Introducción a Sistemas Operativos: Padres e hijos

Clips xxx
Francisco J Ballesteros

1. 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.

1.1. 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.

1.2. 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$
```

1.3. 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.