第6章 链接

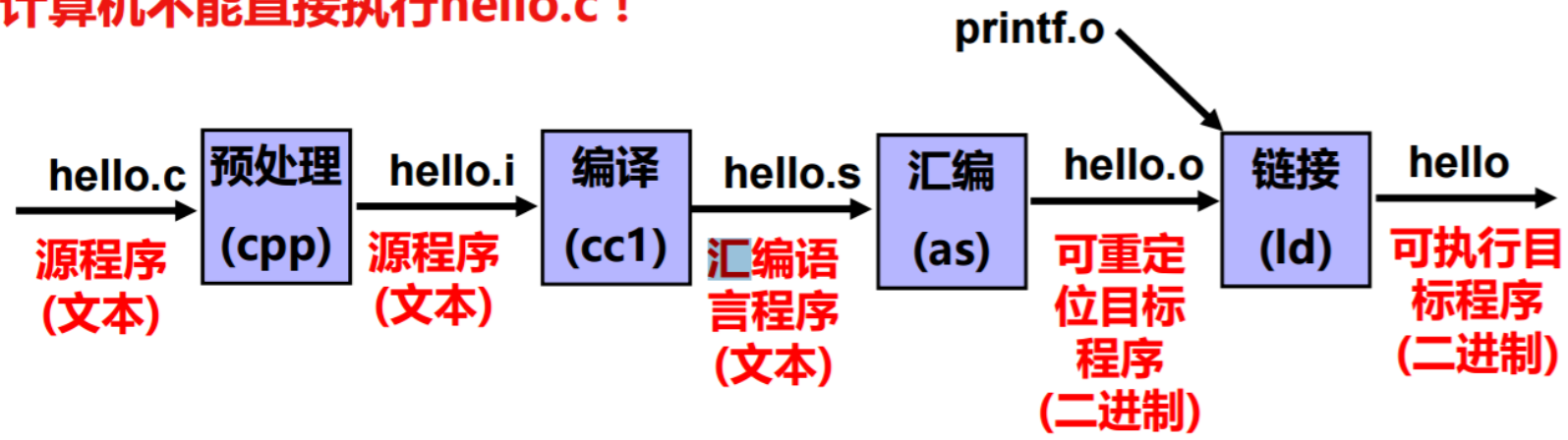
教 师：吴锐
计算机科学与技术学院
哈尔滨工业大学

要点

- **链接**
- **案例学习: 库打桩机制**

一个典型程序的转换过程

计算机不能直接执行hello.c！



C程序例子

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

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

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

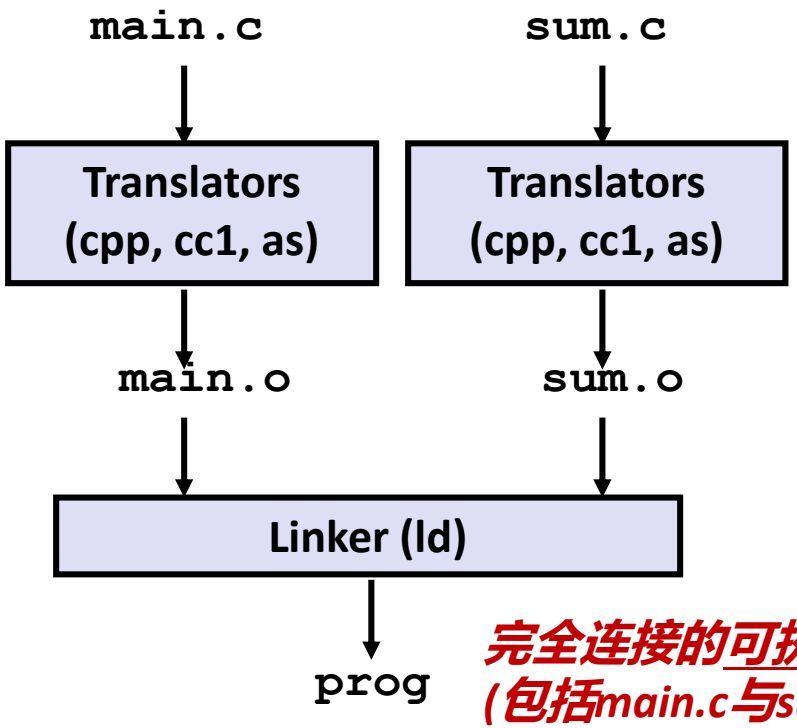
main.c

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

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

sum.c

静态链接

- 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

完全连接的可执行的目标文件
(包括main.c与sum.c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...
    - 然而，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.

链接器干啥?

- **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
                    */
```

  - 符号定义存储在目标文件中(由汇编器) 的符号表中.
    - 符号表示一个结构型的数组
    - 每个条目包括名称、大小和符号的位置.
  - **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.**

链接器干啥?(cont)

- **步骤 2: 重定位**
  - 将多个单独的代码节和数据节合并为单个节
  - 将符号从它们的在.o文件的相对位置重新定位到可执行文件的最终绝对内存位置。
  - 更新所有对这些符号的引用来反映它们的新位置.

让我们更详细地了解这两个步骤....

三种目标文件(模块)

- **可重定位目标文件(.o 文件)**
  - 包含与其他可重定位目标文件相结合的代码和数据，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)**
  - 包含可以直接复制到内存并执行的代码和数据.
- **共享目标文件(.so 文件)**
  - 特殊类型的可重定位目标文件，在加载时或运行时，它可以被动态加载到内存并链接。
  - 在Windows中称为动态链接库.DLL

可执行可链接格式(ELF)

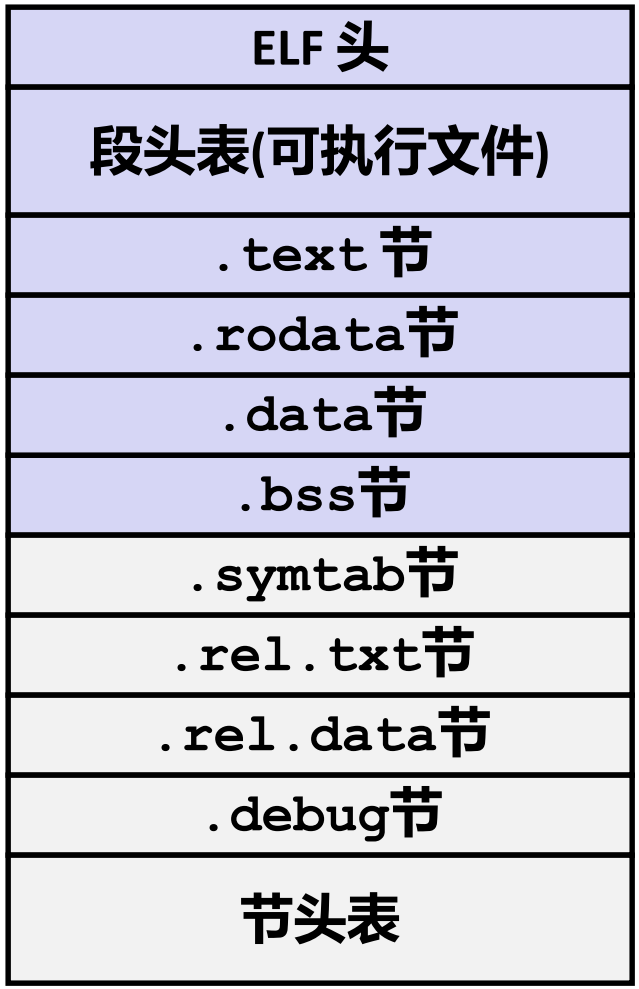
- 目标文件的标准二进制格式

- 一个统一的格式：
  - 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

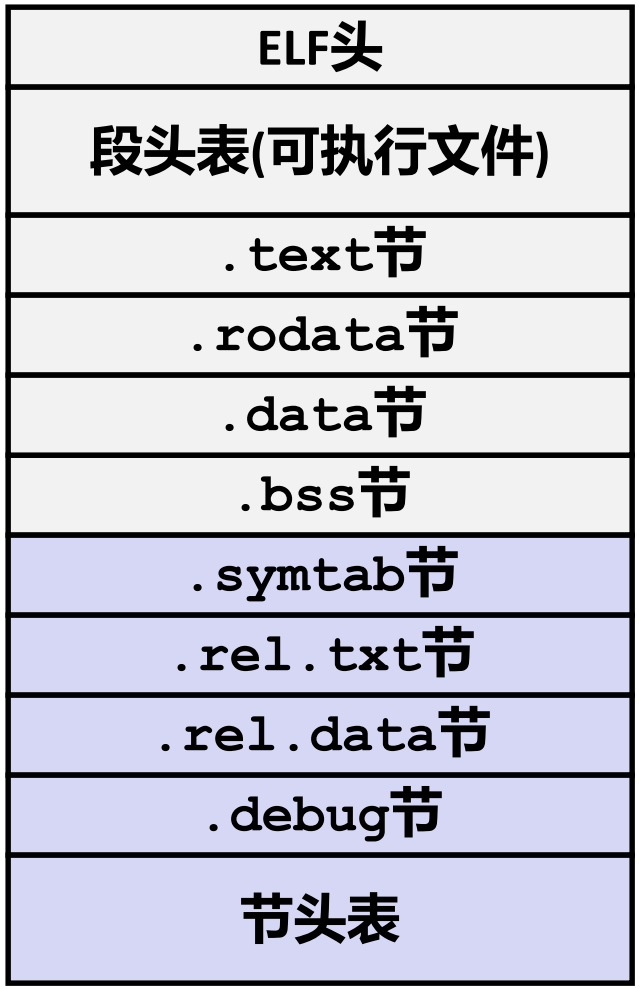
ELF 目标文件格式

- **Elf 头**
  - 字大小、字节顺序、文件类型(.o，exec，.so)，机器类型，等等
- **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：跳转表, ...
- **.data 节（数据/可读写）**
  - 已初始化全局变量
- **.bss 节（未初始化全局变量）**
  - 未初始化的全局变量
  - “Block Started by Symbol” 符号开始的块
  - “Better Save Space” 更加节省空间
  - 有节头，但不占用空间



ELF目标文件格式(cont.)

- .symtab 节（符号表）
  - 符号表
  - 函数和静态变量名
  - 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）
  - 为符号调试的信息 (gcc -g)
- 节头表Section header table
  - 每个节的偏移量和大小



ELF目标文件

可重定位目标文件格式

ELF 头

- ✓ 包括16字节标识信息、文件类型（.o, exec, .so）、机器类型（如 IA-32）、节头表的偏移、节头表的表项大小以及表项个数

.text 节

- ✓ 编译后的代码部分

.rodata 节

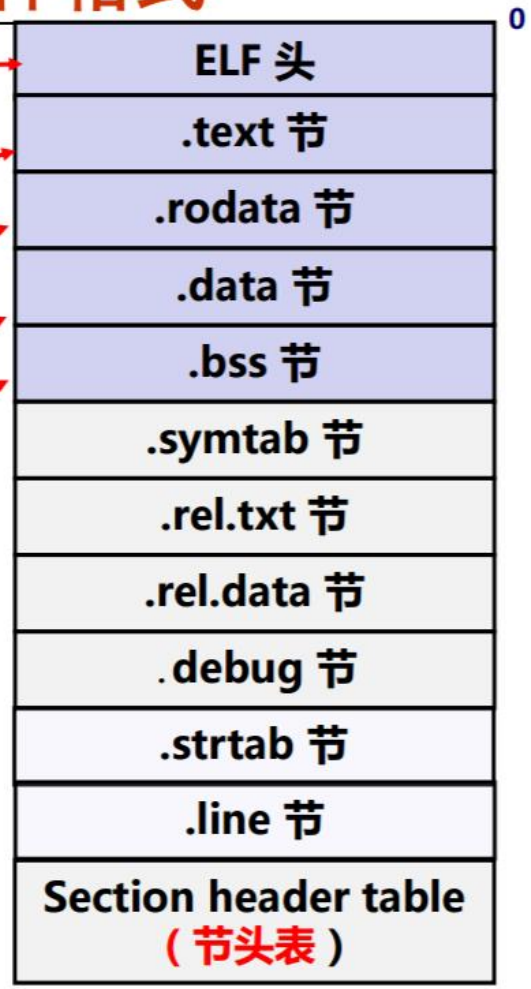
- ✓ 只读数据，如 printf 格式串、switch 跳转表等

.data 节

- ✓ 已初始化的全局变量

.bss 节

- ✓ 未初始化全局变量，仅是占位符，不占据任何实际磁盘空间。区分初始化和非初始化是为了空间效率



ELF目标文件

.symtab 节
- ✓ 存放函数和全局变量（符号表）信息，它不包括局部变量

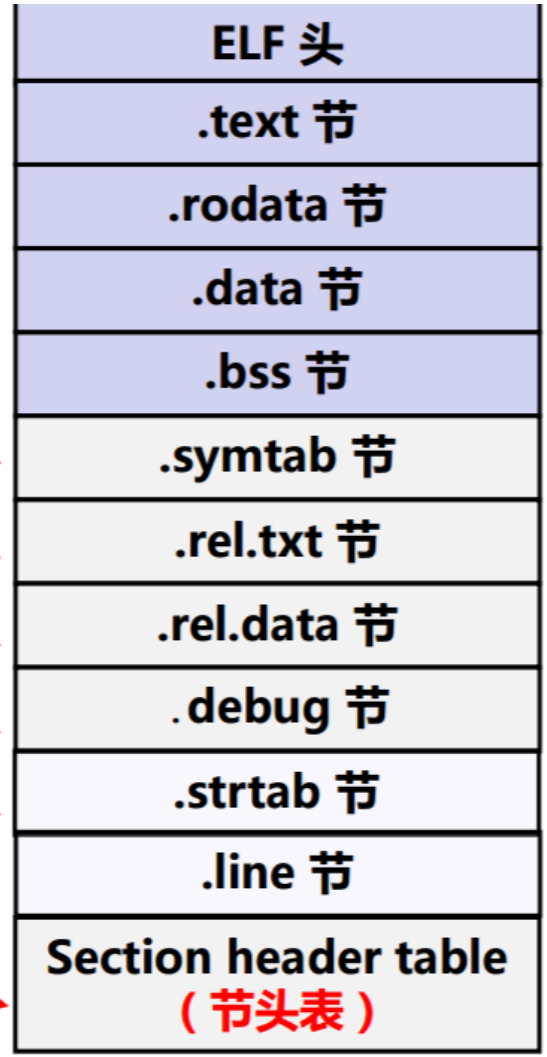
.rel.text 节
- ✓ .text节的重定位信息，用于重新修改代码段的指令中的地址信息

.rel.data 节
- ✓ .data节的重定位信息，用于对被模块使用或定义的全局变量进行重定位的信息

.debug 节
- ✓ 调试用符号表 (gcc -g)

strtab 节
- ✓ 包含symtab和debug节中符号及节名

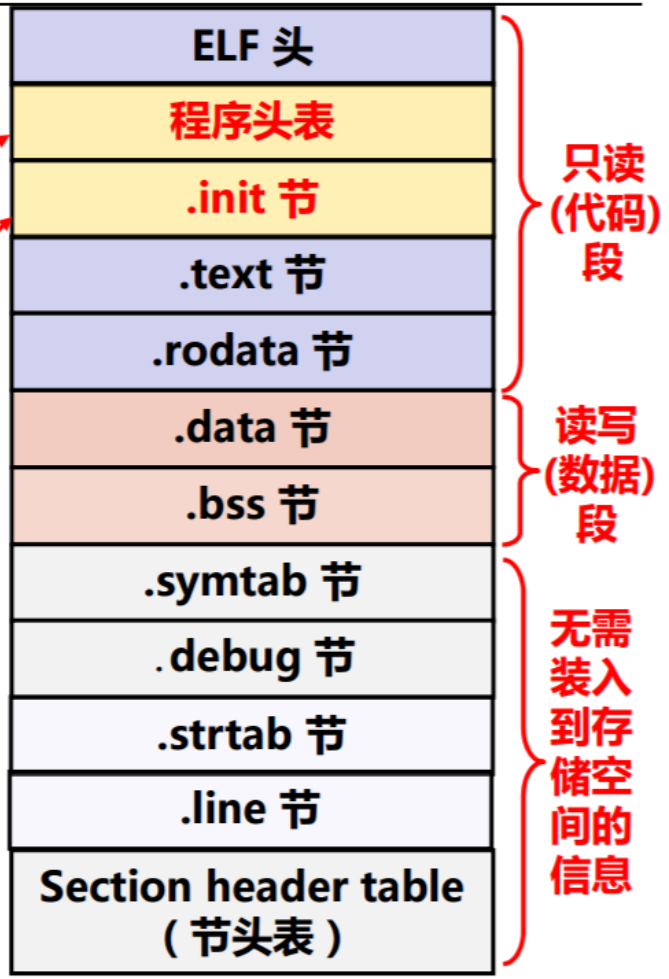
Section header table（节头表）
- ✓ 每个节的节名、偏移和大小



ELF目标文件

可执行目标文件格式

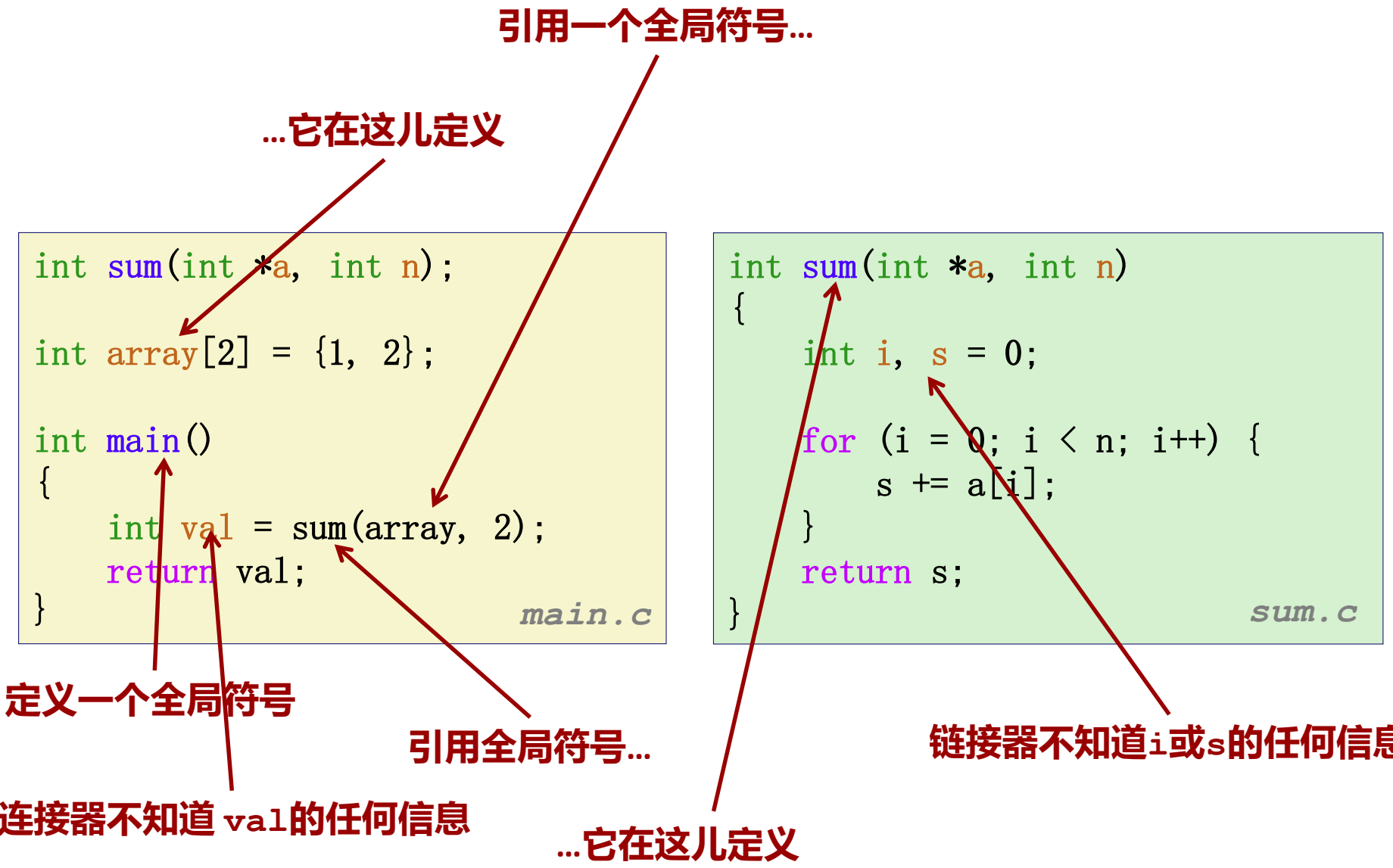
- 与可重定位文件稍有不同：
  - ELF头中字段e_entry给出执行程序时第一条指令的地址，而在可重定位文件中，此字段为0
  - 多一个程序头表，也称段头表（segment header table），是一个结构数组
  - 多一个.init节，用于定义_init函数，该函数用来进行可执行目标文件开始执行时的初始化工作
  - 少两个.rel节（无需重定位）



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 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定义和仅由m唯一引用的符号
  - 如:使用静态属性定义的C函数和全局变量
  - 本地链接符号不包括非静态本地程序变量

步骤 1: 符号解析


引用一个全局符号...
...它在这儿定义
int sum(int *a, int n);
int array[2] = {1, 2};
int main()
{
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
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
连接器不知道 val的任何信息
...它在这儿定义

本地符号

- **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
  - 本地的静态C变量: 存储在 `.bss` 或 `.data`

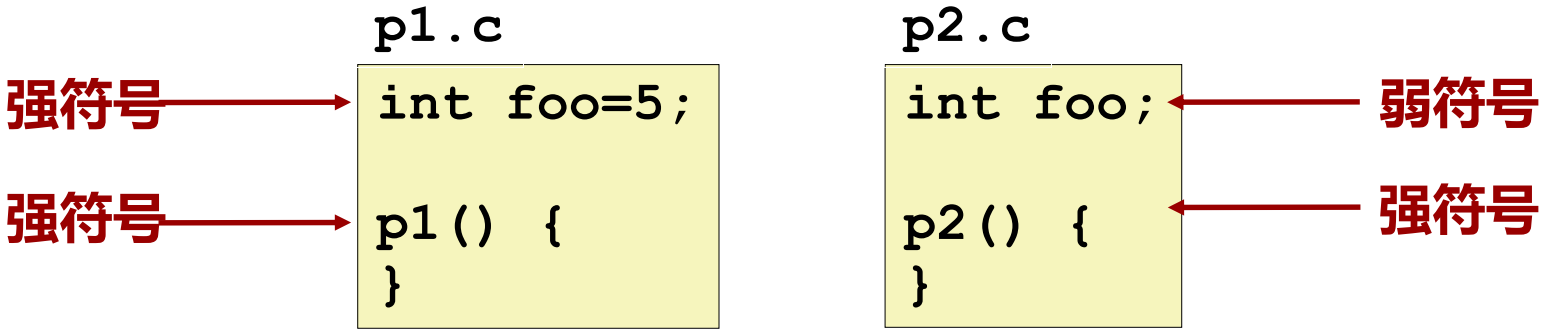
```
int f()
{
    static int x = 0;
    return x;
}

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惟一名称的本地符号。如 **`x.1`** 与 **`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**
  - 对弱符号的引用解析为强符号
- **规则 3:如果有多个弱符号，选择任意一个**
  - 可以用 `gcc -fno-common` 来覆盖这个规则

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 的引用将是相同的未初始化的 int. 这就是你真正想要的？

`int x;`
`int y;`
`p1() {}`
	`double x;`
	`p2() {}`
		在p2中写入x可能会覆盖y！ 邪恶!

`int x=7;`
`int y=5;`
`p1() {}`
	`double x;`
	`p2() {}`
		在p2中写入x将覆盖y！讨厌!

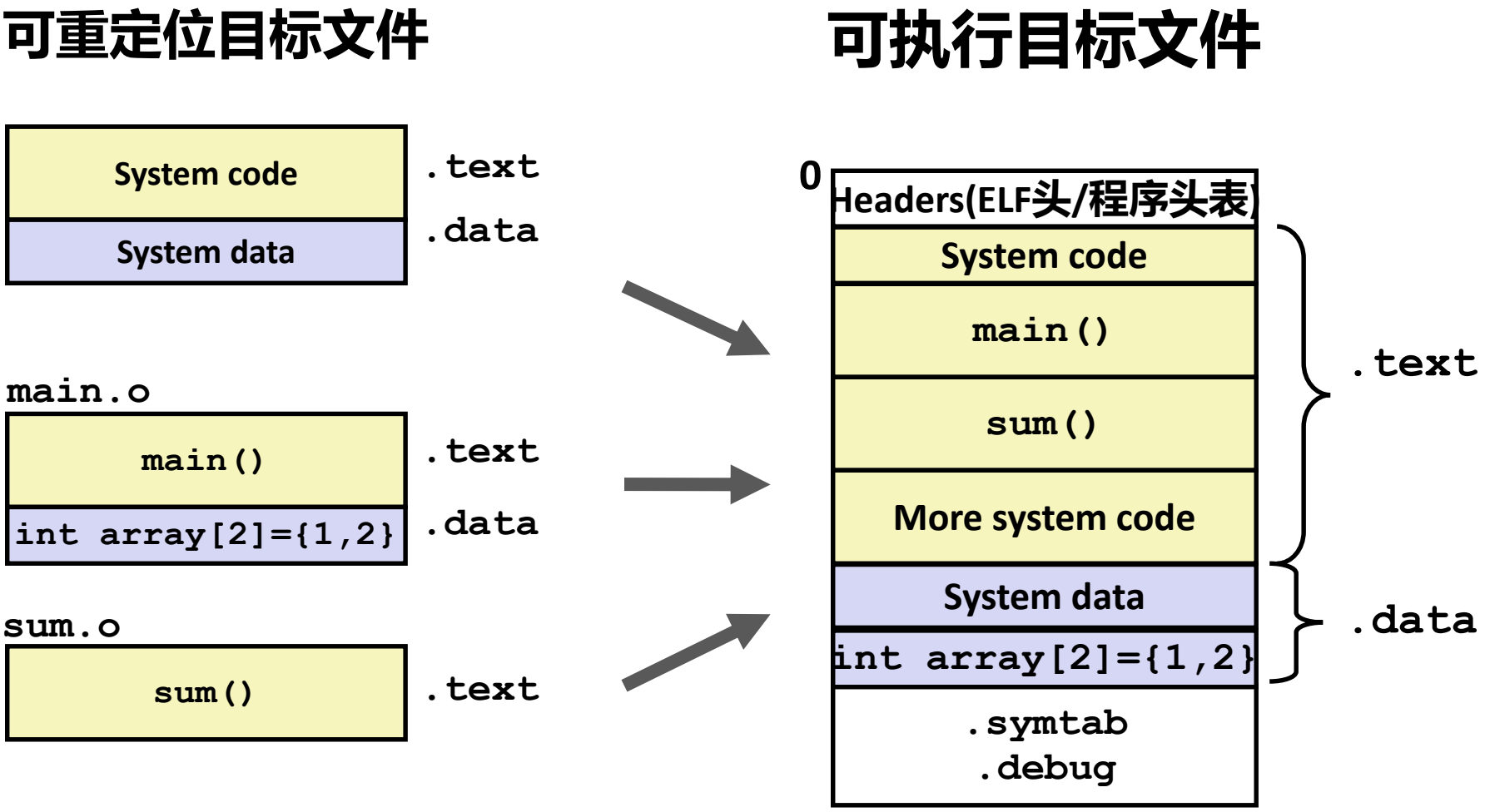
`int x=7;` `p1() {}`	`int x;` `p2() {}`	对x的引用将指向同名的初始化变量

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：如果你能**
- **否则**
  - 使用 `static` **：如果你能**
  - 定义了一个全局变量，就初始化它
  - 使用 `extern` ：如果你引用了一个外部全局符号

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

```c
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`

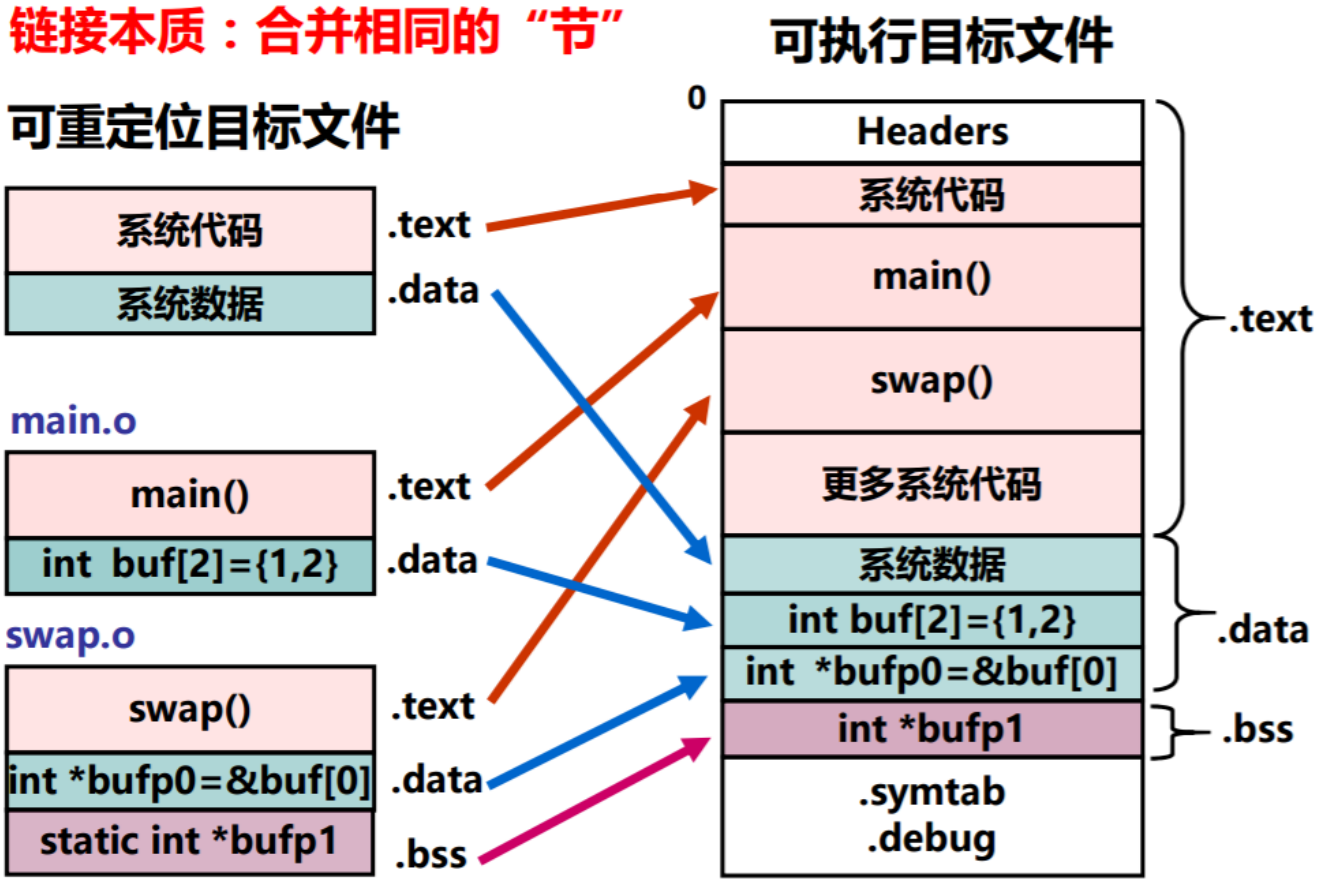
可重定位 .text 节

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

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

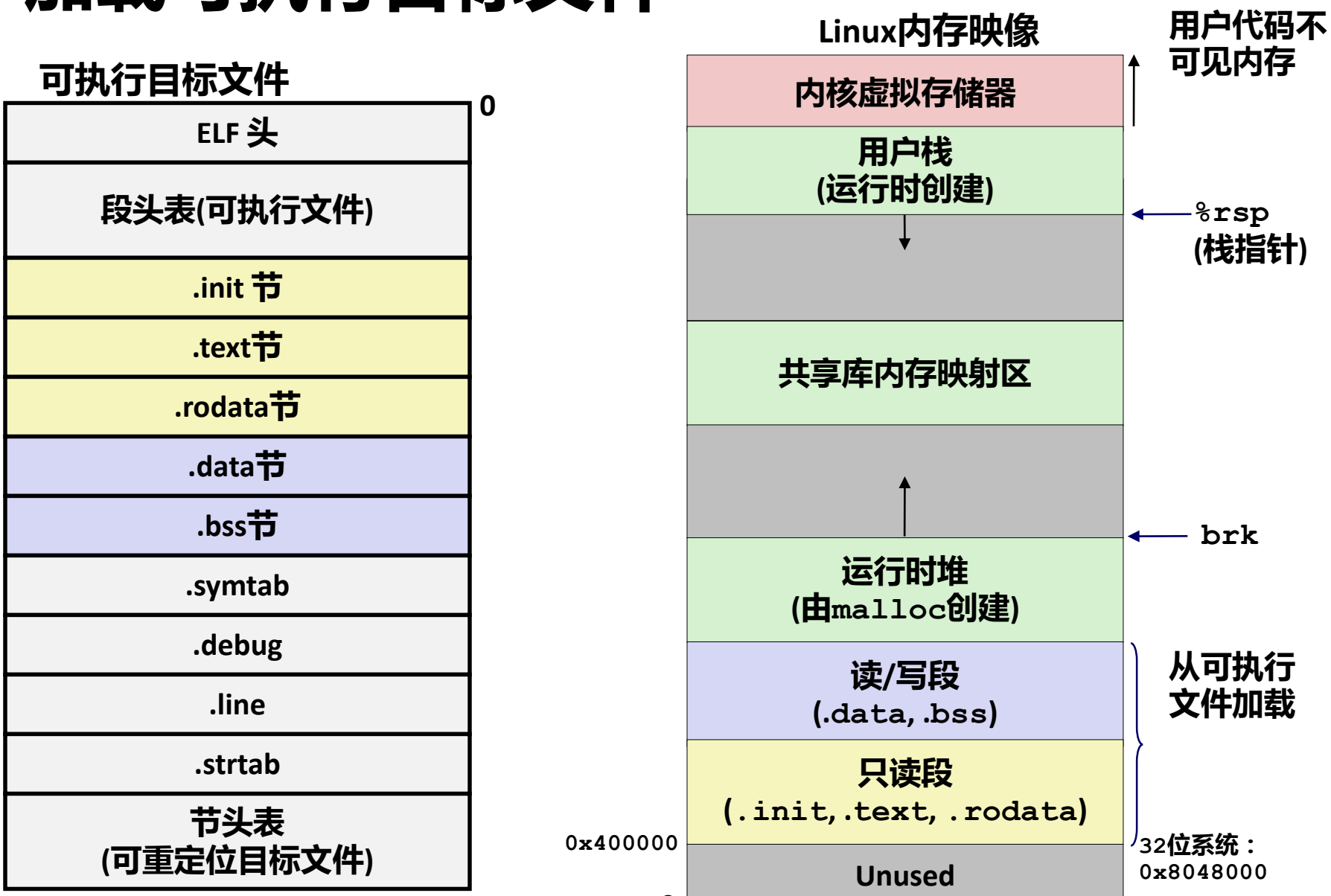
使用PC相对寻址 sum(): 0x4004e8 = 0x4004e3 + 0x5

来源: objdump -dx prog

链接过程的本质


链接本质：合并相同的“节”
可执行目标文件
可重定位目标文件
0
系统代码
系统数据
.text
.data
main.o
main()
int buf[2]={1,2}
.text
.data
swap.o
swap()
int *bufp0=&buf[0]
static int *bufp1
.text
.data
.bss
Headers
系统代码
main()
swap()
更多系统代码
系统数据
int buf[2]={1,2}
int *bufp0=&buf[0]
int *bufp1
.symtab
.debug
.text
.data
.bss

加载可执行目标文件


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
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)
从可执行
文件加载
只读段
(.init, .text, .rodata)
0x400000
32位系统：
0x8048000
Unused
0

常用的函数打包

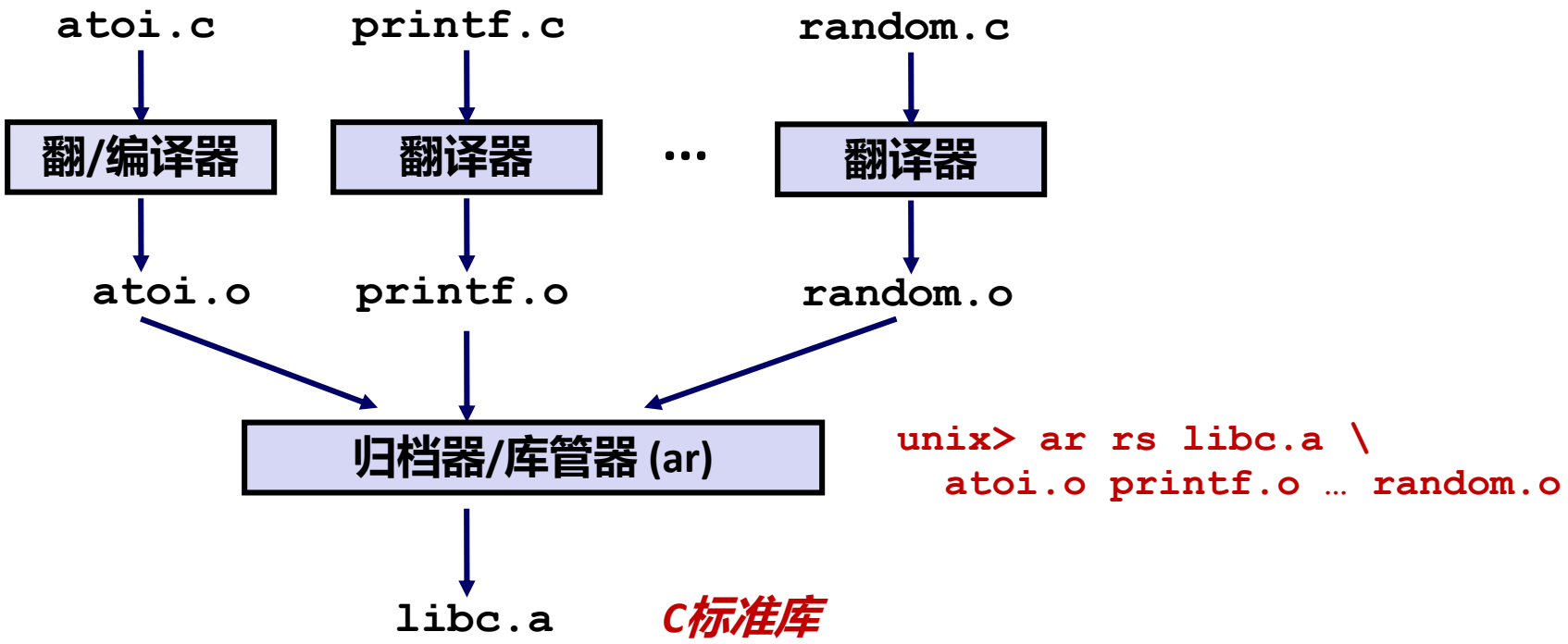
- **如何打包程序员常用的函数?**
  - Math, I/O, 存储管理, 串处理,等等.

- **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，使它尝试通过查找一个或多个存档文件中的符号来解决未解析的外部引用.
  - 如果一个存档成员文件解析了符号引用，就把它链接入可执行文件

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

常用用户库

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, …)

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

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

与静态库链接

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

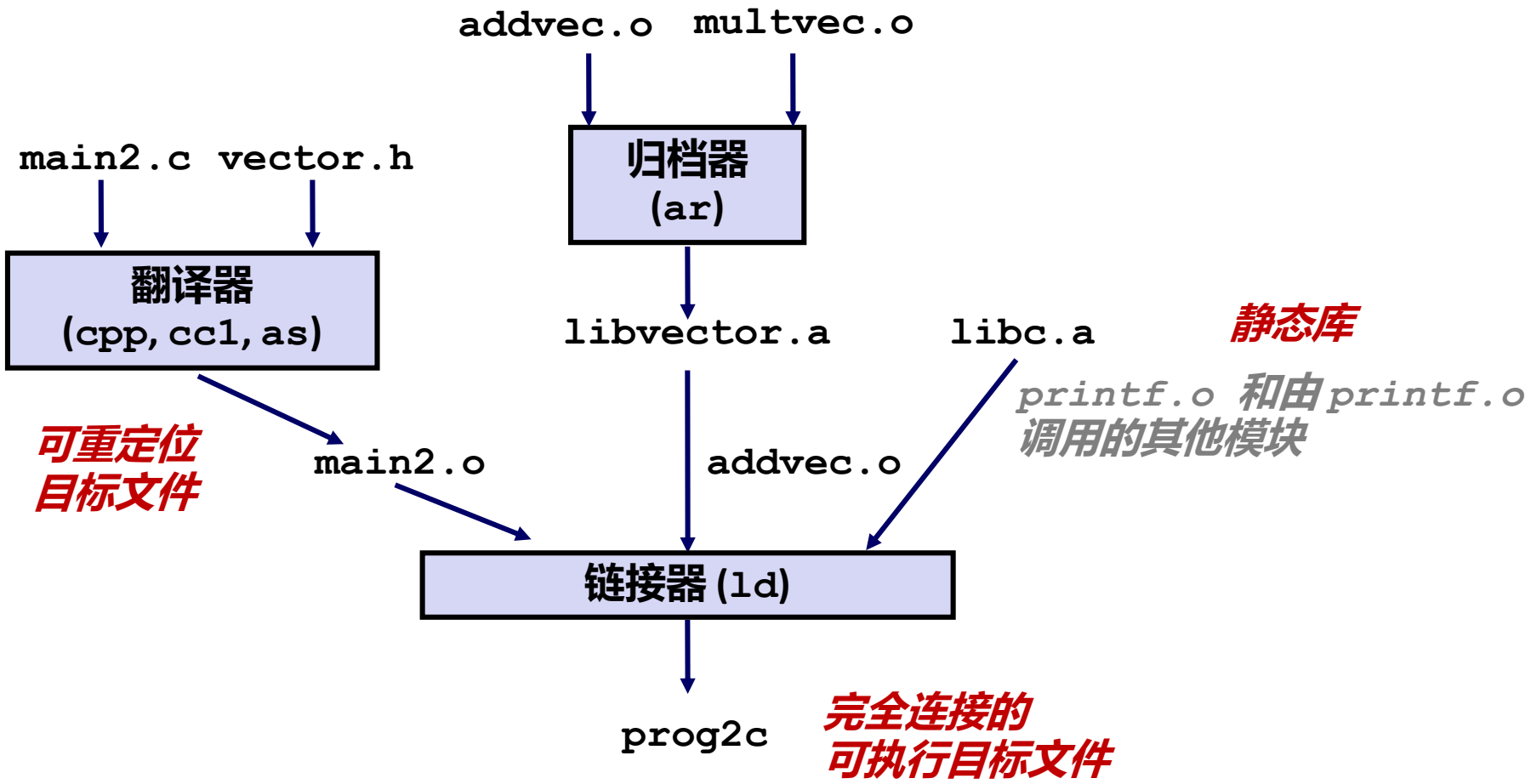
```c
void addvec(int *x, int *y,
            int *z, int n) {
    int i;

    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



"c" for "编译时 compile-time"

使用静态库

- **链接器的解析外部引用的算法:**
  - 按照在命令行的顺序扫描.o与 .a文件
  - 在扫描期间，保持一个当前未解析的引用列表.
  - 扫到每一个新的 **.o或 .a文件**, 遇到目标 *obj*,尝试解析列表中每个未解析的符号引用，而不是在obj中定义的符号。
  - 如果在扫描结束时，在未解析符号列表中仍存在任一条目，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

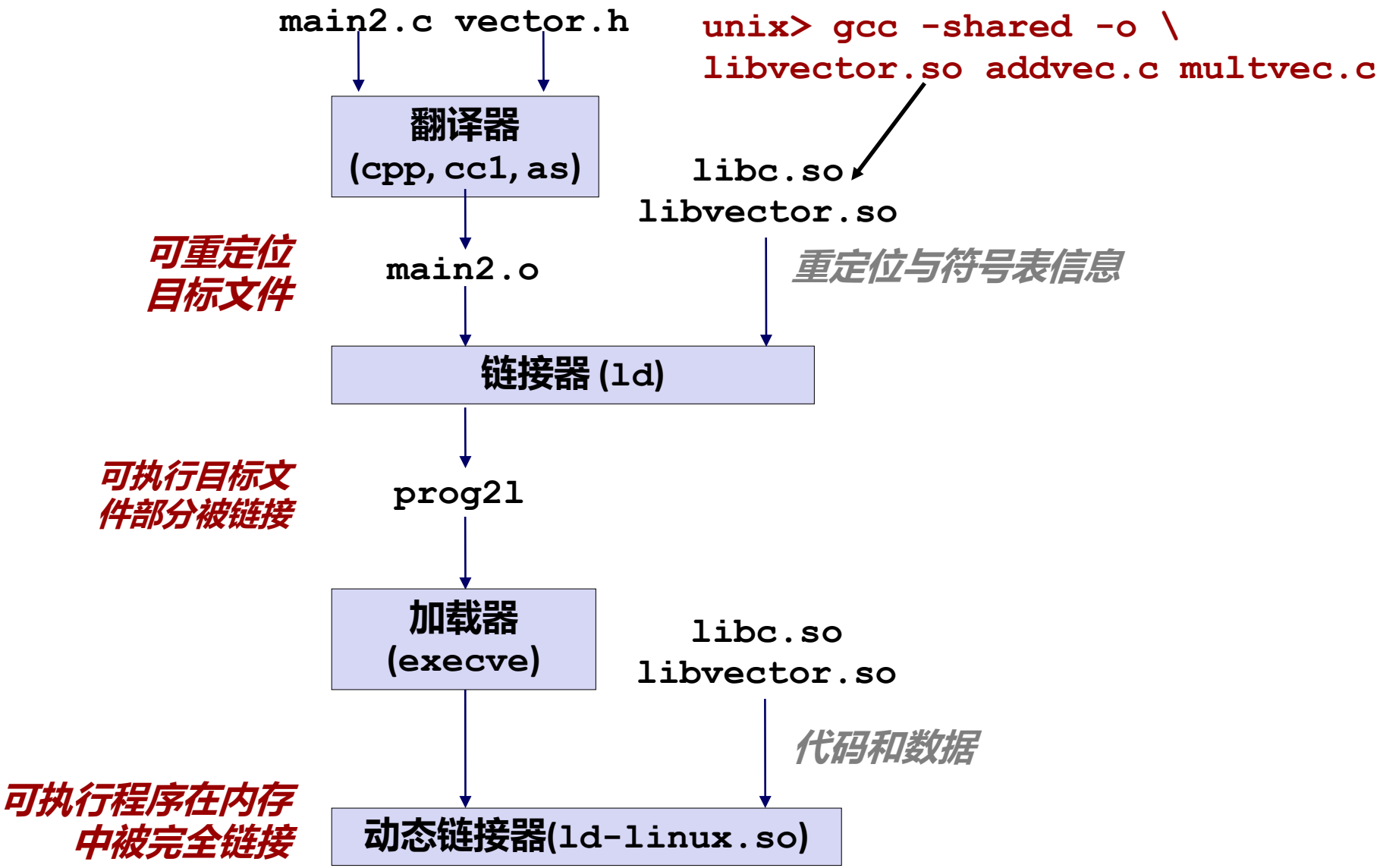
- **静态库有以下缺点:**
  - 在保存的可执行文件中存在重复 (每个函数都需要libc)
  - 在运行的可执行文件中重复
  - 系统库的小错误修复要求每个应用程序显式地重新链接

- **现代的解决方案:共享库**
  - 包含代码和数据的目标文件，在它们的加载时或运行时，被动态地加载并链接到应用程序中
  - 也称: 动态链接库, DLLs, `.so` 文件

共享库 (cont.)

- **当执行文件第一次加载和运行时(加载时链接)，动态链接就会出现.**
  - 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.
- **动态链接也可以在程序启动后进行(运行时链接).**
  - 在Linux中，这是通过调用dlopen()接口完成的 .
    - 分布式软件.
    - 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关于这个的内容

在加载时的动态链接


main2.c vector.h
unix> gcc -shared -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)
可执行目标文
件部分被链接
prog21
加载器
(execve)
libc.so
libvector.so
代码和数据
可执行程序在内存
中被完全链接
动态链接器(ld-linux.so)

在运行时的动态链接

```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);
    }
```

dll.c

在运行时的动态链接

```
    ...

    /* 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);
    }

    /*现在我们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;
}
```

dll.c

链接汇总

- **链接是一个技术： 允许从多个目标文件创建程序.**
- **链接可以在程序的生命周期的不同时间发生:**
  - 链接时(当程序被链接时) /GCC时（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)
- **理解链接可以帮助你避免讨厌的错误，让你成为一个更优秀的程序员。**

要点

- Linking
- **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)
  - 在幕后加密
- **调试**
  - 2014年，两名Facebook工程师使用了打桩机制，调试了他们的iPhone应用程序中一个危险的1年之久的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

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
  - 描述Characterize调用地点sites和函数的参数
  - Malloc 跟踪
    - 检测内存泄露
    - **生成地址痕迹traces**

程序实例

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

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

int.c

- **目标:跟踪已分配和自由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lib malloc和free函数进行打桩

编译时打桩

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

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

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

mymalloc.c

编译时打桩

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

void *mymalloc(size_t size);
void myfree(void *ptr);
```

malloc.h

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

链接时打桩

```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) = 0x1aa0010
free(0x1aa0010)
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:
  - 对 `malloc` 的引用，必须被解析为 `__wrap_malloc`
  - 对 `__real_malloc` 必须被解析为 `malloc`

加载/运行时打桩

```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

Load/Run-time Interpositioning

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

- **LD_PRELOAD环境变量告诉动态链接器，首先通过查看mymalloc.so，解析未解析的符号引用(例如malloc)。**

打桩回顾

- **编译时**
  - 对malloc/free的显式调用将宏扩展到对mymalloc/myfree的调用
- **链接时**
  - 使用链接技巧trick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：用不同的名字，使用动态链接来加载库malloc/free