Introducción a Sistemas Operativos: Ficheros

Clips xxx
Francisco J Ballesteros

1. Offsets

Hay una pregunta que seguramente te has hecho. ¿Cómo saben `read` y `write` en qué posición del fichero han de trabajar? Esto es, ¿En qué *offset* debe escribir `write`? o ¿Desde qué *offset* debe leer `read`?

La respuesta procede de `open`. Cuando se abre un fichero, el sistema mantiene la pista de en qué posición del fichero se está trabajando. Inicialmente este offset es cero. Cuando se escribe, se escribe en el offset para el fichero abierto y un efecto lateral de escribir *n* bytes es que el offset aumenta en *n* unidades. Igual sucede en `read`. Cuando se lee, se lee desde el offset que UNIX mantiene para el fichero abierto. Si se leen *n* bytes, el offset aumenta en *n*. Dicho de otro modo, los fichero se leen y escriben **secuencialmente** utilizando `read` y `write`. Si vuelves a mirar la salida de *lsof* que mostramos anteriormente, verás que una de las columnas indica cuál es el offset para cada descriptor.

Por ejemplo, considera este programa:

```
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <err.h>
#include <string.h>

int
main(int argc, char* argv[])
{
    int    fd, i, n;

    fd = open("afile", O_WRONLY|O_CREAT, 0644);
    if (fd < 0) {
        err(1, "afile");
    }
    for (i = 1; i < argc; i++) {
        n = strlen(argv[i]);
        if (write(fd, argv[i], n) != n) {
            close(fd);
            err(1, "write");
        }
    }
    if (close(fd) < 0) {
        err(1, "close");
    }
    exit(0);
}
```

El programa escribe sus argumentos en el fichero `afile`. La llamada a `open` abre el fichero `afile` para

escribir en el (`O_WRONLY`). Como verás, el segundo argumento son flags en un único entero. Se utiliza un *or* para activar los bits que se desean en el entero. `O_CREAT` indica que si no existe queremos crear el fichero. En dicho caso, cuando el fichero no existe y se crea, el tercer argumento indica qué permisos queremos para el nuevo fichero.

Vamos a ejecutarlo por primera vez tras comprobar que `afile` no existe.

```
unix$ ls -l afile
ls: afile: No such file or directory
unix$ writef aa bb cc
```

Si inspeccionamos `afile` veremos que tiene 9 bytes

```
unix$ ls -l afile
-rw-r--r--  1 nemo  staff  9 Aug 20 15:49 afile
```

y que su contenido son los argumentos de `writef` (con un fin de línea añadido tras cada uno):

```
unix$ cat afile
aa
bb
cc
```

Podemos utilizar *xd(1)* para ver los bytes y caracteres que contiene el fichero:

```
unix$ xd -b -c afile
0000000  61 61 0a 62 62 0a 63 63 0a
      0   a  a \n  b  b \n  c  c \n
0000009
```

La primera columna indica el offset en que aparecen los bytes que *xd* muestra a continuación. Los fines de línea son bytes con el valor `0xa` (`10` en decimal). Como verás, hemos hecho seis writes y se han escrito secuencialmente en el fichero. Esto quiere decir que el primer `write` ha escrito en la posición `0` del fichero: ha utilizado el offset `0`. El segundo `write` ha escrito un "`\n`" tras escribir el primer argumento, y ha escrito a partir del offset `2` en el fichero. El tercer `write` ha escrito el segundo argumento en la posición `3`. Y así sucesivamente.

Por cierto, el tamaño del fichero es `9` puesto que el mayor offset escrito ha sido el `8` y empezamos a contar en `0`. Luego el fichero tiene `9` bytes escritos en total. Más allá no hemos escrito nunca. Dicho de otro modo, el tamaño del fichero está determinado por hasta dónde hemos escrito en el fichero.

Si ejecutamos el programa una segunda vez, veremos mejor qué supone utilizar un offset.

```
unix$ writef x y
unix$ ls -l afile
-rw-r--r--  1 nemo  staff  9 Aug 20 15:49 afile
unix$ cat afile
x
y
b
cc
unix$ xd -b -c afile
0000000  78 0a 79 0a 62 0a 63 63 0a
      0   x \n  y \n  b \n  c  c \n
0000009
```

Lo primero que vemos es que ¡el fichero sigue teniendo 9 bytes!. Dicho de otro modo, aunque hemos

escrito más veces, el tamaño no ha aumentado. El primer argumento se ha escrito al principio, lo que quiere decir que el primer `write` ha utilizado un offset `0` y ha escrito un byte. El segundo `write` ha escrito el fin de línea en el offset `1`, dado que el primer write escribió un byte. Etcétera.

Otro detalle curioso es que después del texto escrito ("`x\ny\n`") puedes ver que el fichero contiene "`b\ncc\n`". Lo único que sucede es que esos bytes se escribieron la vez anterior que ejecutamos `writef` y esta vez no los hemos vuelto a escribir, luego tienen el mismo valor que tenían. Recuerda que `write` **no** inserta, `write` (re)escribe.

Y una cosa más. Ahora el fichero tiene 4 líneas, pero no obstante su tamaño sigue siendo 9 bytes. Aunque tenga más líneas, el fichero no es mas grande que antes. Que tenga más líneas se debe simplemente a que hay más "`\n`" escritos en el fichero.

Para ver estas cosas te puede resultar cómodo utilizar *wc(1)*, que cuenta líneas, palabras y caracteres en un fichero, o en la entrada si no indicas fichero alguno.

```
unix$ wc afile
       4       4       9 afile
unix$
```

¿Dónde está el offset? La figura 1 muestra qué aspecto tiene la estructura de datos que usa UNIX.

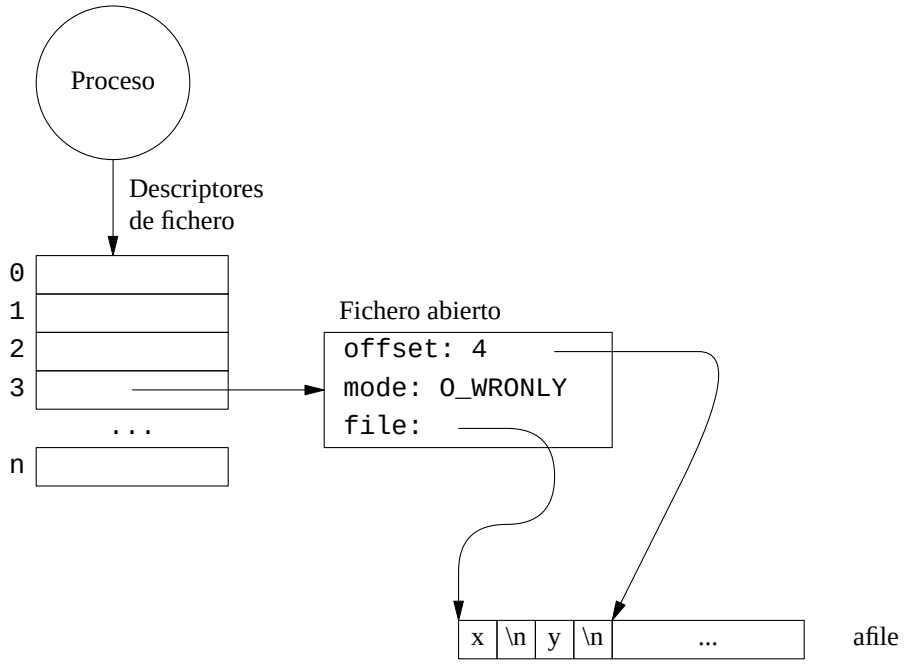

Figura 1: *El offset para lecturas y escrituras se guarda fuera del descriptor de fichero, en una entrada en la tabla de ficheros abiertos.*

En cada descriptor de fichero hay un puntero hacia un record (como siempre). En dicho record, llamado *fichero abierto* por UNIX y almacenado en una *tabla de ficheros abiertos* se guarda:

- El offset en que hay que leer/escribir.
- El modo en que se abrió el fichero (lectura, escritura, o lectura/escritura).
- El puntero hacia la estructura de datos que representa el fichero en UNIX.

En la figura se ve qué valor tenía el contenido de `afile` y el `offset` del fichero abierto justo al final de

ejecutar por segunda vez `afile`, antes de llamar a `close`.

Si queremos que cada vez que ejecutemos `writef` el fichero `afile` quede justo con lo que hemos escrito, la solución es eliminar el contenido de `afile` cuando el fichero ya existe, durante `open`. Esto es, si el fichero no existe lo creamos y si existe lo *truncamos* a 0-bytes. El flag `O_TRUNC` de `open` consigue este efecto. Basta utilizar

```
fd = open("afile", O_WRONLY|O_TRUNC|O_CREAT, 0644);
```

en lugar de la llamada a `open` que hacíamos antes. La próxima vez que ejecutemos `writef` no quedarán restos del contenido que `afile` pudiera tener antes si es que existía. Aunque claro, si has leído *open(2)* ya lo sabías.

Existe otro flag que podemos utilizar con *open(2)* que afecta al offset que utiliza `write`. Se trata del flag `O_APPEND`. Si indicamos `O_APPEND` en `open`, las llamadas a `write` ignoran el offset del fichero y escriben siempre al final del mismo. En realidad, se escribe justo en la posición indicada por el tamaño del fichero. Pero cuidado aquí si el fichero es un fichero compartido en red: cada UNIX que tiene abierto el fichero podría tener su propia idea del tamaño del mismo (por ejemplo si un UNIX acaba justo de hacer crecer el fichero pero otro no se ha dado cuenta todavía).

2. Ajustando el offset

Para evitar errores utilizando `read` y `write` basta con que pienses que UNIX es muy obediente y hace justo lo que se supone que cada llama hace. No hay magia. Si recuerdas lo que hemos visto, podrás entender lo que ocurre. ¡Vamos a comprobarlo!

Hay otra llamada, *lseek(2)*, que permite cambiar el offset de un descriptor de fichero. Bueno, en realidad, queremos decir "*el offset de un fichero abierto al que apunta un descriptor*". Se le suministra el descriptor de fichero, cuantos bytes queremos mover el offset y desde donde contamos. Por ejemplo,

- `lseek(fd, 10, SEEK_SET)` fija el offset a 10 en `fd`.
- `lseek(fd, 10, SEEK_CUR)` añade 10 al offset de `fd`.
- `lseek(fd, 10, SEEK_END)` fija el offset 10 bytes contando desde el final del fichero hacia atrás en `fd`.

Las constantes `SEEK_SET`, `SEEK_CUR` y `SEEK_END` son simplemente `0`, `1` y `2`. De hecho, dado que desde el comienzo de UNIX han tenido esos valores, es muy poco probable que cambien nunca de valor.

Entendido esto, fíjate en este programa:

```
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <err.h>

int
main(int argc, char* argv[])
{
    int    fd;

    fd = open("afile", O_WRONLY|O_TRUNC|O_CREAT);
    if (fd < 0) {
        err(1, "open: afile");
    }
    lseek(fd, 32, 0);
    if (write(fd, "xx\n", 3) != 3) {
        close(fd);
        err(1, "write");
    }
    if (close(fd) < 0) {
        err(1, "close");
    }
    exit(0);
}
```

Ejecutándolo podemos ver lo que sucede con el nuevo `afile` en este caso.

```
unix$ seekwrite
unix$ ls -l afile
-rw-r--r--  1 nemo  staff  35 Aug 20 16:32 afile
unix$ cat afile
xx
unix$ xd -b -c afile
0000000  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
      0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000010  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
     10  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000020  78 78 0a
     20   x  x \n
0000023
```

¡Curioso!, ¿No? Resulta que *ls* dice que el fichero tiene 35 bytes. Pero sólo hemos escrito 3 bytes. Veamos...

- `lseek` ha cambiado el offset a 32, contando desde el comienzo del fichero.
- `write` ha escrito tres bytes en dicho offset.

Luego en total, el byte más alto que hemos escrito es el 34 (contando desde 0), lo que quiere decir que mirando los 35 primeros bytes tenemos el contenido del fichero que hemos escrito alguna vez. Además, puedes ver que los bytes que no hemos escrito nunca están a cero. Eso es cortesía de UNIX.

El otro detalle curioso es que cuando hemos hecho un *cat* del fichero sólo hemos visto una línea con `xx`, que era lo que cabría esperar. Pero en realidad, sí que `cat` ha escrito un total de 35 bytes en su salida:

```
unix$ cat afile >/tmp/out
unix$ ls -l /tmp/out
-rw-r--r--  1 nemo  wheel  35 Aug 20 16:37 /tmp/out
unix$
```

Lo que ocurre es que cuando *cat* escribe un byte a cero en su salida estándar, y esta es el terminal, el terminal no sabe cómo mostrar dicho byte (¿Cómo dibujar un símbolo para el byte nulo?) y no lo escribe en absoluto. Es normal, `/dev/tty` sólo espera que escribas texto en el teclado y que muestres texto en la pantalla. No espera efectos especiales.

Utilizar `lseek` como hemos hecho es muy habitual. Por ejemplo, para crear un fichero con 1GiB basta con que hagas un `lseek` a la posición `1024*1024*1024-1` y escribas un byte. El tamaño del nuevo fichero será justo de 1GiB. Lo que es más, es muy posible que UNIX no asigne espacio en disco para el nuevo fichero salvo para el último *bloque* que use el fichero (que es dónde has escrito el byte). Todos los trozos anteriores del fichero están sin escribir y, si se leen, UNIX sabe que se leen con todos los bytes a cero. Siendo esto así, ¿Para qué guardar los ceros en el disco si UNIX sabe que están a cero? A estos ficheros se les llama *ficheros con huecos*. ¡Puedes tener ficheros más grandes que el tamaño del disco en que los guardas!

Otro uso de `lseek` es para obtener el offset. Basta con cambiar el offset a 0 bytes contando a partir del offset actual, y utilizar el valor que retorna `lseek` (que es el nuevo offset). Por ejemplo:

```
off = lseek(fd, 0, 1);
```