Introducción a Sistemas Operativos: Padres e hijos

Clips xxx
Francisco J Ballesteros

1. ¿Compartidos o no?

Cuando `fork` crea un proceso, dado que es un clon del padre, dicho proceso (hijo) tiene una copia de los descriptores de fichero del padre. Lo mismo sucede con las variables de entorno y otros recursos.

Naturalmente, sólo los descriptores de fichero se copian, ¡no los ficheros!. Piensa lo absurdo que sería (además de ser imposible) copiar el disco duro entero si un proceso tiene abierto el dispositivo del disco y hace un `fork`. Ni siquiera se copian las entradas de la tabla de ficheros abiertos (los record a que apuntan los descriptores de fichero).

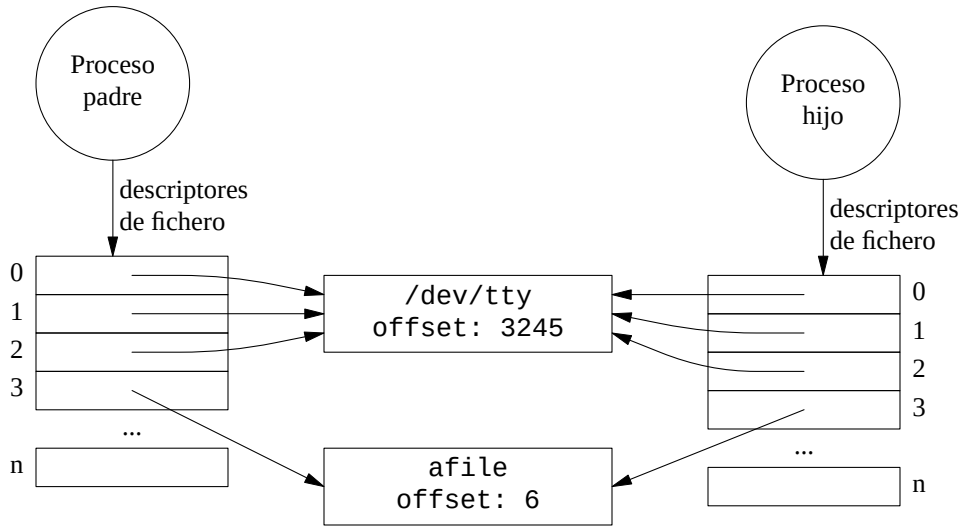

Figura 1: *Descriptores en los procesos padre e hijo tras un fork.*

La figura 1 muestra dos procesos padre e hijo tras una llamada a `fork`, incluyendo los descriptores de fichero de ambos procesos. Esta figura podría corresponder a la ejecución del siguiente programa (en el que hemos ignorado los valores devueltos por llamadas que hacemos para que el código sea más compacto, aunque hacer tal cosa es un error).

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

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

    fd = creat("afile", 0644);
    if (fd < 0) {
        err(1, "creat afile");
    }
    write(fd, "hello\n", 6);
    if(fork() == 0) {
        write(fd, "child\n", 6);
    } else {
        write(fd, "dad\n", 4);
    }
    close(fd);
    exit(0);
}
```

La efecto de ejecutar el programa podría ser este:

```
unix$ before
unix$ cat afile
hello
child
dad
unix$
```

Inicialmente, el padre tiene abierta la entrada estándar, la salida estándar y la salida de error estándar. Todas ellas van al fichero `/dev/tty`. En ese punto el padre abre el fichero `afile` (creándolo si no existe) y obtiene un nuevo descriptor (el `3` en nuestro caso, partiendo con offset `0`). Después de escribir 6 bytes en dicho fichero, el offset pasa a ser `6`.

Es ahora cuando `fork` crea el proceso hijo y vemos ambos procesos tal y como muestra la figura 1. Naturalmente, si cualquiera de los dos procesos abre un fichero en este punto, se le dará un nuevo descriptor al proceso que lo abre y el otro proceso no tendrá ningún nuevo descriptor. Los dos procesos son independientes y cada uno tiene si tabla de descriptores de fichero. Igualmente, si ambos abren el mismo fichero tras el `fork`, cada uno obtiene un descriptor que parte con el offset a `0`. Incluso si en ambos procesos el descriptor es, por ejemplo, `4`, los dos descriptores son distintos. ¿Puedes verlo?

Volviendo a nuestro programa, ambos procesos continúan y cada uno escribe su mensaje. Dado que comparten el (record que representa el) fichero abierto, comparten el offset. UNIX garantiza que writes pequeños en el mismo fichero (digamos de uno o pocos KiB) ejecutan atómicamente, o de forma indivisible sin que otros writes ejecuten durante el que UNIX está ejecutando. Así pues cuando el primer proceso haga su `write`, el offset avanzará y el segundo proceso encontrará el offset pasado el texto que ha escrito el primer proceso. Un mensaje se escribirá a continuación de otro.

Comparemos lo que ha sucedido con el efecto de ejecutar este otro programa:

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

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

    fd = creat("afile", 0644);
    if (fd < 0) {
        err(1, "creat afile");
    }
    close(fd);

    if(fork() == 0) {
        fd = open("afile", O_WRONLY);
        if (fd < 0) {
            err(1, "open afile");
        }
        write(fd, "child\n", 6);
    } else {
        fd = open("afile", O_WRONLY);
        if (fd < 0) {
            err(1, "open afile");
        }
        write(fd, "dad\n", 4);
    }
    close(fd);
    exit(0);
}
```

Esta vez...

```
unix$ after
unix$ cat afile
dad
d
unix$ xd -b -c afile
0000000  63 68 69 6c 64 0a
      0   d  a  d \n  d \n
0000006
unix$
```

¿Por qué? Simplemente porque cada proceso tiene su propio descriptor de fichero con su propio offset. Podríamos pensar que cada vez que abrimos un fichero nos dan un offset. En el programa anterior lo compartían ambos procesos, pero no esta vez. La consecuencia es que ambos procesos realizan el `write` en el offset `0`, con lo que el primero en hacer el `write` escribirá antes en el fichero. El segundo en hacerlo sobreescribirá lo que escribiese el primer proceso. En nuestro caso, como el padre parece que ha escrito después y su escritura era de menos bytes, quedan restos de la escritura del hijo a continuación del mensaje que ha escrito el padre. Ten en cuenta que aunque `write` en el hijo avanza el offset, avanza el offset en el fichero que ha abierto el hijo. Pero esta vez el padre tiene su propio offset que todavía sigue siendo `0` cuando llama a `write`.

Otra posibilidad habría sido ver esto...

```
unix$ after
unix$ cat afile
child
unix$
```

¿Es que el padre no ha escrito nada en este caso?

Si recuerdas que en *open(2)* puedes utilizar el flag `O_APPEND` comprenderás que en este programa podríamos haberlo utilizado para hacer que los writes siempre se realicen al final de los datos existentes en el fichero en lugar de en la posición que indica el offset. Pero no hemos hecho tal cosa.

1.1. Condiciones de carrera

Lo que acabamos de ver es realmente importante. Aunque el programa es el mismo, dado que hay más de un proceso involucrado, el resultado de la ejecución depende del orden en que ejecuten los distintos procesos. Concretamente, en el orden en que se ejecuten sus trozos de código (piensa que en cualquier momento UNIX puede hacer que un proceso abandone el procesador y que otro comience a ejecutar, esto es, en cualquier momento puede haber un cambio de contexto).

A esta situación se la denomina **condición de carrera**, y normalmente es un bug. No es un bug sólo si no nos importa que el resultado varíe, lo que no suele ser el caso.

Estamos adentrándonos en un mundo peligroso, llamado *programación concurrente*. La programación concurrente trata de cómo programar cuando hay múltiples procesos involucrados y dichos procesos comparten recursos. Es justo ese el caso en que pueden darse condiciones de carrera. Recuerda que decimos "concurrente" puesto que nos da exactamente igual si lo que sucede es que el sistema cambia de contexto de un proceso a otro o que los procesos ejecutan realmente en paralelo en distintos cores.

Los programas con condiciones de carrera son impredecibles y muy difíciles de depurar. Es mucho más práctico tener cuidado a la hora de programarlos y evitar que puedan suceder condiciones de carrera. Más adelante veremos algunas formas de conseguirlo.