Introducción a Sistemas Operativos: Ficheros

Clips xxx
Francisco J Ballesteros

1. 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 1.

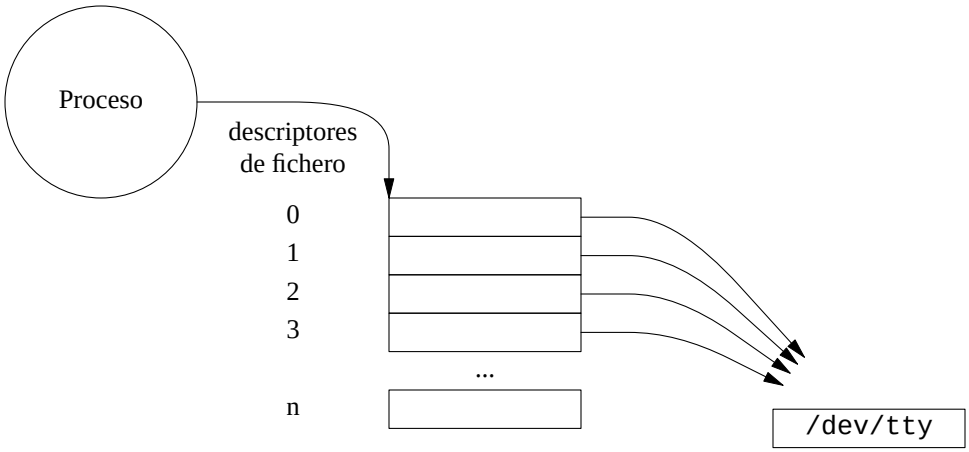

Figura 1: *Descriptores tras abrir el terminal usando* `open`.

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.

2. 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.

3. 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!