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

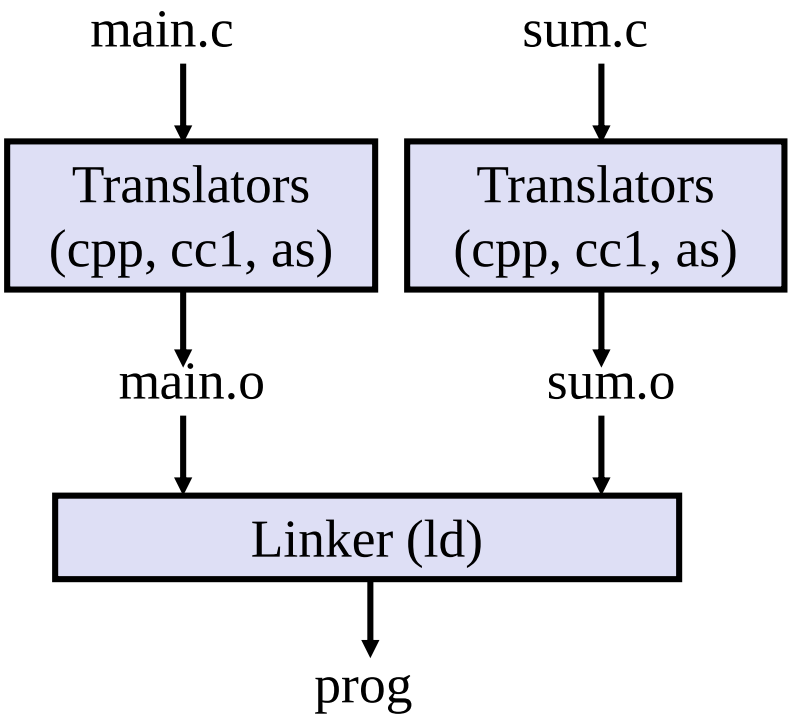
C程序例子

```
//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)

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() {…} /* define symbol swap */**
  - **swap(); /* reference symbol swap */**
  - **int *xp = &x; /* define symbol xp, reference 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
- 在符号解析步骤中，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)

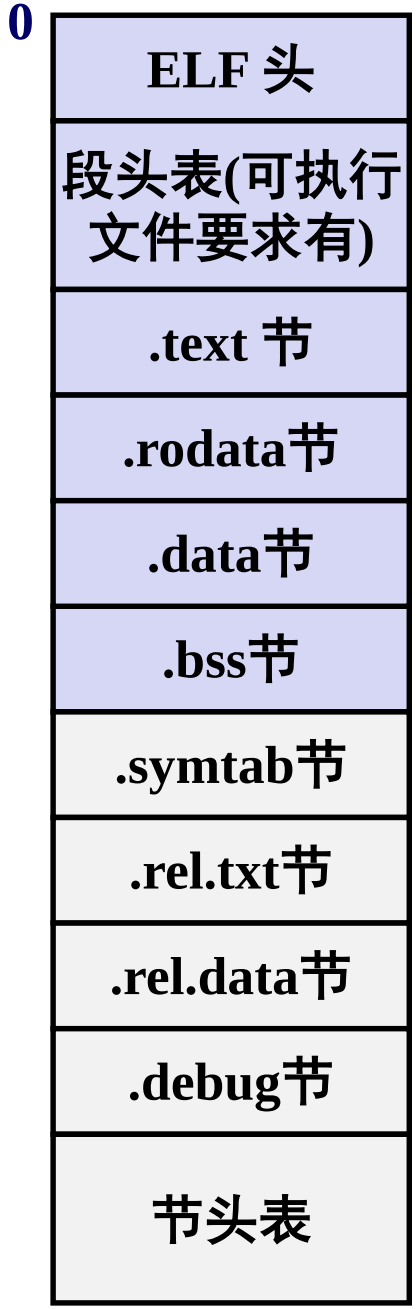
- Executable and Linkable Format， ELF
  - X86-64 Linux 、Unix系统
  - 目标文件的标准二进制格式

- 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

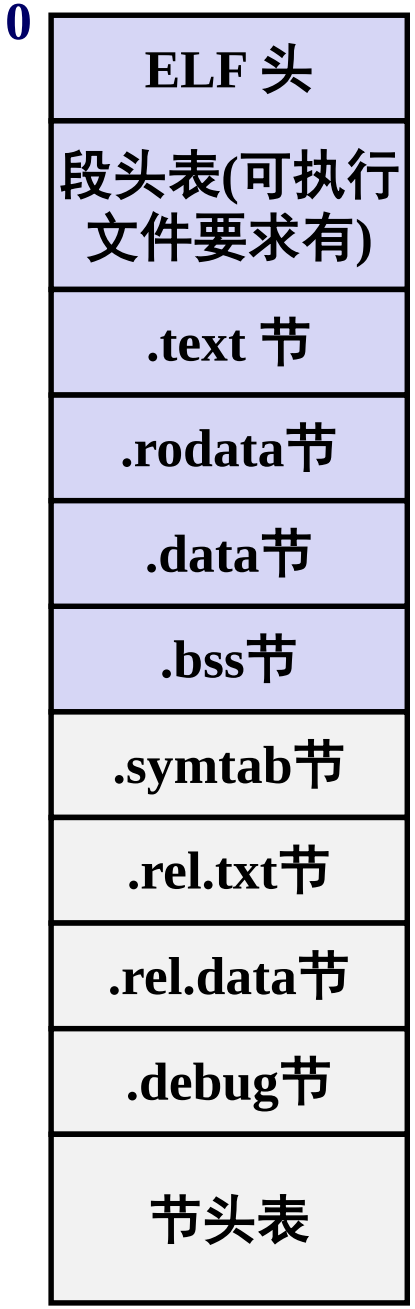
ELF目标文件格式

- **ELF 头**
  - 字大小、字节序：16字节
  - 文件类型(.o，exec，.so)、机器类型、节头表位置等
- 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.)

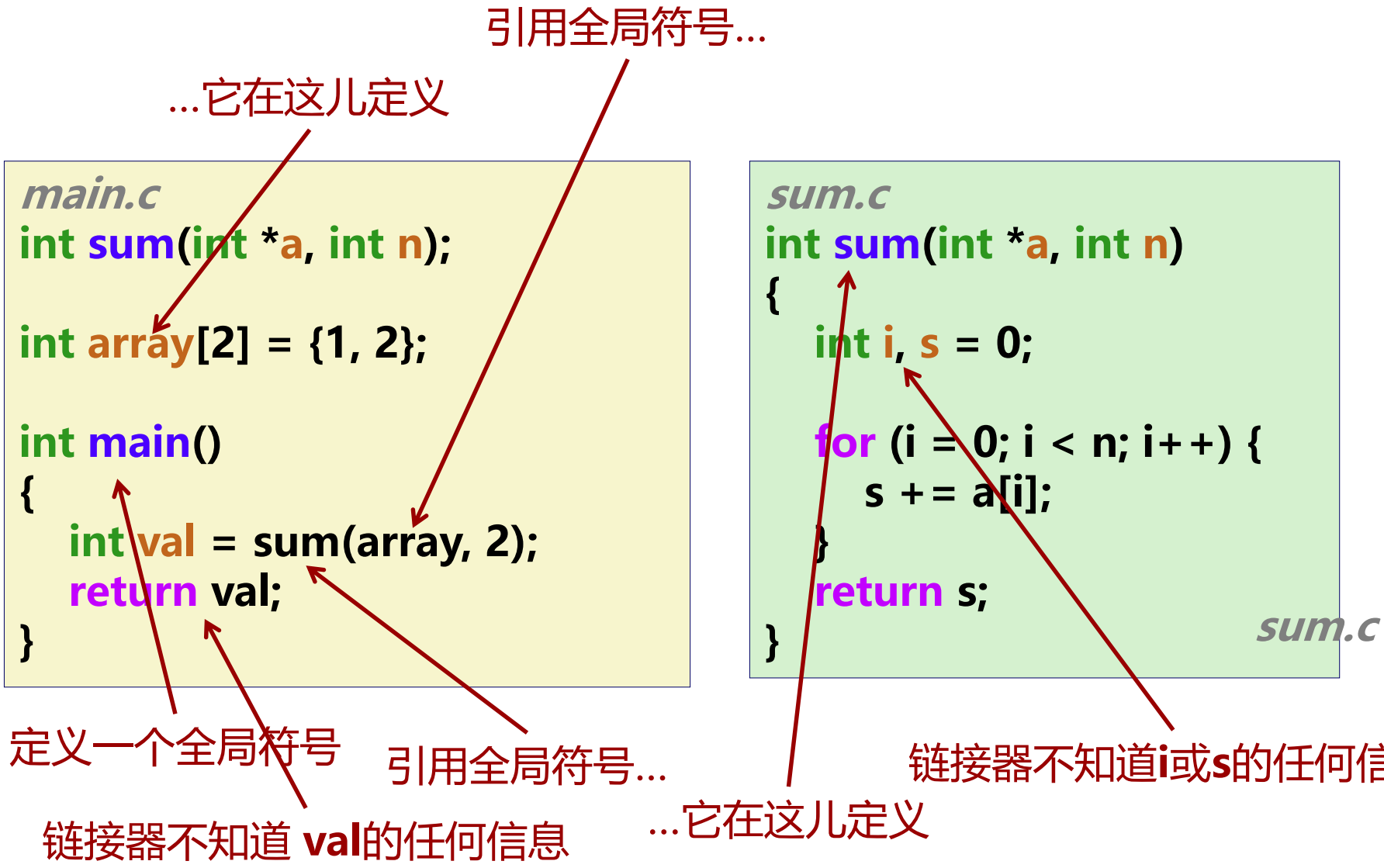
- **.symtab** 节（符号表）
  - 函数和全局/静态变量名
  - 节名称和位置
- **.rel.text** 节（可重定位代码）
  - .text 节的可重定位信息
  - 在可执行文件中需要修改的指令地址
  - 需修改的指令
- **.rel.data** 节（可重定位数据）
  - **.data** 节的可重定位信息
  - 在合并后的可执行文件中需要修改的指针数据的地址
- **.debug** 节（调试符号表）
  - 为符号调试的信息 (**gcc -g**)
- 节头表**Section header table**
  - 每个节的偏移量、大小



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

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

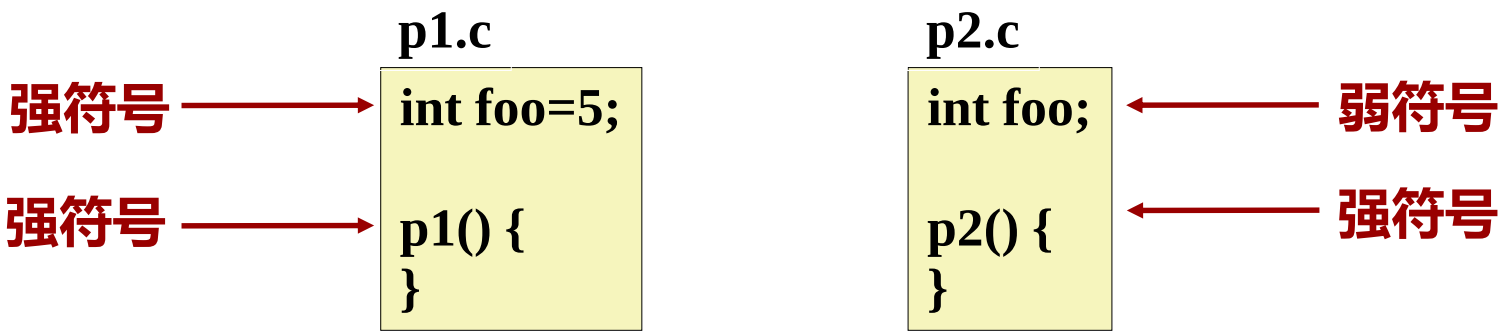
```
int f()
{
    static int x = 0;
    return x;
}

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. 这是你真正想要的？
int x; int y; p1() {}	double x; p2() {}	在p2中如对x写入，可能覆盖y！ 作恶!

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

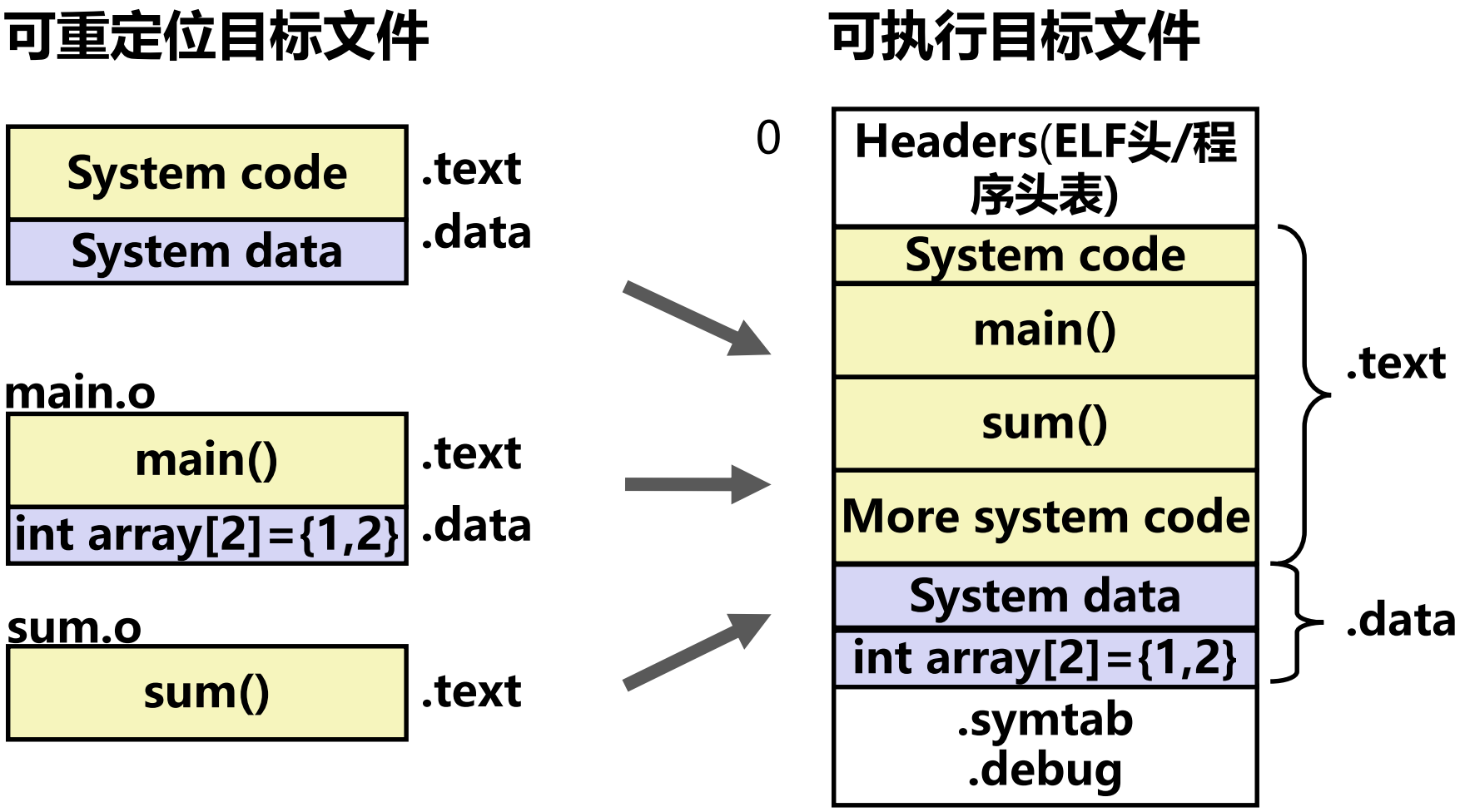
int x=7; p1() {}	int x; p2() {}	对x的引用将指向相同的初始化变量

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
.text
System data
.data
main.o
main()
.text
int array[2]={1,2}
.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 -r -d 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**

```
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

或：

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)

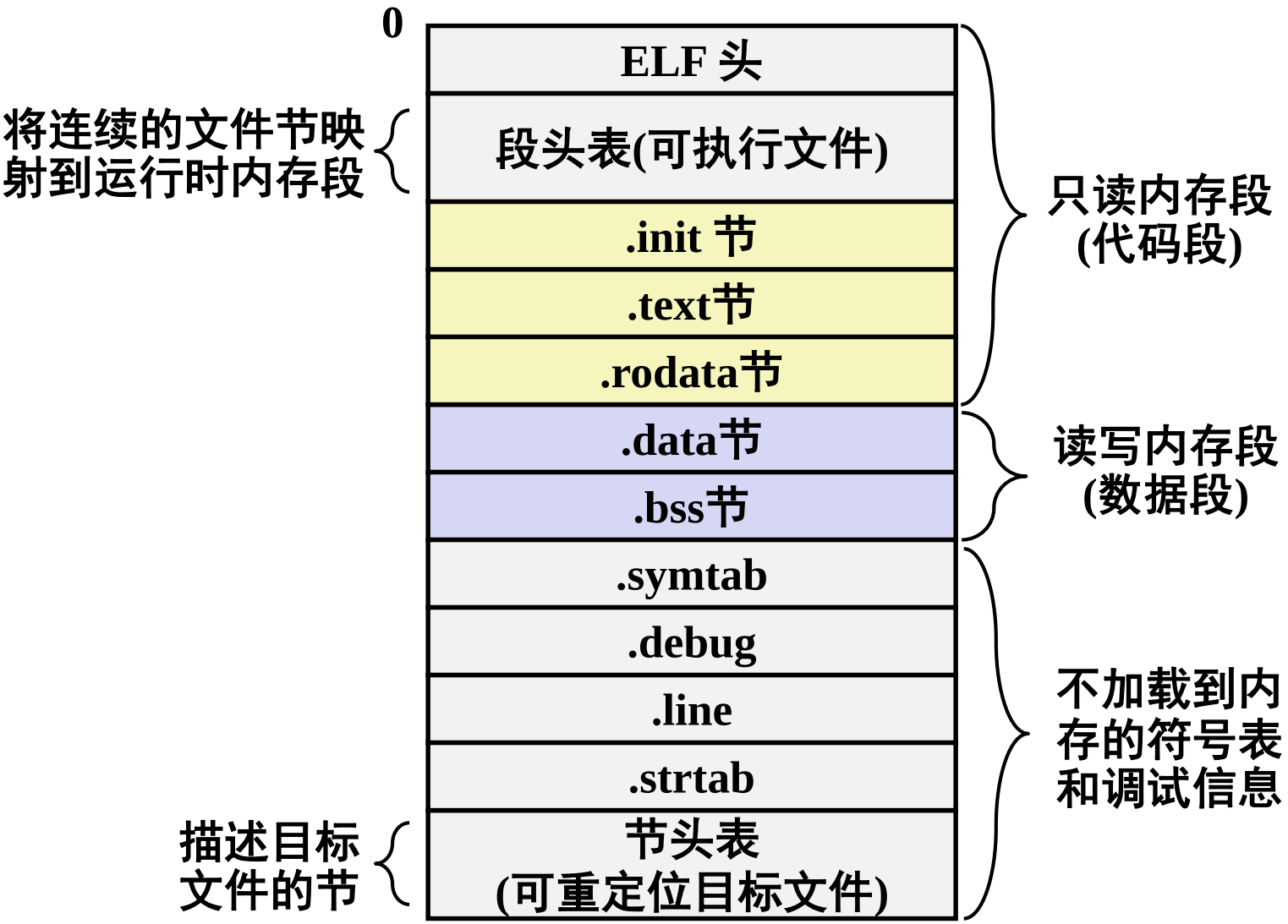
已经重定位的 .text 节

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
```

使用PC相对寻址 sum(): 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)**

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 221**
filesz 0x000000000000069c memsz 0x000000000000069c flags r-x

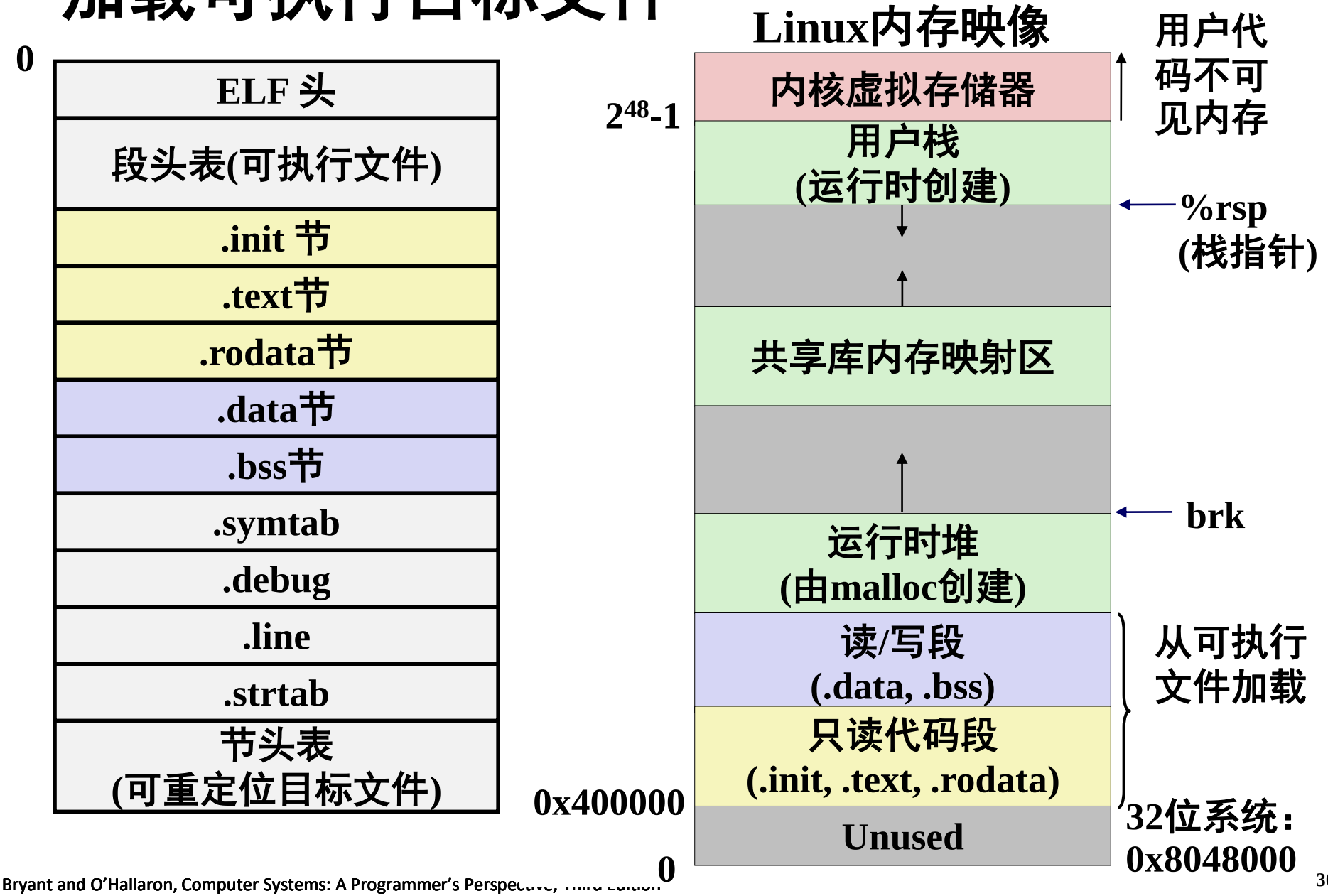
LOAD off 0x0000000000000df8 vaddr 0x0000000000600df8 paddr 0x0000000000600df8 align 221**
filesz 0x0000000000000228 memsz 0x0000000000000230 flags rw-

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

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
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
0
32位系统:
0x8048000
用户代
码不可
见内存

常用函数打包

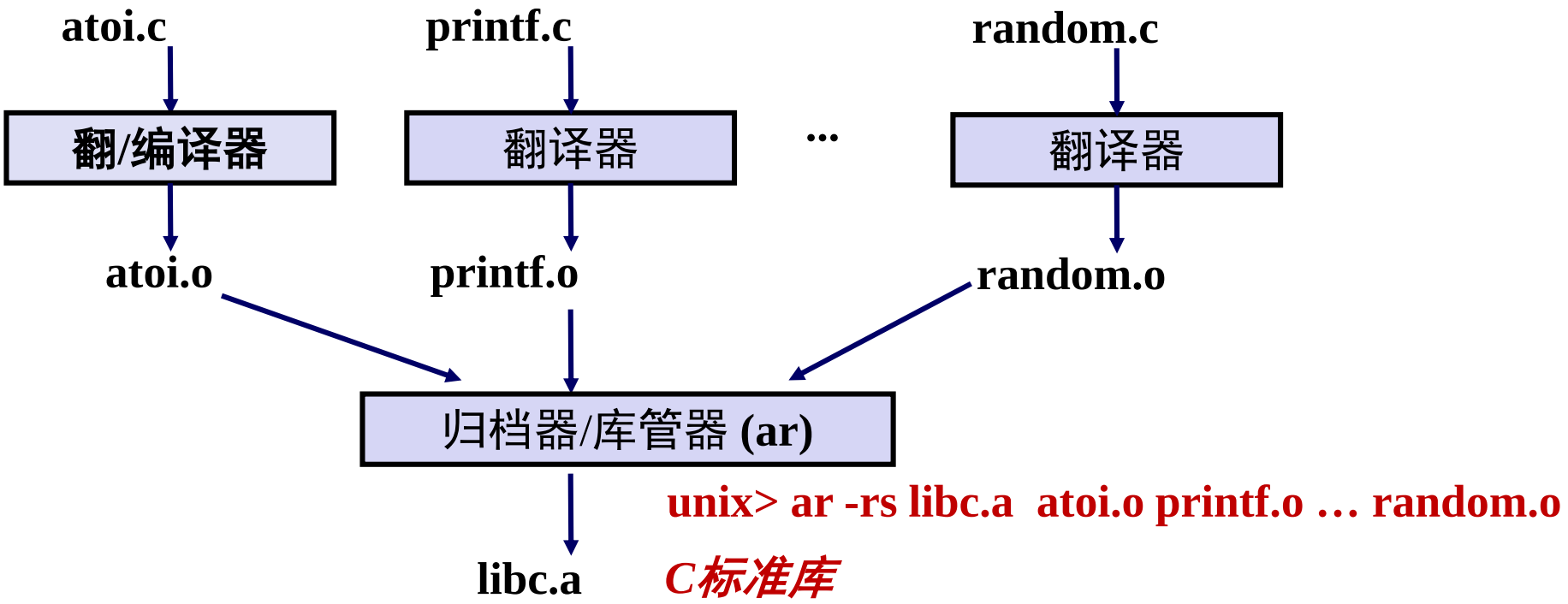
- 如何打包程序员常用的函数?
  - Math、I/O、存储管理、串处理,等等。

- 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中(叫做存档文件)
  - 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

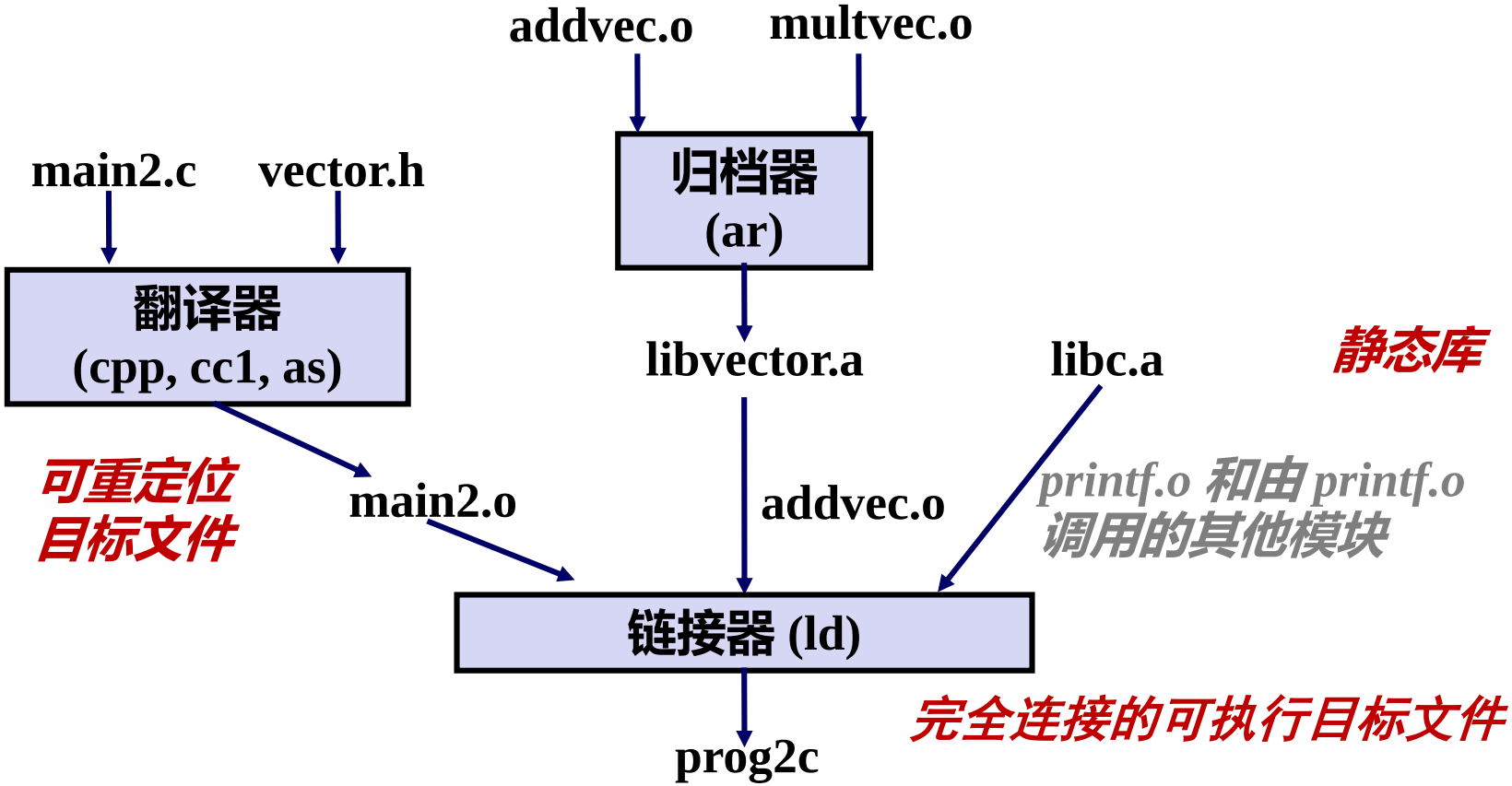
libvector.a

```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 res 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

```
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'
```

使用静态库

- 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。
  - 如在扫描结束时，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

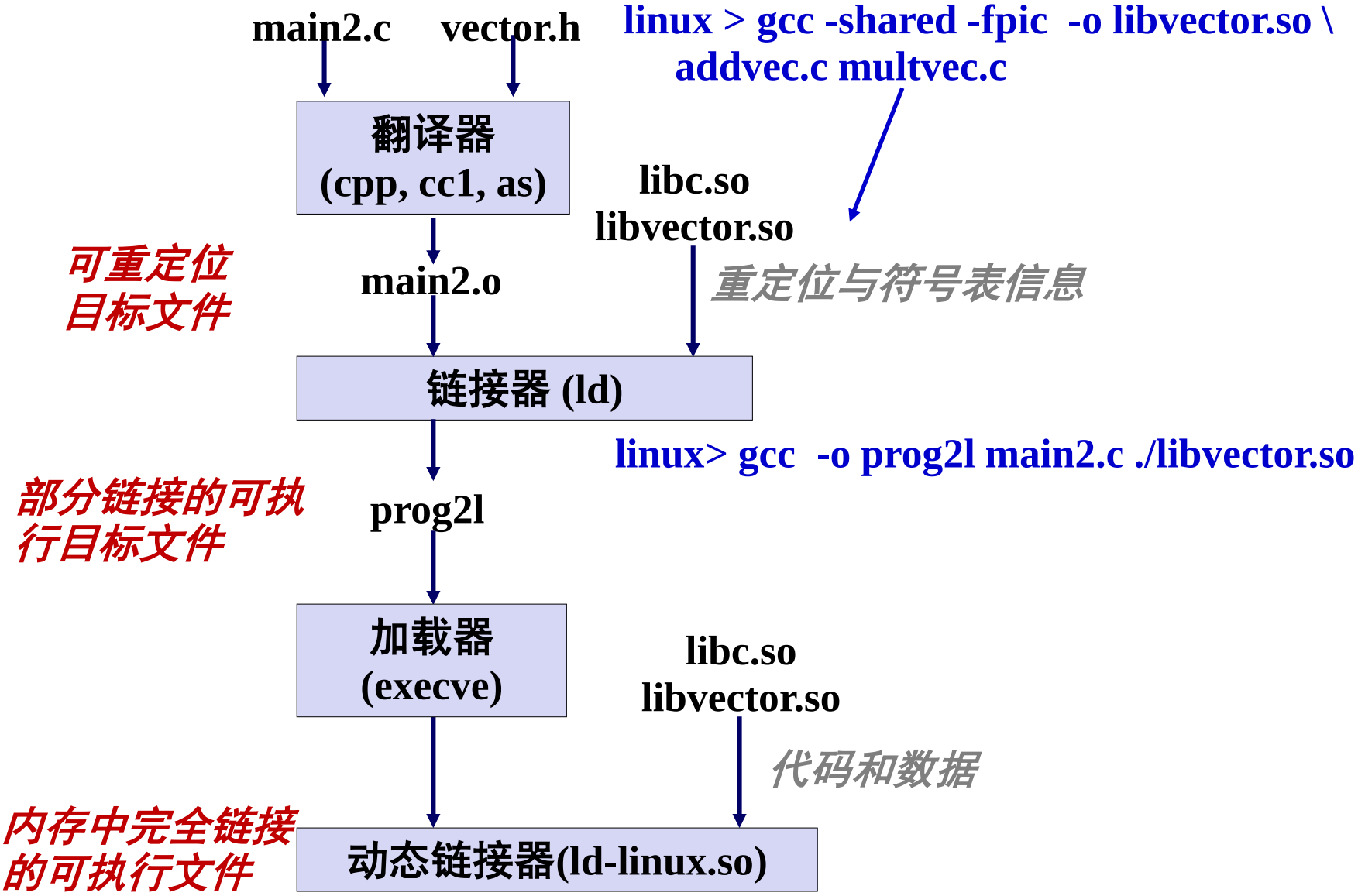
- 静态库缺点
  - 在存储的可执行文件中存在重复 (例如每个程序都需libc)
  - 在运行的可执行文件中存在重复
  - 系统库的小错误修复要求每个应用程序显式地重新链接

- 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的常见情况是，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
- **共享库的例程可以由多个进程共享**
  - 当我们学习虚拟内存时有更多这方面的内容

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 prog2l main2.c ./libvector.so
部分链接的可执
行目标文件
prog2l
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，以便使用 dlopen 实现向后跟踪

-ldl:运行时/显式加载动态库的动态函数库

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

```
#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

```
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="./mymalloc.so"   /usr/bin/uptime
```

- 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