第7章 链接 Link

100076202： 计算机系统导论

任课教师：

宿红毅　张艳　黎有琦　颜珂

原作者：

Randal E. Bryant and David R. O'Hallaron

提纲

- **链接 Linking**
  - 动机 Motivation
  - 完成的工作 What it does
  - 如何工作 How it works
  - 动态链接 Dynamic linking
- **案例研究：库打桩 Case study: Library interpositioning**

示例C语言程序 Example C Program

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

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

int main(int argc, char** argv)
{
    int val = sum(array, 2);
    return val;
}
```

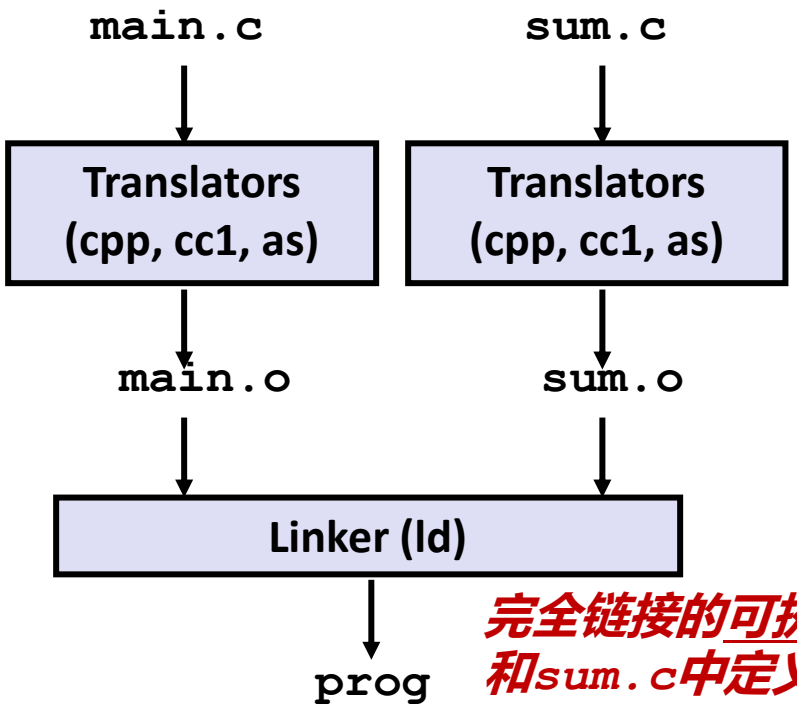
main.c

```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 Linking

- **用编译器来翻译和链接程序：** Programs are translated and linked using a *compiler driver*:
  - `linux> gcc -Og -o prog main.c sum.c`
  - `linux> ./prog`



源文件 Source files

独立编译的
可重定位目标文件
Separately compiled
relocatable object files

完全链接的可执行目标文件（包含`main.c`和`sum.c`中定义的所有函数的代码和数据）
Fully linked executable object file
(contains code and data for all functions defined in `main.c` and `sum.c`)

为什么要用链接器？ Why Linkers?

- **理由1：模块化 Reason 1: Modularity**
  - 将程序组织成若干较小的源文件，而不是一个整体 Program can be written as a collection of smaller source files, rather than one monolithic mass.
  - 可以将常用的函数构造成库（后面会更多地讲到） 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

为什么要用链接器?(续)Why Linkers? (cont)

- **理由2：高效性 Reason 2: Efficiency**
  - 时间：分别编译 Time: Separate compilation
    - 修改一个源文件，编译并重链接 Change one source file, compile, and then relink.
    - 没必要重新编译其他的源文件 No need to recompile other source files.
    - 可以并发编译多个文件 Can compile multiple files concurrently.
  - 空间：库 Space: Libraries
    - 可以将常用函数聚合成单一文件… Common functions can be aggregated into a single file...
    - **选项1：静态链接 Option 1: *Static Linking***
      - 可执行文件和运行时内存镜像中只包含它们实际使用的函数的代码 Executable files and running memory images contain only the library code they actually use
    - **选项2：动态链接 Option 2: *Dynamic linking***
      - 执行文件不包含库代码 Executable files contain no library code
      - 执行期间单一的库代码拷贝可以给所有执行中的进程共享 During execution, single copy of library code can be shared across all

链接器做些什么？ What Do Linkers Do?

- **第一步：符号解析 Step 1: Symbol resolution**
  - 程序中定义和引用***符号***（全局变量和函数） Programs define and reference *symbols* (global variables and functions):
    - `void swap() {…}  /* define symbol swap */`
    - `swap();  /* reference symbol swap */`
    - `int *xp = &x;  /* define symbol xp, reference x */`
  - 符号的定义（被汇编器）存放在目标文件中的*符号表*中 Symbol definitions are stored in object file (by assembler) in *symbol table*.
    - 符号表是一个条目结构的数组 Symbol table is an array of entries
    - 每个条目结构包括名字、大小和符号的位置 Each entry includes name, size, and location of symbol.
  - **在符号解析这个步骤中，链接器将每个符号引用与恰好一个符号定义关联起来 During symbol resolution step, the linker associates each symbol reference with exactly one symbol definition.**

示例C语言程序中的符号
Symbols in Example C Program

定义 Definitions

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

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

int main(int argc, char** argv)
{
    int val = sum(array, 2);
    return val;
}
```

main.c

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

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

sum.c

引用 Reference

链接器做些什么？ （续）
What Do Linkers Do? (cont'd)

- **第二步：重定位 Step 2: Relocation**
  - 将独立的代码和数据节合并为单一的节 Merges separate code and data sections into single sections
  - 将符号在.o文件中的相对位置重定位到可执行文件中的最终绝对内存位置 Relocates symbols from their relative locations in the .o files to their final absolute memory locations in the executable.
  - 更新所有对这些符号的引用，以反应它们的新位置 Updates all references to these symbols to reflect their new positions.

下面让我们详细来看看这两个步骤……
Let's look at these two steps in more detail….

三种类型的目标文件（模块）
Three Kinds of Object Files (Modules)

- **可重定位目标文件 （.o 文件） Relocatable object file (.o file)**
  - 包含二进制代码和数据，其形式使之可以在编译时与其他可重定位目标文件合并起来，创建一个可执行目标文件 Contains code and data in a form that can be combined with other relocatable object files to form executable object file.
    - 每个.o文件恰好来自于一个源（.c）文件 Each .o file is produced from exactly one source (.c) file

- **可执行目标文件（a.out文件） Executable object file (a.out file)**
  - 包含二进制代码和数据，其形式使之可以被直接复制到内存并执行 Contains code and data in a form that can be copied directly into memory and then executed.

三种类型的目标文件（模块）
Three Kinds of Object Files (Modules)

- **共享目标文件（.so文件） Shared object file (.so file)**
  - 特殊类型的可重定位目标文件，可以在加载或者运行时被动态地加载进内存并链接 Special type of relocatable object file that can be loaded into memory and linked dynamically, at either load time or run-time.
  - Windows中称为***动态链接库***（DLLs） Called *Dynamic Link Libraries* (DLLs) by Windows

可执行可链接格式（ELF）
Executable and Linkable Format (ELF)

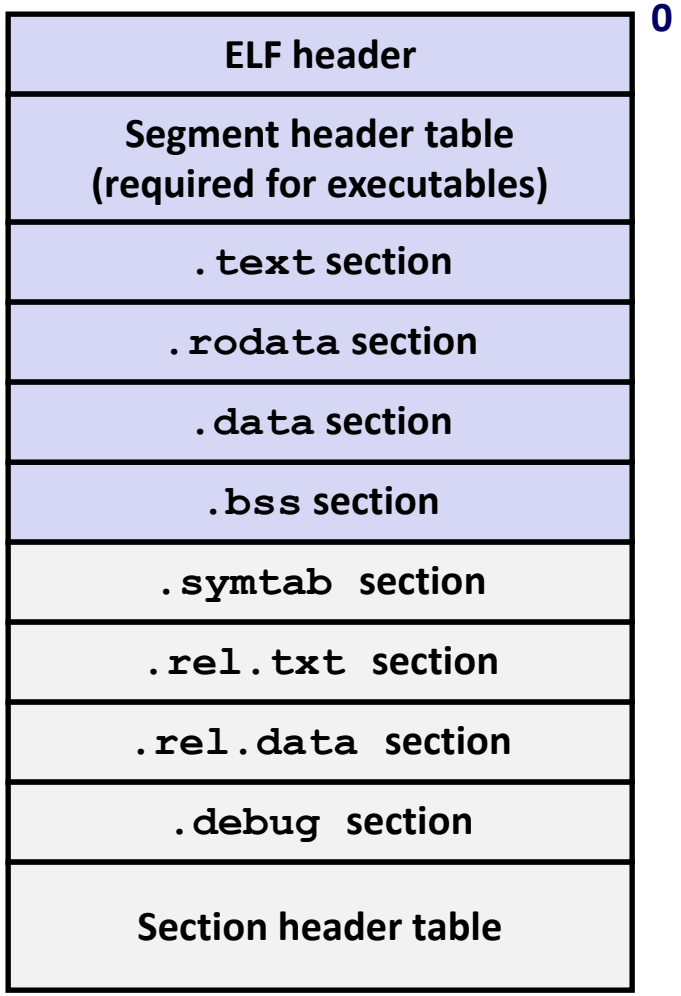
- **目标文件的标准二进制格式 Standard binary format for object files**

- **三种目标文件都是统一的格式： One unified format for**
  - 可重定位目标文件 (`.o`) Relocatable object files (`.o`),
  - 可执行目标文件 (`a.out`) Executable object files (`a.out`)
  - 共享目标文件 (`.so`) Shared object files (`.so`)

- **共有的名字：ELF 二进制文件 Generic name: ELF binaries**

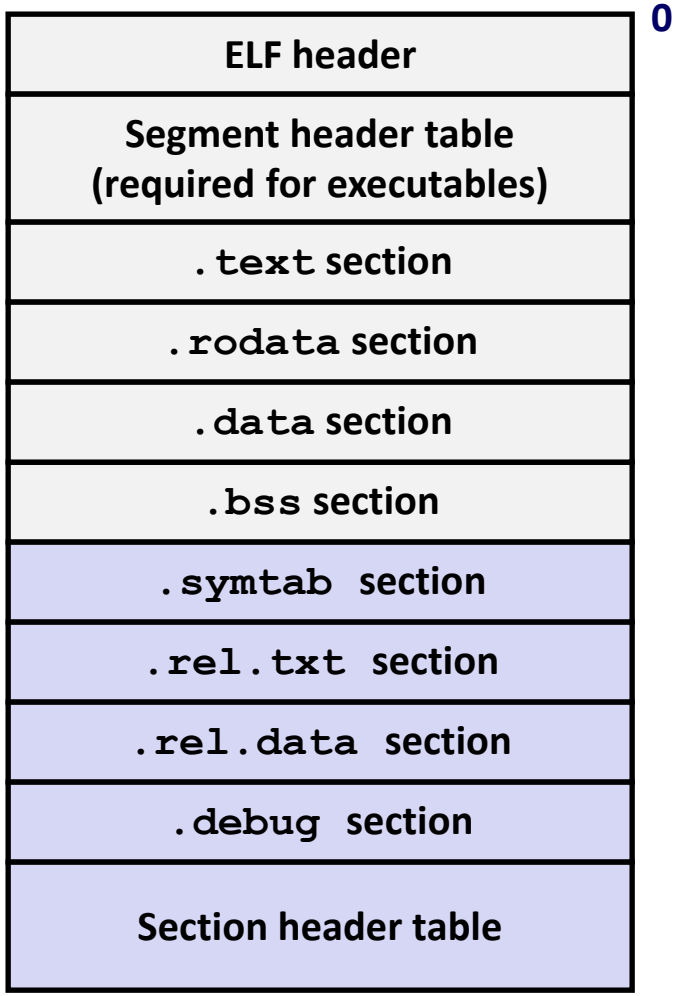
ELF目标文件格式 ELF Object File Format

- **ELF头 Elf header**
  - 字大小、字节序、文件类型（.o, exec, .so）、机器类型、等等 Word size, byte ordering, file type (.o, exec, .so), machine type, etc.
- **段头部表（可执行文件需要） Segment header table**
  - 页大小、虚拟地址内存段（节）、段的大小 Page size, virtual address memory segments (sections), segment sizes.
- **`.text`节 `.text` section**
  - 代码 Code
- **`.rodata`节 `.rodata` section**
  - 只读数据：跳转表、串常量… Read only data: jump tables, string constants, ...
- **`.data`节 `.data` section**
  - 已初始化的全局变量 Initialized global variables



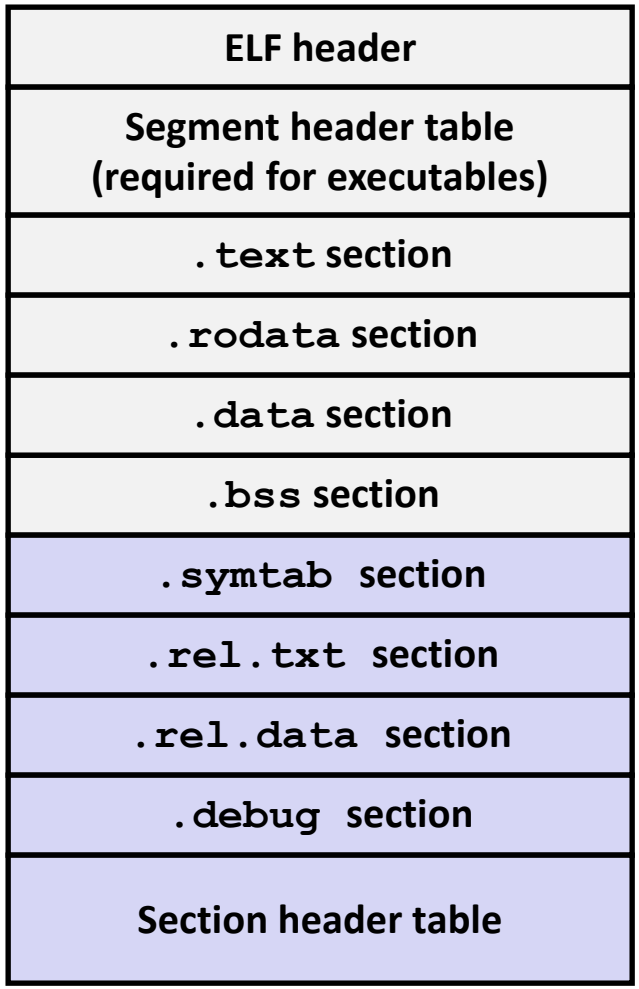
ELF目标文件格式（续）
ELF Object File Format (cont.)

- **.bss节 .bss section**
  - 未初始化的全局变量 Uninitialized global variables
  - “块符号开始bss” “Block Started by Symbol”
  - “更好节省空间bss” “Better Save Space”
  - 有节头部但不占空间 Has section header but occupies no space
- **.symtab节 .symtab section**
  - 符号表 Symbol table
  - 过程和静态变量名字 Procedure and static variable names
  - 节名和位置 Section names and locations
- **.rel.text节 .rel.text section**
  - .text节的可重定位信息 Relocation info for .text section
  - 需要在可执行文件中被修改的指令的地址 Addresses of instructions that will need to be modified in the executable
  - 修改的指令 Instructions for modifying



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

- **`.rel.data`节 `.rel.data` section**
  - `.data`节的可重定位信息
  - Relocation info for `.data` section
  - 需要在合并的可执行文件中被修改的数据的地址 Addresses of pointer data that will need to be modified in the merged executable
- **`.debug`节 `.debug` section**
  - 调试符号 (`gcc -g`)信息 Info for symbolic debugging (`gcc -g`)
- **节头部表 Section header table**
  - 每个节的偏移量和大小 Offsets and sizes of each section



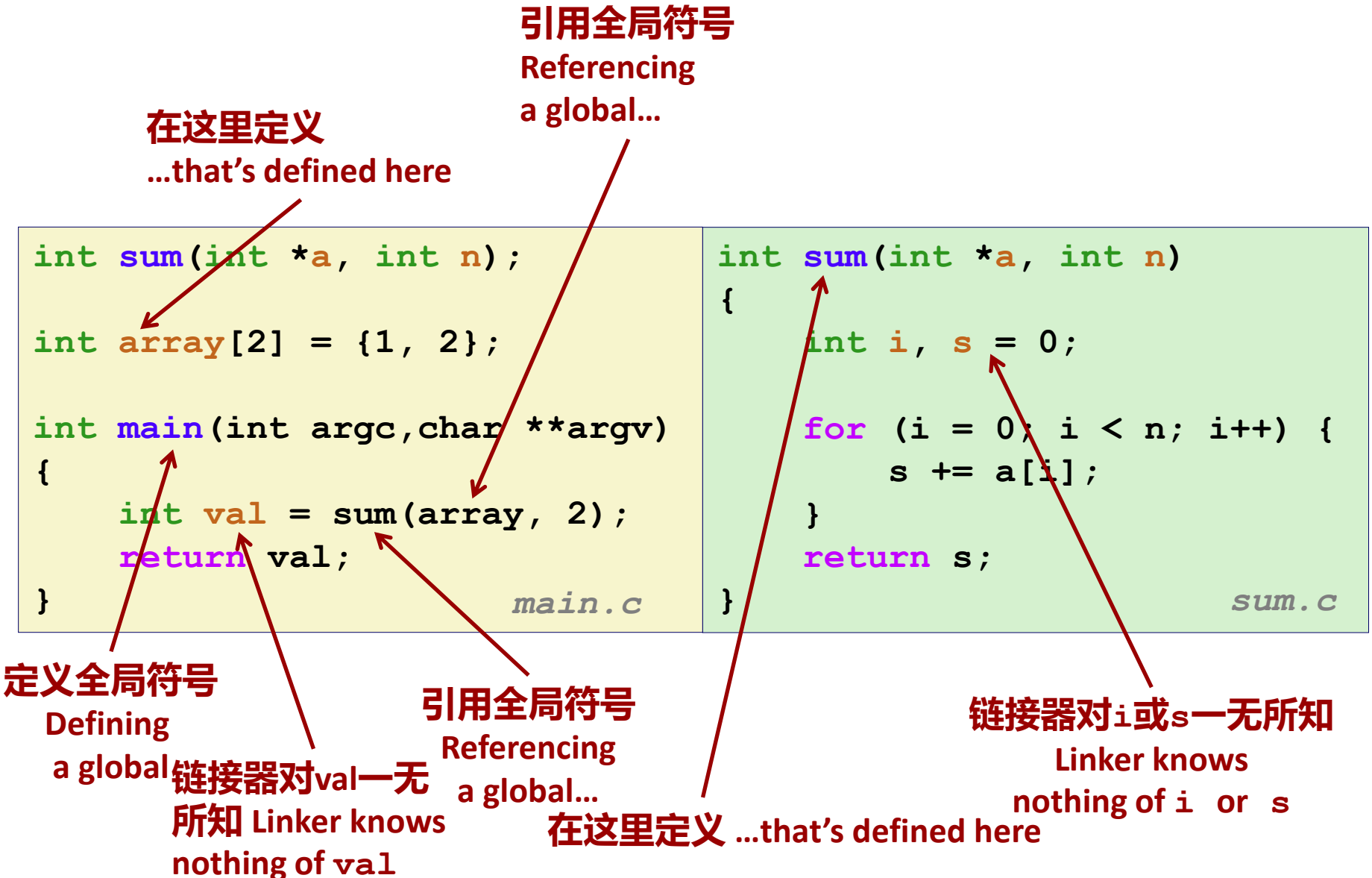
链接器符号 Linker Symbols

- **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语言函数和非**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、其他模块定义的全局符号 Global symbols that are referenced by module *m* but defined by some other module.

- **局部符号 Local symbols**
  - 模块*m* 自己定义并使用的符号 Symbols that are defined and referenced exclusively by module *m*.
  - 例如：用**`static`**限定定义的C语言函数和全局变量 e.g, C functions and global variables defined with the **`static`** attribute.
  - **局部链接器符号*不是*局部程序变量 Local linker symbols are *not* local program variables**

第一步：符号解析 Step 1: Symbol Resolution



符号标识 Symbol Identification

下面*哪些*名字会在symbols.o的符号表中？ *Which* of the following names will be in the symbol table of `symbols.o`?

symbols.c:

```
int incr = 1;
static int foo(int a) {
  int b = a + incr;
  return b;
}

int main(int argc,
         char* argv[]) {
  printf("%d\n", foo(5));
  return 0;
}
```

名字 Names:

- incr
- foo
- a
- argc
- argv
- b
- main
- printf
- "%d\n"

能够用readelf找到符号表

Can find this with `readelf`:

```
linux> readelf -s symbols.o
```

- readelf -s main.o

链接器内部使用的局部变量

```
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
```

大小　对象　函数　不确定　.data段　.text段

未定义

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

```
8       return 0;
9   }
```

code/link/m.c

a) m.c

```
5
6   void swap()
7   {
8       int temp;
9
10      bufp1 = &buf[1];
11      temp = *bufp0;
12      *bufp0 = *bufp1;
13      *bufp1 = temp;
14  }
```

code/link/swap.c

b) swap.c

图 7-5 练习题 7.1 的示例程序

局部符号 Local Symbols

- **局部非静态C变量 vs. 局部静态C变量 Local non-static C variables vs. local static C variables**
  - 局部非静态C变量：存储在栈上 Local non-static C variables: stored on the stack
  - 局部静态C变量：存储在.bss或.data中 Local static C variables: .data

```
static int x = 15;

int f() {
    static int x = 17;
    return x++;
}

int g() {
    static int x = 19;
    return x += 14;
}

int h() {
    return x += 27;
}
```

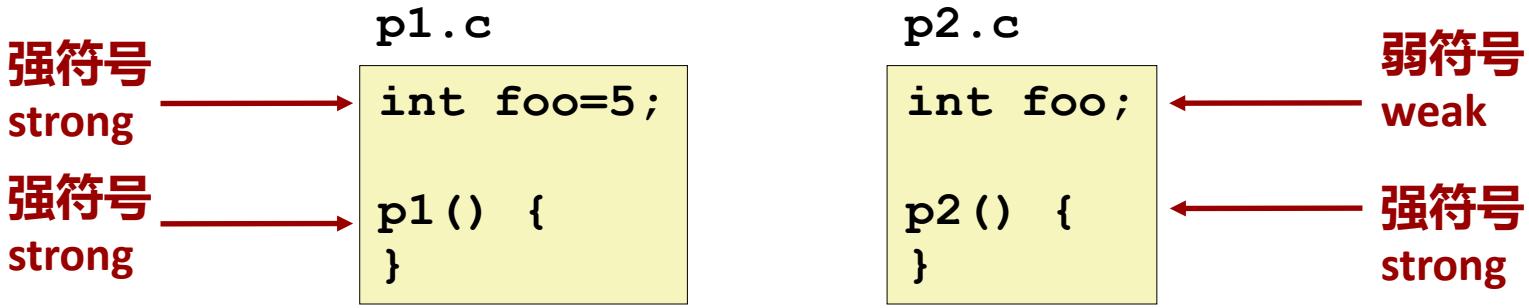
static-local.c

编译器在.data中为每个x的定义分配空间 Compiler allocates space in .data for each definition of x

在符号表中创建局部符号，每个都具有唯一的名字，例如：x.1721和x.1724 Creates local symbols in the symbol table with unique names, e.g., x, x.1721 and x.1724.

链接器如何解析多重定义的全局符号 How Linker Resolves Duplicate Symbol Definitions

- 程序符号要么是*强*符号，要么是*弱*符号 Program symbols are either *strong* or *weak*
  - ***强符号***：过程和初始化了的全局符号 ***Strong***: procedures and initialized globals
  - ***弱符号***：未初始化的全局符号 ***Weak***: uninitialized globals
    - 或者用限定符**extern**声明的 Or ones declared with specifier **`extern`**



链接器的符号规则 Linker's Symbol Rules

- **规则1： 不允许有多个同名的强符号 Rule 1: Multiple strong symbols are not allowed**
  - 每一项只能被定义一次 Each item can be defined only once
  - 否则会报：链接错误 Otherwise: Linker error
- **规则2：如果有一个强符号和多个弱符号同名，那么选择强符号 Rule 2: Given a strong symbol and multiple weak symbols, choose the strong symbol**
  - 对弱符号的引用会被解析成强符号 References to the weak symbol resolve to the strong symbol
- **规则3：如果有多个弱符号同名，那么从这些弱符号中任意选择一个 Rule 3: If there are multiple weak symbols, pick an arbitrary one**
  - 可以用`gcc -fno-common`覆盖掉这个规则（该选项现在是缺省的了）Can override this with `gcc -fno-common`
- **下一页会有一些谜题 Puzzles on the next slide**

链接器谜题 Linker Puzzles

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

```
p1() {}
```

链接时错误：两个强符号（**p1**）
Link time error: two strong symbols (**p1**)

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

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

对 **x** 的引用会可能会指向同一个未初始化的int。这真的是你的本意吗？
References to **x** will refer to the same uninitialized int. Is this what you really want?

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

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

在p2中对x的写操作可能会覆盖y！太可怕了！
Writes to **x** in **p2** might overwrite **y**!
Evil!

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

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

在p2中对x的写会覆盖y！太糟糕了！
Writes to **x** in **p2** might overwrite **y**!
Nasty!

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

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

对x的引用会指向同一个已经被初始化了的变量。
References to **x** will refer to the same initialized variable.

重要点：链接器不会做类型检查 Important: Linker does not do type checking.
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.

为什么要用Common和.bss?

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
- 链接器允许多个模块定义同名的全局符号，编译器翻译时遇到一个弱全局符号（放到COMMON中），不能决定使用哪个定义，把决定权留给链接器

为什么未初始化的静态变量放到.bss中而不是COMMON中？

类型不匹配示例 Type Mismatch Example

```c
long int x;  /* Weak symbol */

int main(int argc,
         char *argv[]) {
    printf("%ld\n", x);
    return 0;
}
```

mismatch-main.c

```c
/* Global strong symbol */
double x = 3.14;
```

mismatch-variable.c

- 编译时没有任何错误和警告 Compiles without any errors or warnings
- 打印的什么？ What gets printed?

```
-bash-4.2$ ./mismatch
4614253070214989087
```

全局变量 Global Variables

- **如果可以请尽量避免 Avoid if you can**

- **否则 Otherwise**
  - 如果可以使用**`static`** Use **`static`** if you can
  - **如果要定义全局变量，初始化之** Initialize if you define a global variable
  - 如果引用一个外部全局变量，使用**`extern`** Use **`extern`** if you reference an external global variable
    - 以弱符号来对待 Treated as weak symbol
    - 如果没有在某个文件中定义仍然会导致链接器错误 But also causes linker error if not defined in some file

在.h文件中使用extern（#1）
Use of `extern` in .h Files (#1)

c1.c

```
#include "global.h"

int f() {
  return g+1;
}
```

global.h

```
extern int g;
int f();
```

c2.c

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

int g = 0;

int main(int argc, char argv[]) {
  int t = f();
  printf("Calling f yields %d\n", t);
  return 0;
}
```

.h文件的使用（#2） Use of .h Files (#2)

c1.c

```
#include "global.h"

int f() {
  return g+1;
}
```

global.h

```
extern int g;
static int init = 0;
```

```
#else
  extern int g;
  static int init = 0;
#endif
```

c2.c

```
#define INITIALIZE
#include <stdio.h>
#include "global.h"

int main(int argc, char** argv) {
  if (init)
    // do something, e.g., g=31;
  int t = f();
  printf("Calling f yields %d\n", t);
  return 0;
}
```

```
int g = 23;
static int init = 1;
```

链接示例 Linking Example

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

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

int main(int argc,char **argv)
{
    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

第2步：重定位 Step 2: Relocation

可重定位目标文件
Relocatable Object Files

System code — .text

System data — .data

main.o

main() — .text

int array[2]={1,2} — .data

sum.o

sum() — .text

可执行目标文件
Executable Object File

0

Headers

System code

main()

sum()

More system code — .text

System data

int array[2]={1,2} — .data

.symtab
.debug

重定位条目 Relocation Entries

```
int array[2]

int main(int
argv)
{
    int val
    return v
}
```

code/link/elfstructs.c

```
1  typedef struct {
2      long offset;    /* Offset of the reference to relocate */
3      long type:32,   /* Relocation type */
4           symbol:32; /* Symbol table index */
5      long addend;    /* Constant part of relocation expression */
6  } 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
```

main.o

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

来源：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 c
  4004e7:       c3

00000000004004e8 <sum>
  4004e8:       b8 00 0
  4004ed:       ba 00 0
  4004f2:       eb 09
  4004f4:       48 63 c
  4004f7:       03 04 8
  4004fa:       83 c2 0
  4004fd:       39 f2
  4004ff:       7c f3
  400501:       f3 c3
```

```
1   foreach section s {
2       foreach relocation entry r {
3           refptr = s + r.offset;  /* ptr to reference to be relocated */
4
5           /* Relocate a PC-relative reference */
6           if (r.type == R_X86_64_PC32) {
7               refaddr = ADDR(s) + r.offset; /* ref's run-time address */
8               *refptr = (unsigned) (ADDR(r.symbol) + r.addend - refaddr);
9           }
10
11          /* Relocate an absolute reference */
12          if (r.type == R_X86_64_32)
13              *refptr = (unsigned) (ADDR(r.symbol) + r.addend);
14      }
15  }
```

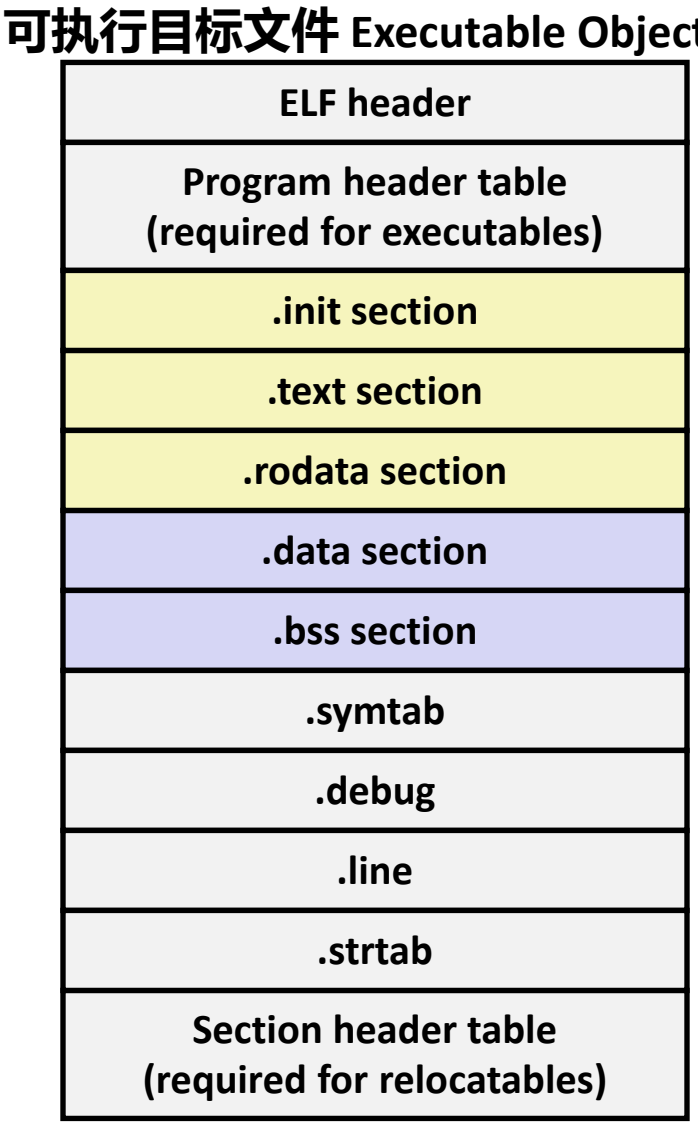
Figure 7.10 Relocation algorithm.

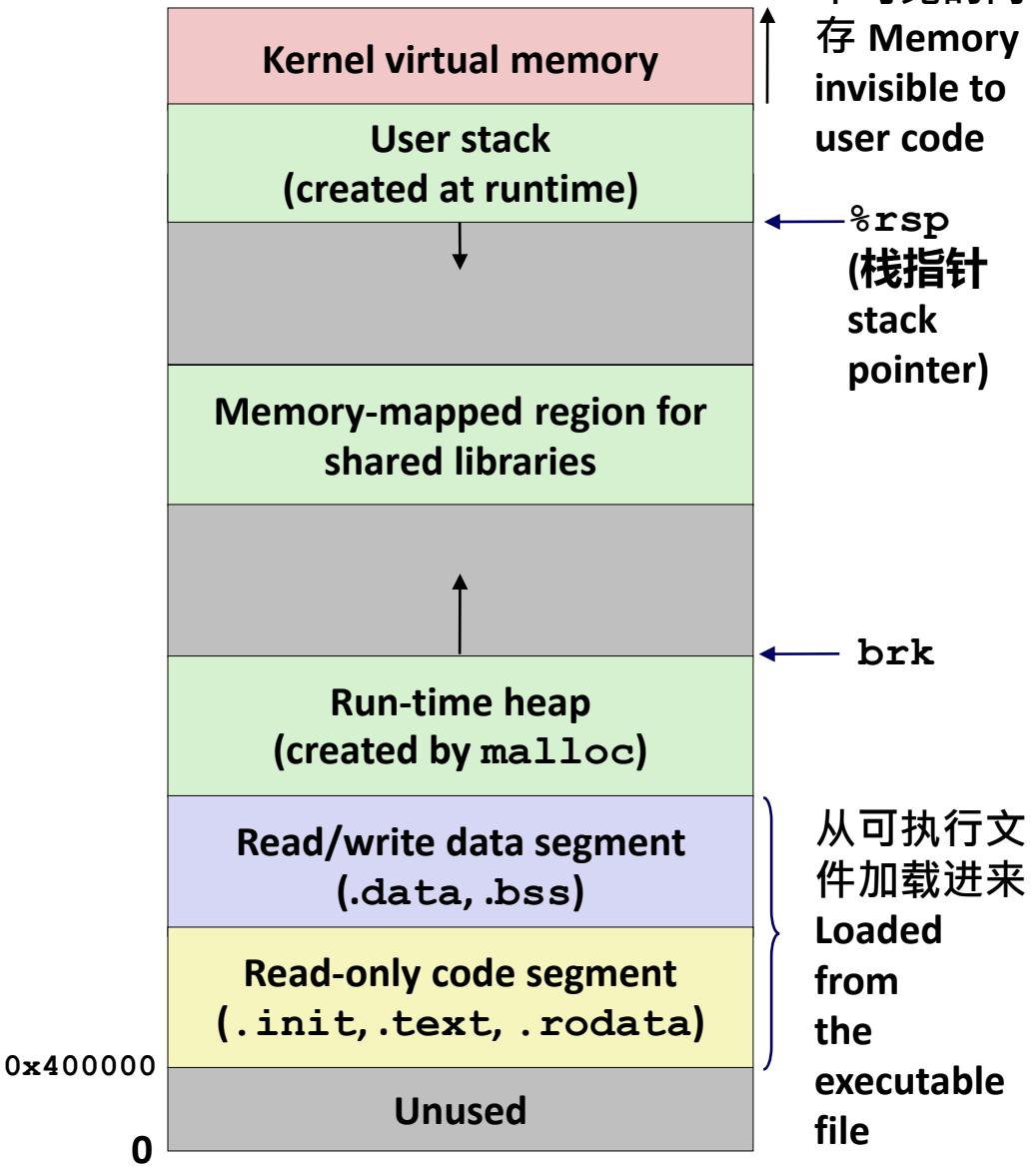
对sum()使用PC相对寻址：`callq` instruction uses PC-relative addressing for sum()

0x4004e8 = 0x4004e3 + 0x5

来源：Source: objdump -d prog

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





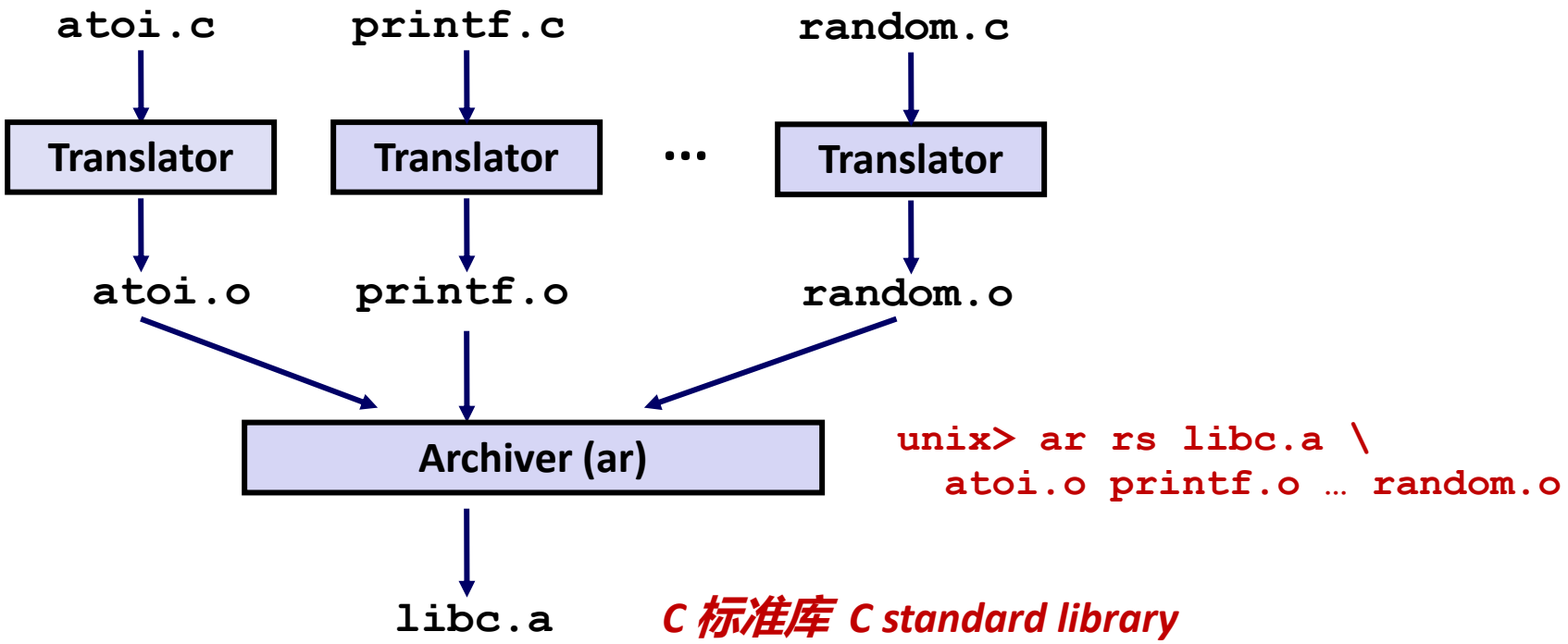
库：将一组函数打包
Libraries: Packaging a Set of Functions

- **如何将程序员常用的函数打包到一起？ How to package functions commonly used by programmers?**
  - 数学、I/O、内存管理、字符串处理等等 Math, I/O, memory management, string manipulation, etc.
- **令人棘手的，对目前的链接器框架来说： Awkward, given the linker framework so far:**
  - **选择1:** 将所有的函数都放进一个源文件中 **Option 1:** Put all functions into a single source file
    - 程序员将这个大目标文件链接进他们的程序中 Programmers link big object file into their programs
    - 空间和时间上效率都不高 Space and time inefficient
  - **选择2:** 将每个函数存放在一个独立的源文件中 **Option 2:** Put each function in a separate source file
    - 程序员明确说明将适当的二进制代码链接到他们的程序中 Programmers explicitly link appropriate binaries into their programs
    - 更高效，但对程序员来说负担较重 More efficient, but burdensome on the programmer

以往的解决方案：静态库
Old-Fashioned Solution: Static Libraries

- **静态库 (.a存档文件) Static libraries (.a archive files)**
  - 将相关的可重定位目标文件连接成一个单一的带索引的文件（称为 *archive*) Concatenate related relocatable object files into a single file with an index (called an *archive*).
  - 改进链接器使之在一个或多个存档文件里查找并解析未被解析的外部引用 Enhance linker so that it tries to resolve unresolved external references by looking for the symbols in one or more archives.
  - 如果一个存档文件中的成员文件解析了一个引用，就把它链接到可执行文件中 If an archive member file resolves reference, link it into the executable.

创建静态库 Creating Static Libraries




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

C 标准库 C standard library

- 存档器允许增量式的更新 Archiver allows incremental updates
- 重新编译修改过的函数，在存档文件中替换相应的.o文件 Recompile function that changes and replace .o file in archive.

常用的库 Commonly Used Libraries

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

- 4.6MB存档文件，包含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、数据与时间、随机数、整数运算 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)

- 2 MB存档文件，包含 444个目标文件 2 MB archive of 444 object files.
- 浮点数运算 floating point math (sin, cos, tan, log, exp, sqrt, …)

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

```
% ar -t /usr/lib/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
…
```

与静态库链接
Linking with Static Libraries

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

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

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

main2.c

libvector.a

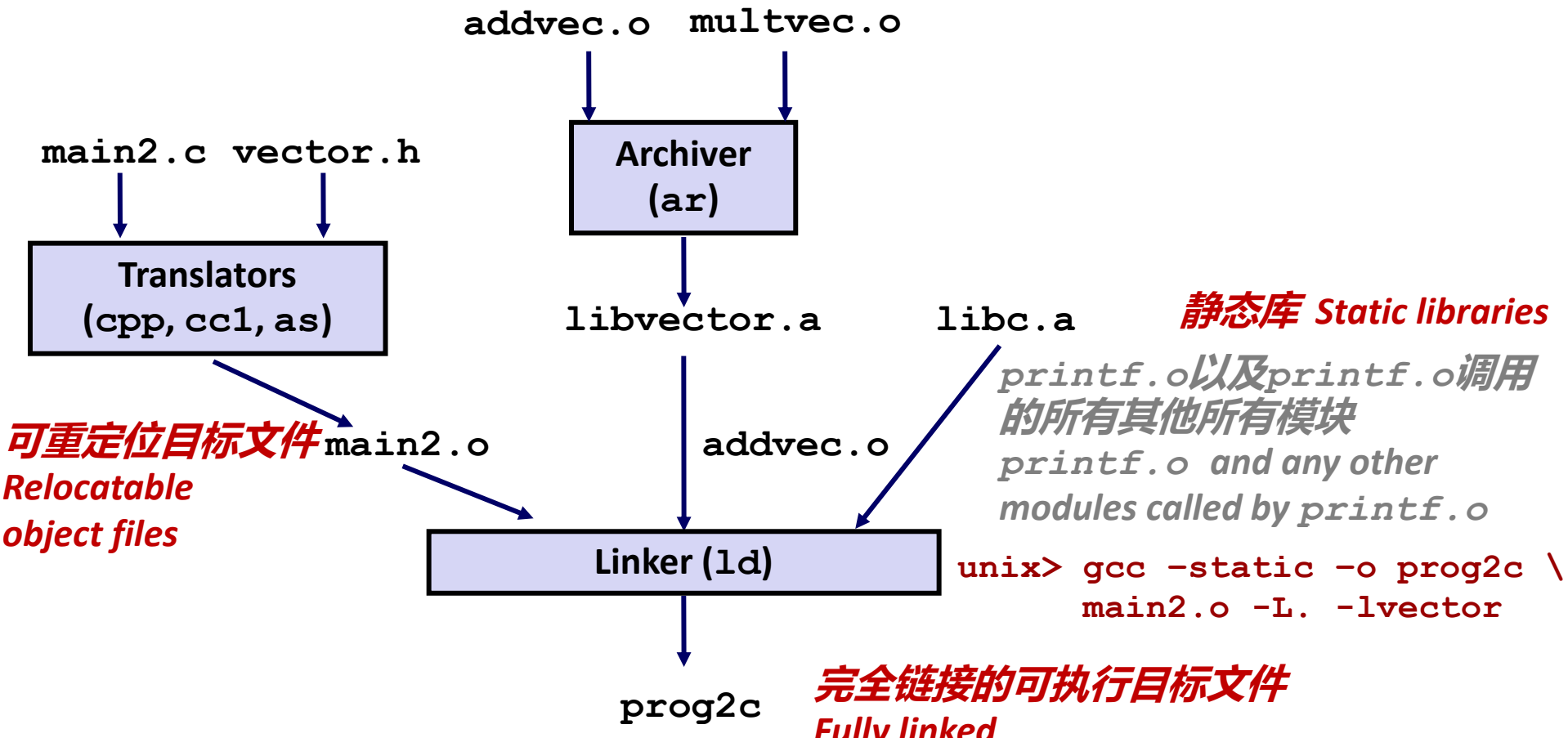
```c
void addvec(int *x, int *y,
            int *z, int n) {
    int i;

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

addvec.c

```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

与静态库链接 Linking with Static Libraries



使用静态库 Using Static Libraries

- **链接器用以解析外部引用的算法： Linker's algorithm for resolving external references:**
  - 按照命令行给出的顺序扫描.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，记录一个当前未解析引用的列表 During the scan, keep a list of the current unresolved references.
  - 每当遇到一个新的.o或者.a file文件，*obj*，时，就试着用*obj*中定义的符号来解析列表中每个未被解析的引用 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*.
  - 如果扫描结束后，未解析列表中还有条目，那么就报错 If any entries in the unresolved list at end of scan, then error.
- **问题： Problem:**
  - 命令行上文件的顺序很重要！ Command line order matters!
  - 规范：将库函数放到命令行的结尾处 Moral: put libraries at the end of the command line.

```
unix> gcc -static -o prog2c -L. -lvector main2.o
main2.o: In function `main':
main2.c:(.text+0x19): undefined reference to `addvec'
collect2: error: ld returned 1 exit status
```

现代解决方法：共享库
Modern Solution: Shared Libraries

- **静态库有如下缺点： Static libraries have the following disadvantages:**
  - 在被存储起来的可执行文件中存在冗余（每个函数都需要libc）Duplication in the stored executables (every function needs libc)
  - 在运行的可执行文件中存在冗余 Duplication in the running executables
  - 对系统库函数最微小的bug修订都需要每个应用程序进行重新链接 Minor bug fixes of system libraries require each application to explicitly relink
    - 用glibc重新构建一切？ Rebuild everything with glibc?
    - https://security.googleblog.com/2016/02/cve-2015-7547-glibc-getaddrinfo-stack.html

现代解决方法：共享库
Modern Solution: Shared Libraries

- **现代解决方法：共享库 Modern solution: shared libraries**
  - 包含代码和数据的目标文件，在*加载时（load-time）*或者*运行时（run-time）*被动态地被链接进应用程序中 Object files that contain code and data that are loaded and linked into an application *dynamically,* at either *load-time* or *run-time*
  - 也称为：动态链接库（DLL），`.so`文件 Also called: dynamic link libraries, DLLs, `.so` files

共享库（续） Shared Libraries (cont.)

- **动态链接可以发生在可执行文件首次被加载进内存开始执行时（加载时链接） Dynamic linking can occur when executable is first loaded and run (load-time linking)**
  - Linux上的一般情况，由动态链接器(`ld-linux.so`)自动处理 Common case for Linux, handled automatically by the dynamic linker (`ld-linux.so`)
  - 标准C库(`libc.so`) 通常是自动链接的 Standard C library (`libc.so`) usually dynamically linked
- **动态链接也可以发生在程序开始执行后（运行时链接） Dynamic linking can also occur after program has begun (run-time linking)**
  - 在Linux上，是通过调用`dlopen()`接口来实现的 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（续） Shared Libraries (cont.)

- **共享库例程可以多进程共享 Shared library routines can be shared by multiple processes**
  - 在学过虚拟内存后可以更好的理解这一点 More on this when we learn about virtual memory

动态库需要什么?
What dynamic libraries are required?

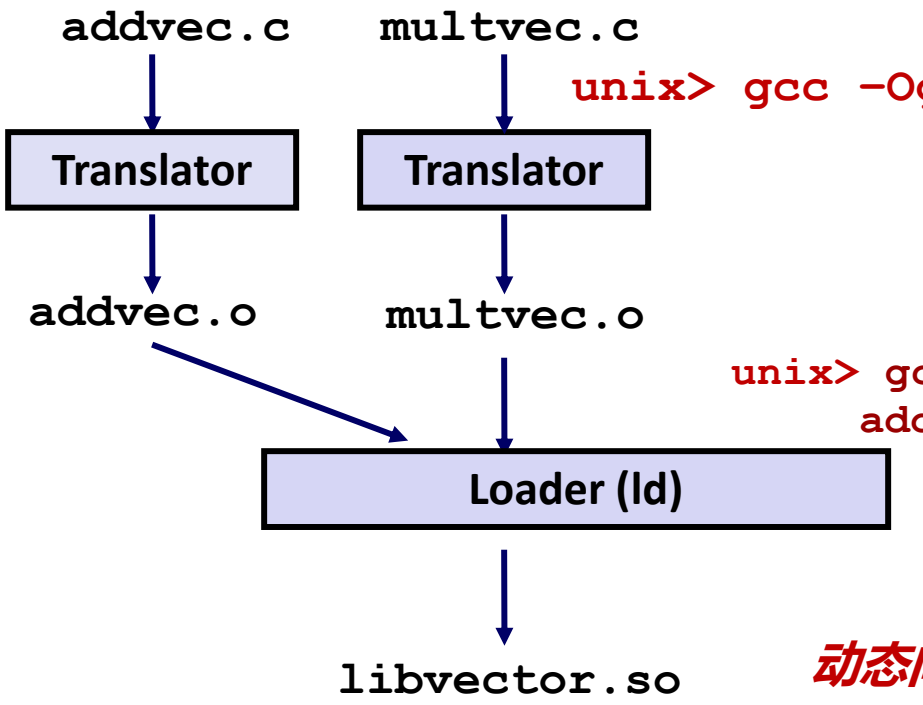
- **`.interp`节 `.interp` section**
  - 指定使用的动态链接器（例如**`ld-linux.so`**） Specifies the dynamic linker to use (i.e., **`ld-linux.so`**)
- **`.dynamic`节 `.dynamic` section**
  - 指定使用的动态库名字等 Specifies the names, etc of the dynamic libraries to use
  - 下面是prog示例 Follow an example of **`prog`**

（需要 NEEDED）　　共享库 Shared library: [libm.so.6]

- **找到库在哪儿? Where are the libraries found?**
  - 使用“ldd”找到 Use “**`ldd`**” to find out:

```
unix> ldd prog
  linux-vdso.so.1 =>  (0x00007ffcf2998000)
  libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f99ad927000)
  /lib64/ld-linux-x86-64.so.2 (0x00007f99adcef000)
```

动态库示例 Dynamic Library Example



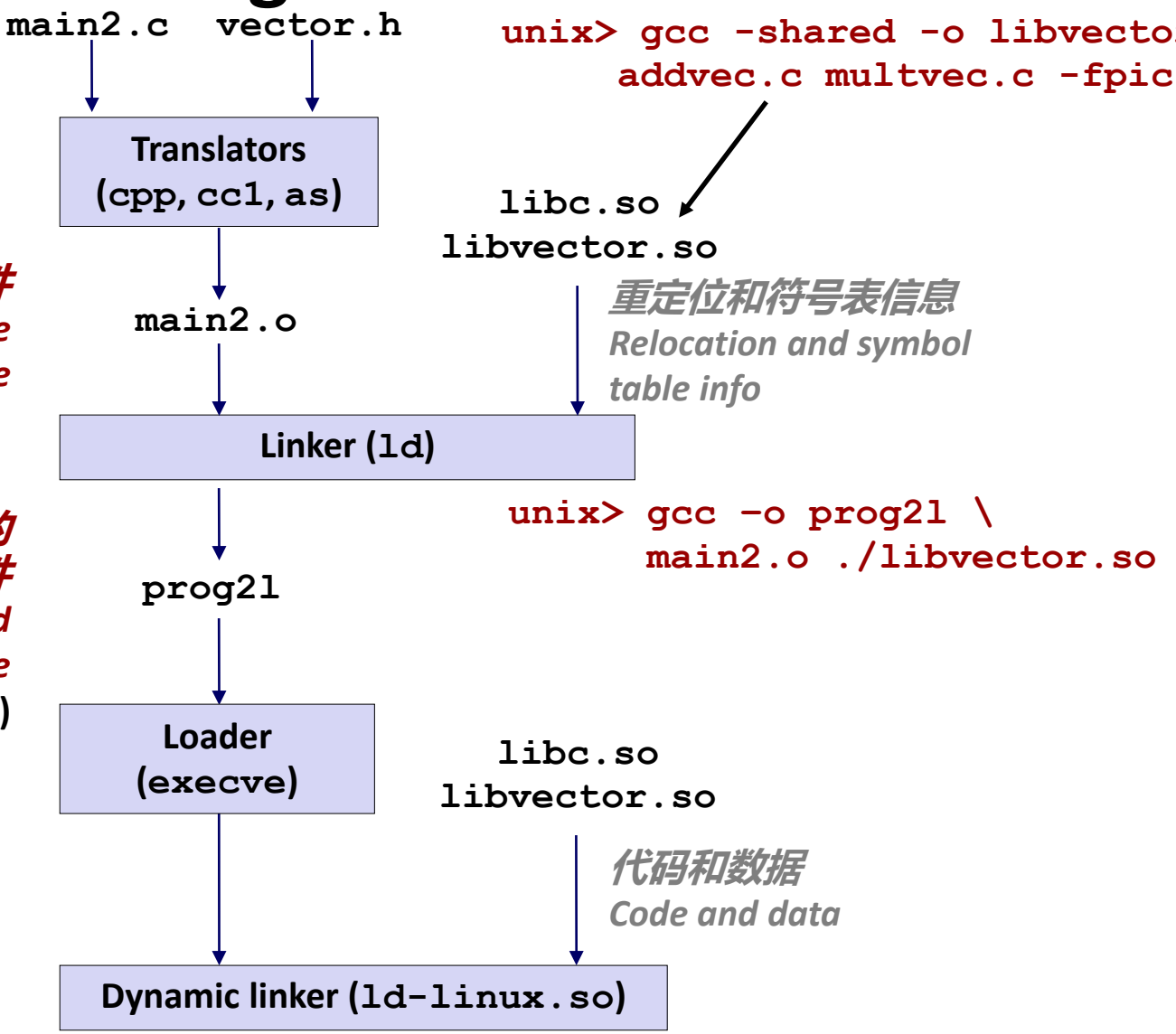

```
unix> gcc -Og -c addvec.c multvec.c -fpic
```

```
unix> gcc -shared -o libvector.so \
      addvec.o multvec.o
```

动态向量库

Dynamic vector library

加载时动态链接
Dynamic Linking at Load-time



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(int argc, char** argv)
{
    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);
    }
  . . .
```

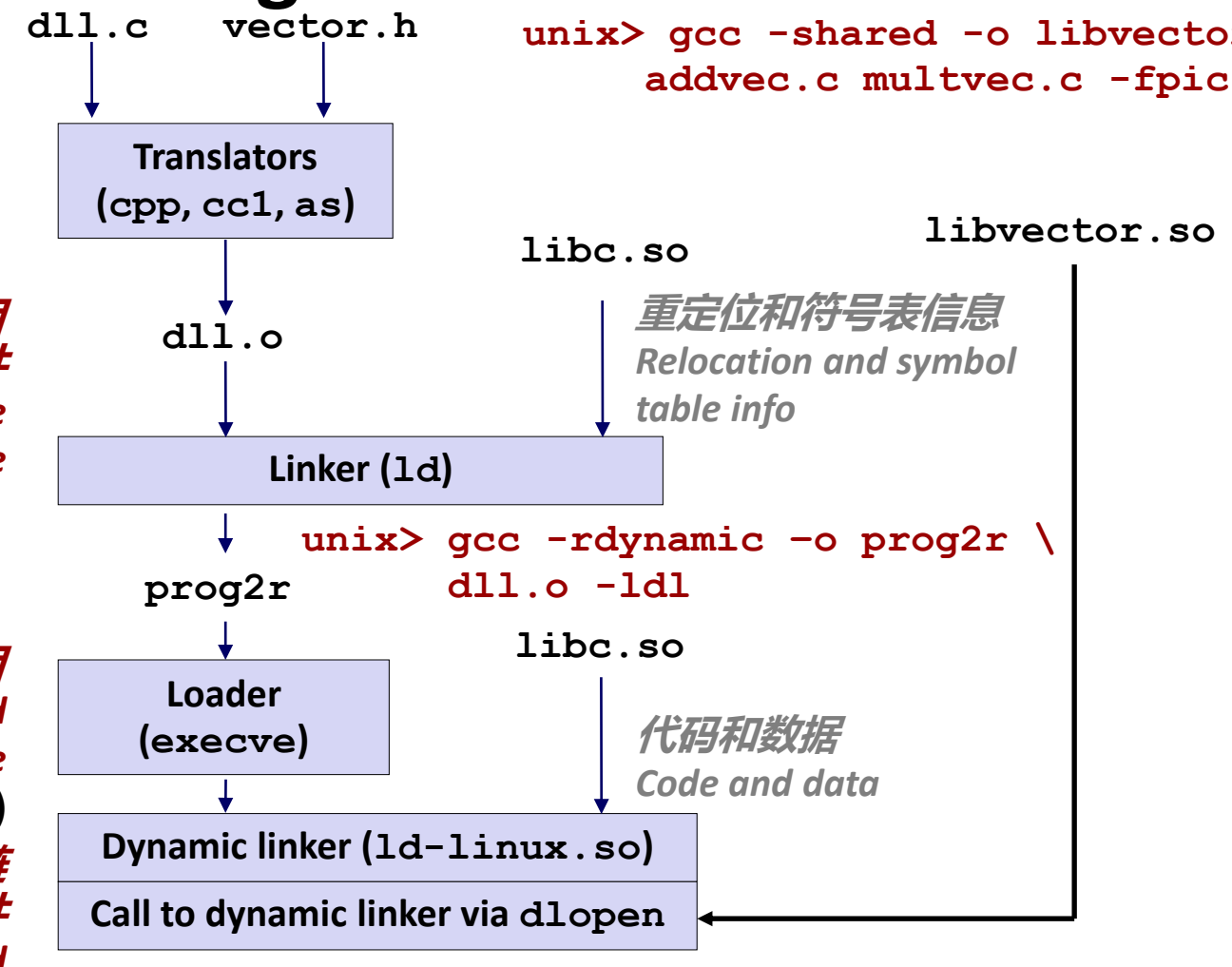
dll.c

运行时动态链接 （续）
Dynamic Linking at Run-time (cont'd)

```
    ...

    /* 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

运行时动态链接
Dynamic Linking at Run-time



链接小结 Linking Summary

- **链接是一种允许从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**

- **链接可以在程序生命周期的不同时间发生： Linking can happen at different times in a program's lifetime:**
  - 编译时间（编译程序时） Compile time (when a program is compiled)
  - 加载时间（当程序加载到内存中时） Load time (when a program is loaded into memory)
  - 运行时间（程序执行时） Run time (while a program is executing)

- **理解链接可以帮助你避免讨厌的错误，让你成为一个更好的程序员 Understanding linking can help you avoid nasty errors and make you a better programmer**

提纲

- 链接 Linking
- **案例研究：库打桩 Case study: Library interpositioning**

案例研究：库打桩
Case Study: Library Interpositioning

- **参见教材7.13节内容 Documented in Section 7.13 of book**
- **库打桩：一项强大的链接技术，使得程序员可以截获对任意函数的调用 Library interpositioning: powerful linking technique that allows programmers to intercept calls to arbitrary functions**
- **库打桩可以发生在： Interpositioning can occur at:**
  - 编译时：在编译源代码时 Compile time: When the source code is compiled
  - 链接时：在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：在可执行目标文件被加载进内存、动态链接并被执行时 Load/run time: When an executable object file is loaded into memory, dynamically linked, and then executed.

库打桩的应用
Some Interpositioning Applications

- **安全 Security**
  - 限制（沙盒） Confinement (sandboxing)
  - 幕后加密 Behind the scenes encryption
- **调试 Debugging**
  - 2014年，两个Facebook的工程师利用库打桩找到了一个隐匿在他们iPhone app里长达一年的bug In 2014, two Facebook engineers debugged a treacherous 1-year old bug in their iPhone app using interpositioning
  - SPDY网络栈的代码写入了一个错误的位置 Code in the SPDY networking stack was writing to the wrong location
  - 用对Posix写函数functions (write, writev, pwrite)库打桩的方法解决的 Solved by intercepting calls to Posix write functions (write, writev, pwrite)

来源：Facebook的工程博客贴 Source: Facebook engineering blog post at:
https://code.facebook.com/posts/313033472212144/debugging-file-corruption-on-ios/

库打桩的应用
Some Interpositioning Applications

- **监控和侧写 Monitoring and Profiling**
  - 对函数调用计数 Count number of calls to functions
  - 描述函数调用的位置和参数的特征 Characterize call sites and arguments to functions
  - malloc轨迹追踪 Malloc tracing
    - 检测内存渗漏 Detecting memory leaks
    - **生成地址跟踪 Generating address traces**
- **差错检测 Error Checking**
  - C编程实验使用定制版本的malloc/free进行仔细的错误检查 C Programming Lab used customized versions of malloc/free to do careful error checking
  - 其他实验（malloc、shell、proxy）也使用库打桩来增强检查能力 Other labs (malloc, shell, proxy) also use interpositioning to enhance checking capabilities

示例程序 Example program

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

int main(int argc,
         char *argv[])
{
  int i;
  for (i = 1; i < argc; i++) {
    void *p =
          malloc(atoi(argv[i]));
    free(p);
  }
  return(0);
}
```

int.c

- 目标：在不破坏程序，而且不修改源代码的情况下，记录分配和释放的块的地址和大小 Goal: trace the addresses and sizes of the allocated and freed blocks, without breaking the program, and without modifying the source code.

示例程序 Example program

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

int main(int argc,
         char *argv[])
{
  int i;
  for (i = 1; i < argc; i++) {
    void *p =
          malloc(atoi(argv[i]));
    free(p);
  }
  return(0);
}
```

int.c

- 对库函数 malloc和free进行库打桩三种解决方法：在 Three solutions: interpose on the library `malloc` and `free` functions at
  - 编译时 compile time,
  - 链接时 link time,
  - 加载/运行时 and load/run time

编译时库打桩
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

编译时库打桩
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 10 100 1000
malloc(10)=0x1ba7010
free(0x1ba7010)
malloc(100)=0x1ba7030
free(0x1ba7030)
malloc(1000)=0x1ba70a0
free(0x1ba70a0)
linux>
```

搜索<malloc.h>会指向
Search for <malloc.h> leads to
/usr/include/malloc.h

搜索<malloc.h>会指向
Search for <malloc.h> leads to

链接时库打桩 Link-time Interpositioning

```c
#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

链接时库打桩 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 10 100 1000
malloc(10) = 0x91a010
free(0x91a010)
. . .
```

搜索<malloc.h>会指向
Search for <malloc.h> leads to
/usr/include/malloc.h

- “-Wl”标志将参数传给链接器，其中每个逗号都替换为一个空格 The “-Wl” flag passes argument to linker, replacing each comma with a space.

--wrap=*symbol*

Use a wrapper function for *symbol*. Any undefined reference to *symbol* will be resolved to "__wrap_*symbol* ". Any undefined reference to "__real_ *symbol* " will be resolved to *symbol*.

链接时库打桩 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 10 100 1000
malloc(10) = 0x91a010
free(0x91a010)
. . .
```

搜索<malloc.h>会指向
Search for <malloc.h> leads to
/usr/include/malloc.h

- “--wrap,malloc”参数指示链接器以一种特殊的方式来解析引用: The “--wrap,malloc” arg instructs linker to resolve references in a special way:
  - 对malloc的引用应该被解析为__wrap_malloc Refs to malloc should be resolved as __wrap_malloc
  - 对__real_malloc的引用应该被解析为malloc Refs to __real_malloc should be resolved as malloc

加载/运行时库打桩
Load/Run-time Interpositioning

观察没有#include <malloc.h>
Observe that DON'T have
#include <malloc.h>

```c
#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

加载/运行时库打桩
Load/Run-time Interpositioning

```c
/* 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

加载/运行时库打桩
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 10 100 1000)
malloc(10) = 0x91a010
free(0x91a010)
. . .
linux>
```

搜索<malloc.h>会指向
Search for <malloc.h> leads to
/usr/include/malloc.h

- LD_PRELOAD环境变量告诉动态链接器在对未被解析的引用（例如对malloc的引用）进行解析时，要先在mymalloc.so中进行查找 The LD_PRELOAD environment variable tells the dynamic linker to resolve unresolved refs (e.g., to malloc) by looking in mymalloc.so first.
- 某些外壳中输入：Type into (some) shells as:

```
env LD_PRELOAD=./mymalloc.so ./intr 10 100 1000
```

回顾：库打桩 Interpositioning Recap

- **编译时 Compile Time**
  - 对malloc/free的调用被宏扩展成对mymalloc/myfree的调用 Apparent calls to **`malloc`**/**`free`** get macro-expanded into calls to **`mymalloc`**/**`myfree`**
  - 简单方法，必须能访问源代码并重新编译 Simple approach. Must have access to source & recompile
- **链接时 Link Time**
  - 利用链接器的特殊的名字解析功能 Use linker trick to have special name resolutions
    - **`malloc`** → **`__wrap_malloc`**
    - **`__real_malloc`** → **`malloc`**
- **加载/运行时 Load/Run Time**
  - 实现自己定制的malloc/free版本，使用动态链接以不同的名字来加载malloc/free函数库 Implement custom version of **`malloc`**/**`free`** that use dynamic linking to load library **`malloc`**/**`free`** under different names
  - 可以使用任何动态链接库 Can use with ANY dynamically linked binary

```
env LD_PRELOAD=./mymalloc.so gcc -c int.c
```

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

- **位置无关代码 PIC**
  - 使用gcc选项–fpic　　-fpic option to gcc
  - 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 The code segments of shared modules can be loaded anywhere in memory without having to be modified by the linker.
- **同一个可执行目标模块中符号的引用 references to symbols in the same executable object module**
  - PC相对寻址 using PC-relative addressing
  - 构造目标文件时由静态链接器重定位　relocated by the static linker when it builds the object file.
- **不需任何重定位即可装入的代码称为位置无关代码（PIC） Code that can be loaded without needing any relocations is known as position independent code (PIC).**

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

- **位置无关代码 PIC**
  - 使用gcc选项–fpic　　-fpic option to gcc
  - 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 The code segments of shared modules can be loaded anywhere in memory without having to be modified by the linker.
- **同一个目标模块中符号的引用 references to symbols in the same executable object module**
  - PC相对寻址

```
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
PIC Data References

- **事实：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 fact: no matter where we load an object module (including shared object modules) in memory, the data segment is always the same distance from the code segment. Thus, the distance between any instruction in the code segment and any variable in the data segment is a run-time constant.**
- **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字节条目 The GOT contains an 8-byte entry for each global data object that is referenced by the object module.
  - 链接器在构造这个模块时解析它 At load time, the dynamic linker relocates each GOT entry.

PIC数据引用
PIC Data References

- 事实：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 fact: no matter where we load an object module (including shared object modules) in memory, the data segment is always the same distance from the code segment. Thus, the distance between n the

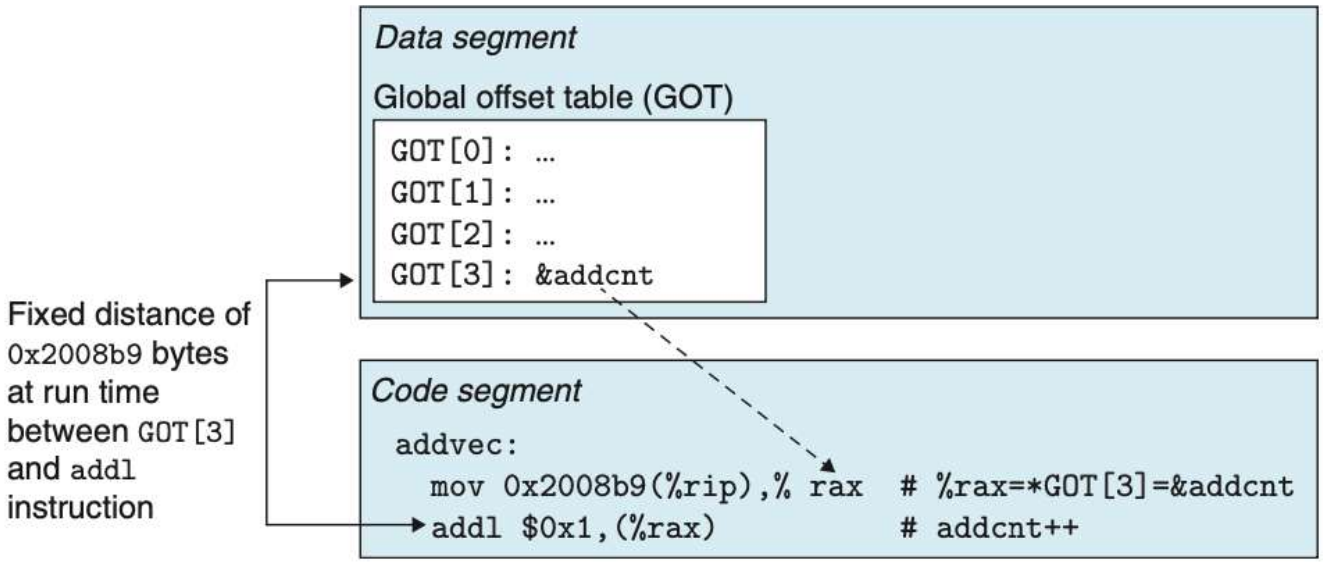


Figure 7.18 Using the GOT to reference a global variable. The addvec routine in libvector.so references addcnt indirectly through the GOT for libvector.so.

PIC数据引用
PIC Data References

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

```
1  Sections:
2  Idx Name          Size      VMA               LMA               File off  Algn
3   15 .got          00000030  0000000000200fd0  0000000000200fd0  00000fd0  2**3
4                    CONTENTS, ALLOC, LOAD, DATA
5   17 .data         00000008  0000000000201018  0000000000201018  00001018  2**3
6                    CONTENTS, ALLOC, LOAD, DATA
7
8  SYMBOL TABLE:
9  0000000000200fd0 l    d  .got   0000000000000000 .got
10
11 000000000000060a <addvec>:
12  60a:   4c 8b 05 cf 09 20 00    mov    0x2009cf(%rip),%r8        # 200fe0 <addcnt-0x44>
13  611:   41 8b 00                mov    (%r8),%eax
14
15 0000000000000639 <multvec>:
16  639:   4c 8b 05 98 09 20 00    mov    0x200998(%rip),%r8        # 200fd8 <multcnt-0x50>
17  640:   41 8b 00                mov    (%r8),%eax
```

PIC函数调用 PIC Function Calls

- 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。 One approach：would be to generate a relocation record for the reference, which the dynamic linker could then resolve when the program was loaded. However, this approach would not be PIC, since require the linker to modify the code segment of the calling module.
- PIC的方法：延迟绑定 PIC：called lazy binding

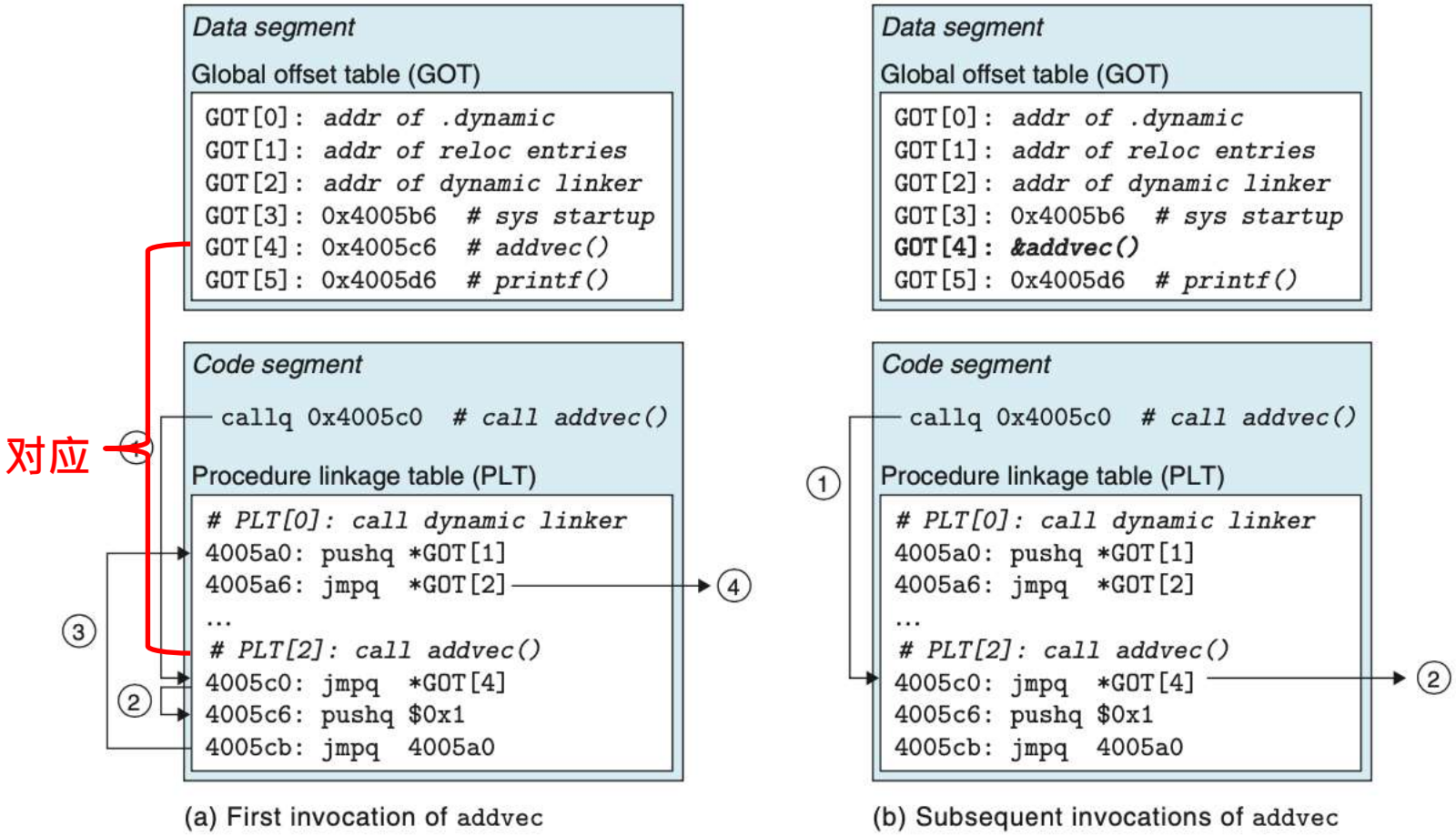
PIC函数调用 PIC Function Calls

- PIC的方法：延迟绑定　PIC：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 defers the binding of each procedure address until the first time the procedure is called.
  - 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。 GOT is an array of 8-byte address entries. GOT[0] and GOT[1] contain information that the dynamic linker uses when it resolves function addresses. GOT[2] is the entry point for the dynamic linker in the ld-linux.so module. Each of the remaining entries corresponds to a called function. Each has a matching PLT entry.

PIC函数调用 PIC Function Calls

- PIC的方法：延迟绑定　PIC：lazy binding
  - 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 The PLT is an array of 16-byte code entries. PLT[0] is a special entry that jumps into the dynamic linker. Each shared library function called by the executable has its own PLT entry. Each of these entries is responsible for invoking a specific function.

PIC函数调用 PIC Function Calls



(a) First invocation of addvec

(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 PIC Function Calls

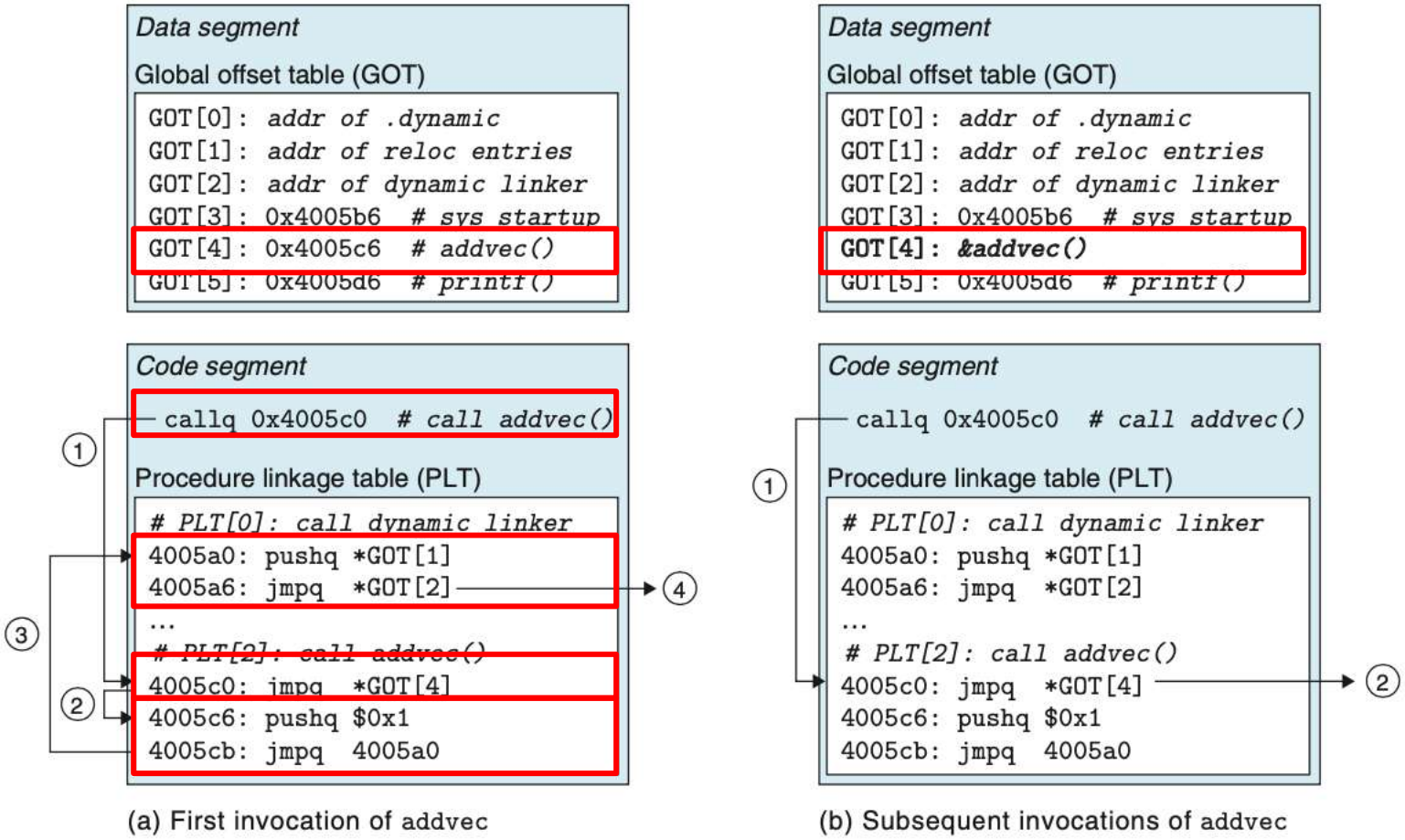


Figure 7.19 Using the PLT and GOT to call external functions. The dynamic linker resolves the address of addvec the first time it is called.

PIC函数调用 PIC Function Calls

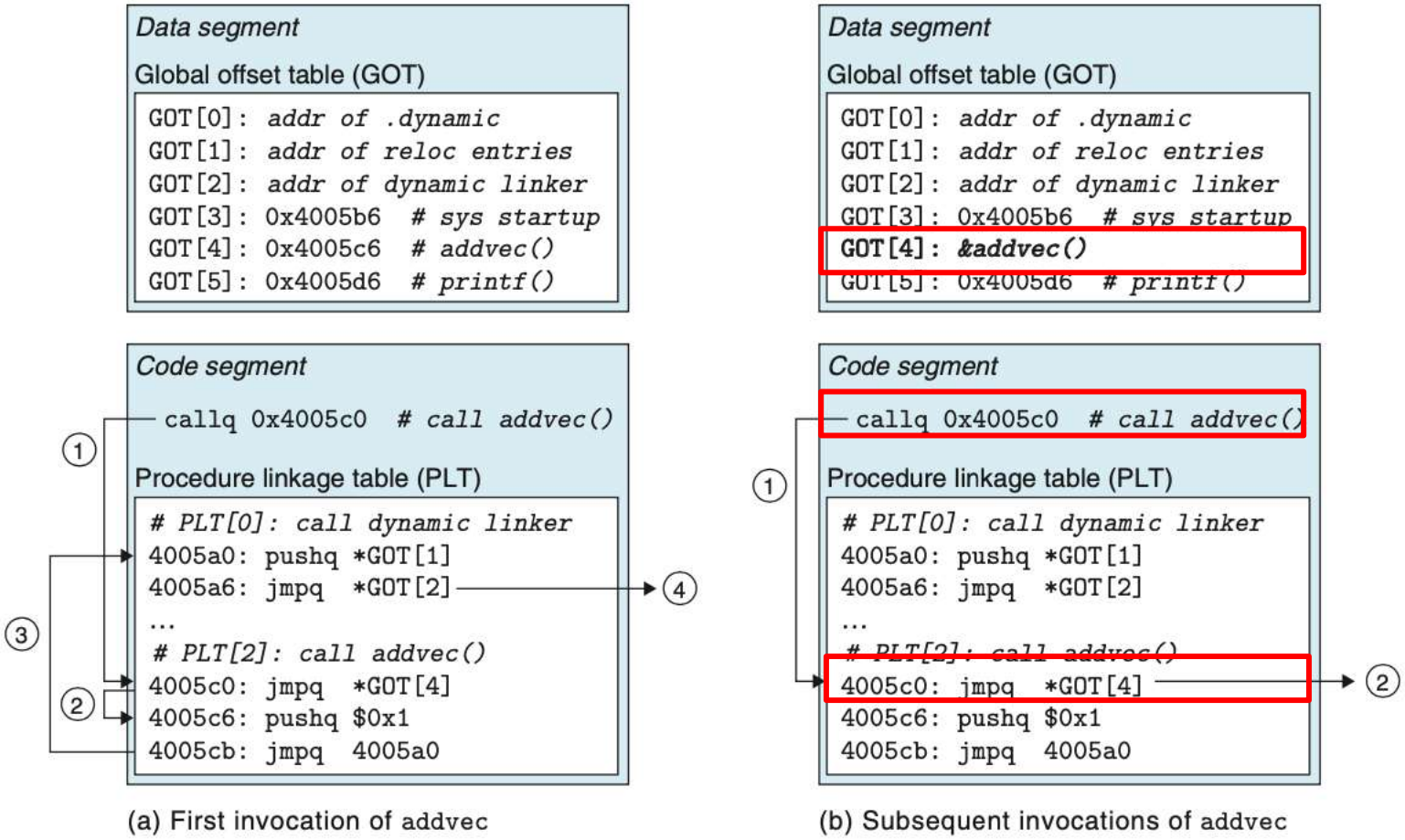


Figure 7.19 **Using the PLT and GOT to call external functions.** The dynamic linker resolves the address of addvec the first time it is called.

链接概括 Linking Recap

- **通常：只是发生，不是大事 Usually: Just happens, no big deal**
- **有时：奇怪的错误 Sometimes: Strange errors**
  - 错误的符号解析 Bad symbol resolution
  - 排序链接的.o、.a和.so文件的依赖关系 Ordering dependence of linked .o, .a, and .so files
- **对于高级用户： For power users:**
  - 库打桩以跟踪有源代码和无源代码的程序 Interpositioning to trace programs with & without source