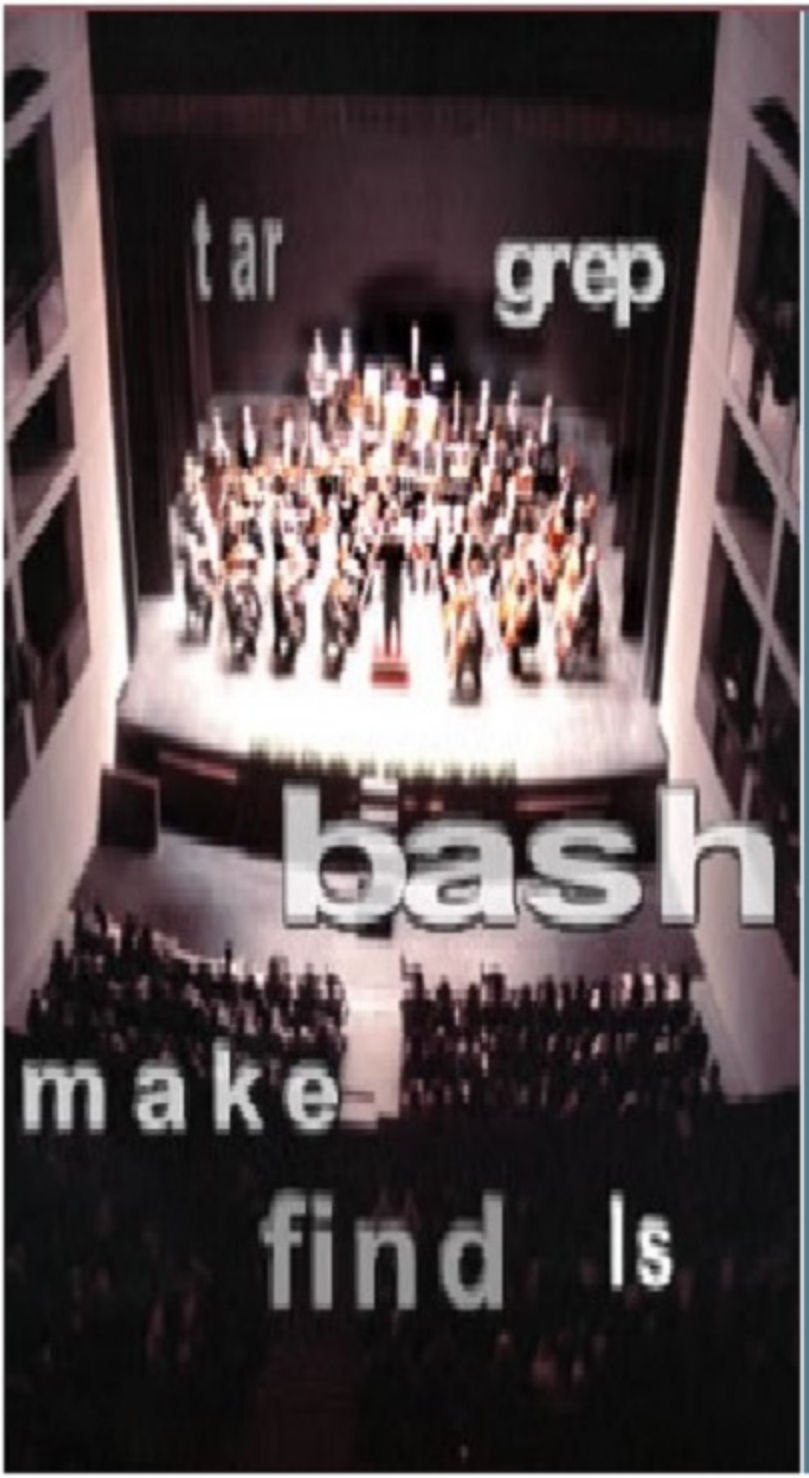
Sistemas Abiertos - Unix



LUIS JOSÉ CEARRA ZABALA

Departamento de Informatica Aplicada.
Area de Arquitectura y tecnología de computadores.
Escuela Universitaria de Informática.

Imagen: CNICE

Autor: Javier Trabadela Robles

Prefacio

Este texto surgió como apuntes de la asignatura *Sistemas Abiertos* de la Escuela Universitaria de Informática en la Universidad Politécnica de Madrid.

No presenta temas nuevos. Aporta la selección de unos pocos comandos y conceptos que (espero que) ayudarán a algunos usuarios a introducirse en el uso de sistemas tipo UNIX.

A lo largo del texto algunos conceptos aparecen en diversos lugares. El *intérprete de comandos* nos acompaña de principio a fin. El editor `vi` se introduce pronto para poder usarlo cuanto antes, y más tarde se estudia exhaustivamente.

No hay un orden estricto en la lectura de los temas. Por ejemplo, el penúltimo capítulo presentado (*en red*) podría adelantarse hasta la parte central. El tema de `make` podría ir al final.

Acaba el libro con un capítulo titulado *Cuentos* y una colección de ejercicios organizados en exámenes. Los *cuentos* presentan unas sesiones, unos ejemplos, de uso de UNIX. Los exámenes están precedidos de unos consejos.

Uso caracteres `sans sherif` para representar texto que aparece o puede aparecer en el terminal de un ordenador, para nombres de comandos (como `who`) y para nombres concretos de ficheros (como `/etc/passwd`) . Como las comillas se usan con mucha frecuencia al interactuar con el ordenador no las uso para señalar textualmente algo. En lugar de comillas aumento la separación blanca de los elementos citados textualmente. Uso caracteres VERSALITA para marcas de productos y para algunos caracteres ASCII. Espero conseguir una buena comunicación con el lector.

En `http://lem.eui.upm.es/~luisjc/sa/apuntes-ssaa.html` hay enlaces a material relacionado con este libro: transparencias, resumen de comandos de UNIX, ejercicios con *expresiones regulares*.

Cualquier comentario o sugerencia que me hagan sobre el texto será de gran ayuda. Espero los mensajes en `sapuntes@lem.eui.upm.es` .

Quiero agradecer a todos aquellos que me han ayudado a la confección de estos apuntes. En particular a Antonio Fernández Anta, Pedro José Fuentes Moreno, Jose Gabriel Pérez Díez, José Luis González González, David González Gordon, Araceli Lorenzo Prieto, Pilar Manzano García, Isabel Muñoz Fernández, Braulio Núñez Lanza y Eduardo Pérez Pérez.

Esta es la cuarta edición. Ha pasado bastante tiempo desde la tercera edición. Unix va cambiando. Linux llegó, y sigue. Es obligado acutalizar la

descripción de algunos comandos. He cambiado tambien el orden de acuerdo con el seguido en las clases los últimos años.

Dedico este texto a los lectores.

Índice general

Capítulo 1

Una sesión

1.1. Una sesión tranquila

Miramos a un terminal de un sistema UNIX (o parecido a UNIX) y vemos:

```
Welcome to Linux 1.2.1

pio login:
```

Está libre. Nos sentamos. Tecleamos nuestro *identificador de usuario*, por ejemplo `a0007` , seguido de un *retorno de carro* .

> La tecla de 'retorno de carro' suele estar situada en la parte derecha del teclado y ser de mayor tamaño. A continuación tenemos la disposición de teclas en un teclado. En la figura 1.1 el retorno de carro está marcado con ↵ . En otro teclados pone ENTER, RET o RETURN.

> El identificador de usuario es el nombre por el que nos conoce el sistema. A veces está limitado a 8 caracteres. Puede ser el nombre, o el apellido, o el alias en un ordenador con pocos usuarios. Puede ser un número de expediente, como `a0007`, en un sistema para prácticas de un centro universitario con cientos de alumnos. En sistemas con decenas de usuarios es costumbre usar la inicial del nombre seguida del primer apellido.

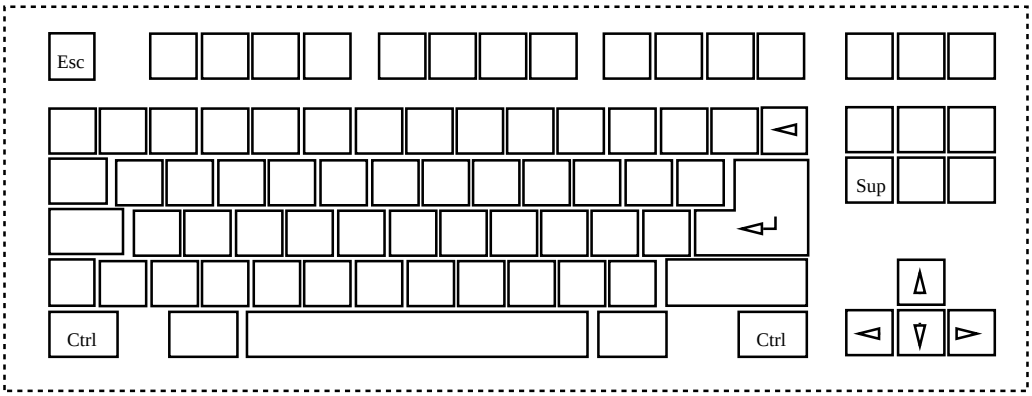


Figura 1.1: Un teclado

Según hemos tecleado hemos visto aparecer los caracteres en la pantalla. Al teclear el 'retorno de carro' el *cursor* baja a la línea siguiente (o si está en la última línea, se desplaza hacia arriba todo lo que está escrito [*scroll* en inglés]), y aparece en la pantalla una petición de *contraseña* . En la pantalla veremos:

```
Welcome to Linux 1.2.1

pio login: a0007
Password: _
```

> El cursor indica donde se va a escribir el siguiente carácter en la pantalla. Puede tomar la apariencia de un rectángulo o un subrayado. En unos casos destella y en otros no. En estos ejemplos tiene el aspecto `_` .

Tecleamos nuestra contraseña , por ejemplo `esTaNoo` . El cursor no se mueve hasta que damos el retorno de carro. No aparecen en el terminal los caracteres que tecleamos. No hay *eco*. La intención es que nadie se entere de nuestra contraseña mirando a la pantalla.

Aparece en la pantalla un mensaje, unos pocos caracteres más, y se queda el cursor cerca del borde izquierdo:

```
pio login: a0007
Password:
```

```
El dia 21 comienza el horario de verano.

% _
```

Al carácter o los caracteres de esta última línea se les llama (indicación de) *espera*. En el ejemplo que ponemos es `%` . Nos indica que el sistema UNIX nos atiende, está preparado para hacer lo que digamos. Al menos lo intentará.

Podemos preguntarle o pedirle al sistema muchas cosas. Por ejemplo:

- ¿Quién está conectado al ordenador? (`who`)
- ¿Qué hora es ? (`date`)
- ¿Cuál es el calendario de este mes? (`cal`)
- ¿Dónde estoy ? (`pwd`)
- ¿Qué tengo ? (`ls`)
- ...

Preguntamos quién está conectado al ordenador: `who` . Nos responde el ordenador con una línea por usuario conectado. Escribe algo así como el nombre de cada usuario (el identificador de usuario), el nombre de la 'línea' a la que está conectado ese usuario y el comienzo de su conexión. Después el ordenador escribe el indicador de espera. En la pantalla veremos:

```
El dia 21 comienza el horario de verano.

% who
a0007    ttyp4   Mar  4 11:56
a0086    ttyp2   Mar  4 10:32
% _
```

Nos despedimos tecleando `exit` . Reaparece el mensaje inicial y `login:` . El ordenador está de nuevo esperando un usuario. Hemos completado una *sesión*.

Cuando queramos volver a trabajar con el sistema, tecleamos nuestro identificador de usuario en un terminal libre, en el que veamos `login:` , y luego tecleamos la contraseña.

1.2. Unos consejos

1. En los sistemas UNIX (y parecidos) las **mayúsculas** se consideran **distintas** de las **minúsculas** [1] . En general, se hace más uso de las minúsculas.

2. No use las flechas. En UNIX son más una molestia que una ayuda. Las flechas consiguen que lo que vemos en la pantalla no se corresponda con lo que el ordenador quiere comunicarnos.

3. Podemos borrar (repetidamente) el último carácter. En unos sistemas se borra un carácter con la tecla de *retroceso* y en otros con la tecla *borrado*. Cuando damos el retorno de carro el sistema atiende nuestra petición y ya no podemos borrar caracteres.

   > La tecla de borrado suele estar rotulada con 'del' o 'supr'. La tecla de retroceso suele estar rotulada como 'back' o ◅ . (En la figura 1.1 la tecla de retroceso está encima de la de cambio de línea.)

4. Podemos descartar una línea a medio escribir tecleando CONTROL-U. Se aprieta la tecla rotulada 'ctrl' y se pulsa `u` .

5. Al comenzar la sesión no es fácil corregir los errores. Escriba despacio. Si se equivoca, pulse 'retorno de carro' una o dos veces (tranquilamente) hasta que aparezca `login:` .

6. Para que no se difunda su contraseña:

   - No la teclee antes de tiempo.
   - No la teclee cuando miran su teclado.

   Por educación, no mire cuando otros van a teclearla.

7. Conviene no olvidarse de cerrar la sesión tecleando `exit` o `logout` .

[1] Este comentario está dedicado a los usuarios de MSDOS, que no hace esta diferencia.

1.3. Si nos despistamos ...

En muchos sistemas UNIX es posible que el ordenador adopte un comportamiento extraño si no seguimos los consejos anteriores.

1.3.1. ... con las flechas

El uso de las flechas nos puede llevar, por ejemplo, a que cuando preguntamos por el nombre de un objeto, veamos en una pantalla `abc` , mientras que el ordenador ha escrito `ad^]]Dc` , y tiene como información válida `ad^]]Dc` .

En algunos ordenadores personales el uso de las flechas puede ser seguro.

1.3.2. ... al comenzar la sesión

- Si, por tener prisa, tecleamos la contraseña (o parte de ella) antes de que aparezca el texto `Password:` , el ordenador visualizará esos caracteres de la contraseña en la pantalla. Podrán verlos otras personas. Y no aceptará nuestra contraseña.

- Mientras tecleamos el identificador de usuario algunos sistemas tienen como tecla de borrado una que no nos esperábamos: `#` (almohadilla) . Además, en esos sistemas, aunque ese carácter borra, el cursor no retrocede. (!!).

- Igualmente, en algunos sistemas, si el primer carácter tecleado después `login:` es una mayúscula, el ordenador utiliza las letras mayúsculas como minúsculas, y representa las letras mayúsculas mediante mayúsculas precedidas por el carácter '`\`' .

  En vez de escribir `procedente de EEUU` el ordenador escribirá `PROCEDENTE DE \E\E\U\U` .

  El ordenador abandona este convenio cuando acaba la sesión (si había empezado) o aproximadamente un minuto después si no llegamos a comenzar la sesión.

Lo mejor es no correr mientras tecleamos nuestro identificador de usuario y contraseña.

1.3.3. ... y alguien sabe nuestra contraseña

Puede usarla para entrar con nuestro identificador de usuario. Somos (hasta cierto punto) responsables de lo que haga después de entrar con nuestro identificador de usuario.

Decir a alguien nuestra contraseña podemos considerarlo semejante a prestar nuestra identidad.

1.3.4. ... y dejamos la sesión abierta

Alguien puede usar el sistema en nombre nuestro hasta que cierre la sesión: por ejemplo toda la tarde. Es más, puede dejar un programa que le permita volver a usurpar nuestra identidad ante el sistema en una sesión posterior.

Capítulo 2

Para cambiar de contraseña

Para cambiar de contraseña tecleamos `passwd` . El sistema nos pregunta (una vez) por la contraseña antigua y dos veces por la contraseña nueva.

```
% passwd
passwd:  Changing password for a0007
Old password:_
New password:_
Retype new passwd:_
%
```

Cuando tecleamos las contraseñas el sistema no hace eco, no se ve en la pantalla. El sistema pregunta dos veces por la contraseña nueva para evitar dar por buena una contraseña equivocada.

El comando `passwd` pide que la contraseña nueva no sea breve y no sea demasiado parecida a la anterior. Permite intentarlo hasta tres veces.

```
% passwd
passwd:  Changing password for a0007
Old password:_
New password:_
Password is too short - must be at least 6 digits
New password:_
Password must contain at least two alphabetic
      characters and at least one numeric
      or special character.
New password:_
```

```
Passwords must differ by at least 3 positions
Too many failures - try later
% _
```

La contraseña suele estar limitada a 8 ó menos caracteres cualesquiera. Incluir algún carácter como '¿' puede resultarnos engorroso si cambiamos de terminal y no lo encontramos fácilmente.

Cuentan que hace años hicieron una encuesta en una empresa grande y encontraron que el 25 % de las contraseñas era `maria` seguido de un número, o simplemente `maria` . Con contraseñas así un sistema no es seguro.

Conviene que la contraseña no sea fácil de deducir o adivinar conociendo a la persona o algunos de sus datos (identificador del usuario, cumpleaños, número de matrícula, teléfono, nombre, apellido, etc.). También es conveniente que la contraseña no se pueda encontrar fácilmente por tanteo, es decir que no sea una palabra de diccionario, o nombre de un lugar, o un apellido, o el nombre de una persona, etc. Hay que recordar que no es prudente apuntar la contraseña en un lugar público, cercano del terminal, o fácil de encontrar.

Capítulo 3

Correo

El correo lo escribimos y enviamos. Alguien lleva el correo a un buzón del destinatario. El destinatario recoge el correo y lo lee.

3.1. `mail`

Para leer el correo podemos usar `mail` .

```
% mail
No mail for a0007
% _
```

No tenemos mensajes. Vamos a enviarnos nosotros mismos un mensaje. Usaremos el mismo comando pero poniendo a continuación el destinatario. El progama `mail` nos pregunta por el tema (`Subject:`) del mensaje. Despues podemos escribir el mensaje. Terminaremos con un punto (`.`) al comienzo de una línea. Algunas versiones de `mail` no terminan con un punto sino con CONTROL-D.

```
% mail a0007
Subject: he prestado
.. el Tanembaum a Javier
.
% _
```

Cuando escribimos un mensaje con `mail` no debemos utilizar las flechas. Eso nos recuerda que no podemos corregir el texto que está en las

líneas anteriores a la del cursor. Esta forma de trabajar nos vale para mensajes cortos (unas pocas líneas). En la sección 9.3.7 (pág. 55), veremos como componer mensajes con el editor `vi` .

Vamos a ver si tenemos mensajes.

```
% mail
"/usr/spool/mail/a0007": 2 messages 2 new
>N  1 a0007    Mon Mar 11 16:36  11/408   "he prestado"
 N  2 a0086    Mon Mar 11 16:38  13/445   "cena"
&
```

Aparece una primera línea con el número de mensajes en el buzón `/usr/spool/mail/a0007`, y una línea por cada mensaje con el número de orden, quién lo envía, fecha de recepción, tamaño en líneas y en caracteres, y tema. Una de las líneas está señalada por `>` , *cursor de mail*. Indica el mensaje que se supone nos interesa cuando no indicamos otro. Al final aparece `&` que es una indicación de espera de `mail` .

Leyendo el correo, cuando aparece su indicación de espera `&` podemos:

- Leer un mensaje cualquiera. Por ejemplo `2` lee el mensaje segundo. Despues de leer un mensaje, si hay más mensajes el cursor pasa al siguiente.

- Leer el mensaje indicado por el cursor de mail, dando retorno de carro.

- Abandonar la lectura de mensajes dando `q` (de *quit*).

- Abandonar la lectura de mensajes dejando en el buzón los mensajes que había al comenzar al comenzar el uso de `mail` , dando `x` .

- Salvar un mensaje en un fichero. Por ejemplo
  `s 1 prestamos`
  lleva una *copia* del primer mensaje al fichero `prestamos` . Estando el cursor (`>`) en el primer mensaje `s prestamo` tiene el mismo efecto. Si el fichero no existía se crea. Si ya existía, pone este mensaje al final.

  Podemos salvar varios mensajes consecutivos indicando el primero y el último. Por ejemplo, `s 3-5 publicidad`

- Contestar a un mensaje. Con `r` (*reply*) pone como destino el origen del mensaje de referencia, y como tema `Re:` seguido del tema del mensaje de referencia. Por ejemplo, `r 2` manda un mensaje a `a0086` y pone como tema `Re: cena` . Luego espera que tecleemos el mensaje acabando con un punto en la columna primera o con un CONTROL-D.

- Marcar un mensaje, con `d` (*delete*), para que se borre si salimos normalmente (`q`).

3.2. Buzones

A cada usuario se le asigna un buzón para que el sistema deje los mensajes que le envían con `mail` . En el ejemplo anterior al usuario `a0007` le asigna el buzón `/usr/spool/mail/a0007`

Si acabamos el uso de `mail` con `q` , los mensajes leídos se borran del buzón de entrada y se dejan en un segundo buzón cuyo nombre suele ser `mbox` .

Si salvamos mensajes con `s` , creamos ficheros semejantes a los buzones anteriores.

Podemos leer mensajes de los buzones poniendo la opción `-f` y el nombre del buzón. Por ejemplo `mail -f prestamos` . Si ponemos `-f` sin nombre de buzón, toma `mbox` .

3.3. Otros lectores de correo

`mutt` En la línea de `mail`. Permite *agregados* (*attachments*).

`mh` Conjunto de programas para leer y administrar el correo.

`pine` Incluye un editor sencillo y permite *agregados*.

`elm`

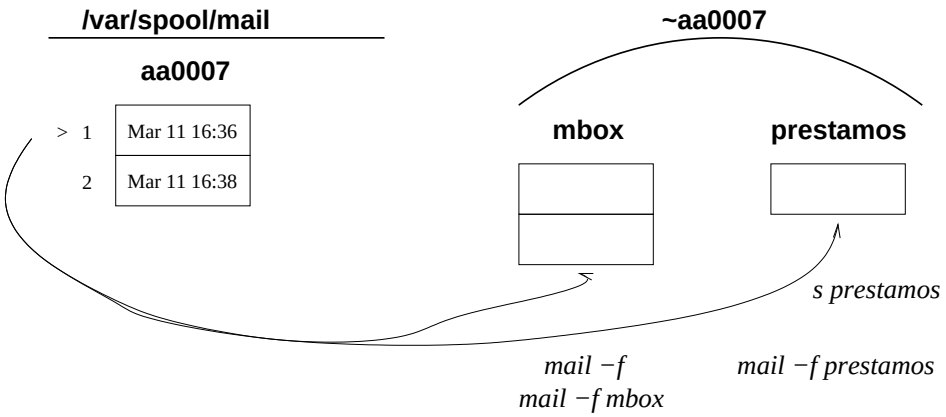


Figura 3.1: Buzones de un usuario

.

Capítulo 4

¿Qué hay?

De donde no hay no se puede sacar.

Para responder a la pregunta anterior utilizaremos el comando `ls`.

`ls` muestra los nombres de los objetos accesibles en nuestro entorno.

La mayoría de los objetos serán ficheros, pero también hay otros tipos de objetos.

> Normalmente, cuando empezamos a usar un sistema con UNIX, nos dan un directorio que está vacío y que nosotros iremos llenando. Los más ordenados hasta harán alguna limpieza por primavera. Como no hemos llegado a *directorios*, escribo (impropiamente) entorno. Vamos a suponer que hay algo en el directorio.

```
$ ls
desenlace
indice
nudo
presentacion
```

`ls -l` nos muestra más información de esos objetos.

```
$ ls -l
total 153
-rw-r--r-- 1 a0007     186 Sep 31 16:33 desenlace
-rw-r--r-- 1 a0007    4795 Sep 31 16:34 indice
-rw-r--r-- 1 a0007  133440 Sep 31 16:32 nudo
-rw-r--r-- 1 a0007    3038 Sep 31 16:33 presenta
```

El primer carácter de cada línea nos indica el tipo de objeto. En este caso es un guión o signo menos (`-`), lo que significa que los cuatro objetos son ficheros.

Los nueve caracteres siguientes indican los permisos asociados a los objetos. Su dueño puede leerlos (`r`) y escribir en ellos (`w`), y el resto de los usuarios sólo puede leerlos (`r`).

El segundo campo nos indica cuántos nombres (enlaces) tiene un objeto. Los cuatro ficheros del ejemplo tienen un único nombre.

El tercer campo nos presenta el dueño del objeto.

El cuarto campo nos presenta el tamaño del objeto medido en caracteres.

Los campos quinto, sexto y séptimo nos indican la fecha y hora de la última modificación del fichero. Si esto sucedió hace más de seis meses se utiliza el formato: mes, día y año. La fecha que aparece en el ejemplo la he cambiado a mano. Es una broma.

Por último, en el octavo campo aparece el nombre del objeto.

Algunas versiones de `ls` presentan la fecha de última modificación con todo detalle cuando se pide con la opción `--full-time` .

```
$ ls -le
total 153
-rw-r--r-- 1 a0007     186 Thu Sep 31 16:33:05 1998 desenlace
...

$ ls -x
desenlace  nudo
indice     presenta
$ ls -ax
.          .delEditor  indice     presenta
..         desenlace   nudo
```

`ls -x` presenta los nombres de los objetos en columnas. Así es más fácil que se vean todos los nombres en la misma pantalla.

La opción `-a` hace que aparezcan también los nombres de los objetos que empiezan por punto. Los dos primeros nombres (`.` y `..`) aparecerán siempre con la opción `-a` .

Normalmente las opciones se pueden combinar como en este ejemplo.

4.1. ¿Es imprimible?

Sabemos preguntar por los nombres y podemos determinar cuáles corresponden a ficheros. Queremos conocer su contenido. Podemos pedir que el contenido aparezca en la pantalla o en la impresora.

No conviene que enviemos cualquier fichero a pantalla sin cuidado. Algunos ficheros pueden enviarse a pantalla sin preocuparse. Todos sus caracteres son imprimibles. Otros ficheros conviene no enviarlos directamente a pantalla. Tienen caracteres no imprimibles.

Si se envía a pantalla un fichero con caracteres no imprimibles puede suceder que lo que veamos no se corresponda con el contenido, puede que la pantalla quede en un modo tal que todo lo que venga a continuación sea ilegible, o puede que quede bloqueada. Nada de esto es irreversible, pero es bastante molesto.

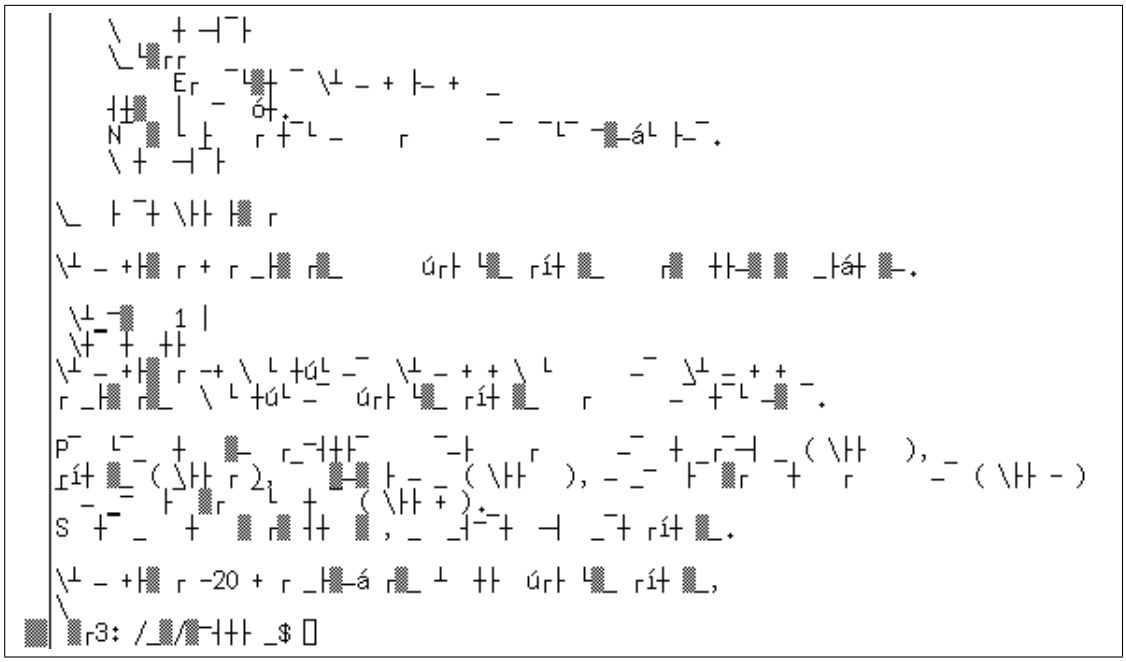


Figura 4.1: Apariencia de una pantalla en modo gráfico cuando debía estar en modo texto.

Se puede hacer afirmaciones parecidas con respecto a la mayor parte

de las impresoras. El envío directo para impresión de ficheros con caracteres no imprimibles, por ejemplo ficheros con programas en código máquina, puede producir el despilfarro de papel con dibujos incoherentes, hojas de papel blanco abarquilladas que no se pueden volver a usar porque atascan la impresora, y gasto inútil de tinta o tóner. Una pena. :,-(.

Sólo hay que matizar que algunas impresoras tienen su propio lenguaje (por ejemplo el de las impresoras de HP) que incluye caracteres que he llamado no imprimibles. Estos caracteres (en general no imprimibles) no son inconveniente para que el fichero se imprima bien en la impresora correspondiente.

Vamos a cambiar de escenario respecto a la sección anterior.

```
$ ls -x
allons   date     quij     xx
aprende  log      texto.z
$ echo Hola Eco
Hola Eco
```

Vemos que en el entorno hay siete objetos.

`echo` repite los parámetros.

`echo` hace el papel de sentencia de escritura al igual que en otros lenguajes tenemos `WriteLn` , `print` y `printf` .

```
$ echo *
allons aprende date log quij texto.z xx
```

Aquí hay algo extraño. Si `echo` repite los parámetros esperamos que escriba un asterisco (`*`). La clave está en que ha intervenido el *intérprete de comandos*.

El *intérprete de comandos* ha substituido el asterisco por la lista ordenada (según el código ASCII) de nombres de objetos que no empiezan por punto. Luego, el *intérprete de comandos* ha llamado al programa `echo` pasándole como parámetros esa lista de nombres.

```
$ file *
allons:    English text
aprende:   ascii text
```

```
date:      sparc pure dynamically linked execut
log:       executable shell script
quij:      English text
texto.z:   packed data
xx:        data
```

El *intérprete de comandos* toma la línea `file *` y la substituye por `file allons aprende date log quij texto.z xx`

El comando `file` intenta determinar el tipo de los objetos cuyos nombres aparecen como parámetros. La mayor parte de las veces los nombres corresponden a ficheros. Empieza observando si el fichero tiene caracteres no imprimibles.

Cuando el fichero tiene caracteres no imprimibles estudia sus dos o cuatro primeros octetos. Si esos primeros octetos toman unos valores reconocidos, `file` indica el tipo de objeto asociado. De esta manera deduce el tipo de los ficheros `date` y `texto.z` en el ejemplo.

> Inicialmente los valores conocidos como marca del tipo estaban en el código del programa (comando) `file` . Al ir aumentando el número de valores-marca y tipos, se ha pasado a consultar estos números en el fichero `/etc/magic` .

Cuando el fichero tiene caracteres no imprimibles y el comando `file` no puede determinar el tipo escribe `data` (del latín *datum data*, *datos*).

Cuando el fichero **no** tiene caracteres no imprimibles, `file` intenta determinar el lenguaje. Para ello compara la frecuencia de caracteres y de ciertas palabras con las estadísticas de uso del inglés y de algunos lenguajes de programación. Basándose en estas frecuencias aventura que el texto es inglés, fuente de un programa en lenguaje C o programa para el intérprete de comandos. Cuando no parece nada de lo anterior lo clasifica como "texto ascii".

`file` trabaja con indicios. En ficheros grandes sólo lee los dos mil primeros octetos. No está garantizado que lo que escribe sea exacto. Con los ficheros del ejemplo veremos que se confunde al suponer que algunos textos están en inglés. La confusión se debe a que los ficheros son pequeños.

Aun así es útil. En particular, nos permite determinar si un fichero es imprimible o no. En el ejemplo, nos muestra que `allons` , `aprende` , `log` y `quij` son imprimibles.

4.2. ¿Cuántas líneas tiene?

Antes de enviar a pantalla o a impresora un fichero imprimible queremos saber su tamaño.

`wc` nos informa del tamaño de un fichero contando sus líneas, palabras y caracteres. La última información la podíamos conocer con `ls` . En este momento considero que lo más significativo son las líneas.

```
$ wc allons aprende quij
       2      11      57 allons
       2       9      42 aprende
      52     539    3007 quij
      56     559    3104 total
```

Si aparecen varios ficheros como parámetros, `wc` escribe también el total de líneas, palabras y caracteres. El nombre del comando viene de *word count*.

4.3. Para ver ficheros pequeños

Si el fichero es pequeño utilizaremos el comando `cat`. `cat` pone a continuación (concatena) las líneas de los ficheros que aparecen como parámetros y las escribe en pantalla.

```
$ cat allons
Allons enfants de la patrie
le jour de gloire est arrive
```

Cuando `cat` tenga un solo parámetro enviará a pantalla ese fichero. Este método es útil para ficheros que quepan en una sola pantalla o ventana. En el caso de usar ventanas, si tienen memoria y una barra para ver las últimas líneas (de *scroll*) puede usarse con ficheros un poco mayores.

4.4. Para ver ficheros medianos o grandes

Si enviamos a pantalla un fichero grande, las líneas vuelan, pasan ante nuestros ojos sin que seamos capaces de leerlas. Aunque vayan despacio

es posible que queramos examinar con cuidado una fórmula, una sentencia (¿judicial?, ¿informática?), ...

Queremos que hasta que no digamos: - **más** - la salida de caracteres por pantalla se detenga y no se pierda información por la parte superior de la pantalla. Ese "más" da nombre al comando que nos ayudará: `more` .

```
$ more quij
```

```
En un lugar de la Mancha de cuyo nombre no quie
no ha mucho tiempo que vivia un hidalgo de los
adarga antigua, rocin flaco y galgo corredor. U
que carnero, salpicon las mas noches, duelos y
lentejas los viernes, algun palomino de anadidu
--More--(11%)
```

El comando `more` nos presenta el fichero pantalla a pantalla. En la figura anterior suponemos que estamos trabajando con una pantalla de 6 líneas. En las 5 primeras líneas de pantalla aparecen las correspondientes líneas del fichero. En la última línea de pantalla se nos recuerda que estamos trabajando con `more` y que hemos visto hasta el 11 por ciento del texto.

```
lentejas los viernes, algun palomino de anadidu
consumian las tres partes de su hacienda. El re
sayo de velarte, calzas de velludo para las fie
con sus pantuflos de lo mesmo, y los dias de se
vellori de lo mas fino.
--More--(20%)
```

Cuando tecleamos un espacio blanco, `more` presenta casi una pantalla nueva. En el ejemplo presenta cuatro líneas más.

`more` lee caracteres sin esperar el *retorno de carro*.

Si tecleamos *retorno de carro* el texto avanza una línea.

Si tecleamos eslás (`/`) el cursor baja a la última línea. Los siguientes caracteres que tecleamos aparecen en la última línea. Cuando damos *retorno de carro*, `more` busca la secuencia indicada en el texto no presentado. La pantalla presenta en una de sus primeras líneas la tira de caracteres buscada.

Un espacio blanco o un *retorno de carro* al final del fichero acaba la sesión con el comando `more` . Otra forma de acabar la sesión es tecleando `q` (de *quit*).

`more` admite más comandos. Esperamos que el lector acuda al manual (enseguida contaremos cómo).

El comando `more` se escribió cuando la mayor parte de los ordenadores tenían poca memoria.

> Tenían poca porque era cara, la tecnología no daba para más. En 1985 más o menos la E.U.I. compró una ampliación de memoria para un miniordenador que compartían simultáneamente hasta 8 usuarios. Costó unas 500.000 pesetas, 512 koctetos.

El comando `more` debía trabajar de igual forma cuando los datos le llegasen de otro programa. Al disponer de poca memoria se especificó que `more` no retrocedería.

`less`

Al pasar el tiempo los ordenadores disponían de más memoria. Parece lógico no verse limitado a volver a ejecutar el comando `more` si queremos ir al principio, por ejemplo, o retroceder un poco.

`less` es un comando que admite todas las posibilidades de `more` y acepta además otros movimientos. Para no obligar al usuario a aprender nuevas asociaciones *letra - movimiento*, `less` toma los movimientos del editor `vi` y los obedece.

En particular, conviene adelantar que `^B` (CONTROL-B) retrocede una pantalla. Más adelante hay dos capítulos dedicados al editor `vi` .

El comando `less` es un programa realizado por el ¿grupo? GNU. Es software que se puede compartir (¡legalmente!) sin pagar.

Su nombre parece que viene de un juego (trivial) de palabras: *less is more than more*, (*menos es más que más*).

4.5. Para ver ficheros con caracteres no imprimibles

Es posible que un día necesitemos ver el contenido de un fichero en el que hay cierta información en un formato que no está pensado para ser visualizado o impreso directamente.

También es posible que queramos ver si hay caracteres blancos o tabuladores después del último carácter visible y antes del cambio de línea.

En ambos casos nos resultará útil el comando `od` . Su nombre viene de *octal dump* (volcado octal), operación consistente en escribir en octal el contenido de la memoria y usada inicialmente para corregir programas.

Empezamos aplicando `od -c` al fichero `allons` del que conocemos el contenido. La opción `-c` viene de *carácter*.

```
$ od -c allons
0000000    A   l   l   o   n   s       e   n
0000020    e       l   a       p   a   t   r
0000040    o   u   r       d   e       g   l
0000060    t       a   r   r   i   v   e  \n
0000071
```

En la primera columna vemos (en octal) el desplazamiento del carácter (octeto) siguiente. En cada línea se presentan dieciséis caracteres (octetos) del fichero separados por tres espacios en blanco. Aparece `\n` que es una representación del cambio de línea.

Si pedimos ver la misma información como octetos con `od -b` obtendremos una serie de números representados en octal. La opción `-b` viene de *byte* (*octeto*).

```
$ od -b allons
0000000  101 154 154 157 156 163 040 145 156 14
0000020  145 040 154 141 040 160 141 164 162 15
0000040  157 165 162 040 144 145 040 147 154 15
0000060  164 040 141 162 162 151 166 145 012
0000071
```

Podemos ver las dos representaciones juntas poniendo las dos opciones (`-bc`).

```
$ od -bc allons
0000000  101 154 154 157 156 163 040 145 156 14
           A   l   l   o   n   s       e   n
0000020  145 040 154 141 040 160 141 164 162 15
           e       l   a       p   a   t   r
0000040  157 165 162 040 144 145 040 147 154 15
           o   u   r       d   e       g   l
0000060  164 040 141 162 162 151 166 145 012
           t       a   r   r   i   v   e  \n
```

Vemos que el segundo octeto (desplazamiento 1) puede considerarse que es el carácter `l` o el número 154 octal ($64 + 5 * 8 + 4 = 108$ decimal).

Después de probar el comando `od` con un fichero conocido, lo utilizamos para ver el contenido del fichero `xx`.

```
$ od -c xx
0000000   \0 013 220  \n     377 320   /   @  \
0000020      001 222   "    ‘ 001 322   &
```

4.6. Para ver (parte de) el contenido

A veces sólo nos interesa ver las primeras líneas de un fichero (unas pocas) y no hace falta hacer `more` seguido de `q` que nos mostraría las 22 primeras líneas.

```
$ head -3 quij
En un lugar de la Mancha de cuyo nombre no quie
no ha mucho tiempo que vivia un hidalgo de los
adarga antigua, rocin flaco y galgo corredor. U
$
```

`head -3 quij` nos muestra las 3 primeras líneas del fichero nombrado. `head` toma por omisión 10 líneas.

`tail -6 quij` nos muestra las 6 últimas líneas del fichero nombrado. `tail` toma por omisión 10 líneas.

`grep` extrae las líneas que contengan la tira de caracteres indicada.

```
$ grep una quij
Tenia en su casa una ama que pasaba de los cuar
y una sobrina que no llegaba a los veinte,
que en esto hay alguna diferencia en los autore
$
```

4.7. Para imprimir

`lpr quij` manda a cola de impresión el fichero nombrado. Si no hay trabajo pendiente se imprimirá inmediatamente.

`lpq` nos informa de los trabajos que hay pendientes para imprimir.

```
$ lpq
Rank   Owner      Job  Files                         Total Size
active a0007      331  quij                          3007 bytes
```

`lprm 331` borra de la cola de impresión el trabajo cuyo número se indica. Si sólo hay un trabajo no hace falta dar el número.

Capítulo 5

Manuales

A lo largo del libro veremos bastantes comandos, pero no vamos a ver todas las opciones de cada comando. Ni siquiera veremos la décima parte. Llegará el momento en que convenga consultar los manuales. Es costumbre que los manuales estén accesibles en el mismo ordenador (*on-line*).

> Hay quien considera que consultar los manuales es lo último. Tiene al menos un poco de razón considerando el tamaño que han alcanzado los manuales. La primera versión comercial de UNIX (III) tenía unas 2500 páginas (1984). En 1992 adquirimos los manuales de un sistema, 30.000 hojas aproximadamente. En 1994 nos dieron la documentación en disco compacto. El manual de instalación sólo se podía consultar después de instalar el sistema operativo :-) .
>
> Pero no hay que exagerar. Si aprendemos a buscar encontraremos enseguida las hojas que nos interesan. Otra cuestión es entender el manual. Las primeras veces puede resultar extraño. Ayuda bastante familiarizarse con la estructura de los manuales.
>
> Consultar los manuales tampoco es lo primero. Lo más fácil es preguntar a un compañero/a. A veces sucede que el compañero nos lo acaba de preguntar, o que nos han contratado como expertos (se dice *gurú(s)* en el gremio) y no nos quedará más remedio que ... . En resumen, consultar el manual una vez a la semana es cosa sana.

`man who` presenta la(s) hoja(s) de manual correspondientes al comando `who` .

> Los manuales de UNIX están formateados mediante un programa (`nroff`) y unas macros (`me`) y siguiendo unas pautas que garantizan una presentación homogénea.

```
WHO(1)                    USER COMMANDS                    WHO(1)

NAME
     who - who is logged in on the system

SYNOPSIS
     who [ who-file ] [ am i ]

DESCRIPTION
     Used without arguments, who lists the login  name,  terminal
     name,  and  login time for each current user.  who gets this
     information from the /etc/utmp file.

     If a filename argument is given, the named file is  examined
     instead   of   /etc/utmp.    Typically  the  named  file  is
     /var/adm/wtmp, which contains a record of all  logins  since
     it  was  created.   In this case, who lists logins, logouts,
     and crashes.  Each login is listed with user name,  terminal
     name  (with  /dev/  suppressed), and date and time.  Logouts
     produce a similar line without a user name.  Reboots produce
     a  line  with  `~' in place of the device name, and a fossil
     time indicating when the system  went  down.   Finally,  the
     adjacent  pair  of  entries `|' and `}' indicate the system-
     maintained time just before and after a date command changed
     the system's idea of the time.

     With two arguments, as in `who  am  i'  (and  also  `who  is
     who'), who tells who you are logged in as:  it displays your
     hostname, login name, terminal name, and login time.

EXAMPLES
          example% who am i
          example!ralph  ttyp0     Apr 27 11:24
          example%

          example% who
          mktg   ttym0   Apr 27 11:11
          gwen   ttyp0   Apr 27 11:25
          ralph       ttyp1   Apr 27 11:30
          example%

FILES
     /etc/utmp
     /var/adm/wtmp

SEE ALSO
     login(1), w(1), whoami(1), utmp(5V), locale(5)
```

Figura 5.1: Página de manual de `who`

Conviene familiarizarse con los apartados de los manuales y en particular con el orden en que aparecen.

En el apartado "SYNOPSIS" nos presentan el significado y posición de los parámetros y opciones de los comandos. Cuando algo está entre corchetes `[ ]` significa que es opcional. Cuando detrás de los corchetes aparecen unos puntos suspensivos, se indica que puede haber 0, 1 ó varias apariciones de lo que se encierra entre corchetes.

El apartado "EXAMPLES" presenta ejemplos que muchas veces no son triviales, y a veces incluso aparece resuelto nuestro problema.

El apartado "SEE ALSO" nos propone otras consultas. A veces aparecen comandos de funcionalidad semejante. Otras veces son comandos que deben usarse antes, o que realizan operaciones complementarias.

Es curiosa la existencia de un apartado "BUGS" (errores, gazapos). Si un comando se distribuye inacabado en ese apartado aparecerán los aspectos incompletos.

En la mayor parte de los sistemas, el comando `man` nos presenta la información pantalla a pantalla e interactuamos con el comando `more` (sin haberlo invocado explícitamente).

Si estamos en un sistema antiguo y el comando `man` produce un chaparrón de líneas en el terminal que no podemos leer, escibiremos `man ls |more` (suponiendo que queríamos consultar la información del comando `ls`). También podemos usar `less` en lugar de `more` .

5.1. Organización de los manuales en UNIX

Con el comando `man` accedemos a las secciones de manual (*References*) organizadas como un diccionario. Se consulta especificando una palabra. (Hay otros manuales (*Tutorials*) organizados en forma más parecida a un libro de texto).

Tradicionalmente los manuales de referencia están organizados en 8 secciones.

1. Comandos.

   La mayor parte de estos apuntes se centra en los comandos. Por ejemplo `ls`, `who`, `mail` .

2. Llamadas al sistema.

3. Funciones de librería.

   Esta sección y la anterior interesan a los que programan, en particular a los que programan en lenguaje C . Por ejemplo `sin`, `putc`.

4. Ficheros especiales.

   Aquí se documenta cómo acceder a los periféricos: cintas, discos flexibles, dispositivos SCSI, etc.

5. Formato de (ciertos) ficheros.

   Se describen los registros y campos que organizan la información. Por ejemplo `passwd` documenta el formato del fichero de contraseñas.

6. Juegos.

   Para los primeros usuarios de UNIX merecen una sección.

7. Varios.

   Cajón de - sastre. Por ejemplo `ascii` .

8. Procedimientos de administración.

   Por ejemplo `mount` .

   En algunos sistemas se han añadido unas secciones (`l`) para aplicaciones locales y (`n`) para aplicaciones nuevas. No está clara su utilidad.

Puede suceder que queramos preguntar por una palabra que aparece en dos secciones. Por ejemplo `passwd`. Elegimos una de las secciones poniendo el número de la sección antes de la palabra.

`man 5 passwd` nos presenta la información sobre el formato del fichero `/etc/passwd` .

En algunos sistemas (por ejemplo LINUX) se puede acceder a información complementaria con el comando `info` .

El comando `info` organiza la información con estructura de *hipertexto* lo que permite mayor flexibilidad en las consultas.

Capítulo 6

Cuenta

Tener *cuenta* en una máquina equivale a *haber sido presentado* a la máquina. Esta presentación se concreta en una línea del fichero de contraseñas `/etc/passwd` .

```
$ grep a0007 /etc/passwd
a0007:ABp60oBls4JsU:5100:5100:Garcia Severon,Vicente:
                          /eui/alum/curso3/a0007:/bin/sh
```

Por cada usuario con cuenta hay una línea. En el ejemplo hemos roto la línea para que quepa en la página. Algunas cuentas no corresponden a ningún usuario, sino a una función. Es algo parecido a las *personas jurídicas*.

Cada línea del fichero de contraseñas tiene siete campos separados por el cáracter *dos puntos* (`:`) .

1. Nombre de la cuenta.

   Como máximo tendrá 8 caracteres. Lo utiliza el ordenador para responder al comando `who` y para indicar el dueño de los ficheros. Lo usamos para indicar el destino del correo.

2. Contraseña codificada.

   La contraseña se transforma con una función tal que su (función) inversa es muy costosa de calcular. Se supone que la única forma de calcular la contraseña original a partir de la versión codificada es mediante tanteos (búsqueda). Si la contraseña está escogida *al azar entre* 128^8 posibles y se prueba una cada microsegundo, harán falta tres mil

años (más o menos) para encontrarla. En cursiva aparece una hipótesis que no se suele cumplir.

3. Número de cuenta.

   Internamente el ordenador identifica al usuario con un número (usualmente de dos octetos). Cuando presenta información para las personas convierte este número en el nombre de cuenta correspondiente.

4. Número de grupo.

   Se pueden definir grupos de usuarios con el fin de que compartan recursos (por ejemplo ficheros). Los grupos no tienen que ser disjuntos. Un usuario puede estar en más de un grupo. En este campo figura el número correspondiente al grupo primero de un usuario.

5. Nombre del usuario.

   En este campo se suele poner el nombre y los apellidos del usuario. A veces se incluye el número de teléfono. Este campo aparece en el primer párrafo de las cartas que envía el usuario.

6. `HOME`

   Directorio de entrada del usuario. Si no se ha explicado, lo veremos en unas pocas hojas.

7. Intérprete de comandos.

   En el último campo se indica el programa que atiende al usuario después de hacer `login` (conectarse, presentarse).

Aunque es suficiente con una de estas líneas para que un usuario tenga cuenta, en la práctica conviene que exista el directorio nombrado en el campo sexto de su línea en `/etc/passwd` , y que el usuario sea dueño de ese directorio.

6.1. `last`

`last` presenta las sesiones que ha habido en el sistema ordenándolas de más reciente a más antigua. Normalmente se usa con parámetros.

`last -10` presenta las 10 sesiones más recientes.

`last juan` presenta las sesiones del usuario `juan` .

Los parámetros se pueden combinar.

```
$ last  -4 k0999
k0999  tty03  Fri Oct  2 18:23 - 18:24  (00:00)
k0999  tty04  Thu Oct  1 14:22 - 14:29  (00:06)
k0999  tty05  Mon Sep 28 11:30 - 11:30  (00:00)
k0999  tty06  Mon Sep 28 11:16 - 11:29  (00:12)
$
```

`last` nos presenta el usuario, el terminal en el que trabaja, las fechas y horas de conexión y de desconexión, y la duración de la sesión.

Capítulo 7

Metacaracteres

El intérprete de comandos separa las líneas en partes. Corta por los tabuladores y espacios en blanco excepto si están entre comillas o precedidos por un eslás inverso (`\`) . Con cada trozo:

- Lo contrasta con todos los nombres de fichero.
- Lo substituye por la lista ordenada de nombres de fichero que se ajustan al *patrón* de cada trozo.

A continuación vemos cómo se forman los patrones.
La mayor parte de los caracteres se representan a sí mismos. Algunos caracteres (*metacaracteres*) sirven para representar a otros.

Para el intérprete de comandos son metacaracteres los cinco siguientes:

```
* ? [ - ]
```

`*` representa cualquier secuencia de caracteres que no empiece por el carácter punto (`.`) . Puede representar la secuencia vacía.

`?` representa cualquier carácter.

`[ax]` representa el carácter `a` o el carácter `x` . Los corchetes indican alternativa, alguno de los caracteres enumerados.

`[i-z]` representa cualquier carácter de los comprendidos entre `i` y `z` inclusive. Entre corchetes se puede indicar un rango señalando los extremos del mismo.

Ejemplos

```
$ ls
C    a    a1   a11  a12  a21  a3   b    c    d
$ echo a*
a a1 a11 a12 a21 a3
```

Hemos hecho `echo` de los nombres de fichero que empiezan por `a` .

```
$ echo a?
a1 a3
```

Obtenemos los nombres formados por `a` y otro carácter.

```
$ echo a?*
a1 a11 a12 a21 a3
```

Obtenemos los nombres que empiezan por `a` y otro carácter.

```
$ echo a1? a?1
a11 a12 a11 a21
```

Vemos que aparece dos veces `a11`.

```
$ echo ?
C a b c d
```

Obtenemos los nombres de un solo carácter.

```
$ echo [b-d]
b c d

$ echo *[0-9][0-9]*
a11 a12 a21
$
```

Obtenemos algunos nombres de un solo carácter, y luego los nombres que contienen dos cifras seguidas.

Para crear ficheros y hacer pruebas como las anteriores puede usarse el comando `touch` .

`touch f1` crea el fichero `f1` si no existe previamente.

Esto que hemos contado para nombres de ficheros también vale para nombres de otros objetos del *sistema de ficheros* de UNIX.

Capítulo 8

Ficheros

Podemos ver las propiedades de los ficheros con el comando `ls` y opción `-l` . También sabemos preguntar por el tipo de un fichero con el comando `file` y presentar en pantalla el contenido de un fichero con los comandos `cat` y `more`.

Queremos conocer mejor los objetos de tipo fichero.

Un fichero es un recipiente (o continente, o contenedor) que en cada momento **admite un valor** de un conjunto de valores posibles.

Para quien ya sepa programar, **un fichero es** como **una variable** de un lenguaje de programación. Para quien no sepa y vaya a aprender a programar esto le sirve de aperitivo.

> (Algunos lenguajes de programación, como ML, no usan variables.)
>
> La única peculiaridad de los ficheros es que su existencia puede ser mucho más prolongada que la de algunas variables de programas. No parece extraño el caso de variables locales que se crean y desaparecen al cabo de un microsegundo. Algunos ficheros se crean el día que compramos la máquina e instalan el sistema operativo, y los damos por perdidos el día que el ordenador va al desguace.
>
> Lo veo como una diferencia cuantitativa. Las variables globales pueden tener una vida prolongada. Hay programas que comienzan a ejecutarse en una máquina y si no hay fallos no dejan de ejecutarse.
>
> Se puede considerar que toda la vida útil de una máquina constituye la ejecución de un gran programa, y los programitas que invocamos son subrutinas de ese programa grande. Los ficheros son las variables globales de ese programa grande.

El fichero y su contenido son cosas distintas. (La variable y su valor son conceptos distintos. No es lo mismo el continente que el contenido. No es lo mismo un vaso de agua que el agua de un vaso.)

En un fichero podemos ver (con `cat`) una carta, una poesía, un programa en lenguaje *pascal*, un chiste, la altura del monte Everest, el guión de 'Aeropuerto 2035', etc. Aunque no sea tan intuitivo, podemos creer que un fichero contiene el programa en lenguaje máquina del comando `mail`, las notas (codificadas) del Ilmo(a). Ministro(a) de Educación, o el censo (comprimido) de pingüinos en la Antártida.

El sistema operativo no atribuye significado al contenido de los ficheros. No trata los ficheros de forma distinta en función de su contenido. Son los usuarios y sus programas los que tienen en cuenta esos posibles significados del contenido.

8.1. Valores y operaciones

La información se almacena como una **secuencia de octetos**. Un octeto puede tomar 256 valores distintos. Esos valores se suelen interpretar como los números comprendidos entre 0 y 255 inclusive.

> Un octeto está compuesto por 8 bits y un bit puede tomar dos valores. A los valores de los bits se les atribuye frecuentemente el significado 0 y 1, o *cierto* y *falso*.

Los octetos de un fichero se localizan por su desplazamiento respecto al comienzo. Del primer octeto de un fichero se dice que su desplazamiento respecto al comienzo es cero.

Son muchas las operaciones sobre los ficheros que el sistema operativo considera elementales.

- Se puede crear un fichero. Inicialmente el fichero será la secuencia vacía de octetos.
- Se puede escribir en el fichero añadiendo al final.
- Se puede situar uno al comienzo del fichero o en medio de él.
- Se puede leer. Avanza la situación de la próxima lectura o escritura.

- Se puede escribir donde se esté situado.
- Se puede pedir acceso exclusivo del fichero o de parte. Si lo conceden no podrá usarlo nadie más hasta que se libere.
- Se puede pedir acceso exclusivo como lector del fichero o de parte. Si lo conceden no podrá escribir nadie hasta que se libere.
- ...

No todas las operaciones que se nos pueden ocurrir son primitivas (elementales) del sistema operativo.

Por ejemplo, tenemos un fichero con 1000 octetos y queremos borrar los octetos cuyo desplazamiento está comprendido entre 200 y 299. Esta operación no es una primitiva del sistema operativo.

Muchos sistemas UNIX utilizan el código ASCII. El código ASCII relaciona los caracteres más usados con los números comprendidos entre 0 y 127. Es una simplificación asimilar octeto y carácter. Con esta simplificación se dice que un fichero es una secuencia de caracteres.

8.1.1. Fichero texto

Parte de los ficheros están destinados a ser leídos por personas. La forma más sencilla de presentar un fichero es a través de una pantalla o imprimiéndolo en una impresora. Las versiones básicas de pantalla e impresora se comportan como los teletipos y las máquinas de escribir antiguas.

El **texto** aparece como una **secuencia de líneas**.

> Tanto en el teletipo como en la máquina de escribir la cabeza de escritura tiene que retroceder y el soporte del papel girar. A estas operaciones se las conoce como *retorno de carro* y *cambio de línea*. El juego de caracteres ASCII tiene sendos códigos para estas operaciones (números 13 y 10). Algunos sistemas operativos almacenan ambos caracteres en los ficheros que van a imprimirse o a visualizarse.

En los sistemas tipo UNIX las líneas de texto son **secuencias de caracteres terminadas por un cáracter *cambio de línea*** (código ASCII 10).

> En los sistemas tipo UNIX cuando se imprime un fichero o se envía a pantalla, el sistema operativo añade el carácter *retorno de carro*.

A lo largo de estos apuntes o de la asignatura que los siga, los ficheros de tipo texto serán los protagonistas.

> Esta historia de *retorno de carro* y *cambio de línea* parece pasada de moda cuando el lenguaje *postscript* permite situar los caracteres por coordenadas, y con editores del tipo 'así-lo-ves-así-lo-tendrás' (*wysiwyg*).
>
> Los ficheros de tipo texto descritos tienen la ventaja de que la estructura de datos *secuencia* es sencilla y robusta. (Los ficheros *postscript* se representan como ficheros texto.)
>
> Para convertir entre el formato texto de UNIX y el de MSDOS se pueden usar las utilidades `unix2dos` y `dos2unix`.

8.2. Nombre

Los ficheros tienen nombre. Nombre del fichero y fichero son conceptos distintos.

Entre los nombres y ficheros no hay una relación *1 a 1* sino *n a m* . Veremos que un nombre (sencillo) puede corresponder a varios ficheros. Distinguiremos estos ficheros como a los tocayos con los apellidos. Un fichero puede tener varios nombres, aunque lo más frecuente es ficheros con un solo nombre.

En los sistemas UNIX más antiguos la longitud de los nombres estaba limitada a 14 caracteres. La mayor parte de los sistemas UNIX actuales permiten nombres de ficheros de más de 80 caracteres de longitud.

Conviene hacer buen uso de esta libertad para elegir los nombres de los ficheros. El criterio es el mismo que para las variables u otros objetos en lenguajes de programación. Si la vida (duración) del fichero o de la variable va a ser corta y su nombre sólo interesa a un usuario, el nombre tenderá a ser breve. Según aumente la vida (duración) del fichero o de la variable y según aumente el número de usuarios a los que interesa, puede ser necesario hacer más largo el nombre del fichero o de la variable.

> Si dos ficheros tienen nombre largo que se diferencia en los primeros caracteres es posible que admitan un recorte en sus nombres. Si la diferencia de los nombres se sitúa en los últimos caracteres quizá convenga tomar la parte común como nombre de un directorio que los contenga y eliminarla del nombre de ambos ficheros.

Es prácticamente imposible[1] que en el nombre de un fichero se incluyan los caracteres `0C` o `/` .

El carácter `0C` (su código ASCII es cero) se utiliza como elemento de terminación de las tiras de caracteres. Si un carácter indica dónde ha terminado una tira de caracteres, no puede formar parte de ella.

> Si las tiras de caracteres se representasen en estos sistemas mediante un *array* y un campo de longitud, el razonamiento no sería útil.

El carácter `/` se usa como separador de nombres en los sistemas tipo UNIX. (Veremos que separa nombres de directorios, pero mejor no liar a un lector nuevo.) Los programas del sistema operativo toman buen cuidado de no permitir la aparición del carácter `/` dentro de un nombre.

> He leído que en un intercambio de información entre un sistema VS y una máquina UNIX alguna vez se había colado un carácter `/` dentro de un nombre de fichero. Manipulando los octetos de un disco también es posible. Como experimento en un ordenador de pruebas, es válido. Como objetivo a conseguir en un entorno de producción lo considero masoquista.

Se debe evitar que formen parte del nombre de un fichero (y de otros objetos que iremos presentando) una serie de caracteres.

- El carácter blanco o espacio (código ASCII 32). Si tenemos un fichero cuyo nombre es `a␣␣c` y usamos el comando `ls` para ver los nombres de los ficheros que tenemos, podemos pensar que tenemos dos ficheros de nombres `a` y `c` respectivamente.
- Caracteres de control (código ASCII menor que 32 ó igual a 127). En general, los caracteres de control no aparecen representados en pantalla de forma clara.
- `*` `?` `[` pueden dar lugar a problemas al ser interpretados por el intérprete de comandos. Supongamos que alguien tiene un fichero de nombre `*` y quiere borrarlo. Si teclea *borrar* `*` borrará todos los ficheros. (Tiene solución: *borrar* `'*'` .)

[1] Me da repelús poner eso de imposible, vamos a dejarlo en muy difícil.

- `< > ; | ( ) ’ " & ^ ! \ ‘`
  también pueden dar lugar a problemas por ser interpretados por el *intérprete de comandos*. Quedará más claro según conozcamos su significado.

La mayor parte de los nombres de ficheros estarán formados sobre todo por letras minúsculas y también mayúsculas, cifras, puntos, guiones y subrayados. No se considera que los ficheros tengan *extensión* en el nombre.

8.3. Estructura

En estos apuntes, en general explico **qué** hacen los comandos, qué hace el intérprete de comandos, qué hace el sistema operativo. No explico **cómo** lo hacen.

Voy a explicar la estructura de los ficheros en los sistemas tipo UNIX. El interés de la siguiente explicación no está en el mecanismo, sino en que ofrece un modelo que ayuda a entender el comportamiento de los ficheros y de algunos comandos (y de los permisos).

Asociada con un fichero hay información. Con `cat` vemos el contenido del fichero. Esa no es toda la información asociada a un fichero. Si hacemos `ls -l` vemos los atributos del fichero.

> Un fichero cuyo tamaño sea 0 también tiene información asociada. Puede ser útil. El hecho de existir o no supone un bit. Tiene un nombre. Tiene una fecha de última modificación. Puede servirnos para recordarnos cuándo sucedió algo.

En el ordenador hay una primera tabla (fig. 8.1) y en cada elemento de esa tabla hay una pareja: *nombre-de-fichero - número*. Esa primera tabla es consultada para ejecutar el comando `ls` sin opciones.

El número de la primera tabla nos lleva a una posición de una segunda tabla, la tabla de *inodos*. En el elemento correspondiente de la tabla de *inodos* se almacenan los atributos del fichero (salvo el nombre) y unos punteros que permiten acceder (directa e indirectamente) al contenido del fichero. Toda la información que aparece al hacer `ls -l` salvo el nombre está almacenada en el *inodo*. La relación entre el *inodo* y el fichero es una relación *1 a 1*.

Desde otros nombres se puede acceder al mismo *inodo* y fichero. En el ejemplo de la figura aparecen tres accesos. En el *inodo* hay un campo en el

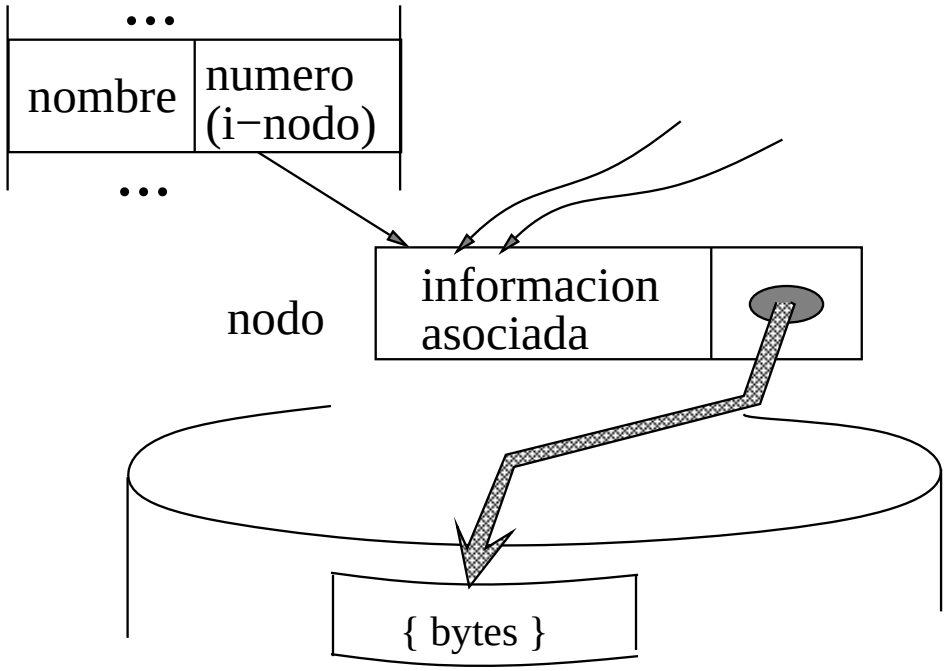


Figura 8.1: Estructura de los ficheros.

que se lleva la cuenta del número de nombres del fichero. Esta información es redundante, pero permite que el sistema funcione más rápidamente.

El número del *inodo* es único en cada disco virtual o partición de disco.

8.4. Operaciones básicas con ficheros

8.4.1. `cp`

`cp f1 f2` copia el fichero `f1` creando un nuevo fichero de nombre `f2` y cuyo contenido es una copia del contenido de `f1`. Si el fichero `f2` ya existía, borra su contenido y pone como contenido una copia del contenido de `f1`.

El nombre del comando viene de *copy*.

8.4.2. `ln`

`ln f1 f3` añade el nombre (enlace) `f3` al fichero `f1` . Un nombre no es una copia, no es un fichero nuevo. Sigue habiendo los mismos ficheros que antes. Cualquier cambio que se haga al fichero a través de un nombre aparece a través del otro (u otros) nombre (o nombres). El primer nombre y sucesivos nombres añadidos mediante `ln` son indistinguibles.

El nombre del comando viene de *link*.

8.4.3. `mv`

`mv f4 f5` cambia el nombre del fichero `f4` y pasa a ser `f5` .

El nombre del comando viene de *move*.

8.4.4. `rm`

`rm f6` quita el nombre (enlace) `f6` al fichero correspondiente. Si el nombre eliminado es el último o único que tenía el fichero, se borra quedando disponible su inodo y bloques de contenido para otros ficheros.

No es exacto decir que `rm` borra ficheros. No lo hace siempre, pero como la mayoría de los ficheros tienen sólo un nombre, casi siempre `rm` borra el fichero.

El nombre del comando viene de *remove* (borrar).

8.4.5. Un ejemplo

Tenemos un fichero, cuyo nombre es `f` (figura 8.2). La opción `-i` del comando `ls` nos presenta el número del *inodo* de ese fichero.

```
$ls -ix
  17 f
$cat f
hola


$cp f cf
$ls -ix
  30 cf   17 f
```

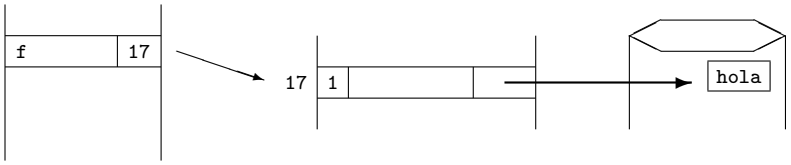


Figura 8.2: Un fichero, un nombre.

Copiamos el fichero `f` y a la copia le llamamos `cf` . En la figura 8.3 mostramos el resultado. Tenemos dos nombres, dos *inodos* y dos grupos de bloques distintos con su información (el texto `hola`). A cada *inodo* sólo se accede a través de un nombre.

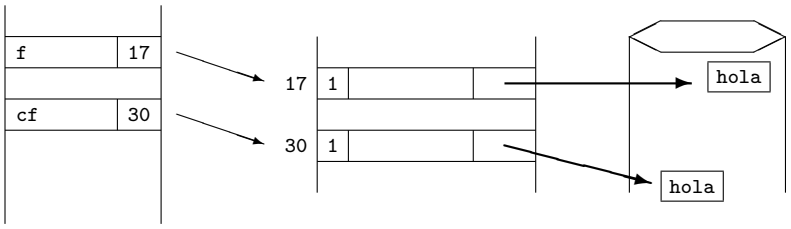


Figura 8.3: Dos ficheros, dos nombres.

```
$ln f lf
$ls -ix
  30 cf   17 f    17 lf
```

Creamos un segundo nombre, `lf` , para el fichero de nombre `f` . En la figura 8.4 mostramos el resultado. Tenemos tres nombres, dos *inodos* y dos grupos de bloques distintos. Al *inodo* cuya posición es `17` se accede a través de dos nombres: `f` y `lf` .

```
$mv f mf
$ls -ix
  30 cf   17 lf   17 mf
```

Cambiamos el nombre `f` por el nuevo nombre `mf` . Seguimos teniendo la misma estructura (fig.8.5). El cambio ha sido mínimo.

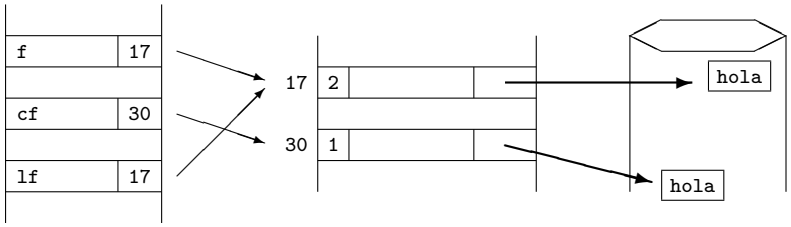


Figura 8.4: Dos ficheros, tres nombres.

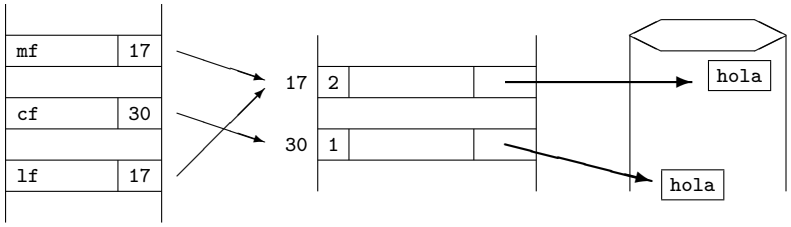


Figura 8.5: Hemos cambiado un nombre.

```
$rm mf
$ls -ix
  30 cf   17 lf
```

Se borra **el nombre** `mf` . Como el *inodo* correspondiente sigue accesible a través de otro nombre (`lf`) no se borra el *inodo* ni los bloques del fichero. Tenemos dos nombres, dos *inodos* y dos grupos de bloques distintos. El resultado se muestra en la figura 8.6.

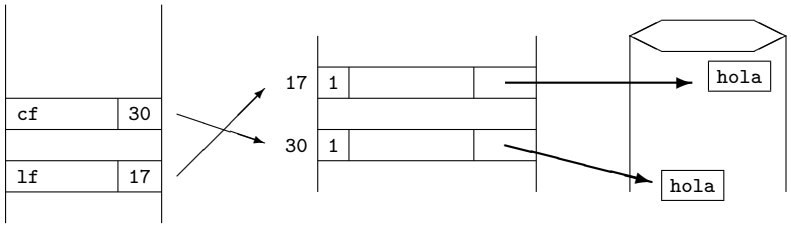


Figura 8.6: Hemos borrado un nombre.

```
$ cat lf
hola
```

El fichero que inicialmente se llamaba `f` sigue existiendo. El comando `ln` crea (añade) nombres de primera categoría. Basta con uno de ellos para mantener la existencia del fichero.

Supongamos que modificamos el contenido de un fichero (`cf` o `f`) después de llegar a la situación de la figura 8.4. ¿Qué aparecerá en la pantalla al ejecutar el comando `cat` ?

> La forma más natural de modificar el contenido de un fichero es editándolo, con `vi` por ejemplo, o copiando el contenido de otro. Para modificar `f` escribimos `cp queseYo f` .
>
> Si nos apoyamos sólo en lo que hemos visto hasta ahora, podemos cambiar el contenido del fichero salvando un mensaje.
>
> ```
> mail
> s f
> x
> ```

Hemos visto que el comando `mv` es el que menos trabaja. El comando `ln` tampoco hace mucho: pide una nueva línea en la tabla de nombres, la rellena, e incrementa el número de accesos al `inodo` correspondiente. El comando `cp` es el que más trabaja: escribe en las tres partes del sistema de ficheros. El comando `rm` podemos situarlo en cuanto a trabajo entre `ln` y `cp`, más cerca del primero si el fichero tiene más de un nombre y del segundo en caso contrario.

8.4.6. Opciones de `cp`, `ln`, `mv` y `rm`

`rm f6 f8 f9` borra los nombres de fichero indicados.

En algunos casos `rm` pide confirmación antes de borrar un fichero.

`rm -f f6 f8 f9` borra sin pedir confirmación. (Viene de *force*.)

`rm -i f?` antes de borrar cada fichero (nombre mejor dicho) pide confirmación. Se confirma con una `y` (de *yes*). Viene de *interactive*.

Se ha ido ampliando la funcionalidad de `cp`, `ln`, `mv` y `rm` .

`cp -p ...` mantiene en el fichero destino las fechas, dueño y grupo del fichero origen. (Viene de *preserve*.)

8.5. Enlaces simbólicos

`ln -s f1 f11` crea un enlace **simbólico** de nombre `f11` para acceder al fichero de nombre `f1`.

```
$ln -s f1 f11
$ls -l
-rw-r--r--   1 a0007        150 Aug 10 12:28 f1
lrwxrwxrwx   1 a0007          2 Aug 10 12:28 f11 -> f1
$ls -F
f1    f11@
$file f11
f11: symbolic link to f1
```

Con `ls -l` aparece una `l` en la primera columna de los enlaces simbólicos. Con `ls -F` vemos un carácter `@` (arroba) a continuación del nombre. También podemos saber que un nombre corresponde a un enlace simbólico con `file` .

Los enlaces simbólicos no crean copias de los ficheros. Si modificamos el fichero `f1` y hacemos `cat f11` veremos la modificación.

Los enlaces simbólicos son nombres de segunda categoría. Si `f1` no tiene otro nombre de primera categoría y lo borramos (`rm`), desaparece el fichero. `f11` no es capaz de garantizar su existencia.

```
$rm f1
$cat f11
cat: f11: No such file or directory
$
```

8.6. Ejercicios

Ejercicios sobre `cp`, `ln`, `mv` y `rm` :
94f.1 (pág. 296), 94j.1 (pág. 302), 95f.1 (pág. 314), 96s.1 (pág. 344), 98j.4 (pág. 374), 98s.7 (pág. 380). 93f.2 (pág. 278), 98f.1 (pág. 368).

Capítulo 9

El editor vi

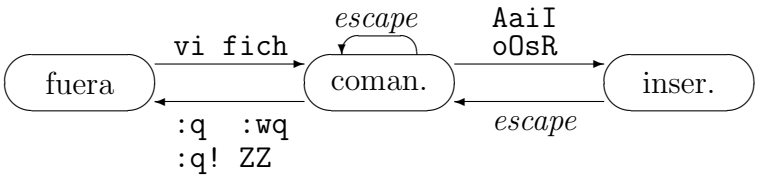
Para crear o modificar ficheros de tipo texto podemos usar el editor `vi` . También nos puede servir para hojear un fichero sin modificarlo.

> Es discutible la conveniencia de explicar un editor, y su elección. Para que un usuario pueda practicar es casi imprescindible que conozca algún editor.
>
> Incluyo el editor `vi` por varias razones. Está muy extendido. Permite un uso rápido (una vez conocido). Tiene cierta regularidad: combina movimientos con acciones. Permite búsquedas mediante *expresiones regulares*. Incorpora muchas posibilidades del entorno UNIX: filtrado y `:r!` . Permite definir (y usar) macros.
>
> A `vi` se le encuentran inconvenientes, p. ej. la falta de ayudas visuales. Además bastantes veces se encuentra con que es el segundo editor que aprende un usuario. El aprendizaje de un segundo editor casi siempre lleva un esfuerzo que no se esperaba.

Esperamos que un editor nos permita: introducir texto, movernos, borrar parte del texto, buscar un cierto contenido, traer el contenido de otros ficheros. Además, es fácil que un editor tenga muchas formas de hacerlo, y admita otras posibilidades.

Para hacer rápido el uso de los editores se suele acortar el nombre de los comandos, o teclas pulsadas para que se realice una acción. Usando editores, los comandos tecleados (casi siempre) se obedecen inmediatamente sin esperar al *retorno de carro*.

Como se diseñan editores con más comandos que caracteres tiene el teclado, se recurre a comandos de más de un carácter y al uso de *modos*.



9.1. Modos de `vi`

Dependiendo de en qué modo se encuentre el editor `vi` el efecto de pulsar una tecla será distinto.

El editor `vi` se entiende con dos *modos*: *modo-inserción* y *modo-comando* (o modo-corrección).

Para editar un fichero, normalmente se teclea `vi` *nombreDeFichero* . Aunque es posible comenzar una sesión de edición sin dar nombre del fichero, es mejor ponerlo. Más tarde podemos cambiarlo.

La sesión de edición comienza en *modo-comando*.

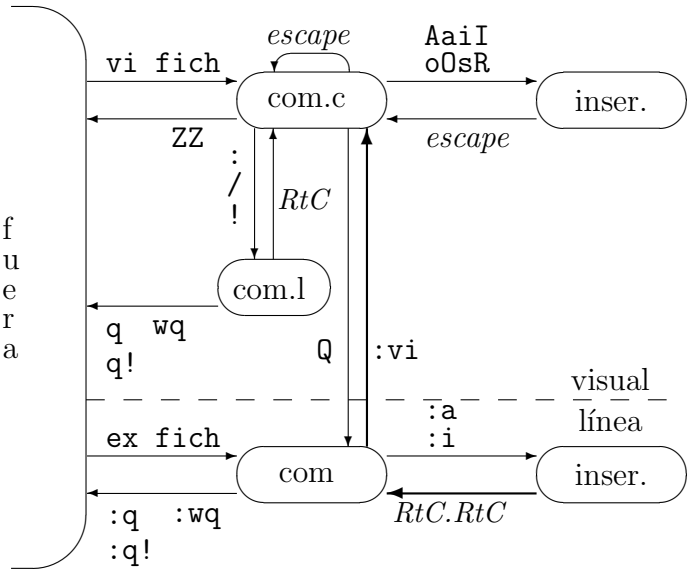
Si en medio de una sesión de edición no sabemos en qué modo estamos, basta teclear *escape*[1]. Si estábamos en modo inserción pasamos a modo comando. Si estábamos en modo comando, no hace nada. Después de teclear *escape* estamos seguros de estar en modo comando.

> Si tecleamos el carácter *escape* en modo comando, sonará el terminal, o la pantalla destellará (poniéndose en vídeo inverso por un tiempo breve). Cualquiera de las dos posibilidades es molesta. Conviene no dar al tuntún caracteres *escape* para no molestar.
>
> En algunos teclados la situación del carácter *escape* puede no ser obvia. Antes de editar, búscalo.

De modo comando podemos pasar a modo inserción con alguno de los siguientes caracteres: `A`, `a`, `i`, `I`, `o`, `O`, `s`, `r` o `R` .

Podemos querer acabar una sesión de edición después de modificar el fichero. Lo haremos tecleando `:wq` . (`ZZ` es lo mismo que `:wq` .) En otras ocasiones sólo habremos ojeado el fichero. Si queremos salir sin modificarlo tecleamos `:q` . Cuando hayamos realizado cambios en (la copia de) el fichero, y queramos dejar el fichero como estaba debemos salir tecleando `:q!` .

[1] *escape* es un carácter y normalmente le corresponde una tecla.



No es conveniente acostumbrarse a teclear `:q!` de forma automática, sin pensar. Una buena costumbre es trabajar más despacio, y pensar un poco antes de realizar acciones que puedan comprometer algo importante. (En este caso una sesión de edición.)

9.1.1. Más modos de vi

Un usuario de `vi` puede usar este editor durante cien horas sin necesidad de más guía que la dada en la sección anterior. Pero es posible que se salga de los dos modos descritos inicialmente. Es posible pero infrecuente.

El editor `vi` tiene dos *grandes modos*: *visual* y *línea*. Normalmente estamos en modo *visual*: *visual.comando visual.inserción*. Si accidentalmente pasamos a *modo línea.comando*, podemos volver con `:vi`*retornoDeCarro* a *modo visual*.

Más remota es la posibilidad de que casualmente lleguemos a *modo línea.inserción*. En ese caso, con: *retornoDeCarro*.*retornoDeCarro*`:vi` *retornoDeCarro* volvemos a *modo visual*.

¿Cómo hemos podido pasar a modo línea?. Lo más probable que por haber tecleado `Q` sin querer en modo comando.

¿Cómo podemos saber que estamos en modo línea?. El cursor está en la última línea de la pantalla. Si damos un *retorno de carro* vuelve a

aparecer : y seguimos en la última línea de la pantalla. La pantalla desplaza el texto hacia arriba. Si tienes curiosidad pruébalo.

En algún clónico de `vi` han quitado el modo línea.

Entre la primera y la segunda figura, el *modo comando* (visual) se ha desdoblado en comando-corto y comando-largo. En los comandos largos el cursor baja a la última línea de la pantalla. Hasta que no tecleamos *retorno de carro* el comando no se obedece.

Términos que aparecen en vi

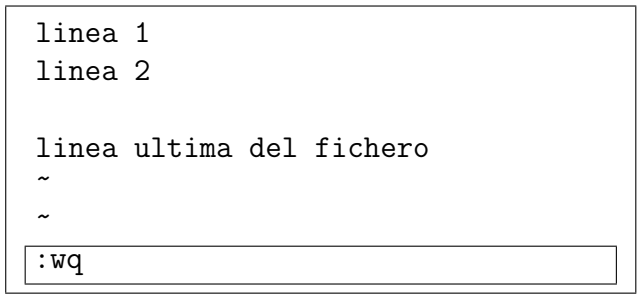
En `vi` aparecen una serie de objetos: carácter, línea, pantalla, fichero, palabra, frase, párrafo, sección, expresión regular, búfer (depósito); y acción: filtrar. Los iremos conociendo en este capítulo y en el de ampliación de `vi` .

9.2. La apariencia

Normalmente, cuando editamos un fichero, a cada línea del fichero le corresponde una línea de la pantalla. Si el fichero es pequeño, o si la pantalla presenta el final del fichero, algunas líneas de la pantalla no se corresponden con ninguna línea del fichero. Esto se representa mediante un carácter tilde (~) en la columna 1.

Es frecuente que después del último cáracter visible de cada línea esté el cambio de línea, y que cada posición anterior en la línea sin caracteres visibles corresponda a un carácter blanco.

La última línea está reservada para los comandos largos y para información que nos dé el editor `vi`.



9.2.1. ¿engaña?

Sucede a veces (pocas, creo), que las cosas no son lo que parecen.

- Puede haber ficheros en los que en la primera columna haya una tilde (`~`) .

  ¿Quién nos dice que el fichero del ejemplo anterior no tiene 6 líneas?

- Puede que haya una línea más larga que la anchura de la pantalla, y con los caracteres visibles puestos de tal forma que parezcan varias.

  Es posible que la tira `linea 2` sea el final de la línea primera (o la mitad).

- Puede que haya caracteres blancos y/o tabuladores después del último carácter visible de cada línea.

  Esto ha causado más de un error en un programa que no lo prevé.

- Donde suponemos que hay varios caracteres blancos, puede que haya menos de los que suponemos y algún(os) tabulador(es).

  Por ejemplo, entre `linea` y `del` puede haber un tabulador o un tabulador y un carácter blanco.

  > Además, los textos con tabuladores pueden tener distinta apariencia dependiendo del estado (programable) de la pantalla, de la ventana o de la impresora.

- Si el fichero tiene caracteres de control, no imprimibles, utiliza una representación del estilo `^Z` . En principio podemos dudar si esa secuencia corresponde a dos caracteres o al CONTROL-Z.

Estáticamente, viendo y sin actuar, no podemos estar seguros del contenido del fichero.

Moviendo (el cursor) o modificando el fichero se rompe la ambigüedad, podemos estar seguros del contenido del fichero.

En el ejemplo anterior, borramos el primer carácter de la primera línea, y vemos que la `l` de la segunda línea se mueve y va a parar a la última columna de la primera línea, y que también se desplaza el resto de la línea 2 de la pantalla. Conclusión: las dos primeras líneas de la pantalla se deben a una sola línea del fichero.

```
linea 1                          l
inea 2

linea ultima del fichero
~
~
```

Estas situaciones no son frecuentes, y la actitud propuesta es trabajar confiados. (Confiados, pero sin dormirse).

9.3. Comandos

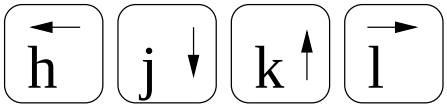
9.3.1. Movimientos

`h` mueve el cursor hacia atrás un carácter. No sale de la línea actual.

`j` mueve el cursor a la línea siguiente, a ser posible a la misma columna.

`k` mueve el cursor a la línea anterior, a ser posible a la misma columna.

`l` mueve el cursor adelante un carácter. No sale de la línea actual.



`h`, `j`, `k` y `l` están a mano, en las posiciones en que normalmente descansa la mano derecha. (Un poco molesto para los zurdos.)

> El uso de las flechas no suele dar problemas, pero considero preferible el uso de `h`, `j`, `k` y `l`. Resulta más rápido. En algunos sistemas, las flechas generan una secuencia del estilo *escape*`[A`*retornoDeCarro* . Estas secuencias casi siempre son correctamente interpretadas por el editor `vi`, pero en ciertas circunstancias no.

`w` mueve el cursor hacia adelante una palabra. El comando viene de *word* (palabra en inglés).

retorno de carro mueve el cursor a la primera posición no blanca de la línea siguiente.

`^F` mueve el cursor hacia adelante casi una pantalla. Viene de *forward*.

> Con `^F` queremos representar CONTROL-F, es decir el carácter enviado al ordenador al presionar (manteniendo) la tecla de CONTROL y luego la `f`.

`^B` mueve el cursor hacia atrás casi una pantalla. Viene de *backward*.

`1G` lleva el cursor al principio del fichero. Viene de *go*.

`G` lleva el cursor al final del fichero.

9.3.2. Pasan a modo inserción

`I` inserta caracteres al comienzo de la línea.

`i` inserta caracteres inmediatamente antes del cursor.

`a` añade caracteres inmediatamente después del cursor.

`A` añade caracteres al final de la línea.

`O` abre una línea antes (encima) de la que estaba el cursor y queda en modo inserción.

`o` abre una línea después (debajo) de la que estaba el cursor y queda en modo inserción.

> Usando el editor `vi` a veces podemos asociar mayúscula con arriba y minúscula con abajo, según esta(ba)n situados los caracteres en las máquinas de escribir.
>
> Otras veces mayúscula y minúscula se asociarán con grande y pequeño, como en el caso de `a A i I` por el desplazamiento del cursor.
>
> En general, en UNIX, es más frecuente el uso de las minúsculas.

Si estamos en modo inserción, no hace hace falta salir a modo comando para un pequeño borrado. En modo inserción, *backespace* retrocede y borra un carácter.

> En modo inserción el carácter CONTROL-W retrocede y borra una palabra. En algunas versiones de `vi` el cursor retrocede y al teclear se escribe donde estaba la palabra. En otras versiones desaparece la palabra.

9.3.3. Borrado

`x` borra el carácter que está bajo el cursor.

Si es posible, el cursor queda sobre el carácter siguiente de la misma línea; si estaba al final de la línea, el cursor queda sobre el anterior al carácter borrado.

Podemos acordarnos asociando la `x` a tachar.

`X` borra el carácter situado antes del cursor.

`dd` borra la línea sobre la que se encuentra el cursor.

Si es posible, el cursor pasa a la línea siguiente, sobre el primer carácter visible (no blanco).

`dw` borra (el resto de) la palabra situada bajo el cursor.

Si es posible sin cambiar de línea, el cursor queda al comienzo de la palabra siguiente a la borrada.

`J` junta las líneas actual y siguiente. Dicho de otra forma, borra el cambio de línea de la línea sobre la que se encuentra.

El cambio de línea y caracteres blancos al inicio de la línea siguiente se substituyen por un carácter blanco. El cursor queda sobre ese espacio blanco.

9.3.4. Deshace (el último cambio)

`u` deshace la última modificación.

`uu` la segunda `u` deshace lo que hizo la primera `u`. El resultado de `uu` es igual a no hacer nada.

`uuu` equivale a una sola `u`. Solo importa si el número de `u`es consecutivas es par o impar.

> Algún programa clónico del editor `vi` ha modificado el comportamiento de `u` .
>
> En el editor `vim`, la segunda `u` deshace la penúltima modificación; la tercera `u` deshace la tercera modificación contando por el final; etc.
>
> En algunos sistemas LINUX, `vi` es un alias (un segundo nombre) de `vim`.

9.3.5. Búsqueda

/tiraDeCaracteres retornoDeCarro lleva el cursor a la siguiente aparición de la *tiraDeCaracteres* nombrada.

Al teclear `/` el cursor baja a la última línea de la pantalla. Según vamos tecleando, la tira de caracteres buscada aparece en la línea inferior de la pantalla. Mientras no tecleemos *retornoDeCarro* podemos corregir la tira a buscar usando el carácter *backespace*. Cuando tecleamos *retornoDeCarro* se realiza la búsqueda.

La búsqueda es cíclica. Al llegar a la última línea del fichero la búsqueda continúa en la primera línea.

`n` repite la última búsqueda.

9.3.6. Para salir

Mientras dura la sesión de edición no se trabaja con el fichero sino con una copia. Si damos por buenas las modificaciones realizadas en la copia, ésta se escribe sobre el fichero.

`:q` *retornoDeCarro* sale de la sesión de edición sin escribir (si no hemos modificado la copia del fichero).

`:q!` *retornoDeCarro* sale de la sesión de edición sin escribir aunque se pierdan modificaciones que hemos hecho en la copia del fichero.

`:wq` *retornoDeCarro* sale de la sesión de edición escribiendo los últimos cambios.

9.3.7. Uso desde `mail`

`~v` Si estamos usando el comando `mail` para enviar un mensaje, podemos abrir una sesión de edición tecleando `~v` con el carácter tilde [2] (`~`) en la columna 1. Veremos la parte de mensaje que hayamos escrito.

Cuando acabemos la sesión de edición (`:wq`), seguiremos dentro de una sesión de envío de correo. Cuando tecleemos CONTROL-D o `.` (punto) en la columna 1, el mensaje se enviará a su destino.

[2] En algunos teclados no se encuentra el carácter tilde. Se puede conseguir con la tecla de componer y `126` .

Capítulo 10

Directorios

Los papalagi guardan cosas dentro de cajas, dentro de cajas, dentro de cajas, ...

Los *directorios* son contenedores de nombres. A los ficheros accedemos por su nombre. Decir que un fichero *está* en un directorio no es exacto, pero casi siempre nos basta para expresarnos y razonar. Además los directorios tienen nombre.

Los *directorios* pueden contener (nombres de) ficheros y directorios.

```
$ ls
desenlace indice    nudos     presenta
```

`ls` nos presenta una serie de nombres de objetos que están accesibles.

```
$ ls -l
total 15
-rw-r--r-- 1 a0007     186 Sep 31 16:33 desenlace
-rw-r--r-- 1 a0007    4795 Sep 31 16:34 indice
drw-r--r-- 2 a0007    1024 Oct  2 16:32 nudos
-rw-r--r-- 1 a0007    3038 Sep 31 16:33 presenta
```

`ls -l` nos da más información de cada objeto. En particular, un guión (`-`) en la primera columna nos indica que el nombre corresponde a un fichero. Un

carácter `d` en la primera columna nos indica que el nombre corresponde a un directorio.

```
$ ls -F
desenlace   nudos/
indice      presenta
$
```

Una forma más sencilla de saber si un nombre corresponde a un directorio es la opción `-F` de `ls` porque añade un carácter `/` (eslás).

Conviene precisar un poco más: los *directorios* pueden contener *nombres* de ficheros y de directorios.

Hay varias razones para afinar:

- Parece más correcto decir que tenemos dos nombres para un fichero, cada uno en un directorio, que decir que el fichero está en dos directorios. (Ya sé que de las dos formas nos entendemos.)
- Cuando consideremos los permisos y queramos 'borrar un fichero', este matiz puede ser importante.
- De hecho, el directorio es la primera de las tablas mostradas al presentar la organización de los ficheros en UNIX.

Los directorios sólo tienen un nombre.

Al estudiar el comando `ln` vimos que un fichero puede tener varios nombres. Estamos refiriéndonos a nombres de primera categoría, no a los nombres creados con `ln -s` .

En las primeras versiones de UNIX (hasta la versión 6) se permitía que un directorio tuviese más de un nombre, pero esto hacía difícil evitar que se formasen ciclos. Si se forman ciclos con los directorios, el método del contador de referencias para saber cuándo se puede borrar un directorio no vale. Hay que acudir a técnicas de *recogida de migas* (*garbage collection*) que son lentas en memoria y más lentas sobre disco.

En los todos los directorios tenemos dos entradas:

`.` es una referencia en un directorio a sí mismo.

`..` es una referencia en un directorio a su *directorio padre* (el directorio que lo contiene).

> Al decir que los directorios sólo tienen un nombre nos olvidamos de `.` y `..` obviamente.

El conjunto de directorios de un sistema forma un árbol. Un árbol es un grafo dirigido (sin ciclos) y con un único elemento (raíz) al que no se llega desde otro. Al único directorio que no está contenido en otro se le llama *directorio raíz*.

En el directorio raíz `..` es una referencia a sí mismo.

El nombre del directorio raíz es `/` (un eslás o barra inclinada como las agujas del reloj a las siete y cinco).

Se llama *sistema de ficheros* al conjunto organizado de ficheros y directorios de una máquina. *Organizado* hace referencia a la estructura en árbol de los directorios e inclusión de ficheros en directorios.

Directorio actual

En cada momento el intérprete de comandos **está** en un directorio. Se llama *directorio actual* a ese directorio.

La afirmación puede hacerse más general:
- En cada momento todo proceso está en un directorio.

> ....
> - El intérprete de comandos es un proceso.
> - El intérprete de comandos en cada momento está en un directorio. [1]

> El *directorio actual* es una variable local a cada proceso. Como no hemos llegado al tema de procesos nos quedamos con la afirmación del comienzo de la subsección y guardamos ésta en el baúl de la culturilla.

Cuando comienza la sesión, el directorio actual es el campo sexto de la línea de `/etc/passwd` . A ese directorio se le llama *directorio de entrada*.

Nombres

Para hacer referencia a un fichero o a un directorio, enumeramos los directorios que atravesamos y los separamos por un carácter eslás (`/`) .

[1] Esto es un silogismo.

- Si la enumeración comienza por eslás (`/`), hemos partido desde el *directorio raíz*.
- En caso contrario, hemos partido del *directorio de actual*.

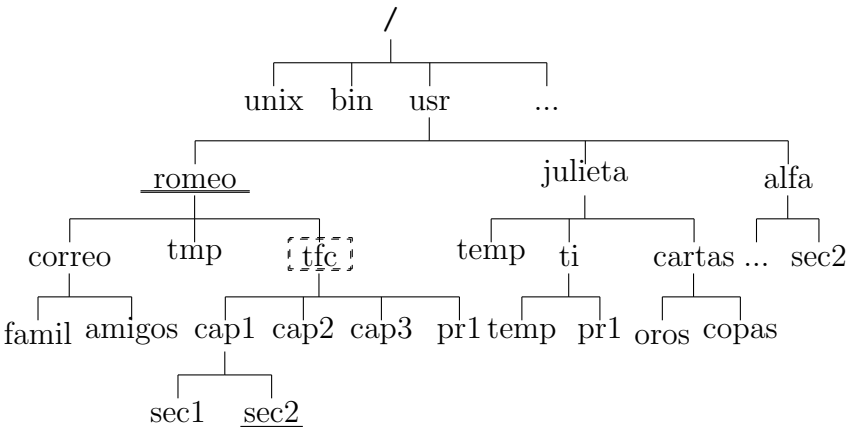


Figura 10.1: Un sistema de ficheros

Vamos a ver unos ejemplos en el *sistema de ficheros* de la figura 10.1. Suponemos que el directorio actual es `tfc` . Si queremos hacer referencia la fichero `sec2` que aparece subrayado podemos hacerlo de dos formas.

`cap1/sec2` es el nombre *relativo* al directorio actual.

`/usr/romeo/tfc/cap1/sec2` es el nombre *absoluto*.

Si cambia el *directorio actual*, cambia el nombre relativo y no cambia el nombre absoluto. El nombre relativo depende del proceso y del momento. El nombre absoluto es independiente de ambos factores.

> Aunque es infrecuente, a veces aparecen otros nombres para el fichero `sec2` subrayado: `cap2/../cap1/sec2` o `./cap1/./sec2` . En estos otros nombres aparecen las notaciones del directorio actual (`.`) o del directorio padre (`..`) .

Hay también dos formas de hacer referencia al otro fichero `sec2`.

`../../alfa/sec2` es el nombre *relativo* al directorio actual.

`/usr/alfa/sec2` es el nombre *absoluto*.

Vamos a suponer que en el ordenador en el que se encuentra el sistema de ficheros de la figura 10.1 el contenido de `/etc/passwd` es el siguiente.

```
...
alfa:x:3001:200:Alfa Centauro:/usr/alfa:/bin/bash
julieta:x:3003:202:Julieta Montesco:/usr/julieta:/bin/bash
romeo:x:3002:201:Romeo Capuleto:/usr/romeo:/bin/bash
```

> Vemos que al usuario `romeo` le corresponde el directorio de entrada `romeo`, e igualmente a `julieta` y `alfa` les corresponden directorios de entrada de igual nombre.
>
> Considero buena costumbre esta coincidencia de nombres de usuario y directorio. Salvo algún despiste inicial, no debe crear problemas.

El intérprete de comandos se encarga de substituir `$HOME` por el directorio de entrada de un usuario.

El usuario *romeo* puede dar como nombre absoluto de su fichero subrayado `$HOME/tfc/cap1/sec2` . El usuario *alfa* puede dar como nombre absoluto del fichero tocayo del anterior `$HOME/sec2` .

Algunas variantes de intérprete de comandos (al menos `csh`, `tcsh` y `bash`) interpretan el carácter tilde (`~`) como `$HOME` y la secuencia `~`*usuario* como el directorio de entrada de ese *usuario*.

El usuario *romeo* puede referirse a los dos ficheros de nombre `sec2` como `~/tfc/cap1/sec2` y `~alfa/sec2` .

10.1. cd , pwd , mkdir y rmdir

`cd dir` cambia el *directorio actual* a `dir` .

`cd` cambia el *directorio actual* al directorio de entrada (`$HOME`) .

El cambio de directorio lo realiza el intérprete de comandos. No usa otro fichero ejecutable. Se dice que `cd` es un comando *interpretado*.

> El intérprete de comandos es como un personaje que toma nota de nuestras peticiones y normalmente delega en otro personaje para realizar el trabajo. Si delegase la función de cambiar el *directorio actual* en otro personaje, como el directorio actual se almacena en una *variable local*, el resultado sería el cambio de una variable local de otro personaje. Dicho cambio delegado *no afectaría al intérprete de comandos*, y queremos que afecte.

`pwd` escribe el camino desde la raíz hasta el directorio actual.

`mkdir dir1` crea el directorio de nombre `dir1` . Admite varios parámetros y en ese caso creará varios directorios.

`mkdir d1 d1/d2 d1/d2/d3` crea los directorios nombrados.
`mkdir -p d1/d2/d3` hace lo mismo. La opción `-p` indica que se creen los directorios que faltan.

`rmdir dir` borra el directorio nombrado. Admite varios parámetros y en ese caso borrará varios directorios.

Un directorio no se puede borrar si contiene ficheros o directorios.

> Tampoco se puede borrar un directorio si en ese directorio está *montado* (enganchado) un sistema de ficheros. (De esto se verá algo en 20.10).

10.2. `ls` y otros

Podemos ver si un fichero está vacío o no con la opción `-a` del comando `ls` .

`ls -a` muestra también los ficheros cuyo nombre empieza por punto (.).

Con `ls -a` siempre veremos las referencias *punto* (`.`) y *dos puntos* (`..`). Si sólo aparecen estas dos el directorio se considera vacío.

```
$ ls -a
.           .delEditor  indice     presenta
..          desenlace   nudos
$
```

`ls -d` La opción `-d` hace que sólo se listen directorios.

`ls -R` **además** aplica `ls -R` a cada directorio que encuentra. La `R` viene de *recursive*. (Una opción `r` indicaría orden inverso (*reverse*)). Lista un subárbol.

```
$ pwd
/usr/romeo
$ ls -R
```

```
correo  tfc     tmp

correo:
amigos  famil

tfc:
cap1  cap2  cap3  pr1

tfc/cap1:
sec1  sec2
```

Hay otras formas de escribir lo que hay en un subárbol.

`find romeo -print` escribe todos los objetos del subárbol bajo el directorio `romeo` .

`du -a dir` escribe todos los objetos del subárbol bajo `dir` y lo que ocupan en bloques.

10.3. cp, ln, mv

Con la introducción de los directorios, los comandos `mv`, `cp` y `ln` se enriquecen admitiendo sendas formas abreviadas.

`cp f1 f2 dir` copia los ficheros `f1 f2` al directorio `dir` .

Podríamos expresar lo mismo con `cp f1 dir/f1` y `cp f2 dir/f2`. La forma anterior es más breve. Nos va a venir muy bien para escribir líneas del tipo `cp f* dir` .

`ln f1 f2 dir` crea sendos enlaces para los ficheros `f1 f2` con nombre `dir/f1` y `dir/f2` .

`mv f1 f2 dir` mueve los ficheros `f1 f2` al directorio `dir` .

Además, el comando `mv` se enriquece con funcionalidad nueva.

`mv dir1 dir2` cambia de nombre al directorio `dir1` . Pasa a llamarse `dir2` .

10.4. Ejercicios

Ejercicios con directorios, `mv`, `cp`, `ln` y `rm` :
93f.1 (pág. 278), 96f.1 (pág. 332), 96f.2 (pág. 332), 96s.4 (pág. 344), 99f.2

(pág. 386).

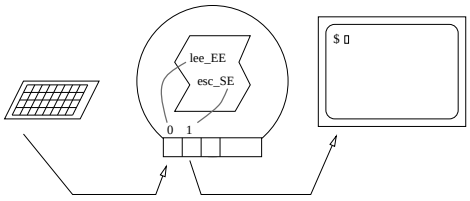
Capítulo 11

Redirección

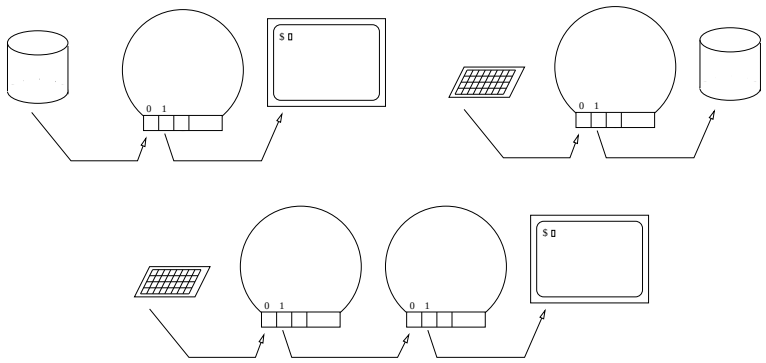
- Muchos comandos escriben por la salida estándar.
- Muchos comandos leen de la entrada estándar.
- Si no se modifica, la entrada estándar es el teclado.
- Si no se modifica, la salida estándar es la pantalla.

En el código de los programas (los comandos han sido programados) no se indica (casi nunca) el origen concreto de los caracteres de entrada ni el destino concreto de la salida.

En el código de los programas suele indicarse (p. ej.) leer un carácter de la entrada estándar. El programa se compila. En el momento de ejecutar, el programa dirige un proceso. Ese proceso tiene conectada la entrada estándar al teclado. Igualmente, en el programa suele poner (p.ej.) escribir cuatro caracteres en la salida estándar, y es en el proceso donde está la conexión de la salida estándar con la pantalla.



Es posible hacer que la entrada estándar se tome de un fichero y que la salida estándar vaya a un fichero. También es posible hacer que la salida estándar de un proceso vaya a la entrada estándar de otro proceso.



El intérprete de comandos es quien normalmente se encarga de establecer la conexión entre la entrada y la salida estándar con los ficheros o con otros procesos. Se llama *redirección* a este cambio del origen y destino de las lecturas y escrituras. El usuario indica al intérprete de comandos las redirecciones que desea mediante unos convenios.

- `< fich1` hace que la entrada estándar sea `fich1` .
- `> fich2` hace que la salida estándar sea `fich2` . Si existe el fichero `fich2` se pierde su contenido. Si no existe el fichero, se crea.
- `>> fich2` hace que la salida estándar sea `fich2` . Si existe el fichero `fich2` no se pierde su contenido. Se escribe a continuación. Si no existe el fichero, se crea.
- `<< centinela` hace que la entrada estándar sean las líneas que vienen a continuación, terminando al encontrar *retornoDeCarro* `centinela` *retornoDeCarro*
- `com1 | com2` hace que la salida estándar de `com1` se lleve a la entrada estándar de `com2`

Algunos usuarios consideran peligroso el comportamiento de `>` porque se puede borrar sin querer el contenido de un fichero valioso. Algunos intérpretes de comandos tienen una opción para restringir la actuación de las redirecciones de salida: `>` y `>>` .

Cuando está activa la opción `noclobber` el intérprete de comandos considera un error la redirección simple (`>`) sobre un fichero que existe, o la redirección doble (`>>`) sobre un fichero que no existe.

> ¿ Qué podemos hacer si nos abren una cuenta con la opción `noclobber` y preferimos el funcionamiento original?. Si esa opción se activa (`set noclobber`) en un fichero del que somos dueños, podemos convertir en comentario la línea anteponiendo una almohadilla (`#`). Si no somos dueños del fichero en que se activa la opción, podemos corregirlo poniendo `unset noclobber` en el fichero `$HOME/.cshrc` o `$HOME/.bash_profile` . (Depende del dialecto de nuestro intérprete de comandos).

Algunos comandos modifican su comportamiento en función de que su salida esté redirigida a un fichero o vaya a un terminal. Por ejemplo `ls -x` escribe los nombres de los objetos en columnas cuando la salida va a un terminal, y un nombre por línea cuando se redirige la salida a un fichero. Es poco frecuente.

11.1. Ejemplos

```
$cat >carta
Subject: prueba
una linea
otra
^D
```

Cuando el comando `cat` no tiene parámetros copia la entrada estándar a la salida estándar.

En el ejemplo anterior lo que vamos escribiendo por el teclado está redirigido al fichero `carta` . El carácter CONTROL-D en la columna primera actúa como *fin de fichero*. Hemos creado un fichero con tres líneas.

> Ésta no es la forma normal de crear ficheros. Sólo es un ejemplo de redirección, y una disculpa para mostrar un comportamiento nuevo del comando `cat`.

Si el fichero `carta` ya existía las tres líneas (`Subject: ... otra`) substituyen al contenido anterior del fichero.

`mail a0007 <carta` redirige la entrada estándar de `mail` al fichero `carta` . Es como si tecleásemos esas tres líneas.

`date >unaFecha` redirige la salida estándar del comando `date` al fichero `unaFecha` .

En principio, no parece muy útil esta información porque también la tenemos haciendo `ls -l` . Puede venirnos bien en el caso de que copiemos el fichero. En la copia, la fecha de última modificación cambia, mientras que la fecha guardada como contenido no cambia.

`who > quienes` redirige la salida estándar del comando `who` al fichero `quienes` . En el fichero `quienes` tenemos algo así como una foto, un registro, de quienes estaban usando el sistema en un momento dado.

`cal 2001 >unaOdisea` redirige la salida estándar del comando `cal` al fichero `unaOdisea` . En ese fichero guardamos el calendario del año 2001, año de la novela de Arthur Clarke.

`date ; who >saco` escribe la fecha (`date`) en la pantalla y redirige la salida estándar del comando `who` al fichero `saco` [1].

El carácter punto y coma (`;`) es un separador del intérprete de comandos. Hace el mismo papel que el *cambio de línea*.

La redirección sólo se produce a partir del punto y coma.

`(date ; who) > quienyCuando` redirige la salida de los comandos `date` y `who` al fichero `quienyCuando` .

La redirección afecta a los dos comandos porque están encerrados entre paréntesis. La salida de `who` va a continuación de la salida de `date`.

`(date ; who) >> quienyCuando` redirige la salida de los comandos `date` y `who` al fichero `quienyCuando` escribiendo a continuación de lo que había. Tenemos así dos fechas y dos listas de sesiones.

```
$ mail <<fin
1
x
fin
$
```

En la primera línea indicamos, con `<<`, al comando `mail` que la entrada estándar viene a continuación, y que acabará cuando se encuentre una línea con la tira de caracteres `fin` .

Las dos líneas siguientes son la entrada estándar para el comando `mail`. Presentan el primer mensaje (`1`) y dejan el buzón como estaba (`x`) .

Este mecanismo (`<<`) está diseñado para usarse en programas del intérprete de comandos. No ofrece ventajas si se usa tecleando. Trabajaríamos más

[1] un nombre trivial para un fichero de usar y tirar.

para conseguir lo mismo.

```
$ man mail | wc
Reformatting page. Wait... done
  1320  5419 51740
```

Redirigimos la salida estándar del comando `man` para que sea entrada estándar del comando `wc`. Queremos saber cuántas líneas tiene la información de manual de `mail` . Tiene 1.320 líneas (20 páginas de 66 líneas).

En el terminal aparece un mensaje, `Reformatting ... done`, que no parece salida normal del comando `wc` . Según lo que hemos visto, tampoco puede ser salida estándar del comando `man` porque está redirigida hacia el comando `wc`. Ese mensaje lo ha escrito el comando `man` por su *salida estándar de errores*.

> Es posible que un comando, o en general un programa, tenga su salida estándar redirigida y haya que enviar un aviso al operador. Si ese aviso se envía a la salida estándar puede pasar desapercibido e incluso puede perderse si es la entrada de un programa como `wc`.
>
> Se reserva un canal específico para los mensajes de error o avisos (*warning*), su número es el 2 y su nombre es *salida estándar de errores*.
>
> (Hoy llamo *operador* a la persona que está junto al sistema cuando se ejecuta un comando. Hace años (muchos en informática) era un puesto de trabajo junto a casi todos los ordenadores.)

```
cat quij|tr -cs  A-Za-z '\012' |sort -f|uniq -c|sort -nr
```

> Hay dos variantes del comando `tr`. Dependiendo de cuál se encuentre en su máquina, la forma adecuada de invocar `tr` puede ser:
> `... | tr -cs  '[A-Z][a-z]' '[\012*]' | ...`

Cinco comandos colaboran en una cadena de trabajo. Todos salvo el primero toman la información de la entrada estándar. Todos envían el resultado de su trabajo a la salida estándar. La salida estándar de los cuatro primeros va a la entrada estándar del comando siguiente. La salida estándar del quinto comando va a la pantalla.

- `cat quij` envía el contenido de un fichero a la salida estándar. Unos párrafos del Quijote.

- `tr -cs ...` recorta las palabras del texto que llega. Cambia (`tr`) todos los caracteres consecutivos que no sean (`-c`) letras (`A-Za-z`) por un solo (`-s`) cambio de línea (`'\012'`).
- `sort -f` ordena las palabras sin separar mayúsculas y minúsculas.
- `uniq -c` elimina las líneas repetidas consecutivas y cada línea va precedida por su frecuencia.
- `sort -nr` ordena las líneas de mayor a menor.

En la salida tenemos el vocabulario del texto de partida, ordenado por frecuencia de mayor a menor.

Este ejemplo puede parecer exagerado, y sin embargo no lo es (tanto). Al cabo de un tiempo, un usuario se acostumbra a pensar en los comandos como piezas con las que construir. Hemos construido una herramienta específica a partir de piezas de uso general.

UNIX invita al usuario a que, para resolver un problema, aproveche los programas que ya están escritos, y sólo escriba las partes imprescindibles. Combinando estas partes nuevas con el software ya escrito resuelve más rápidamente su problema. Y si los programas nuevos que escribe están bien diseñados es probable que también puedan reutilizarse.

La ejecución de estos comandos puede realizarse en paralelo. Este posible paralelismo se aprovechará si el ordenador dispone de varios procesadores, o solapando operaciones de entrada-salida con uso del procesador.

> Cuando el paralelismo es posible y no es obligado se habla de *concurrencia* (ejecución concurrente).

`who | sort > f` envía al fichero `f` la salida del comando `who`, sesiones en el sistema, ordenada alfabéticamente por nombre de usuario.

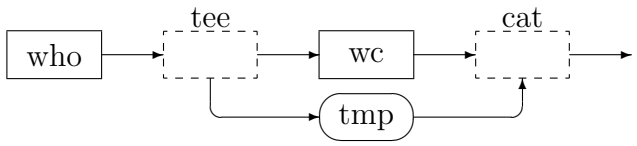
11.2. `tee`

```
$ who | tee tmp | wc -l | cat - tmp
       5
root     ttyb      Oct 12 12:12
p0160    tty099    Oct 12 16:55
```

```
p0160    tty021    Oct 12 16:58
a0007    tty041    Oct 12 17:24
gestor   ttyv008   Oct 12 10:27
$
```

Cuatro comandos trabajan en cadena. La salida de un comando sirve de entrada al siguiente.

- `who` describe las sesiones en el sistema. En el ejemplo hay 5 sesiones, dos corresponden al usuario `p0160`.
- `tee tmp` saca dos copias de los caracteres que lee. Una copia va a la salida estándar, y la otra al fichero `tmp` .
- `wc -l` cuenta las líneas que le llegan.
- `cat - tmp` concatena lo que lee por la entrada estándar (`-`) con el fichero `tmp` .



El comando `tee` nos sirve para sacar dos copias de una información, una para su proceso inmediato.

Capítulo 12

filtros

- Un **filtro** mecánico no deja pasar la arena.
- Un **filtro** óptico sólo deja pasar la luz roja.
- Otro **filtro** óptico gira 90^o el plano de polarización de la luz.
- Hay filtros para el aire en los automóviles, para el humo en los cigarrillos, para las personas en los casinos, etc.

Un filtro de información selecciona la información que pasa, o la transforma.

En el entorno UNIX, un filtro es un comando en el cual (la mayor parte de) la información fluye de la entrada estándar a la salida estándar.

Si tenemos los comandos:
`f1` : escribe el cuadrado de los números que lee.
`f2` : escribe el siguiente de los números que lee.
`cat num | f1 | f2` es semejante a

$f2(f1(num))$ o $(f2 \circ f1)(num)$

La composición de filtros con `|` (tubos, *pipes*) es semejante a la composición de funciones.

> Los comandos pueden modelizarse mediante funciones y autómatas. Es mucho más sencilla la composición de funciones que la de autómatas.

El dominio de definición de estos filtros será frecuentemente el conjunto de secuencias de líneas. Traduciendo: vamos a ver filtros que procesan ficheros de tipo texto.

Agruparemos los filtros según la operación que realizan.

Cortan:

... según la dirección horizontal

12.1. `head`

`head -`*número* *fichero* lista las *número* primeras líneas de un fichero.

Si se omite la opción `-`*número* , toma las 10 primeras líneas.

Si se omite el parámetro *fichero* , toma las n primeras líneas de la entrada estándar.

La mayor parte de los filtros admitirá nombrar un fichero como entrada. Cuando no se nombre el fichero tomará como entrada la entrada estándar.

> El comando `tr` es una excepción. No admite el nombre del fichero como parámetro.

12.2. `tail`

`tail` lista las diez últimas líneas de la entrada estándar.

`tail -`*número* *fichero* lista las *número* últimas líneas del fichero nombrado.

Podemos indicar el punto de corte del fichero en bloques (`b`), líneas (`l`), o caracteres (`c`), respecto al fin del fichero (`-`) o respecto al comienzo (`+`). Si no se indica la unidad, se supone que son líneas.

`tail -20` listará las veinte últimas líneas, `tail -n +101` listará partir de la línea 101 inclusive, `tail -30c` listará los treinta últimos caracteres.

> Algunas implementaciones antiguas fallan cuando se pide más líneas que las contenidas en los últimos 20 bloques.

12.3. split

`split` parte la entrada estándar en trozos de 1000 líneas, y crea ficheros de nombre `xaa`, `xab`, etc.

`split` *fichero* hace lo mismo con el fichero nombrado.

`split` no es propiamente un filtro, porque su efecto no se observa por la salida estándar. Se explica aquí porque su funcionalidad es próxima a la de los comandos anteriores y siguientes.

Se puede cambiar el tamaño de los trozos (`-`*número*). También se puede indicar que los nombres tengan otra raíz.

`split -200 f trozo.` parte el fichero `f` en trozos de nombre `trozo.aa`, `trozo.ab`, etc, de tamaño 200 líneas.

`split -200 - trozo.` hace lo mismo con la entrada estándar.

En muchos filtros se utiliza `-` para indicar la entrada estándar.

... según la dirección vertical

12.4. cut

Los cortes pueden hacerse siguiendo columnas (`-c`) o campos (`-f`) (de *field*). A continuación de `-c` irá una lista de columnas e intervalos de columnas separadas por comas. A continuación de `-f` irá una lista de campos e intervalos de campos separados por comas.

Al usar el comando `cut` las columnas o campos nombrados son los que aparecen en la salida estándar. Los campos y las columnas se numeran a partir de 1.

`cut -c1-10,21-` selecciona los caracteres 1 a 10 y del 21 al último de la entrada estándar. El intervalo abierto `21-` indica 'hasta el final de cada línea'.

`cut -d: -f1,5-7 fich` selecciona los campos 1, 5, 6 y 7 del fichero `fich` , tomando `:` como delimitador de campos.

Si no se pone la opción `-d` se supone que el delimitador es el carácter *tabulador*.

Con la opción `-f`, si en una línea no aparece el carácter delimitador, se escribe esa línea en la salida estándar. Con la opción `-s` (añadida a la opción `-f`) si en una línea no aparece el carácter delimitador, **no** se escribe en la salida estándar.

Entre dos delimitadores consecutivos, y entre el comienzo de línea y un delimitador, `cut` siempre contabiliza un campo (vacío). Veremos que `awk`, `join`, y `sort` a veces no.

Unen:

... poniendo a continuación

12.5. `cat`

`cat f1 - f4` lista consecutivamente `f1`, la entrada estándar y `f4` .

`cat` sin parámetros lista la entrada estándar (igual que `cat -`).

Su nombre viene de *con***cat***enate*.

... poniendo al lado

12.6. `paste`

Cada línea de la salida es la concatenación de las líneas de entrada.

`paste f1 f2 f3` La línea i-ésima de la salida esta formada por las líneas i-ésimas de `f1`, `f2` y `f3` separadas por *tabuladores*.

`paste -d= f1 f2 f3` idem, utilizando `=` como separador.

Si hacemos `paste` con ficheros de líneas de longitud variable el resultado puede no ser agradable a la vista.

Si tenemos :

```
$ cat f1                    $ cat f2
11111                       aaa
22222222222222222222        bbb
33333                       ccc
```

la salida de `paste -d: f1 f2` será:

```
11111:aaa
22222222222222222222:bbb
33333:ccc
```

y a veces nos vendrá bien. Otras veces preferiremos usar el comando `pr` para obtener:

```
11111                 aaa
22222222222222222222  bbb
33333                 ccc
```

Combina:

12.7. join

Vamos a suponer que tenemos dos ficheros `menu` y `dieta` . En el fichero `menu` tenemos pares 'tipo_de_alimento - plato' y en el fichero `dieta` tenemos pares 'tipo_de_alimento - cliente'.

`join menu dieta` escribirá todas las combinaciones 'tipo_de_alimento plato cliente'.

Para dar la funcionalidad indicada, el comando `join` (de UNIX) supone que los ficheros de entrada (o la entrada estándar) tienen sus campos 'clave' ordenados alfabéticamente.

En nuestro ejemplo, `join` supone que los ficheros `menu` y `dieta` están ordenados alfabéticamente por el campo 'tipo_de_alimento' .

> La filosofía de UNIX es **no repetir funcionalidad** en los distintos comandos, y en este caso no incluir la ordenación en el comando `join` . Si sabemos que los ficheros están ordenados usamos `join` directamente. Si no sabemos si los ficheros están ordenados, primero los ordenamos con `sort` y luego combinamos la versión ordenada de los ficheros de partida. Por ejemplo:
>
> ```
> sort f1 >f1.tmp
> sort f2 |join f1.tmp -
> rm f1.tmp
> ```

El comando `join` recorre ambos ficheros, y cuando encuentra la misma clave en ambos ficheros, escribe el producto cartesiano de los restos de línea del fichero primero con los restos de línea del fichero segundo.

> El producto cartesiano de dos conjuntos es el conjunto de todas las combinaciones de elementos del primer conjunto con elementos del segundo conjunto.
>
> El producto cartesiano de los conjuntos {*1, 2, 3, 4, 5, 6, 7, sota, caballo, rey*} y {*oros, copas, espadas, bastos*} resulta ser el conjunto 'baraja española de 40 cartas'.

```
$ cat menu
carne cordero
carne ternera
fruta naranja
pasta macarrones
pescado anchoas
pescado mero
verdura acelgas
```

```
$ cat dieta
carne antonio
carne juan
fruta antonio
pasta isabel
pescado alberto
```

Con estos datos, `join` encuentra `carne` en ambos ficheros y escribe las cuatro combinaciones de `cordero` y `ternera` con `antonio` y `juan`. Luego escribe una combinación para la `fruta`, otra para la `pasta`, etc. Como no hay `verdura` en el fichero `dieta`, no escribe nada para la `verdura` en la salida estándar (¡con lo sana que es la verdura!).

```
$ join menu dieta
carne cordero antonio
carne cordero juan
carne ternera antonio
carne ternera juan
fruta naranja antonio
pasta macarrones isabel
pescado anchoas alberto
pescado mero alberto
```

En otro ejemplo de `join` tenemos dos ficheros: `poblaciones` con pares 'población - código_de_provincia' y `codigos` con pares 'código - nombre_de_la provincia'.

```
$ cat poblaciones
Mataro 08
Alcorcon 28
Mostoles 28
```

```
$ cat codigos
08 Barcelona
28 Madrid
```

`join -1 2 -o 1.1 2.2 poblaciones codigos` obtiene los pares 'población - nombre_de_la_provincia'.

```
$ join -1 2 -o 1.1 2.2 poblaciones codigos
Mataro Barcelona
Alcorcon Madrid
Mostoles Madrid
```

`-1 2` indica que en el primer fichero se use el campo segundo.

`-o 1.1 2.2` pide que la salida contenga el campo primero (`.1`) del primer fichero (`1.`) y el campo segundo (`.2`) del segundo fichero (`2.`) . (La `o` viene de *output*).

La opción `-t:` indicaría que se usa el carácter `:` como separador de campos.

> Los campos están separados normalmente por tabuladores o caracteres blancos. En este caso, varios separadores consecutivos cuentan como uno solo, y los separadores al comienzo de la línea no cuentan. Si el separador de campos se ha puesto mediante la opción `-t` todos los separadores cuentan.

> Cuando hacemos `join` y uno de los ficheros no tiene líneas con la clave repetida, podemos considerar que estamos haciendo un cambio de código o traducción. En el ejemplo anterior, `08` es un código, `Barcelona` es otro código.
>
> Podemos utilizar `join` para obtener las combinaciones de pares de líneas. Basta añadir a un fichero una columna que haga el papel de clave repetida, y hacer `join` del fichero modificado consigo mismo.

Cambian:

... caracteres (1 por 1)

12.8. `tr`

`tr` espera dos parámetros. El primero es la lista de caracteres a cambiar. El segundo es la lista de caracteres resultantes del cambio. Cambia el primer carácter del primer parámetro por el primer carácter del segundo parámetro, etc.

`tr` sólo puede actuar como filtro, no admite un nombre de fichero como entrada.

`tr aeiou aaaaa` cambia las vocales minúsculas por aes.

`tr aeiou a` hace lo mismo.

Cuando el segundo parámetro es más corto que el primero, se prolonga con la repetición del último carácter.

Se puede indicar como parámetro un intervalo de caracteres.

`tr A-Z _` cambia las letras mayúsculas por subrayados (`_`) .

La opción `-c` indica que se tomen como caracteres a cambiar el conjunto complementario de los caracteres nombrados.

`tr -c A-Za-z =` cambia los caracteres que **no** son letras por el signo igual (`=`) .

La opción `-s` quita las repeticiones de caracteres resultantes de los cambios.

`tr -s aeiou a` cambia las vocales por aes, y quita las repeticiones de aes.

`tr -d 0-9` borra las cifras.

La opción `-d` indica que se borren los caracteres del primer parámetro. No hay segundo parámetro con esta opción.

Hasta aquí se ha explicado la versión BSD de `tr` .

> Hay dos versiones del comando `tr`. Una versión es la asociada al UNIX de BSD. La otra versión es la asociada al UNIX SISTEMA V.
>
> En algunos ordenadores sólo está disponible una de las dos versiones. En otros están disponibles las dos versiones. Para usar una de las dos

versiones bastará con poner `tr` . En el caso de que haya dos versiones, para usar la segunda habrá que poner el nombre completo del fichero, por ejemplo: `/usr/ucb/tr` . ¡Qué cosas!.

La versión 'Sistema V' de `tr` necesita corchetes `[ ]` alrededor de los intervalos de caracteres y el intérprete de comandos nos obliga a encerrar estos corchetes entre comillas. Las repeticiones, en Sistema V, se indican mediante un asterisco detrás del carácter a repetir y todo ello entre corchetes (`[ *]`), y se añaden comillas para el intérprete de comandos.

El segundo ejemplo de `tr` se convierte en: `tr aeiou '[a*]'` ; para cambiar de mayúsculas a minúsculas escribimos: `tr '[A-Z]' '[a-z]'` y para cambiar las letras por almohadillas: `tr '[A-Z][a-z]' '[#*]'` .

... series de caracteres

12.9. `sed`

`sed` es un editor no interactivo. Es útil si se va realizar varias veces modificaciones similares sobre uno o varios ficheros.

Motivación

Supongamos que un usuario todos los días realiza las mismas operaciones de edición, hasta el punto de no necesitar mirar lo que hace.

Por ejemplo, borra la línea 2. Puede llegar a hacerlo usando las tres líneas siguientes:

```
vi -s b2.vi f3
                jdd:wq  #  en b2.vi
```

Si más tarde quiere borrar las líneas 2 y 14, escribe:

```
vi -s b214.vi  f3
                2Gdd14Gdd:wq  #  en b214.vi
```

No funciona. Al borrar la línea segunda se renumeran las líneas y ha borrado las líneas 2 y 15 del fichero original. Lo puede arreglar borrando primero la línea 14 y luego la 2.

Este método no parece muy cómodo según aumenta el número de las modificaciones.

Le conviene un editor no interactivo como `sed`. Su problema lo puede arreglar con:

```
sed -e '2d' -e '14d' f3
```

comparación

`vi`	`sed`
Uso (previsto) interactivo	Uso no interactivo
El fichero original queda modificado	El fichero original no queda modificado. Los cambios aparecen en la salida estándar.
La numeración de las líneas cambia al borrar una.	La numeración de las líneas no cambia al borrar una. Las líneas se numeran inicialmente.

invocación

Las instrucciones de edición (o comandos) para `sed` forman un programa.

`sed` aplica todos los comandos de edición a cada línea de texto editado. Empieza aplicando todos los comandos a la primera línea, luego aplica todos a la segunda, etc.

`sed -f` *fichero* toma los comandos de edición del fichero nombrado a continuación de `-f` .

`sed -e` *comando*
`sed -e` *comando* `-e` *comando* toma el (o los) comando(s) de la línea misma de invocación.

Si no se nombran ficheros, `sed` filtra la entrada estándar. Si se nombran ficheros, `sed` envía a la salida estándar el resultado de aplicar los comandos sobre las líneas de esos ficheros.

`sed -f` *ficheroCom* *ficheroDatos1* *ficheroDatos2*
aplica los comandos de edición de *ficheroCom* sobre las líneas de *ficheroDatos1* y *ficheroDatos2*, y envía el resultado a la salida estándar dejando los ficheros de datos como estaban.

comandos de edición

Los comandos de edición tienen el formato:

[*dirección* [, *dirección*] [`!`]] *función* [*argumentos*]

> Escribir algo entre corchetes indica que es opcional. En el ejemplo anterior vemos que si ponemos dos direcciones deben ir separadas por una coma.

Si no se pone dirección la *función* se aplica a todas las líneas.

Las direcciones pueden ser números, `$` (que representa la última línea), o expresiones regulares entre eslases (`/`) .

Si se pone un número o un carácter `$` la *función* se aplica sobre la línea indicada.

Si se pone una expresión regular entre eslases, la *función* se aplica sobre las líneas en las que se encuentre (un ajuste a) la expresión regular.

Si se ponen dos números, la *función* se aplica a las líneas cuyo número esté en el rango indicado.

Si se ponen dos expresiones regulares, éstas definen unos *a modo de párrafos*.

El primer *a modo de párrafo* comenzará con la primera línea en que se encuentre (un ajuste a) la primera expresión regular. Acabará en la primera línea en que (después) se encuentre (un ajuste a) la segunda expresión regular. Acabado el primero, el segundo *a modo de párrafo* comenzará con la primera línea en que se encuentre (un ajuste a) la primera expresión regular.

Supongamos que queremos procesar un fichero cuyo contenido son los días de la semana, cada uno en una línea y empezando por `lunes`.

`/u/,/a/` tomará como primer *a modo de párrafo* las líneas `lunes` y `martes` y como segundo las líneas `jueves` , `viernes` y `sabado` .

`/n/,/v/` tomará como primer *a modo de párrafo* las líneas `lunes` a `viernes` y como segundo la línea `domingo` .

Un carácter de exclamación (`!`) después de la dirección o direcciones hace que la función se aplique al conjunto de líneas complementario al que se tomaría sin ese carácter. Podemos traducirlo como 'ahí no' .

`/u/,/a/ !...` con el fichero días de la semana haría que la función se aplicase a las líneas `miercoles` y `domingo` .

funciones

Presentaremos ejemplos con las funciones: escribir o imprimir (`p`), borrar (`d`), substituir (`s`) y leer (`r`) . (Vienen de *print*, *delete*, *substitute* y *read*.)

`sed -e '3,5p'` lista las líneas 3 a 5.

`sed` escribe las líneas en la salida estándar siempre que no se indique lo contrario. Una forma para que no aparezca una línea en la salida estándar es pidiendo que se borre.

> La opción `-n` suprime la escritura en la salida estándar cuando no se indique explícitamente (por ejemplo con `p`) .

Si procesásemos un fichero con los nombres de los doce meses del año mediante el comando anterior, la salida sería:
`enero`, `febrero`, `marzo`, `marzo`, `abril`, `abril`, `mayo`, `mayo`, `junio`, `julio`, `agosto`, ... , `diciembre` en líneas distintas.

`sed -e '4,6d' -e '10,$d'` borra las líneas 4 a 6 y 10 en adelante.

Si procesásemos un fichero con los nombres de los doce meses del año mediante el comando anterior, la salida sería:
`enero`, `febrero`, `marzo`, `julio`, `agosto` y `septiembre` en líneas distintas.

`sed -e '/^$/,/FIN/p'` lista los grupos de líneas comprendidos entre una línea vacía y una línea que contenga `FIN` .

> En las expresiones regulares el carácter `^` representa el comienzo de línea. Algo situado antes del primer carácter de una línea. El carácter `$` representa el fin de línea. Algo situado después del último carácter de una línea.
>
> `^$` representa en las expresiones regulares una línea vacía.

`sed 's/mate /matematicas /'` substituye (en la salida estándar) las apariciones de '`mate `' por '`matematicas `'. Substituye la **primera** aparición en cada la línea.

> El carácter eslás (`/`) que aparece delimitando los parámetros de la substitución puede cambiarse por cualquier otro.

`sed -e '/DESDE/,/HASTA/!s/no/creo que no/g'` substituye **todas** las apariciones de `no` por `creo que no` salvo en los *a modo de párrafos* encerrados por líneas que contienen `DESDE` y líneas que contienen `HASTA` .

La `g` después de los parámetros de la substitución hace que se substituyan todas las apariciones. El carácter `!` hace que las substituciones se efectúen **fuera** de los *a modo de párrafos*.

`sed '/aqui/r fich'` añade el fichero `fich` a continuación de **cada** línea en que haya un una secuencia de caracteres `aqui` .

```
$ cat atex.sed
s/'a/\\'a/g
s/'e/\\'e/g
$ cat atex
sed -f atex.sed
```

Tenemos el fichero `atex.sed` con comandos de edición de `sed` . Indicamos que se substituyan todas las apariciones de `'a` por `\'a` . La terminación del nombre del fichero no viene de ninguna obligación, sino que es una costumbre de un usuario para recordar la función (¿o tipo?) del fichero.

> El carácter eslás inverso (`\`) tiene significado especial cuando aparece en los parámetros de substitución. Quita el significado especial al siguiente carácter.
>
> Para nombrar un eslás inverso sin significado especial, tenemos que poner dos eslases inversos.
>
> Este ejemplo es un apaño antes de disponer de una herramienta de proceso de texto, LaTeX, en versión adaptada al castellano (dícese también español).

En el fichero `atex` tenemos el uso del comando `sed` para filtrar un texto. Más adelante veremos cómo `atex` se convierte en un comando, e incluso la forma de usarlo desde dentro del editor `vi`.

No hemos agotado las posibilidades de `sed`. El manual presenta unas 26 funciones.

Reorganizan:

12.10. `sort`

`sort` ordena líneas. Toma como base la ordenación de los caracteres ASCII.

Conviene aprenderse cuatro cosas del juego de caracteres ASCII.

- El carácter blanco corresponde al código 32.
- Las cifras `0` a `9` ocupan posiciones consecutivas con códigos 48 a 57 .
- Las letras mayúsculas `A` a `Z` ocupan 26 posiciones consecutivas con códigos 65 a 90.
- Las letras minúsculas `a` a `z` ocupan 26 posiciones consecutivas con códigos 97 a 122.

```
$ cat prSort
Delta
 10
?donde?
ala
2
```

```
$ sort prSort
  10
 2
 ?donde?
 Delta
 ala
```

Como el `10` tiene en la columna primera un carácter blanco, aparece antes que el `2`. El `2` va antes que las palabras porque las cifras van antes que las letras. `Delta` va antes que `ala` porque las mayúsculas son anteriores a las minúsculas.

`sort` *opciones* *fichero1 ... ficheroN* equivale a
`cat` *fichero1 ... ficheroN* `|` `sort` *opciones*

`sort -t: -k 3` Ordena. Compara los campos tercero y sucesivos. Toma el carácter `:` como separador de campos.

Sin la opción `-t` los campos son secuencias no vacías de caracteres distintos del blanco o del tabulador. Actúa como separador una secuencia de blancos y tabuladores.

Con la opción `-t` el carácter que se indique actúa como separador. Puede haber campos vacíos. Un separador al comienzo de una línea define un campo (vacío). Es decir, los campos se delimitan como en el comando `join` .

> Al indicar los campos que se tienen en cuenta para ordenar, también se puede indicar los caracteres que queremos saltarnos de un campo. `sort` k 3.4 - indica que queremos ordenar considerando el campo tercero, a partir del carácter quinto (nos saltamos cuatro caracteres).

`sort -t: -k 3,5 -k 7` Ordena. Compara los campos tercero, cuarto, quinto, séptimo y siguientes.

`sort -nr` Ordena en orden decreciente, interpretando los campos como números. `-r` indica que la ordenación se haga en orden inverso, de mayor a menor. `-n` indica que la ordenación sea numérica.

Las ordenaciónes numérica y alfabética pueden diferir debido a que los campos no estén encolumnados o debido a la presencia de ceros por la izquierda.

```
$ sort -n f1234              $ sort f1234
  1                             2
   2                           1
 03                            4
  4                           03
```

`sort +0 -3 +4r` Ordena y a igualdad de los campos 0..2, ordena en orden decreciente según el valor del campo 4 y siguientes.

Se puede restringir la aplicación de ciertas opciones, como `-n` y `-r` , a algunos campos.

`sort -u` Ordena. Si hay varias líneas consecutivas idénticas escribe sólo una de ellas. Mediante `-u` indicamos que no queremos las líneas repetidas.

Si consideramos ficheros en que las líneas son elementos de un conjunto, `sort -u` aplicado sobre un fichero actúa como un filtro que lleva al conjunto a una forma normalizada. Con la misma hipótesis, `sort -u f1 f2` actúa como el operador *unión de conjuntos* sobre `f1` y `f2`.

12.11. uniq

`uniq` Elimina las líneas repetidas **consecutivas**.

`uniq -c` Idem. Escribe cuántas veces se repite de forma consecutiva cada línea.

```
$ sort votos |uniq -c
      2 abstencion
      1 no
      1 nulo
     12 si
```

`uniq -d` Sólo escribe las líneas repetidas consecutivas. Sólo las escribe una vez.

`sort -u` equivale a `sort | uniq` , y la primera forma es más rápida.

Es muy frecuente usar `uniq` después de `sort`, pero en modo alguno es fijo ni obligatorio.

`sort |uniq -c |sort -nr |head -40` nos da 'los 40 principales', las 40 líneas más frecuentes.

Buscan:

12.12. grep, fgrep, egrep

`grep` y `egrep` casi son dos versiones del mismo comando. Ambos buscan líneas que contengan una *expresión regular*. `grep` se apoya en el reconocimiento de *expresiones regulares antiguas*. `egrep` se apoya en el reconocimiento de *expresiones regulares modernas*. Como ambas variantes de *expresiones regulares* tienen sus ventajas, ambos comandos siguen siendo útiles.

`fgrep` es una versión de `grep` especializada en búsquedas de tiras fijas, sin caracteres especiales (metacaracteres).

> El editor de líneas de UNIX `ed`, y su sucesor `ex` admiten el comando: `g/`*expresión regular*`/p` , que lista (imprime, *print*), todas (`g`) las líneas que contengan una expresión regular dada. En inglés *expresión regular* lo representan por $\mathcal{RE}$, y ya tenemos `grep`.
>
> La `e` de `egrep` viene de *extended*. La `f` de `fgrep` viene de *fixed*.

`grep AQUI` Busca la secuencia de caracteres `AQUI` en la entrada estándar. Escribe todas las líneas en las que aparece.

`grep -v AQUI` Idem. Escribe las líneas en las que no aparece.

`grep AQUI *` Busca en todos (`*`) los ficheros del directorio. Si hay más de un fichero, indica dónde ha encontrado las líneas.

```
$ grep go dias meses
dias:domingo
meses:agosto
```

`fgrep AQUI` busca igual que `grep AQUI` . Es más rápido.

`fgrep -f dias quijote` busca en el fichero `quijote` las secuencias de caracteres contenidas en `dias` .

`fgrep` puede buscar varias tiras de caracteres simultáneamente.

```
$ head -2 dias
lunes
martes
$ fgrep -f dias quijote
que carnero, salpicon las mas noches, duelos y quebrantos los sabados,
lentejas los viernes, algun palomino de anyadidura los domingos,
....
```

`grep 'aqui[0-9]'` Busca la tira de caracteres `aqui` seguida de una cifra.

`[0-9]` es (parte de) una *expresión regular*. Significa 'cualquier carácter incluido en el rango 0..9'. Las comillas no forman parte de la expresión regular, sino que están para evitar que el intérprete de comandos busque significado a `[0-9]` .

Las *expresiones regulares* se estudian aparte porque son algo común a una serie de comandos. Ya han aparecido en `sed`, y vamos a ver un poco más en `grep` y familia.

El último ejemplo se podía haber abordado con `fgrep -f` y un fichero de 10 líneas: `aqui0`, `aqui1`, ..., `aqui9`. Es un caso particular; en seguida veremos que `grep` y `fgrep` tienen usos específicos.

`grep 'aqui[0-9]*no'` Busca la tira de caracteres `aqui` seguida de cifras (ninguna, una o varias) y de `no` .

El asterisco (`*`) en una expresión regular afecta al carácter inmediatamente anterior. Significa 0, 1, o más repeticiones de lo anterior.

> El asterisco (`*`) aparece como carácter con significado especial para el intérprete de comandos y en expresiones regulares. Son dos contextos distintos y espero que no haya confusiones. Ni siquiera en el examen.

`grep '^c'` Busca las líneas que empiezan por `c` .
En las expresiones regulares el carácter `^` representa el comienzo de línea.

`grep 'on$'` Busca las líneas que acaban en `on` .
El carácter `$` representa el fin de línea.

El comando `egrep` básicamente difiere de `grep` en la versión de expresiones regulares que acepta. Los ejemplos puestos hasta ahora pertenecen a la parte común de ambas versiones; funcionan igual con `egrep` y con `grep`.

Comparan:

12.13. `comm`

`comm` compara conjuntos .
`comm` espera que la entrada pueda verse como líneas en las que la información está autocontenida. En estos casos cada línea es como una ficha, como una tupla en una tabla de una base de datos. Al ordenar las líneas el fichero no debe perder el sentido.
El texto de un libro es un buen ejemplo de fichero que **no** cumple el requisito anterior. Si se cambia el orden de las líneas (muy probablemente) se pierde el sentido del fichero. `comm` no sirve para comparar dos novelas o dos actas.
`comm` supone que los ficheros están ordenados . Aunque un conjunto es el mismo independientemente del orden en que se enumeren sus miembros, la representación ordenada de los miembros del conjunto hace más eficaces algunos algoritmos.
`comm` no responde del resultado de su salida si los ficheros no están ordenados.

`comm` se usará normalmente sin líneas repetidas. En un conjunto no procede enumerar dos veces un miembro.

```
$ cat f1                                $ cat f2
blanco                                  blanco
naranja                                 naranja
verde                                   rojo

$ comm f1 f2
                blanco
                naranja
        rojo
verde
```

`comm` presenta en la salida estándar (la unión de) todas las líneas presentes en sus entradas. Las líneas presentes en las entradas se mezclan, de forma que la salida está ordenada[1]. Con lo dicho hasta ahora la salida de `comm` sería igual que la de `sort -u`.

Las líneas de las entradas aparecen en la salida en tres columnas:

- En la columna 1^a presenta las líneas que sólo están en el primer fichero.

  Si los ficheros representan conjuntos: conjunto primero **menos** conjunto segundo.

- En la columna 2^a presenta las líneas que sólo están en el segundo fichero.

  Conjunto segundo **menos** conjunto primero.

- En la columna 3^a presenta las líneas que están en ambos ficheros.

  Intersección de conjuntos.

Las columnas 2^a y 3^a están desplazadas (con) 8 y 16 caracteres (blancos).

En este comando deben figurar dos ficheros como parámetros o un fichero y la entrada estándar. La entrada estándar se representa por `-`.

Se puede eliminar 1 ó 2 columnas de la salida de `comm`. Se indica con las opciones `-`n_1 o `-`$n_1 n_2$. Las columnas nombradas son las que se eliminan.

[1] La salida está lógicamente ordenada. Si quitásemos los desplazamientos que forman las tres columnas estaría físicamente ordenada

```
$ comm -1 f1 f2
        blanco
        naranja
rojo
$ comm -23 f1 f2
verde
$
```

12.14. cmp

`cmp` compara cualquier tipo de ficheros.

Si los ficheros son iguales no escribe nada. Si los ficheros son distintos señala la posición del primer octeto en que difieren. Su papel básico es el de función booleana:
- ¿Son iguales los ficheros?
- No. o - Sí.

`cmp f1 f2` compara los ficheros `f1` y `f2` octeto a octeto.

`cmp - f2` compara el fichero `f2` y la entrada estándar, octeto a octeto.

```
$ head /etc/termcap |cmp - /etc/termcap
cmp: EOF on -
```

En el ejemplo anterior, `cmp` nos informa de que la entrada estándar y el fichero `/etc/termcap` no diferían hasta el punto en que se acabó (la corriente de caracteres de) dicha entrada estándar, pero que `/etc/termcap` tiene más caracteres.

> `EOF` viene de *end of file*, fin de fichero, o para la entrada estándar, fin del flujo (corriente) de caracteres.

12.15. diff

`diff` compara ficheros cuyo contenido son textos.

`diff` permite comparar fuentes de programas semejantes, versiones de actas de reuniones o de textos literarios. Sólo es de utilidad cuando los ficheros a comparar tienen una mayoría de líneas idénticas.

> `diff` se apoya en un algoritmo cuyo objetivo es: dadas dos secuencias, hallar la subsecuencia común más larga. Este algoritmo se usa en biología y en genética. Compara secuencias de aminoácidos en proteínas y secuencias de ACGT en ADN, y así permite estudiar la distancia evolutiva entre especies.

`diff` tiene dos parámetros que serán nombres de ficheros o la entrada estándar, representada por `-` . Cuando las entradas sean idénticas, `diff` no escribirá nada.

```
$ diff f1 f2
3c3
< verde
---
> rojo
```

`diff f1 f2` compara los ficheros `f1` y `f2` línea a línea.

Las diferencias en el primer fichero (primera entrada) comienzan con `<` . Las del segundo fichero (segunda entrada) con `>` .

> `diff` agrupa las diferencias en bloques. Cuando el bloque supone un cambio pone una `c` , cuando es algo presente sólo en la segunda entrada pone una `a` , cuando es algo nuevo en la primera entrada pone una `d` .
>
> A la izquierda de las letras pone los números de las líneas que cambian en la primera entrada. A la derecha de las letras pone los números de las líneas que cambian en la segunda entrada.

`diff -e` *fichero-1 fichero-2* genera una secuencia de caracteres. Esa secuencia de caracteres introducida como entrada de `ed` transforma el primer fichero en el segundo.

```
$ diff -e f1 f2 | tee de.f1.a.f2
3c
rojo
.
$ (cat de.f1.a.f2; echo '1,$p') | ed - f1 >f2a
$ cmp f2 f2a
$
```

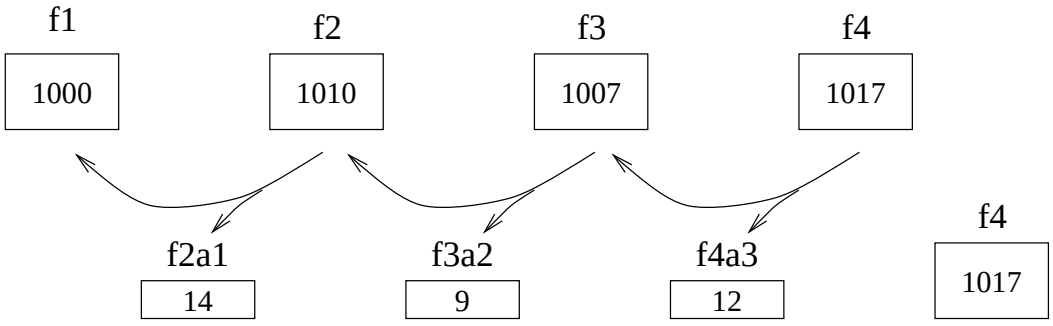
1. `diff -e f1 f2` genera la secuencia de caracteres para cambiar `f1` en `f2` .
2. `| tee de.f1.a.f2` nos permite ver la secuencia y la guarda en el fichero `de.f1.a.f2` .
3. `( ..... ; echo '1,$p') |` añade `1,$p` al contenido del fichero `de.f1.a.f2` y lo manda a la entrada estándar de `ed` .
4. `ed - f1 >f2a` edita el fichero `f1` tomando los comandos de la entrada estándar y no del teclado, y lleva la salida al fichero `f2a` .
5. `cmp f2 f2a` compara el fichero `f2` y la versión modificada del primero `f2a` .
6. Como la salida de `cmp` ha sido nula consideramos que `f2` y `f2a` son iguales.

una aplicación

Supongamos que tenemos 4 versiones consecutivas de un programa en sendos ficheros de nombre: `f1`, `f2`, `f3` y `f4`.

```
diff -e f4 f3 >f4a3
diff -e f3 f2 >f3a2
diff -e f2 f1 >f2a1
```

Obtenemos tres ficheros 'diferencia', y nos quedamos con `f4` y los tres ficheros diferencia: `f4a3`, `f3a2` y `f2a1`. Borramos los tres primeros ficheros originales (`rm f[123]`).



En la figura podemos ver los ficheros originales, los ficheros de diferencias, y sus tamaños (en líneas). Vemos que se ahorra bastante espacio.

Si en un momento queremos recuperar el fichero `f2` , por ejemplo, nos basta hacer:

```
(cat f4a3 f3a2; echo '1,$p') | ed - f4
```

Hemos guardado el último fichero y las diferencias hacia atrás porque consideramos que las versiones más solicitadas son las últimas, y así las obtenemos con menor cálculo.

Hemos hecho un cambio de espacio (en disco, supongo) por tiempo (de cálculo) de la CPU.

A veces no nos basta con ver la diferencia entre dos ficheros, sino que queremos verla en un contexto.

`diff -c4` escribe las diferencias con cuatro líneas de contexto. Si no decimos cuántas líneas presenta tres .

```
$ diff doc1 doc2
168c168
< personal o social.
---
> personal, social o economica.
$ diff -c2 doc1 doc2
*** doc1    Wed Apr 29 18:39:49 1998
--- doc2    Wed Apr 29 18:47:20 1998
***************
*** 166,170 ****
  discriminacion alguna  por  razon  de  nacimiento,  raza,  sexo,
  religion, opinion o cualquier  otra  condicion  o  circunstancia
! personal o social.

                              SECCION 1
--- 166,170 ----
  discriminacion alguna  por  razon  de  nacimiento,  raza,  sexo,
  religion, opinion o cualquier  otra  condicion  o  circunstancia
! personal, social o economica.

                              SECCION 1
```

12.16. patch

`patch` es capaz de regenerar un fichero f_1 a partir de otro fichero f_2 y la diferencia entre f_1 y f_2 obtenida mediante `diff -c` .

Como la salida de `diff -c` es parcialmente redundante con el fichero f_1 (y con el fichero f_2), `patch` es capaz de detectar incoherencias. Es más, previo consentimiento por parte del usuario, intenta superarlas.

Con los datos del ejemplo anterior, alguien tiene un fichero `doc11`, parecido a `doc1` pero con las líneas `discriminacion .... SECCION 1` en las posiciones 167–171, y recibe la salida de `diff -c2 doc1 doc2 >doc1a2`

`patch doc11 doc1a2` pregunta al usuario si quiere seguir. En caso afirmativo guarda en un fichero de nombre `doc11.orig` el contenido previo de `doc11` y modifica (parchea) `doc11` cambiando su línea 169 .

`patch` puede aplicarse al fichero modificado y recupera el fichero original.

`patch` es uno de los muchos programas disponibles sin costo, y con fuentes accesibles. :-) .

12.17. diff3

`diff3 f1 f2 f3` escribe las diferencias entre tres ficheros.

```
$ cat f1          $ cat f2          $cat f3
negro             negro             negro
marron            rojo              marron
$ diff3 f1 f2 f3
====2
1:2c
3:2c
  marron
2:2c
  rojo
```

`====2` indica que el fichero segundo es diferente.

`diff3` admite un signo `-` para indicar que la entrada estándar entra en la comparación.

Visten:

12.18. `nl`

`nl` numera las líneas que no están vacías.

`nl -ba` numera (todas) las líneas. `-b` viene de *boolean*, opción binaria, `a` viene de *all* .

12.19. `pr`

`pr` prepara para imprimir, ajusta el texto a formato de página.

Vamos a considerar una página con 80 columnas y 66 líneas. En el caso más común, se escribe la información en 56 líneas. Se dejan 4 líneas blancas por abajo de margen, y en una de las 6 líneas de margen superior se escribe la cabecera. La cabecera usual tiene la fecha de última modificación si se parte de un fichero o la de proceso si se lee de la entrada estándar, el nombre del fichero en su caso, y el número de la página.

> Los márgenes evitan que algunas líneas resulten ilegibles por caer en la zona de pliegue del papel continuo, o en el borde de una hoja mal impresa, o con problemas de fijación del tóner. La cabecera de los listados evita muchas confusiones entre hojas de listado muy parecidas. Paginar no supone un gasto de papel, sino un ahorro.

> 80x66 viene de un tamaño de papel: 11x8 en pulgadas, y de un espaciado de los caracteres: 10 caracteres por pulgada en horizontal, y 6 líneas por pulgada (en vertical).

Los valores por omisión son 66 líneas y 56 líneas de impresión.

`pr f1` prepara para imprimir el fichero `f1` .

`pr f1 f2 f4` prepara para imprimir los ficheros `f1` , `f2` y `f4` . Cada fichero empieza página nueva y numeración nueva.

`pr -m numeros numbers` prepara para imprimir varios ficheros (`numeros` y `numbers` en este caso) en columnas paralelas. El aspecto de las páginas es:

```
--------------------------------

Oct 27 13:48 1992         Page 1

             uno             one
             dos             two
            tres            thre
          cuatro            four
          .....
```

`pr -2 numeros` prepara para imprimir a dos columnas. El aspecto de las páginas es:

```
-----------------------------------

Oct 27 13:49 1992             Page 1

             uno     cincuentaysiete
             dos      cincuentayocho
            tres     cincuentaynueve
          cuatro             sesenta
           .....
```

> Cuando queramos preparar la impresión en varias columnas tendremos que contar con la longitud de la línea más larga. Con la anchura de impresión por omisión (72 caracteres, 80 menos 8 de margen) y a dos columnas sólo se imprimen los 35 primeros caracteres de cada línea. A tres columnas sólo caben 23 caracteres por columna.

Otras opciones de `pr` :

- `-w128` cambia la anchura de la página a 128 caracteres.
- `-l72` cambia la longitud de la página a 72 líneas.
- `+11` la salida empieza en la página 11.

`pr +11 f1` es lo mismo que `pr f1 | tail +661` . Saltarse 10 páginas de 66 líneas es lo mismo que saltarse 660 líneas y empezar en la línea 661.

- `-h 'otra Cabecera'` pone `otra cabecera` en lugar la cabecera habitual.
- `-t` suprime la cabecera, líneas previas y líneas finales.

`pr -t -l1 -4 numeros` escribe el contenido del fichero `f1` a cuatro columnas *fila a fila*.

```
  uno          dos          tres        cuatro
cinco          seis         siete         ocho
```

Cuenta:

12.20. `wc`

`wc` cuenta líneas, palabras y caracteres. Si hay varios ficheros indica los totales.

```
$ wc f1 f2
      2       2      13 f1
      2       2      11 f2
      4       4      24 total
```

`wc -l` sólo cuenta líneas. Igualmente con `-w` sólo cuenta palabras y con `-c` sólo cuenta caracteres .

aquí no acaba ...

Quedan más filtros. Entre los más importantes y extendidos señalo `awk`.

Supongamos que queremos extraer los campos tercero y primero de un fichero `fich` . Con los filtros vistos una forma es:

```
cut -f1 fich > ftmp
cut -f3 fich | paste - ftmp
rm ftmp
```

Con `awk` la solución es mucho más sencilla:

```
awk '{print $3, $1}' fich
```

`awk` tiene un lenguaje propio y merece un capítulo aparte.

También `sh` se puede usar como un filtro, pero creo que no es su uso normal y presentarlo como filtro es forzar el concepto.

> Un recuerdo para el comando `dd` que parte ficheros, y que tiene su utilidad con los ficheros binarios, es decir con los que no están organizados en líneas.

12.21. Ejercicios

Ejercicios sobre `head`, `tail`, `split` y `cut`:
93j.11 (pág. 285), 93s.2 (pág. 290), 94f.6 (pág. 296), 94s.1 (pág. 308), 95f.3 (pág. 314), 95j.3 (pág. 320), 95s.10 (pág. 327), 96j.2 (pág. 338), 97f.3 (pág. 350), 97j.8 (pág. 357), 97s.4 (pág. 362), 98f.2 (pág. 368), 98s.3 (pág. 380).

Ejercicios sobre `cat` y `paste`:
93j.3 (pág. 284), 94j.6 (pág. 302), 95j.8 (pág. 321), 98j.6 (pág. 374).

En muchos ejercicios aparece `cat`, aunque se supone que no está ahí la dificultad.

Ejercicios sobre `join`:
93f.5 (pág. 278), 93j.4 (pág. 284), 94f.2 (pág. 296), 94j.12 (pág. 304), 94s.7 (pág. 309), 95f.4 (pág. 314), 95j.4 (pág. 320), 96f.7 (pág. 333), 96s.5 (pág. 344), 97f.6 (pág. 351), 97j.1 (pág. 356), 98f.6 (pág. 368), 98j.1 (pág. 374), 98s.1 (pág. 380).

Ejercicios sobre `tr`:
93s.3 (pág. 290), 96j.1 (pág. 338), 98f.4 (pág. 368), 98j.5 (pág. 374), 99f.10 (pág. 387).

Ejercicios sobre `sed`:
93j.5 (pág. 284), 93s.7 (pág. 291), 94f.12 (pág. 298), 94f.18 (pág. 299), 94j.2 (pág. 302), 94j.18 (pág. 305), 95j.9 (pág. 321), 95j.18 (pág. 323), 95s.3 (pág.

326), 96j.19 (pág. 341), 96s.6 (pág. 344), 97f.17 (pág. 353), 98f.3 (pág. 368), 98s.11 (pág. 381).

Hay más en el capítulo de expresiones regulares.

Ejercicios sobre `sort`:
93j.8 (pág. 285), 95f.5 (pág. 314), 97f.2 (pág. 350), 97j.14 (pág. 358).

En muchos ejercicios aparece `sort`, aunque se supone que no está ahí la dificultad.

Ejercicios sobre `uniq`:
94s.9 (pág. 309).

Ejercicios sobre `grep`, `fgrep` y `egrep`:
96f.8 (pág. 333).

Hay más ejercicios en el capítulo de expresiones regulares.

Ejercicios sobre `comm`:
93f.8 (pág. 279), 94f.4 (pág. 296), 94j.4 (pág. 302), 95f.6 (pág. 314), 96j.8 (pág. 339), 97s.14 (pág. 364), 98j.2 (pág. 374), 99f.12 (pág. 388).

Ejercicios sobre `diff`:
94f.7 (pág. 297).

Ejercicios sobre `nl`:
96j.5 (pág. 338), 99f.14 (pág. 388).

Ejercicios sobre `pr`:
97j.3 (pág. 356), 98f.7 (pág. 369).

Ejercicios sobre `wc`:
96s.9 (pág. 345).

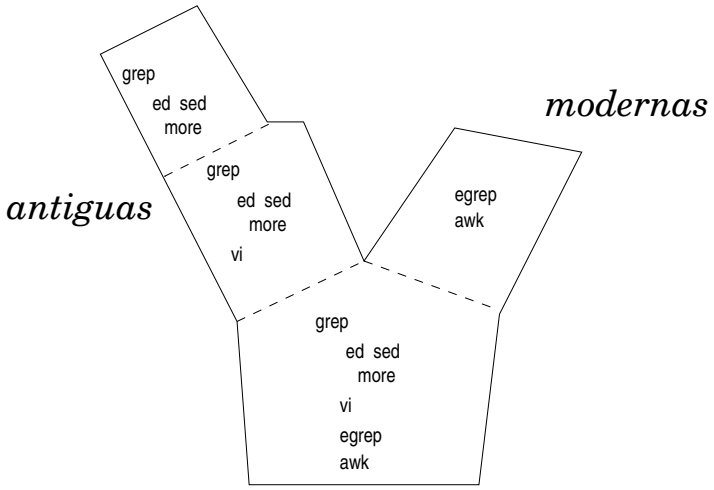
Ejercicios sobre varios filtros (sin clasificar):
93f.11 (pág. 279), 93f.16 (pág. 281), 93j.16 (pág. 287), 96f.18 (pág. 335), 98s.5 (pág. 380), 98s.8 (pág. 381), 98s.16 (pág. 382), 99f.4 (pág. 386).

Capítulo 13

Expresiones regulares

Las expresiones regulares constituyen un lenguaje para describir tiras de caracteres.

Empezaron a usarse con `ed`, un editor orientado a líneas. Más tarde se incluyeron en otros comandos. Siguiendo una buena metodología de programación, no estaban en el código de cada comando, sino en una biblioteca de funciones, y su comportamiento era el mismo para los distintos comandos.



Expresiones regulares

En cierto momento se replantearon la funcionalidad de las expresiones regulares y surgió una variante. Llamaré a estos dialectos *expresiones regulares antiguas* y *expresiones regulares modernas*. Ambos dialectos comparten una serie de reglas.

Generalmente las expresiones regulares describen tiras de caracteres incluidas en una línea, es decir tiras que no incluyen el *cambio de línea*.

13.1. Parte común

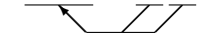
Válido para `grep`, `vi`, `sed`, `ed`, `more`, `egrep` y `awk`.

`c`	Un carácter no especial vale por él mismo. Un carácter puede ser especial en expresiones regulares modernas, y no especial en expresiones regulares antiguas, por ejemplo `|` .
`\$`	`\` **cancela** significado de caracteres especiales. Este carácter, eslás inverso (`\`), es también carácter especial.
`^`	**inicio de línea**, algo situado antes del primer carácter de una línea.
`$`	**fin de línea**, algo situado después del último carácter de una línea.
`.`	**cualquier carácter**, excepto el *cambio de línea* (ASCII 10 decimal).
`[abz]`	**alternativa** de caracteres. `a` o `b` o `z`. En general, cualquiera de los caracteres entre corchetes.
`[i-n01]`	alternativa. Permite especificar **rangos** de caracteres. En este caso indica cualquier minúscula desde la `i` a la `n` , o la cifra `0`, o la cifra `1`.
`[^a-zA-Z]`	cualquier carácter **salvo** los citados y el *cambio de línea*. El carácter `^` inmediatamente después de `[` indica complemento de los caracteres nombrados respecto del juego de caracteres sin el *cambio de línea*. El carácter `^` en otra posición entre corchetes se representa a sí mismo.
`a*`	cero o más **repeticiones** de `a`. El asterisco indica cero o más repeticiones de lo inmediatamente anterior.

13.2. En expresiones regulares antiguas

Válido para `grep`, `vi`, `sed`, `ed`, y `more`.

`\(a*b\)`	una (sub)***expresión marcada*** en la que cero o más caracteres `a` van seguidos de un carácter `b` . Las (sub)expresiones se marcan poniéndolas entre `\(` y `\)` .
`\2`	**referencia** a la segunda (sub)*expresión marcada*. Cuando hacemos una referencia a una *expresión marcada* estamos diciendo: 'lo mismo, **exactamente lo mismo** que antes'.
`\<`	**comienzo de palabra**, algo situado antes del primer carácter de una palabra. Se considera *palabra* una secuencia de una o más letras, cifras y caracteres de subrayado. (Más que palabras me parecen identificadores del lenguaje C, o `awk`). Cada palabra incluirá el máximo de caracteres posibles.
`\>`	**fin de palabra**, algo situado después del último carácter de una palabra.
`[a-z][a-z][a-z]`	representa tres letras minúsculas cualesquiera.
`\([a-z]\)\1\1`	representa una letra minúscula repetida tres veces.



En algunos comandos

Válido para `grep`, `sed`, y `ed`.

`[a-z]\{4,6\}`	entre 4 y 6 letras minúsculas consecutivas. Se pone entre `\{` y `\}` el mínimo y máximo de **veces** que se puede **repetir** lo anterior.
`[a-z]\{7\}`	7 letras minúsculas consecutivas.

13.3. En expresiones regulares modernas

Válido para `egrep`, y `awk`.

`([ab]c)`	el paréntesis **agrupa**.
`([0-9]_)*`	**cero o más repeticiones** de `[0-9]_` . En las expresiones regulares antiguas el asterisco sólo afectaba a un carácter o alternativa.
`(bc)+`	**una o más repeticiones** de `bc` .
`[de]?`	**cero o una vez** lo anterior, es decir `d` o `e` o nada.
`(a*)|(b*)`	**alternativa**: una secuencia de caracteres `a` , o una secuencia de caracteres `b` (incluyendo la secuencia vacía). En las expresiones regulares modernas el operador `*` tiene mayor prioridad que el operador `|` . La expresión anterior se puede escribir `a*|b*` .

13.4. Ejemplos

`^$` Un comienzo de línea seguido de un fin de línea o, lo que es lo mismo, una línea vacía.

`[a-zA-Z0-9]*`
una secuencia (quiza vacía) de letras y cifras. Es parecida a un identificador del lenguaje pascal.

`[a-zA-Z0-9]+`
una secuencia no vacía de letras y cifras. He supuesto que estamos en expresiones regulares modernas.

`[a-zA-Z0-9][a-zA-Z0-9]*`
idéntico a lo anterior. (Una o más ..., es lo mismo que una ... seguida de cero o más ...). Válida para todas las expresiones regulares.

`[a-zA-Z][a-zA-Z0-9]*`
Una letra seguida de cero o más letras o cifras. Ésta es la forma de los identificadores del lenguaje pascal.

`\([0-9]\)\1` Un número capicúa de dos cifras.

`p.p.` da positivo si encuentra `pepa` .

`\(p.\)\1` no se ajusta `pepa`, y sí `pepe` .

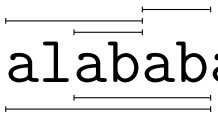
`.*`	cualquier tira de caracteres (incluyendo la tira vacía o de longitud nula).
`..*`	cualquier tira de caracteres (con al menos un carácter).
`[aeiou]`	cualquier vocal en minúsculas.
`[^aeiou]`	cualquier carácter *excepto* las vocales en minúsculas.
`[aeiou^]`	
`[a^eiou]`	cualquier vocal en minúsculas o *el acento circunflejo* '`^`'.
`[0-9]`	cualquier cifra.
`[0-9,]`	cualquier cifra o la coma.
`[^0-9]`	cualquier carácter excepto las cifras.
`[0-9][^0-9][0-9]`	dos cifras separadas por un carácter que no es cifra.
`[0-9].*[^0-9].*[0-9]`	dos cifras separadas por algún o algunos caracteres que *no son todos* cifras.
`[0-9][^0-9][^0-9]*[0-9]`	dos cifras separadas por algún o algunos caracteres que *no es ninguno* cifra.
`\(.\)\1`	dos caracteres *consecutivos* iguales.
`\(.\).*\1`	dos caracteres repetidos.
`\(..*\)\1`	dos secuencias repetidas y *consecutivas*.
`\(..*\).*\1`	dos secuencias repetidas.
`^\([0-9]\)\([0-9]\)[0-9]\2\1$`	una línea con un número de cinco cifras capicúa.

13.5. Substituciones con `vi` o `sed`

Las expresiones regulares exploran las líneas de izquierda a derecha. Entre las secuencias situadas más a la izquierda seleccionan la más larga.

Por ejemplo, en una línea tenemos `alababa` y queremos substituir los caracteres entre una `a` y una `b` inclusive por una `X` . Hay cinco posibles secuencias. Más tarde imponemos que no haya ninguna `b` en medio de la

secuencia a substituir. Hay tres posibles secuencias.




```
$ cat ersed
alababa
$ sed -e 's/a.*b/X/' ersed
Xa
$ sed -e 's/a[^b]*b/X/' ersed
Xaba
```

Se pueden poner referencias a subexpresiones marcadas en la segunda parte (segundo parámetro) de una substitución.

`sed -e 's/a\([0-9]*\)b/A\1B/g'`
substituye la secuencia: un carácter `a`, seguido de cifras (o ninguna), seguido de un carácter `b` por :
un carácter `A`, seguido de **esas mismas** cifras seguido de un carácter `B` . (Todas las veces que lo encuentre).

`sed -e 's/\([aeiou][aeiou]\)/\1\1/'`
duplica el primer par de vocales (consecutivas) que encuentre en cada línea.

13.6. Ejercicios

Ejercicios sobre expresiones regulares:
93f.4 (pág. 278), 93f.7 (pág. 279), 93j.7 (pág. 285), 93s.8 (pág. 291), 93s.9 (pág. 291), 94f.3 (pág. 296), 94j.3 (pág. 302), 94s.5 (pág. 308), 94s.6 (pág. 308), 95f.2 (pág. 314), 95j.1 (pág. 320), 95s.9 (pág. 327), 95s.12 (pág. 328), 96f.4 (pág. 332), 96f.12 (pág. 334), 96j.7 (pág. 338), 96j.11 (pág. 339), 96s.2 (pág. 344), 96s.7 (pág. 344), 97f.5 (pág. 350), 97f.8 (pág. 351), 97j.4 (pág. 356), 97j.7 (pág. 357), 97s.1 (pág. 362), 97s.2 (pág. 362), 97s.5 (pág. 362), 97s.6 (pág. 362), 98f.5 (pág. 368), 98j.9 (pág. 375), 98s.4 (pág. 380), 98s.9 (pág. 381), 98s.18 (pág. 383), 99f.6 (pág. 386), 99f.9 (pág. 387).

Capítulo 14

Programas del intérprete de comandos

Es fácil escribir un *programa para el intérprete de comandos* o *script*[1]. Basta con crear un fichero con una o varias líneas. En general, cada línea se interpreta como un comando. Para que las líneas de este fichero se ejecuten tenemos dos métodos.

- `sh fich` . Llamamos al intérprete con el nombre de fichero `fich` como parámetro.
- Ponemos permiso de ejecución al fichero: `chmod u+x fich` . Llamamos al comando: `fich` .

Dentro de un programa para el intérprete de comandos, `$1` hace referencia al primer parámetro, `$2` al segundo, etc.

Podemos incluir comentarios. Comienzan con el carácter almohadilla (`#`) y acaban con el cambio de línea.

14.1. Ejemplos

```
$ cat > prueba
echo esto es una prueba
date
```

[1] *script* quiere decir 'apunte' e indica que en general los programas del intérprete de comandos son pequeños. *Escripto* quiere decir 'escrito' y está en el D.R.A.

```
^D
```

Creamos un fichero de nombre `prueba` con dos líneas. En la primera línea llamamos al comando `echo` y en la segunda al comando `date`. Lo normal es que para crear el fichero usemos un editor como el `vi` . El uso de `cat` del ejemplo es una broma.

```
$ sh prueba
esto es una prueba
Tue Nov  3 21:00:01 WET 1992
$ chmod u+x prueba
$ prueba
....
$ prueba
....
```

Llamamos al intérprete de comandos (`sh`) con el nombre del fichero como parámetro. Vemos la salida correspondiente a la ejecución de los comandos `echo` y `date`.

Más tarde, con `chmod u+x prueba` , ponemos permiso de ejecución al fichero `prueba` . A partir de ese momento nos basta con escribir el nombre del fichero (`prueba`) para que se ejecute como un comando cualquiera de los que vienen con el sistema al adquirirlo.

> Los programas que escribimos se usan cómodamente si estamos en el directorio en que residen, o si están en alguno de los directorios nombrados en la variable `PATH`. En caso contrario hay que incluir los directorios que nos llevan al programa. En el capítulo dedicado a `sh` se explica con más detalle.

```
$ vi ultima
...
$ cat ultima
last -10 $1
$ chmod u+x ultima
$ ultima q0987
....
```

Con el editor `vi` creamos un fichero de nombre `ultima` para conocer las últimas sesiones de un usuario. Ponemos permiso de ejecución al fichero `ultima`. Cuando escribimos `ultima q0987`, `q0987` substituye a `$1` en el fichero.

```
$ cat quien
grep $1 /etc/passwd |cut -d: -f1,5
$ sh quien q0987
....
```

Queremos obtener el nombre completo a partir del identificador de un usuario. Escribimos un programa que busca su línea en el fichero de contraseñas `/etc/passwd` , y de esa línea obtiene el quinto campo con el comando `cut`. El ejemplo puede mejorarse poniendo `grep $1: /etc/passwd` .

El comando `finger` realiza esta función y algo más.

```
$ cat masFrequentes
sort | uniq -c | sort -nr
$ ls -l masFrequentes
-rw-r--r-- 1 a0007 23 Oct 20 17:47 masFrecuentes
$ chmod u+x masFrequentes
$ ls -l masFrequentes
-rwxr--r-- 1 a0007 23 Oct 20 17:47 masFrecuentes
```

En el capítulo de redirección vimos un ejemplo de comandos para hacer un análisis de la frecuencia de palabras en un texto. Escribimos un fichero `masFrecuentes` que realiza el análisis de frecuencias. El efecto de poner permiso de ejecución se puede observar mediante el comando `ls -l` .

```
$ cat separaPals
tr -cs A-Za-z '\012' <$1
$ chmod u+x separaPals
$ separaPals quij | masFrecuentes
....
```

En el programa `separaPals` están los detalles de cómo extraer palabras. Tenemos dos programas `separaPals` y `masFrecuentes` que podemos usar juntos o por separado en otras aplicaciones.

14.2. Eslás inverso y comillas : \ y ' '

`sh` trata de forma especial una serie de caracteres:
`* ? [ - ] < > | & $ # ' " ‘ \ ( )` , *blanco*, *tabulador*, *retornoDeCarro*.

Tenemos tres mecanismos para que estos caracteres no sean tratados de forma especial.

- Si encerramos estos caracteres entre comillas sencillas (`'`) estos caracteres pierden su tratamiento especial.

  Entre comillas sencillas no podemos incluir comillas sencillas.

- Si ponemos un *eslás* (barra inclinada) hacia atrás (como las agujas del reloj a las cinco menos cinco) (`\`) el siguiente carácter pierde su tratamiento especial.

  Si se antepone `\` a varios caracteres consecutivos, resulta un texto confuso. Es mejor ponerlo entre comillas sencillas.

- Entre comillas dobles (`"`) el intérprete sólo da tratamiento especial a tres caracteres: `$ \ ‘` .

```
$ echo *
a b c
$ echo '**'
**
$ echo \**
echo: No match
```

En el directorio actual sólo hay tres objetos y su nombre es: `a` , `b` y `c` .

Entre las comillas sencillas el asterisco pierde su significado especial. Al comando `echo` le llega como parámetro dos asteriscos y los escribe.

El carácter `\` quita el tratamiento especial al primer asterisco. El segundo asterisco conserva su tratamiento especial. El intérprete `sh` busca objetos cuyo nombre empiece por un asterisco. No los hay.

Las versiones antiguas de `sh` devolvían la tira de caracteres de longitud nula y `echo` no escribía nada. Las versiones modernas de `sh` cuando no encuentran nada pasan el *patrón* (`**`) al comando `echo` el cual escribe el mensaje: `echo: No match` .

```
$ cat c
echo uno: $1 dos: $2
$ c 1 2 3
uno: 1 dos: 2
$ c '1 2' 3
uno: 1 2 dos: 3
```

En el fichero `c` tenemos un programa que muestra cuáles son los parámetros primero y segundo.

La segunda vez que llamo a `c` el intérprete `sh` parte por blancos que no estén entre comillas, y el primer parámetro es '`1 2`' y el segundo parámetro es `3` .

```
$ echo $TERM "$TERM" '$TERM' "\$TERM" '\$TERM'
vt100 vt100 $TERM $TERM \$TERM
```

`$TERM` se substituye por su valor, tanto si no hay comillas como entre comillas dobles. Entre comillas sencillas el carácter dólar (`$`) no se trata especialmente, se representa a sí mismo. Entre comillas dobles el carácter `\` quita el significado especial al `$` que le sigue. Entre comillas sencillas ni el carácter `\` ni el `$` son especiales y en el último caso se escribe `\$TERM` .

Un carácter `\` seguido de un cambio de línea nos permite seguir un comando en otra línea.

Capítulo 15

Permisos

Unix incluye unos mecanismos que permiten restringir '*quién* puede hacer *qué* con qué *objeto*'.

Para concretar los permisos con todo detalle, podemos dar todos los tríos (ternas) 'usuario operación objeto' permitidos, y el resto está prohibido. También podríamos especificar todo lo prohibido y el resto estaría permitido.

> En general se habla de objetos, pero podemos empezar pensando en ficheros y directorios.
>
> Las operaciones que se pueden hacer sobre (con) un objeto dependen del objeto. Fijándonos en un fichero, podemos pensar en leerlo, escribir a continuación de lo último que había, pedirlo para usarlo sin que nadie lo cambie, borrar su contenido y dejar un fichero de tamaño 'cero octetos', etc.

Esas ternas 'usuario operación objeto' pueden ser muchas. En un ordenador en el que hubiese 1000 usuarios, 20000 ficheros y 20 operaciones, nos saldrían del orden de 200 millones de ternas. Más difícil todavía, si se crease un fichero nuevo, habría que incluir las operaciones autorizadas para cada uno de los 1000 usuarios. Igualmente si se diese de alta a un nuevo usuario. Y para completar el panorama, habría que regular los permisos para conocer y cambiar los permisos (!).

15.1. En UNIX

Unix simplifica agrupando los usuarios en clases, y agrupando también las operaciones.

- En UNIX todo fichero (y directorio) pertenece a un usuario, su dueño.
- En UNIX todo fichero (y directorio) está asociado a un grupo.
- El dueño y el grupo se pueden ver con `ls -l` o con `ls -lg` .
- Respecto a un fichero o directorio, los usuarios se clasifican en:
  - *dueño*.
  - *grupo*: perteneciente al grupo del objeto y no dueño del mismo.
  - *otros*: ni dueño, ni perteneciente al grupo del objeto.

15.2. Ficheros

- Para los ficheros, la mayoría de las operaciones se asocian a tres tipos de permiso:
  - lectura `r` , de *read*,
  - escritura `w` , de *write*, y
  - ejecución `x` , de *execute* .

Cada fichero tiene sus permisos independientes. Dado un fichero y una clase de usuario se puede conceder o denegar permiso para cada uno de los tres tipos de operaciones (r, w, x). Para cada fichero tenemos nueve permisos (básicos) [1], y $2^9 = 512$ combinaciones posibles.

Los permisos se pueden ver con `ls -l` y aparecen en las columnas 2^a a 10^a.

```
% ls -l l.tex
- rw- rw- r--   1 a0007  3007     185 Mar  9 00:55 l.tex
  dueño grupo otros
  permisos
```

[1] Hay más permisos. Se ven más adelante

El fichero anterior tiene permiso de lectura y escritura para el dueño y para el grupo, y permiso de lectura para los demás. No da permiso de ejecución a nadie.

Los permisos se pueden expresar (codificar) mediante un número octal de tres cifras. Para los permisos del fichero anterior tendríamos 110 110 100 en base dos, ó 664 en octal (base ocho).

Normalmente:

- los permisos para el grupo son iguales o menores que los del dueño e iguales o mayores que los del resto de los usuarios.
- el dueño tiene permiso de escritura y el resto de los usuarios no tiene permiso de escritura.
- los ficheros que no lo necesitan no tienen permiso de ejecución.

Es frecuente que todos los usuarios tengan permiso de lectura.

Los permisos más frecuentes son (en formato de `ls -l`):

`-rw-rw-r--` ficheros de texto, programas fuente, datos.

`-rwxrwxr-x` ejecutables de lenguajes compilados, por ejemplo con `cc`, y programas del intérprete de comandos. No es el caso general de ficheros de otros lenguajes interpretados.

`-r--r--r--` datos que queremos no perder ni borrar. Si hacemos `rm` de un fichero con estos permisos nos pide confirmación para borrar el fichero.

> Conviene no confirmar por rutina.
> Puede ser nuestra ruina.

`-rw-rw----` información de difusión restringida (en este caso al grupo).

Es poco frecuente encontrar los siguientes permisos:

`-rwxrwx--x` fichero ejecutable pero no legible. Este permiso funciona para ejecutables de lenguajes compilados, y no funciona para programas del intérprete de comandos. El resto de los usuarios no puede copiar el programa, ni desensamblarlo, ni buscar tiras de caracteres en su interior.

`-rw--w--w-` Parece extraño poder escribir en un fichero y no poder leer. Esto sucede con los dispositivos asociados a terminales, como por ejemplo `dev/tty0..` . Con estos permisos otro usuario puede escribir en un terminal del que no aparece como dueño, pero no puede leer del teclado correspondiente.

> Con estos permisos nos pueden mandar mensajes a la pantalla. Si queremos cerrar esa puerta hacemos `mesg n` .

A continuación apunto unos permisos a los que no veo utilidad:

`----rw-rw-` los demás sí pueden escribir y leer pero el dueño no (¡!).

`----------` nadie puede hacer nada.

La filosofía del diseño de UNIX es no prohibir estas combinaciones que hoy nos parecen sin sentido. No hacen mal a nadie. El código del sistema operativo es más breve, tendrá menos errores y será más fácil de mantener. No es imposible que alguien encuentre utilidad a alguna de esas combianciones.

Los permisos presentados anteriormente se pueden codificar en octal como:

```
664  775  444  660
771  622
066  000
```

15.3. Directorios

- Para los directorios se consideran 3 tipos de operaciones: lectura `r`, escritura `w` y uso (del i-nodo) `x`.
- Con el permiso de lectura `r`, podemos conocer los nombres de los objetos en un directorio.
- Con el permiso `x` , dado un nombre en un directorio, podemos acceder al *nodo* y al objeto (fichero o subdirectorio).

Lo más frecuente es que los permisos de lectura `r` y de uso `x` vayan juntos.

Los permisos más frecuentes son :

`drwxrwxr-x` los demás usuarios no pueden escribir.

`drwxrwxrwx` directorio público (`/tmp`), todos pueden escribir.

`drwxrwx---` los demás usuarios no pueden acceder.

15.3.1. Diferencia entre `r` y `x`

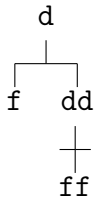
El permiso de lectura permite acceder a los nombres de los objetos contenidos en el directorio. El permiso de uso permite acceder al nodo correspondiente a un cierto objeto. Sin acceder a un nodo no se puede hacer `ls -l` de ese objeto, ni `cd` si es un directorio, ni leerlo o escribirlo. Con permisos `--x` podemos acceder a aquellos objetos de los que conocemos el nombre. Si no sabemos el nombre no podemos hacer nada.

> Es un poco la situación que se produce cuando le preguntan a uno:
> – ¿Dónde va Vd.?.
> (Por la razón que sea se lo preguntan más a unos que a otros.)

Supongamos unos directorios `d` y `dd`, y unos ficheros `f` y `ff`, y que los permisos que afectan a un usuario son:

`dd : r-x` `f : r--` `ff : r--`

Según los permisos del directorio `d` para el usuario, podrán realizarse o no las siguientes operaciones:



	permiso de `d`	
operación	`r--`	`--x`
`echo *`	sí	no
`ls`	sí	no
`ls -l`	no	no
`ls -l f`	no	sí
`cd dd`	no	sí
`cat f`	no	sí
`cat dd/ff`	no	sí

15.4. `chmod`

Los permisos se cambian con el comando `chmod` .

`chmod u+x p1` añade (al dueño) permiso de ejecución de `p1` .

Se puede cambiar un permiso: $\left\{ \begin{array}{c} u \\ g \\ o \end{array} \right\} \left\{ \begin{array}{c} + \\ - \end{array} \right\} \left\{ \begin{array}{c} r \\ w \\ x \end{array} \right\}$

`chmod a-w f1` quita (a todos) el permiso de escritura en `f1` .

Se puede cambiar tres permisos: $a \left\{ \begin{array}{c} + \\ - \end{array} \right\} \left\{ \begin{array}{c} r \\ w \\ x \end{array} \right\}$

`chmod 644 f1` pone los permisos de `f1` a `rw-r--r--` .

Esta forma permite fijar los nueve permisos.

Hay más formas de usar `chmod`, pero éstas son las más usuales.

15.5. Permisos iniciales

Cuando se crea un fichero o un directorio se pone como dueño al usuario, y como grupo el grupo del usuario.

> Un usuario puede pertenecer a varios grupos. Se pone como grupo del objeto el grupo activo para el usuario en ese momento.

Los permisos iniciales de los directorios y ficheros dependen de dos factores.

- En el código fuente de cada programa que crea un fichero *se propone* unos permisos. En una línea podríamos ver:
  `create ( "f3" , ... , 666 )`
  El tercer parámetro, `666`, son los permisos propuestos.

- Cuando un usuario usa (ejecuta) ese programa, en su proceso hay una variable local, la *máscara de creación* que indica qué permisos se quieren quitar de los ficheros creados. Supongamos que el valor de la máscara es `022` .

Se creará un fichero con permisos `644` .

Si el valor de la máscara hubiese sido `033` los permisos del fichero creado serían también `644` (¡!).

Los permisos de los ficheros y directorios creados se obtienen a partir de los permisos propuestos y de la *máscara de creación* (máscara-u) haciendo la **resta bit a bit sin acarreo** o, lo que es lo mismo, haciendo la operación **Y-lógico** del primer término con la máscara **negada**.

```
6 6 6        1 1 0   1 1 0   1 1 0
0 3 3      - 0 0 0   0 1 1   0 1 1
           ---------------------------
           1 1 0   1 0 0   1 0 0       6 4 4
```

La mayor parte de los programas proponen como permisos `666` . Este es el caso de los intérpretes de comandos (`sh`, `tcsh` o `bash`) cuando crean ficheros por redirección (`>` , `>>`), el editor `vi`, los comandos `tail`, `split`, etc.

En la fase final del proceso de compilación de bastantes lenguajes (Ada, pascal, C) se propone como permisos `777` .

El comando `mkdir` propone como permisos para los directorios `777` . Los permisos de los directorios se deciden con los mismos criterios que para los ficheros.

15.6. umask

`umask` nos informa del valor de la *máscara de creación* (máscara-u).

`umask 037` cambia el valor de la *máscara de creación* a 037.

Los valores habituales de la *máscara de creación* son 002 y 022 .

EL comando `umask` es *interpretado*. El intérprete de comandos no delega en otro proceso porque lo que queremos es cambiar el valor de una variable local. Es el mismo caso que el comando `cd`.

15.7. chown y chgrp

`chown o0137 f1` regala (cambia el dueño de) el fichero `f1` al usuario `o0137` .

Esto lo puede hacer el dueño en unos sistemas y en otros no. No puede deshacerlo (Santa Rita, ...). Lo puede hacer (y deshacer) el administrador (`root`). (¡Cómo no!).

> Inicialmente UNIX lo permitía. No aparecen inconvenientes a primera vista. El riesgo de que a alguien le regalen un programa ejecutable de comportamiento oculto y malicioso (*caballo de troya*) y que lo use parece pequeño.

> En un momento se introduce en algunos sistemas UNIX la posibilidad de limitar la ocupación de disco por parte de los usuarios. (Comportamiento de los usuarios con los discos: se expanden como los gases en sus recipientes, se comprimen como los sólidos o líquidos viscosos (según)).
>
> Si se permite *regalar* ficheros, un usuario al límite de su cuota de disco puede poner a nombre de otro usuario un fichero grande en un directorio inaccesible al supuesto beneficiario. (¡Trampa!).

`chgrp 5007 f1` cambia el grupo asociado al fichero `f1` a `5007`.

Esto lo puede hacer el dueño en unos sistemas y en otros no. Si puede hacerlo, podrá deshacerlo (seguirá siendo el dueño).

Un ejemplo curioso

```
$ cp f1 f2
$ chown a0008 f2
```

Copiamos el fichero `f1` y la copia se llama `f2`. Regalamos la copia a otro usuario (`a0008`).

```
$ vi f2
...
$ cal >>f2
...
```

Intentamos editar el fichero `f2` y no nos deja modificarlo. Intentamos añadir información (`$ cal >>f2`) al fichero `f2` y tampoco podemos.

```
$ rm f2
...
```

Intentamos borrar el fichero `f2`, nos pide confirmación, decimos que sí, ... y lo hemos borrado. No podemos escribir en un fichero ni a continuación, pero podemos borrarlo. Más de un usuario ha pensado que el sistema funcionaba mal.

`rm` borra un **nombre**. El nombre está en un directorio nuestro en el que tenemos permiso de escritura. Podemos borrar el nombre `f2`. Como es el

único nombre de ese fichero, al borrar el nombre se borra el fichero. Paradoja resuelta.

Si llegamos a avisar al usuario `a0008` de que le habíamos regalado un fichero y él hubiese puesto un nombre (segundo) al fichero `f2`, mediante el comando `ln`, no habríamos borrado el fichero.

> Esta historia me recuerda a uno de los pleitos que resolvió Sancho en la ínsula Barataria en el Quijote (segunda parte, cap. XLV). Un hombre reclama el pago de una deuda que otro asegura haber pagado ... .

15.8. Consideraciones

Un secreto parece mejor guardado cuantas menos personas lo sepan. Aunque sólo lo sepa una persona, ya se sabe que esa persona puede tener cosquillas, puede hablar dormida, etc. Incluso si no lo sabe ninguna persona puede desvelarse el secreto. Este sería el caso de la demostración de la conjetura de Fermat.

La moraleja que yo saco es que no hay secretos perfectos. Al menos en este mundo/pañuelo.

Para equilibrar opinaré que tampoco creo que exista el rompeSistemas (*cracker*) perfecto. (Tanto va el cántaro ...).

Y como consejo, propongo no dejar información crítica en sistemas conectados o compartidos.

Un sistema en general es más seguro si está aislado (no está conectado a internet, o ...net), si está a menudo apagado, si se desconoce su existencia, si lo guarda alguien, está inaccesible. En el límite es más seguro si no funciona, pero no queríamos llegar tan lejos.

Capítulo 16

Procesos

16.1. Ejecución no interactiva

Queremos buscar y contar líneas que contengan `Sancho` entre una serie de ficheros del directorio `libros`.

```
$ grep Sancho libros/* | wc > sancho.con &
7342
7343
$ vi tfc01.tex
....
```

Como la búsqueda puede tardar un rato le indicamos al ordenador que no espere a acabar para atendernos poniendo un carácter *et* (`&`, *ampersand*) detrás de los comandos. El sistema nos responde con un par de números. Los números son los *identificadores de proceso* (*pid*) de los procesos que ejecutan `grep` y `wc`. En seguida aparece el carácter `$` . El sistema está preparado para ejecutar otro comando. Pasamos a editar un fichero mientras `grep` trabaja.

`&` hace que un comando sencillo o una serie de comandos unidos por *tubos* (*pipes*) se ejecute de forma *no interactiva*.

> `&` es un carácter *separador* para el intérprete de comandos, al igual que el punto y coma (`;`) y el *retorno de carro*. Teniéndolo en cuenta, se puede escribir: `com1 & com2` . El comando `com1` se ejecutará en modo no interactivo y `com2` en modo interactivo.

A la ejecución interactiva también se la llama síncrona, atendida, en primer plano, en *foreground*. A la ejecución no interactiva también se la llama asíncrona, desatendida, en último plano, en *background*.

> Cuando se ejecuta un comando en modo no atendido lo normal es redirigir la salida estándar para que no interfiera con la ejecución de otros comandos interactivos.
>
> Los comandos ejecutados en modo no atendido no dependerán del teclado. O no intentan leerlo o tienen la entrada estándar redirigida. Prueba con `wc &` . Alguna razón habrá.

16.1.1. `ps`

`ps` nos muestra el estado de los procesos asociados al terminal en el que estamos trabajando.

```
$ ps
   PID TTY      TIME COMMAND
  8211 tty006   0:00 ps
  7310 tty006   0:01 bash
  7342 tty006   0:03 grep
  7343 tty006   0:00 wc
```

En la primera columna aparece el *identificador de proceso* o *pid*. Es un número entero positivo y en la mayor parte de los sistemas es menor que 30000. En los sistemas tipo UNIX tenemos la garantía de que no hay simultáneamente dos procesos con el mismo *pid*.

> Los números (libres) se suelen dar secuencialmente comenzando por 0. Los procesos del sistema toman los primeros números.

En la segunda columna aparece el terminal al que está asociado el proceso. En este caso el nombre completo es `/dev/tty006` .

En la tercera columna tenemos el tiempo de CPU consumido por el proceso.

En la cuarta columna aparece el nombre del comando. Si corresponde a un *script* aparecerá el nombre del intérprete de comandos.

En el ejemplo vemos el intérprete de comandos de entrada (`bash`), el mismo proceso que escribe el estado (`ps`), y los dos procesos que hemos iniciado en modo no interactivo (`grep` y `wc`).

`ps` presenta dos dialectos: *Sistema V* y *BSD*. El estilo presentado es el *BSD*. El estilo *Sistema V* presenta una columna más con el estado del proceso.

`ps -ax` presenta información amplia sobre todos los procesos del sistema en el dialecto (estilo) *BSD*.

`ps -ef` hace lo mismo en el dialecto *Sistema V*.

`pstree` informa sobre el estado de los procesos en forma gráfica. Está disponible al menos en los sistemas linux.

```
$ pstree a0007
init---bash---pstree
```

16.1.2. `kill`

`^C` (CONTROL-C) termina con el proceso (o grupo de procesos) que se está ejecutando de forma interactiva. Se descarta la entrada pendiente y la salida pendiente (casi toda).

Si queremos cortar la ejecución de los procesos que se están ejecutando en modo no interactivo no nos vale `^C` .

`kill` envía una *señal* al proceso o procesos cuyo número indicamos. Generalmente esa señal suele bastar para que el proceso termine.

> En UNIX las *señales* para los procesos son muy parecidas a las interrupciones para la CPU.

Supongamos que queremos hacer que termine la ejecución de los comandos que mandamos ejecutar con `grep ... | wc ... &` . Quizá nos acordemos de los números que escribió el sistema en respuesta. Quizá apuntamos esos números. En caso contrario, hacemos `ps` y los vemos.

`kill 7342 7343` nos permite terminar la ejecución de esos procesos.

Algunos programas (y comandos) inhiben la terminación cuando se recibe la *señal* de `kill` . En estos casos `kill -9` *pid* envía al proceso una señal que no puede ignorar y que obliga la finalización del proceso.

> No es buena costumbre enviar la señal de terminación a los procesos con `kill -9` . La mayor parte de los programas (bien escritos) cuando reciben la señal normal de `kill` (su nombre es *TERM*) procuran dejar los ficheros y directorios con que trabajaban lo mejor posible.

> Por ejemplo, si se está copiando un fichero, parece conveniente copiarlo entero o borrar la copia. Si dejamos el fichero a medio copiar podríamos borrar el original y quedarnos sin parte de la información.

`kill -9` hace que los programas acaben **ya**, sin tener tiempo para ordenar su entorno. Sólo se debe usar como último recurso.

Un proceso sólo atiende señales enviadas por su usuario (mismo *uid*), o por el administrador ($uid = 0$).

El *uid* (*user identifier*) de un usuario es el campo segundo de su línea en `/etc/passwd` . El *uid* de un proceso es el del usuario que lo hace ejecutar. Es una variable local a cada proceso.

16.1.3. `sleep`

Puede resultar interesante probar un poco qué sucede con los procesos. Nuestra herramienta de observación va a ser el comando `ps`.

Para observar procesos ejecutándose de forma no interactiva mediante el comando `ps` necesitamos que no acaben en seguida. Una forma de que no acaben enseguida es obligarles a trabajar mucho: o muchos cálculos o mucha entrada/salida (lectura de disco, por ejemplo).

Si estamos usando un ordenador personal el método anterior no tiene apenas inconvenientes. Si estamos compartiendo el ordenador con otros usuarios, el método anterior molestará a los otros usuarios. La molestia puede ser mayor si coinciden la mitad de los usuarios con estas pruebas.

Tenemos a nuestra disposición un comando que apenas consume recursos y puede servirnos.

`sleep` *Nsegundos* espera los segundos indicados y acaba.

En su código se indica arrancar un temporizador del sistema operativo. Luego debe pasar a estado de espera (*pausa*), y cuando llega la señal de que ha terminado el plazo terminar normalmente.

Aunque molesta poco, `sleep` consume un recurso cuyo número no es ilimitado. En cada sistema hay un número máximo de procesos. Como al ejecutarse es un proceso, ocupa una de las casillas de proceso. Conviene por ello no hacer pruebas con valores mayores de 2 ó 3 minutos. Si no somos lentos tecleando pueden bastar 30 segundos.

```
$ date; sleep 10; date
```

```
Thu Dec 24 09:24:56 GMT 1992
Thu Dec 24 09:25:06 GMT 1992
```

La línea que hemos tecleado parece pensada para ejecutar dos veces el comando `date` con un intervalo de 10 segundos. Aparentemente lo consigue. Los sistemas con UNIX ofrecen de forma estándar una resolución de 1 segundo. Esquemas del tipo del anterior presentan una deriva que puede ser importante.

> Como causas de la deriva tenemos la limitada precisión del reloj en muchos sistemas UNIX, el intervalo entre ejecución de comandos (nadie nos asegura que la máquina no se toma un descanso entre `sleep` y `date`), y el tiempo de ejecución de `date`. (Se toma un descando ejecutando otros procesos).
>
> En *tiempo real* explican que para evitar la deriva se deben usar comandos del tipo *a las 8 y cuarto me despiertas*, en lugar de *despiértame dentro de dos horas*.
>
> Hay versiones de UNIX útiles en tiempo real, pero resultan caras para quien no necesita garantizar que se cumplen unos plazos en el rango de los milisegundos.

```
$ date
Thu Dec 24 09:25:30 GMT 1992
$ (sleep 86400; echo ya es navidad ^G) &
$ date
Thu Dec 24 09:26:05 GMT 1992
$
```

`(sleep 86400; echo ya es navidad ^G) &`
hace que se ejecute en modo no interactivo una secuencia que primero espera un día (86400 segundos), y luego (el día 25) escribe `ya es navidad` y hace "pii". Sin esperar por esa secuencia (el mismo día 24 a las 9 y 26 minutos) el intérprete de comandos escribe un carácter `$` indicando que ya está dispuesto para aceptar otros comandos.

> Los teletipos tenían una campanilla, formada por una pequeña bola que golpeaba media esfera metálica. Para activar esa campanilla se reservó el código ASCII número 7 (CONTROL-G). Actualmente los terminales y ordenadores producen un pitido corto al recibir ese código.

16.1.4. Un terminal bloqueado

A un usuario cuyo identificador es `a0007` no le responde el terminal. Ha probado a teclear `^C` , `^Q` , etc. Va a otro terminal. Comienza una sesión (hace `login`).

```
$ tty
/dev/tty09
$ ps -ef | grep a0007
a0007  2517 2036  2 13:57:55 tty09 0:00 grep a0007
a0007  1360    1  0 12:56:46 tty06 0:01 -csh
a0007  2516 2036 21 13:57:55 tty09 0:00 ps -ef
a0007  2036    1  0 13:53:05 tty09 0:01 -csh
a0007  1508 1360  0 13:47:51 tty06 0:20 pru3
$ kill 1508 1360
```

Teclea `tty` . Este comando indica el nombre del terminal asociado. El usuario sabe que ahora está en `tty09` .

Luego teclea `ps -ef ...` . Ha pedido el estado completo de todos los procesos y ha seleccionado (con `grep`) aquellos de los que es dueño (o responsable). Aparecen dos números de dispositivo: `tty06` y `tty09`. Deduce que antes estaba en el terminal `tty06` . El *pid* de los procesos asociados a ese terminal es 1360 y 1508.

> Podía haber hecho `ps -ef | grep a0007 | grep -v tty09` y sólo habrían aparecido las líneas relacionadas con `tty06` .

Envía una señal de terminar al proceso de `pid=1508` (`pru3`) y al de `pid=1360` (`csh`).

Podemos sacar más partido del ejemplo anterior. La tercera columna de la salida de `ps -ef` es el *pid* del proceso padre. Por ejemplo, la primera línea nos dice que el proceso cuyo *pid* es 2036 es padre del proceso cuyo *pid* es 2517. Con esta información podemos representar el árbol de procesos de la figura 16.1.

16.2. Llamadas al sistema

> El objetivo al explicar detalles sobre el esquema de procesos en los sistemas UNIX es:

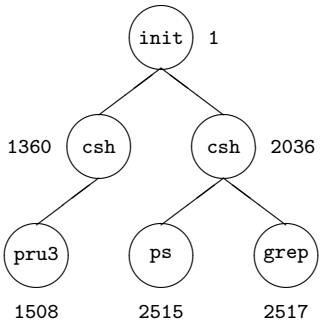


Figura 16.1: Árbol de procesos

1. Que los alumnos conozcan el efecto de: ejecutar comandos interactivos, no interactivos, programas del intérprete de comandos, `cd` , `umask` , comandos conectados por tubos (pipes), comandos entre paréntesis, y redirigir la salida de los comandos.

2. Que los alumnos conozcan la causa de estos efectos.

   Mediante una receta, se puede transmitir que "el efecto de un `cd` (cambio de directorio) no se observa fuera de un programa del intérprete de comandos (*script*), ni de una construcción `( .... )` " . No satisfechos con recetas queremos que los alumnos sepan **por qué**.

3. Introducir información (y formación) útil en otras asignaturas, en particular Sistemas Operativos I y II.

Muchos programas de usuario se ejecutan conviviendo con un sistema operativo.

Cuando se ejecuta un programa de usuario, por ejemplo una suma (ADD cuenta movimiento), estamos utilizando instrucciones de la CPU. El sistema operativo no nos ha ocultado la CPU. El Sistema Operativo nos permite utilizar (casi) todas las instrucciones de la CPU. Además, el Sistema Operativo nos ofrece operaciones (o instrucciones) nuevas, y tipos de datos (objetos) nuevos. A estas operaciones nuevas las llamamos *llamadas al sistema*.

En este momento sólo nos interesan cuatro *llamadas al sistema* que nos permiten crear y terminar procesos, cambiar el programa que ejecutan los procesos, y una forma primitiva de sincronizarlos.

16.2.1. Proceso: - Máquina virtual

Supongamos que tenemos una oficina con tres ventanillas: Envío de paquetes, Recepción de paquetes, Venta de sellos. Un funcionario dedica 20 segundos a cada ventanilla, cambiando de aspecto cada vez que asoma por una ventanilla. Es posible que el público piense que hay tres funcionarios.

En cierta terminología (jerga) dicen que tenemos tres funcionarios virtuales y un funcionario real. Cambiamos 'funcionario' por máquina, y funcionario virtual por máquina virtual, o por proceso. También podemos cambiar 20 segundos por 20 milisegundos.

16.2.2. Proceso: - Programa en ejecución

En todo momento un proceso ejecuta un programa. No es obligada la relación 1 a 1.

- Un programa está en un papel y no lo está ejecutando ningún proceso.
- En un momento puede haber varios procesos ejecutando el mismo programa.
- Un proceso puede ejecutar un programa durante un rato, y en un momento dado cambiar el programa que ejecuta.

Una metáfora: Si convertimos los procesos en actores y los programas en papeles.

En un momento varios actores de un coro siguen el mismo papel. Un personaje sigue un papel (médico de familia), en el que pone que tome una pócima y se convierta (nuevo papel) en el doctor Monstruo...

16.2.3. Llamadas:

- `fork ()`

Resultan dos procesos casi idénticos (padre - hijo).

`fork` es conceptualmente un procedimiento, pero en C se usa con aspecto de función, devuelve un valor. En el proceso padre la función *fork* devuelve el identificador *pid* del hijo, en el proceso hijo la función *fork* devuelve 0 .

- `exit (`*res*`)`

El proceso termina. Cuando en un lenguaje como 'pascal' escribimos `END.` el

compilador incluye en ese punto una llamada al sistema *exit* (esto es una simplificación).

Al hacer *exit* un proceso comunica (brevemente) el resultado de sus gestiones. *Sin novedad* se codifica como `0` (cero), y en caso contrario se devuelve un valor distindo de cero. Pueden usarse distintos valores para reflejar distintas incidencias. Por ejemplo, un compilador puede indicar con `1` que ha detectado errores, y con `2` que no ha podido leer el fichero fuente.

¿Se muere?. Si el proceso padre no está esperando en un *wait* no se muere. Queda esperando para que el proceso padre reciba información suya. Cuando un proceso ha hecho `exit` y su proceso padre no le esté esperando en un *wait*, se dice que está en estado *zombi*.

• `wait` (*resh*)
Con `wait` un proceso padre espera por un proceso hijo que termine. Si tiene varios hijos espera a que acabe uno cualquiera. En *resh* obtiene el valor de terminación del proceso hijo. La función *wait* devuelve como valor el identificador (*pid*) del proceso hijo.

• `exec` (*fichero-ejecutable*)
El proceso pasa a ejecutar el programa contenido en `fichero-ejecutable`. El proceso sigue siendo el mismo. Tiene el mismo *pid* pero cambia el programa. (Si fuese un actor cambiaría el papel).

¿Qué pasa si termina antes el proceso padre que el proceso hijo?. Cuando el proceso padre de uno dado desaparece, el hijo es adoptado por el proceso `init` .

16.2.4. Uso más frecuente

El uso más frecuente y característico de estas cuatro llamadas al sistema aparece en el intérprete de comandos. En la figura 16.2 presentamos un resumen mínimo de la actividad del intérprete de comandos.

```
10  loop   ...   140   end
```

refleja que el intérprete de comandos repite un ciclo en el que recibe una petición y atiende esa petición. En esas pocas líneas no aparece la posibilidad de acabar (con `logout` o `exit`).

```
30   lee linea
```

Supongamos que se lee `cat f1` .

```
 10  loop
 20    escribe el 'preparado
 30    lee linea
 40    if fork () # 0 then
 50      (* proceso padre *)
 60      wait
 70    else
 80      (* proceso hijo *)
 90      expansion de '* '?
100      redireccion(es)
110      exec linea
120               --> (*con el t. exit *)
130      end
140    end
```

Figura 16.2: Seudocódigo del intérprete de comandos.

`fork () # 0`
En la línea 40 se ejecuta la llamada al sistema *fork*. El control pasa al sistema operativo, y éste crea un segundo proceso idéntico salvo en el valor devuelto por la función. **A partir de este momento tenemos dos procesos ejecutando el mismo programa**.

Vamos con el proceso padre:
El valor devuelto por *fork* será distinto de cero. La expresión booleana de la línea 40 será cierta. Se ejecutará la rama `then` de la sentencia `if`. El proceso padre hace *wait* y se queda esperando a que acabe su proceso hijo.

> Podíamos haber comenzado con el proceso hijo, y dejado para después el proceso padre. Es posible que suceda a la vez (concurrencia).
>
> Quizá alguna persona esté harta de tanto 'padre e hijo'. Puede leerlo como 'madre e hija' cambiando 'proceso' por 'máquina virtual'.
> (`sed -e 's/proceso/maquina virtual/g`) :-)

Ahora el proceso hijo:
En el proceso hijo, el valor devuelto por *fork* será cero. La expresión booleana de la línea 40 será falsa. Se ejecutará la rama `else` de la sentencia

`if`. El proceso hijo está ejecutando código del intérprete de comandos todavía. Se realiza cierto trabajo, como expansión de metacaracteres (`*`, `?`, ...) redirecciones, etc.

En un momento dado se hace *exec* y en el proceso hijo se cambia el programa pasando a ejecutarse el programa indicado en la línea que se leyó. En este ejemplo pasaría a ejecutarse el programa `cat` con `f1` como parámetro.

Normalmente, con el tiempo acaba el programa que se leyó en `30 lee linea` (por ejemplo `cat`). En el código correspondiente habrá un *exit*. Acaba el proceso hijo, y ya puede continuar el proceso padre.

Vuelta al proceso padre:
Después de cumplir con la línea: `60 wait` el proceso padre salta la rama `else` y vuelve a la línea 20.

> Podemos ver una analogía entre el comportamiento del código de este ejemplo y el funcionamiento de una llamada a procedimiento. El proceso padre se corresponde con el programa que invoca. El proceso hijo se corresponde con el procedimiento llamado. Tanto el proceso hijo como el procedimiento llamado, heredan o ven el entorno de quien los originó.
>
> Como diferencia tenemos que puede acabar el proceso padre y continuar el proceso hijo, y no sucede así con los procedimientos. También es diferente la comunicación. En muchos lenguajes el procedimiento llamado puede modificar la memoria del procedimiento que lo llamó. Entre procesos el hijo no puede modificar la memoria *ram* del padre, pero sí puede modificar un fichero compartido (que también es memoria, aunque de acceso más lento).
>
> Puestos a sintetizar, la combinación de las cuatro llamadas al sistema explicadas permite implementar una *llamada a procedimiento tolerante a fallos*. Un fallo en el proceso hijo no afecta a la memoria del proceso padre. esto parece coherente con el hecho de que UNIX fuese desarrollado por los 'laboratorios bell' que son también pioneros en *sistemas tolerantes a fallos*.

16.3. Los primeros procesos

Al encender el sistema se carga el programa `/unix` (puede ser otro nombre parecido). Inicialmente sólo hay un proceso y su nombre es `init` (fig. 16.3*a*).

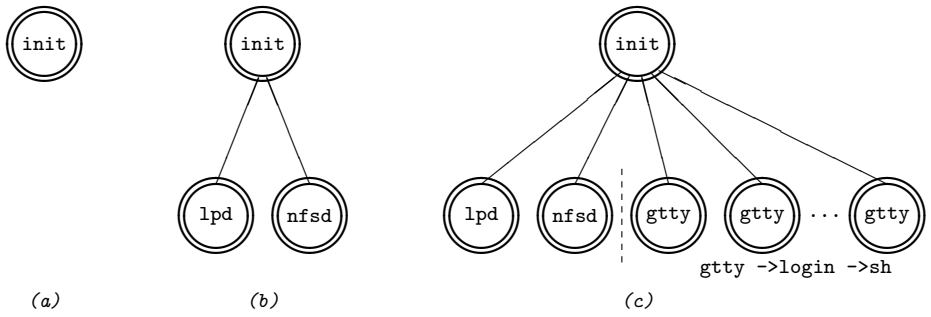


Figura 16.3: Los primeros procesos.

`init` lee uno o varios ficheros de inicialización. El nombre de estos ficheros suele empezar por `/etc/rc` . `init` realiza las operaciones indicadas en esos ficheros. Son típicas tareas tales como borrar los ficheros del directorio `/tmp` , configurar los dispositivos de red, revisar los discos por si hay algún problema con los ficheros. Además, es normal que los ficheros `/etc/rc...` incluyan líneas del estilo:

```
/usr/bin/lpd &
```

Con esta línea `init` crea un proceso hijo que ejecuta el programa `lpd` . Este programa no acaba, e `init` no espera (`&`). A programas como éste los llaman *deamon*. Esta palabra no tiene el sentido de 'demonio' (malo) sino más bien de 'duende'. Son de hecho servidores. Su código consiste en un ciclo: leo petición, sirvo petición, leo petición, etc... .

> El servidor de impresora recibe una petición de impresora. Si puede pone el fichero a imprimir. Si no puede, pone la petición en cola. Cuando acaba una impresión, mira si hay trabajo pendiente en la cola de impresión, y en caso afirmativo elige uno para imprimirlo.

Quedan varios procesos servidores, hijos de `init`, ejecutándose sin terminar (fig. 16.3*b*).

Luego `init` sigue leyendo el fichero `/etc/inittab` . En este fichero se indica qué líneas tienen conectados terminales, su velocidad, etc. Para cada una de esas líneas crea mediante *fork* un proceso con código `gtty` (fig. 16.3*c*).

Cuando un usuario hace `login` , el proceso de esa línea hace `exec sh`.

16.4. Creación de procesos según los comandos

En los cuatro casos que veremos a continuación consideraremos un usuario que acaba de comenzar su sesión. Le atiende el intérprete de comandos de entrada y el proceso es hijo de `init`.

En las figuras representaremos los procesos activos como círculos remarcados y los procesos que esperan (*wait*) sin remarcar. No remarcamos `init` aunque esté activo porque no interactúa con el usuario y para no distraer.

16.4.1. Un comando sencillo en modo interactivo

Supongamos que un usuario teclea, por ejemplo, `date >f1` .

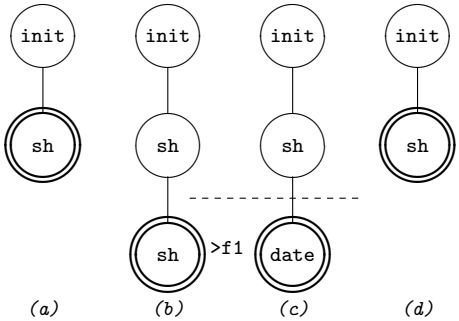


Figura 16.4: Ejecución interactiva de un comando.

Su intérprete de comandos (`sh`) hace *fork* y crea un proceso nieto del proceso `init` (e hijo del intérprete de comandos de entrada)(fig. 16.4*b*). El proceso intérprete de comandos de entrada espera en un *wait*. El intérprete de comandos nieto de `init` redirige su salida hacia el fichero `f1` .

El intérprete de comandos nieto de `init` hace *exec* para ejecutar `date` (fig. 16.4*c*). El programa `date` escribe la fecha por la salida estándar, es decir en el fichero `f1` .

El programa `date` acaba, hace *exit*. El intérprete de comandos de entrada puede continuar (fig. 16.4*d*). Escribe el carácter `$` . Está preparado para atender otro comando.

16.4.2. Un comando en modo no interactivo

Un usuario teclea, por ejemplo, `com1 &` .

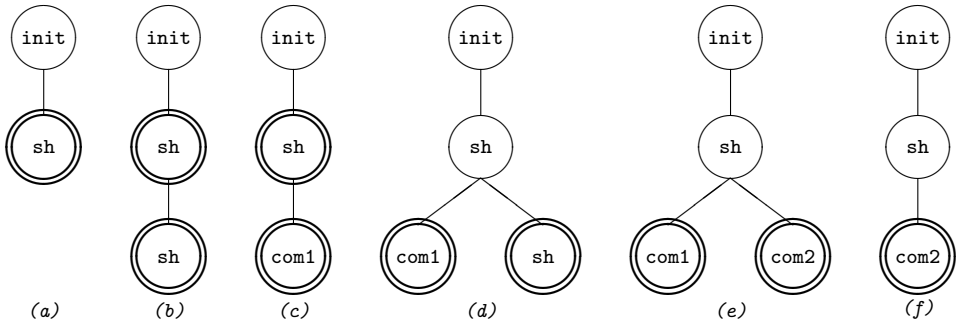


Figura 16.5: Ejecución no interactiva de un comando.

Su intérprete de comandos (`sh`) hace *fork*. Se crea un proceso nieto del proceso `init` (e hijo del intérprete de comandos de entrada)(fig. 16.5*b*). El proceso intérprete de comandos de entrada **no** espera en un *wait*. Escribe el prompt y está preparado para atender al usuario.

El proceso nieto de `init` (e hijo del intérprete de comandos de entrada) hace *exec* del código de `com1` (fig. 16.5*c*). Hay dos procesos activos.

La evolución depende de la duración de `com1`, de cuándo se teclee el siguiente comando, de la duración del siguiente comando, etc. La ejecución no interactiva introduce concurrencia, y con la concurrencia la evolución de un sistema deja de ser determinista.

Vamos a suponer que `com1` tiene una duración intermedia, que al poco tiempo se pide la ejecución interactiva de otro comando, `com2`, y que `com2` acaba después de `com1`. Esta es una secuencia posible, entre muchas otras, que hemos escogido arbitrariamente (hasta cierto punto).

El intérprete de comandos lee `com2` del teclado. Hace `fork`. Se crea otro proceso nieto del proceso `init` (e hijo del intérprete de comandos de entrada)(fig. 16.5*d*). El proceso intérprete de comandos de entrada espera en un *wait*. No le vale que acabe cualquier hijo, sino que espera específicamente la terminación del segundo hijo.

El segundo proceso nieto de `init` (e hijo del intérprete de comandos de entrada) de esta historia hace *exec* del código de `com2` (fig. 16.5*e*).

Acaba el primer proceso nieto de `init`. Hace *exit*. El proceso intérprete

de comandos de entrada hace *wait*. Como no ha acabado el que buscaba, vuelve a hacer *wait* (fig. 16.5*f*).

Acaba el segundo proceso nieto de `init`. Hace *exit*. El proceso intérprete de comandos de entrada hace *wait*. Ha acabado el que buscaba. Ya está activo otra vez (fig. 16.5*a*).

El seudocódigo presentado anteriormente para el intérprete de comandos (fig. 16.2) se puede modificar para adaptarlo al caso no interactivo.

```
35    nh := fork ();
40    if nh # 0 then
50      (* proceso padre *)
54      if interactivo then
55        repeat
60        until nh = wait ()
```

Figura 16.6: Parche para procesos no interactivos.

En la figura 16.6, la línea 54 se añade para que sólo se haga *wait* para comandos ejecutados en modo interactivo. Las líneas 55 y 60 repiten la llamada *wait* hasta que acabe el último proceso interactivo. La línea 35 se encarga de guardar el número del último proceso creado.

16.4.3. Varios comandos conectados por tubos

Supongamos que un usuario teclea, por ejemplo, `who | wc -l` .

El intérprete de comandos (`sh`) pide un *pipe* o *tubo* (fig. 16.7*b*).

Luego hace *fork* dos veces. Se crean dos procesos nietos del proceso `init` (e hijos del intérprete de comandos de entrada) (fig. 16.7*c*). Ambos ejecutan el comando `sh` o intérprete de comandos. El primero de ellos redirige la salida estándar hacia el *tubo*. El segundo redirige la entrada estándar leyendo los caracteres del *tubo*. El proceso intérprete de comandos de entrada queda esperando. En su código hay dos *wait*.

El primer proceso nieto de 1`init` hace *exec* del código de `who` , mientras que el segundo proceso nieto hace *exec* del código de `wc -l` (fig. 16.7*d*).

Con el tiempo ambos programas (`who` y `wc`) acabarán. Si tenemos dos procesos conectados por un *tubo* llamamos proceso *productor* al que tiene redirigida la salida estándar y proceso *consumidor* al que tiene redirigida

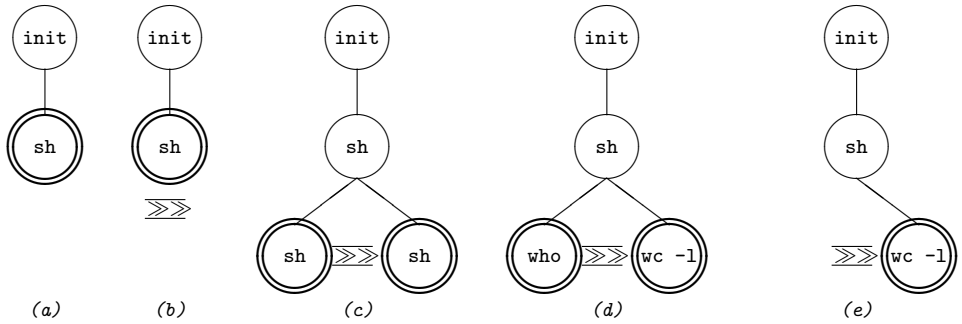


Figura 16.7: Ejecución de dos procesos conectados por un *tubo*.

la entrada estándar. En general es más común que acabe antes el proceso productor que el proceso consumidor.

Algunas veces puede acabar antes el consumidor. Por ejemplo con el comando `muchasLineas | head -1` . En la figura 16.7*e* hemos presentado el caso en que acaba primero el proceso productor.

Si acaba primero el proceso productor, en el tubo tendremos un *fin de fichero*, y normalmente el proceso consumidor acabará cierto tiempo después de leer la última información del tubo. Si acaba primero el proceso consumidor, el proceso productor recibe una notificación del sistema operativo: –¡Nadie te atiende (lee) en el tubo!. Y normalmente acaba.

Cuando acaban ambos procesos se han realizado dos llamadas al sistema *exit*. El intérprete de comandos de entrada ha cumplido con dos *wait* por sus dos procesos hijos recientes y ya puede continuar. Ya está activo (figura 16.7*a*).

16.4.4. Varios comandos entre paréntesis

Un usuario teclea, por ejemplo, `(com1; com2) >f` .

Su intérprete de comandos (`sh`) hace *fork* y queda esperando en un *wait*. Se crea un proceso nieto de `init` (e hijo del intérprete de comandos de entrada) (fig. 16.8*b*) para procesar todos los comandos encerrados entre paréntesis. Como la salida de los comandos entre paréntesis está redirigida, este proceso nieto de `init` redirige su salida hacia el fichero `f` .

A partir de este momento, es como si al intérprete de comandos nieto de `init` le llegase `com1 ; com2` seguido de un *fin de fichero*. Se repite dos

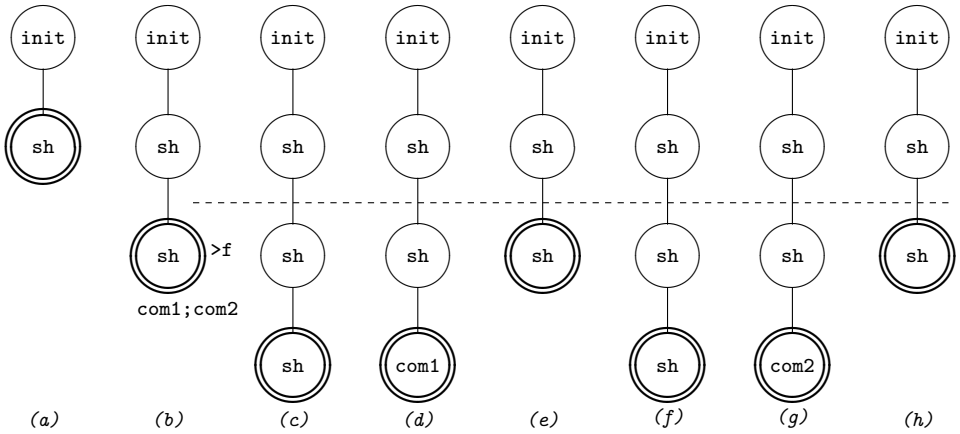


Figura 16.8: Ejecución de comandos entre paréntesis.

veces el mecanismo para la ejecución de un comando interactivo sencillo. La única diferencia con el caso del apartado 16.4.1 es que hay un proceso más.

> El intérprete de comandos nieto de `init` lee `com1`, hace *fork*, queda a la espera en un `wait`. Se crea un proceso biznieto de `init` con código de intérprete de comandos (fig. 16.8*c*).
>
> El intérprete de comandos biznieto de `init` hace *exec* para ejecutar `com1` (fig. 16.8*d*).
>
> Normalmente, a menos que sea un servidor, el programa de `com1` acaba, hace *exit*. El intérprete de comandos nieto de `init` puede continuar (fig. 16.8*e*).
>
> El intérprete de comandos nieto de `init` lee `com2`, hace *fork*, queda a la espera en un `wait`. Se crea otro proceso biznieto de `init` con código de intérprete de comandos (fig. 16.8*f*).
>
> El intérprete de comandos biznieto de `init` hace *exec* para ejecutar `com2` (fig. 16.8*g*).
>
> El programa de `com2` acaba, hace *exit*. El intérprete de comandos nieto de `init` puede continuar (fig. 16.8*h*).

El intérprete de comandos nieto de `init` ha llegado al final de los comandos entre paréntesis (como un fin de fichero) y hace *exit*. El intérprete de comandos de entrada puede continuar (fig. 16.8*a*).

En el ejemplo que hemos puesto (fig. 16.8), hay tres procesos con su salida estándar redirigida hacia el fichero `f` : el proceso nieto de `init` y los dos procesos biznietos de `init` que lo heredan del primero.

> Alguna implementación de intérprete de comandos aprovecha su conocimiento de cuál es el último de los comandos entre paréntesis para ahorrar un *fork* y hace directamente un *exec* en el paso *e* de la figura 16.8. Esto no afecta a los programas más frecuentes. Hay que empeñarse un poco para detectarlo. Por ejemplo, se puede ver poniendo `ps` al final de un grupo de comandos entre paréntesis.

16.5. Otros casos

Cuando se ejecuta un programa del intérprete de comandos (*script*) se crea mediante *fork* un proceso hijo del proceso intérprete actual. Este es el caso más frecuente.

Es posible hacer que el intérprete de comandos tome directamente las líneas de un fichero sin hacer *fork*. Se usa excepcionalmente.

Los comandos interpretados, como `cd` y `umask`, no siguen el esquema del apartado 16.4.1. No se ejecuta ninguna de las líneas 40 ... 130 del seudocódigo que aparece en la figura 16.2. El intérprete de comandos modifica su propia variable local.

Algunos programas, por ejemplo `vi` y `mail`, permiten usar el intérprete de comandos u otros comandos. Estos programas hacen *fork* y crean un proceso hijo que ejecuta (*exec*) el programa correspondiente, y se quedan esperando en un *wait* a que acabe el proceso hijo.

16.6. Ejercicios

Ejercicios sobre procesos en UNIX:
93f.10 (pág. 279), 93j.10 (pág. 285), 93s.10 (pág. 291), 94f.10 (pág. 297), 94j.10 (pág. 303), 94s.10 (pág. 309), 95f.8 (pág. 315), 95f.10 (pág. 315), 95j.7 (pág. 321), 95s.4 (pág. 326), 95s.13 (pág. 328), 96f.9 (pág. 333), 96f.14 (pág. 334), 96j.4 (pág. 338), 96s.8 (pág. 345), 97f.9 (pág. 351), 97j.10 (pág. 357), 98f.10 (pág. 369), 98j.7 (pág. 375), 98s.10 (pág. 381), 99f.7 (pág. 387).

Capítulo 17

awk

`awk` es un filtro para ficheros de texto. Es programable. En esto se parece al comando `sed` . `awk` define un lenguaje de programación. Sus operaciones básicas son la búsqueda de pautas (patrones, *patterns*) y las operaciones sobre campos o líneas.

> El nombre del comando `awk` viene de sus autores: Alfred V. **A**ho, Peter J. **W**einberger y Brian W. **K**ernigham.

17.1. Uso

`awk` *programa* *ficheros* aplica el *programa* sobre cada línea de cada fichero nombrado.

`awk '/o/;/o/' f1 f3` escribirá dos veces cada línea de los ficheros `f1` y `f3` que tenga una `o` . Las comillas no forman parte del programa, sino que están ahí para que el intérprete de comandos (`sh`, `tcsh`, ...) no interprete, husmee, meta la nariz.

El programa puede estar en un fichero, por ejemplo `dosoes` . El programa en la línea del comando `awk` se especifica a través de la opción `-f` *ficheroPrograma* . El ejemplo anterior se escribiría:

```
awk -f dosoes f1 f3
```

El contenido del fichero `dosoes` puede ser:

```
$ cat dosoes          $ cat dosoes
/o/                   /o/;/o/
/o/
```

En los programas `awk` el punto y coma (`;`) se puede cambiar por el *cambio de línea* y viceversa.

Si no se nombra un fichero de datos, `awk` procesará la entrada estándar. Mediante el carácter `-` se puede incluir la entrada estándar en la lista de ficheros.

`awk '/^1/' f1 - f3` escribirá las líneas del fichero `f1` , de la entrada estándar y del fichero `f3` que empiecen por `1` .

17.2. Estructura del programa

Un programa `awk` es una secuencia de pares:

dónde `{` *acción* `}`

La parte *dónde* puede tomar los valores especiales `BEGIN` y `END` .

La estructura de un programa `awk` queda:

`BEGIN {` *acción* `}`	prólogo (opcional)
dónde `{` *acción* `}`	cuerpo del programa (opcional)
...	
`END {` *acción* `}`	epílogo (opcional)

`awk`

- ejecuta la acción asociada a `BEGIN` antes de tratar ningún registro [1] de entrada.
- para cada registro de entrada:

  recorre la secuencia de pares *dónde* - *acción*. Para cada par:

  Si la parte *dónde* incluye ese registro, se ejecuta la *acción* asociada.

[1] En seguida veremos que en muchos casos *registro* se puede substituir por *línea*.

- ejecuta la acción asociada a `END` después de tratar el último registro de entrada.

  Como cada una de las tres partes del programa: prólogo (parte `BEGIN`), cuerpo del programa y epílogo (parte `END`), son opcionales, hay 7 combinaciones para formar un programa. Un programa puede tener sólo cuerpo del programa, o sólo prólogo, o sólo epílogo. Estas dos últimas son equivalentes, la salida no dependerá de la entrada. Un programa puede tener dos partes, o las tres partes.

  Quiero remarcar que el uso de `BEGIN` y `END` en `awk` es completamente diferente del uso de esas palabras en lenguajes de programación como *pascal*.

  En *pascal* cada `BEGIN` lleva asociado un `END` y viceversa. El papel de estas palabras es el de *paréntesis de sentencias*.

  En `awk` `BEGIN` y `END` son independientes, y cada uno lleva asociada una *acción*.

Tanto la parte *dónde* como la parte *acción* se pueden omitir. Las llaves sirven para saber si se trata de una u otra.

La acción por omisión es escribir el registro de entrada.

Si se omite la parte *dónde*, la acción se aplica a todos los registros de entrada.

Parte *dónde*

La parte *dónde* de `awk` es parecida a la de `sed` . El formato de la parte *dónde* es:

[*dirección* [, *dirección*]]

Las *direcciones* pueden ser *expresiones booleanas* o *pautas* (patrones, expresiones regulares entre eslases (`/`)).

Si se pone una pauta (expresión regular entre eslases), la acción se aplica a todos los registros en los que se encuentre la expresión regular.

Si se pone una expresión booleana la acción se aplica a todos los registros en los que la expresión booleana evalúe *cierto*.

Si se ponen dos pautas (expresiones regulares entre eslases), éstas definen unos *a modo de párrafos*.

El primer *a modo de párrafo* comenzará con el primer registro en que se encuentre (un ajuste a) la primera expresión regular. Acabará en el primer registro en que (después) se encuentre (un ajuste a) la segunda expresión regular. Acabado el primero, el segundo *a modo de párrafo* comenzará con el primer registro en que se encuentre (un ajuste a) la primera expresión regular.

Si se ponen dos expresiones booleanas, éstas también definen unos *a modo de párrafos*.

`/^>>/` selecciona los registros que comienzan por `>>` .

`NR >= 10` selecciona el registro décimo y los siguientes. `NR` es el número de registro.

`/^Comienzo/,/^Fin/` selecciona los *a modo de párrafos* delimitados por `Comienzo` y `Fin` (en la columna 1) .

Parte *acción*

La parte *acción* es una *secuencia de sentencias*.

Normalmente, las sentencias van en líneas distintas. Puede ponerse varias sentencias en la misma línea separándolas con puntos y comas (`;`).

`awk` tiene las *sentencias* siguientes:

- sentencias de *salida*:
  - `print` con formato implícito.
  - `printf` con formato explícito.
- sentencias de *asignación*.
- sentencias de *control de flujo*:
  - `if-else`
  - `while`
  - `for ( ; ; )` , `for ( in )`

Para hacer posibles las sentencias de asignación y las sentencias de control de flujo es necesario introducir las *expresiones*.

> Las expresiones de las sentencias de control de flujo serán de tipo booleano. También hemos visto que puede haber expresiones booleanas en las *partes dónde*.

17.3. Campos

`awk` considera el texto como una secuencia de registros y cada registro como una secuencia de campos.

El *tipo de datos* predominante en `awk` es el tipo *tira de caracteres*. Sus valores son las secuencias de caracteres, incluyendo la secuencia vacía. No hay tipo asociado a un solo carácter, se le considera una secuencia de longitud uno.

> Las constantes de tipo tira de caracteres se escriben entre comillas dobles.

El *delimitador de registros* por omisión es el cambio de línea (*line feed*, ascii 10). Si no se cambia el delimitador de registros, un registro es una línea.

El *delimitador de campos* por omisión es el carácter blanco. Se puede cambiar el valor del delimitador de campos en la línea de invocación a `awk` o en el prólogo (parte `BEGIN`).

`awk ... -F= ...` cambia el separador de campos a `=` .

`BEGIN { FS=":" }` cambia el separador de campos a `:` .

Si el separador de campos es el carácter `:` y en una línea (en un registro) tenemos:

```
:::agua::fuego
```

la tira `agua` es el cuarto campo, `fuego` es el sexto campo, hay seis campos y cuatro de ellos son tiras de caracteres vacías.

Si el separador de campos es el carácter blanco (␣) y en un registro tenemos:

```
␣␣␣agua␣␣fuego
```

la tira `agua` es el primer campo, `fuego` es el segundo campo, y hay dos campos. Con este separador no puede haber campos cuyo valor sea la tira vacía.

- Si el separador de campos es distinto del carácter blanco, entre cada dos separadores consecutivos y entre el comienzo de línea y el primer separador hay un campo.
- Si el separador de campos es el carácter blanco, las secuencias de caracteres blancos y tabuladores actúan como un único separador, y los

blancos y tabuladores situados al comienzo de la línea no delimitan ningún campo.

- Si el *separador de registros* es la tira (de caracteres) vacía, los registros vendrán delimitados por líneas vacías. En este caso, haciéndose caso omiso del separador de campos, cada línea será un campo.

17.4. Variables

Campos

En `awk` los campos se consideran un caso particular de *variable*.

`$1` es el campo primero,
`$2` es el campo segundo,
...
`$99` es el campo nonagesimonoveno.
`$0` es todo el registro.

El valor de un campo inexistente es la tira vacía.

Aunque se puede realizar asignaciones a campos, creo que en general es mejor evitarlo.

Si se conoce el lenguaje del intérprete de comandos, y se intenta establecer analogías puede haber cierta confusión. Para los intérpretes de comandos, `$`*algo* representa el **valor de** *algo*. Para `awk` `$` es el comienzo del nombre de un campo (o del registro).

Variables predefinidas

Algunas de las variables predefinidas de `awk` son:

- `NR` número de orden del registro actual. Si se procesan dos ficheros de 10 registros, el primer registro del segundo fichero el valor de `NR` será 11. En el epílogo, `NR` es el número de registros total.
- `NF` número de campos del registro actual.
- `FS` separador de campos (para la entrada).

- `RS` separador de registros (para la entrada).
- `FILENAME` nombre del fichero de entrada que se está procesando.
- `OFS` separador de campos para la salida.

  Su valor inicial es el carácter blanco (espacio, ascii 32).

Variables del usuario

`awk` admite variables del usuario. Los nombres de las variables son una secuencia de letras, cifras y subrayados (`_`) que no empiece por una cifra. El tamaño (mayúscula o minúscula) de una letra es significativo. Son variables diferentes `sol`, `Sol` y `SOL` .

> Dado que los programas de `awk` no serán grandes (normalmente), parece propio no utilizar nombres de variables largos, aunque siempre será bueno que los nombres sean significativos.

Las variables toman inicialmente el valor *tira vacía*. En muchos casos este valor es el adecuado y ahorra una asignación en el prólogo.

17.5. `print`

`print` *secuencia de expresiones y comas* imprime los valores de las expresiones.

Las comas en la *secuencia de expresiones y comas* generan caracteres *separadores de campos* en la salida.

```
print FILENAME, NR, $1, $2
```

escribe el nombre del fichero que está procesando, el número de registro, el primer y el segundo campos del registro. Intercala tres separadores de campos de salida.

```
$ cat diastros
luna-lunes-monday
marte-martes-tuesday
$ awk -F- '{print FILENAME, NR, $1, $2}' diastros
diastros 1 luna lunes
diastros 2 marte martes
```

Hemos construido un programa poniendo una sentencia entre llaves. El programa podía haber estado en un fichero. Para que el intérprete no partiese el programa por los caracteres blancos y para que no interpretase los `$1` , ..., hemos puesto el programa entre comillas sencillas.

```
print "fichero", FILENAME ":", NR, $1, $3
```

escribe la palabra `fichero`, el nombre del fichero que está procesando seguido de `:` , el número de registro y los campos primero y tercero. Intercala separadores de campo en la salida salvo entre el nombre del fichero y el carácter `:` .

```
$ awk -F- '{print "fichero", FILENAME ":", NR, $1, $3}' diastros
fichero diastros: 1 luna monday
fichero diastros: 2 marte tuesday
```

`print NF, $NF` escribe el número de campos en el registro y el último campo del registro. Si en un registro hay 4 campos al tratar ese registro `NF` valdrá 4. `$NR` será `$4` , es decir el cuarto campo, el último.

`print` equivale a `print $0` . Escribe todo el registro.

`print ""` escribe una línea vacía.

17.6. Funciones sobre tiras de caracteres

Suponemos que `s`, `s1` y `s2` son tiras de caracteres y `p` y `n` número enteros.

- `length (s)` devuelve la longitud de `s` .
- `substr (s, p, n)` devuelve la parte de `s` que empieza en la posición `p` y de longitud `n` caracteres (como máximo).

  La posición de los caracteres en la tira se numera a partir de uno.
- `substr (s, p)` devuelve la parte de `s` que empieza en la posición `p` y llega hasta el final de `s` .
- `index (s, s1)` devuelve la primera posición de `s` en que aparece la tira de caracteres `s1` . Si `s1` no aparece en `s` la función devuelve 0 .

- `s1 s2` devuelve el resultado de encadenar las tiras `s1` y `s2` .

  El carácter blanco actúa como operador de concatenación. Esto justifica el comportamiento de dos expresiones sin comas en la sentencia `print`. ¡Resuelto el caso del operador invisible!.

- `sprintf` se ve en la sección de `printf`, y `split` se ve en la sección de arrays.

Ejemplos

```
escala = "doremifasollasido"
```

`length (escala)`	devuelve 17.
`substr (escala,5,7)`	devuelve `mifasol` .
`substr (escala,5,100)`	devuelve `mifasollasido` .
`substr (escala,5)`	devuelve `mifasollasido` .
`index (escala,"aso")`	devuelve 8.
`substr (escala, index(escala,"re")+2 ,2)`	devuelve `mi` .
`substr(escala,1,2) substr(escala,5,2)`	devuelve `domi` .

```
apellYnombre = "Ramon y Cajal, Santiago"
posComa = index (apellYnombre, ",")
apell  = substr (apellYnombre, 1, posComa-1)
nombre = substr (apellYnombre, posComa+1 )
```

`index` nos permite, entre otras cosas, buscar subcampos en un campo o en una variable.

17.7. Operadores relacionales

`awk` tiene los seis operadores de comparación.

- `<` menor
- `<=` menor o igual

- `==` igual
- `!=` distinto
- `>` mayor
- `>=` mayor o igual

Hay que tener cuidado de no confundir la escritura del operador *igual*. Se utilizan dos caracteres. El signo igual (`=`) suelto se reserva para la asignación).

> En `awk` original esta equivocación se detecta y da lugar a un mensaje de error.
>
> En alguna versión ampliada, como `gawk`, no se detecta el uso de `=` en el lugar en que queríamos una comparación.

Los dos operadores siguientes sirven para preguntar si una tira de caracteres contiene un patrón. El segundo operando será una expresión regular entre eslases (`/`) .

- `~` contiene
- `!~` no contiene

`$1 ~ /e+d/` valdrá *cierto* si el primer parámetro contiene (`~`) una o más caracteres `e` seguido(s) de un carácter `d` .

`$1`	`$1 ~ /e+d/`
`leed`	*cierto*
`enredo`	*cierto*
`dado`	*falso*

17.8. Operadores lógicos

- `&&` y (secuencial)
- `||` o (secuencial)
- `!` no

Los operadores secuenciales sólo evalúan el segundo operando cuando es necesario para determinar el resultado.

> `(z > 0) && (y/z > x)` no dará nunca error de división por cero. Cuando `z` valga cero `(z > 0)` será falso y el operador `&&` (y-secuencial) **no evaluará** su segundo operando.

17.9. Sentencias de control de flujo

`awk` ofrece como sentencias de control de flujo `if`, `if-else`, `while` y `for`. A continuación se muestra su sintaxis.

```
if ( expresiónBooleana ) {
   sentencias }

if ( expresiónBooleana ) {
   sentencias }
else {
   sentencias }

while ( expresiónBooleana ) {
   sentencias }

for ( sentencia1 ;
      expresiónBooleana ;
      sentencia2 ) {
   sentencias }
```

Las llaves (`{ ... }`) están para delimitar las sentencias afectadas por la estructura de control de flujo. En el caso de ser una única sentencia, se pueden omitir las llaves.

La sentencia `for` descrita anteriormente es **equivalente** al grupo de sentencias con `while` que se escribe a continuación. Son dos formas de escribir lo mismo; cualquiera de ellas puede sustituir a la otra.

```
sentencia1
while ( expresiónBooleana ) {
   sentencias
   sentencia2 }
```

Ejemplos

```
{ if ($2 > max) max = $2 }
END { print max }
```

Queremos hallar el máximo de los valores del campo segundo. Suponemos que en `max` guardamos el valor máximo de los registros anteriores. Si el valor del campo segundo del registro actual (`$2`) es mayor que el guardado, asignamos el valor actual a `max`. Cuando hayamos tratado todos los registros (`END`) escribimos el valor de `max`.

Para resolver el problema en el caso general tenemos que asignar inicialmente (`BEGIN`) a la variable `max` un valor mínimo, o alternativamente tenemos que tratar de forma distinta el registro primero.

Si los valores de los campos segundos son tiras de caracteres, el programa funciona correctamente porque la variable `max` se inicializa al valor de tira vacía, que es el menor posible.

Si los valores de los campos segundos son números positivos, el programa funciona correctamente. La variable `max` se inicializa al valor de tira vacía, y la tira vacía se interpreta como 0 (cero), que es el menor posible.

> Aquí parece que hay trampa. Sí, un poco. No sólo los poetas tienen sus licencias. El tema lo trataremos con cuidado hablando de números y conversiones.

Si los valores de los campos segundos son números arbitrarios no nos queda más remedio que arreglar el programa anterior. Por ejemplo añadimos: `NR=1 {max = $2}` .

```
{ #  posicion mas a la derecha de $2 en $1
  t = $1 ;  i = 0
  while ( index (t,$2) > 0 ) {
    t = substr (t, 2)
    i = i + 1 }
  print i, $1, $2
}
```

Tenemos una serie de registros con al menos dos campos. Queremos hallar la posición más a la derecha de una tira (campo segundo) en otra (campo primero). Por ejemplo, la posición más a la derecha de `ki` en `kikiriki` es 7.

Se copia el campo primero en la variable `t`. Se va recortando por la derecha la tira guardada en la variable `t` mientras siga conteniendo el valor `$2` . Cuando ya no encontramos el valor `$2` , lo vemos porque `index` devuelve cero, el número de caracteres recortados (`i`) es la respuesta.

Los comentarios comienzan con un carácter almohadilla (`#`) y acaban con el fin de línea.

```
{ suma = 0
  for (i=1; i <= NF; i = i+1) {
    suma = suma + $i }
  print suma
}
```

Tenemos un fichero con varios números en cada línea. Queremos escribir la suma de los valores de cada línea. Con una sentencia `for` recorremos todos los campos `(i=1; i <= NF; i = i+1)` . No podemos prescindir de la inicialización de la variable `suma` porque cada línea es necesario que comience con el valor `0`.

```
if (posComa != 0) {
  ...  }
```

En un ejemplo anterior separábamos un apellido de un nombre contenidos ambos en una variable. Buscábamos una coma que los separaba. Dos de aquellas líneas podrían ir rodeadas por el `if` anterior.

17.10. Números

`awk` admite también el tipo *número*. Los números aparecen como resultado de funciones y de operaciones. Internamente se guardan como números reales[2].

17.10.1. Operadores aritméticos

- `+` suma

[2]Aunque en informática muchas veces se habla de *números reales* de hecho se trabaja con un subconjunto de los números racionales.

- `-` diferencia
- `*` producto
- `/` cociente
- `%` resto
- `( )` agrupamiento

```
$ cat p11.aw
END {print 5 % 3 , -5 % 3 , 5 % -3 , -5 % -3
     print 4 % 2.2
     print 6 / 3 ,  5 / 3, (1/7) *7 }
$ awk -f p11.aw /dev/null
2 -2 2 -2
1.8
2 1.66667 1
```

En la prueba anterior vemos el comportamiento del operador *resto* con números negativos y racionales. También se ve que la precisión es grande, y no se aprecia el error introducido al dividir y multiplicar por 7. El usuario curioso puede escribir un programa para conseguir visualizar ese error.

El usuario muchas veces tiene la impresión de que también se ofrece el tipo número entero. (De ilusión, ..., cuidado). Mientras no realicemos divisiones y hayamos comenzado con números enteros no tiene por qué haber sorpresas. (Supongo que no llegamos al desbordamiento).

17.10.2. Funciones aritméticas

- `int ( )` parte entera
- `exp ( )` exponencial
- `log ( )` logaritmo neperiano
- `sqrt ( )` raíz cuadrada

Para elevar un número a una potencia no entera usamos la fórmula $a^b = \exp(b \log(a))$.

Para calcular la raíz cuadrada es más claro (y rápido) usar la función `sqrt`.

Si el exponente es entero, es más preciso y rápido usar una instrucción `for`.

17.11. Conversión

`awk` convierte automáticamente los valores de tipo tira de caracteres a tipo numérico (y viceversa) cuando aparecen como parámetros de una función o como operandos de un operador que lo requiere. Por ejemplo, los operadores aritméticos fuerzan la conversión a tipo numérico.

```
dos = 2 ;  tres = 3 ; print ( dos tres ) + 4
```

escribe `27` .

En la conversión de tipo numérico, se genera la tira de caracteres que resultaría de la impresión de ese número.

Para convertir un objeto de tipo tira de caracteres, `awk` considera si esa tira corresponde a la representación de un número. En caso afirmativo se convierte a ese número. En caso negativo resulta el valor `0` (cero).

constantes numéricas:

constante **numérica**	valor tira de caracteres
0	0
1	1
.5	.5
.5e2	50

constantes *tira de caracteres*:

constante **tira de caracteres**	valor numérico
	0
“a”	0
“o”	0
”1”	1
”.5”	0.5
”.5e2”	50

Hay ambigüedad en el caso de los 6 operadores de comparación.

Si los operandos son de tipo numérico `awk` realiza la comparación como números. Si son de tipo tira de caracteres y no tienen interpretación como número, realiza la comparación como tira de caracteres.

> Hay un estándar POSIX de 1992 que especifica que cuando dos tiras de caracteres proceden de campos, `split`, etc. y tienen aspecto numérico deben compararse como números.

La posible ambigüedad de una subexpresión se puede romper. Si le sumamos cero o si le concatenamos la tira vacía determinamos que el tipo resultante es numérico o tira de caracteres respectivamente.

Con `if ($2+0 > max+0) max = $2` estamos seguros de que comparamos números.

17.12. printf

`printf (` *formato* `,` *lista de parámetros* `)`

`printf` escribe siguiendo un *formato* que se expresa como una tira de caracteres y que casi siempre es un literal (una tira fija). `printf` escribe en la salida casi todos los caracteres del *formato*. Trata de forma específica las secuencias que empiezan con eslás inverso (`\`) y las que empiezan con `%` .

Las secuencias que empiezan con eslás inverso se usan para representar los caracteres no imprimibles y algunos de movimiento. Por cada secuencia que empieza por `%` debe haber un parámetro en la *lista de parámetros* de `printf` . Los parámetros (expresiones) (su valor) se escriben siguiendo las indicaciones de la correspondiente secuencia que empieza por `%` ,

```
END{ printf ("muestras: %4d promedio: %8.2f\n", NR, suma / NR)}
```

La línea anterior:

- escribe literalmente `muestras:` y `promedio:` con los caracteres blancos que se ven entre comillas.
- `%4d` reserva cuatro espacios para escribir en formato decimal el valor de `NR` y

- `%8.2f` reserva ocho espacios para escribir en *coma flotante* el valor de `suma / NR` (dos de ellos para los decimales).

- Acaba escribiendo un cambio de línea (`\n`).

Secuencias que empiezan por eslás inverso:

`\n` representa el carácter *cambio de línea*. `printf` no escribe el cambio de línea si no se le indica explícitamente.

`\r` representa el carácter *retorno de carro*.

`\t` representa el carácter tabulador.

`\033` representa el carácter cuyo ascii es 33 (octal) (*escape*). Se puede indicar cualquier carácter por su código octal.

Las secuencias que empiezan por `%` acaban por una letra que indica el tipo y formato del objeto:

`%d` decimal.

`%o` octal.

`%x` hexadecimal.

`%f` coma fija. Por ejemplo π es `3.14` en formato `%5.2f` .

`%e` notación científica. Por ejemplo π es `0.314e+1` en formato `%6.2e` .

`%s` tira de caracteres (string).

Los números y signos modifican el formato. Cuando sobra espacio, los números tienden a escribirse a la derecha y las tiras de caracteres a la izquierda.

`%6d` la anchura del campo es 6 caracteres. El primer número indica la anchura.

`%06d` Un cero a la izquierda de la anchura indica que el campo numérico se rellene a ceros.

`%-6d` Anchura 6 y ajustado a la izquierda (por el signo menos).

`%+9s` Anchura 9 y ajustado a la derecha (por el signo mas).

`%7.3f` Tres caracteres reservados para la parte decimal y la anchura total es 7 caracteres, incluyendo el punto y eventualmente el signo (menos).

17.12.1. sprintf

`sprintf` es semejante a `printf`. Aplica el formato a la lista de parámetros. No escribe, la salida formateada es el resultado de la función.

```
{ ...
  horaMinuto = sprintf ("%02d:%02d", hora, minuto)
  ... }
```

17.13. Arrays

`awk` permite arrays[3] de una dimensión.

No hace falta declararlos. (Ni hay forma de hacerlo).

Sus componentes se crean al citarlos (dinámicamente). Su valor inicial es la tira vacía.

Los valores de los índices pueden ser de tipo numérico y de **tipo tira de caracteres**.

`for (` *variable* `in` *variableArray* `)`
permite recorrer todos los componentes de un array.

Espero que al lector no le falten ejemplos de uso de arrays cuyo índice sea un número. Algunos de los ejercicios del fin del capítulo están resueltos.

Un ejemplo

Supongamos que tenemos unos registros con informes de ventas.

```
%cat facturado
juan   1000
jorge   500
jaime   700
juan    100
```

Ejecutamos un programa `awk` cuyo cuerpo es
`{totalVentas [$1] = totalVentas[$1] + $2}` .

[3] No me convence el término *array*, pero no encuentro otro mejor.

Cuando va a tratar el primer registro no existe ningún elemento del array `totalVentas`. Se hace referencia al elemento `totalVentas["juan"]`. Se crea ese elemento del array con valor la tira vacía. La suma implica una conversión de la tira vacía al valor numérico 0. Se incrementa en `1000` (`$2`, el segundo campo). Se almacena el número `1000` en el elemento `totalVentas["juan"]`.

De forma semejante se procesan los registros segundo y tercero.

Cuando toca procesar el registro cuarto el elemento `totalVentas["juan"]` ya existe. Se incrementa y se almacena `1100`.

```
END { for (v in totalVentas)
        print v, totalVentas [v] }
```

Cuando recorremos, con `v`, todos los valores del índice del array no vemos ningún orden razonable.

> Podemos sospechar que estos arrays se implementan mediante *tablas hash*.

Podemos arreglar el desorden de salida poniendo a continuación del programa `awk` un `sort`:

`awk ... | sort`	orden alfabético de vendedores, o
`awk ... | sort +1nr`	orden por cifra de ventas.

17.13.1. `split`

`n = split(s, varr, sep)`
parte la tira `s` en `n` componentes `varr[1]` , ... , `varr[n]` usando `sep` como separador .

```
n = split ("hola, que tal estas", pals, " ")
```

Después de ejecutar la sentencia anterior: `n` vale `4` , `pals[1]` vale `hola,` y `pals[4]` vale `estas` .

17.14. Operadores de asignación

Contracciones

En `awk` se puede escribir `x += d` como contracción de `x = x + d` . En este caso el ahorro al escribir o leer es pequeño. Mayor es el ahorro si tenemos que programar `ornitorrincos +=1` .

Hay más razones para estas contracciones. Los compiladores y los intérpretes ven facilitada su decisión de cuándo usar registros de CPU para ciertas variables. (Creo que esto era más importante antes que ahora).

Para las personas es más fácil leer la forma contraída, y el código tendrá menos propensión a errores.

- `+=` `x += y` equivale a `x = x + y`
- `-=` `x -= y` equivale a `x = x - y`
- `*=` ...
- `/=`
- `%=` `x %= y` equivale a `x = x % y`

En toda esta sección `y` se puede substituir por cualquier expresión numérica.

Operador

- `=` asignación.
  El valor de `(x = y)` es el valor de `y` .

En `awk` se pueden escribir sentencias tales como `a = (b = c)` , que es equivalente a la pareja de sentencias `b=c ; a=c` .

Nadie (casi) está obligado a escribir usando la primera forma. Pero si se encuentra escrita creo que todos deben saber interpretarla.

La asignación presenta dos posibilidades. Es una sentencia y es una expresión. (¿Será *géminis*?).

En informática, en situaciones como ésta, se habla de *efecto lateral*. ¿Cuál es el significado de `i=4; a[i]=(i=5)` ? Los prudentes dicen: "*efecto lateral*, con muchísimo cuidado". Los radicales dicen: "*efecto lateral*: no, gracias".

Las contracciones del comienzo de la sección también son operadores.

- `++` situado antes de una variable primero la incrementa en una unidad y luego toma el valor (incrementado).

  Situado después de una variable toma su valor, y luego incrementa la variable.

- `--` situado antes de una variable primero la decrementa en una unidad y luego toma el valor (decrementado).

  Situado después de una variable toma su valor, y luego decrementa la variable.

`++x` **equivale a** (se puede substituir por) `x += 1` .
`x++` es semejante (parecido) a `x += 1`
.

`--x` **equivale a** (se puede substituir por) `x -= 1` .
`x--` es semejante (parecido) a `x -= 1`
.

Ejemplo

Supongamos que `x` vale 5.
`b = a[x++]` es equivalente a `b = a[x] ; x = x + 1` . El valor de `b` será el mismo que el valor de `a[5]`.
`b = a[++x]` es equivalente a `x = x + 1 ; b = a[x]` . El valor de `b` será el mismo que el valor de `a[6]`.
`b = a[x+=1]` es equivalente a ... (ejercicio para el lector).

El uso de los operadores `++` y `--` como sentencias y 'desaprovechando' su aspecto de operador mantiene los programas legibles y no introduce ningún problema. Podemos substituir la línea del ejemplo de `for` :

```
for (i=1; i <= NF; i++) {
```

17.15. El programa entre comillas

Entre comillas sencillas un cambio de línea no acaba un comando. Así se puede escribir un programa como el que sigue:

```
join f1 f2 | awk '{s += $2 * $3
                   print $0, $2*$3 }
```

```
                END {print "total = ", s}' | \
pr
```

Podemos componer un único parámetro con texto y entrecomillados consecutivos sin blancos en medio.

```
$ cat columna
awk '{print $'$1'}'
```

Si escribimos `columna 3` tenemos un programa que filtra la tercera columna. El programa equivalente escrito con comillas dobles es:

```
awk "{print \$$1}"
```

El primer carácter dólar no es especial y forma parte del programa *awk*, mientras que el segundo indica que substituyamos `$1` por el primer parámetro.

Este programa también se puede escribir con eslases inversos:

```
awk {print\ \$$1}
```

17.16. Limitaciones / extensiones

`awk` no tiene declaración de tipos, construcción de tipos, tipo *record*, funciones o procedimientos del usuario, etc. No es un lenguaje pensado para escribir programas grandes. Su terreno está entre varias unidades o decenas de líneas.

Por no tener, no tiene ni valores booleanos. No hay un *cierto* en el lenguaje. Si el usuario lo necesita, se define un valor como *cierto* y sigue el convenio hasta el final de su programa.

En 1985, los autores de `awk` ampliaron el lenguaje. Durante cierto tiempo no lo ofrecieron. Ahora lo ofrecen (¿y cobran?). El nombre del comando es `nawk`.

El grupo GNU ofrece una versión ampliada de `awk` con el nombre de `gawk`. En las distribuciones actuales de LINUX, `awk` es un alias de `gawk`.

Ambas versiones ofrecen la posibilidad de que el usuario defina funciones.

```
$ cat pru1.aw
function intercambia(t1,t2,   taux){
taux = t1
t1   = t2
t2   = taux}

END { v1 = "hola"
      v2 = "adios"
      intercambia (v1,v2)
      print v1, v2 }
```

El ejemplo anterior es trivial como programa. Hay que señalar que quiero que `taux` sea una variable local. La pongo como parámetro formal de la función, separada por blancos de los demás parámetros. Los parámetros formales de verdad se escriben sin blancos intercalados. (No parece muy elegante, aunque funcione.)

17.17. Ejercicios

Escribe un (¿breve?) programa `awk` que ...:

1. se comporte como `cat` (Pista, tiene 7 caracteres o menos.)
2. se comporte como `uniq`
3. se comporte como `uniq -c`
4. se comporte como función inversa de `uniq -c` . Con la salida de `uniq -c` genere la entrada.
5. se comporte como `paste` (de dos ficheros).
6. se comporte como `nl -ba`
7. se comporte como `nl`
8. se comporte como `head -1`
9. se comporte como `tail -1`
10. se comporte como `join` (o parecido, o mejor).

11. escriba la longitud de la línea más larga.
12. .. y también la línea (una de ellas) más larga.
13. ponga las líneas del revés letra a letra.
14. ponga el fichero del revés (línea a línea).
15. pinte un histograma (líneas)(leyendo números pequeños).
16. escriba las 365 fechas de este año (una por línea: 1 Enero, 2 Enero ..)
17. escriba la última línea de **cada** fichero (`awk -f p17.aw f1 f3 f6`)
18. cambie las mayúsculas por minúsculas
19. cambie cada letra minúscula por la siguiente minúscula, y `z` por `a` .
20. se comporte como un `grep` con contexto (± 3 líneas)(para un tira fija).
21. ...
22. escriba el mismo programa.
23. escriba otro programa, que a su vez escriba el anterior (flip/flop).

Ejercicios de examen sobre `awk` tipo seguimiento:
93f.6 (pág. 278), 93s.5 (pág. 290), 93s.6 (pág. 290), 94j.7 (pág. 303), 94s.8 (pág. 309), 95f.9 (pág. 315), 95j.5 (pág. 320), 95s.5 (pág. 326), 96f.5 (pág. 332), 96j.3 (pág. 338), 96j.10 (pág. 339), 96s.3 (pág. 344), 97f.4 (pág. 350), 97j.5 (pág. 356), 97j.11 (pág. 357), 97s.2 (pág. 362), 97s.3 (pág. 362), 97s.6 (pág. 362), 98f.8 (pág. 369), 98j.3 (pág. 374), 98j.10 (pág. 375), 98s.2 (pág. 380), 99f.3 (pág. 386).

Ejercicios de examen sobre `awk` tipo programación:
93f.15 (pág. 280), 93f.18 (pág. 281), 93j.15 (pág. 286), 93j.18 (pág. 287), 93s.15 (pág. 292), 93s.18 (pág. 293), 94f.16 (pág. 298), 94f.19 (pág. 299), 94j.16 (pág. 305), 94j.19 (pág. 305), 94s.16 (pág. 310), 94s.19 (pág. 311), 95f.16 (pág. 316), 95f.19 (pág. 317), 95j.16 (pág. 322), 95j.19 (pág. 323), 95s.16 (pág. 328), 95s.19 (pág. 329), 96f.16 (pág. 334), 96f.17 (pág. 335), 96j.16 (pág. 340), 96s.16 (pág. 346), 96s.19 (pág. 347), 97f.15 (pág. 352), 97f.18 (pág. 353), 97j.15 (pág. 358), 97s.15 (pág. 364), 97s.18 (pág. 365), 98f.15 (pág. 370), 98j.15 (pág. 376), 98s.15 (pág. 382), 99f.15 (pág. 388).

Capítulo 18

make

`make` facilita la actualización de ficheros.

18.1. Ejemplos

18.1.1. Un caso sencillo

Supongamos que otra persona actualiza de vez en cuando un fichero de nombre `fno` . Nuestro trabajo consiste en realizar una serie de operaciones a partir del fichero `fno` y obtener un fichero `fce` actualizado cuando nos lo pidan.

Puestos a simplificar, la serie de operaciones se puede reducir a una copia (operación identidad).

Escribimos (editamos) un fichero `mande` de contenido:

```
cp fno fce
```

y ponemos permiso de ejecución al fichero `mande` .

```
chmod u+x mande
```

Cuando nos piden la versión actualizada de `fce` tecleamos `mande` .

Este fichero `mande` supone un adelanto, sobre todo si los nombres de los ficheros en juego se alargan. (Los ficheros podrían llamarse `fnotedirequeno` y `fnotedirequena`).

Si el fichero `fno` no ha cambiado desde la última copia, podemos ahorrarnos la operación de copiar. Para saberlo basta con comparar las fechas

de última modificación de los ficheros `fno` y `fce`. (fechas que salen con el comando `ls -l`).

El comando `make` se apoya en un fichero con reglas cuyo nombre usual es `Makefile` . Escribimos en `Makefile` :

```
fce: fno
        cp fno fce
```

que podemos leer como:

> "el fichero `fce` depende del fichero `fno`,
> cuando haya que recrear `fce` haz `cp fno fce` ."

En este fichero `Makefile` tengo una regla con tres partes:

objetivo : `fce`

lista de requisitos : (uno sólo) `fno`

lista de acciones : (una sola) `cp fno fce`

El objetivo va seguido del carácter '`:`' , y de la lista de requisitos. A continuación va la lista de acciones. Aunque sólo sea sintaxis, es IMPRESCINDIBLE poner un carácter tabulador al comienzo de cada línea de acciones.

Tecleamos `make` . Si la fecha de última actualización de `fno` es posterior a la de `fce`, hace `cp fno fce`; si es anterior, dice `fce` *esta al día* (en inglés). Si no existe el fichero `fce` , asimila este caso a tener un `fce` no actualizado y hace `cp fno fce` .

> Las fechas de los ficheros podrían ser iguales. ¿ Que haría `make` ? En este ejemplo parece más prudente hacer `cp` y asegurar que `fce` está al día.
>
> La mayoría de los sistemas tipo UNIX (escribo en 1996) guardan la fecha con resolución de un segundo.
>
> Usando `ls -l` sólo vemos la fecha de última modificación con resolución de minutos. Para hacer pruebas necesitamos conocer las fechas de última modificación de los ficheros objetivo y requisitos. Si creamos (o actualizamos) varios ficheros: o bien nos acordamos del orden en que lo hacemos, o bien esperamos un minuto (o más) entre una actualización (o creación) y la siguiente.

1) Para ver el comportamiento de `make` creamos los dos ficheros con contenidos distintos. P. ej.

```
echo uno >fno
echo dos >fce
```

Si hemos creado `fno` y `fce` en este orden y hacemos `make` nos responderá que `fce` esta al día.

2) Queremos que la fecha de última actualización de `fno` sea posterior. Podemos editar `fno` y poner por ejemplo `tres`, saliendo con `:wq` si usamos `vi` . También podemos teclear

```
echo tres >fno
```

La forma más sencilla de cambiar la fecha de última modificación de un fichero es con el comando `touch` .

`touch fno` pone como fecha de última modificación del fichero `fno` la fecha actual. Si no existe, lo crea.

Si ahora hacemos `make` en la pantalla veremos

```
cp fno fce
```

El comando `make` escribe en la pantalla (en la salida estándar) los comandos que va a ejecutar antes de ejecutarlos.

Si escribimos `cat fce` comprobamos que se ha realizado la copia.

3) Podemos probar a borrar `fce` (`rm fce`) y hacer `make` .

Si ejecutamos dos veces el comando `make`, la segunda vez nos dirá que `fce` esta al día independientemente de la situación inicial.

4) Borramos `fno` y hacemos `make` .

El sistema nos responde que no sabe cómo hacer `fno` .

18.1.2. Versión segunda

Ante el éxito de nuestra administración del proyecto “fno-fce” nos encargan mantener actualizado un fichero `fsu` . El fichero `fsu` tendrá las 10 primeras líneas del fichero `fce`.

Editamos el fichero `Makefile` y añadimos nuestro conocimiento de los requisitos y comandos para construir `fsu`. Añadimos la regla:

```
fsu: fce
        head fce > fsu
```

Ahora al comando `make` , con este fichero `Makefile` , podemos pedirle `make fce` o `make fsu` . Le indicamos cuál es el objetivo (`fce` o `fsu`).

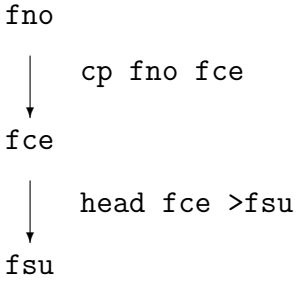
Si escribimos simplemente `make` el objetivo del comando `make` será el de la regla situada primera en el fichero `Makefile` .

¿ Qué hace `make fsu` ? :

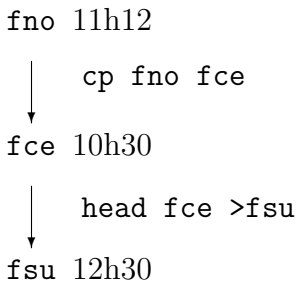
- Como `fsu` depende de `fce` , marca como subobjetivo actualizar `fce`.
- Si `fce` está al día respecto a `fno`:
  - Si `fsu` está al día respecto a `fce` no hay que hacer nada.
  - Si `fsu` no está al día respecto a `fce` , señala y hace `head fce > fsu` .
- Si `fce` no está al día respecto a `fno`:
  - señala y hace `cp fno fce` y luego
  - señala y hace `head fce > fsu` .

18.1.3. Lo mismo contado de otra forma

- Pintamos el grafo de dependencia correspondiente al fichero `Makefile` :



- Ponemos las fechas de última actualización de los ficheros. (Suponemos que es el mismo día, por ejemplo, y ponemos)



- A partir del objetivo (`fsu` en este caso) sólo nos interesa el subgrafo formado por el objetivo, los requisitos, los requisitos de los requisitos, etc.

- Se recorre hacia arriba el subgrafo. Si la fecha de un fichero objetivo es anterior a la fecha de un fichero requisito se marca la acción como pendiente.

Con las fechas del ejemplo marcamos como pendiente '`cp fno fce`' .

- Se recorre hacia abajo el subgrafo. Si una acción estaba pendiente se señala y se realiza, y se marcan como pendientes las acciones situadas debajo.

Con las fechas del ejemplo:

- se señala y se realiza `cp fno fce` , se marca como pendiente '`head fce > fsu`' .
- se señala y se realiza `head fce > fsu`

18.1.4. Pruebas

Probar todos los casos de esta versión es muy laborioso.

Si hacemos `make fce` estamos en la misma situación que en la primera versión.

Si hacemos `make fsu`:

- puede estar actualizado `fsu`
- puede estar actualizado `fce`
- puede ser `fno` posterior a `fce` y `fce` posterior a `fsu`
- puede ser `fno` posterior a `fce` y `fsu` posterior a `fce`
- puede no existir `fsu`
- puede no existir `fce`

- puede no existir `fno`
- ...

18.2. make

make *objetivo* ≡

- **make** *requisitos*
- **sii** fecha (*objetivo*) < max (fechas (*requisitos*))
  aplica acciones de la regla de *objetivo*

18.2.1. Salida estándar de make

`make` normalmente antes de realizar cada acción la señala, la escribe por la salida estándar.

Por ejemplo si en `Makefile` tenemos:

```
fsu: fce
        head fce | tee fsu
```

y hacemos `make` , la salida puede ser (dependiendo de fechas)

```
head fce | tee fsu
linea 1 de fce
linea 2 de fce
...
```

Con la opción `-n` , (`make -n`) señala (amaga) sin realizar las acciones. La salida de `make -n` con el `Makefile` anterior sería

```
head fce | tee fsu
```

Cuando queremos, al revés, que realice las acciones sin señalarlas, podemos usar la opción '`-s`' (silencioso). Prefiero no usar el modo silencioso para ver el avance y localizar mejor la causa de los diagnósticos.

Cuando queremos que una acción particular no se señale anteponemos un carácter '`@`' a la acción. (Sigue habiendo un tabulador al comienzo de la línea). Un caso típico es

```
        ...
        @ echo  CUIDADO xxxxx
```

18.2.2. Fechas

`make` supone que las fechas son correctas, o que al menos el tiempo del sistema crece con el tiempo real.

Si en algún momento el tiempo decrece, p.ej. alguien corrige una hora adelantada retrasando el reloj, se pueden obtener resultado incoherentes. Si se detecta este problema, la solución es borrar todos los ficheros que figuren como objetivo de alguna regla.

`make` también supone que no hay otros procesos trabajando con sus ficheros (requisito(s) u objetivo(s)). Si no es cierta esta hipótesis es posible obtener resultados incoherentes.

18.2.3. Comentarios

Podemos introducir comentarios con un carácter '`#`' en la primera columna, o a continuación del carácter '`#`' añadido a una línea útil de `Makefile` .

18.3. Reglas ...

18.3.1. ... sin prerrequisitos

Si escribimos reglas como la siguiente:

```
clean:
       rm -rf *.o a.out config.h
```

y hacemos `make clean` , utilizamos `make` como un *script*. Esta regla sólo tiene la pega de que deja de funcionar si accidentalmente creamos un fichero de nombre `clean` .

En este ejemplo se ve que no es necesario que el objetivo de una regla sea un nombre de fichero.

Para imprimir ...

... podemos usar un esquema parecido.

```
imprimir:
        lpr f1 f2
```

En este caso si invocamos dos veces `make imprimir` , la segunda no nos dice 'imprimir está actualizado'. Este `imprimir` es incondicional; siempre imprime.

Una regla un poco mejor es:

```
imprime: f1 f2
        lpr f1 f2
        touch imprime
```

En este caso si invocamos dos veces `make imprime` , la segunda nos dice 'imprime esta actualizado'. Este `imprime` es condicional; sólo imprime si ha cambiado `f1` o `f2` .

Si lo que queremos es que sólo se imprima lo más reciente la regla es

```
imprime: f1 f2
         lpr $?
         touch imprime
```

y esto funciona porque `$?` es la lista de requisitos más nuevos que el objetivo.

18.3.2. ... sin acciones ...

... son posibles. Se usan para agrupar varios objetivos bajo un nombre. Por ejemplo:

```
all: sube baja izquierda derecha
sube: sube.c ...
        ...
```

18.3.3. ... con varios objetivos

Una regla puede tener dos o más objetivos. Por ejemplo:

```
f0 f9 :  F
        head F >f0
        tail F >f9
```

Este estilo tiene sentido cuando los ficheros (`f0` y `f9` en el ejemplo) forman una sola entidad lógica, y se quiere siempre actualizarlos simultáneamente.

18.4. Ejecución de las acciones

`make` invoca una `sh` para cada línea de la lista de acciones. Si hacemos `cd` en una línea, no afecta a la línea siguiente. Si queremos que una acción se extienda a más de una línea, pondremos un carácter '\' seguido inmediatamente del cambio de línea, y un tabulador al comienzo de la línea siguiente.

Si una línea con acciones devuelve un resultado distinto de cero (interpretado como error) se acaba la ejecución del comando `make` . Si queremos que el resultado de una línea se ignore, se pone un signo menos al comienzo de la línea (y detrás del carácter tabulador). Por ejemplo:

```
objn: re1 re2
        ...
        - rm tmp2
        ...
```

18.5. Macros (o variables) de `make`

A veces, nos encontramos con una tira de caracteres que aparece en dos o más lugares del fichero `Makefile` , y además es previsible que haya que cambiar esa tira de caracteres, o es una lista larga de ficheros. En cualquiera de estos casos es conveniente definir una macro de `make` . (También les llaman variables).

las macros se definen:

```
NOMBRE = valor
```

y se obtiene su valor con

```
$(NOMBRE)    o    ${NOMBRE}
```

Se puede omitir los paréntesis o llaves si el nombre de la macro es un sólo carácter.

En el ejemplo de imprimir se podía poner:

```
FIMPRIMIR = f1 f2
...
imprime: $(FIMPRIMIR)
```

También es frecuente que el nombre de un comando esté en una macro:

```
#CC = cc
CC = gcc
...
hola : hola.c
        $(CC) -o hola hola.c
```

Como `make` utiliza `$` para indicar el valor de una variable, cuando queramos introducir un carácter `$` en una acción escribiremos `\$` o `$$` . (`make` es un usurero, se queda con un dólar de cada dos :-).

18.6. Esquemas

18.6.1. ... del usuario

(En los manuales en inglés a esto le llaman 'reglas' (Rules)).

A veces el usuario descubre que su `Makefile` se repite, se repite ... . Por ejemplo:

```
...
f1.dos: f1.unix
        unix2dos f1.unix f1.dos
f2.dos: f2.unix
        unix2dos f2.unix f2.dos
indice.dos: indice.unix
        unix2dos inidice.unix indice.dos
...
```

> (en inglés '2' escrito 'two' se pronuncia parecido a 'to' y en jerga informática se usa para nombrar traductores, en este caso `unix2dos` traduce de formato UNIX a formato DOS) .

Podemos enseñarle a `make` cómo obtener ficheros con terminación `.dos` a partir de ficheros con terminación `.unix` . Lo conseguimos mediante:

```
.SUFFIXES: .unix .dos
.unix.dos:
        unix2dos $< $@
```

La línea `.SUFFIXES` es una presentación de sufijos nuevos. La macro '`<`' guarda el valor del requisito. La macro '`@`' guarda el valor del objetivo.

El esquema anterior lo podemos leer:

> " Te presento los nuevos sufijos '`.unix`' y '`.dos`'
> cuando quieras obtener un fichero acabado en '`.dos`'
> mira si tienes un fichero de nombre semejante acabado en '`.unix`'.
> En caso afirmativo ejecuta `unix2dos` con parámetros el prerrequisito y el objetivo "

Las seis líneas anteriores (`f1.dos` .... `indice.dos` ...) serían innecesarias.

18.6.2. Esquemas predefinidos

El comando `make` conoce previamente una serie de sufijos y esquemas sin que el usuario le diga nada.

Por ejemplo conoce al menos las terminaciones `.c .o .a .l .y` y los esquemas para obtener ejecutables y pasos intermedios a partir de estas terminaciones.

Se llega a que si tenemos un fichero fuente del lenguaje-c '`prueba.c`' con un programa sencillo, y sin tener fichero `Makefile` , hacemos `make prueba` y se compila correctamente el programa obteniendo el ejecutable `prueba` .

18.7. Letra pequeña

El primer objetivo de una regla no puede empezar por el carácter '`.`' .

Se pueden usar ficheros cuyo nombre comience por '`.`' anteponiendo un falso objetivo. Por ejemplo:

```
carabina .fichero : pre
```

18.7.1. `make` y directorios

Es costumbre que la mayoría de los ficheros nombrados en `Makefile` estén en el mismo directorio. Esto se cumple en mayor grado para los objetivos.

En los proyectos grandes, el trabajo se distribuye entre varios directorios, y generalmente están agrupados formando un árbol.

En estos casos, desde el fichero `Makefile` de un directorio se puede/suele invocar ejecuciones de `make` en los subdirectorios.

18.7.2. Más ...

... se puede aprender leyendo los manuales, leyendo buenos ejemplos y practicando. GNU ofrece su versión de `make`. SunOs ofrece la suya. Tienen ampliaciones, pero si no son compatibles podemos tener problemas de portabilidad, (al cambiar de una máquina/sistema a otra).

Una solución puede ser instalar make de GNU en (todas?) las máquinas. Por ahí están los fuentes, y no hay que pagar con dinero. Pero quizá sí con tiempo. Si encontramos una versión compilada, pues tan contentos (si no tenemos problemas por tener dos `make` y usar a veces el que no es). Si tenemos que partir de los fuentes y compilarlos, hay que leer con cuidado los `README`, `*doc*`, ... Quizá nos piden que compilemos con el compilador de GNU `gcc` . Y si no tenemos el compilador `gcc` , podemos buscar el binario, con sus librerías, e instalarlo en su sitio, o en otro sitio y usar variables de entorno, o conseguir los fuentes del compilador de `gcc` y compilarlo con otro compilador (tres veces !!!).

La otra vía (solución) es aprender a usar `imake`. `make` carece de condicionales para las variables. `imake` es una combinación ingeniosa de macros de `cpp` que prepara el trabajo de `make` .

18.7.3. Es un error ...

... declarar en el fichero Makefile reglas que indican dependencias circulares. P.ej.:

```
b: a
   cp a b
a: b
   cat b b | head -30 > a
c:
   @ echo regla para c
```

Si hacemos `make a` el sistema responde 'cuidado, `a` depende de si mismo' o algo parecido. Si hacemos `make c` no protesta.

18.7.4. Es arriesgado ...

que un fichero `Makefile` deje el objetivo no actualizado aunque parezca que sí.

```
b: a
   cp a b
d: b
   cat b b | tee a | head -30 > d
```

`a` depende de `b`. Hay una dependencia circular oculta. `make d` no deja el fichero `d` actualizado de forma estable. El fichero `a` queda con fecha de última modificación posterior a la (fecha) del fichero `b` .

18.7.5. Lenguaje de programación

Al escribir el fichero `Makefile` programamos en un lenguaje declarativo. En las reglas escribimos relaciones entre objetivos requisitos y acciones. El orden en que aparecen las reglas en `Makefile` no es importante (excepto la primera, cuando se llama a `make` sin objetivo).

Al no ser importante el orden de las reglas es más fácil escribir y modificar ficheros `Makefile` .

Por ser un lenguaje declarativo se parece (un poco) a prolog.

18.7.6. Orden de actualización

Si tenemos un fichero `Makefile`

```
fsu: fes foe
        cat fes foe > fsu
fes: fne
        grep  A  fne > fes
foe: fno
        grep  B  fno >foe
```

y hacemos `touch fno fne` , el comando `make fsu` tiene que actualizar `fes` y `foe` antes que realizar la acción para actualizar `fsu` .

El orden en que actualiza `fes` y `foe` es indiferente. Puede actualizar ambos fichero en paralelo. Hay versiones de `make` que aprovechan esta posibilidad para conseguir que la ejecución acabe antes. Estas versiones son más útiles en sistemas con varios procesadores.

Hablando técnicamente, `Makefile` establece una relación de orden entre los objetivos. Esta relación de orden es parcial, y al añadir las acciones equivale a un programa concurrente.

18.8. Ejercicios

Ejercicios de examen sobre `make`:
93f.14 (pág. 280), 93j.14 (pág. 286), 93s.14 (pág. 292), 94f.11 (pág. 297), 94j.11 (pág. 303), 94s.13 (pág. 310), 95f.11 (pág. 315), 95j.11 (pág. 321), 95j.12 (pág. 322), 95s.1 (pág. 326), 95s.2 (pág. 326), 96f.6 (pág. 332), 96j.6 (pág. 338), 96s.10 (pág. 345), 96s.11 (pág. 345), 97f.10 (pág. 351), 97j.9 (pág. 357), 97s.10 (pág. 363), 98f.9 (pág. 369), 98j.14 (pág. 376), 98s.12 (pág. 381), 99f.5 (pág. 386).

Capítulo 19

vi

En las primeras clases dábamos una introducción a `vi` . Sólo explicábamos unos pocos comandos, los necesarios para editar, crear, ver y cambiar, ficheros de unas pocas líneas.

Vamos a explicar más comandos del editor `vi` . Empezaremos viendo más comandos para movernos por el texto. Los comandos para moverse son muy importantes porque los movimientos van a combinarse con otros verbos de acción como 'borrar', 'copiar', 'llevar', etc.

19.1. Movimientos

Agruparemos los movimientos por su alcance, y por los objetos en que se basan.

al fondo .. , o a la línea número ..

`1G` nos lleva a la primera línea del fichero.
`G` nos lleva a la última línea del fichero.
número `G` nos lleva a la línea del fichero que hayamos indicado. La 'g' viene de *go*, *ir* en inglés.

19.1.1. una / media pantalla

`^B` mueve el cursor una pantalla hacia atrás. De *backward*, *atrás* en inglés.

Con `^B` queremos representar CONTROL-B, es decir el carácter enviado al ordenador al presionar (manteniendo) la tecla de CONTROL y luego la `b`.

El cursor no se desplaza exactamente una pantalla hacia arriba, sino un poco menos.

`^U` mueve el cursor **media** pantalla hacia atrás. De *up*.
`^D` mueve el cursor **media** pantalla hacia adelante. De *down*.

Si el terminal lo permite, con `^U` y con `^D` las líneas se desplazan suavemente, lo que resulta más agradable a la vista.

`^F` mueve el cursor una pantalla hacia adelante. De *forward*.

una línea

`-` lleva al comienzo de la línea anterior.
`+` y *retorno-de-carro* llevan al comienzo de la línea siguiente.

en la pantalla

Movimientos relativos al texto visible, sin desplazar la pantalla.

`H` lleva el cursor a la primera línea visible en la pantalla. De *high*.
`L` lleva el cursor a la última línea visible en la pantalla. De *low*.

Estos comandos se pueden modificar anteponiendo un número.
`3H` lleva el cursor a la tercera línea de la pantalla.
`2L` lleva el cursor a la penúltima línea de la pantalla.

`M` lleva a una línea situada a la mitad de la pantalla (aproximadamente). De *middle*.

palabras

Empezamos pensando que una palabra es una secuencia de letras. Luego alguien extiende el concepto para que incluya los identificadores de un lenguaje de programación. Puestos a pensar en un lenguaje de programación quienes diseñaron `vi` eligieron C.

palabra es (para `vi`) una secuencia de letras, cifras y caracteres subrayado (`_`). (Palabra pequeña).

En algunas circunstancias interesa extender la palabra.

Llamamos **palabra grande** a una secuencia de caracteres imprimibles consecutivos distintos de: carácter blanco, tabulador y cambio de línea.

`w` lleva al comienzo de la siguiente palabra. 'w' viene de *word*.
`e` lleva al siguiente fin de palabra. De *end*.
`b` lleva hacia atrás al comienzo de una palabra. De *backward*.
`W` , `E` y `B` realizan una función semejante, con *palabras grandes*.

dentro de la línea

`|` lleva a la columna 1 (de la línea actual).
número `|` lleva a la columna indicada por el número.
`0` también lleva a la columna 1 (de la línea actual).
`^` lleva a la primera posición no blanca de la línea.
`$` lleva al final de la línea.

> `$` dentro de una línea indica el fin de la línea. `$` en el ámbito de un fichero indicará la última línea del fichero.

frase, párrafo y sección

Una **frase** (para `vi`) termina en un punto (`.`), signo de admiración (`!`) o de interrogación(`?`) seguido de un fin de línea o de dos caracteres blancos.

> Se hace caso omiso de los caracteres de cierre `)` , `]` , `"` y `'` que aparezcan después del punto, interrogación o exclamación y antes del cambio de línea o los caracteres blancos.

Después de cada línea vacía empieza un **párrafo**.

Si en una línea hay un carácter blanco o un tabulador ya no es una línea vacía, no da comienzo a un párrafo.

> También comienza un **párrafo** con cada conjunto de macros de párrafo, especificadas por los pares de caracteres de la *variable de* `vi` de nombre *paragraphs*. Por ejemplo `.IP` al comienzo de una línea da comienzo a un párrafo. (Es un honor ver a vuecencia leyendo esta letra pequeña. Este párrafo y el siguiente de letra pequeña son más bien historia.)

Cada frontera de párrafo es también una frontera de frase.

Para el editor `vi` existe también el concepto de **sección**.

> Las **secciones** empiezan después de las macros indicadas por los pares de caracteres de la *variable de* `vi` de nombre *sections*.
>
> Si tecleamos `:set sections` en el modo comando del editor `vi` veremos `sections=SHNHH HUnhsh` en la última línea de la pantalla.
>
> `.SH` , `NH` , etc. son referencias a macros de antiguos lenguajes de proceso de texto: *nroff*, *troff*, *mm*, *me*. Todavía se utilizan para dar forma a las páginas de manual.

Cada frontera de sección es también una frontera de frase y párrafo.

`(` lleva al anterior comienzo de frase.
`)` lleva al comienzo de la frase siguiente.
`{` lleva al anterior comienzo de párrafo.
`}` lleva al comienzo del párrafo siguiente.
`[[` lleva al anterior comienzo de sección, o a una llave (`{`) en la columna 1 (lo que encuentre antes).
`]]` lleva al comienzo de la sección siguiente, o a una llave (`{`) en la columna 1 (lo que encuentre antes).

> Es costumbre extendida poner en la columna 1 las llaves que rodean las funciones de C.

Lo más interesante de este apartado es la posibilidad de estructurar un fichero o parte de él en párrafos delimitados por líneas vacías.

movimientos de la pantalla

Los movimiento de esta sección son distintos. Los movimientos que hemos explicado hasta ahora y los que explicaremos en la siguiente sección son movimientos del cursor.

Cuando el cursor llega al extremo superior o inferior de la pantalla obligan a desplazar la pantalla sobre la imagen del fichero. Si en la pantalla se ven las líneas 101..124, el cursor está en la línea 124, y tecleamos `j` , el cursor bajará a la línea 125, y la pantalla pasará a presentar las líneas 102..125 .

A veces interesa desplazar la pantalla sin mover el cursor.

Supongamos que en la pantalla se ven las líneas 101..124 y el cursor está en la línea 124. Queremos borrar la línea 124 dependiendo del contenido de la línea 125 (que de momento no vemos). Una solución puede ser bajar (`j`) una línea para obligar a que aparezca la línea 125, y luego quizá subir (`k`) para borrar (`dd`) la línea 124.

`^Y` desplaza la pantalla una línea hacia arriba, expone una línea por arriba.

`^E` desplaza la pantalla una línea hacia abajo, expone una línea por abajo.

El problema anterior lo solucionamos mejor tecleando `^E` en lugar de `jk` para exponer la línea 125.

búsquedas en la línea

Las búsquedas son también movimientos.

Dentro de una línea buscaremos un carácter, su primera o enésima aparición. Estos movimientos no nos cambiarán de línea. Estas búsquedas no son circulares.

`f` *carácter* avanza el cursor hasta situarlo en la primera aparición hacia la derecha de ese carácter.

`F` *carácter* mueve hacia atrás hasta situar el cursor en la primera aparición hacia la izquierda de ese carácter.

`t` *carácter* avanza hasta el carácter anterior a la primera aparición hacia la derecha de ese carácter.

`T` *carácter* mueve hacia atrás hasta el carácter anterior a la primera aparición hacia la izquierda de ese carácter.

`;` repite la búsqueda.

`,` repite la búsqueda, cambiando el sentido.

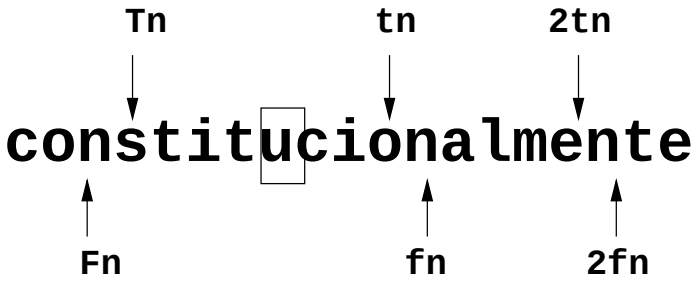


Figura 19.1: El cursor está inicialmente en la `u` .

En la figura anterior vemos distintos movimientos de búsqueda dentro de la línea. Un número previo equivale a realizar *n* movimientos.

búsquedas en todo el fichero

Estas búsquedas son **circulares**. Si estamos en medio de un fichero y buscando hacia adelante llegamos a la última línea del fichero sin encontrar lo indicado, la búsqueda continúa en la primera línea del fichero. En caso de no encontrarse el objetivo, la búsqueda se detendrá al llegar al punto de partida. Igualmente, en una búsqueda hacia atrás, después de la línea primera se buscará en la última línea del fichero.

`/` *expresión regular* busca hacia adelante en el fichero la *expresión regular*.

> En el caso más sencillo las expresiones regulares están formadas por caracteres que se representan a sí mismos. Se puede aprovechar la expresividad de las expresiones regulares para realizar búsquedas sofisticadas. Por ejemplo:
> `/^[^ ]*$` nos permite buscar líneas que no contengan caracteres blancos.

`?` *expresión regular* busca hacia atrás en el fichero la *expresión regular*.

> Después de acabar la expresión regular tenemos que pulsar la tecla *retorno de carro*. Tanto en las búsquedas con `/` como con `?` .

`n` repite la última búsqueda en el fichero en la misma dirección que se indicó con el último eslás (`/`) o interrogación (`?`).
`N` repite la última búsqueda en el fichero en dirección contraria a la que se indicó con el último eslás (`/`) o interrogación (`?`).

Supongamos que estamos editando un fichero que tiene los nombres de los días de la semana, empezando por el lunes, cada uno en una línea. Comenzamos situados en la primera columna de la primera línea y en modo comando. Tecleamos:

1. `/es` *retorno de carro* vamos casi al final de la primera línea (`lunes`),
2. `nnn` vamos casi al final de la cuarta línea (`jueves`),
3. `N` vamos casi al final de la tercera línea (`miercoles`),

4. `N` vamos casi al final de la segunda línea (`martes`),
5. `?o` *retorno de carro* vamos al final de la séptima línea (`domingo`),
6. `n` vamos casi al comienzo de la séptima línea (`domingo`),
7. `n` vamos al final de la sexta línea (`sabado`),
8. `N` vamos casi al comienzo de la séptima línea (`domingo`),
9. `N` vamos al final de la séptima línea (`domingo`).

19.2. Se combinan con los movimientos

`d` *movimiento* borra el objeto abarcado por el movimiento y lo lleva a un búfer (depósito) anónimo. La 'd' viene de *delete*.

> *objeto abarcado por el movimiento*: El cursor antes del movimiento está en un sitio. Después del movimiento va a otro. *Objeto abarcado por el movimiento* es el comprendido entre el punto de partida y el punto de llegada.
>
> Es necesaria una matización. Hay movimientos *orientados a caracteres*, por ejemplo `w`, y *movimientos orientados a líneas*, por ejemplo `j`. Si el movimiento es orientado a líneas, el objeto incluirá desde la línea de partida, hasta la línea de llegada. Si el movimiento es orientado a caracteres, el objeto incluirá desde el carácter de partida, hasta el carácter de llegada.

`y` *movimiento* lleva el objeto abarcado por el movimiento a un búfer (depósito) anónimo. La 'y' viene de *yank*.

`P` trae (una copia de) el contenido del búfer anónimo.

- sobre la línea en que está el cursor,
  si el objeto se compone de líneas enteras, o
- antes del cursor, en caso contrario.

`p` trae (una copia de) el contenido del búfer anónimo.

- bajo la línea en que está el cursor,
  si el objeto se compone de líneas enteras, o

- después del cursor, en caso contrario.

Algunos movimientos: `G` , `^B` , `^U` , `^D` , `^F` , `H` , `M` , `L` , `-` , `+` , `j` , `k` , ... delimitan objetos con líneas enteras. Son movimientos orientados a líneas.

Otros movimientos: `b` , `B` , `e` , `E` , `w` , `W` , `|` , `^` , `$` , `(` , `)` , `{` , `}` , `[[` , `]]` , `F` , `T` , `f` , `t` , `;` , `,` , `/..` , `?..` , `n` , `N` , `h` , `l` , `%` , `'` , ... delimitan objetos con líneas parciales. Son movimientos orientados a caracteres. (Y son la mayoría).

En el búfer queda información de cómo, con un movimiento orientado a líneas o a caracteres, se ha recortado un objeto y se tiene en cuenta al recuperarlo con `P` o con `p` .

mover un objeto se consigue borrándolo (`d` *movimiento1*), yendo al destino (*movimiento2*), y recuperándolo (`p` o `P`).

duplicar un objeto se consigue guardando una copia (`y` *movimiento1*), yendo al destino (*movimiento2*), y recuperándolo (`p` o `P`).

`c` *movimiento* borra el objeto abarcado por el movimiento, lo lleva a un búfer (depósito) anónimo, y queda en modo inserción permitiéndonos cambiar el objeto. La 'c' viene de *change*.

`!` *movimiento comando* filtra el objeto abarcado por el movimiento, a través del comando.

Filtrar un objeto a través de (con) un comando, es poner el objeto como entrada estándar al comando, tomar la salida estándar de esa ejecución y ponerla en el lugar ocupado por el objeto inicial.

Se puede filtrar con un comando sencillo, o una serie de comandos conectados mediante `|` (tubos).

No se modifica el búfer anónimo.

Al teclear `!` no se ve ninguna indicación en pantalla. Al teclear el movimiento aparece el carácter `!` en la última línea de la pantalla. El comando lo vemos en esa última línea. Acabaremos el comando cuando tecleemos *retorno de carro*. Hasta ese momento podemos corregir el comando retrocediendo con el carácter de borrado.

es cómodo ...

`dd` borra una línea (entera). (Ya sabemos que la segunda 'd' no es un movimiento).

`yy` guarda una línea (entera). (Ya sabemos que la segunda 'y' no es un movimiento).

`Y` guarda una línea (entera).

trucos

`xp` intercambia el carácter bajo el cursor y el siguiente.

En una línea tenemos el texto `1234`, el cursor está sobre el `2`, y estamos en modo comando. Tecleamos:

1. `x` se borra el carácter `2`, va al búfer, y el cursor queda sobre el `3`.
2. `p` se recupera el carácter del búfer, en la línea queda `1324` y el cursor queda sobre el `2`.

`ddp` intercambia la línea actual y la siguiente.

Esto funciona porque después de borrar una línea con `dd` el cursor queda en la línea siguiente. (Queda al principio de la línea).

ejemplos

Supongamos que tenemos un fichero de contenido:

```
... marcan:

pedro
antonio
pedro

y acaba el partido.
```

Estamos editando ese fichero y estamos en modo comando, en la línea primera, columna primera. Tecleamos:

1. `}` avanzamos un párrafo. Nos situamos en la línea vacía encima de los nombres.
2. `!}sort` las tres líneas con nombres pasan a ser la entrada estándar del comando `sort`. La salida estándar está formada por esos nombres ordenados alfabéticamente. En el fichero quedan los nombres ordenados alfabéticamente.

3. `u` deshacemos el filtrado y dejamos el fichero como estaba. El cursor nos ha quedado encima de los nombres (línea vacía).

4. `j` el cursor nos queda en la primera línea con nombres.

5. `!}sort |uniq -c` las tres líneas con nombres pasan a ser la entrada estándar del comando `sort |uniq -c` . La salida estándar está formada por

```
1 antonio
2 pedro
```

En el fichero nos queda:

```
... marcan:

 1 antonio
 2 pedro

y acaba el partido.
```

> La entrada estándar del ejemplo anterior eran las tres líneas con nombres y la línea vacía anterior a los nombres. Por eso hemos bajado una línea con `j` .

Supongamos que tenemos un fichero de contenido:

```
... tras muchos cálculos:
12345679 * 18
```

Estamos editando ese fichero y estamos en modo comando, en la línea última, en la columna primera. Tecleamos:

1. `Y` guarda la línea en el búfer,

2. `P` saca arriba una copia de la línea,

3. `!$ bc` la última línea es la entrada estándar del comando `bc` . Este comando `bc` es una calculadora y nos da el resultado 222222222 .

En el fichero nos queda:

```
... tras muchos calculos:
12345679 * 18
222222222
```

Tecleamos `kJi=` *escape* para dejar mejor el fichero.

El mecanismo de filtrar introduce dentro del editor `vi` las posibilidades que tenemos con los comandos de UNIX.

Por si no se ha notado, confirmo: "Al autor le gusta la posibilidad de filtrar que ofrece el editor `vi`".

El ejemplo siguiente se puede considerar algo arriesgado.

Estamos en un directorio de pruebas en el que hay tres ficheros: `f1` , `f2` y `f3` , y de los que podemos prescindir.

Tecleamos:

1. `vi pru1` entramos a editar un fichero que no existía.
2. `i` *escape* creamos una línea vacía que nos resulta cómoda para filtrar.
3. `Omv f1 f4` *escape* en la primera línea editamos un comando del intérprete (`sh` o parecido) para cambiar el nombre de un fichero.
4. `Yp` sacamos una copia y nos situamos en la segunda línea.
5. `!}sh` *retorno de carro* hacemos que el `mv ...` sea la entrada estándar del intérprete de comandos. No hay salida (no se escribe nada por la salida estándar). Hay un efecto: se ha cambiado el nombre de un fichero de `f1` a `f4` .

   ¡ Manejamos el intérprete de comandos desde el editor y tenemos una copia del comando ejecutado!
6. `Orm f?` *escape* en la segunda línea editamos un comando del intérprete (`sh` o parecido) para borrar los ficheros de nombre dos letras y que empiece por `f`.
7. `!}sh` *retorno de carro* hacemos que `rm ...` sea la entrada estándar del intérprete de comandos. No hay salida.
8. `u` Deshacemos lo hecho. Reaparece la línea con `rm ...` .

   ¡No hemos deshecho todo lo hecho!. Los ficheros `f4` , `f2` y `f3` ya no están. `u` deshace los efectos de un comando en lo que respecta al editor `vi`. `u` ignora cómo deshacer el efecto lateral de los comandos que se ha llamado al filtrar.

marcha atrás

`U` deshace los últimos cambios en la línea actual, en la línea en la que está el cursor. Deja la línea como estaba cuando llegamos.

Si vamos a una línea y damos `U` no deshacemos nada.

`.` (punto) repite la última modificación. (modificación, no movimiento).

Por ejemplo, `ddj...` borra la línea actual, salta una y borra tres líneas más.

Un apaño

Estamos editando un fichero. Hemos borrado varios trozos de texto que nos interesaría recuperar. Tecleamos `"1P..` :

1. `"1` citamos el búfer (depósito) de nombre `1` ,
2. `P` traemos lo que hay en el búfer citado. En ese búfer está lo último que hemos borrado.
3. `.` incrementa el nombre del búfer (el `2`). Trae lo que hay en ese búfer, es decir trae lo penúltimo borrado.
4. `.` incrementa el nombre del búfer (el `3`). Trae lo que hay en ese búfer, es decir trae lo tercero borrado (contando hacia atrás).

Este método nos permite recuperar hasta nueve elementos borrados.

En el ejemplo se pone `P` en vez de `p` porque en general deja las cosas más cerca de donde estaban.

Una de las mejoras del editor `vi` , `vim` (*vi-improved*), ha modificado el comportamiento del comando `u` , de forma que en sucesivas invocaciones van deshaciendo las modificaciones.

19.3. Macros

En informática se ha usado frecuentemente el mecanismo que consiste en que un texto se substituye por otro texto siguiendo ciertas reglas. A veces a este mecanismo se le llama *macro*.

En los años 50 y 60 se programaba bastante usando ensamblador, que era una forma de referirse a las instrucciones de la máquina con nombres (y algo más). Ciertas secuencias de instrucciones se repetían,

> y en vez de volverlas a escribir se las empaquetaba y daba nombre y constituían una instrucción grande: *macro-instrucción* (o *macro* para los colegas). Las macros eran substituidas por instrucciones sencillas (bastante) antes de ser ejecutadas.
>
> La substitución de texto también la usan los matemáticos. P. ej. en:
> $$(a+b)^2 = a^2 + 2ab + b^2$$
> $$m + n \to \text{predecesor}(m) + \text{sucesor}(n)$$
> Y creo que (a veces) lo llaman *reescritura*.

En el editor `vi`, las macros primero se definen y luego se usan. Al definir se asocia un valor a un nombre. Definimos una macro escribiendo
`:map` *nombre valor*
El nombre de la macro está limitado a un sólo carácter.

¿Qué caracteres escogemos como nombre de las macros? Hay algunos caracteres sin significado en modo comando, otros caracteres poco usados, como `#`, `q` o `v`. No usaremos como nombre de macro caracteres (casi) imprescindibles como `:` o `d`. El hecho de que las macros de `vi` no puedan ser *recursivas* también limita los posibles nombres. El nombre no puede formar parte del valor de la macro.

`:map q hde$p` define una macro de nombre `q` que retrocede un carácter, borra hasta el fin de la palabra, va al final de la línea y añade la palabra borrada.

Usaremos la macro tecleando su nombre en modo comando. El resultado será el mismo que si hubiésemos tecleado los caracteres del valor de la macro.

Al teclear `:` el cursor baja a la última línea de la pantalla. Podemos escribir y corregir en la última línea con la tecla de borrado hasta que tecleamos retorno de carro, terminando en ese momento la definición de la macro.

A veces, queremos incluir el carácter *retorno de carro* (`^M`) dentro del valor de una macro. Contamos para ello con el carácter `^V` (CONTROL-V) que quita el significado especial al carácter siguiente que se teclee.

Para definir una macro que busque la secuencia de caracteres `pino` tecleamos:
`:map v /pino` *control-V* retorno-de-carro
(sin blancos ni antes ni después de *control-V*). Al teclear `:` el cursor baja a la última línea. El resto lo vemos tal cual en la última línea hasta que tecleamos *control-V* que vemos un carácter `^`. Cuando damos retorno-de-carro aparece una `M`. Tecleamos un segundo retorno de carro para acabar la definición de la macro.

`^V` es el *carácter de escape* del editor `vi`.

Otras veces (?) queremos incluir el carácter *escape* en el valor de una macro. Pues hay que pulsar tres veces `^V` (CONTROL-V) y luego *escape* (una vez).

Cuando estemos definiendo una macro con el carácter *escape* veremos que aparece: primero un carácter `^` , luego una `V` , otro carácter `^` , y por último un carácter `[` .

Para definir una macro que inserte `Hola` al comienzo de la línea actual teclearemos:
`:map # IHola` *control-V control-V control-V escape*
y en la última línea veremos: `:map # IHola^V^[` . Acabamos la definición de la macro con un retorno de carro.

> Sucede que distintas versiones de `vi` permiten sólo una parte de las macros que definimos sobre el papel. El primer ejemplo lo tenemos en el `vi` de ATT que no permite definir como macro `ddp` . Y sobre este aspecto no tengo más documentación que los fuentes de alguna implementación. La situación es la de los sedientos exploradores ante la botella 'medio vacía' o 'medio llena'. Mi propuesta es que aprovechemos las posibilidades que nos ofrecen las macros. Una señal de esperanza es que las últimas versiones soportan la mayoría de las macros que se me ocurren. (¿o será señal de que disminuye la imaginación?.)
>
> También se puede definir macros para utilizarlas en modo inserción. Definimos una macro para modo inserción escribiendo
> `:map!` *nombre valor*

Si las macros dan trabajo y no son imprescindibles, ¿para qué este lío?. Una macro se define una vez (o dos o tres), y se usa muchas veces. Parece que es más útil si editamos varios ficheros en una sesión (de una tirada). Podemos conseguir que el editor `vi` tenga una inicialización en la que se realice automáticamente la definición de varias macros útiles en un entorno de trabajo.

19.4. Opciones y parámetros

Podemos modificar una serie de opciones y parámetros del editor escribiendo:

`:set` *opción*
`:set no`*opción*
`:set` *opción* `=` *valor*

Me parecen útiles las siguientes opciones:

`:set showmode` pone un indicador en la parte inferior derecha de la pantalla para que sepamos si estamos en modo inserción o en modo modificación. Se puede abreviar poniendo `:set sm` .

`:set wm=` *número* permite la escritura continua de texto para personas. Supongamos que estamos en una pantalla de 80 caracteres de anchura, y ponemos un margen de al menos 8 (`.. wm=8`). Cuando al insertar texto, una palabra sobrepase la columna 72 (80-8), se introduce un cambio de línea y esa palabra pasa a la línea siguiente.

`:set number` presenta el número de línea. Estos números de línea no quedan en el fichero, son sólo una ayuda. Pueden molestar si hay líneas largas en el fichero y pasan a ocupar dos líneas de la pantalla. Se puede abreviar poniendo `:set nu` .

`:set all` nos presenta todas las opciones del editor y sus valores.

19.5. Varios

`^G` (CONTROL-G) nos informa del estado del fichero y de la posición del cursor. Una muestra:

```
"cw.tex" [Modified] line 936 of 970 --96%-- col 1
```

`^L` refresca la pantalla.

Si alguien nos manda un mensaje, o si un comando que se está ejecutando desatendido (en *background*) escribe un mensaje por la salida estándar, se nos descabala la pantalla. No coincide la posición del cursor y del texto que estamos editando en la pantalla con la imagen del cursor y de la pantalla accesible al editor `vi`. Trabajar en estas circunstancias puede ser catastrófico.

> Supongamos que estamos editando un fichero y el cursor está en la línea 1, columna 1. A la pantalla llegan 10 caracteres que se insertan en la posición del cursor quedando éste en la columna 11. Tecleamos

`j` . Baja el cursor a la línea 2. En la pantalla vemos el cursor en la columna 11 de la línea 2. El programa `vi` supone que el cursor está sobre el primer carácter de la línea 2. No va a la par lo que vemos con lo que realiza al ordenador. El problema surge si modificamos el fichero , por ejemplo borramos algo (`x` , `dw` , ...), con esta visión errónea.

El problema se resuelve tecleando `^L` (CONTROL-L).

`r` cambia (reemplaza) el carácter bajo el cursor por el carácter que se teclee a continuación. El editor sigue en modo comando. Es más cómodo que borrar, entrar y salir de modo inserción.

`R` comienza una substitución de caracteres. Todo lo que tecleamos se introduce en el fichero y se pierden los caracteres que ocupaban el lugar de los recién tecleados. Si escribimos una línea más larga, los últimos caracteres no eliminan ninguno antiguo. Acaba la substitución con el carácter *escape*.

`~` hace avanzar el cursor y si está sobre una letra la cambia de tamaño, de mayúscula a minúscula o viceversa. Al avanzar sobre cualquier otro carácter lo deja como estaba.

Una serie de `~` convertiría `Gran Via 32` en `gRAN vIA 32` .

`%` mueve el cursor al paréntesis o a la llave asociados.

`(`$_1$ `a + b * (`$_2$ `c - d )`$_3$ `* (`$_4$ `e - f / (`$_5$ `g + h )`$_6$ `)`$_7$ `)`$_8$

Si el cursor está sobre el paréntesis-1 con un `%` irá al 8, y con otro `%` volverá al paréntesis-1. Si el cursor está sobre el paréntesis-4 con un `%` irá al 7, y con otro `%` volverá al paréntesis-4.

Este movimiento viene bien para comprobar que es correcto el anidamiento de paréntesis en expresiones o el anidamiento de llaves en programas de lenguajes como C y 'awk'.

19.6. Números

La mayor parte de los comandos de `vi` modifican su comportamiento si se ha tecleado antes un número. El efecto más frecuente es repetir el comando ese número de veces.

`2j` baja 2 líneas.

`3w` avanza 3 palabras.

`3dw` borra 3 palabras.

`d3w` también borra 3 palabras.

`4fa` avanza a la cuarta `a` .

`f4a` avanza hasta el `4` y queda en modo inserción.
(El emmental[1], Watson).

`5dd` borra cinco líneas.

`6.` (a continuación de borrar 5 líneas) borra 6 líneas (y no 30).

`7i....+....|` *escape* inserta una secuencia de 70 caracteres con una `+` en las posiciones 5, 15, ..., 65 y un `|` en las posiciones 10, 20, ... 70

En algunos casos, como no tenía sentido repetir el movimiento, lo modifica. Por ejemplo: `2H` , `7G` .

Algunos comandos no modifican su comportamiento. Por ejemplo: `7^F` . Se supone que la gente no cuenta pantallas.

> Si alguien usase `vi` a través de un módem de 300 bits/seg. podría ver la diferencia entre `^F` y `7^F` , pero no creo que sea frecuente trabajar con esos módems.

19.7. Búferes (depósitos)

Empezamos hablando de un búfer (depósito) anónimo, donde va todo lo que se borra (con `d`), o guarda (con `y`), o se cambia (con `c`), siempre que no se indique su destino.

Más tarde, en 'apaños', aparecieron 9 búferes (depósitos) más. Su nombre es `1`, `2`, ... , `9` y se corresponden con las nueve últimas cosas (objetos) borradas. Creo que lo que llamamos búfer anónimo y búfer `1` son el mismo búfer.

Pues ahí no acaba la cosa. Hay otros 26 búferes (depósitos) más. Sus nombres son las letras minúsculas: `a`, ..., `z`.

> No han puesto búfer de nombre `ñ` .

`"` *búfer* `y` *movimiento* guarda el objeto abarcado por el movimiento en el búfer indicado a continuación de la doble comilla (`"`).

[1] Queso, variedad de gruyere

" *búfer* p trae el objeto del búfer indicado a continuación de la doble comilla.

Los búferes se pueden usar con d , y , c , p y P .

"adw borra una palabra y la guarda en el búfer a .

"ap trae lo que hay en el búfer a .

19.8. Marcas

m *letra* marca una posición con el nombre dado. Podemos marcar 26 posiciones con las letras minúsculas a, ..., z.

' *letra* va a la posición marcada con la *letra*. Esto (') es una comilla (sencilla).

19.9. Comandos ex

El editor vi tiene también otro nombre: ex . Hay unas hojas de manual con el comportamiento del editor bajo el título vi. Hay información complementaria bajo el título ex. A continuación presento algunos comandos documentados bajo el título ex.

Algunos de estos comandos se aplican sobre una o varias líneas, (de forma parecida al editor no interactivo sed).

Puede indicarse una línea o un rango de líneas:
primera-línea,*última-línea*

Las referencias a líneas pueden ser:

- números,
- . (punto) que es una referencia a la línea actual (del cursor),
- $ que indica la última línea,
- relativas a la línea actual, como .-2 , o .+4 , o
- relativas a la última línea, como $-2 .

`:`*rango* `s/`*expresión-regular-1*`/`*expresión-regular-2*`/g`
cambia en el *rango* todas (`g`) las apariciones de la *expresión-regular-1* por la *expresión-regular-2*.

Sin la `g` sólo cambiaría la primera aparición en todas las líneas del rango.

> Esto difiere del comportamiento de `sed`. El editor no interactivo `sed` cuando no se pone `g` cambia la primera aparición en cada línea.

`:1,$s/can/perro/g` substituye todas las apariciones en el fichero de `can` por `perro` .

`:.+1,$-1 s/\([0-9]A\)/\1B/g`
Substituye (`s`) desde la línea siguiente a la actual (`.+1`) , hasta la penúltima (`$-1)` , todas las apariciones (`g`) de una cifra seguida de una `A` (`[0-9]A`), por esa misma cifra (`\( ... \) ... \1`) seguida de una `B`. (Ya sé que esto es complejo, pero a todo (lo bueno :-) se acostumbra uno. Si tienes dudas repasa el capítulo 13 que trata de las *expresiones regulares*.)

`:g/`*expresión-regular*`/d` borra (`d`) todas (`g`) las líneas que se ajusten a la *expresión-regular*.

`:`*rango* `m` *línea* mueve las líneas indicadas por *rango* debajo de la línea indicada al final.

`:2,3 m 4` aplicado en un fichero con los días de la semana, dejaría las líneas `martes` y `miercoles`, debajo de la línea `jueves`.

`:`*línea* lleva el cursor al comienzo de la línea indicada.

19.10. Ficheros

escritura

Cuando se empieza a editar un fichero, se lleva una copia del contenido del fichero a la memoria de un proceso cuyo código es el programa `vi`. Cuando salimos del editor con `:wq` se escribe la versión del fichero en memoria del proceso `vi` al sistema de ficheros. Podemos decir que entonces se escribe en el disco. (No hay por qué entrar en más detalles hoy).

`:w` escribe el contenido de la memoria en el fichero de partida, y seguimos editando.

`:w` *nombre de fichero* escribe el contenido de la memoria en el fichero cuyo nombre se indica. (Y seguimos editando).

Alguna versión de `vi` no deja escribir con `:w` *nombre de fichero* cuando el fichero existe. En ese caso insistiremos con:
`:w!` *nombre de fichero*

`5,.w trozo` un rango (`r,.`) antes de `w` indica que sólo se escriban esas líneas.

lectura

`:r` *nombre de fichero* trae debajo de la línea del cursor el contenido del fichero indicado.

`:r` trae debajo de la línea del cursor el contenido del fichero que se está editando.

edición de múltiples ficheros

Al comenzar la sesión de edición podemos nombrar varios ficheros, por ejemplo: `vi f2 f4 f7` . El editor comenzará con el primero nombrado.

`:n` pasa a editar el siguiente fichero. Si hemos modificado el fichero actual, se supone que hemos salvado las modificaciones, por ejemplo con `:w` .

Si hemos modificado el fichero actual, no hemos salvado las modificaciones y tecleamos `:n` , el editor se niega a pasar al siguiente fichero. Insistimos con: `:n!`

ejecución de comandos

En caso necesario, el editor `vi` crea un nuevo proceso mediante *fork*, y *exec* y le encarga que ejecute el código de un comando de UNIX tal como `date`, `sort`, o incluso `sh` o `tcsh`. De hecho, esto ya a sido necesario para filtrar objetos. También se puede usar de otras formas.

`:r !`*comando* trae debajo de la línea del cursor la salida estándar del comando nombrado.

`:r !date` trae la fecha y hora debajo de la línea del cursor.

`:!` *comando* ejecuta el comando nombrado, presenta su salida y espera a que tecleemos un retorno de carro para volver al texto que estamos editando.

`:!sh` inicia una sesión con el intérprete de comandos `sh` . Es un caso particular del anterior. Se puede poner (obviamente) cualquier intérprete disponible en nuestro sistema. Acabaremos con `exit` o `logout`. Tendremos que dar también un retorno de carro para volver al modo comando.

19.11. Inicialización

`vi` , igual que muchos otros programas, admite que se le de una configuración inicial. Para inicializar `vi` hay dos pasos. En primer lugar se mira si:

1. existe el fichero `./.exrc` , o

   `elvis` y `vim` no consideran `./.exrc` .

2. está definida la *variable de entorno* de nombre `EXINIT` , o

3. existe el fichero `$HOME/.exrc` .

Se toma el contenido (fichero) o valor (variable) del primero de éstos que se encuentre y lo acepta como comandos que teclease el usuario.

Después ejecutará lo que pongamos en la línea de invocación del `vi` detrás de la opción (flag) `-c` .

ejemplos

Un contenido típico de la variable `EXINIT` o de un fichero `.exrc` puede ser: `set wm=8 nu showmode` .

Dependiendo del intérprete de comandos cambia la forma de asignar valor a una variable de entorno.

1. En el caso de `sh` o `bash` escribiremos:
   ```
   export EXINIT='set ... mode'
   ```

2. En el caso de `csh` o `tcsh` escribiremos:
   ```
   setenv EXINIT 'set ... mode'
   ```

Se suele asignar valor a la variable `EXINIT` en el fichero de inicialización del intérprete de comandos.

intérprete de comandos	fichero de inicialización
`sh`	`.profile`
`csh`	`.cshrc`
`tcsh`	`.tcshrc`
	`.cshrc`
`bash`	`.bash_profile`

Podemos ver la inicialización para un usuario con `csh` o `tcsh`

```
$ grep EXINIT $HOME/.cshrc
setenv EXINIT 'set wm=8 nu showmode'
```

`vi -c ':$' f11` comienza la edición del fichero con el cursor en la última línea.

19.12. Ejercicios

Ejercicios sobre el editor `vi` : 93f.12 (pág. 280), 93f.13 (pág. 280), 93f.17 (pág. 281), 93j.12 (pág. 286), 93j.13 (pág. 286), 93j.17 (pág. 287), 93s.12 (pág. 292), 93s.13 (pág. 292), 93s.17 (pág. 293), 94f.13 (pág. 298), 94f.14 (pág. 298), 94f.17 (pág. 299), 94j.13 (pág. 304), 94j.14 (pág. 304), 94j.17 (pág. 305), 94s.14 (pág. 310), 94s.15 (pág. 310), 94s.18 (pág. 311), 95f.13 (pág. 316), 95f.17 (pág. 317), 95j.13 (pág. 322), 95j.14 (pág. 322), 95s.13 (pág. 328), 95s.14 (pág. 328), 95s.15 (pág. 328), 96f.13 (pág. 334), 96f.14 (pág. 334), 96j.13 (pág. 339), 96j.14 (pág. 340), 96j.18 (pág. 341), 96s.13 (pág. 345), 96s.14 (pág. 346), 96s.18 (pág. 347), 97f.12 (pág. 352), 97f.13 (pág. 352), 97j.12 (pág. 358), 97j.13 (pág. 358), 97j.17 (pág. 359), 97s.12 (pág. 363), 97s.17 (pág. 365), 98f.11 (pág. 369), 98f.17 (pág. 371), 98j.13 (pág. 376), 98s.13 (pág. 381), 98s.17 (pág. 383), 99f.13 (pág. 388), 99f.17 (pág. 389).

Capítulo 20

Más comandos

20.1. `cal`

- sin parámetros, imprime el calendario del mes actual.
- con un parámetro, imprime el calendario del año que se indica.
- con dos parámetros, imprime el calendario del mes y año que se indica.

```
$ cal 12 2001
   December 2001
 S  M Tu  W Th  F  S
                   1
 2  3  4  5  6  7  8
 9 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28 29
30 31
```

> El calendario escrito por `cal` tiene el domingo en la primera columna, los meses y días de la semana están en inglés, y corresponde al calendario gregoriano. No sirve para fechas anteriores a 1582 en que se produjo el cambio de calendario juliano a gregoriano. Hay más información en *History of Astronomy: Items: Calendars, Time and Chronology*
> `http://www.astro.uni-bonn.de/~pbrosche/.../hist_astr/ha_items_calendar.html` Wolfgang R. Dick.

20.2. bc

`bc` nos ofrece una calculadora básica. De ahí viene su nombre. Se puede usar interactivamente. Recibe operandos y operaciones por la entrada estándar. Además de las operaciones suma, resta, multiplicación, cociente, resto y elevar a una potencia, permite usar variables (de nombre una letra).

```
% bc
a=987654321
a^9
894220687831492019250297979565814868578\
5
966650988309889495771664151944421814072\
81
scale=30
3/19
.157894736842105263157894736842
15*15
225
obase=8
.
341
^D
%
```

Como ejemplo asignamos a la `a` un valor de nueve cifras, elevamos ese valor de la `a` a la novena potencia y obtenemos (en este caso) un resultado de 82 cifras con total precisión. La precisión de la calculadora `bc` no está limitada (a un número pequeño de cifras). Cuando el resultado no cabe holgadamente en una línea, lo presenta en dos o más líneas.

Después dividimos 3 entre 19 con 30 (`scale`) cifras decimales. Vemos que las cifras se repiten :-) .

Más tarde multiplicamos 15 por 15.

El comando `bc` nos permite cambiar la base de entrada y/o la base de salida (una sola vez).

Indicamos el cambio de base de salida (output) mediante `obase` . (Para cambiar la base de entrada pondríamos `ibase=..`)

El punto representa para `bc` el último resultado obtenido.

Si escribimos `.` obtenemos la representación en octal de 225.

`bc` también se puede usar como filtro.

```
% echo '5 * 4' | bc
20
% vi ....
!}bc
```

ejercicios

Ejercicios sobre el comando `bc` : 99f.11 (pág. 387).

20.3. spell

`spell` sirve para presentarnos una lista de palabras probablemente mal deletreadas (en inglés).

Podíamos haber construido este comando basándonos en `partepals` (`tr -cs \'A-Za-z' '\012'`) que parte un texto en palabras.

Si disponemos de una colección suficientemente amplia de textos base (100 libros?, 1000 informes?) podemos obtener su vocabulario (`cat ...|partepals|sort -u`), y luego eliminar los nombres de variables, y otros conjuntos de caracteres sin sentido.

Si queremos analizar un documento, obtenemos su vocabulario igualmente y lo comparamos (`comm -23 - vocRef`) con el vocabulario de referencia obtenido anteriormente. Las palabras que aparezcan podrán ser lugares geográficos, términos comerciales o quizá palabras mal deletreadas.

Realmente el programa `spell` está escrito en C. Así es más rápido. Se apoya en un fichero de más de 60000 términos. Puede construir plurales, y formas verbales ???? .

Un programa equivalente a `spell` en castellano sería más complicado por la mayor complejidad de los verbos con todos sus tiempos y personas, y por los prefijos y sufijos que existen en este lenguaje.

```
% echo one two tree four cinco see seven eleven | spell
cinco
```

Existen otras utilidades de análisis de texto. Alguna de ellas mide cómo de 'pesado' es un texto y se basa en una fórmula lineal $c_f l_f + c_p l_p$ basada en la longitud media de las frases en palabras l_f y la longitud media de las palabras en sílabas l_p . (c_f y c_p son dos constantes).

20.4. units

`units` convierte unas unidades a otras. Se apoya en el fichero `/usr/lib/units` (o `unittab`).

El comando `units` está diseñado para que se use de forma interactiva. Se le pueden dar unidades sencillas:

```
$ units
you have: inch
you want: cm
        * 2.540000e+00
        / 3.937008e-01
...
```

Tenemos pulgadas y queremos centímetros y nos da las constantes de conversión directa e inversa. En este primer ejemplo escribe unos datos que tiene en el fichero `units`.

> Vemos que los datos los expresa en notación científica. Por ejemplo, `e-01` nos indica que el número `3.937008` hay que multiplicarlo por 10^{-1}. La `e` viene de exponente.

$$3{,}937008e-01 = 3{,}937008 * 10^{-1} = 0{,}3937008 = \frac{1}{2{,}54}$$

```
...
you have: 6 feet
you want: meter
        * 1.828800e+00
        / 5.468066e-01
...
```

Podemos decirle que tenemos 6 pies y que queremos metros. Tiene que hacer una operación sobre los datos de `/usr/lib/units`.

```
...
you have: $
you want: peseta
        * 6.993007e+01
        / 1.430000e-02
...
```

Podemos decirle que tenemos dólares y que queremos pesetas, y nos da el valor de cambio cuando se escribió el fichero de datos (1980 (!)).

```
grep peseta /usr/lib/units
spainpeseta             .0143 $
peseta                  spainpeseta
```

Si cambiamos la primera línea que aparece en la búsqueda con `grep` por

```
euro                    1.147 $
spainpeseta             .006010121 euro
```

tendremos un valor un poco más actual (1999-01-28).

```
...
you have: 1000 peseta / meter
you want: $ / feet
        * 4.358640e+00
        / 2.294294e-01
...
```

Por último damos un valor 'precio por metro', y pedimos el correspondiente en dólares cada pie. Para obtener el resultado utiliza varias líneas del fichero de datos.

```
...
you have: 1000 peseta / meter
you want: $ / pound
conformability
        1.451000e-02 dollar/m
        5.818351e-01
```

El programa `units` analiza las dimensiones y rechaza conversiones imposibles.

20.5. Compresión, consideraciones

Buscamos algoritmos que compriman la información. Estos algoritmos pueden ser

no reversibles: no existe una operación inversa para obtener la información original.

Podemos usarlos para comprimir la información que hay en un fichero que representa una foto, música o vídeo. Comprimimos una foto, la descomprimimos, y podemos ver la misma foto, ya que nuestro ojo/cerebro tolera que algunos pixels cambien de valor sin que tengamos problemas en reconocer un paisaje o a una persona.

Suelen conseguir que los ficheros comprimidos ocupen menos que con algoritmos reversibles.

En general, son específicos para el tipo de datos representados, porque tienen que conocer qué pérdida de información es tolerable y cuál no.

reversibles: existe una operación inversa para obtener la información original.

Los usaremos con ficheros que representen fuentes de programas, programas compilados, protocolos notariales, etc. No nos conformaremos con que el fichero descomprimido con los fuentes de un programa se parezca al original, queremos que sea *idéntico*. Ni siquiera admitimos que nos quiten una línea en blanco.

Dado el enfoque de la asignatura nos centraremos en los algoritmos reversibles.

No existe un algoritmo de compresión reversible que siempre consiga que el fichero procesado ocupe menos que el fichero original.

Si tuviéramos un algoritmo que comprimiese todos los ficheros bastaría aplicarlo un número finito veces para obtener una versión comprimida de tamaño cero (!). Esta aplicación no tendría inversa.

> Alguien argumenta que es capaz de comprimir todos los ficheros del párrafo anterior añadiendo el contenido del fichero al nombre y dejando el fichero con tamaño cero.
>
> Matizamos: “No existe un algoritmo de compresión reversible que siempre consiga que el fichero procesado (contenido y nombre) ocupe menos que el fichero original (contenido y nombre)”.

Podemos plantear también la dificultad ante un directorio con ficheros que no sabemos si están comprimidos (y cuántas veces) o no. (Esta situación me recuerda a la descrita por Stanislaw Lem en *Memorias encontradas en una bañera* cuando un agente secreto tras desencriptar un mensaje con una máquina, duda si la salida debe volver a introducirla en la máquina, ...).

En la vida diaria, la realidad no es tan mala [Descartes].

- Los ficheros comprimidos acostumbran a usar extensiones específicas `.z`, `.gz`, y llevar una marca o contraseña en los primeros octetos (llamada número mágico). Resulta muy improbable tomar por comprimido un fichero que no lo está.
- La mayor parte de los ficheros que usamos admiten ser comprimidos mediante algoritmos sencillos.

Los criterios de calidad de un algoritmo de compresión son:

1. Menor tamaño del fichero comprimido.
2. Menor tiempo de descompresión.
3. Menor tiempo de compresión.
4. Menores requisitos de memoria.

Suponemos que la descompresión de ficheros es más frecuente que la compresión. Igualmente suponemos que la lectura es más frecuente que la escritura. (Lo contrario nos llevaría a un modelo de mundo incomunicado (más?)).

Compresión, comandos en UNIX

Los ficheros comprimidos mantienen la fecha de última modificación (y otros datos del i-nodo) del fichero original, para que al descomprimir la situación sea lo más parecida posible a la de partida.

20.5.1. `pack`

`pack` es el más antiguo de los programas de compresión de uso frecuente en UNIX (UNIX III de ATT). Codifica los octetos del fichero con un código Huffman.

`pack f1 f2` substituye (si conviene) los ficheros `f1` y `f2` por sus versiones comprimidas, con nombre `f1.z` y `f2.z` .

`unpack f1.z f2.z` recupera los ficheros originales.

20.5.2. `compress`

`compress` ofrece la misma funcionalidad que `pack` .

`compress f1 f2`

`compress` tiene un esquema de compresión más eficaz que `pack` . Los ficheros comprimidos por `compress` añaden a su nombre `.Z` . Para descomprimir se usa la opción `-d` .

`compress -d f1.Z f2.Z`

`compress` se ha usado durante cierto tiempo (198x) como si fuese software público, hasta que Unisys ha reclamado la propiedad del algoritmo en que se basa, y ha manifestado su intención de ejercer su derecho sobre la utilización para comprimir, dejando libre la posibilidad de descomprimir.

20.5.3. `gzip`

`gzip` surge para ofrecer la posibilidad de comprimir usando software libre o *copyleft*. Se basa en codificación Lempel-Ziv. Para comprimir y descomprimir los ficheros de los ejemplos anteriores haríamos

```
gzip f1 f2

gunzip f1.gz f2.gz
```

Los ficheros comprimidos añaden al nombre `.gz` .

`gzip` es más eficaz que los anteriores.

20.5.4. `bzip2`

Usa el algoritmo de Burrows-Wheeler. Los ficheros comprimidos añaden al nombre `.bz2` . Su uso es:

```
bzip2 f1 f2                    bunzip2 f1.bz2 f2.bz2
```

`bzip2` es el que más comprime de los cuatro. En algunas circunstancias puede resultar lento.

20.5.5. `pcat, zcat, gzcat, bzcat`

A veces interesa ver el contenido de un fichero comprimido enviando los caracteres resultantes a la salida estándar.

Nombres:

- Un fichero comprimido con `pack` (`xx.z`) se descomprime y envía a la salida estándar mediante `pcat` .
- Un fichero comprimido con `compress` (`xx.Z`) se descomprime y envía a la salida estándar mediante `zcat` .
- Un fichero comprimido con `gzip` (`xx.gz`) se descomprime y envía a la salida estándar **según** en el sistema
  - **existe** software de `compress` mediante `gzcat` .
  - **no existe** software de `compress` mediante `zcat` .
- Un fichero comprimido con `bzip2` (`xx.bz2`) se descomprime y envía a la salida estándar mediante `bzcat` .

Queremos consultar un servidor de ftp, por ejemplo ftp.funet.fi . En este servidor, todas las noches hacen algo así como

```
ls -lR-a | tee ls-lR | gzip >ls-lR.gz
```

con lo que tienen un índice de todo lo que hay en el servidor y lo guardan en versión comprimida (`ls-lR.gz`) y descomprimida (`ls-lR`). Supongamos que la versión comprimida ocupa 4 Megas y la versión descomprimida 40 Megas.

Nos traemos a nuestra máquina el fichero `ls-lR.gz` . Queremos ver dónde aparece el fichero `adjtime*` . Podemos hacer:

```
gunzip ls-lR.gz
grep adjtime
gzip ls-lR
```

o

```
zcat ls-lR.gz | grep adjtime
```

La segunda forma es más rápida, y no ocupa temporalmente 40 Megas de disco.

(Nota: la función del último ejemplo la realiza el comando `zgrep` . Casi siempre hay otro comando ...).

20.6. `btoa`

En algún momento, un transporte de correo sólo aceptaba ficheros con caracteres imprimibles y líneas de longitud máxima 80 caracteres. (Usaba código EBCDIC e imágenes de tarjeta; sí, tarjeta perforada, como las del censo americano de comienzos de siglo XX; y conversión desde ASCII y a ASCII en los extremos). Para enviar ficheros con caracteres no imprimibles, se busca una representación de los mismos que sólo use caracteres imprimibles.

> Se nos puede ocurrir substituir cada octeto por su representación decimal usando siempre tres dígitos. El fichero ocupará el triple.
>
> Podemos substituir cada octeto por su representación hexadecimal. El fichero ocupará el doble.
>
> Podemos tomar los octetos del fichero de partida, por cada tres octetos formar cuatro grupos de 6 bits, y asociar a cada combinación de 6 bits (hay $2^6 = 64$ combinaciones distintas) un carácter imprimible. El fichero resultante ocupará un 33 % más que el original.
>
> Podemos tomar los octetos del fichero de partida, considerar que cada 4 octetos representan un número, que puesto en base 89 se puede representar con 5 caracteres imprimibles. Se consideran caracteres imprimibles del ASCII-32 al ASCII-126 (además del retorno de carro y quizá algún otro) por lo que hay suficientes. El fichero resultante ocupará un 25 % más que el original.

`btoa` convierte de binario a ascii (e imprimible). Hace algún tiempo se usaba una versión de `btoa` basada en el paso de 3 a 4 octetos. La versión actual pasa de 4 a 5 octetos.

Si `btoa` tiene un parámetro tomará como entrada el fichero nombrado, y si no tiene parámetros convertirá la entrada estándar.

Además `btoa` forma líneas de 80 caracteres, añade una suma de redundancia para detectar errores, y guarda el nombre del fichero original si lo conoce.

`btoa` deja la información en la salida estándar.

`btoa -a` realiza la función inversa a `btoa` . También se puede llamar a `atob` (ascii to binary) que es otro nombre (enlace) del mismo comando.

20.7. tar

El comando `tar` maneja objetos en *formato tar*. El formato tar es una forma pensada para agrupar y guardar (*archive*) uno o varios ficheros en cinta magnética (*tape*).

> Los dispositivos de cinta magnética son bastante antiguos. Se desarrollaron inicialmente para grabar sonido. Desde los años 195x represetaron el soporte más económico de almacenamiento masivo de información. (Esto de económico es relativo al precio del resto del sistema y dispositivos alternativos). Tenían 7, 8 ó 9 pistas, y se grababa la información a 200, 800, 1600 bits por pulgada. La cinta viajaba a 20 m/seg. (72 km/hora), y se detenía en un cm. Para aprovechar la cinta y poder acceder a parte de la información, ésta se agrupaba en bloques. Una cifra típica del tamaño de bloque era 2048 octetos.

Con el formato *tar* queremos llevar uno o varios ficheros a cinta y más tarde recuperar esos ficheros en (quizá otro) disco con el mismo aspecto para el usuario.

En formato *tar*, por cada fichero se escribe el nombre (posiblemente cualificado con algunos directorios) de ese fichero, la información contenida en el i-nodo para ese fichero (toda, salvo los números de los bloques en que está el fichero), y la información interna de ese fichero, seguida de caracteres 0-ascii hasta situarse en posición múltiplo de 2048 octetos.

`tar` se usa con una de las tres opciones siguientes:

c para *crear* objetos de tipo *tar* .

t para *examinar* el contenido (nombres, tamaños, .., de los ficheros) en objetos de tipo *tar*

x para *extraer* todos o parte de los ficheros contenidos en objetos de tipo *tar* .

Estas opciones no van precedidas del carácter `-` , habitual en otros comandos de UNIX.

`tar cf /dev/rmtp f1 f3` escribe en la cinta colocada en el dispositivo `/dev/rmtp` los ficheros `f1` y `f3` .

La opción `f` indica que a continuación se nombra el destino (`/dev/rmtp`) del objeto de formato tar. Esta opción `f` aparece en un 99 % de los casos.

En UNIX es costumbre que los dispositivos físicos figuren como ficheros (de tipo carácter o bloque) bajo el directorio `/dev` . `rm` viene de removible, y `tp` de cinta (tape).

Suponemos que hay un dispositivo de cinta (armario le llaman) etiquetado `rmtp` , en el cual se ha colocado un carrete con la cinta, y en esa cinta se graba una imagen de `f1` y `f3` en formato tar.

`tar cvf cin.dd dd` con todos los ficheros contenidos debajo del directorio `dd` crea el fichero `cin.dd` en formato 'tar'.

Aunque `cin.dd` probablemente esté en un disco duro, contiene una secuencia de octetos, idéntica a la que se graba en una cinta magnética.

Si debajo de `dd` tuviésemos cuatro ficheros de 20 octetos, `cin.dd` podría ocupar 8k octetos o más debido al relleno de caracteres 0-ascii para cada fichero.

La opción `v` viene de *verbose* (prolijo, charlatán). Al crear un objeto tar, escribe el nombre de los objetos añadidos. Al extraer ficheros o examinar un objeto tar escribe una línea por cada fichero semejante a la que escribe `ls -l` .

`tar tvf cin.dd` lista el contenido del fichero `cin.dd` . Su formato debe ser 'tar' .

`tar xvf cin.dd` extrae todos los ficheros contenidos en `cin.dd` . Su formato debe ser 'tar' . Crea directorios cuando sea necesario.

> Con el formato tar se guardan los ficheros con su nombre, incluyendo los directorios hasta llegar al directorio indicado (`dd` en este caso). No se ha guardado el i-nodo de los directorios que cualifican a los nombres de los ficheros. Cuando más tarde recupere la información del objeto tar puedo perder algo de información: la información asociada a los directorios, su dueño, grupo, fechas de última actualización, etc. Si no existen los directorios que contienen a los ficheros del objeto tar, se crean con unos valores, en particular se pone como dueño de un directorio el dueño del primer fichero que obliga a su creación.

`tar xvf cin.dd dd/f1` extrae `dd/f1` contenido en `cin.dd` .

Al extraer podemos poner a continuación unos ficheros o subdirectorios de lo salvado y se extrae solamente los ficheros nombrados o los ficheros contenidos en los subdirectorios nombrados.

`tar cvf cin.ee -T flista ee` incluye los ficheros y directorios indicados (uno por línea) en `flista` , y luego el fichero (o directorio) `ee` .

Cuando la lista de ficheros o directorios a salvar creando un objeto tipo tar es muy larga, tendríamos una línea difícil de teclear sin errores, y podemos incluir dicha lista en un fichero (`flista` en el ejemplo), al que luego hacemos referencia mediante la opción `-T` .

El uso de `-T` no impide (ni obliga) añadir algunos ficheros o directorios más en la línea de comandos de `tar` (`ee` en el ejemplo).

Alberto quiere enviar a Bárbara todos los ficheros bajo un directorio por correo electrónico a través de unos sistemas que no transportan ficheros no imprimibles, ni mayores de 100k .

Alberto teclea:

```
$ tar cvf - dd | gzip | btoa | split
```

`tar cvf - dd` crea un objeto tar a partir de los ficheros bajo el directorio `dd` y lo envía a la salida estándar (`c.f -`).

`gzip` comprime la información. Las secuencias de caracteres 0-ascii ocupan la mayor parte del espacio en objetos tipo 'tar' cuando los ficheros son pequeños, y si hay ficheros con contenido semejante, esto también favorece una mayor compresión. No es raro reducir el tamaño de objetos *tar* a la cuarta, o incluso décima parte.

`btoa` convierte el flujo de caracteres comprimidos y no imprimibles a caracteres imprimibles.

`split` trocea la salida de `btoa` en ficheros de 1000 líneas o menos.

Después envía a Bárbara esos ficheros (`xaa` , `xab` , etc).

Supongamos que Bárbara recibe los mensajes, les quita la cabecera y los llama `xaa` , `xab` , ..., etc.

Luego Bárbara teclea:

```
$ cat xa? |btoa -a| gunzip |tar xvf -
```

`cat xa?` realiza la operación inversa de `split` .

`btoa -a` realiza la operación inversa de `btoa` .

`gunzip` realiza la operación inversa de `gzip` .

`tar xvf -` extrae del objeto tar que le llega por la entrada estándar (`x.f -`) los ficheros, creando los directorios necesarios.

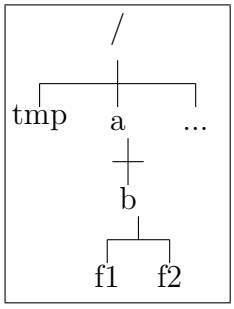
Es costumbre que el nombre de los ficheros tipo tar acabe en `.tar` y si guardan los ficheros que estaban bajo un directorio, su nombre suele ser el del directorio acabado en `.tar` .

El nombre de los objetos tar comprimidos con `gzip` a veces acaba en `.tar.gz` o en `.tgz` . Igualmente se usa `.taz` y `.taZ` para los comprimidos con `pack` o `compress` .

El comando `tar` que viene con Linux admite las opciones `z` y `y` . `tar cz..` crea objetos tar comprimidos con `gzip` . `tar xz..` extrae los ficheros de objetos tar comprimidos. `tar cj..` crea objetos tar comprimidos con `bzip2` . `tar xj..` extrae los ficheros de objetos tar comprimidos.

`tar`, tres escenarios y una regla

Tenemos el siguiente árbol en el sistema de ficheros:



Vamos a realizar tres operaciones:

1. Salvar los ficheros `f1` y `f2` en formato tar, en el fichero `/tmp/cin` .
2. Borrar `f1` y `f2` con `rm /a/b/f[12]` .
3. Recuperar los ficheros `f1` , y `f2` del fichero `/tmp/cin` .

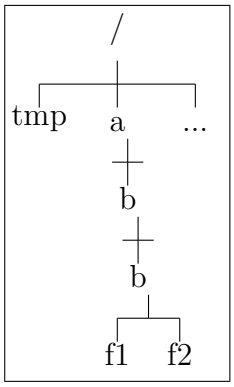
Esto, que parece tan sencillo, admite varias formas de hacerlo. A cada forma le llamamos escenario.

esc. A
```
cd /a/b ; tar cf /tmp/cin f1 f2
rm /a/b/f[12]
cd /a/b ; tar xf /tmp/cin
```

El sistema de ficheros queda como estaba originalmente.

esc. B
```
cd /a   ; tar cf /tmp/cin b/f1 b/f2
rm /a/b/f[12]
cd /a/b ; tar xf /tmp/cin
```

El sistema de ficheros nos queda:



esc. C
```
cd /a   ; tar cf -P /tmp/cin /a/b/f1 /a/b/f2
rm /a/b/f[12]
cd /a/b ; tar xf /tmp/cin
```

El sistema de ficheros queda como estaba originalmente.

Moraleja :

- Si los objetos a guardar con formato tar se nombran con el camino absoluto (desde `/`), y se indica `-P`, al recuperarlos irán al lugar en que estaban originalmente.

  Esto es rígido, y alguna vez nos puede resultar útil si queremos asegurar dónde quedarán.

- Si los objetos a guardar con formato tar no se nombran con el camino absoluto, para que queden en el mismo lugar en que estaban originalmente, debemos ejecutar el comando `tar x..` en la misma posición en que hicimos `tar c..` .

  Lo más frecuente y flexible es no nombrar los ficheros con el camino absoluto. Esto nos deja más libertad. En el último momento decidimos donde dejamos los ficheros.

ejercicios

Ejercicios sobre el comando `tar` : 95j.10 (pág. 321), 95s.7 (pág. 327), 96j.12 (pág. 339), 97f.14 (pág. 352), 97s.11 (pág. 363), 98j.8 (pág. 375).

20.8. `find`

`find` busca objetos (ficheros o directorios) bajo un directorio dado.

Se le invoca dando el directorio bajo el que se busca y una secuencia de expresiones. En principio, las expresiones se evalúan secuencialmente con lógica *y* (*and*) de izquierda a derecha. Cuando una expresión toma valor *falso* no se evalúan las siguientes.

Con expresiones simples podemos poner condiciones al:

- nombre:
  `-name nm` su nombre es `nm` .

  Para el nombre podemos utilizar los metacaracteres de `sh` : `*` , `?` , `[` , `-` y `]` . En caso de usarlos los incluiremos entre comillas o los prefijaremos con un `\` para que `sh` no les dé significado especial.

- número de enlaces:
  `-links 2` tiene 2 enlaces.

  Cuando en las expresiones utilicemos números, podrá ponerse también +*número* o -*número* para indicar >*número* o <*número* respectivamente.

- fecha de última modificación:
  `-mtime -7` modificado hace menos de 7 días.

  La unidad es días.

- fecha de último acceso:
  `-atime +7` accedido hace más de 7 días.

- tamaño del fichero o directorio:
  `-size +40` tamaño mayor que 40 bloques (de 512 octetos).

  La unidad es el bloque de 512 octetos. Es un valor independiente de la unidad de asignación física de disco a los ficheros.

- fecha de última modificación relativa a la de otro objeto:
  `-newer f1` ha sido modificado después que `f1` .

  Por ejemplo, instalamos un software nuevo, creamos un fichero al acabar la instalación: `nuevaConf` . Más tarde podemos buscar ficheros modificados o creados después de la instalación.

- tipo del objeto:
  `-type _` su tipo es :

  - `f` fichero normal.
  - `d` directorio.
  - `c` fichero especial caracteres. ...

- número de i-nodo:
  `-inum 1022` su número de nodo es 1022 .

  El número de nodo es único para todos los objetos contenidos en la misma partición (o disco virtual).

- –
  `-print` evalúa siempre cierto.

  Escribe el nombre en la salida estándar.

- –
  `-ls` evalúa siempre cierto.

  Escribe la información que da `ls -l` .

- resultado tras un comando:
  `-exec com` cierto si `com` devuelve 0 .

  Los comandos de UNIX devuelven un valor que indica si ha habido o no novedad. A la llamada al sistema *exit* se le llama con un parámetro y ese parámetro es el valor de terminación. Sin novedad, 0, se asocia con 'bien', 'true'.

Hay expresiones con efecto lateral, y expresiones sin efecto lateral. Las tres últimas (`-print` , `-ls` , y `-exec`) tienen efecto lateral, las otras nombradas no tienen efecto lateral.

La forma más frecuente de uso del comando `find` es con una serie de expresiones sin efecto lateral, seguidas de una única expresión con efecto lateral. Si por ejemplo, ponemos:

`find .` *expresión1 expresión2* `-print`

para cada fichero o directorio situado bajo el directorio actual (`.`),

- se evalúa la *expresión1*. Si resulta falsa, se pasa al siguiente fichero o directorio.
- Si resulta cierta, se evalúa la *expresión2*. Si resulta falsa se pasa al siguiente fichero o directorio.
- Si han sido ciertas las dos primeras expresiones, se imprime el nombre del objeto (`-print`) .

Si se cambia el orden, por ejemplo:

`find . -print` *expresión1 expresión2*

se imprime el nombre de todos los ficheros y directorios, y luego se evalúa *expresión1*, y a veces *expresión2*.

Moraleja: las expresiones con efecto lateral no conmutan (no se puede cambiar su orden).

Las expresiones sin efecto lateral conmutan, se puede cambiar su orden. Puede haber un pequeño aumento o disminución de velocidad de ejecución. Por ejemplo, no importa cambiar a `find ...` *expresión2 expresión1* .

Tenemos mecanismos para construir expresiones más complejas:

- `!` : negación.
- `-a` : y (and)
  expresión1 `-a` *expresión2*
- `-o` : o (or)
  expresión1 `-o` *expresión2*
- `( )` : agrupamiento
  `(` *expresión* `)`

Como `(` y `)` son caracteres especiales para `sh` , los prefijaremos con un `\` para que `sh` no les de significado especial.

`find . -name set.mod -print`
busca bajo el directorio actual (`.`) objetos de nombre `set.mod` . Imprime los nombres con los directorios que conducen hasta ellos desde el directorio actual.

`find lib -name '*.def' -print`
busca bajo el directorio `lib` objetos de nombre acabado en `.def` e imprime sus nombres.

`find .. -print`
imprime todos los nombres de los objetos bajo el directorio padre del directorio actual.

```
find . \( -name a.out -o -name '*.o' \) -atime +7 -exec rm {} \;
```

Estamos buscando:

- debajo del directorio actual (`.`), todos los objetos
- cuyo nombre sea `a.out` o acabe en `.o` ,

  > Estos nombres son propios de resultados de compilación en lenguaje C. El primero para ejecutables a los que no se da nombre específico. El segundo, de resultados intermedios, antes de enlazar.

- a los que haga más de 7 días que no se ha accedido (ni lectura ni escritura), (`- atime +7`)
- y se borran (`-exec rm {} \;`).

La sintaxis para la expresión `exec` es singular:

- Acaba con `;` debido a que la longitud de un comando en principio no la conoce `find` . Como `;` es carácter especial para `sh` , hay que anteponer `\` para que el punto-y-coma llegue al `find` .
- `{}` indica el lugar a ocupar por el nombre del objeto.

  En otro ejemplo podíamos poner `-exec mv {} {}.v \;` para cambiar un montón de nombres de fichero añadiéndoles `.v` .

20.8.1. Ejercicios

Ejercicios sobre el comando `find` : 95s.18 (pág. 329), 96j.15 (pág. 340), 96s.15 (pág. 346), 97s.7 (pág. 363), 98f.12 (pág. 370), 98s.14 (pág. 382), 99f.8 (pág. 387).

20.9. du

`du` nos da la ocupación de disco.

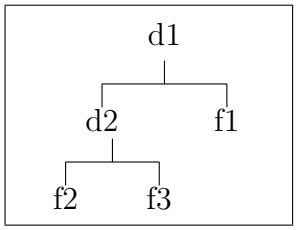
`du dir` escribe los bloques usados por `dir` y sus subdirectorios.

`du -a dir` escribe los bloques usados por `dir` y cada fichero situado bajo `dir` y bajo los subdirectorios de `dir` . (`-a` viene de *all*, todos)

`du -s dir` escribe el total de bloques ocupados por los ficheros y directorios bajo `dir` . (`-s` viene de *summary*, resumen)

Si no se indica el directorio, se toma por omisión el directorio actual.

Supongamos que tenemos los siguientes ficheros y directorios:




```
% wc -c d1/f1 d1/d2/f?
   4159 d1/f1
     29 d1/d2/f2
    145 d1/d2/f3
```

```
% du -a d1
5       d1/f1
1       d1/d2/f2
1       d1/d2/f3
3       d1/d2
9       d1
```

```
% du d1
3       d1/d2
9       d1
```

```
% du -s d1
9       d1
```

Vemos que el directorio `d1/d2` , sin los ficheros `f2` y `f3` , ocupa un bloque $(3 - (1 + 1))$. Es el espacio ocupado por los nombres de los ficheros, los números de los i-nodos, y quizá huecos de (nombres de) ficheros que hubo antes.

Los bloques en que `du` mide la ocupación son bloques de asignación para la información interna del fichero. Su tamaño depende del sistema y son valores frecuentes 512, 1024, 2048 y 4096 bytes.

Para conocer el tamaño del bloque de asignación podemos fijarnos en un fichero de unos pocos miles de octetos de tamaño. En el ejemplo anterior valdría `f1` . Vemos que para un tamaño de 4158 octetos, utiliza 5 bloques. Podemos suponer razonablemente que el tamaño del bloque es 1024 octetos.

Alguna vez puede fallarnos el método anterior.

```
% ls -l
-rw-r--r--  1 a0007    10981808 Jan 13 16:35 core
-rw-r--r--  1 a0007        1823 Jan 13 16:36 saco
% du -a
2585    core
2       saco
```

En el ejemplo anterior tenemos un fichero (`core`) en el que el último octeto está en la posición 10.981.808, pero en el que sólo se ha escrito en 2.585 bloques distintos. Es un fichero con 'huecos'. Este es un caso infrecuente. (El tamaño del bloque en este sistema es 1024 octetos.)

20.10. df

`df` informa del espacio libre en particiones (discos virtuales). Su nombre viene de *disk free*.

```
% df
Filesystem kbytes   used  avail capa Mounted on
/dev/sd6a   14882  10418   2975  78% /
/dev/sd6g  256786 200350  30757  87% /usr
/dev/sd4g  299621 244769  24889  91% /eui
/dev/sd0g  299621  36833 232825  14% /eui/alum
```

- La primera columna indica el nombre del dispositivo físico (partición o disco) que soporta el sistema de ficheros.

  El primer dispositivo tiene por nombre `/dev/sd6a` . Está en un disco con una controladora SCSI, con *número scsi* 6, y es la primera partición (`a`) .

- La segunda columna indica el total de espacio disponible para los ficheros en esa partición o disco.
- La tercera, los bloques usados.
- La cuarta, los bloques disponibles.
- La quinta es el tanto por ciento ocupado sobre el total.

  Fijándonos en la línea de `/dev/sd6g` vemos que

  $$usados + disponibles \neq total$$

  $$200350 + 30757 = 231107 \neq 256786$$

  El sistema no deja a los usuarios ocupar todo el disco; considera que el espacio útil es el 90 % del total.

  Si el sistema se usase hasta acercarse al 100 %, los últimos ficheros creados tendrían los bloques muy dispersos por el disco, y el acceso secuencial a estos ficheros sería muy lento.

  El administrador (`root`) puede llegar a usar todo el disco, lo cual le da oportunidad, por ejemplo, de comprimir un fichero. Esto sería imposible si se le pusiese las limitaciones de los demás usuarios. Al comprimir un fichero, temporalmente se ocupa más disco (coexisten las versiones comprimida y sin comprimir).

- La sexta columna es el directorio en que se ha montado ese dispositivo.

  Los ficheros cuyo nombre comience por:

  - `/eui/alum` están en el dispositivo `/dev/sd0g` ,
  - `/usr` están en el dispositivo `/dev/sd6g` ,
  - `/eui` están en el dispositivo `/dev/sd4g` ,
  - y todos los demás están en el dispositivo `/dev/sd6a`

  Si envían muchos mensajes de correo electrónico puede suceder que se llene la partición `/dev/sd6a` , ya que el correo se almacena en `/var/spool/mail` [1]. Veríamos:

[1] en esta máquina `/usr/spool` es un enlace simbólico a `/var/spool` .

```
% df
Filesystem kbytes   used  avail capa Mounted on
.....
/dev/sd6a   14882  13393      0 100% /
.....
```

20.11. Variables de entorno

Motivación

Escribimos un programa editor `mivi` . Para editar un fichero `f1` en un terminal de tipo *vt52* escribimos:

```
mivi -term vt52 f1
```

Escribimos un programa para ver ficheros página a página `mimas` . Para hojear el fichero `f3` en el terminal anterior escribimos:

```
mimas -term vt52 f3
```

Si durante una sesión usamos varias veces los programas `mivi` y `mimas` nos conviene ahorrar la opción `-term vt52` porque durante la sesión no cambia el tipo de terminal.

Alternativas

- Podemos hacer que los programas `mivi` y `mimas` lean un fichero , por ejemplo `$HOME/.term` , cuyo contenido se fija al comienzo de cada sesión.
- Podemos hacer que estos programas reciban el valor del *tipo de terminal* a través de las **variables de entorno**.

Qué es

- Es una lista de pares nombre-valor asociada con (en la memoria de usuario de) cada proceso. El orden no se considera importante.
- Los valores son del tipo *tira de caracteres*. Un posible valor es la tira de caracteres vacía.

- Cuando el código de un proceso hace *fork* se crea un proceso idéntico (salvo el valor de la cima de la pila), y por lo tanto con variables de entorno idénticas e idénticos valores.
- Es muy frecuente que cuando el código de un proceso haga *exec* se le pasen las mismas variables de entorno con los mismos valores.

Ventajas

- Se accede a la información más rápido que leyendo un fichero.
- El tipo *tira de caracteres* es aceptable para la mayoría de los lenguajes.

Un caso concreto

Durante el arranque de UNIX, el proceso `init` en cierto momento lee el fichero `/etc/gtty` en el que se dice a cada línea qué tipo de terminal hay conectado, por ejemplo:

```
# name  getty                       type     status
console "/usr/etc/getty cons8"      sun      on local secure
ttya    "/usr/etc/getty std.9600"   vt100    on local
```

Por cada línea para conexión de usuarios `init` crea un proceso mediante *fork*, el proceso creado hace *exec*. Para la línea `ttya` el proceso creado hará *exec* con la variable de entorno `TERM=vt100` y código `getty`. Este proceso (hijo de `init`) hace a su vez *exec*, y llega a ser el intérprete de comandos de un usuario. El intérprete de comandos de un usuario trabajando en el teminal conectado a la línea `ttya` tendrá la variable de entorno `TERM=vt100`.

Cuando el usuario hace `vi f` el intérprete de comandos hace *fork*, y su proceso hijo hace *exec* con código de `vi` con lo que el programa editor `vi` tiene acceso a la variable de entorno `TERM` que le informa del tipo de terminal que se usa.

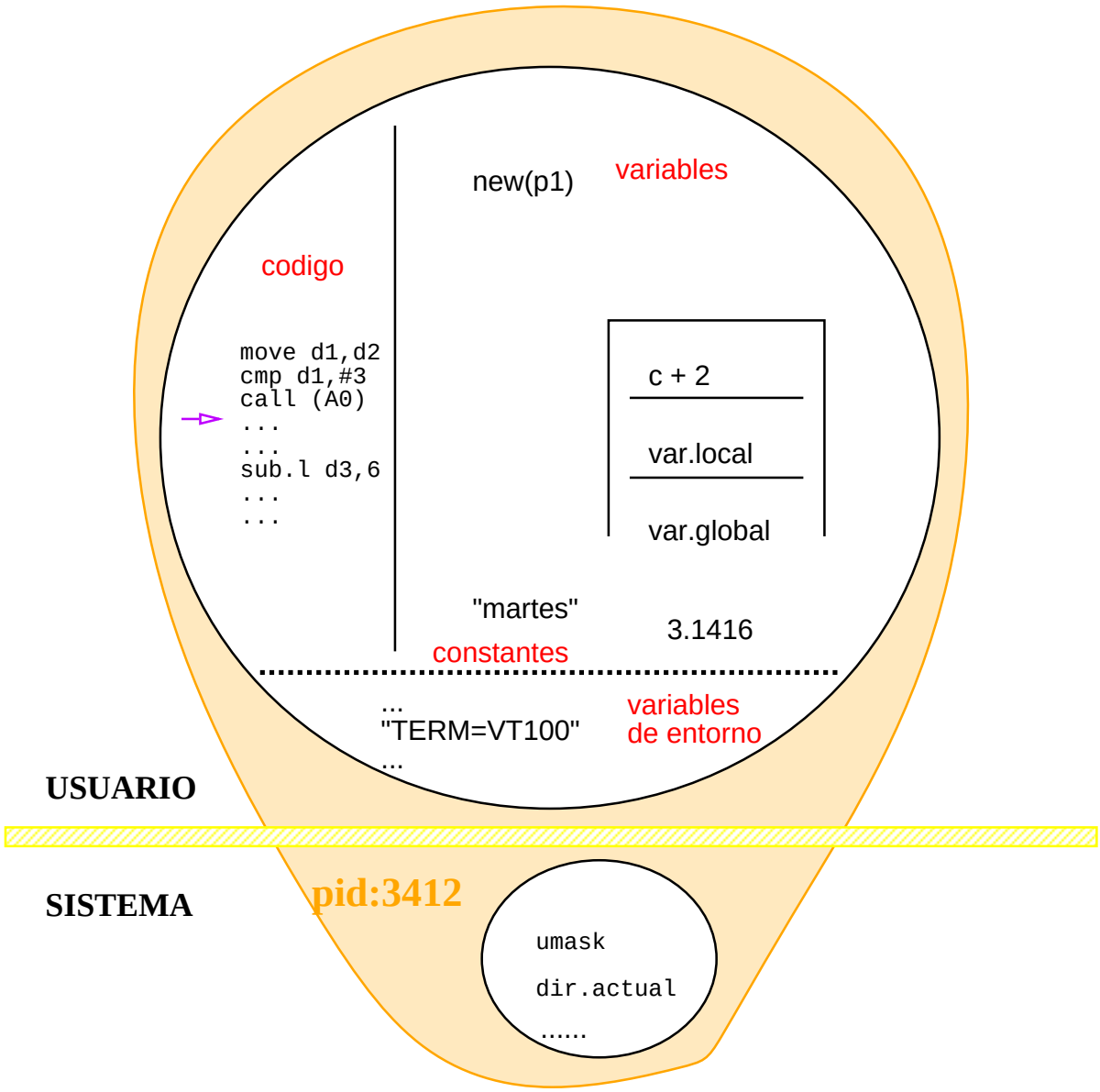


Figura 20.1: Memoria de un proceso en UNIX

Capítulo 21

sh

21.1. Variables

`sh` , como muchos otros intérpretes, admite variables. A estas variables se asigna un valor poniendo el nombre de la variable, el signo igual, y el valor que queremos que tome la variable.

Los valores de las variables son de tipo tira de caracteres.

`hoy="Marzo18"` asigna el valor `Marzo18` a la variable `hoy` .

> Los caracteres `"` `"` sólo son necesarios si queremos que el valor asignado tenga blancos, tabuladores, u otro carácter especial para `sh` . En este ejemplo no eran necesarios.
>
> El signo igual debe seguir inmediatamente a la variable, para que el nombre de la variable no se confunda con el nombre de un comando. También se impone que el valor a asignar a la variable siga inmediatamente al signo igual.

El nombre de las *variables sh* puede formarse con una secuencia de letras, cifras y subrayados (`_`). El tamaño de las letras (mayúsculas o minúsculas) hace distintas a las variables.

Obtenemos el valor de una variable poniendo `$` , y el nombre de la variable.

Se puede usar `{ }` para indicar dónde acaba el nombre de una variable:

- `$hoy3` se substituye por el valor de la variable `hoy3` .

- `${hoy}3` se substituye por el valor de la variable `hoy` seguido de un carácter `3` .

> Las *variables de entorno* son algo propio de todos los procesos en sistemas tipo UNIX. `sh` , al igual que muchos otros intérpretes, tiene variables. Las llamaremos *variables sh*. Son conceptos distintos aunque están relacionados.

Al comenzar a ejecutarse, `sh` lee las *variables de entorno*, crea *variables sh* del mismo nombre y valor, y les pone marca de exportar. Cuando hay que asignar un valor a una variable, si no existe se crea sin marca de exportar. Mediante el comando

`export` *nombreDeVariable*

se puede poner marca de exportar a una variable. Cuando `sh` hace `exec` para ejecutar un comando, pone como *variables de entorno* una copia de aquellas de sus *variables sh* que tienen marca de exportar.

`sh` incluye una serie de variables: `HOME` , `PATH` , ...

`$HOME` : es el argumento por omisión del comando `cd` .

Cuando escribimos programas de `sh` y queremos usar un fichero poniendo el camino desde `/` , conviene expresar su nombre en términos de `HOME` . Si el administrador cambia la posición del directorio de entrada, y mueve todo nuestro árbol de directorios y ficheros a otro lugar, el programa seguirá funcionando sin ningún cambio. Es (trans)portable.

En lugar de

```
..... >> /home/cajal/a0007/tfc/log
```

es mejor poner

```
..... >> $HOME/tfc/log
```

`$PATH` es la lista de directorios en los que se buscan los comandos. Los directorios están separados por el carácter `:` . Cuando escribamos un comando, se buscará en primer lugar en el primer directorio. Si existe el fichero, y es ejecutable, se ejecutará. En caso contrario, se buscará en el segundo directorio, ... Si no lo encuentra en el último directorio, `sh` escribirá 'comando no encontrado'.

Si la lista comienza por el carácter `:` , esto equivale a buscar en primer lugar en el directorio actual. Hay un punto implícito. Es como si la lista comenzase por `.:/usr1/alum..` .

```
$ echo $PATH
:/usr1/alumn/o0123/bin:/usr/bin
```

`sh` busca los comandos en `.` , luego en `/usr1/alumn/o0123/bin` , y por último en `/usr/bin` .

> No parece buena costumbre que el directorio actual (`.`) esté al comienzo de `$PATH` , antes que los directorios estándar de comandos. Podría suceder que fuésemos al directorio de otro usuario, `cd ~vecino` , e hiciésemos `ls` para ver qué hay. Si en ese directorio hubiese un fichero de nombre `ls` y con permiso de ejecución, no se ejecutaría el `ls` situado en `/usr/bin` . Se ejecutaría el programa (fichero) de(l) `vecino` . Este programa, podría leer, escribir o borrar nuestros ficheros sin nuestro permiso.
>
> Podemos evitar esto tecleando siempre el camino largo del comando. Por ejemplo, escribir `/usr/bin/ls` en vez de `ls` .

```
$ echo $PATH
/usr1/alumn/o0123/bin:/usr/bin:.
```

Parece mejor poner el directorio actual al final de `$PATH` .

```
$ echo $PATH
/usr1/alumn/o0123/bin:/usr/bin
```

En este caso `sh` no busca los comandos en el directorio actual (`.`) . `$PATH` no contiene explícita ni implícitamente el directorio actual.

> `csh` y `tcsh` , al comenzar a ejecutarse crean un índice (con una *tabla hash*) con el contenido de esos directorios. Toman todos los nombres de ficheros ejecutables que se encuentran en los directorios de `$PATH` , excluyendo el directorio actual.
>
> Puede suceder que un usuario (*o0123*) trabaje en su directorio de ejecutables `/usr1/alumn/o0123/bin` y ponga a punto el programa `reves` . Prueba el programa y funciona. Va a otro directorio e intenta usar el programa `reves` . El intérprete de comandos le responde 'comando no encontrado'. No lo encuentra porque `reves` no está en su índice (*tabla hash*). El usuario debe teclear `rehash` para que el intérprete actualice su índice.

‘ ‘

Un texto entre comillas inversas (‘ ‘), se ejecuta y se substituye por la salida estándar generada al ejecutar ese comando. Los *cambios de línea* se cambian por blancos.

```
$ date +%m%d
0112
$ who > foto.‘date +%m%d‘
```

`date` puede escribir la fecha en diferentes formatos. Mediante `+` se indica que el formato no es el estándar. `%m` indica que queremos el mes en formato numérico y con dos cifras. `%d` indica que queremos el día del mes con dos cifras. En las hojas de manual de `date` se documentan otras opciones de formato de fecha. No es lo mismo que `printf` . Ejecutada la primera línea el 12 de Enero, nos escribe `0112` .

Cuando `sh` explora toda la línea y encuentra `who ... ‘ .... ‘` .

- Primero ejecuta el comando entre comillas inversas (‘ ‘)
- Substituye el comando y las comillas inversas por lo obtenido en la salida estándar.
- Luego `sh` encuentra `who >foto.0112` y lo ejecuta.

Así, podemos preparar un programa de `sh` que cada día vaya grabando la información en un fichero distinto.

```
d=‘pwd‘
cd /usr1/alumn/terce/b0321
...
cd $d
```

En medio de un programa de `sh` queremos ir a un directorio, hacer un trabajo, y volver donde estábamos.

Guardamos en una variable el nombre del directorio donde estamos: `d=‘pwd‘` . Vamos al directorio que queríamos: `cd /usr1/.../b0321` . Hacemos el trabajo en ese directorio, y volvemos: `cd $d` .

Una forma de conseguir algo parecido sería:

```
...
(cd /usr1/alumn/terce/b0321; ......)
...
```

pero de esta forma no podríamos usar el directorio `$d` mientras estuviésemos en `.../b0231` . También está la pega de que se crea un proceso más.

Secuencias $

`sh` suministra una serie de valores útiles mediante unas *secuencias dólar*.

- `$$` da el identificador (número) de proceso en que se ejecuta `sh` .

  Una aplicación de `$$` es crear nombres de ficheros únicos.

  Supongamos que queremos escribir un programa que se pueda ejecutar concurrentemente, y que ese programa utiliza un fichero temporal. Por ejemplo, un comando redirige la salida estándar a un fichero temporal, y más tarde se borra.

  Si dos usuarios ejecutan el programa simultáneamente, o un usuario lanza dos ejecuciones simultáneas de este programa, puede suceder que una instancia del programa borre el fichero que la otra instancia acaba de crear, o que las dos instancias escriban simultáneamente sobre el mismo fichero, o que una instancia lea (casi) a la vez que otra está escribiendo. Todas estas posibilidades se deben evitar.

  Para no tener problemas basta con utilizar ficheros distintos. Si se usan directorios distintos es sencillo. Si se usa el mismo directorio debe ser distinto el nombre del fichero. Esto se consigue con una secuencia de caracteres que incluya `$$` . Los sistemas tipo UNIX garantizan en ningún momento hay dos procesos con el mismo identificador de proceso.

```
ftemp=/tmp/nombreCom.$$
...
...  >$ftemp
...
rm $ftemp
```

- `$#` da el número de parámetros con que se invoca al programa `sh` .

  Podemos comprobar que se invoca al programa con el número de parámetros adecuado, o modificar ligeramente el comportamiento del programa en función del número de parámetros.

- `$?` da el valor devuelto por el último comando ejecutado síncronamente.

- `$0` da el nombre con que se ha invocado al comando.

  Podemos escribir un programa de `sh` y darle dos nombres con `ln` . Podemos hacer que el comportamiento del programa cambie ligeramente dependiendo del nombre con el que se le llama.

  También es cómodo `$0` para informar del nombre del programa a qué se debe el error. Por ejemplo:

```
echo $0 : error , no se puede leer /etc/hosts
```

- `$*` se substituye por la lista de parámetros.

operadores Y, O : `&&` `||`

Los procesos de UNIX devuelven un valor que indica si ha habido o no novedad. A la llamada al sistema *exit* se le llama con un parámetro (0..255) y ese parámetro es el valor de terminación en la mayor parte de los casos. Si los procesos acaban por una señal, por ejemplo por un `kill` , el valor es uno mayor, relacionado con el número de la señal. Sin novedad, 0, se asocia con 'bien', 'true', 'cierto'.

El valor **0** se reserva para: ejecución con éxito, situación más frecuente, etc. .

> Los conceptos de *situación más frecuente* y *éxito*, hablando de comandos en UNIX, no son absolutos en el caso general. Es una decisión del implementador del comando, en qué casos acaba la ejecución con *exit(0)* o *exit(6)*. Más precisamente, es una decisión de los primeros implementadores, ya que parece normal que a partir de cierta difusión de un comando no cambien los criterios.
>
> Esperamos que en comandos como `cp` , `ln` , `mv` , y `rm` , el valor devuelto se ajuste al éxito de la operación indicada. `grep` devuelve 0

si encuentra la tira buscada, devuelve 1 si no la encuentra, y devuelve 2 si no puede abrir el/los fichero(s), o la sintaxis de la expresión regular es incorrecta. `cmp` , y `diff` devuelven 0 si los ficheros son idénticos.

Como norma general, conviene mirar el manual, en la sección "DIAGNOSTICS", al final. Creo que no conviene trabajar sólo con el resultado de una prueba (`$?`).

`sh` tiene los operadores:

- `&&` **y** lógico (secuencial),
- `||` **o** lógico (secuencial).

Con estos operadores se evalúa de izquierda a derecha sólo los operandos necesarios para conocer el resultado (evaluación perezosa).

- Con el **y** lógico
  - si el primer operando resulta cierto, se evalúa el segundo operando.
  - si el primer operando resulta falso, no se evalúa el segundo operando.
- Con el **o** lógico
  - si el primer operando resulta cierto, no se evalúa el segundo operando.
  - si el primer operando resulta falso, se evalúa el segundo operando.

Tenemos ejecución condicional del segundo operando (comando) dependiendo del resultado del primer operando (comando).

```
ln b z || ln c z
```

Si falla el primer `ln` , intenta el segundo. La línea habrá tenido éxito si `z` se ha enlazado a `b` o a `c` .

```
cp e f && rm e
```

Si falla el `cp` , no borra `e` . Si `cp` no falla, borra `e` .

> Alguno se dará cuenta que la línea anterior es muy parecida a `mv e f` .
> Si quiere otro ejemplo puede cambiar `cp` por `rcp` .

Una serie de comandos unidos por `|` devuelve como resultado el de su último comando.

> Parece sensato que en un trabajo 'en cadena', si el último proceso acaba normalmente, la ejecución de la cadena se considere normal. Si en una aplicación esta regla no nos conviene, podemos reescribir la 'cadena', por ejemplo usando ficheros temporales.

21.2. `sh` . Comandos compuestos

Vamos a presentar cuatro estructuras o comandos compuestos de `sh` : `for` , `if` , `while` y `case` .

```
for variable in listaPalabras
do
   listaComandos
done
```

```
if expresiónBooleana
then
   listaComandos
else
   listaComandos
fi
```

```
while expresiónBooleana
do
   listaComandos
done
```

```
case expresiónTira in
   patrón ) listaComandos ;;
   ...
esac
```

21.2.1. Sintaxis

Para que las palabras `for` , `do` , `done` , `if` , `else` , `fi` , `while` , `case` , `esac` sean reconocidas como *palabras reservadas* sólo pueden ir precedidas de (cero o más) caracteres blanco y tabulador. La palabra `in` toma carácter de palabra reservada después de `for` o `case`.

Es conveniente desplazar unos (pocos) caracteres a la derecha el texto de los comandos (*listaComandos*) afectados por un comando compuesto. El nombre correcto de este desplazamiento es *sangrado* y no 'indentado'. Este último término es una mala traducción del término inglés *indent*.

El punto y coma (`;`) hace el papel de cambio de línea. Podemos poner el primer comando de *listaComandos* a continuación de `do` , `then` o `else` sin poner un `;` . Es más, a continuación de `do` , `then` o `else` no podemos poner un `;` para continuar con el primer comando de *listaComandos*.

En pocas líneas, la sentencia `for` puede escribirse:
`for` *variable* `in` *listaPalabras* `; do` *listaComandos* `; done`

21.2.2. Semántica

• `for` va seguido de una variable, la partícula `in` y una secuencia de palabras (piezas o tokens). Este `for` repetirá la lista de comandos tantas veces como palabras encuentre a continuación de `in` . En cada iteración la variable tomará como valor cada una de las palabras que sigue a `in` , y en el mismo orden en que se nombran.

> Este `for` , a diferencia del comando del lenguaje pascal del mismo nombre, no tiene un incremento automático. Para recorrer diez valores consecutivos tenemos que nombrar los diez valores. Tenemos a cambio posibilidad de poner esos valores en cualquier orden arbitrario. Por ejemplo, podemos poner: `for j in 1 3 2 4` .
>
> Muchas veces, los valores de `for` son nombres de fichero o parte del nombre de ficheros. Por ejemplo:
> `for k in 2 3 5 ; do pr cap$k |lpr ; done`

Se puede suprimir `in $*` al final de la primera línea de `for` . Cuando sólo tenemos `for` *variable* , se supone (se añade) `in $*` .

```
for i in $*
do
  echo $i
  mv $i $i.v
done
```

`$*` se substituye por todos los parámetros. Suponemos que los parámetros son nombres de fichero. Con cada parámetro (`for i in $*`), escribimos el nombre del parámetro/fichero (`echo $i`) y cambiamos el nombre del fichero añadiéndole `.v` (`mv $i $i.v`).

- `if` va seguido de una expresión booleana, tiene una rama `then` , una rama `else` y está terminado por `fi` (la palabra del comienzo al revés).

Si la expresión booleana es cierta, se ejecuta la rama `then` . En caso contrario, se ejecuta la rama `else` .

Una versión reducida de `if` carece de rama `else` .

```
if ln b z || ln c z
then
  :
else
  echo no se pudo crear el enlace z
fi
```

Intentamos que `z` sea el segundo nombre de `b` o de `c` . En caso de que no podamos, ambos `ln` han fracasado, escribiremos un mensaje (`echo no se pudo crear el enlace z`). En caso de que se haya creado, no se hace nada. ‘ `:` ’ hace el papel de comando nulo.

- `while` va seguido de una expresión booleana, la partícula `do` y una lista de comandos.

`while` evalúa la expresión booleana. Si el resultado es falso, acaba el comando compuesto `while` . En caso contrario ejecuta la lista de comandos y vuelve a la situación inicial.

```
while true
do
  sleep 10
  echo tic tac
done
```

En el ejemplo anterior, la expresión booleana es el comando `true` . Este comando siempre devuelve 0 (cierto). Este ciclo está escrito para que no acabe. (El programa no considera la posibilidad de que corten la corriente eléctrica). Se espera 10 segundos (`sleep 10`) y se escribe `tic tac` (`echo ...`). Este reloj atrasa (como otros que meten ruido :-) .

- `case` va seguido de una expresión de tipo tira de caracteres (valor discriminante), la partícula `in` y varias alternativas. Cada alternativa comienza con una lista de patrones, separados por comas (`,`), un paréntesis de cierre `)` , seguida de una lista de comandos, y acaba con dos puntos

y coma seguidos (;;). Los patrones están construidos con caracteres normales, y los metacaracteres * ? [-] que tienen el mismo significado que cuando se comparan con nombres de fichero (? equivale a cualquier carácter, etc.).

La expresión que sigue a case se compara consecutivamente con los patrones de cada alternativa. Cuando se produce un ajuste entre la expresión y un patrón se ejecuta la secuencia de comandos de esa alternativa.

> A diferencia de la sentencia *case* del lenguaje pascal, en el comando case de sh puede haber solapamiento entre los valores que se ajustan a los patrones de las alternativas. Por ello el orden de las alternativas es significativo, importante.

```
case $resp in
  SI)  rm $i ;;
  NO)  exit 1 ;;
  *)   echo 'Responder "SI" o "NO"' ;;
esac
```

- Si el valor de la variable resp es SI , se borra el fichero de nombre el valor de i (SI) rm $i ;;).
- Si el valor de la variable resp es NO , acaba el programa con resultado 1 (NO) exit 1 ;;).
- En cualquier otro caso se escribe Responder ...
  (*) echo 'Responder "SI" o "NO"' ;;).

El asterisco situado como *patrón* de la última alternativa equivale a una cláusula *ELSE* (en cualquier otro caso) de otros lenguajes.

Si la alternativa *) .. se pusiese en segundo lugar, nunca se ejecutaría la tercera alternativa.

21.2.3. break, exit

El comando break sale de la estructura for o while más interior de las que le incluyen.

Normalmente estará dentro de una estructura if .

```
while .... ; do
  ...
  if ...... ; then ..... ; break ; fi
  ...
done
```

`exit` acaba el programa de `sh` y si lo tecleamos trabajando en modo interactivo acaba la sesión.

`exit` *número* , acaba igualmente un programa de `sh` . El proceso que ejecuta `sh` devuelve como resultado *número*. El valor por omisión es 0 .

21.3. `shift`

`shift` (desplazar) quita el primer parámetro y renumera. Así se puede acceder a más de 9 parámetros.

Tenemos un programa de `sh` de nombre `ponap` . Su contenido es:

```
ap=$1 ;  shift
for i  ; do
  mv $i $i.$ap
done
```

En la primera línea guardamos en la variable `ap` el valor del primer parámetro. Después desplazamos, perdemos lo que había sido primer parámetro y se renumeran todos. Luego iteramos, repetimos. Como no ponemos para qué valores (`for i`) es para todos los parámetros (`for i in $*`) que quedan. Se ha perdido el primer parámetro al haber hecho `shift` . Para cada parámetro se cambia (`mv`) `$i` por `$i.$ap` . Supongamos que escribimos:

```
ponap .v f1 f2
```

Inicialmente con `$1` hacemos referencia a `.v` , con `$2` hacemos referencia a `f1` , y con `$3` a `f2` .

A la variable `ap` le asignamos `.v` , y al hacer shift `$1` representa a `f1` y `$2` representa a `f2` .

	.v	f1	f2
antes de shift	$1	$2	$3
después de shift		$1	$2

Al iterar con `for` invocamos dos veces el comando `mv` , y cambiamos el nombre de `f1` por `f1..v` , y el nombre de `f2` por `f2..v` .

> Es posible que quisiésemos obtener `f1.v` y no `f1..v` . Tenemos dos posibilidades: o cambiamos el uso del programa `ponap v f1 f2` , o cambiamos el programa `mv $i $i$ap` . Tomaremos la segunda opción si pensamos añadir alguna vez una terminación sin punto.

21.4. `expr`

`expr` es un comando no interpretado. Evalúa expresiones formadas por los parámetros. Se suele usar colaborando con `sh` para hacer cálculos.

Queremos incrementar el valor de una variable en una unidad.

```
i=‘expr $i + 1‘
```

Supongamos que `i` vale inicialmente 8. Tenemos comillas inversas. Se va a ejecutar lo acotado por las comillas inversas. Previamente se substituye `$i` por su valor. Se ejecuta `expr 8 + 1` . En la salida estándar hay un `9` . Se substituye lo acotado por las comillas inversas, incluyendo las mismas comillas, por el `9` . Queda `i=9` . Se asigna ese valor a la variable `i` .

Es imprescindible dejar al menos un espacio blanco entre los operandos y operadores de `expr` . Si ponemos `expr 4+2` no obtenemos `6` .

> También `bc` es capaz de evaluar expresiones. Obtiene los datos por la entrada estándar, mientras que `expr` trabaja con los parámetros, en la misma línea de comandos.
>
> Para incrementar el valor de la variable `i` escribiríamos
>
> ```
> i=‘bc <<FIN
> $i + 1
> FIN‘
> ```

`expr` admite los siguientes operadores:

- `|` `&` : **or** y **and** respectivamente;
- `=` `>` `<` `>=` `<=` `!=` : igual, mayor, ... y distinto;
- `+` `-` `*` `/` `%` : suma, resta, producto, cociente y resto;
- `:` : el primer operando **se ajusta** al segundo operando. Sigue las reglas de las expresiones regulares antiguas, como `ed` .

Para que `sh` no interprete los caracteres que tendrían un significado especial, se debe anteponer `\` a los operadores `| & > < >= <= *` .

21.5. `test`

En un programa de `sh` queremos tomar decisiones. Por ejemplo, queremos escribir un mensaje de error si no podemos escribir en un fichero, o si no existe.

> Esto se puede conseguir redirigiendo la salida de `ls -l` y con un programa que analice su salida, pero no es sencillo. Además es un problema que se presenta frecuentemente.

`test` *expresión* devuelve 0 si la expresión es cierta.
`[` *expresión* `]` es equivalente.

`test` tiene un segundo nombre : `[` . Cuando invoquemos al comando con el nombre `[` , debemos acabar el comando con `]` . Este corchete derecho debe ser una pieza (*token*) separada. No debe ir junto al parámetro anterior. Cuando invoquemos a este comando con `test` no se debe acabar con `]` .

```
$ cd /usr/bin
$ ls -li test \[
 17122 -rwxr-xr-x  2 root         5200 Jul  2  1993 [
 17122 -rwxr-xr-x  2 root         5200 Jul  2  1993 test
```

`test`, expresiones

- sobre el sistema de ficheros
  - `-r f` da cierto cuando `f` existe y se puede leer.
    "... se puede ..." equivale a "... el usuario que ejecuta el programa puede ..."
  - `-w f` : `f` (existe y) se puede escribir en él.
  - `-x f` : `f` (existe y) tiene bit `x` . En el caso de que `f` sea un fichero, es ejecutable.
  - `-f f` : `f` (existe y) es un fichero normal.
  - `-d d` : `d` (existe y) es un directorio.
  - `-c f` : `f` (existe y) es un dispositivo de caracteres.
  - `-u f` : `f` (existe y) tiene bit `s` de usuario.
    (Al final de esta lista comentamos el bit `s` .)
- sobre la salida estándar.
  - `-t` : da cierto cuando la salida estándar va a un terminal.

    La mayor parte de los programas se comportan igual, y su salida es igual independientemente de que se redirija su salida estándar o no. Algún programa se comporta de forma distinta dependiendo de a dónde va su salida estándar. Por ejemplo, algunas implementaciones de `ls` escriben en varias columnas en pantalla, y escriben a una columna cuando la salida va a un fichero. Esto es útil y razonable. De todas formas, conviene usar el mínimo de veces esta posibilidad, ya que confunde a los usuarios.
- sobre tiras de caracteres.
  - `-z s1` : la longitud de la tira `s1` es cero.
  - `-n s1` : la longitud de la tira `s1` no es cero.
  - `s1 = s2` las tiras son idénticas.
    (se usa un solo `=` porque no existe una operación de asignación.)
  - `s1 != s2` las tiras no son idénticas.

- sobre números (tiras de caracteres consideradas como números)
  - `n1 -eq n2` : los números son iguales.
    `2` y `02` son iguales considerados como números, y son distintas consideradas como tiras de caracteres.
  - `n1 -gt n2` : `n1` es mayor que `n2` .
    `-ne -lt -ge -le` corresponden a *distintos*, *menor*, *mayor o igual* y *menor o igual* respectivamente.

  Conviene estar atento y, por ejemplo, no usar `-eq` con tiras de caracteres.

- compuestas
  - *expr1* `-a` *expr2* : **and**
  - *expr1* `-o` *expr2* : **or**
  - `-!` *expr2* : negación (**not**)
  - `( ... )` : agrupamiento
    Los caracteres `(` y `)` llevarán normalmente `\` para evitar que `sh` los intérprete (como agrupación de comandos).

21.5.1. El bit s

El uso del mecanismo de permisos '`rwx`' nos permite conceder o denegar permiso en un modo de 'todo o nada'. El *bit s* nos va a permitir mayor flexibilidad. Podremos describir mediante un programa los permisos.

Si un proceso ejecuta (a través de la llamada al sistema `exec`) un programa que tiene *bit s* de usuario, pasa a tener como *identificador efectivo de usuario* el del dueño del fichero (programa). El proceso podrá hacer (casi todo) lo que podía hacer el dueño del programa, pero a través del programa.

un ejemplo

Supongamos que queremos administrar una pista de tenis de un club. Este club tiene como norma que la pista se puede usar entre las 10 y las 21 horas. Puede efectuar la reserva cualquier miembro del club, con una

antelación máxima de una semana, y siempre que la pista esté previamente libre (obvio), y que ninguno de los jugadores haya efectuado otra reserva. (Nota: no juego al tenis, y no sé si estas normas son razonables.)

Vamos a utilizar un fichero `hojaReservas` para apuntar en él las reservas efectuadas y las horas libres. Puede tener, por ejemplo, una línea por cada hora y día de la semana, y el nombre de los jugadores en caso de que esté reservada. El fichero `hojaReservas` tendrá como dueño a `admTenis` , y tendrá permisos `rw-r--r--` .

Escribimos un programa de nombre `reservar` del que será dueño `admTenis`, y que tendrá permisos `rwsr-xr-x` . La `s` en la tercera posición indica que tiene *bit s* de usuario.

```
$ ls -l
-rw-r--r--   1 admTenis        145 Jan 23 15:34 hojaReservas
-rws--x--x   1 admTenis       6520 Jan 23 15:14 reservar
```

- Cualquier usuario puede ejecutar el programa `reservar` dado que sus permisos son `rws--x--x` .
- Su proceso pasará a tener como *identificador efectivo* de usuario `admTenis`, porque éste es el dueño de `reservar` .
  (`-rwsx--x--x  1 admTenis`) .
- Su proceso podrá escribir en `hojaReservas` porque los permisos son `rwx-r--r--` .
- Escribirá sólo lo que el programa le deje. Si el programa está bien escrito sólo realizará apuntes conforme a las normas.
- Un usuario cualquiera no puede escribir directamente en `hojaReservas` porque no tiene permiso (`rw-r--r-x-`) .

21.6. `read`

`echo` hace el papel de sentencia de salida de `sh` para comunicarnos con el usuario.

`read` es el comando de entrada de `sh` .

`read` *variable* lee una línea de la entrada estándar y la asigna a la *variable* .

Si esperamos varias palabras (piezas) en la línea de la entrada estándar, podemos poner varias variables. A cada variable se le asignará una palabra (pieza). Si faltan palabras (piezas), las últimas variables se quedarán sin que se les asigne nada. Si faltan variables, se asignará la tira formada por las últimas palabras a la última variable.

```
$ cat p1
read v1 v2
echo =$v1=  =$v2=

$ p1 <<FIN          $ p1 <<FIN          $ p1 <<FIN
unodos              uno dos             uno dos tres
FIN                 FIN                 FIN
=unodos= ==         =uno= =dos=        =uno= =dos tres=
```

21.7. dirname

`dirname` *nombreDeObjeto* toma todo lo situado antes del último `/` del *nombreDeObjeto*.

```
$ dirname /a/b/c/d.e
/a/b/c
$ dirname d.e
.
```

Como el nombre de un objeto que no empieza por `/` no cambia si se le añade `./` por la izquierda, `dirname d.e` es '.' .

```
cd ‘dirname $0‘
(date; who) >>quienes
```

Las dos líneas anteriores están en un fichero ejecutable, de nombre `foto` . Si el fichero `foto` no está en un directorio de `$PATH` , independientemente del lugar desde el que se llame al programa, `cd` nos lleva al directorio donde está el fichero `foto` . El fichero `quienes` que aparece en la segunda línea será el que está en el mismo directorio que `foto` .

21.8. basename

`basename` *nombreDeObjeto* toma todo lo situado después del último `/` del *nombreDeObjeto*.

`basename` *nombreDeObjeto terminación* toma todo lo situado después del último `/` del *nombreDeObjeto* y elimina la *terminación* si puede.

```
$ basename /a/b/c/d.e
d.e
$ basename /a/b/c/d.e .e
d
$ basename /a/b/c/d.e .f
d.e
```

```
nom=‘basename $1 .c‘
cc $1 -o $nom
```

En el ejemplo anterior queremos compilar el programa `prog/dado.c`, y que el ejecutable tome el nombre (`-o ....`) `dado`.

21.9. Desviación (temporal) de la entrada

`.` *fichero* toma la entrada del *fichero*. Si lo tecleamos, es como si tecleásemos el contenido del *fichero*. Si lo ponemos en un programa de `sh` es como si incluimos en ese punto el contenido del *fichero*. No se crea un proceso nuevo para esas líneas. Acabado el *fichero*, `sh` vuelve a tomar las líneas del lugar en el que encontró `.` *fichero*.

¿ En qué se diferencia esto de un programa de `sh` ?

En la mayor parte de los casos no se nota la diferencia. Pero puede haber diferencia.

Para un programa de `sh` se crea un nuevo proceso. Los cambios en variables locales no son observables después de la ejecución de un programa de `sh`, y sí son observables después de una desviación temporal de la entrada por el mecanismo ‘ `.` ’.

Por ejemplo, si dentro de un fichero hacemos: `cd`, `umask 077`, o asignamos valor a una variable de `sh`, el efecto no se verá después de la ejecución de un programa de `sh` y sí se verá después de una desviación temporal de la entrada.

Trabajando con `csh` o `tcsh` el mecanismo equivalente de desviación temporal de la entrada se consigue mediante `source` *fichero* .

21.10. Inicialización

Como muchos programas interactivos, `sh` se puede personalizar. `sh` lee, acepta inicialmente, como si se teclease, los ficheros `/etc/profile` y luego `$HOME/.profile` .

El fichero `/etc/profile` es el mismo para todos los usuarios del sistema. Su contenido lo decide el administrador del sistema (*root*).

Cada usuario tiene o puede tener su fichero `$HOME/.profile` . Puede crearlo, modificarlo o borrarlo. (Suponemos que los usuarios tienen permiso de escritura en su directorio `$HOME`).

Al leerse después el fichero `$HOME/.profile` , cada usuario tiene la última palabra, y puede modificar las decisiones de `/etc/profile` que no le convienen.

Eso de 'la última palabra' es en el caso normal. No es cierto en casos extremos. Si en `/etc/profile` hay una línea que pone `exit` no sé qué se puede hacer.

La última palabra está en el tomo II del diccionario de la Real Academia. Dicen que dentro de poco estará en disco compacto. ;-) .

```
$ cat .profile
umask 022

PATH=$PATH:/usr/bin/graph
export PATH
EXINIT='set nu wm=8'
export EXINIT
PS1="!"
export PS1

mesg n

cat pers/pendiente
```

En el ejemplo anterior, el usuario:

- Pone la *máscara u* a 022 ,
- Añade el directorio `/usr/bin/graph` al valor de `$PATH` .

  El usuario piensa utilizar programas que están en ese directorio.
- Marca para exportar la variable `PATH` .
- Pone la variable de inicialización de `vi` , y la marca para exportar.
- Cambia el *indicador de preparado* (prompt) de `sh` .
- Indica que no quiere mensajes en el terminal (`mesg n`) .
- Envía a pantalla el contenido del fichero `pers/pendiente` . (Un poco primitiva esta agenda).

21.11. Trazas

Es posible que un programa de `sh` tenga un error. Para encontrar el error conviene repasar el programa. Si repasamos una ejecución del programa, vamos a decir que hacemos un *seguimiento*. También es posible que queramos hacer un seguimiento para entender mejor el programa.

Una forma sencilla de observar el comportamiento de un programa es introducir sondas del tipo:

```
echo "el programa 'nudoGordiano' pasa por superSur"
...
echo "la variable 'veleta' toma el valor $veleta"
```

Tenemos dos opciones (*flags*) para que `sh` nos dé más información. Ambas opciones son independientes. Se puede activar una, ninguna o las dos.

- `sh -v` imprime las líneas según las lee el intérprete.
- `sh -x` imprime los comandos antes de ejecutarlos, precedidos de `+` y con las referencias a 'valor de variable' substituidas.

```
$ cat com
for i
do
  echo $i $i
done
echo fin
```

Vamos a ver el efecto de las trazas en la ejecución. El programa del ejemplo repite dos veces cada parámetro (`for .. done`) y termina escribiendo `fin` .

```
$ sh -v com 1 2
for i
do
echo $i $i
done
1 1
2 2
echo fin
fin
```

```
$ sh -x com 1 2
+ echo 1 1
1 1
+ echo 2 2
2 2
+ echo fin
fin
```

La traza debida a las opciones `-v` y `-x` está señalada mediante recuadros.

Para activar la traza-x hemos ejecutado el programa poniendo `sh -x` *programaSh* . Una forma alternativa es incluir en el programa una línea con `set -x` . Cuando no queramos traza, podemos borrar esta línea o convertirla en comentario con un `#` en la primera columna.

Igualmente podemos incluir una línea con `set -v` para activar el otro tipo de traza.

Es frecuente que las trazas alarguen la salida de información y ocupen más de una pantalla. Conviene redirigir la salida de los programas que tienen activada la traza para poder estudiarla detenidamente. La cuestión es que las trazas se escriben por la *salida estándar de errores*. Tenemos que redirigir la salida estándar de errores, y además se suele redirigir al mismo lugar que la *salida estándar*.

21.11.1. Redirección de la salida estándar de errores

Para redirigir la *salida estándar de errores* al mismo lugar que la *salida estándar* escribimos `2>&1` . Supongamos que queremos redirigir las salidas

estándar y de errores del programa `p7` al fichero `log` . Escribiremos:
`p7 >log 2>&1` ,
mientras que si además queremos ver las salidas por pantalla las redirigimos a un `tee` escribiendo:
`p7 2>&1 | tee log` .

Es importante poner `2>&1` en su sitio: después, cuando va con `>` , y antes, cuando va con `|` .

La forma de redirigir la salida estándar de errores es específica del tipo de intérprete de comandos. Lo expuesto hasta ahora es la forma de trabajar con `sh` y `bash` .

> Para redirigir la salida estándar de errores con `csh` y `tcsh` en el ejemplo anterior se escribe
> `p7 >&log` o `p7 |& log` .

21.12. Funciones

La *función* y el *procedimiento* son mecanismos de abstracción que disminuyen los errores, aumentan la productividad y dan lugar a la reutilización.

`sh` ofrece con el nombre de *función* algo que más bien corresponde al concepto de *procedimiento* en otros lenguajes de programación. Esta función de `sh` equivale a dar nombre a un conjunto de líneas o comandos.

Una función de `sh` tiene la forma:

nombreDeLaFunción `()`
`{`
  listaDeComandos
`}`

En el cuerpo de la función (*listaDeComandos*) los parámetros de *nombreDeLaFunción* substituirán a `$1` , `$2` , ...

```
$ cat com
letrasYblancos ()
{
  if [ -r $1 ] ; then
    otros=‘tr -d ’[a-z][A-Z][\012] ’ \
                      <$1 | wc -c‘
```

```
    if test $otros -ne 0 ; then
      echo "hay car. distintos de letras y blancos"
    fi
  else
    echo no existe: $1
  fi
}

for i in uno dos tres
do
  letrasYblancos $i
done
```

En el ejemplo anterior, recorro tres ficheros (`for i in uno dos tres`), llamando tres veces a la función `letrasYblancos` . La primera vez se ejecuta `letrasYblancos uno` .

- Se pregunta si el fichero `uno` existe y es legible (`if [ -r $1 ]`)
- En caso de respuesta negativa se escribe un mensaje (`else echo no existe: $1`).

  No estaría de más escribir el nombre del programa que falla (`$0`), incluir en el mensaje la posibilidad de que el fichero exista y no tenga permiso de lectura, y terminar con un código distinto de 0 (por ejemplo con `exit 1`).
- En caso de respuesta positiva, se manda a la salida estándar el contenido del fichero `uno` (`$1`) borrando las letras, blancos y cambios de línea (`tr -d ...`) .
- Se cuentan con los caracteres restantes (`| wc -c`).
- Y se asigna a la variable `otros` el resultado en la salida estándar de lo anterior (`otros=‘ ... ‘`).
- Se pregunta si el valor de `otros` es distinto de 0 (`test $otros -ne 0`).
- En caso afirmativo se escribe un mensaje (`echo "hay car... "`) .

El programa del ejemplo anterior se puede escribir usando dos ficheros: `com` y `letrasYblancos`. En el primer fichero (`com`) estarían las cuatro últimas líneas, es decir el ciclo `for`. En el segundo fichero estaría el texto de la función, sin la línea del nombre y `()`, y sin las llaves que encierran la lista de comandos.

El uso de funciones tiene varias ventajas:

- Es más fácil copiar, mover o editar una aplicación con un fichero que con varios.
- Las funciones no crean procesos, por lo que su uso es más eficaz. Se ahorra memoria y el tiempo de creación de nuevos procesos.

La pega de usar funciones, en vez de ficheros separados, es que desde fuera del fichero no se tiene acceso a las funciones.

Hay que tener en cuenta otra diferencia entre el uso de funciones y ficheros. Cuando se usan funciones no se crea un nuevo ámbito para las variables.

```
fun ()
{ for i in 1 2 ; do
    :
  done }

 for i in a b ; do
   fun $i
   echo $i
 done
```

En el ejemplo anterior sólo tenemos una variable `i` y no dos. El programa anterior escribirá:

```
2
2
```

Si en vez de ser una función, `fun` hubiese sido un comando, la salida habría sido:

```
a
b
```

21.13. Ejercicios

Ejercicios sobre el intérprete `sh` : 95f.15 (pág. 316), 95f.18 (pág. 317), 95j.2 (pág. 320), 95j.6 (pág. 320), 95j.15 (pág. 322), 95j.17 (pág. 323), 95s.17 (pág. 329), 95s.18 (pág. 329), 96f.10 (pág. 333), 96f.11 (pág. 333), 96f.19 (pág. 335), 96j.9 (pág. 339), 96j.17 (pág. 340), 96s.12 (pág. 345), 96s.17 (pág. 346), 97f.7 (pág. 351), 97f.11 (pág. 352), 97f.16 (pág. 353), 97j.16 (pág. 358), 97j.18 (pág. 359), 97s.8 (pág. 363), 97s.13 (pág. 364), 97s.16 (pág. 365), 98f.13 (pág. 370), 98f.14 (pág. 370), 98f.16 (pág. 371), 98f.18 (pág. 371), 98j.11 (pág. 375), 98j.12 (pág. 376), 98j.16 (pág. 377), 98j.17 (pág. 377), 98j.18 (pág. 377), 98s.6 (pág. 380), 99f.1 (pág. 386), 99f.16 (pág. 388), 99f.18 (pág. 389).

21.14. `csh`

`set history=100` para guardar los 100 últimos comandos tecleados. (Hemos asignado 100 a la variable `history` .)

`history` escribe los comandos que recuerda. Si hemos tecleado la línea anterior, escribirá los 100 últimos comandos.

`!!` se substituye por el último comando.

`!`*caracteres* se substituye por el último comando tecleado que empiece por *caracteres* . Por ejemplo: `!so` se substituye por el último comando que empiece por `so` .

> Durante un trabajo, puede bastar repetir: `!v` , `!p` , y `!f` para editar el programa `fac.p` , compilarlo con pascal y ejecutar el programa `fac` varias veces, mientras se quita errores .
>
> Esta posibilidad ha encontrado competencia. En los intérpretes `tcsh` (y `bash`) usando las flechas se consigue una funcionalidad parecida.

`^t1^t2` cambia `t1` del último comando por `t2` .

`alias` *tira-1 tira-2* hace que en adelante cuando *tira-1* aparezca como comando (`$0`) se substituya por *tira-2*. Por ejemplo: `alias ls 'ls -xF'` .

`~` es lo mismo que `$HOME` .

`~`*identificador-de-usuario* es el directorio de entrada de ese usuario.

Se usa por ejemplo en: `cp ~a0009/hoja h` .

El intérprete `csh` inicialmente lee el fichero `~/.cshrc` . Si es 'de entrada' lee también el fichero `~/.login` a continuación del anterior.

> 'De entrada' hace referencia a que ese intérprete es el primero en atendernos después de habernos conectado a una máquina, es decir, después de hacer 'login' dando nuestro identificador y contraseña.
>
> Por ejemplo, en `.login` se suele poner la presentación del correo que llegó mientras no estábamos conectados.
>
> En `.cshrc` se suele poner el valor de variables que interesa para todas las instancias de `csh` .
>
> Hay más detalles a tener en cuenta cuando trabajamos con un entorno con Xwindows.

La sintaxis de las estructuras de control (if, while, for, case) y de las expresiones (`expr` y `test`) es distinta que en la `sh` .

Por ejemplo, para cambiar el nombre de tres ficheros escribiríamos:

```
foreach i (f1 f2 f3)
  mv $i $i.v
end
```

> Hay quien recomienda no usar `csh` como lenguaje para programar. *unix-faq.shell.csh-whynot:*
>
> > The following periodic article answers in excruciating detail the frequently asked question "Why shouldn't I program in csh?". It is available for anon FTP from convex.com in /pub/csh.whynot

`tcsh` es una mejora de `csh` .

El intérprete `tcsh` para inicializarse intenta leer el fichero `~/.tcshrc` y si no existe, lee el fichero `~/.cshrc` .

Capítulo 22

en la red

22.1. `rlogin`, `rsh` y `rcp`

Hay tres comandos relacionados `rsh` , `rcp` y `rlogin` . Para usarlos tendremos cuenta en otra máquina conectada por red. Normalmente en la máquina remota estaremos autorizados mediante una línea en el fichero `~/.rhosts` .

> Los permisos del fichero `~/.rhosts` deben ser 400 ó 600. No conviene que los demás usuarios conozcan desde qué otros equipos se puede acceder a la cuenta.

`rlogin` *nombre-de-otra-máquina* nos permite entrar en nuestra cuenta de la otra máquina.

Si en la máquina remota tenemos el mismo nombre de cuenta y en el fichero `~/.rhosts` de la máquina remota autorizamos a la máquina local, entraremos directamente. En caso contrario tendremos que dar la contraseña.

Si los nombres de usuario remoto y local son distintos se puede escribir: `rlogin -l` *nombre-usuario-remoto nombre-de-otra-máquina* .

Se usa `rlogin` cuando se quiere ejecutar **interactivamente** varios comandos en la máquina remota.

`rsh` *nombre-de-máquina comando* ejecutará el *comando* en la *máquina* indicada. El comando estará formado por todo lo que aparezca a continuación del nombre de la máquina. El comando se ejecutará en el directorio de entrada en la máquina indicada. La entrada estándar se toma de la máquina local, y la salida estándar se lleva a la máquina local.

`rsh -n` *nombre ...* ejecuta el *comando* tomando la entrada de `/dev/null` .

`/dev/null` conectado a la entrada estándar de un proceso, ofrece de inmediato el *fin de fichero*. Equivale a un fichero vacío.

`/dev/null` conectado a la salida estándar de un proceso, no tiene ningún efecto. Desaparece la salida sin ocupar disco, ni molestar en pantalla. Actúa como un agujero negro.

Es más, las dos afirmaciones anteriores son ciertas también para entradas y salidas distintas de la estándar.

`cp /dev/null f` deja el fichero `f` con tamaño cero sin cambiar el dueño ni los permisos.

Al escribir el comando a ejecutar en la máquina remota, podemos usar comillas (`'`), o eslás inverso (`\`) si queremos que expanda los caracteres especiales `* ? [ - ]` en la máquina remota. A veces se usan paréntesis y entrecomillado para cambiar el directorio de ejecución del comando.

`rsh` se usa cuando se quiere ejecutar interactivamente un comando en una máquina remota, y para escribir programas de `sh` (*scripts*) que realicen parte de su trabajo en otras máquinas.

`rcp` copia ficheros, generalmente de una máquina a otra. Tanto el origen como el destino de la copia pueden ser locales o remotos. Es posible la combinación de copia en que el origen y el destino sean ambos remotos, en el mismo o en distintos ordenadores. `rcp` tiene similar sintaxis y significado que `cp` .

Cuando un fichero o directorio estén en otra máquina se indica con el formato *máquina*:*objeto*. Cuando el nombre del objeto remoto no empieza por eslás (`/`) se considera relativo al directorio de entrada en la máquina remota.

`rcp a genio:b` copia el fichero `a` de la máquina local en el fichero `b` del directorio de entrada de la máquina `genio` .

`rcp quijote:/tmp/cap\* genio:tmp` copia todos los ficheros del directorio `/tmp` de la máquina `quijote` cuyo nombre empiece por `cap` al directorio `$HOME/tmp` en la máquina `genio` . El eslás inverso (`\`) sirve para que el asterisco (`*`) se expanda en la máquina `quijote` y no en la máquina local.

```
rsh quijote '(cd d; ls)' >saco
```

```
sed -e 's/^/aplica /' saco >saco2  ; rm saco
cat cabecera saco2 >remoto  ; rm saco2
rcp remoto cajal:
rsh cajal chmod 744 remoto
rsh cajal remoto
rsh cajal rm remoto
```

En el ejemplo anterior, en la primera línea, se obtiene cierta información del ordenador `quijote` . A partir de esa información, en las líneas segunda y tercera, se construye (localmente) un comando (`remoto`). Se copia ese comando al ordenador `cajal` (`rcp`). Se pone permiso de ejecución al comando en `cajal` . Se ejecuta (`rsh`) ese comando y se borra.

22.2. Correo

> En los primeros tiempos del correo entre máquinas distintas, el transporte de correo era semejante al transporte informal de paquetes aprovechando los viajes de un conocido. De un pueblo salía un autobús para otro y se encargaba a un viajero: "lleva este paquete para Tomás en Pinto". Si de Pinto salía otro autobús para Valdemoro, se encargaba: "lleva este paquete a Pinto, y allí encargas que lo lleven a Paco en Valdemoro"

En los primeros tiempos del correo entre máquinas distintas, una máquina se conectaba (por la noche, por módem, p.ej.) a otras pocas (1, 2, 3, ..). Cuando el usuario iba a leer el correo en una máquina con la que existía conexión directa, se mandaba correo indicando *máquina*`!`*usuario* . Por ejemplo, se escribía `mail pinto!tomas` . Si el destinatario estaba en una máquina accesible a través de una intermedia, había que ponerla en la dirección. Así escribiríamos `mail pinto!valdemoro!paco` .

> En el esquema anterior, el camino (encaminamiento) aparece explícitamente en la dirección. Según crece la red de ordenadores, o se modifica la topología de la red, este esquema resulta incómodo y poco flexible.

Hace ya unos años se pasó al uso de nombres de dominios.

Hay unos cuantos dominios de primer nivel (*top*), cuyo nombre (por brevedad) suele ser parte del nombre de un país, o de un tipo de entidad.

> `es` para España, `fr` para Francia, ..., `com` para entidades con ánimo de lucro en EEUU, `edu` para universidades en EEUU, ... Al ser el primer usuario EEUU, no usa un dominio para el país.

En cada dominio, hay un administrador de los nombres de dominio o máquina debajo suyo.

> En España, REDIRIS adjudica nombres de dominio de segundo nivel debajo de `es` . Uno de esos dominios es `upm.es` y corresponde a la Universidad Politécnica de Madrid. Uno de los subdominios de `upm.es` corresponde a la Escuela Universitaria de Informática y su nombre es `eui.upm.es`. En el Centro de Cálculo administran el dominio `eui.upm.es` y autorizan la asignación de nombres a las máquinas. Así llegamos a que el nombre de una máquina es `cajal.eui.upm.es` .

Al enviar correo con nombres de dominio se pone `mail` *usuario* `@`*nombre-de-dominios*. Cuando la parte final del nombre de dominios coincide con la de la máquina local se puede omitir.

`mail gaspar@cajal.eui.upm.es` envía correo al usuario `gaspar` en el ordenador `cajal` del dominio `eui.upm.es` . Lo mismo se consigue con `mail gaspar@cajal` si estamos en el dominio `eui.upm.es` .

El esquema de nombre de dominios se puede usar también con otros comandos como `rcp`, `rsh`, `rlogin` y `ftp` .

22.3. Otros comandos

22.3.1. `ping`

Para saber si una máquina está accesible desde donde estamos, podemos hacer `ping` *máquina* .

```
$ ping elvis
elvis is alive
```

En el ejemplo anterior tenemos que la máquina de nombre `elvis` del mismo dominio está accesible. (Hay un guiño para los que consideran a Elvis un viejo rockero).

```
$ ping cajal
PING cajal.eui.upm.es (138.100.56.6): 56 data bytes
64 bytes from 138.100.56.6: icmp_seq=0 ttl=255 time=4.3 ms
64 bytes from 138.100.56.6: icmp_seq=1 ttl=255 time=3.0 ms

--- cajal.eui.upm.es ping statistics ---
2 packets transmitted, 2 packets received, 0% packet loss
round-trip min/avg/max = 3.0/3.6/4.3 ms
```

En este otro ejemplo, la versión de `ping` de LINUX nos informa del tiempo que tarda un paquete hasta `cajal` . Hemos tecleado rápidamente `^C` (Control-C) para interrumpir el comando. Si no lo interrumpimos nos llena la pantalla ... . Sale una línea cada segundo.

> El comando `ping` es el primero que tecleamos cuando dudamos de la conexión de nuestro ordenador o del ordenador remoto a la red. Es algo así (y algo más) como el tono del teléfono.
>
> El número de milisegundos nos da una idea de la calidad de la conexión y disponibilidad del ordenador remoto. Conectados a través de un módem (P. ej. a 28800 bits/s.) los valores se incrementan en casi 200 milisegundos. Los paquetes perdidos en la última línea indican una conexión cargada o problemática.
>
> En general, estas conexiones funcionan como una cadena que será tan débil (lenta) como el eslabón más débil (lento).

22.3.2. `telnet`

`telnet` es una forma de conexión de máquina a máquina. Esta función ya la hacía `rlogin` . `rlogin` está pensado para conectar máquinas tipo UNIX y pasa el entorno (variables de) a la máquina remota. `telnet` no pasa el entorno, pero permite conectar máquinas no-unix a máquinas UNIX.

22.3.3. `ftp`

`ftp` es un programa para transferir ficheros sobre internet.
`ftp` es también el nombre del protocolo usado por esta aplicación.
`ftp` no es la utilidad más cómoda, pero es muy eficaz para llevar grandes cantidades de información.

> Para ir de un sitio a otro con una maleta (o menos) lo más cómodo es ir con un turismo. Para hacer una mudanza o para abastecer un almacén lo más útil es un camión. `ftp` hace el papel de camión.

`ftp` se puede usar con máquinas en las que tenemos cuenta, y si no tenemos cuenta generalmente nos podemos conectar como usuarios anónimos (anonymous), y dar como identificador nuestra dirección de correo a efectos (generalmente) sólo de estadística.

Podemos ver lo que hay en el directorio con `dir` y con `ls`, y podemos cambiar de directorio con `cd` . Si queremos ver todo lo que hay en el servidor de ftp nos transferimos el fichero de nombre `ls-lR` (o `ls-lR.Z` o `ls-lR.gz`), lo descomprimimos localmente, y con `vi` o con `grep` vemos lo que nos puede interesar.

> Ha surgido el término *servidor*. Algunos de los comandos que usamos `rlogin`, `rsh`, `rcp` `ftp`, `xarchie`, ..., funcionan con un esquema *cliente-servidor*.
>
> En una máquina se ejecuta un programa *servidor* que espera peticiones, las atiende, y envía la respuesta a quien hizo la petición. En otra máquina (o a veces en la misma) se ejecuta un programa *cliente* que hace una petición al programa servidor, y obtiene de él una respuesta. El programa cliente está más cerca del usuario.
>
> Los programas servidores reciben frecuentemente el nombre de *daemon* en los sistemas tipo UNIX, y de los *daemon* ya hemos tratado antes.

Si queremos transmitir ficheros que pueden tener caracteres no imprimibles ponemos `ftp` en modo binario (`binary`).

Por encima de `ftp` hay una serie de herramientas como `archie`/`xarchie`, y `mirror`.

22.3.4. `xarchie`

`xarchie` es un cliente sobre xwindows (ventanas) de un *servidor archie*.

Los *servidores archie* son servidores de índices de `ftp`. Un servidor archie se conecta periódicamente (quizá cada 15 días) con gran número de servidores de ftp y consigue sus índices (`ls-lR`), y crea un índice compuesto. Cuando se le pregunta por un nombre, indica las máquinas y directorios en los que se puede encontrar esos objetos. La búsqueda puede hacerse conociendo el nombre exacto o sólo aproximado.

22.3.5. `news`

`news` son los círculos/foros/corros de noticias en los que se habla casi de cualquier tema. Hay varios miles de grupos de `news`. En algunos participan unas pocas docenas y en otros varios miles de personas.

Conviene escuchar (leer) durante un tiempo antes de participar, y también conviene leer el fichero de respuestas a preguntas más frecuentes (FAQ's, Frecuently Asked Questions) para evitar repetir preguntas de recién llegados.

Si alguien tiene algo que decir a otro particular, debe mandarle el mensaje por *mail*, en vez de mandarlo a todo el grupo.

Si alguien hace una pregunta, las respuestas y comentarios irán al que hizo la pregunta. Si la respuesta no es obvia se considera de buena educación escribir un resumen con lo más interesante de las respuestas y enviarlo al grupo de news.

22.3.6. `mirror`

`mirror` es un comando que apoyándose en `ftp` mantiene un directorio o árbol local idéntico o lo más parecido posible a otro directorio o árbol remoto.

`mirror` lleva una copia de `ls-lR` de la máquina remota a la máquina local, compara los datos con los del `ls-lR` local, y transfiere los ficheros remotos que no existan en la máquina local y los que sean más recientes que los del mismo nombre en la máquina local.

22.3.7. FAQ

FAQ's son respuestas a preguntas más frecuentes. Su conjunto forma una pequeña enciclopedia. Es la mejor manera de introducirse en un tema. Normalmente cada grupo de noticias (news) tiene su FAQ.

22.3.8. WWW

www , w^3 (alguien lo lee guau guau guau), también MultiMalla Mundial (mmm o m^3). Es la aplicación que se apoya en un lenguaje de marcas . Ese lenguaje es un hipertexto con punteros a objetos ditribuidos (*URL, Universal Resource Locator*). Es muy popular, resulta cómodo e interactivo, aunque no llega a eliminar a las demás aplicaciones de la red.

Desde navegadores de *www* se puede acceder a otros servicios de internet como ftp, archie, gopher, news, mail.

glosario

X/OPEN (1984) Grupo de fabricantes europeos.

OSF (Open Software Fundation) Agrupación de fabricantes mundial, liderada por IBM, con HP, DIGITAL, ...).

USL (Unix Software Laboratories) algo parecido, liderado por el grupo ATT.

UCB (Universidad de California, Berkeley)

Linux El UNIX bueno, bonito y barato. Sólo le falta venir instalado de serie.

GNU El mayor grupo de usuarios contribuyentes de software de dominio público.

X-Window Sistema de ventanas asíncronas (y distribuidas).

Xlib Interfaz de bajo nivel para X-Window.

Motif Interfaz de alto nivel para X-Window. Aspecto, apariencia (look).

Xview *y* **Athena** Otros aspectos para X-Windows.

ACM Asociación de profesionales de informática. Tiene más de 50000 socios y 50 años de antigüedad (80 %? USA).

Surveys.ACM Una revista con resúmenes de informática. Nivel: de introducción para cursos de doctorado (tercer ciclo). Tiene algunos socios más que ACM, y también 50 años de antigüedad.

IEEE Asociación hermana de ACM con un enfoque un poco más cercano al hardware.

ATI Asociación de informática de habla hispana. Tiene unos 6000 socios.

hasta la próxima ...

Saludos a todos, que se puede escribir:

```
saludos *
```

Muy contento de haber estado con vds.

```
:-)
```

Con `banner` se sacan caracteres grandes y podemos poner:

```
$ banner 'THE END'

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

Capítulo 23

Cuentos

23.1. Por las líneas del metro

Un usuario quiere ver lo que hay en los ficheros cuyo nombre empieza por `linea` y teclea:

```
cat linea*
```

Las líneas (de texto) de las líneas (de metro) vuelan. Para verlo más tranquilo teclea:

```
more linea*
```

Sale del comado `more`. Quiere saber cuántas estaciones hay en la línea 3 y teclea:

```
wc linea.3
```

Alguien le comenta que para saber las estaciones podía haber tecleado:

```
wc -l linea.3
```

Luego quiere saber cuántas estaciones hay en las líneas de metro 7, 8 y 9, y teclea:

```
wc linea.[79]
```

No sale la información que quería, tiene que teclear:

```
wc linea.[7-9]
```

(o `wc linea.[789]` o `wc linea.[798]` o ...)
Como quiere saber cuántas estaciones hay en la red de metro se le ocurre teclear:

```
cat linea.* | wc
```

y luego se da cuenta que ha contado varias veces las estaciones que están en más de una línea. Teclea sucesivamente:

```
cat linea.* |sort
!! |uniq
!! |wc
```

o si tiene prisa, ve el resultado con:

```
cat linea.* |sort |uniq |wc
```

Quiere saber qué estaciones están en las línea 5 y 6 a la vez. Ordena alfabéticamente las estaciones en `l5` y `l6` . Teclea:

```
sort linea.5 >l5
sort linea.6 >l6
```

Quiza no está muy seguro y comprueba que todo va bien tecleando:

```
more linea.5 l5
more linea.6 l6
```

Ahora ya puede hallar la intersección de `l5` y `l6` , (porque están ordenadas), y escribe:

```
comm -12 l5 l6
```

Lo de `comm -12` le parece una receta. Para ver el funcionamiento de `comm` teclea:

```
comm l5 l6
```

y obtiene todas las estaciones de las líneas 5 y 6 en tres columnas:

- En la 1^a columna aparecen las estaciones que solo están en la línea 5,
- En la 2^a columna aparecen las estaciones que solo están en la línea 6,
- En la 3^a columna aparecen las estaciones comunes a las dos líneas.

(Y todo ordenado alfabéticamente, con las mayúsculas por delante).

Borra `l5` y `l6` escribiendo:

```
rm l5 l6
```

Edita un fichero con nombre `intersec` :

```
vi intersec
```

y consigue que el contenido del fichero sea:

```
sort linea.5 >l5
sort linea.6 >l6
comm -12 l5 l6
rm l5 l6
```

(ha entrado en modo inserción con `i` , ha salido de modo inserción con *escape*, ha borrado algún carácter con `x` o `X` , ha salido con `:wq`).

Hace ejecutable el fichero tecleando:

```
chmod a+x intersec
```

y lo prueba ecribiendo:

```
intersec
```

¿Es su primer programa para el intérprete de comados? Esto hay que celebrarlo. ¿Lleva 30 años escribiendo programas?. También puede celebrarlo. :-)

Luego (¿después de celebrarlo?) decide mejorar el programa `intersec` . Edita el fichero:

```
vi intersec
```

y cambia los números `5` por `$1` y los números `6` por `$2` . Prueba la nueva versión:

```
intersec 1 2
intersec 6 9
```

Por último, quiere saber cuáles son las estaciones que están en más líneas, y escribe:

```
cat linea.*|sort|uniq -c|sort -nr|more
```

y llega a la conclusión: "casi todas las líneas conducen a ...".

Se queda con la duda de cómo hallar el camino más corto entre dos estaciones, pero no todo van a ser respuestas. ¿Qué es *más corto* ?.

23.2. Lista de notas

Un maestro quiere poner poner las notas a dos alumnos. Con `vi` crea un fichero (de nombre `nota.prob`) cuyo contenido es:

```
Ramon   3  4
Cajal   0  5
```

Como quiere obtener la nota total (con la suma de las notas de los dos problemas) escribe un programa:

```
{ print $0, $2+$3 }
```

en un fichero cuyo nombre es `nota.aw` .

Luego ejecuta el programa con

```
awk -f nota.aw nota.prob | tee nota.sal
```

guardando la salida y viéndola en el terminal.

Para ahorrar trabajo, crea el fichero `Makefile` cuyo contenido es:

```
nota.sal: nota.aw nota.prob
	awk -f nota.aw nota.prob | tee nota.sal
```

pueba a repetir la ejecución con

```
make
```

Como no ha conseguido otra ejecución pregunta a un vecino, y siguiendo su consejo hace

```
touch nota.prob
```

Y para ver qué sucede hace

```
make -n
```

Ahora

```
make
```

ejecuta `awk`, pero otro

```
make
```

le dice que no. Algo hay con el `touch`. (Lee el manual de `touch`, de `make` y se entera).

Le viene un tercer alumno, y le pone en la lista con sus notas

```
Iturrigorriecheandia 2 2
```

Cuando ve sus notas en la lista (con `make`) no le gusta como queda.

La salida de las notas será con formato. Cambia el programa a:

```
{ printf ("%25s %3d %3d %3d\n", $1, $2, $3, $2+$3) }
```

y lo prueba.

Como en una lista larga de alumnos no se leen bien las notas, cambia el programa para que cada 3 líneas se escriba una línea blanca. Decide también añadirle una cabecera. Queda

```
BEGIN { print "  nombre           p1 p2 tot"
        print "-----------------  -- -- ---" }
{ printf ("%25s %3d %3d %3d\n", $1, $2, $3, $2+$3)
  if ((NR % 3) == 0) print "" }
```

Añade líneas a `nota.prob` para ver la separación cada tres alumnos.

Un curioso le dice que quite el `BEGIN` , y que quite `""` . Lo hace, lo prueba, y lo deja como mejor le parece.

Quiere escribir una estadística con la frecuencia de las notas. Añade líneas con:

```
f [$2+$3] ++
```

y

```
END { print ""
      for (i=0; i<=10; i++) print i, f [i] }
```

Más tarde piensa escribir un histograma al final de la lista de notas y cambia la última línea por

```
      for (i=0; i<=10; i++){
        printf ("%3d", i)
        for (j=0; j<f[i]; j++){
          printf ("%s","*") }
        print ""
        }
      }
```

y prueba esta versión del programa.

Capítulo A

Exámenes

A continuación incluimos una serie de exámenes. Cada examen consta de (una mayoría de) ejercicios de análisis:
... Escriba la salida del último comando:
y unos ejercicios de síntesis.

Para los ejercicios de análisis se presenta un escenario: directorio actual, valor de *umask*, nombres de ficheros en el directorio actual, contenido, etc.

Se supone que un usuario teclea una serie líneas, casi todas de comandos. Se pide la salida en terminal debida a la última línea de cada pregunta. El efecto de las líneas tecleadas en cada pregunta pueden afectar a las preguntas siguientes. (De hecho sólo afecta en menos del 20 % de los casos.)

A.1. Modo de uso

Los ejercicios pueden hacerse por exámenes (consecutivamente) o por temas (por ejemplo, todos los del comando `find` seguidos). Quizá lo más práctico sea reservar la mitad de los exámenes para ir haciendo los ejercicios por temas, y la otra mitad por exámenes.

Los ejercicios deben realizarse sin tener el ordenador delante.

Lo mejor es coger un lápiz, una goma, un libro de UNIX, los enunciados, ir a un lugar tranquilo y responder **pensando**.

Más tarde se va a un ordenador con UNIX, y con el mismo escenario se van tecleando los comandos. Si el ejercicio está bien se pone una $\mathcal{B}$

Si no está bien se pone una $\mathcal{M}$, **no se apunta la respuesta correcta**, y se vuelve a pensar sin ordenador y con apuntes.

Y así al menos tres veces (si no respondemos correctamente).

Y si no ha salido bien, escribimos en un papel aparte cuál creemos, desde nuestro estudio sin ordenador, que es el contenido, permisos, etc. de los ficheros implicados en los pasos intermedios. Vamos luego al ordenador a ver en qué paso nos hemos equivocado, y **por qué**.

> Hay dos formas de usar mal estos exámenes. Algunas personas teclean directamente en el ordenador y escriben la salida. Otras lo intentan una vez, y luego escriben la respuesta correcta obtenida en el ordenador.
>
> Si nos cuentan que alguien acumula el resultado de muchas multiplicaciones para aprender a multiplicar nos parecería ingenuo y despistado. Creo que es parecido.

De los ejercicios de síntesis se incluye una solución.

> Normalmente habrá más de una solución correcta. Es posible que alguna más sencilla, y otras más ingeniosas.

Por idéntica razón (aprende el que piensa) se debe intentar resolver estos ejercicios sin mirar la solución propuesta.

> Si no he puesto estas soluciones al revés o legibles con un espejo es *por el qué dirás*.

A.1.1. detalles

Los exámenes de los años 93 y 94 no incluían `find`, `tar` y el último capítulo de `sh`.

Al final de cada examen se adjunta una hoja con respuestas a las ¿dudas? (¿nervios?) más frecuentes.

Por favor, el número de matrícula en **todas** las hojas: una letra y cuatro cifras. Por ejemplo:

z 0983

Si la salida es en minúsculas, poned minúsculas. Un carácter en cada espacio.

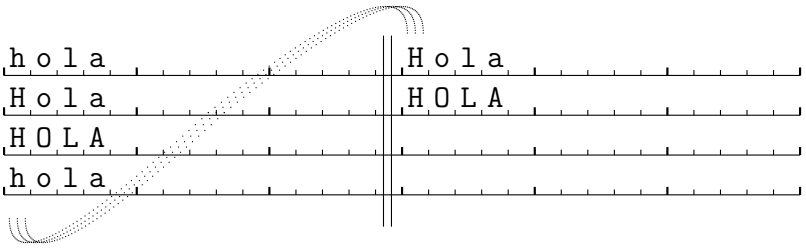
Salida : lo que se ve por el terminal.

último-comando : (sólo la salida de la última línea de comandos)

después : a continuación, con los ficheros modificados.

Se puede duplicar el número de líneas de salida:

```
echo hola >/tmp/saco
echo Hola >>/tmp/saco
echo HOLA >>/tmp/saco
cat /tmp/saco /tmp/saco
```



ascii (mayúscula) < ascii (minúscula)

La redirección simple (`>`) sobre un fichero que ya exite trunca ese fichero (tamaño = 0), y escribe en el fichero.

La redirección doble (`>>`) sobre un fichero que no existe crea ese fichero y escribe en el fichero.

Algunos códigos ASCII ‘ ’: 32, ‘0’: 48, ‘A’: 65, ‘a’: 97.

Se supone que los comandos se ejecutan sin *alias*.

Índice de figuras

Índice alfabético