Introducción a Sistemas Operativos: Padres e hijos

Clips xxx
Francisco J Ballesteros

1. Ejecutando un nuevo programa

Hemos visto antes cómo es el proceso que ejecuta nuestro código. UNIX ha creado este proceso cuando se lo hemos pedido utilizando el shell y, hasta el momento, sólo hemos utilizado el shell para crear nuevos procesos.

Vamos a ver ahora cómo crear nuevos procesos y ejecutar nuevos programas pidiéndoselo a UNIX directamente. Aunque en otros sistemas tenemos llamadas similares a

```
spawn("/bin/ls");
```

para ejecutar `ls` en un nuevo proceso, ese *no* es el caso en UNIX. En su lugar, tenemos dos llamadas:

- Una sirve para crear un nuevo proceso
- Otra sirve para ejecutar un nuevo programa.

Las razones principales para esto es que podríamos querer un nuevo proceso que ejecute el mismo programa que estamos ejecutando y que podríamos querer configurar el entorno para un nuevo programa en un nuevo proceso antes de cargar dicho programa.

Antes de ver dichas llamadas detenidamente, veamos un ejemplo completo. En este programa utilizamos *fork(2)* para crear un nuevo proceso y hacemos que dicho proceso ejecute `/bin/ls` mediante una llamada a *execl(3)*:

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

int
main(int argc, char* argv[])
{
    switch(fork()){
    case -1:
        err(1, "fork failed");
        break;
    case 0:
        execl("/bin/ls", "ls", "-l", NULL);
        err(1, "exec failed");
        break;
    default:
        printf("ls started\n");
    }
    exit(0);
}
```

El programa empieza su ejecución como cualquier otro proceso y continúa hasta la llamada a `fork`. En este punto sucede algo curioso: se crea un *clon exacto del proceso* y tanto el proceso original (llamado *proceso padre*) como el nuevo proceso (llamadado *proceso hijo*) continúan su ejecución normalmente a partir de dicha llamada. Dicho de otro modo,

- hay una única llamada a `fork` (en el proceso padre),
- pero `fork` retorna dos veces: una vez en el proceso padre y otra en el hijo.

Ambos procesos son totalmente independientes, y ejecutarán según obtengan procesador (no sabemos en qué orden).

En el proceso padre `fork` retorna un número positivo (a menos que `fork` falle, en cuyo caso retorna `-1`). Luego el padre continúa su ejecución en el `default`, imprime su mensaje y luego termina en la llamada a `exit`.

En el proceso hijo `fork` siempre retorna `0`, con lo que el hijo entra en el `case` para `0` y ejecuta `execl`. Esta llamada *borra* por completo el contenido de la memoria del proceso hijo y carga un nuevo programa desde `/bin/ls`, saltando a la dirección de memoria en que está su punto de entrada (`main` para `ls`) y utilizando una pila que tiene argumentos `argc` y `argv` para dicha llamada *copiados* a partir de los que se han suministrado a `execl`.

Si todo va bien, `execl` *no retorna*. ¡Normal!, el programa original que hizo la llamada ya no está y no hay nadie a quién retornar. Estamos ejecutando un nuevo programa desde el comienzo, y este terminará cuando llame a `exit` (o `main` retorne y se llame a `exit`).

Si ejecutamos el programa, podemos ver una salida similar a esta:

```
unix$
ls started
total 112
-rw-r--r--  1 nemo  staff    10 Oct 21  2014 afile
-rw-r--r--  1 nemo  staff  1018 Oct 28  2014 guide
-rw-r--r--  1 nemo  staff   363 Aug 25 12:11 runls.c
-rwxr-xr-x  1 nemo  staff  8600 Aug 25 12:11 runls
unix$
```

La pregunta es... ¿tendremos siempre esta salida? Piensa que son procesos independientes, así pues ¿no podría aparecer el mensaje "`ls started`" del proceso padre en otro sitio? Piénsalo.