第七章　链接

- 教　师：郑贵滨
- 计算机科学与技术学院
- 哈尔滨工业大学

要点

- 链接
- 案例学习: 库打桩机制

- 两个重要工具
  - objdump

    http://manpages.ubuntu.com/manpages/hirsute/en/man1/objdump.1.html
  - readelf

    http://manpages.ubuntu.com/manpages/hirsute/en/man1/readelf.1.html

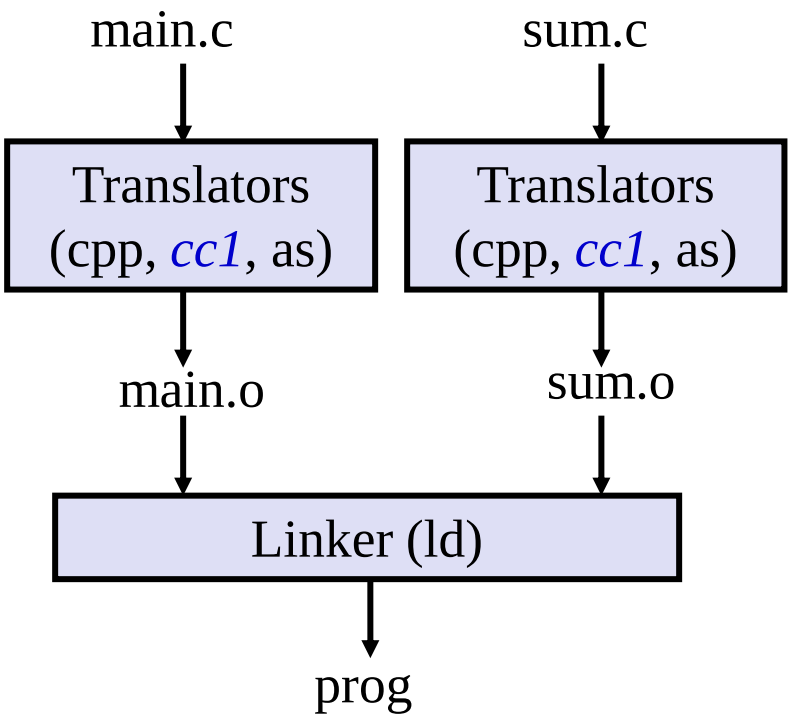
C程序例子

```
//main.c
int sum(int *a, int n);

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

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

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

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

静态链接

- 使用编译器驱动程序(*compiler driver*)进行程序的翻译和链接:
  - **linux> *gcc -Og -o prog main.c sum.c***
  - **linux> *./prog***



源程序

分开编译成可重定位目标文件

完全链接的可执行的目标文件
(包括main.c与sum.cd定义的所有函数的代码和数据)

/usr/lib/gcc/x86_64-linux-gnu/7/cc1

为什么用链接器?

- 理由 1: 模块化
  - 程序可以编写为一个较小的源文件的集合，而不是一个整体巨大的一团。
  - 可以构建公共函数库 (稍后详述)
    - 例如：数学运算库, 标准C库

为什么用链接器?(cont)

- 理由 2: 效率
  - 时间: 分开编译
    - 更改一个源文件，编译，然后重新链接。
    - 不需要重新编译其他源文件。
  - 空间: 库
    - 可以将公共函数聚合为单个文件...
    - 而可执行文件和运行内存映像只包含它们实际使用的函数的代码.

链接器如果工作

- 步骤 1:符号解析
- 步骤 2: 重定位

链接步骤 1:符号解析

- 程序定义和引用符号 (全局变量和函数):
  - **void swap() {…} /* 定义(define)符号swap */**
  - **swap(); /* 引用(reference)符号swap */**
  - **int *xp = &x; /*定义符号xp,引用符号x */**
- 由汇编器将符号定义存储在目标文件中的符号表中
  - 符号表是一个结构体的数组
  - 每个条目包括符号的名称、大小和位置
  - 汇编器生成符号表!
- 符号解析：链接器将每个符号引用与一个确定的符号定义关联起来

链接步骤 2: 重定位

- 将多个单独的代码节(sections)和数据节合并为单个节。
- 将符号从它们在.o文件中的相对位置重新定位到可执行文件中的最终绝对内存位置。
- 用它们的新位置，更新所有对这些符号的引用。

我们将详细地介绍这两个步骤....

三种目标文件(模块)

- 可重定位目标文件(.o 文件)
  - 包含的代码和数据，其形式能与其他可重定位目标文件相结合，以形成可执行的目标文件。
    - 每一个.o文件是由一个源(.c)文件生成的
- 可执行目标文件(a.out 文件)
  - 包含的代码和数据，其形式可以直接复制到内存并执行。
- 共享目标文件(.so 文件)
  - 特殊类型的可重定位目标文件，它可以在加载时或运行时，动态地加载到内存并链接。
  - 在Windows中称为动态链接库(Dynamic Link Libraries, DLL)

可执行与可链接格式(ELF)

- Executable and Linkable Format， ELF
  - X86-64 Linux 、Unix系统
  - 目标文件的标准二进制格式

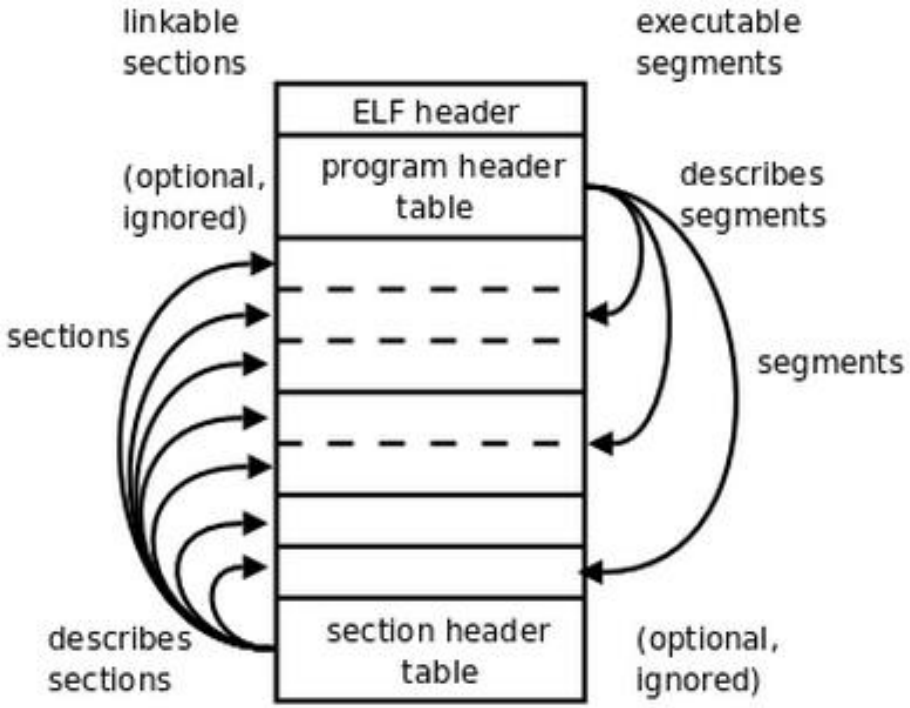
- 三种目标文件的统一格式：
  - 可重定位目标文件(.o)
  - 可执行目标文件(a.out)
  - 共享目标文件(.so)

- 通用名字: ELF二进制文件

ELF目标文件格式

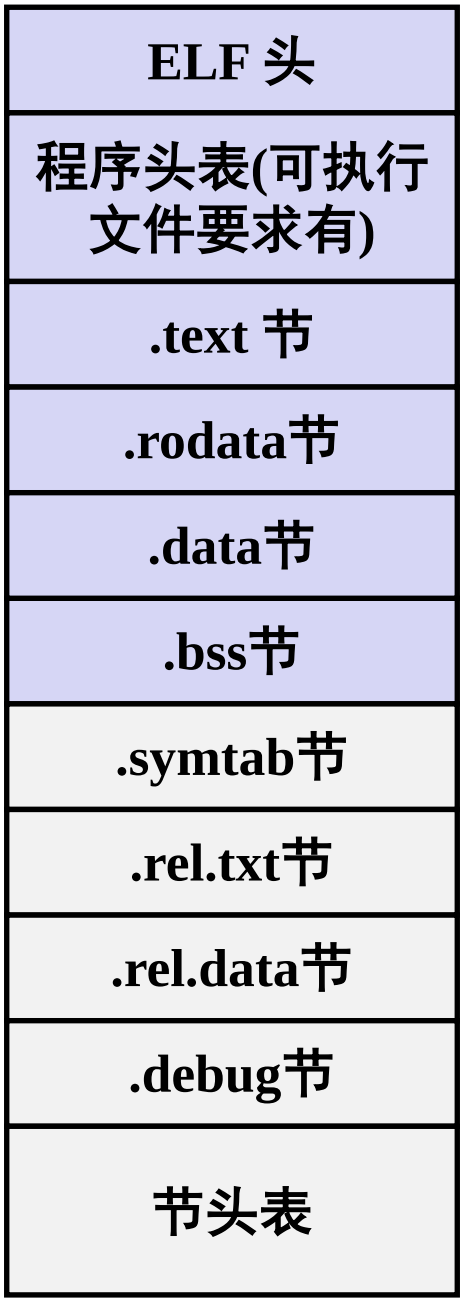
- Executable and Linkable Format， ELF
  - 链接视图、执行视图

链接视图	执行视图
ELF 头部	ELF 头部
程序头部表（可选）	程序头部表
节区 1	段 1
...	
节区 n	段 2
...	
...	...
节区头部表	节区头部表（可选）



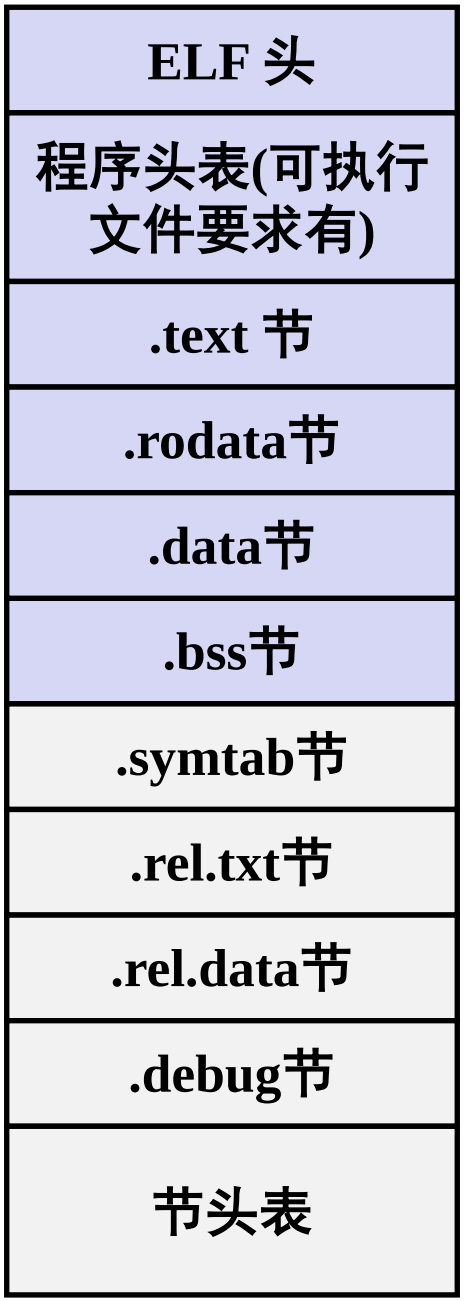
ELF目标文件格式

- **ELF 头**: readelf -*l* main查看
  - 字大小、字节序：16字节
  - 文件类型(.o，exec，.so)、机器类型、节头表的位置、条目大小、数量等
- 段头表/程序头表
  - 页面大小，虚拟地址内存段(节)，段大小
- **.text 节**（代码）
  - 代码
- **.rodata 节**（只读数据）
  - 只读数据： printf的格式串、跳转表, ...
- **.data 节** （数据/可读写）
  - 已初始化全局和静态变量
- **.bss 节** （未初始化全局变量）
  - 未初始化/初始化为0的全局和静态变量
  - 仅有节头，但节本身不占用磁盘空间



ELF目标文件格式(cont.)

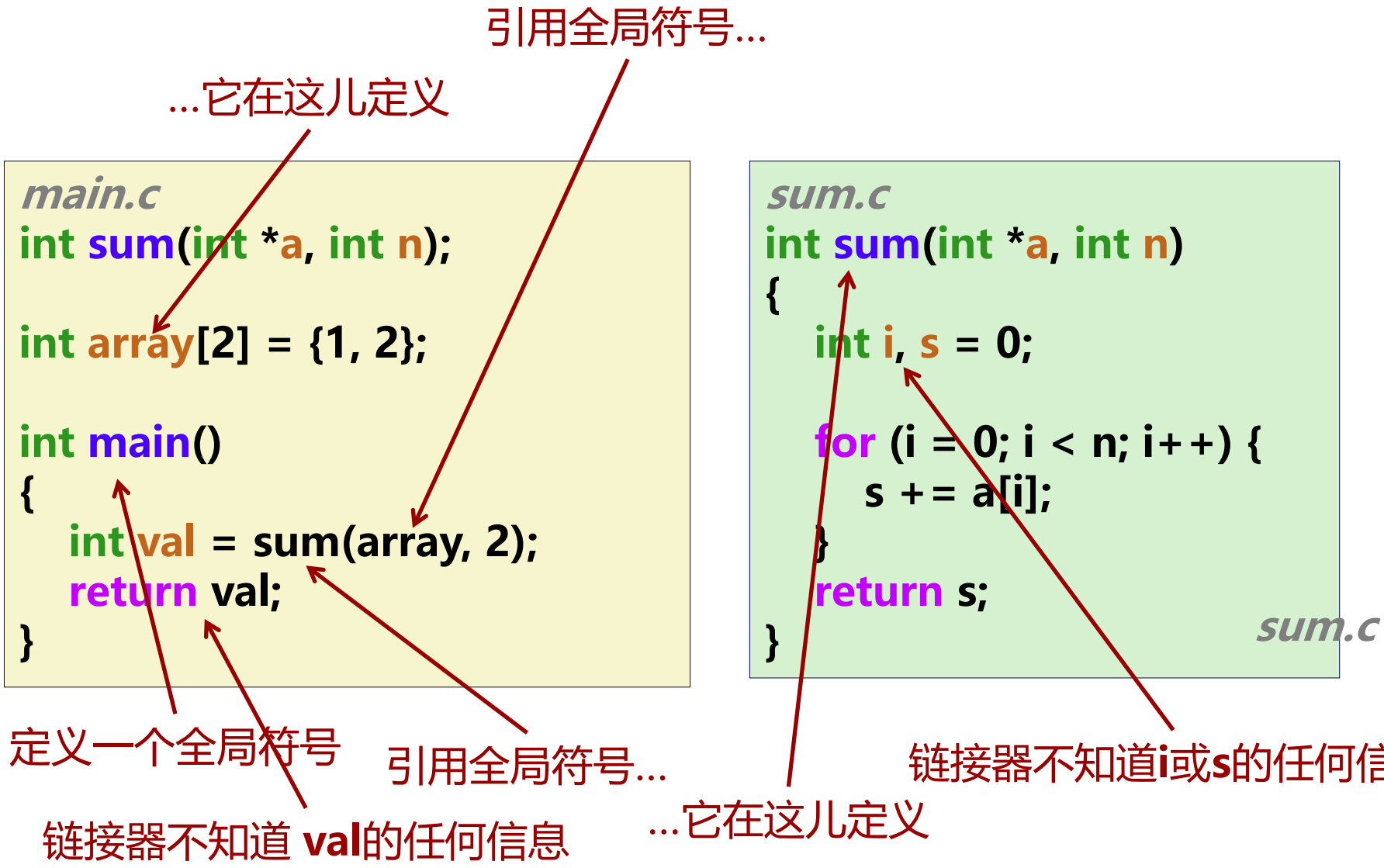
- **.symtab 节（符号表）**
  - 函数和全局/静态变量名
  - 节名称和位置
- **.rel.text 节（可重定位代码）**
  - .text 节的可重定位信息
  - 在可执行文件中需要修改的指令地址
  - 需修改的指令
- **.rel.data 节（可重定位数据）**
  - **.data 节的可重定位信息**
  - 在合并后的可执行文件中需要修改的指针数据的地址
- **.debug 节（调试符号表）**
  - 为符号调试的信息 (**gcc -g**)
- 节头表**Section header table**
  - 每个节的在文件中的偏移量、大小等



链接器符号

- 全局符号
  - 由模块m定义的，可以被其他模块引用的符号。
  - 例如: 非静态(non-static) C 函数与非静态全局变量。
- 外部符号
  - 由模块m引用的全局符号，但由其他模块定义。
- 本地/局部符号
  - 由模块m定义、并仅由m引用的符号。
    - 如: 带static属性的C函数和全局变量。
  - **本地链接器符号不是程序的局部变量。**

链接步骤 1: 符号解析


引用全局符号...
...它在这儿定义
main.c
int sum(int *a, int n);
int array[2] = {1, 2};
int main()
{
int val = sum(array, 2);
return val;
}
sum.c
int sum(int *a, int n)
{
int i, s = 0;
for (i = 0; i < n; i++) {
s += a[i];
}
return s;
}
sum.c
定义一个全局符号
引用全局符号...
链接器不知道i或s的任何信息
链接器不知道 val的任何信息
...它在这儿定义

局部符号

- 局部/本地非静态C变量 vs. 局部/本地静态C变量
  - 局部非静态C变量： 存储在栈上
  - 局部静态C变量： 存储在 **.bss** 或 **.data**

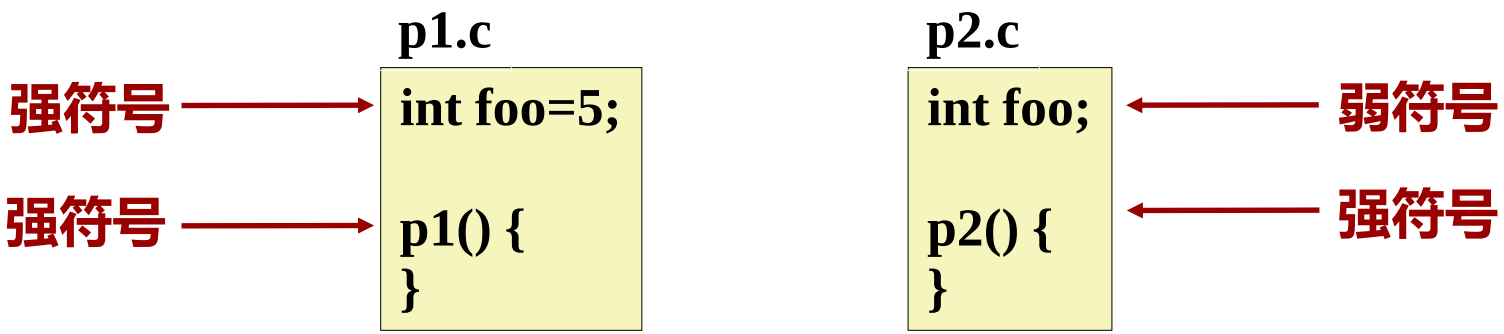
```
int f()
{
    static int x = 0;
    return x;
}

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

编译器在.data为每个x的定义分配空间。

在符号表中创建具有唯一名称的局部符号（本地链接器符号），如 x.1 和 x.2

链接器如何解析重复的符号定义

- 程序符号要么是强符号，要么是弱符号
  - *强*：函数和初始化全局变量
  - *弱*：未初始化的全局变量



链接器的符号处理规则

- 规则 1:不允许多个同名的强符号
  - 每个强符号只能定义一次
  - 否则: 链接器错误

- 规则 2:若有一个强符号和多个弱符号同名，则选择强符号
  - **对弱符号的引用将被解析为强符号**

- 规则 3:如果有多个弱符号，选择任意一个
  - `gcc -fno-common`：对多重定义的全局符号报错
  - `gcc -Werror`：所有警告都变为错误

链接器谜题

int x; p1() {}	p1() {}	链接时错误:两个强符号(p1)
int x; p1() {}	int x; p2() {}	对 x 的引用将指向同一个未初始化的 int x 你真想这样？
int x; int y; p1() {}	double x; p2() {}	在p2中如对x写入，可能覆盖y！邪恶!

int x=7;
int y=5;
p1() {}
	double x;
	p2() {}
		在p2中如对x写入，必将覆盖y！作恶!

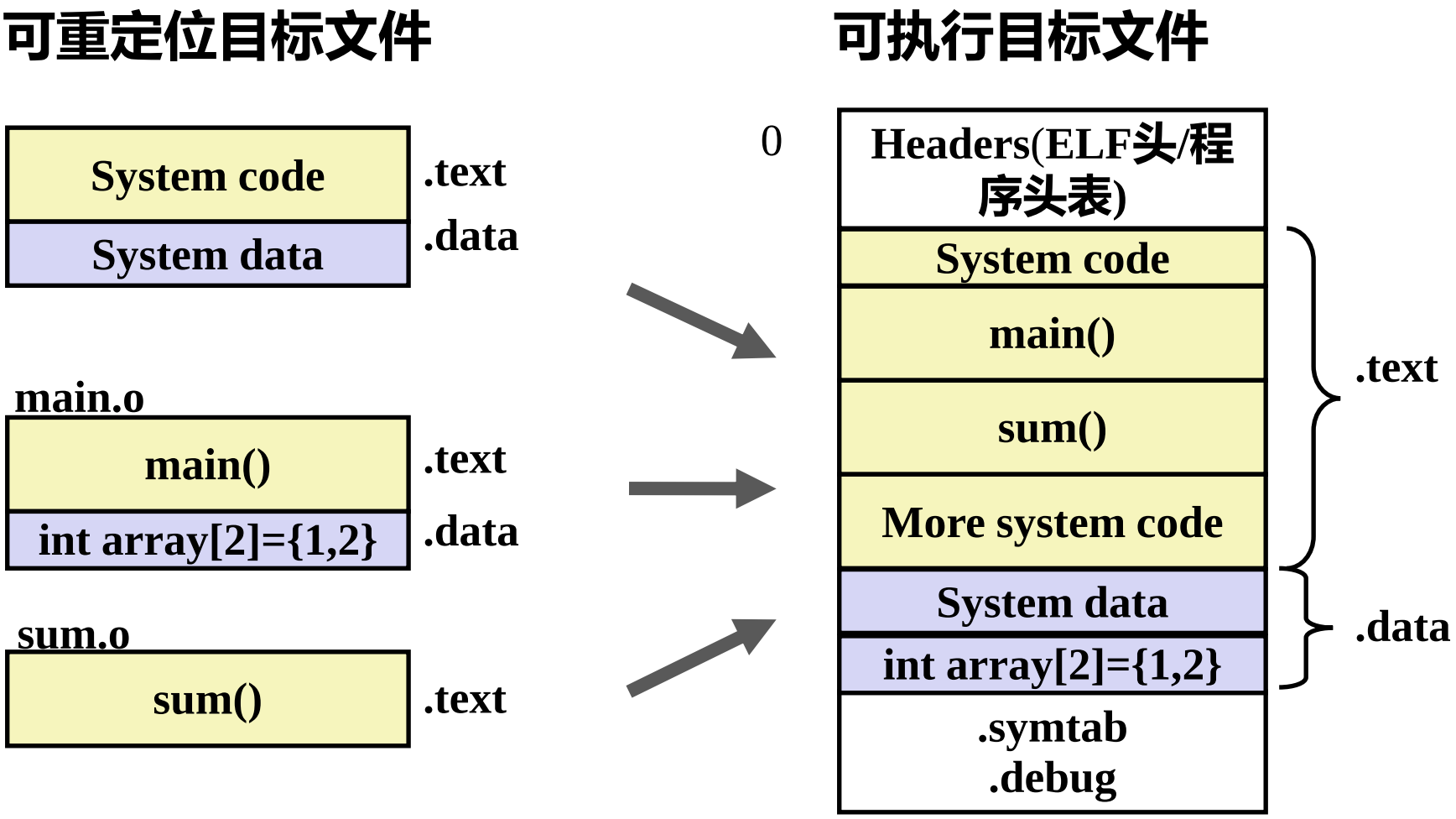
int x=7; p1() {}	int x; p2() {}	对x的引用将指向相同的初始化变量

噩梦场景:两个同名弱符号，由不同的编译器来编译会采用不同的排列规则.

全局变量

- 避免：如果能的话
- 否则
  - 如果可以，使用 **`static`**
  - 定义时初始化它
  - 使用 **`extern`** 声明引用的外部全局符号

链接步骤 2: 重定位


可重定位目标文件
可执行目标文件
System code
System data
.text
.data
main.o
main()
int array[2]={1,2}
.text
.data
sum.o
sum()
.text
0
Headers(ELF头/程序头表)
System code
main()
sum()
More system code
System data
int array[2]={1,2}
.symtab
.debug
.text
.data

可重定位条目

```
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        # 可重定位条目

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

来源: `objdump -rd main.o`

重定位算法

ADDR(r.symbol) - (refaddr - r.addend);

```
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){//PC相对寻址的引用
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)//使用32位绝对地址
13       *refptr = (unsigned)(ADDR(r.symbol) + r.addend);
14   }
15 }
```

ADDR(s)：节s的运行时地址
ADDR(r.symbol)重定位条目r的符号symbol的运行时地址

重定位计算示例#1

- **readelf -r main.o　　//读出main.o中需要重定位的信息**

重定位节 '.rela.text' 位于偏移量 0x1e8 含有 2 个条目：

偏移量	信息	类型	符号值	符号名称 + 加数
00000000000a	00090000000a	R_X86_64_32	0000000000000000	array + 0
00000000000f	000a00000002	R_X86_64_PC32	0000000000000000	sum **– 4**

English：

Relocation section '.rela.text' at offset 0x1e8 contains 2 entries:

Offset	Info	Type	Sym. Value	Sym. Name + Addend
00000000000a	00090000000a	R_X86_64_32	0000000000000000	array +0
00000000000f	000a00000002	R_X86_64_PC32	0000000000000000	sum **- 4**

重定位PC相对引用sum

- 重定位条目信息
  - r.offset = 0xf
  - r.symbol = sum
  - r.type = R_X86_64_PC32
  - r.addend = -4
- 假设链接器已经确定
  - ADDR(s)=ADDR(.text)=0x4004d0
  - ADDR(r.symbol) = ADDR(sum)=0x4004e8
- 重定位
  - refaddr = ADDR(s) + r.offset = 0x4004d0 + 0xf = 0x4004df
  - *refptr = (unsigned)(ADDR(r.symbol) + r.addend - refaddr)
    =(unsigned)(0x4004e8 + (-4) - 0x4004df)
    =(unsigned)(0x5)

重定位绝对引用array

- 重定位条目信息
  - r.offset = 0xa
  - r.symbol = array
  - r.type = R_X86_64_32
  - r.addend = 0
- 假设链接器已经确定
  - ADDR(s)=ADDR(.text)=0x4004d0
  - ADDR(r.symbol) = ADDR(array)=0x601018
- 重定位
  - *refptr = (unsigned)(ADDR(r.symbol) + r.addend)
=(unsigned)(0x601018 + 0)
=(unsigned)(0x601018)

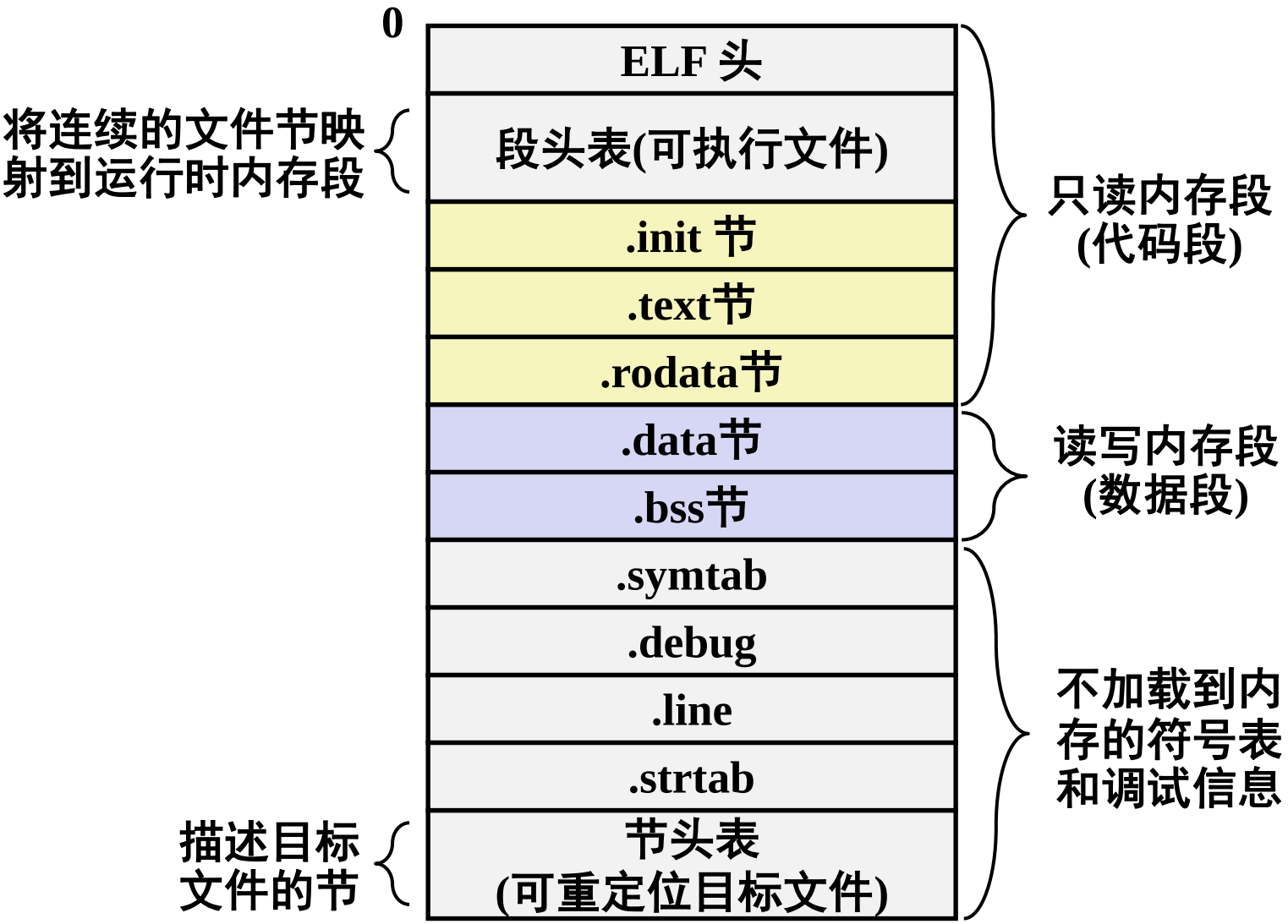
已经重定位的 .text 节

来源: objdump -dx -j .text prog

```
0000000004004d0 <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

0000000004004e8 <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
```

sum()的调用指令使用PC相对寻址： 0x4004e8 = 0x4004e3 + 0x5

可执行目标文件


0
ELF 头
段头表(可执行文件)
.init 节
.text节
.rodata节
.data节
.bss节
.symtab
.debug
.line
.strtab
节头表
(可重定位目标文件)
将连续的文件节映射到运行时内存段
描述目标文件的节
只读内存段(代码段)
读写内存段(数据段)
不加载到内存的符号表和调试信息

可执行目标文件

vaddr 有对齐要求：
vaddr mod align = off mod align

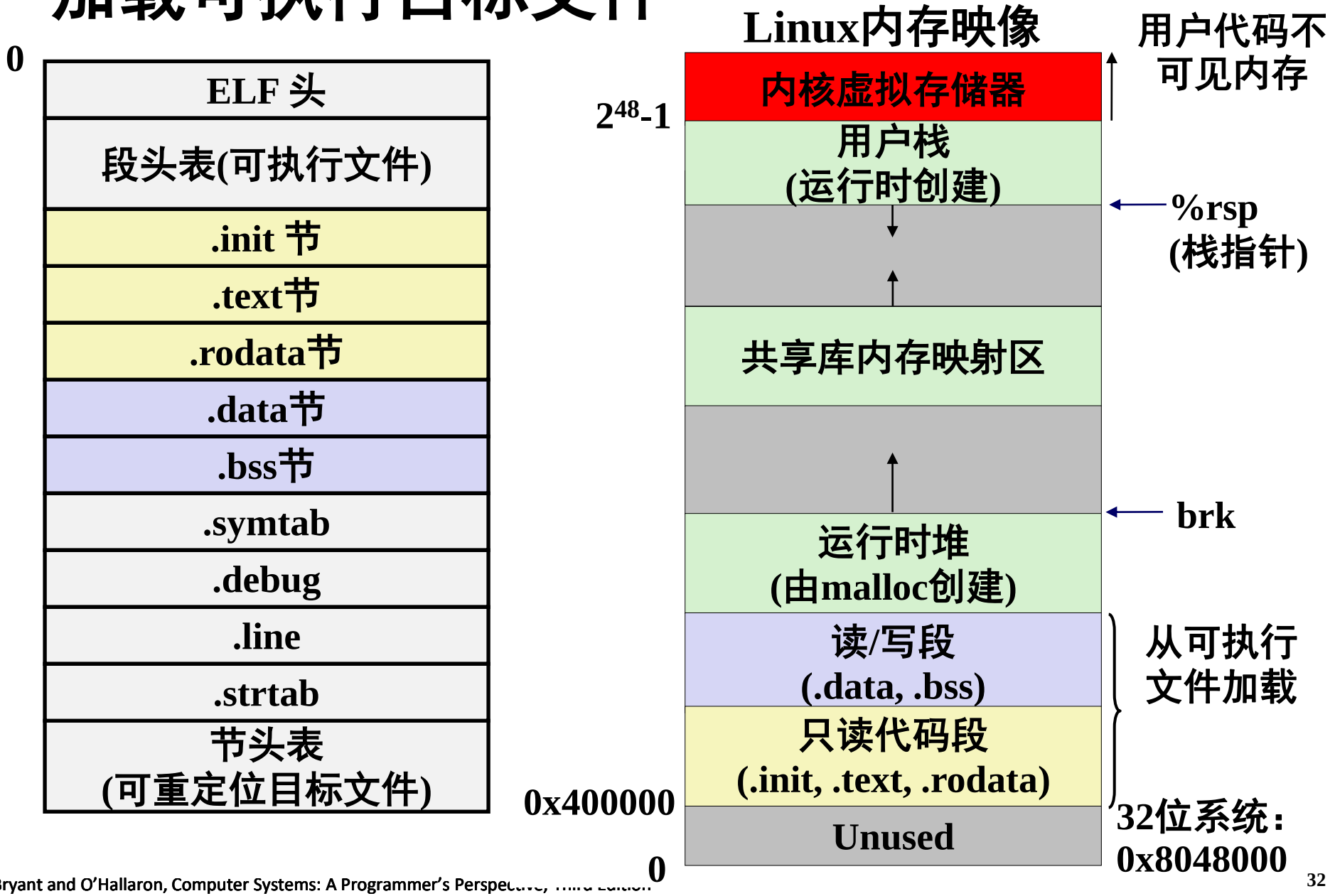
- 程序头部表(program header table)
  - objdump -dx prog > prog_dump.txt

```
Program Header:
PHDR off    0x0000000000000040 vaddr 0x0000000000400040 paddr 0x0000000000400040 align 2**3
      filesz 0x00000000000001f8 memsz 0x00000000000001f8 flags r-x
INTERP off    0x0000000000000238 vaddr 0x0000000000400238 paddr 0x0000000000400238 align 2**0
      filesz 0x000000000000001c memsz 0x000000000000001c flags r--
LOAD off    0x0000000000000000 vaddr 0x0000000000400000 paddr 0x0000000000400000 align 2**21
      filesz 0x000000000000069c memsz 0x000000000000069c flags r-x
LOAD off    0x0000000000000df8 vaddr 0x0000000000600df8 paddr 0x0000000000600df8 align 2**21
      filesz 0x0000000000000228 memsz 0x0000000000000230 flags rw-
DYNAMIC off    0x0000000000000e10 vaddr 0x0000000000600e10 paddr 0x0000000000600e10 align 2**3
      filesz 0x00000000000001d0 memsz 0x00000000000001d0 flags rw-
NOTE off    0x0000000000000254 vaddr 0x0000000000400254 paddr 0x0000000000400254 align 2**2
      filesz 0x0000000000000044 memsz 0x0000000000000044 flags r--
EH_FRAME off    0x00000000000005b4 vaddr 0x00000000004005b4 paddr 0x00000000004005b4 align 2**2
      filesz 0x0000000000000034 memsz 0x0000000000000034 flags r--   STACK off
```

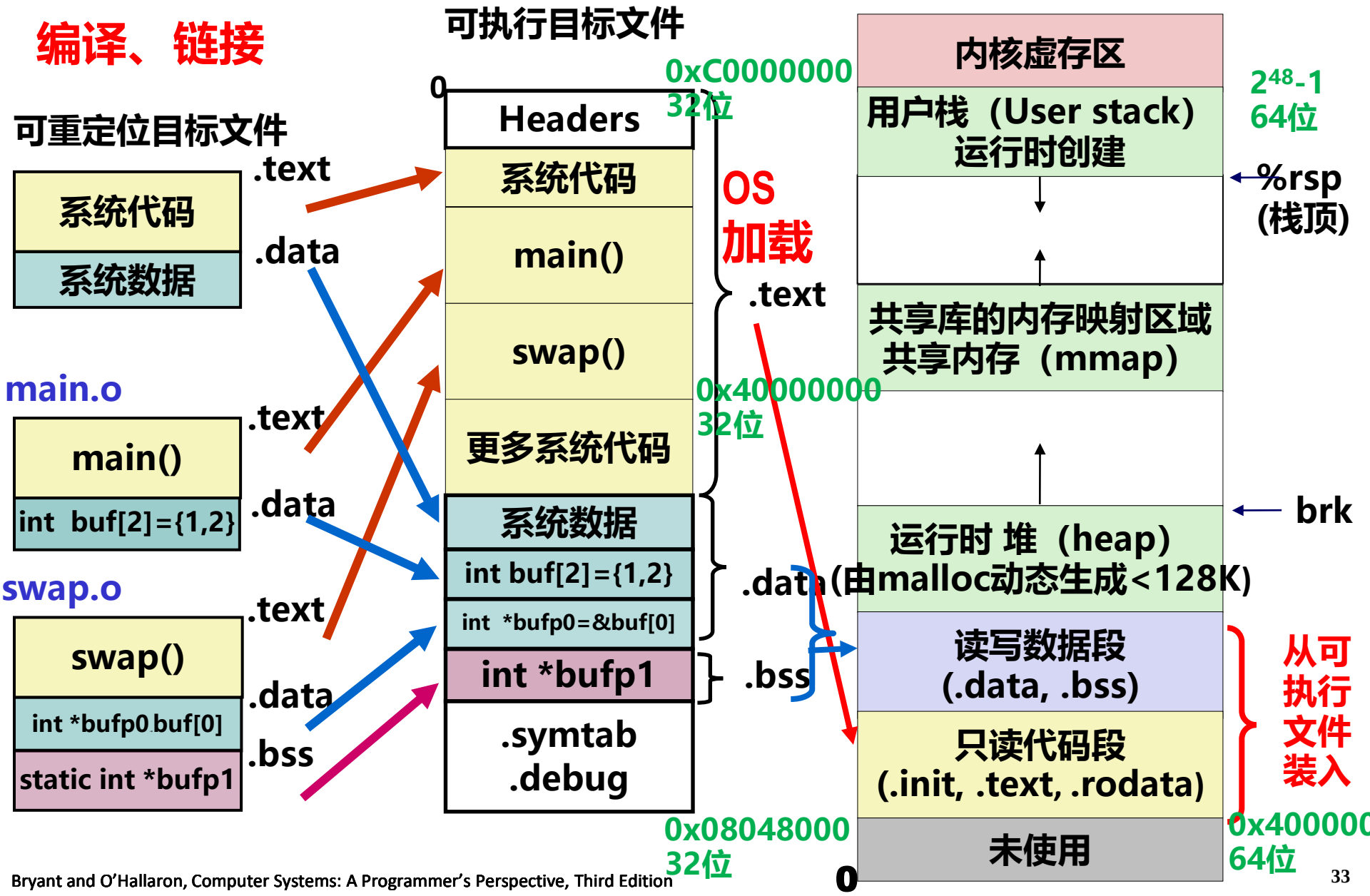
可执行目标文件

- 传统：可执行程序载入内存的位置为0x400000（64位程序）或0x8048000（32位程序）
- 缺点：易受攻击
- 改进：现代链接器都是非固定地址的连接，程序可以加载到内存任意位置
  - readelf看到的信息是**text段的vaddr为0**，这样在加载时使用动态地址，可以防止攻击。
  - 最新链接器默认采用这种编译方式
  - 关闭这个功能：连接时加-no-pie选项
    - 否则： **text段的vaddr为0**

加载可执行目标文件


0
ELF 头
段头表(可执行文件)
.init 节
.text节
.rodata节
.data节
.bss节
.symtab
.debug
.line
.strtab
节头表
(可重定位目标文件)
Linux内存映像
用户代码不
可见内存
2^48-1
内核虚拟存储器
用户栈
(运行时创建)
%rsp
(栈指针)
共享库内存映射区
brk
运行时堆
(由malloc创建)
读/写段
(.data, .bss)
只读代码段
(.init, .text, .rodata)
从可执行
文件加载
0x400000
Unused
32位系统:
0x8048000
0

源程序、目标文件、执行程序、虚拟内存映像



链接实例：

```
/* main.c */
#include <stdio.h>
#include "vector.h"
int a;
int x[2] = {1, 2};
int y[2] = {3, 4};
static int z[2];
static int w[2]= {5, 6};
extern int addcnt;

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

```
/* addvec.c */
int addcnt = 0;
int addcnt_1 = 0x563412;
int z[2];

void addvec(int *x, int *y,
            int *z, int n)
{
    int i;
    addcnt++;
    addcnt_1++;
    for (i = 0; i < n; i++)
        z[i] = x[i] + y[i];
}
```

```
gcc -m64 -no-pie -fno-PIC -c addvec.c main.c
gcc -m64 -no-pie -fno-PIC -o prog addvec.o main.o
```

链接实例：

objdump -dxs addvec.o > addvec.d

```
addvec.o:     file format elf64-x86-64
addvec.o
architecture: i386:x86-64, flags 0x00000011:
HAS_RELOC, HAS_SYMS
start address 0x0000000000000000
Sections:
Idx Name          Size      VMA               LMA               File off  Algn
  0 .text         0000008d  0000000000000000  0000000000000000  00000040  2**0
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1 .data         00000004  0000000000000000  0000000000000000  000000d0  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000004  0000000000000000  0000000000000000  000000d4  2**2
                  ALLOC
  3 .comment      0000002b  0000000000000000  0000000000000000  000000d4  2**0
                  CONTENTS, READONLY
```

链接实例：

objdump -dxs addvec.o > addvec.d

```
SYMBOL TABLE:
0000000000000000 l    df *ABS*    0000000000000000 addvec.c
0000000000000000 l    d  .text    0000000000000000 .text
0000000000000000 l    d  .data    0000000000000000 .data
0000000000000000 l    d  .bss     0000000000000000 .bss
0000000000000000 g     O .bss     0000000000000004 addcnt
0000000000000000 g     O .data    0000000000000004 addcnt_1
0000000000000008       O *COM*    0000000000000008 z
0000000000000000 g     F .text    000000000000008d addvec

Contents of section .data:
 0000 12345600                             .4V.
```

链接实例：

objdump -dxs main.o > main.d

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

Sections:Idx Name          Size      VMA               LMA               File off  Algn  0
.text         00000048  0000000000000000  0000000000000000  00000040  2**0
              CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
1 .data       00000018  0000000000000000  0000000000000000  00000088  2**3
              CONTENTS, ALLOC, LOAD, DATA
2 .bss        00000008  0000000000000000  0000000000000000  000000a0  2**3
              ALLOC
3 .rodata     00000017  0000000000000000  0000000000000000  000000a0  2**0
              CONTENTS, ALLOC, LOAD, READONLY, DATA
```

local (l), global (g), unique global (u), neither global nor local (a space) or both global and local (!)

```
SYMBOL TABLE:
0000000000000000 l    df *ABS*  0000000000000000 main.c
0000000000000000 l    d  .text  0000000000000000 .text
0000000000000000 l    d  .data  0000000000000000 .data
0000000000000000 l    d  .bss   0000000000000000 .bss
0000000000000000 l    O .bss    0000000000000008 z
0000000000000010 l    O .data   0000000000000008 w
0000000000000000 l    d  .rodata 0000000000000000 .rodata
0000000000000000 l    d  .note.GNU-stack 0000000000000000 .note.GNU-stack
0000000000000000 l    d  .eh_frame       0000000000000000 .eh_frame
0000000000000000 l    d  .comment        0000000000000000 .comment
0000000000000004       O *COM*  0000000000000004 a
0000000000000000 g    O .data   0000000000000008 x
0000000000000008 g    O .data   0000000000000008 y
0000000000000000 g     F .text  0000000000000048 main
0000000000000000         *UND*  0000000000000000 addvec
0000000000000000         *UND*  0000000000000000 addcnt
0000000000000000         *UND*  0000000000000000 printf
```

链接实例： main.o

objdump -dxs main.o > main.d

```
Contents of section .text:
 0000 554889e5 b9020000 00ba0000 0000be00  UH..............
 0010 000000bf 00000000 e8000000 008b0d00  ................
 0020 0000008b 15000000 008b0500 00000089  ................
 0030 c6bf0000 0000b800 000000e8 00000000  ................
 0040 8b050000 00005dc3                    ......].
Contents of section .data:
 0000 01000000 02000000 03000000 04000000  ................
 0010 05000000 06000000                    ........
Contents of section .rodata:
 0000 7a203d20 5b256420 25645d20 61646463  z = [%d %d] addc
 0010 6e743d25 640a00                      nt=%d..

Contents of section .comment:
 0000 00474343 3a202855 62756e74 7520372e  .GCC: (Ubuntu 7.
 0010 332e302d 32377562 756e7475 317e3138  3.0-27ubuntu1~18
 0020 2e303429 20372e33 2e3000             .04) 7.3.0.
....
```

objdump -dxs main.o > main.d

```
Disassembly of section .text:
0000000000000000 <main>:
   0:   55                      push   %rbp
   1:   48 89 e5                mov    %rsp,%rbp
   4:   b9 02 00 00 00                 mov    $0x2,%ecx
   9:   ba 00 00 00 00                 mov    $0x0,%edx
                        a: R_X86_64_32 .bss
   e:   be 00 00 00 00                 mov    $0x0,%esi
                        f: R_X86_64_32 y
  13:   bf 00 00 00 00                 mov    $0x0,%edi
                        14: R_X86_64_32         x
  18:   e8 00 00 00 00                 callq  1d <main+0x1d>
                        19: R_X86_64_PC32       addvec-0x4
  1d:   8b 0d 00 00 00 00              mov    0x0(%rip),%ecx        # 23 <main+0x23>
                        1f: R_X86_64_PC32       addcnt-0x4
  23:   8b 15 00 00 00 00              mov    0x0(%rip),%edx        # 29 <main+0x29>
                        25: R_X86_64_PC32       .bss
  29:   8b 05 00 00 00 00              mov    0x0(%rip),%eax        # 2f <main+0x2f>
                        2b: R_X86_64_PC32       .bss-0x4
  2f:   89 c6                  mov    %eax,%esi
  31:   bf 00 00 00 00                 mov    $0x0,%edi
                        32: R_X86_64_32         .rodata
  36:   b8 00 00 00 00                 mov    $0x0,%eax
  3b:   e8 00 00 00 00                 callq  40 <main+0x40>
                        3c: R_X86_64_PC32       printf-0x4
  40:   8b 05 00 00 00 00              mov    0x0(%rip),%eax        # 46 <main+0x46>
                        42: R_X86_64_PC32       a-0x4
  46:   5d                             pop    %rbp
  47:   c3                             retq
```

```
Disassembly of section .text:
0000000000000000 <main>:
   0:   55                      push   %rbp
   1:   48 89 e5                mov    %rsp,%rbp
   4:   b9 02 00 00 00                 mov    $0x2,%ecx
   9:   ba 00 00 00 00                 mov    $0x0,%edx
                        a: R_X86_64_32 .bss
   e:   be 00 00 00 00                 mov    $0x0,%esi
                        f: R_X86_64_32 y
  13:   bf 00 00 00 00                 mov    $0x0,%edi
                        14: R_X86_64_32         x
  18:   e8 00 00 00 00                 callq  1d <main+0x1d>
                        19: R_X86_64_PC32       addvec-0x4
  1d:   8b 0d 00 00 00 00              mov    0x0(%rip),%ecx        # 23 <ma
                        1f: R_X86_64_PC32       addcnt-0x4
  23:   8b 15 00 00 00 00              mov    0x0(%rip),%edx        # 29
                        25: R_X86_64_PC32       .bss
  29:   8b 05 00 00 00 00              mov    0x0(%rip),%eax        # 2f
                        2b: R_X86_64_PC32       .bss-0x4
  2f:   89 c6                  mov    %eax,%esi
  31:   bf 00 00 00 00                 mov    $0x0,%edi
                        32: R_X86_64_32         .rodata
  36:   b8 00 00 00 00                 mov    $0x0,%eax
  3b:   e8 00 00 00 00                 callq  40 <main+0x40>
                        3c: R_X86_64_PC32       printf-0x4
  40:   8b 05 00 00 00 00              mov    0x0(%rip),%eax        # 46 <main+0x46>
                        42: R_X86_64_PC32       a-0x4
  46:   5d                             pop    %rbp
  47:   c3                             retq
```

objdump -dxs main.o > main.d

链接时：需要确定红色部分的数值！

bss节的static int z

y

x

函数addvec

addcnt

z[1]

z[0]

printf的格式串

函数printf

int a

链接实例： prog

objdump -dxs prog > prog.d

```
prog:     file format elf64-x86-64
architecture: i386:x86-64, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x0000000000400400
...
Sections:
Idx Name          Size      VMA               LMA               File off  Algn
 10 .init         00000017  00000000004003c8  00000000004003c8  000003c8  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 11 .plt          00000020  00000000004003e0  00000000004003e0  000003e0  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 12 .text         00000232  0000000000400400  0000000000400400  00000400  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 14 .rodata       0000001b  0000000000400640  0000000000400640  00000640  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 20 .got          00000010  0000000000600ff0  0000000000600ff0  00000ff0  2**3
                  CONTENTS, ALLOC, LOAD, DATA
 21 .got.plt      00000020  0000000000601000  0000000000601000  00001000  2**3
                  CONTENTS, ALLOC, LOAD, DATA
 22 .data         00000030  0000000000601020  0000000000601020  00001020  2**3
                  CONTENTS, ALLOC, LOAD, DATA
 23 .bss          00000020  0000000000601050  0000000000601050  00001050  2**3
                  ALLOC
```

链接实例： prog

objdump -dxs prog > prog.d

```
Contents of section .rodata:
 400640 01000200 7a203d20 5b256420 25645d20  ....z = [%d %d]
 400650 61646463 6e743d25 640a00             addcnt=%d..
```

格式字符串的起始地址：0x400644

```
Contents of section .data:
 601020 00000000 00000000 00000000 00000000  ................
 601030 12345600 00000000 01000000 02000000  .4V.............
 601040 03000000 04000000 05000000 06000000  ................

Disassembly of section .text:
00000000004004e7 <addvec>:
  4004e7:       55                      push   %rbp
  4004e8:       48 89 e5                mov    %rsp,%rbp
  4004eb:       48 89 7d e8             mov    %rdi,-0x18(%rbp)
```

链接实例

objdump -dxs prog > prog.d

```
SYMBOL TABLE:
0000000000400238 l    d  .interp    0000000000000000              .interp
0000000004003c8 l    d  .init      0000000000000000              .init
0000000004003e0 l    d  .plt       0000000000000000              .plt
0000000000400400 l    d  .text      0000000000000000              .text
0000000000400640 l    d  .rodata    0000000000000000              .rodata
0000000000600e20 l    d  .dynamic   0000000000000000              .dynamic
0000000000600ff0  l    d  .got       0000000000000000              .got
0000000000601000 l    d  .got.plt   0000000000000000              .got.plt
0000000000601020 l    d  .data      0000000000000000              .data
0000000000601050 l    d  .bss       0000000000000000              .bss
0000000000601058 l    O .bss        0000000000000008              z
0000000000601048 l    O .data       0000000000000008              w
0000000000601060 g    O .bss        0000000000000008              z
0000000000601038 g    O .data       0000000000000008              x
0000000000000000      F *UND*       0000000000000000              printf@@GLIBC_2.2.5
0000000000000000      F *UND*       0000000000000000              __libc_start_main@@GLIBC_2.2.5
00000000004004e7 g     F .text      000000000000008d              addvec
0000000000601030 g     O .data      0000000000000004              addcnt_1
0000000000400400 g     F .text      000000000000002b              _start
0000000000601068 g     O .bss       0000000000000004              a
0000000000601040 g     O .data      0000000000000008              y
0000000000400574 g     F .text      0000000000000047              main
0000000000601054 g     O .bss       0000000000000004              addcnt
```

```
0000000000400574 <main>:
 400574:  55                    push   %rbp
 400575:  48 89 e5              mov    %rsp,%rbp
 400578:  b9 02 00 00 00        mov    $0x2,%ecx
 40057d:  ba 00 00 00 00        mov    $0x0,%edx
                                a: R_X86_64_32 .bss
 400582 : be 00 00 00 00                mov    $0x0,%esi
                                f: R_X86_64_32     y
 400587 : bf 00 00 00 00                mov    $0x0,%edi
                                14: R_X86_64_32   x
 40058c : e8 00 00 00 00                callq  1d <main+0x1d>
                                19: R_X86_64_PC32       addvec-0x4
 400591 : 8b 0d 00 00 00 00     mov    0x0(%rip),%ecx  # 23<main+0x2
                                1f: R_X86_64_PC32       addcnt-0x4
 400597 : 8b 15 00 00 00 00     mov    0x0(%rip),%edx   # 29<main+
                                25: R_X86_64_PC32       .bss
 40059d: 8b 05 00 00 00 00      mov    0x0(%rip),%eax   # 2f <main+
                                2b: R_X86_64_PC32       .bss-0x4
 4005a3: 89 c6                   mov    %eax,%esi
 4005a5: bf 00 00 00 00          mov    $0x0,%edi
                                32: R_X86_64_32         .rodata
 4005aa: b8 00 00 00 00         mov    $0x0,%eax
 4005af:  e8 00 00 00 00         callq  40 <main+0x40>
                                3c: R_X86_64_PC32       printf-0x4
 4005b4: 8b 05 00 00 00 00      mov    0x0(%rip),%eax        # 46 <main+0x46>
                                42: R_X86_64_PC32       a-0x4
 4005ba: 5d                         pop    %rbp
 4005bb: c3                         retq
 4005bc: 0f 1f 40 00                nopl   0x0(%rax)
```

已知所有符号的地址(包括main)

确定引用、红色部分

bss节的static int z

y

x

函数addvec

addcnt

z[1]

z[0]

printf的格式串

函数printf

int a

链接实例：prog 的重定位

```
0000000000601058 l    O .bss     0000000000000008              z
0000000000601060 g    O .bss     0000000000000008              z
0000000000601038 g    O .data    0000000000000008              x
00000000004004e7 g     F .text   000000000000008d              addvec
0000000000601030 g     O .data   0000000000000004              addcnt_1
0000000000601068 g     O .bss    0000000000000004              a
0000000000601040 g     O .data   0000000000000008              y
0000000000601054 g     O .bss    0000000000000004              addcnt
```

```
Disassembly of section .plt:

00000000004003e0 <.plt>:
4003e0: ff 35 22 0c 20 00  pushq  0x200c22(%rip) #601008<_GLOBAL_OFFSET_TABLE_+0x8>
4003e6: ff 25 24 0c 20 00  jmpq *0x200c24(%rip) #601010<_GLOBAL_OFFSET_TABLE_+0x10>
4003ec:  0f 1f 40 00   nopl   0x0(%rax)

00000000004003f0 <printf@plt>:
  4003f0: ff 25 22 0c 20 00   jmpq   *0x200c22(%rip) # 601018 <printf@GLIBC_2.2.5>
  4003f6: 68 00 00 00 00  pushq  $0x0
  4003fb: e9 e0 ff ff ff      jmpq   4003e0 <.plt>
```

```
0000000000400574 <main>:
 400574:  55                    push   %rbp
 400575:  48 89 e5              mov    %rsp,%rbp
 400578:  b9 02 00 00 00        mov    $0x2,%ecx
 40057d:  ba 00 00 00 00        mov    $0x0,%edx
                                a: R_X86_64_32 .bss
 400582 : be 00 00 00 00                mov    $0x0,%esi
                                f: R_X86_64_32     y
 400587 : bf 00 00 00 00                mov    $0x0,%edi
                                14: R_X86_64_32   x
 40058c : e8 00 00 00 00                callq  1d <main+0x1d>
                                19: R_X86_64_PC32   addvec-0x4
 400591 : 8b 0d 00 00 00 00     mov    0x0(%rip),%ecx  # 23<ma
                                1f: R_X86_64_PC32    addcnt-0x4
 400597 : 8b 15 00 00 00 00     mov    0x0(%rip),%edx   # 29
                                25: R_X86_64_PC32    .bss
 40059d: 8b 05 00 00 00 00      mov    0x0(%rip),%eax   # 2f <
                                2b: R_X86_64_PC32   .bss-0x4
 4005a3: 89 c6                   mov    %eax,%esi
 4005a5: bf 00 00 00 00          mov    $0x0,%edi
                                32: R_X86_64_32    .rodata
 4005aa: b8 00 00 00 00         mov    $0x0,%eax
 4005af:  e8 00 00 00 00         callq  40 <main+0x40>
                                3c: R_X86_64_PC32       printf-0x4
 4005b4: 8b 05 00 00 00 00      mov    0x0(%rip),%eax       # 46 <main+0x46>
                                42: R_X86_64_PC32       a-0x4
 4005ba: 5d                         pop    %rbp
 4005bb: c3                         retq
 4005bc: 0f 1f 40 00                nopl   0x0(%rax)
```

确定引用、红色部分

static int &z:601058

&y:601040

&x:601038

addvec:4004e7

&addcnt:601054

static &z[1] :601058

static &z[0]:60105C

printf的格式串: 0x400644

函数printf:4003f0

&a:601068

链接实例：prog 重定位

- 重定位方法
  - R_X86_64_32

    直接将引用对象的地址按小尾顺序写入待修改字段
  - R_X86_64_PC32

    将引用对象地址 – rip（下一条指令地址）差的补码，以小尾顺序写入待修改字段

链接实例：prog 重定位

objdump -dxs prog > prog.d

```
0000000000400574 <main>:
  400574:  55                    push  %rbp
  400575:  48 89 e5              mov   %rsp,%rbp
  400578:  b9 02 00 00 00        mov   $0x2,%ecx
  40057d:  ba 58 10 60 00        mov   $0x601058,%edx
  400582:  be 40 10 60 00        mov   $0x601040,%esi
  400587:  bf 38 10 60 00        mov   $0x601038,%edi
  40058c:  e8 56 ff  ff   ff     callq 4004e7 <addvec>
  400591:  8b 0d bd 0a 20 00     mov   0x200abd(%rip),%ecx  #601054 <addcnt>
  400597:  8b 15 bf  0a 20 00    mov   0x200abf(%rip),%edx  #60105c <z+0x4>
  40059d:  8b 05 b5 0a 20 00     mov   0x200ab5(%rip),%eax  #601058 <z>
  4005a3:  89 c6                 mov   %eax,%esi
  4005a5:  bf 44 06 40 00        mov   $0x400644,%edi
  4005aa:  b8 00 00 00 00        mov   $0x0,%eax
  4005af:   e8 3c fe  ff  ff     callq 4003f0 <printf@plt>
  4005b4:  8b 05 ae 0a 20 00     mov   0x200aae(%rip),%eax  # 601068 <a>
  4005ba:  5d                    pop   %rbp
  4005bb:  c3                    retq
  4005bc:  0f 1f 40 00           nopl  0x0(%rax)
```

常用函数打包

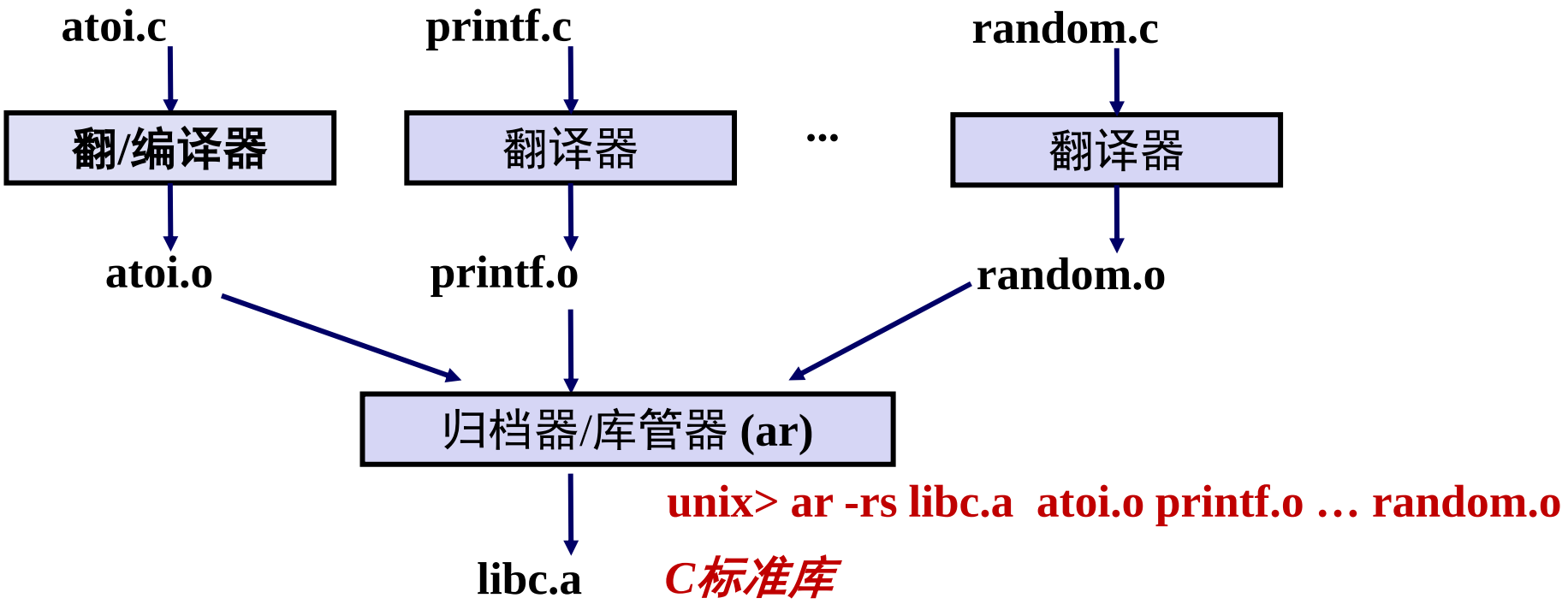
- 如何打包程序员常用的函数?
  - Math、I/O、存储管理、串处理,等等。

- 尴尬，考虑到目前的链接器框架:
  - **选择 1:**将所有函数都放入一个源文件中
    - 程序员将大目标文件链接到他们的程序中
    - 时间和空间效率低下
  - **选择 2:**将每个函数放在一个单独的源文件中
    - 程序员明确地将适当的二进制文件链接到他们的程序中
    - 更高效，但对程序员来说是负担

传统的解决方案:静态库

- 静态库 (.a 存档文件)
  - 将多个相关的可重定位目标文件连接到一个带有索引的单个文件
  - 增强链接器通过查找一个或多个存档文件中的符号来解析尚未解析的外部引用
  - 如果存档的一个成员文件解析了符号引用，就把它链接到可执行文件中

创建静态库



- 存档文件可以增量更新
- 重新编译变化的函数，在存档文件中替换.o文件

常用库

libc.a (C 标准库)

- **4.6 MB 存档文件：1496 目标文件。**
- **I/O, 存储器分配,信号处理，字符串处理，日期和时间，随机数，整数数学运算。**

libm.a (C 数学库)

- **2 MB 存档文件：444 object 目标文件。**
- **浮点数学运算(sin, cos, tan, log, exp, sqrt, …)**

```
/usr/lib32> ar -t libc.a | sort
…
fork.o
…
fprintf.o
fpu_control.o
fputc.o
freopen.o
fscanf.o
fseek.o
fstab.o …
```

```
/usr/lib32> 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 …
```

与静态库链接

```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

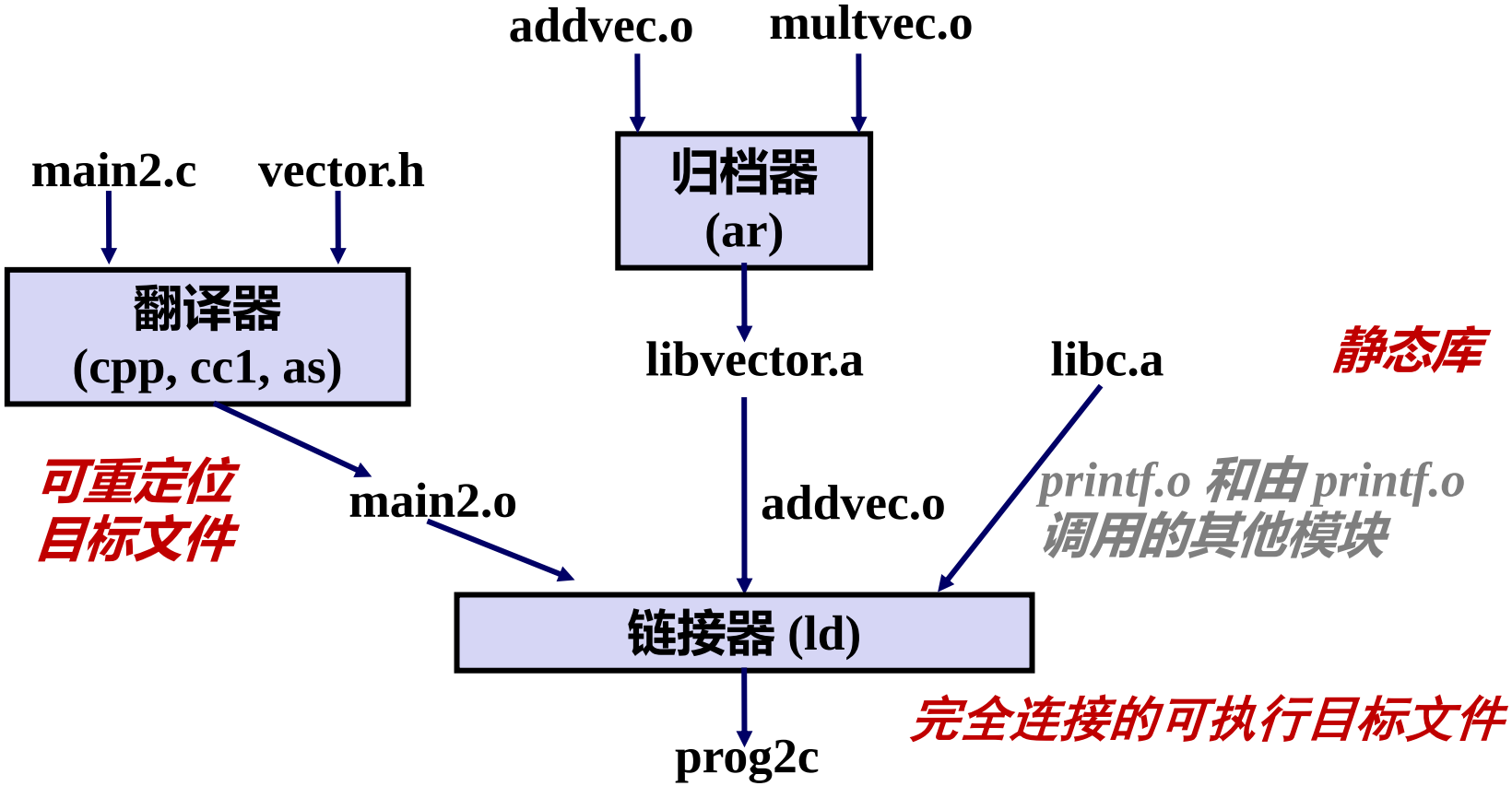
libvector.a

```c
#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

linux>gcc -c addvec.c multvec.c
linux>ar rcs libvector.a addvec.o multvec.o

与静态库链接



编译时用-c选项生成main2.o：gcc –c main2.c
链接方法1：gcc -static -o prog2c main2.o -L. -lvector
链接方法2：gcc -static -o prog2c main2.o ./libvector.a

直接编译链接方法1：gcc -static -o prog2c main2.c ./libvector.a
直接编译链接方法2：gcc -static -o prog2c main2.c -L. -lvector

使用静态库

搞不定依赖顺序：
-Xlinker --start-group -la -lb **-Xlinker --end-group**

- 链接器解析外部引用的算法:
  - 按照命令行的顺序扫描.o与 .a文件
  - 在扫描期间，保持一个当前未解析的引用列表U
  - 对于每个新的**.o或 .a文件**（*obj文件*）,利用该目标文件中定义的符号，尝试解析列表U中尚未解析的符号引用。
  - 若扫描结束时，在未解析符号列表U中仍存在条目，那么就报错！
- 问题:
  - 命令行中的顺序很重要!
  - 准则:将库放在命令行的末尾

```
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'
```

现代的解决方案:共享库

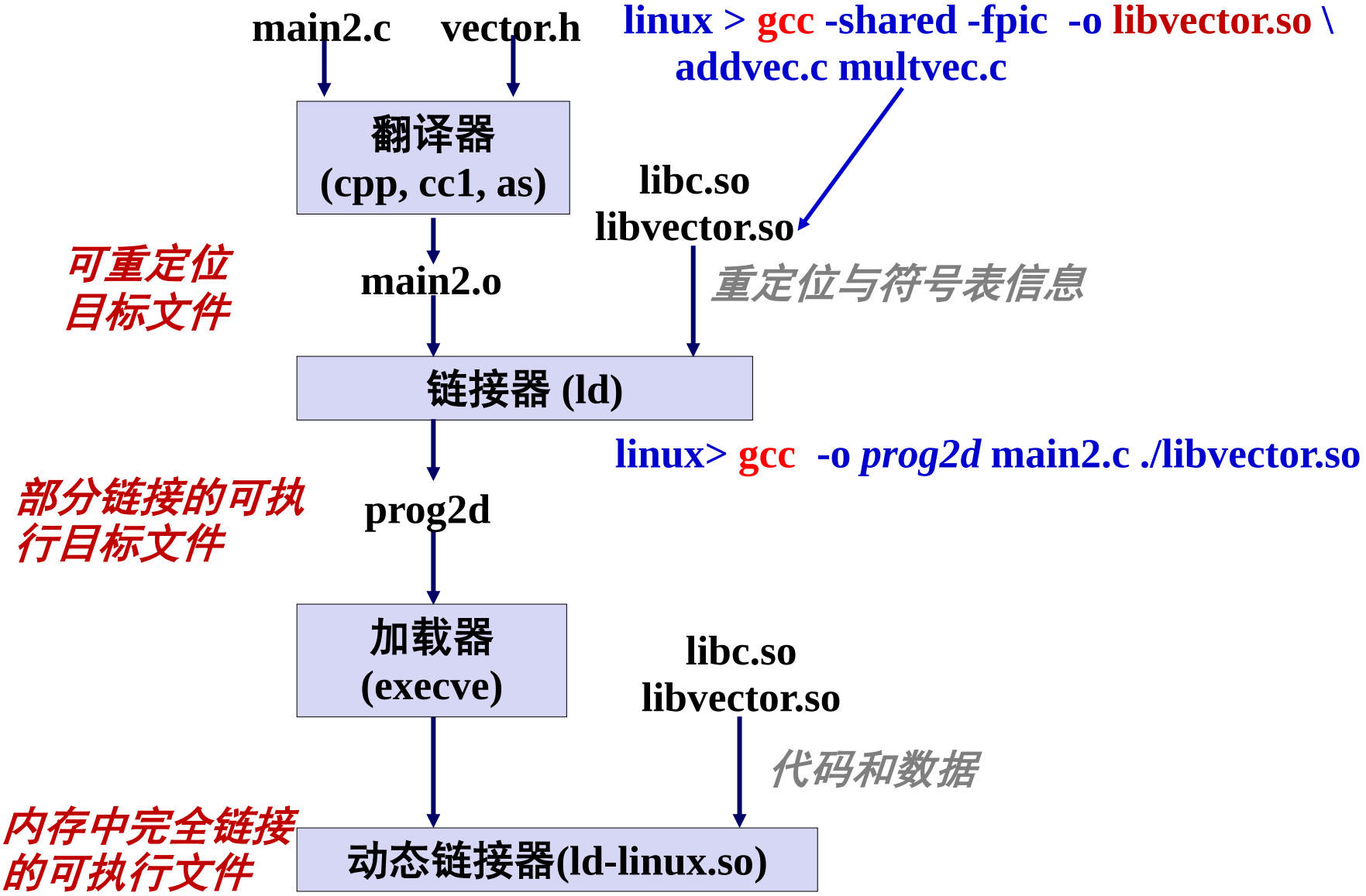
- 静态库缺点
  - 在存储的可执行文件中存在重复 (例如每个程序都需libc)
  - 在运行的可执行文件中存在重复
  - 系统库的小错误修复要求每个应用程序显式地重新链接

- 现代的解决方案——共享库
  - 包含代码和数据的目标文件，在(程序)加载时或运行时，共享库被动态地加载并链接到应用程序中
  - 也称: 动态链接库(DLL)、
  - .so文件(linux)
  - .dll文件（windows）

共享库 (cont.)

- **加载时链接：当可执行文件首次加载和运行时进行动态链接**
  - Linux通常由动态链接器(`ld-linux.so`) 自动处理
  - 标准C 库 (`libc.so`)通常是动态链接的
- **运行时链接：在程序开始运行后(通过编程指令)进行动态链接**
  - 在Linux中，通过调用dlopen()接口完成的
    - 分发软件
    - 高性能web服务器
    - 运行时库打桩
- **共享库载入内存后，可以由多个进程共享**
  - 第九章的虚拟内存会介绍共享内存的知识

加载时的动态链接


main2.c
vector.h
linux > gcc -shared -fpic -o libvector.so \
addvec.c multvec.c
翻译器
(cpp, cc1, as)
libc.so
libvector.so
可重定位
目标文件
main2.o
重定位与符号表信息
链接器 (ld)
linux> gcc -o prog2d main2.c ./libvector.so
部分链接的可执
行目标文件
prog2d
加载器
(execve)
libc.so
libvector.so
代码和数据
内存中完全链接
的可执行文件
动态链接器(ld-linux.so)

运行时动态链接

dll.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;

    /*动态加载包含addvec()的共享库*/
    handle = dlopen("./libvector.so", RTLD_LAZY);
    if (!handle) {
        fprintf(stderr, "%s\n", dlerror());
        exit(1);
    }
```

运行时动态链接

dll.c

```
...
/* 获取我们刚刚加载的addvec()函数的指针 */
addvec = dlsym(handle, "addvec");
if ((error = dlerror()) != NULL) {
    fprintf(stderr, "%s\n", error);
    exit(1);
}

/*现在就可以像其他函数一样调用addvec() */
addvec(x, y, z, 2);
printf("z = [%d %d]\n", z[0], z[1]);

/*卸载共享库*/
if (dlclose(handle) < 0) {
    fprintf(stderr, "%s\n", dlerror());
    exit(1);
}
return 0;
}
```

编译：

linux> gcc –rdynamic –o prog2r dll.c -ldl

-rdynamic :通知链接器将所有符号添加到动态符号表中，这些符号将在dlsym、向后追踪(backtrace)时使用

-ldl:运行时显式加载动态库的动态函数库

dlopen()

- void *dlopen(const char *filename, int flag)
- 功　能：打开指定共享库，并返回文件描述符。
- 返回值： 成功时返回文件描述符，否则返回NULL。
- 说　明：

（1）filename: 库文件路径

如不以/开头，则为非绝对路径名，按以下顺序搜索库文件：

① 环境变量中的LD_LIBRARY_PATH值指定的那些路径；

② 动态链接缓冲文件/etc/ld.so.cache;

③ 库文件默认目录/lib、/usr/lib

dlopen说明

（2）flag：如何解析符号

① 解析方式

- RTLD_LAZY：暂缓决定，在dlopen返回前，对于动态库中的未定义符号不执行解析（只对函数引用有效，对于变量引用总是立即解析）。等有需要时，即在调用动态链接库中函数的时候再解析符号。
- RTLD_NOW：立即决定，在dlopen返回前，解析出所有未定义符号，如果解析不出来，dlopen会返回NULL，错误为： "undefined symbol: xxxx..... "。

② 作用范围，可与解析方式通过"|"组合使用

- RTLD_GLOBAL：动态库中定义的符号可被其后打开的其它库重定位。
- RTLD_LOCAL： 与RTLD_GLOBAL作用相反，动态库中定义的符号不能被其后打开的其它库重定位。如果没有指明是RTLD_GLOBAL还是RTLD_LOCAL，则缺省为RTLD_LOCAL。

dlsym()

- void *dlsym(void *handle, char *symbol);
  - 功能：返回共享库中指定函数的入口地址
  - 说明：根据共享库文件描述符handle与符号symbol，返回symbol对应的函数入口地址，相当于返回一个函数指针
  - 默认是C语言函数名，C++程序中需要使用**extern "C"**

```
extern "C" {   int func1(void  *param){return 0;};
    int func2(char *buf, int len){return 0;}
    int func3(){return 0;}
}
```

- void dlclose(void *handle);
  - 功能：关闭已经打开的指定共享库文件。
- const char *dlerror(void);
  - 功能：返回动态共享库操作状态信息（失败信息）。

链接汇总

- 链接是一个技术： 允许从多个目标文件创建程序
- 链接可以在程序生命周期的不同时间发生:
  - 编译时(当程序被编译链接时，GCC编译时)
  - 加载时(将程序加载到内存中)
  - 运行时(当程序正在执行时)
- 理解链接可以帮助我们避免讨厌的错误，做一个更优秀的程序员。

要点

- 链接
- 案例学习: 库打桩机制

案例学习: 库打桩机制

- 库打桩机制： 强大的链接技术--- 允许程序员拦截对任意函数的调用
- 打桩可出现在:
  - 编译时：源代码被编译时
  - 链接时间：当可重定位目标文件被静态链接来形成一个可执行目标文件时
  - 加载/运行时：当一个可执行目标文件被加载到内存中，动态链接，然后执行时

一些打桩应用程序

- 安全
  - 监禁**confinement** (沙箱sandboxing)
  - 幕后加密
- 调试
  - 2014年，两名Facebook工程师使用了打桩机制，调试了他们的iPhone APP中一个1年之久的危险bug
  - SPDY网络堆栈中的代码正在写入错误的位置
  - 通过拦截Posix的write函数(write, writev, pwrite)来解决问题

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

一些打桩应用程序

- 监控和性能分析
  - 计算函数调用的次数
  - 刻画函数的调用位置(sites)和参数
  - Malloc 跟踪
    - 检测内存泄露
    - **生成地址痕迹(traces)**

程序实例

- 目标：跟踪已分配/释放的内存块的地址和大小，不破坏程序，也不修改源代码
- 三个解决方案：在编译时、链接时和加载/运行时，对库函数malloc和free进行打桩

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

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

int.c

编译时打桩

```
#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

```
#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)=0x9ee010
free(0x9ee010)
linux>
```

链接时打桩

```
#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

```
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) = 0x18cf010
free(0x18cf010)
linux>
```

- “-Wl” 标志将参数传递给链接器，将每个逗号替换为空格
- “--wrap,malloc ” 参数 指示链接器以一种特殊的方式解析引用:
  - 将malloc 的引用被解析为 __wrap_malloc
  - 将__real_malloc的引用解析为 malloc

加载/运行时打桩

```
#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/运行时打桩

```
/* 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/运行时打桩

```
gcc -Wall -DRUNTIME -shared -fpic -o mymalloc.so mymalloc.c -ldl
gcc -Wall -o intr int.c
```

```
linux> make intr
linux> make runr
malloc(32) = 0xe60010
free(0xe60010)
linux> LD_PRELOAD="./mymalloc.so"  /usr/bin/uptime
....
```

(LD_PRELOAD="./mymalloc.so" ./intr)

- LD_PRELOAD环境变量告诉动态链接器，首先查看mymalloc.so，解析尚未解析的符号引用(例如malloc)。

打桩回顾

- 编译时
  - 采用宏，将malloc/free的显式调用转换成对mymalloc/myfree的调用
- 链接时
  - 使用链接技巧，来获得特殊的符号名解析
    - malloc → __wrap_malloc
    - __real_malloc → malloc
- 加载/运行时
  - 实现malloc/free的自定义版本：使用动态链接，用不同的名字来加载库函数malloc/free

CALL指令格式(对被调用函数的寻址方式)

CALL—Call Procedure

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
E8 *cw*	CALL *rel16*	N.S.	Valid	Call near, relative, displacement relative to next instruction. Not supported in 64-bit mode.
E8 *cd*	CALL *rel32*	Valid	Valid	Call near, relative, displacement relative to next instruction. 32-bit displacement sign extended to 64-bits in 64-bit mode.
FF /2	CALL *r/m16*	N.E.	Valid	Call near, absolute indirect, address given in *r/m16.*
FF /2	CALL *r/m32*	N.E.	Valid	Call near, absolute indirect, address given in *r/m32.* 32-bit displacement sign extended to 64-bits in 64-bit mode
FF /2	CALL *r/m64*	Valid	N.E.	Call near, absolute indirect, address given in *r/m64.*
9A *cd*	CALL *ptr16:16*	Inv.	Valid	Call far, absolute, address given in operand
9A *cp*	CALL *ptr16:32*	Inv.	Valid	Call far, absolute, address given in operand
FF /3	CALL *m16:16*	Valid	Valid	Call far, absolute indirect, address given in *m16:16* In 32-bit mode if selector points to a gate then RIP = 32-bit zero extended displacement taken from gate else RIP = zero extended 16-bit offset from far pointer referenced in the instruction.
FF /3	CALL *m16:32*	Valid	Valid	In 64-bit mode of operation If selector points to a gate then RIP = 64-bit displacement taken from gate else RIP = zero extended 32-bit offset from far pointer referenced in the instruction.
FF /3	CALL *m16:64*	Valid	N.E.	In 64-bit mode of operation If selector points to a gate then RIP = 64-bit displacement taken from gate else RIP = 64-bit offset from far pointer from far pointer referenced in the instruction.

JMP指令格式(对跳转目的地址的寻址方式)

JMP—Jump

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
EB *cb*	JMP *rel8*	Valid	Valid	Jump short, RIP = RIP + 8-bit displacement sign extended to 64-bits
E9 *cw*	JMP *rel16*	N.S.	Valid	Jump near, relative, displacement relative to next instruction. Not supported in 64-bit mode.
E9 *cd*	JMP *rel32*	Valid	Valid	Jump near, relative, RIP = RIP + 32-bit displacement sign extended to 64-bits
FF /4	JMP *r/m16*	N.S.	Valid	Jump near, absolute indirect, address = sign-extended *r/m16.* Not supported in 64-bit mode.
FF /4	JMP *r/m32*	N.S.	Valid	Jump near, absolute indirect, address = sign-extended *r/m32.* Not supported in 64-bit mode.
FF /4	JMP *r/m64*	Valid	N.E.	Jump near, absolute indirect, RIP = 64-Bit offset from register or memory
EA *cd*	JMP *ptr16:16*	Inv.	Valid	Jump far, absolute, address given in operand
EA *cp*	JMP *ptr16:32*	Inv.	Valid	Jump far, absolute, address given in operand
FF /5	JMP *m16:16*	Valid	Valid	Jump far, absolute indirect, address given in *m16:16*
FF /5	JMP *m16:32*	Valid	Valid	Jump far, absolute indirect, address given in *m16:32* In 32-bit mode of operation If selector points to a gate then RIP = 32-bit zero extended displacement taken from gate else RIP = zero extended 32-bit offset from far pointer referenced in the instruction.
FF /5	JMP *m16:64*	Valid	N.E.	In 64-bit mode of operation If selector points to a gate then RIP = 64-bit displacement taken from gate else RIP = zero extended 32-bit offset from far pointer referenced in the instruction.