Introducción a Sistemas Operativos: Ficheros

Francisco J Ballesteros

1. Entrada/Salida

Es importante saber cómo utilizar ficheros. En UNIX, es aún más importante dado que gran parte de los recursos, dispositivos y abstracciones del sistema aparentan ser ficheros. Por ejemplo, el teclado, la salida de texto en la pantalla, las conexiones de red, una impresora USB..., casi todo es un fichero o se comporta como tal.

Antes de UNIX la mayoría de los sistemas utilizaban abstracciones diferentes para cada dispositivo (para el teclado, la impresora, etc.). Una de las razones por las que UNIX se hizo popular es que utilizó la abstracción *fichero* como interfaz para todos ellos. Eso permite utilizar los comandos y llamadas al sistema que operan en ficheros para operar en casi cualquier dispositivo. ¡Al contrario que en MS-DOS y en Windows aunque estos sistemas sean posteriores!

En cuanto a ficheros contenidos en un disco, normalmente el disco se suele dividir en trozos más pequeños llamados particiones. Cada partición no es muy diferente a un disco: es una secuencia de bloques siendo cada bloque un array de bytes (normalmente de 512 bytes, 1KiB, 8KiB o 16KiB). UNIX guarda en cada partición estructuras de datos para implementar los ficheros. Se suele llamar **sistema de ficheros** al programa que implementa esa abstracción, el *fichero*, que suele ser parte del kernel. Cuando das formato a una partición para guardar ficheros, determinas el tipo de sistema de ficheros que creas. Veremos más sobre sistemas de ficheros más adelante.

En realidad ya conoces mucho sobre ficheros. En cursos previos has utilizado la librería de tu lenguaje de programación para abrir, leer y escribir ficheros. Y probablemente encontrases problemas que te costaba explicar. Ahora vamos a utilizar el interfaz suministrado por UNIX para manipular ficheros, que es casi el interfaz universal en todos los sistemas operativos, y verás cómo desaparecen los problemas tanto en UNIX como al usar cualquier otro lenguaje.

Considera `printf`. Es una función de librería documentada en *printf(3)* que imprime mensajes con formato. Escribe siempre en un fichero ¡incluso cuando escribes en la pantalla! y para ello llama a *write(2)*. Veamos un programa...

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <err.h>
#include <string.h>

int
main(int argc, char* argv[])
{
    char    msg[] = "hello\n";
    int    n, nw;

    n = strlen(msg);
    nw = write(1, msg, n);
    if (nw != n) {
        err(1, "write");
    }
    exit(0);
}
```

Es algo más elaborado de lo que podría ser. El mensaje `msg` lo guardamos en un array de caracteres en lugar de utilizar `"hello\n"` directamente para que puedas ver lo que hace `write` lo más claramente posible. Vamos a ejecutarlo:

```
unix$ write
hello
unix$
```

¡Hace lo mismo que si utilizamos `printf("hello\n")` o `puts("hello")`!

La llamada *write(2)* escribe bytes en un fichero. El primer parámetro es un `int` que representa un fichero abierto en que se desea escribir. El segundo parámetro es una dirección de memoria, un puntero. Indica dónde tienes los bytes que deseas escribir en el fichero. El último parámetro es el número de bytes que deseas escribir. Tal y como indica el manual, el valor devuelto es el número de bytes que se han escrito y se considera un error que write escriba un número de bytes distinto al que se desea escribir. ¿Entiendes mejor el código ahora?

El fichero en que hemos escrito es simplemente "`1`" en este caso. Los ficheros tienen nombres, como sabes. Los nombres de fichero son strings que UNIX interpreta como paths absolutos o relativos, dependiendo de si comienzan por "`/`" o no. Pues bien, los ficheros abiertos que utiliza un proceso también tienen nombres. En este caso los llamamos **descriptores de fichero**.

Un *descriptor de fichero* es un entero que indica qué fichero abierto se desea usar. Cuando abres un fichero, UNIX te da un nuevo descriptor de fichero. Cuando lo cierras, dicho descriptor deja de existir. El descriptor de fichero es simplemente un índice en un array de ficheros abiertos. Este array lo mantiene UNIX para cada proceso. Mira la figura 1.

El convenio en UNIX es que todos los procesos parten con tres descriptores de fichero abiertos: `0`, `1` y `2`. Corresponden a los ficheros llamados **entrada estándar**, **salida estándar** y **salida de error estándar**. La idea es que se espera que el programa lea sus datos de entrada del fichero abierto `0`, o de la entrada estándar o de *stdin*. Igualmente, se espera que el programa escriba cualquier resultado en el fichero abierto `1`, o en la salida estándar o en *stdout*. En cambio, los mensajes de error se escriben en la salida de error estándar, que corresponde con el descriptor `2`. La razón para tener una salida de error estándar separada de la salida estándar es evitar que se mezclen los resultados de la ejecución de un programa con los mensajes de error.

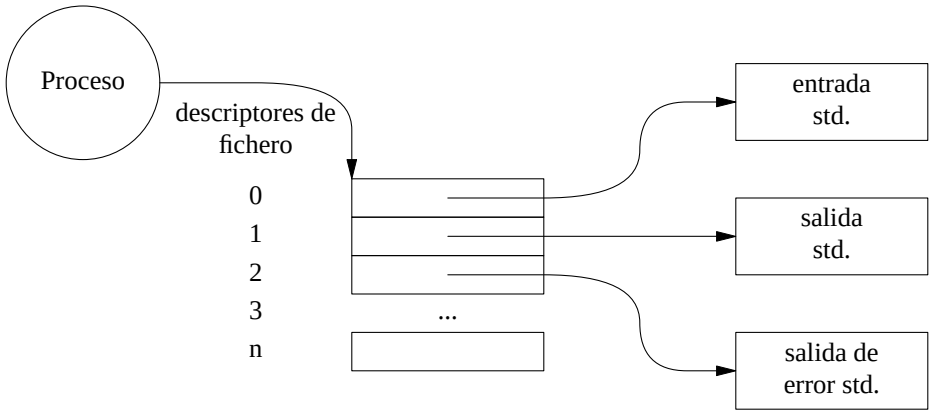

Figura 1: *Los descriptores de fichero son entradas en un array que apuntan a los ficheros abiertos de un proceso. Se utilizan para la entrada estándar, la salida estándar, la salida de error estándar y cualquier otro fichero abierto.*

Por ejemplo, podríamos ejecutar un programa guardando sus resultados en un fichero y, aún así, querremos que los mensajes de error aparezcan en la pantalla.

Cuando ejecutas un programa desde el shell en una ventana de tu sistema la entrada estándar es el teclado en dicha ventana y tanto la salida estándar como la salida de error estándar son la pantalla en dicha ventana. Esto es así a no ser que lo cambies. Recuerda cuando ejecutamos...

```
unix$ echo hola >unfichero
```

En este caso el shell habrá hecho que la salida estándar de `echo` sea `unfichero`. Ya veremos más adelante como funciona esto.

Para leer un fichero (abierto) se utiliza *read(2)*. Igual que `write`, esta función recibe tres argumentos: un descriptor de fichero, una dirección de memoria y un número entero. La llamada `read` lee bytes del fichero indicado por el descriptor de fichero y los deja en la memoria a partir de la posición indicada por el segundo argumento (un puntero). Como mucho se leen tantos bytes como indica el tercer argumento, pero podrían leerse menos. Veamos un programa que lee de la entrada estándar y escribe lo que lee en la salida estándar:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <err.h>

int
main(int argc, char* argv[])
{
    char    buffer[1024];
    int    nr;

    nr = read(0, buffer, sizeof buffer);
    if (nr < 0) {
        err(1, "read");
    }
    if (write(1, buffer, nr) != nr) {
        err(1, "write");
    }
    exit(0);
}
```

Y así es cómo ejecuta...

```
unix$ read1
hola
hola
unix$
```

El primer "`hola`" lo hemos escrito nosotros. El segundo en cambio lo ha escrito el programa `read1`. Una vez lo ha escrito, el programa termina.

Como verás, el programa lee del descriptor `0` y escribe en el `1`. O dicho de otro modo, lee de la entrada y escribe en la salida lo que lee. Hasta que hemos escrito "`hola`" y pulsado *enter* el programa estaba esperando a que escribiéramos. O, más precisamente, el programa estaba dentro de la llamada a *read* y UNIX tenía el proceso bloqueado a la espera de tener algo que leer (esto es, no estaba ejecutando ni listo para ejecutar).

Cuando pulsamos teclas para escribir "`hola`" el teclado envía interrupciones que atiende UNIX. El manejador de dichas interrupciones ha ido guardando los caracteres correspondientes en un buffer hasta que hemos pulsado *enter*. En dicho momento, UNIX ha copiado los caracteres de dicho buffer hacia la memoria del proceso (para atender la llamada a *read* que estaba haciendo) y listo.

¡Esa es la razón por la que puedes borrar caracteres cuando escribes en el teclado! Cuando pulsas la tecla de borrar, UNIX lee el carácter (que es tan bueno como cualquier otro) y entiende que quieres eliminar el último carácter que había en el buffer de lectura de teclado. Cuando pulsas *enter*, UNIX entiende que dicha línea está lista para quien quiera que lea de teclado.

La mayoría de los programas en UNIX aceptan nombres de fichero como argumentos para trabajar con ellos y, lo normal, es que si no indicas ningún nombre de fichero el programa en cuestión trabaje con su entrada estándar.

Por ejemplo, en este caso

```
unix$ echo hola >/tmp/fich
unix$ cat /tmp/fich
hola
unix$ cat /tmp/fich /tmp/fich
hola
hola
unix$
```

el comando *cat* lee `/tmp/fich` y escribe su contenido en la salida estándar. Pero si llamamos a *cat* sin indicar ningún nombre de fichero...

```
unix$ cat
xxx
xxx
yyy
yyy
^D
unix$
```

cat se limita a leer de su entrada estándar hasta el fin de fichero y a escribir todo cuanto lea. Nosotros escribimos una línea con `xxx` en el teclado y *cat* la escribe en su salida. Después escribimos una línea con `yyy` en el teclado y *cat* la escribe en su salida. Hasta el fin de fichero.

¡Un momento! ¿Fin de fichero para el teclado? Pues si. El fichero que corresponde al teclado es una abstracción, ¿Recuerdas?. Cuando mantienes pulsado *control* y pulsas "`d`", (esto es, *control-d*, a veces escrito como `^D` como hemos hecho nosotros en el ejemplo) UNIX entiende que quieres que quien esté leyendo de teclado reciba una indicación de fin de fichero. Así pues, *cat* termina. (Acabamos de mentir respecto a lo que hace *control-d* pero desharemos la mentira en breve).

Cabe otra pregunta... ¿Cómo es que *cat* copia múltiples líneas y nuestro programa sólo una? La respuesta es simple si piensas que no hay magia y piensas en lo que hace *read(2)*. Cuando `read` lee los bytes que puede leer y te dice cuántos bytes ha leído, `read` ha terminado su trabajo. Nuestro programa llamaba a `read` una única vez, por lo que al leer de teclado ha leído una única línea.

Vamos a arreglarlo haciendo un nuevo programa.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <err.h>

int
main(int argc, char* argv[])
{
    char    buffer[1024];
    int    nr;

    for(;;) {
        nr = read(0, buffer, sizeof buffer);
        if (nr < 0) {
            err(1, "read");
        }
        if (nr == 0) {
            break;    // EOF
        }
        if (write(1, buffer, nr) != nr) {
            err(1, "write");
        }
    }
    exit(0);
}
```

Si lo ejecutamos veremos que se comporta como *cat*:

```
unix$ readin
xxx
xxx
yyy
yyy
^D
unix$
```

Esta vez llamamos a `read` las veces que sea preciso para leer *toda* la entrada. En cada llamada pueden ocurrir tres cosas:

- Puede que leamos algunos bytes.
- Puede que no tengamos nada más que leer.
- Puede que suframos un error.

En el último caso la página de manual *read(2)* nos dice que `read` devuelve `-1` y actualiza `errno`. En dicho caso nuestro programa aborta tras imprimir un mensaje explicativo del error.

En el penúltimo caso (EOF) el programa termina con normalidad. Y, en el primer caso, el programa escribe en la salida los bytes que ha conseguido leer.

Recuerda que aunque tu desees leer *n* bytes, `read` no garantiza que pueda leerlos todos. Así pues, si deseas leer todo un fichero o un número dado de bytes, deberás llamar a `read` múltiples veces hasta que consigas leer todo lo que quieres. Un error frecuente cuando no se sabe utilizar UNIX (¡Cuando no se sabe leer!) es llamar a `read` una única vez.

2. Open, close, el terminal y la consola

Vamos a cambiar nuestro programa para que use *open(2)* y veamos que ocurre.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <err.h>

int
main(int argc, char* argv[])
{
    char    buffer[1024];
    int    fd, nr;

    fd = open("/dev/tty", O_RDWR);
    if (fd < 0) {
        err(1, "open %s", "/dev/tty");
    }
    for(;;) {
        nr = read(fd, buffer, sizeof buffer);
        if (nr < 0) {
            close(fd);
            err(1, "read");
        }
        if (nr == 0) {
            break;    // EOF
        }
        if (write(fd, buffer, nr) != nr) {
            close(fd);
            err(1, "write");
        }
    }
    if (close(fd) < 0) {
        err(1, "close");
    }
    exit(0);
}
```

Esto es lo que sucede al ejecutarlo:

```
unix$ readtty
hola
hola
caracola
caracola
^D
unix$
```

¡Lo mismo que al utilizar el descriptor `0` para leer y el descriptor `1` para escribir! Esta vez estamos utilizando como descriptor (para leer y escribir) el que nos ha devuelto `open`. Y hemos utilizado `open` para abrir el fichero `/dev/tty` para lectura/escritura. El primer parámetro de `open` es un nombre de fichero que queremos abrir y el segundo es un entero que has de interpretar como un conjunto de bits. El valor `O_RDWR` indica que queremos abrir el fichero para leer y escribir. El resultado de `open` es un entero que indica qué descriptor podemos utilizar para el nuevo fichero abierto. Por ejemplo, si en nuestro ejemplo

resulta que `open` ha devuelto `3`, tendríamos posiblemente unos descriptores como los que puedes ver en la figura 2.

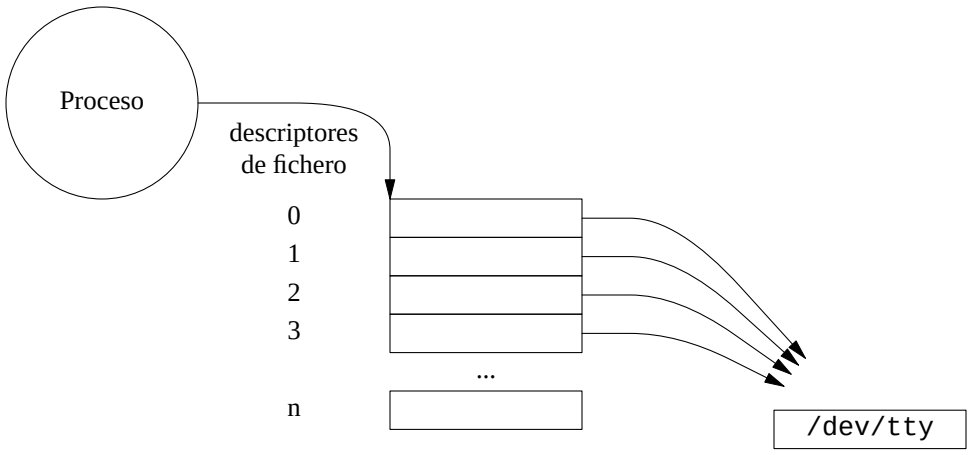

Figura 2: *Descriptores tras abrir el terminal usando* `open`.

Cuando abrimos un fichero, se espera que lo cerremos en el momento en que deje de sernos útil, cosa que se consigue llamando a *close(2)* con el descriptor de fichero que se desea cerrar. Y cuidado aquí... `close` podría fallar y hay que comprobar si ha podido hacer su trabajo o no. De no hacerlo, puede que no detectes que todas tus escrituras han alcanzado su destino. Una vez has cerrado un descriptor, podría ser que `open` en el futuro devuelva justo ese descriptor para otro fichero. Un descriptor de fichero es simplemente un índice en la tabla de ficheros abiertos del proceso.

En este punto, deberíamos preguntarnos... ¿Qué fichero es la entrada estándar? ¿Y la salida? La mayoría de las veces la entrada y la salida corresponden al fichero `/dev/tty`, que representa el *terminal* en que ejecuta nuestro programa. Es por esto que nuestro programa consigue el mismo efecto leyendo de `/dev/tty` que leyendo del descriptor `0` y escribiendo en `/dev/tty` en lugar de escribir en el descriptor `1`. Simplemente `0` y `1` ya se referían a `/dev/tty`.

Pero mira esto:

```
unix$ readin >/tmp/fich
hola
^D
unix$ cat /tmp/fich
hola
unix$
unix$ readtty >/tmp/fich
hola
hola
^D
unix$ cat /tmp/fich
unix$
```

Si utilizamos el programa que lee de la entrada estándar y escribe en la salida estándar (¡Que es lo que se espera de un programa en UNIX!) vemos que el programa se comporta de forma diferente a cuando utilizamos el programa que utiliza `/dev/tty` para leer y escribir en el. En este caso, hemos pedido al shell que ejecute `readin` enviando su salida estándar al fichero `/tmp/fich`, por lo que el programa que

escribe en `1` envía su salida correctamente a dicho fichero. En cambio, `readtty` sigue escribiendo en la ventana (en el terminal). ¡Normal!, considerando que dicho programa abre el terminal y escribe en el.

Cuando el sistema arranca, antes de que ejecute el sistema de ventanas, los programas utilizan la pantalla y el teclado. Ambos están abstraídos en el fichero `/dev/cons`, llamado así por ser la *consola* (Hace tiempo, las máquinas eran mucho mas grandes y tenían aspecto de mueble siendo la pantalla y el teclado algo con aspecto de consola).

Una vez ejecuta el sistema de ventanas (que es un programa como todo lo demás), éste se queda con la consola para poder leer y escribir y se inventa las *ventanas* como abstracción para que ejecuten nuevos programas. Igualmente, cuando un usuario remoto establece una conexión de red y se conecta para utilizar la máquina, se le asigna un *terminal* que es de nuevo una abstracción y tiene aspecto de ser un fichero similar a la consola.

En cualquier caso, `/dev/tty` es siempre el terminal que estamos utilizando. Si leemos, leemos del teclado. Cuando se trata de una ventana, el teclado naturalmente sólo escribe en esa ventana cuando la ventana tiene el *foco* (hemos dado click con el ratón en ella o algo similar).

Los ficheros que representan terminales pueden encontrarse en `/dev`:

```
unix$ ls /dev/tty*
/dev/ttyp0    /dev/ttyqa    /dev/ttys4    /dev/ttyte    /dev/ttyv8
/dev/ttyp1    /dev/ttyqb    /dev/ttys5    /dev/ttytf    /dev/ttyv9
/dev/ttyp2    /dev/ttyqc    /dev/ttys6    /dev/ttyu0    /dev/ttyva
/dev/ttyp3    /dev/ttyqd    /dev/ttys7    /dev/ttyu1    /dev/ttyvb
/dev/ttyp4    /dev/ttyqe    /dev/ttys8    /dev/ttyu2    /dev/ttyvc
/dev/ttyp5    /dev/ttyqf    /dev/ttys9    /dev/ttyu3    /dev/ttyvd
/dev/ttyp6    /dev/ttyr0    /dev/ttysa    /dev/ttyu4    /dev/ttyve
...
unix$
```

Pero para que sea trivial encontrar el fichero que corresponde al terminal que usa nuestro proceso, `/dev/tty` *siempre* corresponde al terminal que usamos. Eso sí, en cada proceso `/dev/tty` corresponderá a un fichero de terminal distinto. Esto no es un problema. Dado que UNIX sabe qué proceso está haciendo la llamada para abrir `/dev/tty`, UNIX puede dar "el cambiazo" perfectamente y hacer que se abra en realidad el terminal que está usando el proceso.

Ya ves que el **terminal de control** de un proceso (que es como se denomina) es en realidad otro de los atributos o elementos que tiene cada proceso en UNIX.

3. Ficheros abiertos

Resulta instructivo ver qué ficheros tiene abierto un proceso. Vamos a hacer un programa que abra un fichero y luego se limite a dormir durante un tiempo, para darnos tiempo a jugar con el.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <err.h>

int
main(int argc, char* argv[])
{
    int    fd;

    fd = open("sleepfd.c", O_RDONLY);
    if (fd < 0) {
        err(1, "open: %s", "sleepfd.c");
    }
    sleep(3600);
    if (close(fd) < 0) {
        err(1, "close");
    }
    exit(0);
}
```

Primero vamos a ejecutarlo, pero pidiendo al shell que no espere a que termine antes de leer mas líneas de comandos...

```
unix$ sleepfd &
[1] 93552
unix$
```

El "&" al final de una línea de comandos es sintaxis de shell (de nuevo) y hace que shell continúe leyendo líneas de comandos sin esperar a que el comando termine. El shell ha sido tan amable de decirnos que el proceso tiene el pid `93552`, pero vamos a ver qué procesos tenemos en cualquier caso.

```
unix$ ps
  448 ttys000    0:00.01 -bash
  519 ttys000    0:00.01 acme
93552 ttys002    0:00.00 sleepfd
...
unix$
```

Ahora que tenemos a `sleepfd` esperando, podemos utilizar el comando *lsof(1)* que lista ficheros abiertos. Este comando es útil tanto para ver qué procesos tienen determinado fichero abierto como para ver los ficheros abiertos de un proceso. Con la opción `-p` permite indicar el pid del proceso en que estamos interesados.

```
unix$ lsof -p 93552
COMMAND     PID USER   FD   TYPE DEVICE  SIZE/OFF    NODE NAME
sleepfd 93552 nemo  cwd    DIR    1,4      1020 7766070 /home/nemo/sot
sleepfd 93552 nemo  txt    REG    1,4      8700 7781483 /home/nemo/sot/sleepfd
sleepfd 93552 nemo  txt    REG    1,4    642448 4991292 /usr/lib/dyld
sleepfd 93552 nemo    0u   CHR   16,2  0t638511    1037 /dev/tty02
sleepfd 93552 nemo    1u   CHR   16,2  0t638511    1037 /dev/tty02
sleepfd 93552 nemo    2u   CHR   16,2  0t638511    1037 /dev/tty02
sleepfd 93552 nemo    3r   REG    1,4       398 7781474 /home/nemo/sot/sleepfd.c
unix$
```

La primera línea muestra que el proceso está usando `/home/nemo/sot`, que es un directorio (la columna `TYPE` muestra `DIR`). Mirando la columna llamada `FD`, vemos que indica `cwd`, lo que quiere decir que en realidad se trata del directorio actual (*current working directory*) del proceso.

La segunda línea muestra que también está usando `/home/nemo/sot/sleepfd`, ¡el ejecutable que hemos ejecutado!. Y la columna `TYPE` muestra `txt`, indicando que se está usando ese fichero como código (o texto) para paginar código hacia el segmento de texto, posiblemente. Igualmente, la tercera línea muestra que se está utilizando el código del enlazador dinámico `dyld` para suministrar código.

Las últimas cuatro filas son nuestro objetivo. Como puedes ver, la columna `FD` indica `0u`, `1u`, `2u` y `3r`. Además, puedes ver que `0`, `1` y `2` se refieren a `/dev/ttys002` (¡un terminal!). Estos tres son la entrada, salida y salida de error estándar de nuestro proceso. La cuarta fila indica que el descriptor `3` se refiere al fichero `sleepfd.c`, que es el fichero que nuestro programa ha abierto.

Para no dejar programas danzando inútilmente, vamos a matar nuestro proceso...

```
unix$ kill 93552
[1]+  Terminated: 15          sleepfd
unix$
```

El comando *kill(1)* puede utilizarse para matar procesos, basta darle el pid de los mismos. El shell, de nuevo, ha sido tan amable de informar que uno de los comandos que había dejado ejecutando ha terminado.

En el futuro, si te preguntas si se te ha olvidado en tu programa cerrar algún descriptor, podrías incluir un flag que haga que tu programa duerma al final y luego utilizar *lsof(1)* para inspeccionarlo.

4. Permisos y control de acceso

Resulta instructivo considerar los permisos y las comprobaciones que hace UNIX para intentar asegurar el acceso a ficheros. Como sabes, cada fichero tiene una *lista de control de acceso*, implementada en un único entero donde cada bit indica un permiso para el dueño, el grupo de usuarios al que pertenece el fichero o el resto del mundo.

Pues bien, esta lista se comprueba *durante open*. En ningún caso `read` o `write` comprueban los permisos del fichero en que operan. Se supone que si un proceso ha tenido permisos para abrir un fichero para escribir en el, por ejemplo, es legítimo que `write` pueda proceder para dicho fichero desde ese proceso.

Las **listas de control de acceso** (ACL en inglés) son similares a los vigilantes de seguridad en la puerta de una fiesta. En este caso el vigilante es UNIX y comprueba para cada proceso (visitante de la fiesta) si puede o no acceder a la misma (al fichero). Una alternativa a una lista de control de acceso es utilizar algo similar a una "llave" que permite la entrada. En este caso, esa "llave" suele denominarse **capability** e indica que quien la posee puede hacer algo con algún recurso (abrir una puerta en el ejemplo).

Aunque los ficheros están protegidos con una ACL, `read`, `write` y el resto de operaciones sobre un fichero abierto operan utilizando el descriptor como *capability*. Una vez tienes el descriptor abierto para

escribir puedes escribir en el fichero. Ya no es preciso volver a comprobar los permisos.

La estructura de datos a la que apunta un descriptor de fichero, que veremos más adelante, contiene un campo que registra para qué se abrió el fichero (leer, escribir, leer y escribir) y posteriormente `read` y `write` tan sólo han de comprobar si el descriptor es válido para ellos.

¡Curiosamente el uso de ficheros en UNIX combina tanto ACLs como capabilities!

5. Offsets

Hay una pregunta que seguramente te has hecho. ¿Cómo saben `read` y `write` en qué posición del fichero han de trabajar? Esto es, ¿En qué *offset* debe escribir `write`? o ¿Desde qué *offset* debe leer `read`?

La respuesta procede de `open`. Cuando se abre un fichero, el sistema mantiene la pista de en qué posición del fichero se está trabajando. Inicialmente este offset es cero. Cuando se escribe, se escribe en el offset para el fichero abierto y un efecto lateral de escribir *n* bytes es que el offset aumenta en *n* unidades. Igual sucede en `read`. Cuando se lee, se lee desde el offset que UNIX mantiene para el fichero abierto. Si se leen *n* bytes, el offset aumenta en *n*. Dicho de otro modo, los fichero se leen y escriben **secuencialmente** utilizando `read` y `write`. Si vuelves a mirar la salida de *lsof* que mostramos anteriormente, verás que una de las columnas indica cuál es el offset para cada descriptor.

Por ejemplo, considera este programa:

```
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <err.h>
#include <string.h>

int
main(int argc, char* argv[])
{
    int    fd, i, n;

    fd = open("afile", O_WRONLY|O_CREAT, 0644);
    if (fd < 0) {
        err(1, "afile");
    }
    for (i = 1; i < argc; i++) {
        n = strlen(argv[i]);
        if (write(fd, argv[i], n) != n) {
            close(fd);
            err(1, "write");
        }
    }
    if (close(fd) < 0) {
        err(1, "close");
    }
    exit(0);
}
```

El programa escribe sus argumentos en el fichero `afile`. La llamada a `open` abre el fichero `afile` para escribir en el (`O_WRONLY`). Como verás, el segundo argumento son flags en un único entero. Se utiliza un *or* para activar los bits que se desean en el entero. `O_CREAT` indica que si no existe queremos crear el

fichero. En dicho caso, cuando el fichero no existe y se crea, el tercer argumento indica qué permisos queremos para el nuevo fichero.

Vamos a ejecutarlo por primera vez tras comprobar que `afile` no existe.

```
unix$ ls -l afile
ls: afile: No such file or directory
unix$ writef aa bb cc
```

Si inspeccionamos `afile` veremos que tiene 9 bytes

```
unix$ ls -l afile
-rw-r--r--  1 nemo  staff  9 Aug 20 15:49 afile
```

y que su contenido son los argumentos de `writef` (con un fin de línea añadido tras cada uno):

```
unix$ cat afile
aa
bb
cc
```

Podemos utilizar *xd(1)* para ver los bytes y caracteres que contiene el fichero:

```
unix$ xd -b -c afile
0000000  61 61 0a 62 62 0a 63 63 0a
      0   a  a \n  b  b \n  c  c \n
0000009
```

La primera columna indica el offset en que aparecen los bytes que *xd* muestra a continuación. Los fines de línea son bytes con el valor `0xa` (`10` en decimal). Como verás, hemos hecho seis writes y se han escrito secuencialmente en el fichero. Esto quiere decir que el primer `write` ha escrito en la posición `0` del fichero: ha utilizado el offset `0`. El segundo `write` ha escrito un "`\n`" tras escribir el primer argumento, y ha escrito a partir del offset `2` en el fichero. El tercer `write` ha escrito el segundo argumento en la posición `3`. Y así sucesivamente.

Por cierto, el tamaño del fichero es `9` puesto que el mayor offset escrito ha sido el `8` y empezamos a contar en `0`. Luego el fichero tiene `9` bytes escritos en total. Más allá no hemos escrito nunca. Dicho de otro modo, el tamaño del fichero está determinado por hasta dónde hemos escrito en el fichero.

Si ejecutamos el programa una segunda vez, veremos mejor qué supone utilizar un offset.

```
unix$ writef x y
unix$ ls -l afile
-rw-r--r--  1 nemo  staff  9 Aug 20 15:49 afile
unix$ cat afile
x
y
b
cc
unix$ xd -b -c afile
0000000  78 0a 79 0a 62 0a 63 63 0a
      0   x \n  y \n  b \n  c  c \n
0000009
```

Lo primero que vemos es que ¡el fichero sigue teniendo 9 bytes!. Dicho de otro modo, aunque hemos escrito más veces, el tamaño no ha aumentado. El primer argumento se ha escrito al principio, lo que quiere decir que el primer `write` ha utilizado un offset `0` y ha escrito un byte. El segundo `write` ha escrito el fin

de línea en el offset `1`, dado que el primer write escribió un byte. Etcétera.

Otro detalle curioso es que después del texto escrito ("`x\ny\n`") puedes ver que el fichero contiene "`b\ncc\n`". Lo único que sucede es que esos bytes se escribieron la vez anterior que ejecutamos `writef` y esta vez no los hemos vuelto a escribir, luego tienen el mismo valor que tenían. Recuerda que `write` **no** inserta, `write` (re)escribe.

Y una cosa más. Ahora el fichero tiene 4 líneas, pero no obstante su tamaño sigue siendo 9 bytes. Aunque tenga más líneas, el fichero no es mas grande que antes. Que tenga más líneas se debe simplemente a que hay más "`\n`" escritos en el fichero.

Para ver estas cosas te puede resultar cómodo utilizar *wc(1)*, que cuenta líneas, palabras y caracteres en un fichero, o en la entrada si no indicas fichero alguno.

```
unix$ wc afile
       4       4       9 afile
unix$
```

¿Dónde está el offset? La figura 3 muestra qué aspecto tiene la estructura de datos que usa UNIX.

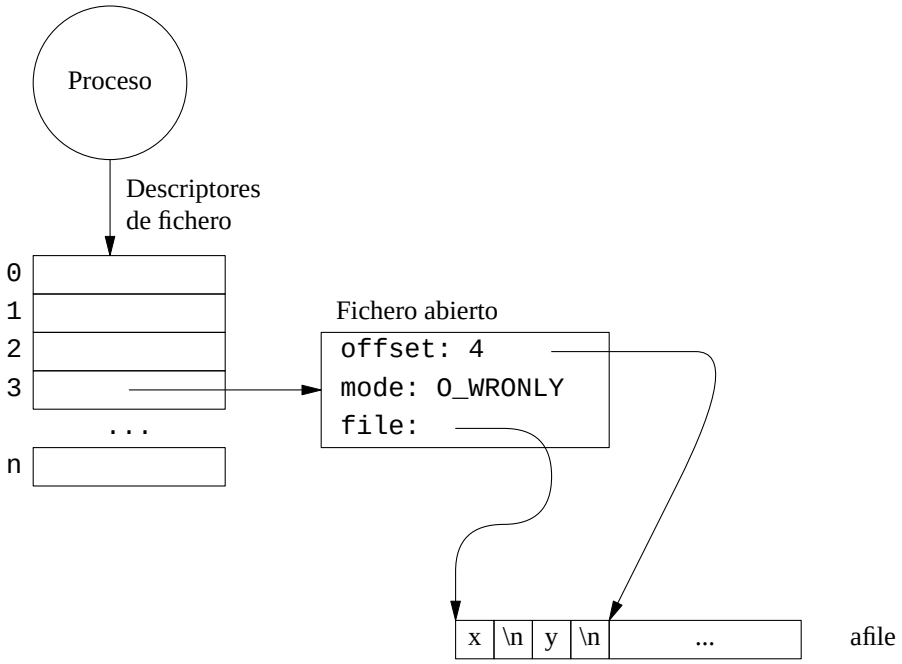

Figura 3: *El offset para lecturas y escrituras se guarda fuera del descriptor de fichero, en una entrada en la tabla de ficheros abiertos.*

En cada descriptor de fichero hay un puntero hacia un record (como siempre). En dicho record, llamado *fichero abierto* por UNIX y almacenado en una *tabla de ficheros abiertos* se guarda:

- El offset en que hay que leer/escribir.
- El modo en que se abrió el fichero (lectura, escritura, o lectura/escritura).
- El puntero hacia la estructura de datos que representa el fichero en UNIX.

En la figura se ve qué valor tenía el contenido de `afile` y el `offset` del fichero abierto justo al final de ejecutar por segunda vez `afile`, antes de llamar a `close`.

Si queremos que cada vez que ejecutemos `writef` el fichero `afile` quede justo con lo que hemos escrito, la solución es eliminar el contenido de `afile` cuando el fichero ya existe, durante `open`. Esto es, si el fichero no existe lo creamos y si existe lo *truncamos* a 0-bytes. El flag `O_TRUNC` de `open` consigue este efecto. Basta utilizar

```
fd = open("afile", O_WRONLY|O_TRUNC|O_CREAT, 0644);
```

en lugar de la llamada a `open` que hacíamos antes. La próxima vez que ejecutemos `writef` no quedarán restos del contenido que `afile` pudiera tener antes si es que existía. Aunque claro, si has leído *open(2)* ya lo sabías.

Existe otro flag que podemos utilizar con *open(2)* que afecta al offset que utiliza `write`. Se trata del flag `O_APPEND`. Si indicamos `O_APPEND` en `open`, las llamadas a `write` ignoran el offset del fichero y escriben siempre al final del mismo. En realidad, se escribe justo en la posición indicada por el tamaño del fichero. Pero cuidado aquí si el fichero es un fichero compartido en red: cada UNIX que tiene abierto el fichero podría tener su propia idea del tamaño del mismo (por ejemplo si un UNIX acaba justo de hacer crecer el fichero pero otro no se ha dado cuenta todavía).

6. Ajustando el offset

Para evitar errores utilizando `read` y `write` basta con que pienses que UNIX es muy obediente y hace justo lo que se supone que cada llama hace. No hay magia. Si recuerdas lo que hemos visto, podrás entender lo que ocurre. ¡Vamos a comprobarlo!

Hay otra llamada, *lseek(2)*, que permite cambiar el offset de un descriptor de fichero. Bueno, en realidad, queremos decir "*el offset de un fichero abierto al que apunta un descriptor*". Se le suministra el descriptor de fichero, cuantos bytes queremos mover el offset y desde donde contamos. Por ejemplo,

- `lseek(fd, 10, SEEK_SET)` fija el offset a 10 en `fd`.
- `lseek(fd, 10, SEEK_CUR)` añade 10 al offset de `fd`.
- `lseek(fd, 10, SEEK_END)` fija el offset 10 bytes contando desde el final del fichero hacia atrás en `fd`.

Las constantes `SEEK_SET`, `SEEK_CUR` y `SEEK_END` son simplemente `0`, `1` y `2`. De hecho, dado que desde el comienzo de UNIX han tenido esos valores, es muy poco probable que cambien nunca de valor.

Entendido esto, fíjate en este programa:

```
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <err.h>

int
main(int argc, char* argv[])
{
    int    fd;

    fd = open("afile", O_WRONLY|O_TRUNC|O_CREAT);
    if (fd < 0) {
        err(1, "open: afile");
    }
    lseek(fd, 32, 0);
    if (write(fd, "xx\n", 3) != 3) {
        close(fd);
        err(1, "write");
    }
    if (close(fd) < 0) {
        err(1, "close");
    }
    exit(0);
}
```

Ejecutándolo podemos ver lo que sucede con el nuevo `afile` en este caso.

```
unix$ seekwrite
unix$ ls -l afile
-rw-r--r--  1 nemo  staff  35 Aug 20 16:32 afile
unix$ cat afile
xx
unix$ xd -b -c afile
0000000  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
      0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000010  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
     10  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000020  78 78 0a
     20   x  x \n
0000023
```

¡Curioso!, ¿No? Resulta que *ls* dice que el fichero tiene 35 bytes. Pero sólo hemos escrito 3 bytes. Veamos...

- `lseek` ha cambiado el offset a 32, contando desde el comienzo del fichero.
- `write` ha escrito tres bytes en dicho offset.

Luego en total, el byte más alto que hemos escrito es el 34 (contando desde 0), lo que quiere decir que mirando los 35 primeros bytes tenemos el contenido del fichero que hemos escrito alguna vez. Además, puedes ver que los bytes que no hemos escrito nunca están a cero. Eso es cortesía de UNIX.

El otro detalle curioso es que cuando hemos hecho un *cat* del fichero sólo hemos visto una línea con `xx`, que era lo que cabría esperar. Pero en realidad, sí que `cat` ha escrito un total de 35 bytes en su salida:

```
unix$ cat afile >/tmp/out
unix$ ls -l /tmp/out
-rw-r--r--  1 nemo  wheel  35 Aug 20 16:37 /tmp/out
unix$
```

Lo que ocurre es que cuando *cat* escribe un byte a cero en su salida estándar, y esta es el terminal, el terminal no sabe cómo mostrar dicho byte (¿Cómo dibujar un símbolo para el byte nulo?) y no lo escribe en absoluto. Es normal, `/dev/tty` sólo espera que escribas texto en el teclado y que muestres texto en la pantalla. No espera efectos especiales.

Utilizar `lseek` como hemos hecho es muy habitual. Por ejemplo, para crear un fichero con 1GiB basta con que hagas un `lseek` a la posición `1024*1024*1024-1` y escribas un byte. El tamaño del nuevo fichero será justo de 1GiB. Lo que es más, es muy posible que UNIX no asigne espacio en disco para el nuevo fichero salvo para el último *bloque* que use el fichero (que es dónde has escrito el byte). Todos los trozos anteriores del fichero están sin escribir y, si se leen, UNIX sabe que se leen con todos los bytes a cero. Siendo esto así, ¿Para qué guardar los ceros en el disco si UNIX sabe que están a cero? A estos ficheros se les llama *ficheros con huecos*. ¡Puedes tener ficheros más grandes que el tamaño del disco en que los guardas!

Otro uso de `lseek` es para obtener el offset. Basta con cambiar el offset a 0 bytes contando a partir del offset actual, y utilizar el valor que retorna `lseek` (que es el nuevo offset). Por ejemplo:

```
off = lseek(fd, 0, 1);
```

7. Crear y borrar

Ya hemos creado ficheros utilizando el flag `O_CREAT` de `open`. Otra forma es utilizar *creat(2)*. Llamar a

```
fd = creat(path, mode);
```

es lo mismo que llamar a

```
fd = open(path, O_CREAT|OTRUNC|O_WRONLY, mode);
```

así que en realidad ya sabes utilizar `creat`.

No es posible crear directorios utilizando `creat`. Hay que utilizar *mkdir(2)*. Su uso es similar al de `creat`, salvo porque `mkdir` no devuelve ningún descriptor. Tan sólo devuelve `-1` si falla y `0` en caso contrario:

```
if (mkdir("/tmp/mydir", 0755) < 0) {
    ...mkdir ha fallado...
}
```

Los permisos (o como UNIX lo denomina, el *modo*) con que se crean los ficheros y directorios no sólo dependen de los permisos que indiques en la llamada a `open`, `creat` o `mkdir`. Por seguridad, cada proceso tiene un atributo denominado *umask*. El *umask* es la *máscara de creación de ficheros* y actúa como una máscara para evitar que permisos no deseados se den a ficheros que crea un proceso. La llamada al sistema es *umask(2)* y el comando que utiliza dicha llamada y podemos utilizar en el shell es *umask(1)*.

Para cambiar la *umask* desde C podemos utilizar código como

```
umask(077);
```

Con esta llamada ponemos la máscara a 0077, lo que hace que en ningún caso se den permisos de lectura, escritura o ejecución para el grupo de usuarios o para el resto del mundo. Si con esta máscara utilizamos 0755 como modo en `creat`, los permisos del nuevo fichero serán en realidad 0700. Aunque no hemos

usado el valor que devuelve `umask`, la llamada devuelve la máscara anterior, por si queremos volverla a dejar como estaba.

Normalmente, la máscara es 0022, lo que hace que ni el grupo ni el resto del mundo pueda escribir los ficheros o directorios que creamos. ¿Puedes ahora explicar lo que sucede en esta sesión de shell?

```
unix$ umask
0022
unix$ touch a
unix$ ls -l a
-rw-r--r--  1 nemo  staff  0 Aug 20 21:32 a
unix$ umask 077
unix$ umask
0077
unix$ rm a
unix$ touch a
unix$ ls -l a
-rw-------  1 nemo  staff  0 Aug 20 21:32 a
unix$
```

Cuando un proceso crea un fichero hace en realidad dos cosas:

- crear el fichero
- darle un nombre en un directorio

Ambas cosas suceden dentro de la llamada. Esto hace que resulte natural que no exista una llamada para borrar un fichero. Para borrar un fichero lo que se hace es eliminar el nombre del fichero. Si nadie está utilizando el fichero y no hay ningún otro nombre para el fichero, este se borra.

Por ejemplo, este programa es una versión simplificada de *rm(1)*.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <err.h>

int
main(int argc, char* argv[])
{
    int i, sts;

    sts = 0;
    if (argc == 1) {
        fprintf(stderr, "usage: %s file...\n", argv[0]);
        exit(1);
    }
    for (i = 1; i < argc; i++) {
        if (unlink(argv[i]) < 0) {
            warn("%s: unlink", argv[i]);
            sts = 1;
        }
    }
    exit(sts);
}
```

Podemos utilizarlo para borrar ficheros:

```
unix$ touch /tmp/a /tmp/b /tmp/c
unix$ ls -l /tmp/?
-rw-r--r--  1 nemo  wheel  0 Aug 20 18:57 /tmp/a
-rw-r--r--  1 nemo  wheel  0 Aug 20 18:57 /tmp/b
-rw-r--r--  1 nemo  wheel  0 Aug 20 18:57 /tmp/c
unix$ chmod -w /tmp/b
unix$ ls -l /tmp/b
-r--r--r--  1 nemo  wheel  0 Aug 20 18:57 /tmp/b
unix$ rem /tmp/[abc]
unix$ ls -l /tmp/?
ls: /tmp/?: No such file or directory
unix$
```

Hemos utilizado `/tmp/?` para que el shell escriba por nosotros en la línea de comandos todos los nombres de fichero que están en `/tmp` y tienen como nombre un sólo carácter. También hemos utilizado `/tmp/[abc]` para que el shell escriba por nosotros los nombres de ficheros en `/tmp` que sean `a`, `b` o `c`. Si esto te resulta confuso, piensa que hemos usando siempre los argumentos que hemos dado a `touch` en lugar de `/tmp/[abc]` o de `/tmp/?`. Más adelante explicaremos estas expresiones con más detalle.

Como verás, que no tengas permiso de escritura en un fichero no quiere decir que no puedas borrarlo. Pero mira esto:

```
unix$ mkdir /tmp/d
unix$ touch /tmp/d/a
unix$ chmod -w /tmp/d
unix$ 8.rem /tmp/d/a
rem: /tmp/d/a: unlink: Permission denied
unix$ chmod +w /tmp/d
unix$ rem /tmp/d/a
unix$
```

Como indica la página de manual *unlink(2)*, borrar un fichero requiere poder escribir el directorio en que está. Cuando tengas dudas respecto a permisos, consulta el manual.

Otra cosa curiosa sucede si intentamos borrar `/tmp/d`.

```
unix$ rem /tmp/d
rem: /tmp/d: unlink: Operation not permitted
unix$
```

La llamada `unlink` no sabe borrar directorios.

Para borrar un directorio hay que utilizar *rmdir(2)*, del mismo modo que para crear directorios hay que utilizar *mkdir(2)* y no podemos utilizar *creat(2)*.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <err.h>

int
main(int argc, char* argv[])
{
    int i, sts;

    sts = 0;
    if (argc == 1) {
        fprintf(stderr, "usage: %s dir...\n", argv[0]);
        exit(1);
    }
    for (i = 1; i < argc; i++) {
        if (rmdir(argv[i]) < 0) {
            warn("%s: rmdir", argv[i]);
            sts = 1;
        }
    }
    exit(sts);
}
```

Y ahora podemos...

```
unix$ remdir /tmp/d
unix$
```

Naturalmente, no podemos utilizar `rmdir` para borrar ficheros:

```
unix$ touch /tmp/a
unix$ remdir /tmp/a
remdir: /tmp/a: rmdir: Not a directory
unix$
```

¡Y tampoco para borrar directorios que no están vacíos! (sin contar ni "`.`" ni "`..`"). Si `rmdir` pudiese borrar directorios no vacíos nos divertiríamos mucho ejecutando

```
unix$ rm /
```

8. Enlaces

Dado un fichero que existe, podemos darle un nuevo nombre dentro de la misma partición o del mismo sistema de ficheros. Esto se hace con la llamada *link(2)*. Este programa es similar al comando *ln(1)*, que establece un nuevo nombre para un fichero:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <err.h>

int
main(int argc, char* argv[])
{

    if (argc != 3) {
        fprintf(stderr, "usage: %s old new\n", argv[0]);
        exit(1);
    }
    if (link(argv[1], argv[2]) < 0) {
        err(1, "link %s", argv[1]);
    }
    exit(0);
}
```

Vamos a verlo despacio y paso a paso utilizando un par de ficheros. Primero creamos un fichero `afile`:

```
unix$ echo hola >afile
unix$ ls -i afile
7782892 afile
unix$ cat afile
hola
```

Hemos utilizado `ls` con la opción `-i` para que nos de el número que usa UNIX para identificar el fichero dentro su partición o sistema de ficheros (UNIX lo llama *i-nodo*).

Ahora podemos ejecutar nuestro programa `lnk` o el comando `ln` (funcionan igual en este caso) para dar un nuevo nombre para `afile`:

```
unix$ lnk afile another
unix$ ln afile another
ln: another: File exists
unix$
```

La segunda vez, el fichero `another` ya existe por lo que no se puede utilizar ese nombre como un nuevo nombre. En cualquier caso, `lnk` ha llamado a `link` haciendo que `another` sea otro nombre para `afile`. ¡Ambos ficheros son el mismo!

Aunque en este caso ambos nombres están dentro del mismo directorio, es posible crearlos en directorios distintos. Pero sigamos explorando los enlaces y el uso que hace UNIX de los nombres. Primero, podemos comprobar que el fichero es en realidad el mismo:

```
unix$ ls -i another
7782892 another
```

El número de i-nodo sólo significa algo dentro de la misma partición en el mismo disco, pero ese es el caso por lo que si el número coincide entonces el fichero es el mismo.

Veámoslo mirando y cambiando uno de los ficheros:

```
unix$ cat another
hola
unix$ echo adios >afile
unix$ cat another
adios
```

Tras cambiar `afile`, resulta que `another` ha cambiado del mismo modo. ¡Naturalmente!, son el mismo fichero.

Si borramos `afile`, todavía sigue existiendo el fichero

```
unix$ rm afile
unix$ cat another
adios
```

puesto que aún tiene otro nombre. Si borramos el único nombre que le queda al fichero

```
unix$ rm another
```

UNIX borra realmente el fichero.

Para saber cuántos nombres tiene un fichero, UNIX guarda en la estructura de datos que lo implementa (llamada *i-nodo*) un contador que cuenta cuántos nombres tiene. Se lo suele llamar contador de referencia. Podemos verlo utilizando `ls`. Fíjate en el número de la segunda columna cada vez que llamamos a `ls`:

```
unix$ touch afile
unix$ ln afile another
unix$ ls -l afile another
-rw-r--r--  2 nemo  staff  0 Aug 20 19:20 afile
-rw-r--r--  2 nemo  staff  0 Aug 20 19:20 another
unix$ rm another
unix$ ls -l afile
-rw-r--r--  1 nemo  staff  0 Aug 20 19:20 afile
```

¿Recuerdas que "`.`" es otro nombre para el directorio actual? Observa la salida de este comando:

```
unix$ ls -ld .
drwxr-xr-x  2 nemo  staff  1258 Aug 20 19:21 .
```

El flag `-d` de `ls` hace que si listamos un nombre de directorio, `ls` liste el fichero del directorio y no los ficheros que contiene. ¿Puedes explicar por qué "`.`" tiene 2 enlaces?

Podemos jugar más...

```
unix$ mkdir /tmp/d
unix$ mkdir /tmp/d/1 /tmp/d/2 /tmp/d3
unix$ ls -ld /tmp/d
drwxr-xr-x  4 nemo  wheel  136 Aug 20 19:26 /tmp/d
```

¿Aún no ves por qué "`/tmp/d`" tiene 4 enlaces? Aquí tienes una pista...

```
unix$ ls -ld /tmp/d/1/..
drwxr-xr-x  4 nemo  wheel  136 Aug 20 19:26 /tmp/d/1/..
```

9. Lectura de directorios

Los directorios en UNIX son ficheros que contienen una tabla de ficheros pertenecientes al directorio. En cada entrada de la tabla hay dos cosas:

- Un nombre de fichero
- Un número (de *i-nodo*) que identifica al fichero.

Cuando se crea un fichero con `creat`, `open`, o `mkdir` se añade automáticamente la entrada de directorio para el fichero o directorio que se ha creado. Recordarás que `link` (o `ln`) puede usarse para establecer nuevos enlaces para ficheros que existen (nuevas entradas de directorio con el número de un fichero existente) y que `unlink` o `rmdir` pueden utilizarse para eliminar nombres.

Hace tiempo UNIX permitía utilizar `read` y `write` para leer y escribir directorios. No obstante, esto causó tantos problemas cada vez un usuario escribía incorrectamente la tabla que desde hace tiempo no se puede utilizar `read` o `write` para leer una tabla de directorio.

En lugar de *read(2)*, para directorios tenemos la llamada al sistema *getdirentries(2)*. No obstante, es mucho más fácil utilizar las funciones *opendir(3)*, *readdir(3)* y *closedir(3)* que tenemos en la librería de C. No sólo es más fácil. Es más portable. Piensa que el formato exacto de una entrada de directorio depende del tipo de sistema de ficheros que utilices (del programa que implementa los ficheros en esa partición o donde quiera que estén los ficheros, y que forma parte del kernel).

Aquí tienes un ejemplo de cómo listar un directorio en C.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/dir.h>
#include <dirent.h>
#include <err.h>

int
main(int argc, char* argv[])
{
    DIR *d;
    struct dirent *de;

    d = opendir(".");
    if(d == NULL) {
        err(1, "opendir");
    }
    while((de = readdir(d)) != NULL) {
        printf("%s\n", de->d_name);
    }
    if (closedir(d) < 0) {
        err(1, "closedir");
    }
    exit(0);
}
```

Este programa lista el contenido del directorio actual. Por ejemplo,

```
unix$ lsdot
.
..
lsdot.c
lsdot
ch03e.w
unix$
```

Como verás, aparecen tanto "`.`" como "`..`".

Cada llamada a `readdir` devuelve una estructura de tipo `struct dirent`. Dicha estructura no debes liberarla, según indica el manual, y si llamas de nuevo a `readdir` la estructura antigua se sobreescribe con la nueva muy posiblemente.

Esta estructura cambia mucho de un tipo de UNIX a otro, aunque en la mayoría las entradas de directorio son similares. Por ejemplo, esta es la que utiliza Linux:

```
struct dirent {
    ino_t          d_ino;       /* inode number */
    off_t          d_off;       /* not an offset; see NOTES */
    unsigned short d_reclen;    /* length of this record */
    unsigned char  d_type;      /* type of file; not supported
                                   by all filesystem types */
    char           d_name[256]; /* filename */
};
```

Normalmente, siempre tienes el campo `d_name`. En Linux y sistemas BSD tienes `d_type` también. El resto de campos suelen variar de nombre y tal vez no estén en tu UNIX. Si utilizas algo más que `d_name`, seguramente tu código deje de ser portable a otros tipos de UNIX.

Cuando tienes `d_type`, este campo vale `DT_DIR` para directorios, `DT_REG` para ficheros (ficheros regulares o normales) y otros valores para otros tipos de fichero que veremos más adelante.

Naturalmente, nunca se escribe un directorio, este se actualiza creando, borrando y renombrando los ficheros que contiene. Ya conoces todo lo necesario, salvo por cómo renombrar ficheros. La llamada al sistema en cuestión es *rename(2)*.

Este programa renombra un fichero, similar al uso más sencillo del comando *mv(1)*.

```
#include <stdio.h>
#include <stdlib.h>
#include <err.h>

int
main(int argc, char* argv[])
{

    if (argc != 3) {
        fprintf(stderr, "usage: %s old new\n", argv[0]);
        exit(1);
    }
    if (rename(argv[1], argv[2]) < 0) {
        err(1, "rename %s", argv[1]);
    }
    exit(0);
}
```

Podemos usarlo para mover un fichero a notro nombre:

```
unix$ echo hola >a
unix$ cat a
hola
unix$ mvf a b
unix$ ls -l a b
ls: a: No such file or directory
-rw-------  1 nemo  staff  5 Aug 20 22:13 b
unix$ cat b
hola
unix$
```

Has de tener en cuenta que si el fichero de destino existe, se borra (su nombre) durante el *rename*. Además, es un error hacer un *rename* hacia un directorio no vacío. Y otra cosa, si recuerdas el comando *mv(1)*, verás que

```
unix$ mv a /tmp
```

mueve el fichero `a` a `/tmp/a`. No obstante,

```
unix$ mvf a /tmp
mvf: rename a: Permission denied
```

nuestro programa no es tan listo e intenta hacer que `a` tenga como nombre `/tmp` (que es un directorio que ya existe).

10. Globbing

Dado que gran parte de las "palabras" que escribimos en una línea de comandos corresponden a nombres de fichero, el shell da facilidades para expresar de un modo compacto nombres de fichero que tienen cierto aspecto. Dicho de otro modo, el shell permite utilizar expresiones que *generan* nombres de fichero o que *encajan* con nombres de fichero. A esta herramienta se la conoce como *globbing*.

Ya sabes que en general una buena forma de saber cómo funcionan las cosas es experimentar con ellas, así que vamos a crear un directorio con ciertos ficheros dentro para experimentar, tal y como harías tu.

```
unix$ mkdir -p /tmp/d/d2/d3
unix$ touch /tmp/d/a /tmp/d/ab /tmp/d/abc
unix$ touch /tmp/d/d2/j /tmp/d/d2/k
unix$ touch /tmp/d/d2/d3/x  /tmp/d/d2/d3/y
unix$
```

El flag `-p` de `mkdir` hace que se cree el directorio indicado como argumento y los directorios padre si es que no existen.

Podemos utilizar el comando *du(1)* (*disk usage*) para que liste el árbol de ficheros que hemos creado. Este comando en realidad lista cuándo disco consumen ficheros y directorios, pero es una forma práctica de listar árboles de ficheros (¡Más práctica que utilizando `ls`!).

```
unix$ du -a /tmp/d
0    /tmp/d/1
0    /tmp/d/2
0    /tmp/d/a
0    /tmp/d/ab
0    /tmp/d/abc
0    /tmp/d/d2/d3/x
0    /tmp/d/d2/d3/y
0    /tmp/d/d2/d3
0    /tmp/d/d2/j
0    /tmp/d/d2/k
0    /tmp/d/d2
0    /tmp/d
```

El flag `-a` de `du` hace que liste no sólo los directorios, sino también los ficheros.

Pero continuemos. Cuando el shell ve que un nombre en la línea de comandos contiene ciertos caracteres especiales, intenta encontrar paths de fichero que encajan con dicho nombre y, si los encuentra, cambia ese nombre los pos paths, separados por espacio. En realidad, deberíamos llamar *expresión* a lo que estamos llamando "nombre".

Por ejemplo, en este comando

```
unix$ echo /tmp/d/*
/tmp/d/1 /tmp/d/2 /tmp/d/a /tmp/d/ab /tmp/d/abc /tmp/d/d2
```

el shell ha tomado "`/tmp/d/*`" y, como puedes ver por la salida de `echo`, lo ha cambiado por los nombres de fichero en "`/tmp/d`".

Pero mira este otro comando:

```
unix$ echo /tmp/d/a*
/tmp/d/a /tmp/d/ab /tmp/d/abc
```

Aquí el shell ha cambiado "`/tmp/d/a*`" por los nombres de fichero en `/tmp/d` que comienzan por "`a`".

Veamos otro más...

```
unix$ echo /*/ls
/bin/ls
```

El shell ha buscado en "`/`" cualquier fichero que sea un directorio y contenga un fichero llamado "`ls`".

Las expresiones de globbing son fáciles de entender:

- El shell las recorre componente a componente mirando cada una de las subexpresiones que corresponden a nombres de directorio en el path.
- Naturalmente, "`/`" separa unos componentes de otros.
- En cada componente intenta encontrar todos los ficheros cuyo nombre encaja en la subexpresión para dicho componente.
- En cada componente podemos tener caracteres normales o alguna de estas expresiones:
  - "`*`" encaja con cualquier string, incluso con el string vacío.
  - "`[...]`" encaja con cualquier carácter contenido en los corchetes. Aquí es posible indicar rangos de caracteres como en "`[a-z]`" (las letras de la "`a`" a la "`z`").
  - "`?`" encaja con un sólo carácter.

Por ejemplo,

```
unix$ cd /tmp/d
unix$ ls
1    2    a    ab    abc    d2
unix$ echo [12]*
1 2
unix$ echo [ab]*
a ab abc
unix$
```

La primera expresión quiere decir: "un `1` o un `2` y luego cualquier cosa" La segunda quiere decir: "una `a` o una `b` y luego cualquier cosa".

Otro ejemplo:

```
unix$ ls /tmp/*/d*/?
/tmp/d/d2/j    /tmp/d/d2/k
```

Esto es, dentro de "`/tmp`", cualquier nombre de directorio que tenga dentro un directorio que comience por "`d`" y tenga dentro un fichero cuyo nombre sea un sólo carácter y nada más.

Quizá el uso más común sea en comandos como

```
unix$ ls *.[ch]
```

para listar los fuentes en C (ficheros cuyo nombre es cualquier cosa, luego un "`.`" y luego o bien una "`c`" o una "`h`", y como

```
unix$ rm *.o
```

para borrar todos los ficheros objeto (cualquier nombre terminado en "`.o`").

Recuerda que ninguno de estos comandos sabe nada respecto a globbing, ni entienden qué quiere decir "`*`" ni lo ven siquiera. Es el shell cuando analiza la línea de comandos el que detecta que hay una expresión que contiene caracteres de globbing, y cambia dicha expresión por los nombres de los ficheros que encajan en la expresión. Los comandos simplemente reciben los argumentos con los paths y se limitan a hacer su trabajo.

En ocasiones resultará útil hacer que el shell no haga globbing, como en esta sesión:

```
unix$ touch '*'
```

Hemos creado un fichero llamado "`*`". Esta vez a sabiendas, pero podría haber sido por error. Para hacerlo simplemente utilizamos las comillas simples para hacer que el shell tome *literalmente* y sin cambiar nada lo que incluyen las comillas (y como una sóla palabra).

Y ahora tenemos el dilema.

```
unix$ ls
*    1    2    a    ab    abc    d2
unix$ echo *
* 1 2 a ab abc d2
```

Si utilizamos

```
unix$ rm *
```

para eliminar "`*`", ¡borraremos todos los ficheros! y no es lo que queremos. Pero podemos hacer esto:

```
unix$ rm '*'
unix$ ls
1    2    a    ab    abc    d2
```

11. Metadatos

Los ficheros, como ya sabes en este punto, no sólo tienen datos. Tienen además atributos, o datos sobre los datos, o **metadatos**. Por ejemplo, el *uid* del dueño del fichero, el *gid* del grupo al que pertenece, los permisos, el contador de referencias, etc.

En UNIX, todos estos metadatos se guardan en la estructura de datos que representa al fichero, que es un record que UNIX llama *i-nodo*. La llamada al sistema *stat(2)* sirve para recuperar los metadatos de un fichero. El siguiente programa mejora nuestro programa para listar directorios y muestra distintos atributos de cada fichero contenido en los directorios cuyo nombre se indica como argumento (el directorio actual si no se indica ninguno).

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/dir.h>
#include <sys/stat.h>
#include <dirent.h>
#include <string.h>
#include <time.h>
#include <err.h>
typedef unsigned long long uvlong_t;
static int
list(char *fname)
{
    struct stat st;

    if (stat(fname, &st) < 0)
        return -1;
    printf("%s:\n\t", fname);
    if ((st.st_mode & S_IFMT) == S_IFDIR)
        printf("dir");
    else if ((st.st_mode & S_IFMT) == S_IFREG)
        printf("file");
    else
        printf("type %xu", st.st_mode&S_IFMT);
    printf(" perms %o", st.st_mode & 0777);
    printf(" sz %llu", (uvlong_t) st.st_size);
    printf(" uid %d gid %d", st.st_uid, st.st_gid);
    printf("\n\tlinks %d", (int)st.st_nlink);
    printf(" dev %d ino %llu\n\t", (int)st.st_dev, (uvlong_t) st.st_ino);
    printf("mtime %llus = %s", (uvlong_t)st.st_mtime, ctime(&st.st_mtime));
    return 0;
}
```

El propósito de `list` es listar los atributos de fichero cuyo path se indica en `fname`. La llamada a `stat` rellena una estructura `st` con dichos atributos. Igual que sucedía con la lectura de directorios, el contenido de esta estructura varía de unos sistemas UNIX a otros. Así que hay que tener cuidado para que el código sea portable. Los campos que utilizamos en nuestro programa suelen estar disponibles en todos los UNIX. En el caso de Linux, la estructura está definida así:

```
struct stat {
    dev_t     st_dev;     /* ID of device containing file */
    ino_t     st_ino;     /* inode number */
    mode_t    st_mode;    /* protection */
    nlink_t   st_nlink;   /* number of hard links */
    uid_t     st_uid;     /* user ID of owner */
    gid_t     st_gid;     /* group ID of owner */
    dev_t     st_rdev;    /* device ID (if special file) */
    off_t     st_size;    /* total size, in bytes */
    blksize_t st_blksize; /* blocksize for filesystem I/O */
    blkcnt_t  st_blocks;  /* number of 512B blocks allocated */
    time_t    st_atime;   /* time of last access */
    time_t    st_mtime;   /* time of last modification */
    time_t    st_ctime;   /* time of last status change */
};
```

El campo `st_mode` contiene tanto los permisos como otros bits que indican el tipo de fichero. La constante `S_IFMT` indica qué bits contienen el tipo de fichero. Además, tenemos constantes `S_IFDIR`, `S_IFREG`, etc. para comprobar qué tipo de fichero tenemos. Observa que utilizamos "`&`" dado que el campo `st_mode` tiene más bits y no es posible utilizar sólo "`==`".

Otros campos como `st_size`, `st_uid` y `st_gid` debieran ser obvios a estas alturas. El campo `st_nlink` indica cuántas referencias (nombres) tiene el fichero.

Para ver si dos ficheros son el mismo habría que comprobar si tanto `st_dev` como `st_ino` coinciden. El primero indica en qué dispositivo se encuentra el fichero y el segundo indica qué número de fichero dentro de dicho dispositivo (qué número de *i-nodo*) tenemos.

Los campos `st_mtime`, `st_atime` y `st_ctime` contienen las fechas de la última modificación del fichero, del último acceso a los datos del mismo y de la última vez que se cambiaron los atributos (o de cuándo se creó el fichero si no se han cambiado). Estas fechas están codificadas como un número de segundos desde una fecha data, pero podemos utilizar *ctime(3)* para convertir dicho número a un string con la fecha en un formato que un humano pueda leer. En breve veremos la gestión del tiempo en UNIX más despacio.

Lamentablemente, hoy en día, UNIX utiliza tipos como `ino_t`, `dev_t`, `size_t` y otros muchos en lugar de `int`, `long`, etc. Eso quiere decir que tendremos problemas para utilizar los formatos de `printf` con cada uno de ellos. Una solución portable para imprimir dichos valores es la que puedes ver en el código: convertimos el entero en cuestión a otro entero de mayor o igual tamaño, en nuestro caso a `unsigned long long`, y utilizamos el formato para dicho entero (en nuestro caso "`%llu`").

Habría sido mejor utilizar otro tamaño de entero y usar simplemente "`int`" y "`%d`" dado que hoy día la memoria es barata. Pero esto es tan sólo una opinión.

Para que puedas ver el programa en su conjunto y repasar cómo se leían directorios, comprobaban argumentos y otros detalles, mostramos a continuación el resto del fichero fuente que contiene nuestro programa. Observa cómo las funciones se toman molestias en devolver una indicación de error si tienen problemas y cómo se comprueban los errores e imprimen mensajes de error. En aquellos casos en que es posible continuar el trabajo tras informar de un error, el código hace justo eso. Además, se cierran los ficheros que se abren, tengamos errores o no. Pero antes de verlo, esta es una ejecución del programa:

```
unix$ ll
./ll:
    file perms 755 sz 9092 uid 501 gid 20
    links 1 dev 16777220 ino 7791046
    mtime 1471771999s = Sun Aug 21 11:33:19 2016
./guide:
    file perms 644 sz 1396 uid 501 gid 20
    links 1 dev 16777220 ino 7766071
    mtime 1471700478s = Sat Aug 20 15:41:18 2016
./ll.c:
    file perms 644 sz 1731 uid 501 gid 20
    links 1 dev 16777220 ino 7788145
    mtime 1471771995s = Sun Aug 21 11:33:15 2016
./writef.c:
    file perms 644 sz 512 uid 501 gid 20
    links 1 dev 16777220 ino 7782848
    mtime 1471700968s = Sat Aug 20 15:49:28 2016
```

Y este es el código que falta:

```
static int
ldir(char *dir)
{
    DIR *d;
    struct dirent *de;
    char path[1024];
    int n, rc;

    d = opendir(dir);
    if(d == NULL) {
        warn("opendir: %s", dir);
        return -1;
    }
    rc = 0;
    while((de = readdir(d)) != NULL) {
        n = snprintf(path, sizeof path, "%s/%s", dir, de->d_name);
        if (n >= sizeof path-1) { // -1 for '\0'
            warn("path %s/%s too long", dir, de->d_name);
            rc = -1;
        } else if (list(path) < 0) {
            warn("list: %s", path);
            rc = -1;
        }
    }
    if (closedir(d) < 0) {
        warn("closedir: %s", dir);
        return -1;
    }
    return rc;
}
```

```
int
main(int argc, char* argv[])
{
    int sts, i;

    sts = 0;
    if (argc == 1) {
        if (ldir(".") < 0) {
            sts = 1;
        }
    } else {
        for (i = 1; i < argc; i++) {
            if (ldir(argv[i]) < 0) {
                sts = 1;
            }
        }
    }
    exit(sts);
}
```

Por cierto, en ocasiones tendrás un descriptor de fichero del que desees obtener los metadatos, en lugar de tener el path para dicho fichero. En ese caso puedes utilizar *fstat(2)* en lugar de *stat(2)*, que recibe un descriptor de fichero en lugar de un path. En muchas otras llamadas se sigue el mismo convenio y dispones de funciones que comienzan por "`f`" para utilizar un descriptor en lugar de un path para identificar el fichero con que quieres trabajar. ¡El manual es tu amigo!

12. El tiempo

UNIX mantiene su propia idea de la fecha y hora. Por un lado, el hardware habitualmente dispone de un reloj que está continuamente alimentado por baterías y que se ocupa de mantener la noción del tiempo.

Naturalmente, este reloj es un contador que se incrementa cada unidad de tiempo. Normalmente dispone de una serie de ticks por segundo y es programable respecto a la frecuencia a la que opera.

El sistema además suele programar el temporizador hardware para que cada `HZ` (una constante entera) veces por segundo genere una interrupción de reloj. Esto se usa, como ya mencionamos, entre otras cosas para expulsar del procesador aquellos procesos que llevan suficiente tiempo ejecutando (que han agotado su cuanto). Es tan simple como retornar de la interrupción en un proceso distinto.

Desde el shell, ya sabes que el comando *date(1)* informa respecto a la fecha y hora. Desde C, el principal interfaz para obtener el tiempo es *gettimeofday(2)* (y puede utilizarse *settimeofday(2)* para ajustarlo).

Esta función rellena una estructura de tipo *timeval* con los segundos y microsegundos desde una fecha convenida, llamada *epoch*. Normalmente desde el 1 de enero de 1970 en el caso de UNIX.

```
struct timeval {
        time_t       tv_sec;   /* seconds since Jan. 1, 1970 */
        suseconds_t  tv_usec;  /* and microseconds */
};
```

Además, rellena otra estructura llamada *timezone* que indica la zona horaria en que nos encontramos y si estamos en horario de verano (*daylight saving time*).

```
struct timezone {
        int     tz_minuteswest; /* of Greenwich */
        int     tz_dsttime;     /* type of dst correction to apply */
};
```

Pero *no deberías* dejar que `gettimeofday` rellene información sobre la zona horaria. Es mejor utilizar funciones de *ctime(3)* si deseas jugar con zonas horarias. Por ello nosotros vamos a utilizar `NULL` en el argumento correspondiente a la zona horaria para que `gettimeofday` lo ignore.

Este programa es similar a *date(1)*, pero con la opción "`-n`" imprime el número de segundos y microsegundos y los datos de la zona horaria en lugar de escribir la fecha normalmente.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <err.h>
#include <time.h>
#include <sys/time.h>

typedef unsigned long long uvlong_t;
static char *argv0;

static void
usage(void)
{
    fprintf(stderr, "usage: %s [-n]\n", argv0);
    exit(1);
}
```

```
int
main(int argc, char* argv[])
{
    struct timeval tv;
    int nflag;

    nflag = 0;
    argv0 = argv[0];
    if (argc == 2) {
        if (strcmp(argv[1], "-n") == 0) {
            nflag = 1;
        } else {
            usage();
        }
    }
    if (argc > 2) {
        usage();
    }

    if (gettimeofday(&tv, NULL) < 0) {
        err(1, "gettimeofday");
    }
    if (nflag) {
        printf("%llds %lldµs\n", (uvlong_t)tv.tv_sec, (uvlong_t)tv.tv_usec);
    } else {
        printf("%s", ctime(&tv.tv_sec));
    }
    exit(0);
}
```

La función *ctime(3)* se ocupa de generar un string para la fecha dada como un número de segundos desde *epoch*. Todas los argumentos de funciones de *ctime(3)* que aceptan un `time_t` suelen ser dicho número de segundos.

Para partir la fecha dada por un `time_t` (número de segundos desde *epoch*) y obtener el año, el mes, el día del mes, etc. normalmente se utilizan las funciones `localtime` (hora local) y `gmtime` (hora en greenwich). Por ejemplo, como en

```
struct tm *t;
         struct timeval tv;
if (gettimeofday(&tv, NULL) < 0) {
    err(1, "gettimeofday");
}
t = localtime(&tv.tv_sec);
```

o bien

```
t = gmtime(&tv.tv_sec);
```

Y luego podemos usar los siguientes campos de la estructura `tm`:

```
int tm_sec;     /* seconds (0 - 60) */
int tm_min;     /* minutes (0 - 59) */
int tm_hour;    /* hours (0 - 23) */
int tm_mday;    /* day of month (1 - 31) */
int tm_mon;     /* month of year (0 - 11) */
int tm_year;    /* year - 1900 */
int tm_wday;    /* day of week (Sunday = 0) */
int tm_yday;    /* day of year (0 - 365) */
int tm_isdst;   /* is summer time in effect? */
```

Podríamos tener otros campos dependiendo del UNIX que usemos, pero seguramente no estén disponibles en todos los sistemas.

La función `mktime`, también documentada en *ctime(3)* hace el proceso inverso y genera un tiempo en segundos desde *epoch* a partir de una estructura de tipo `tm`.

13. Cambiando los metadatos

Los metadatos cambian muchas veces simplemente con operar sobre el fichero para cambiar los datos. Por ejemplo, el tamaño del fichero cambia si lo haces crecer utilizando `write`, al igual que las fechas de acceso y modificación.

Aunque ya podemos vaciar un fichero utilizando `open` y hacerlo crecer utilizando `seek` y `write`, en ocasiones querremos truncar un fichero o cambiar su tamaño a un número dado de bytes. Esto puede hacerse con *truncate(2)*, que ajusta el tamaño al número indicado. Si el fichero era más grande, se tiran los bytes del final que sobran. Si el fichero era más pequeño, se hace crecer con ceros. La llamada puede utilizarse como en

```
if (truncate("myfile", 1024) < 0) {
    // truncate ha fallado...
}
```

que dejaría `myfile` con 1024 bytes exactamente. Como podrás suponer, existe *ftruncate(2)* que trunca el fichero indicado por un descriptor de fichero en lugar de por un path.

En realidad, `truncate` cambia el tamaño en los metadatos y si ello require liberar bloques en disco para los datos que ya no se usan, así lo hace.

Los permisos pueden cambiarse con la llamada *chmod(2)*, que es la que utiliza el comando *chmod(1)* que hemos usado anteriormente. Aunque esta vez sí diremos que también tienes *fchmod(2)*, en el futuro no mencionaremos más las funciones que operan con descriptores, ya estás acostumbrado a leer el manual y sabes cómo encontrarlas y usarlas.

Este programa ajusta los permisos de los ficheros que se pasan como argumento al valor indicado por el primer argumento, similar a *chmod(1)*.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <err.h>
```

```
int
main(int argc, char* argv[])
{
    long mode;
    int i, sts;

    if (argc < 3) {
        fprintf(stderr, "usage: %s mode file...\n", argv[0]);
        exit(1);
    }
    mode = strtol(argv[1], NULL, 8);
    sts = 0;
    for (i = 2; i < argc; i++) {
        if (chmod(argv[i], mode) < 0) {
            warn("chmod: %s", argv[i]);
            sts = 1;
        }
    }
    exit(sts);
}
```

Podemos utilizarlo como puedes ver:

```
unix$ chm 664 chm.c
unix$ ls -l chm.c
-rw-rw-r--  1 nemo  staff  452 Aug 21 11:59 chm.c
unix$
```

El propietario y el grupo a que pertenece un fichero se puede cambiar como se describe en *chown(2)*, como en:

```
if (fchown(fd, newuid, newgid) < 0) {
    // fchown ha fallado
}
```

¿Y si tienes un path en lugar de un descriptor? ¿Cómo lo harás?

Las fechas de acceso y modificación de un fichero pueden cambiarse con *utimes(2)*. Por ejemplo, este program es similar a *touch(1)*, aunque nunca crea los ficheros si no existen.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <err.h>

int
main(int argc, char* argv[])
{
    int i, sts;
    struct timeval tv[2];

    if (argc < 2) {
        fprintf(stderr, "usage: %s file...\n", argv[0]);
        exit(1);
    }
    if (gettimeofday(&tv[0], NULL) < 0) {
        err(1, "gettimeofday");
    }
    tv[1] = tv[0];
    sts = 0;
    for (i = 1; i < argc; i++) {
        // could use just NULL instead of tv.
        if (utimes(argv[i], tv) < 0) {
            warn("utimes: %s", argv[i]);
            sts = 1;
        }
    }
    exit(sts);
}
```

Aquí hemos utilizado *gettimeofday(2)* para pedir a UNIX la fecha y hora actual y cambiamos los tiempos de los ficheros justo a ese momento. El mismo efecto podría haberse conseguido utilizando "`NULL`" como argumento (que hace que `utimes` use la fecha actual).

14. Enlaces simbólicos

Recientemente hemos visto que podemos enlazar ficheros desde varios nombres. A este tipo de enlaces se los denomina **hard links** o *enlaces duros*. La ventaja que tienen es que todos los enlaces se comportan bien. Por ejemplo, borrando cualquiera de los nombres para un fichero seguimos teniendo el fichero accesible.

No obstante, no es posible hacer un enlace a un fichero que esté en otro sistema de ficheros (en otra partición, otro disco y otra máquina). Esto es obvio si piensas que el enlace simplemente crea una entrada de directorio que se refiere a un número de *i-nodo* dado, y piensas que el número de *i-nodo* sólo tiene sentido dentro de la misma partición o sistema de ficheros.

Para solucionar este problema, UNIX dispone de **enlaces simbólicos** además de enlaces duros. Un enlace simbólico es similar a un *acceso directo* en Windows. Se trata de un fichero cuyos datos son el path de otro fichero. ¡Tan sencillo como eso!. La consecuencia es que un enlace simbólico puede apuntar a cualquier fichero dado su path. Y, naturalmente, la desventaja es que si borramos el fichero original perdemos los datos del fichero y el enlace queda apuntando a un fichero que no existe.

Para crear un enlace simbólico puedes utilizar el flag `-s` de *ln(1)*. Por ejemplo:

```
unix$ echo hola >fich
unix$ ln -s fich link
unix$ ls -l fich link
-rw-r--r--  1 nemo  wheel  5 Aug 21 19:24 fich
lrwxr-xr-x  1 nemo  wheel  4 Aug 21 19:24 link -> fich
unix$ cat link
hola
unix$ cat fich
hola
unix$ rm fich
unix$ cat link
cat: link: No such file or directory
unix$ ls -l link
lrwxr-xr-x  1 nemo  wheel  4 Aug 21 19:24 link -> fich
```

Es interesante fijarse en la salida de `ls` para `link`. El primer carácter es "`l`", lo que indica que el tipo de fichero es *symbolic link*. Si recuerdas *stat(2)*, puedes utilizar código como

```
if ((st.st_mode & S_IFMT) == S_IFLNK) {
    // el fichero es un enlace simbolico
}
```

para ver si el fichero que tienes entre manos es un enlace simbólico o no.

En general, las llamadas al sistema y funciones que operan con ficheros suelen seguir los enlaces como cabe esperar. Por ejemplo, si `open` recibe como argumento el path de un fichero que es un enlace simbólico, UNIX se da cuenta de ello y procede como sigue:

- Se leen los datos del fichero
- Se interpretan como el path del que hay que hacer el `open`
- Se efectúa el `open` de dicho fichero

Esto hace que normalmente tus programas trabajen correctamente aunque encuentren enlaces simbólicos.

No obstante, hay ocasiones en que hay que prestar atención a este tipo de ficheros. Por ejemplo, si hacemos un `unlink` de un enlace simbólico tan sólo se borra el enlace y no el fichero enlazado. Aquí UNIX no sigue el enlace de forma automática. Pero es mejor así, dado que lo que cabría esperar al ejecutar

```
unix$ rm fich
```

es que se borre `fich`, sea un enlace o no. En ningún caso esperamos que se borre el fichero al que apunta `fich` si es un enlace simbólico.

Llamadas como *stat(2)* son más delicadas. Utilizar `stat` sobre un fichero que es un enlace simbólico hace que UNIX (como hace en general) atraviese el enlace automáticamente. Esto hace que en realidad obtengamos los metadatos del fichero al que apunta el enlace. Dicho de otro modo, ¡`stat` nunca te dirá que un fichero es un enlace simbólico!

Una forma de solucionar este problema es llamar a `lstat`, que funciona igual que `stat` pero cuando el fichero es un enlace devuelve los atributos del enlace en lugar de los del fichero enlazado.

Sucede lo mismo con llamadas como *chown(2)*, *chmod(2)*, etc. Estas llamadas atraviesan los enlaces simbólicos y operan sobre los ficheros enlazados, pero dispones de llamadas con igual nombre pero comenzando por "`l`" (por ej., "`lchmod`") que, cuando el fichero es un enlace simbólico, trabajan sobre el enlace en sí y no sobre el fichero enlazado.

¡El manual es tu amigo! Aunque creas que sabes usar UNIX, consulta rápidamente la página de manual de

la llamada que piensas usar. Quizá recuerdes que debieras utilizar `lstat` y no `stat` o `fstat`... ¡y ahorrarás mucho tiempo depurando bugs que podrías haber evitado!

Para crear un enlace simbólico desde C puedes utilizar *symlink(2)* que funciona igual que *link(2)*, pero crea un enlace simbólico en lugar de uno duro. Por ejemplo, este programa es similar a *ln(1)* bajo el flag `-s`:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <err.h>

int
main(int argc, char* argv[])
{

    if (argc != 3) {
        fprintf(stderr, "usage: %s old new\n", argv[0]);
        exit(1);
    }
    if (symlink(argv[1], argv[2]) < 0) {
        err(1, "symlink %s", argv[1]);
    }
    exit(0);
}
```

15. Dispositivos

Ya conocemos diversos tipos de fichero:

- Ficheros regulares
- Directorios
- Enlaces simbólicos

Pero es hora de mencionar dos más:

- Dispositivos de tipo carácter
- Dispositivos de tipo bloque

Los dispositivos son *i-nodos* que no contienen datos en realidad. Se utilizan para representar dispositivos que pueden ser artefactos hardware (como una impresora o un disco) o pueden ser invenciones de software (como `/dev/null`). La mayoría de los dispositivos suelen estar en `/dev`.

UNIX distingue entre dispositivos de modo carácter y de modo bloque principalmente por que los primeros se espera que correspondan a streams de caracteres (el teclado, la pantalla al escribir caracteres, el ratón, etc.) y los segundos se espera que correspondan a almacenes de bloques de bytes de los que UNIX podría mantener una cache.

De hecho, los discos duros (y las particiones) suelen tener un par de ficheros de dispositivo cada uno: uno de tipo carácter y uno de tipo bloque. El primero se utiliza para formatear el disco y para leerlo saltándose la cache. El segundo se utiliza para acceder al disco beneficiándose de la cache de bloques que mantiene UNIX. Por ejemplo, presentamos los dispositivos de la primera partición del primer disco en nuestro sistema:

```
unix$ ls -l /dev/rdisk0s1
crw-r-----  1 root  operator    1,   1 Jul 13 07:30 /dev/rdisk0s1
unix$ ls -l /dev/disk0s1
brw-r-----  1 root  operator    1,   1 Jul 13 07:30 /dev/disk0s1
```

Cuando abres, lees o escribes un fichero, UNIX localiza el *i-nodo* del fichero y en función del tipo de fichero hace una cosa u otra. En el caso de los dispositivos, simplemente se llama a una función distinta (en el kernel) para cada tipo de dispositivo y se deja que ella haga el trabajo.

El código de un dispositivo suele llamarse **manejador** o **driver**. Si se trata de hardware, además de atender las interrupciones de dicho trozo de hardware y de detectar si está instalado en el sistema o no, el manejador implementará las operaciones del *i-nodo* para el tipo de dispositivo de que se trate.

En un *i-nodo* de un dispositivo tienes dos números:

- Número principal (*major number*)
- Número secundario (*minor number*)

El primero identifica un manejador de dispositivo (por ejemplo, discos duros SATA). El segundo identifica de qué dispositivo concreto se trata (por ejemplo, el primer disco SATA en el primer bus SATA).

Así pues, si un *i-nodo* es un dispositivo de bloque que identifica un disco y el número principal corresponde al driver de discos SATA, todas las llamadas `open`, `read`, ... que operen sobre ese *i-nodo* llamarán a funciones concretas del manejador de discos SATA. A dichas funciones se les dará además el *i-nodo* de que se trate y podrán ver el número secundario (ver de qué disco concreto hay que leer, escribir, etc.).

Ahora puedes entender la salida de este comando:

```
unix$ ls -l /dev/tty
crw-rw-rw-  1 root  wheel    2,   0 Aug 20 12:30 /dev/tty
```

El primer carácter es una "`c`", lo que indica *dispositivo de caracteres*. Y los números `2` y `0` corresponden al número principal y secundario.

Los dispositivos sólo son ficheros que representan a los dispositivos, no son hardware. Esto es, que tengas en `/dev` mil dispositivos para discos duros no quiere decir que tengas mil discos instalados.

Para crear un fichero de dispositivo podemos utilizar *mknod(1)*, que llama a *mknod(2)*. Pero, normalmente hay que ser *root* para poderlo hacer. !Así que vamos a convertirnos en *root* y crear uno!:

```
unix$ sudo su
Password:
unix# mknod term c 2 0
sh-3.2# echo hola >term
hola
unix# ls -l term
crw-r--r--  1 root  staff    2,   0 Aug 21 19:56 term
unix# rm term
unix# exit
unix$
```

Como verás, hemos creado un fichero llamado `term`, que es un dispositivo de modo carácter ("`c`") y tiene números `2` y `0` como *major* y *minor*. Si recuerdas el listado de `/dev/tty`, verás que estamos creando un dispositivo para el mismo artefacto. Una vez creado, podemos utilizar `term` igual que `/dev/tty`, como puedes ver por el efecto de `echo` en nuestro ejemplo.

16. Entrada/salida y buffering

El interfaz proporcionado por `open`, `close`, `read`, `write`, etc. es suficiente la mayoría de las veces. Lo que es más, en muchas ocasiones es el más adecuado. Por ejemplo, *cat(1)* debería utilizar un buffer cierto tamaño (8KiB, quizá) y usar `read` y `write` para hacer su trabajo. Es lo más simple y además lo más eficiente la mayoría de las veces.

No obstante, hay ocasiones en que deseamos leer carácter a carácter o línea a línea, o deseamos escribir poco a poco la salida del programa o escribir con cierto formato. En estos casos es mucho más adecuado utilizar la librería de C que las llamadas al sistema. Además, dicha librería suministra entrada/salida con buffering, lo que hace que el programa sea mucho mas eficiente si leemos o escribimos poco a poco.

Veamos un programa que copia un fichero en otro, pero byte a byte. Esta versión utiliza las llamadas al sistema que hemos visto.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <err.h>

int
main(int argc, char* argv[])
{
    int sfd, dfd, nr;
    char buf[1];

    if(argc != 3) {
        fprintf(stderr, "usage: %s src dst\n", argv[0]);
        exit(1);
    }


    sfd = open(argv[1], O_RDONLY);
    if(sfd < 0){
        err(1, "open: %s", argv[1]);
    }
    dfd = creat(argv[2], 0664);
    if(dfd < 0){
        close(sfd);
        err(1, "creat: %s", argv[2]);
    }
```

```
    for(;;){
        nr = read(sfd, buf, sizeof buf);
        if(nr == 0){
            break;
        }
        if(nr < 0){
            close(sfd);
            close(dfd);
            err(1, "read: %s:", argv[1]);
        }
        if(write(dfd, buf, nr) != nr){
            close(sfd);
            close(dfd);
            err(1, "write: %s:", argv[2]);
        }
    }
    close(sfd);
    if (close(dfd) < 0) {
        err(1, "close: %s:", argv[2]);
    }
    exit(0);
}
```

Podemos crear un fichero de 10MiB utilizando el comando *dd(1)*. Basta pedirle que copie 10240 bloques de 1024 bytes desde `/dev/zero` (una fuente ilimitada de ceros) hasta el fichero en cuestión:

```
unix$ dd if=/dev/zero of=/tmp/10m bs=1024 count=10240
10240+0 records in
10240+0 records out
10485760 bytes transferred in 0.028481 secs (368166762 bytes/sec)
unix$ ls -l /tmp/10m
-rw-r--r--  1 nemo  wheel  10485760 Aug 21 22:06 /tmp/10m
```

Este comando se llama así por usarse hace tiempo para copiar *device to device*, en los tiempos en que las cintas magnéticas eran muy puntillosas respecto a qué tamaños de lectura y escritura admitían.

Ahora veamos cuánto tarda nuestro programa en copiar nuestro nuevo fichero.

```
unix$ time cp1 /tmp/10m /tmp/10mbis
real    0m16.850s
user    0m1.563s
sys    0m15.230s
unix$ ls -l /tmp/10mbis
-rw-r--r--  1 nemo  wheel  10485760 Aug 21 22:06 /tmp/1mbis
```

¡Un total de 16.8 segundos! ¡Un eón! Y para sólo 10MiB de datos. Esto quiere decir que copiar un sólo MiB tardaría aproximadamente 1.68 segundos. Muchísimo tiempo para las máquinas de hoy día.

Por cierto, el comando *time(1)* que hemos utilizado ejecuta la línea de argumentos como un comando e imprime, como has podido ver, cuánto tiempo ha empleado dicho comando en total, cuánto ejecutando código de usuario y cuánto ejecutado código del kernel en su favor. Es una herramienta indispensable para ver si cuándo haces un cambio en un programa has mejorado realmente las cosas o las has empeorado.

Veamos ahora el mismo programa, pero esta vez escrito on *stdio*, la librería estándar de C con buffering para entrada/salida. Igual que antes, vamos a copiar byte a byte.

```
#include <stdio.h>
#include <stdlib.h>
#include <err.h>

int
main(int argc, char* argv[])
{
    FILE *in, *out;
    char buf[1];
    size_t nr;

    if(argc != 3) {
        fprintf(stderr, "usage: %s src dst\n", argv[0]);
        exit(1);
    }

    in = fopen(argv[1], "r");
    if(in == NULL){
        err(1, "%s", argv[1]);
    }
    out = fopen(argv[2], "w");
    if(out == NULL){
        err(1, "%s", argv[2]);
    }


    for(;;){
        nr = fread(buf, sizeof buf, 1, in);
        if(nr == 0){
            if(ferror(in)) {
                err(1, "read");
            }
            break;
        }
        if(fwrite(buf, nr, 1, out) != nr){
            err(1, "write");
        }
    }
    fclose(in);
    fclose(out);
    exit(0);
}
```

En este caso, hemos utilizado *fopen(3)* en lugar de *open(2)* para obtener un "`FILE*`". Esta estructura representa un buffer (quizá) para operar en un descriptor de fichero que la estructura mantiene internamente. Si lees *fopen(3)* verás que hay llamadas para construir un `FILE*` a partir de un descriptor de fichero, en lugar de un path.

En realidad ya conoces este tipo de datos. Anteriormente hemos utilizado `stderr` para imprimir mensajes de error con `fprintf`. Pues bien, `stderr` es un `FILE*` para el descriptor `2`. ¿Resulta claro ahora por qué utilizamos

```
fprintf(stderr, "usage: %s [-n]\n", argv[0]);
```

para informar del uso de un programa? Igualmente tienes `stdin` para la entrada estándar y `stdout` para

la salida estándar.

La función `printf(...)` es equivalente a `fprintf(stdout,...)`. La librería *stdio* tiene funciones que comienzan por "`f`" y tienen nombres similares a funciones que ya conoces, como `fopen`, `fread`, etc. Naturalmente, operan sobre `FILE*` y no sobre descriptores de fichero: para eso están.

Pero volvamos a nuestro experimento. Vamos a ver cuánto tarda este programa en hacer el mismo trabajo:

```
unix$ time 8.bcp2 /tmp/1m /tmp/1mbis

real    0m0.912s
user    0m0.881s
sys    0m0.022s
```

Hemos tardado menos de un segundo. Bastante más rápido que antes. Y hay que decir que la máquina en que hemos hecho este experimento posee un disco SSD que en realidad es memoria de estado sólido, por lo que la diferencia de tiempos no es tanta como sería utilizando un disco duro magnético. ¡La última vez que probamos con discos magnéticos pudimos hacer el descanso de una clase mientras el primero de los programas terminaba su trabajo!

¿Cómo ha podido tardar menos esta segunda versión? La respuesta radica en que se utiliza un buffer intermedio dentro del `FILE*`. La primera vez que se llama a `fread`, *stdio* lee una cantidad razonable de bytes con una llamada a `read`. Si se estaba leyendo un sólo byte no importa, el resto ya están en el buffer esperando que los leas en el futuro. Las llamadas a procedimiento son mucho más rápidas que las llamadas al sistema, dado que éstas han de entrar y salir del kernel y dado que el kernel comprueba cada vez que lo llamas que todos tus argumentos son correctos y que todo está en orden para hacer la llamada.

En las escrituras sucede lo mismo, cuando escribes con `fwrite` (o con `fprintf`), los bytes se quedan normalmente en el buffer. Sólo cuando el buffer se llena o cierras el `FILE*` se escriben los bytes con un `write`. Eso quiere decir que aunque escribamos byte a byte, en realidad estamos haciendo muchos menos `write` y estos escriben muchos bytes a la vez.

Un detalle importante cuando se utiliza buffering, es llamar a `fclose`, dado que, si hemos escrito, los bytes pueden estar en el buffer y no haberse escrito en absoluto en el fichero. Por esta razón `stderr` no utiliza buffering en absoluto, para que los mensajes de error aparezcan inmediatamente en el terminal.

La función `fflush` vuelca el buffer de un `FILE*`, por ejemplo,

```
if (fflush(out) < 0) {
    // el flush ha fallado (fallo en write?)
}
```

escribe lo que pueda haber en el buffer de `out` en el descriptor de fichero que hay bajo `out`. Sabiendo esto... ¿Cuánto crees que tardaría el programa si haces un `fflush` en cada iteración del bucle `for`? ¿Y por qué crees tal cosa?

Hay otras muchas funciones y utilidades en *stdio(3)* que deberías conocer para programar en C. Por ejemplo, un par de funciones para leer línea a línea entre otras. Pero nosotros vamos a continuar explorando UNIX.

17. Buffering en el kernel

Hemos visto hace poco que los dispositivos de modo bloque utilizan una cache de bloques en el kernel. Esto implica que cuando escribes un fichero lo habitual es que tus datos aún no estén en el disco. Lo que es más, incluso ficheros que has creado o borrado podrían no haberse actualizado en el disco.

Dependiendo del tipo de sistema de ficheros que utilices, es posible que incluso si copias el disco en este momento (utilizando su dispositivo crudo o de modo carácter) las estructuras de datos sean incoherentes. Por ejemplo, puede que un directorio contenga una entrada de directorio con un número de *i-nodo* que todavía aparece como libre en el disco.

Una vez UNIX sincroniza su cache en disco, el sistema de ficheros en el disco será coherente, hasta que se hagan más modificaciones. La llamada al sistema *sync(2)*, y el comando *sync(1)* se utilizan precisamente para edir a UNIX que sincronice la cache en disco. Por ejemplo,

```
unix$ sync ; sync
unix$
```

es una buena forma de asegurarse de que las escrituras se han realizado, si pensamos hacer algo peligroso que podría hacer que el sistema fallase estrepitosamente.

Hemos ejecutado dos veces `sync` puesto que lo normal es que UNIX continúe sincronizando el disco tras la llamada a *sync(2)*. La segunda vez que ejecutamos este comando ha de esperar a que acabe la sincronización anterior antes de empezar.

Disponemos de otra llamada, *fsync(2)*, a la que podemos pasar un descriptor de fichero para sincronizar las escrituras de ese fichero. Lo habitual es utilizarla tras escrituras en ficheros importantes que queremos mantener coherentes en el disco aunque falle la alimentación y el sistema muera de repente antes de que sincronice su cache.

Normalmente no es preciso utilizar `sync` dado que UNIX lo hace cada 30 segundos (o cada poco tiempo). Además, cuando se detiene la operación del sistema usando *shutdown(8)* o *halt(8)* UNIX sincroniza los sistemas de ficheros.

Además, es posible que utilicemos un sistema de ficheros que presta atención al orden de las escrituras en el disco de tal forma que el estado del sistema de ficheros en el disco es siempre coherente, lo que hace menos necesario utilizar esta llamada.

Si todo falla y el sistema de ficheros se corrompe en el disco tras un apagón o un fallo del sistema, el comando *fsck(8)* ejecutará durante el siguiente arranque del sistema y tratará de dejar los sistemas de ficheros coherentes de nuevo... ¡Si es que puede! Si llega ese caso... ¡prepárate para perder ficheros! Recuerda mantener tus backups al día.

18. Ficheros proyectados en memoria.

Utilizar `read` y `write` no es la única forma de utilizar un fichero. La abstracción fichero en UNIX tiene una relación muy estrecha con la abstracción *segmento de memoria*.

Ya sabemos que un proceso tiene diversos segmentos y sabemos que el segmento de texto procede del fichero ejecutable y se *pagina en demanda*. Recordemos que un segmento en UNIX es una abstracción. Posee una dirección de memoria virtual de comienzo, un tamaño, unos permisos (leer, escribir, ejecutar) y habitualmente se pagina desde un fichero. Para BSS y otros segmentos sin inicializar (que UNIX inicializa a cero) el fichero en cuestión suele ser `/dev/zero`.

Inicialmente, el segmento será simplemente una estructura de datos y no se le asigna memoria física (salvo que sea preciso utilizar parte de la memoria en realidad como en el caso de la pila). Una vez el programa ejecuta y utiliza direcciones de memoria del segmento, el hardware provoca un fallo de página (una excepción) si no existe memoria física asignada. El manejador, el kernel, asigna la memoria física y la inicializa, poniendo la traducción adecuada de memoria virtual a física (de *página* a *marco*).

Sabes que dado que no es posible tener una traducción en la tabla que utiliza el hardware (*tabla de páginas*)

para cada dirección, las traducciones de memoria virtual a memoria física siempre traducen bloques de 4KiB de direcciones virtuales a bloques de 4KiB de direcciones físicas. A la memoria utilizada con direcciones virtuales es a lo que llamamos página y a la memoria utilizada con direcciones físicas (memoria de verdad) es a lo que llamamos marco de página. Tanto páginas como marcos deben comenzar en direcciones que sean un múltiplo del tamaño de página (normalmente 4KiB, como hemos supuesto antes en este mismo párrafo). Pero todo esto ya lo conoces de arquitectura de computadores.

Pues bien, si consideras que los ficheros (aunque procedan de disco) utilizan una cache de bloques de disco, y piensas que UNIX puede hacer que la cache para los datos del fichero sea en realidad memoria física que puede asignarse a memoria virtual... ¡Ya lo tienes!

Es posible pedir a UNIX que un segmento de memoria virtual corresponda a un fichero existente en disco. A partir de dicho momento, leer direcciones del segmento implica leer los datos del fichero (igual que se lee el código del programa conforme ejecuta). Del mismo modo, escribir direcciones del segmento implica escribir en la cache del fichero en memoria. Más adelante UNIX sincronizará el contenido del disco.

La llamada al sistema que consigue este efecto es *mmap(2)*. Tiene un número significativo de parámetros:

```
void *
mmap(void *addr, size_t len, int prot, int flags, int fd, off_t offset);
```

Para usarla bien, debes pensar en lo que hemos discutido antes. UNIX mantendrá páginas de memoria virtual y hará que correspondan a trozos del mismo tamaño en el fichero. La primera dirección virtual será `addr`, o bien UNIX elige una si `addr` es `0`. En total se proyectan `len` bytes desde el fichero a memoria. Como comprenderás, lo mejor será que `len` sea un múltiplo del tamaño de página. El fichero está identificado por el descriptor `fd` y los bytes proyectados comienzan en `offset` dentro del fichero. Los parámetros `prot` y `flags` sirven para indicar los permisos para la memoria proyectada e indicar a unix propiedades que queremos para el nuevo segmento de memoria.

Por ejemplo, algunos valores interesantes para `flags` son

- `MAP_PRIVATE`: Las escrituras hechas por el proceso no se verán ni en el fichero y en la proyecciones a memoria que otros procesos hagan. Básicamente las páginas que se modifican se copian en cuanto el proceso intenta modificarlas. A esto se le llama **copy on write** y es en realidad la técnica que utiliza UNIX para hacer creer que en un `fork` el hijo es una copia del padre. Sólo se copia lo que cambia alguno de los procesos.

- `MAP_SHARED`: Las escrituras terminan en el fichero y son compartidas con todos los demás. Esto suele ser lo habitual.

- `MAP_ANON`: En realidad queremos un nuevo segmento de memoria *anónima* (que no procede de ningún fichero).

Este programa proyecta un fichero en memoria y lo modifica.

```
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/mman.h>
#include <err.h>
#include <string.h>

enum {KiB = 1024};
```

```
int
main(int argc, char* argv[])
{
    int fd;
    void *addr;
    char *p;

    fd = open("/tmp/afile", O_RDWR|O_CREAT, 0644);
    if (fd < 0) {
        err(1, "open: /tmp/afile");
    }
    if (ftruncate(fd, 12*KiB) < 0) {
        close(fd);
        err(1, "truncate: /tmp/afile");
    }



    addr = mmap(0, 8*KiB, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
    close(fd);
    if (addr == MAP_FAILED) {
        err(1, "mmap: /tmp/afile");
    }
    p = addr;
    strcpy(&p[2*KiB], "hi at 2k\n");
    strcpy(&p[6*KiB], "hi at 6k\n");
    if (munmap(addr, 8*KiB) < 0) {
        err(1, "munmap: /tmp/afile");
    }
    exit(0);
}
```

Observa como dejamos que UNIX determine la dirección de memoria en que ponemos el nuevo segmento (es siempre lo mejor). Además, los tamaños están elegidos para que sean múltiplos de 4KiB (que presuponemos como tamaño de página, lo que hoy día suele ser cierto en todos los sistemas). Además, cuando terminamos, llamamos a *munmap(2)* para pedir a UNIX que termine la proyección.

Si ejecutamos nuestro nuevo programa podemos ver qué tiene `/tmp/afile`:

```
unix$ map
unix$ ls -l /tmp/afile
unix$ ls -l /tmp/afile
-rw-r--r--  1 nemo  wheel  12288 Aug 27 22:07 /tmp/afile
unix$ echo '12*1024' | hoc
12288
unix$ cat /tmp/afile
hi at 2k
hi at 6k
unix$
```

Y, para ser mas precisos...

```
unix$ od -A d -c /tmp/afile
0000000   \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
*
0002048    h   i       a   t       2   k  \n  \0  \0  \0  \0  \0  \0  \0
0002064   \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
*
0006144    h   i       a   t       6   k  \n  \0  \0  \0  \0  \0  \0  \0
0006160   \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
*
0012288
```

Hemos pedido a `od` con "`-A d`" que imprima las direcciones en base 10. Como puedes ver, en el offset 2048 tenemos el string que escribió nuestro programa en su memoria. Igual sucede en el offset 6144 (6KiB) con el string escrito allí. El resto de bytes está a cero dado que nunca los hemos escrito.

Si en el futuro proyectamos el fichero de nuevo, digamos en la dirección `p`, tendremos en `&p[2*KiB]` nuestro primer string tal cual esté en el fichero. Ni que decir tiene que aunque hemos escrito y leído bytes, es memoria como todo lo demás y podríamos haber utilizado cualquier otro tipo de datos.

Con todo esto ya sabes cómo crear nuevos segmentos en tu proceso. Hace tiempo se utilizaba una llamada *sbrk(2)* para aumentar el tamaño del segmento de datos, antes de disponer de las facilidades que nos da en la actualidad la memoria virtual. La implementación de *malloc(3)* utilizaba *sbrk(2)* para pedir más memoria virtual en el segmento de datos cuando agotaba la que tenía. Hoy en día, si necesitas una cantidad ingente de memoria virtual es mejor que pidas tu propio segmento y lo uses como te plazca. De hecho, algunos sistemas UNIX crean un segmento independiente llamado *heap* para dar servicio a *malloc(3)*. Otros usan *sbrk(2)* o bien crean un segmento con suficiente tamaño de memoria virtual. En cualquier caso, es mejor dejar la implementación de `malloc` en paz y pedir lo que necesites sin molestar a la librería de C.

19. Montajes y nombres

En UNIX tienes todos los ficheros dispuestos en un único árbol como ya sabes. El directorio raíz es "`/`" y en dicho directorio están contenidos todos los ficheros a que puedes acceder. Esto es así incluso si tienes varias particiones con ficheros o varios discos.

Una vez más, estamos ante otra abstracción suministrada por el sistema, la idea de un **espacio de nombres** único que podemos adaptar para crear la ilusión de un sólo árbol, aunque tengamos múltiples árboles de ficheros. Normalmente cada árbol está guardado en una partición utilizando un formato concreto de sistema de ficheros, pero recuerda que existen árboles de ficheros implementados en software que no tienen almacenamiento en disco (como `/proc`).

La implementación es simple: UNIX dispone de una *tabla de montajes* que hace que, al recorrer paths, el kernel pueda saltar de un directorio en el árbol al directorio raíz de de otro árbol.
El efecto puedes verlo en la figura 4. En ella puedes ver que el administrador de este sistema ha utilizado dos discos (o dos particiones, a UNIX le da lo mismo) y ha hecho que en el directorio `/home` se monte el segundo disco (o la segunda partición). Tras el montaje, cuando el kernel resuelva paths y alcance "`/home`", saltará al directorio raíz del segundo disco, por lo que aparentemente tenemos paths como "`/home/nemo`", lo que es una ilusión.

A la vista de esto puedes imaginar que si `/home` contenía ficheros o directorios antes del montaje, dicho contenido resulta ahora inaccesible. Aparentemente `/home` contiene los directorios `root`, `nemo`, etc. y eso es todo lo que pueden ver los usuarios del sistema.

El comando `mount(1)` muestra los montajes en el sistema:

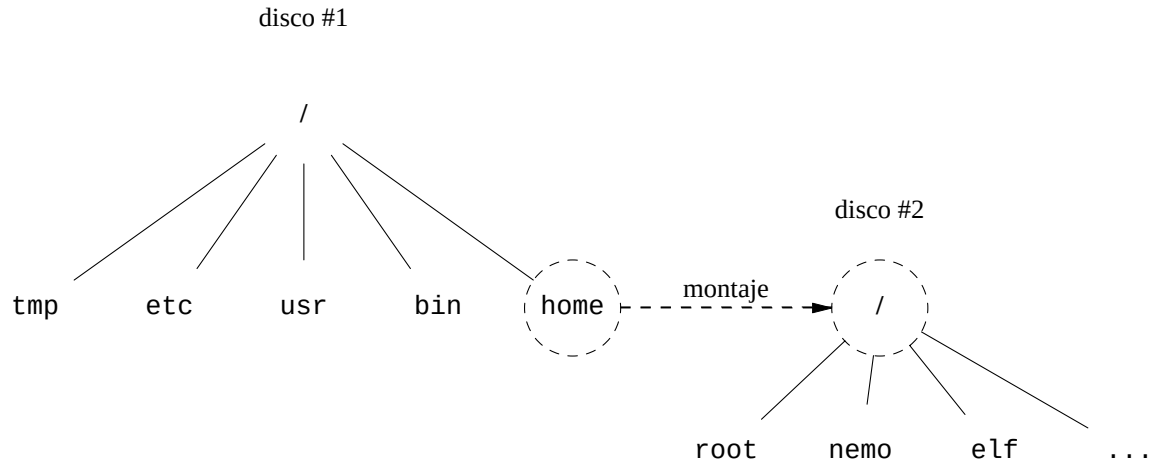

Figura 4: *Un montaje hace que UNIX salte de un directorio en el árbol al raíz de otro árbol creando el efecto de un único árbol.*

```
unix$ mount
/dev/disk1 on / (hfs, local, journaled, noatime)
devfs on /dev (devfs, local, nobrowse)
elf@fs.lsub.org:/home/dump on /dump (osxfusefs, nodev, nosuid)
elf@fs.lsub.org:/home/lsub on /zx (osxfusefs, nodev, nosuid)
elf@fs.lsub.org:/home/once on /once (osxfusefs, nodev, nosuid)
unix$
```

El resultando dependerá mucho no sólo del tipo de UNIX sino también de cómo esté administrado. En este caso vemos que hay una sóla partición en el primer disco que está montada en el directorio raíz. Por otra parte, no hay `/proc` en este sistema (OS X) y `/dev` es un sistema de ficheros sintético (igual que lo es `/proc` en otros UNIX). Puede verse también que los directorios `/dump`, `/zx` y `/once` están montados y proceden de otra máquina llamada `fs.lsub.org`.

Existe otro comando que resulta útil no sólo para ver qué tenemos montado sino también para ver cuánto espacio libre resta en cada uno de los sistemas de ficheros. hablamos de *df(1)*. Para que puedamos ver otro ejemplo, vamos a utilizar un sistema OpenBSD esta vez:

```
unix$ mount
/dev/sd2a on / type ffs (local, noatime, softdep)
unix$ df -h
Filesystem     Size    Used   Avail Capacity  Mounted on
/dev/sd2a      1.8T    239G    1.5T    14%    /
unix$
```

En este sistema hay un único sistema de ficheros montado en el raíz, procedente de la primera partición (la "`a`" en "`sd2a`") del tercer disco (el "`2`" en "`sd2a`", empezando a contar en cero). Con el flag `-h` (*human readable sizes*) `df` informa que en dicha partición de 1.8TiB estamos usando 239GiB.

El flag `-i` de `df` es muy útil e informa de cuántos *i-nodos* estamos usando en cada sistema de ficheros.

```
unix$ df -ih
Filesystem     Size    Used   Avail Capacity iused   ifree  %iused  Mounted on
/dev/sd2a      1.8T    239G    1.5T    14% 1727771 58895843     3%   /
```

Como podrás suponer, una vez hemos usado todos los *i-nodos* de que dispone un sistema de ficheros ya no es posible crear nuevos ficheros dentro del mismo.

Para montar sistemas de ficheros hay que ser *root* (a no ser que dicho usuario se ocupe de configurar el sistema para que un usuario normal pueda realizar ciertos montajes, como por ejemplo sucede con los discos USB hoy día). Un ejemplo es

```
unix$ mount -t mfs -o rw /dev/sd0b /tmp
```

que monta (en un sistema OpenBSD) en `/tmp` un sistema de ficheros en memoria virtual, respaldado por una partición de swap en disco. (Dicha partición se utiliza para mantener en ella la parte de la memoria virtual que no nos cabe en memoria física).

Para deshacer el efecto de *mount(8)* disponemos de *umount(8)*. Por ejemplo,

```
unix$ umount /tmp
```

deshace el montaje del ejemplo anterior. El sistema se negará a desmontar un sistema de ficheros que esté utilizándose. Por ejemplo, si un proceso tiene su directorio actual dentro de dicho sistema de ficheros o tiene ficheros abiertos procedentes del mismo, o está paginando su código desde un ejecutable almacenado en él. Si lo intentamos...

```
unix$ umount /
/dev/sd2a: device busy
unix$
```

Tenemos llamadas al sistema en *mount(2)* que utilizan los comandos que hemos visto, pero es muy poco probable que las necesites.