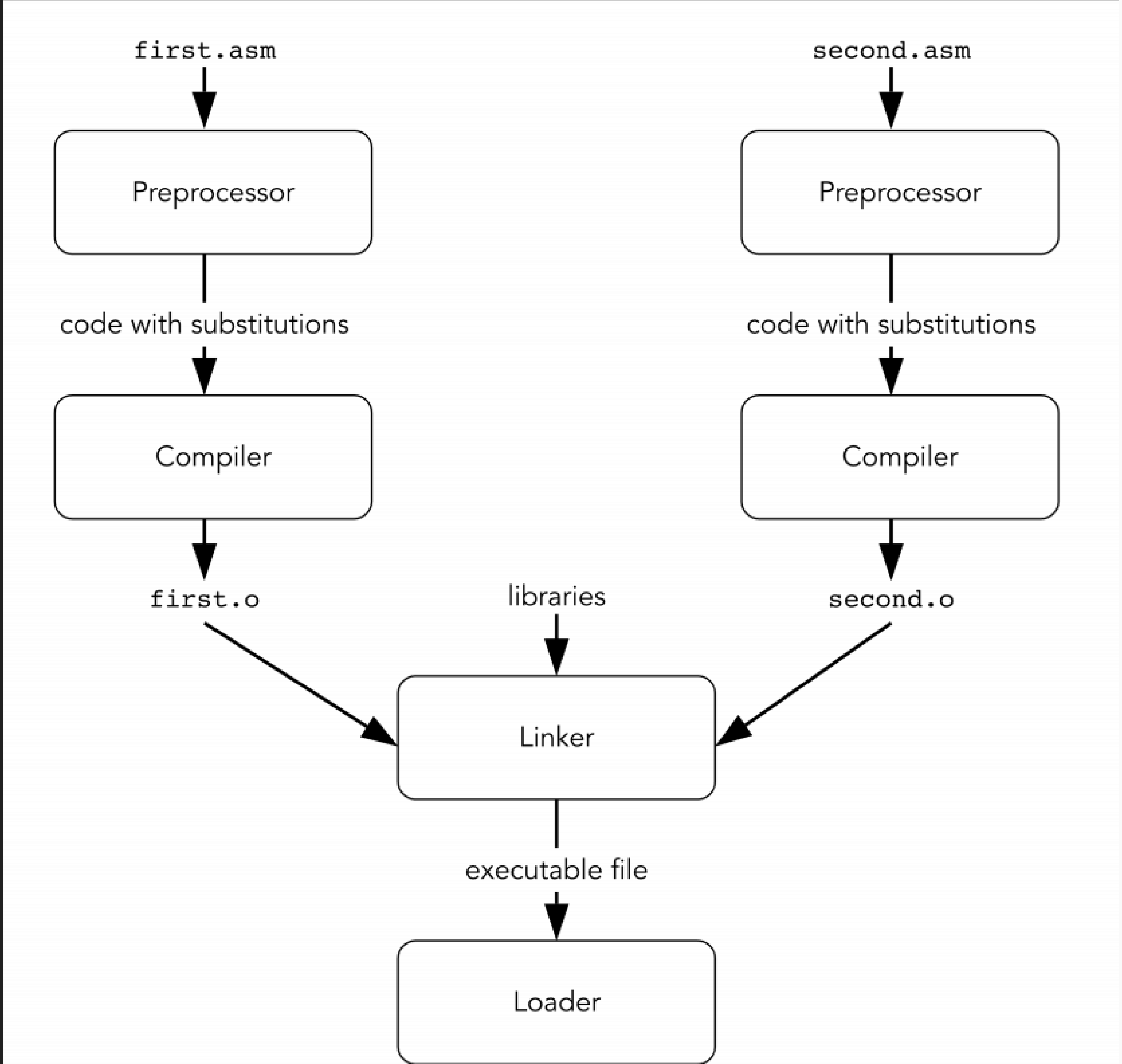
Низкоуровневый взгляд на динамические ELF-библиотеки

Introduction

- Intel 64 aka AMD64 aka x86_64.
- GNU/Linux
- Object file format: ELF files.
- Languages: C, Assembly (NASM)

module ->

object file ->



Introduction

- ELF – Executable and Linkable Format. Эти файлы можно разделить на три категории:
- Relocatable files – .o (то что получается после компиляции) является элементом static libraries (.a), т.е. может включать 1 или больше.
- Executable – программы после этапа линковки, готовые к запуску.
- Shared – .so dynamic libraries, они должны быть скомпонованы с запускаемым файлом в run-time.

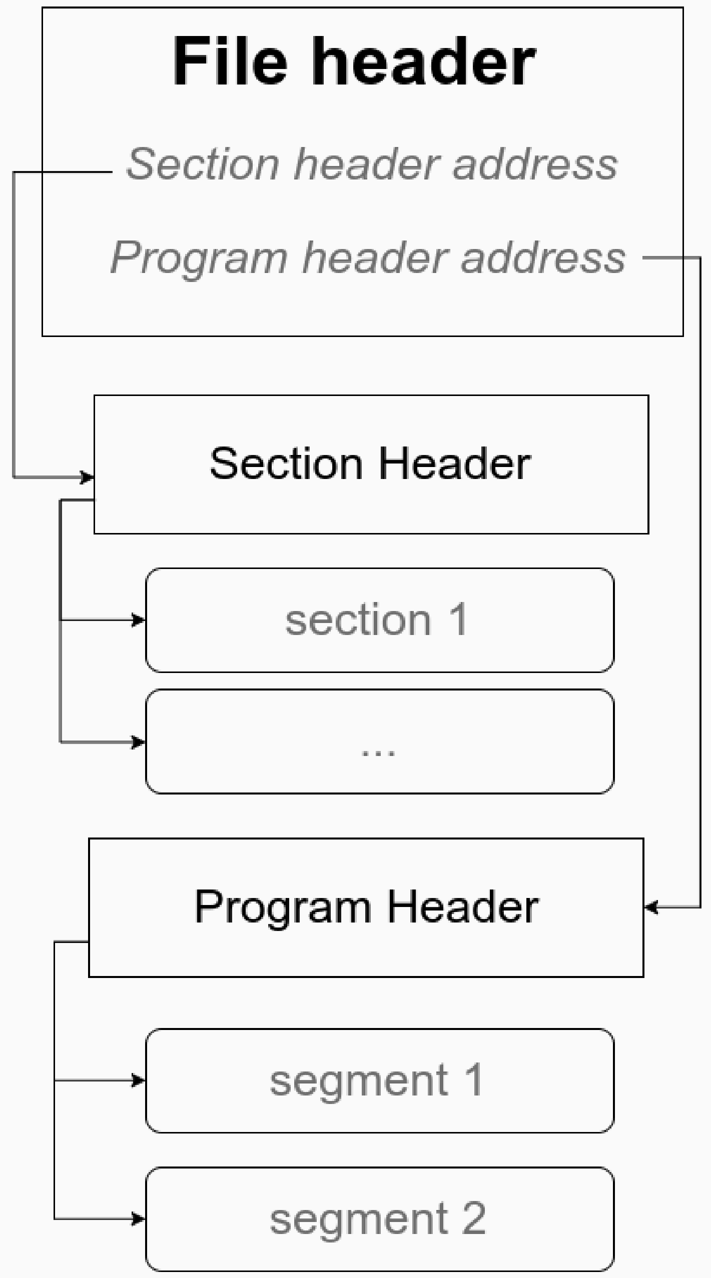
Tools

- Tools to examine object files:
  - readelf – ELF meta-information
  - objdump – meta-information of any format, disassembler
  - nm – only symbols.

  What we use:
  - objdump usually, less specific
  - readelf for verbose ELF structure

ELF header, Static Linkage

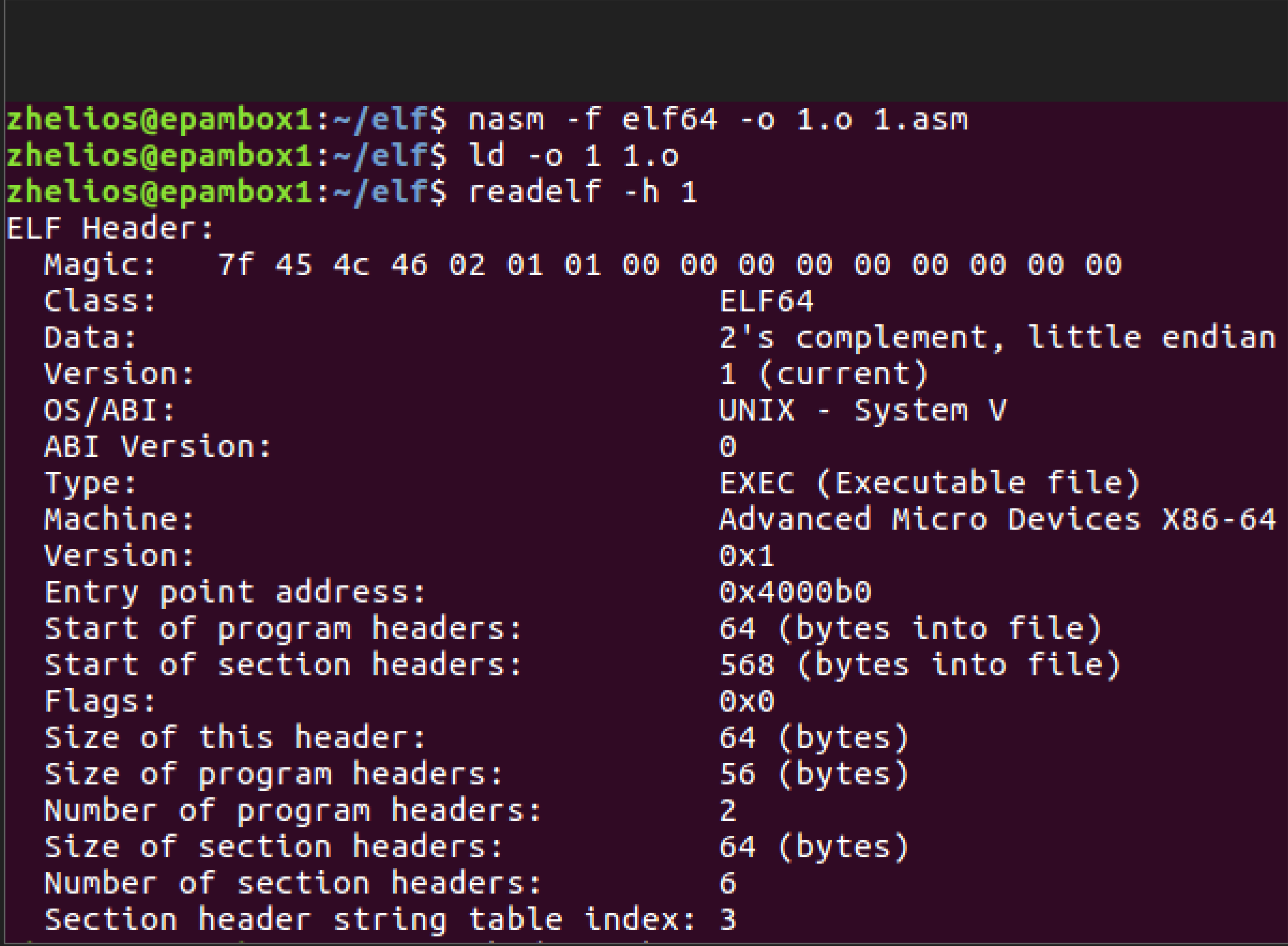
- Three headers:
  - File header
    - General info.
    - Links to Program and Section headers.
  - Section header
    - Information about sections.
    - Each section stores code or meta-information.
    - Needed for linking.
  - Program header
    - Instructions on how to create process image.
    - Information about segments.
    - Segment is a virtual memory region;
    - Needed for execution.



Typical sections:

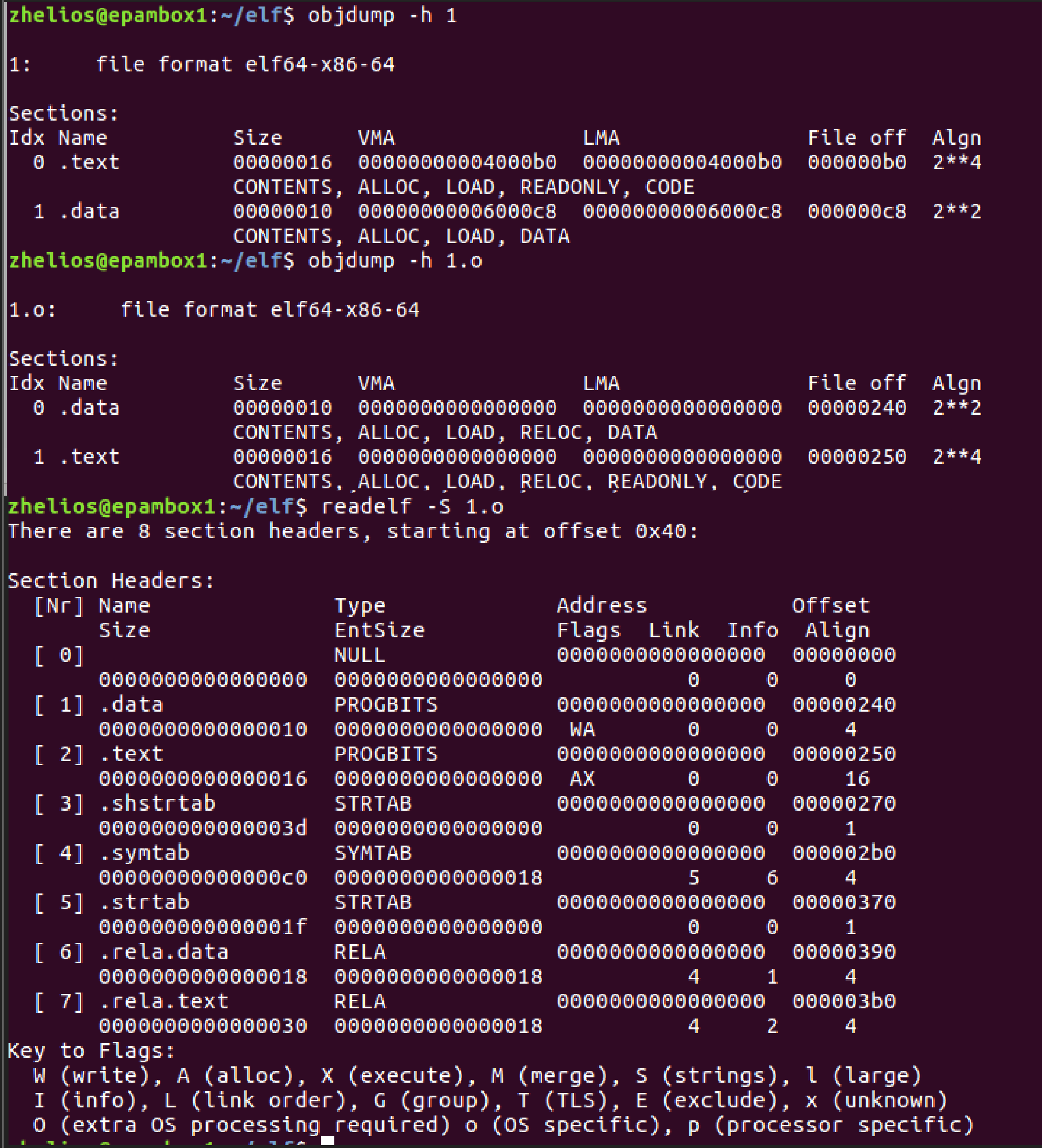
- **.data**
- **.text** – compiled instructions.
- **.rodata** – read only.
- **.bss** – zero-initialized data (only size is stored).
- **.line** – line numbers in source code.
- **.symtab** – symbol table.
- ...

```
section .data     ; global variables:
a: dq 123    ; int a = 123
b: dq a        ; int* b = &a
extern ext_variable

global _start ; visible to other
modules

section .text ;

_start:
mov rax, a ; rax := &a
mov rbx, b ; rbx := &b

jmp _start ;
```



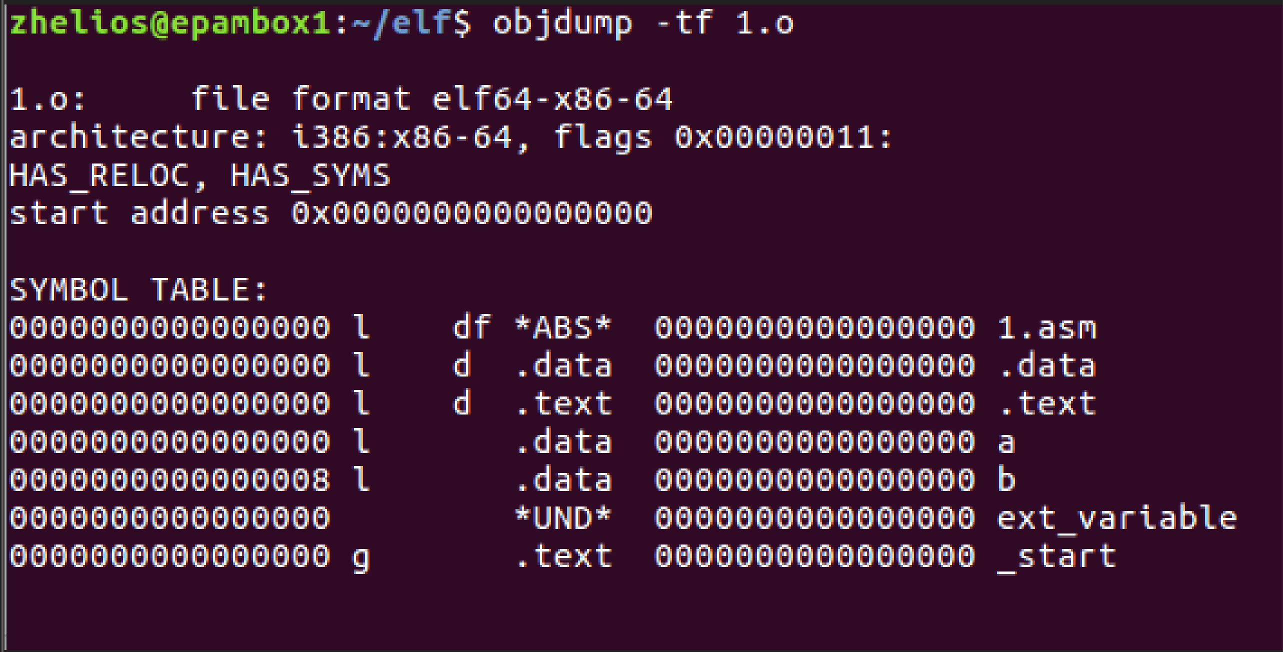
```
zhelios@epambox1:~/elf$ nasm -f elf64 -o 1.o 1.asm
zhelios@epambox1:~/elf$ ld -o 1 1.o
zhelios@epambox1:~/elf$ readelf -h 1
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little 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:          568 (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: 3
```

```
section .data     ; global variables:
a: dq 123    ; int a = 123
b: dq a         ; int* b = &a
extern ext_variable

global _start ; visible to other
modules

section .text ;

_start:
mov rax, a ; rax := &a
mov rbx, b ; rbx := &b

jmp _start ;
```



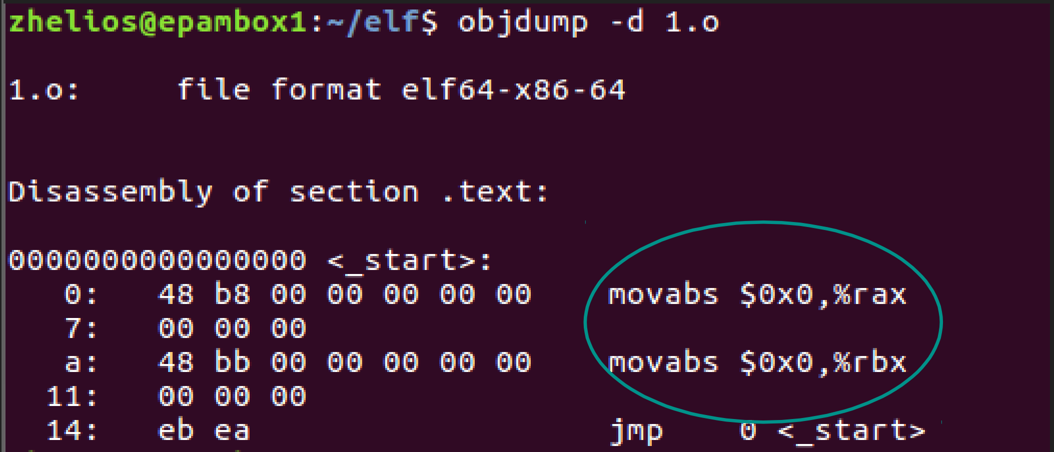
```
zhelios@epambox1:~/elf$ objdump -h 1

1:     file format elf64-x86-64

Sections:
Idx Name          Size      VMA               LMA               File off  Algn
  0 .text         00000016  00000000004000b0  00000000004000b0  000000b0  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .data         00000010  00000000006000c8  00000000006000c8  000000c8  2**2
                  CONTENTS, ALLOC, LOAD, DATA
zhelios@epambox1:~/elf$ objdump -h 1.o

1.o:     file format elf64-x86-64

Sections:
Idx Name          Size      VMA               LMA               File off  Algn
  0 .data         00000010  0000000000000000  0000000000000000  00000240  2**2
                  CONTENTS, ALLOC, LOAD, RELOC, DATA
  1 .text         00000016  0000000000000000  0000000000000000  00000250  2**4
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
zhelios@epambox1:~/elf$ readelf -S 1.o
There are 8 section headers, starting at offset 0x40:

Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .data             PROGBITS         0000000000000000  00000240
       0000000000000010  0000000000000000  WA       0     0     4
  [ 2] .text             PROGBITS         0000000000000000  00000250
       0000000000000016  0000000000000000  AX       0     0     16
  [ 3] .shstrtab         STRTAB           0000000000000000  00000270
       000000000000003d  0000000000000000           0     0     1
  [ 4] .symtab           SYMTAB           0000000000000000  000002b0
       00000000000000c0  0000000000000018           5     6     4
  [ 5] .strtab           STRTAB           0000000000000000  00000370
       000000000000001f  0000000000000000           0     0     1
  [ 6] .rela.data        RELA             0000000000000000  00000390
       0000000000000018  0000000000000018           4     1     4
  [ 7] .rela.text        RELA             0000000000000000  000003b0
       0000000000000030  0000000000000018           4     2     4
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), l (large)
  I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
```

```
section .data     ; global variables:
a: dq 123    ; int a = 123
b: dq a       ; int* b = &a
extern ext_variable

global _start ; visible to other
modules

section .text ;

_start:
mov rax, a ; rax := &a
mov rbx, b ; rbx := &b

jmp _start ;
```



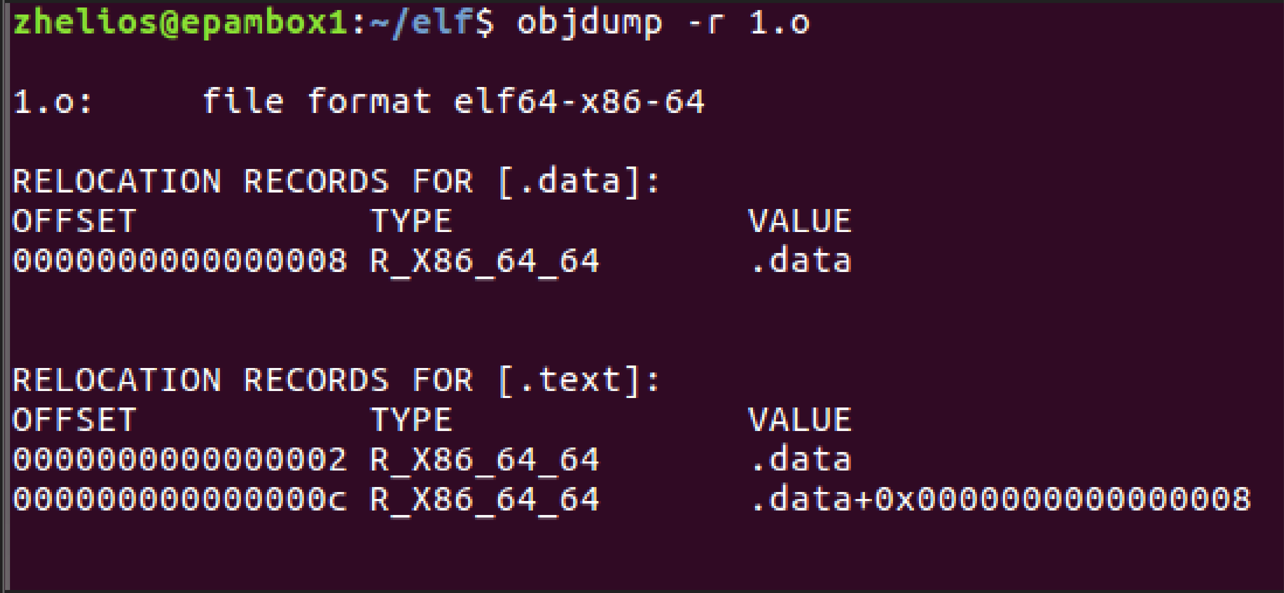
```
zhelios@epambox1:~/elf$ objdump -tf 1.o

1.o:     file format elf64-x86-64
architecture: i386:x86-64, flags 0x00000011:
HAS_RELOC, HAS_SYMS
start address 0x0000000000000000

SYMBOL TABLE:
0000000000000000 l    df *ABS*  0000000000000000 1.asm
0000000000000000 l    d  .data  0000000000000000 .data
0000000000000000 l    d  .text  0000000000000000 .text
0000000000000000 l       .data  0000000000000000 a
0000000000000008 l       .data  0000000000000000 b
0000000000000000         *UND*  0000000000000000 ext_variable
0000000000000000 g       .text  0000000000000000 _start
```

l – local
g – global (visible to other object files)
d – debug symbol
f – file name



```
zhelios@epambox1:~/elf$ objdump -d 1.o

1.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <_start>:
   0:   48 b8 00 00 00 00 00    movabs $0x0,%rax
   7:   00 00 00
   a:   48 bb 00 00 00 00 00    movabs $0x0,%rbx
  11:   00 00 00
  14:   eb ea                   jmp    0 <_start>
```

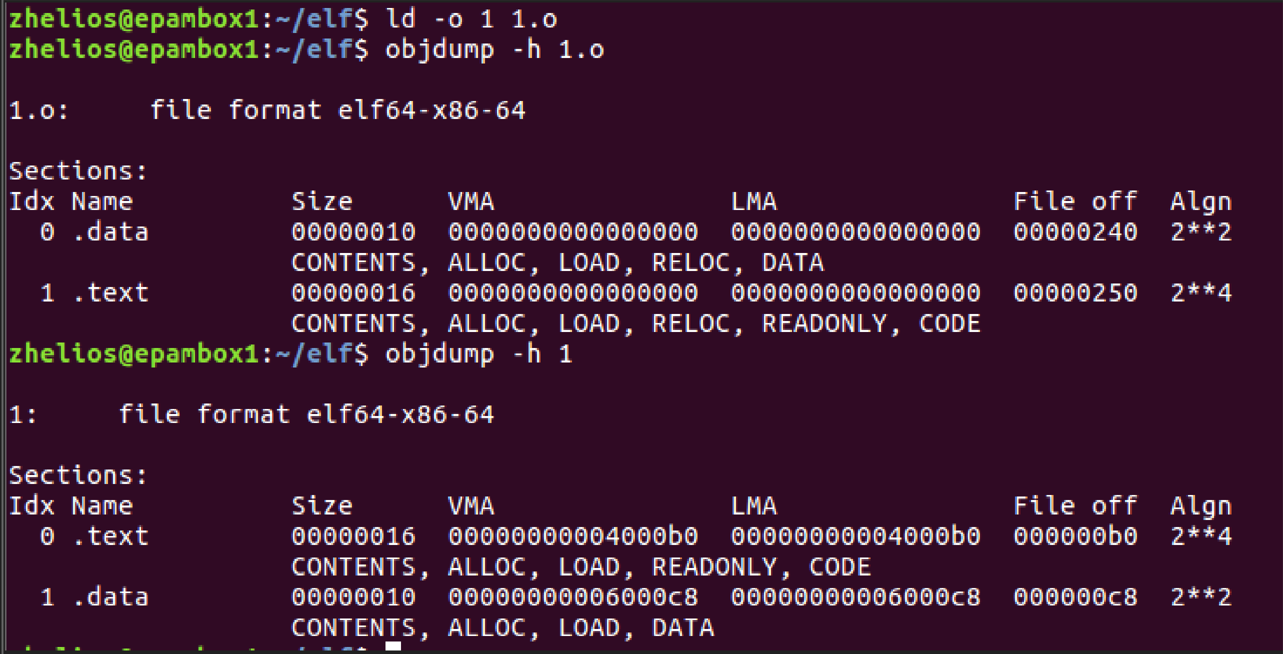
```
section .data     ; global variables:
a: dq 123    ; int a = 123
b: dq a        ; int* b = &a
extern ext_variable

global _start ; visible to other
modules

section .text ;

_start:
mov rax, a ; rax := &a
mov rbx, b ; rbx := &b

jmp _start ;
```



```
zhelios@epambox1:~/elf$ objdump -r 1.o

1.o:     file format elf64-x86-64

RELOCATION RECORDS FOR [.data]:
OFFSET           TYPE              VALUE
0000000000000008 R_X86_64_64       .data


RELOCATION RECORDS FOR [.text]:
OFFSET           TYPE              VALUE
0000000000000002 R_X86_64_64       .data
000000000000000c R_X86_64_64       .data+0x0000000000000008
```

Почему OFFSET в .data 0x8 ? ‘b’ хранит адрес ‘a’
Что это за адреса ? 0x2 , 0xC ?
На пред. сл. показано что опкод занимает 2 байта, а размер переменной 8
т.е. вставлять правильный адрес мы будем начиная с 0x2 байта и начиная с 0xC
R_X86_64_64 – самый базовый тип.

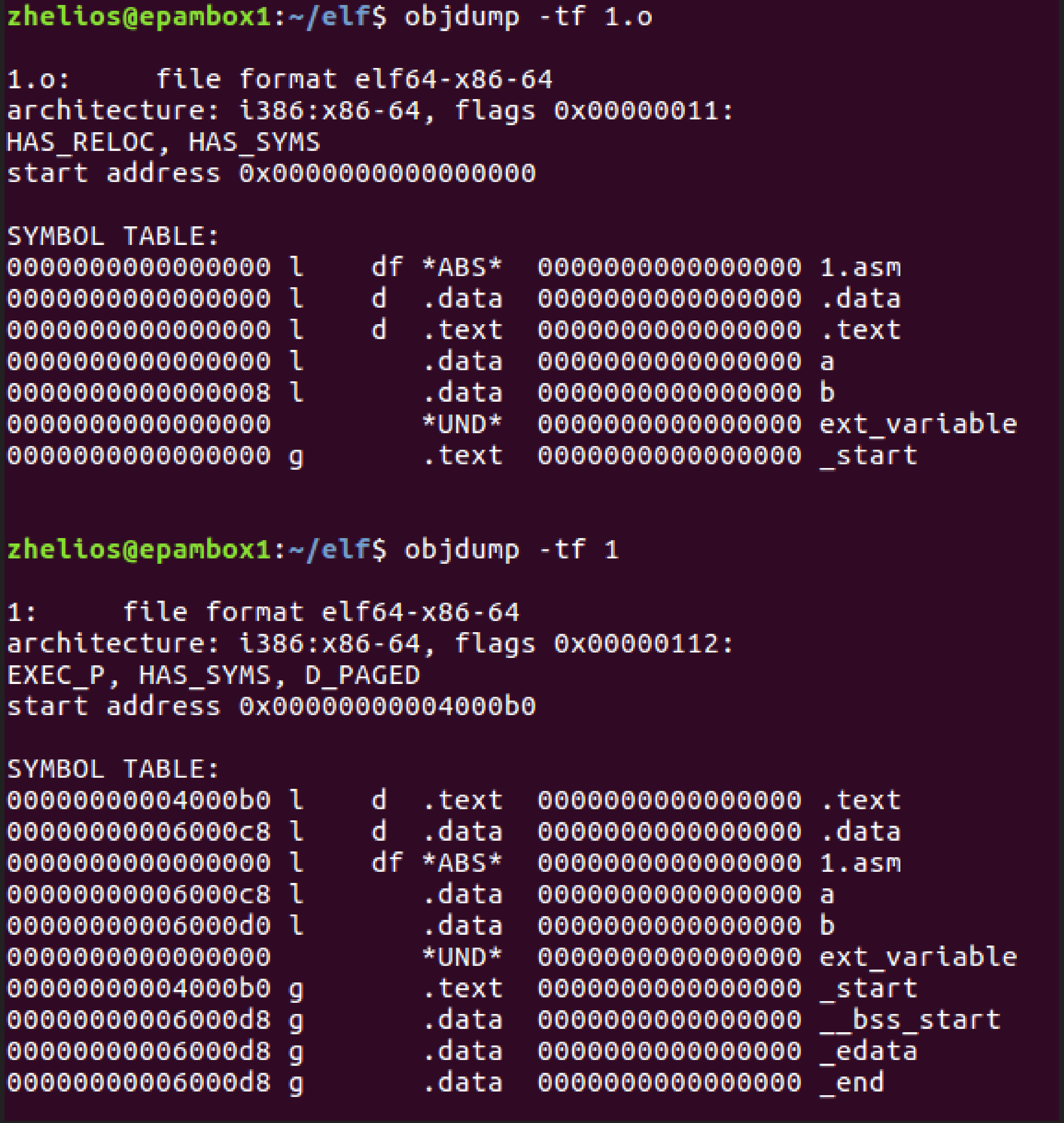
```
section .data     ; global variables:
a: dq 123    ; int a = 123
b: dq a        ; int* b = &a
extern ext_variable

global _start ; visible to other modules

section .text ;

_start:
mov rax, a ; rax := &a
mov rbx, b ; rbx := &b

jmp _start ;
```



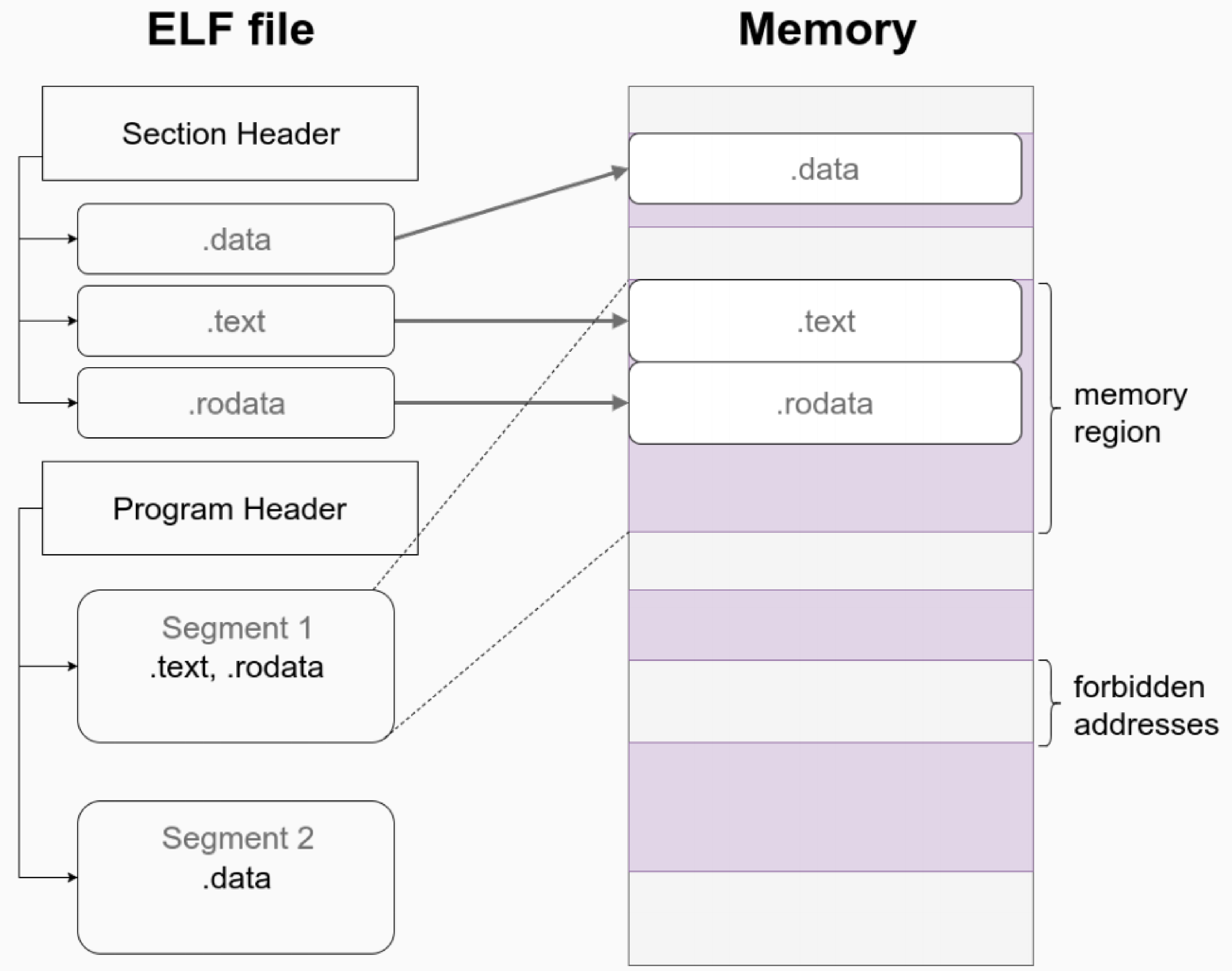
```
zhelios@epambox1:~/elf$ ld -o 1 1.o
zhelios@epambox1:~/elf$ objdump -h 1.o

1.o:     file format elf64-x86-64

Sections:
Idx Name          Size      VMA               LMA               File off  Algn
  0 .data         00000010  0000000000000000  0000000000000000  00000240  2**2
                  CONTENTS, ALLOC, LOAD, RELOC, DATA
  1 .text         00000016  0000000000000000  0000000000000000  00000250  2**4
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
zhelios@epambox1:~/elf$ objdump -h 1

1:     file format elf64-x86-64

Sections:
Idx Name          Size      VMA               LMA               File off  Algn
  0 .text         00000016  00000000004000b0  00000000004000b0  000000b0  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .data         00000010  00000000006000c8  00000000006000c8  000000c8  2**2
                  CONTENTS, ALLOC, LOAD, DATA
```

- Addresses are chosen.
- No more reloc mark.

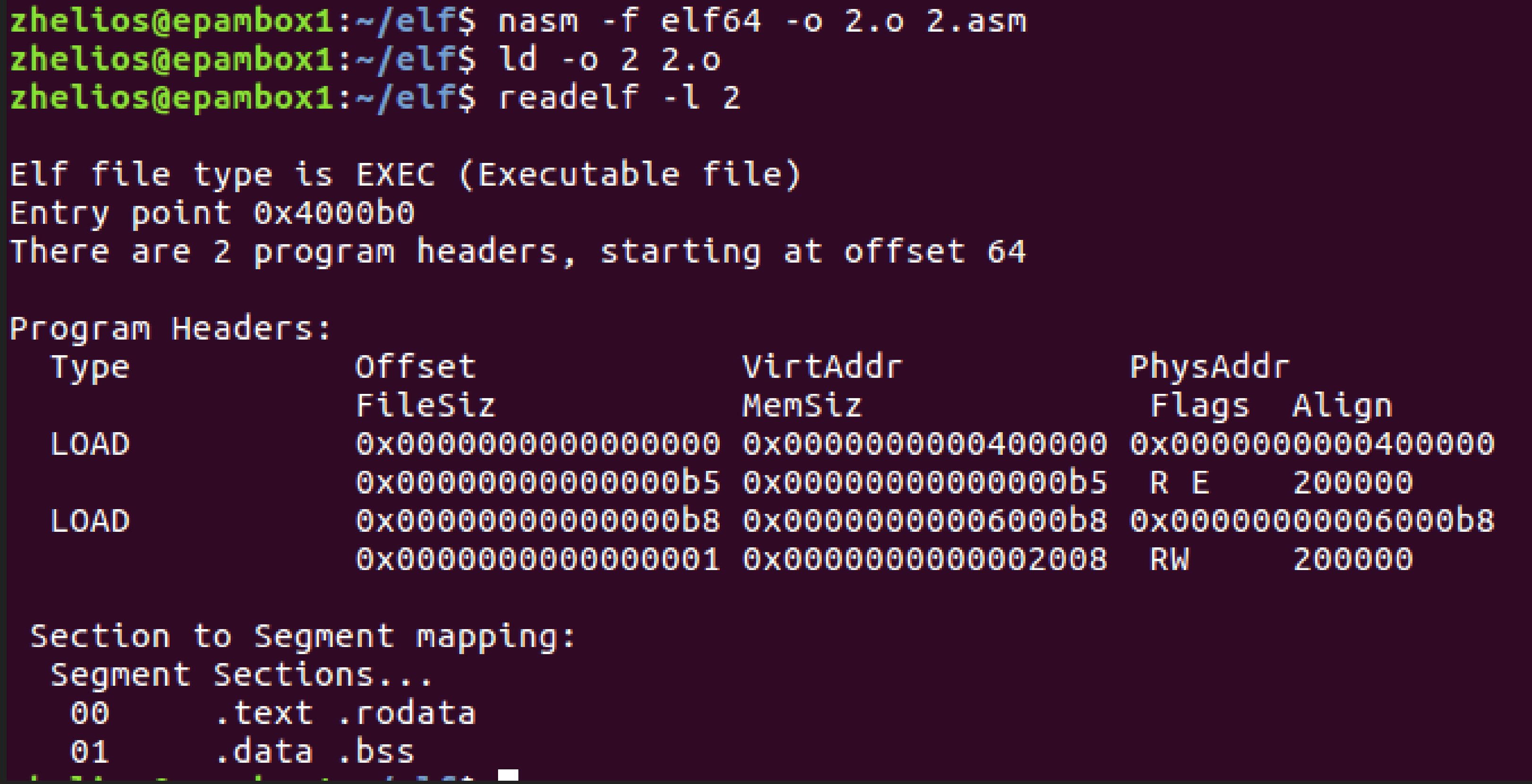
```
section .data     ; global variables:
a: dq 123     ; int a = 123
b: dq a         ; int* b = &a
extern ext_variable

global _start ; visible to other
modules

section .text ;

_start:
mov rax, a ; rax := &a
mov rbx, b ; rbx := &b

jmp _start ;
```



```
zhelios@epambox1:~/elf$ objdump -tf 1.o

1.o:     file format elf64-x86-64
architecture: i386:x86-64, flags 0x00000011:
HAS_RELOC, HAS_SYMS
start address 0x0000000000000000

SYMBOL TABLE:
0000000000000000 l    df *ABS*  0000000000000000 1.asm
0000000000000000 l    d  .data  0000000000000000 .data
0000000000000000 l    d  .text  0000000000000000 .text
0000000000000000 l       .data  0000000000000000 a
0000000000000008 l       .data  0000000000000000 b
0000000000000000         *UND*  0000000000000000 ext_variable
0000000000000000 g       .text  0000000000000000 _start


zhelios@epambox1:~/elf$ objdump -tf 1

1:     file format elf64-x86-64
architecture: i386:x86-64, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x00000000004000b0

SYMBOL TABLE:
00000000004000b0 l    d  .text  0000000000000000 .text
00000000006000c8 l    d  .data  0000000000000000 .data
0000000000000000 l    df *ABS*  0000000000000000 1.asm
00000000006000c8 l       .data  0000000000000000 a
00000000006000d0 l       .data  0000000000000000 b
0000000000000000         *UND*  0000000000000000 ext_variable
00000000004000b0 g       .text  0000000000000000 _start
00000000006000d8 g       .data  0000000000000000 __bss_start
00000000006000d8 g       .data  0000000000000000 _edata
00000000006000d8 g       .data  0000000000000000 _end
```


ELF file
Memory
Section Header
.data
.text
.rodata
Program Header
Segment 1
.text, .rodata
Segment 2
.data
.data
.text
.rodata
memory region
forbidden addresses

```
global _start
section .text

_start: jmp _start
section .data
db 10

section .rodata
db 1

section .bss
resq 1024
```



```
zhelios@epambox1:~/elf$ nasm -f elf64 -o 2.o 2.asm
zhelios@epambox1:~/elf$ ld -o 2 2.o
zhelios@epambox1:~/elf$ readelf -l 2

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              Flags  Align
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x00000000000000b5 0x00000000000000b5  R E    200000
  LOAD           0x00000000000000b8 0x00000000006000b8 0x00000000006000b8
                 0x0000000000000001 0x0000000000002008  RW     200000

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

Dynamic Linkage

- Third type of ELF files.
- Separate file, after linking.
  - .dll, .so
- Can be updated separately.
- Exposes some of global variables and functions.
- Relocation is partially performed.
- Reusable by other running processes.
- Spares memory, but has additional costs when using. Executable files use many libraries.

Dynamic linker’s job

- 1. Find and load dependencies.
  2. Perform relocation.
  3. Initialize the application and its dependencies
  4. Pass the control to the application.

How to find which libraries we need?

- Search locations:
    - rpath – to be found in section .dynamic
    - LD_LIBRARY_PATH environment variable.
    - runpath – to be found in section .dynamic
    - List in the file /etc/ld.so.conf.
    - Standards such as /lib
  - Depth-first-search order, dependencies and their dependencies.
  - Remember, there is an order on dependencies!
  - Does not load the same library twice.

How to select a symbol?

- As in static linking, we search by name through the symbol tables.
  Symbol can be defined in multiple objects, only one will exist in runtime.
  Depending on a set of existing objects, its location may change.
  Lookup scope of an object file an ordered list of a subset of the loaded objects.

Lookup scopes

- Last to first priority.
  • Global: the executable and all its dependencies recursively, in a breadth-first search order. Starts with the executable.
  • Legacy: look in metadata if DF_SYMBOLIC flag is set. If yes, local definitions are preferred.
  • Everything opened by dlopen call have a common additional separated scope. Not searched for normal lookups.

  LD_PRELOAD allows to add a library to global scope right after the executable itself.

Sharing library between processes

- • .data and .bss can not be shared anyway (each process should have its own global variables).
  • .text can be shared if consists of position independent code (PIC).
  • .rodata can be shared if it has no relocations (e.g. an address of a variable).

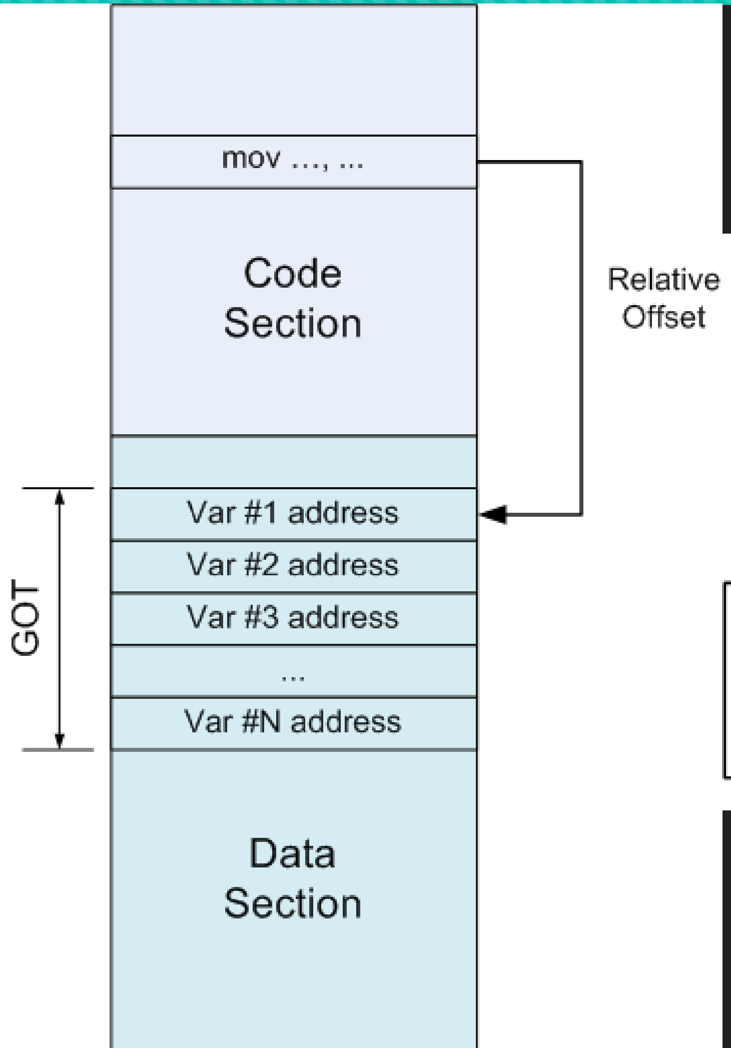
RIP-relative addressing

- RIP – register holding the address of current instruction (Program Counter) Intel 64 supports RIP-relative addressing out-of-the-box. Can we just change all addressing to RIP-relative?

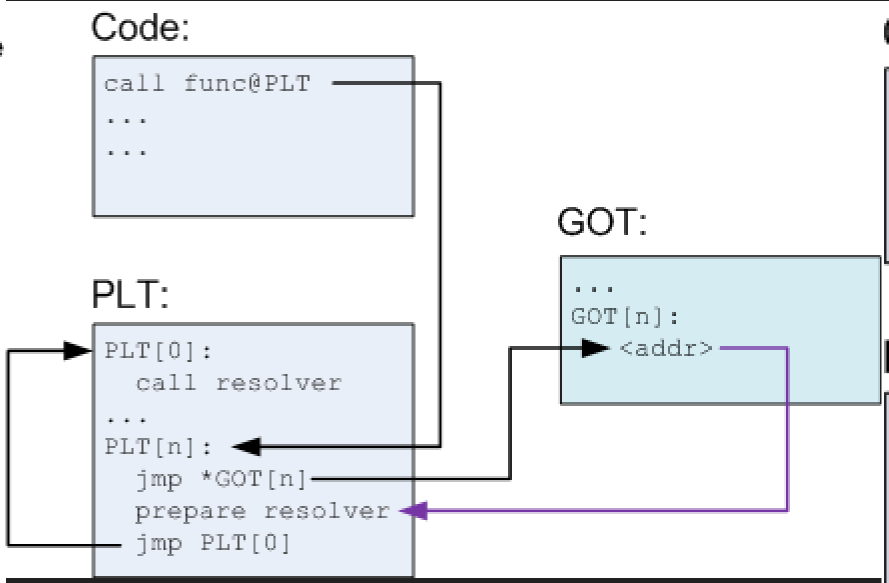
  • Works for addresses of local variables and functions: we know the offsets between current position in code and everything from the same object file.
  • Not for exported or imported symbols: we do not know which object will provide them.

  Solution: add level of indirection using Global Offset Table.

Global Offset Table & Procedure Linkage Table



1st time resolving

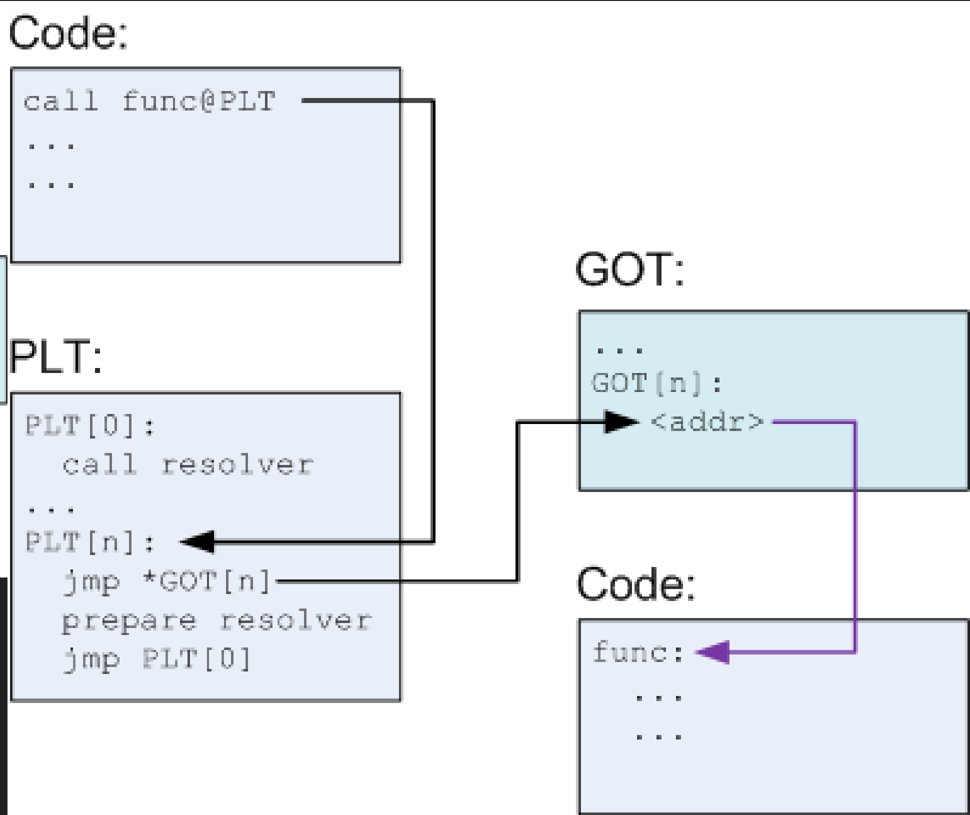


```
call func@PLT
...
...
```

```
PLT[0]:
  call resolver
...
PLT[n]:
  jmp *GOT[n]
  prepare resolver
  jmp PLT[0]
```

```
...
GOT[n]:
  <addr>
```

After 1st time resolving



```
call func@PLT
...
...
```

```
PLT[0]:
  call resolver
...
PLT[n]:
  jmp *GOT[n]
  prepare resolver
  jmp PLT[0]
```

```
...
GOT[n]:
  <addr>
```

```
func:
  ...
  ...
```