第7章　链接

要点

- **链接**
  - 符号解析
  - 重定位（重点）
- **目标文件格式**
  - 可重定位目标文件
  - 可执行目标文件
- **静态链接**
- **动态链接**

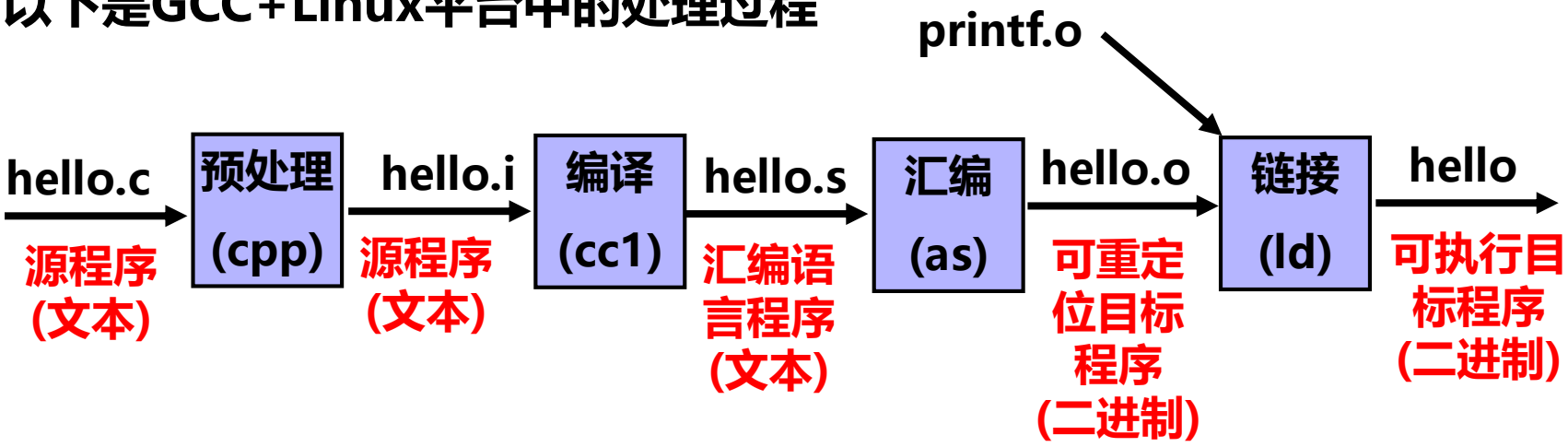
可执行程序是怎么生成的?

经典的“ hello.c ” C-源程序

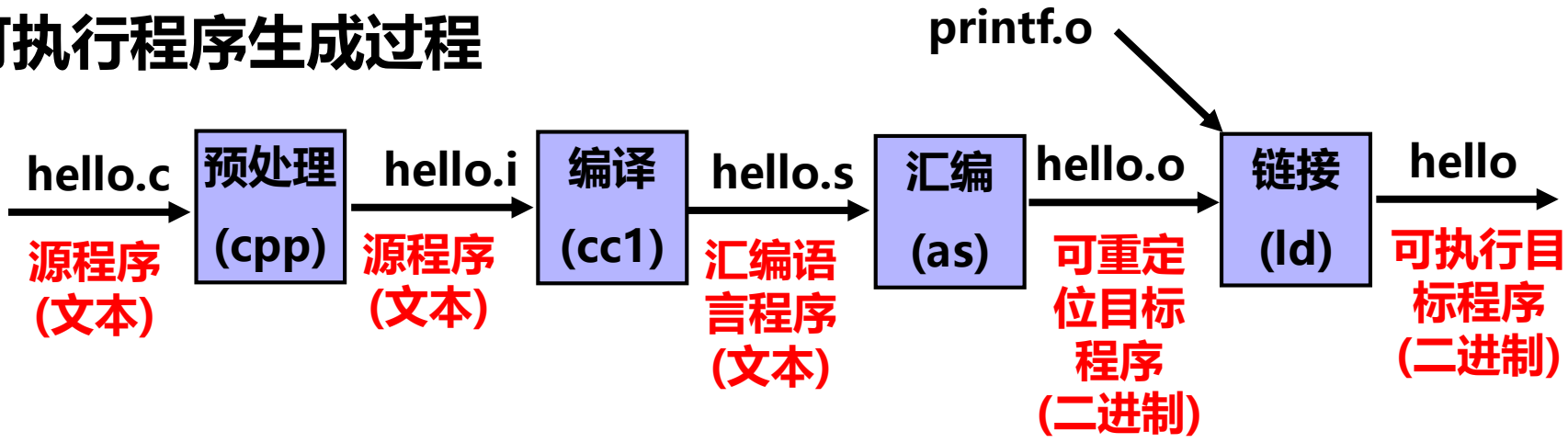
```
#include <stdio.h>

int main()
{
  printf("hello, world\n");
}
```

功能：输出“hello,world”　　计算机不能直接执行hello.c!

以下是GCC+Linux平台中的处理过程



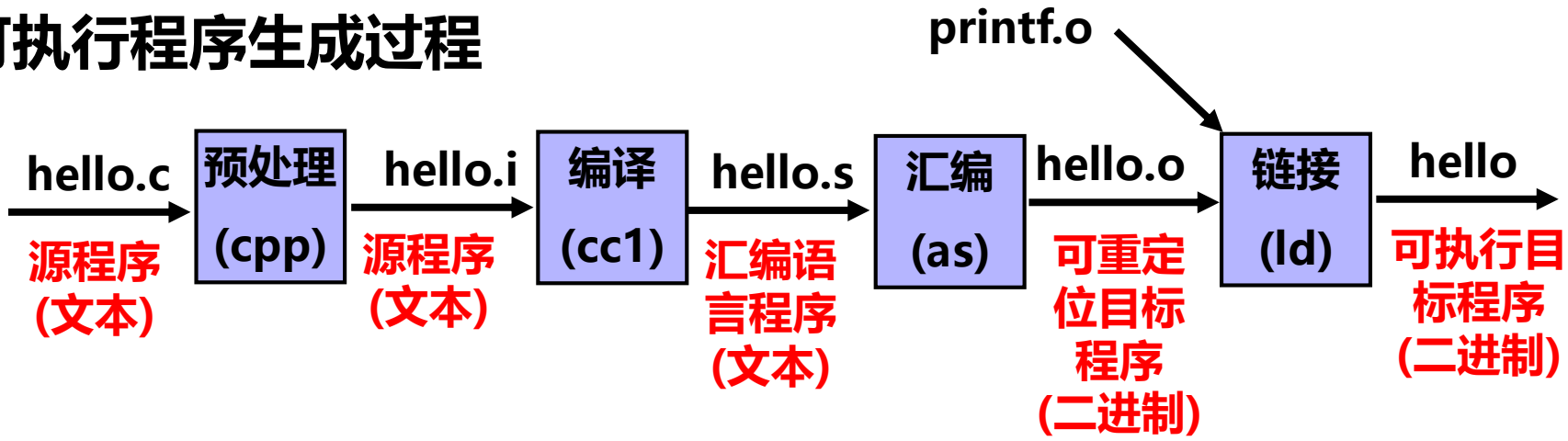
可执行程序生成过程



- **预处理**
  - 预处理命令（cpp是C preprocessor “即C预处理程序”的英文缩写）
    - $gcc –E hello.c –o hello.i
    - $cpp hello.c > hello.i
  - 处理源文件中以“#”开头的预编译指令，包括：
    - 删除“#define”并展开所定义的宏
    - 处理所有条件预编译指令，如“#if”，“#ifdef”，“#endif”等
    - 插入头文件到“#include”处，可以递归方式进行处理
    - 删除所有的注释“//”和“/**/”
    - 添加行号和文件名标识，以便编译时产生调试用的行号信息
    - 保留所有#pragma编译指令（编译器需要用）

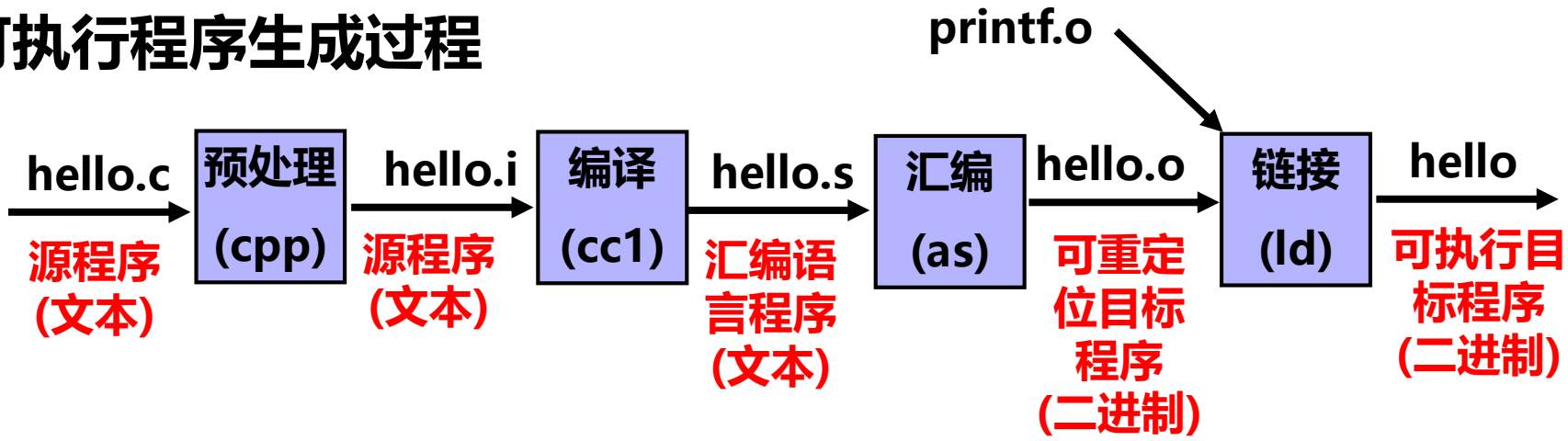
    备注： #pragma 指令的作用是设定编译器的状态或者是指示编译器完成一些特定的动作。
  - 经过预处理后得到的预处理文件（如hello.i），它还是一个可读的文本文件，但不包含任何宏定义

可执行程序生成过程



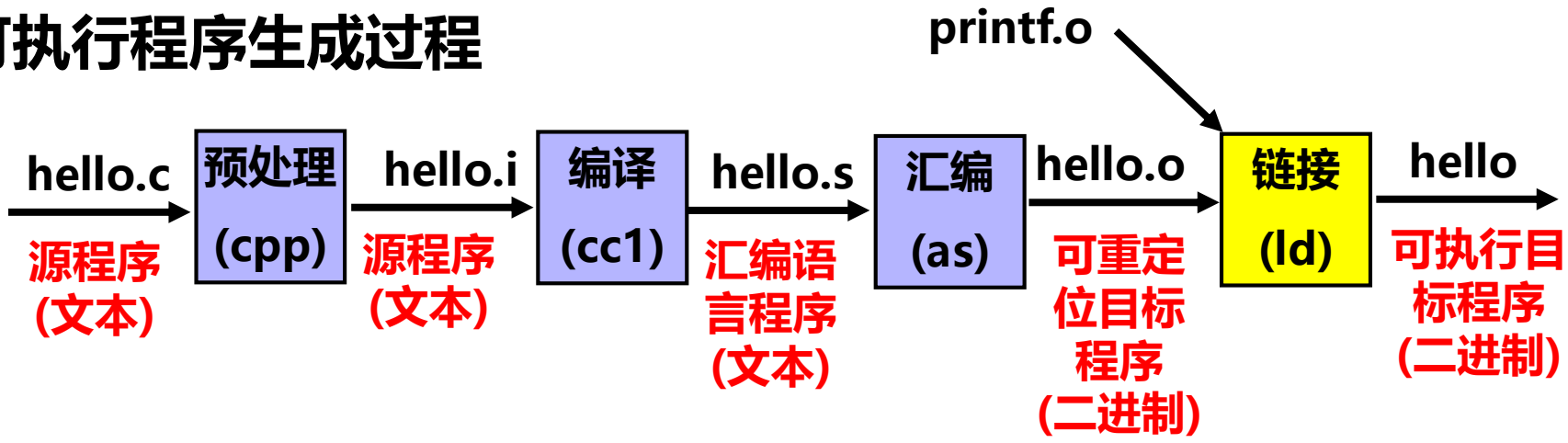
- **编译**
  - 编译过程就是将预处理后得到的预处理文件（如hello.i）进行词法分析、语法分析、语义分析、优化后，生成汇编代码文件
  - 用来进行编译处理的程序称为**编译程序**
  - 编译命令
    - $gcc –S hello.i –o hello.s
    - $gcc –S hello.c –o hello.s
    - $/user/lib/gcc/i486-linux-gnu/4.1/cc1 hello.c
  - 经过编译后，得到的汇编代码文件（如hello.s）还是可读的文本文件，CPU无法理解和执行它

可执行程序生成过程



- **汇编**
  - 汇编代码文件（由汇编指令构成）称为汇编语言源程序
  - 汇编程序（汇编器）用来将汇编源程序转换为机器指令序列（机器语言程序）
  - 汇编指令和机器指令一一对应，前者是后者的符号表示，它们都属于机器级指令，所构成的程序称为机器级代码
  - 汇编命令
    - $gcc –c hello.s –o hello.o
    - $gcc –c hello.c –o hello.o
    - $as hello.s –o hello.o
  - 汇编结果是一个**可重定位目标文件**（如hello.o），其中包含的是不可读的二进制代码，必须用相应的工具软件来查看其内容

可执行程序生成过程



- **链接**
  - 预处理、编译和汇编三个阶段针对一个模块（一个*.c文件）进行处理，得到对应的一个可重定位目标文件（一个*.o文件）
  - 链接过程将多个可重定位目标文件合并以生成可执行目标文件
  - 链接命令
    - $gcc –static –o myproc main.o test.o
    - $ld –static –o myproc main.o test.o
    - -static表示静态链接，如果不指定-o选项，则可执行文件名为“a.out”

两个可重定位文件

链接的定义：将代码和数据收集并组合成一个文件的过程，最终得到的文件可以被加载到内存执行。或者说，把可重定位目标文件以及必要的系统文件组合起来，生成一个可执行目标文件的操作。

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，而不是一个整体的大文件
  - 可以构建公共函数库 (代码复用)
    - 例如, 数学运算库, 标准C库
- **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

静态链接

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

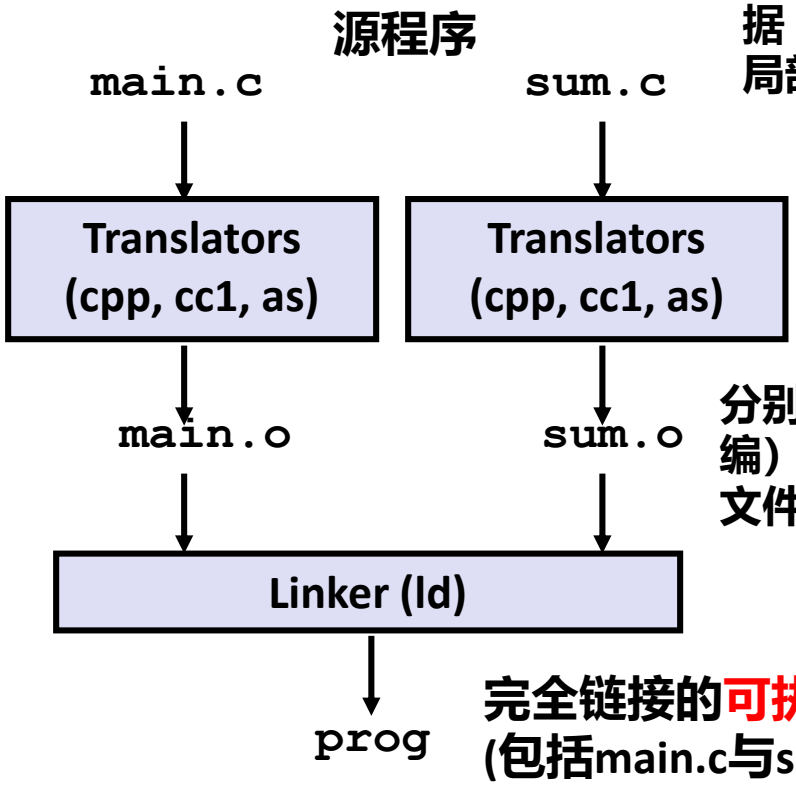
备注：-Og代表生成的机器代码要符合原始C代码的结构，方便调试

main.c文件内容：

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

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

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

sum.c文件内容

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

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



每个模块有自己的代码、数据（全局变量、静态变量、局部变量等）

问题：如何进行合并？？

分别转换（预处理、编译、汇编）为可重定位目标文件（.o文件）

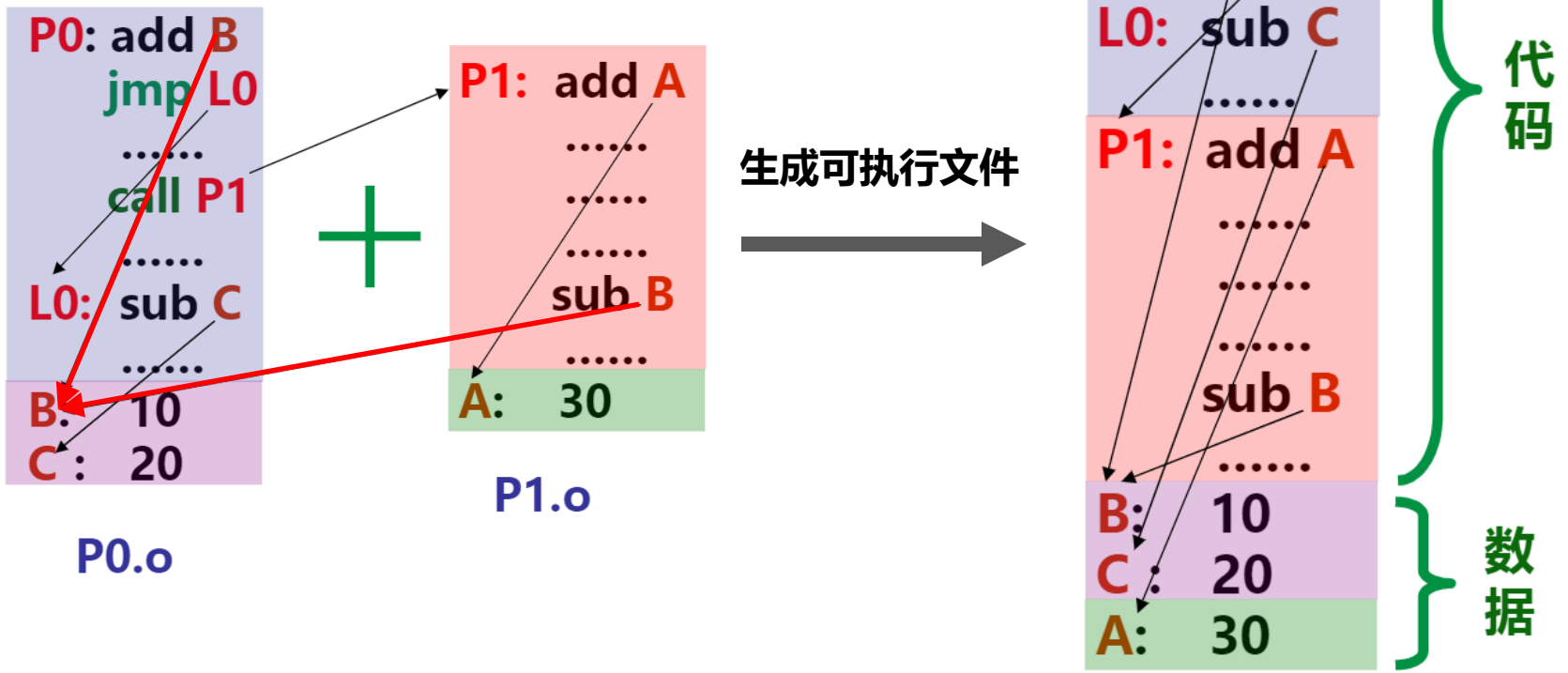
完全链接的可执行目标文件
(包括main.c与sum.c定义的所有函数的代码和数据)

链接过程：1. 符号解析；2. 重定位

出填空题

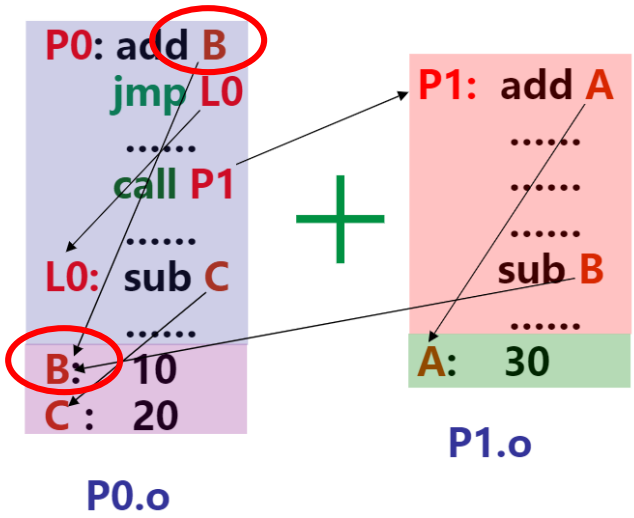
- **链接过程：**
  - 确定符号引用关系（**符号解析**）→ **确定符号引用和符号定义的关系**
  - 合并相关.o文件
  - 确定每个符号的地址
  - 在指令中填入新地址

  } **重定位**



链接过程：1. 符号解析；2. 重定位

- **步骤 1: 符号解析→什么是符号?**
  - 程序定义和引用符号 (全局变量和函数):
    - `void swap() {…}` /* 定义符号swap */
    - `swap();` /* 引用符号swap */
    - `int *xp = &x;` /* 定义符号xp，引用符号x */
  - **局部变量和参数等等经编译器处理后不再是符号了！！！**
  - 符号定义存储在目标文件中(由汇编器) 的符号表中.
    - 符号表是一个结构型的数组
    - 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正好与一个符号定义关联起来



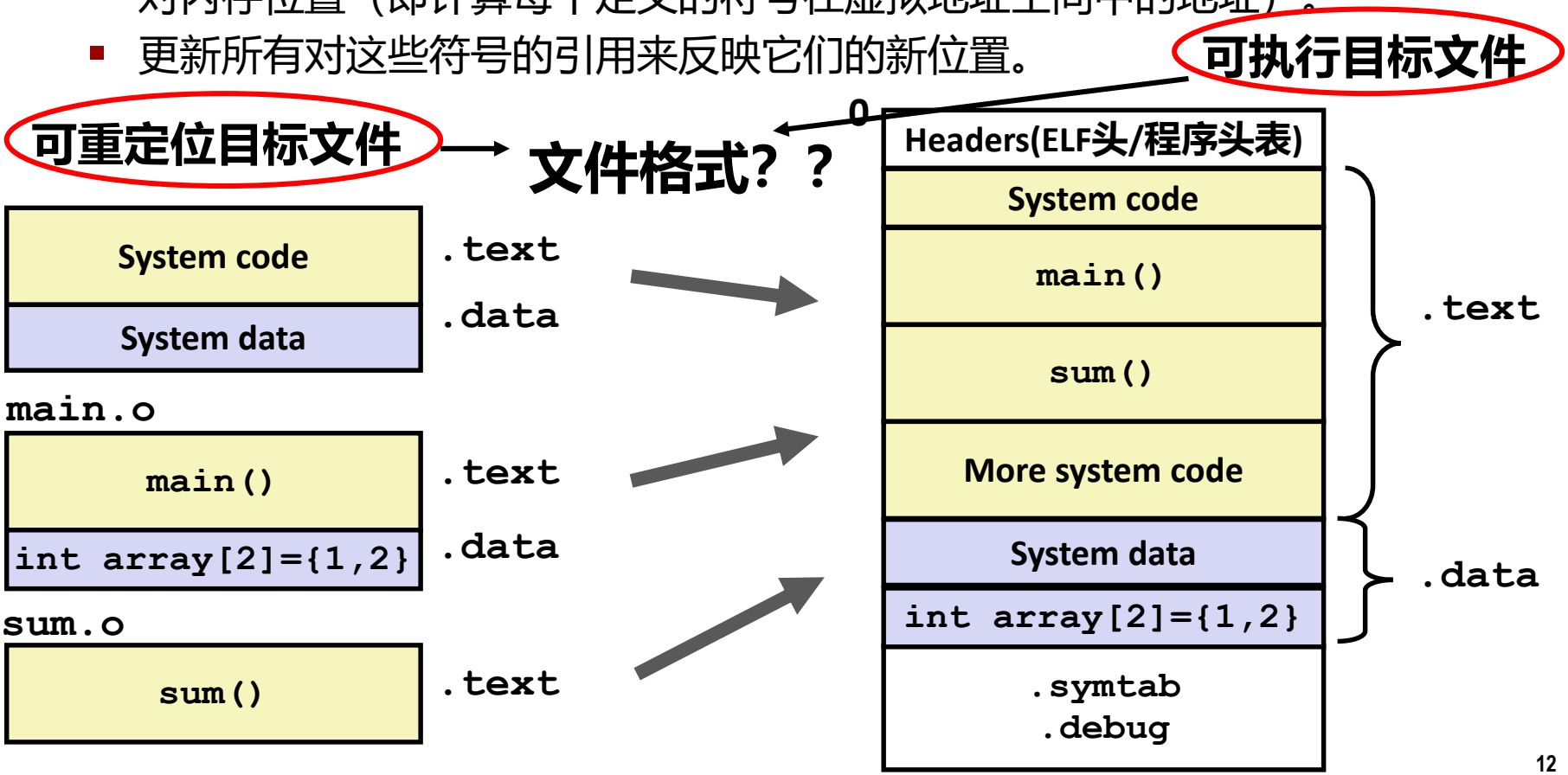
虚拟地址

符号定义的实质是什么?

- 指被分配了存储空间。如果是函数名即指其代码所在区；变量名即指其所占的静态数据区
- 所有定义符号的值就是其目标所在的首地址

链接过程：1. 符号解析；2. 重定位

- **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（即计算每个定义的符号在虚拟地址空间中的地址）。
  - 更新所有对这些符号的引用来反映它们的新位置。



目标文件格式

- **标准的几种目标文件格式**
  - Linux&Unix: Executable and Linkable Format (ELF)
  - Windows: Portable Executable(PE)

- **三种目标文件都采用ELF二进制格式**
  - 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）

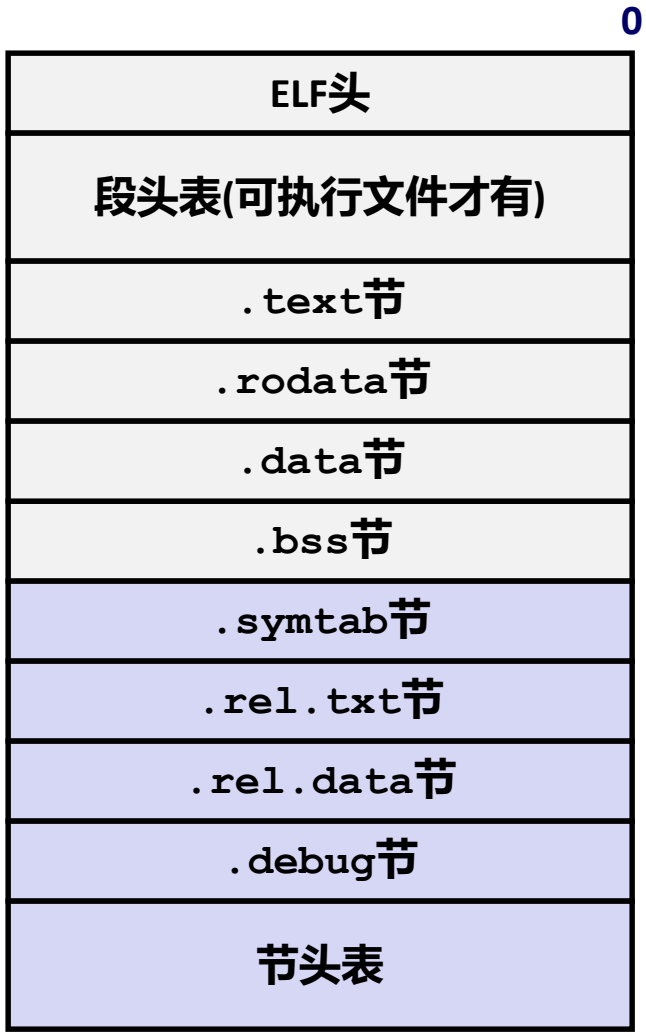
- **工具: readelf 文件名**

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
    - 每一个.o文件代码和数据地址都从0开始
- **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
  - 代码和数据地址为虚拟地址空间中的地址
- **共享目标文件(.so 文件)**
  - 特殊类型的可重定位目标文件，在其他的程序加载时或运行时，它可以被动态加载到内存，并被动态链接
  - 在Windows中称为动态链接库(.DLL)

ELF目标文件格式(cont.)

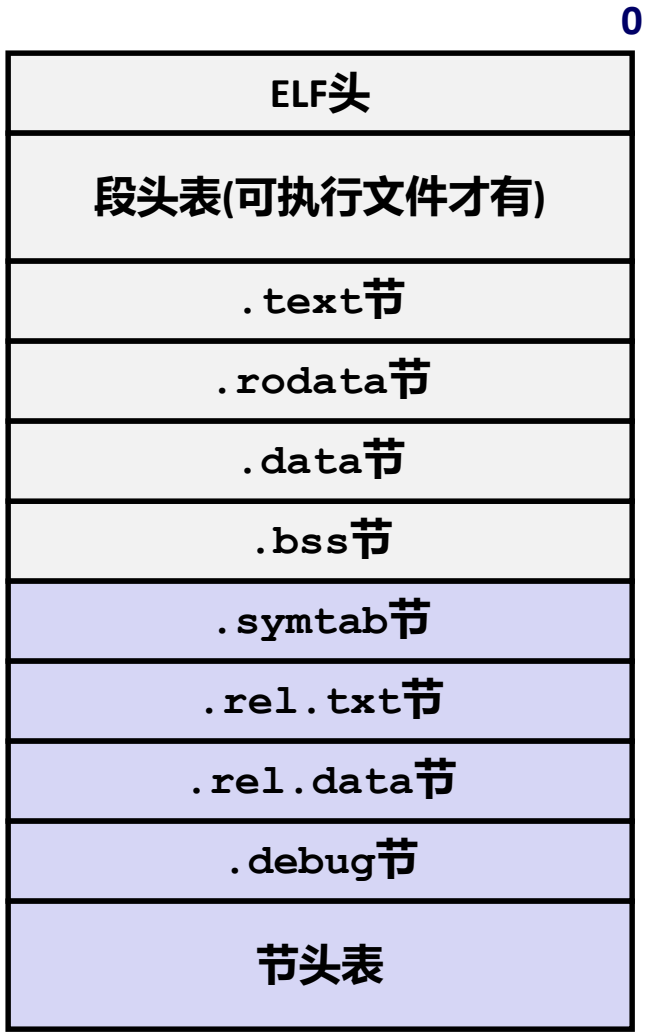
- **ELF 头**
  - 字大小、字节顺序、文件类型(.o，exec，.so)，机器类型（如IA-32），节头表的偏移、节头表的表项大小以及表项个数等等
- **段头表/程序头表（可执行文件）**
  - 页面大小，虚拟地址内存段(节)，段大小
- **节头表Section header table**
  - 每个节的节名、偏移量和大小
- **.text 节（代码）**
- **.rodata 节（只读数据）**
  - 只读数据 : printf格式串、 跳转表, ...
- **.data 节 （数据/可读写）**
  - 已初始化全局变量
- **.bss 节 （未初始化全局变量）**
  - 未初始化的全局变量，仅是占位符，不占据任何实际磁盘空间。(Better Save Space)



ELF目标文件格式(cont.)

- 各个节所包含的内容
- 可重定位和可执行文件的区别

- **.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)
- **.strtab节**
  - 包含symtab和debug节中的符号及节名



每个节是如何区分开来的？ ⟶ 节头表

节头表

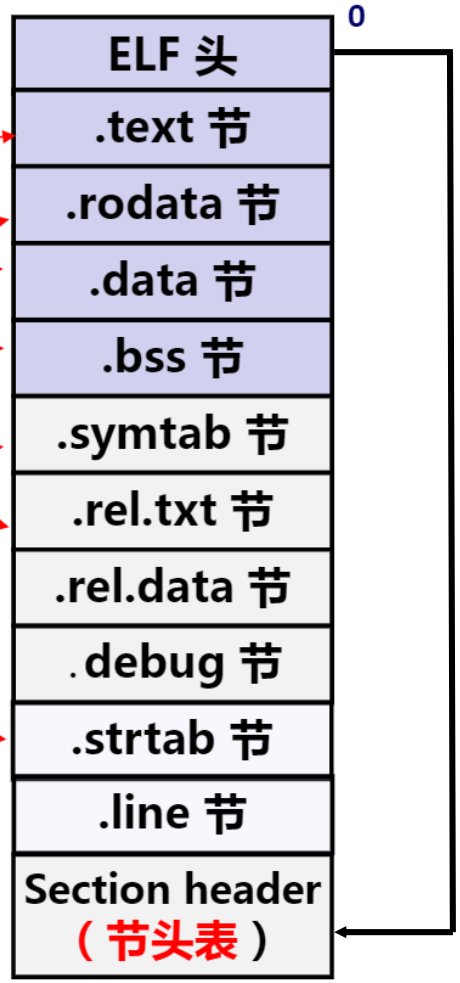
节头表是ELF可重定位目标文件中最重要的内容，描述每个节的节名、在文件中的偏移、大小、访问属性、对齐方式等。

从ELF头中知道节头表的文件偏移，以及节头表中条目的大小和数量

```
$ readelf -S test.o
```

打印节头表信息

```
There are 11 section headers, starting at offset 0x120:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        00000000 000034 00005b 00  AX  0   0  4
  [ 2] .rel.text         REL             00000000 000498 000028 08      9   1  4
  [ 3] .data             PROGBITS        00000000 000090 00000c 00  WA  0   0  4
  [ 4] .bss              NOBITS          00000000 00009c 00000c 00  WA  0   0  4
  [ 5] .rodata           PROGBITS        00000000 00009c 000004 00   A  0   0  1
  [ 6] .comment          PROGBITS        00000000 0000a0 00002e 00      0   0  1
  [ 7] .note.GNU-stack   PROGBITS        00000000 0000ce 000000 00      0   0  1
  [ 8] .shstrtab         STRTAB          00000000 0000ce 000051 00      0   0  1
  [ 9] .symtab           SYMTAB          00000000 0002d8 000120 10     10  13  4
  [10] .strtab           STRTAB          00000000 0003f8 00009e 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
```



可重定位目标文件中，每个可装入节的起始地址总是0

- Name: 节名
- Type: 节的类型（无效/保存程序相关信息/符号…）
- Addr: 虚拟地址
- Off: 在文件中的偏移地址
- Size: 节在文件中所占长度
- ES(Entsize): 节中每个表项的长度，0表示无固定长度表项
- Lk、Inf: 用于与链接相关的节（如.rel.text节、.rel.data节、.symtab节等）
- Al(Addralign): 节的对齐要求

节头表信息举例

$ readelf -S test.o

There are 11 section headers, starting at offset 0x120:

Section Headers:

[Nr] Name	Off	Size	ES	Flg	Lk	Inf	Al
[0]	000000	000000	00		0	0	0
[1] .text	000034	00005b	00	AX	0	0	4
[2] .rel.text	000498	000028	08		9	1	4
[3] .data	000090	00000c	00	WA	0	0	4
[4] .bss	00009c	00000c	00	WA	0	0	4
[5] .rodata	00009c	000004	00	A	0	0	1
[6] .comment	0000a0	00002e	00		0	0	1
[7] .note.GNU-stack	0000ce	000000	00		0	0	1
[8] .shstrtab	0000ce	000051	00		0	0	1
[9] .symtab	0002d8	000120	10		10	13	4
[10] .strtab	0003f8	00009e	00		0	0	1

Key to Flags:

W (write), A (alloc), X (execute), M (merge), S (strings)

I (info), L (link order), G (group), x (unknown)

.....

4

3

1

2

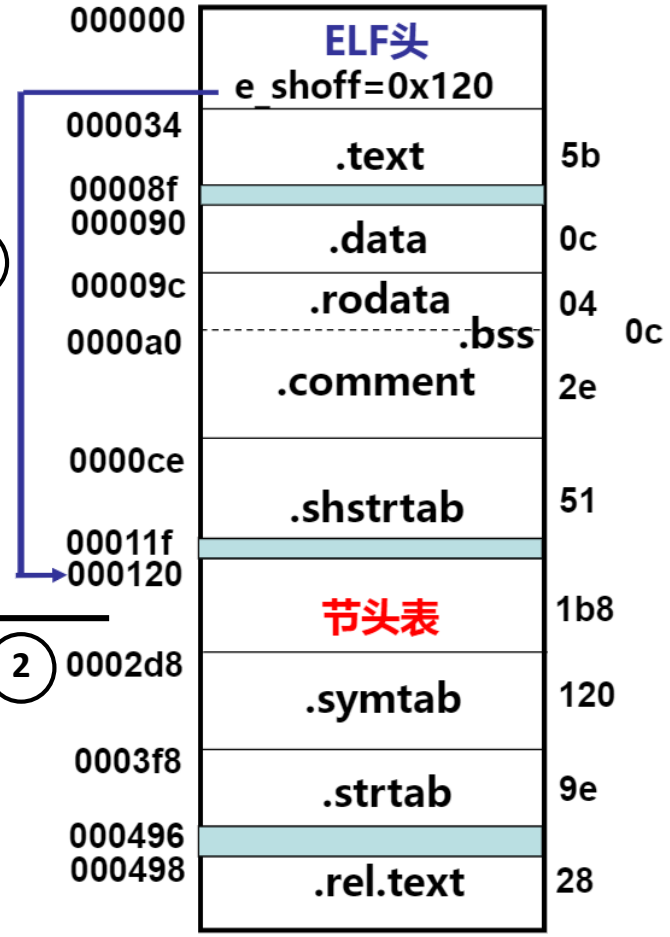
有4个节将会分配存储空间

.text：可执行

.data和.bss：可读可写

.rodata：可读

可重定位目标文件test.o的结构



步骤 1: 符号解析

- **链接过程：**
  - 确定符号引用关系（**符号解析**）
  - 合并相关.o文件
  - 确定每个符号的地址
  - 在指令中填入新地址

  } **重定位**

每个重定位目标模块m都有一个符号表(.symtab节)，它包含m定义和引用的符号信息，有三种不同的符号：

- **全局符号**
  - 由该模块定义的，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**
  - 由该模块引用的全局符号，但由其他模块定义。
- **本地/局部符号**
  - 由该模块定义和仅由该模块唯一引用的符号，不能被外部模块引用
  - 如:使用静态属性定义的C函数和全局变量
  - 注意：程序的非static局部变量不是本地/局部符号

符号：对应于一个函数、一个全局变量或一个静态变量
（注意，这里是C 语言中任何以static 属性声明的变量）

步骤 1: 符号解析

在符号解析步骤中，链接器将每个符号引用与一个符号定义关联起来

- Value: 在对应节的偏移量
- Size: 符号对应目标字节数（函数大小或变量长度）
- Type、Bind: 类型和绑定属性（全局/局部符号）
- Ndx: 符号对应目标所在节(例如1表示在.text节)

可重定位目标文件main.o**符号表**(.symtab节)中的最后三个条目

Num	Name	Value	Size	Type	Bind	Vis	Ndx
8:	main	0000000000000000	24	FUNC	GLOBAL	DEFAULT	1
9:	array	0000000000000000	8	OBJECT	GLOBAL	DEFAULT	3
10:	sum	0000000000000000	0	NOTYPE	GLOBAL	DEFAULT	UND

…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

引用一个全局符号…

定义一个全局符号

链接器不知道 val的任何信息

引用全局符号…

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

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

…它在这儿定义

链接器不知道i或s的任何信息

本地符号

- **本地非静态C变量（非本地符号）vs 本地静态C变量（本地符号）**
  - 本地非静态C变量: 存储在栈或寄存器上（运行时），不是符号
  - 本地的静态C变量: 存储在 `.bss` 或 `.data`，是符号

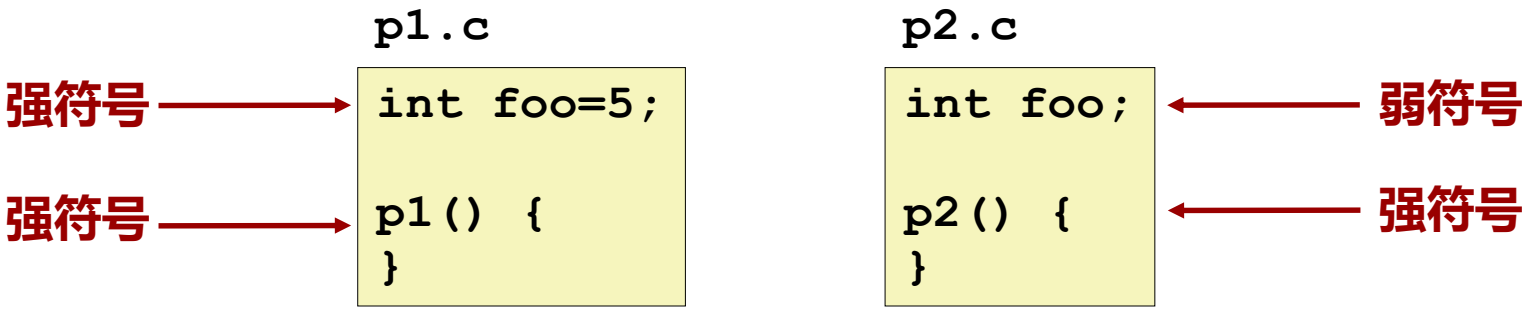
```
int f()
{
    static int x = 0;
    return x;
}

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.

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

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

`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.

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

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

在p2中写入x可能会覆盖y！

由于x都是弱符号，所以有两种情况：

1）当链接器选择了double x，此时x是double（8个字节），y是int（4个字节），x和y的描述相关信息存在符号表里，p1仅仅使用x前面的四个字节，p2则使用8个字节，此时double x和int y在内存没有重叠地址部分。

2）当链接器选择了int x，此时x是int（4个字节），y也是int（4个字节），x和y的描述相关信息存在符号表里，注意p2使用的是基于x首地址的连续8个字节，这时候就覆盖了y，此时double x的高4个字节和int y在内存上重叠。

总结：在p2中写入x可能会覆盖y，“可能”表示发生哪种情况取决于编译器。

注意：gcc在链接的时候通常选择空间大的比如double x，这样避免了覆盖y，见《程序员的自我修养》P92

区别

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

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

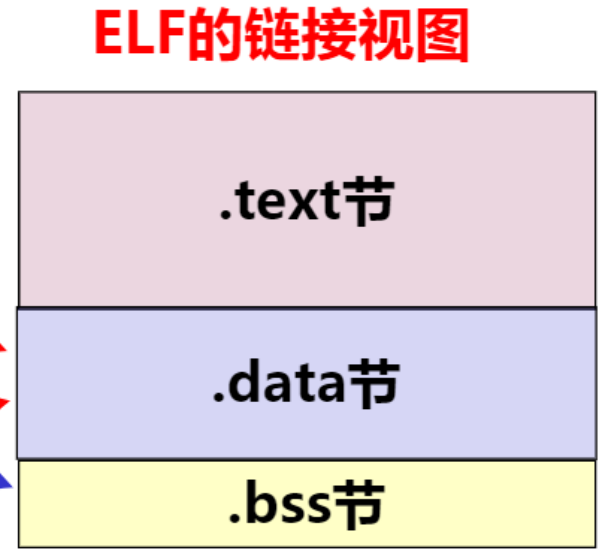
在p2中写入x将覆盖y！

由于x=7是强符号，所以链接器选择了int x=7，此时x是int（4个字节），y也是int（4个字节），x和y的描述相关信息存在符号表里，注意p2使用的是基于x首地址的连续8个字节，这时候就覆盖了y，此时double x的高4个字节和int y在内存上重叠。

总结：在p2中写入x将覆盖y，此种情况一定发生。

各部分存放位置

```
int x=100;
int y;
void prn(int n)
{
   printf( "%d\n" ,n);
}

void main( )
{
    static int a=1;
    static int b;
    int i=200,j;
    prn(x+a+i);
}
```



为了进行链接，还需要其他许多信息，如符号表、重定位信息等许多其他的节（Section）

.text节: 已编译程序的机器代码

.data节: 已初始化的全局和静态C变量

.bss: 未初始化的全局和静态C变量

有时候还这样进一步划分:
1）COMMON 未初始化的全局变量
2）.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

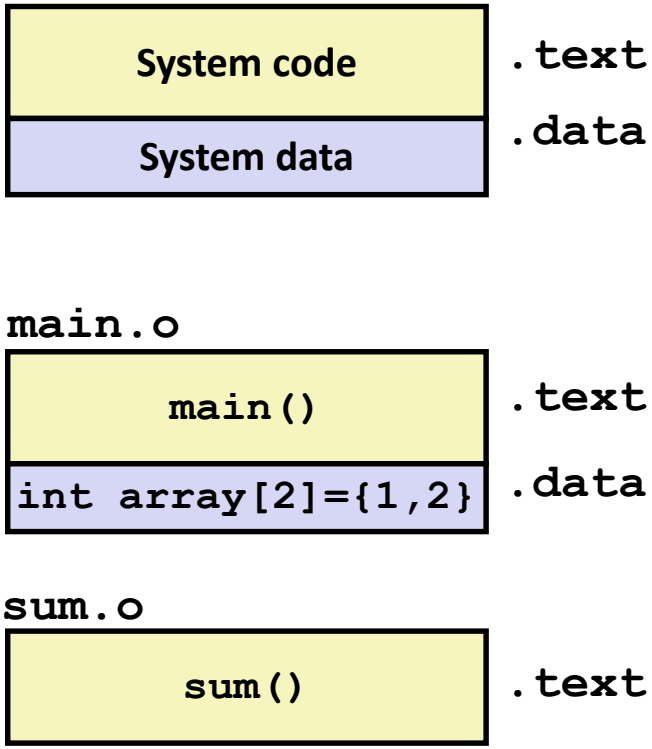
不同类型的符号存放位置不同

步骤 2: 重定位

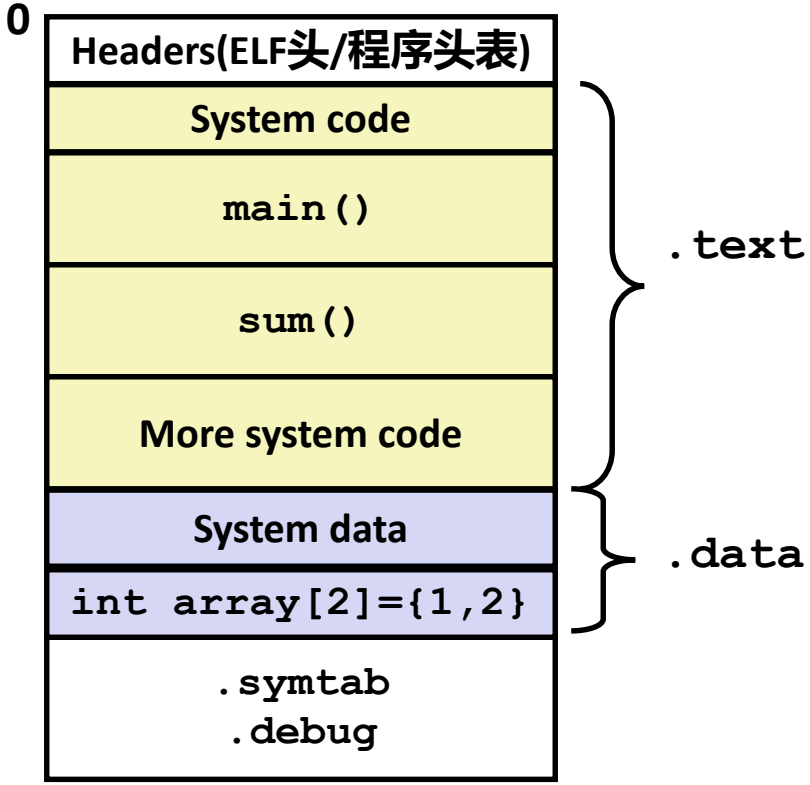
- **链接过程：**
  - 确定符号引用关系（**符号解析**）
  - 合并相关.o文件
  - 确定每个符号的地址
  - 在指令中填入新地址

  } **重定位**

可重定位目标文件



可执行目标文件



步骤 2: 重定位

- **符号解析完成后，可进行重定位操作，分三步**
  - 第一步：合并相同的节
    - 将所有目标模块中相同的节合并成新节
      - 例如，所有.text节合并作为可执行文件中的.text节
  - 第二步：对定义符号进行重定位（确定地址）
    - 确定新节中所有定义符号在虚拟地址空间中的地址
      - 例如，为函数确定首地址，进而确定每条指令的地址，为变量确定首地址
    - 完成这一步后，每条指令和每个全局或局部变量都可确定地址
  - 第三步：对引用符号进行重定位（确定地址）
    - 修改.text节和.data节中每个符号的引用（地址）
    - 需要用到在.rel.data和.rel.text节中保存的重定位信息

↓

可重定位条目（见下页）

问题：为什么要进行重定位?

当汇编器生成一个可重定位目标模块时，它并不知道数据和代码最终将放在内存中的什么位置。它也不知道这个模块引用的任何外部定义的函数或者全局变量的位置。所以，无论何时汇编器遇到对最终位置未知的目标引用，它就会生成一个重定位条目，告诉链接器在将目标文件合并成可执行文件时如何修改这个引用。代码的重定位条目放在.rel.text中。已初始化数据的重定位条目放在.rel.data中。

可重定位条目

```
1   typedef struct {
2       long offset;
3       long type:32,
4            symbol:32;
5       long addend;
6   } Elf64_Rela;
```

可重定位条目（以两类为例）

需要被修改的引用的节内偏移

重定位类型：1. R_X86_64_PC32（PC相对地址）
2. R_X86_64_32（绝对地址）

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

被修改引用应该指向的符号

偏移调整（由重定位类型确定）

存放在.rel.data (可重定位数据) 或者rel.text (可重定位代码)

例如，在.rel.text节中有重定位条目：
offset: 0x1
symbol: B
type: R_x86_64_32

用命令readelf –r main.o可显示main.o中的重定位条目

可重定位条目

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

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

- 对引用符号进行重定位（确定地址）
  - 修改.text节和.data节中每个符号的引用（地址）
  - 需要用到在.rel.data和.rel.text节中保存的重定位信息

占位符00 00 00 00

```
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
```

重定位条目

r.offset = 0xf

r.symbol = sum

r.type = R_X86_64_PC32

r.addend = -4

main.o

来源: `objdump -r -d main.o`

重定位算法→对引用符号进行重定位

引用的运行时地址

节s的运行时地址

节内偏移

3. 如果是采用PC相对引用的方式

```
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  }
```

修正量

1. 根据重定位类型进行选择

2. 如果是采用绝对地址的方式

重定位条目
r.offset = 0xf
r.symbol = sum
r.type = R_X86_64_PC32
r.addend = -4

*refpr即要填入00 00 00 00处的值

具体过程

1. 重定位PC相对引用

```
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
```

重定位条目

r.offset = 0xf

r.symbol = sum

r.type = R_X86_64_PC32

r.addend = -4

从节头表获知代码节地址： ADD(s) = ADDR(.text) = 0x4004d0

节头表+符号表获知符号sum地址： ADD(r.symbol) = ADDR(sum) = 0x4004e8

计算过程：

①占位符的起始地址

refaddr = ADDR(s) + r.offset

= 0x4004d0 + 0xf

= 0x4004df

②sum函数的相对地址（sum地址减去当前PC地址）

*refptr = (unsigned)(ADD(r.symbol) + r.addend - refaddr)

= (unsigned)(0x4004e8 – (0x4004df+4))

= (unsigned) (0x5)

具体过程

2. 重定位绝对引用

```
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
```

重定位条目

r.offset = 0xa

r.symbol = array

r.type = R_X86_64_32

r.addend = 0

节头表+符号表获知符号array地址： ADD(r.symbol) = ADDR(array) = 0x601018

计算过程（绝对地址）：

*refptr = (unsigned)(ADD(r.symbol) + r.addend)

= (unsigned)(0x601018 + 0)

= (unsigned) (0x601018)

最后结果

```
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
```

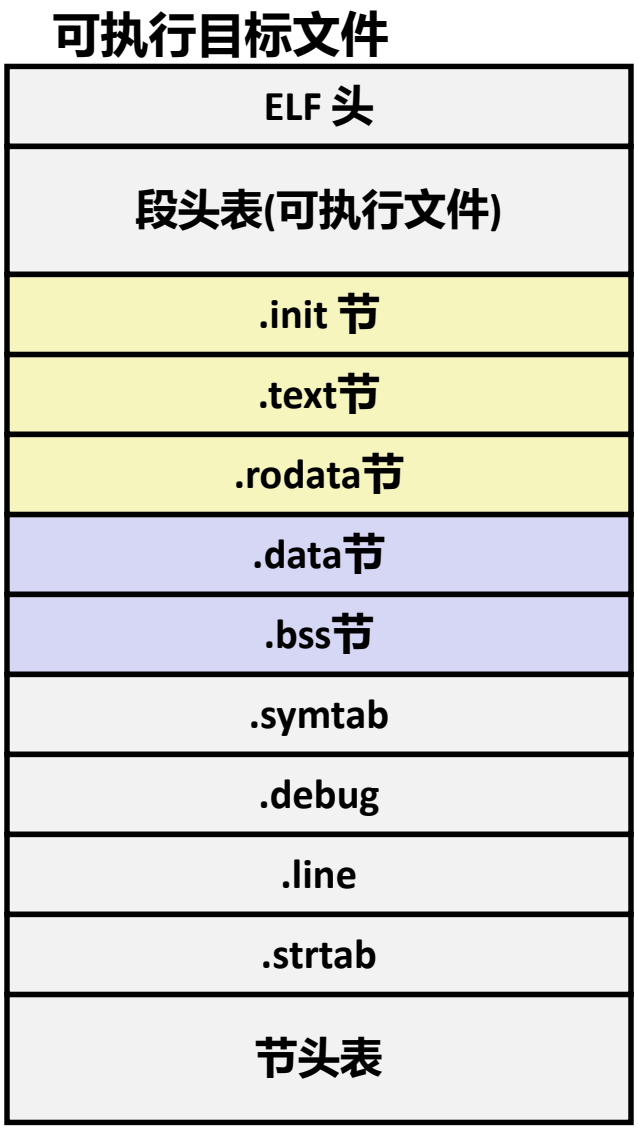
出题，跳转到的地址减下一条指令地址

为什么那两条指令立即数是09和f3?

来源：`objdump -dx prog`

对比可重定位目标文件和可执行目标文件

可执行目标文件

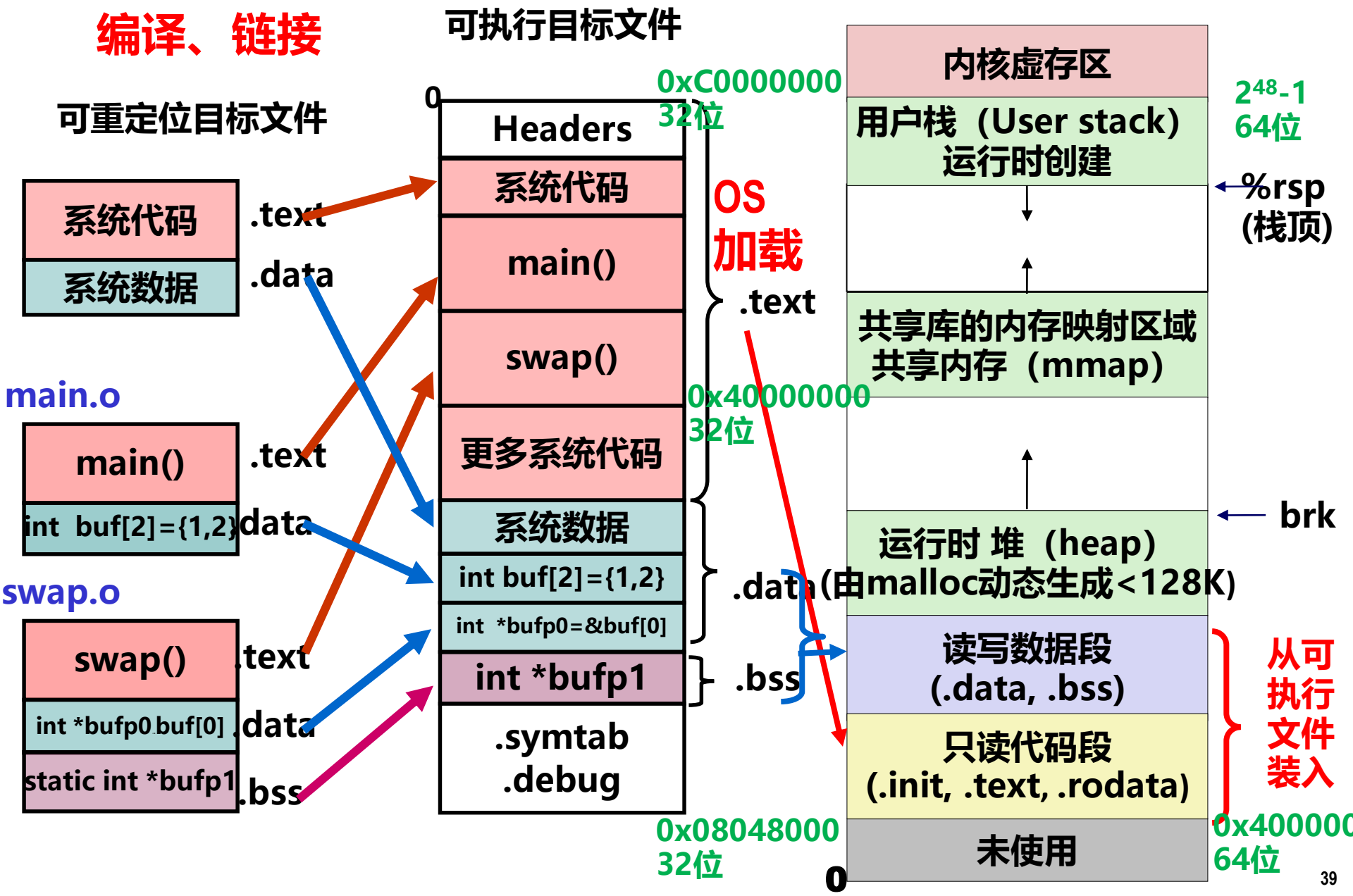


与可重定位文件稍有不同：

- 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，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）

工具：readlf –h 文件名

源程序、目标文件、执行程序、虚拟内存映像



常用的函数打包 .o法

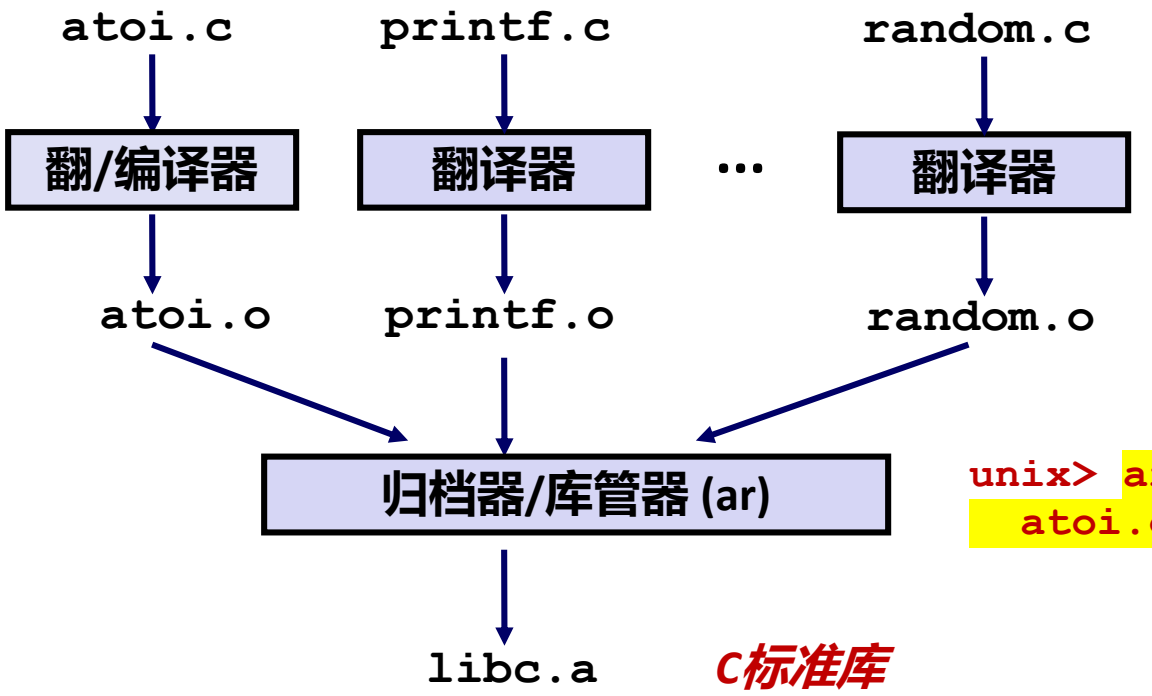
- **如何打包程序员常用的函数?**
  - Math, I/O, 存储管理, 串处理,等等.

- **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 .a法

- **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).
  - 链接器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.
  - 如果存档的一个成员文件.o解析了符号引用，就把它链接入可执行文件

创建静态库




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

- 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（用ar –t libc.a可以查看所有.o文件列表）

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];
}
```

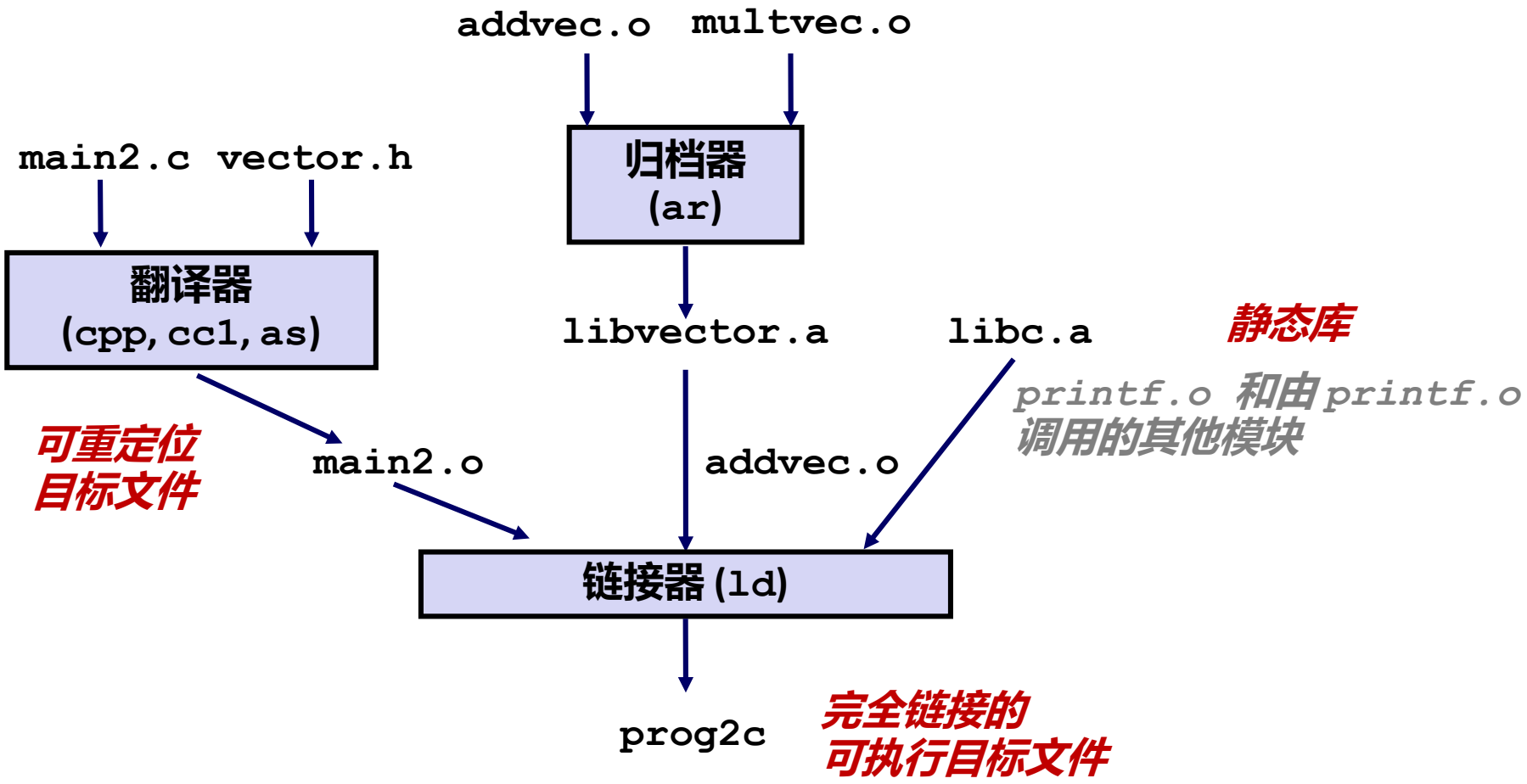
addvec.c

```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

```
gcc –c addvec.c mulvec.c
ar rcs libvector.a addvec.o mulvec.o
gcc –o prog1 main2.c ./libvector.a
```

与静态库链接



当链接器运行时，它判定main2.o引用了addvec.o定义的**addvec**符号，所以复制addvec.o到可执行文件。因为程序不引用任何由**multvec.o**定义的符号，所以链接器就不会复制这个模块到可执行文件。链接器还会复制**libc.a**中的**printf.o**模块，以及许多C运行时系统中的其他模块。

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：**
  - 在使用静态链接的过程中，文件的输入顺序十分重要！
  - 准则：将库放在命令行的末尾（-lmine等价于libmine.a）

```
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'
```

关于gcc的-l参数，-L参数，-l参数，请查看：
https://blog.csdn.net/sinat_31608641/article/details/122513674

习题

- **请给出命令行，使得静态链接器能够解析所有的符号引用**
  - A. p.o -> libx.a
  - B. p.o -> libx.a -> liby.a
  - C. p.o -> libx.a ->liby.a且liby.a->libx.a->p.o

答案

- A. gcc p.o libx.a
- B. gcc p.o libx.a liby.a
- C. gcc p.o libx.a liby.a libx.a

注：要求使用最少数量的目标文件和库参数命令
a->b表示a依赖于b，也就是a中引用了定义在b中的符号

习题

- **p.o -> libx.a ->liby.a且liby.a->libx.a->p.o**
- **答案：gcc p.o libx.a liby.a libx.a**

- 在这次扫描中，链接器维护一个可重定位目标文件的集合（刚开始里面只有p.o），一个未解析的符号（即引用了但尚未定义的符号）集合，以及一个在前面输入文件中已定义的符号集合，p.o中定义的符号都在第三个集合中，所以如果后面命令行中的库依赖p.o就不用在命令行中重复出现p.o，而对于库之间的依赖关系，则必须要在命令行上重复库引用

现代的解决方案

- **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)
  - 在运行的多个进程中代码重复复制，浪费内存空间
  - 系统库的更新或者小错误修复要求每个应用程序显式地重新链接，维护复杂

- **现代的解决方案：共享库** 静态.so,动态.so
  - 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)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
  - 软件的部分更新，更轻量便捷
- **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**
- **共享库也适合跨语言函数调用，比如java调用C生成的共享库**

对比静态链接和动态链接

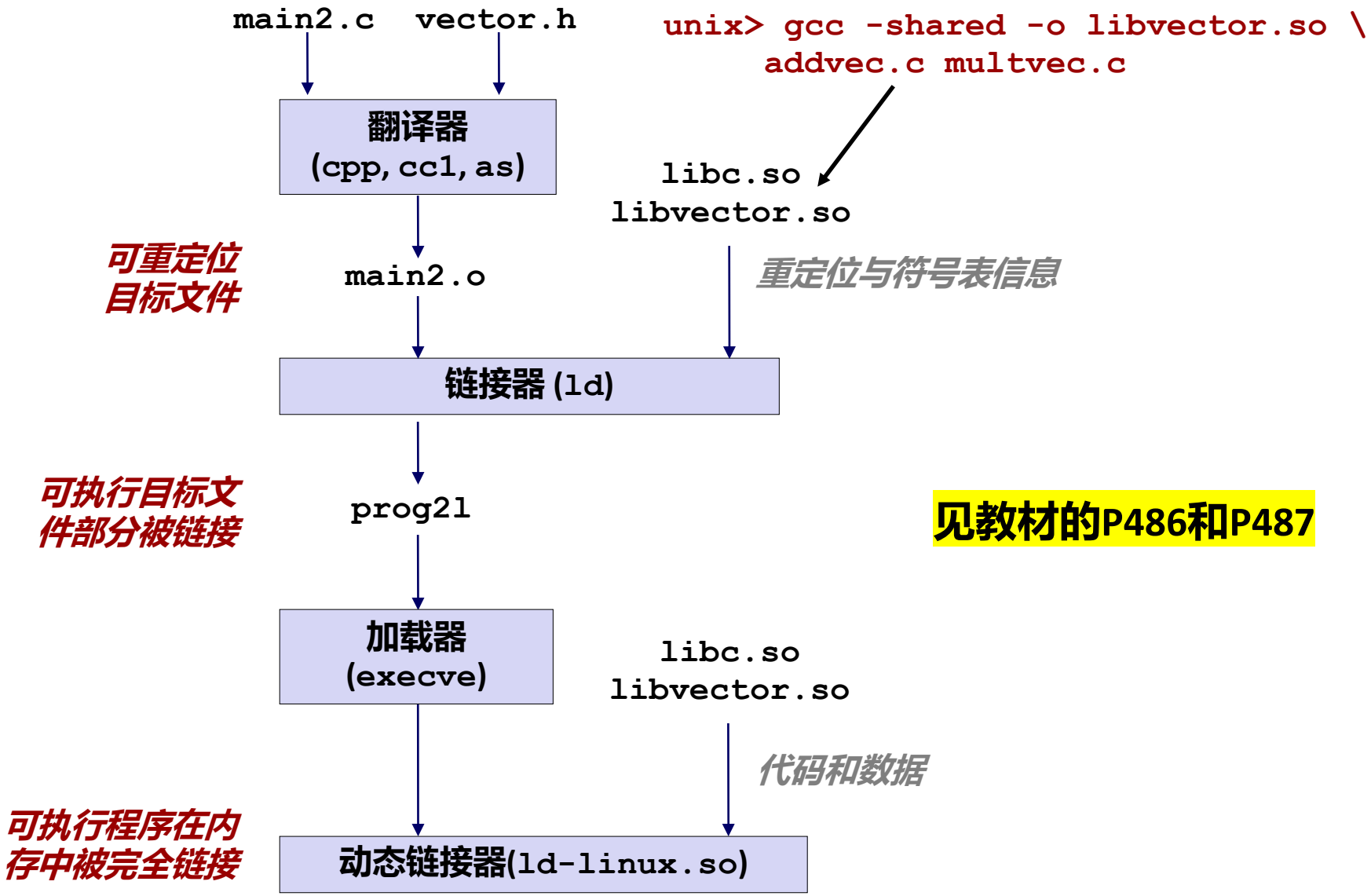
- 静态链接库的生成

```
gcc –c addvec.c mulvec.c
ar rcs libvector.a addvec.o mulvec.o
gcc –o prog1 main.c ./libvector.a
```

- 动态链接库(共享库)的生成

```
gcc -shared –fpic –o libvector.so  addvec.c mulvec.c
gcc –o prog2 main.c ./libvector.so
```

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
见教材的P486和P487
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

```
#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);
    }
```

备注： RTLD_LAZY 标志，该标志指示链接器推迟符号解析直到执行来自库中的代码

dll.c

dlopen函数见https://baike.baidu.com/item/dlopen/1967576?fr=aladdin

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:**
  - 编译时(当程序被链接时) /GCC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

- 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：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)来解决

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---各种监控与分析软件、生成日志等
  - 计算函数调用的次数
  - 描述个性化的调用地点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—自己记录统计分析

```
#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)=0x1edc010
free(0x1edc010)
linux>
```

关于make，请查看:https://blog.csdn.net/FRS2023/article/details/120839849
Makefile里面有一个编译标签(目标)叫intc:，其后跟着目标文件及相关编译指令，上面命令会根据make命令后面的参数选择并编译Makefile里的带intc标签下对应的编译命令，虽然标签名和最终可执行目标文件是重名，都叫intc。

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

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

习题

- **1.链接过程中，赋初值的局部变量名，正确的是**

  A.强符号　　B.弱符号　　C.若是静态的则为强符号

  D.以上都错

- **2.关于动态库的描述错误的是**
  - A.可在加载时链接，即当可执行文件首次加载和运行时进行动态链接。
  - B.更新动态库，即便接口不变，也需要将使用该库的程序重新编译。
  - C.可在运行时链接，即在程序开始运行后通过程序指令进行动态链接。
  - D.即便有多个正在运行的程序使用同一动态库，系统也仅在内存中载入一份动态库。

1.D　　2.B

Hope you enjoyed the CSAPP course!