Strumenti di sviluppo

G. Lettieri

25 Marzo 2019

Per ottenere un programma eseguibile a partire da dei file sorgenti è necessario passare attraverso diversi strumenti, di cui vogliamo ora capire lo scopo e il funzionamento interno.

Nel seguito faremo riferimento al piccolo esempio di programma misto contenuto nelle Figure 1, 2 e 3.

1 Il preprocessore

Il preprocessore si preoccupa di interpretare le direttive che si trovano nelle line che iniziano con `#`, espandere eventuali *macro* ed eliminare commenti e spazi superflui.

La direttiva più comune è **`#include`**, che può essere seguita da un nome di file tra doppi apici o tra parentesi angolate. Quando il preprocessore incontra questa direttiva la sostituisce con tutto il contenuto del file. Nel caso di doppi apici, il preprocessore cerca il file prima nella stessa directory in cui si trova il file corrente e (se non lo trova), in una serie di directory di sistema (per es., `/usr/include`). Nel caso di parentesi angolate, il preprocessore cerca solo nelle directory di sistema.

```
1  #include "lib.h"
2
3  long var1 = 8;
4  long var2 = 4;
5
6  int main()
7  {
8          // un commento
9          var1 =    foo(var2);
10         return var1;
11 }
```

Figura 1: file main.cpp

```
1 extern "C" long foo(long);
```

Figura 2: file lib.h

```
1  .data
2  foovar1:
3          .quad 5
4  foovar2:
5          .quad 6
6  .text
7  .global foo
8  foo:
9          pushq %rbp
10         movq %rsp, %rbp
11         movq %rdi, %rax
12         movq foovar1, %rax
13         addq foovar2(%rip), %rax
14         leave
15         ret
```

Figura 3: file foo.s

Le macro possono essere definite con la direttiva **#define**. Per esempio, **#define** `PI 3.14` definisce la macro `PI` e le assegna il testo `3.14`. Da questa linea in poi il preprocessore sostituirà ogni occorrenza della parola `PI` con `3.14`.

Mentre le macro sono essenziali in C, in C++ si tende ad evitarle in quanto il linguaggio dispone di costrutti migliori per molti dei loro utilizzi più tipici.

Con il comando `g++ -E main.cpp` possiamo osservare l'output del preprocessore (Figura 4). Si noti che la linea 1 di Figura 1 è stata sostiutita dal contenuto del file `lib.h` all linea 8 di Figura 4 (sono state aggiunte anche le linee 1–7 e 9, che servono al compilatore per emettere eventuali errori di sintassi in maniera più precisa). È sparito il commento che si trovava alla linea 8 di Figura 1 e gli spazi all'interno di ciascuna riga sono stati ridotti all'essenziale.

2 Il compilatore

Il compilatore C++ (o C) riceve un unico file in ingresso e produce un unico file assembler in uscita. È uno strumento molto sofisticato che richiederebbe da solo un intero corso. Tramite gli esercizi di traduzione da C++ ad assembler, in cui ci sostituiamo al compilatore, possiamo farci un idea di come deve operare.

La cosa che più ci interessa in questo momento è osservare che il compilatore vede esclusivamente il contenuto del file di Figura 4. L'inclusione del file `lib.h`

```
1  # 1 "main.cpp"
2  # 1 "<built-in>"
3  # 1 "<command-line>"
4  # 1 "/usr/include/stdc-predef.h" 1 3 4
5  # 1 "<command-line>" 2
6  # 1 "main.cpp"
7  # 1 "lib.h" 1
8  extern "C" long foo(long);
9  # 2 "main.cpp" 2
10
11 long var1 = 8;
12 long var2 = 4;
13
14 int main()
15 {
16
17  var1 = foo(var2);
18  return var1;
19 }
```

Figura 4: L'output di `g++ -E main.cpp`

serve a fare in modo che il compilatore veda la dichiarazione della funzione `foo`, che è l'unica cosa di cui ha bisogno per poter tradurre la linea 17.

Per semplicità traduciamo a mano il file `main.cpp` ottenendo il file in Figura 5. Per semplificare ulteriormente il contenuto dei vari file oggetto, abbiamo anche definito direttamente l'etichetta `_start` al posto di `main`.

3 L'assemblatore

Nel nostro esempio l'assemblatore entra in gioco sia per tradurre l'output prodotto dal compilatore, sia per tradurre il file `foo.s` che è scritto direttamente in assembler. Anche l'assemblatore lavora esclusivamente osservando un file alla volta.

Lo scopo dell'assemblatore è di generare il contenuto di *sezioni* della memoria del programma. Una sezione è denominata `.text` ed è destinata a contenere il codice del programma, e un'altra è denominata `.data` ed è destinata a contenere i dati globali. Ci possono essere altre sezioni, anche definite dall'utente.

L'assemblatore lavora senza sapere (in genere) a quale indirizzo le sezioni verrano poi caricate. Questo comporta che in molti casi la sua traduzione è incompleta e deve essere completata successivamente, dal collegatore o dal caricatore. L'assemblatore genera anche una serie di tabelle contenenti le informazioni necessarie al collegatore (o caricatore) per completare la traduzione:

```
1  .data
2  .global var1, var2
3  var1:
4          .quad   8
5  var2:
6          .quad   4
7  .text
8  .global _start
9  _start:
10         pushq   %rbp
11         movq    %rsp, %rbp
12         movq    var2(%rip), %rax
13         movq    %rax, %rdi
14         call    foo
15         movq    %rax, var1(%rip)
16         movq    var1(%rip), %rax
17         popq    %rbp
18         ret
```

Figura 5: file main.s

la *tabella delle sezioni*, la *tabella dei simboli* (locali e globali) e la *tabella di rilocazione*.

Per ogni sezione l'assemblatore mantiene un *contatore*, inizialmente pari a zero. Mentre legge il file sorgente, l'assemblatore fa riferimento sempre ad una sezione corrente, che è l'ultima che è stata nominata (o `.text`, se non è stata nominata nessuna). Ogni comando, come `.quad 5` oppure **`movq %rsp,%rbp`**, corrisponde ad una richiesta di aggiungere un certo numero di byte nella sezione corrente (avanzando di conseguenza il contatore). Il comando `.quad 5` aggiunge 8 byte contenenti la rappresentazione binaria di 5, mentre il comando **`movq %rsp,%rbp`** aggiunge i byte necessari a codificare questa istruzione in linguaggio macchina. Si noti che ogni comando può far parte di qualunque sezione: il risultato è comunque l'aggiunta dei corrispondenti byte alla sezione.

La definizione di una etichetta, come alle linee 2, 4 e 8 di Figura 3, corrisponde alla richiesta di aggiungere un nuovo simbolo alla tabella dei simboli. Per ogni simbolo l'assemblatore deve sapere il nome, la sezione a cui appartiene e il valore. La linea 2 aggiunge il simbolo di nome "`foovar1`" appartenente alla sezione `.data` (che è quella corrente). Il valore di un simbolo è il valore del contatore della sezione corrente nel momento in cui il simbolo viene introdotto: rappresenta, dunque, l'offset rispetto alla base della sezione del primo byte che verrà aggiunto dopo il simbolo. Nel caso di `foovar1` si tratta dunque dell'offset del quad 5 introdotto alla linea 3. L'effetto complessivo è quello di dare un nome ad una certa locazione di memoria (anche se l'indirizzo di questa locazione non è ancora noto).

L'assemblatore lavora concettualmente in due "passate" (letture del file sorgente): una prima passata per raccogliere tutte le definizioni dei simboli e una seconda per effettuare la traduzione vera e propria. Questo perché una istruzione (come un salto in avanti) può riferire un simbolo che è stato definito più avanti nel file.

Alla linea 12 vediamo un caso in cui l'assemblatore non può completare la traduzione: l'istruzione ha bisogno dell'indirizzo completo di `foovar1`, ma l'assemblatore non lo conosce (perché non sa dove verrà caricata la sezione `.data`). In questo caso l'assemblatore usa temporanemente l'indirizzo zero, ma aggiunge una nuova entrata alla *tabella di rilocazione*. Questa tabella contiene le istruzioni che permetteranno al collegatore di inserire l'indirizzo `foovar` una volta noto.

3.1 Il formato ELF

Vediamo ora in concreto cosa l'assemblatore GNU produce su Linux dopo aver assemblato il file `foo.s`. Il risultato è un file oggetto in formato ELF (Executable and Linking Format), uno standard adottato da molti sistemi Unix e che è in grado di contenere sia file oggetto, sia eseguibili, sia librerie dinamiche (che non vedremo).

Per assemblare il file `foo.s` usiamo il comando

```
as -o foo.o foo.s
```

Ottenendo il file `foo.o`. I file in Unix e (Linux) sono sempre solo sequenze di byte il cui significato dipende da varie convenzioni. Possiamo esaminare il contenuto di qualunque file con il comando `hexdump` (Figura 6). Ogni riga mostra 16 byte del file, in esadecimale. La prima colonna contiene l'offset del primo byte della riga (in esadecimale), mentre le 16 colonne successive mostrano i byte in questione. L'ultima colonna (tra barre verticali) mostra il carattere ASCII che corrisponde ad ogni byte della riga, quando possibile. Il carattere "." indica che il byte corrispondente non contiene un codice ASCII stampabile.

Nel caso di file ELF abbiamo altri strumenti per ispezionarne il contenuto. Quelli più comunamente disponibili sono `readelf` e `objdump`. Il file ELF inizia con una intestazione standard, che possiamo leggere con l'opzione `-h` di `readelf` (Figura 7). Tra le varie informazioni, ci interessa lo "start of section headers". Questo dice che la tabella delle sezioni si trova nel file a un offeset di 400 byte. Possiamo chiedere a `readelf` di mostrarci il contenuto della tabella delle sezioni (Figura 8). Per ogni sezione abbiamo il nome, il tipo, l'indirizzo a cui deve essere caricata in memoria (Address), l'offset a cui la sezione inizia all'interno del file (Off) e la sua dimensione (Size). Delle collonne successive notiamo i flag (Flg) che fornisce ulteriori informazioni su come caricare la sezione. Notiamo subito che tutti i campi Address sono zero, perché l'assemblatore non sa ancora a che indirizzo le sezioni debbano essere caricate. Inoltre, solo le sezioni con il flag A vanno effettivamente caricate, mentre le altre servono ad altri scopi.

```
00000000  7f 45 4c 46 02 01 01 00  00 00 00 00 00 00 00 00  |.ELF............|
00000010  01 00 3e 00 01 00 00 00  00 00 00 00 00 00 00 00  |..>.............|
00000020  00 00 00 00 00 00 00 00  90 01 00 00 00 00 00 00  |................|
00000030  00 00 00 00 40 00 00 00  00 00 40 00 08 00 07 00  |....@.....@.....|
00000040  55 48 89 e5 48 89 f8 48  8b 04 25 00 00 00 00 48  |UH..H..H..%....H|
00000050  03 05 00 00 00 00 c9 c3  05 00 00 00 00 00 00 00  |................|
00000060  06 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000070  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000080  00 00 00 00 03 00 01 00  00 00 00 00 00 00 00 00  |................|
00000090  00 00 00 00 00 00 00 00  00 00 00 00 03 00 03 00  |................|
000000a0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
000000b0  00 00 00 00 03 00 04 00  00 00 00 00 00 00 00 00  |................|
000000c0  00 00 00 00 00 00 00 00  01 00 00 00 00 00 03 00  |................|
000000d0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
000000e0  09 00 00 00 00 00 00 03  00 08 00 00 00 00 00 00  |................|
000000f0  00 00 00 00 00 00 00 00  11 00 00 00 10 00 01 00  |................|
00000100  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000110  00 66 6f 6f 76 61 72 31  00 66 6f 6f 76 61 72 32  |.foovar1.foovar2|
00000120  00 66 6f 6f 00 00 00 00  0b 00 00 00 00 00 00 00  |.foo............|
00000130  0b 00 00 00 02 00 00 00  00 00 00 00 00 00 00 00  |................|
00000140  12 00 00 00 00 00 00 00  02 00 00 00 02 00 00 00  |................|
00000150  04 00 00 00 00 00 00 00  00 2e 73 79 6d 74 61 62  |..........symtab|
00000160  00 2e 73 74 72 74 61 62  00 2e 73 68 73 74 72 74  |..strtab..shstrt|
00000170  61 62 00 2e 72 65 6c 61  2e 74 65 78 74 00 2e 64  |ab..rela.text..d|
00000180  61 74 61 00 2e 62 73 73  00 00 00 00 00 00 00 00  |ata..bss........|
00000190  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
000001d0  20 00 00 00 01 00 00 00  06 00 00 00 00 00 00 00  | ...............|
000001e0  00 00 00 00 00 00 00 00  40 00 00 00 00 00 00 00  |........@.......|
000001f0  18 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000200  01 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000210  1b 00 00 00 04 00 00 00  40 00 00 00 00 00 00 00  |........@.......|
00000220  00 00 00 00 00 00 00 00  28 01 00 00 00 00 00 00  |........(.......|
00000230  30 00 00 00 00 00 00 00  05 00 00 00 01 00 00 00  |0...............|
00000240  08 00 00 00 00 00 00 00  18 00 00 00 00 00 00 00  |................|
00000250  26 00 00 00 01 00 00 00  03 00 00 00 00 00 00 00  |&...............|
00000260  00 00 00 00 00 00 00 00  58 00 00 00 00 00 00 00  |........X.......|
00000270  10 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000280  01 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000290  2c 00 00 00 08 00 00 00  03 00 00 00 00 00 00 00  |,...............|
000002a0  00 00 00 00 00 00 00 00  68 00 00 00 00 00 00 00  |........h.......|
000002b0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
000002c0  01 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
000002d0  01 00 00 00 02 00 00 00  00 00 00 00 00 00 00 00  |................|
000002e0  00 00 00 00 00 00 00 00  68 00 00 00 00 00 00 00  |........h.......|
000002f0  a8 00 00 00 00 00 00 00  06 00 00 00 06 00 00 00  |................|
00000300  08 00 00 00 00 00 00 00  18 00 00 00 00 00 00 00  |................|
00000310  09 00 00 00 03 00 00 00  00 00 00 00 00 00 00 00  |................|
00000320  00 00 00 00 00 00 00 00  10 01 00 00 00 00 00 00  |................|
00000330  15 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000340  01 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000350  11 00 00 00 03 00 00 00  00 00 00 00 00 00 00 00  |................|
00000360  00 00 00 00 00 00 00 00  58 01 00 00 00 00 00 00  |........X.......|
00000370  31 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |1...............|
00000380  01 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000390
```

Figura 6: L'output di `hexdump -C foo.o`.

```
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              REL (Relocatable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x0
  Start of program headers:          0 (bytes into file)
  Start of section headers:          400 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           0 (bytes)
  Number of program headers:         0
  Size of section headers:           64 (bytes)
  Number of section headers:         8
  Section header string table index: 7
```

Figura 7: L'output di `readelf -h foo.o` (intestazione).

```
There are 8 section headers, starting at offset 0x190:

Section Headers:
  [Nr] Name              Type            Address          Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            0000000000000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        0000000000000000 000040 000018 00  AX  0   0  1
  [ 2] .rela.text        RELA            0000000000000000 000128 000030 18   I  5   1  8
  [ 3] .data             PROGBITS        0000000000000000 000058 000010 00  WA  0   0  1
  [ 4] .bss              NOBITS          0000000000000000 000068 000000 00  WA  0   0  1
  [ 5] .symtab           SYMTAB          0000000000000000 000068 0000a8 18      6   6  8
  [ 6] .strtab           STRTAB          0000000000000000 000110 000015 00      0   0  1
  [ 7] .shstrtab         STRTAB          0000000000000000 000158 000031 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  l (large), p (processor specific)
```

Figura 8: L'output di `readelf -WS foo.o` (tabella delle sezioni).

```
foo.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <foo>:
   0: 55                    push   %rbp
   1: 48 89 e5              mov    %rsp,%rbp
   4: 48 89 f8              mov    %rdi,%rax
   7: 48 8b 04 25 00 00 00  mov    0x0,%rax
   e: 00
   f: 48 03 05 00 00 00 00  add    0x0(%rip),%rax        # 16 <foo+0x16>
  16: c9                    leaveq
  17: c3                    retq
```

Figura 9: L'output di `objdump -d foo.o` (disassemblato).

```
Symbol table '.symtab' contains 7 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 SECTION LOCAL  DEFAULT    1
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    3
     3: 0000000000000000     0 SECTION LOCAL  DEFAULT    4
     4: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT    3 foovar1
     5: 0000000000000008     0 NOTYPE  LOCAL  DEFAULT    3 foovar2
     6: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT    1 foo
```

Figura 10: L'output di `readelf -s foo.o` (tabella dei simboli).

```
Symbol table '.symtab' contains 8 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 SECTION LOCAL  DEFAULT    1
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    3
     3: 0000000000000000     0 SECTION LOCAL  DEFAULT    4
     4: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT    3 var1
     5: 0000000000000008     0 NOTYPE  GLOBAL DEFAULT    3 var2
     6: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT    1 _start
     7: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND foo
```

Figura 11: L'output di `readelf -s main.o` (tabella dei simboli).

Osserviamo per esempio la sezione `.text`. Il tipo PROGBITS dice che il contenuto della sezione è deciso dal programma e non dallo standard ELF. La sezione inizia all'offset 0x40 nel file. In Figura 6 vediamo che a questo offset troviamo i byte 0x55, 0x48, 0x89, ... Si tratta della traduzione in linguaggio macchina delle istruzioni che abbiamo scritto nella sezione `.text` in Figura 3. Possiamo verificarlo usando il comando `objdump`, che ci permette di *disassemblare* il contenuto della sezione `.text` di un file ELF (Figura 9). L'output è su più colonne: la prima contiene l'offset all'interno della sezione, seguito da due punti; seguono i byte che si trovano a partire da quell'offset e infine l'interpretazione di quei byte come istruzione assembler (si noti che l'istruzione è ottenuta senza guardare il sorgente). La sezione va caricata (flag A) e deve essere eseguibile (flag X).

La sezione 5 è la tabella dei simboli, che possiamo osservare in Figura 10. È una delle sezioni che non va caricata (niente flag A) e serve solo al procedimento di costruzione dell'eseguibile. Le colonne importanti sono il valore (Value), lo scopo (Bind), la sezione a cui il simbolo appartiene (Ndx) e il suo nome (Name). I simboli il cui tipo è SECTION servono in realtà a contenere l'indirizzo, ancora ignoto, dell'inizio delle sezioni `.text` (Ndx=1), `.data` (Ndx=3) e `.bss` (Ndx=4). Per gli altri simboli il campo Value contiene l'offset all'interno della sezione. Si noti, per esempio, il Value=8 per il simbolo `foovar2`. Tutti i simboli sono marcati come LOCAL (campo Bind): vuol dire che il linker non li userà per risolvere i riferimenti indefiniti. Il simbolo `foo` è marcato come GLOBAL, perché lo abbiamo dichiarato tale (linea 7 di Figura 3).

In Figura 11 osserviamo anche la tabella dei simboli di `main.o`, per notare

```
Relocation section '.rela.text' at offset 0x128 contains 2 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
00000000000b  00020000000b R_X86_64_32S      0000000000000000 .data + 0
000000000012  000200000002 R_X86_64_PC32     0000000000000000 .data + 4
```

Figura 12: L'output di `readelf -r foo.o` (tabella di rilocazione).

un altro caso che si può presentare: il simbolo numero 7 (`foo`) risulta non definito (valore UND nella colonna Ndx). Questo avviene perché il file di Figura 5 riferisce il simbolo `foo` (riga 14) ma non lo definisce. La dichiarazione alla riga 8, infatti, è solo un prototipo che dichiara l'esistenza di `foo` e il suo tipo, ma non ne definisce il codice.

Osserviamo ora l'istruzione all'offset 7 in Figura 9. Questa traduce l'istruzione alla riga 12 di Figura 3, che contiene il riferimento a `foovar1` di cui, come abbiamo detto, l'assemblatore non conosce l'indirizzo. Possiamo osservare che l'assemblatore ha usato zero al posto dell'indirizzo di `foovar1`. In Figura 12 vediamo invece la tabella delle rilocazioni che l'assemblatore ha generato. Nel formato ELF c'è in realta una tabella di rilocazione diversa per ogni sezione che ne ha bisogno. La tabella stessa si trova in una sezione del file e la possiamo vedere in Figura 8, riga `.rela.text`. La collonna Inf ci dice, in questo caso, che la tabella è relativa alla sezione 1 (cioé la sezione `.text`).

Ogni entrata della tabella di rilocazione fornisce le indicazioni per il collegatore (nel suo ruolo di *link editor*) su come e dove scrivere gli indirizzi. Il collegatore è in grado di eseguire semplici calcoli che, in genere, comportano di sommare il valore di un simbolo con una costante. L'operazione da svolgere è codificata nel campo Type, e i possibili tipi sono specificati dallo standard ELF. La prima riga in Figura 12 dice all'offset 0xb della sezione `.text` è necessario scrivere il valore di `.data + 0`. Questo non è altro che l'indirizzo `foovar1`, e l'offset 0xb corrisponde ai 4 byte nulli all'interno dell'istruzione che si trova all'offset 7 in Figura 9. La seconda riga chiede di fare una operazione leggermente diversa, in quanto è relativa all'istruzione alla riga 13 di Figura 3 che usa un indirizzamento relativo a **`%rip`**. Il Type è infatti diverso e chiede di eseguire una differenza tra `.data + 4` e il campo Offset. Si noti che l'assemblatore ha deciso di usare il simbolo `.data` invece dei simboli `foovar1` e `foovar2`, aggiustando opportunamente la costante da sommare[1].

Per completezza, le Figure 13 e 14 mostrano la tabella delle sezioni e la tabelle di rilocazione del file `main.o`

4 Il collegatore

Il collegatore ha tre compiti:

- decidere dove caricare ogni sezione;

[1] Questo è accaduto perché non abbiamo dichiarato **`.global`** i simboli `foovar1` e `foovar2`.

```
There are 8 section headers, starting at offset 0x1e8:

Section Headers:
  [Nr] Name              Type             Address           Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL             0000000000000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS         0000000000000000 000040 000023 00  AX  0   0  1
  [ 2] .rela.text        RELA             0000000000000000 000150 000060 18   I  5   1  8
  [ 3] .data             PROGBITS         0000000000000000 000063 000010 00  WA  0   0  1
  [ 4] .bss              NOBITS           0000000000000000 000073 000000 00  WA  0   0  1
  [ 5] .symtab           SYMTAB           0000000000000000 000078 0000c0 18      6   4  8
  [ 6] .strtab           STRTAB           0000000000000000 000138 000016 00      0   0  1
  [ 7] .shstrtab         STRTAB           0000000000000000 0001b0 000031 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  l (large), p (processor specific)
```

Figura 13: L'output di `readelf -WS main.o` (tabella delle sezioni).

```
Relocation section '.rela.text' at offset 0x150 contains 4 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000000007  000500000002 R_X86_64_PC32     0000000000000008 var2 - 4
00000000000f  000700000002 R_X86_64_PC32     0000000000000000 foo - 4
000000000016  000400000002 R_X86_64_PC32     0000000000000000 var1 - 4
00000000001d  000400000002 R_X86_64_PC32     0000000000000000 var1 - 4
```

Figura 14: L'output di `readelf -r main.o` (tabella di rilocazione).

- risolvere tutti i riferimenti ai simboli non definiti;
- eseguire tutte le rilocazioni.

Il collegatore è il primo che vede il programma nella sua interezza: riceve infatti la lista di tutti i file oggetto da collegare.

In una prima fase, il collegatore cerca di ottenere un'unica sezione `.text`, `.data` e un'unica tabella dei simboli mettendo insieme le rispettive sezioni di ogni file oggetto. Il risultato di questa fase è una nuova tabella delle sezioni e una nuova tabella dei simboli.

Per creare l'unica sezione `.text` il collegatore si limita a concatenare le sezioni `.text` dei file oggetto, nell'ordine in cui gli sono stati passati. Lo stesso vale per la sezione `.data`. Per creare l'unica tabella dei simboli esamina in ordine tutte le tabelle dei simboli dei file oggetto e le unisce. Durante questa operazione deve:

- aggiustare gli offset dei simboli dei file dal secondo in poi, sommandovi le dimensioni delle sezioni dei file precedenti;
- controllare che uno stesso simbolo GLOBAL non sia definito due volte (altrimenti termina con un errore);
- creare una lista di tutti i simboli non definiti.

```
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x4000b0
  Start of program headers:          64 (bytes into file)
  Start of section headers:          680 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         2
  Size of section headers:           64 (bytes)
  Number of section headers:         6
  Section header string table index: 5
```

Figura 15: L'output di `readelf -h prog` (intestazione).

Una volta esaminati tutti i file, deve controllare che ogni simbolo non definito in un certo file sia stato definito in qualche altro file, cioé che ciascuno dei simboli nella lista dei non definiti si trovi nella lista dei simboli complessivi, marcato come GLOBAL. Se questo non accade, il collegatore termina con un errore.

A questo punto il collegatore può assegnare un indirizzo ad ogni sezione e, di conseguenza, calcolare il valore definitivo di ogni simbolo.

Fatto ciò, può procedere a consultare le tabelle di rilocazione di tutti i file oggetto, mettendo in atto le istruzioni che vi trova.

Il contenuto di tutte le sezioni è ormai pronto e il collegatore deve solo creare la *tabella di caricamento*, che dirà al caricatore come e dove caricare le sezioni in memoria, ogni volta che il programma dovrà essere eseguito.

4.1 Il formato ELF per gli eseguibili

Torniamo al nosto esempio e supponiamo che `main.o` e `foo.o` siano stati collegati insieme per ottenere il file `prog`. Anche questo è un file ELF e ne possiamo esaminare l'intestazione (Figura 15). Notiamo che sono ora presenti dei *program headers* (dopo 64 byte nel file) e che il campo Entry Point Address ha un valore diverso da zero.

I program headers compongono la tabella di caricamento, che possiamo osservare in Figura 16. La tabella contiene una riga per ogni *segmento* dove, nel gergo ELF, un segmento corrisponde ad un insieme di sezioni. La prima riga dice che all'offset zero nel file si trova un segmento grande 0xeb (FileSiz) che deve essere caricato all'indirizzo 0x400000 (VirtAddr o PhysAddr, che sono sempre uguali) e reso leggibile (Flg R) ed eseguibile (Flg E). Si tratta, come vediamo più sotto, del segmento che contiene la sezione `.text`. La seconda riga dice che all'offset 0xeb nel file si trova un segmento grande 0x20 (FileSiz) che deve essere caricato all'indirizzo 0x6000eb (VirtAddr o PhysAddr) e reso leggibile (Flg

```
Elf file type is EXEC (Executable file)
Entry point 0x4000b0
There are 2 program headers, starting at offset 64

Program Headers:
  Type           Offset   VirtAddr           PhysAddr           FileSiz  MemSiz   Flg Align
  LOAD           0x000000 0x0000000000400000 0x0000000000400000 0x0000eb 0x0000eb R E 0x200000
  LOAD           0x0000eb 0x00000000006000eb 0x00000000006000eb 0x000020 0x000020 RW  0x200000

 Section to Segment mapping:
  Segment Sections...
   00     .text
   01     .data
```

Figura 16: L'output di `readelf -Wl prog` (tabella di caricamento).

```
There are 6 section headers, starting at offset 0x2a8:

Section Headers:
  [Nr] Name              Type            Address          Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            0000000000000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        00000000004000b0 0000b0 00003b 00  AX  0   0  1
  [ 2] .data             PROGBITS        00000000006000eb 0000eb 000020 00  WA  0   0  1
  [ 3] .symtab           SYMTAB          0000000000000000 000110 000138 18      4   6  8
  [ 4] .strtab           STRTAB          0000000000000000 000248 000033 00      0   0  1
  [ 5] .shstrtab         STRTAB          0000000000000000 00027b 000027 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  l (large), p (processor specific)
```

Figura 17: L'output di `readelf -WS prog` (tabella delle sezioni).

R) e scrivibile (Flg W). Si tratta del segmento che contiene contiene la sezione `.data`.

Anche se non servono più una volta ottenuto l'eseguibile, il collegatore ha lasciato nel file ELF anche la tabella delle sezioni e la tabella dei simboli complessive. Possiamo osservarle nelle Figure 17 e 18. Notiamo come la dimensione delle sezioni `.text` e `.data` sia la somma delle dimensini delle corrispondenti sezioni nei file `main.o` e `foo.o`. Notiamo anche come i simboli abbiano ora tutti il loro valore definitivo. Possiamo vedere che il campo Entry Point Address di Figura 15 è esattamente il valore del simbolo `_start`.

Si noti che il collegatore ha aggiunto dei simboli propri (`__bss_start`, `_edata` ed `_end`, che marcano l'inizio o la fine di varie parti del programma.

Infine, osseriamo il disassemblato del file `prog` (Figura 19), che contiene il risultato finale di tutte le operazioni di rilocazione eseguite dal collegatore. In particolare osserviamo come nella funzione `foo`, all'indirizzo 0x4000da, l'istruzione **`movq`** `foovar1,%`**`rax`** sia finalmente completa con l'indirizzo `foovar1`, che è 0x600fb, come possiamo confermare dalla tabella dei simboli in Figura 18.

```
Symbol table '.symtab' contains 13 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 00000000004000b0     0 SECTION LOCAL  DEFAULT    1
     2: 00000000006000eb     0 SECTION LOCAL  DEFAULT    2
     3: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS foo.o
     4: 00000000006000fb     0 NOTYPE  LOCAL  DEFAULT    2 foovar1
     5: 0000000000600103     0 NOTYPE  LOCAL  DEFAULT    2 foovar2
     6: 00000000006000eb     0 NOTYPE  GLOBAL DEFAULT    2 var1
     7: 00000000004000b0     0 NOTYPE  GLOBAL DEFAULT    1 _start
     8: 000000000060010b     0 NOTYPE  GLOBAL DEFAULT    2 __bss_start
     9: 00000000006000f3     0 NOTYPE  GLOBAL DEFAULT    2 var2
    10: 00000000004000d3     0 NOTYPE  GLOBAL DEFAULT    1 foo
    11: 000000000060010b     0 NOTYPE  GLOBAL DEFAULT    2 _edata
    12: 0000000000600110     0 NOTYPE  GLOBAL DEFAULT    2 _end
```

Figura 18: L'output di `readelf -s prog` (tabella dei simboli).

```
prog:     file format elf64-x86-64


Disassembly of section .text:

00000000004000b0 <_start>:
  4000b0: 55                    push   %rbp
  4000b1: 48 89 e5              mov    %rsp,%rbp
  4000b4: 48 8b 05 38 00 20 00  mov    0x200038(%rip),%rax        # 6000f3 <var2>
  4000bb: 48 89 c7              mov    %rax,%rdi
  4000be: e8 10 00 00 00        callq  4000d3 <foo>
  4000c3: 48 89 05 21 00 20 00  mov    %rax,0x200021(%rip)        # 6000eb <var1>
  4000ca: 48 8b 05 1a 00 20 00  mov    0x20001a(%rip),%rax        # 6000eb <var1>
  4000d1: 5d                    pop    %rbp
  4000d2: c3                    retq

00000000004000d3 <foo>:
  4000d3: 55                    push   %rbp
  4000d4: 48 89 e5              mov    %rsp,%rbp
  4000d7: 48 89 f8              mov    %rdi,%rax
  4000da: 48 8b 04 25 fb 00 60  mov    0x6000fb,%rax
  4000e1: 00
  4000e2: 48 03 05 1a 00 20 00  add    0x20001a(%rip),%rax        # 600103 <foovar2>
  4000e9: c9                    leaveq
  4000ea: c3                    retq
```

Figura 19: L'output di `objdump -d prog` (disassemblato).

5 Il caricatore

Il caricatore ha il compito di caricare in memoria i segmenti del programma, agli indirizzi specificati nella tabella di caricamento, e di inizializzare i registri del processore in modo che il programma possa partire. In particolare, il registro `%rip` verrà inizializzato con il valore dell'entry point del programma (Figura 15, campo Entry point address) e il registro `%rsp` con un valore prefissato.

In Linux il caricatore è parte del nucleo del sistema.