Universidad Tecnológica Nacional
Facultad Regional Buenos Aires

Departamento de Electrónica

Asignatura: Informática I

Hola Mundo en profundidad.

Autor: Ing. Alejandro Furfaro. Profesor Titular

Índice de contenido

Introducción

Cualquier curso de programación comienza con el paradigmático ejemplo "Hola Mundo". Es como una tradición. Comparado con los programas de aplicación habituales, muchos de los cuales emplean interfaz gráfica de modo de darle un "look and feel" muy intuitivo, amigable, y en ocasiones muy agradable a la vista, el pobre "Hola mundo", parece muy poco interesante. Sin embargo es muy útil para comprender los conceptos generales de programación y entorno que existen en su trasfondo, y que son los mismos que cualquiera de los demás programas.

Programa fuente

```
#include <stdio.h>
int main(void)
{
     printf("Hello World!\n");
     return 0;
}
```

La primer línea del programa es una directiva para el compilador que le indica a éste la inclusión de un archivo denominado *stdio.h*. Este archivo, contiene definiciones de funciones, macros y variables. En particular, necesitamos incluirlo ya que entre las funciones definidas en el mismo se encuentra *printf*, que estamos invocando en nuestro programa y cuyo código objeto se encuentra en la librería estándar de C denominada *libc*.

Todos los programas escritos en C deben incluir una función *main*, la que representa el punto de entrada ("entry point"), es decir, la primer instrucción que ejecutará el computador al cargar el programa en la memoria. En el lenguaje C se ha tomado como convención darle obligatoriamente este nombre a la función que se ejecutará al inicio del programa. En nuestro sencillo ejemplo la función *main* no toma ningún parámetro ya que este programa no se ejecutará con argumentos desde la línea de comandos. Por este motivo se coloca *void* (vacío) como argumento, al solo efecto de ser mas específicos ya que de no haber colocado nada entre los paréntesis no habría diferencia a efectos de la compilación. Esta función devuelve un entero. En el caso de la función principal del programa este entero se le devuelve al proceso padre que es quien ha invocado su ejecución, en este caso el *shell* del sistema operativo. Por convención se retorna un número de 8 bits, que será 0 en caso de terminación normal del proceso, un valor n / 0<n<128, para procesos que hayan terminado con alguna anomalía, y n >128 para procesos terminados por medio de alguna *señal*[1].

1 ¿Recuerdan la orden kill en la Guía 000, cuando vimos los comandos del shell? Kill envía una señal a través del

Las líneas encerradas entre las llaves componen el programa en sí, que no hace otra cosa que imprimir *Hola Mundo!*, en la consola desde la que se invocó el proceso.

El compilador

Para compilar cualquier programa en Linux utilizamos el universalmente difundido GNU C Compiler (*gcc*) y sus herramientas asociadas que generalmente vienen en el paquete *binutils*.

Para compilar nuestro pequeño programa tipeamos desde la consola el siguiente comando:

```
$ gcc –c –o hola.o hola.c
```

La opción *-c* le indica al *gcc* que compile solamente, es decir que genere un archivo objeto. Si no la incluimos el *gcc* seguirá automáticamente a la siguiente etapa, invocando al linker, para producir directamente el programa ejecutable.

En ésta etapa del proceso de aprendizaje no es conveniente incurrir en el confort del automatismo, y en cambio trabajar las diferentes etapas para poder seguir de cerca cada pequeño detalle que será un gran aporte a nuestro conocimiento.

La opción *-o* sirve para indicarle a *gcc* el nombre del archivo de salida (*o* de **o**utput). En este caso lo llamamos igual que el de entrada pero terminándolo en .o en lugar de .c, para darle la nomenclatura estándar de los archivos objeto.

El comando *file* de Linux nos puede dar alguna idea de las características de este archivo objeto:

```
$ file hola.o
hola.o: ELF 32-bit LSB relocatable, Intel 80386, version 1
(SYSV), not stripped
```

El archivo objeto generado por *gcc*, es del tipo reubicable. Un archivo objeto es reubicable cuando contiene referencias simbólicas de variables y/o funciones a direcciones relativas contenidas dentro de la Unidad de Compilación o fuera de ella. Al poder invocar variables o funciones externas a la Unidad de Compilación permite utilizar otros archivos objetos obtenidos a partir de otras unidades de compilación para proveer al linker de las interfaces a las rutinas para que pueda construir el programa ejecutable.

Una referencia simbólica en nuestro caso consiste en almacenar la dirección numérica correspondiente a una etiqueta que identifica a una variable o a una función. Por ejemplo: que dirección ocupará dentro del archivo la etiqueta

Kernel al proceso para que finalice su ejecución.

main, o que habrá que colocar como dirección destino en la instrucción que efectúa la llamada a *printf*. La Unidad de Compilación no es otra cosa que lo que estamos compilando, es decir, nuestro programa. Entonces como nuestro programa posee referencias simbólicas para acceder a *printf* y lograr imprimir *Hola Mundo!* en la pantalla, es entonces reubicable.

El archivo objeto contiene en su encabezado información para la reubicación. El Linker reemplazará la información simbólica con la información de dirección actual en el momento de construir el archivo binario ejecutable. En nuestro ejemplo, la llamada al código de la función *printf*, será resuelto por el Linker ya que el *gcc* no conoce la dirección de comienzo de *printf*, ya que este código no está definido en nuestro pequeño programa. El Linker la obtendrá del encabezado de la librería estándar de C, *libc* (contenida en el archivo /lib/libc-2.7.so, para el kernel 2.6.26-1-686).

La otra característica saliente de nuestro archivo objeto es el formato ELF[2] de 32 bits que contiene una tabla de símbolos que no ha sido removida. Se puede remover la tabla de símbolos con el programa *strip*.

```
$ strip hola.o
$ file hola.o
hola.o: ELF 32-bit LSB relocatable, Intel 80386, version 1
(SYSV), stripped
```

Como vemos el resultado del comando es la remoción (*strip*) de la tabla de símbolos.

No es este el único síntoma visible. Queda a cargo del lector mirar con *ls -las* el tamaño del archivo *hola.o*, antes de la orden *strip*, y después de la misma, para verificar que se reduce el tamaño de este archivo en unos 300 bytes. Esto es lógico ya que estamos quitando información de su encabezado.

El formato ELF puede variar en función del tipo de procesador que se trate: la salida de nuestros ejemplos corresponde a una PC de escritorio de arquitectura IA-32, y un S.O. Debian Lenny de *32* bits. Por lo tanto es ELF *32.*

Con el comando *objdump* podemos comprobar varias cosas adicionales del archivo *ELF*. En el siguiente vuelco de pantalla hemos utilizado las opciones *-h*, para que presente los headers de las diferentes secciones del archivo objeto *hola.o*, *-r* para que imprima las entradas de reubicación del archivo objeto *hola.o*, y *t* para que imprima la tabla de información simbólica de dicho archivo.

```
$ objdump -hrt hola.o

hola.o:     file format elf32-i386

Sections:
```

2 Tools Interface Standard (TIS) Comitee, Executable and Linkable Format (ELF), Portable Formats Specification, Version 1.1

```
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000025  00000000  00000000  00000034  2**2
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1 .data         00000000  00000000  00000000  0000005c  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000000  00000000  00000000  0000005c  2**2
                  ALLOC
  3 .rodata.str1.1 0000000c  00000000  00000000  0000005c  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .comment      0000001f  00000000  00000000  00000068  2**0
                  CONTENTS, READONLY
  5 .note.GNU-stack 00000000  00000000  00000000  00000087  2**0
                  CONTENTS, READONLY
SYMBOL TABLE:
00000000 l    df *ABS*  00000000 hola.c
00000000 l    d  .text  00000000 .text
00000000 l    d  .data  00000000 .data
00000000 l    d  .bss   00000000 .bss
00000000 l    d  .rodata.str1.1 00000000 .rodata.str1.1
00000000 l    d  .note.GNU-stack        00000000 .note.GNU-stack
00000000 l    d  .comment       00000000 .comment
00000000 g     F .text  00000025 main
00000000         *UND*  00000000 puts


RELOCATION RECORDS FOR [.text]:
OFFSET   TYPE              VALUE
00000012 R_386_32          .rodata.str1.1
00000017 R_386_PC32        puts
```

Vemos que *hola.o* tiene 5 secciones. Antes de seguir, veamos que son las secciones en un archivo *ELF*. Básicamente la organización de un archivo objeto en formato *ELF* contiene un encabezado en donde está el mapa de organización del archivo, y luego secciones en donde se agrupan, instrucciones que componen el programa, las variables que hemos definido en el programa, y eventualmente listas de símbolos y demás elementos que hacen a los aspectos del lenguaje en el que se escribió el programa original.

Cuando pasemos la fase del linker, el archivo ejecutable también tendrá el formato *ELF*, aunque con una estructura algo diferente. En particular las secciones en un objeto, se denominan segmentos en el ejecutable derivado del mismo.

Aclarado este concepto, vamos a estudiar el contenido de cada sección:

.text, que contiene el código máquina compilado que imprime "Hola Mundo!" en la consola. El programa cargador de Linux (*loader*) toma esta sección y la copia en el segmento de código del proceso (que es el área de memoria en la que se carga la secuencia de instrucciones que componen el programa).

.data,. Que en este caso está vacío ya que nuestro sencillo ejemplo no hemos definido entidades en nuestro programa que se guarden en esta sección: variables globales y variables locales estáticas inicializadas. De otro modo aquí encontraríamos los valores iniciales de aquellas variables globales o locales estáticas que hemos definido en nuestro programa, con sus correspondientes valores iniciales definidos al momento de su declaración, las que serán copiadas con esos valores en el segmento de datos del proceso.

.bss, que también en este caso está vacía ya que nuestro sencillo ejemplo no tiene ninguna variable no inicializada (de hecho no tiene variables). De otro modo aquí encontraríamos aquellas variables locales o globales no inicializadas en nuestro programa al momento de su declaración. En tal caso indicaría cuantos bytes se deben allocar[3] y completar con ceros, en el segmento de datos del proceso, además de los ya allocados para la sección ***.data***.

.rodata, que contiene la string "HolaMundo!\n", etiquetada como Read Only. En general las áreas de datos de los programa no son Read Only ya que los datos en general es de esperar que puedan modificarse durante la operación del programa. Sin embargo hay datos como las constantes de programa que definimos como macros con la directiva #define, o strings de mensajes que no es de esperar que se modifiquen que el compilador pone en una sección aparte para diferenciarlas de los datos comunes. Luego dependiendo del Sistema Operativo, pueden copiarse en el segmento de datos del proceso o en el segmento de código.

.comment, contiene 0x1f bytes de comentarios cuya relación no podemos determinar ahora ya que sencillamente no hemos escrito comentario alguno en nuestro código (tan sencillo es.....). Mas adelante determinaremos de donde proviene este valor.

A esta altura es oportuno proponer una primer experiencia para el lector: Compilar nuevamente el archivo *hola.c*, pero agregándole la opción *-g* para que el *gcc* incluya en el archivo objeto *hola.o* la información simbólica que, en caso de requerirse *debug* del programa, permita al *debuger* recorrer el texto del archivo fuente, en lugar de pasar por el código *assembler* que de otro modo sería el único que podría determinar. Una vez recompilado con *-g*, compara el tamaño resultante del nuevo archivo *hola.o*, y por sobre todas las cosas re ejecutar el comando *objdump* con idénticas opciones y comparar con el listado en este documento. ¿Hay cambios?

Volviendo a nuestra salida de *objdump*, podemos observar que aparece una tabla de símbolos, en la que el símbolo *main* ocupa el offset 0 y *puts* figura *UND* (por Undefined). Esta tabla indica como se efectuarán la reubicaciones en la sección *.text* de las referencias efectuadas a secciones externas. El primer símbolo reubicable corresponde a la string "Hola Mundo!\n" contenida en la sección *.rodata,* y el segundo símbolo reubicable, *puts*, corresponde a

3 O alojar, reservar espacio para. Sucede que esta como otras ciencias, la programación ha sido desarrollada en países de habla anglosajona, y en ocasiones los profesionales para simplificar la interpretación de los textos originales en Inglés, muchas veces castellanizamos la palabra sin cambios. De este modo todos saben de que se habla.

una función de la librería *libc*, que se ha generado como resultado de llamar a *printf*.

Para entender mejor el contenido de *hola.o*, conviene analizar su código *assembler*. Para ello empleamos en *gcc* la opción *-S* para que genere el código *assembler* resultante, y *-o* – para que en lugar de guardarlo en *hola.s*, lo presente en la consola.

```
$ gcc -S hola.c -o -
        .file   "hola.c"
        .section        .rodata
.LC0:
        .string "Hola mundo!"
        .text
.globl main
        .type   main, @function
main:
        leal    4(%esp), %ecx
        andl    $-16, %esp
        pushl   -4(%ecx)
        pushl   %ebp
        movl    %esp, %ebp
        pushl   %ecx
        subl    $4, %esp
        movl    $.LC0, (%esp)
        call    puts
        movl    $0, %eax
        addl    $4, %esp
        popl    %ecx
        popl    %ebp
        leal    -4(%ecx), %esp
        ret
        .size   main, .-main
        .ident  "GCC: (Debian 4.3.2-1.1) 4.3.2"
        .section        .note.GNU-stack,"",@progbits
```

Podemos ver algunas cuestiones interesantes. Una es que en lugar de invocar a *printf* el código *assembler* llama a la función *puts* de la *libc*, como vemos en la línea que figura resaltada, utilizando la instrucción *call*, y pasando previamente justo en la instrucción anterior la etiqueta *.LCO*, que es el inicio de la string "Hola Mundo!\n" que presentará por la consola, y que también hemos resaltado en los dos puntos del programa en los que aparece.

Además podemos ver de donde proviene la sección *.comment* en el *objdump* anterior: La ante última línea, es introducida automáticamente por el compilador.

Volviendo a *puts*, el compilador debe poner en el campo de la instrucción *call* que corresponde a la dirección, el valor relativo al comienzo del programa que ocupará el código de *puts* cuando se arme todo el bloque de código. Cabe

preguntarse como sabe cual es este valor. La respuesta es muy simple (y algo dramática): no lo sabe.

Tal vez el comando *nm* nos pueda dar alguna pista.

```
$ nm hola.o
00000000 T main
         U puts
```

El comando *nm* lista los símbolos declarados en el encabezado de un archivo objeto, asumiendo que éste está en formato *ELF*.

Básicamente la salida tiene tres columnas:

En la primera de ellas se tiene el valor que colocará el compilador en reemplazo de la etiqueta en el momento de crear el archivo objeto y que corresponde a la distancia en bytes que habrá desde el lugar que ocupa esta etiqueta respecto del primer byte del segmento en el que se cargará nuestro programa.

En la segunda columna se tiene el tipo de etiqueta (los diferentes significados de cada letra se obtienen con *man nm*), que determina si es mayúscula que la etiqueta corresponde a un símbolo Global / Estático, o minúscula si es una etiqueta local.

En el caso de *main*, la *T* indica que está contenida en la sección *Text*, es decir, código.

La tercer columna es la etiqueta definida en el programa.

Lo mas importante: la etiqueta *puts*, figura con una U, que significa *Undefined*.

Estas son las cosas que no puede resolver el compilador ya que trabaja sobre el archivo fuente especificado.

El Linker

El linker entonces es un programa que toma uno o más archivos objetos y los combina (linkea) en un archivo ejecutable. Los archivos objetos pueden provenir de diferentes archivos fuente compilados en nuestro proyecto, y también de librerías de objetos externas. Tal el caso de *puts*.

Llegó el momento de analizar como se genera un programa ejecutable a partir del objeto *hola.o*.

Leyendo por encima las *man pages* de *ld*, y asumiendo que no sería muy diferente de lo que requiere *gcc*, nuestro primer intento es:

```
$ ld -o hola hola.o -lc
ld: warning: cannot find entry symbol _start; defaulting to
00000000080481a4
ls -las
```

```
total 20
4 drwxr-xr-x 2 alejandro alejandro 4096 dic 28 01:08 .
4 drwxr-xr-x 5 alejandro alejandro 4096 dic 26 18:33 ..
4 -rwxr-xr-x 1 alejandro alejandro 1932 dic 28 01:08 hola
4 -rw-r--r-- 1 alejandro alejandro   74 dic 26 19:22 hola.c
4 -rw-r--r-- 1 alejandro alejandro  860 dic 27 17:56 hola.o
$ ./hola
bash: ./hola: No existe el fichero o el directorio
```

No hemos generado un archivo que pueda ser reconocido como ejecutable por el sistema operativo. Por eso el mensaje de error.
Regla Nº 1: Nunca lea por encima las *man pages*.
Regla Nº2: los primeros intentos rara vez nos llevan a buen puerto (en especial si violamos la Regla Nº1).
En algunas plataformas el uso de un linker es bastante trivial. Pero en los sistemas operativos "Unix like" (tal el caso de Linux), la estabilidad se suele pagar con algunas complejidades. Esto hace que el pasaje del rótulo *main* que está al inicio del programa a un punto de entrada binario para ser tomado por el sistema, no resulte una tarea trivial. Esta es probablemente una de las razones por las cuales el compilador *gcc* invoca por default en forma automática al linker. Para evitarnos algunas complejidades.
Esto es muy saludable cuando estamos en un ambiente profesional en donde ya hemos pasado por la etapa formativa, y ahora formamos parte de un equipo de profesionales especializados en desarrollo, contexto en el cual la productividad es lo importante, y por ende echamos mano de cualquier simplificación posible, seguros de tener los conocimientos necesarios que nos permitan tomar el control cuando las simplificaciones no funcionen (lo cual ocurre mas a menudo de lo que el lector pueda imaginar).
Justamente es en la etapa formativa en donde las complejidades no deben esquivarse ya que junto con ellas se esquiva el conocimiento. Y esto nos dejaría muy mal preparados para la etapa profesional.
Entonces, hay que entender algunas cosas, antes de continuar.
El *gcc* arma un llamado a un módulo de *GNU utils* llamado *collect2*, que hace las veces de linker. Para saber como lo arma, podemos observarlo incluyendo en la compilación la opción *-v*. Esta opción hace que el *gcc* imprima por *stderr* los comandos que arma para ejecutar las diferentes etapas de compilación, además de información sobre versiones y demás.
Probemos entonces:

```
$ gcc -o hola hola.o -v
Using built-in specs.
Target: i486-linux-gnu
Configured with: ../src/configure -v --with-pkgversion='Debian
4.3.2-1.1'                                                  --with-
bugurl=file:///usr/share/doc/gcc-4.3/README.Bugs         --enable-
languages=c,c++,fortran,objc,obj-c++    --prefix=/usr    --enable-
```

```
shared --with-system-zlib --libexecdir=/usr/lib --without-
included-gettext --enable-threads=posix --enable-nls --with-gxx-
include-dir=/usr/include/c++/4.3 --program-suffix=-4.3 --enable-
clocale=gnu --enable-libstdcxx-debug --enable-objc-gc --enable-
mpfr --enable-targets=all --enable-cld --enable-checking=release
--build=i486-linux-gnu --host=i486-linux-gnu --target=i486-linux-
gnu
Thread model: posix
gcc version 4.3.2 (Debian 4.3.2-1.1)
COMPILER_PATH=/usr/lib/gcc/i486-linux-
gnu/4.3.2/:/usr/lib/gcc/i486-linux-gnu/4.3.2/:/usr/lib/gcc/i486-
linux-gnu/:/usr/lib/gcc/i486-linux-gnu/4.3.2/:/usr/lib/gcc/i486-
linux-gnu/:/usr/lib/gcc/i486-linux-gnu/4.3.2/:/usr/lib/gcc/i486-
linux-gnu/
LIBRARY_PATH=/usr/lib/gcc/i486-linux-
gnu/4.3.2/:/usr/lib/gcc/i486-linux-gnu/4.3.2/:/usr/lib/gcc/i486-
linux-
gnu/4.3.2/../../../../lib/:/lib/../lib/:/usr/lib/../lib/:/usr/lib
/gcc/i486-linux-gnu/4.3.2/../../../:/lib/:/usr/lib/
COLLECT_GCC_OPTIONS='-o' 'hola' '-v' '-mtune=generic'
  /usr/lib/gcc/i486-linux-gnu/4.3.2/collect2 --eh-frame-hdr -m
elf_i386 --hash-style=both -dynamic-linker /lib/ld-linux.so.2 -o
hola     /usr/lib/gcc/i486-linux-gnu/4.3.2/../../../../lib/crt1.o
/usr/lib/gcc/i486-linux-gnu/4.3.2/../../../../lib/crti.o
/usr/lib/gcc/i486-linux-gnu/4.3.2/crtbegin.o -L/usr/lib/gcc/i486-
linux-gnu/4.3.2               -L/usr/lib/gcc/i486-linux-gnu/4.3.2
-L/usr/lib/gcc/i486-linux-gnu/4.3.2/../../../../lib -L/lib/../lib
-L/usr/lib/../lib    -L/usr/lib/gcc/i486-linux-gnu/4.3.2/../../..
hola.o -lgcc --as-needed -lgcc_s --no-as-needed -lc -lgcc --as-
needed    -lgcc_s    --no-as-needed    /usr/lib/gcc/i486-linux-
gnu/4.3.2/crtend.o /usr/lib/gcc/i486-linux-gnu/4.3.2/../../../../
lib/crtn.o
```

Bastante complejo por cierto. Nuestra parte de interés se reduce a *COLLECT_GCC_OPTIONS*. Es decir a la última parte.
Si lo depuramos eliminando rutas redundantes y simplificando los *paths*, nos queda el siguiente comando

```
$/usr/lib/gcc/i486-linux-gnu/4.3.2/collect2 --eh-frame-hdr -m
elf_i386 --hash-style=both -dynamic-linker /lib/ld-linux.so.2 -o
hola /usr/lib/crt1.o /usr/lib/crti.o /usr/lib/gcc/i486-linux-gnu/
4.3.2/crtbegin.o -L/usr/lib/gcc/i486-linux-gnu/4.3.2
-L/usr/lib/gcc/i486-linux-gnu/4.3.2 -L/lib -L/usr/lib hola.o
-lgcc --as-needed -lgcc_s --no-as-needed -lc -lgcc --as-needed
-lgcc_s --no-as-needed /usr/lib/gcc/i486-linux-gnu/4.3.2/crtend.o
```

```
/usr/lib/crtn.o
```

Que funciona perfectamente generando el programa ejecutable *hola*, y que ahora si puede ejecutarse sin errores.
Lo mismo ocurre si en lugar de collect2 utilizamos *ld* (queda a cargo del lector reemplazar *ld* por *collect2* para comprobarlo), que es el linker "oficial" por llamarlo de algún modo a la luz de la aparición del misterioso *collect2*.
Es interesante discutir algunas de las opciones empleadas por *gcc* para invocar a *collect2*. En principio el uso de la opción *-dynamic-linker*, denota que se trabajará con librerías de código dinámicas. Esta opción trabaja con un argumento que es el archivo que trabajará como linker dinámico en tiempo de ejecución. En nuestro caso el argumento de la línea de comandos es *-dynamic-linker /lib/ld-linux.so.2.* Este módulo es el linker dinámico por default de linux, que se encarga de cargar el programa ejecutado desde el prompt y resolver en tiempo de carga las dependencias con código de librerías dinámicas necesario.
Por si no queda claro que significa una librería dinámica, veamos las opciones con que contamos:
Linkeo estático: El linker coloca todo el código máquina adentro del ejecutable. Cuando linkeamos de esta forma todas las referencias se resuelven en tiempo de linkeo. Resultado: un archivo ejecutable con todo lo necesario (sin ninguna dependencia a resolver en el momento de su ejecución, pero muy voluminoso.
Linkeo Dinámico: En este modo algunas bibliotecas pueden ser compartidas. Es decir que el código de las mismas no va a aparecer dentro de nuestro ejecutable. Pero entonces vamos a necesitar que esas bibliotecas formen parte del sistema. En este modo hay algunas referencias que se resuelven al momento de cargar el programa para su ejecución.
Este último es el caso en que se tiende a trabajar debido a que se genera código mucho mas compacto. Este es el caso de nuestro ejemplo.
Ampliaremos el tema librerías en la Guía002.
Aparecen en la línea de argumentos varios archivos objeto que corresponden a otras Unidades de Compilación que se necesitan incluir para que el linker pueda convertir a nuestro sencillo Hola Mundo en un ejecutable ELF, y determinar su punto de entrada. Estos son:

/usr/lib/crt1.o

/usr/lib/crti.o

/usr/lib/gcc/i486-linux-gnu/4.3.2/crtbegin.o

/usr/lib/gcc/i486-linux-gnu/4.3.2/crtend.o

/usr/lib/crtn.o

Estos objetos son imprescindibles para conectar nuestra etiqueta *main* con el punto de entrada "físico" por llamarlo de algún modo distintivo de nuestro programa. Son Unidades de Procesamiento que vienen con el sistema operativo y se utilizan junto con los objetos del paquete *binutils*, para generar las aplicaciones.

Si se mira sus contenidos mediante el comando *objdump*, siempre con las opciones *-hrt*, es posible ver que *crt1.0*, y *crti.o* tienen relación directa con *main*. En particular *crt1.o* tiene a *main* en su tabla de símbolos como "U" (undefined), y *crti.o*, tiene la etiqueta *_start*.
Para nas detalle en el Apéndice A se listan las salidas del mencionado comando para cada uno de los archivos.
Puede verse que los otros tres complementarios. Pero los dos mencionados tienen relación directa con el punto de entrada del programa (el cual se suele encontrar como *entry point*, su nombre original, en inglés).
Ahora tenemos un ejecutable que funciona, y hemos logrado desentrañar lo complejo del proceso de compilación y linkeo.
Queda para el lector ejecutar el comando *objdump -hrt hola*, para ver la diferencia entre la salida del objeto *hola.o* y del ejecutable hola. Tal vez esto de una dimensión de la cantidad de información irrelevante para nuestro programa pero imprescindible para que este pueda ser ejecutado en Linux que ha agregado el linker.
Idéntica conclusión se obtiene comparando sus tamaños con el comando *ls -las hola**

```
$ ls -las hola*
total 24
4 drwxr-xr-x 2 alejandro alejandro 4096 dic 30 16:53 .
4 drwxr-xr-x 5 alejandro alejandro 4096 dic 26 18:33 ..
8 -rwxr-xr-x 1 alejandro alejandro 6272 dic 30 13:45 hola
4 -rw-r--r-- 1 alejandro alejandro   74 dic 26 19:22 hola.c
4 -rw-r--r-- 1 alejandro alejandro  856 dic 30 16:53 hola.o
```

Veamos que ocurre si ejecutamos *nm* sobre el programa ya linkeado. La salida es mucho mas amplia que la que se obtiene cuando se lo ejecuta sobre el objeto.

```
$ nm hola
080494b4 d _DYNAMIC
08049588 d _GLOBAL_OFFSET_TABLE_
0804848c R _IO_stdin_used
         w _Jv_RegisterClasses
080494a4 d __CTOR_END__
080494a0 d __CTOR_LIST__
080494ac D __DTOR_END__
080494a8 d __DTOR_LIST__
0804849c r __FRAME_END__
080494b0 d __JCR_END__
080494b0 d __JCR_LIST__
080495a8 A __bss_start
080495a0 D __data_start
08048440 t __do_global_ctors_aux
08048320 t __do_global_dtors_aux
080495a4 D __dso_handle
```

```
         w __gmon_start__
0804843a T __i686.get_pc_thunk.bx
080494a0 d __init_array_end
080494a0 d __init_array_start
080483d0 T __libc_csu_fini
080483e0 T __libc_csu_init
         U __libc_start_main@@GLIBC_2.0
080495a8 A _edata
080495b0 A _end
0804846c T _fini
08048488 R _fp_hw
08048274 T _init
080482f0 T _start
080495a8 b completed.5706
080495a0 W data_start
080495ac b dtor_idx.5708
08048380 t frame_dummy
080483a4 T main
         U puts@@GLIBC_2.0
```

Vemos que se han agregado una cantidad importante de referencias y etiquetas. Ahora las etiquetas marcadas con U tienen la información acerca de la Unidad de Compilación desde la cual se han obtenido y completado las referencias. En el caso de la referencia a *puts* que ya había aparecido al correr *nm* sobre el objeto, figura *ptus@@GLIBC_2.0*, que identifica quien contiene la definición de esa etiqueta, en este caso Gnu LIBC, lo que significa que la etiqueta es externa.

Vale también la pena analizar la etiqueta remarcada en el listado que aparece en el ejecutable. Esta etiqueta referencia a la etiqueta *_start* que en general es el punto de entrada que el linker define para nuestro programa y que como vemos se ha definido a partir de Unidades de Compilación externas a nuestro programa *hola*.

Carga y ejecución

En los sistemas "UNIX like", o mas específicamente en aquellos que adhieren al estándar POSIX, existe claramente definida una jerarquía de procesos basada en lo que se conoce como parentesco. El árbol de procesos comienza con *init*, cuyo *PID* (*Process ID* = número de 16 bits mediante el cual se identifica unívocamente a un proceso) es 1.
Cada vez que se enciende una terminal o se abre una ventana de sesión *init* crea una instancia de */bin/login* que será encargado de validar el *user ID* y la contraseña que el usuario que abrió una terminal ingresará en la misma para su ingreso al sistema. Si los datos ingresados por el usuario son válidos, login ejecuta las *scripts* de inicio personalizadas para el usuario y crea una instancia

del programa *shell* definido para ese usuario (por lo general *bash* (Bourne Again SHell). *bash*, es entonces el responsable de presentar al usuario el *prompt* ('$' para usuarios comunes, '#' para *root*).
Esto puede verificarse mediante el comando *ps* ejecutado en una consola de modo texto:

```
$ ps -f
UID        PID  PPID  C STIME TTY          TIME CMD
500       6602  3459  0 10:56 tty1     00:00:00 bash
500       6686  6602  0 10:58 tty1     00:00:00 ps -f
```

Podemos observar que *bash* (PID=6602) es un proceso hijo del que tiene *PID*=3459. A su vez el comando *ps -f* ejecutado en esa misma consola de texto resulta ser hijo de *bash* ya que en sus datos figura el *PID* de *bash* en la columna *PPID* (Parent PID). La pregunta es ¿quien es el proceso 3459?. Con la opción e del comando *ps* para listar todos los procesos del sistema (no los de la consola que por default son los que se listan), mas los buenos oficios del filtro *grep*, tenemos:

```
$ ps -ef | grep 3459
UID        PID  PPID  C STIME TTY          TIME CMD
500       3459  1     1 09:37 tty1     00:00:00 /bin/login
500       6602  3459  0 10:56 tty1     00:00:00 bash
500       6686  6602  0 10:58 tty1     00:00:00 grep 3459
```

aquí aparece *login*, cuyo padre es *init* (PPID=1).
De modo que al tipear en el prompt del sistema *./hola*, estamos haciendo que *bash* cree un proceso hijo (que en principio es una réplica de sí mismo), acción que se lleva adelante invocando a la función *fork ()*, al que luego reemplaza por la imagen de *hola*, acción que se logra mediante la familia de *system calls exec()*. Un viaje... pero no es el momento de emprenderlo en este punto.
Podemos comprobarlo mediante algunos comandos del sistema operativo. El comando *strace* permite trazar las llamadas a *system calls* del proceso que recibe como argumento y las señales recibidas por este. Tipeando

```
$ strace -i hola
```

, hacemos que además de proveer la información provista por el comando, agregue el valor del puntero de instrucciones del procesador al inicio de la línea (opción *-i*).
Omitimos volcar en este documento la salida de *strace*, debido a su extensión. El lector la puede ver en su propia instalación.
La primer línea de su extensa salida es:

```
[b7f79424] execve("./hola", ["hola"], [/* 32 vars */]) = 0
```

Tipeando *man execve* puede verse que el primer argumento es el nombre del archivo binario que contiene el código por el que se debe reemplazar la réplica del proceso padre que invocó a *fork ()*.
El segundo argumento es la lista de argumentos que se ingresan por línea de

comandos. Como es de práctica es un *char * argv []*, y el tercero es la lista de variables de entorno que se deseen pasar al nuevo programa.
Si *execve* funciona correctamente no regresa al programa invocador.
Las dos líneas finales del listado generado por *strace* son:

```
[b7f63424] write(1, "Hola mundo!\n"..., 12) = 12
[b7f63424] exit_group(0)                = ?
```

La función *write ()* es producto del código invocado desde *puts* y que no hace mas que escribir en *stdout* (o sea en la pantalla), y *exit ()* es la salida del programa al proceso padre (es decir a *bash*)
Para agregar algunos detalles interesantes el comando siguiente provee información de interés:

```
$ readelf -l hola

Elf file type is EXEC (Executable file)
Entry point 0x80482f0
There are 7 program headers, starting at offset 52

Program Headers:
 Type      Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
 PHDR      0x000034 0x08048034 0x08048034 0x000e0 0x000e0 R E 0x4
 INTERP    0x000114 0x08048114 0x08048114 0x00013 0x00013 R   0x1
     [Requesting program interpreter: /lib/ld-linux.so.2]
 LOAD      0x000000 0x08048000 0x08048000 0x004a0 0x004a0 R E 0x1000
 LOAD      0x0004a0 0x080494a0 0x080494a0 0x00108 0x00110 RW  0x1000
 DYNAMIC   0x0004b4 0x080494b4 0x080494b4 0x000d0 0x000d0 RW  0x4
 NOTE      0x000128 0x08048128 0x08048128 0x00020 0x00020 R   0x4
 GNU_STACK 0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x4

 Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02  .interp .note.ABI-tag .hash .gnu.hash .dynsym .dynstr
.gnu.version .gnu.version_r .rel.dyn .rel.plt .init .plt .text
.fini .rodata .eh_frame
   03     .ctors .dtors .jcr .dynamic .got .got.plt .data .bss
   04     .dynamic
   05     .note.ABI-tag
   06
```

La salida del comando *readelf* permite ver la estructura completa del programa, que ahora tiene un formato similar al del archivo objeto pero con la vista de un ejecutable en lugar que la de un objeto.
La principal diferencia está en que lo que en el encabezado de un archivo objeto se denominan secciones en el encabezado de un ejecutable se denominan Segmentos. Esto puede verse claramente indicado en la salida de *readelf*.

La salida se divide claramente en dos partes: El encabezado ELF, y los segmentos. Ambas están relacionadas como veremos a continuación.
El tercer *Program header*,

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

corresponde al segmento de código que es la sección *.text* generada en el programa objeto por el compilador.
La columna offset define el lugar en el que inicia el segmento relativo al comienzo del programa, en nuestro caso 0x00000000, la columna *Virtual Address*[4] indica 0x08048000, al igual que la Columna correspondiente a la *Physical Address*.
El valor de la columna *FileSiz* indica el tamaño del segmento en el archivo en disco y el de la columna *MemSiz* indica el tamaño que ocupará el segmento en la memoria del sistema. El valor de la columna *Align*, establece la alineación que llevará en la memoria (es decir a partir de que múltiplo de valores de address el Sistema Operativo cargará a ese segmento). El segmento de código correspondiente a la sección *.text* de nuestro programa, y los datos de solo lectura que el archivo *ELF* organiza en la sección *.rodata* se cargarán entonces en este segmento a partir de la dirección virtual 0x08048000, alineado a direcciones múltiplo de 0x1000, es decir con sus 12 bits menos significativos en '0'[5]. Además de acuerdo a las necesidades aparecidas durante el proceso de linkeo es probable que en este segmento se carguen secciones de código y datos de solo lectura que incluya el linker en el *ELF* ejecutable. Tratándose entonces de código y datos para lectura solamente no es extraño que en la columna *Flg* de la salida del comando *readelf*, este segmento esté marcado R (Read only), y E (Executable) en lugar de W (Writable)
El cuarto header corresponde al segmento *.data* del proceso y su interpretación responde a las mismas reglas del caso anterior. Veámoslo:

```
  03     .ctors .dtors .jcr .dynamic .got .got.plt .data .bss
```

La diferencia de tamaño entre el archivo (*FileSiz*) y la imagen en memoria (*MemSiz*) se debe a que se incluye en este segmento la sección *.bss*.
No obstante en este programa dicha sección no contiene nada y se llena con ceros. Al igual que en el caso anterior se alinea a 0x1000, y como corresponde a un segmento de datos se marca W (Writable).
El quinto header (DYNAMIC) corresponde al proceso de linkeo (recordemos que el programa *collect2* realizó un linkeo a librerías dinámicas.

```
  04     .dynamic
```

En general los sistemas Linux poseen un file system mapeado en RAM

4 La dirección Virtual es propia de Linux y puede ser la misma para todos los programas, ya que le asigna a cada uno un espacio virtual que luego el procesador traduce a direcciones físicas diferentes en el momento de cargarlo en memoria para su ejecución. Por eso el campo Dirección física que arroja *readelf* no tiene significado ya que esta dirección será calculada en tiempo de ejecución, por el procesador.

5 Durante el proceso de traducción de dirección virtual a dirección física el procesador organiza a la memoria del sistema en páginas iguales de 4Kbytes de tamaño. De esto surge la necesidad de alinear a 0x1000.

directamente en tiempo de boot (no está presente en el disco rígido), denominado */proc*.
En éste se almacena información de suma utilidad y que es administrada solamente por el kernel. Los usuarios sin embargo pueden ver su contenido y muchas veces obtener información sumamente útil de él.
En particular el kernel construye un directorio para cada proceso cuyo nombre es el PID del proceso, y dentro de este un árbol de archivos y directorios con información sumamente útil e interesante.
En particular el directorio */proc/[PID_de_nuestro_proceso]/maps* contiene información muy similar a la obtenida de la salida de *readelf.*
Claramente, el árbol de directorio de cada proceso existe en */proc*, solo durante la vida del proceso. En nuestro caso ésta es efímera ya que todo lo que hacemos es presentar la leyenda "Hola mundo", y terminar el programa. ¿Como podemos hacer para detener nuestro programa de modo tal de darnos el tiempo para explorar el directorio?. Una opción es debugearlo. Eso creará una instancia que estará disponible en memoria y con un PID asociado durante todo el tiempo que queramos.
Una opción (rústica pero muy potente) es utilizar el programa *gdb* (GNU DeBuger).
La secuencia de comandos necesarios y su salida es la siguiente:

```
$ gdb
GNU gdb 6.8-debian
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
<http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show
copying"
and "show warranty" for details.
This GDB was configured as "i486-linux-gnu".
(gdb) file hola
Reading symbols from /home/alejandro/InfoI/hola...done.
(gdb) break main
Breakpoint 1 at 0x80483b2
(gdb) run
Starting program: /home/alejandro/InfoI/first/hola

Breakpoint 1, 0x080483b2 in main ()
Current language:  auto; currently asm
(gdb)
```

Los tres comandos que se ejecutan una vez dentro de *gdb* son los que hemos resaltado en negrita, y que por orden de ejecución se encargan de cargar el ejecutable hola en el contexto de *gdb*, colocar un *breackpoint* en la etiqueta *main* (es decir en el punto de inicio del programa), y ejecutarlo.
Obviamente al ejecutar el programa este se detendrá en el *breackpoint* hasta que continuemos ejecutándolo en forma completa o paso a paso.

Lo importante aquí es que el proceso está cargado en memoria y detenido, lo cual nos permite explorar su entrada en */proc*. Pero antes debemos tener su *PID*.
Para ello la siguiente secuencia nos permite alcanzar el resultado:

```
$ ps -C hola -o pid
  PID
 5398
```

El comando *ps* como ya vimos lista los procesos de acuerdo con determinados criterios que podemos acotar mediante las opciones disponibles (como siempre *man ps* ampliará la información de las opciones del comando). En este caso *-C* indica que presente los procesos cuyo nombre coincida con la cadena que se tipea a continuación, en nuestro caso, *hola*, y *-o* permite customizar la salida colocando a continuación el nombre de la(s) columnas(s) que deseamos presentar a la salida.

```
$ cat /proc/5398/maps
08048000-08049000 r-xp 00000000 08:02 3145866    /home/alejandro/InfoI/
hola
08049000-0804a000 rw-p 00000000 08:02 3145866    /home/alejandro/InfoI/
hola
b7dc5000-b7dc6000 rw-p b7dc5000 00:00 0
b7dc6000-b7f1b000         r-xp        00000000        08:02        294942
/lib/i686/cmov/libc-2.7.so
b7f1b000-b7f1c000         r--p        00155000        08:02        294942
/lib/i686/cmov/libc-2.7.so
b7f1c000-b7f1e000         rw-p        00156000        08:02        294942
/lib/i686/cmov/libc-2.7.so
b7f1e000-b7f23000 rw-p b7f1e000 00:00 0
b7f23000-b7f24000 r-xp b7f23000 00:00 0          [vdso]
b7f24000-b7f3e000 r-xp 00000000 08:02 278572     /lib/ld-2.7.so
b7f3e000-b7f40000 rw-p 0001a000 08:02 278572     /lib/ld-2.7.so
bff2b000-bff40000 rw-p bffeb000 00:00 0          [stack]
```

Aquí vemos el segmento de código en la primer línea, con las direcciones a partir de las cuales se ha papeado, y a continuación el segmento de datos (contiene las secciones *.data .bss* y el *heap*).
A continuación una serie de líneas correspondientes a las librerías dinámicas con las que el linker enlazó nuestro breve código para poderlo convertir en algo ejecutable dentro de Linux, y en el final el *stack*. Este último no tiene correspondencia alguna con el formato ELF.

Finalización

Cuando nuestro programa ejecuta dentro de *main* la función *return*, devuelve el control a una de las funciones de la librería dinámica contra la que se linkea nuestro programa para poder ejecutarse, y dicha función invoca a la *system*

call exit pasándole el mismo argumento que lleva *return* en su código.
Por su parte *exit*, actúa sobre el proceso padre del que ejecuta *hola*, el cual está a su vez ejecutando una *system call* denominada *wait*. Como resultado el proceso padre estará bloqueado en esa función esperando (waiting) a que el proceso hijo, en nuestro caso *hola*, finalice su ejecución.
De este modo nuestro proceso devolverá al sistema operativo de manera ordenada todos los recursos que se le han asignado y su padre (en este caso el proceso *shell*) quedará liberado en lo que a este proceso hijo se refiere.
Podemos mediante el comando *strace*, desentrañar en parte el proceso de finalización, tipeando lo siguiente:

```
$ strace -e trace=process -f sh -c "hola; echo $?" > /dev/null
execve("/bin/sh", ["sh", "-c", "hola; echo 0"], [/* 32 vars */]) = 0
clone(Process 6142 attached
child_stack=0,   flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD,
child_tidptr=0xb7da96f8) = 6142
[pid  6142] execve("./hola", ["hola"], [/* 32 vars */] <unfinished ...>
[pid  6141] waitpid(-1, Process 6141 suspended
 <unfinished ...>
[pid  6142] <... execve resumed> )      = 0
[pid  6142] exit_group(0)               = ?
Process 6141 resumed
Process 6142 detached
<... waitpid resumed> [{WIFEXITED(s) && WEXITSTATUS(s) == 0}], 0) =
6142
--- SIGCHLD (Child exited) @ 0 (0) ---
waitpid(-1, 0xbf856c98, WNOHANG)              = -1 ECHILD (No child
processes)
exit_group(0)                           = ?
```

El comando *strace* es sumamente potente, y por lo tanto involucra una cantidad de conceptos que aún no estamos en condiciones de abordar debido al nivel inicial de esta guía. Sin embargo, podemos ver en su salida todas las llamadas al sistema operativo que hizo nuestro proceso. Esto es debido al empleo en el comando de la opción *-e trace=process*, que le indica que se presente a la salida del comando todas las *system calls* y señales involucradas.
La opción -f sirve para especificar allí el comando que queremos ejecutar, y que el proceso *strace* siga las *system calls* y señales del proceso especificado y de sus procesos hijos. En nuestro caso lo que hemos hecho es ejecutar una instancia del *shell* (*sh*) con la opción -c para que el comando lo tome de la string de caracteres contigua en lugar del *stdin* (teclado).

Conclusiones

El objeto de esta guía es que se entienda la complejidad oculta detrás de cualquier aplicación y los recursos del sistema operativo que entran en juego,

para la ejecución de una aplicación cualquiera, independientemente del lenguaje en el que ésta se haya programado. Lo que aquí vimos ocurre tanto para un programa escrito en C como para una applet Java, o scritp en Phyton. El proceso es siempre el mismo.

Apéndice A

```
$ objdump -hrt /usr/lib/crt1.o

/usr/lib/crt1.o:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .note.ABI-tag 00000020  00000000  00000000  00000034  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  1 .text         00000024  00000000  00000000  00000054  2**2
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  2 .rodata       00000004  00000000  00000000  00000078  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .rodata.cst4  00000004  00000000  00000000  0000007c  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .data         00000004  00000000  00000000  00000080  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  5 .bss          00000000  00000000  00000000  00000084  2**2
                  ALLOC
  6 .comment      0000001f  00000000  00000000  00000084  2**0
                  CONTENTS, READONLY
  7 .debug_pubnames 00000025  00000000  00000000  000000a3  2**0
                  CONTENTS, RELOC, READONLY, DEBUGGING
  8 .debug_info   0000008d  00000000  00000000  000000c8  2**0
                  CONTENTS, RELOC, READONLY, DEBUGGING
  9 .debug_abbrev 0000004b  00000000  00000000  00000155  2**0
                  CONTENTS, READONLY, DEBUGGING
 10 .debug_line   00000027  00000000  00000000  000001a0  2**0
                  CONTENTS, READONLY, DEBUGGING
 11 .debug_str    000000dc  00000000  00000000  000001c7  2**0
                  CONTENTS, READONLY, DEBUGGING
 12 .note.GNU-stack 00000000  00000000  00000000  000002a3  2**0
                  CONTENTS, READONLY
SYMBOL TABLE:
00000000 l    d  .note.ABI-tag  00000000 .note.ABI-tag
00000000 l    d  .text  00000000 .text
00000000 l    d  .rodata        00000000 .rodata
00000000 l    d  .rodata.cst4   00000000 .rodata.cst4
00000000 l    d  .data  00000000 .data
00000000 l    d  .bss   00000000 .bss
00000000 l    d  .comment       00000000 .comment
00000000 l    d  .debug_pubnames        00000000 .debug_pubnames
00000000 l    d  .debug_info    00000000 .debug_info
00000000 l    d  .debug_abbrev  00000000 .debug_abbrev
00000000 l    d  .debug_line    00000000 .debug_line
00000000 l    d  .debug_str     00000000 .debug_str
00000000 l    d  .note.GNU-stack        00000000 .note.GNU-stack
00000000 l    df *ABS*  00000000 init.c
00000000 g     O .rodata        00000004 _fp_hw
00000000         *UND*  00000000 __libc_csu_fini
00000000 g     F .text  00000000 _start
00000000         *UND*  00000000 __libc_csu_init
```

```
00000000         *UND*  00000000 main
00000000  w      .data  00000000 data_start
00000000 g     O .rodata.cst4   00000004 _IO_stdin_used
00000000         *UND*  00000000 __libc_start_main
00000000 g       .data  00000000 __data_start


RELOCATION RECORDS FOR [.text]:
OFFSET   TYPE              VALUE
0000000c R_386_32          __libc_csu_fini
00000011 R_386_32          __libc_csu_init
00000018 R_386_32          main
0000001d R_386_PC32        __libc_start_main


RELOCATION RECORDS FOR [.debug_pubnames]:
OFFSET   TYPE              VALUE
00000006 R_386_32          .debug_info


RELOCATION RECORDS FOR [.debug_info]:
OFFSET   TYPE              VALUE
00000006 R_386_32          .debug_abbrev
0000000c R_386_32          .debug_str
00000011 R_386_32          .debug_str
00000015 R_386_32          .debug_str
00000019 R_386_32          .text
0000001d R_386_32          .text
00000021 R_386_32          .debug_line
00000028 R_386_32          .debug_str
0000002f R_386_32          .debug_str
00000036 R_386_32          .debug_str
0000003d R_386_32          .debug_str
00000044 R_386_32          .debug_str
0000004b R_386_32          .debug_str
00000059 R_386_32          .debug_str
00000060 R_386_32          .debug_str
00000067 R_386_32          .debug_str
00000071 R_386_32          .debug_str
00000076 R_386_32          .debug_str
00000083 R_386_32          _IO_stdin_used
```

```
$ objdump -hrt /usr/lib/crti.o

/usr/lib/crti.o:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000000  00000000  00000000  00000034  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .data         00000000  00000000  00000000  00000034  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000000  00000000  00000000  00000034  2**2
                  ALLOC
  3 .init         00000022  00000000  00000000  00000034  2**2
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
```

```
  4 .fini         00000013  00000000  00000000  00000058  2**2
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  5 .comment      0000001f  00000000  00000000  0000006b  2**0
                  CONTENTS, READONLY
  6 .note.GNU-stack 00000000  00000000  00000000  0000008a  2**0
                  CONTENTS, READONLY
  7 .debug_line   000000aa  00000000  00000000  0000008a  2**0
                  CONTENTS, RELOC, READONLY, DEBUGGING
  8 .debug_info   0000008d  00000000  00000000  00000134  2**0
                  CONTENTS, RELOC, READONLY, DEBUGGING
  9 .debug_abbrev 00000012  00000000  00000000  000001c1  2**0
                  CONTENTS, READONLY, DEBUGGING
 10 .debug_aranges 00000028  00000000  00000000  000001d8  2**3
                  CONTENTS, RELOC, READONLY, DEBUGGING
 11 .debug_ranges 00000020  00000000  00000000  00000200  2**3
                  CONTENTS, RELOC, READONLY, DEBUGGING
SYMBOL TABLE:
00000000 l    df *ABS*  00000000 initfini.c
00000000 l    d  .text  00000000 .text
00000000 l    d  .data  00000000 .data
00000000 l    d  .bss   00000000 .bss
00000000 l    d  .init  00000000 .init
00000000 l    d  .fini  00000000 .fini
00000000 l    d  .note.GNU-stack        00000000 .note.GNU-stack
00000000 l    d  .debug_info    00000000 .debug_info
00000000 l    d  .debug_abbrev  00000000 .debug_abbrev
00000000 l    d  .debug_line    00000000 .debug_line
00000000 l    d  .debug_ranges  00000000 .debug_ranges
00000000 l    d  .comment       00000000 .comment
00000000 l    d  .debug_aranges 00000000 .debug_aranges
00000000  w      *UND*  00000000 __gmon_start__
00000000 g     F .init  00000000 _init
00000000         *UND*  00000000 _GLOBAL_OFFSET_TABLE_
00000000 g     F .fini  00000000 _fini


RELOCATION RECORDS FOR [.init]:
OFFSET   TYPE              VALUE
0000000f R_386_GOTPC       _GLOBAL_OFFSET_TABLE_
00000015 R_386_GOT32       __gmon_start__
0000001e R_386_PLT32       __gmon_start__


RELOCATION RECORDS FOR [.fini]:
OFFSET   TYPE              VALUE
0000000f R_386_GOTPC       _GLOBAL_OFFSET_TABLE_


RELOCATION RECORDS FOR [.debug_line]:
OFFSET   TYPE              VALUE
0000007c R_386_32          .init
00000097 R_386_32          .fini


RELOCATION RECORDS FOR [.debug_info]:
OFFSET   TYPE              VALUE
00000006 R_386_32          .debug_abbrev
```

```
0000000c R_386_32          .debug_line
00000010 R_386_32          .debug_ranges


RELOCATION RECORDS FOR [.debug_aranges]:
OFFSET   TYPE              VALUE
00000006 R_386_32          .debug_info
00000010 R_386_32          .init
00000018 R_386_32          .fini


RELOCATION RECORDS FOR [.debug_ranges]:
OFFSET   TYPE              VALUE
00000008 R_386_32          .init
0000000c R_386_32          .init
00000010 R_386_32          .fini
00000014 R_386_32          .fini
```

```
$ objdump -hrt /usr/lib/crtn.o

/usr/lib/crtn.o:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000000  00000000  00000000  00000034  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .data         00000000  00000000  00000000  00000034  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000000  00000000  00000000  00000034  2**2
                  ALLOC
  3 .init         00000004  00000000  00000000  00000034  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  4 .fini         00000004  00000000  00000000  00000038  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  5 .comment      0000001f  00000000  00000000  0000003c  2**0
                  CONTENTS, READONLY
  6 .note.GNU-stack 00000000  00000000  00000000  0000005b  2**0
                  CONTENTS, READONLY
  7 .debug_line   00000054  00000000  00000000  0000005b  2**0
                  CONTENTS, RELOC, READONLY, DEBUGGING
  8 .debug_info   00000065  00000000  00000000  000000af  2**0
                  CONTENTS, RELOC, READONLY, DEBUGGING
  9 .debug_abbrev 00000012  00000000  00000000  00000114  2**0
                  CONTENTS, READONLY, DEBUGGING
 10 .debug_aranges 00000028  00000000  00000000  00000128  2**3
                  CONTENTS, RELOC, READONLY, DEBUGGING
 11 .debug_ranges 00000020  00000000  00000000  00000150  2**3
                  CONTENTS, RELOC, READONLY, DEBUGGING
SYMBOL TABLE:
00000000 l    df *ABS*  00000000 initfini.c
00000000 l    d  .text  00000000 .text
00000000 l    d  .data  00000000 .data
00000000 l    d  .bss   00000000 .bss
00000000 l    d  .init  00000000 .init
00000000 l    d  .fini  00000000 .fini
00000000 l    d  .note.GNU-stack        00000000 .note.GNU-stack
```

```
00000000 l    d  .debug_info    00000000 .debug_info
00000000 l    d  .debug_abbrev  00000000 .debug_abbrev
00000000 l    d  .debug_line    00000000 .debug_line
00000000 l    d  .debug_ranges  00000000 .debug_ranges
00000000 l    d  .comment       00000000 .comment
00000000 l    d  .debug_aranges 00000000 .debug_aranges


RELOCATION RECORDS FOR [.debug_line]:
OFFSET   TYPE              VALUE
00000033 R_386_32          .init
00000045 R_386_32          .fini


RELOCATION RECORDS FOR [.debug_info]:
OFFSET   TYPE              VALUE
00000006 R_386_32          .debug_abbrev
0000000c R_386_32          .debug_line
00000010 R_386_32          .debug_ranges


RELOCATION RECORDS FOR [.debug_aranges]:
OFFSET   TYPE              VALUE
00000006 R_386_32          .debug_info
00000010 R_386_32          .init
00000018 R_386_32          .fini


RELOCATION RECORDS FOR [.debug_ranges]:
OFFSET   TYPE              VALUE
00000008 R_386_32          .init
0000000c R_386_32          .init
00000010 R_386_32          .fini
00000014 R_386_32          .fini
```

```
$ objdump -hrt /usr/lib/gcc/i486-linux-gnu/4.3.2/crtbegin.o

/usr/lib/gcc/i486-linux-gnu/4.3.2/crtbegin.o:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000083  00000000  00000000  00000040  2**4
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1 .data         00000004  00000000  00000000  000000c4  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000008  00000000  00000000  000000c8  2**2
                  ALLOC
  3 .ctors        00000004  00000000  00000000  000000c8  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  4 .dtors        00000004  00000000  00000000  000000cc  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  5 .jcr          00000000  00000000  00000000  000000d0  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  6 .fini         00000005  00000000  00000000  000000d0  2**0
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  7 .init         00000005  00000000  00000000  000000d5  2**0
```

```
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  8 .comment      0000001f  00000000  00000000  000000da  2**0
                  CONTENTS, READONLY
  9 .note.GNU-stack 00000000  00000000  00000000  000000f9  2**0
                  CONTENTS, READONLY
SYMBOL TABLE:
00000000 l    df *ABS*  00000000 crtstuff.c
00000000 l    d  .text  00000000 .text
00000000 l    d  .data  00000000 .data
00000000 l    d  .bss   00000000 .bss
00000000 l    d  .ctors 00000000 .ctors
00000000 l     O .ctors 00000000 __CTOR_LIST__
00000000 l    d  .dtors 00000000 .dtors
00000000 l     O .dtors 00000000 __DTOR_LIST__
00000000 l    d  .jcr   00000000 .jcr
00000000 l     O .jcr   00000000 __JCR_LIST__
00000000 l     F .text  00000000 __do_global_dtors_aux
00000000 l     O .bss   00000001 completed.5706
00000004 l     O .bss   00000004 dtor_idx.5708
00000000 l    d  .fini  00000000 .fini
00000060 l     F .text  00000000 frame_dummy
00000000 l    d  .init  00000000 .init
00000000 l    d  .note.GNU-stack        00000000 .note.GNU-stack
00000000 l    d  .comment       00000000 .comment
00000000 g     O .data  00000000 .hidden __dso_handle
00000000         *UND*  00000000 .hidden __DTOR_END__
00000000  w      *UND*  00000000 _Jv_RegisterClasses


RELOCATION RECORDS FOR [.text]:
OFFSET   TYPE              VALUE
00000009 R_386_32          .bss
00000012 R_386_32          .bss
00000017 R_386_32          __DTOR_END__
0000001c R_386_32          .dtors
00000034 R_386_32          .bss
0000003b R_386_32          .dtors
00000041 R_386_32          .bss
0000004b R_386_32          .bss
00000067 R_386_32          .jcr
00000070 R_386_32          _Jv_RegisterClasses
0000007b R_386_32          .jcr


RELOCATION RECORDS FOR [.fini]:
OFFSET   TYPE              VALUE
00000001 R_386_PC32        .text


RELOCATION RECORDS FOR [.init]:
OFFSET   TYPE              VALUE
00000001 R_386_PC32        .text
```

```
$ objdump -hrt /usr/lib/gcc/i486-linux-gnu/4.3.2/crtend.o
```

```
/usr/lib/gcc/i486-linux-gnu/4.3.2/crtend.o:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         0000002a  00000000  00000000  00000040  2**4
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1 .data         00000000  00000000  00000000  0000006c  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000000  00000000  00000000  0000006c  2**2
                  ALLOC
  3 .ctors        00000004  00000000  00000000  0000006c  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  4 .dtors        00000004  00000000  00000000  00000070  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  5 .eh_frame     00000004  00000000  00000000  00000074  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  6 .jcr          00000004  00000000  00000000  00000078  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  7 .init         00000005  00000000  00000000  0000007c  2**0
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  8 .comment      0000001f  00000000  00000000  00000081  2**0
                  CONTENTS, READONLY
  9 .note.GNU-stack 00000000  00000000  00000000  000000a0  2**0
                  CONTENTS, READONLY
SYMBOL TABLE:
00000000 l    df *ABS*  00000000 crtstuff.c
00000000 l    d  .text  00000000 .text
00000000 l    d  .data  00000000 .data
00000000 l    d  .bss   00000000 .bss
00000000 l    d  .ctors 00000000 .ctors
00000000 l     O .ctors 00000000 __CTOR_END__
00000000 l    d  .dtors 00000000 .dtors
00000000 l    d  .eh_frame      00000000 .eh_frame
00000000 l     O .eh_frame      00000000 __FRAME_END__
00000000 l    d  .jcr   00000000 .jcr
00000000 l     O .jcr   00000000 __JCR_END__
00000000 l     F .text  00000000 __do_global_ctors_aux
00000000 l    d  .init  00000000 .init
00000000 l    d  .note.GNU-stack        00000000 .note.GNU-stack
00000000 l    d  .comment       00000000 .comment
00000000 g     O .dtors 00000000 .hidden __DTOR_END__


RELOCATION RECORDS FOR [.text]:
OFFSET   TYPE              VALUE
00000008 R_386_32          .ctors
00000012 R_386_32          .ctors


RELOCATION RECORDS FOR [.init]:
OFFSET   TYPE              VALUE
00000001 R_386_PC32        .text
```