Introducción a Sistemas Operativos: Padres e hijos

Clips xxx
Francisco J Ballesteros

1. Todo junto

La mayoría de las veces no vamos a llamar a `exec` (`execl`, `execv`, ...) en el proceso que ejecuta nuestro programa. Normalmente creamos un proceso y utilizamos dicho proceso para ejecutar un programa dado. Bueno... aunque *login(1)* llama directamente a `exec`. Lo hace tras preguntar un nombre de usuario y comprobar que su password es correcto, ajusta el entorno del proceso y hace un `exec` del shell del nuevo usuario (que ejecuta a nombre del usuario que ha hecho *login* en el sistema).

¡Cuidado! Aquellos que no saben utilizar UNIX algunas veces hacen un `exec` de un comando de shell por no saber utilizar el manual y no saber que existe una función en C que hace justo lo que querían hacer. No tiene sentido utilizar `fork`, `exec`, y *date(1)* para imprimir la fecha actual. Basta una línea de C si sabes leer *gettimeofday(2)* y *ctime(3)*, como hemos visto antes. Recuerda que cuando buscas código en internet no puedes saber si lo ha escrito un humano o un simio. Tu eres siempre responsable del código que incluyes en tus programas.

En cualquier caso, vamos a programar una función en C que nos permita ejecutar un programa en otro proceso dado el path de su ejecutable y su vector de argumentos. La cabecera de la función podría ser algo como

```
int run(char *path, char *argv[])
```

Haremos que devuelva `-1` si falla y `0` si ha conseguido hacer su trabajo, como suele ser costumbre.

Esta es nuestra primera versión:

```
int
run(char *path, char *argv[])
{
    switch(fork()){
    case -1:
        return -1;
    case 0:
        execv(path, argv);
        err(1, "exec %s failed", cmd);
    default:
        return 0;
    }
}
```

El proceso hijo llama a `execv` (dado que tenemos un vector, `execl` no es adecuado) y termina su ejecución si dicha llamada falla. No queremos que el hijo retorne de `run` en ningún caso. ¡Un sólo flujo de control ejecutando código en el padre es suficiente!

El proceso padre retorna tras crear el hijo, aunque esto es un problema. Lo deseable sería que `run` no termine hasta que el programa que ejecuta el proceso hijo termine. Lo que necesitamos es una forma de esperar a que un proceso hijo termine, y eso es exactamente lo que vamos a ver a continuación.

2. Esperando a un proceso hijo

La llamada al sistema *wait(2)* se utiliza para esperar a que un hijo termine. Además de esperar, la llamada retorna el valor que suministró dicho proceso en su llamada a *exit(3)* (su *exit status*). Luego podemos utilizarla tanto para esperar a que nuestro nuevo proceso termine como para ver qué tal le fué en su ejecución. Ya sabemos que el convenio en UNIX es que un estatus de salida `0` significa "todo ha ido bien" y que cualquier otro valor indica "algo ha ido mal".

Vamos a mejorar nuestra función, ahora que sabemos qué utilizar.

```
int
run(char *cmd, char *argv[])
{
    int pid, sts;

    pid = fork();
    switch(pid){
    case -1:
        return -1;
    case 0:
        execv(cmd, argv);
        err(1, "exec %s failed", cmd);
    default:
        while(wait(&sts) != pid)
            ;
        if (sts != 0) {
            return -1;
        }
        return 0;
    }
}
```

En esta versión, el proceso padre llama a `wait` hasta que el valor devuelto concuerde con el *pid* del hijo, y en ese caso el entero `sts` que ha rellenado la llamada a `wait` contiene el estatus del hijo.

El bucle en la llamada a `wait` es preciso puesto que, si nuestro proceso ha creado otros procesos antes de llamar a `wait` dentro de `run`, no tenemos garantías de que `wait` informe del proceso que nos interesa.

La llamada a `wait` espera hasta que *alguno* de los procesos hijo ha muerto y retorna con el pid y estatus de dicho hijo. Si ningún hijo ha muerto aún, `wait` se bloquea hasta que alguno muera. Y si no hay ningún proceso hijo creado... ¡Nos mereceremos lo que nos pase!

El programa que llame a `run` sólo estará interesado en si `run` ha podido hacer su trabajo o no. Por eso, si el estatus del hijo indica que el programa que ha ejecutado no ha podido hacer su trabajo, `run` retorna `-1`.

2.1. Zombies

Cuando un proceso muere en UNIX, el kernel debe guardar su estatus de salida hasta que el proceso padre hace un `wait` y el kernel puede informarle de la muerte del hijo.

¿Qué sucede si el padre nunca hace la llamada a `wait` para esperar a ese hijo? Simplemente que UNIX debe mantener en el kernel la información sobre el hijo que ha muerto. A partir de aquí, lo que ocurra dependerá el sistema concreto que utilizamos. En principio, la entrada en la tabla de procesos sigue ocupada para almacenar el estatus del hijo, por lo que tenemos un proceso (muerto) correspondiente al hijo. Pero dado que el hijo ha muerto, nunca volverá a ejecutar.

A estos procesos se los conoce como *zombies*, dado que son procesos muertos que aparecerán en la salida de *ps(1)* si el sistema que tenemos se comporta como hemos descrito. Una vez el padre llame a `wait`, UNIX podrá informarle respecto al hijo y la entrada para el hijo en la tabla de procesos quedará libre de nuevo. El zombie desaparece.

En otros sistemas el kernel mantiene en la entrada de la tabla de procesos del padre la información de los hijos que han muerto. En este caso, aunque técnicamente no tenemos un proceso zombie, el kernel sigue manteniendo recursos que no son necesarios si no vamos a llamar a `wait` en el padre.

Esta relación padre-hijo es tan importante en UNIX que cuando un proceso muere sus hijos suele adoptarlos el proceso con pid `1` (conocido como *init* habitualmente). Dicho proceso se ocupa de llamar a `wait` para que dichos procesos puedan por fin descansar en paz.

Lo importante para nosotros es que si nuestro programa crea procesos hemos de llamar a `wait` para esperarlos, o informar a UNIX del hecho de que no vamos a llamar a `wait` en ningún caso. Esto último se hace utilizando la llamada:

```
signal(SIGCHLD, SIG_IGN);
```

Aunque esta llamada no tiene nada que ver con la creación o muerte de procesos, así es como son las cosas. Más adelante veremos qué es *signal(3)* en realidad y para qué se utiliza.

3. Ejecución en background

Anteriormente hemos utilizado "`&`" en el shell, para ejecutar un comando y recuperar la línea de comandos (obtener un nuevo prompt) antes de que dicho comando termine. Como ya sabrás en este punto, para implementar "`&`" no es preciso ejecutar nada en el programa que implementa el shell. De hecho, hay que *no ejecutar* algo. Concretamente, basta con que el shell no llame a `wait` tras el `fork` que crea el proceso para el nuevo comando.

El comando *wait(1)* es un *built-in* del shell y espera hasta que los comandos que aún quedan por terminar terminen. Por ejemplo...

```
unix$ sleep 5 & echo hola ; wait
[1] 13796
hola
[1]+  Done                    sleep 5
unix$
```

y aparece "`hola`" en la salida en el acto, pero el prompt para un nuevo comando aparece 5 segundos después, cuando *wait(1)* ha terminado tras esperar que `sleep` termine.