Introducción a Sistemas Operativos: Padres e hijos

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.

2. 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 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.

2.1. 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:

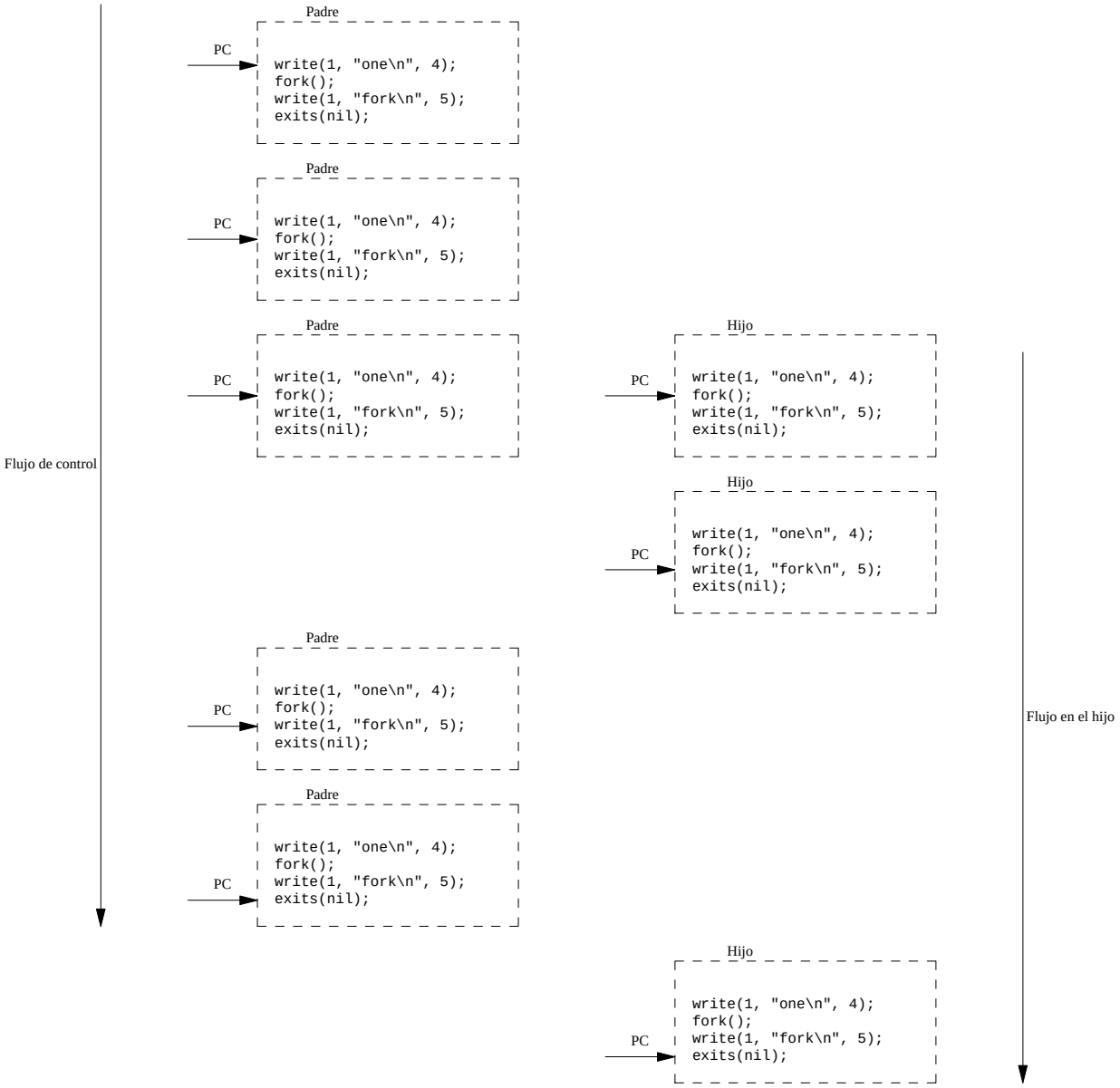

Figura 1: *La llamada a fork crea un clon del proceso original y ambos continúan su ejecución desde ese punto.*

```
#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?

2.2. 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.

3. 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).

4. ¿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).

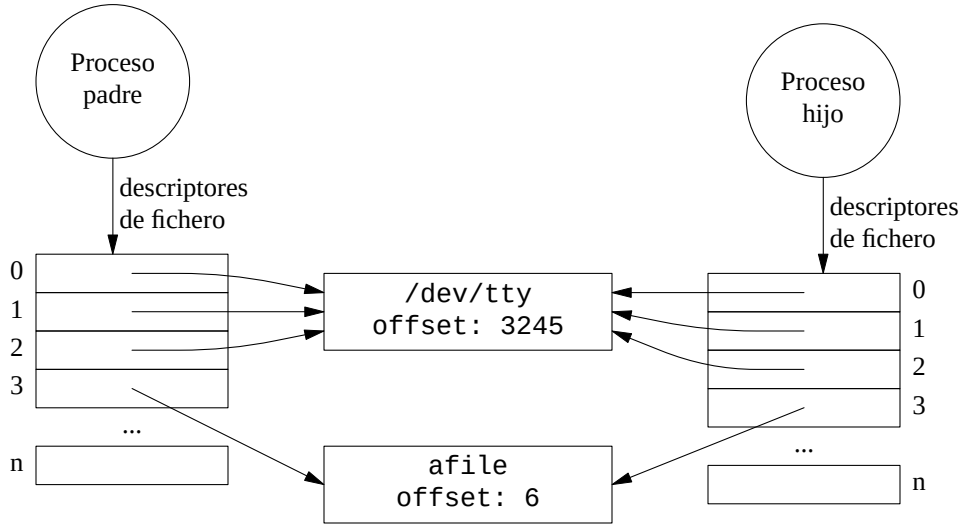

Figura 2: *Descriptores en los procesos padre e hijo tras un fork.*

La figura 2 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 2. 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.

4.1. 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.

5. 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?, ...

6. 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.

7. 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`.

7.1. 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.

8. 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.

9. Ejecutables

Para UNIX, un ejecutable es simplemente un fichero que tiene permiso de ejecución. UNIX es optimista e intentará ejecutar lo que se le pida, si es posible.

Durante la llamada al sistema `exec`, UNIX inspecciona el comienzo del fichero que ha de cargar para ejecución leyendo los primeros bytes. Dependiendo del contenido de dichos bytes pasará una cosa u otra.

9.1. Binarios

Consideremos de nuevo un ejecutable obtenido tras compilar y enlazar un "*hola mundo*" en C.

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

int
main(int argc, char *argv[])
{
    puts("hola mundo");
    exit(0);
}

unix$ cc -g hi.c
unix$ ls -l a.out
-rwxrwxr-x 1 elf elf 9654 Aug 26 08:38 a.out
```

El formato del fichero `a.out` dependerá mucho del tipo de UNIX que utilizamos. En general, es muy posible que sea un fichero en formato *ELF* (*Executable and Linkable Format*). No obstante, la estructura del fichero será prácticamente la misma en todos los casos:

- Una tabla al principio que indica el formato del fichero
- Una o más *secciones* con los bytes de código, datos inicializados, etc.

El comando *file(1)* en UNIX intenta determinar el tipo de fichero que tenemos entre manos. Simplemente lo lee y hace una apuesta, no hay garantías respecto a la mayoría de ficheros. Recuerda que para UNIX los ficheros son arrays de bytes y poco más.

```
unix$ file hi.c
hi.c: C source, ASCII text
unix$ file a.out
a.out: ELF 64-bit LSB  executable, x86-64, version 1 (SYSV)
dynamically linked (uses shared libs), for GNU/Linux 2.6.24
unix$
```

Como `hi.c` contiene texto típico de fuente en C, `file` cree que contiene tal cosa (y en este caso acierta). Pero, ¿Cómo sabe que `a.out` es un ELF? Simplemente mira al comienzo del fichero y ve si hay cierta constante con cierto valor. Si la hay, se supone que es un ELF puesto que el enlazador que genera ficheros

ELF deja en esa posición ese valor. A estos valores se los llama *números mágicos* (o *magic numbers*). Simplemente sirven como una comprobación de tipos para un hombre pobre. En nuestro caso hemos utilizado Linux esta vez, como puedes ver, y el formato de los ejecutables es ELF, descrito en *elf(5)*.

Podemos utilizar *readelf(1)* para inspeccionar nuestro ejecutable. Con la opción "`-h`" podemos pedirle que vuelque los primeros bytes del fichero suponiendo que es una cabecera de un fichero en formato ELF (un record al comienzo del fichero, nada más).

```
unix$ readelf -h a.out
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x400630
  Start of program headers:          64 (bytes into file)
  Start of section headers:          4520 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         9
  Size of section headers:           64 (bytes)
  Number of section headers:         30
  Section header string table index: 27
unix$
```

Si miramos los bytes al principio del fichero utilizando `xd` (el resto de la línea hace que sólo mostremos dos líneas de la salida de `xd`), esto es lo que vemos:

```
unix$ xd -b -c a.out | sed 2q
0000000  7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
      0  7f  E  L  F 02 01 01 00 00 00 00 00 00 00 00 00
```

Como puedes ver, el fichero comienza por un número mágico que, por convenio, está presente en esa posición para todos los ficheros ELF. Así es cómo sabe UNIX que tiene un ELF entre manos. Si luego resulta que no es un ELF... ¡Mala suerte!

Pero vamos a un sistema OpenBSD y veamos qué sucede...

```
unix$ cc -g hi.c
unix$ file a.out
a.out: ELF 64-bit LSB shared object, x86-64, version 1,
for OpenBSD, dynamically linked (uses shared libs), not stripped
unix$
unix$ readelf -h a.out
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              DYN (Shared object file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0xb40
  Start of program headers:          64 (bytes into file)
  Start of section headers:          5920 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         11
  Size of section headers:           64 (bytes)
  Number of section headers:         35
  Section header string table index: 32
unix$
```

La constante (mágica) es la misma. Pero puedes ver que el resto de datos varía. Por ejemplo, en Linux el programa comenzará a ejecutar en la dirección `0x400630` (que es el punto de entrada al programa). En cambio, en OpenBSD la dirección de comienzo es `0xb40`. El enlazador en cada sistema está programado de acuerdo con los convenios del sistema para el que enlaza, y el kernel sigue dichos convenios para cargar el ejecutable.

El resto del ejecutable son *secciones*. Cada una de ellas es simplemente una serie de bytes descritos por una cabecera (otro record). Tendremos una para el código ejecutable (el texto), otra para los datos inicializados, otra para la información de depuración, y quizá algunas más.

¿Qué sucede si un fichero binario no es del formato adecuado para nuestro sistema? Pues que UNIX no puede ejecutarlo. Por ejemplo, esto sucede si copiamos el ELF de nuestro OpenBSD hacia un OSX e intentamos ejecutarlo:

```
unix$ /tmp/a.out
bash: /tmp/a.out: cannot execute binary file
unix$
```

La constante mágica lo identifica como ELF, y `exec` intentó leerlo e interpretarlo. Pero, tras mirar la cabecera este UNIX descubre que no sabe ejecutarlo y `exec` falla.

9.2. Programas interpretados

¿Y qué sucede si un fichero ejecutable no es un binario? Vamos a ver un ejemplo...

```
unix$ echo echo hola > /tmp/fich
unix$ chmod +x /tmp/fich
unix$ /tmp/fich
hola
unix$
```

Para UNIX, un fichero que tiene permiso de ejecución y no es un binario conocido es un **fichero interpretado**. UNIX denomina **script** a un fichero interpretado. Lo que hace `exec` con este tipo de fichero es ejecutar un programa que hace de *intérprete*. Ya sabes que un intérprete es simplemente un programa que interpreta otro (lo lee y ejecuta las operaciones del programa interpretado).

En el caso de UNIX, el intérprete es `/bin/sh`. Esto explica que en nuestro ejemplo, ejecutar un fichero que contiene comandos es lo mismo que ejecutar un shell y hacer que dicho shell ejecute los comandos que contiene el fichero.

Dado que existen múltiples lenguajes interpretados, es posible indicarle a UNIX qué intérprete queremos para un fichero interpretado. El convenio es que si un fichero es un *script* y comienza por "`#!`", entonces el resto de bytes hasta el primer fin de línea indica la línea de comandos que hay que utilizar para interpretar el fichero.

Por ejemplo, vamos a crear un script con este contenido

```
#!/bin/echo
ya sabemos que echo(1) no lee de stdin
```

en el fichero `ecoeco` y a ejecutarlo:

```
unix$ ecoeco
./ecoeco
unix$ ecoeco -abc hola caracola
./ecoeco -abc hola caracola
unix$
```

Cuando escribimos la línea de comandos "`ecoeco`" en el shell, éste la lee y decide hacer un `fork` y un `exec` de `./ecoeco`. El resto depende del código de `exec` en el kernel de UNIX. El shell ya ha hecho su trabajo llamando a `exec` y no tiene ni idea de si el fichero que se quiere ejecutar es un binario o un script.

Sabemos lo que hace *echo(1)*. Y que no hay magia en nada de lo que ha sucedido. El kernel de UNIX ha leído los primeros bytes de `ecoeco` y ha visto que el intérprete para dicho fichero es `/bin/echo`. Así pues, el kernel se comporta como si la llamada a `exec` fuese del estilo a

```
execl("/bin/echo", "ecoeco", "./ecoeco", NULL);
```

en el primer caso y,

```
execl("/bin/echo", "ecoeco", "./ecoeco", "-abc", "hola", "caracola", NULL);
```

en el segundo caso.

Es importante que veas que "`./ecoeco`" es `argv[1]` cuando `echo` ha ejecutado. Por eso `echo` lo ha escrito.

En resumen, en el caso de un script `exec` ejecuta el intérprete (siendo este `/bin/sh` si no se utiliza "`#!...`" y cambia el vector de argumentos para indicarle al intérprete qué fichero hay que interpretar (el que se indicó en la llamada a `exec`).

Veamos otro ejemplo para ver si esto resulta más claro ahora. Vamos a ejecutar este script

```
#!/bin/echo a b c
```

y ver lo que sucede

```
unix$ eco2 x y z
a b c ./eco2 x y z
unix$
```

Al llamar a

```
execl("./eco2", "eco2", "x", "y", "z", NULL);
```

UNIX se ha comportado como si la llamada hubiera sido

```
execl("/bin/echo", "eco2", "a", "b", "c", "eco2", "x", "y", "z", NULL);
```

Ha dejado `argv[0]` con el nombre del script y ha cambiado el resto de argumentos para incluir al principio los argumentos indicados en la línea "#!...". En cuanto al fichero ejecutable, ha ejecutado el indicado tras "#!".

¿Comprendes por qué en este caso da igual el contenido del fichero tras la línea "#!..."? ¡`echo` no lee ningún fichero!

Otro ejemplo más:

```
unix$ cat /tmp/catme
#!/bin/cat
uno
dos
unix$ /tmp/catme
#!/bin/cat
uno
dos
unix$
```

El comando *hoc(1)* es una calculadora. Quizá no esté instalado en tu UNIX, pero [1] tiene el fuente y explica cómo está programa. Puedes ver que `hoc` evalúa expresiones que lee de la entrada e imprime su valor:

```
unix$ hoc
2 + 2
4
^D
unix$
```

¡Vamos a crear un script!

```
unix$ cat >/tmp/exprs
#!/bin/hoc
2 + 2
3 * 5
^D
unix$
unix$ chmod +x /tmp/exprs
```

¿Comprendes por qué al ejecutarlo sucede esto?

```
unix$ /tmp/exprs
4
15
unix$
```

9.3. Scripts de shell

De ahora en adelante, puedes escribir ficheros que contienen comandos de shell para ejecutar tareas que repites múltiples veces. Por ejemplo, si estás todo el tiempo compilando y ejecutando un programa podrías hacer un script que haga tal cosa en lugar de hacerlo a mano.

Suponiendo que nuestro fichero fuente es `f.c`, podríamos crear este script

```
#!/bin/sh
cc -g f.c
./a.out
```

en el fichero `xc` y en futuro podemos ejecutar

```
unix$ xc
hola mundo
unix$
```

en lugar de compilar y ejecutar a mano el programa cada vez.

No obstante, si tenemos un error de compilación el script ejecuta el fichero `a.out` aunque no corresponda al fuente que hemos intentado compilar (sin éxito).

Podemos aprovecharnos de que el shell es en realidad un lenguaje de programación. Para el shell, los comandos pueden utilizarse como condiciones de "`if`s". Si al comando que utilizamos como condición le ha ido bien (su estatus de salida es `0`) entonces el shell considera que hay que ejecutar el cuerpo del `then`. En otro caso el shell interpreta la condición como falsa.

Este es nuestro script utilizando un "`if`" del shell:

```
#!/bin/sh
if cc f.c
then
    ./a.out
fi
```

Y ahora, si cambiamos `f.c` para que tenga un error sintáctico y no compile...

```
unix$ xc
f.c:10:20: error: expected ';' after expression
1 error generated.
unix$
```

el script no ejecuta `a.out`. Si arreglamos el error

```
unix$ xc
hola mundo
unix$
```

el comando `cc` hará un "`exit(0)`", por lo que el shell recibirá `0` cuando llame a `wait` esperando que `cc` termine. Puesto que `cc` se ha utilizado como condición en un `if`, el shell entiende que hay que considerar que la condición es cierta y ejecutará las líneas de comandos contenidas entre la línea "`then`" y la línea "`fi`".

Recuerda que el shell lee líneas de comandos, no es C:

```
unix$ if echo hola ; then date ; fi
hola
Fri Aug 26 09:59:55 CEST 2016
unix$
```

Luego si escribimos...

unix$ if echo hola then date fi >
]

el shell escribe otro prompt para indicarnos que el comando `if` no está completo y necesitamos escribir más líneas. Concretamente, el shell está leyendo otra línea de comandos y esta debería ser un "`then`". Tras "`if`" todos los argumentos se ejecutarán como un comando que el shell utiliza como condición, luego ha de seguir un "comando" `then` (que es parte de la sintaxis de shell para el *if*, y no un comando en si mismo).

El shell define variables de entorno para permitir que procesemos los argumentos en scripts, y podemos utilizarlas para que nuestro script `xc` compile y ejecute cualquier fichero, así escribimos menos y no tenemos que editar el script cada vez que lo usemos con un programa distinto. Concretamente

- `$*` equivale a los argumentos del script
- `$#` contiene cuántos argumentos hay (es un string, como el valor de cualquier otra variable de entorno)
- `$0` es el nombre del script.
- `$1` es el primer argumento, `$2` el segundo, etc.

Así pues, este script `xc` compila el fichero que se indica como argumento:

```
#!/bin/sh
if cc $1
then
    ./a.out
fi
```

y lo podemos utilizar para compilar y ejecutar cualquier fuente en C

```
unix$ xc f.c
hola mundo
unix$
```

Podemos mejorarlo un poco más si hacemos que el script compruebe que de verdad recibe un argumento.

```
#!/bin/sh
if test $# -eq 0
then
    echo usage: $0 fich
    exit 1
fi
if cc $1
then
    ./a.out
fi
```

Aquí hemos utilizado el comando *test(1)* para comprobar que "`$#`" es igual al "`0`". Este comando es muy útil para evaluar condiciones en los `if` en el shell. Si no hay argumentos, el script utiliza `echo` para escribir un mensaje indicando su uso (y utilizamos "`$0`" como nombre del script).

```
unix$ xc
usage: ./xc fich
unix$
```

En otro caso el script hace su trabajo como antes.

Referencias

1. The UNIX Programming Environment. Brian W. Kernighan,Rob Pike. Prentice-Hall. 1984.