Introdução

A partição dos programas por unidades de tradução é uma prática necessária na produção, na manutenção e na reutilização de *software*. Alteração de código fonte numa unidade de tradução não implica a necessidade de recompilar a totalidade do projeto.

Unidade de tradução (ficheiro ou módulo)

Uma unidade de tradução é formada por uma sequência de definições externas – variáveis ou funções. Externas porque são globalmente acessíveis, podem ser usadas por funções definidas nesta ou noutras unidades de tradução.

As variáveis definidas dentro de funções são internas.

Uma parte de um programa, numa unidade de tradução, interage com outra parte do programa, noutra unidade de tradução, através de variáveis ou funções. As referências para variáveis ou para funções, são feitas através de **símbolos**.

Também se usa para unidade de tradução a designação de módulo.

Declaração

Declarar uma variável ou função consiste em anunciar a sua existência.

`extern int z;` – declara que a variável **`z`**, do tipo **`int`**, está definida em âmbito externo.

`int strlen(char *str);` – declara que a função **`strlen`** recebe como argumento um ponteiros para **`char`** e devolve um **`int`**.

A sua definição encontra-se noutro local, pode ser no mesmo ficheiro.

Definição

Definir uma variável ou função implica reservar espaço de memória.

No caso das variáveis a definição pode vir acompanhada da definição do valor inicial.

```
int z = 33;
```

No caso das funções implica explicitar as instruções que determinam o seu comportamento.

```
int strlen(char * str) {
      int len = 0;
      while (*str++)
            ++len;
      return len;
}
```

Uma definição também é declaração.

Âmbito das declarações

O âmbito dos identificadores pode ser bloco, função, unidade de tradução (ficheiro ou módulo) ou global (de todo o programa).

A utilização do mesmo símbolo numa declaração mais interior "esconde" uma declaração mais exterior desse mesmo símbolo.

A uma declaração está associada um *storage class*: automático ou estático.

O *storage class* depende dos especificadores **`auto`**, **`register`**, **`static`**, **`extern`** ou do local da

declaração.

```
int a = 2;
extern int b;
static int c = 2;
int main() {
     int a;
     auto int b;
     register int c;
     static int d;
     extern int e;
}
```

Uma declaração diz-se externa se estiver fora de uma função ou for precedida do especificador **`extern.`**

Numa declaração interna a palavra **`static`** significa que a variável vai ser alojada num local permanente.

Visibilidade

Para que um símbolo definido num módulo possa ser referenciado noutro módulo é necessário que seja classificado como globalmente visível.

Na **linguagem C**, por omissão, uma declaração externa produz um símbolo globalmente visível (símbolo global). Para restringir a visibilidade de um símbolo ao módulo onde é definido, usa-se o especificador **`static`**.

Na **linguagem *assembly* GNU**, por omissão, um símbolo é visível apenas no módulo onde é definido. Para o tornar globalmente visível é necessário explicitar através da diretiva **`.global`** como no seguinte exemplo:

```
.global    main
main:
```

Um símbolo global precisa ser conhecido no módulo onde é referenciado. O **as** assume que um símbolo referenciado e não declarado no presente módulo é global e está definido noutro módulo.

As declarações são escritas em ficheiros com extensão **h**, que por sua vez são intercalados pela diretiva **`#include`** nos ficheiros fonte, a fim de dar a conhecer as propriedades dos objetos.

Compilação separada

Tomemos como exemplo um programa constituído pelos módulos: **main.c** e **add.c.**

```
int a = 4, b = 5;

int res;

int add();

int main() {
      add();
      return 0;
}
```

main.c

```
extern int a, b;
static const int x = 44;
static int y = 33;
static int *pa = &a;
static int *pb;
int res;

int add() {
      pb = &b;
      res = *pa + *pb + x + y;
}
```

add.c

Numa única invocação, o **gcc** pode processar ambos os ficheiros fonte e produzir o executável.

```
$ gcc main.c add.c -o main
```

Efetivamente, **gcc** não é apenas o compilador é um *compiler driver*. No processo de geração do executável o **gcc** invoca sucessivamente, para cada ficheiro:

1. o pré-processador

   ```
   $ cpp <outros argumentos> -c main.c -o main.i
   ```

2. o compilador

   ```
   $ gcc <outros argumentos> -S main.i -o main.s
   ```

3. o *assembler*

   ```
   $ as <outros argumentos> -S main.s -o main.o
   ```

No final invoca o *linker*:

```
$ ld <outros argumentos> <objetos de sistema> main.o add.o -o main
```

O método designado por **compilação separada** consiste em invocar os tradutores (**cpp. cc, as**) individualmente para cada ficheiro fonte, produzindo o **ficheiro objeto relocalizável**.

```
$ gcc -c main.c
```

```
$ gcc -c add.c
```

No final invoca-se o *linker* (**ld**) para proceder à geração do **ficheiro objeto executável**. Por conveniência invoca-se novamente o *driver* **gcc** porque este adiciona os objetos de sistema e as bibliotecas normalizadas.

```
$ gcc main.o add.o -o main
```

```
$ gcc -v
```

```
...
```

```
gcc version 9.3.0 (Ubuntu 9.3.0-17ubuntu1~20.04)
```

Ficheiro objeto relocalizável

O ficheiro objeto relocalizável é produzido pelo *assembler* e tem extensão **o**. Pode ter tido origem num programa fonte em C ou em *assembly.*

Nos sistemas Linux atuais os ficheiros objeto relocalizáveis e os objetos executáveis, utilizam o formato ELF[1]. O seu conteúdo é essencialmente composto por um cabeçalho e um conjunto de blocos que se designam por secções que contêm o binário do programa e outras informações.

Secções

```
$ readelf -S add.o

Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align

  [ 1] .text             PROGBITS         0000000000000000  00000040
       0000000000000042  0000000000000000  AX       0     0     1
  [ 2] .rela.text        RELA             0000000000000000  00000338
       0000000000000090  0000000000000018   I      13     1     8
  [ 3] .data             PROGBITS         0000000000000000  00000084
       0000000000000004  0000000000000000  WA       0     0     4
  [ 4] .bss              NOBITS           0000000000000000  00000088
       0000000000000008  0000000000000000  WA       0     0     8
  [ 5] .rodata           PROGBITS         0000000000000000  00000088
       0000000000000004  0000000000000000   A       0     0     4
  [ 6] .data.rel         PROGBITS         0000000000000000  00000090
       0000000000000008  0000000000000000  WA       0     0     8
  [ 7] .rela.data.rel    RELA             0000000000000000  000003c8
       0000000000000018  0000000000000018   I      13     6     8

  [13] .symtab           SYMTAB           0000000000000000  00000150
       00000000000001c8  0000000000000018          14    15     8
  [14] .strtab           STRTAB           0000000000000000  00000318
       0000000000000019  0000000000000000           0     0     1
  [15] .shstrtab         STRTAB           0000000000000000  000003f8
       0000000000000083  0000000000000000           0     0     1
```

.text binário das instruções *assembly*;

.rela.text locais na secção **.text** que precisam ser atualizadas na ligação;

.data variáveis com valor inicial definido e determinável em compilação;

.data.rel variáveis com valor inicial definido determinado em ligação;

.rela.data.rel locais na secção **.data.rel** que precisam ser atualizadas na ligação;

.bss variáveis com valor inicial zero;

.rodata dados apenas de leitura – constantes, strings;

.symtab tabela de símbolos – nome das funções e variáveis, definidas e invocadas;

.strtab tabela de strings com o nome dos símbolos;

.shstrtab tabela de strings usadas nos cabeçalhos de secções.

1 ELF-64 Object File Format - http://ftp.openwatcom.org/devel/docs/elf-64-gen.pdf

Símbolos

```
$ readelf -s add.o

Symbol table '.symtab' contains 20 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS add.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1
     3: 0000000000000000     0 SECTION LOCAL  DEFAULT    3
     4: 0000000000000000     0 SECTION LOCAL  DEFAULT    4
     5: 0000000000000000     0 SECTION LOCAL  DEFAULT    5
     6: 0000000000000000     4 OBJECT  LOCAL  DEFAULT    5 x
     7: 0000000000000000     4 OBJECT  LOCAL  DEFAULT    3 y
     8: 0000000000000000     0 SECTION LOCAL  DEFAULT    6
     9: 0000000000000000     8 OBJECT  LOCAL  DEFAULT    6 pa
    10: 0000000000000000     8 OBJECT  LOCAL  DEFAULT    4 pb
    11: 0000000000000000     0 SECTION LOCAL  DEFAULT    9
    12: 0000000000000000     0 SECTION LOCAL  DEFAULT   10
    13: 0000000000000000     0 SECTION LOCAL  DEFAULT   11
    14: 0000000000000000     0 SECTION LOCAL  DEFAULT    8
    15: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND a
    16: 0000000000000004     4 OBJECT  GLOBAL DEFAULT  COM res
    17: 0000000000000000    66 FUNC    GLOBAL DEFAULT    1 add
    18: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND b
```

Detalhe dos símbolos: posição relativa ao início da secção a que pertencem (**`Value`**), dimensão (**`Size`**); tipo função (**`FUNC`**) ou variável (**`OBJECT`**); âmbito (**`Bind`**) local ou global, secção a que pertence (**`Ndx`**), e nome do símbolo (**`Name`**).

Outra forma de visualizar os símbolos de um módulo:

```
$ nm add.o
                 U a
0000000000000000 T add
                 U b
0000000000000000 d pa
0000000000000000 b pb
0000000000000004 C res
0000000000000000 r x
0000000000000000 d y
```

Símbolos locais definidos e invocados apenas por este módulo – **`pa, pb, x, y`**.

Símbolos globais definidos em **`add.c`** e eventualmente referidos por outros módulos – **`add`**.

Símbolos globais não definidos em **`add.c`** mas referidos – **`a`**, **`b`**.

Símbolos globais definidos ou declarados (símbolos fracos) em **`add.c`** – **`res`**.

Os símbolos são representados no ficheiro objeto pela estrutura **`Elf64_Sym`**.

```
typedef struct {
  Elf64_Word     st_name;     índice na tabela de strings com o nome dos símbolos
  unsigned char  st_info;     bits 0 a 3 - âmbito local ou global; bit 4 a 7 - tipo OBJECT, FUNC
  unsigned char  st_other;    reservado
  Elf64_Half     st_shndx;    secção onde é definido - há três casos especiais ABS, COM , UND
  Elf64_Addr     st_value;    valor associado ao símbolo (endereço de memória ou offset)
  Elf64_Xword    st_size;     dimensão de memória que ocupa
} Elf64_Sym;
```

Visualização em hexadecimal da tabela de símbolos. Os dados assinalados correspondem ao

símbolo **add**.

```
$ readelf -x .symtab  add.o

Hex dump of section '.symtab':
  0x00000000 00000000 00000000 00000000 00000000 ................
  0x00000010 00000000 00000000 01000000 0400f1ff ................
  0x00000020 00000000 00000000 00000000 00000000 ................
  0x00000030 00000000 03000100 00000000 00000000 ................
  0x00000040 00000000 00000000 00000000 03000300 ................
  0x00000050 00000000 00000000 00000000 00000000 ................
  0x00000060 00000000 03000400 00000000 00000000 ................
  0x00000070 00000000 00000000 00000000 03000500 ................
  0x00000080 00000000 00000000 00000000 00000000 ................
  0x00000090 07000000 01000500 00000000 00000000 ................
  0x000000a0 04000000 00000000 09000000 01000300 ................
  0x000000b0 00000000 00000000 04000000 00000000 ................
  0x000000c0 00000000 03000600 00000000 00000000 ................
  0x000000d0 00000000 00000000 0b000000 01000600 ................
  0x000000e0 00000000 00000000 08000000 00000000 ................
  0x000000f0 0e000000 01000400 00000000 00000000 ................
  0x00000100 08000000 00000000 00000000 03000900 ................
  0x00000110 00000000 00000000 00000000 00000000 ................
  0x00000120 00000000 03000a00 00000000 00000000 ................
  0x00000130 00000000 00000000 00000000 03000b00 ................
  0x00000140 00000000 00000000 00000000 00000000 ................
  0x00000150 00000000 03000800 00000000 00000000 ................
  0x00000160 00000000 00000000 0c000000 10000000 ................
  0x00000170 00000000 00000000 00000000 00000000 ................
  0x00000180 11000000 1100f2ff 04000000 00000000 ................
  0x00000190 04000000 00000000 15000000 12000100 ................
  0x000001a0 00000000 00000000 42000000 00000000 ........B.......
  0x000001b0 0f000000 10000000 00000000 00000000 ................
  0x000001c0 00000000 00000000                   ........

$ readelf -x .strtab add.o

Hex dump of section '.strtab':
  0x00000000 00616464 2e630078 00790070 61007062 .add.c.x.y.pa.pb
  0x00000010 00726573 00616464 00                 .res.add.
```

Ligação (*linking*)

A operação de ligação é realizada em duas fase:

1. **resolução de símbolos** – a cada referência corresponde apenas uma definição;
2. **relocalização** – calcula endereços dos símbolos e completa o código binário.

Resolução de símbolos

Consiste em encontrar a única definição para cada símbolo.

Nos símbolos locais isso é realizado pelo compilador – símbolos com ligação interna.

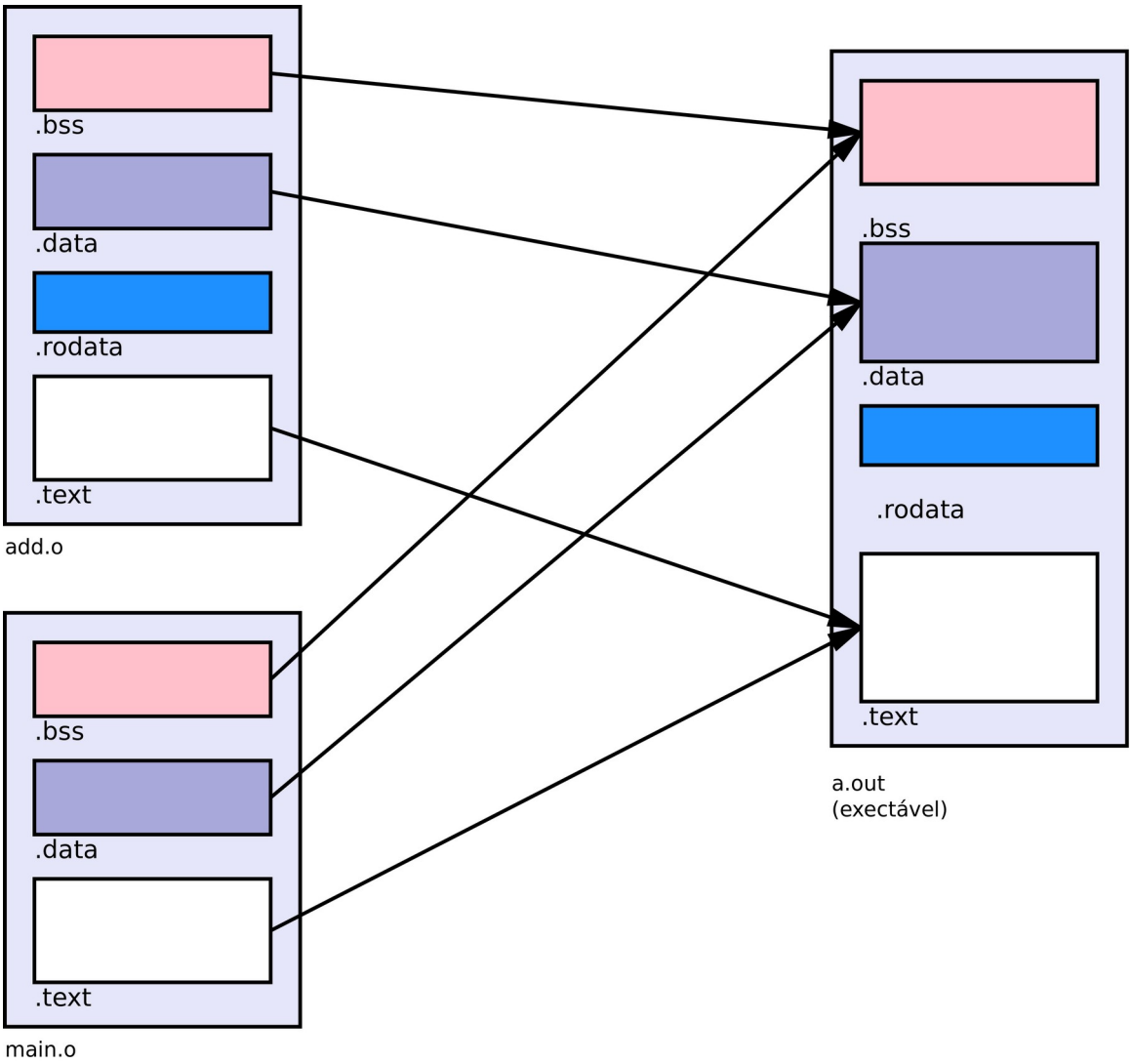
Nas resolução de símbolos de ligação externa pode acontecer a definição do símbolo não existir ou existirem múltiplas definições. Neste segundo caso a resolução poderá ter êxito se:

- existir apenas uma definição forte devendo as outras ser fracas;
- haver só definições fracas – neste caso escolher uma qualquer.

Relocalização

A relocalização começa depois de se conhecerem todas as definições de símbolos e engloba as seguintes operações:

1. agrega todas as secções do mesmo tipo definidas nos módulos numa única secção de saída;
2. calcula e associa a cada símbolo o endereço de execução;
3. modifica o conteúdo das secções nos locais indicados na respetiva tabela de relocalização.



Quando o *assembler* codifica uma instrução com referência a um símbolo, não sabe a localização final dessa instrução, nem a localização do símbolo referido. Nessa altura gera uma ***relocation entrie***.

```
$ objdump -d add.o

0000000000000000 <add>:
   0: f3 0f 1e fa             endbr64
   4: 55                      push   %rbp
   5: 48 89 e5                mov    %rsp,%rbp
   8: 48 8d 05 00 00 00 00    lea    0x0(%rip),%rax        # f <add+0xf>
   f: 48 89 05 00 00 00 00    mov    %rax,0x0(%rip)        # 16 <add+0x16>
  16: 48 8b 05 00 00 00 00    mov    0x0(%rip),%rax        # 1d <add+0x1d>
  1d: 8b 10                   mov    (%rax),%edx
  1f: 48 8b 05 00 00 00 00    mov    0x0(%rip),%rax        # 26 <add+0x26>
  26: 8b 00                   mov    (%rax),%eax
  28: 01 d0                   add    %edx,%eax
  2a: ba 2c 00 00 00          mov    $0x2c,%edx
  2f: 01 c2                   add    %eax,%edx
  31: 8b 05 00 00 00 00       mov    0x0(%rip),%eax        # 37 <add+0x37>
  37: 01 d0                   add    %edx,%eax
  39: 89 05 00 00 00 00       mov    %eax,0x0(%rip)        # 3f <add+0x3f>
  3f: 90                      nop
  40: 5d                      pop    %rbp
  41: c3                      ret
```

```
$ objdump -r add.o

RELOCATION RECORDS FOR [.text]:
OFFSET           TYPE              VALUE
000000000000000b R_X86_64_PC32     b-0x0000000000000004
0000000000000012 R_X86_64_PC32     .bss-0x0000000000000004
0000000000000019 R_X86_64_PC32     .data.rel-0x0000000000000004
0000000000000022 R_X86_64_PC32     .bss-0x0000000000000004
0000000000000033 R_X86_64_PC32     .data-0x0000000000000004
000000000000003b R_X86_64_PC32     res-0x0000000000000004


RELOCATION RECORDS FOR [.data.rel]:
OFFSET           TYPE              VALUE
0000000000000000 R_X86_64_64       a
```

Cada entrada na tabela de relocalização tem a seguinte informação:

```
typedef struct {
      long  offset;
      long  type:32,
            symbol:32;
      long  addend;
} Elf64_Rela;
```

`offset` – *offset* da referência em relação ao início da secção (local onde é preciso modificar);
`symbol` – identificação do símbolo referido;
`type` – tipo de relocalização (absoluta – `R_X86_64_64` ou relativa ao PC – `R_X86_64_PC32`);
`addend` – constante a ser incorporada na expressão.

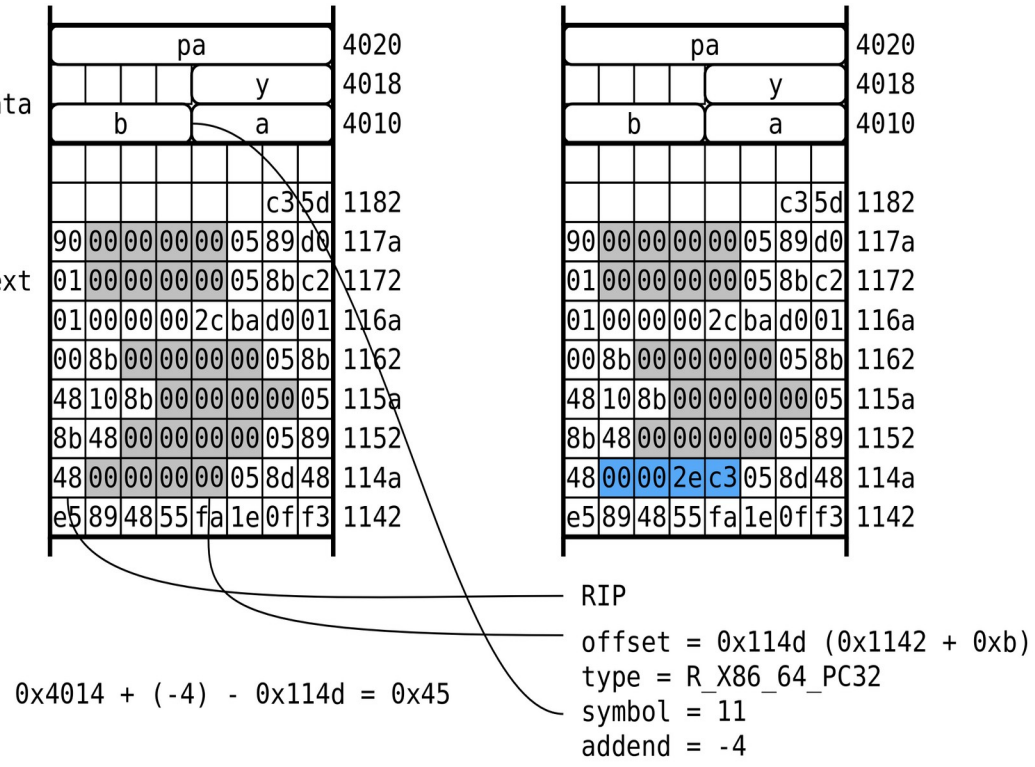
```
$ readelf -x .rela.text  add.o

Hex dump of section '.rela.text':
  0x00000000 0b000000 00000000 02000000 13000000 ................
  0x00000010 fcffffff ffffffff 12000000 00000000 ................
  0x00000020 02000000 04000000 fcffffff ffffffff ................
  0x00000030 19000000 00000000 02000000 08000000 ................
  0x00000040 fcffffff ffffffff 22000000 00000000 ........".......
  0x00000050 02000000 04000000 fcffffff ffffffff ................
  0x00000060 33000000 00000000 02000000 03000000 3...............
  0x00000070 fcffffff ffffffff 3b000000 00000000 ........;.......
  0x00000080 02000000 11000000 fcffffff ffffffff ................


$ objdump -d -r add.o

0000000000000000 <add>:
   0: f3 0f 1e fa              endbr64
   4: 55                       push   %rbp
   5: 48 89 e5                 mov    %rsp,%rbp
   8: 48 8d 05 00 00 00 00     lea    0x0(%rip),%rax        # f <add+0xf>
                  b: R_X86_64_PC32  b-0x4
   f: 48 89 05 00 00 00 00     mov    %rax,0x0(%rip)        # 16 <add+0x16>
                  12: R_X86_64_PC32 .bss-0x4
  16: 48 8b 05 00 00 00 00     mov    0x0(%rip),%rax        # 1d <add+0x1d>
                  19: R_X86_64_PC32 .data.rel-0x4
  1d: 8b 10                   mov    (%rax),%edx
  1f: 48 8b 05 00 00 00 00     mov    0x0(%rip),%rax        # 26 <add+0x26>
                  22: R_X86_64_PC32 .bss-0x4
  26: 8b 00                   mov    (%rax),%eax
  28: 01 d0                   add    %edx,%eax
  2a: ba 2c 00 00 00          mov    $0x2c,%edx
```

```
  2f: 01 c2                   add    %eax,%edx
  31: 8b 05 00 00 00 00       mov    0x0(%rip),%eax        # 37 <add+0x37>
                  33: R_X86_64_PC32 .data-0x4
  37: 01 d0                   add    %edx,%eax
  39: 89 05 00 00 00 00       mov    %eax,0x0(%rip)        # 3f <add+0x3f>
                  3b: R_X86_64_PC32 res-0x4
  3f: 90                      nop
  40: 5d                      pop    %rbp
  41: c3
```

A instrução localizada no endereço **4** (`lea 0x0(%rip),%rax`) tem associada uma relocalização para atualizar o *offset* relativo ao PC (`R_X86_64_PC32`) no acesso à variável **b**.
A instrução localizada no endereço **b** (`mov %rax,0x0(%rip)`) tem associada uma relocalização para atualizar o *offset* relativo ao PC (`R_X86_64_PC32`) no acesso à variável **pb**.

Algoritmo de relocalização

```
foreach section s {
      foreach relocation entry r {
            ptr = s + r.offset   local do ficheiro onde é preciso afetar
            if (r.type ==  R_X86_64_PC32) {
                  address = address(s) + r.offset;     endereço de execução
                  *ptr = adress(r.symbol) + r.addend - address;
            }
            if (r.type ==  R_X86_64_64)
                  *ptr = adress(r.symbol) + r.addend;
}     }
```



(ver endereços finais com nm a.out)

Ficheiro objeto executável

O ficheiro objeto executável contém toda a informação necessária para carregar o programa em memória para ser executado. As secções e os símbolos têm posições definidas e as referências a símbolos estão preenchidas (foram preenchidas através das *relocations*).

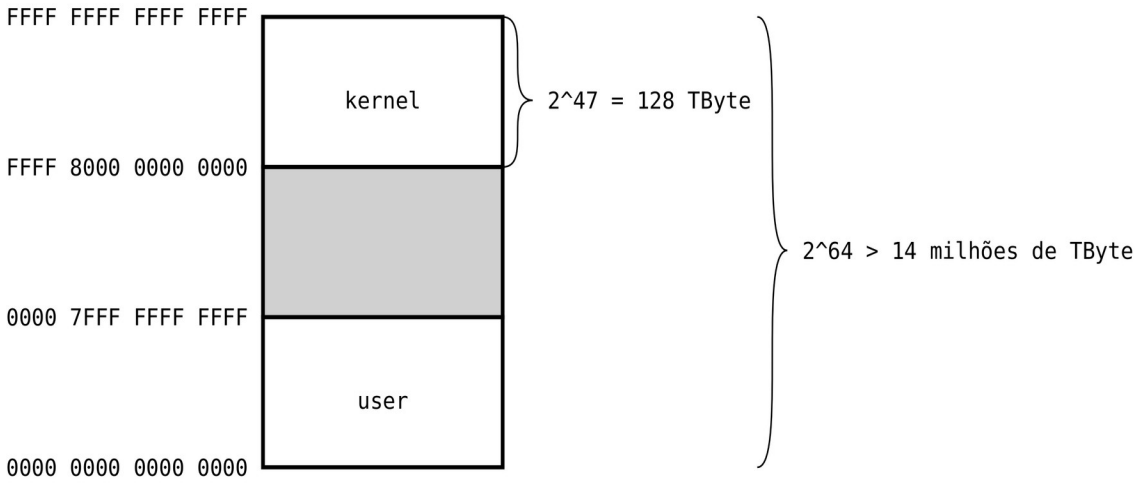
```
$ objdump -h main

main:     file format elf64-x86-64

Sections:
Idx Name          Size      VMA               LMA               File off  Algn
  0 .interp       0000001c  0000000000000318  0000000000000318  00000318  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  1 .note.gnu.property 00000020  0000000000000338  0000000000000338  00000338
2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .note.gnu.build-id 00000024  0000000000000358  0000000000000358  00000358
2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .note.ABI-tag 00000020  000000000000037c  000000000000037c  0000037c  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .gnu.hash     00000024  00000000000003a0  00000000000003a0  000003a0  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  5 .dynsym       00000090  00000000000003c8  00000000000003c8  000003c8  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  6 .dynstr       0000007d  0000000000000458  0000000000000458  00000458  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  7 .gnu.version  0000000c  00000000000004d6  00000000000004d6  000004d6  2**1
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  8 .gnu.version_r 00000020  00000000000004e8  00000000000004e8  000004e8  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  9 .rela.dyn     000000d8  0000000000000508  0000000000000508  00000508  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 10 .init         0000001b  0000000000001000  0000000000001000  00001000  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 11 .plt          00000010  0000000000001020  0000000000001020  00001020  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 12 .plt.got      00000010  0000000000001030  0000000000001030  00001030  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 13 .text         000001c5  0000000000001040  0000000000001040  00001040  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 14 .fini         0000000d  0000000000001208  0000000000001208  00001208  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 15 .rodata       00000008  0000000000002000  0000000000002000  00002000  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 16 .eh_frame_hdr 00000044  0000000000002008  0000000000002008  00002008  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 17 .eh_frame     00000110  0000000000002050  0000000000002050  00002050  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 18 .init_array   00000008  0000000000003df0  0000000000003df0  00002df0  2**3
                  CONTENTS, ALLOC, LOAD, DATA
 19 .fini_array   00000008  0000000000003df8  0000000000003df8  00002df8  2**3
                  CONTENTS, ALLOC, LOAD, DATA
 20 .dynamic      000001c0  0000000000003e00  0000000000003e00  00002e00  2**3
                  CONTENTS, ALLOC, LOAD, DATA
 21 .got          00000040  0000000000003fc0  0000000000003fc0  00002fc0  2**3
                  CONTENTS, ALLOC, LOAD, DATA
 22 .data         00000030  0000000000004000  0000000000004000  00003000  2**3
                  CONTENTS, ALLOC, LOAD, DATA
 23 .bss          00000010  0000000000004030  0000000000004030  00003030  2**3
```

```
                  ALLOC
 24 .comment      0000002a  0000000000000000  0000000000000000  00003030  2**0
                  CONTENTS, READONLY
```

Execução de um programa no Linux

Os processadores atuais, dos 64 *bits* de endereço, apenas implementam 48 *bits* em endereço virtual e cerca de 40 *bits* em endereço físico.

Nos sistemas operativos, a primeira metade do espaço de endereçamento virtual **`0 – 7fff ffff ffff`** é usada para processos e a segunda metade **`ffff 8000 0000 0000 – ffff ffff ffff ffff`** é usada pelo *kernel*.



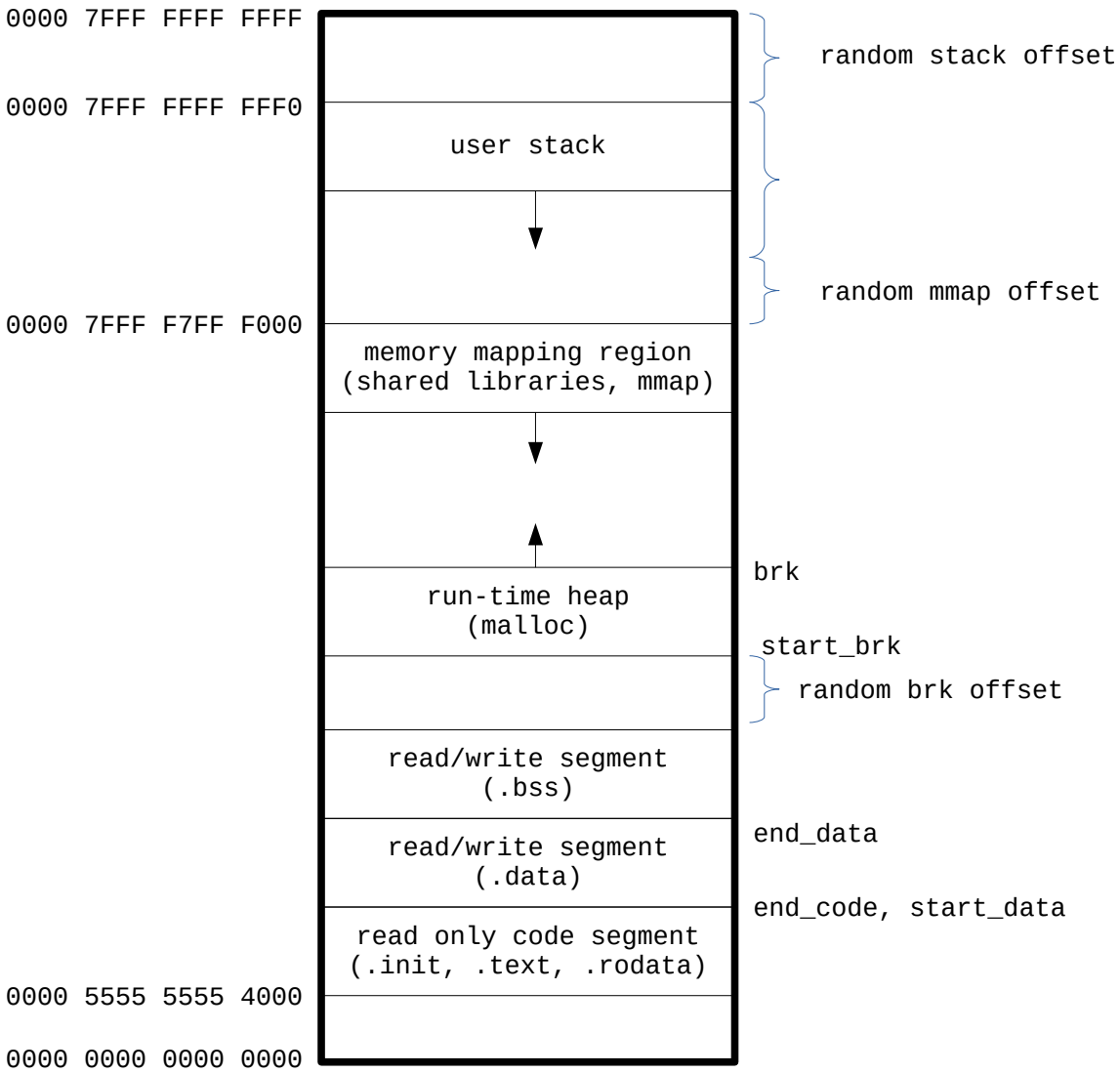
Ao lançar um processo, o Linux faz a seguinte ocupação do espaço de endereçamento virtual:

```
$ cat /proc/self/maps
55c202d2e000-55c202d36000 r-xp 00000000 08:21 524313   /bin/cat
55c202f35000-55c202f36000 r--p 00007000 08:21 524313   /bin/cat
55c202f36000-55c202f37000 rw-p 00008000 08:21 524313   /bin/cat
55c204b54000-55c204b75000 rw-p 00000000 00:00 0        [heap]
7f4070819000-7f407125c000 r--p 00000000 08:21 2103794  /usr/lib/locale/locale-archive
7f407125c000-7f4071443000 r-xp 00000000 08:21 2495886  /lib/x86_64-linux-gnu/libc-2.27.so
7f4071443000-7f4071643000 ---p 001e7000 08:21 2495886  /lib/x86_64-linux-gnu/libc-2.27.so
7f4071643000-7f4071647000 r--p 001e7000 08:21 2495886  /lib/x86_64-linux-gnu/libc-2.27.so
7f4071647000-7f4071649000 rw-p 001eb000 08:21 2495886  /lib/x86_64-linux-gnu/libc-2.27.so
7f4071649000-7f407164d000 rw-p 00000000 00:00 0
7f407164d000-7f4071674000 r-xp 00000000 08:21 2495858  /lib/x86_64-linux-gnu/ld-2.27.so
7f4071831000-7f4071855000 rw-p 00000000 00:00 0
7f4071874000-7f4071875000 r--p 00027000 08:21 2495858  /lib/x86_64-linux-gnu/ld-2.27.so
7f4071875000-7f4071876000 rw-p 00028000 08:21 2495858  /lib/x86_64-linux-gnu/ld-2.27.so
7f4071876000-7f4071877000 rw-p 00000000 00:00 0
7ffdae2ad000-7ffdae2ce000 rw-p 00000000 00:00 0              [stack]
7ffdae3c9000-7ffdae3cc000 r--p 00000000 00:00 0              [vvar]
7ffdae3cc000-7ffdae3ce000 r-xp 00000000 00:00 0              [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0      [vsyscall]
```

Os endereços de execução são atribuídos aleatoriamente no momento do carregamento do programa em memória.

O insight atribui endereços fixos. Podem ser observados assim:

```
$ ps -x

$ cat /proc/<pid>/maps/
```



Exercício

Verificar os endereços indicados.

Momentos importantes na vida de um programa

- compilação (*compile time*)
- ligação (*link time*)
- carregamento (*load time*)
- execução (*run time*)

Código independente da posição (PIC)

Considere-se o ficheiro **`main.c`** como o código fonte do programa de aplicação e o ficheiro **`libexa.c`** como o código fonte da biblioteca.

Nestas experiências foram utilizados as seguintes ferramentas: gcc 9.3 e binutils 2.35.50

```
extern int lib_var;
void lib_func();

int prog_var = 4;

int prog_func() {
      return prog_var + lib_var;
}

int main() {
      lib_func();
      prog_func();
}
```

`main.c`

```
extern int prog_var;
int prog_func();

int lib_var = 3;

void lib_func() {
      lib_var = prog_var;
      prog_var = prog_func();
}
```

`libexa.c`

O código binário é produzido sob o controlo do seguinte *makefile*:

```
CFLAGS = -c -Wall -std=c11 -g --save-temps -Og

all: libexa.so main

libexa.so: libexa.o
      gcc -shared libexa.o -o libexa.so

libexa.o: libexa.c
      gcc $(CFLAGS) -fPIC libexa.c -o libexa.o

main: main.o
      gcc main.o -L. -lexa -o main

main.o: main.c
      gcc $(CFLAGS) main.c -o main.o
```

Acesso a variáveis

As variáveis que interessa analisar são as variáveis globais com ligação externa. As variáveis com ligação interna são resolvidas pelo compilador e acedidas com endereçamento relativo ao RIP.

Acesso a variáveis globais a partir da aplicação

As variáveis globais, as definidas na aplicação – **`prog_var`** e as definidas na biblioteca - **`lib_var`**, são mapeadas na secção **`.data`** da aplicação. São acedidas com endereçamento relativo ao RIP e o seu endereço é determinado em *link time*.

```
$ objdump -d main

000000000000115a <prog_func>:
    115a:   8b 05 b8 2e 00 00 mov    0x2eb8(%rip),%eax    # 4018 <lib_var>
    1160:   03 05 aa 2e 00 00 add    0x2eaa(%rip),%eax    # 4010 <prog_var>
    1166:   c3                       retq
```

Acesso a variáveis globais a partir da biblioteca

A posição das variáveis, relativamente ao código da biblioteca, só é determinável em tempo de carregamento (*load time*). O acesso a estas variáveis é feito via tabela GOT (**Global Offset Table**) quer sejam definidas no programa de aplicação quer sejam definidas na biblioteca.

Observar o código de acesso às variáveis globais **`prog_var`** e **`lib_var`** (instruções de **`112b`** a **`113b`**):

```
$ objdump  -d libexa.so
0000000000001119 <lib_func1>:

   1119: f3 0f 1e fa          endbr64
   111d: 53                   push   %rbx
   111e: 48 8b 1d cb 2e 00 00 mov    0x2ecb(%rip),%rbx    # 3ff0 <prog_var>
   1125: 8b 13                mov    (%rbx),%edx
   1127: 48 8b 05 a2 2e 00 00 mov    0x2ea2(%rip),%rax    # 3fd0 <lib_var-0x58>
   112e: 89 10                mov    %edx,(%rax)
   1130: b8 00 00 00 00       mov    $0x0,%eax
   1135: e8 16 ff ff ff       callq  1050 <prog_func@plt>
   113a: 89 03                mov    %eax,(%rbx)
   113c: 5b                   pop    %rbx
   113d: c3                   ret
```

A tabela GOT tem uma entrada por cada variável externa referida.

Cada módulo em biblioteca possui a secção .**`got`** onde é alojada a tabela GOT. Esta secção é localizada a uma distância fixa em relação à secção .**`text`**, determinada em *link time*. O acesso ao conteúdo da tabela GOT é realizado com endereçamento relativo ao RIP.

Durante o carregamento do programa, as entradas da tabela GOT são preenchidas pelo *linker* dinâmico, na fase de relocalização, com o endereço absoluto das variáveis.

A instrução **`mov 0x2ecb(%rip),%rbx`** coloca em RBX o conteúdo da entrada da GOT relativa à variável **`prog_var`**, que é o endereço absoluto desta variável.

A instrução **`mov (%rbx), %edx`** carrega o conteúdo de **`prog_var`** em EDX.

Invocação de funções

Tal como nas variáveis, o endereço das funções da biblioteca só é determinado em tempo de carregamento. O acesso ao endereço das funções para chamadas entre programa de aplicação e biblioteca é diferente do utilizado para as variáveis, mas poderia ser igual.

O objetivo é manter o código de invocação de função igual quer na ligação estática quer na ligação dinâmica, isto é, não é necessário gerar o código de utilizador diferente para cada tipo de ligação. No caso de ligação estática chama logo a função, no caso de ligação dinâmica chama a PLT.

Invocação da função lib_func a partir do programa

```
$ objdump -d main --section=.text

000000000000117a <main>:
    117a:   f3 0f 1e fa             endbr64
    117e:   48 83 ec 08             sub    $0x8,%rsp
    1182:   b8 00 00 00 00          mov    $0x0,%eax
    1187:   e8 e4 fe ff ff          call   1070 <lib_func1@plt>
    118c:   b8 00 00 00 00          mov    $0x0,%eax
    1191:   e8 ca fe ff ff          call   1060 <lib_func2@plt>
    1196:   b8 00 00 00 00          mov    $0x0,%eax
    119b:   e8 c9 ff ff ff          call   1169 <prog_func>
    11a0:   48 83 c4 08             add    $0x8,%rsp
    11a4:   c3                      ret
```

A chamada a funções de biblioteca é feita por via da PLT (**Procedure Linkage Table**). A partir da versão 8 do compilador GCC passou a haver uma PLT primária **`.plt`** e PLT secundária **`.plt.sec`**.

```
$ objdump -d main

Disassembly of section .plt:

0000000000001020 <.plt>:
    1020:       ff 35 92 2f 00 00       push   0x2f92(%rip)        # 3fb8 <_GLOBAL_OFFSET_TABLE_+0x8>
    1026:       f2 ff 25 93 2f 00 00    bnd jmp *0x2f93(%rip)      # 3fc0 <_GLOBAL_OFFSET_TABLE_+0x10>
    102d:       0f 1f 00                nopl   (%rax)
    1030:       f3 0f 1e fa             endbr64
    1034:       68 00 00 00 00          push   $0x0
    1039:       f2 e9 e1 ff ff ff       bnd jmp 1020 <.plt>
    103f:       90                      nop
    1040:       f3 0f 1e fa             endbr64
    1044:       68 01 00 00 00          push   $0x1
    1049:       f2 e9 d1 ff ff ff       bnd jmp 1020 <.plt>
    104f:       90                      nop

Disassembly of section .plt.sec:

0000000000001060 <lib_func2@plt>:
    1060:       f3 0f 1e fa             endbr64
    1064:       f2 ff 25 5d 2f 00 00    bnd jmp *0x2f5d(%rip)        # 3fc8 <lib_func2>
    106b:       0f 1f 44 00 00          nopl   0x0(%rax,%rax,1)

0000000000001070 <lib_func1@plt>:
    1070:       f3 0f 1e fa             endbr64
    1074:       f2 ff 25 55 2f 00 00    bnd jmp *0x2f55(%rip)        # 3fd0 <lib_func1>
    107b:       0f 1f 44 00 00          nopl   0x0(%rax,%rax,1)
```

Em cada entrada de **`.plt.sec`** existe um **`jmp`** indireto sobre uma entrada da tabela GOT.
A instrução **`jmpq *0x2f55(%rip)`** acede à entrada GOT de endereço **`0x3fd0`** onde se encontra o endereço absoluto da função **`lib_func1`**.

O *linker* aplica um procedimento designado por *lazy binding* que consiste em protelar a atualização da entrada GOT para o momento da primeira chamada. Até lá, todas as entradas da tabela GOT apontam para a PLT primária, para a entrada da respetiva função. Esta instrução carrega no *stack* um argumento que corresponde ao número identificador da função e em seguida salta para a PLT[0] que por sua vez invoca o *linker* dinâmico apontado por GOT[2].

.bss	**lib_var**	**4018**	
.data	**prog_var**	**4010**	
		4008	
		4000	
.got	**GOT[9]**	**3ff8**	
	GOT[8]	**3ff0**	
	GOT[7]	**3fe8**	
	GOT[6]	**3fe0**	
	GOT[5]	**3fd8**	
	GOT[4]	**3fd0**	address of lib_func1
	GOT[3]	**3fc8**	address of lib_func2
	GOT[2]	**3fc0**	address of dynamic linker
	GOT[1]	**3fb8**	address of reloc entries
	GOT[0]	**3fb0**	address of .dynamic 0x3db
.text			
		1187	**call 1070 <lib_func1@plt> .plt.sec[1]**
.plt.sec	**[1]**	**103b**	**nop**
		1036	**bnd jmpq *0x2f5d(%rip) 3fc8 GOT[4]**
		1070	**endbr64**
	[0]	**106b**	**nop**
		1064	**bnd jmpq *0x2f55(%rip) 3fd0 GOT[3]**
		1060	**endbr64**
.plt		**1020**	**pushq 0x2f92(%rip) # 3fb8 GOT[1]**
		1026	**bnd jmpq *0x2f93(%rip) # 3fc0 GOT[2]**
		102d	**nop**
		1030	**endbr64**
		1034	**pushq $0x0**
		1039	**bnd jmpq 1020 <.plt>**
		103f	**nop**
		1040	**endbr64**
		1044	**pushq $0x1**
		1049	**bnd jmpq 1020 <.plt>**
		104f	**nop**

Execução na primeira chamada à função **`lib_func`**:

RIP	Ação
`1187`	Chamar a função **`lib_func1`** a partir da função **`main`** – salta para **`.plt.sec[1]`**.
`1070`	Na **`.plt.sec[1]`** salta para o endereço contido na **`GOT[4]`**. Da primeira vez é um endereço na PLT primária.
`1030`	Empilhar o identificador da função **`lib_func`** (**`pushq 0x0`**) e saltar para **`PLT[0]`**.
`1020`	Em **`PLT[0]`**, depois de empilhar **`GOT[1]`** salta para **`GOT[2]`** que é o endereço do *linker* dinâmico. O *linker*, baseado nos parâmetros recebidos, substitui o conteúdo de **`GOT[3]`** pelo o endereço de execução de **`lib_func`**.

Execução das chamadas seguintes à função **`lib_func`**:

RIP	Ação
`1187`	Chamar a função **`lib_func1`** a partir da função **`main`** – salta para **`.plt.sec[1]`**.
`1070`	Na **`.plt.sec[1]`** salta para o endereço contido na **`GOT[4]`**. Que é o endereço de **`lib_func`**.

`bnd` – faz parte do Intel Memory Protection Extention (MPX)

Invocação da função prog_func a partir da biblioteca

A operação é muito idêntica à anterior, serve apenas como exercício.

```
$ objdump -d libexa.so --section=.text

000000000000112a <lib_func>:
  112a:  53                      push   %rbx
  112b:  48 8b 1d be 2e 00 00    mov    0x2ebe(%rip),%rbx   # 3ff0 <prog_var>
  1132:  8b 13                   mov    (%rbx),%edx
  1134:  48 8b 05 95 2e 00 00    mov    0x2e95(%rip),%rax   # 3fd0 <lib_var-0x58>
  113b:  89 10                   mov    %edx,(%rax)
  113d:  b8 00 00 00 00          mov    $0x0,%eax
  1142:  e8 e9 fe ff ff          callq  1030 <prog_func@plt>
  1147:  89 03                   mov    %eax,(%rbx)
  1149:  b8 00 00 00 00          mov    $0x0,%eax
  114e:  e8 dd fe ff ff          callq  1030 <prog_func@plt>
  1153:  5b                      pop    %rbx
  1154:  c3                      retq

$ objdump -d main -section=.plt

0000000000001020 <.plt>:
 1020: ff 35 e2 2f 00 00 pushq 0x2fe2(%rip)  # 4008 <_GLOBAL_OFFSET_TABLE_+0x8>
 1026: ff 25 e4 2f 00 00 jmpq  *0x2fe4(%rip) # 4010 <_GLOBAL_OFFSET_TABLE_+0x10>
 102c: 0f 1f 40 00       nopl  0x0(%rax)

0000000000001030 <prog_func@plt>:
 1030: ff 25 e2 2f 00 00 jmpq  *0x2fe2(%rip) # 4018 <prog_func>
 1036: 68 00 00 00 00    pushq $0x0
 103b: e9 e0 ff ff ff    jmpq  1020 <.plt>
```

.bss		402c	
.data	lib_var	4028	
		4020	
.got.plt	[3]	4018	address of prog_func
	[2]	4010	address of dynamic linker
	[1]	4008	address of reloc entries
	[0]	4000	address of .dynamic 0x3e50
.got		3ff8	
		3ff0	
		3fe8	
		3fe0	
		3fd8	
		3fd0	
.text			
		1135	callq 1050 <prog_func@plt> .plt.sec[0]
.plt.sec		105b	nop
		1054	bnd jmp *0x2fbd(%rip) # 4018 <prog_func>
		1050	endbr64
.plt		103f	nop
		1039	bnd jmp 1020 <.plt>
		1034	push $0x0
		1030	endbr64
		102d	nop
		1026	bnd jmp *0x2fe3(%rip) # 4010
		1020	push 0x2fe2(%rip) # 4008