Introducción a Sistemas Operativos: Padres e hijos

Clips xxx
Francisco J Ballesteros

1. Creación de procesos

La llamada al sistema `fork(2)` crea un *clone exacto* del proceso que hace la llamada. Pero, ¿qué significa esto? Experimentemos con un nuevo programa:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int
main(int argc, char* argv[])
{
    write(1, "one\n", 4);
    fork();
    write(1, "fork\n", 5);
    exit(0);
}
```

No hemos comprobado errores (¡mal hecho!), pero esta es la salida:

```
unix$ onefork
one
fork
fork
unix$
```

El primer `write` ejecuta y vemos `one` en la salida. Pero después, llamamos a `fork`, lo que crea otro proceso que es un clon exacto y, por tanto, están también dentro de la llamada a `fork`. Ambos procesos (padre e hijo) continúan desde ese punto y, claro, llamarán al segundo `write` de nuestro programa. Pero, naturalmente, los *dos* llaman a `write`, por lo que vemos dos veces `fork` en la salida del programa. Una pregunta que podemos hacernos es... ¿será el primer `fork` que vemos el escrito por el padre o será el escrito por el hijo? ¿Qué opinas al respecto?

La figura 1 muestra un ejemplo de ejecución para ambos procesos en puntos diferentes del tiempo (que fluye hacia abajo en la figura).

Si seguimos la figura desde arriba hacia abajo vemos que inicialmente sólo existe el padre. Las flechas representan el contador de programa y vemos que el padre está ejecutando primero la primera llamada a `write` del programa. Después, el padre llama a `fork` y ¡aparece un nuevo proceso hijo! Cuando `fork` termina su trabajo, tanto el proceso padre como el hijo están retornando de la llamada a `fork`. Esto es lógico si piensas que el hijo es una copia exacta del padre en el punto en que llamó a `fork`, y esa copia incluye también la pila (no sólo los segmentos de código y datos).

Así pues, ambos procesos retornan del mismo modo y aparentan haber llamado a `fork` del mismo modo (aunque el hijo nunca ha hecho ninguna llamada a `fork`). Aunque UNIX hace que en el hijo `fork` retorne

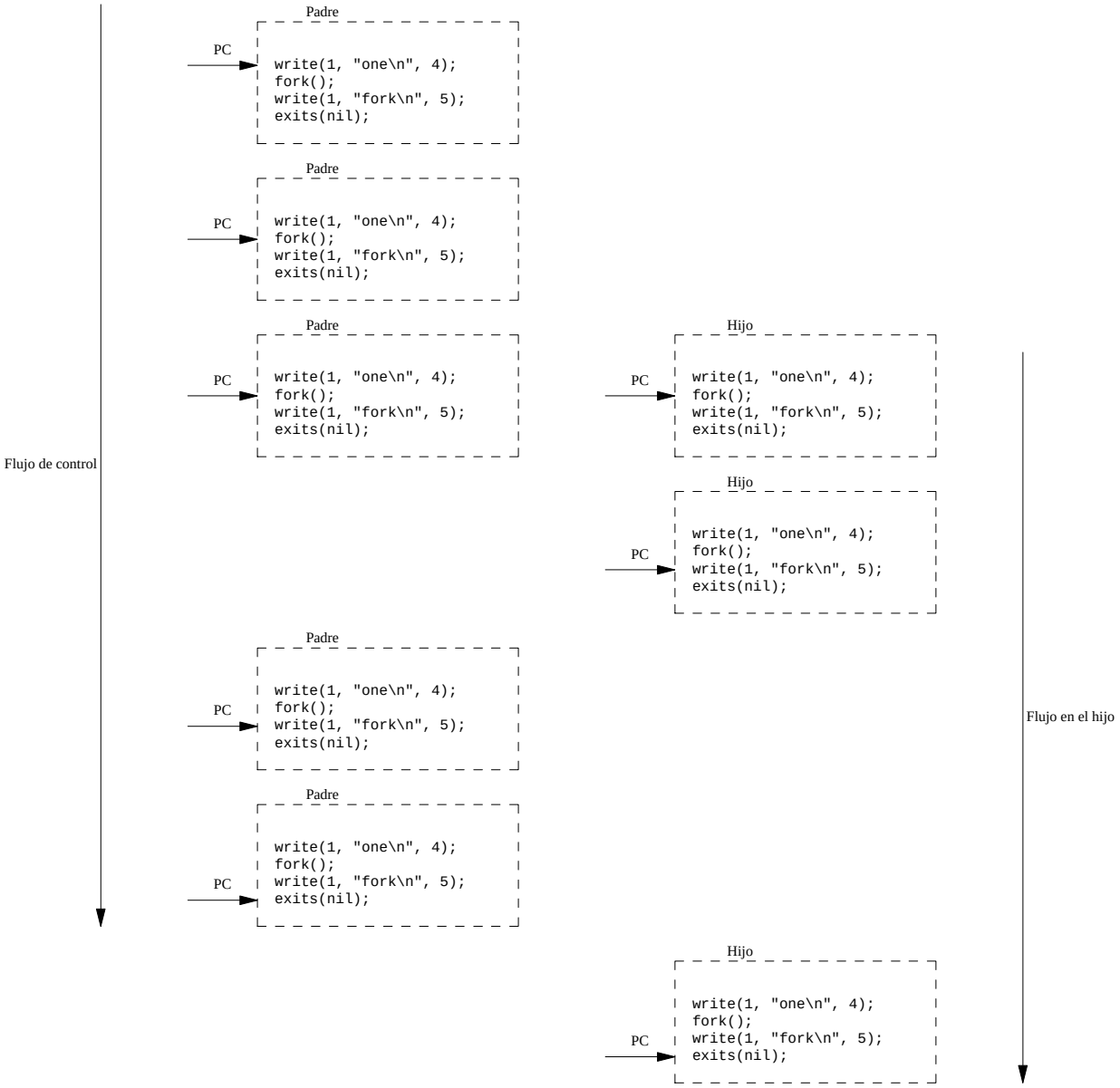

Figura 1: *La llamada a fork crea un clon del proceso original y ambos continúan su ejecución desde ese punto.*

siempre `0`, lo que no importa en este programa. En la figura parece que el hijo ejecuta después su segundo `write` y entonces el padre continúa hasta que termina. A continuación el hijo ejecuta el código que le queda por ejecutar antes de terminar.

Naturalmente, desde el punto en que se llama a `fork`, padre e hijo pueden ejecutar en cualquier orden (o incluso de forma realmente paralela si disponemos de varios cores o CPUs en la máquina). Esto es precisamente lo que hace la abstracción *proceso*: nos permite pensar que cada proceso ejecuta independientemente del resto del mundo.

Nunca has pensado en el código del shell o el del sistema de ventanas o en ningún otro cuando has escrito un programa. Siempre has podido suponer que tu programa comienza en su programa principal y continúa según le dicte el código de forma independiente a todos los demás. Igual sucede aquí. Todo ello es gracias a la abstracción que suponen los procesos. Puedes pensar que una vez que llamamos a `fork` y se crea un

proceso hijo, el hijo abandona la casa inmediatamente y continúa la vida por su cuenta.

1.1. Las variables

Dado que el proceso hijo es una copia, no comparte variables con el padre. El segmento de datos en el hijo (y la pila) son una copia de los del padre, igual que sucede con el segmento BSS y todo lo demás. Así pues, después de `fork` tu programa vive en dos procesos y cada uno tiene su propio valor para cada variable. El flujo de control (los registros y la pila) también se divide en dos (uno para cada proceso), de ahí el nombre "`fork`" ("tenedor" en inglés).

Veamos otro programa:

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

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


    p = strdup("hola");
    n = 0;
    switch(fork()){
    case -1:
        err(1, "fork failed");
        break;
    case 0:
        p[0] = 'b';
        break;
    }
    n++;
    printf("pid %d: n=%d; %s at %p\n", getpid(), n, p, p);
    free(p);
    exit(0);
}
```

Intenta pensar cuál puede ser su salida y por qué antes de que lo expliquemos.

Las variables `n` y `p` están en la pila del proceso padre. La primera se inicializa a `0` y la segunda se inicializa apuntando a memoria dinámica (que está dentro de un segmento de datos, sea este el BSS o un segmento *heap* dependiendo del sistema UNIX). En dicha memoria `strdup` copia el string "`hola`".

Una vez hecho el `fork`, el proceso hijo hace que la posición a la que apunta `p` contenga `'b'`. Tras el `switch`, ambos procesos incrementan (su version de) `n`. La direcciones que que está `n` en ambos procesos coinciden (tienen el mismo valor). Pero cada proceso tiene su propia memoria virtual y su propia copia del segmento de pila. Igualmente, desde la llamada a `fork`, aunque `p` tiene el mismo valor en ambos procesos, la memoria a la que apunta `p` en el proceso hijo es distinta a la que tiene el proceso padre (¡Aunque las direcciones de memoria virtual sean las mismas!).

¿Entienes ahora por qué la salida es como sigue?

```
unix$ onefork2
pid 13083: n=1; hola at 0x7fd870c032a0
pid 13084: n=1; bola at 0x7fd870c032a0
unix$
```

Dado que *cada* proceso ha incrementado su variable `n`, ambos escriben `1` como valor de `n`. Además, los strings a que apunta `p` en cada proceso difieren, aunque las direcciones de memoria en que están en cada proceso coincidan.

Habitualmente se utiliza un `if` o `switch` justo tras la llamada a `fork` para que el código del proceso hijo haga lo que sea que tenga que hacer el hijo y el padre continúe con su trabajo. Ya dijimos que en el hijo `fork` devuelve siempre 0. En el padre `fork` devuelve el *pid* del hijo, que puede usarse para identificar qué proceso se ha creado y para diferenciar la ejecución del padre de la del hijo en el código que escribimos. Por ejemplo,

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

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

    write(1, "first\n", 7);
    pid = fork();
    switch(pid) {
    case -1:
        err(1, "fork");
        break;
    case 0:
        printf("child pid %d\n", getpid());
        break;
    default:
        printf("parent pid %d child %d\n", getpid(), pid);
    }
    printf("last\n");
    exit(0);
}
```

escribe al ejecutar

```
unix$
first
parent pid 13172 child 13173
last
child pid 13173
last
unix$
```

¿En qué otro orden pueden salir los mensajes?

1.2. El efecto de las cachés

Vamos a reescribir ligeramente uno de los programas anteriores y ver qué sucede. Concretamente, utilizaremos *stdio* en lugar de *write(2)* para escribir mensajes. Este es el programa

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int
main(int argc, char* argv[])
{
    printf("one\n");
    fork();
    printf("fork\n");
    exit(0);
}
```

Y esta es la salida

```
unix$ stdiofork
one
fork
one
fork
unix$
```

¿Qué sucede? ¿Por qué hay dos "`one` en la salida? Según entendemos lo que hace `fork`... ¿No debería salir el mensaje una única vez?

Bueno, en realidad... ¡No!. Como sabemos, `printf` escribe utilizando un `FILE*` que dispone de buffering. No tenemos garantías de que `printf` llame a `write` en cada ocasión. Tan sólo cuando el buffer se llena o la implementación de `printf` decide hacerlo se llamará a `write`.

En nuestro programa, los bytes con "`one\n`" están en el buffer de `stdout` en el momento de llamar a `fork`. En este punto, `fork` crea el proceso hijo como un *clon exacto*. Luego el hijo dispone naturalmente de los mismos segmentos de datos que el padre y el buffer de `stdout` tendrá el mismo contenido en el hijo que en el padre.

Así pues, cuando *stdio* llame a `write` para escribir el contenido del buffer, ambos mensajes aparecen en el terminal, en cada uno de los dos procesos.

2. Juegos

Este programa es curioso:

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

int
main(int argc, char* argv[])
{
    while(fork() == 0)
        ;    // catch me!
    exit(0);
}
```

El proceso padre llama a `fork` y luego muere (dado que para él `fork` devuelve `0` lo que hace que el bucle termine). No obstante, el proceso hijo continúa en el bucle y llama a `fork`. Esta vez, el hijo termina tras crear un nieto. Y así hasta el infinito. Es realmente difícil matar este programa dado que cuando estemos intentando matar al proceso, este ya habrá muerto tras encarnarse en otro.

Este otro programa es aún peor.

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

int
main(int argc, char* argv[])
{
    while(1) {
        fork();
    }
    exit(0);
}
```

Un proceso crea otro. Ambos continúan en el bucle y cada uno de ellos crea otro. Los cuatro continúan...

¡Pruébalo! (y prepárate a tener que rearrancar el sistema cuando lo hagas).