# Introducción a Sistemas Operativos: Padres e hijos

*Clips xxx*
*Francisco J Ballesteros*

## 1. Cargando un nuevo programa

Ya sabemos cómo crear un proceso. Ahora necesitamos poder cargar nuevos programas o estaremos condenados a implementar *todo* cuanto queramos ejecutar en un único programa. Naturalmente, no se hacen así las cosas.

Para cargar un nuevo programa basta con utilizar la llamada al sistema `execl`, o una de las variantes descritas en *exec(3)*. Esta llamada recibe:

- El nombre (path) de un fichero que contiene el ejecutable para el nuevo programa
- Un vector de argumentos para el programa (`argv` para su `main`)

y, opcionalmente, dependiendo de la función de *exec(3)* que utilicemos,

- Un vector de variables de entorno.

Normalmente se utiliza o bien `execl` o bien `execv`. La primera acepta el vector de argumentos como argumentos de la función, por lo que se utiliza si al programar ya sabemos cuántos argumentos queremos pasarle al nuevo programa (si se conocen en tiempo de compilación, o *de forma estática*). La segunda acepta un vector de strings para el vector de argumentos y suele utilizarse si queremos construir un vector de argumentos en tiempo de ejecución o si resulta más cómo utilizar el vector que escribir un argumento tras otro en la llamada.

Veamos un programa con `execl`:

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

int
main(int argc, char* argv[])
{
    fprintf(stderr, "running ls\n");
    execl("ls", "ls", "-l", NULL);
    fprintf(stderr, "trying again\n");
    execl("/bin/ls", "ls", "-l", NULL);
    fprintf(stderr, "exec is done\n");
    exit(0);
}
```

Para que los mensajes salgan inmediatamente, el programa escribe en `stderr` (que no posee buffering) y así podemos utilizar `fprintf`. De nuevo, igual que en muchos ejemplos de los que siguen, hemos omitido las comprobaciones de error para hacer que los programas distraigan menos de la llamada con la que estamos experimentando.

Pero vamos a ejecutarlo...

```
unix$ execls
running ls
trying again
total 304
-rw-r--r--  1 nemo  staff     6 Aug 25 16:22 afile
-rwxr-xr-x  1 nemo  staff  8600 Aug 25 12:20 execls
-rw-r--r--  1 nemo  staff   363 Aug 25 12:11 execls.c
unix$
```

Claramente nuestro programa no ha leído ningún directorio ni lo ha listado. No hemos programado tal cosa. Es más, la mayoría de la salida claramente procede de ejecutar "`ls -l`". ¡Hemos ejecutado código de `ls` de igual modo que cuando ejecutamos "`ls -l`" en el shell!

Eso es exactamente lo que ha hecho la llamada a `execl`, cargar el código de `/bin/ls` en la memoria, tras lobotomizar el proceso y tirar el contenido de su memoria a la basura.

Mirando la salida más despacio, puede verse que el mensaje "`trying again`" ha salido en el terminal, pero no así el mensaje "`exec is done`". Esto quiere decir que la primera llamada a `execl` ha fallado: no ha ejecutado programa alguno y nuestro programa ha continuado ejecutando. El mero hecho de que `execl` retorne indica que ha fallado. Igual sucede con cualquiera de las variantes de *exec(3)*.

Vamos a cambiar ligeramente el programa para ver qué ha pasado:

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

int
main(int argc, char* argv[])
{
    fprintf(stderr, "running ls\n");
    execl("ls", "ls", "-l", NULL);
    fprintf(stderr, "trying again\n");
    execl("/bin/ls", "ls", "-l", NULL);
    fprintf(stderr, "exec is done\n");
    exit(0);
}
```

Y ahora sí podemos ver cuál fué el problema.

```
unix$ execls2
running ls
execls2: exec: ls: No such file or directory
trying again
total 304
...
```

No existe ningún fichero llamado "`./ls`" y naturalmente UNIX no ha podido cargar ningún programa desde dicho fichero dado que el primer argumento de `execl` (el path hacia el fichero que queremos cargar y ejecutar) es "`ls`" y no existe dicho fichero.

En la segunda llamada a `execl` resulta que hemos pedido que ejecute "`/bin/ls`" y UNIX no ha tenido problema en ejecutarlo: el fichero existe y tiene permiso de ejecución.

Inspeccionando el resto de argumentos de `execl` puede verse que la "linea de comandos" o, mejor dicho, el vector de argumentos para el nuevo programa está indicado tal cual como argumentos de la llamada. Dado que no hay magia, `execl` necesita saber dónde termina el "vector" y requiere que el último

argumento sea `NULL` para marcar el fin de los argumentos.

Pero probemos a ejecutar con otro vector de argumentos:

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

int
main(int argc, char* argv[])
{
    fprintf(stderr, "running ls\n");
    execl("ls", "ls", "-ld" "$HOME", NULL);
    err(1, "exec failed");
}
```

Ahora esta es la salida:

```
unix$ execls3
running ls
ls: $HOME: No such file or directory
unix$
```

Como puedes ver, `execl` *no* ha fallado: no puede verse el mensaje que imprimiría la llamada a `err`, con lo que `execl` no ha retornado nunca. Esto quiere decir que ha podido hacer su trabajo. Lo que es más, `ls` ha llegado a ejecutar y ha sido el que imprime el mensaje de error quejándose de que el fichero no existe.

¡Naturalmente!, ¡Claro que no existe "`$HOME`"! Si queremos ejecutar `ls` para que liste nuestro directorio casa, habría que llamar a `getenv` para obtener el valor de la variable de entorno `HOME` y pasar dicho valor como argumento en la llamada a `execl`".

Recuerda que `execl` no es el shell. Pero... si quieres el shell, ¡Ya sabes dónde encontrarlo! Este programa

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

int
main(int argc, char* argv[])
{
    fprintf(stderr, "running ls\n");
    execl("/bin/sh", "sh", "-c", "ls -l $HOME", NULL);
    err(1, "exec failed");
}
```

ejecuta una línea de comandos desde C. Simplemente carga el shell como nuevo programa y utiliza su opción "`-c`" para pasarle como argumento el "comando" que queremos utilizar. Claro está, el shell sí entiende "`$HOME`" y sabe qué hacer con esa sintáxis.

Piensa siempre que no hay magia y piensa con quién estás hablando cuando escribes código: ¿C?, ¿El shell?, ...