FUNDAMENTOS DEL SOFTWARE - 2021_rosana@ugr.es

Ejemplo del ciclo de vida de un programa en GNU/Linux

¿Por qué no vemos las tres primeras fases cuando compilamos? Para estudiar el proceso vamos primero a proponer un ejemplo sencillo que es una variante del programa "Hola Mundo",como se muestra a continuación

```
% cat archi.c

#include "archi.h"

main()

{

printf("hola … %d\n", VAR);

}

% cat archi.h

#define VAR 100
```

Nosotros compilamos en general nuestro programa con `gcc` de la forma:

```
% gcc -o hola hola.c
```

En realidad, `gcc` es un *wrapper* (envoltorio o programa que controla el acceso a otros programas) que invoca de forma ordenada a los programas que realizan cada una de las fases antes citadas. Podemos ver lo que hacer la orden con la opción -v:

```
% gcc –v hola.c

Reading specs from /usr/lib/gcc-lib/i386-redhat-linux/2.96/specs

gcc version 2.96 20000731 (Red Hat Linux 7.1 2.96-81)

/usr/lib/gcc-lib/i386-redhat-linux/2.96/cpp0 -lang-c -v -D__GNUC__=2 . . .

#include <...> search starts here:

/usr/local/include

/usr/lib/gcc-lib/i386-redhat-linux/2.96/include

/usr/include

End of search list.

/usr/lib/gcc-lib/i386-redhat-linux/2.96/cc1 /tmp/ccLbIetn.i -quiet -dumpbase hola.c -version -o /tmp/ccQ2g4QC.s . . .

as -V -Qy -o /tmp/ccXeGPDX.o /tmp/ccQ2g4QC.s

GNU assembler version 2.10.91 (i386-redhat-linux) using BFD version 2.10.91.0.2
```

```
/usr/lib/gcc-lib/i386-redhat-linux/2.96/collect2 -m elf_i386 -dynamic-linker /lib/ld-linux.so.2 /usr/lib/gcc-lib/i386-redhat-linux/2.96/../../../crt1.o /usr/lib/gcc-lib/i386-redhat-linux/2.96/../../../crti.o /usr/lib/gcc-lib/i386-redhat-linux/2.96/crtbegin.o -L/usr/lib/gcc-lib/i386-redhat-linux/2.96 -L/usr/lib/gcc-lib/i386-redhat-linux/2.96/../../.. /tmp/ccXeGPDX.o -lgcc -lc -lgcc /usr/lib/gcc-lib/i386-redhat-linux/2.96/crtend.o /usr/lib/gcc-lib/i386-redhat-linux/2.96/../../../crtn.o
```

En negrita, hemos marcado la invocación a los programas que realiza y que son:

- **cpp1**: el preprocesador
- **cc**: el compilador
- **as**: el ensamblador
- **collect2:** que a su vez es otro wrapper para invocar al enlazador, **ld**.

La salida también muestra las numerosas opciones y argumentos con las que se invoca a cada una de estas órdenes. Por qué hacerlo así, en lugar de llamar separadamente a cada programa. Una razón es la conveniencia, la otra portabilidad. La herramienta *gcc* nos suministra un gran número de conmutadores portables que afectan al comportamiento de los programas compilados. Esto nos asegura que nuestros programas compila correctamente en varias plataformas.

A continuación, vamos a compilar nuestro programa paso a paso, para ver con más detalle cada una de las fases. Cuando pasamos un archivo a gcc, este pasa por cuatro etapas. En la primera de ellas, el preprocesador examina los archivos .c en busca de declaraciones que comiencen por #. Cuando son de tipo `#include archivo`, localiza el archivo y lo añade al programa. Después, el preprocesador busca y sustituye todas las macros (`#define`). Una vez realizadas todas las sustituciones se invoca al compilador. Ojo: el compilador no ve el archivo .c original si no la suma del .c y .h con la macros sustituidas. Esto puede producir algunos problemas serios si el archivo de cabecera contiene errores. Una macro que se comportan como función con errores puede generar bastante confusión: el compilador no sabe nada sobre cabeceras, por tanto, cuando ve un error informa del número de línea que él ve; así cuando comprobamos el .c original, podemos ver una línea completamente inocente.

```
% gcc -save-temps archi.c

% ls -l

total 44

-rwxr-xr-x 1 root root 13640 abr 21 20:28 a.out

-rw-r--r-- 1 root root 62 abr 21 20:28 archi.c

-rw-r--r-- 1 root root 17 abr 21 20:27 archi.h

-rw-r--r-- 1 root root 95 abr 21 20:28 archi.i

-rw-r--r-- 1 root root 912 abr 21 20:28 archi.o

-rw-r--r-- 1 root root 376 abr 21 20:28 archi.s
```

El archivo generado por el preprocesador:

```
% cat archi.i

# 1 "archi.c"

# 1 "archi.h" 1

# 2 "archi.c" 2

main()

{

printf("Hola ...%d\n", 100);

}
```

Podemos ver el código en ensamblador (el ensamblador se puede generar solo con `gcc -S`):

```
% cat archi.s
```

```
.file "archi.c"

.version "01.01"

gcc2_compiled.:

.section .rodata

.LC0:

.string "Hola ...%d\n"

.text

.align 4

.globl main

.type main,@function

main:

pushl %ebp

movl %esp, %ebp

subl $8, %esp

subl $8, %esp

pushl $100

pushl $.LC0

call printf

addl $16, %esp

leave

ret

.Lfe1:

.size main,.Lfe1-main

.ident "GCC: (GNU) 2.96 20000731 (Red Hat Linux 7.1 2.96-81)"
```

Pasemos a examinar el enlazador, para ello utilizamos la opción `-c` indica que no se realice el enlazado, es decir, que solo genere el archivo objeto *archi.o*:

```
% gcc –c archi.c
```

El paso final, es enlazar el archivo:

```
% ld archi.o –o archi

ld: warning: cannot find entry symbol _start; defaulting to 08048074

archi.o: In function `main':

archi.o(.text+0x11): undefined reference to `printf'
```

Pero, de ¿dónde viene el error? Hemos utilizado la función `printf` de la biblioteca `libc.so` del directorio */usr/lib*. Por tanto, debemos instruir al enlazador para que localice el código:

```
% ld archi.o –o archi /usr/lib/libc.so
```

ó bien

```
% ld archi.o –o archi –lc

ld: warning: cannot find entry symbol _start; defaulting to 08048184
```

Dejando a un lado el aviso, si intentamos ejecutar nuestro programa, el shell se quejará indicándonos de que no encuentra el archivo (aunque este existe). El problema no es el shell. El shell es una interfaz con el sistema y no sabe como cargarlo ni ejecutarlo. Necesita los servicios de cargador de archivos ELF, *ld-linux.so.2*. Al enlazador le debemos que cargador añadir al ejecutable cuando se enlaza. Por tanto, enlazaremos nuestro programa:

```
% ld archi.o –o archi –lc ld archi.o –o archi –lc-dymamic-linker /lib/ls-linux.so.2

ld: warning: cannot find entry symbol _start; defaulting to 08048184
```

Cuando ejecutamos nuestro programa, *main* no es el primer fragmento de código que se llama. Se ejecuta primero el *código de arranque* que es quien invoca al `main`. Por tanto, debemos indicar a `ld` donde esta el `main`. Realmente, la primera línea ejecutable dentro de un programa esta etiquetada como `_start`, por convenio. Debemos indicar, por consiguiente, a `ld` que utilice `main` como el inicio de nuestro programa.

```
% ld –o archi archi.o –lc –e main –dynamic-linker /lib/ld-linux.so.2

% ./archi

Hola … 100

Segmentation fault (core dumped)
```

¡Bueno, persisten los problemas! Hasta ahora nos hemos ocupado del arranque del programa, consideremos ahora su finalización. Para que nuestro programa termine correctamente, debemos incluir la llamada al sistema *exit()* que termina la ejecución de un programa (cuando se compila directamente, no paso a paso, el enlazador incluye esta función al detectar la última instrucción del programa o el return final).

```
% cat archi.c

#include “archi.h”

main()

{

printf(“hola … %d\n”, VAR);

exit(1);

}

% gcc –c a.c

% ld –o archi archi.o –lc –e main –dymanic-linker /lib/ld-linux.so.2

% ./archi; echo $?

Hola… 100

1
```

Cuando se lanza la ejecución de un programa (llamada al sistema `exec()` en Unix, o `CreateProcess()` en Windows) se crea el mapa de memoria de este en el que se crean las regiones de memoria de acorde a la información contenida en el ejecutable.

El cargador debe:

1. Validar los acceso y memoria – el cargador que es parte del SO lee la cabecera del ejecutable y realiza la validación de tipos, permisos de acceso y requisitos de memoria.
2. Construye el procesos – esto incluye:
    1. Asignar memoria principal para la ejecución del programa.
    2. Copiar el espacio de direcciones desde disco a memoria principal.
    3. Copiar las secciones `.text` y `.data` del ejecuta en memoria.
    4. Copiar los argumentos de invocación del programa en la pila.
    5. Inicializar los registros: ajusta SP para que apunte a la pila, y limpia el resto.

6. Salta a la dirección de inicio, que copia los argumentos salvado en la pila, y salta al `main()`.

Para comprender los detalles que se esconden tras el procedimiento de carga realizado por `execve`, echemos un vistazo la ejecutable ELF:

```
% readelf -l hola

Elf file type is EXEC (Executable file)

Entry point 0x8048360

There are 6 program headers, starting at offset 52

Program Headers:

Type Offset VirtAddr PhysAddr FileSiz MemSiz Flg Align

PHDR 0x000034 0x08048034 0x08048034 0x000c0 0x000c0 R E 0x4

INTERP 0x0000f4 0x080480f4 0x080480f4 0x00013 0x00013 R 0x1

[Requesting program interpreter: /lib/ld-linux.so.2]

LOAD 0x000000 0x08048000 0x08048000 0x004f5 0x004f5 R E 0x1000

LOAD 0x0004f8 0x080494f8 0x080494f8 0x000e8 0x00100 RW 0x1000

DYNAMIC 0x000540 0x08049540 0x08049540 0x000a0 0x000a0 RW 0x4

NOTE 0x000108 0x08048108 0x08048108 0x00020 0x00020 R 0x4

Section to Segment mapping:

Segment Sections...

00

01 .interp

02 .interp .note.ABI-tag .hash .dynsym .dynstr .gnu.version .gnu.version_r .rel.got .rel.plt .init .plt .text .fini
.rodata

03 .data .eh_frame .ctors .dtors .got .dynamic .bss

04 .dynamic

05 .note.ABI-tag
```

La salida muestra la estructura general del proceso *hola*. La primera cabecera de programa corresponde al segmento de código del proceso, que se cargará del archivo desde el desplazamiento `0x000000` en una región de memoria que será proyectada en el espacio de direcciones del proceso en la dirección `0x08048000`. El segmento de código tendrá `0x004f5` bytes y debe estar alineado a página (`0x1000`). Este segmento comprenderá los segmentos ELF `.text` y `.rodata`, discutidos anteriormente, más segmentos adicionales generados durante el procedimiento de enlazado. Como esperábamos, esta marcado como sólo lectura(`R`) y ejecutable (`E`), pero no se puede escribir (`W`).

La segunda cabecera de programa corresponde al segmento de datos. La carga de este segmento sigue los mismos pasos mencionados anteriormente. Sin embargo, observar que la longitud del segmento es de tamaño `0x000e8` sobre el archivo y `0x00100` en memoria. Esto se debe a la sección `.bss`, que se rellena a ceros y por tanto no debe estar presente en el archivo. Este segmento también debe estar alineado a página y contiene los segmentos ELF `.data` y `.bss`. Esta marcado como lectura/escritua (`RW`). La tercera cabecera de programa resulta del procedimiento de enlazado y es irrelevante a nuestra discusión.

Si disponemos de un sistema de archivos */proc* podemos comprobar esto mientras se ejecuta el proceso "Hola Mundo":

```
% cat /proc/`ps -C hola - pid h|tr " " "\null"/maps

08048000-08049000 r-xp 00000000 08:05 277304 /root/hola

08049000-0804a000 rw-p 00000000 08:05 277304 /root/hola

40000000-40016000 r-xp 00000000 08:05 271030 /lib/ld-2.2.2.so
```

```
40016000-40017000 rw-p 00015000 08:05 271030 /lib/ld-2.2.2.so

40017000-40018000 rw-p 00000000 00:00 0

40018000-40019000 rw-p 00000000 00:00 0

40026000-4014c000 r-xp 00000000 08:05 350657 /lib/i686/libc-2.2.2.so

4014c000-40152000 rw-p 00125000 08:05 350657 /lib/i686/libc-2.2.2.so

40152000-40156000 rw-p 00000000 00:00 0

bfffe000-c0000000 rwxp fffff000 00:00 0
```

La primera región proyectada es el segmento de código del proceso, el segundo constituye el segmento de datos (datos + bss + heap), y el último, que no tiene correspondencia en el archivo ELF, es la pila. Se puede obtener información adicional mediante `time`, `ps`, y `/proc/pid/stat` de GNU.

◄ Depuración en Entornos Integrados de Desarrollo

Ir a...

Ir a...
POSGRADO 20-21
GRADO 21-22
POSGRADO 21-22
Ayuda
Preguntas Frecuentes - General
Preguntas Frecuentes - Profesorado
Preguntas Frecuentes - Alumnado
Consultas e incidencias
Español - Internacional (es)
English (en)
Español - Internacional (es)