第四章 程序的链接

目标文件格式

符号解析与重定位

共享库与动态链接

可执行文件的链接生成

- 主要教学目标
  - 使学生了解链接器是如何工作的，从而能够养成良好的程序设计习惯，并增加程序调试能力。
  - 通过了解可执行文件的存储器映像来进一步深入理解进程的虚拟地址空间的概念。
- 包括以下内容
  - 链接和静态链接概念
  - 三种目标文件格式
  - 符号及符号表、符号解析
  - 使用静态库链接
  - 重定位信息及重定位过程
  - 可执行文件的存储器映像
  - 可执行文件的加载
  - 共享（动态）库链接

程序的链接

- 分以下三个部分介绍
  - 第一讲：目标文件格式
    - 程序的链接概述、链接的意义与过程
    - ELF目标文件、重定位目标文件格式、可执行目标文件格式
  - 第二讲：符号解析与重定位
    - 符号和符号表、符号解析
    - 与静态库的链接
    - 重定位信息、重定位过程
    - 可执行文件的加载
  - 第三讲：动态链接
    - 动态链接的特性、程序加载时的动态链接、程序运行时的动态链接、动态链接举例

一个典型程序的转换处理过程

经典的“ hello.c ”C-源程序

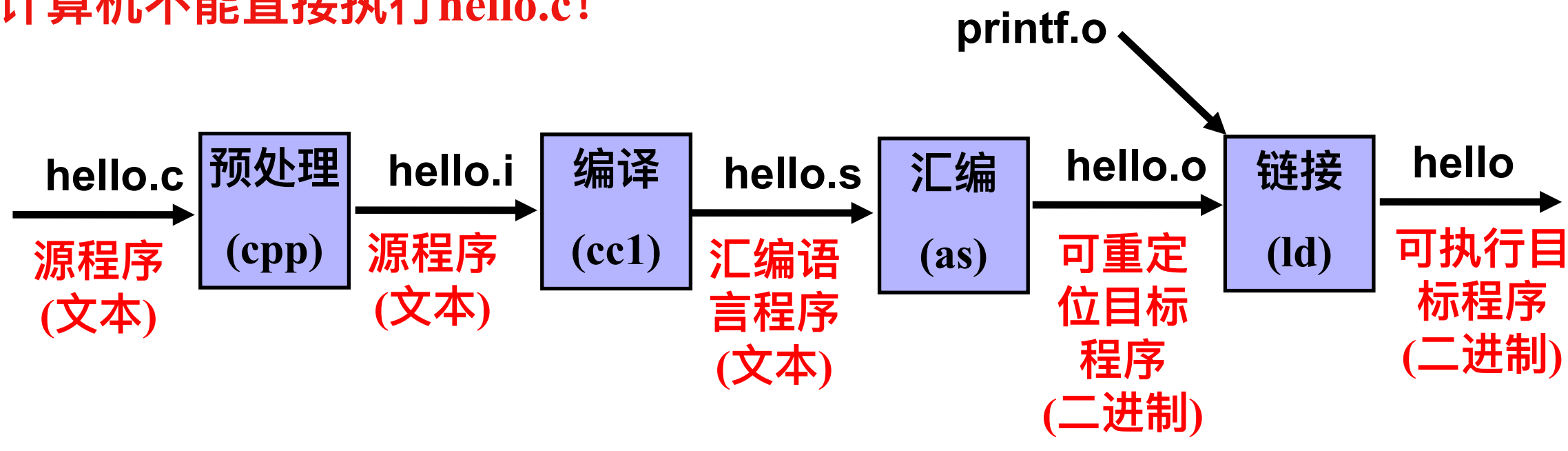
```
#include <stdio.h>

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

hello.c的ASCII文本表示

```
# i n c l u d e <sp> < s t d i o .
35 105 110 99 108 117 100 101 32 60 115 116 100 105 111 46
h > \n \n i n t <sp> m a i n ( ) \n {
104 62 10 10 105 110 116 32 109 97 105 110 40 41 10 123
\n <sp> <sp> <sp> <sp> p r i n t f ( " h e l
10 32 32 32 32 112 114 105 110 116 102 40 34 104 101 108
l o , <sp> w o r l d \ n " ) ; \ n }
108 111 44 32 119 111 114 108 100 92 110 34 41 59 10 125
```

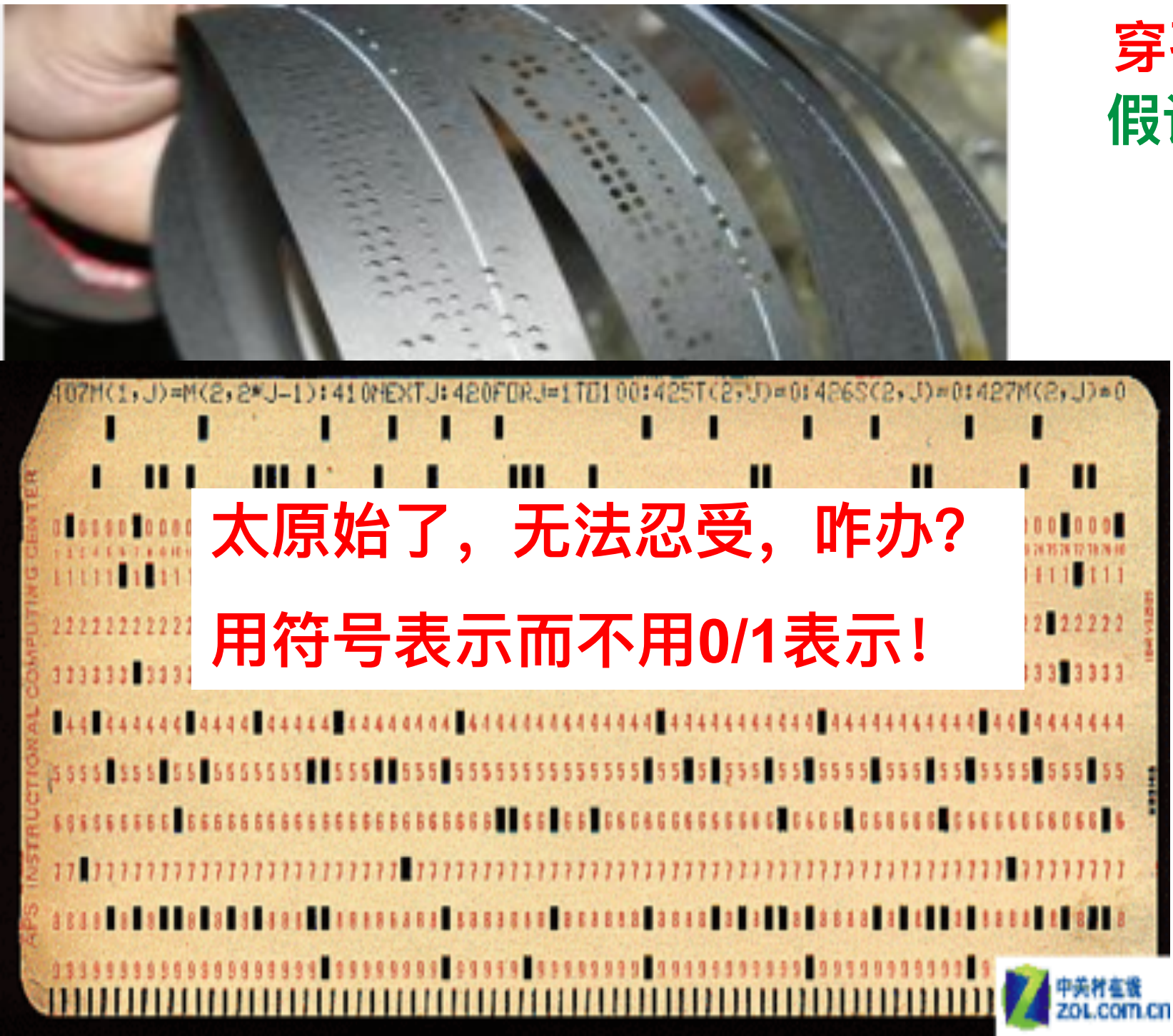
功能：输出“hello,world”

计算机不能直接执行hello.c！



链接器的由来

- 原始的链接概念早在高级编程语言出现之前就已存在
- 最早程序员**用机器语言编写程序**，并记录在纸带或卡片上



穿孔表示0，未穿孔为1

假设：0010-jmp

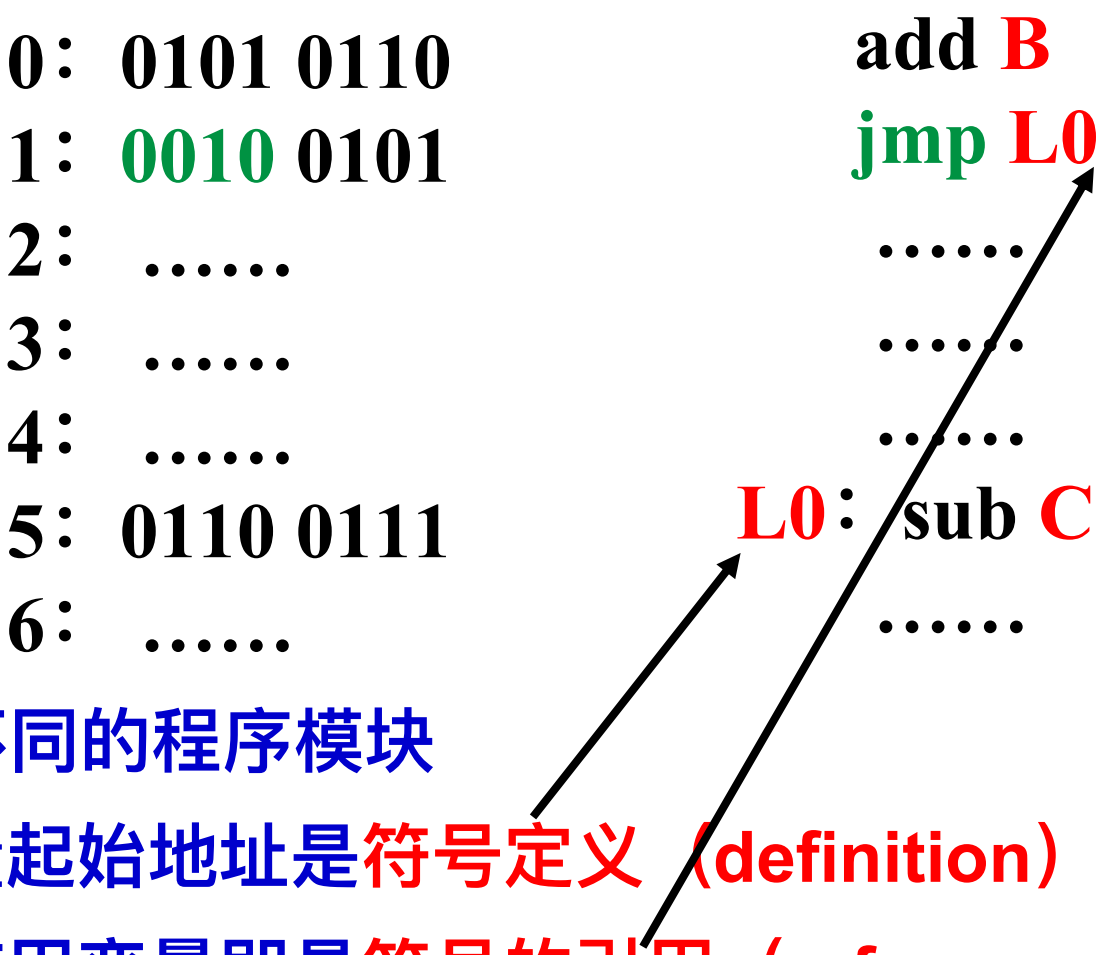
0：0101 0110
1：0010 0101
2：……
3：……
4：……
5：0110 0111
6：……

若在第5条指令前加入指令，则程序员需重新计算jmp指令的目标地址（重定位），然后重新打孔。

链接器的由来

- 用符号表示跳转位置和变量位置，是否简化了问题?
- 汇编语言出现
  - 用助记符表示操作码
  - 用符号表示位置
  - 用助记符表示寄存器
  - .....
- 更高级编程语言出现
  - 程序越来越复杂，需多人开发不同的程序模块
  - 子程序（函数）起始地址和变量起始地址是符号定义（definition）
  - 调用子程序（函数或过程）和使用变量即是符号的引用（reference）
  - 一个模块定义的符号可以被另一个模块引用
  - 最终须链接（即合并），合并时须在符号引用处填入定义处的地址

如上例，先确定L0的地址，再在jmp指令中填入L0的地址



使用链接的好处

链接带来的好处1：模块化

（1）一个程序可以分成很多源程序文件

（2）可构建公共函数库，如数学库，标准C库等

链接带来的好处2：效率高

（1）时间上，可分开编译

只需重新编译被修改的源程序文件，然后重新链接

（2）空间上，无需包含共享库所有代码

源文件中无需包含共享库函数的源码，只要直接调用即可

可执行文件和运行时的内存中只需包含所调用函数的代码

而不需要包含整个共享库

一个C语言程序举例

main.c

```
int buf[2] = {1, 2};
void swap();

int main()
{
  swap();
  return 0;
}
```

swap.c

```
extern int buf[];

int *bufp0 = &buf[0];
static int *bufp1;

void swap()
{
  int temp;
  bufp1 = &buf[1];
  temp = *bufp0;
  *bufp0 = *bufp1;
  *bufp1 = temp;
}
```

你能说出哪些是符号定义？哪些是符号的引用？

局部变量temp分配在栈中，不会在过程外被引用，因此不是符号定义

可执行文件的生成

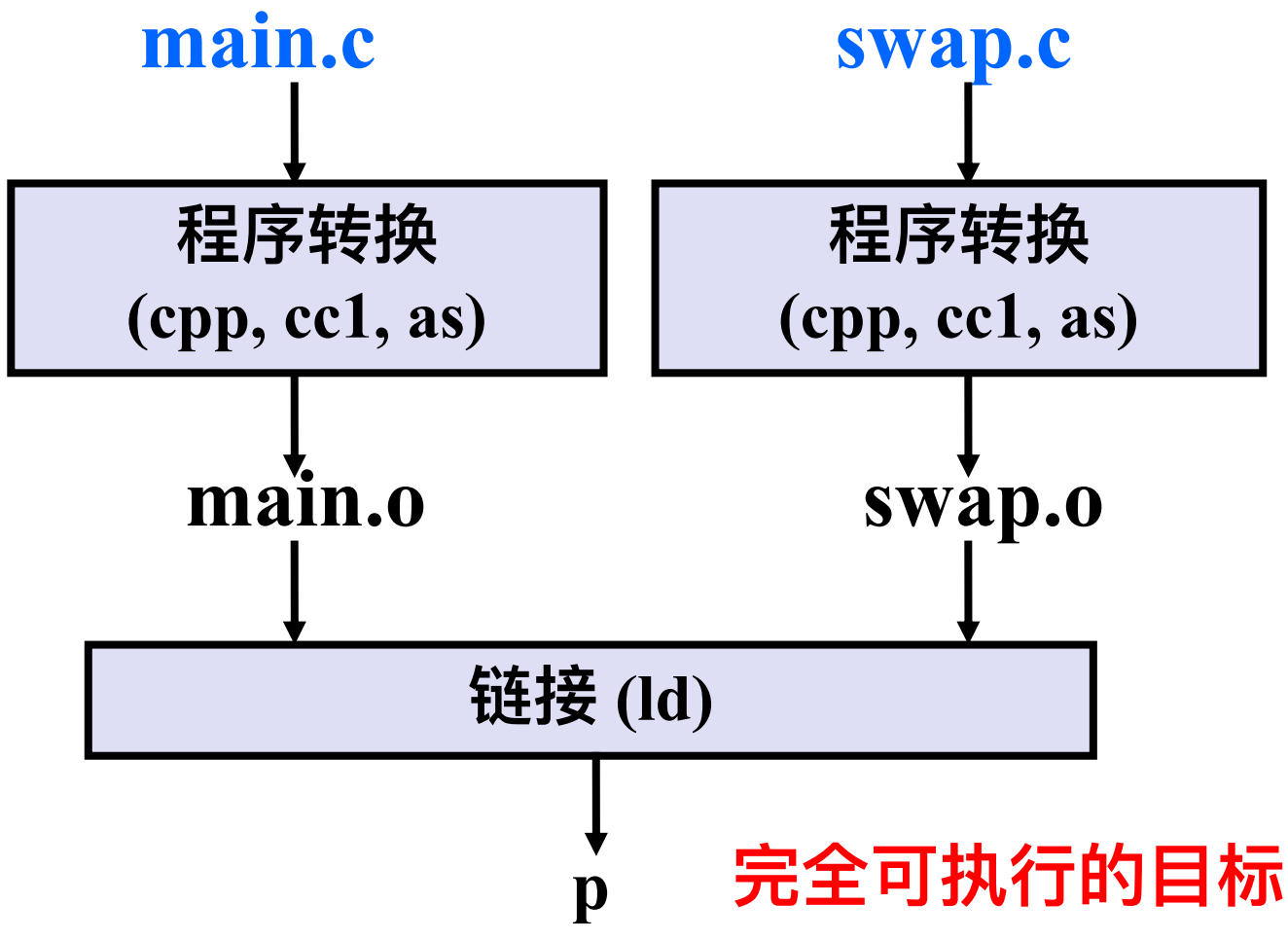
- 使用GCC编译器编译并链接生成可执行程序P:
  - $ gcc -O2 -g -o p main.c swap.c
  - $ ./p

-O2：2级优化

-g：生成调试信息

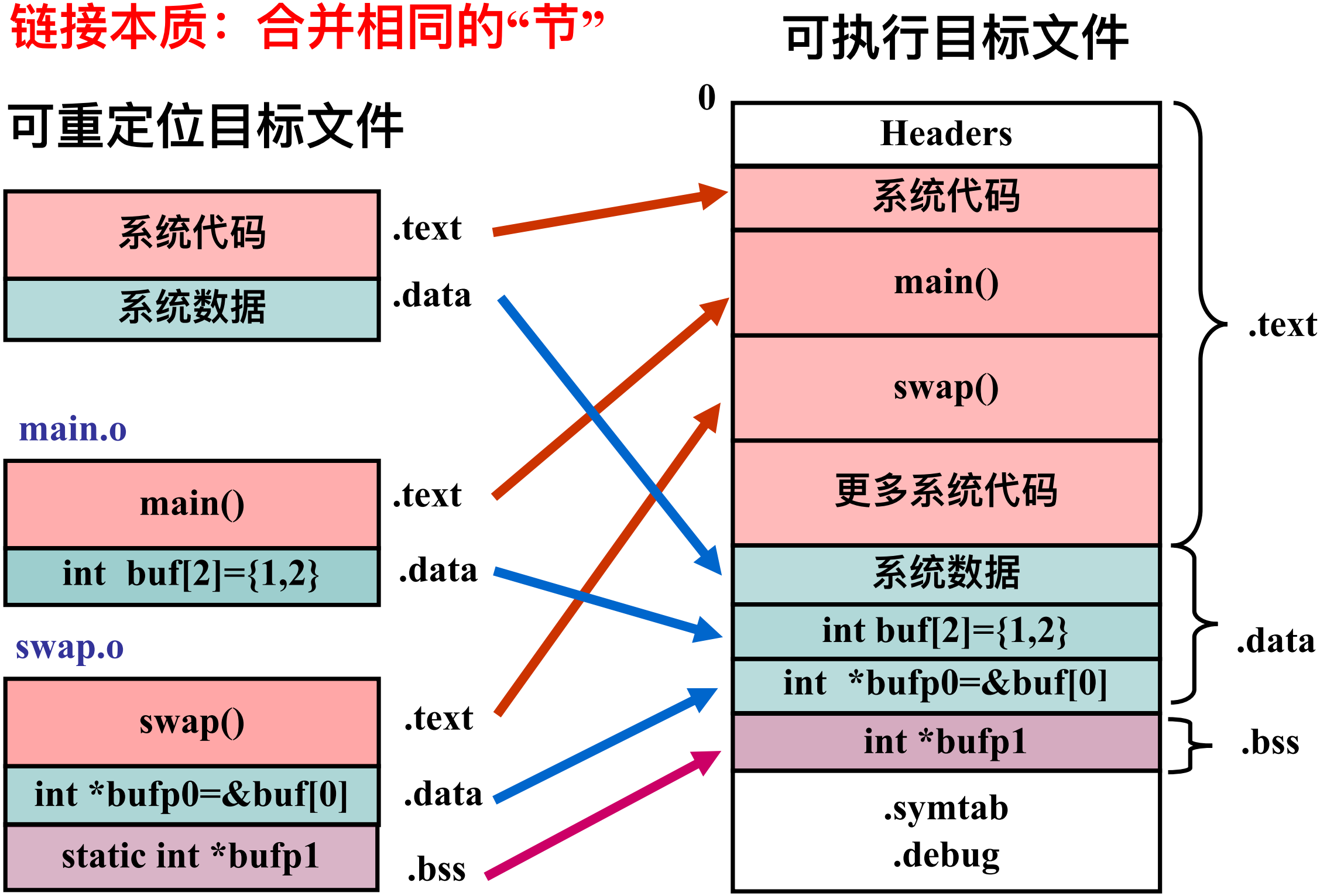
-o：目标文件名

GCC编译器的静态链接过程



链接过程的本质

链接本质：合并相同的“节”



目标文件

```
/* main.c */
int add(int, int);
int main( )
{
   return add(20, 13);
}
```

```
/* test.c */
int add(int i, int j)
{
    int x = i + j;
    return x;
}
```

objdump -d test.o

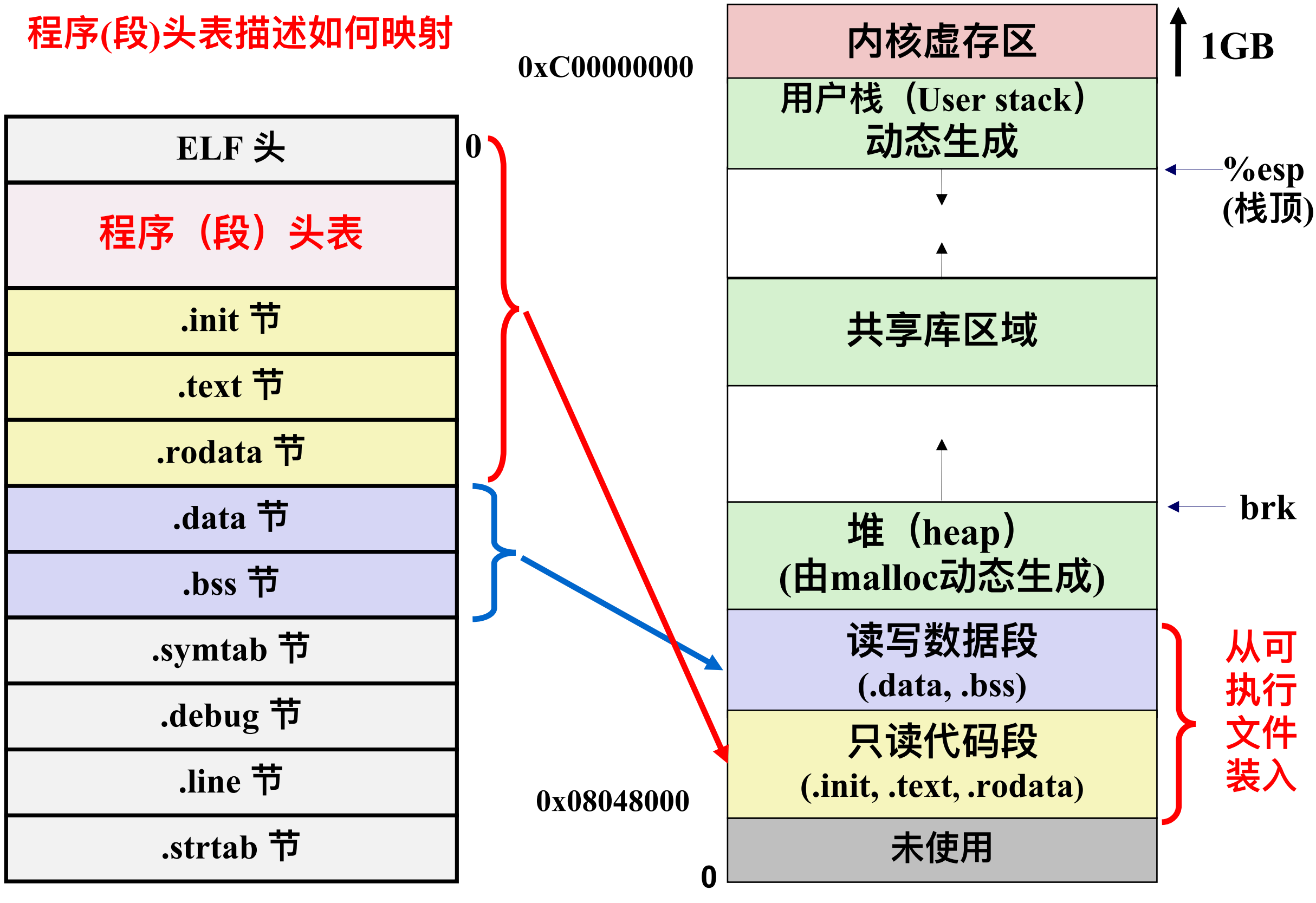
```
00000000 <add>:
   0:   55              push  %ebp
   1:   89 e5           mov   %esp, %ebp
   3:   83 ec 10        sub   $0x10, %esp
   6:   8b 45 0c        mov   0xc(%ebp), %eax
   9:   8b 55 08        mov   0x8(%ebp), %edx
   c:   8d 04 02        lea   (%edx,%eax,1), %eax
   f:   89 45 fc        mov   %eax, -0x4(%ebp)
  12:   8b 45 fc        mov   -0x4(%ebp), %eax
  15:   c9              leave
  16:   c3              ret
```

objdump -d test

```
080483d4 <add>:
 80483d4:   55              push  %ebp
 80483d5:   89 e5           mov   %esp, %ebp
 80483d7:   83 ec 10        sub   $0x10, %esp
 80483da:   8b 45 0c        mov   0xc(%ebp), %eax
 80483dd:   8b 55 08        mov   0x8(%ebp), %edx
 80483e0:   8d 04 02        lea   (%edx,%eax,1), %eax
 80483e3:   89 45 fc        mov   %eax, -0x4(%ebp)
 80483e6:   8b 45 fc        mov   -0x4(%ebp), %eax
 80483e9:   c9              leave
 80483ea:   c3              ret
```

可执行文件的存储器映像

程序(段)头表描述如何映射



链接操作的步骤

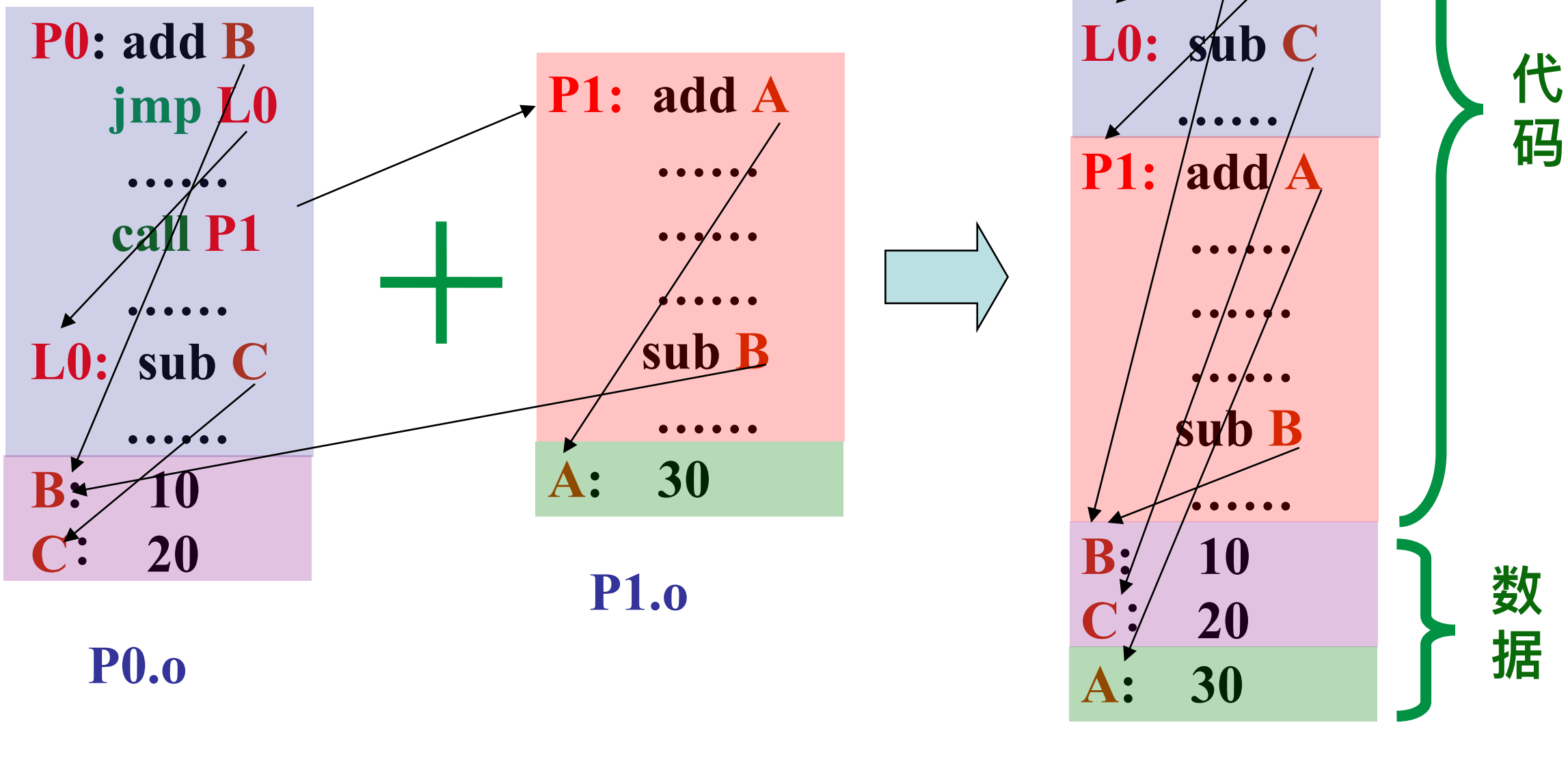
1）确定标号引用关系（符号解析）

2）合并相关.o文件

3）确定每个标号的地址

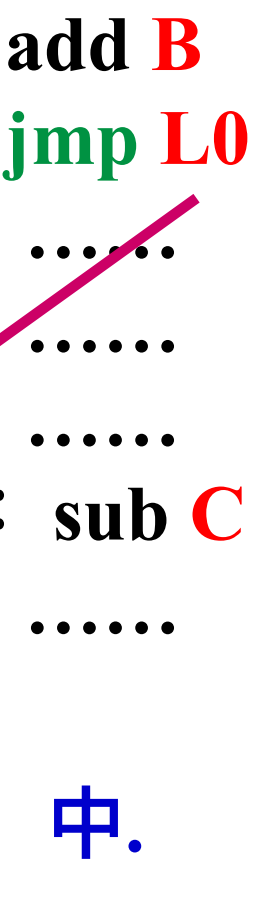
4）在指令中填入新地址

} 重定位



链接操作的步骤

- Step 1. 符号解析（Symbol resolution）
  - 程序中有定义和引用的符号 (包括变量和函数等)
    - 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 */
  - 编译器将定义的符号存放在一个符号表（ symbol table）中.
    - 符号表是一个结构数组
    - 每个表项包含符号名、长度和位置等信息
  - 链接器将每个符号的引用都与一个确定的符号定义建立关联
- Step 2. 重定位
  - 将多个代码段与数据段分别合并为一个单独的代码段和数据段
  - 计算每个定义的符号在虚拟地址空间中的绝对地址
  - 将可执行文件中符号引用处的地址修改为重定位后的地址信息



三类目标文件

- 可重定位目标文件 (.o)
  - 其代码和数据可和其他可重定位文件合并为可执行文件
    - 每个.o 文件由对应的.c文件生成
    - 每个.o文件代码和数据地址都从0开始
- 可执行目标文件 (默认为a.out)
  - 包含的代码和数据可以被直接复制到内存并被执行
  - 代码和数据地址为虚拟地址空间中的地址
- 共享的目标文件 (.so)
  - 特殊的可重定位目标文件，能在装入或运行时被装入到内存并自动被链接，称为共享库文件
  - Windows 中称其为 *Dynamic Link Libraries* (DLLs)

目标文件的格式

- **目标代码（Object Code）**指编译器和汇编器处理源代码后所生成的机器语言目标代码
- **目标文件（Object File）**指包含目标代码的文件
- 最早的目标文件格式是自有格式，非标准的
- 标准的几种目标文件格式
  - DOS操作系统（最简单）：COM格式，文件中仅包含代码和数据，且被加载到固定位置
  - System V UNIX早期版本：COFF格式，文件中不仅包含代码和数据，还包含重定位信息、调试信息、符号表等其他信息，由一组严格定义的数据结构序列组成
  - Windows： PE格式（COFF的变种），称为可移植可执行（Portable Executable，简称PE）
  - Linux等类UNIX：ELF格式（COFF的变种），称为可执行可链接（Executable and Linkable Format，简称ELF）

Executable and Linkable Format (ELF)

- 两种视图
  - 链接视图（被链接）：Relocatable object files
  - 执行视图（被执行）：Executable object files

链接视图

节（section）是ELF 文件中具有相同特征的最小可处理单位

.text节: 代码

.data节: 数据

.rodata: 只读数据

.bss: 未初始化数据

执行视图

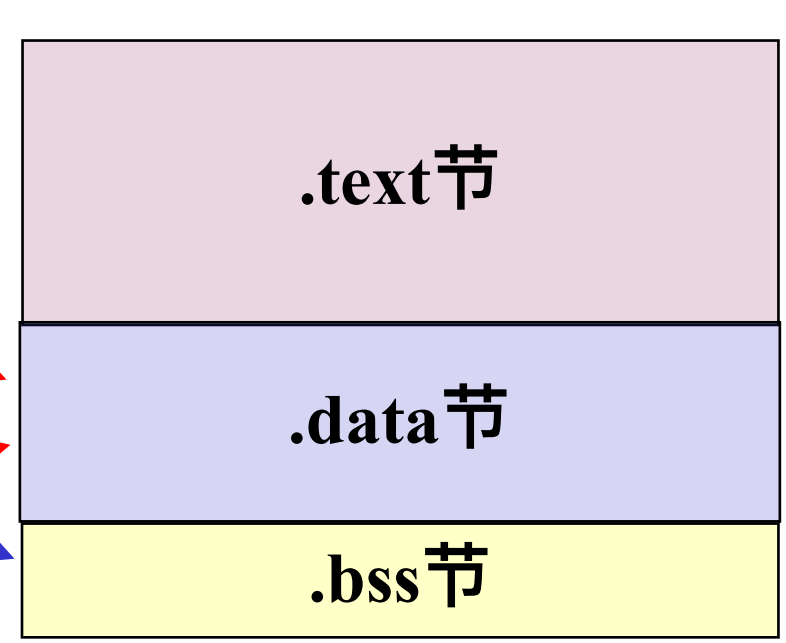
由不同的段（segment）组成，描述节如何映射到存储段中，可多个节映射到同一段，如：可合并.data节和.bss节,并映射到一个可读可写数据段中

链接视图—可重定位目标文件

- 可被链接（合并）生成可执行文件或共享目标文件
- 静态链接库文件由若干个可重定位目标文件组成
- 包含代码、数据（已初始化.data和未初始化.bss）
- 包含重定位信息（指出哪些符号引用处需要重定位）
- 文件扩展名为.o（相当于Windows中的 .obj文件）

```
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);
}
```

ELF的链接视图



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）

可重定位目标文件格式

ELF 头

- ✓ 占16字节，包括字长、字节序（大端/小端）、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

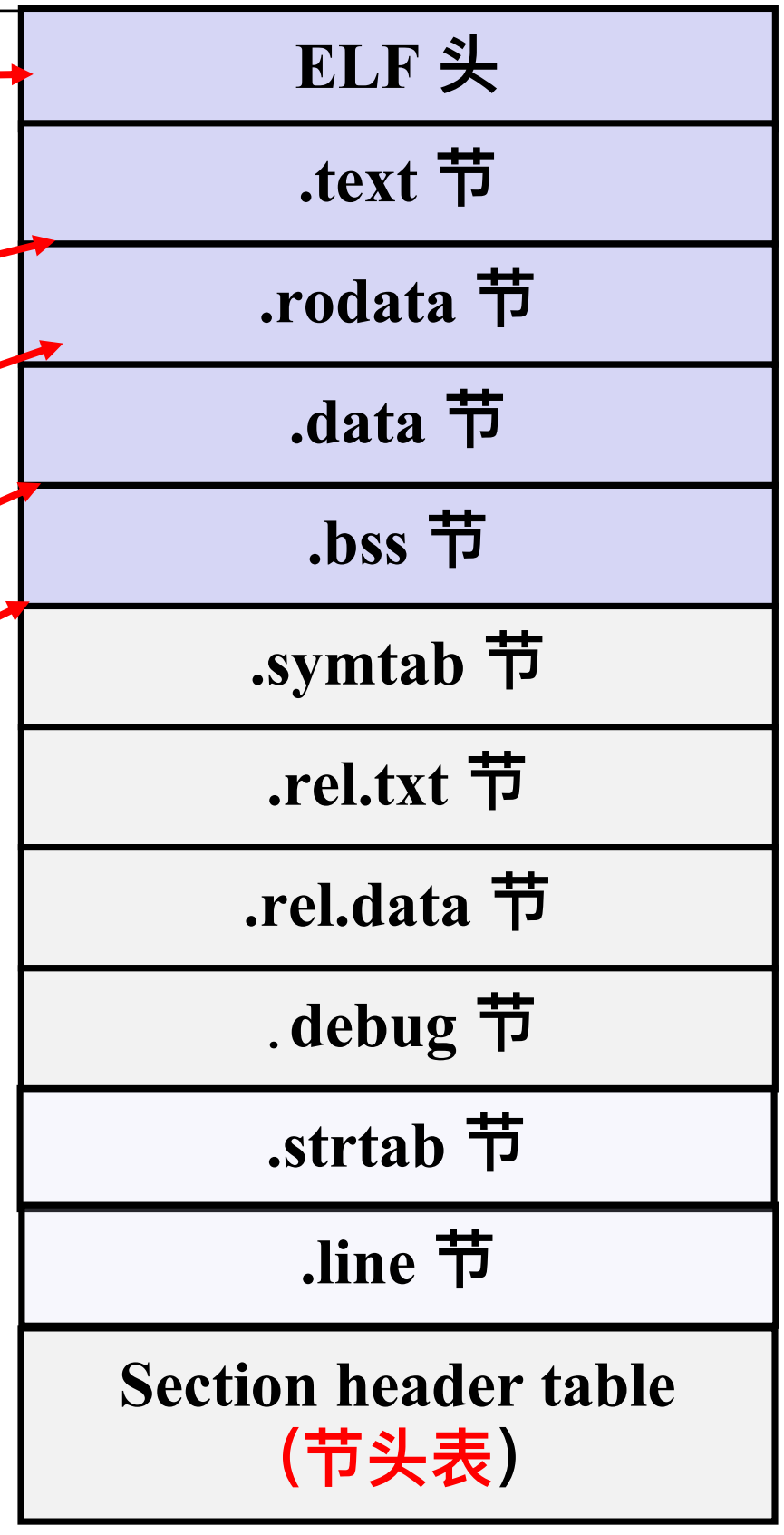
- ✓ 只读数据，如 printf 格式串、switch 跳转表等

.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



switch-case语句举例

```
int sw_test(int a, int b, int c)
{
   int result;
   switch(a) {
   case 15:
      c=b&0x0f;
   case 10:
      result=c+50;
      break;
   case 12:
   case 17:
      result=b+50;
      break;
   case 14:
      result=b
      break;
   default:
      result=a;
   }
   return result;
}
```

```
  movl  8(%ebp), %eax
  subl  $10, %eax
  cmpl  $7, %eax
  ja    .L5
  jmp   *.L8( , %eax, 4)
.L1:
  movl  12(%ebp), %eax
  andl  $15, %eax
  movl  %eax, 16(%ebp)
.L2:
  movl  16(%ebp), %eax
  addl  $50, %eax
  jmp  .L7
.L3:
  movl  12(%ebp), %eax
  addl  $50, %eax
  jmp   .L7
.L4:
  movl  12(%ebp), %eax
  jmp  .L7
.L5:
  addl  $10, %eax
.L7:
```

R[eax]=a-10=i

if (a-10)>7 转 L5

转.L8+4*i 处的地址

跳转表在目标文件的只读节中，按4字节边界对齐。

```
   .section  .rodata
   .align 4
.L8                    a=
   .long    .L2    10
   .long    .L5    11
   .long    .L3    12
   .long    .L5    13
   .long    .L4    14
   .long    .L1    15
   .long    .L5    16
   .long    .L3    17
```

ELF头（ELF Header）

- ELF头位于ELF文件开始，包含文件结构说明信息。分32位系统对应结构和64位系统对应结构（32位版本、64位版本）
- 以下是32位系统对应的数据结构

```
#define EI_NIDENT     16
typedef struct {
        unsigned char     e_ident[EI_NIDENT];
        Elf32_Half        e_type;
        Elf32_Half        e_machine;
        Elf32_Word        e_version;
        Elf32_Addr        e_entry;
        Elf32_Off         e_phoff;
        Elf32_Off         e_shoff;
        Elf32_Word        e_flags;
        Elf32_Half        e_ehsize;
        Elf32_Half        e_phentsize;
        Elf32_Half        e_phnum;
        Elf32_Half        e_shentsize;
        Elf32_Half        e_shnum;
        Elf32_Half        e_shstrndx;
} Elf32_Ehdr;
```

定义了ELF魔数、版本、小端/大端、操作系统平台、目标文件的类型、机器结构类型、程序执行的入口地址、程序头表（段头表）的起始位置和长度、节头表的起始位置和长度等

魔数：文件开头几个字节通常用来确定文件的类型或格式

a.out的魔数：01H 07H

PE格式魔数：4DH 5AH

加载或读取文件时，可用魔数确认文件类型是否正确

ELF头信息举例

```
$ readelf -h main.o
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              REL (Relocatable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x0
  Start of program headers:          0 (bytes into file)
  Start of section headers:          516 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           0 (bytes)
  Number of program headers:         0
  Size of section headers:           40 (bytes)
  Number of section headers:         15
  Section header string table index: 12
```

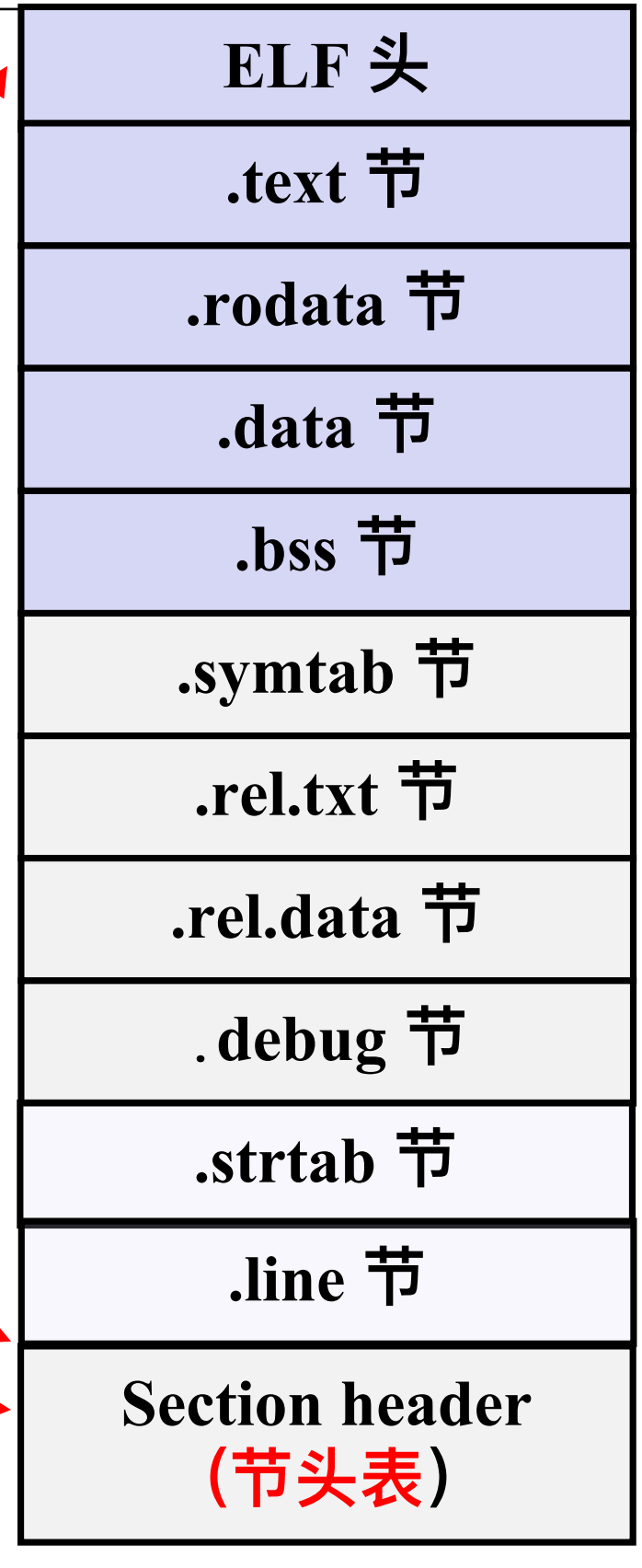
可重定位目标文件的ELF

ELF文件的魔数

没有程序头表

15x40B

.strtab在节头表中的索引



节头表（Section Header Table）

- 除ELF头之外，节头表是ELF可重定位目标文件中最重要的部分内容
- 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
- 以下是32位系统对应的数据结构（每个表项占40B）

```
typedef struct {
    Elf32_Word      sh_name;    节名字符串在.strtab中的偏移
    Elf32_Word      sh_type;    节类型：无效/代码或数据/符号/字符串/...
    Elf32_Word      sh_flags;   节标志：该节在虚拟空间中的访问属性
    Elf32_Addr      sh_addr;    虚拟地址：若可被加载，则对应虚拟地址
    Elf32_Off       sh_offset;  在文件中的偏移地址，对.bss节而言则无意义
    Elf32_Word      sh_size;    节在文件中所占的长度
    Elf32_Word      sh_link;    sh_link和sh_info用于与链接相关的节（如
    Elf32_Word      sh_info;    .rel.text节、.rel.data节、.symtab节等）
    Elf32_Word      sh_addralign;   节的对齐要求
    Elf32_Word      sh_entsize; 节中每个表项的长度，0表示无固定长度表项
} Elf32_Shdr;
```

节头表信息举例

$ readelf -S test.o

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

Section Headers:

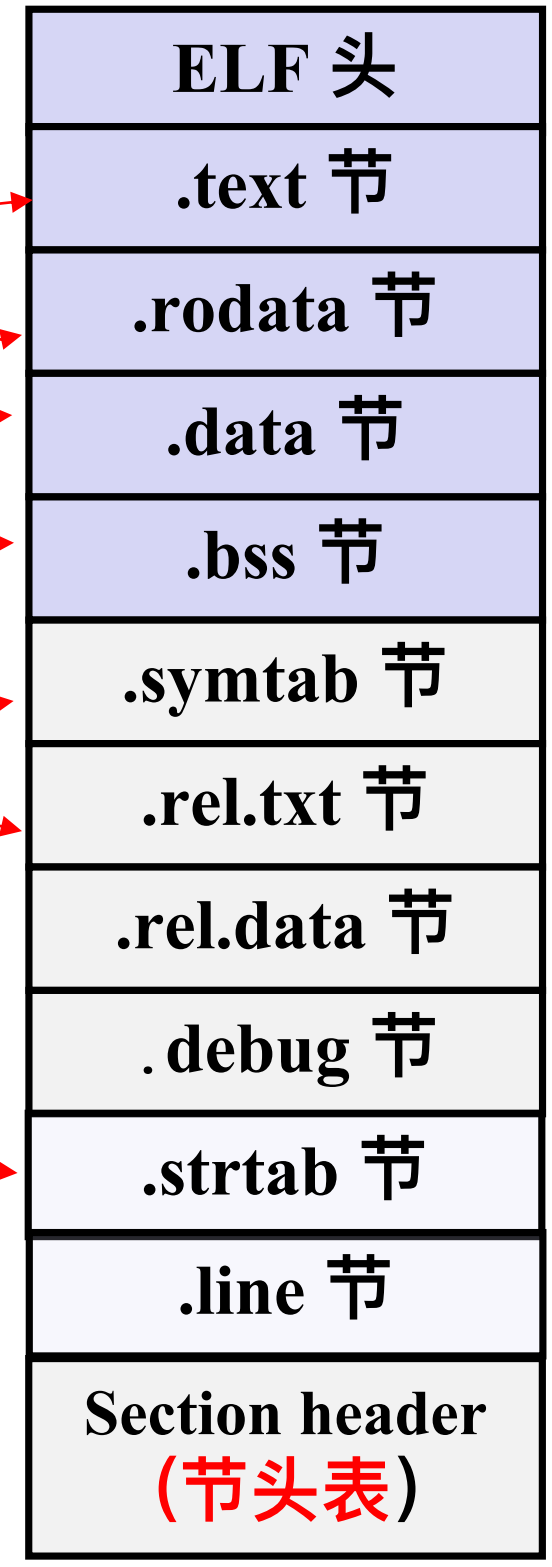
[Nr] Name	Type	Addr	Off	Size	ES	Flg	Lk	Inf	Al
[0]	NULL	00000000	000000	000000	00		0	0	0
[1] .text	PROGBITS	00000000	000034	00005b	00	AX	0	0	4
[2] .rel.text	REL	00000000	000498	000028	08		9	1	4
[3] .data	PROGBITS	00000000	000090	00000c	00	WA	0	0	4
[4] .bss	NOBITS	00000000	00009c	00000c	00	WA	0	0	4
[5] .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

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)

.....

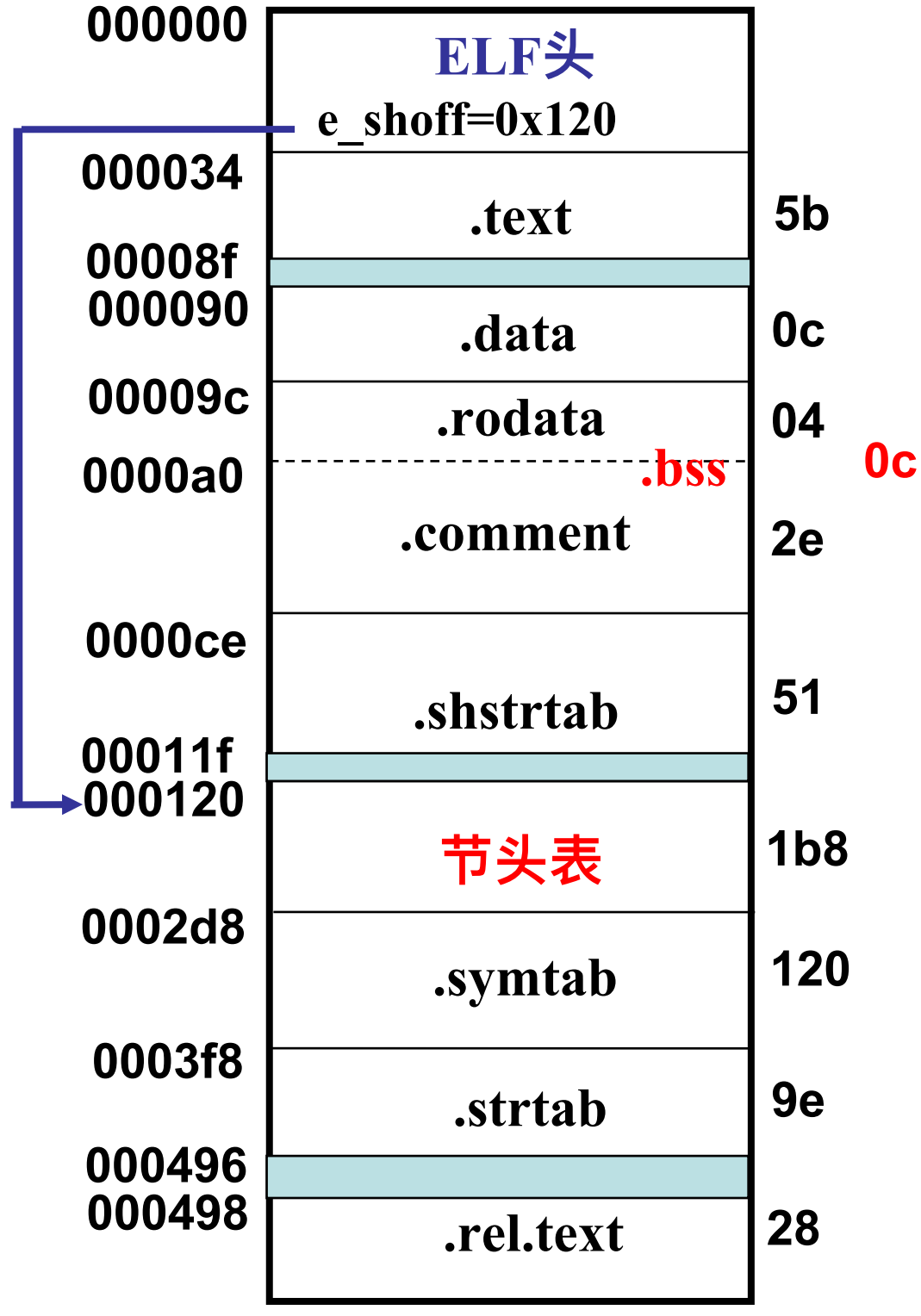
有4个节将会分配（A）存储空间

.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

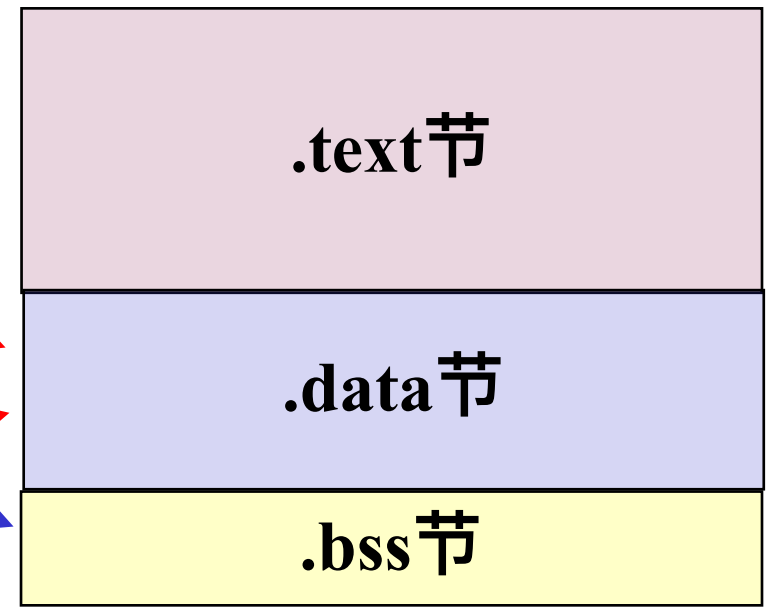


执行视图—可执行目标文件

- 包含代码、数据（已初始化.data和未初始化.bss）
- 定义的所有变量和函数已有确定地址（虚拟地址空间中的地址）
- 符号引用处已被重定位，以指向所引用的定义符号
- 没有文件扩展名或默认为a.out（相当于Windows中的 .exe文件）
- 可被CPU直接执行，指令地址和指令给出的操作数地址都是虚拟地址

```
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);
}
```

ELF的执行视图

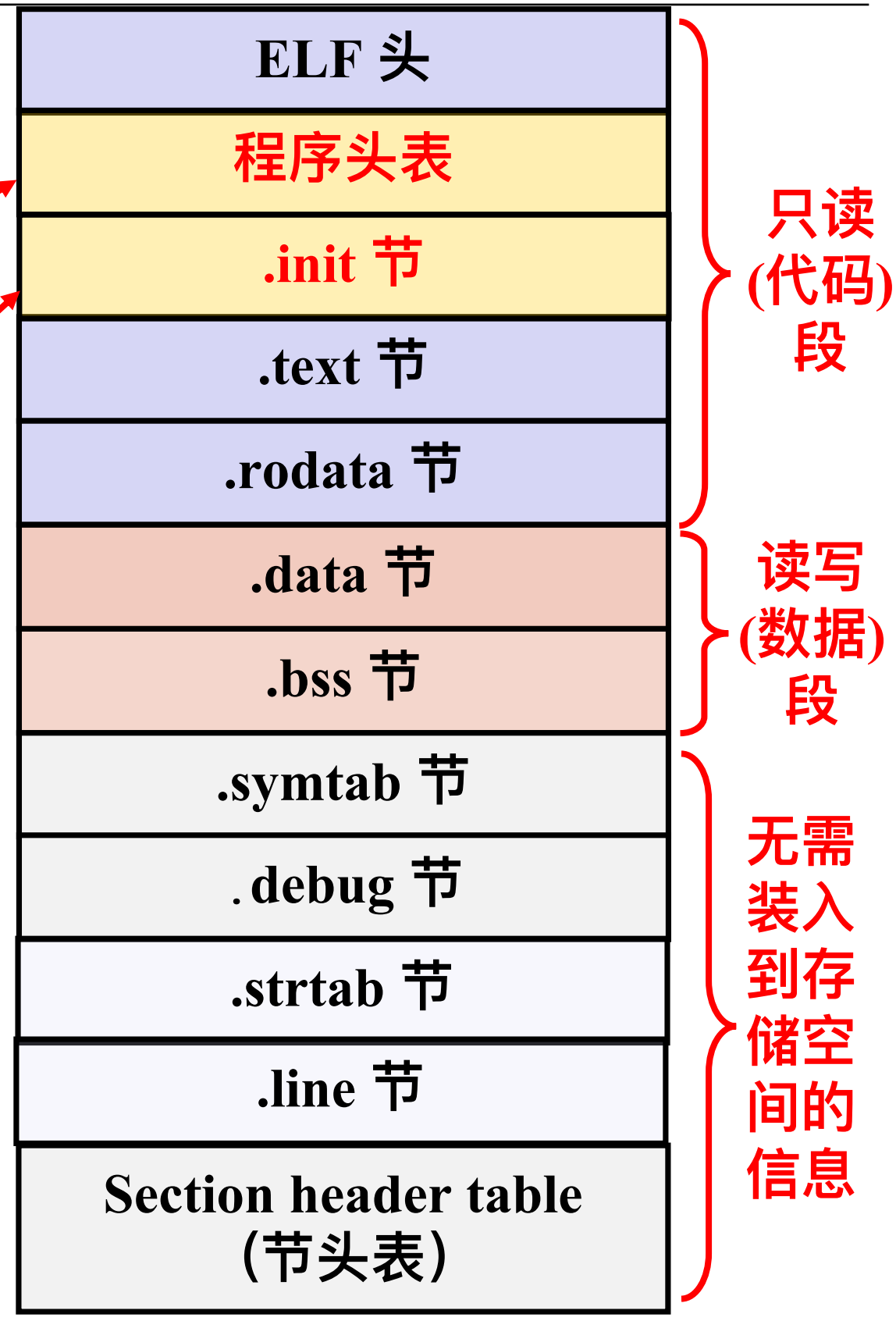


程序头表用来说明段信息，也称段头表

为了能执行，还需将具相同访问属性的节合并成段（Segment），并说明每个段的属性，如：在可执行文件中的位移、大小、在虚拟空间中的位置、对齐方式、访问属性等

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）



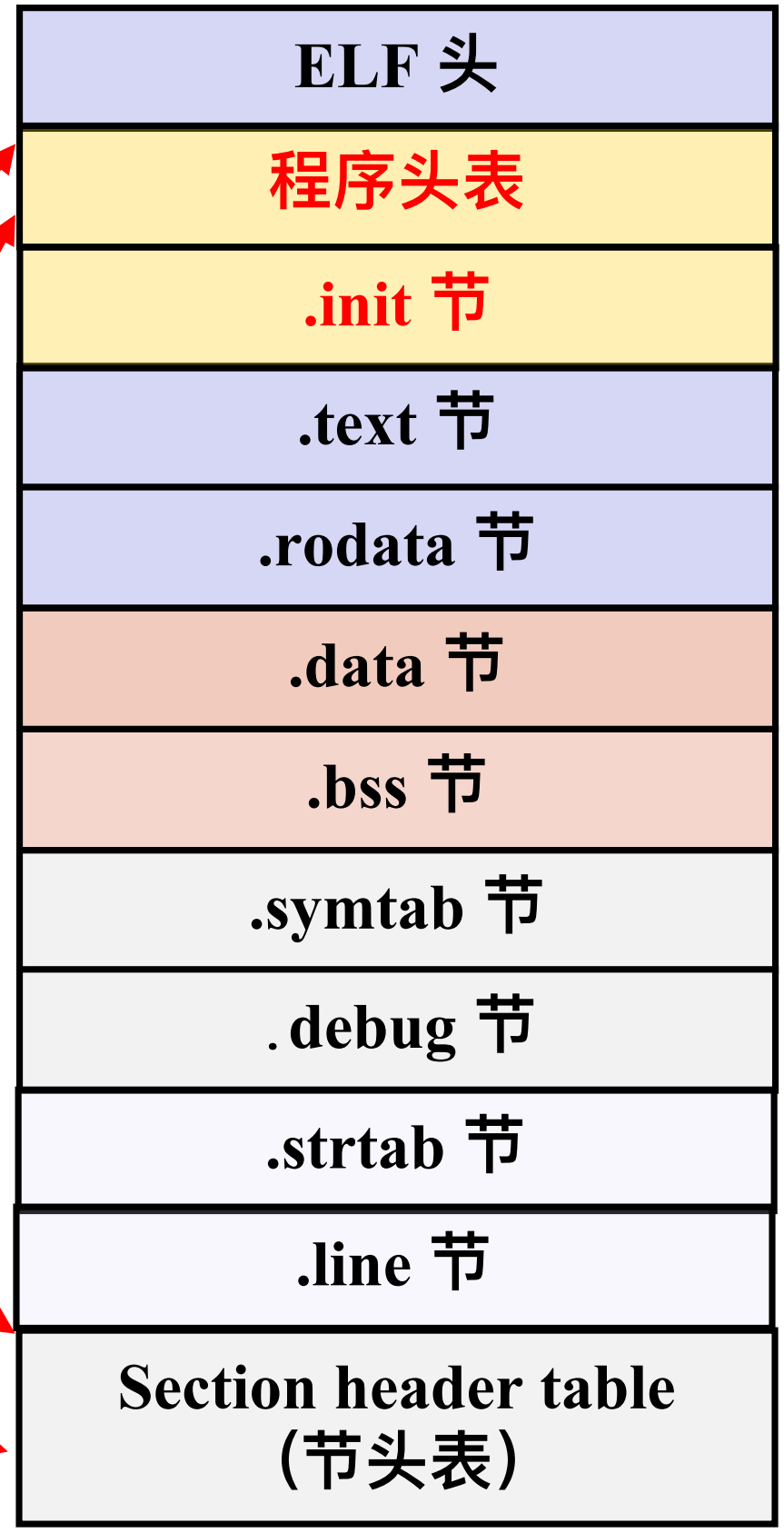
ELF头信息举例

$ readelf -h main　　可执行目标文件的ELF头

```
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:    ELF32
  Data:      2's complement, little endian
  Version:  1 (current)
  OS/ABI:    UNIX - System V
  ABI Version:     0
  Type:    EXEC (Executable file)
  Machine:   Intel 80386
  Version:    0x1
  Entry point address:    x8048580
  Start of program headers:  52 (bytes into file)
  Start of section headers:    3232 (bytes into file)
  Flags:    0x0
  Size of this header:    52 (bytes)
  Size of program headers:    32 (bytes)
  Number of program headers:   8
  Size of section headers:     40 (bytes)
  Number of section headers:    29
  Section header string table index: 26
```

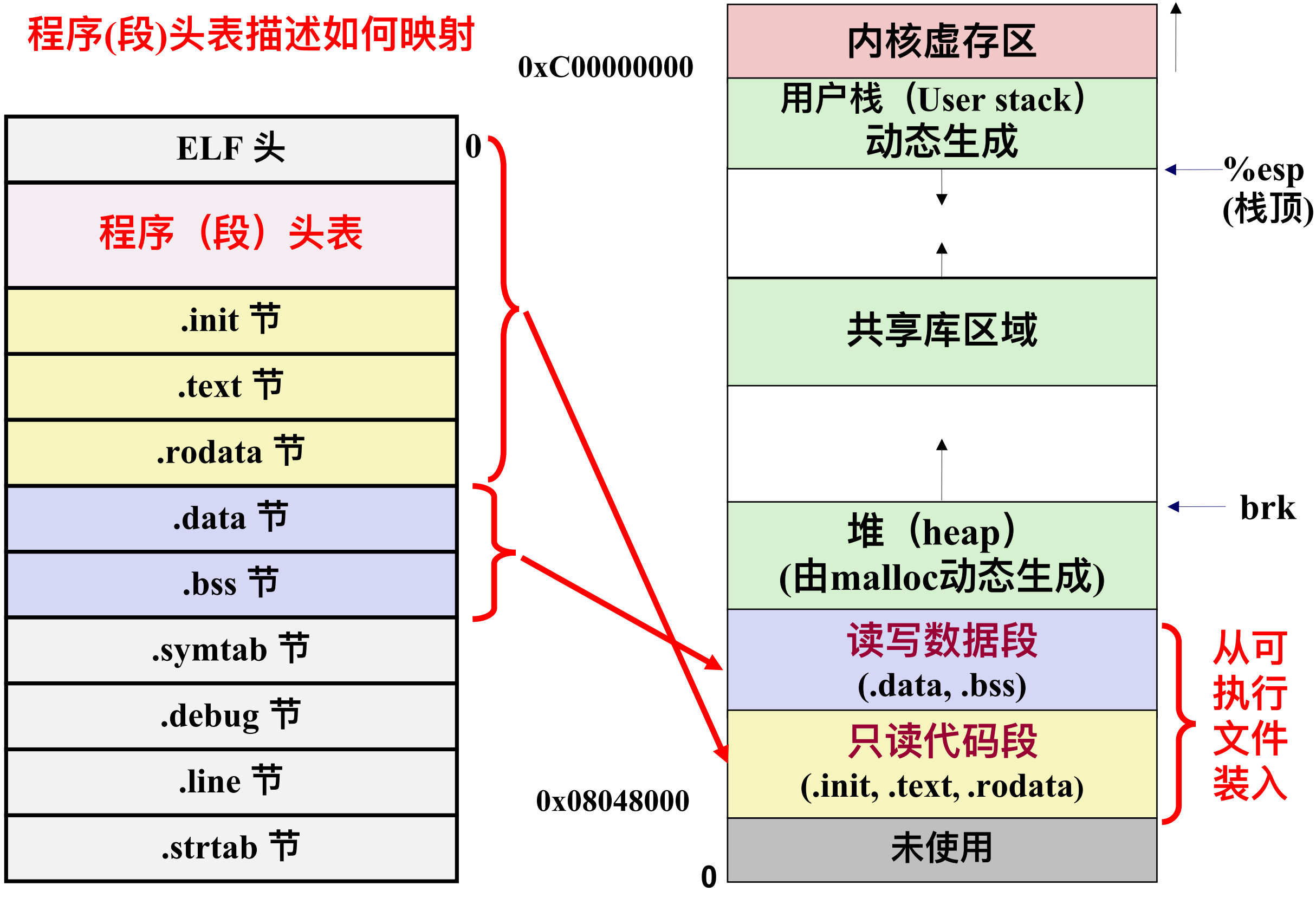
8x32B

29x40B



可执行文件的存储器映像

程序(段)头表描述如何映射



可执行文件中的程序头表

```
typedef struct {
        Elf32_Word    p_type;
        Elf32_Off     p_offset;
        Elf32_Addr    p_vaddr;
        Elf32_Addr    p_paddr;
        Elf32_Word    p_filesz;
        Elf32_Word    p_memsz;
        Elf32_Word    p_flags;
        Elf32_Word    p_align;
} Elf32_Phdr;
```

程序头表描述可执行文件中的节与虚拟空间中的存储段之间的映射关系

一个表项（32B）说明虚拟地址空间中一个连续的段或一个特殊的节

以下是某可执行目标文件程序头表信息

有8个表项，其中两个为可装入段（即Type=LOAD）

$ readelf –l main

```
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000034 0x08048034 0x08048034 0x00100 0x00100 R E 0x4
  INTERP         0x000134 0x08048134 0x08048134 0x00013 0x00013 R   0x1
      [Requesting program interpreter: /lib/ld-linux.so.2]
  LOAD           0x000000 0x08048000 0x08048000 0x004d4 0x004d4 R E 0x1000
  LOAD           0x000f0c 0x08049f0c 0x08049f0c 0x00108 0x00110 RW  0x1000
  DYNAMIC        0x000f20 0x08049f20 0x08049f20 0x000d0 0x000d0 RW  0x4
  NOTE           0x000148 0x08048148 0x08048148 0x00044 0x00044 R   0x4
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x4
  GNU_RELRO      0x000f0c 0x08049f0c 0x08049f0c 0x000f4 0x000f4 R   0x1
```

可执行文件中的程序头表

```
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000034 0x08048034 0x08048034 0x00100 0x00100 R E 0x4
  INTERP         0x000134 0x08048134 0x08048134 0x00013 0x00013 R   0x1
      [Requesting program interpreter: /lib/ld-linux.so.2]
  LOAD           0x000000 0x08048000 0x08048000 0x004d4 0x004d4 R E 0x1000
  LOAD           0x000f0c 0x08049f0c 0x08049f0c 0x00108 0x00110 RW  0x1000
  DYNAMIC        0x000f20 0x08049f20 0x08049f20 0x000d0 0x000d0 RW  0x4
  NOTE           0x000148 0x08048148 0x08048148 0x00044 0x00044 R   0x4
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x4
  GNU_RELRO      0x000f0c 0x08049f0c 0x08049f0c 0x000f4 0x000f4 R   0x1
```

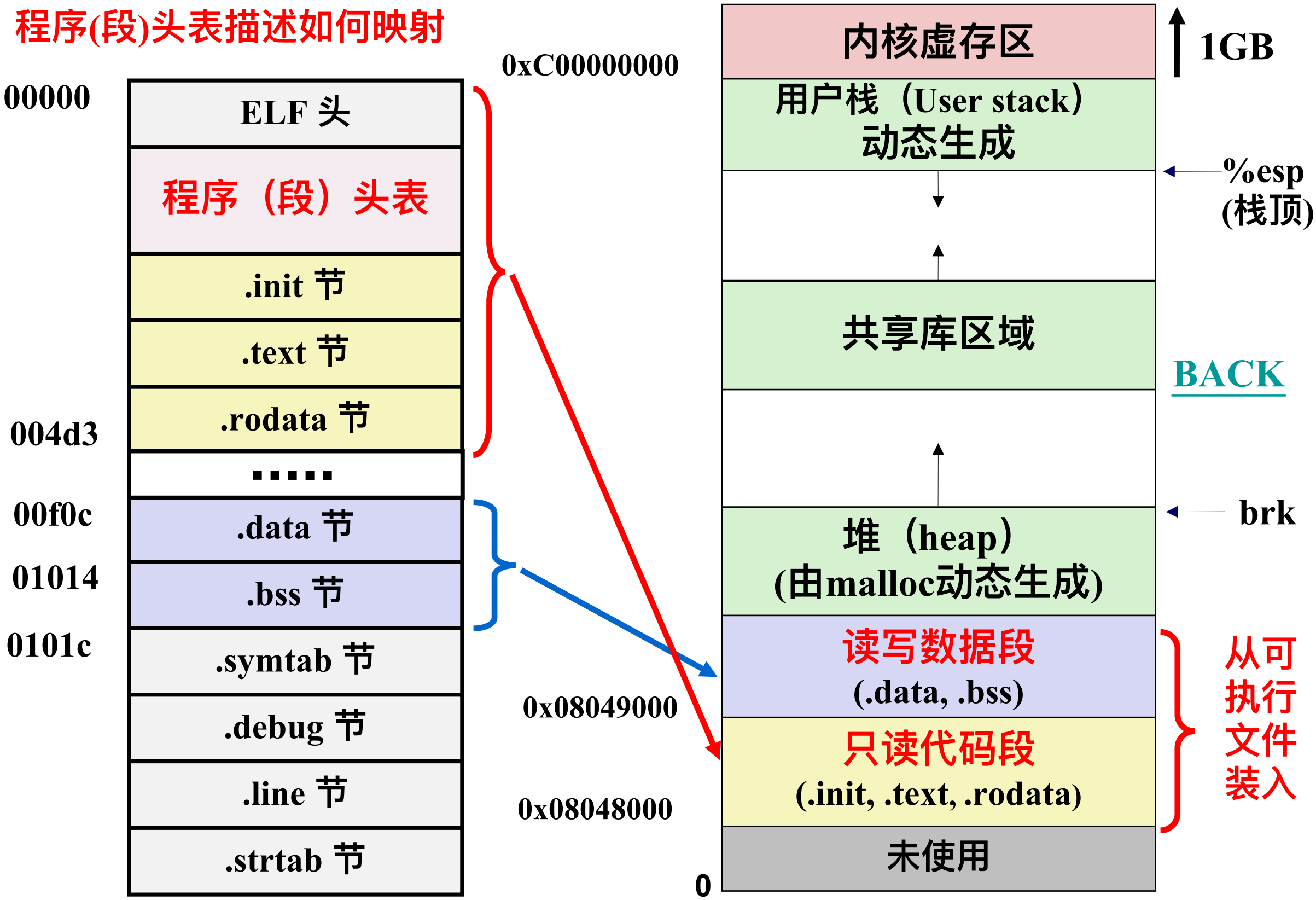
SKIP

第一可装入段：第0x00000~0x004d3字节（包括ELF头、程序头表、.init、.text和.rodata节），映射到虚拟地址0x8048000开始长度为0x4d4字节的区域，按0x1000=2^{12}=4KB对齐，具有只读/执行权限（Flg=RE），是只读代码段。

第二可装入段：第0x000f0c开始长度为0x108字节的.data节，映射到虚拟地址0x8049f0c开始长度为0x110字节的存储区域，在0x110=272B存储区中，前0x108=264B用.data节内容初始化，后面272-264=8B对应.bss节，初始化为0，按0x1000=4KB对齐，具有可读可写权限（Flg=RW），是可读写数据段。

可执行文件的存储器映像

程序(段)头表描述如何映射



程序的链接

- 分以下三个部分介绍
  - 第一讲：目标文件格式
    - 程序的链接概述、链接的意义与过程
    - ELF目标文件、重定位目标文件格式、可执行目标文件格式
  - 第二讲：符号解析与重定位
    - 符号和符号表、符号解析
    - 与静态库的链接
    - 重定位信息、重定位过程
    - 可执行文件的加载
  - 第三讲：动态链接
    - 动态链接的特性、程序加载时的动态链接、程序运行时的动态链接、动态链接举例

符号和符号解析

每个可重定位目标模块m都有一个符号表，它包含了在m中定义的符号。

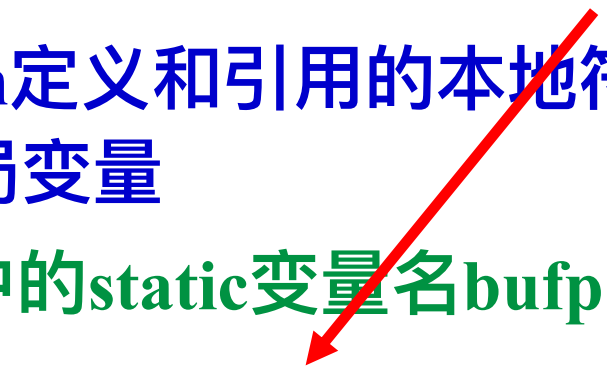
有三种链接器符号：

- Global symbols（模块内部定义的全局符号）
  - 由模块m定义并能被其他模块引用的符号。例如，非static 函数和非static的全局变量（指不带static的全局变量）

    如，main.c 中的全局变量名buf
- External symbols（外部定义的全局符号）
  - 由其他模块定义并被模块m引用的全局符号

    如，main.c 中的函数名swap
- Local symbols（本模块的局部符号）
  - 仅由模块m定义和引用的本地符号。例如，在模块m中定义的带static的函数和全局变量

    如，swap.c 中的static变量名bufp1



链接器局部符号不是指程序中的局部变量（分配在栈中的临时性变量）,链接器不关心这种局部变量

符号和符号解析

main.c

```
int buf[2] = {1, 2};
extern void swap();

int main()
{
  swap();
  return 0;
}
```

swap.c

```
extern int buf[];

int *bufp0 = &buf[0];
static int *bufp1;

void swap()
{
  int temp;

  bufp1 = &buf[1];
  temp = *bufp0;
  *bufp0 = *bufp1;
  *bufp1 = temp;
}
```

你能说出哪些是全局符号？哪些是外部符号？哪些是局部符号？

目标文件中的符号表

.symtab 节记录符号表信息，是一个结构数组

- 符号表（symtab）中每个条目的结构如下：

函数名在text节中

变量名在data节或bss节中

```
typedef  struct {
        int    name;    /*符号对应字符串在strtab节中的偏移量*/
        int    value;   /*在对应节中的偏移量，可执行文件中是虚拟地址*/
        int    size;    /*符号对应目标所占字节数*/
        char  type: 4,  /*符号对应目标的类型：数据、函数、源文件、节*/
              binding: 4; /*符号类别：全局符号、局部符号、弱符号*/
        char  reserved;
        char  section;  /*符号对应目标所在的节，或其他情况*/
} Elf_Symbol;
```

函数大小或变量长度

其他情况：ABS表示不该被重定位；UND表示未定义；COM表示未初始化数据（.bss），此时，value表示对齐要求，size给出最小大小

目标文件中的符号表

- main.o中的符号表中最后三个条目（共10个）

Num:	value	Size	Type	Bind	Ot	Ndx	Name
8:	0	8	Data	Global	0	3	buf
9:	0	33	Func	Global	0	1	main
10:	0	0	Notype	Global	0	UND	swap

buf是main.o中第3节（.data）偏移为0的符号，是全局变量，占8B； main是第1节（.text）偏移为0的符号，是全局函数，占33B；

swap是main.o中未定义全局（在其他模块定义）符号，类型和大小未知

- swap.o中的符号表中最后4个条目（共11个）

Num:	value	Size	Type	Bind	Ot	Ndx	Name
8:	0	4	Data	Global	0	3	bufp0
9:	0	0	Notype	Global	0	UND	buf
10:	0	36	Func	Global	0	1	swap
11:	4	4	Data	Local	0	COM	bufp1

bufp1是未分配地址且未初始化的本地变量(ndx=COM), 按4B对齐且占4B

符号解析（Symbol Resolution）

- 目的：将每个模块中引用的符号与某个目标模块中的定义符号建立关联。
- 每个定义符号在代码段或数据段中都被分配了存储空间，将引用符号与定义符号建立关联后，就可在重定位时将引用符号的地址重定位为相关联的定义符号的地址。
- 本地符号在本模块内定义并引用，因此，其解析较简单，只要与本模块内唯一的定义符号关联即可。
- 全局符号（外部定义的、内部定义的）的解析涉及多个模块，故较复杂

确定L0的地址，再在jmp指令中填入L0的地址

符号解析也称符号绑定

“符号的定义”其实质是什么?

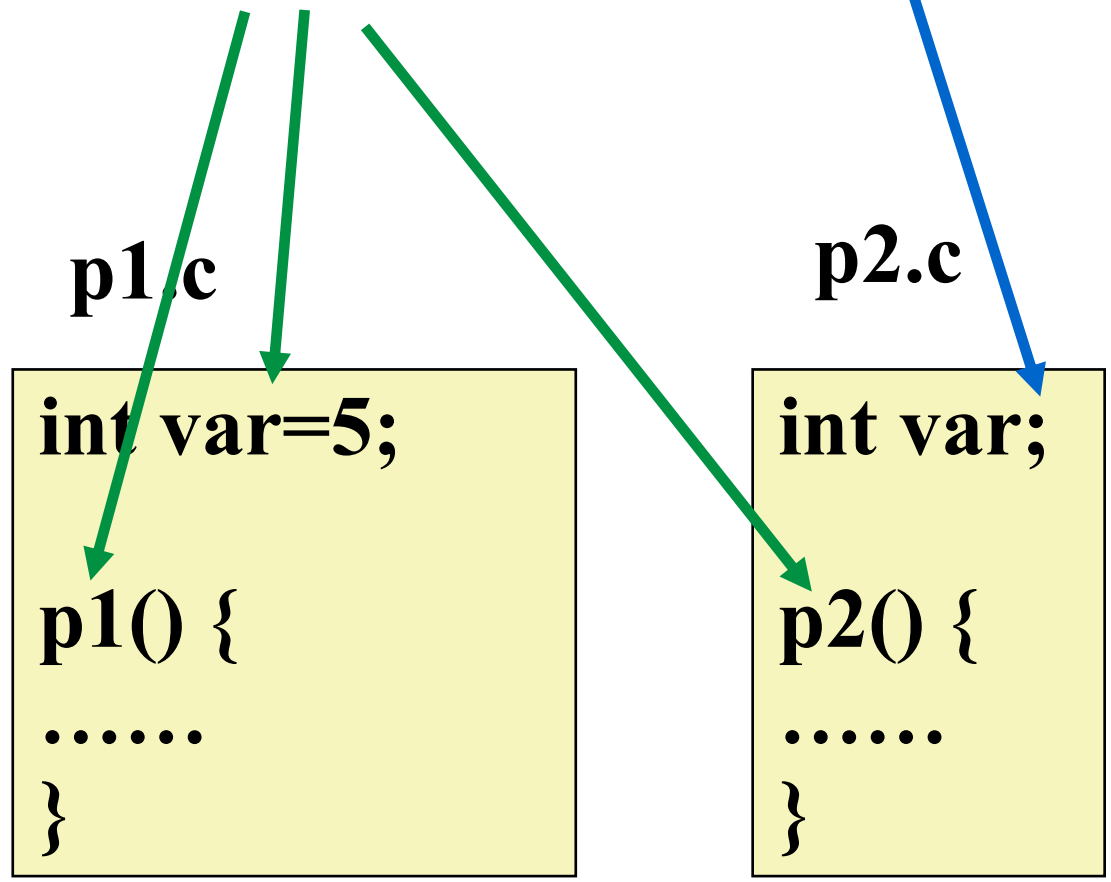
指被分配了存储空间。为函数名即指其代码所在区；为变量名即指其所占的静态数据区。

所有定义符号的值就是其目标所在的首地址

全局符号的符号解析

- 全局符号的强/弱特性
  - 函数名和已初始化的全局变量名是强符号
  - 未初始化的全局变量名是弱符号

以下符号哪些是强符号？哪些是弱符号？



全局符号的符号解析

以下符号哪些是强符号？哪些是弱符号？

main.c

```
int buf[2] = {1, 2};
void swap();

int main()
{
  swap();
  return 0;
}
```

此处为引用

swap.c

```
extern int buf[];

int *bufp0 = &buf[0];
static int *bufp1;

void swap()
{
  int temp;

  bufp1 = &buf[1];
  temp = *bufp0;
  *bufp0 = *bufp1;
  *bufp1 = temp;
}
```

局部变量

本地局部符号

链接器对符号的解析规则

符号解析时只能有一个确定的定义（即每个符号仅占一处存储空间）

- 多重定义符号的处理规则

Rule 1: 强符号不能多次定义

– 强符号只能被定义一次，否则链接错误

Rule 2: 若一个符号被定义为一次强符号和多次弱符号，则按强定义为准

– 对弱符号的引用被解析为其强定义符号

Rule 3: 若有多个弱符号定义，则任选其中一个

– 使用命令 gcc –fno-common链接时，会告诉链接器在遇到多个弱定义的全局符号时输出一条警告信息。

多重定义符号的解析举例

以下程序会发生链接出错吗?

```
int  x=10;
int  p1(void);
int main()
{
    x=p1();
    return x;
}
```

main.c

```
int  x=20;
int p1()
{
    return x;
}
```

p1.c

main只有一次强定义

p1有一次强定义，一次弱定义

x有两次强定义，所以，链接器将输出一条出错信息

多重定义符号的解析举例

以下程序会发生链接出错吗?

```
# include <stdio.h>
int  y=100;
int  z;
void  p1(void);
int main()
{
    z=1000;
    p1( );
    printf("y=%d, z=%d\n", y, z);
    return 0;
}
```

main.c

y一次强定义，一次弱定义
z两次弱定义
p1一次强定义，一次弱定义
main一次强定义

```
int  y;
int  z;
void p1( )
{
    y=200;
    z=2000;
}
```

p1.c

问题：打印结果是什么?

y=200，z=2000

该例说明：在两个不同模块定义相同变量名，很可能发生意想不到的结果！

多重定义符号的解析举例

以下程序会发生链接出错吗?

```
1  #include <stdio.h>
2  int d=100;
3  int x=200;
4  void p1(void);
5  int main()
6  {
7      p1();
8      printf("d=%d,x=%d\n",d,x);
9      return 0;
10 }
```

main.c

```
1  double d;
2
3  void p1()
4  {
5      d=1.0;
6  }
```

p1.c

FLD1
FSTPl &d

问题：打印结果是什么?

d=0,x=1 072 693 248

该例说明：两个重复定义的变量具有不同类型时，更容易出现难以理解的结果！

p1执行后d和x处内容是什么?

	0	1	2	3
&x	00	00	F0	3F
&d	00	00	00	00

1.0：0 01111111111 0…0B

=3FF0 0000 0000 0000H

多重定义符号的解析举例

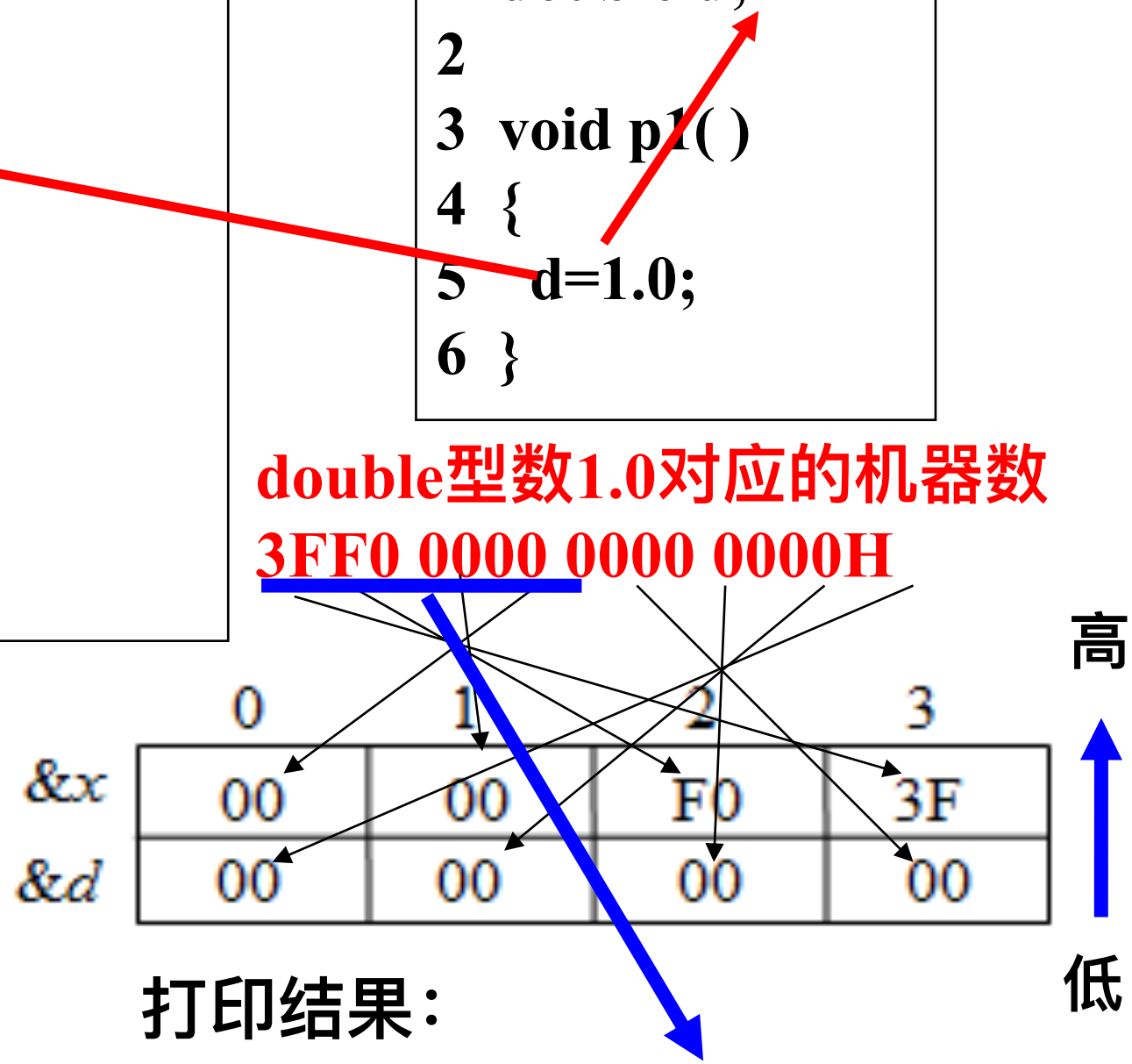
main.c

```
.......
1  int d=100;
2  int x=200;
3  int main()
4  {
5    p1( );
6    printf (“d=%d, x=%d\n”, d, x );
7    return 0;
8  }
```

p1.c

```
1  double d;
2
3  void p1( )
4  {
5    d=1.0;
6  }
```

double型数1.0对应的机器数
3FF0 0000 0000 0000H



IA-32是小端方式

$2^{30}-1-(2^{20}-1)=2^{30}-2^{20}$

$=1024*1024*1023$

$=1\ 072\ 693\ 248$

打印结果：

d=0，x=1 072 693 248

Why?

多重定义全局符号的问题

- 尽量避免使用全局变量
- 一定需要用的话，就按以下规则使用
  - 尽量使用本地变量（static）
  - 全局变量要赋初值
  - 外部全局变量要使用extern

多重定义全局变量会造成一些意想不到的错误，而且是默默发生的，编译系统不会警告，并会在程序执行很久后才能表现出来，且远离错误引发处。特别是在一个具有几百个模块的大型软件中，这类错误很难修正。

大部分程序员并不了解链接器如何工作，因而养成良好的编程习惯是非常重要的。

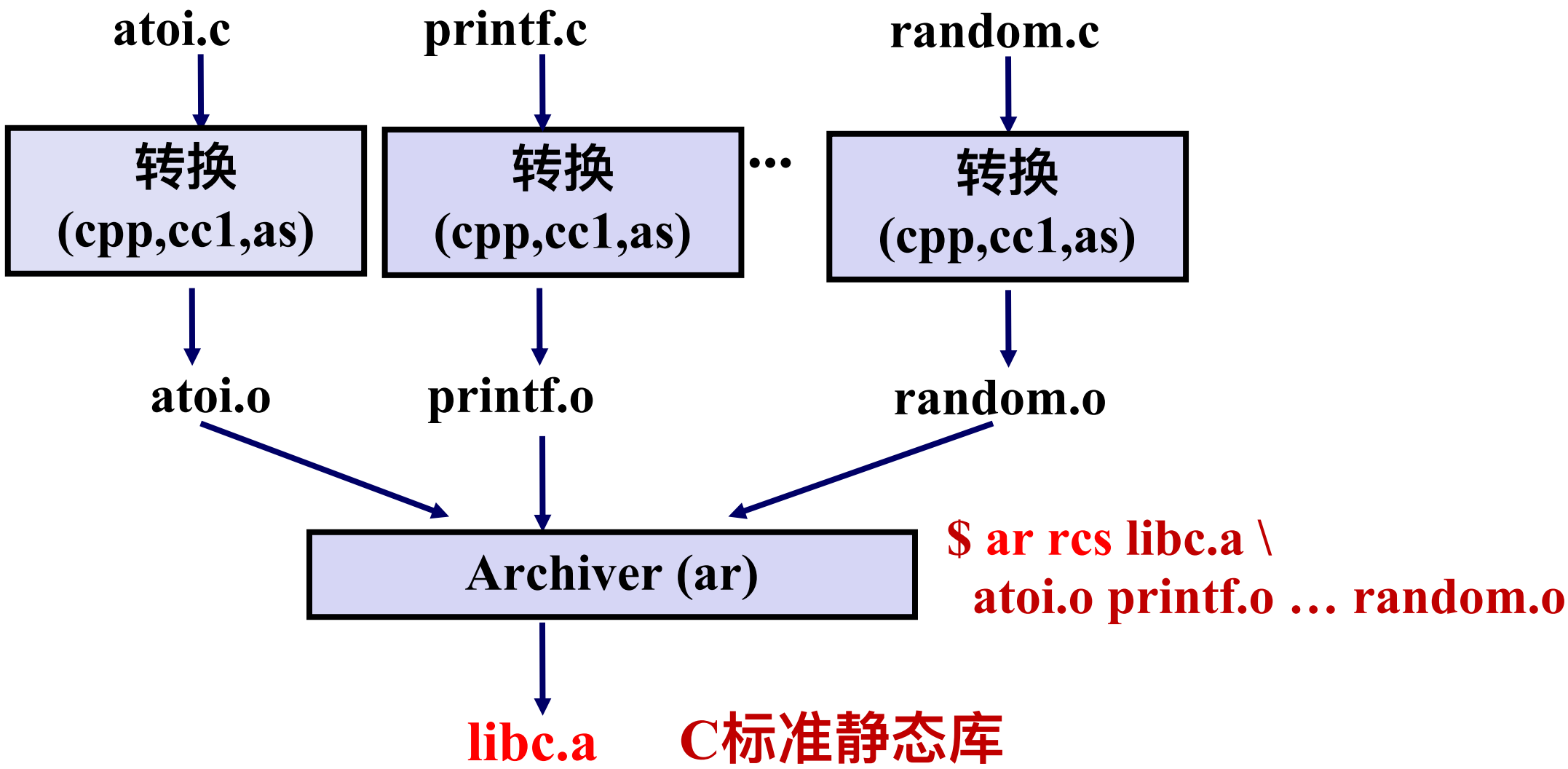
如何划分模块?

- 许多函数无需自己写，可使用共享库函数
  - 如数学库, 输入/输出库, 存储管理库，字符串处理等
- 避免以下两种极端做法
  - 将所有函数都放在一个源文件中
    - 修改一个函数需要对所有函数重新编译
    - 时间和空间两方面的效率都不高
  - 一个源文件中仅包含一个函数
    - 需要程序员显式地进行链接
    - 效率高，但模块太多，故太繁琐

静态共享库

- 静态库 (.a archive files)
  - 将所有相关的目标模块（.o）打包为一个单独的库文件（.a），称为静态库文件，也称存档文件（archive）
  - 增强了链接器功能，使其能通过查找一个或多个库文件中的符号来解析符号
  - 在构建可执行文件时只需指定库文件名，链接器会自动到库中寻找那些应用程序用到的目标模块，并且只把用到的模块从库中拷贝出来
  - 在gcc命令行中无需明显指定C标准库libc.a(默认库)

静态库的创建



- Archiver（归档器）允许增量更新，只要重新编译需修改的源码并将其.o文件替换到静态库中。

常用静态库

libc.a (C标准库)

- 1392个目标文件（大约8 MB）
- 包含I/O、存储分配、信号处理、字符串处理、时间和日期、随机数生成、定点整数算术运算

libm.a (the C math library)

- 401 个目标文件（大约 1 MB）
- 浮点数算术运算(如sin, cos, tan, log, exp, sqrt, …)

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

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

自定义一个静态库文件

举例：将myproc1.o和myproc2.o打包生成mylib.a

myproc1.c

```
# include <stdio.h>
void myfunc1() {
    printf("This is myfunc1!\n");
}
```

myproc2.c

```
# include <stdio.h>
void myfunc2() {
    printf("This is myfunc2\n");
}
```

```
$ gcc –c myproc1.c myproc2.c
$ ar rcs mylib.a myproc1.o myproc2.o
```

main.c

```
void myfunc1(viod);
int main()
{
   myfunc1();
   return 0;
}
```

```
$ gcc –c main.c
$ gcc –static –o myproc main.o ./mylib.a
```

libc.a无需明显指出！

调用关系：main→myfunc1→printf

问题：如何进行符号解析?

链接器中符号解析的全过程

libc.a无需明显指出！

```
$ gcc –c main.c
$ gcc –static –o myproc main.o ./mylib.a
```

调用关系：main→myfunc1→printf

main.c

```
void myfunc1(viod);
int main()
{
   myfunc1();
   return 0;
}
```

E 将被合并以组成可执行文件的所有目标文件集合
U 当前所有未解析的引用符号的集合
D 当前所有定义符号的集合

开始E、U、D为空，首先扫描main.o，把它加入E，同时把myfun1加入U，main加入D。接着扫描到mylib.a，将U中所有符号（本例中为myfunc1）与mylib.a中所有目标模块（myproc1.o和myproc2.o）依次匹配，发现在myproc1.o中定义了myfunc1，故myproc1.o加入E，myfunc1从U转移到D。在myproc1.o中发现还有未解析符号printf，将其加到U。不断在mylib.a的各模块上进行迭代以匹配U中的符号，直到U、D都不再变化。此时U中只有一个未解析符号printf，而D中有main和myfunc1。因为模块myproc2.o没有被加入E中，因而它被丢弃。

接着，扫描默认的库文件libc.a，发现其目标模块printf.o定义了printf，于是printf也从U移到D，并将printf.o加入E，同时把它定义的所有符号加入D，而所有未解析符号加入U。

处理完libc.a时，U一定是空的。

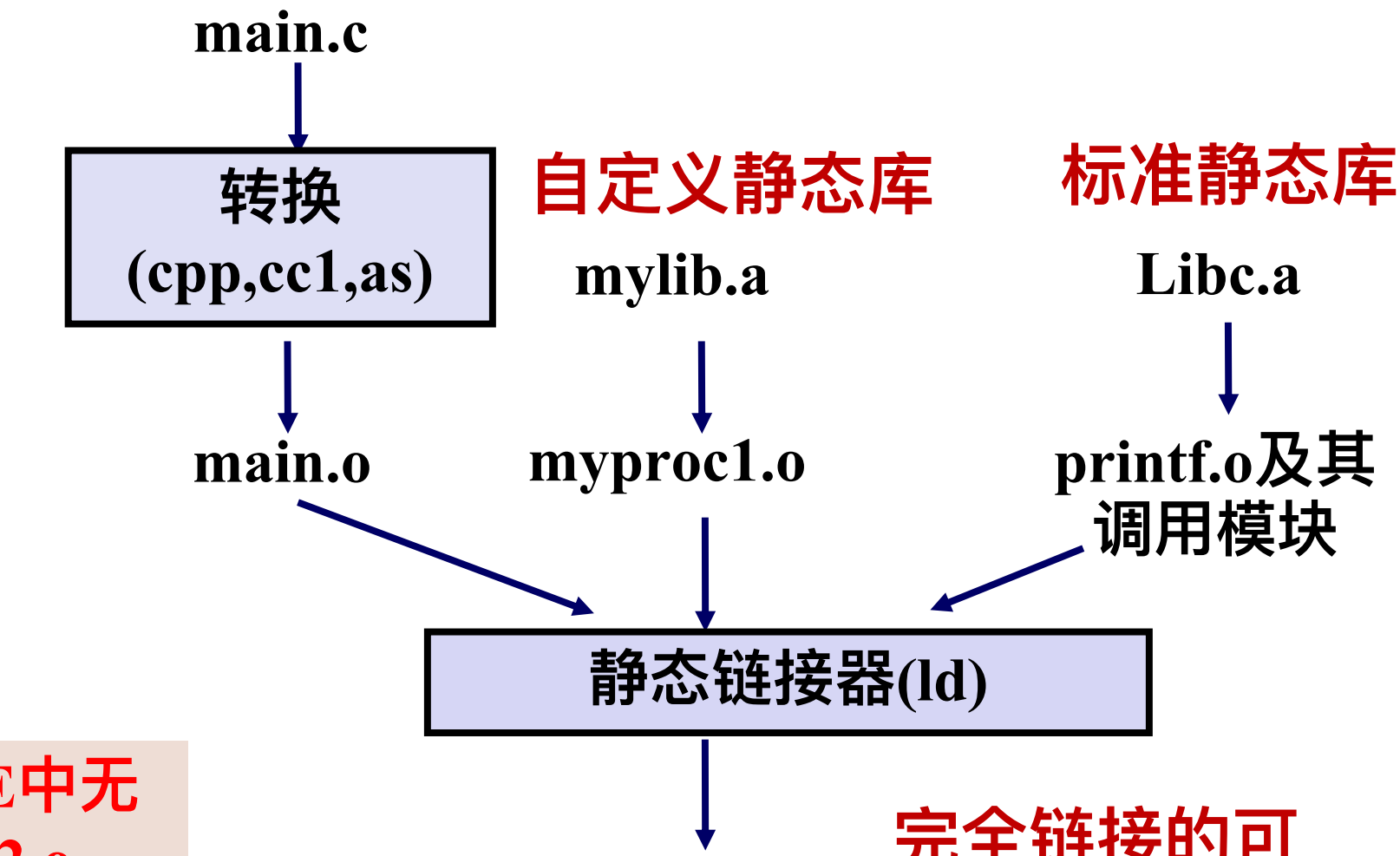
链接器中符号解析的全过程

$ gcc –static –o myproc main.o ./mylib.a

main→myfunc1→printf

main.c

```
void myfunc1(viod);
int main()
{
   myfunc1();
   return 0;
}
```



注意：E中无myproc2.o

解析结果：

E中有main.o、myproc1.o、printf.o及其调用的模块

D中有main、myproc1、printf及其引用的符号

链接器中符号解析的全过程

main.c

```
void myfunc1(viod);
int main()
{
   myfunc1();
   return 0;
}
```

main→myfunc1→printf

$ gcc –static –o myproc main.o ./mylib.a

解析结果：

E中有main.o、myproc1.o、printf.o及其调用的模块

D中有main、myproc1、printf及其引用符号

被链接模块应按调用顺序指定！

若命令为：$ gcc –static –o myproc ./mylib.a main.o，结果怎样?

首先，扫描mylib，因是静态库，应根据其中是否存在U中未解析符号对应的定义符号来确定哪个.o被加入E。因为开始U为空，故其中两个.o模块都不被加入E中而被丢弃。

然后，扫描main.o，将myfunc1加入U，直到最后它都不能被解析。 **Why?**

因此，出现链接错误！

它只能用mylib.a中符号来解析，而mylib中两个.o模块都已被丢弃！

使用静态库

- 链接器对外部引用的解析算法要点如下：
  - 按照命令行给出的顺序扫描.o 和.a 文件
  - 扫描期间将当前未解析的引用记录到一个列表U中
  - 每遇到一个新的.o 或 .a 中的模块，都试图用其来解析U中的符号
  - 如果扫描到最后，U中还有未被解析的符号，则发生错误
- 问题和对策
  - 能否正确解析与命令行给出的顺序有关
  - 好的做法：将静态库放在命令行的最后

libmine.a 是静态库

假设调用关系：libtest.o→libfun.o(在libmine.a中）

-lxxx=libxxx.a (main) →(libfun)

```
$ gcc -L. libtest.o -lmine
$ gcc -L. -lmine libtest.o
libtest.o: In function `main':
libtest.o(.text+0x4): undefined reference to `libfun'
```

扫描libtest.o，将libfun送U，扫描到libmine.a时，用其定义的libfun来解析

说明在libtest.o中的main调用了libfun这个在库libmine中的函数，所以，在命令行中，应该将libtest.o放在前面，像第一行中那样！

链接顺序问题

- 假设调用关系如下：

  func.o → libx.a 和 liby.a 中的函数

  libx.a → libz.a 中的函数

  libx.a 和 liby.a 之间、liby.a 和 libz.a 相互独立

  则以下几个命令行都是可行的：

  - gcc -static –o myfunc func.o libx.a liby.a libz.a
  - gcc -static –o myfunc func.o liby.a libx.a libz.a
  - gcc -static –o myfunc func.o libx.a libz.a liby.a

- 假设调用关系如下：

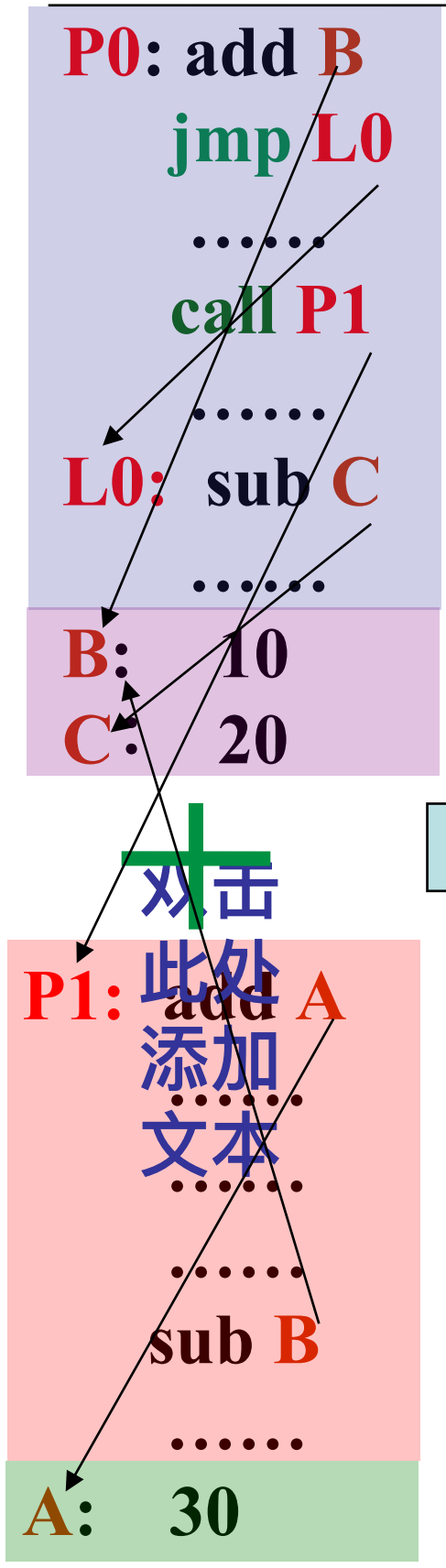
  func.o → libx.a 和 liby.a 中的函数

  libx.a → liby.a 同时 liby.a → libx.a

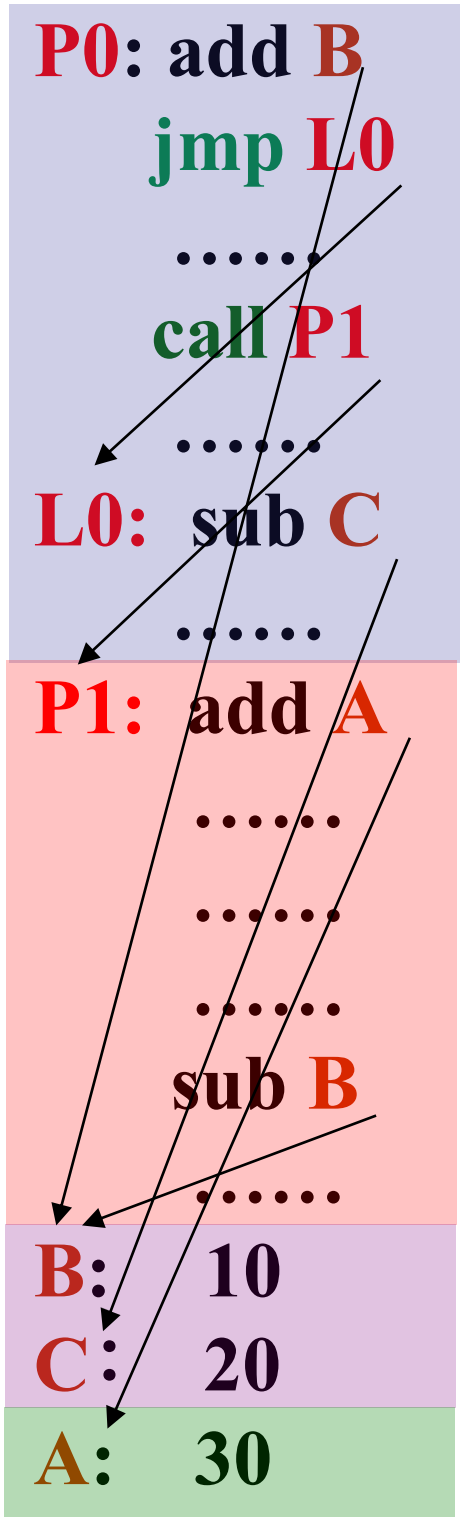
  则以下命令行可行：

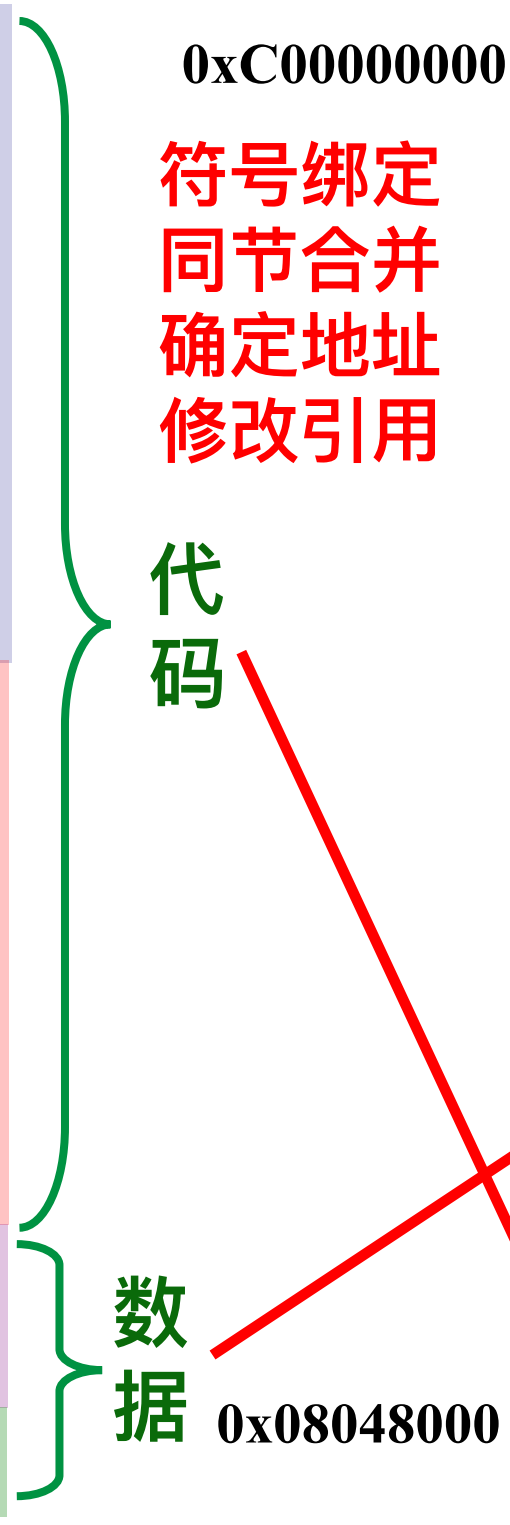
  - gcc -static –o myfunc func.o libx.a liby.a libx.a

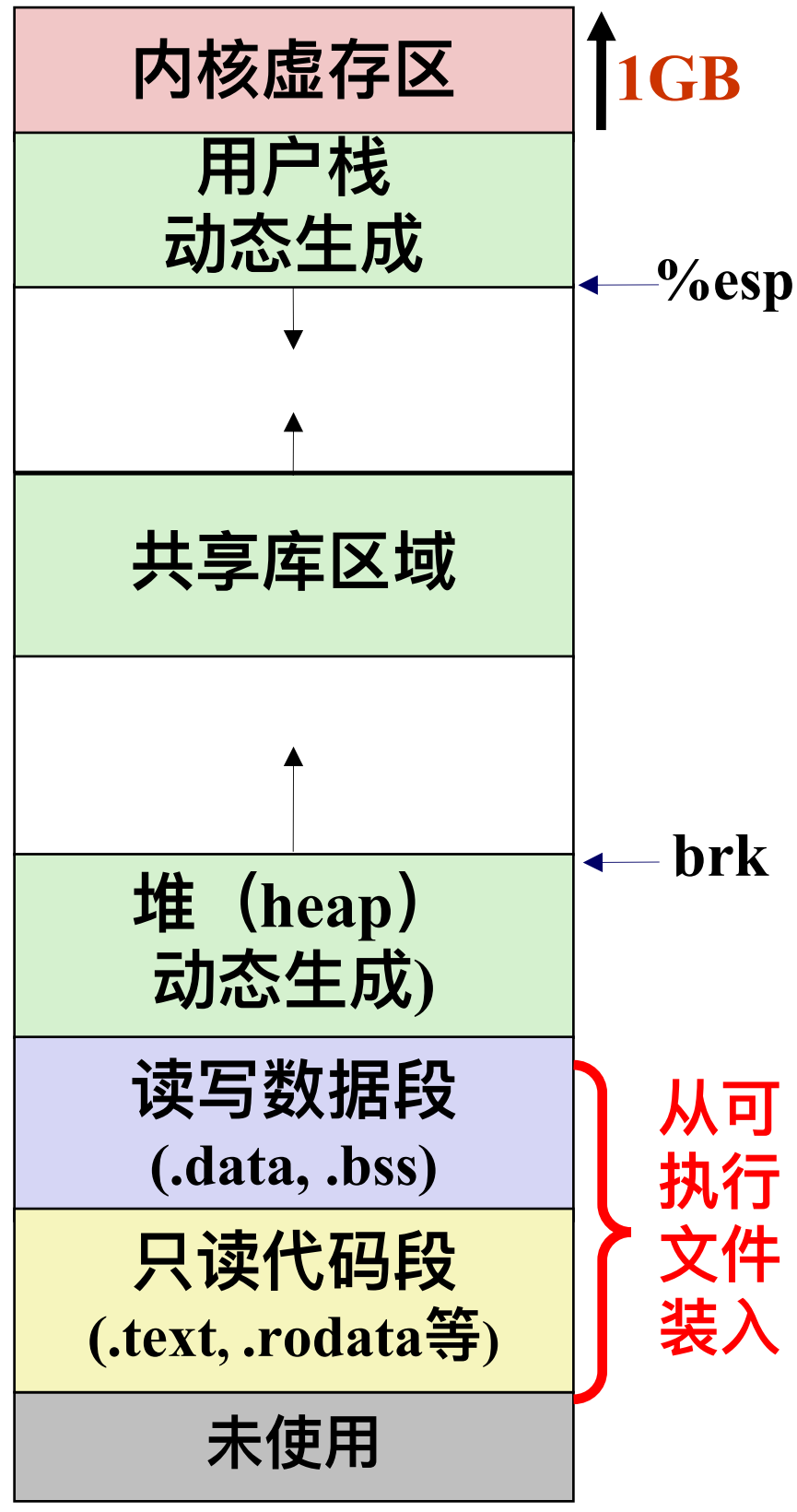
链接操作的步骤


P0: add B
jmp L0
……
call P1
……
L0: sub C
……
B: 10
C: 20

双击
此处
添加
文本
P1: add A
……
……
……
sub B
……
A: 30


P0: add B
jmp L0
……
call P1
……
L0: sub C
……
P1: add A
……
……
……
sub B
……
B: 10
C: 20
A: 30


0xC0000000
符号绑定
同节合并
确定地址
修改引用
代码
数据
0x08048000


内核虚存区
1GB
用户栈
动态生成
%esp
共享库区域
brk
堆（heap）
动态生成)
读写数据段
(.data, .bss)
只读代码段
(.text, .rodata等)
从可执行文件装入
未使用
0

目标文件

```
/* main.c */
int add(int, int);
int main( )
{
   return add(20, 13);
}
```

```
/* test.c */
int add(int i, int j)
{
    int x = i + j;
    return x;
}
```

objdump -d test.o

```
00000000 <add>:
   0:   55              push  %ebp
   1:   89 e5           mov   %esp, %ebp
   3:   83 ec 10        sub   $0x10, %esp
   6:   8b 45 0c        mov   0xc(%ebp), %eax
   9:   8b 55 08        mov   0x8(%ebp), %edx
   c:   8d 04 02        lea   (%edx,%eax,1), %eax
   f:   89 45 fc        mov   %eax, -0x4(%ebp)
  12:   8b 45 fc        mov   -0x4(%ebp), %eax
  15:   c9              leave
  16:   c3              ret
```

objdump -d test

```
080483d4 <add>:
 80483d4:   55              push  %ebp
 80483d5:   89 e5           mov   %esp, %ebp
 80483d7:   83 ec 10        sub   $0x10, %esp
 80483da:   8b 45 0c        mov   0xc(%ebp), %eax
 80483dd:   8b 55 08        mov   0x8(%ebp), %edx
 80483e0:   8d 04 02        lea   (%edx,%eax,1), %eax
 80483e3:   89 45 fc        mov   %eax, -0x4(%ebp)
 80483e6:   8b 45 fc        mov   -0x4(%ebp), %eax
 80483e9:   c9              leave
 80483ea:   c3              ret
```

重定位

符号解析完成后，可进行重定位工作，分三步

- 合并相同的节
  - 将集合E的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
- 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变量都可确定地址
- 对引用符号进行重定位（确定地址）
  - 修改.text节和.data节中对每个符号的引用（地址）

    需要用到在.rel_data和.rel_text节中保存的重定位信息

重定位信息

- 汇编器遇到引用时，生成一个重定位条目
- 数据引用的重定位条目在.rel_data节中
- 指令中引用的重定位条目在.rel_text节中
- ELF中重定位条目格式如下：

```
typedef  struct {
        int  offset;        /*节内偏移*/
        int  symbol:24,  /*所绑定符号*/
          type: 8;       /*重定位类型*/
} Elf32_Rel;
```

- IA-32有两种最基本的重定位类型
  - R_386_32: 绝对地址
  - R_386_PC32: PC相对地址

```
    add B
    jmp L0
    ……
L0：sub 23
    ……
B：  ……
```

```
05 00000000
02 FCFFFFFF
    ……
L0：sub 23
    ……
B：  ……
```

例如，在rel_text节中有重定位条目如下

offset: 0x1
symbol: B
type: R_386_32

offset: 0x6
symbol: L0
type: R_386_PC32

问题：重定位条目和汇编后的机器代码在哪种目标文件中?

在可重定位目标（.o）文件中!

重定位操作举例

main.c

```
int buf[2] = {1, 2};
void swap();

int main()
{
  swap();
  return 0;
}
```

swap.c

```
extern int buf[];

int *bufp0 = &buf[0];
static int *bufp1;

void swap()
{
  int temp;
  bufp1 = &buf[1];
  temp = *bufp0;
  *bufp0 = *bufp1;
  *bufp1 = temp;
}
```

你能说出哪些是符号定义？哪些是符号的引用？

局部变量temp分配在栈中，不会在过程外被引用，因此不是符号定义

重定位操作举例

main.c

```
int buf[2] = {1, 2};
void swap();

int main()
{
  swap();
  return 0;
}
```

swap.c

```
extern int buf[];

int *bufp0 = &buf[0];
static int *bufp1;

void swap()
{
   int temp;
   bufp1 = &buf[1];
   temp = *bufp0;
   *bufp0 = *bufp1;
   *bufp1 = temp;
}
```

符号解析后的结果是什么？
E中有printf.o吗？

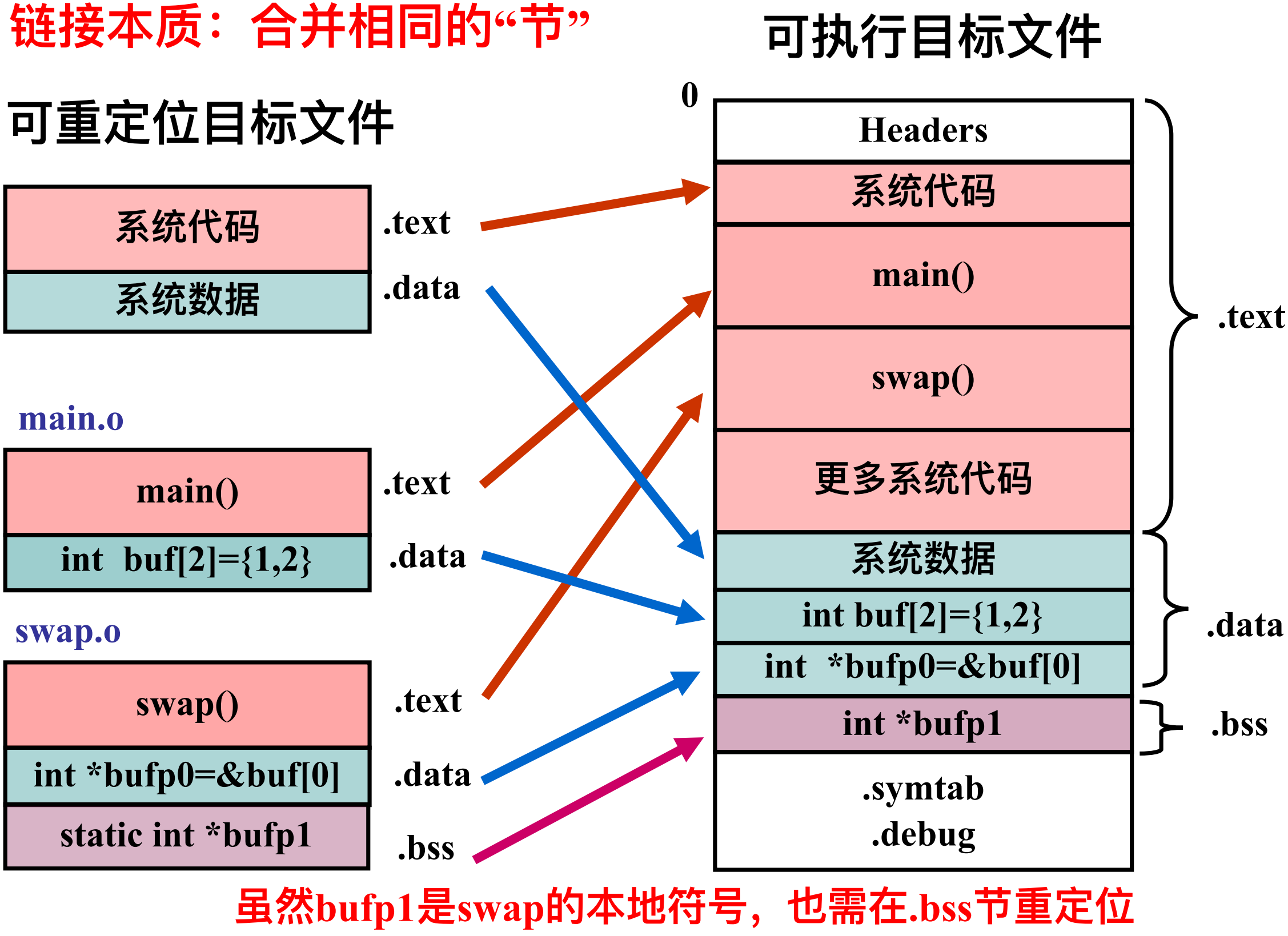
E中有main.o和swap.o两个模块！D中有所有定义的符号！

在main.o和swap.o的重定位节(.rel.text、.rel.data)中有重定位信息，反映符号引用的位置、绑定的定义符号名、重定位类型

用命令readelf -r main.o可显示main.o中的重定位条目（表项）

符号引用的地址需要重定位

链接本质：合并相同的“节”



虽然bufp1是swap的本地符号，也需在.bss节重定位

main.o重定位前

main.c

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

int main()
{
  swap();
  return 0;
}
```

main的定义在.text节中偏移为0处开始，占0x12B。

main.o

```
Disassembly of section .text:
00000000 <main>:
   0:    55                      push   %ebp
   1:    89 e5                   mov    %esp,%ebp
   3:    83 e4 f0                and    $0xfffffff0,%esp
   6:    e8 fc ff ff ff          call   7 <main+0x7>
                                 7: R_386_PC32 swap
   b:    b8 00 00 00 00          mov    $0x0,%eax
  10:    c9                      leave
  11:    c3                      ret
```

```
Disassembly of section .data:

00000000 <buf>:
  0:   01 00 00 00 02 00 00 00
```

buf的定义在.data节中偏移为0处开始，占8B。

在rel_text节中的重定位条目为：
r_offset=0x7, r_sym=10,
r_type=R_386_PC32，dump出来后为“7: R_386_PC32 swap”

r_sym=10说明引用的是swap！

main.o中的符号表

- main.o中的符号表中最后三个条目

Num:	value	Size	Type	Bind	Ot	Ndx	Name
8:	0	8	Data	Global	0	3	buf
9:	0	18	Func	Global	0	1	main
10:	0	0	Notype	Global	0	UND	swap

swap是main.o的符号表中第10项，是未定义符号，类型和大小未知，并是全局符号，故在其他模块中定义。

在rel_text节中的重定位条目为：

r_offset=0x7, r_sym=10,

r_type=R_386_PC32，dump出来后为“7: R_386_PC32 swap”

r_sym=10说明引用的是swap！

R_386_PC32的重定位方式

- 假定：
  - 可执行文件中main
  - swap紧跟main后，
- 则swap起始地址为多
  - 0x8048380+0x12=0x8048392
  - 在4字节边界对齐的情况下，是0x8048394
- 则重定位后call指令的机器代码是什么？
  - 转移目标地址=PC+偏移地址，PC=0x8048380+0x07-init
  - PC=0x8048380+0x07-(-4)=0x804838b
  - 重定位值=转移目标地址-PC=0048394-0x804838b=0x9
  - call指令的机器代码为“e8 09 00 00 00”

```
Disassembly of section .text:
00000000 <main>:
   ......
   6:     e8 fc ff ff ff      call   7 <main+0x7>
                              7: R_386_PC32 swap
   ......
```

重定位值

值为-4

SKIP

PC相对地址方式下，重定位值计算公式为：

ADDR(r_sym) – ((ADDR(.text) + r_offset) – init)

引用目标处　　call指令下条指令地址　　即当前PC的值

确定定义符号的地址

可执行目标文件

0

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

0xC0000000

内核虚存区

1GB

用户栈
动态生成

%esp

共享库区域

堆（heap）
动态生成)

brk

读写数据段
(.data, .bss)

只读代码段
(.text, .rodata等)

未使用

从可执行文件装入

0x08048000

0

BACK

R_386_32的重定位方式

main.o中.data和.rel.data节内容

```
Disassembly of section .data:

00000000 <buf>:
  0:   01 00 00 00 02 00 00 00
```

buf定义在.data节中偏移为0处，占8B，没有需重定位的符号。

main.c

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

int main()
……
```

swap.c

```
extern int buf[];


int *bufp0 = &buf[0];
static int *bufp1;

void swap()
……
```

swap.o中.data和.rel.data节内容

```
Disassembly of section .data:

00000000 <bufp0>:
  0:   00 00 00 00
       0：R_386_32  buf
```

bufp0定义在.data节中偏移为0处，占4B，初值为0x0

重定位节.rel.data中有一个重定位表项：r_offset=0x0, r_sym=9, r_type=R_386_32，OBJDUMP工具解释后显示为“0：R_386_32 buf”

r_sym=9说明引用的是buf!

swap.o中的符号表

- swap.o中的符号表中最后4个条目

Num:	value	Size	Type	Bind	Ot	Ndx	Name
8:	0	4	Data	Global	0	3	bufp0
9:	0	0	Notype	Global	0	UND	buf
10:	0	36	Func	Global	0	1	swap
11:	4	4	Data	Local	0	COM	bufp1

buf是swap.o的符号表中第9项，是未定义符号，类型和大小未知，并是全局符号，故在其他模块中定义。

重定位节.rel.data中有一个重定位表项：r_offset=0x0, r_sym=9, r_type=R_386_32，OBJDUMP工具解释后显示为“0：R_386_32 buf”

r_sym=9说明引用的是buf!

R_386_32的重定位方式

- 假定：
  - buf在运行时的存储地址ADDR(buf)=0x8049620
- 则重定位后，bufp0的地址及内容变为什么？
  - buf和bufp0同属于.data节，故在可执行文件中它们被合并
  - bufp0紧接在buf后，故地址为0x8049620+8= 0x8049628
  - 因是R_386_32方式，故bufp0内容为buf的绝对地址0x8049620，即“20 96 04 08”

可执行目标文件中.data节的内容

```
Disassembly of section .data:

08049620 <buf>:
  8049620:                    01 00 00 00 02 00 00 00

08049628 <bufp0>:
  8049628:                    20 96 04 08
```

swap.o重定位

swap.c

```
extern int buf[];

int *bufp0 = &buf[0];
static int *bufp1;

void swap()
{
  int temp;

  bufp1 = &buf[1];
  temp = *bufp0;
  *bufp0 = *bufp1;
  *bufp1 = temp;
}
```

共有6处需要重定位

划红线处：8、c、11、1b、21、2a

```
Disassembly of section .text:
00000000 <swap>:
   0:   55                      push   %ebp
   1:   89 e5                   mov    %esp,%ebp
   3:   83 ec 10                sub    $0x10,%esp
   6:   c7 05 00 00 00 00 04    movl   $0x4,0x0
   d:   00 00 00
                        8: R_386_32     .bss
                        c: R_386_32     buf
  10:   a1 00 00 00 00          mov    0x0,%eax
                        11: R_386_32    bufp0
  15:   8b 00                   mov    (%eax),%eax
  17:   89 45 fc                mov    %eax,-0x4(%ebp)
  1a:   a1 00 00 00 00          mov    0x0,%eax
                        1b: R_386_32    bufp0
  1f:   8b 15 00 00 00 00       mov    0x0,%edx
                        21: R_386_32    .bss
  25:   8b 12                   mov    (%edx),%edx
  27:   89 10                   mov    %edx,(%eax)
  29:   a1 00 00 00 00          mov    0x0,%eax
                        2a: R_386_32    .bss
  2e:   8b 55 fc                mov    -0x4(%ebp),%edx
  31:   89 10                   mov    %edx,(%eax)
  33:   c9                      leave
  34:   c3                      ret
```

swap.o重定位

buf和bufp0的地址分别是0x8049620和0x8049628

&buf[1](c处重定位值）为0x8049620+0x4=0x8049624

bufp1的地址就是链接合并后.bss节的首地址，假定为0x8049700

8 (bufp1)：00 97 04 08

c (&buf[1])：24 96 04 08

11 (bufp0)：28 96 04 08

1b (bufp0) : 28 96 04 08

21 (bufp1)：00 97 04 08

2a (bufp1)：00 97 04 08

```
bufp1 = &buf[1];
temp = *bufp0;
*bufp0 = *bufp1;
*bufp1 = temp;
```

```
6:    c7 05 00 00 00 00 04 movl   $0x4,0x0
d:    00 00 00
                           8: R_386_32      .bss
                           c: R_386_32      buf
10:   a1 00 00 00 00       mov    0x0,%eax
                           11: R_386_32     bufp0
15:   8b 00                mov    (%eax),%eax
17:   89 45 fc             mov    %eax,-0x4(%ebp)
1a:   a1 00 00 00 00       mov    0x0,%eax
                           1b: R_386_32     bufp0
1f:   8b 15 00 00 00 00    mov    0x0,%edx
                           21: R_386_32     .bss
25:   8b 12                mov    (%edx),%edx
27:   89 10                mov    %edx,(%eax)
29:   a1 00 00 00 00       mov    0x0,%eax
                           2a: R_386_32     .bss
2e:   8b 55 fc             mov    -0x4(%ebp),%edx
31:   89 10                mov    %edx,(%eax)
```

重定位后

```
08048380 <main>:
 8048380:   55              push  %ebp
 8048381:   89 e5           mov   %esp,%ebp
 8048383:   83 e4 f0        and   $0xfffffff0,%esp
 8048386:   e8 09 00 00 00  call  8048394 <swap>
 804838b:   b8 00 00 00 00  mov   $0x0,%eax
 8048390:   c9
 8048391:   c3
 8048392:   90
 8048393:   90
```

你能写出该call指令的功能描述吗?

假定每个函数要求4字节边界对齐,故填充两条nop指令

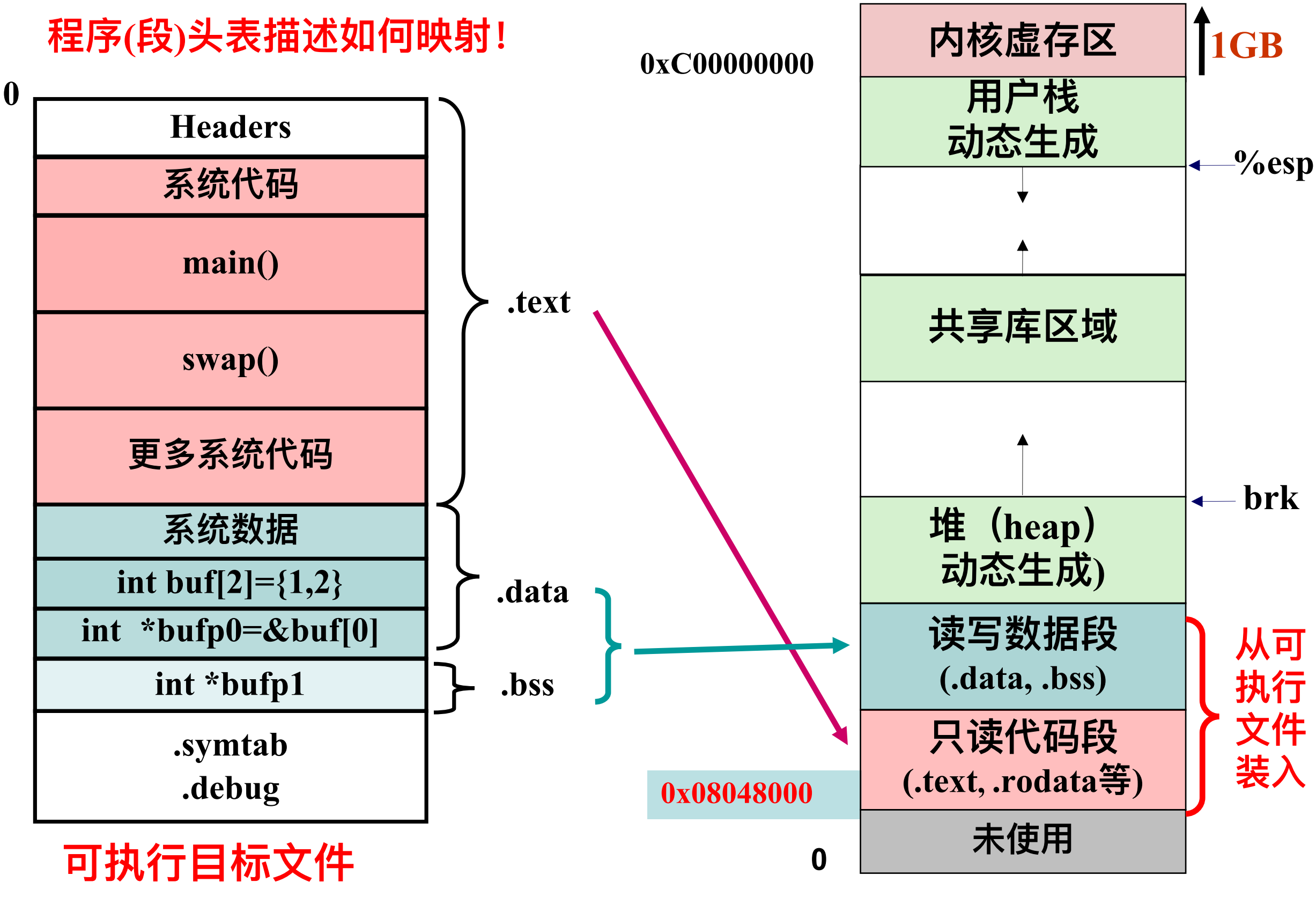
R[eip]=0x804838b

1) R[esp]← R[esp]-4
- M[R[esp]] ←R[eip]
- R[eip] ←R[eip]+0x9

```
08048394 <swap>:
 8048394:   55                    push  %ebp
 8048395:   89 e5                 mov   %esp,%ebp
 8048397:   83 ec 10              sub   $0x10,%esp
 804839a:   c7 05 00 97 04 08 24  mov $0x8049624,0x8049700
 80483a1:   96 04 08
 80483a4:   a1 28 96 04 08        mov  0x8049628,%eax
 80483a9:   8b 00                 mov  (%eax),%eax
 80483ab:   89 45 fc              mov   %eax,-0x4(%ebp)
 80483ae:   a1 28 96 04 08        mov  0x8049628,%eax
 80483b3:   8b 15 00 97 04 08     mov  0x8049700,%edx
 80493b9:   8b 12                 mov  (%edx),%edx
 80493bb:   89 10                 mov  %edx,(%eax)
 80493bd:   a1 00 97 04 08        mov  0x8049700,%eax
 80493c2:   8b 55 fc              mov  -0x4(%ebp),%edx
 80493c5:   89 10                 mov  %edx,(%eax)
 80493c7:   c9                    leave
 80493c8:   c3                    ret
```

可执行文件的存储器映像

程序(段)头表描述如何映射！



回顾：可执行文件中的程序头表

```
typedef struct {
        Elf32_Word    p_type;
        Elf32_Off     p_offset;
        Elf32_Addr    p_vaddr;
        Elf32_Addr    p_paddr;
        Elf32_Word    p_filesz;
        Elf32_Word    p_memsz;
        Elf32_Word    p_flags;
        Elf32_Word    p_align;
} Elf32_Phdr;
```

程序头表能够描述可执行文件中的节与虚拟空间中的存储段之间的映射关系

一个表项说明虚拟地址空间中一个连续的片段或一个特殊的节

以下是GNU READELF显示的某可执行目标文件的程序头表信息

$ readelf –l main

```
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000034 0x08048034 0x08048034 0x00100 0x00100 R E 0x4
  INTERP         0x000134 0x08048134 0x08048134 0x00013 0x00013 R   0x1
      [Requesting program interpreter: /lib/ld-linux.so.2]
  LOAD           0x000000 0x08048000 0x08048000 0x004d4 0x004d4 R E 0x1000
  LOAD           0x000f0c 0x08049f0c 0x08049f0c 0x00108 0x00110 RW  0x1000
  DYNAMIC        0x000f20 0x08049f20 0x08049f20 0x000d0 0x000d0 RW  0x4
  NOTE           0x000148 0x08048148 0x08048148 0x00044 0x00044 R   0x4
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x4
  GNU_RELRO      0x000f0c 0x08049f0c 0x08049f0c 0x000f4 0x000f4 R   0x1
```

程序头（段头）表的信息

- 程序头表中包含了可执行文件中连续的片（chunk）如何映射到连续的存储段的信息。 也可用命令：$ readelf –l main
- 以下是由OBJDUMP得到某可执行文件的段头部表内容

双击此处添加文本

Read-only code segment

LOAD off 0x00000000 vaddr 0x08048000 paddr 0x08048000 align 2**12
filesz 0x00000448 memsz 0x00000448 flags r-x

Read/write data segment

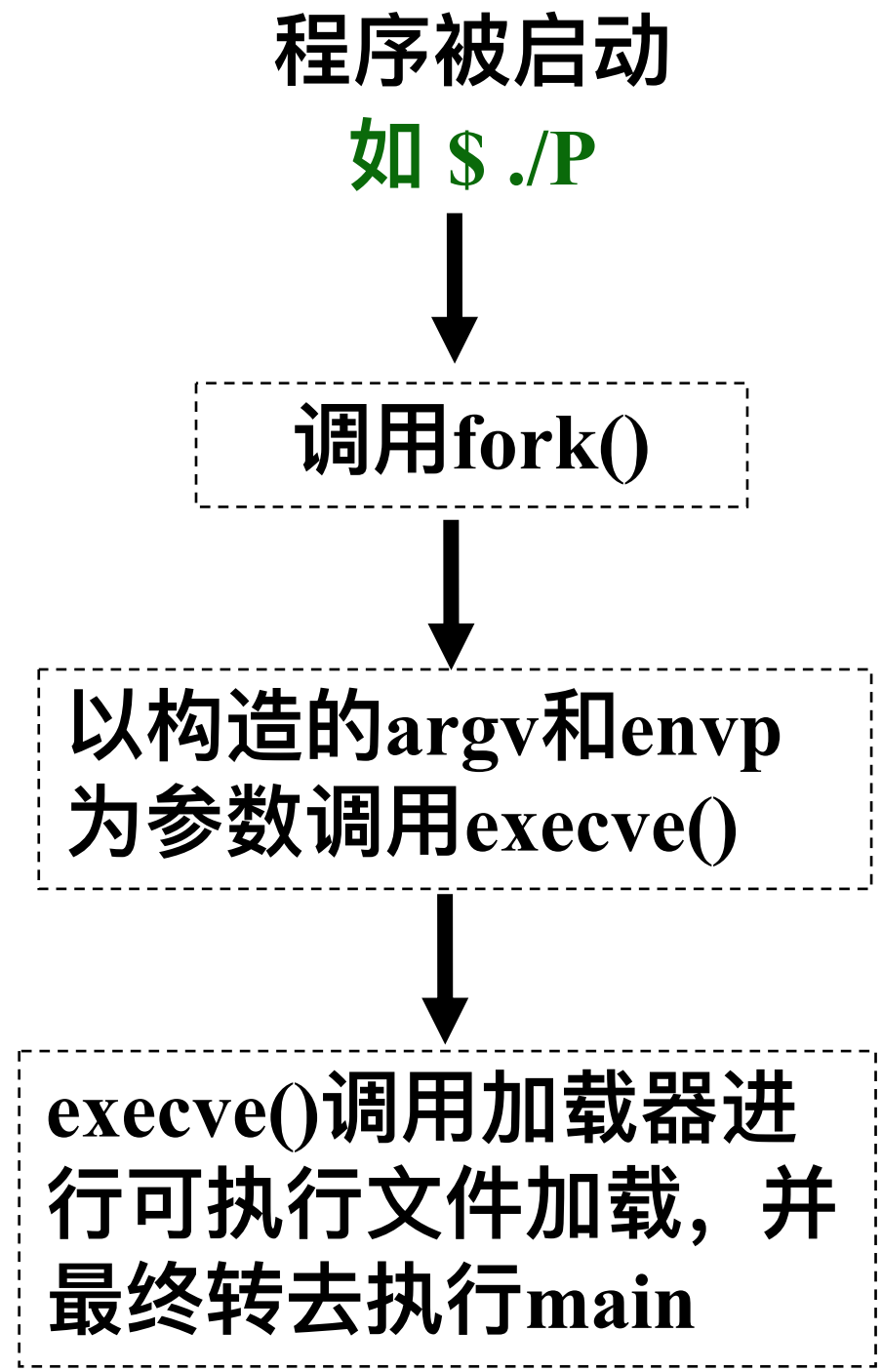
LOAD off 0x00000448 vaddr 0x08049448 paddr 0x08049448 align 2**12
filesz 0x000000e8 memsz 0x00000104 flags rw-

代码段：从0x8048000开始，按4KB对齐，具有读/执行权限，对应可执行文件第0~447H的内容（包括ELF头、段头部表以及.init、.text和.rodata节）

数据段：从0x8049448开始，按4KB对齐，具有读/写权限，前E8H字节用可执行文件.data节内容初始化，后面104H-E8H=10H（32）字节对应.bss节，被初始化为0

可执行文件的加载

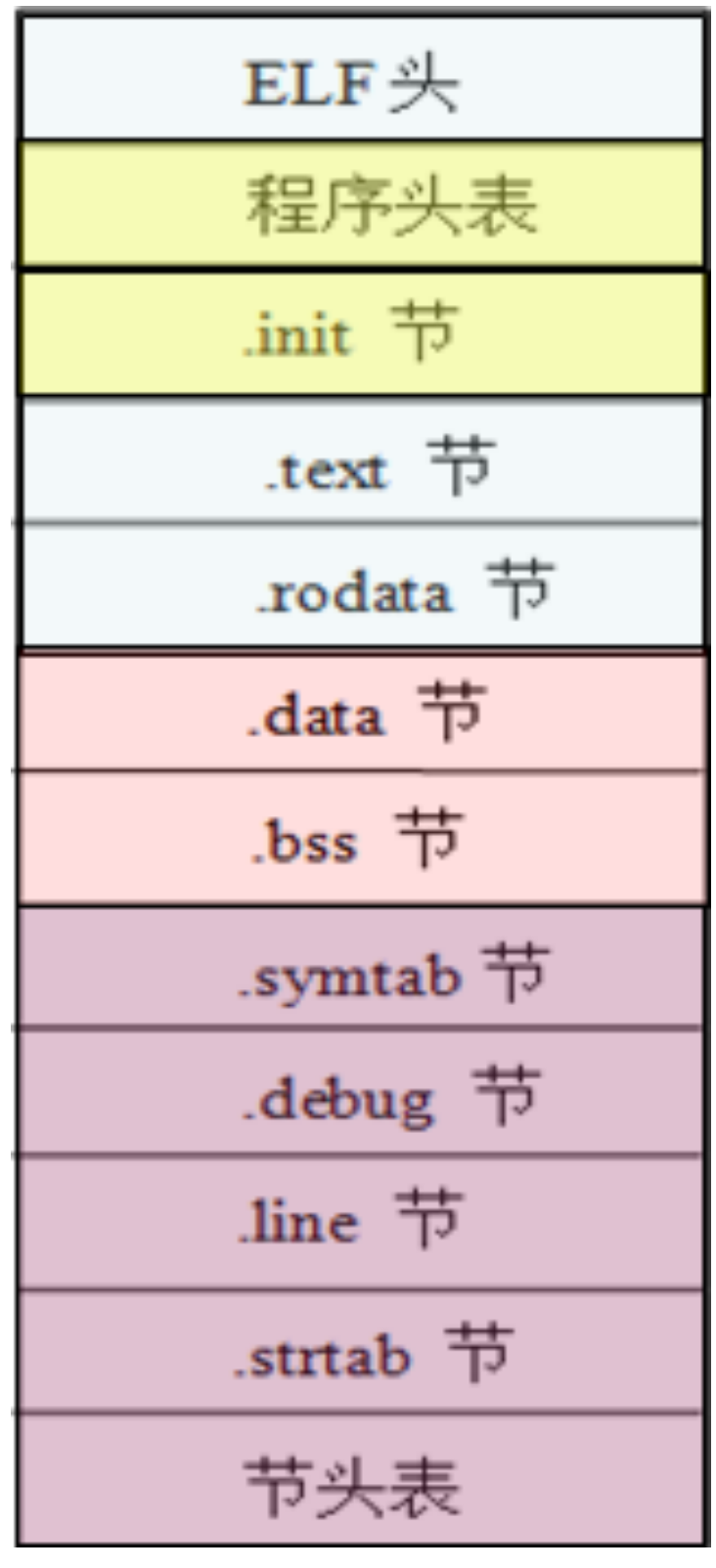
- 通过调用execve系统调用函数来调用加载器
- 加载器（loader）根据可执行文件的程序（段）头表中的信息，将可执行文件的代码和数据从磁盘“拷贝”到存储器中（实际上不会真正拷贝，仅建立一种映像，这涉及到许多复杂的过程和一些重要概念，将在后续课上学习）
- 加载后，将PC（EIP）设定指向Entry point (即符号_start处)，最终执行main函数，以启动程序执行。



ELF文件信息举例

$ readelf -h main　　可执行目标文件的ELF头

```
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:    ELF32
  Data:      2's complement, little endian
  Version:  1 (current)
  OS/ABI:    UNIX - System V
  ABI Version:     0
  Type:    EXEC (Executable file)
  Machine:   Intel 80386
  Version:   0x1
  Entry point address:    x8048580
  Start of program headers:  52 (bytes into file)
  Start of section headers:    3232 (bytes into file)
  Flags:    0x0
  Size of this header:    52 (bytes)
  Size of program headers:    32 (bytes)
  Number of program headers:   8
  Size of section headers:     40 (bytes)
  Number of section headers:    29
  Section header string table index: 26
```



程序的链接

- 分以下三个部分介绍
  - 第一讲：目标文件格式
    - 程序的链接概述、链接的意义与过程
    - ELF目标文件、重定位目标文件格式、可执行目标文件格式
  - 第二讲：符号解析与重定位
    - 符号和符号表、符号解析
    - 与静态库的链接
    - 重定位信息、重定位过程
    - 可执行文件的加载
  - 第三讲：动态链接
    - 动态链接的特性、程序加载时的动态链接、程序运行时的动态链接、动态链接举例

动态链接的共享库（Shared Libraries）

- 静态库有一些缺点：
  - 库函数（如printf）被包含在每个运行进程的代码段中，对于并发运行上百个进程的系统，造成极大的主存资源浪费
  - 库函数（如printf）被合并在可执行目标中，磁盘上存放着数千个可执行文件，造成磁盘空间的极大浪费
  - 程序员需关注是否有函数库的新版本出现，并须定期下载、重新编译和链接，更新困难、使用不便
- 解决方案: Shared Libraries （共享库）
  - 是一个目标文件，包含有代码和数据
  - 从程序中分离出来，磁盘和内存中都只有一个备份
  - 可以动态地在装入时或运行时被加载并链接
  - Window称其为动态链接库（Dynamic Link Libraries，.dll文件）
  - Linux称其为动态共享对象（ Dynamic Shared Objects, .so文件）

共享库（Shared Libraries）

动态链接可以按以下两种方式进行：

- 在第一次加载并运行时进行 (load-time linking).
  - Linux通常由动态链接器(ld-linux.so)自动处理
  - 标准C库 (libc.so) 通常按这种方式动态被链接
- 在已经开始运行后进行(run-time linking).
  - 在Linux中，通过调用 dlopen()等接口来实现
    - 分发软件包、构建高性能Web服务器等

在内存中只有一个备份，被所有进程共享，节省内存空间

一个共享库目标文件被所有程序共享链接，节省磁盘空间

共享库升级时，被自动加载到内存和程序动态链接，使用方便

共享库可分模块、独立、用不同编程语言进行开发，效率高

第三方开发的共享库可作为程序插件，使程序功能易于扩展

自定义一个动态共享库文件

myproc1.c

```
# include <stdio.h>
void myfunc1()
{
    printf("%s","This is myfunc1!\n");
}
```

myproc2.c

```
# include <stdio.h>
void myfunc2()
{
    printf("%s","This is myfunc2\n");
}
```

PIC：Position Independent Code

位置无关代码

1）保证共享库代码的位置可以是不确定的

双击此处添加文本

2）即使共享库代码的长度发生变化，也不会影响调用它的程序

位置无关的共享代码库文件

```
gcc –c myproc1.c myproc2.c
gcc –shared –fPIC –o mylib.so myproc1.o myproc2.o
```

加载时动态链接

libc.so无需明显指出

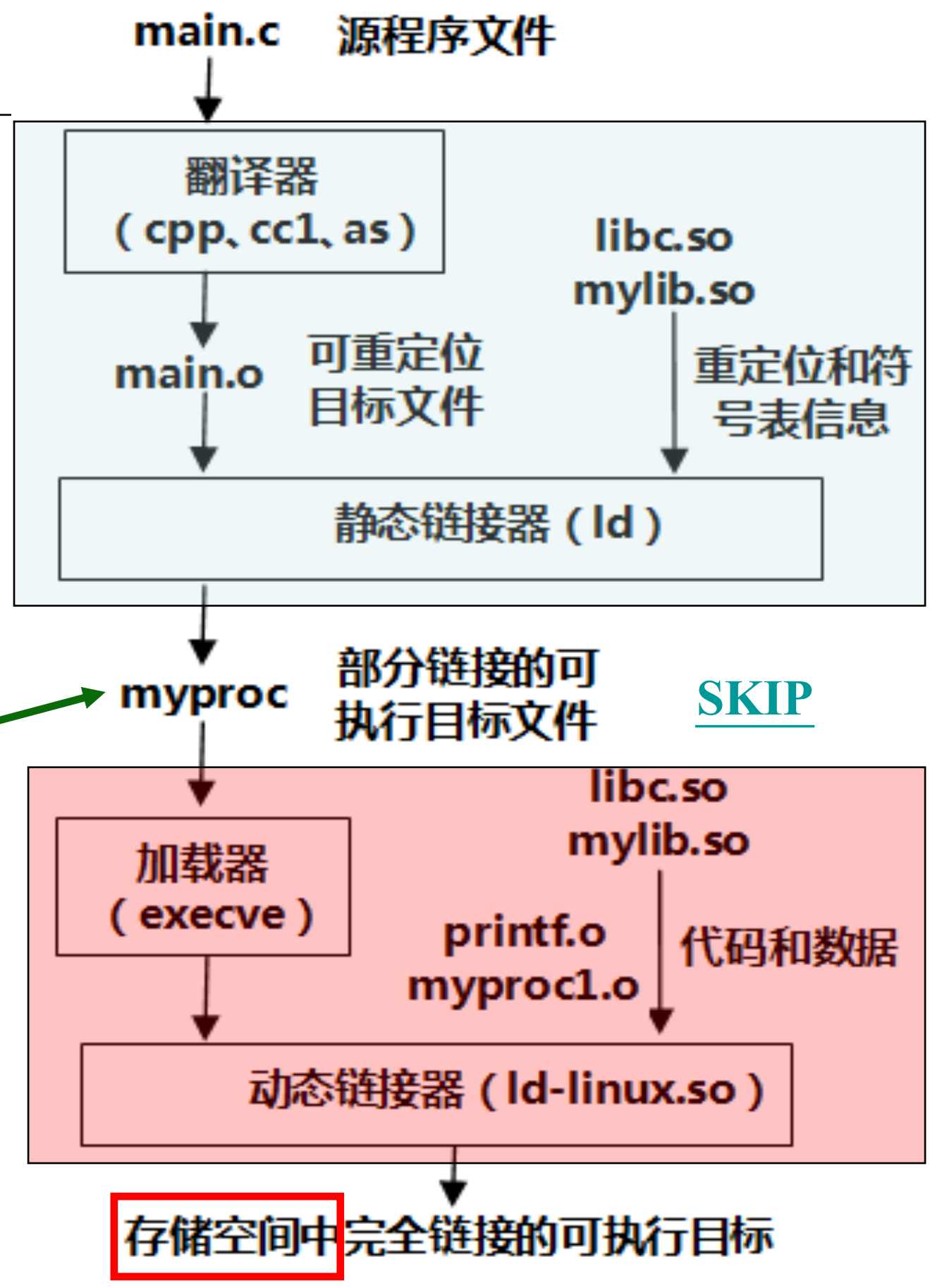
gcc –c main.c

gcc –o myproc main.o ./mylib.so

调用关系：main→myfunc1→printf

main.c

```
void myfunc1(viod);
int main()
{
   myfunc1();
   return 0;
}
```



加载 myproc 时，加载器发现在其程序头表中有 .interp 段，其中包含了动态链接器路径名 ld-linux.so，因而加载器根据指定路径加载并启动动态链接器运行。动态链接器完成相应的重定位工作后，再把控制权交给myproc，启动其第一条指令执行。

加载时动态链接

- 程序头表中有一个特殊的段：INTERP
- 其中记录了动态链接器目录及文件名ld-linux.so

BACK

```
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000034 0x08048034 0x08048034 0x00100 0x00100 R E 0x4
  INTERP         0x000134 0x08048134 0x08048134 0x00013 0x00013 R   0x1
      [Requesting program interpreter: /lib/ld-linux.so.2]
  LOAD           0x000000 0x08048000 0x08048000 0x004d4 0x004d4 R E 0x1000
  LOAD           0x000f0c 0x08049f0c 0x08049f0c 0x00108 0x00110 RW  0x1000
  DYNAMIC        0x000f20 0x08049f20 0x08049f20 0x000d0 0x000d0 RW  0x4
  NOTE           0x000148 0x08048148 0x08048148 0x00044 0x00044 R   0x4
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x4
  GNU_RELRO      0x000f0c 0x08049f0c 0x08049f0c 0x000f4 0x000f4 R   0x1
```

运行时动态链接

可通过动态链接器接口提供的函数在运行时进行动态链接

类UNIX系统中的动态链接器接口定义了相应的函数，如dlopen, dlsym, dlerror, dlclose等，其头文件为dlfcn.h

```
#include <stdio.h>
#include <dlfcn.h>
int main()
{
    void *handle;
    void (*myfunc1)();
    char *error;
     /* 动态装入包含函数myfunc1()的共享库文件 */
    handle = dlopen("./mylib.so", RTLD_LAZY);
    if (!handle) {
            fprintf(stderr, "%s\n", dlerror());
            exit(1);
     }
     /* 获得一个指向函数myfunc1()的指针myfunc1*/
     myfunc1 = dlsym(handle, "myfunc1");
     if ((error = dlerror()) != NULL) {
            fprintf(stderr, "%s\n", error);
            exit(1);
     }
     /* 现在可以像调用其他函数一样调用函数myfunc1() */
     myfunc1();
     /* 关闭（卸载）共享库文件 */
     if (dlclose(handle) < 0) {
            fprintf(stderr, "%s\n", dlerror());
            exit(1);
     }
     return 0;
}
```

位置无关代码（PIC）

- 动态链接用到一个重要概念：
  - 位置无关代码（Position-Independent Code，PIC
  - GCC选项-fPIC指示生成PIC代码
- 共享库代码是一种PIC
  - 共享库代码的位置可以是不确定的
  - 即使共享库代码的长度发生变化，也不影响调用它的程序

要实现动态链接，必须生成PIC代码

- 引入PIC的目的
  - 链接器无需修改代码即可将共享库加载到任意地址运行
- 所有引用情况

(1) 模块内的过程调用、跳转，采用PC相对偏移寻址

(2) 模块内数据访问，如模块内的全局变量和静态变量

(3) 模块外的过程调用、跳转

(4) 模块外的数据访问，如外部变量的访问

} 要生成PIC代码，主要解决这两个问题

(1) 模块内部函数调用或跳转

- 调用或跳转源与目的地都在同一个模块，相对位置固定，只要用相对偏移寻址即可
- 无需动态链接器进行重定位

```
8048344 <bar>:
 8048344:    55                      pushl  %ebp
 8048345:    89 e5                   movl  %esp, %ebp
 ……
 8048352:    c3                      ret
 8048353:    90                      nop

8048354  <foo>:
 8048354:    55                      pushl %ebp
 ……
 8048364:    e8 db ff ff ff     call 8048344 <bar>
 8048369:
 ……
```

```
static int a;
static int b;
extern  void ext();

void bar()
{
    a=1;
    b=2;
}

void foo()
{
    bar();
    ext();
}
```

call的目标地址为：
0x8048369+
0xfffffdb(-0x25)=
0x8048344

JMP指令也可用相对寻址方式解决

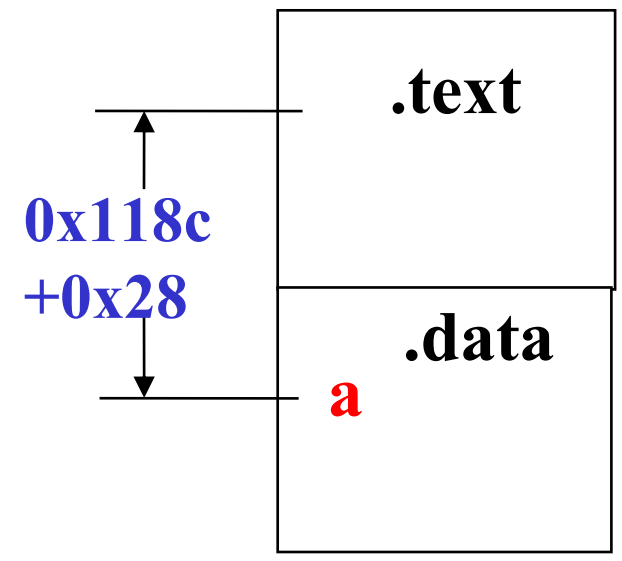
(2) 模块内部数据引用

- .data节与.text节之间的相对位置确定，任何引用局部符号的指令与该符号之间的距离是一个常数

```
static   int a;
extern int b;
extern  void ext();

void bar()
{
    a=1;
    b=2;
}
…….
```

```
0000344 <bar>:
 0000344:    55                          pushl  %ebp
 0000345:    89 e5                     movl   %esp, %ebp
 0000347:    e8 50 00 00 00      call    39c <__get_pc>
 000034c:    81 c1 8c 11 00 00 addl  $0x118c, %ecx
 0000352:   c7 81 28 00 00 00 movl $0x1, 0x28(%ecx)
 ……
 0000362:    c3                          ret

000039c <__get_pc>:
 000039c:     8b 0c 24                  movl  (%esp), %ecx
 000039f:     c3                    ret
```

多用了4条指令



变量a与引用a的指令之间的距离为常数，调用__get_pc后，call指令的返回地址被置ECX。若模块被加载到0x9000000，则a的访问地址为：

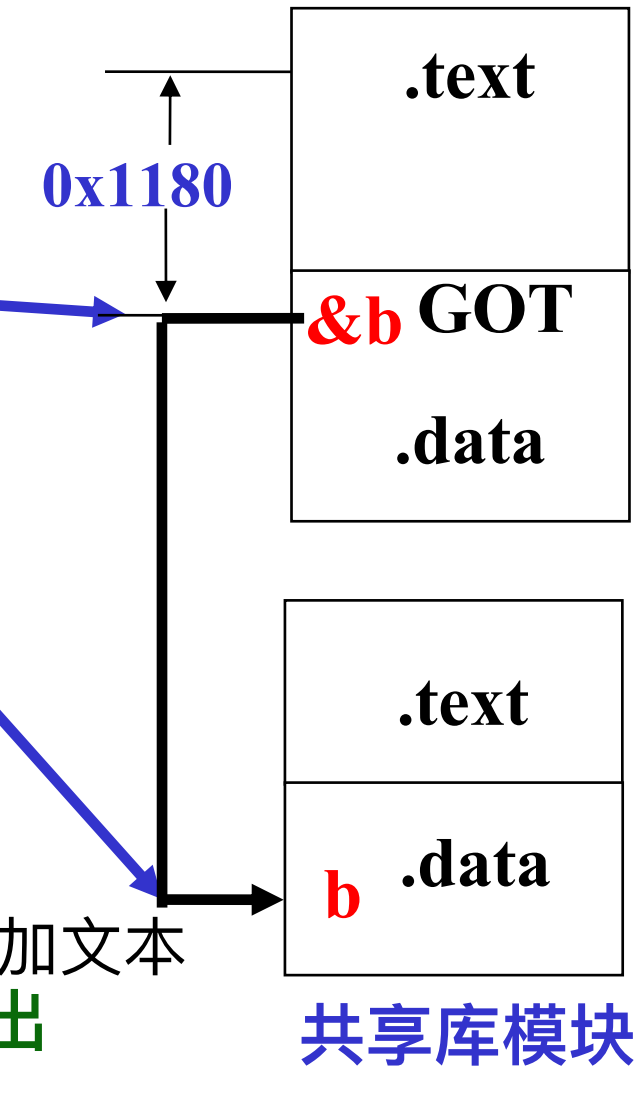
0x9000000+0x34c+0x118c(指令与.data间距离)+0x28(a在.data节中偏移)

(3) 模块外数据的引用

- 引用其他模块的全局变量，无法确定相对距离
- 在.data节开始处设置一个指针数组（全局偏移表，GOT），指针可指向一个全局变量
- GOT与引用数据的指令之间相对距离固定

```
00000344 <bar>:
 00000344:   55              pushl  %ebp
 ……
 00000357:   e8 00 00 00 00   call   0000035c
 0000035c:   5b               popl  %ebx
 000035d:                     addl  $1180, %ebx
 ……                           movl  (%ebx), %eax
 ……                         movl  $2, (%eax)
 ……
```

```
static   int a;
extern int b;
extern  void ext();

void bar()
{
    a=1;
    b=2;
}
……
```



- 编译器为GOT每一项生成一个重定位项（如.rel节…）
- 加载时，动态链接器对GOT中各项进行重定位，填入所引用的地址（如&b）

PIC有两个缺陷：多用4条指令；多了GOT（Global Offset Table），故需多用一个寄存器（如EBX），易造成寄存器溢出

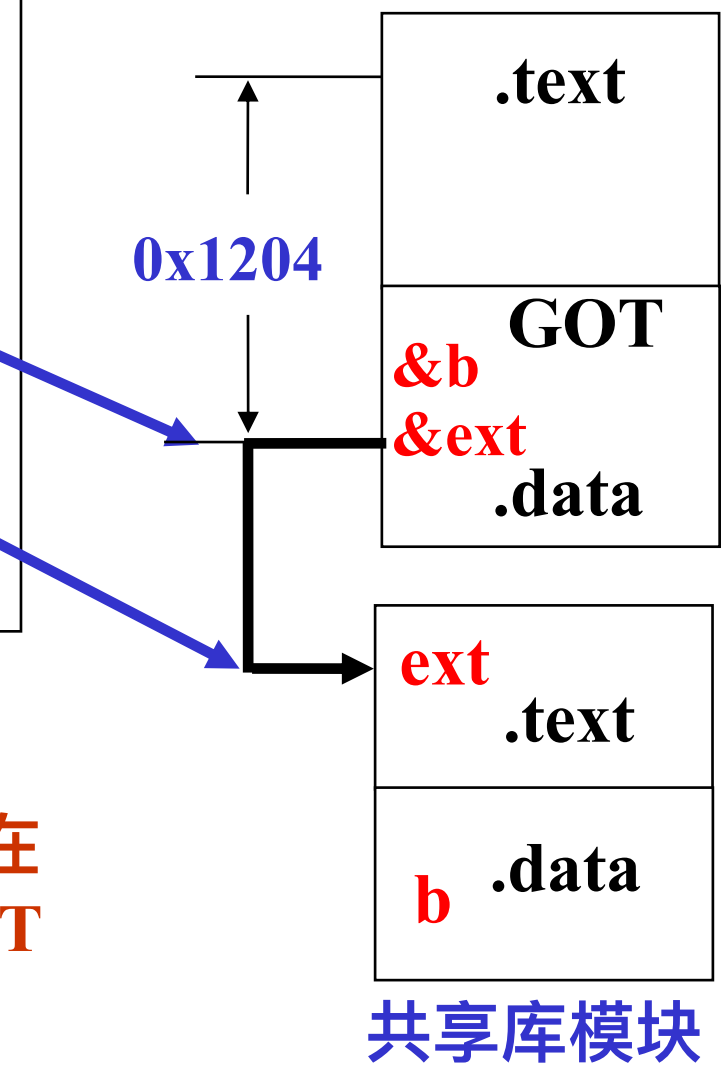
双击此处添加文本

(4) 模块间调用、跳转

- 方法一：类似于(3)，在GOT中加一个项(指针)，用于指向目标函数的首地址（如&ext）
- 动态加载时，填入目标函数的首地址

```
static   int a;
extern int b;
extern  void ext();

void foo()
{
    bar();
    ext();
}
……
```

```
0000050c <foo>:
  0000050c:    55                pushl  %ebp
  ……
  00000557:    e8 00 00 00 00    call   0000055c
  0000055c:    5b                popl  %ebx
  0000055d:                      addl  $1204, %ebx
  ……                             call   *(%ebx)
  ……
        *(%ebx)为间接地址：R[eip]←M[R[ebx]]
```



- 多用三条指令并额外多用一个寄存器（如EBX）

可用“延迟绑定（lazy binding）”技术减少指令条数：不在加载时重定位，而延迟到第一次函数调用时，需要用GOT和PLT（Procedure linkage Table, 过程链接表）

(4) 模块间调用、跳转

```
extern  void ext();
void foo() {
    bar();
    ext();
}
……
```

方法二：延迟绑定

GOT是.data节一部分，开始三项固定，含义如下：

GOT[0]为.dynamic节首址，该节中包含动态链接器所需要的基本信息，如符号表位置、重定位表位置等；

GOT[1]为动态链接器的标识信息

GOT[2]为动态链接器延迟绑定代码的入口地址

调用的共享库函数都有GOT项，如GOT[3]对应ext

延时绑定代码根据GOT[1]和ID确定ext地址填入GOT[3]，并转ext执行，以后调用ext，只要多执行一条jmp指令而不是多3条指令。

PLT是.text节一部分，结构数组，每项16B，除PLT[0]外，其余项各对应一个共享库函数，如PLT[1]对应ext

PLT[0]

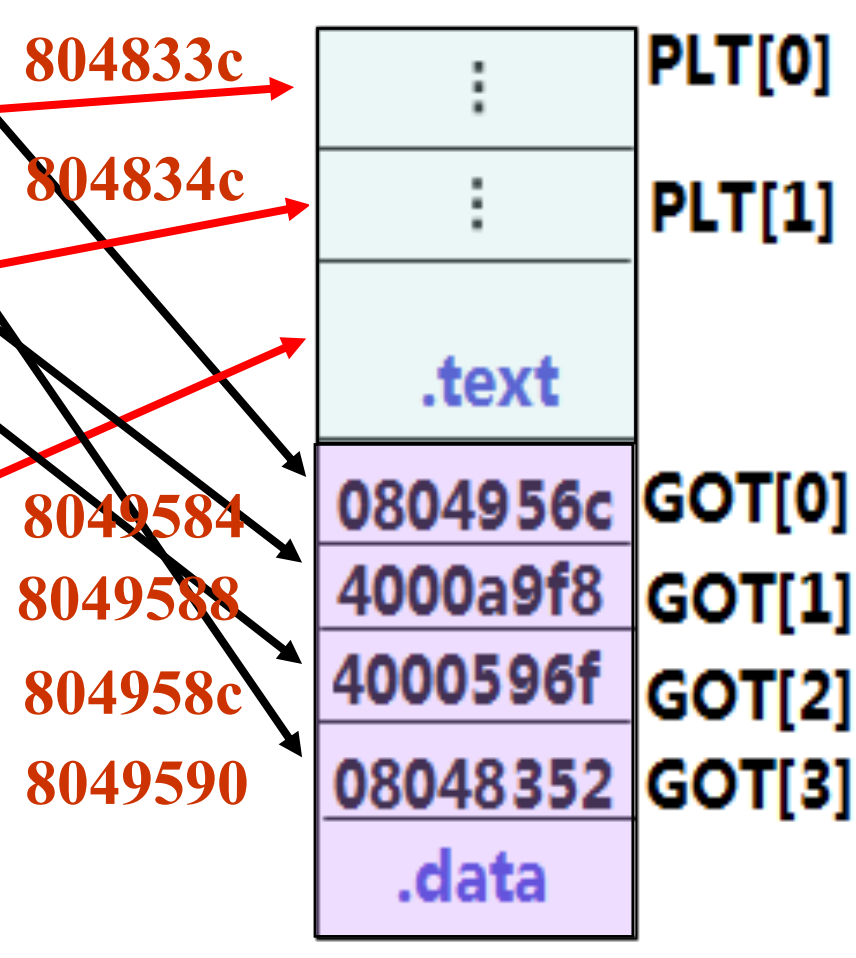
```
0804833c:  ff 35 88 95 04 08    pushl  0x8049588
 8048342:  ff 25 8c 95 04 08     jmp    *0x804958c
 8048348:  00 00 00 00
```

PLT[1] <ext>　　用 ID=0 标识ext()函数

```
0804834c:  ff 25 90 95 04 08    jmp  *0x8049590
 8048352:  68 00 00 00 00        pushl  $0x0
 8048357:  e9 e0  ff  ff  ff     jmp    804833c
```

ext()的调用指令：

```
804845b:  e8 ec fe ff ff    call  804834c <ext>
```



本章小结

- 链接处理涉及到三种目标文件格式：可重定位目标文件、可执行目标文件和共享目标文件。共享库文件是一种特殊的可重定位目标。
- ELF目标文件格式有链接视图和执行视图两种，前者是可重定位目标格式，后者是可执行目标格式。
  - 链接视图中包含ELF头、各个节以及节头表
  - 执行视图中包含ELF头、程序头表（段头表）以及各种节组成的段
- 链接分为静态链接和动态链接两种
  - 静态链接将多个可重定位目标模块中相同类型的节合并起来，以生成完全链接的可执行目标文件，其中所有符号的引用都是在虚拟地址空间中确定的最终地址，因而可以直接被加载执行。
  - 动态链接的可执行目标文件是部分链接的，还有一部分符号的引用地址没有确定，需要利用共享库中定义的符号进行重定位，因而需要由动态链接器来加载共享库并重定位可执行文件中部分符号的引用。
    - 加载时进行共享库的动态链接
    - 执行时进行共享库的动态链接

本章小结

- 链接过程需要完成符号解析和重定位两方面的工作
  - 符号解析的目的就是将符号的引用与符号的定义关联起来
  - 重定位的目的是分别合并代码和数据，并根据代码和数据在虚拟地址空间中的位置，确定每个符号的最终存储地址，然后根据符号的确切地址来修改符号的引用处的地址。
- 在不同目标模块中可能会定义相同符号，因为相同的多个符号只能分配一个地址，因而链接器需要确定以哪个符号为准。
- 编译器通过对定义符号标识其为强符号还是弱符号，由链接器根据一套规则来确定多重定义符号中哪个是唯一的定义符号，如果不了解这些规则，则可能无法理解程序执行的有些结果。
- 加载器在加载可执行目标文件时，实际上只是把可执行目标文件中的只读代码段和可读写数据段通过页表映射到了虚拟地址空间中确定的位置，并没有真正把代码和数据从磁盘装入主存。

本章作业

- 5、7、8、9、10

- 5月11号交本子