Introducción a Sistemas Operativos: Ficheros

Clips xxx
Francisco J Ballesteros

1. 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 /
```

2. 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/..
```