Introducción a Sistemas Operativos: Ficheros

Clips xxx
Francisco J Ballesteros

1. 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.

2. 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.