Compilación y linking

Marcelo Arroyo
Departamento de Computación, FCEFQyN
Universidad Nacional de Río Cuarto

2017

Resumen

En este pequeño tutorial se describen los detalles del proceso de compilación de programas C/C++ (aunque también son aplicables a otros lenguajes). Además de describen los detalles de los formatos de archivos objeto, ejecutables y blbiotecas. Los ejemplos descriptos se basan en el uso de la suite de compilación `GCC (GNU Compiler Collection)` aunque los conceptos son aplicables en otras suites como `CLang-LLVM` o `MS Visual C/C++`.

Finalmente, se describen los detalles de la carga, ejecución e imagen de un proceso en memoria que es aplicable en varios sistemas operativos modernos y la compilación y enlace (linking) de programas con bibliotecas compartidas, tabién conocidas como *shared objects* o *dynamic link libraries (DLLs)*.

1. Compilación de programas C/C++

La compilación de programas C/C++ se realiza en diferentes etapas y con diferentes herramientas, como se muestra en la figura 1.

Cada paso se describe a continuación:

1. **Preprocesamiento**: Se procesan las directivas del pre-procesador (`#include, #define, #ifdef, ...`) contenidas en el archivo fuente. Este preprocesamiento genera archivos temporales de salida (`.i`) con el resultado de la evaluación de las directivas (macros e inclusiones).

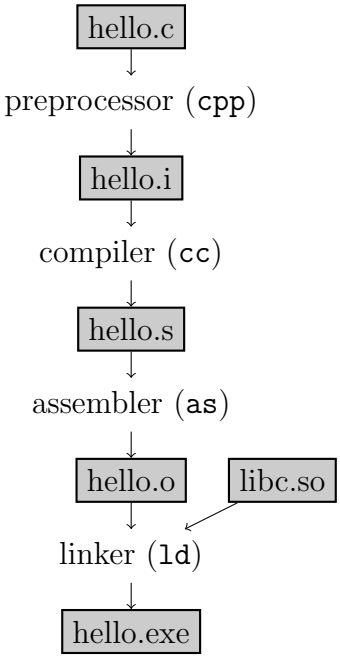

Figura 1: Pasos del proceso de compilación.

2. **Compilación**: Toma la salida de la etapa anterior, checkea sintaxis y semántica del código fuente, realiza optimizaciones y genera código de ensamble *assembly* (`.s`).

3. **Ensamble (assembly)**: Toma un archivo fuente *assembly* y genena un *archivo objeto* (`.o`).

4. **Enlazado (linking)**: El *linker* toma uno o más archivos objectos y las bibliotecas requeridas para producir el *archivo ejecutable o programa*.

El compilador `GCC`[1] incluye todas las herramientas (c-preprocessor, compiler, assembler, ...) y utiliza las siguientes convenciones sobre las extensiones (o sufijos en el caso de los sistemas tipo UNIX) de los archivos.

Extensión	**Descripción**
filename.c	Archivo fuente C
filename.h	Archivo de inclusión C/C++ que no debe ser compilado ni enlazado
filename.hpp	Archivo fuente de inclusión C++
filename.c++ **filename.cc** **filename.cpp** **filename.cxx** **filename.C**	Archivo fuente C++
filename.i	Archivo fuente C que no debe ser preprocesado
filename.ii	Archivo fuente C++ que no debe ser preprocesado
filename.s	Archivo fuente assembly
filename.S	Archivo fuente assembly que debe ser preprocesado
filename.o	Archivo objeto

Cuando se compila un programa fuente C para producir un ejecutable, generalmente se realiza el siguiente comando:

```
gcc -o hello hello.c
```

En realidad el comando `gcc` es un comando de alto nivel, un *front-end (compiler driver)*, que se encarga de ejecutar todos los pasos de la compilación descriptos de manera automática.

Un programador puede utilizar los comandos individuales. A modo de ejemplo, el comando

[1] GCC: GNU compiler collection.

`cpp hello.c`
envía por la consola (salida estándar) el resultado del preprocesamiento del archivo fuente `hello.c`.

El comando `gcc` soporta opciones para detener el proceso de compilación en cualquiera de sus etapas.

A continuación se describen las opciones (*flags*) que pueden utilizarse en `gcc` (y `clang-llvm`).

Opción	**Descripción**
-E	Finaliza antes de la etapa de compilación
-S	Finaliza antes de la etapa de *ensamble* (genera `.s`)
-c	Finaliza antes de la etapa de *linking* (genera `.o`)

2. Archivos objeto, bibliotecas y ejecutables

El *assembler* genera un archivo objeto. Un archivo objeto es una *unidad compilada* que puede ser enlazada con otros archivos objetos y bibliotecas para formar un *ejecutable*.

El formato de los archivos objeto, bibliotecas y ejecutables depende de los formatos soportados por el sistema operativo.

A continuación se listan los formatos más ampliamente utilizados.

- **a.out**: Formato original de los sistemas UNIX[2].
- `Common Object File Format (COFF)`: Introducido por el *System V Release 3 UNIX*.
- `Portable Executable (PE)`: Utilizado en sistemas MS-Windows (es un `COFF` extendido).
- `Executable and Linking Format (ELF)`: Usado en varios sistemas UNIX modernos (GNU-Linux, Solaris, ...)

Si bien hay diferencias entre los formatos, todos tienen secciones comunes (a veces con diferentes nombres). Una sección puede contener instrucciones

[2]Por eso el `gcc` genera ejecutables con ese nombre si se omite el nombre del archivo de salida.

(código) de máquina, datos, tablas de símbolos, información para enlazado dinámico, datos para depuración (relación entre instrucciones de máquina con líneas de código fuente), comentarios y notas, etc.

Las principales secciones son:

- `.text`: Instrucciones de máquina (código).
- `.bss` (*Block Started by Symbol*): Información sobre el tamaño del bloque de datos estáticos (variables globales) no inicializadas. Esto le provee información al sistema operativo para determinar cuánta memoria reservar para las variables globales no inicializadas.
- `.data`: Valores de las variables globales y estáticas inicializadas. Estos valores se *cargan* en la imagen de memoria del proceso.
- `.rodata`: Valores constantes y literales (ej: strings).
- `Symbol table`: Tabla que mapea símbolos (identificadores) con sus direcciones de memoria. Estas direcciones pueden ser virtuales (lógicas) o absolutas.
- `Relocation records`: Registros de información usados por el *linker* para ajustar contenidos de secciones. Por ejemplo, estos registros describen las direcciones de memoria que un linker tiene que ajustar (modificar) cuando se combinan (enlazan) diferentes archivos objeto para formar un ejecutable.

Cada sistema operativo incluye un conjunto de utilidades para analizar y manipular archivos objetos. En sistemas tipo UNIX podemos mencionar `readelf` para formato `ELF` y `objdump`. Estas utilidades se encuentran en el paquete `binutils`. En sistemas MS-Windows puede utilizarse `PEBrowse`.

A modo de ejemplo, dado el siguiente programa C, con dos módulos `main.c` y `mylib.c`:

```
/* main.c */
#include <stdio.h>

int g=1000;

extern int f(int);
```

```
/* mylib.c */
int f(int x)
{
   return x+1;
}
```

```
int main(void)
{
  f(g);
}
```

Luego de ejecutar el comando `gcc -c main.c`, es posible analizar las secciones del archivo objeto `hello.o` generado mediante el comando `objdump -x hello.o`:

```
main.o
arquitectura: i386:x86-64, opciones 0x00000011:
HAS_RELOC, HAS_SYMS
dirección de inicio 0x0000000000000000

Secciones:
Ind Nombre          Tamaño    VMA               LMA               Desp fich Alin
  0 .text           00000020  0000000000000000  0000000000000000  00000220  2**4
                    CONTENTS, ALLOC, LOAD, RELOC, CODE
  1 .data           00000004  0000000000000020  0000000000000020  00000240  2**2
                    CONTENTS, ALLOC, LOAD, DATA
  2 __LD.__compact_unwind 00000020  0000000000000028  0000000000000028  00000248  2**3
                    CONTENTS, RELOC, DEBUGGING
  3 .eh_frame       00000040  0000000000000048  0000000000000048  00000268  2**3
                    CONTENTS, ALLOC, LOAD, READONLY, DATA
SYMBOL TABLE:
0000000000000020 g       0f SECT   02 0000 [.data] _g
0000000000000000 g       0f SECT   01 0000 [.text] _main
0000000000000000 g       01 UND    00 0000 _f

RELOCATION RECORDS FOR [.text]:
OFFSET           TYPE              VALUE
000000000000000f BRANCH32          _f
000000000000000a DISP32            _g

RELOCATION RECORDS FOR [__LD.__compact_unwind]:
OFFSET           TYPE              VALUE
0000000000000000 64                .text
```

El archivo objeto contiene 4 secciones. La sección `.text` contiene el código (de función `main`). La sección `.data` contiene datos globales (inicializados). En particular contiene la variable global `g`, que ocupa 4 bytes. Estas dos secciones debarán serán cargadas (nota que están marcadas con `LOAD`) por el

sistema operativo cuando se deba ejecutar el programa.

La tabla de símbolos contiene tres entradas que definen los símbolos globales del programa: las funciones `main`, `f` y la variable global `g`[3].

Cada entrada en la tabla de símbolos contiene información sobre la dirección (relativa) de memoria asociada al identificador y si está definido o no. Por ejemplo, el símbolo `_main` tiene un offset (dirección relativa al segmento `.text`) cero, la variable global `_g` también (relativa al segmento `.data`), y el símbolo `_f` está indefinido (`UND`) ya que no está definido en éste archivo objeto.

Decimos que `_f` es una *referencia externa* en `_main`.

Los registros de reubicación definen cuáles son los símbolos (en realidad dónde están los valores de las direcciones de memoria en el código) que deberán reubicarse o recomputarse durante el enlazado. En este ejemplo, las direcciones de destino del llamado a la función `_f` (operando de la instrucción `callp _f`) y la dirección de la variable global `g`.

A continuación se muestra el contenido (desensamble) de la sección `.text` de `main.o` (`objdump -d main.o`).

```
main.o:     formato del fichero mach-o-x86-64
Desensamblado de la sección .text:

0000000000000000 <_main>:
   0: 55                    push   %rbp
   1: 48 89 e5              mov    %rsp,%rbp
   4: 48 83 ec 10           sub    $0x10,%rsp
   8: 8b 3d 00 00 00 00     mov    0x0(%rip),%edi        # e <_main+0xe>
   e: e8 00 00 00 00        callq  13 <_main+0x13>
  13: 31 ff                 xor    %edi,%edi
  15: 89 45 fc              mov    %eax,-0x4(%rbp)
  18: 89 f8                 mov    %edi,%eax
  1a: 48 83 c4 10           add    $0x10,%rsp
  1e: 5d                    pop    %rbp
  1f: c3                    retq
```

Es interesante ver que en la instrucción `callq 13 <_main+0x13>` la dirección destino es cero (el contenido del *program counter (PC)* en ese momento),

[3]Notar que el compilador nombra a los identificadores de función precediéndolas de un *underscore* (`'_'`).

no la dirección de la función `_f`. Como ya vimos, la dirección de destino de la instrucción es un *relocation record*.

Ea tarea del *linker* combinar ambos módulos, es decir, concatenar los segmentos `.text` y `.data` para luego calcular las dirección absoluta de `_f` y actualizar la dirección de la invocación en `_main`.

Al generar el archivo objeto `mylib.o` (mediante el comando `gcc -c mylib.c`), podemos observar su contenido:

```
mylib.o
arquitectura: i386:x86-64, opciones 0x00000011:
HAS_RELOC, HAS_SYMS
dirección de inicio 0x0000000000000000

Secciones:
Ind Nombre          Tamaño    VMA               LMA               Desp fich Alin
  0 .text           00000014  0000000000000000  0000000000000000  000001d0  2**4
                    CONTENTS, ALLOC, LOAD, CODE
  1 __LD.__compact_unwind 00000020  0000000000000018  0000000000000018  000001e8  2**3
                    CONTENTS, RELOC, DEBUGGING
  2 .eh_frame       00000040  0000000000000038  0000000000000038  00000208  2**3
                    CONTENTS, ALLOC, LOAD, READONLY, DATA
SYMBOL TABLE:
0000000000000000 g       0f SECT   01 0000 [.text] _f

RELOCATION RECORDS FOR [__LD.__compact_unwind]:
OFFSET           TYPE              VALUE
0000000000000000 64                .text
```

El código de la función `_f`:

```
mylib.o:     formato del fichero mach-o-x86-64
Desensamblado de la sección .text:

0000000000000000 <_f>:
   0: 55                    push   %rbp
   1: 48 89 e5              mov    %rsp,%rbp
   4: 89 7d fc              mov    %edi,-0x4(%rbp)
   7: 8b 7d fc              mov    -0x4(%rbp),%edi
   a: 81 c7 01 00 00 00     add    $0x1,%edi
  10: 89 f8                 mov    %edi,%eax
  12: 5d                    pop    %rbp
  13: c3                    retq
```

Podemos observar que en cada archivo objeto, el código de ambas funciones (`_main` y `_f`) comienzan en la dirección cero, por lo cual el linker cuando tenga que generar el programa final deberá *reubicar* alguna de las funciones.

Al generar el programa (mediante `gcc -o main main.o mylib.o`), lo cual solamente invoca al linker, podemos observar:

```
main
arquitectura: i386:x86-64, opciones 0x00000012:
EXEC_P, HAS_SYMS
dirección de inicio 0x0000000100000f50

Secciones:
Ind Nombre          Tamaño    VMA               LMA               Desp fich Alin
  0 .text           00000034  0000000100000f50  0000000100000f50  00000f50  2**4
                    CONTENTS, ALLOC, LOAD, CODE
  1 __TEXT.__unwind_info 00000048  0000000100000f84  0000000100000f84  00000f84  2**2
                    CONTENTS, ALLOC, LOAD, READONLY, CODE
  2 .eh_frame       00000030  0000000100000fd0  0000000100000fd0  00000fd0  2**3
                    CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .data           00000004  0000000100001000  0000000100001000  00001000  2**2
                    CONTENTS, ALLOC, LOAD, DATA
SYMBOL TABLE:
0000000100000000 g       0f SECT   01 0010 [.text] __mh_execute_header
0000000100000f70 g       0f SECT   01 0000 [.text] _f
0000000100001000 g       0f SECT   04 0000 [.data] _g
0000000100000f50 g       0f SECT   01 0000 [.text] _main
0000000000000000 g       01 UND    00 0100 dyld_stub_binder
```

El código del programa completo ahora es:

```
main:     formato del fichero mach-o-x86-64
Desensamblado de la sección .text:

0000000100000f50 <_main>:
   100000f50: 55                    push   %rbp
   100000f51: 48 89 e5              mov    %rsp,%rbp
   100000f54: 48 83 ec 10           sub    $0x10,%rsp
   100000f58: 8b 3d a2 00 00 00     mov    0xa2(%rip),%edi        # 100001000 <_g>
   100000f5e: e8 0d 00 00 00        callq  100000f70 <_f>
   100000f63: 31 ff                 xor    %edi,%edi
   100000f65: 89 45 fc              mov    %eax,-0x4(%rbp)
   100000f68: 89 f8                 mov    %edi,%eax
   100000f6a: 48 83 c4 10           add    $0x10,%rsp
```

```
   100000f6e: 5d                    pop    %rbp
   100000f6f: c3                    retq

0000000100000f70 <_f>:
   100000f70: 55                    push   %rbp
   100000f71: 48 89 e5              mov    %rsp,%rbp
   100000f74: 89 7d fc              mov    %edi,-0x4(%rbp)
   100000f77: 8b 7d fc              mov    -0x4(%rbp),%edi
   100000f7a: 81 c7 01 00 00 00     add    $0x1,%edi
   100000f80: 89 f8                 mov    %edi,%eax
   100000f82: 5d                    pop    %rbp
   100000f83: c3                    retq
   ...
```

El linker combinó los contenidos de los archivos objetos y reubicó el código de la función `_f`. También resolvió y reescribió el destino de la instrucción `callq`, haciendo que efectivamente sea un llamado a la dirección de la primera instrucción de `_f`.

Este enlazado se denomina *enlazado estático*, ya que todo se resuelve en tiempo de compilación/linking.

3. Bibliotecas compartidas y enlazado dinámico

Normalmente los programas utilizan funciones y datos de uso común. Es por eso que estas funcionalidades se encapsulan en archivos objeto.

Una *biblioteca*[4] es un archivo objeto o un conjunto de archivos objetos. En el caso de bibliotecas estáticas, éstas son contenedores de archivos objetos para ser enlazados estáticamente.

En el mundo UNIX las bibliotecas estáticas tienen extensión (o sufijo `.a`), por *archiver*. En otros SO como MS-Windows tienen extensión `.lib`.

Cada SO provee un conjunto de bibliotecas del sistema que proveen los mecanismos de interacción entre los programas de usuario y los servicios del SO y otras con funciones útiles comunes como funciones matemáticas, estructuras de datos y algoritmos, APIs de interfaces gráficas, etc.

[4]La traducción adecuada de *library* es *biblioteca* y no *librería* (ésta última es una tienda de venta de libros).

Por ejemplo, en los sistemas tipo UNIX, la biblioteca `libc` se conoce como la *biblioteca estándar*. Contiene las funciones que realizan las *llamadas al sistema* como operaciones de entrada/salida, manejo de procesos, gestión de memoria y sistemas de archivos, entre otras.

Ya hemos visto que el *linker* tiene que generar un ejecutable a partir de uno o más archivos objeto y/o bibliotecas combinándolos y resolviendo las referencias externas (direcciones de memoria de funciones y datos definidos en otros módulos).

Un *static linker* combina los archivos concatenando los contenidos de cada uno en el archivo resultante y recalcula y reescribe las direcciones de los datos y funciones referenciados.

Esto tiene como ventaja que es un proceso simple y que el archivo ejecutable resultante contiene todo lo necesario, es decir que está autocontenido. Esto simplifica su distribución ya que sólo es necesario copiar el programa al otro sistema.

Sin embargo, tiene sus desventajas:

1. Cada aplicación que usa la biblioteca estándar y otras de uso común, tendrían incluído todo el código de éstas. Es decir, tendríamos replicación de código en cada aplicación, llevando a un desperidicio en el uso de la memoria.

2. Los archivos ejecutables son grandes.

3. En cada actualización de una biblioteca sería necesario re-enlazar el programa.

Por eso, los sistemas operativos modernos proveen *dynamic linking*: Permitir que el enlace se realice en tiempo de carga o ejecución del programa. Esto permite que una biblioteca se pueda cargar sólo una vez en memoria, por lo que se conocen como *shared libraries*, *shared objects* o *dynamic linking libraries (DLLs)*.

Para lograr esto, el linker se divide en dos partes: Una parte estática, generada por el compilador con la información necesaria sobre las biblotecas requeridas y posiblemente agregando una o más secciones, por ejemplo

`.interp` en `gcc`. Esta sección incluye el nombre de un intérprete del programa[5].

En éste caso de programas compilados, como C/C++, el intérprete es el *dynamic linker* (`ld-linux.so` en sistemas GNU-Linux), el cual también es un *shared object*.

La llamada al sistema `execve(path,args)` carga el programa en memoria, reserva los espacios de memoria correspondientes al código, datos estáticos, heap y stack, carga los segmentos de código y datos globales y finalmente pasa el control al código del dynamic linker que realiza los siguientes pasos:

1. Carga el código y datos de las bibliotecas compartidas requeridas (si no se había hecho antes).

2. Mapea el código (y datos globales) de las bibliotecas compartidas en el mismo espacio de memoria del proceso.

La resolución de los símbolos externos puede hacerse en dos momentos:

1. Durante la carga del programa (*load time*). Esto puede tomar demasiado tiempo en los sistemas modernos ya que requeriría reescribir cada dirección a resolver.

2. En tiempo de ejecución del programa (*run time linking*). Es la técnica más usada en los sistemas modernos, conocida como *lazy linking*.

3.1. Código independiente de posición (PIC)

EL propósito de las bibliotecas compartidas es mantener una única copia en el sistema y que varias procesos los usen simultáneamente. Esto implica que el diseño de una biblioteca compartida no contenga estado interno, ya que funciona como una utilidad de servicios a múltiples clientes (procesos) concurrentemente.

Un SO multitarea carga programas en diferentes áreas físicas de memoria, sin embargo, estos programas fueron compilados y enlazados de la misma manera, es decir todos ven su espacio de direcciones comenzando por ejmplo,

[5]Esto permite cargar el texto de un programa interpretado y hacer que el SO invoque automáticamente su intérprete, como ocurre con programas `python`, por ejemplo.

desde cero hasta un tamaño máximo. Por eso es que los programas deben contener código (instrucciones) que pueda funcionar independientemente de su dirección de carga.

El uso de modos de direccionamiento relativo y los mecanismos de memoria virtual provisto por CPUs modernas, permiten lograr código independiente de su posición. Por ejemplo, una instrucción `call f` en CPUs modernas usa direccionamiento relativo (desplazamniento con respecto al contenido actual del *program counter o PC*), por lo que ésta instrucción es *PIC*.

En muchas arquitecturas, el acceso a los datos se hecen relativos al contenido de un registro que contiene una dirección base para lograr código reubicable.

Con las biblotecas compartidas, queda por resolver cómo se resuelven las llamadas a las funciones desde el proceso. En las arquitecturas modernas, con mecanismos de protección de mamoria[6], se recurren a técnicas de memoria virtual para *mapear* la biblioteca compartida en el propio espacio de memoria de cada proceso que la usa.

¿Cómo se resuelven las direcciones de las llamadas a funciones a la biblioteca compartida?

Veámoslo a partir de un ejemplo simple (`hello.c`):

```
#include <stdio.h>

#define msg "Hello world\n"

int main(void)
{
  printf(msg);
}
```

El compilador (`gcc -c hello.c`) genera el archivo objeto `hello.o`. Como éste tiene la función `main`, éste módulo será el *programa principal* y usa

[6]Pera evitar que un proceso quede confinado a su espacio de memoria y no invada el espacio de otros.

la función de la biblioteca estándar `printf`.

Si analizamos el archivo objeto podemos observar:

```
hello.o
arquitectura: i386:x86-64, opciones 0x00000011:
HAS_RELOC, HAS_SYMS
dirección de inicio 0x0000000000000000

Secciones:
Ind Nombre        Tamaño    VMA               LMA               Desp fich Alin
  0 .text         00000023  0000000000000000  0000000000000000  00000220  2**4
                  CONTENTS, ALLOC, LOAD, RELOC, CODE
  1 .cstring      0000000d  0000000000000023  0000000000000023  00000243  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 __LD.__compact_unwind 00000020  0000000000000030  0000000000000030  00000250  2**3
                  CONTENTS, RELOC, DEBUGGING
  3 .eh_frame     00000040  0000000000000050  0000000000000050  00000270  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
SYMBOL TABLE:
0000000000000000 g       0f SECT   01 0000 [.text] _main
0000000000000000 g       01 UND    00 0000 _printf

RELOCATION RECORDS FOR [.text]:
OFFSET           TYPE              VALUE
0000000000000012 BRANCH32          _printf
000000000000000b DISP32            .cstring-0x0000000000000023

RELOCATION RECORDS FOR [__LD.__compact_unwind]:
OFFSET           TYPE              VALUE
0000000000000000 64                .text

Desensamblado de la sección .text:

0000000000000000 <_main>:
   0: 55                    push   %rbp
   1: 48 89 e5              mov    %rsp,%rbp
   4: 48 83 ec 10           sub    $0x10,%rsp
   8: 48 8d 3d 14 00 00 00  lea    0x14(%rip),%rdi        # 23 <_main+0x23>
   f: b0 00                 mov    $0x0,%al
  11: e8 00 00 00 00        callq  16 <_main+0x16>
  16: 31 c9                 xor    %ecx,%ecx
  18: 89 45 fc              mov    %eax,-0x4(%rbp)
  1b: 89 c8                 mov    %ecx,%eax
  1d: 48 83 c4 10           add    $0x10,%rsp
```

```
  21: 5d                    pop    %rbp
  22: c3                    retq
```

Nuevamente, vemos que el compilador generó *registros de reubicación*, en éste caso para la dirección de destino del *call _printf* y del *literal string* (arreglo de bytes que contiene los caracteres `"Hello world"`). Notar que el literal ocupa 13 bytes.

También debemos notar que el compilador dejó la dirección de destino de la instrucción `callq` en cero ya que no está resuelta y deberá resolverse dinámicamente.

¿Cómo se resuelve? Veamos qué genera el linker (`gcc -o hello hello.o`, lo cual automáticamente enlaza con la biblioteca estándar):

```
hello
arquitectura: i386:x86-64, opciones 0x00000012:
EXEC_P, HAS_SYMS
dirección de inicio 0x0000000100000f40

Secciones:
Ind Nombre          Tamaño    VMA               LMA               Desp fich Alin
  0 .text           00000023  0000000100000f40  0000000100000f40  00000f40  2**4
                    CONTENTS, ALLOC, LOAD, CODE
  1 __TEXT.__stubs 00000006  0000000100000f64  0000000100000f64  00000f64  2**1
                    CONTENTS, ALLOC, LOAD, READONLY, CODE
  2 __TEXT.__stub_helper 0000001a  0000000100000f6c  0000000100000f6c  00000f6c  2**2
                    CONTENTS, ALLOC, LOAD, READONLY, CODE
  3 .cstring        0000000d  0000000100000f86  0000000100000f86  00000f86  2**0
                    CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 __TEXT.__unwind_info 00000048  0000000100000f94  0000000100000f94  00000f94  2**2
                    CONTENTS, ALLOC, LOAD, READONLY, CODE
  5 .eh_frame       00000018  0000000100000fe0  0000000100000fe0  00000fe0  2**3
                    CONTENTS, ALLOC, LOAD, READONLY, DATA
  6 __DATA.__nl_symbol_ptr 00000010  0000000100001000  0000000100001000  00001000  2**3
                    CONTENTS, ALLOC, LOAD, DATA
  7 __DATA.__la_symbol_ptr 00000008  0000000100001010  0000000100001010  00001010  2**3
                    CONTENTS, ALLOC, LOAD, DATA
SYMBOL TABLE:
0000000100000000 g       0f SECT   01 0010 [.text] __mh_execute_header
0000000100000f40 g       0f SECT   01 0000 [.text] _main
0000000000000000 g       01 UND    00 0100 _printf
0000000000000000 g       01 UND    00 0100 dyld_stub_binder
```

```
Desensamblado de la sección .text:

0000000100000f40 <_main>:
   100000f40: 55                    push   %rbp
   100000f41: 48 89 e5              mov    %rsp,%rbp
   100000f44: 48 83 ec 10           sub    $0x10,%rsp
   100000f48: 48 8d 3d 37 00 00 00  lea    0x37(%rip),%rdi        # 100000f86 <_main+0x46>
   100000f4f: b0 00                 mov    $0x0,%al
   100000f51: e8 0e 00 00 00        callq  100000f64 <_main+0x24>
   100000f56: 31 c9                 xor    %ecx,%ecx
   100000f58: 89 45 fc              mov    %eax,-0x4(%rbp)
   100000f5b: 89 c8                 mov    %ecx,%eax
   100000f5d: 48 83 c4 10           add    $0x10,%rsp
   100000f61: 5d                    pop    %rbp
   100000f62: c3                    retq

Desensamblado de la sección __TEXT.__stubs:

0000000100000f64 <__TEXT.__stubs>:
   100000f64: ff 25 a6 00 00 00     jmpq   *0xa6(%rip)        # 100001010 <_main+0xd0>
...
```

Debemos notar que ahora el linker reemplazó (reubicó) la dirección de la instrucción `callq` que en el archivo objeto estaba en cero, por la dirección de una función introducida por el compilador. Esta función es un *stub* encargado de resolver la dirección de la función, invocando al *linker dinámico*.

En sistemas GNU-Linux, éste mecanismo se codifica en dos tablas:

- **Global Offset Table (GOT)**: Contiene los desplazamientos (direcciones relativas) de los símbolos globales del *shared object*. Esta tabla está al comienzo del área de datos globales (`.data`).
- **Procedure Linkage Table (PLT)**: Tabla de direcciones de pequeñas rutinas (*stubs*) incluidos en el código por el compilador y/o linker. Cada *stub* se corresponde con una función externa invocada por el programa.

En cada invocación a una función externa (sea i), el linker estático generó el `call` a la dirección contenida en `PLT[i]` ($i > 0$).

1. Cada stub `PLT[i]` contiene instrucciones que salta a la función cuya dirección está contenida en `GOT[i]`, apilando el identificador del símbolo de función (generado por el linker estático).

2. `GOT[i]` inicialmente apunta a `PLT[0]`.

3. La rutina apuntada por `PLT[0]` invoca a `ld` pasándole como parámetro el identificador de la función (que fue apilada por `PLT[i]`).

   `Ld` resuelve la dirección y la almacena en `GOT[i]`.

De esta forma, en las próximas invocaciones, `PLT[i]`, saltará directamente a la función de la biblioteca en lugar de saltar a `PLT[0]`. El único costo adicional en las sucesivas invicaciones es una instrucción de salto incondicional usando direccionamiento indirecto.

La figura 3.1 muestra el funcionamiento del mecanismo *lazy binding*.

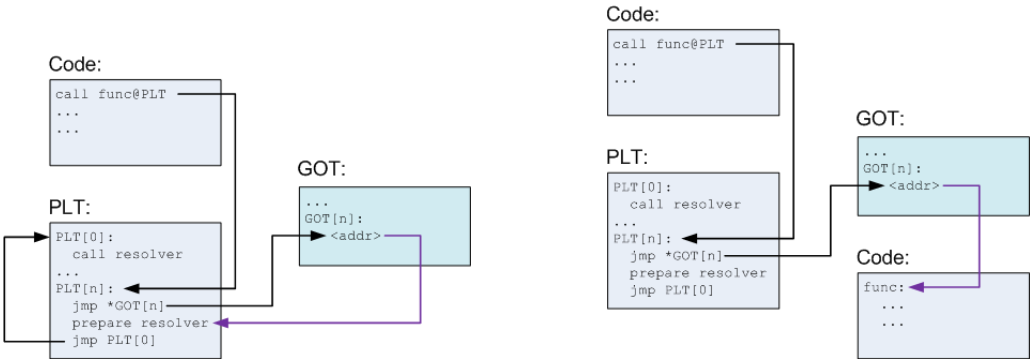

Figura 2: GOT-PLT: Primera invocación (izq.) y próximas (der.).

4. Carga y enlazado de bibliotecas compartidas desde programas

El *linker dinámico* puede usarse en programas de usuario directamente.

De esta forma una aplicación podría cargar y enlazar dinámicamente módulos (componentes o plugins) inclusive desarrollados por terceros con una API[7].

[7] Application Programming Interface.

Estas bibliotecas o plugins pueden cargarse y descargarse *en caliente*, es decir, sin siquiera reiniciar el proceso principal.

Los sistemas tipo UNIX, incluyendo GNU-Linux, ofrecen una interface muy simple al `dynamic linker` (`libld.so`).

```
#include <dlfcn.h>

void * dlopen(const char * filename, in flag);
void * dlsym(void * handle, char * symbol);
int dlclose(void * handle);
```

La función `dlopen()` carga un *shared object* y retorna un puntero a un handle. La función `dlsym()` permite resolver y obtener la dirección del símbolo (función o variable) dada. `dlclose()` descarga el *shared object* de la memoria si no hay otros programas o bibliotecas que lo estén usando.

A continuación se muestra el primer ejemplo, donde ahora el módulo `mylib.c` se compila como un *shared object* y se carga dinámicamente desde la función `main`:

1. Generar código *independiente de posición*: `gcc -c -fpic mylib.c`
2. Generar la biblioteca dinámica: `gcc -c -shared -o libmylib.so mylib.o`
   (En los sistemas tipo UNIX una biblioteca `x` se debería nombrar `libx.o`.)

El programa `main.c` ahora es:

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

int g = 1000;

int main() {
  void *handle;
  int (*f)(int); /* f: int --> int  */
  handle = dlopen("./libmylib.so", RTLD_LAZY);

  if (!handle) {
```

```
    printf("Error loading library...\n");
    return -1;
  }

  f = dlsym(handle, "f");

  if (!f) {
    printf("Error resolving symbol f...\n");
    dlclose(handle);
    return -2;
  }

  printf("Calling f(g): %d\n", f(g));

  dlclose(handle);
  return 0;
}
```

y lo debemos compilar enlazándolo con el *dynamic linker*: `gcc -o main main.c -ldl`. Notar que el enlazado sólo genera la información necesaria que vimos anteriormente para su enlazado dinámico.

Referencias

[1] John Levine. *Linkers and Loaders*. Elsevier Science. 1999. ISBN-13: 9781558604964. Draft chapters: *http://www.iecc.com/linker/*