Introducción a Sistemas Operativos: Empezando

Clips 14 a 16
Francisco J Ballesteros

1. Directorios

Igual que la mayoría de sistemas, UNIX utiliza directorios para agrupar ficheros. Usuarios de Windows suelen llamar "carpeta" a los directorios. Un directorio es tan sólo un fichero que agrupa varios ficheros. Aparentemente, un directorio contiene otros ficheros y directorios, pero esto es otra ilusión implementada por el sistema operativo. Todos los ficheros y directorios están guardados en el disco (en la mayoría de los casos), por separado. No obstante, podemos listar el contenido de un directorio y veremos una lista de ficheros (y directorios). Dos ficheros en dos directorios distintos siempre serán ficheros distintos aunque tengan el mismo nombre.

En pocas palabras, los ficheros están agrupados en un árbol cuya raíz es un directorio (el directorio raíz, llamado "`/`" o *slash* en UNIX). Dentro del raíz tenemos ficheros y directorios, y así sucesivamente. La figura 1 muestra parte del árbol de ficheros que puedes ver en un sistema UNIX.

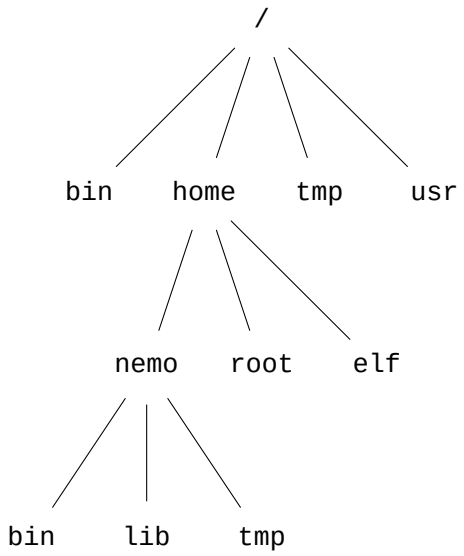

Figura 1: *Ejemplo de árbol de ficheros en UNIX.*

Tendrás muchos mas ficheros en cualquier sistema que utilices. El directorio raíz (`/`) contiene directorios llamados `bin`, `home`, `tmp`, `usr`, y otros que no mostramos. El directorio `bin` suele contener el binario de comandos del sistema. El directorio `home` suele contener un directorio por cada usuario, para que dicho usuario pueda dejar sus ficheros dentro del mismo. El directorio `tmp` suele utilizarse para crear ficheros temporales, que queremos utilizar sólo durante un rato y borrar después. En nuestro ejemplo, el usuario *nemo* ha creado directorios llamados `bin`, `lib`, y `tmp` dentro del directorio `nemo`.

Cada fichero o directorio tiene un nombre, escrito con texto en la figura. Pero para localizar un fichero (o directorio) hay que decirle a UNIX cuál es el camino en el árbol de ficheros para llegar hasta el fichero en

cuestión. Por ejemplo, el camino para el directorio del usuario *nemo* sería "`/home/nemo`". Hemos escrito el cambio, o *path*, empezando por el directorio raíz, "`/`", y nombrando los directorios según bajamos por el árbol, separados unos de otros por un "`/`". La última componente del path es el nombre del fichero.

Puedes ver que hay dos directorios llamados `tmp`. Uno es `/tmp` (el que está directamente dentro del raíz) y otro es `/home/nemo/tmp`. Esto es: Empezamos en el raíz, y seguimos por `home`, luego `nemo` y finalmente `tmp`. Ambos paths se denominan absolutos dado que comienzan por el raíz e identifican un fichero (o directorio) sin duda alguna.

Sería incómodo tener que escribir paths absolutos cada vez que queremos nombrar un fichero. Debido a esto, cada programa que ejecuta en UNIX está asociado a un directorio. Dicho directorio se denomina *directorio actual* o *directorio de trabajo* del programa en ejecución que consideremos.

Cuando entramos al sistema y se ejecuta un shell para que podamos ejecutar comandos, el shell utiliza como directorio de trabajo un directorio que el administrador creó al crear nuestro usuario en UNIX (al darnos de alta como usuario en el sistema o crear nuestra cuenta). Para el usuario *nemo*, ese directorio podría ser `/home/nemo`. Decimos que dicho directorio es el directorio *casa* (o *home*) de dicho usuario.

Pues bien, utilizando el árbol del la figura, el usuario *nemo* podría cambiar el directorio de trabajo del shell a otro directorio utilizando el comando que sigue:

```
unix$ cd /tmp
```

En este caso, el directorio ha pasado a ser `/tmp`.

¡Y ahora podemos crear `/tmp/a` como sigue!

```
unix$ touch a
```

No ha sido preciso ejecutar

```
unix$ touch /tmp/a
```

Dado que *touch* le ha pedido a UNIX que cree el fichero "`a`", y que el nombre de fichero (el path) no comienza por "`/`", UNIX entiende que el path hay que recorrerlo a partir del directorio actual o directorio de trabajo. El comando *touch* ha "heredado" el directorio actual del shell, como se verá más adelante. Dado que el directorio era "`/tmp`", el path resultante es "`/tmp/a`".

Si piensas en los comandos que hemos ejecutado anteriormente, verás ahora como los paths que hemos utilizado se han resuelto a partir del directorio de trabajo. Estos paths se denominan *relativos* (no empiezan desde el raíz).

¿Y en qué directorio estamos? Es fácil verlo, utilizando el comando *pwd* que escribe su directorio de trabajo.

```
unix$ pwd
/tmp
```

Y podemos cambiar el directorio de trabajo del shell utilizando el comando *cd*:

```
unix$ cd /
unix$ pwd
/
```

Hay dos nombres de fichero (de directorio, en realidad) que existen en cada directorio del árbol de ficheros. Uso se llama "`.`" y el otro "`..`". Corresponden al directorio en que están y al directorio que está arriba en el árbol, respectivamente. Así pues,

```
unix$ cd .
```

no hace nada. Si el directorio actual es `/tmp`, "`.`" significa `/tmp/.`, lo cual significa `/tmp`. Así que el comando *cd* deja el directorio actual en el mismo sitio.

Si ahora ejecutamos...

```
unix$ pwd
/tmp
unix$ cd ..
unix$ pwd
/
```

vemos que "`..`" en `/tmp` corresponde al directorio raíz. Si seguimos insistiendo...

```
unix$ cd ..
unix$ pwd
/
```

vemos que no es posible subir más arriba en el árbol. No hay nada fuera del raíz.

Es importante entener que cada programa que ejecuta tiene su propio directorio de trabajo. Por ejemplo, podemos ejecutar un shell y cambiar su directorio

```
unix$ pwd
/home/nemo
unix$ sh
unix$ cd /
unix$ pwd
/
```

Ahora, si escribimos el comando *exit*, que indica al shell que termine, volveremos al shell que teníamos al principio, cuyo directorio era `/home/nemo`.

```
unix$ exit
unix$ pwd
unix$ /home/nemo
```

El directorio *home* es tan importante (para cada usuario el suyo) que el comando *cd* nos lleva a dicho directorio (cambia su directorio de trabajo al directorio casa) si no le damos argumentos. Es fácil averiguar cuál es tu directorio *home*:

```
unix$ cd
unix$ pwd
/home/nemo
```

Dispones del comando *mkdir* para crear nuevos directorios. Por ejemplo, es probable que el usuario *nemo* ejecutó un comando como

```
unix$ mkdir bin
```

para crear el directorio `/home/nemo/bin` que viste en el árbol de ficheros de la figura 1.

Puedes borrar directorios utilizando el comando *rmdir*. Por ejemplo,

```
unix$ rmdir /home/nemo/bin
```

borraría el directorio `bin`. Esta vez hemos utilizado un path absoluto.

Los comandos *cp* y *mv* se comportan de un modo diferente cuando el destino es un directorios. En tal caso,

copian o mueven los ficheros origen hacia ficheros en dicho directorio. Por ejemplo, si empezamos en un directorio vacío:

```
unix$ mkdir fich
unix$ touch a b
unix$ ls
a    b    fich
unix$ cp a b fich
unix$ ls fich
a    b
```

Esto es, *cp* ha copiado tanto `a` como `b` a ficheros `fich/a` y `fich/b`. ¿Entiendes ahora los caminos relativos?

Igualmente podemos mover uno o varios ficheros a un directorio:

```
unix$ mv a b fich
unix$ ls
fich
unix$ ls fich
a    b
```

Si ahora intentamos borrar el directorio...

```
unix$ rmdir fich
rmdir: fich: Directory not empty
```

el comando *rmdir* no se deja. Si se dejase, ¿Qué pasaría si ejecutas por accidente el siguiente comando?

```
unix$ rmdir /
```

Tendrías diversión a raudales...

Para borrar un directorio es preciso borrar antes los ficheros (y directorios) que contiene. El comando *rm* tiene la opción `-r` (recursivo) que hace justo eso:

```
unix$ rm -r fich
```

¿Comprendes por qué no debes intentar en casa el siguiente par de comandos?

```
unix$ cd
unix$ rm -r .
```

2. ¿Qué contienen los ficheros?

El UNIX, los ficheros contienen bytes. Eso es todo. UNIX no sabe qué significan esos bytes ni para qué sirven. ¿Recuerdas el fichero fuente en C que escribimos? Podemos ver el contenido del fichero utilizando el comando *cat*:

```
unix$ cat hola.c
#include <stdio.h>

int
main(int argc, char *argv[])
{
    puts("hola\n");
}
```

En realidad *puts* escribe un fin de línea por su cuenta tras escribir el string que le pasamos, por lo que este

programa deja una línea en blanco tras escribir "`hola`". Pero también podemos utilizar el comando *od* (octal dump) para que nos escriba el valor de los bytes del fichero:

```
od -c hola.c
0000000    #   i   n   c   l   u   d   e       <   s   t   d   i   o   .
0000020    h   >  \n  \n   i   n   t  \n   m   a   i   n   (   i   n   t
0000040        a   r   g   c   ,       c   h   a   r       *   a   r   g
0000060    v   [   ]   )  \n   {  \n  \t   p   u   t   s   (   "   h   o
0000100    l   a   \   n   "   )   ;  \n   }  \n  \n
0000113
```

En cada línea, *od* escribe unos cuantos bytes indicando al principio de la línea el número de byte (en octal). Cada línea contiene 16 bytes y por tanto la segunda línea comienza en la posición 20 (`2*8`).

La opción `-c` de *od* hace que se escriban los caracteres correspondientes al valor de cada byte. Pero podemos pedirle a *od* que escriba además el valor de cada byte en hexadecimal:

```
unix$ od -c -t x1 hola.c
0000000    #   i   n   c   l   u   d   e       <   s   t   d   i   o   .
          23  69  6e  63  6c  75  64  65  20  3c  73  74  64  69  6f  2e
0000020    h   >  \n  \n   i   n   t  \n   m   a   i   n   (   i   n   t
          68  3e  0a  0a  69  6e  74  0a  6d  61  69  6e  28  69  6e  74
0000040        a   r   g   c   ,       c   h   a   r       *   a   r   g
          20  61  72  67  63  2c  20  63  68  61  72  20  2a  61  72  67
0000060    v   [   ]   )  \n   {  \n  \t   p   u   t   s   (   "   h   o
          76  5b  5d  29  0a  7b  0a  09  70  75  74  73  28  22  68  6f
0000100    l   a   \   n   "   )   ;  \n   }  \n  \n
          6c  61  5c  6e  22  29  3b  0a  7d  0a  0a
0000113
```

Si te fijas en el fichero mostrado con *cat* y lo comparas con la salida de *od* podrás ver qué contiene en realidad el fichero con tu código fuente. Por ejemplo, tras el "`>`" de la primera línea, hay un byte cuyo valor es 10 (o `0a` escrito en hexadecimal). Dicho byte se muestra como "`\n`" si hemos pedido a *od* que escriba cada byte como un carácter. En el caso de "`\n`", se trata del carácter que corresponde al fin de línea. Es ese el carácter que hace que la segunda línea se muestre en otra línea cuando *cat* escribe el fichero en el terminal (en la pantalla o en la ventana). Si miras ahora el fuente, verás que hay una línea en blanco tras el *include*. En la salida de *od* vemos que simplemente no hay nada entre el `\n` que termina la primera línea y el `\n` que termina la segunda.

Otro detalle importante es que no existe ningún carácter que marque el fin del fichero. No existe *eof*. Igual que un libro, un fichero se termina cuando no tiene más datos que puedas leer.

El fuente estaba tabulado, utilizando el carácter tabulador, ("`\t`", cuyo valor es 9). Al mostrar ese carácter, el terminal avanza hasta que la columna en que escribe es un múltiplo de 8 (puede cambiarse en ancho del tabulador). Por eso se llama *tabulador*, por que sirve para tabular o escribir tablas o texto con forma de tabla. Hoy día se utiliza para sangrar el fuente más que para otra cosa.

Tanto el fin de línea, como el tabulador y otros caracteres especiales, son especiales sólo por que los programas que los utilizan les dan un significado especial. Pero no tienen nada de especial. Siguen siendo un byte con un valor dado. Eso si, si un editor o un terminal muestra un "`\n`", entiende que ha de seguir mostrando el texto en la siguiente línea. Y si muestra un "`\t`", el programa entenderá que hay que avanzar para tabular. A estos caracteres se los denomina caracteres de control.

Otro carácter de control es "`\b`", o *backspace* (borrar), que hace que se borre el carácter anterior. De nuevo, es el programa que utiliza el carácter el que hace que sea especial. Por ejemplo, al escribir una línea de comandos, "`\b`" borra el carácter anterior. ¡Pero no puedes borrar un *intro* si lo has escrito! Cuando

escribes el fin de línea, UNIX le da el texto al shell, con lo que UNIX (el kernel) no puede cancelar el último carácter que escribiste antes de borrar. Eso si, si pulsas un carácter y luego *backspace*, el kernel tira ambos caracteres a la basura (y actualiza el texto que ves en el terminal). La tecla de borrar borra también en un editor de texto sólo debido a que el programa del editor interpreta que quieres borrar, y actualiza el texto que hay escrito.

En este texto, y en la salida de comandos como *od*, se utiliza la sintaxis del lenguaje C para escribir caracteres de control: Un *backslash* y una letra.

Hay algo más que debes saber sobre ficheros que contienen texto. Hace tiempo se codificaba cada carácter con un único byte, empleando la tabla ASCII (7 bits por carácter). El comando *ascii* existe todavía e imprime la tabla:

```
unix$ ascii
|00 nul|01 soh|02 stx|03 etx|04 eot|05 enq|06 ack|07 bel|
|08 bs |09 ht |0a nl |0b vt |0c np |0d cr |0e so |0f si |
|10 dle|11 dc1|12 dc2|13 dc3|14 dc4|15 nak|16 syn|17 etb|
|18 can|19 em |1a sub|1b esc|1c fs |1d gs |1e rs |1f us |
|20 sp |21  ! |22  " |23  # |24  $ |25  % |26  & |27  ’ |
|28  ( |29  ) |2a  * |2b  + |2c  , |2d  - |2e  . |2f  / |
|30  0 |31  1 |32  2 |33  3 |34  4 |35  5 |36  6 |37  7 |
|38  8 |39  9 |3a  : |3b  ; |3c  < |3d  = |3e  > |3f  ? |
|40  @ |41  A |42  B |43  C |44  D |45  E |46  F |47  G |
|48  H |49  I |4a  J |4b  K |4c  L |4d  M |4e  N |4f  O |
|50  P |51  Q |52  R |53  S |54  T |55  U |56  V |57  W |
|58  X |59  Y |5a  Z |5b  [ |5c  \ |5d  ] |5e  ^ |5f  _ |
|60  ‘ |61  a |62  b |63  c |64  d |65  e |66  f |67  g |
|68  h |69  i |6a  j |6b  k |6c  l |6d  m |6e  n |6f  o |
|70  p |71  q |72  r |73  s |74  t |75  u |76  v |77  w |
|78  x |79  y |7a  z |7b  { |7c  | |7d  } |7e  ~ |7f del|
```

Compara el fuente y la salida de *od* con los valores de la tabla.

Pero vamos a crear un fichero con el carácter "α".

```
unix$ echo α >fich
```

Y ahora vamos a ver qué contiene el fichero:

```
unix$ cat fich
α
unix$ od -c fich
0000000  316 261  \n
0000003
```

¡Contiene 3 bytes! En muchos casos, caracteres con tilde como "ñ" o "ó", o letras como "α", y otras muchas no pueden escribirse en ASCII. Eso hizo que se utilizasen otros formatos para codificar texto. Hoy día suele utilizarse Unicode como formato para guardar caracteres. Cada carácter, llamado *runa* o *code-point* en Unicode, corresponde a un valor que necesita en general varios bytes. Para guardar ese valor se utiliza un formato de codificación que hace que para las runas que existían en ASCII se utilice el mismo valor que en ASCII (por compatibilidad hacia atrás con programas que ya existían), y para las otras se utilicen más bytes. De ese modo, "`\n`" sigue siendo un byte con valor 10 (o `0a` en hexadecimal), pero "α" no. Si utilizamos como formato de codificación UTF-8, corresponde con los bytes `ce b1`:

```
unix$ od -t x1 fich
0000000   ce  b1  0a
0000003
```

En cualquier caso, siguen siendo bytes. Tan sólo recuerda que cada sistema utiliza una tabla de codificación de caracteres que asigna a cada carácter un valor entero. Y para sistemas como Unicode, tienes además un formato de codificación que hace que sea posible escribir cada valor como uno o más bytes. Se hace así para para que el texto que puede escribirse en ASCII siga codificado como en ASCII, para permitir que los programas antiguos sigan funcionando igual.

Para UNIX, los ficheros contienen bytes. Pero para los programas que manipulan ficheros con texto y para los humanos los ficheros de texto contienen bytes que corresponden a UTF-8 o algún otro formato de texto.

Los directorios contienen también bytes. Por cada fichero, contienen el nombre del fichero en cuestión y algo de información extra que permite localizar ese fichero en el disco. Pero para evitar que los usuarios y los programas rompan los datos que se guardan en los directorios, y causen problemas al kernel cuando los usa, UNIX no permite leer o escribir directamente los bytes que se guardan en un directorio. Hace tiempo lo permitía, pero los bugs en programas de usuario causaban muchos problemas y decidieron que era mejor utilizar otras llamadas al sistema para directorios, en lugar de las que se utilizan para leer y escribir ficheros. Veremos esto cuando hablemos de ficheros más adelante.