Introducción a Sistemas Operativos: Empezando

Clip 18
Francisco J Ballesteros

1. Compilado y enlazado

Podemos ver con más detalle qué sucede cuando compilamos, enlazamos y ejecutamos un programa que hace llamadas al sistema.

Volvamos a considerar el programa para escribir un saludo, que reproducimos aquí por comodidad.

`[hola.c]`:

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

Cuando ejecutamos el comilador y generamos un fichero objeto para este fichero fuente, obtenemos el fichero `hola.o`, como vimos antes.

```
unix$ cc hola.o
unix$ ls -l
total 64
-rw-r--r--  1 elf  wheel    75 May  4 16:02 hola.c
-rw-r--r--  1 elf  wheel  1288 May  4 16:03 hola.o
```

El fichero objeto contiene parte del binario para el programa ejecutable. La parte que corresponde al fuente que hemos compilado. También incluye información respecto a qué símbolos (nombres para código y datos) incluye el fichero objeto y qué símbolos necesita de otros ficheros para terminar de construir un ejecutable.

Dado que es muy tedioso intentar entender el binario, lo que podemos hacer es ver el código ensamblador que ha generado el compilador y ha ensamblado para generar el fichero objeto. Esto se puede hacer utilizando el flag `-S` del compilador de C.

```
unix$ cc -S hola.c
unix$ ls -l
total 64
-rw-r--r--  1 elf  wheel    75 May  4 16:02 hola.c
-rw-r--r--  1 elf  wheel  1288 May  4 16:03 hola.o
-rw-r--r--  1 elf  wheel   840 May  5 09:57 hola.s
```

El fichero `hola.s` contiene el código ensamblador: texto fuente para un programa que traduce dicho código a binario, y que está muy próximo al binario.

[hola.s]:

```
        .file   "hola.c"
        .section        .rodata
        .string "hola\n"
        .text
        .globl  main
main:
        pushq   %rbp
        movq    %rsp, %rbp
        subq    $32, %rsp
        movl    %edi, -20(%rbp)
        movq    %rsi, -32(%rbp)
        leaq    .LC0(%rip), %rdi
        call    puts@PLT
        leave
        ret
```

Como puedes ver, simplemente se define el valor para un string en la memoria (con el saludo) y hay una serie de instrucciones para main. Todo esto se verá más claro en el siguiente tema. Estas instrucciones se pueden ya codificar como números en binario para que las entienda el procesador (eso es lo que tiene el fichero objeto). Las instrucciones se limitan a situar en la pila los argumentos de *puts* y a llamar a *puts*. El valor para nuestra constante debe ir en una sección de valores para datos y el valor del código de *main* debe ir en una sección de código (texto).

Para que el enlazador sepa enlazar este fichero objeto junto con otros y obtener un fichero ejecutable, el fichero objeto incluye no sólo los valores de los datos inicializados y el código, sino también una tabla que contiene el nombre y posiciones en el fichero para cada símbolo. El comando *nm* lista esa tabla y nos permite ver lo que contiene un fichero objeto.

```
unix$ nm hola.o
         U _GLOBAL_OFFSET_TABLE_
00000000 F hola.c
00000000 T main
         U puts
```

Como verás, existe un símbolo de tipo "`F`" (fichero fuente) que nombra el fichero del que procede este código. Compara con la directiva "`.file`" que aparece en el listado en ensamblador. Además, aparece un símbolo de código o texto ("`T`") para la función *main*. Por último, hay una anotación respecto a que este código necesita un símbolo llamado "`puts`" para poder construir un ejecutable. Esto es así puesto que *main* llama a *puts*.

El enlazador tomará este fichero objeto y la librería de C, que contiene más ficheros objeto, y construirá un único fichero ejecutable. Para ello irá asignando direcciones a cada símbolo y copiando los símbolos que se necesiten al fichero ejecutable. Las direcciones asignadas serán aquellas que deberán utilizar en la memoria los bytes de cada símbolo.

Por ejemplo, *main* aparece con dirección "`00000000`" en la salida de *nm* para `hola.o`. Esto es normal puesto que se trata de un trozo de un ejecutable. La dirección que utilice en la memoria el código de *main* dependerá de dónde lo decida situar el enlazador. Una vez enlazado, el binario se cargará en la memoria utilizando el sistema operativo, de tal forma que las direcciones utilizadas en la memoria deberán coincidir con las que ha asignado el enlazador.

Podemos construir un ejecutable utilizando el compilador para que llame al enlazador del siguiente modo:

```
unix$ cc -static hola.o
```

Y ahora tenemos un ejecutable llamado `a.out`:

```
unix$ cc hola.o
unix$ ls -l
total 64
-rwxr-xr-x  1 elf  wheel  342427 May  5 10:16 a.out*
-rw-r--r--  1 elf  wheel      75 May  4 16:02 hola.c
-rw-r--r--  1 elf  wheel    1288 May  4 16:03 hola.o
```

Si listamos los símbolos de `a.out` utilizando *nm* podemos ver una salida como la que sigue.

```
unix$ nm -n a.out
00400270 T __init
00400280 T __start
00400280 T _start
00400480 T atexit
004004f4 T main
00400520 T puts
00400710 T exit
......
0040c930 W write
......
```

Esta vez, hemos utilizado el flag `-n` de *nm* para pedirle que liste los símbolos ordenados por dirección de memoria. Como puedes ver, *main* estará en la dirección `004004f4` cuando esté cargado en la memoria, y *puts* en la dirección `00400520`. Esta llamará a *write*, que estará en la dirección `0040c930`. Dado que es una llamada al sistema, este símbolo está declarado de tipo "`W`" en el UNIX que utilizamos, pero habitualmente podrás verlo como "`T`". (La "`W`" significa *símbolo débil* y es similiar a texto pero permite que un fichero objeto contenga un símbolo del mismo nombre para reemplazar al símbolo débil; Puedes ignorar esto).

En realidad, normalmente no utilizamos la opción "`-static`" al compilar y enlazar. Vamos a enlazar de nuevo:

```
unix$ cc hola.o
unix$ ls -l a.out
-rwxr-xr-x  1 elf  wheel  8570 May  5 10:26 a.out*
```

Esta vez, el ejecutable contiene unos 8Kbytes y no unos 342Kbytes. Veamos que nos dice *nm*:

```
unix$ nm -n a.out
         U puts
         U exit
00000a10 T __init
00000a80 T __start
00000a80 T _start
00000c80 T atexit
00000cf4 T main
00000d20 T __fini
00201000 D __guard_local
00201008 D __data_start
00201008 D __progname
00201010 D __dso_handle
00201148 a _DYNAMIC
00301290 a _GLOBAL_OFFSET_TABLE_
00401360 B _dl_skipnum
00401370 B _dl_searchnum
...
```

La vez anterior, la salida de *nm* tenía muchas mas líneas que ahora (aunque hemos borrado la mayoría para omitir detalles que no importan por el momento). Ahora, la salida de *nm* tiene sólo cuatro o cinco líneas más que las que mostramos.

Lo que sucede es que el programa está enlazado pero no del todo. Puedes ver que *puts* sigue "U" (*undefined*). Cuando este programa esté cargado en la memoria y llegue al punto en que necesita llamar a *puts*, llamará a código que tiene enlazado y que deberá completar el enlazado. Este código forma parte del denominado *enlazador dinámico*.

El proceso es igual que cuando enlazamos de forma estática (en tiempo de compilación si quieres verlo así) el ejecutable. La diferencia es que ahora hemos enlazado los ficheros objeto junto con código que completa el enlazado ya en tiempo de ejecución.

Además, a diferencia de antes, cuando el programa se enlaza ya en tiempo de ejecución, el enlazador dinámico utiliza el sistema operativo para poder compartir librerías ya cargadas en la memoria (enlazadas dinámicamente con otros programas que están ejecutando). Si un programa utiliza por primera vez un símbolo de una librería dinámica que no se ha cargado aún, el sistema operativo la carga en la memoria y el enlazador dinámico puede completar el enlazado.

El comando *ldd* muestra las librerías que necesitará el ejecutable para completar el enlazado, ya en tiempo de ejecución. Así pues:

```
unix$ cc -static hola.o
unix$ ldd a.out
a.out:
not a dynamic executable
unix$ cc hola.o
unix$ ldd a.out
a.out:
        Start            End              Type Open Ref GrpRef Name
        00001af940c00000 00001af941002000 exe  1    0   0      a.out
        00001afbe5731000 00001afbe5c1d000 rlib 0    1   0      /usr/lib/libc.so.78.1
        00001afbfd900000 00001afbfd900000 rtld 0    1   0      /usr/libexec/ld.so
```

Por ahora nos da igual que quiere decir cada columna, pero puedes ver que el ejecutable utilizará código procedente de `/usr/libexec/ld.so` (¡Parte del enlazador dinámico!) y código procedente de la

librería de C, procedente del fichero `/usr/lib/libc.so.78.1`.

Por el momento basta como introducción a lo que hace un sistema operativo y a qué relación tiene con tus programas. Ha llegado el momento de empezar a ver las abstracciones que implementa el kernel y las llamadas al sistema y comandos de shell relacionados con cada una de ellas. Empezaremos en el siguiente tema viendo lo que es un *proceso*, o programa en ejecución, lo que en realidad continúa lo que terminados de describir en este epígrafe.