Hard Links y Soft Links

> Tutorial extraído del blog C para Operativos (autor: Matías García Isaia)

¿Qué es un hard link? ¿Qué es un soft link? ¿Qué diferencia hay entre ellos?

Me parece que la clave para entender los enlaces en los filesystems basados en inodos[1] es entender *qué es un archivo*.

Archivos y directorios

Un archivo está compuesto por dos partes: su contenido (*los bloques de datos*) y su metadata (*el inodo*).

Cuando creo un archivo llamado `miarchivo.txt` cuyo único contenido es `hola` (con, por ejemplo, el comando `echo hola > miarchivo.txt`), en mi FS se reserva un bloque de datos en el que se escriben los 5 bytes `hola\n` [2], y además se reserva un inodo que se marca como ocupado, en el que se especifica que el contenido del archivo mide 5 bytes, y se establece en el primer puntero del inodo que el primer bloque de datos es el que acabamos de reservar. Se ponen más datos, obviamente, pero por ahora no nos importan.

Y con eso tenemos un archivo, pero nos falta algo importante: referenciarlo. Ese archivo **no tiene nombre**. *El inodo no guarda el nombre del archivo*. Y yo nunca le pido al sistema operativo que me abra el archivo del inodo 5236: yo le pido rutas. Y una ruta está formada por una lista de directorios, y el nombre del archivo al final.

El nombre existe en el directorio.

Un directorio es un archivo cuyo contenido es una tabla. Cada entrada de esa tabla (las *entradas de directorios*) relaciona un nombre de archivo con un número de inodo. Al querer abrir un archivo (digamos, `miarchivo.txt`), el FS busca en esa tabla la entrada con ese nombre, y mira cuál es el número de inodo que le corresponde (digamos, `402`), y hace todas las operaciones usando ese inodo. Siguiendo el ejemplo, si quisieramos leer todo el archivo `miarchivo.txt` (`cat miarchivo.txt`), el FS encuentra la entrada de directorio de ese archivo, lee que le corresponde el inodo 402, abre el inodo 402, mira que mide 5 bytes, busca cuál es el primer bloque de datos, lee los primeros 5 bytes de ese bloque de datos y se los devuelve a `cat`, que se encarga de imprimirlos por pantalla.

```bash
$ cat miarchivo.txt
hola
$
```

Hard links

Ahora, ¿qué me impide tener otra entrada de directorio que apunte al inodo 402? En el inodo no hay ninguna referencia del estilo `directorioPadre` ni nada por el estilo. Tranquilamente podría crear una nueva entrada en ese directorio (o en otro, es indistinto[3]) que apunte al mismo inodo. Asumamos que creamos una entrada para el nombre `passwords.txt`, que también apunta al inodo 402.

¿Qué va a ocurrir cuando quiera leer el archivo `passwords.txt` haciendo `cat passwords.txt`? Esto:

> el FS encuentra la entrada de directorio de ese archivo, lee que le corresponde el inodo 402, abre el inodo 402, mira que mide 5 bytes, busca cuál es el primer bloque de datos, lee los primeros 5 bytes de ese bloque de datos y se los devuelve a `cat`, que se encarga de imprimirlos por pantalla.

```bash
$ cat passwords.txt
hola
$
```

Exactamente lo mismo que hace un rato, porque, a partir de que encontró que el inodo es el 402, el resto va todo igual: el inodo es lo que representa a nuestro archivo en el FS.

Y si quisiera agregarle contenido a `miarchivo.txt`, ¿qué pasaría? Ejecuto `echo clave123 >> miarchivo.txt`, y entonces el FS busca la entrada correspondiente a `miarchivo.txt`, encuentra que corresponde al inodo 402, va al inodo 402, ve que mide 5 bytes y que si le agrega los 9 bytes correspondientes a `clave123\n` sigue entrando en el primer bloque de datos, entonces abre el primer bloque de datos, y a partir del sexto byte escribe ese contenido. Por último, anota en el inodo que el archivo ahora pesa 14 bytes, cierra el archivo, y todos felices. Si ahora hacemos `cat miarchivo.txt`, encuentra la entrada de directorio[4], abre el inodo 402, abre el bloque de datos, lee los 14 bytes, se los devuelve a `cat`, y `cat` los imprime por pantalla.

```bash
$ cat miarchivo.txt
hola
clave123
$
```

¿Y si ahora leo el contenido de `passwords.txt`? Bueno, *leo la entrada de directorio de `passwords.txt`, veo que es el inodo 402, abro el inodo, veo que mide 14 bytes, abro el primer bloque de datos, leo los 14 bytes, y se los devuelvo a `cat` para que los imprima en pantalla*.

```bash
$ cat passwords.txt
hola
clave123
$
```

Las dos entradas de directorio referencian al mismo archivo. Lo que *escribo en un archivo se ve reflejado en el otro*, pero porque **es mentira que sean dos archivos**: *son el*

mismo. Son *dos referencias al mismo objeto*.

Ahora bien, podemos mirar por la consola que esos dos archivos referencian al mismo inodo. La clave está en el switch `-i` del comando `ls`, que acá vamos a usar junto con el `-1` para ver un archivo por línea:

```bash
$ ls -i1
402 miarchivo.txt
402 passwords.txt
$
```

Ambas entradas referencian al inodo 402.

Ese es el concepto de hard link: varias entradas de directorio referenciando al mismo inodo. *Varias referencias al mismo archivo*.

Pero es interesante que, a partir del momento en que se crea el hard link, ya no existe *el archivo* y *el hard link*: el archivo es el inodo, y todas las referencias a inodos son hardlinks. O sea, técnicamente hablando, *todas las entradas de directorios son hard links*, pero coloquialmente hablando nos referimos a hard links en el momento en que hay más de una entrada refiriendo a ese inodo. Pero es importante tener siempre presente que, a partir de que se crea el hard link (es decir, que se crea la segunda entrada apuntando al mismo inodo), ya no hay diferencia entre la referencia original y la nueva: podríamos borrar `miarchivo.txt`, y `passwords.txt` seguiría existiendo y apuntando al mismo contenido. Podríamos crear un tercer hard link, borrar los dos anteriores, y el inodo seguiría siendo el mismo, con el mismo contenido.

Y, ya que estamos, ¿qué pasa si borro un hard link? ¿Cómo se cuándo liberar el inodo y los bloques de datos? Para eliminar un archivo, siempre uso el mismo comando: `rm`. Y no tengo que especificarle yo si es el último hard link o no, o si liberar o no el inodo: es el FS el que decide eso.

Para eso, el FS usa una técnica llamada reference counting[5] (conteo de referencias). El inodo tiene un campo en el que se cuenta la cantidad de referencias que hay hacia él en el file system. Al crear un archivo, ese contador arranca en 1 (porque hay una única entrada de directorio que lo apunta), cuando se crea un hard link se aumenta en 1, y cuando se elimina una entrada (*se elimina un hard link*) se decrementa en 1. Si ese contador llega a 0, el FS libera el inodo y los bloques de datos asociados. De este modo, en el caso de `miarchivo.txt`, el inodo 402 arrancó con el contador de links en 1, y cuando creamos el hard link `passwords.txt` pasó a 2. Si ahora borramos alguno de los dos (es indistinto cuál, digamos que hacemos `rm miarchivo.txt`), se borra la entrada de directorio correspondiente y se decrementa en uno el contador de links del inodo 402, pasando a 1. Si borramos el link que quedaba (`rm passwords.txt`), el contador de links del inodo 402 pasa a 0, y ahí el FS libera ese inodo y el bloque de datos asociado.

Podemos ver la cantidad de hard links usando `ls -l` (o combinarlo con `-i` para ver también el número de inodo):

```bash
$ ls -li
total 2
402 -rw-r--r--+ 2 mgarcia staff 14 Feb 7 11:53 miarchivo.txt
402 -rw-r--r--+ 2 mgarcia staff 14 Feb 7 11:53 passwords.txt
$
```

Esos `2` que vemos entre los permisos y el owner del archivo es el contador de hard links. Creemos un archivo cualquiera para ver que tiene un único link:

```bash
$ touch vacio
$ ls -li
total 3
402 -rw-r--r--+ 2 mgarcia staff 14 Feb 7 11:53 miarchivo.txt
402 -rw-r--r--+ 2 mgarcia staff 14 Feb 7 11:53 passwords.txt
406 -rw-r--r--+ 1 mgarcia staff 14 Feb 7 13:27 vacio
$
```

Borremos tanto a `vacio` como a `miarchivo.txt` para ver que no les mentí[6]:

```bash
$ rm miarchivo.txt vacio
$ ls -li
total 1
402 -rw-r--r--+ 1 mgarcia staff 14 Feb 7 11:53 passwords.txt
$
```

Si borrara `passwords.txt`, recién ahí se liberaría el inodo 402.

Por último, un detalle de los hard links: toda la relación que existe entre la entrada de directorio y el inodo es el número de inodo. Debido a esto, los hard links sólo pueden realizarse *intra-filesystem*, es decir, el archivo destino y el directorio que contiene al nuevo link **deben** estar dentro del mismo file system[7]. Porque si no, ¿al inodo 402 de qué file system se refiere la entrada? ¿Qué garantías tengo de que el otro filesystem también maneje inodos? Por ese motivo, los hard links son intra-FS.

Y, algo que nos faltaba: ¿con qué creamos el hard link? Con el comando `ln`:

```bash
$ ls -li
total 1
402 -rw-r--r--+ 1 mgarcia staff 14 Feb 7 11:53 miarchivo.txt
$ ln miarchivo.txt passwords.txt
$ ls -li
total 2
402 -rw-r--r--+ 2 mgarcia staff 14 Feb 7 11:53 miarchivo.txt
402 -rw-r--r--+ 2 mgarcia staff 14 Feb 7 11:53 passwords.txt
$
```

Esto es lo que ejecuté (hace un rato) para crear passwords.txt

Soft links

Ahora bien, ¿ya está? Si quiero hacer un enlace a otro FS, ¿estoy en el horno? Nops: para eso hicieron los soft links 😃

Un softlink[8] es un **archivo nuevo** que referencia a otra ruta. Creemos uno:

```bash
$ ls -li
total 2
402 -rw-r--r--+ 2 mgarcia staff 14 Feb 7 11:53 miarchivo.txt
402 -rw-r--r--+ 2 mgarcia staff 14 Feb 7 11:53 passwords.txt
$ ln -s miarchivo.txt blandito
$ ls -li
total 3
415 lrwxr-xr-x  1 mgarcia staff 14 Feb 7 13:46 blandito -> miarchivo.txt
402 -rw-r--r--+ 2 mgarcia staff 14 Feb 7 11:53 miarchivo.txt
402 -rw-r--r--+ 2 mgarcia staff 14 Feb 7 11:53 passwords.txt
$
```

Por lo pronto, para crear un soft link le pasamos el parámetro `-s` a `ln`. Por otro lado, vemos que se creó una nueva entrada de directorio llamada `blandito` [9], **que tiene su propio inodo**. Es decir, es un archivo completamente separado de los otros. De hecho, tiene sus propios atributos: la `l` que vemos al principio es la que indica que ese archivo es un softlink. También vemos que el contador de links está en 1, porque, claro, este inodo (el 415) está referenciado únicamente por la entrada de directorio `blandito`.

Y, si es un inodo aparte, ¿cómo referencia al otro archivo? Bueno, el FS sabe que los archivos con el atributo de enlace (la `l` esa que vimos en los permisos) son softlinks, y entonces sabe que tiene que tratarlos de manera especial: cuando le pidan leerlo (`cat blandito`), no va a hacer lo que hacía siempre (abrir el inodo referenciado - 415 -, leer su contenido y pasárselo a `cat`). Por ser un softlink, el contenido que está guardado en el bloque de datos no es el contenido del archivo, si no **la ruta del archivo al que enlaza**[10]. Entonces, al hacer `cat blandito` el FS abre el inodo 415, ve que es un softlink, lee el bloque de datos, ve que está guardada la ruta `miarchivo.txt` , y entonces, ahora sí, busca la entrada `miarchivo.txt` , abre el inodo correspondiente (el viejo y conocido 402), lee el bloque de datos, etc.

¿Se ve la diferencia? **El contenido del softlink dice cuál es el path del archivo al que uno realmente quiere acceder.** Esto es lo que le permite ser cross-FS: el link se puede hacer a, no se, `/mount/DVD/autorun.inf` , que sea otro FS, y no hay ningún problema: el identificador ya no es el número de inodo, y por eso puede ir a otros FS.

Del `ls` que hicimos después de crear el softlink tenemos que notar otro detalle: el link count del inodo 402 sigue en 2, **no aumentó a 3**. Esto significa que **el archivo *apuntado* no se entera de que hay un softlink que lo enlaza**.

El hecho de que el enlace esté hecho a nivel *ruta* (y no *inodo*) implica que puedo tener lo que se conoce como *enlaces rotos*: links que apuntan a archivos que no existen [más]. Por

ejemplo, ¿qué pasa si eliminamos `miarchivo.txt` ? `blandito` sigue referenciando a un archivo llamado `miarchivo.txt` , pero ese archivo ya no existe, así que cualquier operación sobre el link va a fallar. No importa que siga existiendo `passwords.txt` , que sigue apuntando al inodo 402: el link funciona a nivel de paths. Lo mismo ocurre si renombro el archivo `miarchivo.txt` , o si pongo un archivo distinto llamado `miarchivo.txt` : el enlace apunta a *lo que sea que haya* en la ruta `miarchivo.txt` .

Existe un comando (más *low level*[11] que `ls -l`) para leer el *destino* de un softlink: `readlink` .

```bash
$ readlink blandito
miarchivo.txt
$
```

Rompamos el enlace:

```bash
$ mv miarchivo.txt archivo.txt
$ ls -li
total 3
415 lrwxr-xr-x  1 mgarcia staff 14 Feb 7 13:46 blandito -> miarchivo.txt
402 -rw-r--r--+ 2 mgarcia staff 14 Feb 7 11:53 archivo.txt
402 -rw-r--r--+ 2 mgarcia staff 14 Feb 7 11:53 passwords.txt
$ cat blandito
cat: blandito: No such file or directory
$
```

That's it 😃

Windows

Ok, hablemos (un poco, medio de mala gana, y bastante desde el desconocimiento) de Windows.

En FAT no existen la conceptos de soft ni hard links. NTFS los tiene.

Lo que sí existe en Windows desde antes de NTFS son los viejos y conocidos accesos directos.

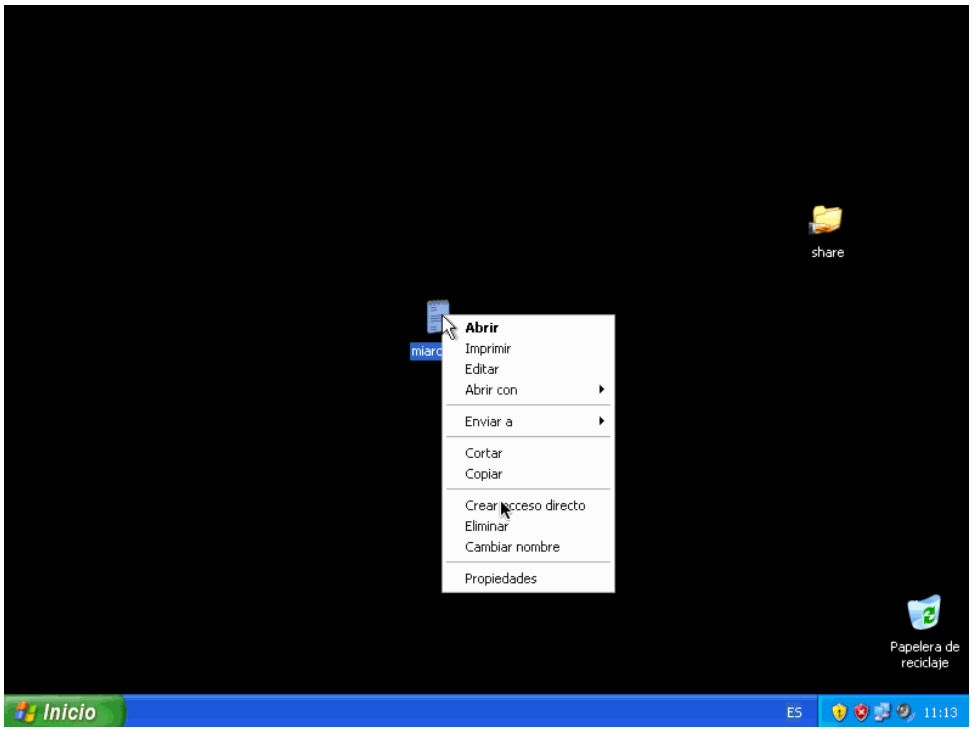
¿Qué son los accesos directos?

Bueno, son archivos que están guardados en el file system, y que cuando los abro en realidad me abren otros archivos a los que "apuntan".

¡ESO ES UN SOFTLINK! ¡AHHHHH IGNORANTE! ¡ME MENTISTE! ¡¡¿¿VISTE QUE SÍ EXISTÍAN??!! Te dije que este pibe era un talibán linuxero y que no puede contar objetivamente nada de Windows.

Sí, bueno, no... Son softlinks, pero leé de nuevo lo que puse: FAT no soporta symlinks; Windows lo soporta.

¿Y qué cambia? Bueno, cambia bastante. En Windows/FAT, los accesos directos los interpreta el shell (es decir, la interfaz de ventanitas que te deja hacer doble click sobre los archivos), y no el file system. Esto significa que, dependiendo de cómo interactúe cada aplicación con la API de Windows, al abrir un acceso directo podría leer el contenido del archivo destino o el del propio acceso directo. Tengo la sensación de que en las versiones nuevas[12] de Windows hicieron algunos cambios para que esto no pase, pero, por ejemplo, uno puede darle doble click a un acceso directo a un txt en Windows XP para abrir el archivo destino en notepad[13], pero si abrimos notepad y hacemos drag&drop del acceso directo vamos a ver un montón de chirimbolos horrendos que tienen codificada de algún modo extraño la ruta al destino[14].



En Unix/ext, el propio sistema de archivos se encarga de resolver el enlace al abrir un softlink, por lo que siempre vemos el contenido del destino[15].

Links útiles

- Hard links and Unix file system nodes (inodes)

---

1. Existen los enlaces ~~en FAT~~ en Windows, pero funcionan a otro nivel, me disgustan, me caen mal ~~, y voy a ignorarlos rotundamente~~ 😃 ↩
2. Por defecto, `echo` imprime un salto de línea, y por eso el `\n` al final del contenido. Se puede evitar el `\n` final haciendo `echo -n` . ↩
3. Sólo importa que sea dentro del mismo filesystem, ya veremos por qué. ↩

4. A esta altura ya te habrás imaginado que tiene bastante sentido aplicar algúna técnica de caché para las entradas de directorios, porque *se usan bastante seguido*, je 😃 ↩

5. Como dice el artículo de Wikipedia, la misma idea de reference counting se usa mucho en los garbage collectors de muchos lenguajes de programación de objetos. ↩

6. Que no les mentí *tanto*. Estoy inventando un poco esos outputs de los comandos, porque no los estoy corriendo realmente en la consola. Hago pruebas y lo fabrico, pero debería ser consistente. ↩

7. Sí, en el sistema operativo se pueden (y de hecho, se suelen) montar más de un sistema de archivos. Cuando metés un CD en la máquina, estás montando otro FS. Cuando tenés más de una partición disponible, estás montando más de un sistema de archivos. Incluso cuando tenés una partición de swap, estás montando otro sistema de archivos. ↩

8. "softlink", "soft link", "enlace dinámico", "enlace *suave*", *you name it* ↩

9. Lo de `-> miarchivo.txt` es información que agrega `ls` para ser *más usable*, pero esa información **no está guardada en la entrada de directorio**. ↩

10. La enorme mayoría de las implementaciones de ext optimizan el uso de disco almacenando la ruta destino *en el inodo* en lugar de en un bloque de datos aparte (escribiéndola en la zona reservada para los punteros a bloques de datos). De hecho, la especificación de ext2 dice que ese es el modo de hacerlo (digamos, no lo nombra como un opcional). De todos modos, esto es una optimización de espacio, y no hace diferencias importantes a la hora de entender el concepto del symlink. ↩

11. En general, trato de usar los comandos de UNIX más *cercanos* a las llamadas al sistema. `ls` hace *muchas* cosas, no existe una llamada al sistema `ls`, pero sí existe una llamada al sistema readlink (o quizá no se llame así, pero cumple esa tarea específica). ↩

12. Llamar "nuevas" a las que tienen menos de 15 años de antigüedad denota mi pasión por gritarle a las nubes ↩

13. Hice recién la misma prueba en Windows 8.1, y funcionó del mismo modo, pero no se por qué siento que alguna vez me pasó distinto. ↩

14. Y, obviamente, si usamos el programa `edit` desde la consola/CMD/DOS, peor todavía. ↩

15. Entiendo que uno podría modificar una implementación de ext para que muestre el contenido de los enlaces en lugar de resolverlos, recompilarla, instalarla y montar un FS con esa implementación, pero sería *llevar las cosas bastante al límite*, y, obviamente, dejaría de ser ext (porque ya no cumple con la especificación de cómo abrir los enlaces, je). ↩