Linking
链接

100076202： 计算机系统导论

任课教师：

计卫星　　宿红毅　　张艳

原作者：

Randal E. **Bryant and** David R. O’Hallaron

提纲/Today

- 链接/Linking
- 实例分析：/Case study: Library interpositioning

C语言例程/Example C Program

```c
int sum(int *a, int n);

int array[2] = {1, 2};

int main()
{
    int val = sum(array, 2);
    return val;
}
```

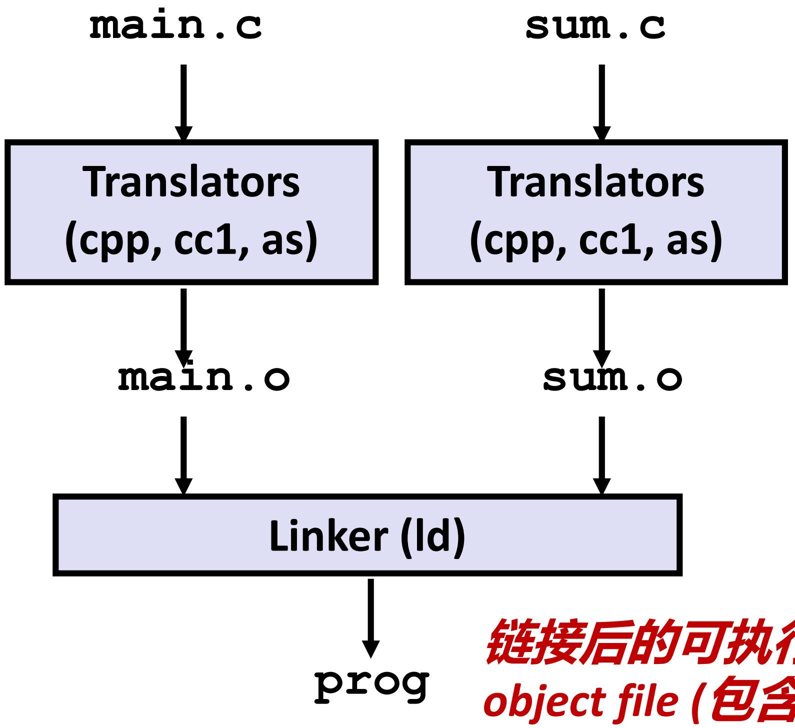
main.c

```c
int sum(int *a, int n)
{
    int i, s = 0;

    for (i = 0; i < n; i++) {
        s += a[i];
    }
    return s;
}
```

sum.c

静态链接/Static Linking

- **编译器主程序负责对程序进行编译和链接/Programs are translated and linked using a *compiler driver*:**
  - `linux> gcc -Og -o prog main.c sum.c`
  - `linux> ./prog`



源码文件/Source files

编译后独立的可重定位目标文件/Separately compiled relocatable object files

链接后的可执行目标文件/Fully linked executable object file (包含了两个文件中定义的所有代码和数据/contains code and data for all functions defined in `main.c` and `sum.c`)

为什么要有链接器？ / Why Linkers?

- **原因1：模块化/Reason 1: Modularity**
  - 程序可以写成更小的源文件的集合，而不是一个整体。
    Program can be written as a collection of smaller source files, rather than one monolithic mass.
  - 可以构建共用的函数库（后续讨论）/ Can build libraries of common functions (more on this later)
    - 例如数学库、标准C库/e.g., Math library, standard C library

为什么要有链接器？ /Why Linkers? (cont)

- **原因2：效率/Reason 2: Efficiency**
  - 时间：分别编译/Time: Separate compilation
    - 修改单个源码文件、编译和重新链接/Change one source file, compile, and then relink.
    - 无需重新编译其他的源码文件/No need to recompile other source files.
  - 空间：库/Space: Libraries
    - 共用的函数可以放进单个文件中共享使用/Common functions can be aggregated into a single file...
    - 可执行文件和运行的内存镜像只包含他们需要的函数和代码/Yet executable files and running memory images contain only code for the functions they actually use.

链接器的作用/What Do Linkers Do?

- **第1步：符号消解/Step 1: Symbol resolution**
  - 程序定义和引用的符号（全局变量和函数）Programs define and reference *symbols* (global variables and functions):
    - `void swap() {…}    /*` **定义符号**`swap/define symbol swap */`
    - `swap();            /*` **引用符号**`swap/reference symbol swap */`
    - `int *xp = &x;      /*` **定义符号**`xp/define symbol xp,` **引用符号** `x/reference x */`
  - 符号定义保存在目标文件的符号表中（汇编器完成）Symbol definitions are stored in object file (by assembler) in *symbol table*.
    - 符号表是structs数组/Symbol table is an array of `structs`
    - 每个条目包括符号的名称、大小以及位置/Each entry includes name, size, and location of symbol.
  - **在符号消解阶段，链接器将每个符号引用与符号定义相关联/During symbol resolution step, the linker associates each symbol reference with exactly one symbol definition.**

链接器的作用/ What Do Linkers Do? (cont)

- **第2步：重定位/Step 2: Relocation**
  - 将多个独立的代码和数据段合并到一个段中/Merges separate code and data sections into single sections
  - 将.o文件中的符号从相对位置改为可执行文件最终的在内存中的绝对地址/Relocates symbols from their relative locations in the .o files to their final absolute memory locations in the executable.
  - 更新所有引用将其改为最新的位置/Updates all references to these symbols to reflect their new positions.

让我们仔细看一下这两步/
Let’s look at these two steps in more detail....

三种不同的目标文件（模块）/Three Kinds of Object Files (Modules)

- **可重定位目标文件/Relocatable object file (`.o` file)**
  - 其中的代码和数据可以和其他可重定位目标文件一起形成可执行目标文件/Contains code and data in a form that can be combined with other relocatable object files to form executable object file.
    - 每个.o文件从一个源码文件.c获得/Each `.o` file is produced from exactly one source (`.c`) file
- **可执行目标文件/Executable object file (`a.out` file)**
  - 其中的代码和数据可以直接拷贝到内存中执行/Contains code and data in a form that can be copied directly into memory and then executed.
- **共享目标文件/Shared object file (`.so` file)**
  - 特定的可重定位目标文件，能以在加载时或者运行时装进内存并进行动态链接/Special type of relocatable object file that can be loaded into memory and linked dynamically, at either load time or run-time.
  - Windows上通常称为动态链接库/Called *Dynamic Link Libraries* (DLLs) by Windows

可重定位目标文件和可执行目标文件的区别

- **可执行目标文件包含了程序的入口点（Entry Point）**
  - 第一条指令的地址
- **.text、.rodata和.data类似，但是可执行目标文件中已完成重定位**
- **.init节中的_init函数定义了程序初始化过程**
- **可执行目标文件中不再需要.rel节**

可执行和可链接格式/Executable and Linkable Format (ELF)

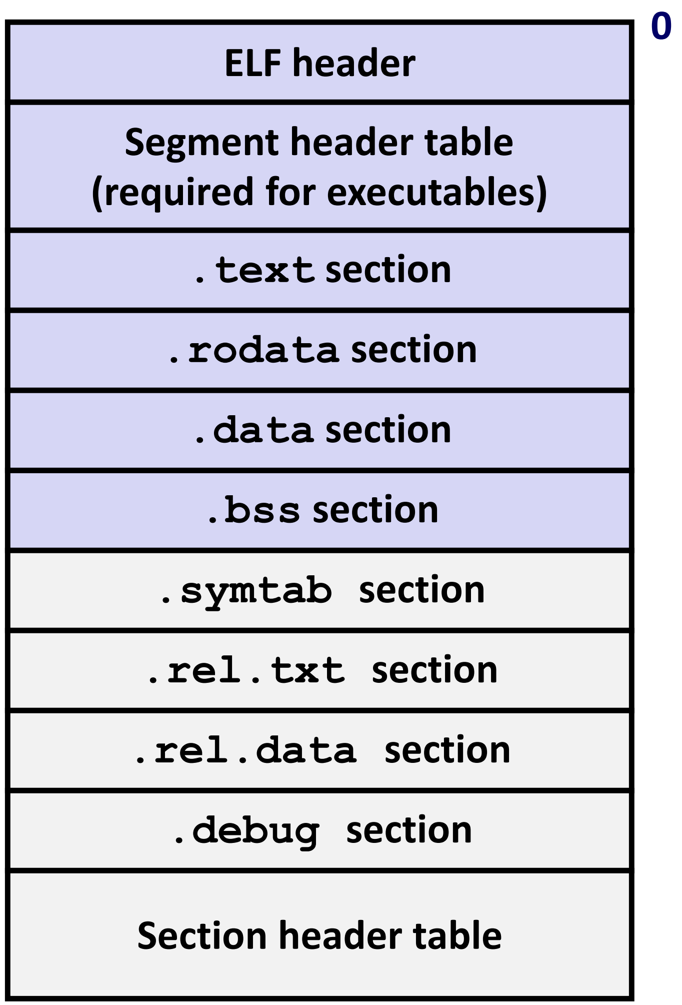
- **Linux类系统上目标文件的标准格式/Standard binary format for object files**

- **以下文件的统一格式/One unified format for**
  - 可重定位目标文件/Relocatable object files (`.o`),
  - 可执行目标文件/Executable object files （`a.out`)
  - 共享目标文件/Shared object files (`.so`)

- **通常称为ELF二进制/Generic name: ELF binaries**

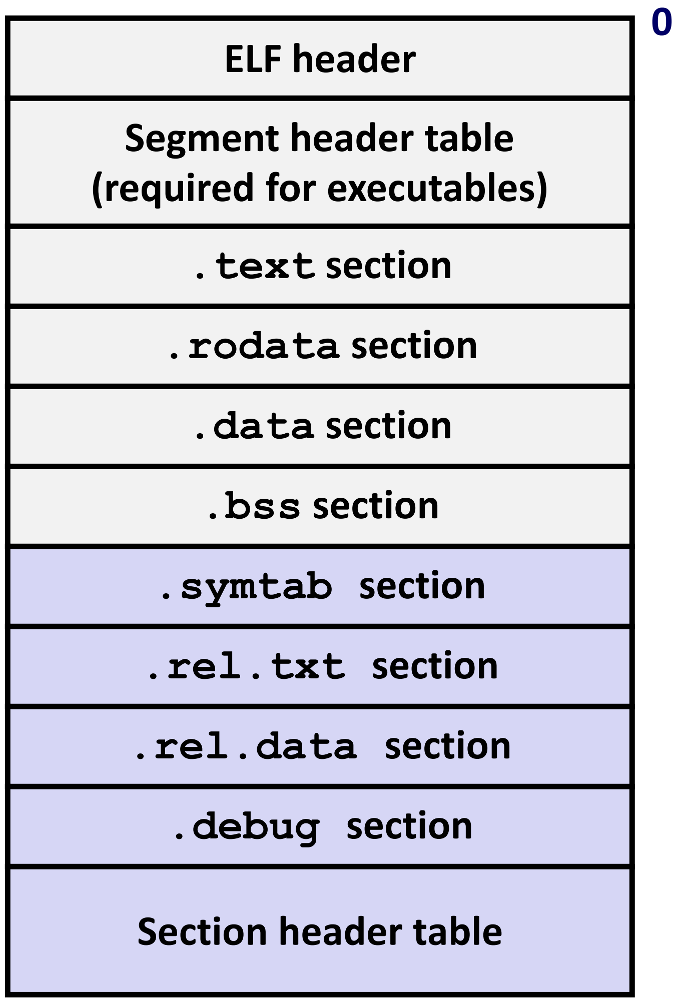
ELF目标文件格式/ELF Object File Format

- ELF头/Elf header
  - 字大小、字节顺序、文件类型、机器类型等/Word size, byte ordering, file type (.o, exec, .so), machine type, etc.
- 段头表/Segment header table
  - 页大小、虚拟地址内存段(节)、段大小/Page size, virtual addresses memory segments (sections), segment sizes.
- `.text`节/`.text` section
  - 代码/Code
- `.rodata`节/`.rodata` section
  - 只读数据：跳转表,.../Read only data: jump tables, ...
- `.data`节/`.data` section
  - 初始化的全局数据/Initialized global variables
- `.bss`节/`.bss` section
  - 未初始化的全局变量/Uninitialized global variables
  - “Block Started by Symbol”
  - “Better Save Space”
  - 不占内存空间/Has section header but occupies no space



ELF目标文件格式/ ELF Object File Format (cont.)

- **.symtab节/.symtab section**
  - 符号表/Symbol table
  - 过程和静态变量名称/Procedure and static variable names
  - 节的名字和位置/Section names and locations
- **.rel.text节/.rel.text section**
  - .text 节的重定位信息/Relocation info for .text section
  - 可执行文件中需要修改的指令的地址/Addresses of instructions that will need to be modified in the executable
  - 需要修改的指令/Instructions for modifying.
- **.rel.data节/.rel.data section**
  - .data 节的重定位信息/Relocation info for .data section
  - 在合并的可执行文件中需要修改的指针数据/Addresses of pointer data that will need to be modified in the merged executable
- **.debug节.debug section**
  - 调试的符号信息/Info for symbolic debugging (gcc -g)
- **节头表/Section header table**
  - 每个节的偏移量和大小/Offsets and sizes of each section



链接符号/Linker Symbols

- **全局符号/Global symbols**
  - 模块m中定义的可以被其他模块引用的符号/Symbols defined by module *m* that can be referenced by other modules.
  - 例如：非静态C函数以及非静态全局变量/E.g.: non-**`static`** C functions and non-**`static`** global variables.
- **外部符号/External symbols**
  - 模块m引用的但是在其他模块中定义的全局符号/Global symbols that are referenced by module *m* but defined by some other module.
- **本地符号/Local symbols**
  - 模块m定义并且只在m中引用的符号/Symbols that are defined and referenced exclusively by module *m*.
  - 例如：C函数以及使用static关键字定义的全局变量/E.g.: C functions and global variables defined with the **`static`** attribute.
  - **本地链接符号并不是局部程序变量/Local linker symbols are *not* local program variables**

第1步：符号消解/Step 1: Symbol Resolution

引用一个全局符号/Referencing a global…

在这里定义/…that's defined here

```c
int sum(int *a, int n);

int array[2] = {1, 2};

int main()
{
    int val = sum(array, 2);
    return val;
}
```

main.c

```c
int sum(int *a, int n)
{
    int i, s = 0;

    for (i = 0; i < n; i++) {
        s += a[i];
    }
    return s;
}
```

sum.c

全局符号定义/Defining a global

引用全局符号/Referencing a global…

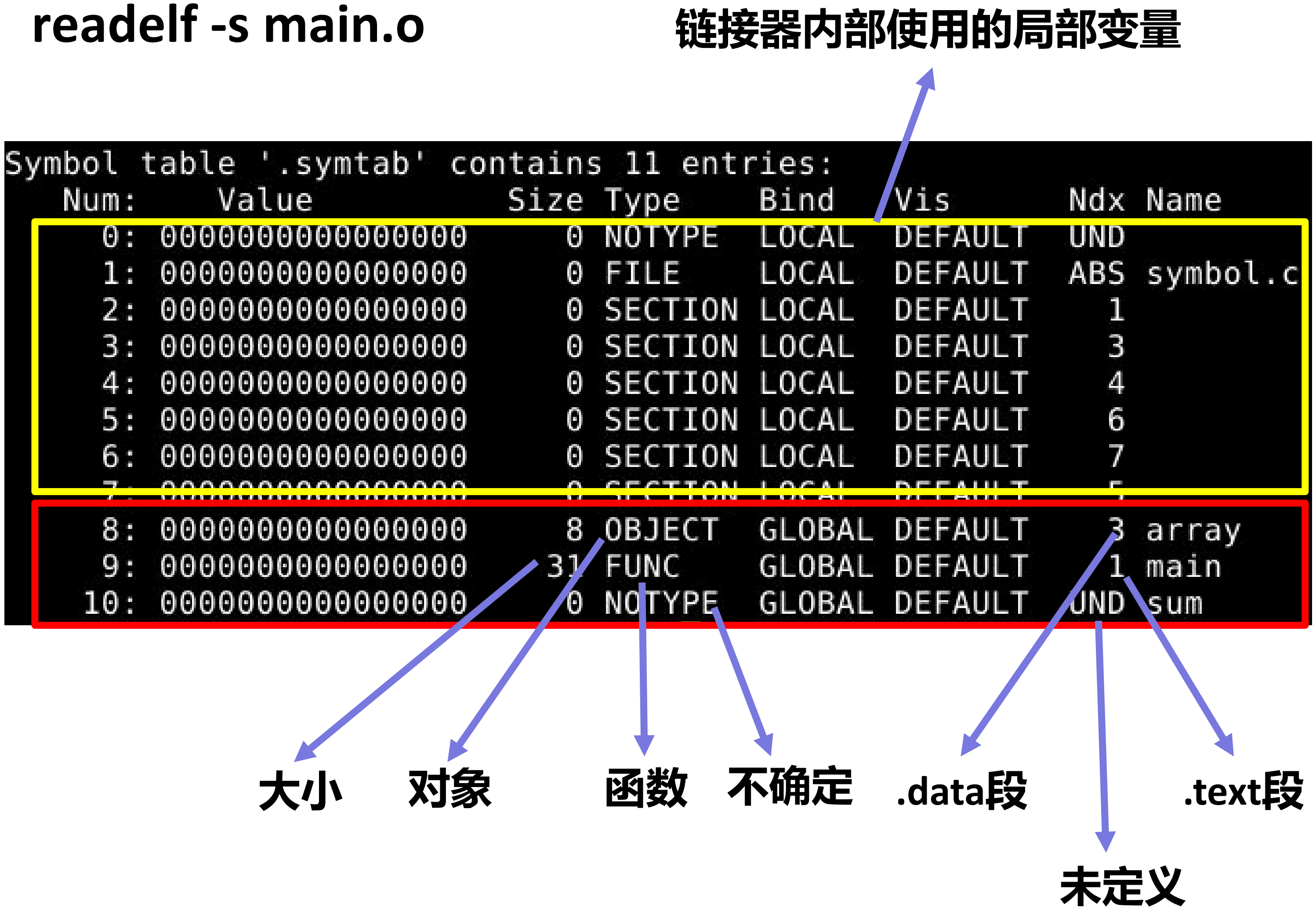
链接器并不知道i和s的存在/Linker knows nothing of `i` or `s`

链接器并不知道val的存在/Linker knows nothing of `val`

在这里定义/…that's defined here

第1步：符号消解/Step 1: Symbol Resolution

- readelf -s main.o

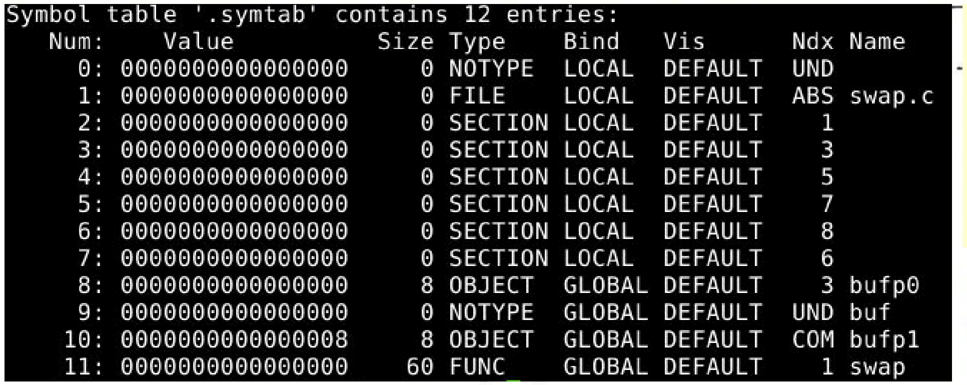


```
Symbol table '.symtab' contains 11 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS symbol.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1
     3: 0000000000000000     0 SECTION LOCAL  DEFAULT    3
     4: 0000000000000000     0 SECTION LOCAL  DEFAULT    4
     5: 0000000000000000     0 SECTION LOCAL  DEFAULT    6
     6: 0000000000000000     0 SECTION LOCAL  DEFAULT    7
     7: 0000000000000000     0 SECTION LOCAL  DEFAULT    5
     8: 0000000000000000     8 OBJECT  GLOBAL DEFAULT    3 array
     9: 0000000000000000    31 FUNC    GLOBAL DEFAULT    1 main
    10: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND sum
```

来源：武汉大学龚奕利老师

练习题 7.1 这个题目针对图 7-5 中的 m.o 和 swap.o 模块。对于每个在 swap.o 中定义或引用的符号，请指出它是否在模块 swap.o 中的 .symtab 节中有一个符号表条目。如果是，请指出定义该符号的模块(swap.o 或者 m.o)、符号类型(局部、全局或者外部)以及它在模块中被分配到的节(.text、.data、.bss 或 COMMON)。

符号	.symtab条目？	符号类型	在哪个模块中定义	节
buf	是	外部	m.o	.data
bufp0	是	全局	swap.o	.data
bufp1	是	全局	swap.o	COMMON
swap	是	全局	swap.o	.text
temp	否	—	—	—



```
Symbol table '.symtab' contains 12 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND 
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS swap.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1 
     3: 0000000000000000     0 SECTION LOCAL  DEFAULT    3 
     4: 0000000000000000     0 SECTION LOCAL  DEFAULT    5 
     5: 0000000000000000     0 SECTION LOCAL  DEFAULT    7 
     6: 0000000000000000     0 SECTION LOCAL  DEFAULT    8 
     7: 0000000000000000     0 SECTION LOCAL  DEFAULT    6 
     8: 0000000000000000     8 OBJECT  GLOBAL DEFAULT    3 bufp0
     9: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND buf
    10: 0000000000000008     8 OBJECT  GLOBAL DEFAULT  COM bufp1
    11: 0000000000000000    60 FUNC    GLOBAL DEFAULT    1 swap
```

gcc中：
COMMON： 未初始化的全局变量
.bss：未初始化的静态变量，以及初始化为0的全局或静态变量

```
        return 0;
    }
```

code/link/m.c

```
    void swap()
    {
        int temp;

        bufp1 = &buf[1];
        temp = *bufp0;
        *bufp0 = *bufp1;
        *bufp1 = temp;
    }
```

code/link/swap.c

a) m.c　　　　b) swap.c

图 7-5 练习题 7.1 的示例程序

来源：武汉大学龚奕利老师

本地符号/Local Symbols

- **本地非静态C变量和本地静态C变量/Local non-static C variables vs. local static C variables**
  - 本地非静态C变量：存储在栈上/local non-static C variables: stored on the stack
  - 本地静态C变量：存储在.bss或者.data区域/local static C variables: stored in either `.bss`, or `.data`

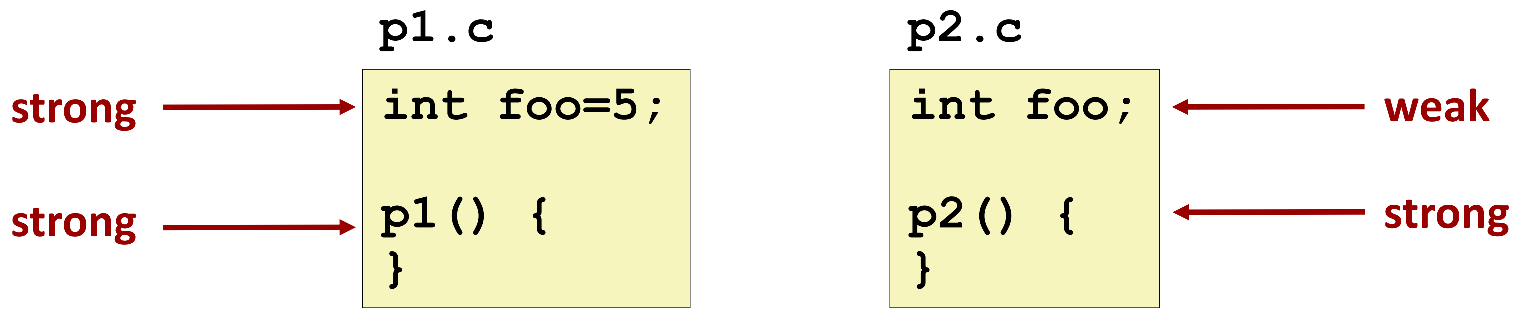
```
int f()
{
    static int x = 0;
    return x;
}

int g()
{
    static int x = 1;
    return x;
}
```

编译器为每个定义的x在.data节分配空间/Compiler allocates space in `.data` for each definition of `x`

在符号表中创建唯一的符号名字，例如x.1和x.2等/Creates local symbols in the symbol table with unique names, e.g., `x.1` and `x.2`.

链接器怎么消解重复的符号定义/How Linker Resolves Duplicate Symbol Definitions

- **程序中的符号分为强和弱两类/Program symbols are either *strong* or *weak***
  - ***强符号/Strong***: 过程和初始化的全局变量/procedures and initialized globals
  - ***弱符号/Weak***: 未初始化的全局符号/uninitialized globals



p1.c

```
int foo=5;

p1() {
}
```

p2.c

```
int foo;

p2() {
}
```

链接器符号消解规则/Linker’s Symbol Rules

- **规则1：不允许有多个强符号/Rule 1: Multiple strong symbols are not allowed**
  - 每个符号只允许被定义一次/Each item can be defined only once
  - 否则会出现链接错误/Otherwise: Linker error

- **规则2：当有一个强符号和多个弱符号时，选择强符号/Rule 2: Given a strong symbol and multiple weak symbols, choose the strong symbol**
  - 对弱符号的引用改为对强符号的引用/References to the weak symbol resolve to the strong symbol

- **规则3：当有多个弱符号时，选择其中任意一个/Rule 3: If there are multiple weak symbols, pick an arbitrary one**
  - 可以通过`gcc -fno-common`指定/Can override this with `gcc -fno-common`

链接谜题/Linker Puzzles

```
int x;
p1() {}
```

```
p1() {}
```

链接错误：两个强符号（p1）
Link time error: two strong symbols (`p1`)

```
int x;
p1() {}
```

```
int x;
p2() {}
```

对x的引用会指向同一个未初始化的int，这是程序员的本意吗？/References to `x` will refer to the same uninitialized int. Is this what you really want?

```
int x;
int y;
p1() {}
```

```
double x;
p2() {}
```

p2中对x的写入操作甚至可能覆盖y/Writes to `x` in `p2` might overwrite `y`!

```
int x=7;
int y=5;
p1() {}
```

```
double x;
p2() {}
```

p2中的对x的写操作会覆盖y/Writes to `x` in `p2` will overwrite `y`!

```
int x=7;
p1() {}
```

```
int x;
p2() {}
```

对x的引用将指向同一个初始化的变量/References to `x` will refer to the same initialized variable.

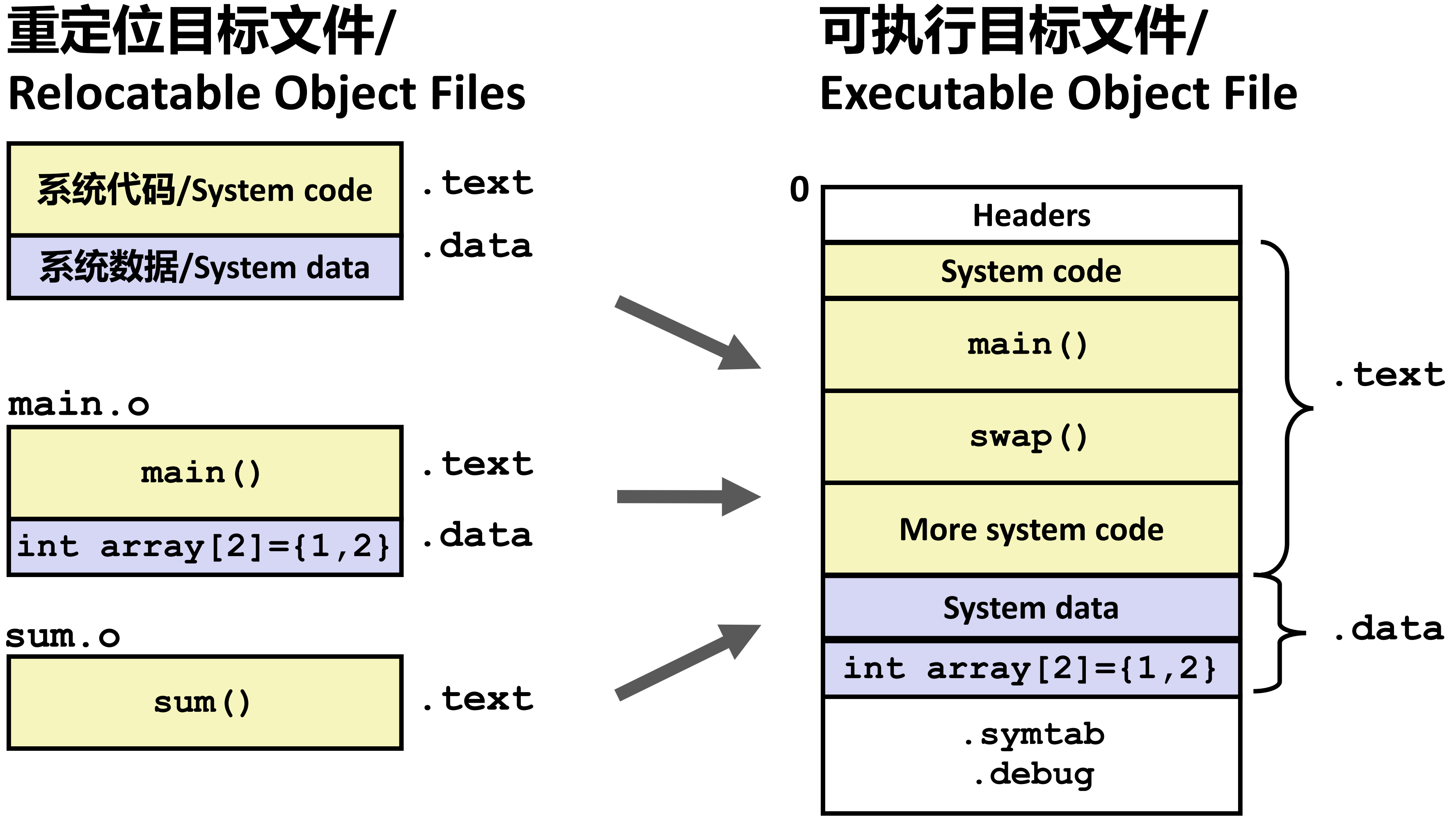
噩梦场景：两个一样的弱结构使用两个不同编译器编译会出现不一样的适用规则/Nightmare scenario: two identical weak structs, compiled by different compilers with different alignment rules.

全局变量/Global Variables

- **尽可能避免/Avoid if you can**

- **否则/Otherwise**
  - 尽可能使用static/Use **`static`** if you can
  - 定义全局变量时初始化/Initialize if you define a global variable
  - 使用extern关键字如果引用的是一个外部的全局变量/Use **`extern`** if you reference an external global variable

第2步：重定位/Step 2: Relocation



重定位条目/Relocation Entries

```
int array[2] = {1, 2};

int main()
{
    int val = sum(array, 2);
    return val;
}
                                main.c
```

```
0000000000000000 <main>:
   0:    48 83 ec 08              sub    $0x8,%rsp
   4:    be 02 00 00 00           mov    $0x2,%esi
   9:    bf 00 00 00 00           mov    $0x0,%edi       # %edi = &array
                        a: R_X86_64_32 array           # Relocation entry

   e:    e8 00 00 00 00           callq  13 <main+0x13> # sum()
                        f: R_X86_64_PC32 sum-0x4       # Relocation entry
  13:    48 83 c4 08              add    $0x8,%rsp
  17:    c3                       retq
                                                                    main.o
```

Source: `objdump -r -d main.o`

重定位后的.text节/Relocated .text section

```
00000000004004d0 <main>:
  4004d0:       48 83 ec 08          sub    $0x8,%rsp
  4004d4:       be 02 00 00 00       mov    $0x2,%esi
  4004d9:       bf 18 10 60 00       mov    $0x601018,%edi   # %edi = &array
  4004de:       e8 05 00 00 00       callq  4004e8 <sum>     # sum()
  4004e3:       48 83 c4 08          add    $0x8,%rsp
  4004e7:       c3                   retq

00000000004004e8 <sum>:
  4004e8:       b8 00 00 00 00             mov    $0x0,%eax
  4004ed:       ba 00 00 00 00             mov    $0x0,%edx
  4004f2:       eb 09                      jmp    4004fd <sum+0x15>
  4004f4:       48 63 ca                   movslq %edx,%rcx
  4004f7:       03 04 8f                   add    (%rdi,%rcx,4),%eax
  4004fa:       83 c2 01                   add    $0x1,%edx
  4004fd:       39 f2                      cmp    %esi,%edx
  4004ff:       7c f3                      jl     4004f4 <sum+0xc>
  400501:       f3 c3                      repz retq
```

Using PC-relative addressing for sum(): 0x4004e8 = 0x4004e3 + 0x5

Source: objdump -dx prog

重定位条目

```c
int array[2] =

int main()
{
    int val =
    return val
}
```

code/link/elfstructs.c

```c
typedef struct {
    long offset;      /* Offset of the reference to relocate */
    long type:32,     /* Relocation type */
         symbol:32;   /* Symbol table index */
    long addend;      /* Constant part of relocation expression */
} Elf64_Rela;
```

code/link/elfstructs.c

Figure 7.9 **ELF relocation entry.** Each entry identifies a reference that must be relocated and specifies how to compute the modified reference.

```
0000000000000000 <main>:
   0:   48 83 ec 08            sub    $0x8,%rsp
   4:   be 02 00 00 00         mov    $0x2,%esi
   9:   bf 00 00 00 00         mov    $0x0,%edi         # %edi = &array
                        a: R_X86_64_32 array              # Relocation entry

   e:   e8 00 00 00 00         callq  13 <main+0x13>    # sum()
                        f: R_X86_64_PC32 sum-0x4          # Relocation entry
  13:   48 83 c4 08            add    $0x8,%rsp
  17:   c3                     retq
```

待重定位引用的位置

32位绝对地址的引用

32位PC相对地址的引用

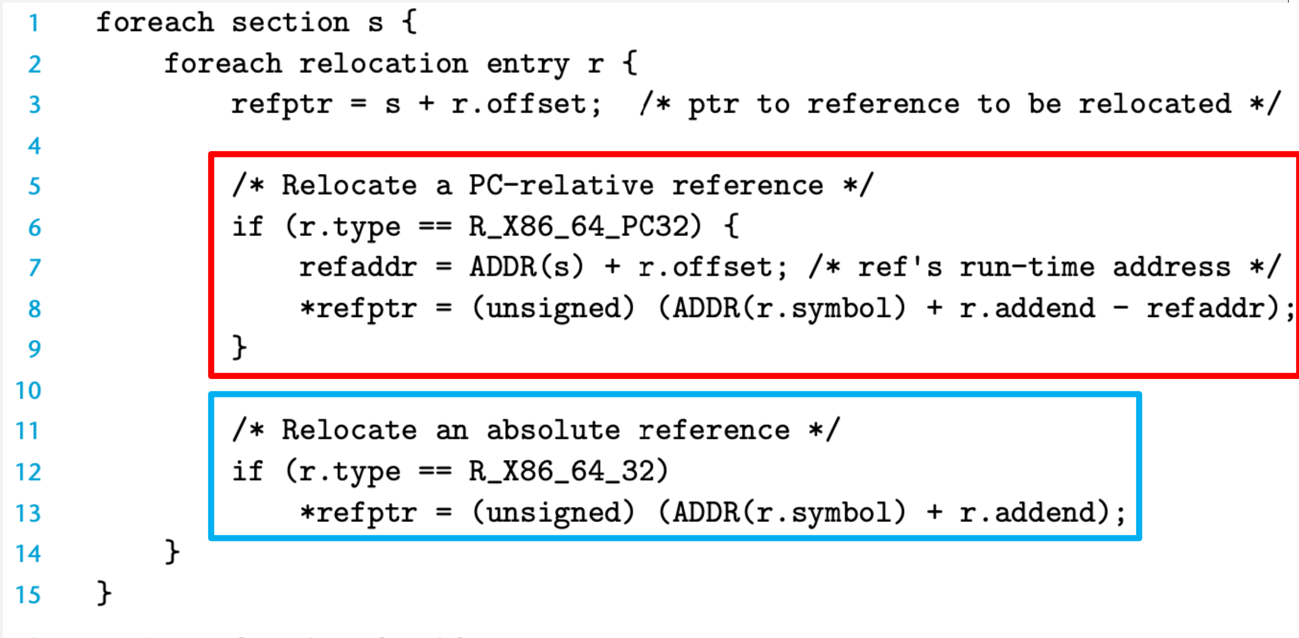
main.o

来源：武汉大学龚奕利老师

Source: `objdump -r -d main.o`

重定位后的.text节

```
00000000004004d0 <main>:
  4004d0:       48 83 ec 08          sub    $0x8,%rsp
  4004d4:       be 02 00 00 00       mov    $0x2,%esi
  4004d9:       bf 18 10 60 00       mov    $0x601018,%edi   # %edi = &array
  4004de:       e8 05 00 00 00       callq  4004e8 <sum>     # sum()
  4004e3:       48 83 c4 08
  4004e7:       c3

00000000004004e8 <sum>:
  4004e8:       b8 00 00 00
  4004ed:       ba 00 00 00
  4004f2:       eb 09
  4004f4:       48 63 ca
  4004f7:       03 04 8f
  4004fa:       83 c2 01
  4004fd:       39 f2
  4004ff:       7c f3
  400501:       f3 c3
```



```
foreach section s {
    foreach relocation entry r {
        refptr = s + r.offset;  /* ptr to reference to be relocated */

        /* Relocate a PC-relative reference */
        if (r.type == R_X86_64_PC32) {
            refaddr = ADDR(s) + r.offset; /* ref's run-time address */
            *refptr = (unsigned) (ADDR(r.symbol) + r.addend - refaddr);
        }

        /* Relocate an absolute reference */
        if (r.type == R_X86_64_32)
            *refptr = (unsigned) (ADDR(r.symbol) + r.addend);
    }
}
```

Figure 7.10 Relocation algorithm.

sum(0x4004e8) – 4 – PC(0x4004df) = 0x5

对sum()使用PC相对寻址： 0x4004e8 = 0x4004e3 + 0x5

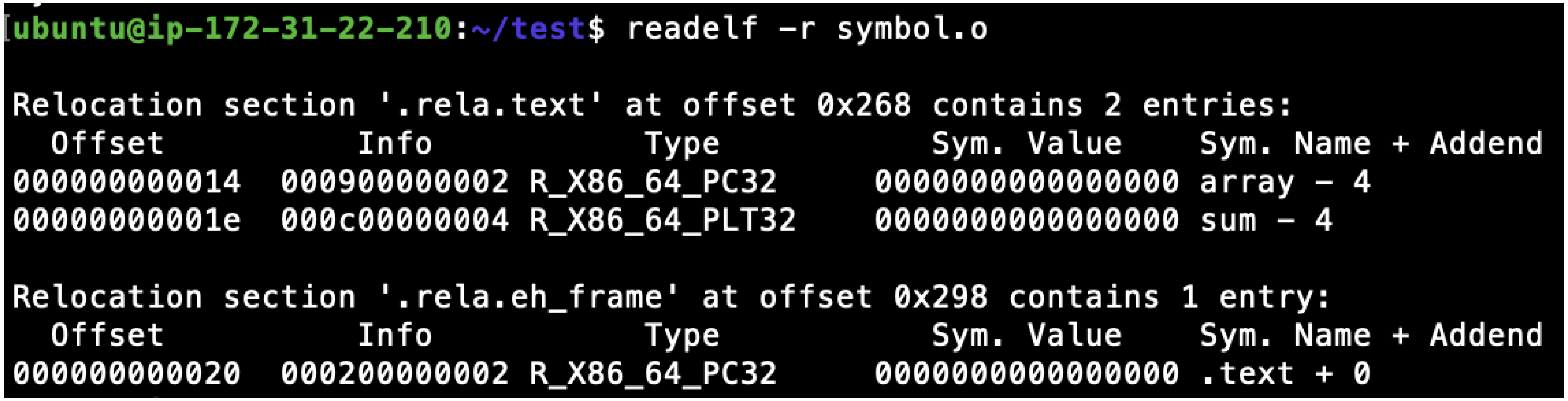
来源：武汉大学龚奕利老师　　**来源： `objdump -dx prog`** *code/link/elfstructs.c*

```
typedef struct {
    long offset;                     rence to relocate */
    long type:32,                    '
         symbol:32;                  : */
    long addend;                     elocation expression */
} Elf64_Rela;
```

```
0xf
R_X86_64_PC32
sum
-4
```

code/link/elfstructs.c

Figure 7.9 ELF relocation entry. Each entry identifies a reference that must be relocated and specifies how to compute the modified reference.



```
ubuntu@ip-172-31-22-210:~/test$ readelf -r symbol.o

Relocation section '.rela.text' at offset 0x268 contains 2 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000000014  000900000002 R_X86_64_PC32     0000000000000000 array - 4
00000000001e  000c00000004 R_X86_64_PLT32    0000000000000000 sum - 4

Relocation section '.rela.eh_frame' at offset 0x298 contains 1 entry:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000000020  000200000002 R_X86_64_PC32     0000000000000000 .text + 0
```

来源：武汉大学龚奕利老师

code/link/elfstructs.c

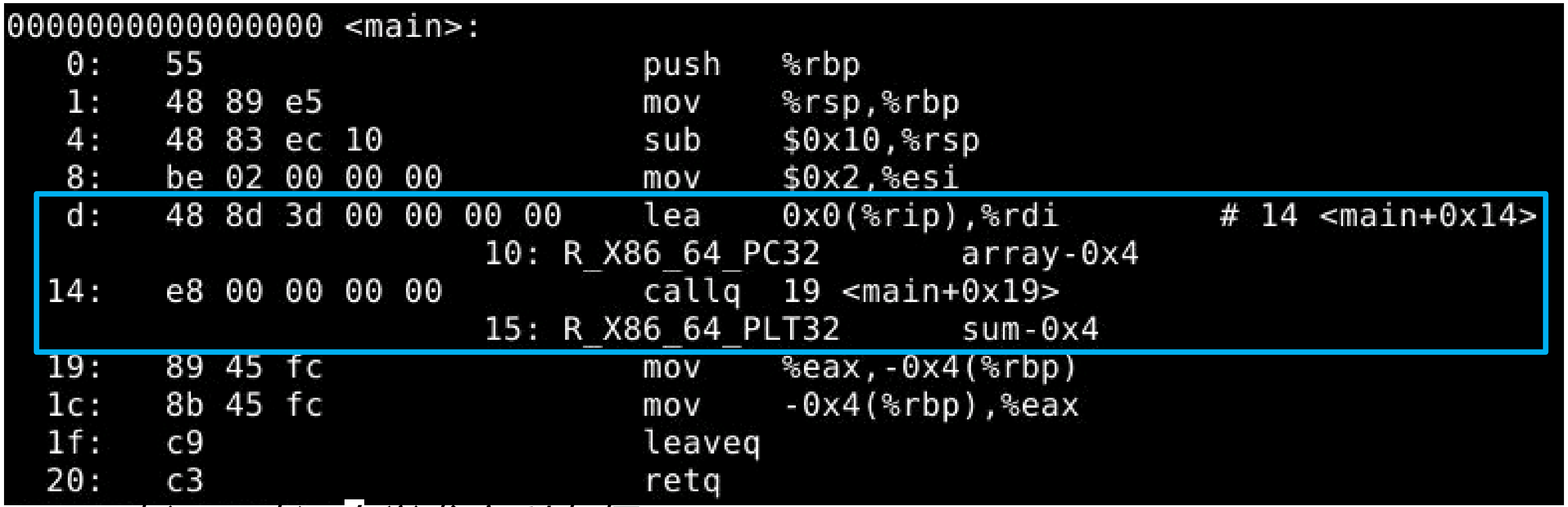
```
typedef struct {
    long offset;                          rence to relocate */
    long type:32,                         /
         symbol:32;                       : */
    long addend;                          relocation expression */
} Elf64_Rela;
```

```
0xf
R_X86_64_PC32
sum
-4
```

code/link/elfstructs.c

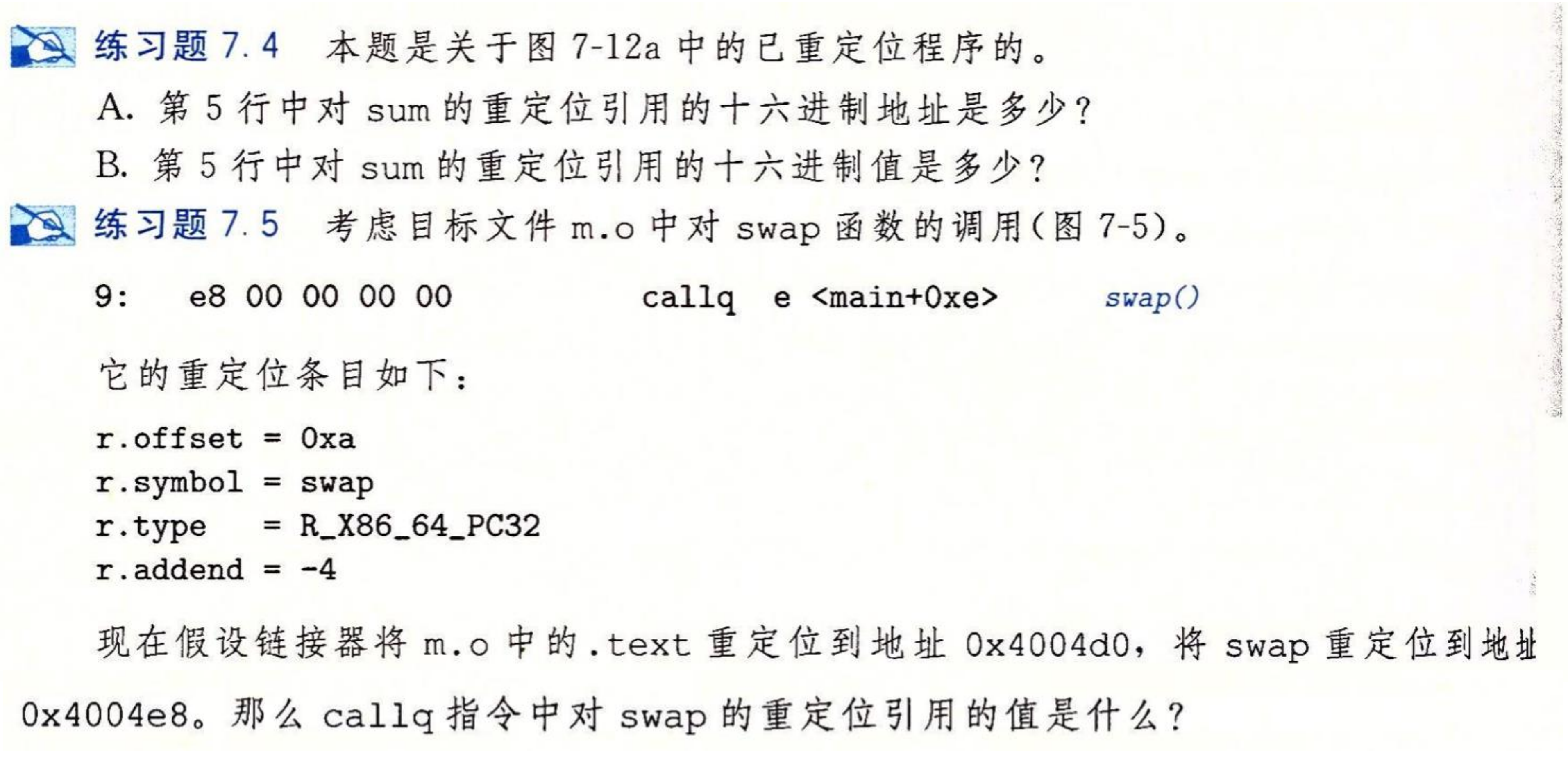
Figure 7.9 **ELF relocation entry.** Each entry identifies a reference that must be relocated and specifies how to compute the modified reference.

linux> objdump –dx main.o



```
0000000000000000 <main>:
   0:	55                   	push   %rbp
   1:	48 89 e5             	mov    %rsp,%rbp
   4:	48 83 ec 10          	sub    $0x10,%rsp
   8:	be 02 00 00 00       	mov    $0x2,%esi
   d:	48 8d 3d 00 00 00 00 	lea    0x0(%rip),%rdi        # 14 <main+0x14>
			10: R_X86_64_PC32	array-0x4
  14:	e8 00 00 00 00       	callq  19 <main+0x19>
			15: R_X86_64_PLT32	sum-0x4
  19:	89 45 fc             	mov    %eax,-0x4(%rbp)
  1c:	8b 45 fc             	mov    -0x4(%rbp),%eax
  1f:	c9                   	leaveq
  20:	c3                   	retq
```

来源：武汉大学龚奕利老师



练习题 7.4 本题是关于图 7-12a 中的已重定位程序的。

A. 第 5 行中对 sum 的重定位引用的十六进制地址是多少？

B. 第 5 行中对 sum 的重定位引用的十六进制值是多少？

练习题 7.5 考虑目标文件 m.o 中对 swap 函数的调用(图 7-5)。

```
9:    e8 00 00 00 00          callq  e <main+0xe>       swap()
```

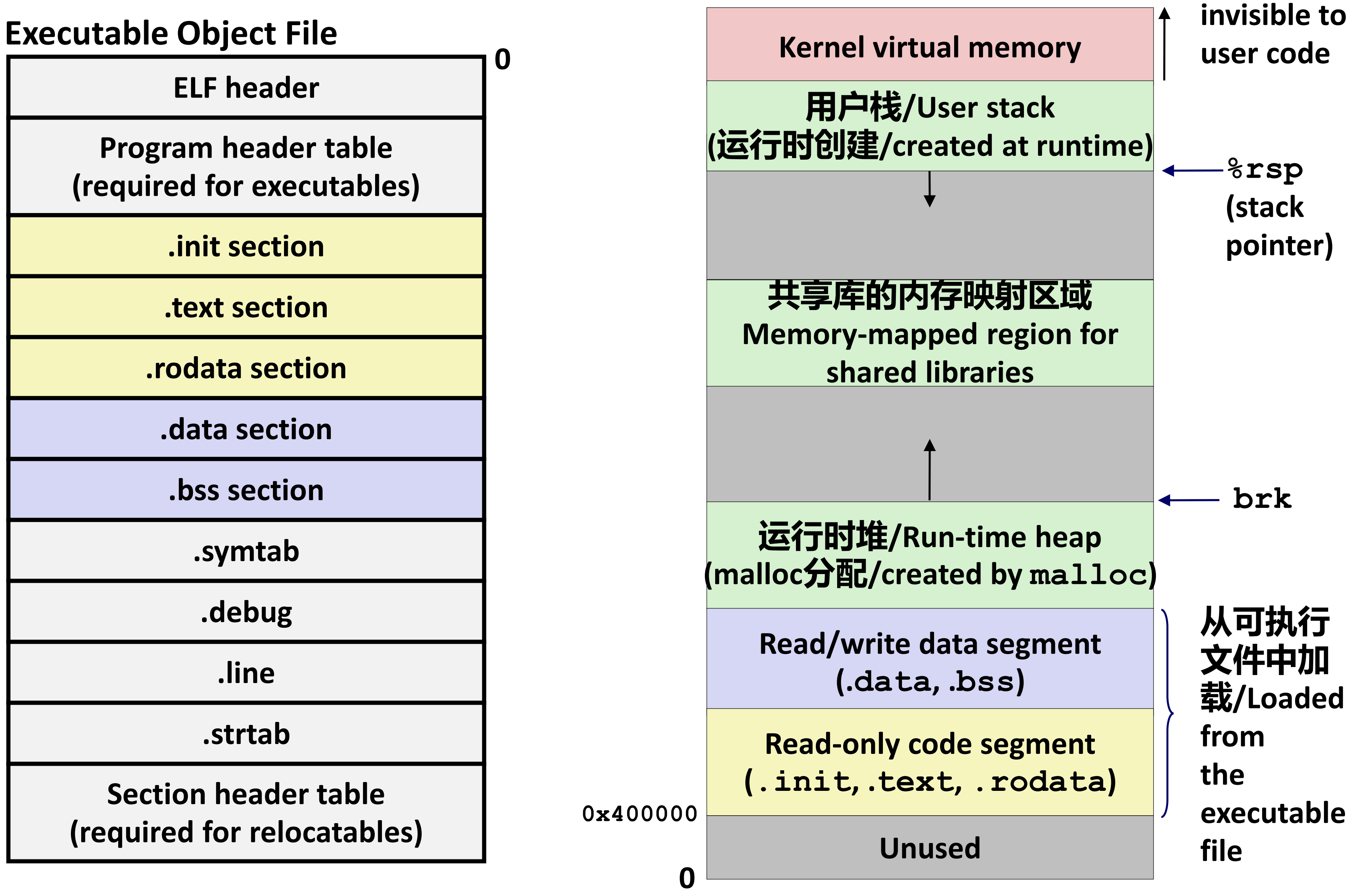
它的重定位条目如下：

```
r.offset = 0xa
r.symbol = swap
r.type   = R_X86_64_PC32
r.addend = -4
```

现在假设链接器将 m.o 中的 .text 重定位到地址 0x4004d0，将 swap 重定位到地址 0x4004e8。那么 callq 指令中对 swap 的重定位引用的值是什么？

来源：武汉大学龚奕利老师

加载可执行目标文件/Loading Executable Object Files



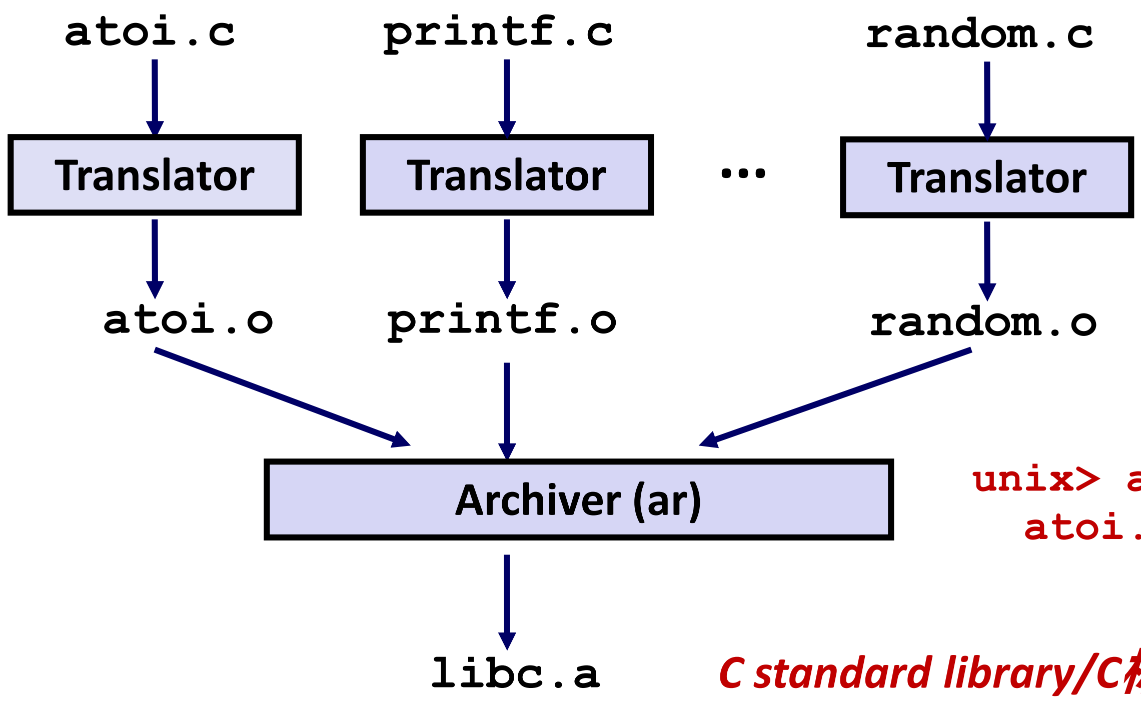
打包公共函数/Packaging Commonly Used Functions

- **如何对程序员共用的函数打包？/How to package functions commonly used by programmers?**
  - 数学、I/O、内存管理、字符串操作等/Math, I/O, memory management, string manipulation, etc.
- **基于目前的框架比较尴尬/Awkward, given the linker framework so far:**
  - **选择1:/Option 1:** 将所有的函数放在单个源码文件中/Put all functions into a single source file
    - 程序员将整个目标文件链接到他们的程序中/Programmers link big object file into their programs
    - 空间和时间上都比较低效/Space and time inefficient
  - **选择2:/Option 2:** 将每个函数放在单独的文件中/Put each function in a separate source file
    - 程序员显式将需要的函数链接到他们的程序中/Programmers explicitly link appropriate binaries into their programs
    - 更加高效，但是程序员的负担较重/More efficient, but burdensome on the programmer

较早的解决方案：静态链接库/Old-fashioned Solution: Static Libraries

- **静态链接库/Static libraries (.a archive files)**
  - 将相关的重定位目标文件按照索引放入单个的文件中（称为档案）/Concatenate related relocatable object files into a single file with an index (called an *archive*).
  - 链接器在不同的档案中查找未消解的符号/Enhance linker so that it tries to resolve unresolved external references by looking for the symbols in one or more archives.
  - 如果在某个档案中找到了引用的符号，则将这个文件链接到可执行文件中/If an archive member file resolves reference, link it into the executable.

创建静态链接库/Creating Static Libraries



```
unix> ar rs libc.a \
  atoi.o printf.o … random.o
```

C standard library/C标准库

- 允许增量更新/Archiver allows incremental updates
- 重新编译修改的文件并替换档案中的.o文件/Recompile function that changes and replace .o file in archive.

常用的静态链接库/Commonly Used Libraries

`libc.a` (C语言标准库/the C standard library)

- 4.6MB的档案中包含了1496个目标文件/4.6 MB archive of 1496 object files.
- I/O、内存分配、信号处理、字符串处理、日期和时间、随机数、整数数学函数/ I/O, memory allocation, signal handling, string handling, data and time, random numbers, integer math

`libm.a` (C语言数学库/the C math library)

- 2MB的档案中包含了444个目标文件/2 MB archive of 444 object files.
- 浮点数学函数/floating point math (sin, cos, tan, log, exp, sqrt, …)

```
% ar -t libc.a | sort
…
fork.o
…
fprintf.o
fpu_control.o
fputc.o
freopen.o
fscanf.o
fseek.o
fstab.o
…
```

```
% ar -t libm.a | sort
…
e_acos.o
e_acosf.o
e_acosh.o
e_acoshf.o
e_acoshl.o
e_acosl.o
e_asin.o
e_asinf.o
e_asinl.o
…
```

链接静态链接库/Linking with Static Libraries

libvector.a

```
#include <stdio.h>
#include "vector.h"

int x[2] = {1, 2};
int y[2] = {3, 4};
int z[2];

int main()
{
    addvec(x, y, z, 2);
    printf("z = [%d %d]\n",
           z[0], z[1]);
    return 0;
}
```

main2.c

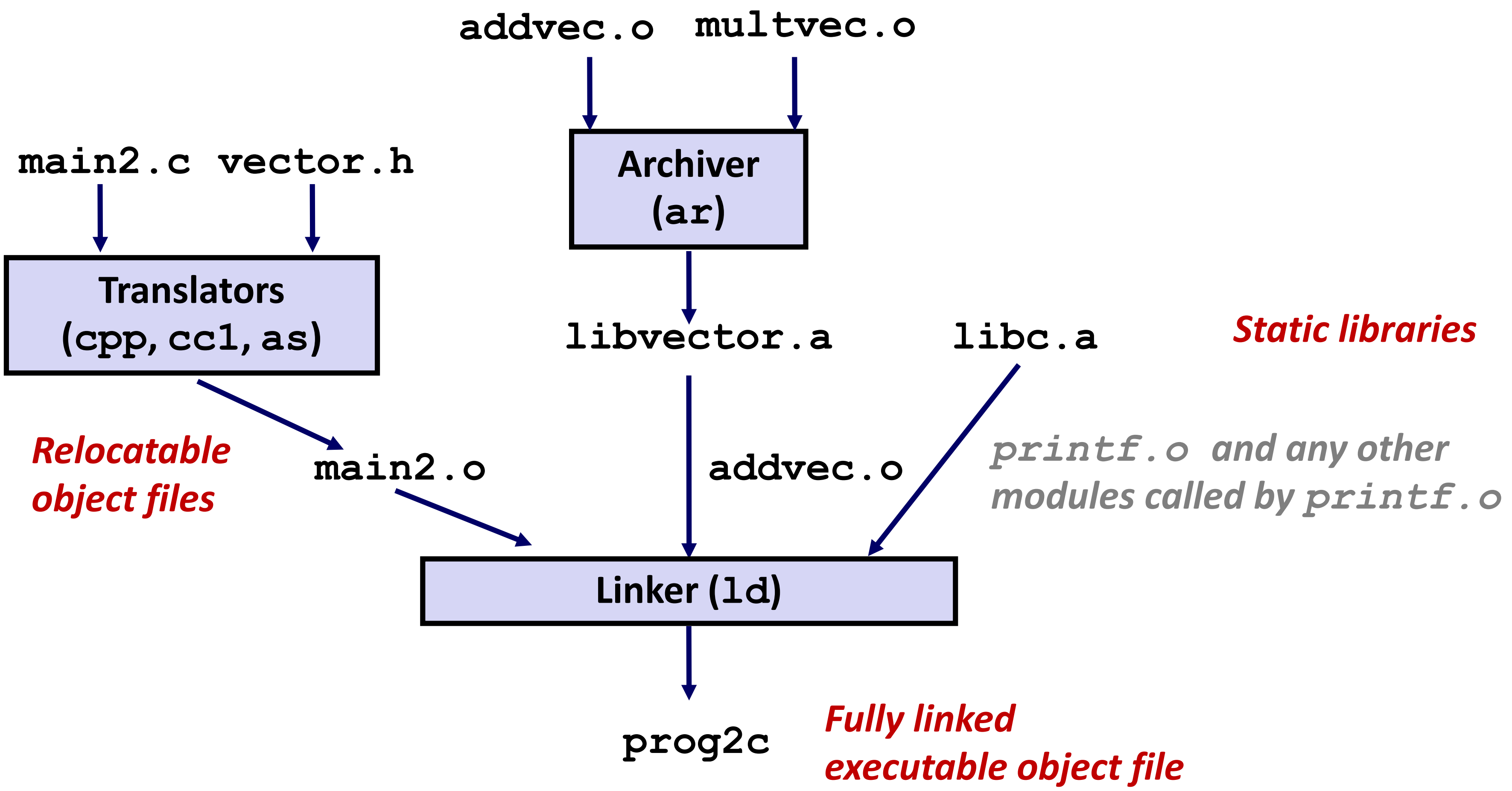
```
void addvec(int *x, int *y,
            int *z, int n) {
    int i;

    for (i = 0; i < n; i++)
        z[i] = x[i] + y[i];
}
```

addvec.c

```
void multvec(int *x, int *y,
             int *z, int n)
{
    int i;

    for (i = 0; i < n; i++)
        z[i] = x[i] * y[i];
}
```

multvec.c

链接静态链接库/Linking with Static Libraries



"c" for "compile-time"

使用静态链接库/Using Static Libraries

- **链接器中消解外部引用的算法/Linker's algorithm for resolving external references:**
  - 按照命令行顺序扫描.o文件和.a文件/Scan `.o` files and `.a` files in the command line order.
  - 在扫描的过程中维护一个未消解的引用/During the scan, keep a list of the current unresolved references.
  - 当每遇到新的.o和.a文件，尝试使用这些目标文件中定义的符号对应那些未消解的符号/As each new `.o` or `.a` file, *obj*, is encountered, try to resolve each unresolved reference in the list against the symbols defined in *obj*.
  - 如果扫描结束还有未消解的符号则报错/If any entries in the unresolved list at end of scan, then error.
- **问题/Problem:**
  - 命令行中的文件顺序很重要/Command line order matters!
  - 启示：将库放在命令行的末尾/Moral: put libraries at the end of the command line.

```
unix> gcc -L. libtest.o -lmine
unix> gcc -L. -lmine libtest.o
libtest.o: In function `main':
libtest.o(.text+0x4): undefined reference to `libfun'
```

当前的解决方案：共享库/Modern Solution: Shared Libraries

- **静态库有如下缺点/Static libraries have the following disadvantages:**
  - 可执行文件中重复包含和存储/Duplication in the stored executables (每个函数都需要libc/every function needs libc)
  - 运行可执行文件存在冗余/Duplication in the running executables
  - 系统库中的bug修复需要每个应用重新链接/Minor bug fixes of system libraries require each application to explicitly relink

- **当前的方案：共享库/Modern solution: Shared Libraries**
  - 包含代码和数据的目标文件是在加载过程中或者运行时动态加载和链接的/Object files that contain code and data that are loaded and linked into an application *dynamically,* at either *load-time* or *run-time*
  - 又称为动态链接库，DLL和.so等/Also called: dynamic link libraries, DLLs, `.so` files

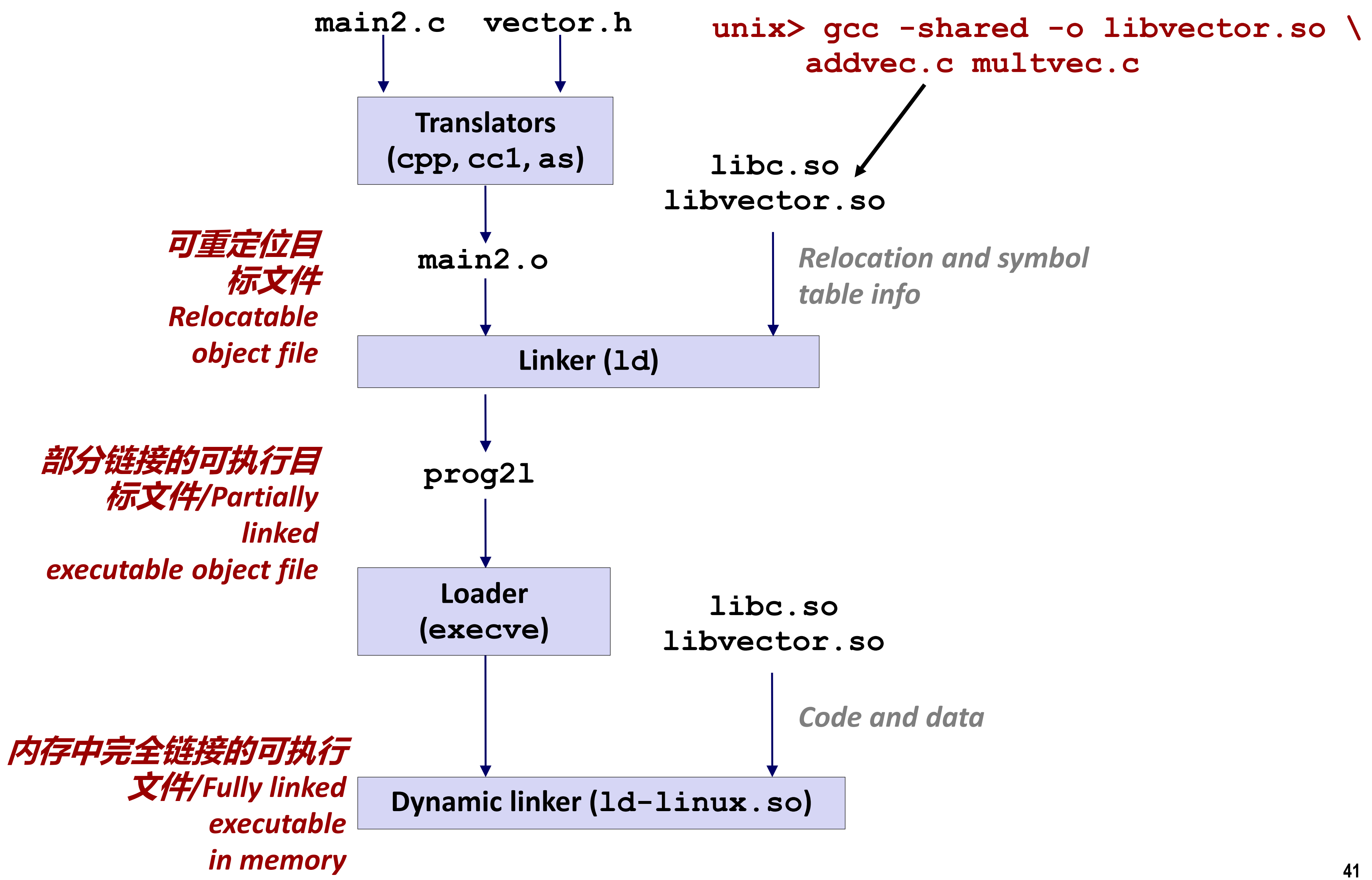
共享库/Shared Libraries (cont.)

- **当可执行文件第一次加载和运行时进行动态链接/Dynamic linking can occur when executable is first loaded and run (load-time linking).**
  - 在Linux系统中是由动态链接器ld-linux.so自动完成的/Common case for Linux, handled automatically by the dynamic linker (**`ld-linux.so`**）.
  - C的标准库通常是动态链接的/Standard C library (**`libc.so`**) usually dynamically linked.

- **程序开始执行后也可以进行动态链接/Dynamic linking can also occur after program has begun (运行时链接/run-time linking).**
  - Linux中是通过调用dlopen()接口完成的/In Linux, this is done by calls to the **`dlopen()`** interface.
    - 分发软件/Distributing software.
    - 高性能web服务器/High-performance web servers.
    - 运行时库打桩/Runtime library interpositioning.

- **共享库可以被多个进程共享/Shared library routines can be shared by multiple processes.**
  - 虚拟内存部分会详细介绍/More on this when we learn about virtual memory

加载时进行动态链接/Dynamic Linking at Load-time


main2.c
vector.h
unix> gcc -shared -o libvector.so \
addvec.c multvec.c
Translators
(cpp, cc1, as)
libc.so
libvector.so
可重定位目标文件
Relocatable object file
main2.o
Relocation and symbol table info
Linker (ld)
部分链接的可执行目标文件/Partially linked executable object file
prog21
Loader
(execve)
libc.so
libvector.so
Code and data
内存中完全链接的可执行文件/Fully linked executable in memory
Dynamic linker (ld-linux.so)

运行时动态链接/Dynamic Linking at Run-time

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

int x[2] = {1, 2};
int y[2] = {3, 4};
int z[2];

int main()
{
    void *handle;
    void (*addvec)(int *, int *, int *, int);
    char *error;

    /* Dynamically load the shared library that contains addvec() */
    handle = dlopen("./libvector.so", RTLD_LAZY);
    if (!handle) {
        fprintf(stderr, "%s\n", dlerror());
        exit(1);
    }
```

dll.c

运行时动态链接/Dynamic Linking at Run-time

```c
    ...

    /* Get a pointer to the addvec() function we just loaded */
    addvec = dlsym(handle, "addvec");
    if ((error = dlerror()) != NULL) {
        fprintf(stderr, "%s\n", error);
        exit(1);
    }

    /* Now we can call addvec() just like any other function */
    addvec(x, y, z, 2);
    printf("z = [%d %d]\n", z[0], z[1]);

    /* Unload the shared library */
    if (dlclose(handle) < 0) {
        fprintf(stderr, "%s\n", dlerror());
        exit(1);
    }
    return 0;
}
```

dll.c

链接总结/Linking Summary

- **链接是一种允许使用多个目标文件构建程序的技术/Linking is a technique that allows programs to be constructed from multiple object files.**

- **链接可以在程序声明周期的不同时间进行/Linking can happen at different times in a program’s lifetime:**
  - 编译时（当程序被编译）Compile time (when a program is compiled)
  - 加载时（当程序被装载进内存时）Load time (when a program is loaded into memory)
  - 运行时（当程序执行时）Run time (while a program is executing)

- **理解链接可以避免讨厌的错误，让你成为一名更好的程序员。Understanding linking can help you avoid nasty errors and make you a better programmer.**

提纲/Today

- 链接/Linking
- 实例分析：库打桩/Case study: Library interpositioning

实例分析：库打桩/Case Study: Library Interpositioning

- **库打桩：一种强大的链接技术，能够让程序员截获任意函数调用/Library interpositioning : powerful linking technique that allows programmers to intercept calls to arbitrary functions**
- **打桩可以发生在/Interpositioning can occur at:**
  - 编译时：当源代码被编译时/Compile time: When the source code is compiled
  - 链接时：当可重定位目标文件被静态链接形成可执行目标文件时/Link time: When the relocatable object files are statically linked to form an executable object file
  - 加载/运行时：当可执行目标文件被装载进内存，动态链接并执行时/Load/run time: When an executable object file is loaded into memory, dynamically linked, and then executed.

打桩应用/Some Interpositioning Applications

- **安全/Security**
  - 限制（沙箱）Confinement (sandboxing)
  - 幕后加密/Behind the scenes encryption
- **调试/Debugging**
  - 2014年，两名Facebook工程师使用打桩技术，在iPhone应用程序中调试了一个潜伏了一年的bug/In 2014, two Facebook engineers debugged a treacherous 1-year old bug in their iPhone app using interpositioning
  - SPDY网络栈中的代码被写入错误的位置/Code in the SPDY networking stack was writing to the wrong location
  - 通过拦截对Posix写入函数（write、writev、pwrite）的调用来解决/Solved by intercepting calls to Posix write functions (write, writev, pwrite)

Source: Facebook engineering blog post at
`https://code.facebook.com/posts/313033472212144/debugging-file-corruption-on-ios/`

打桩应用/Some Interpositioning Applications

- **监控和profiling/Monitoring and Profiling**
  - 统计函数调用次数/Count number of calls to functions
  - 分析函数调用和参数/Characterize call sites and arguments to functions
  - 内存分配追踪/Malloc tracing
    - 发现内存泄露/Detecting memory leaks
    - **生成地址序列/Generating address traces**

例子/Example program

```c
#include <stdio.h>
#include <malloc.h>

int main()
{
    int *p = malloc(32);
    free(p);
    return(0);
}
```
int.c

- 目标：在不干扰程序和修改源码的情况下，记录分配和释放的内存地址和大小/Goal: trace the addresses and sizes of the allocated and freed blocks, without breaking the program, and without modifying the source code.

- 三个方案：在编译、链接、装载/运行时打桩/Three solutions: interpose on the `lib` `malloc` and `free` functions at compile time, link time, and load/run time.

编译时打桩/Compile-time Interpositioning

```c
#ifdef COMPILETIME
#include <stdio.h>
#include <malloc.h>

/* malloc wrapper function */
void *mymalloc(size_t size)
{
    void *ptr = malloc(size);
    printf("malloc(%d)=%p\n",
           (int)size, ptr);
    return ptr;
}

/* free wrapper function */
void myfree(void *ptr)
{
    free(ptr);
    printf("free(%p)\n", ptr);
}
#endif
```

mymalloc.c

编译时打桩/Compile-time Interpositioning

```
#define malloc(size) mymalloc(size)
#define free(ptr) myfree(ptr)

void *mymalloc(size_t size);
void myfree(void *ptr);
                                                          malloc.h
```

```
linux> make intc
gcc -Wall -DCOMPILETIME -c mymalloc.c
gcc -Wall -I. -o intc int.c mymalloc.o
linux> make runc
./intc
malloc(32)=0x1edc010
free(0x1edc010)
linux>
```

链接时打桩/Link-time Interpositioning

```
#ifdef LINKTIME
#include <stdio.h>

void *__real_malloc(size_t size);
void __real_free(void *ptr);

/* malloc wrapper function */
void *__wrap_malloc(size_t size)
{
    void *ptr = __real_malloc(size); /* Call libc malloc */
    printf("malloc(%d) = %p\n", (int)size, ptr);
    return ptr;
}

/* free wrapper function */
void __wrap_free(void *ptr)
{
    __real_free(ptr); /* Call libc free */
    printf("free(%p)\n", ptr);
}
#endif
```

mymalloc.c

链接时打桩/ Link-time Interpositioning

```
linux> make intl
gcc -Wall -DLINKTIME -c mymalloc.c
gcc -Wall -c int.c
gcc -Wall -Wl,--wrap,malloc -Wl,--wrap,free -o intl
int.o mymalloc.o
linux> make runl
./intl
malloc(32) = 0x1aa0010
free(0x1aa0010)
linux>
```

- **"-W1"将参数传递给链接器，将每个,替换为空白/The "`-Wl`" flag passes argument to linker, replacing each comma with a space.**
- **参数"`--wrap,malloc`" 告知链接器对符号消解进行特殊处理/The "`--wrap,malloc`" arg instructs linker to resolve references in a special way:**
  - Refs to `malloc` should be resolved as `__wrap_malloc`
  - Refs to `__real_malloc` should be resolved as `malloc`

Load/Run-time Interpositioning

```
#ifdef RUNTIME
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>

/* malloc wrapper function */
void *malloc(size_t size)
{
    void *(*mallocp)(size_t size);
    char *error;

    mallocp = dlsym(RTLD_NEXT, "malloc"); /* Get addr of libc malloc */
    if ((error = dlerror()) != NULL) {
        fputs(error, stderr);
        exit(1);
    }
    char *ptr = mallocp(size); /* Call libc malloc */
    printf("malloc(%d) = %p\n", (int)size, ptr);
    return ptr;
}
```

mymalloc.c

装载/运行时打桩/Load/Run-time Interpositioning

```
/* free wrapper function */
void free(void *ptr)
{
    void (*freep)(void *) = NULL;
    char *error;

    if (!ptr)
        return;

    freep = dlsym(RTLD_NEXT, "free"); /* Get address of libc free */
    if ((error = dlerror()) != NULL) {
        fputs(error, stderr);
        exit(1);
    }
    freep(ptr); /* Call libc free */
    printf("free(%p)\n", ptr);
}
#endif
```

mymalloc.c

装载/运行时打桩/Load/Run-time Interpositioning

```
linux> make intr
gcc -Wall -DRUNTIME -shared -fpic -o mymalloc.so mymalloc.c -ldl
gcc -Wall -o intr int.c
linux> make runr
(LD_PRELOAD="./mymalloc.so" ./intr)
malloc(32) = 0xe60010
free(0xe60010)
linux>
```

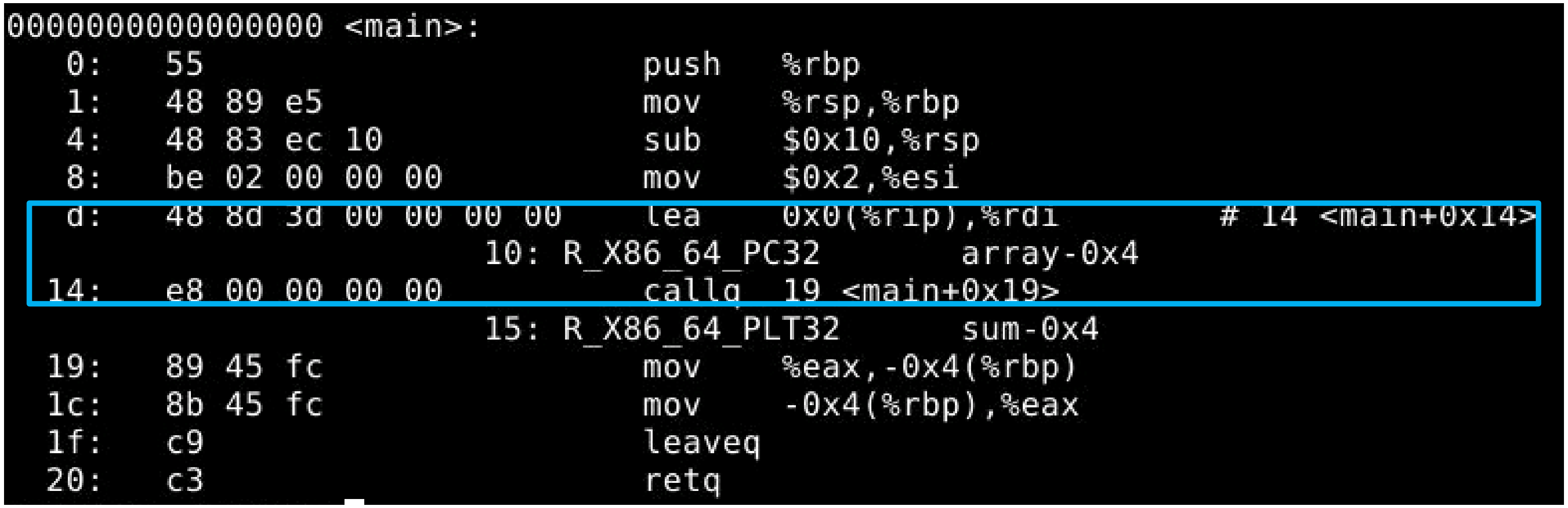
- LD_PRELOAD环境变量指示动态链接器优先在mymalloc.so中查找未消解的符号/The LD_PRELOAD environment variable tells the dynamic linker to resolve unresolved refs (e.g., to malloc) by looking in mymalloc.so first.

打桩回顾/Interpositioning Recap

- **编译时/Compile Time**
  - Apparent calls to malloc/free get macro-expanded into calls to mymalloc/myfree
- **链接时/Link Time**
  - Use linker trick to have special name resolutions
    - malloc → __wrap_malloc
    - __real_malloc → malloc
- **加载/运行时/Load/Run Time**
  - Implement custom version of malloc/free that use dynamic linking to load library malloc/free under different names

7.12 位置无关代码
(Position-Independent Code, PIC)

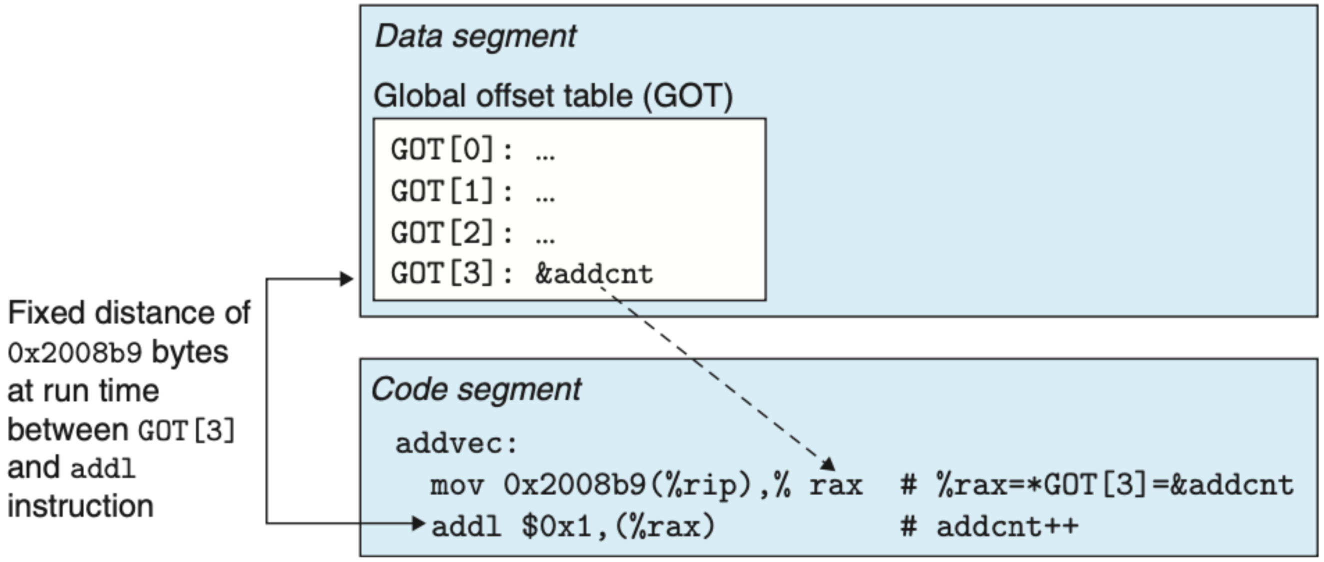
- **位置无关代码**
  - `gcc -fpic`
  - 可以把共享模块的代码段加载到内存的任何位置，而无需链接器做任何修改
- **同一个目标模块中符号的引用**
  - PC相对寻址
  - 构造目标文件时由静态链接器重定位



```
0000000000000000 <main>:
   0:   55                      push   %rbp
   1:   48 89 e5                mov    %rsp,%rbp
   4:   48 83 ec 10             sub    $0x10,%rsp
   8:   be 02 00 00 00          mov    $0x2,%esi
   d:   48 8d 3d 00 00 00 00    lea    0x0(%rip),%rdi        # 14 <main+0x14>
                        10: R_X86_64_PC32       array-0x4
  14:   e8 00 00 00 00          callq  19 <main+0x19>
                        15: R_X86_64_PLT32      sum-0x4
  19:   89 45 fc                mov    %eax,-0x4(%rbp)
  1c:   8b 45 fc                mov    -0x4(%rbp),%eax
  1f:   c9                      leaveq
  20:   c3                      retq
```

PIC数据引用

- **Fact: 无论我们在内存中的何处加载一个目标模块（包括共享目标模块），数据段与代码段的距离总是保持不变的，因而代码段中任何指令和数据段中任何变量之间的距离都是一个运行时常量**
- **全局偏移量表 Global Offset Table, GOT**
  - 每个被目标模块引用的全局数据目标都有一个8字节entry
  - 链接器在构造这个模块时解析它



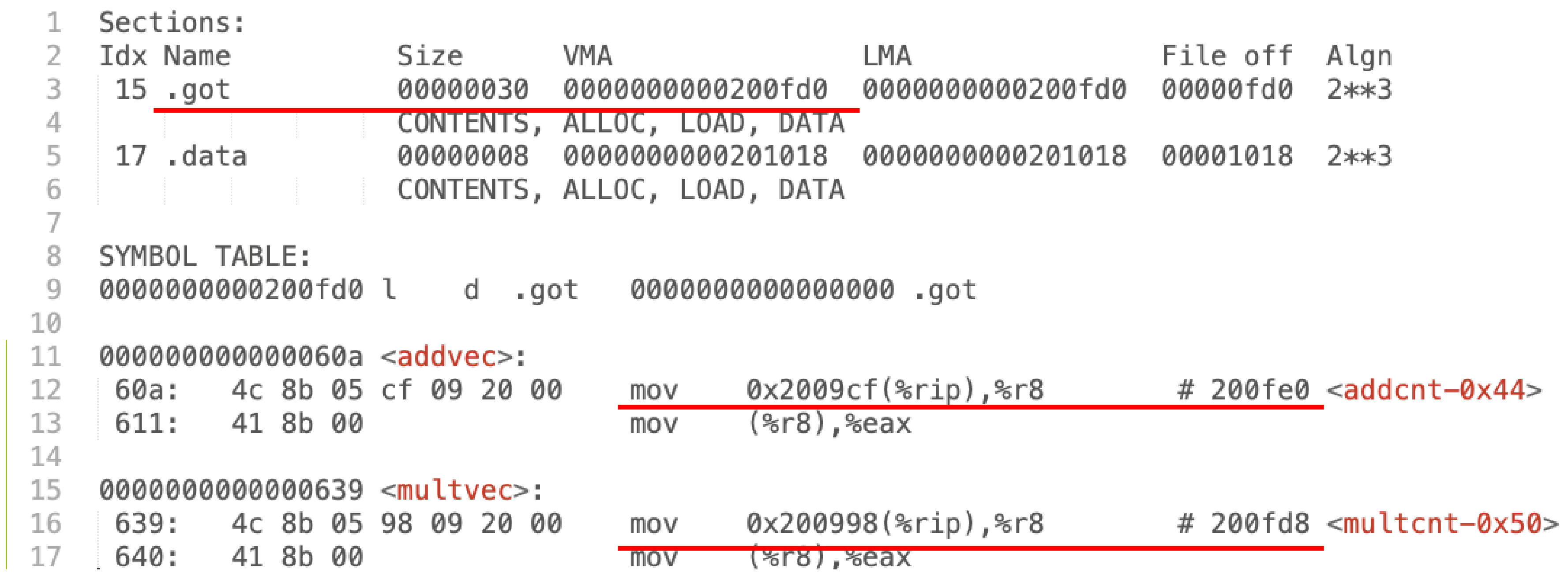
Data segment
Global offset table (GOT)
GOT[0]: …
GOT[1]: …
GOT[2]: …
GOT[3]: &addcnt
Fixed distance of 0x2008b9 bytes at run time between GOT[3] and addl instruction
Code segment

```
addvec:
   mov  0x2008b9(%rip),% rax   # %rax=*GOT[3]=&addcnt
   addl  $0x1,(%rax)           # addcnt++
```

Figure 7.18 Using the GOT to reference a global variable. The addvec routine in libvector.so references addcnt indirectly through the GOT for libvector.so.

PIC数据引用

- 使用 objdump -dx libvector.so 得到部分汇编指令

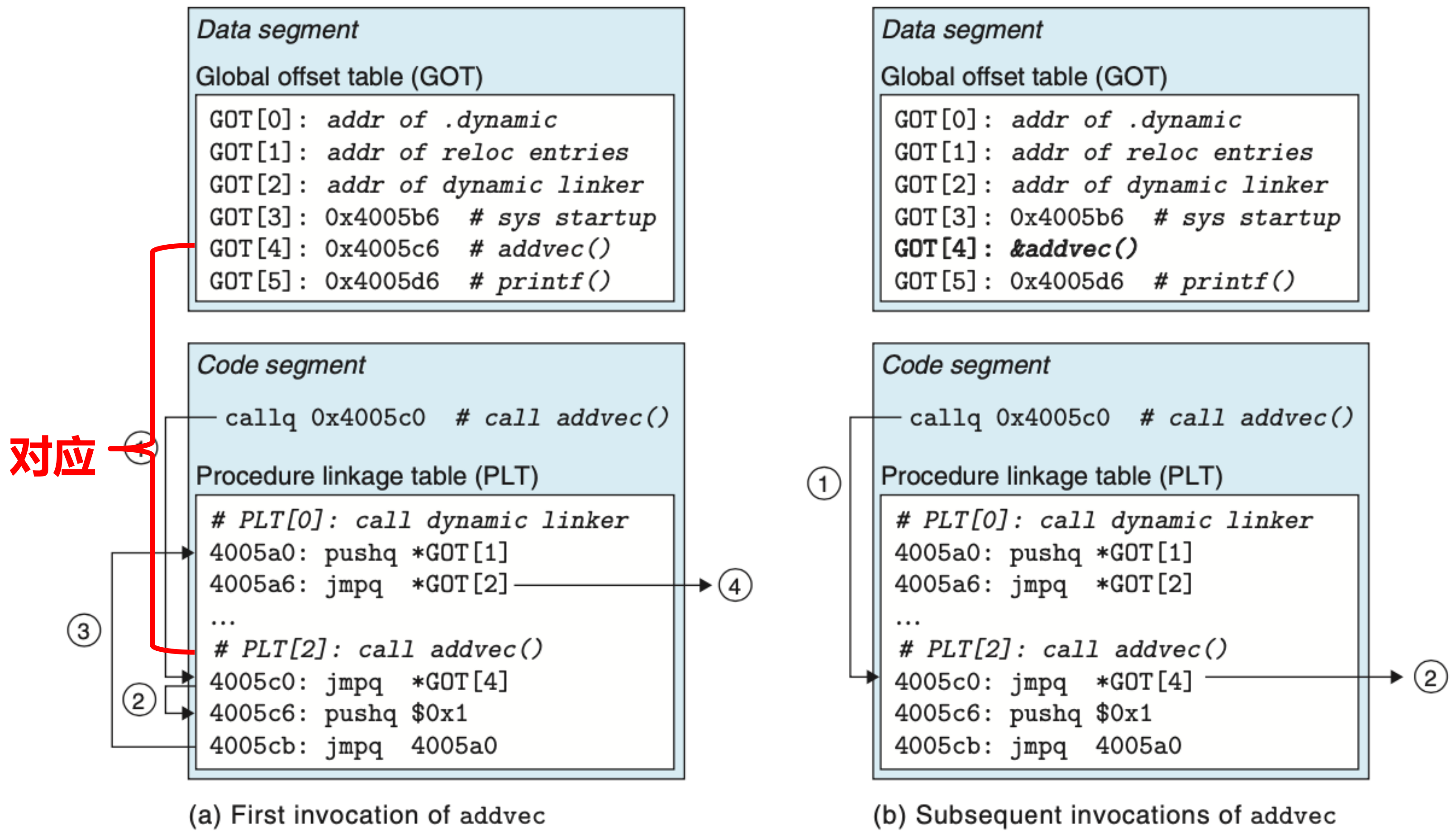


```
Sections:
Idx Name          Size      VMA               LMA               File off  Algn
 15 .got          00000030  0000000000200fd0  0000000000200fd0  00000fd0  2**3
                  CONTENTS, ALLOC, LOAD, DATA
 17 .data         00000008  0000000000201018  0000000000201018  00001018  2**3
                  CONTENTS, ALLOC, LOAD, DATA

SYMBOL TABLE:
0000000000200fd0 l    d  .got   0000000000000000 .got

000000000000060a <addvec>:
 60a:   4c 8b 05 cf 09 20 00    mov    0x2009cf(%rip),%r8        # 200fe0 <addcnt-0x44>
 611:   41 8b 00                mov    (%r8),%eax

0000000000000639 <multvec>:
 639:   4c 8b 05 98 09 20 00    mov    0x200998(%rip),%r8        # 200fd8 <multcnt-0x50>
 640:   41 8b 00                mov    (%r8),%eax
```

PIC函数调用

- 一种方法：为该引用生产一条重定位记录，然后动态链接器在程序加载时再解析它。这不是PIC的，需要链接器修改调用模块的代码段。
- PIC的方法：延迟绑定 lazy binding
  - 第一次调用时加载，其后的每次调用开销为一条指令和一个内存间接引用
  - 借助于GOT和过程链接表 Procedure Linkage Table PLT
  - GOT：每个条目8字节地址；GOT[0]和GOT[1]是动态链接器解析函数地址时需要的信息； GOT[2]为动态链接器的入口地址；其余条目对应一个被调用的函数，对应一个PLT条目。
  - PLT ：每个条目16字节代码；PLT[0]跳转到动态链接器中；每个被调用函数都有一个条目

PIC函数调用



Data segment

Global offset table (GOT)

```
GOT[0]: addr of .dynamic
GOT[1]: addr of reloc entries
GOT[2]: addr of dynamic linker
GOT[3]: 0x4005b6  # sys startup
GOT[4]: 0x4005c6  # addvec()
GOT[5]: 0x4005d6  # printf()
```

Code segment

```
callq 0x4005c0  # call addvec()
```

Procedure linkage table (PLT)

```
# PLT[0]: call dynamic linker
4005a0: pushq *GOT[1]
4005a6: jmpq  *GOT[2]
...
# PLT[2]: call addvec()
4005c0: jmpq  *GOT[4]
4005c6: pushq $0x1
4005cb: jmpq  4005a0
```

(a) First invocation of addvec

Data segment

Global offset table (GOT)

```
GOT[0]: addr of .dynamic
GOT[1]: addr of reloc entries
GOT[2]: addr of dynamic linker
GOT[3]: 0x4005b6  # sys startup
GOT[4]: &addvec()
GOT[5]: 0x4005d6  # printf()
```

Code segment

```
callq 0x4005c0  # call addvec()
```

Procedure linkage table (PLT)

```
# PLT[0]: call dynamic linker
4005a0: pushq *GOT[1]
4005a6: jmpq  *GOT[2]
...
# PLT[2]: call addvec()
4005c0: jmpq  *GOT[4]
4005c6: pushq $0x1
4005cb: jmpq  4005a0
```

(b) Subsequent invocations of addvec

Figure 7.19 **Using the PLT and GOT to call external functions.** The dynamic linker resolves the address of addvec the first time it is called.

PIC函数调用

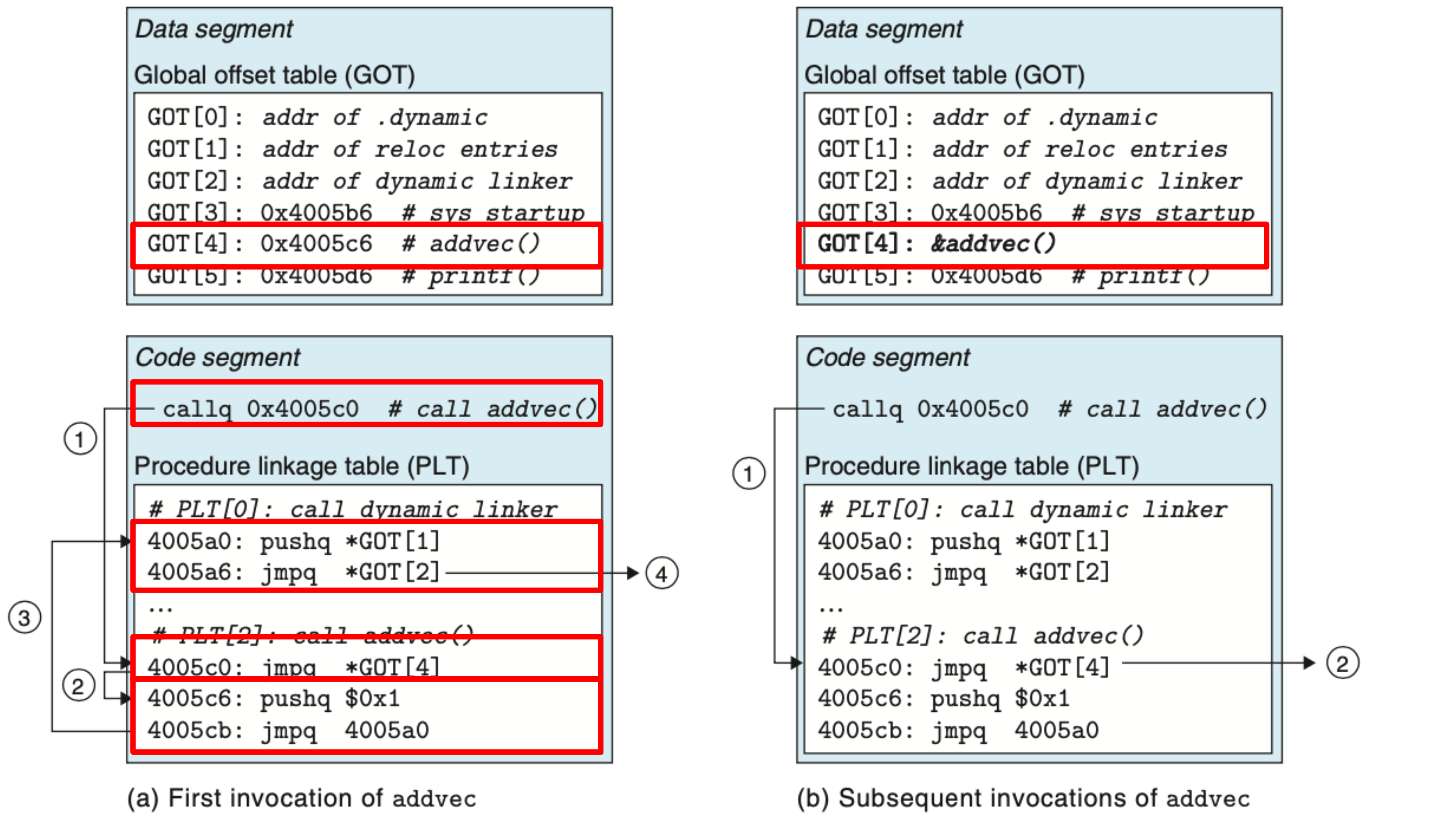
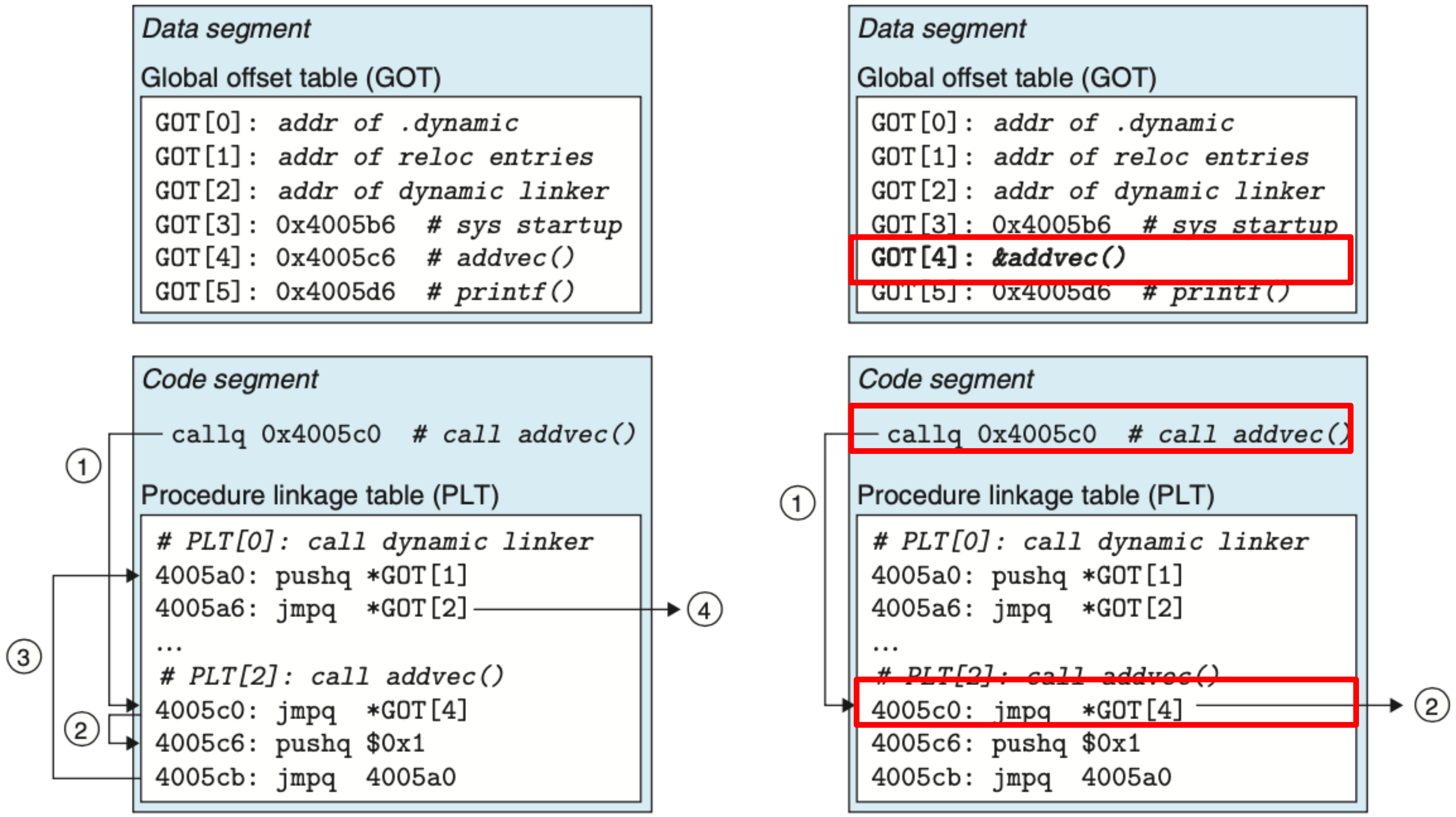

Data segment
Global offset table (GOT)
GOT[0]: addr of .dynamic
GOT[1]: addr of reloc entries
GOT[2]: addr of dynamic linker
GOT[3]: 0x4005b6 # sys startup
GOT[4]: 0x4005c6 # addvec()
GOT[5]: 0x4005d6 # printf()
Code segment
callq 0x4005c0 # call addvec()
Procedure linkage table (PLT)
# PLT[0]: call dynamic linker
4005a0: pushq *GOT[1]
4005a6: jmpq *GOT[2]
...
# PLT[2]: call addvec()
4005c0: jmpq *GOT[4]
4005c6: pushq $0x1
4005cb: jmpq 4005a0
①
②
③
④
(a) First invocation of addvec
Data segment
Global offset table (GOT)
GOT[0]: addr of .dynamic
GOT[1]: addr of reloc entries
GOT[2]: addr of dynamic linker
GOT[3]: 0x4005b6 # sys startup
GOT[4]: &addvec()
GOT[5]: 0x4005d6 # printf()
Code segment
callq 0x4005c0 # call addvec()
Procedure linkage table (PLT)
# PLT[0]: call dynamic linker
4005a0: pushq *GOT[1]
4005a6: jmpq *GOT[2]
...
# PLT[2]: call addvec()
4005c0: jmpq *GOT[4]
4005c6: pushq $0x1
4005cb: jmpq 4005a0
①
②
(b) Subsequent invocations of addvec

Figure 7.19 Using the PLT and GOT to call external functions. The dynamic linker resolves the address of addvec the first time it is called.

PIC函数调用



```
GOT[0]: addr of .dynamic
GOT[1]: addr of reloc entries
GOT[2]: addr of dynamic linker
GOT[3]: 0x4005b6  # sys startup
GOT[4]: 0x4005c6  # addvec()
GOT[5]: 0x4005d6  # printf()

callq 0x4005c0  # call addvec()

# PLT[0]: call dynamic linker
4005a0: pushq *GOT[1]
4005a6: jmpq  *GOT[2]
...
# PLT[2]: call addvec()
4005c0: jmpq  *GOT[4]
4005c6: pushq $0x1
4005cb: jmpq  4005a0
```

(a) First invocation of addvec

```
GOT[0]: addr of .dynamic
GOT[1]: addr of reloc entries
GOT[2]: addr of dynamic linker
GOT[3]: 0x4005b6  # sys startup
GOT[4]: &addvec()
GOT[5]: 0x4005d6  # printf()

callq 0x4005c0  # call addvec()

# PLT[0]: call dynamic linker
4005a0: pushq *GOT[1]
4005a6: jmpq  *GOT[2]
...
# PLT[2]: call addvec()
4005c0: jmpq  *GOT[4]
4005c6: pushq $0x1
4005cb: jmpq  4005a0
```

(b) Subsequent invocations of addvec

Figure 7.19 Using the PLT and GOT to call external functions. The dynamic linker resolves the address of addvec the first time it is called.