Introducción a Sistemas Operativos: Ficheros

Clips xxx
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.

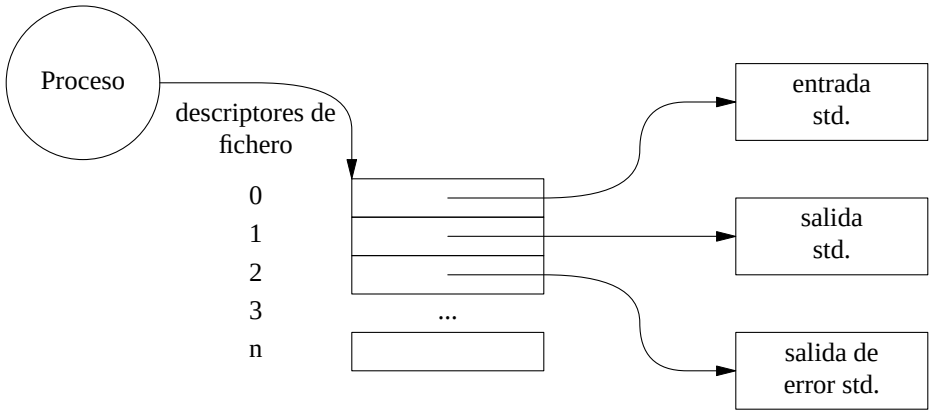

Figura 1: *Los descriptores de fichero son entradas en un array que apuntan a los ficheros abiertos de un proceso. Se utilizan para la entrada estándar, la salida estándar, la salida de error estándar y cualquier otro fichero abierto.*

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.