

第四章 程序的链接

2019.4

王多强

群名称：ICS2019

群 号：1015148119



群名称：ICS2019

群 号：1015148119

一个典型的程序转换处理过程

经典的 “hello.c” C-源程序

```
#include <stdio.h>

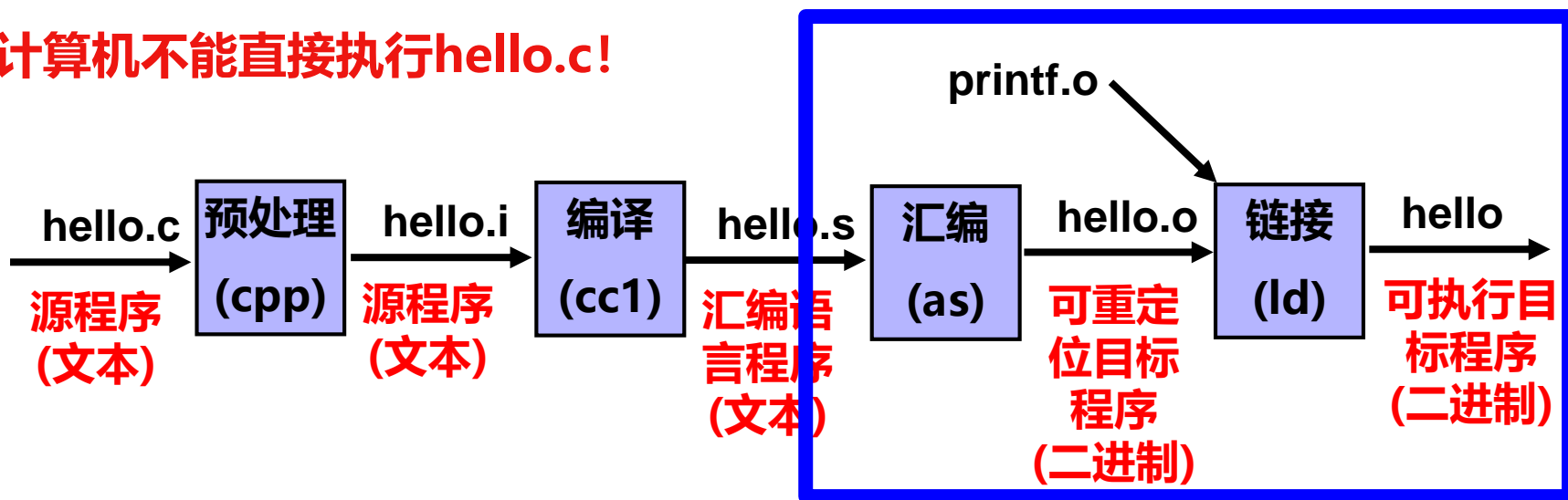
int main()
{
    printf("hello, world\n");
}
```

hello.c的ASCII文本表示

```
# i n c l u d e < s p > < 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 < s p > m a i n ( ) \n {
104 62 10 10 105 110 116 32 109 97 105 110 40 41 10 123
\n < s p > < s p > < s p > < s p > 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 , < s p > 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!



-
- 本章介绍**链接器**的工作原理
 - 链接概念、目标文件格式
 - 符号解析、地址重定位
 - 可执行文件的存储器映像和加载
 - 共享（动态）库链接

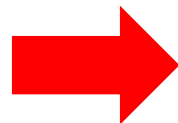
- ◆ 一方面，在**可执行目标程序的统一虚拟地址空间**中，每条指令、每个数据都有一个从**绝对位置0**开始的**统一编制的唯一的地址**。
- ◆ 另一方面，程序是分模块开发的：**多个.c文件**，每个可以单独编译而得到**多个.o文件**。每个.o文件里，有本模块的指令和数据，它们也有一个地址，但这个地址仅是从本模块的开始位置——**相对位置0**处开始编制的**字节偏移**。不同模块的.o文件里，会有相同的相对地址。

怎么把这些模块整合在一起，重新编排所有的指令和数据的位置，然后在统一虚拟地址空间中重新编址而得到全局统一地址呢？——**链接器的工作！**

```
file1.c
main()
{
    .....
}
```

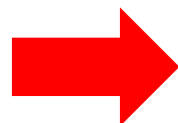
```
file2.c
myfunc()
{
    .....
}
```

源程序



单独编译、
汇编

```
00: .....
08: .....
0f: .....
.....
```



```
00: .....
08: .....
ef: .....
.....
```

相对编址的可重
定位目标文件



统一链接

```
00: .....
08: .....
bf: .....
.....
.....
```

绝对编址的可
执行目标文件

链接：

为了生成一个可执行文件，需要将所有关联到的目标代码文件，包括用到的**标准库函数目标文件**，按照某种形式组合在一起，形成一个具有**统一地址空间**、可被加载到存储器直接执行的程序。

——这种**将一个程序的所有关联模块对应的目标代码文件结合在一起，以形成一个可执行文件的过程**称为**链接**。

➤ 早期计算机系统中，链接靠手工完成

➤ 计算机系统中，链接由专门的链接程序（linker，**链接器**）完成。

使用链接的好处

● 模块化开发

- (1) 一个程序可以分成多个源程序文件，分别进行编辑、编译
- (2) 可构建共享函数库，如数学库，标准C库等

● 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执行代码，而不需要包含整个共享库

一个C语言程序举例， 包含两个模块

main.c

```
int buf[2] = {1, 2};  
void swap();  
  
int main( )  
{  
    swap( );  
    return 0;  
}
```

功能： 交换数组中的两个元素

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;  
}
```


- 使用**GCC**编译器编译并链接生成可执行程序p:

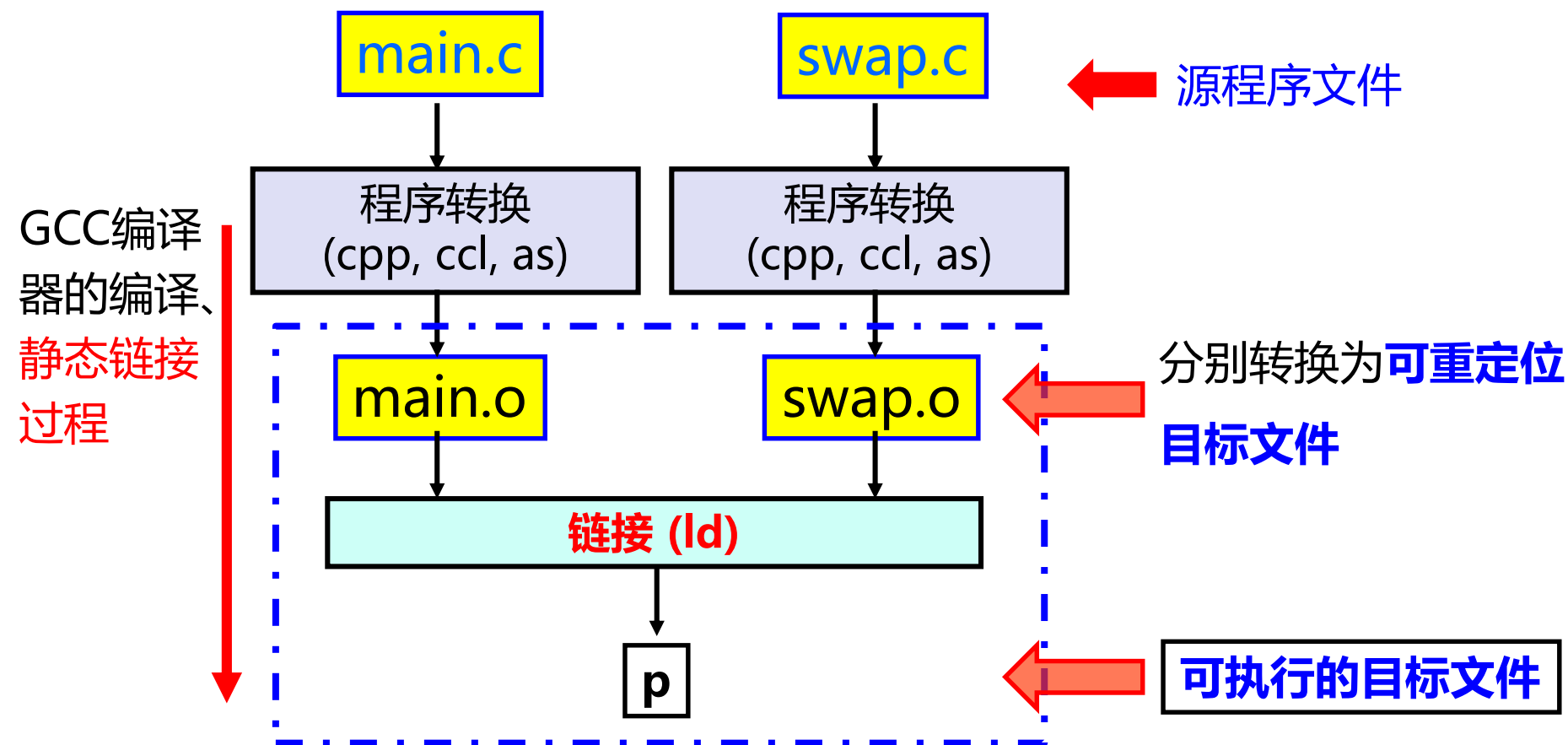
– \$ gcc -O2 -g -o **p** **main.c** **swap.c**

– \$./p

-O2: 2级优化

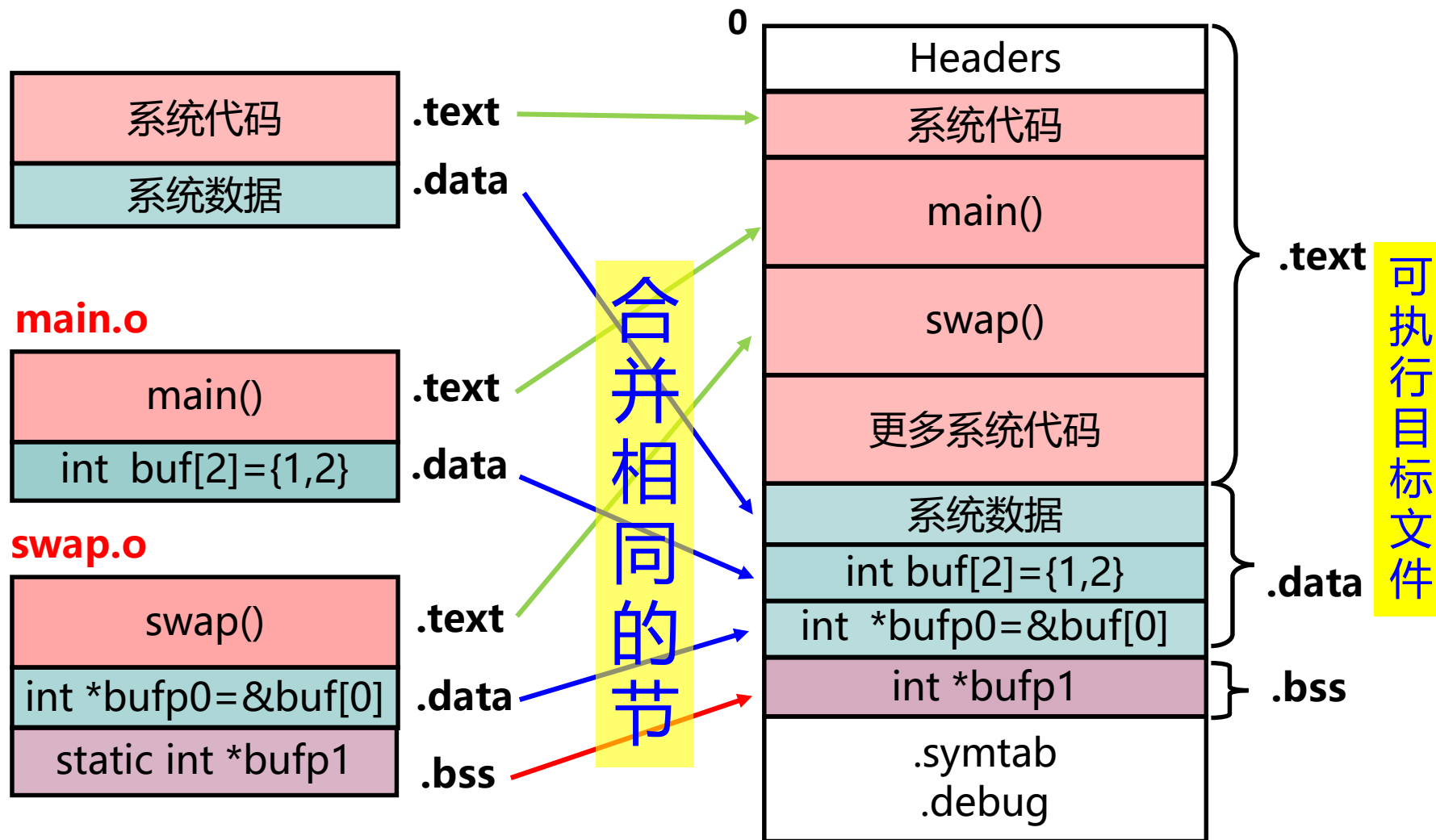
-g: 生成调试信息

-o: 目标 (输出) 文件名



链接过程的本质：

- 将可重定位目标文件中相同的**节合并**，形成具有**统一虚拟地址空间编址**的可执行目标代码文件。



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;  
}
```

存储映像加载地址:

0x08048000

相对地址

```
00000000  
00000000:  
00000001:  
00000003:  
00000006:  
00000009:  
0000000C:  
0000000F:  
00000012:  
00000015:  
00000016:
```

```
<add>:  
55  
89 e5  
83 ec 10  
8b 45 0c  
8b 55 08  
8d 04 02  
89 45 fc  
8b 45 fc  
c9  
c3
```

objdump -d test.o

```
push %ebp  
mov %esp, %ebp  
sub $0x10, %esp  
mov 0xc(%ebp), %eax  
mov 0x8(%ebp), %edx  
lea (%edx,%eax,1), %eax  
mov %eax, -0x4(%ebp)  
mov -0x4(%ebp), %eax  
leave  
ret
```

绝对地址

```
080483d4  
80483d4:  
80483d5:  
80483d7:  
80483da:  
80483dd:  
80483e0:  
80483e3:  
80483e6:  
80483e9:  
80483ea:
```

```
<add>:  
55  
89 e5  
83 ec 10  
8b 45 0c  
8b 55 08  
8d 04 02  
89 45 fc  
8b 45 fc  
c9  
c3
```

objdump -d test

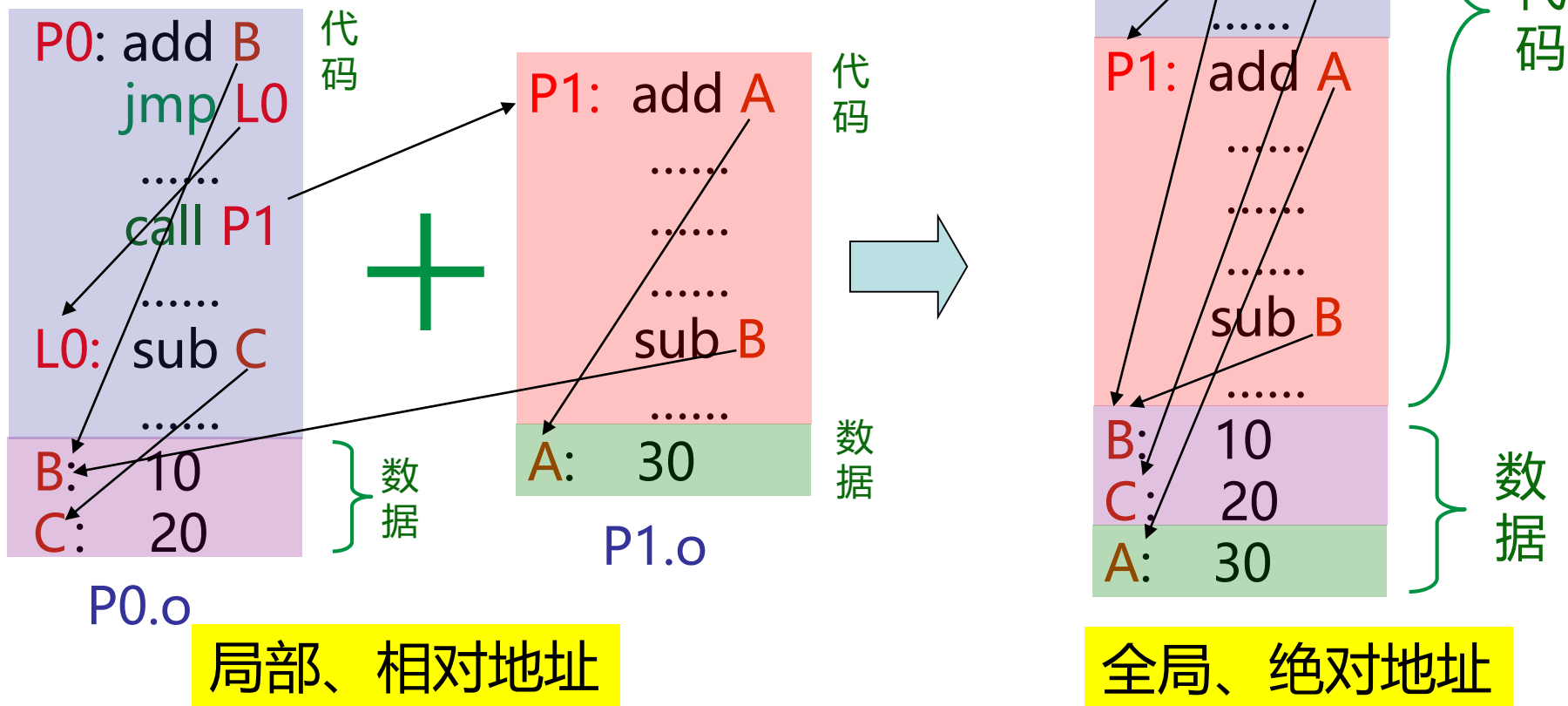
```
push %ebp  
mov %esp, %ebp  
sub $0x10, %esp  
mov 0xc(%ebp), %eax  
mov 0x8(%ebp), %edx  
lea (%edx,%eax,1), %eax  
mov %eax, -0x4(%ebp)  
mov -0x4(%ebp), %eax  
leave  
ret
```

链接操作的步骤

- 1) 确定符号的定义和引用
- 2) 合并相关.o文件, 确定每个符号的统一地址, 并在指令中填入新地址

➡ 符号解析

➡ 地址重定位



链接步骤

• Step 1. 符号解析 (Symbol resolution)

- 符号：程序中的名字，包括变量和函数等
 - void **swap**() {...} /* **定义符号** */
 - **swap**(); /* **引用符号** */
 - int ***xp** = &**x**; /* 定义符号 xp, 引用符号 x */
- 编译器将所有定义的符号存放在一个**符号表** (symbol table) 中。
 - 每个表项包含一个**定义符号**的符号名、长度和位置等信息
- 链接器将每个**引用符号**都与其确定的**定义符号**建立关联，从而可以确定符号地址

add B
jmp L0
.....
.....
.....
L0: sub C
.....

• Step 2. 重定位

- 在将多个可重定位目标文件中的代码段与数据段**合并**为可执行目标文件中的全局统一的代码段和数据段后，计算每个定义符号在可执行目标文件全局虚拟地址空间中的绝对地址。然后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。

一、目标文件的格式

- **目标文件 (Object File)**

- 源代码经**编译和汇编**后所生成的**机器语言代码**称为**目标代码**。
- 存放目标代码的文件称为**目标文件**。

- **有三类目标文件：**

- **可重定位目标文件 (.o)：** 不能执行，需被链接

- 其代码和数据可和其他可重定位文件合并为可执行文件。
- .o文件中的代码和数据的地址都是相对自身0的相对地址。

- **可执行目标文件 (默认为a.out)：** 可以被直接执行

- 其代码和数据可以被直接复制到内存并被执行。
- 代码和数据的地址为统一虚拟地址空间中的绝对地址。

- **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**）

目标文件格式：目标文件的结构规范

- 早期，不同的计算机各有自己的独特格式，非标准的，不兼容
- 现在，几种标准的目标文件格式：**COM**、**COFF**、**PE**、**ELF**等格式
 - **COM格式**：**DOS操作系统使用**，简单，文件中仅包含代码和数据，且被加载到固定位置。
 - **COFF格式**：**System V UNIX早期版本使用**，由一组严格定义的数据结构序列组成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，。
 - **PE格式**：**Windows使用**，COFF的变种，称为**可移植可执行格式**（Portable Executable, 简称PE）
 - **ELF格式**：COFF文件格式的另一变种，称为**可执行可链接格式**（**Executable and Linkable Format**, 简称**ELF**）。**Linux系统中使用。**

ELF目标文件格式

- ELF目标文件格式有两种视图
 - 针对可链接目标文件的**链接视图**
 - 针对可执行目标文件的**执行视图**

1) 链接视图

◆ 可链接目标文件由不同的**节**组成。

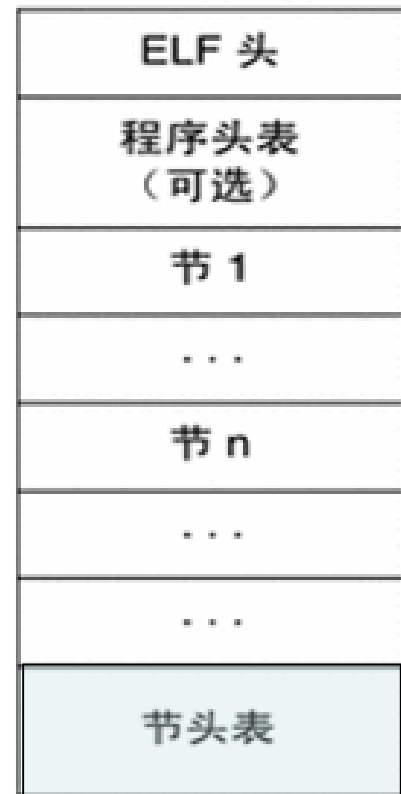
◆ **节**: **section**, 是 ELF 文件中具有相同特征的最小可处理单位。

如: **.text**节: 代码节

.data节: 已初始化的全局数据节

.rodata: 只读数据

.bss: 未初始化的全局数据节



链接视图

不同的节描述了目标文件中不同类型的信息及其特征。

2) 执行视图

➤ 可执行目标文件的由不同的**段**组成

描述节是如何映射到存储空间的段中的。

➤ **段**: **segment**, 可执行目标文件**存储映像**中不同性质的一块区域。

如只读代码段、读写数据段等。

如：合并.data节和.bss节，映射到一个**可读可写数据段**中



执行视图

1. 可重定位目标文件格式（ELF格式）

– 可重定位目标文件由不同的**节**组成：

- ① 代码节：**.text**
- ② 已初始化数据节：**.data**
- ③ 未初始化数据节：**.bss**
- ④ 符号表：**.symtab**
- ⑤ 代码重定位信息：**.rel.txt**
- ⑥ 数据重定位信息：**.rel.dat**等。

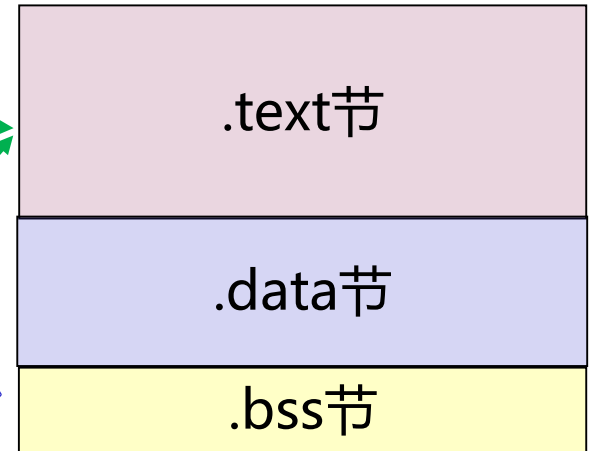
注：Linux中可重定位目标文件的扩展名为.o

Windows中可重定位目标文件的扩展名为.obj

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

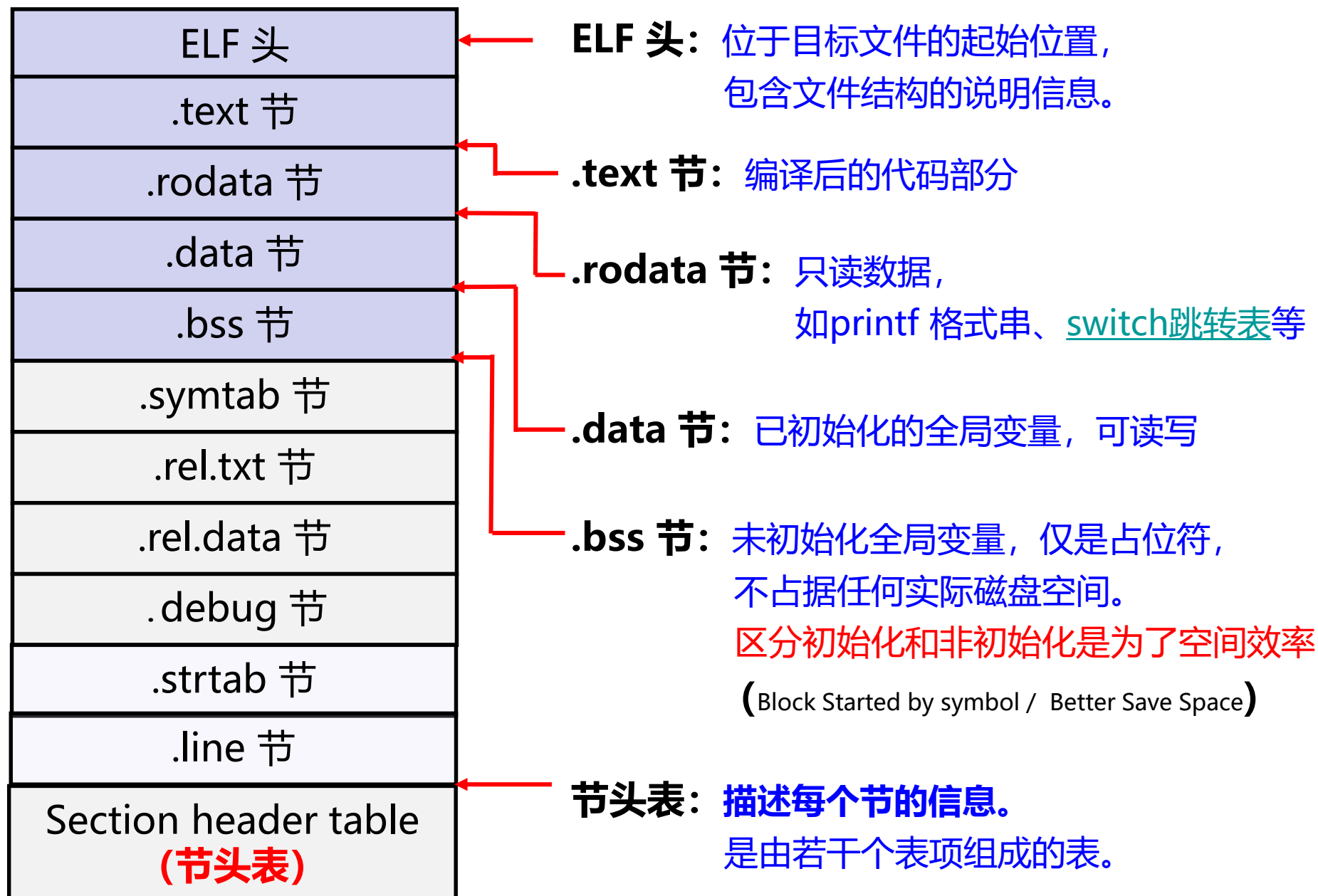
void main( )
{
    static int a=1;
    static int b;
    int i=200,j;
    prn(x+a+i);
}
```

ELF的链接视图



指令映射到代码节
数据映射到数据节

ELF可重定位目标文件格式



1) ELF头 (ELF Header)

描述了本目标文件的一些基本信息，如

- ① **类型：ELF32/64** 有32位ELF文件格式和64位ELF文件格式
- ② 数据存放是大端还是小端
- ③ ELF头的版本、大小
- ④ 目标文件的类型
- ⑤ 机器结构类型
- ⑥ **控制权转移的起始虚拟地址**
- ⑦ 程序头表基本信息：位置、表项数量、表项大小
- ⑧ **节头表基本信息**：位置、表项数量、表项大小等

1) ELF头 (ELF Header)

```
#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;
```

魔数：文件开头的几个字节，用来确定文件类型、格式。
加载或读取文件时，可用魔数确认文件类型是否正确。

16字节序列，ELF魔数 (7f 45 4c 46) + 标识信息，如32/64位格式、数据存放是大端还是小端、ELF头的版本号等

目标文件的类型：可重定位文件/可执行文件/共享库文件/其他类型文件等

指定机器结构类型：IA-32、SPARC V9、AMD64

标识目标文件版本

指定系统将控制权转移的起始虚拟地址（入口点），若没有关联的入口点，则置0，如可重定位文件，此字段为0

程序头表在文件中的偏移量（以字节为单位）

节头表在文件中的偏移量（以字节为单位）

处理器特定标志，对IA32而言，此项为0。

说明ELF文件头的大小（以字节为单位）

程序头表中一个表项的大小（以字节为单位），所有表项大小相同

程序头表中表项的数量。程序头表大小 = $e_phentsize * e_phnum$

节头表中一个表项的大小（以字节为单位），所有表项大小相同

节头表中表项的数量。节头表大小 = $e_shentsize * e_shnum$

.strtab在节头表中的索引

可重定位目标文件ELF头信息举例

\$ readelf -h main.o

ELF Header: ELF文件的魔数
Magic: 7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
Class: ELF32 二进制补码
Data: 2's complement, little endian
Version: 1 (current) ELF头的版本号
OS/ABI: UNIX - System V
ABI Version: 0
Type: REL (Relocatable file) 可重定位目标文件的ELF头
Machine: Intel 80386
Version: 0x1 目标文件版本
Entry point address: 0x0 程序入口为0 没有程序头表
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

ELF 头
.text 节
.rodata 节
.data 节
.bss 节
.symtab 节
.rel.txt 节
.rel.data 节
.debug 节
.strtab 节
.line 节
Section header (节头表)

2) 节头表 (Section Header Table)

- **描述每个节的基本信息**，如每个节的节名、在目标文件中的偏移、大小、访问属性、对齐方式等、
- **以下是32位系统对应的节头表数据结构（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.rel.text节、.rel.data节、.symtab节等）

```
    Elf32_Word    sh_info;
```

```
    Elf32_Word    sh_addralign;
```

节的对齐要求

```
    Elf32_Word    sh_entsize;
```

节中是表，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:

		起始地址总是0	文件内偏移	大小						
[Nr]	Name	Type	Addr	Off	Size	ES	Fig	Lk	Inf	AI
[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)

ELF 头
.text 节
.data 节
.bss节
.rodata节
.symtab 节
.rel.txt 节
.rel.data 节
.debug 节
.strtab 节
.line 节
Section header (节头表)

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	AI
[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

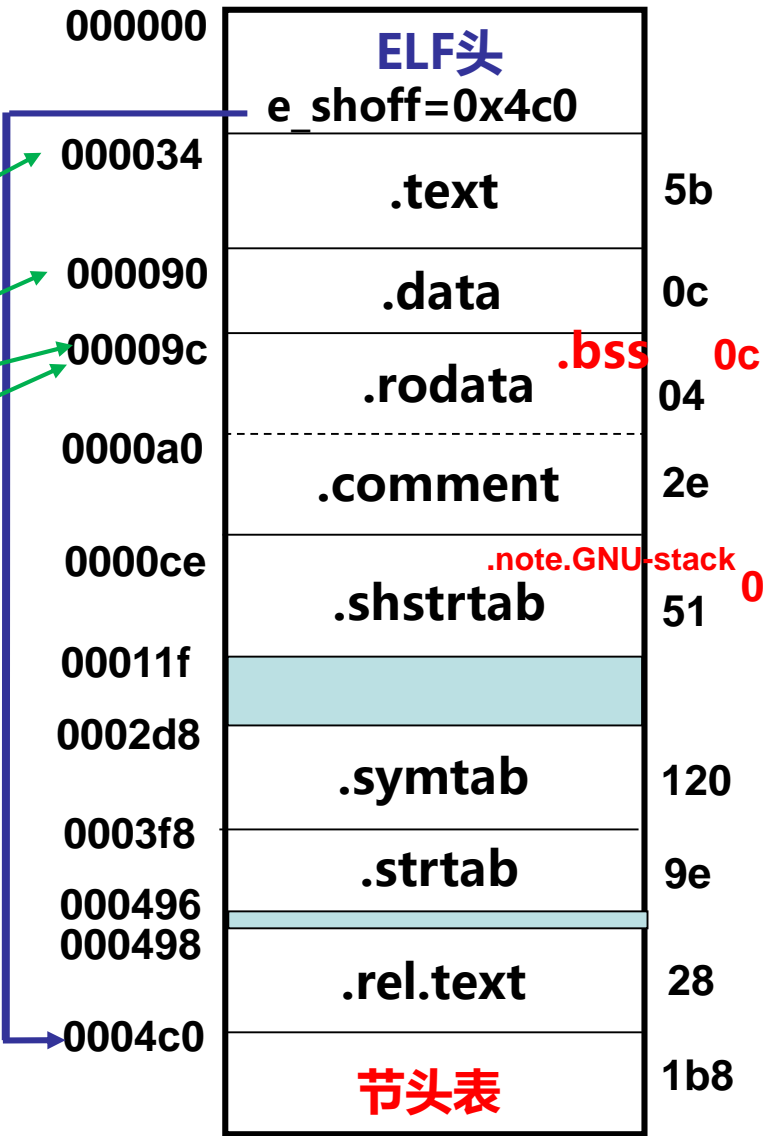
Key to Flags:

W (write), A (alloc), X (execute), M (merge), S (strings)
I (info), L (link order), G (group), x (unknown)

.....

.text: 可执行
.data和.bss: 可读可写
.rodata: 可读
.bss有4个节将会分配存储空间

可重定位目标文件test.o的结构



2. 可执行目标文件

可执行目标文件包含**代码、数据**及其它：

- ◆ 其中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，指向相应的定义符号；
- ◆ 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；

在执行视图中

- ① 需将**具相同访问属性**的节合并成**段**（Segment）。
- ② 段有相应的属性，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。
- ③ **程序头表**用来说明段信息，也称**段头表**（segment header table）。

可执行目标文件格式

与可重定位文件类似，但稍有不同：

- ELF头中字段**e_entry**给出执行程序时第一条指令的地址。

在可重定位文件中，此字段为0；

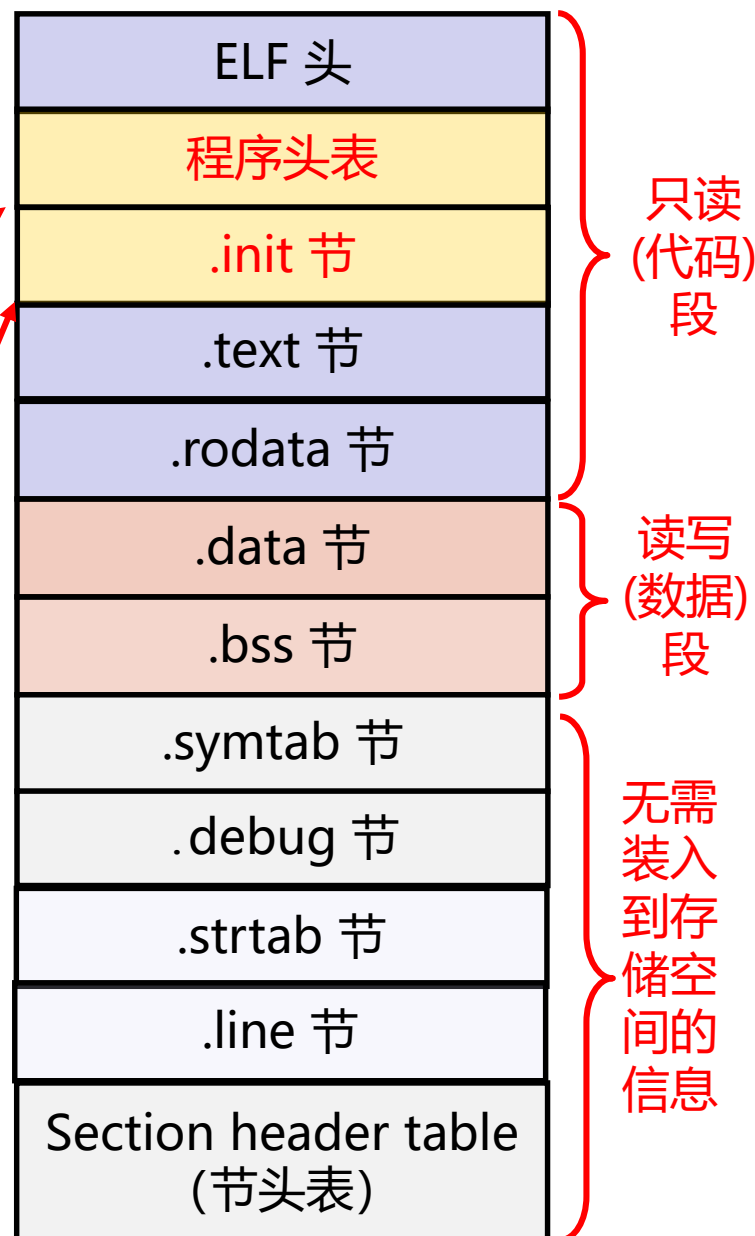
- 多一个**程序头表**（段头表）

用于说明可执行目标文件段的组成。

- 多一个**.init节**，其中定义了一个_init函数。

用于开始执行时的初始化工作；

- 少两个**.rel节**（无需重定位）；



可执行目标文件的ELF头信息举例

\$ readelf -h main

ELF Header: ELF文件的魔数

Magic: 7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00

Class: ELF32 二进制补码

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: 0x8048580 第一条指令的地址

Start of program headers: 52 (bytes into file)

Start of section headers: 3232 (bytes into file)

Flags: 0x0

Size of this header: 52 (bytes)

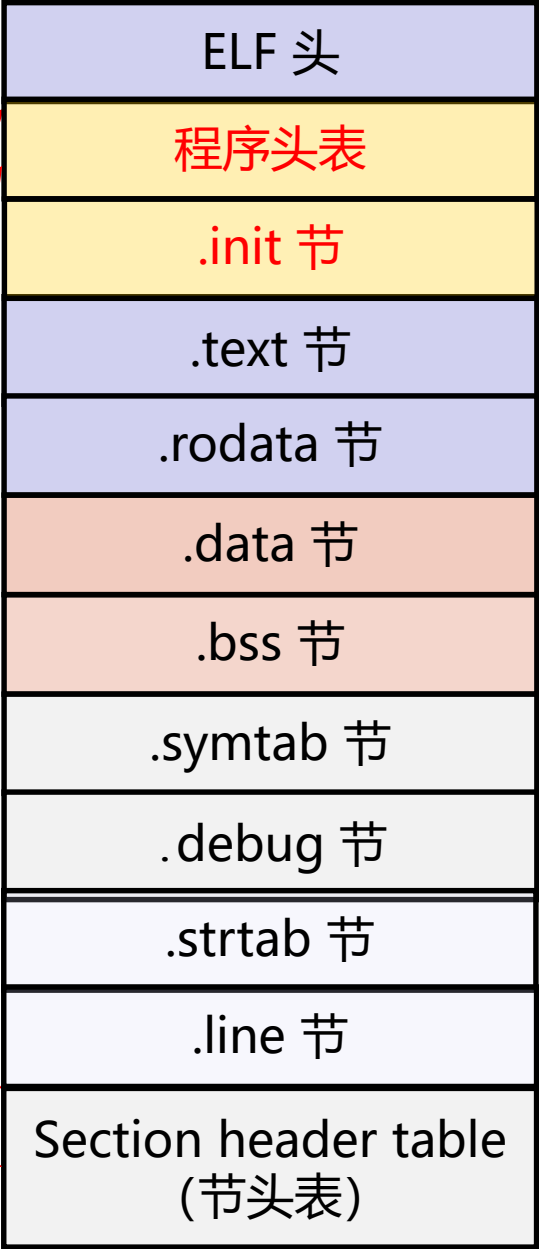
Size of program headers: 32 (bytes)

Number of program headers: 8

Size of section headers: 40 (bytes)

Number of section headers: 29

Section header string table index: 26



可执行文件程序头表

- **程序头表**：一个结构数组，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。
- 每个表项说明虚拟地址空间中的一个连续的段（或一个特殊的节）

程序头表的数据结构

```
typedef struct {
```

```
    Elf32_Word    p_type;
```

描述存储段的类型或特殊节类型，如：是否为可装入段(PT_LOAD)、是否是特殊的动态节(PT_DYNAMIC)、是否是特殊的解释程序节(PT_INTERP)

```
    Elf32_Off     p_offset;
```

指出本段的首字节在文件中的偏移地址

```
    Elf32_Addr    p_vaddr;
```

指出本段的首字节在虚拟地址空间中的虚拟地址

```
    Elf32_Addr    p_paddr;
```

指出本段的首字节的物理地址，但因为物理地址只有在运行时由操作系统动态确定，所以该信息通常无效。

```
    Elf32_Word    p_filesz;
```

指出本段在文件中所占的字节数，可以为0

```
    Elf32_Word    p_memsz;
```

指出本段在**存储器**中所占的字节数，可以为0

```
    Elf32_Word    p_flags;
```

指出存取权限

```
    Elf32_Word    p_align;
```

指出段的对齐方式，用模数表示，为2的正整数幂，通常模数与页面大小相关，若页面大小为4KB，这模数为 2^{12} 。

```
} Elf32_Phdr;
```

以下是某可执行目标文件程序头表信息

\$ 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

解释程序节

动态链接器的路径名

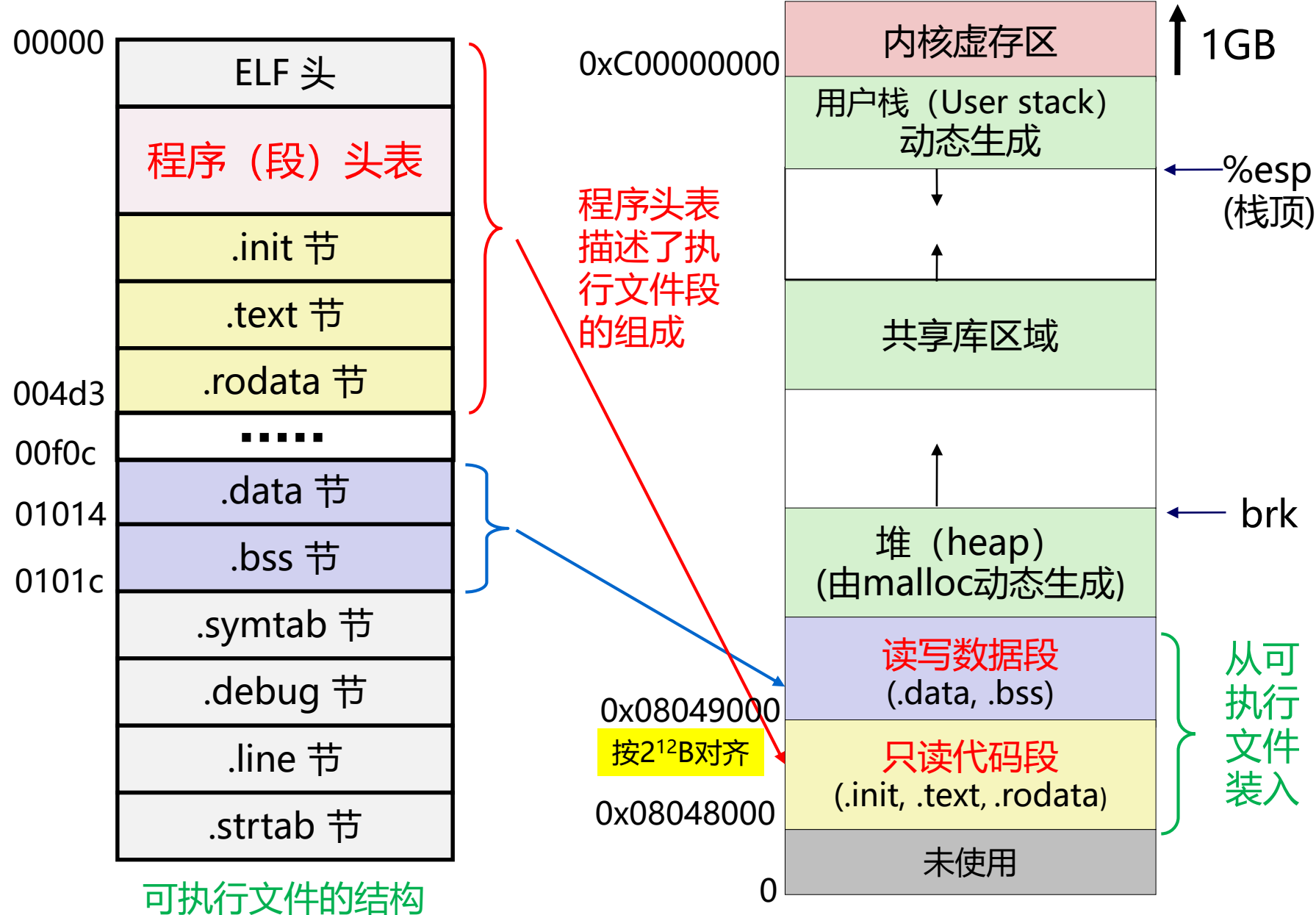
有8个表项，其中两个为可装入段（即Type=LOAD）

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 = 2¹² = 4KB对齐，具有只读/执行权限（Flg=RE），是只读代码段。

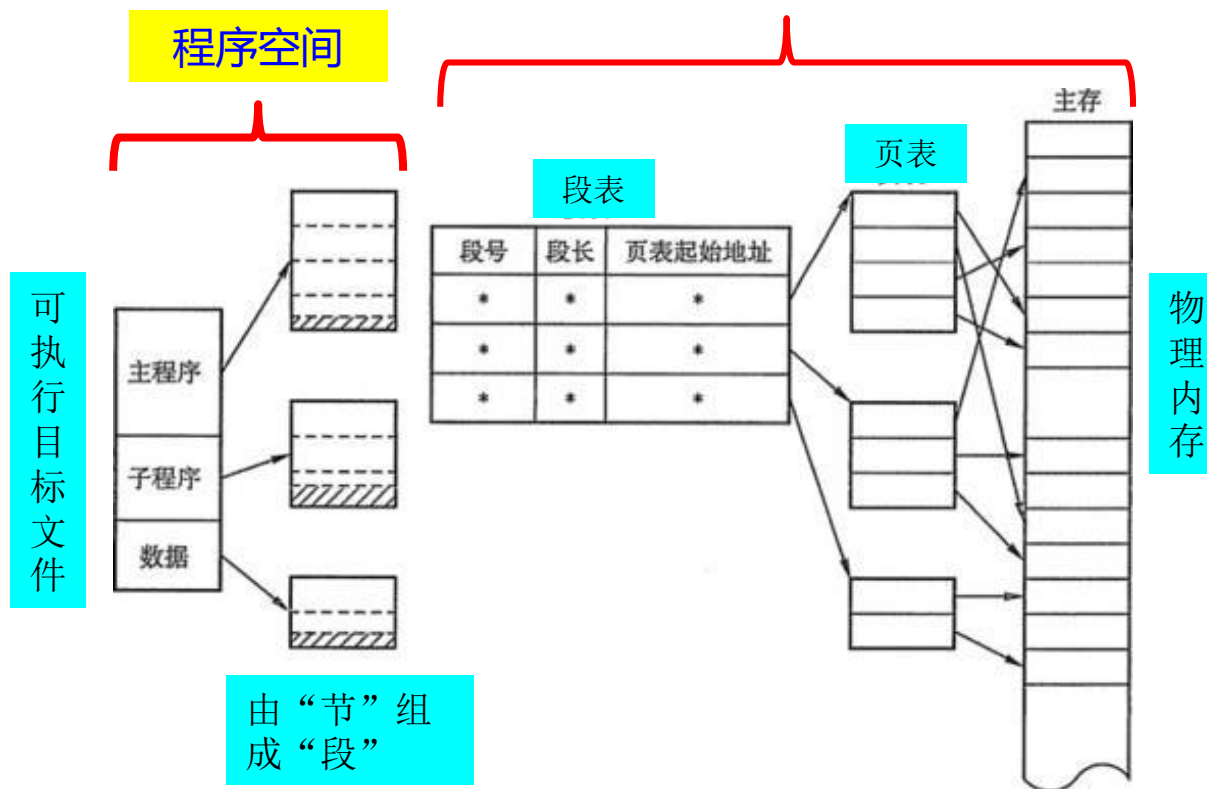
第二可装入段：从0x000f0c开始长度为0x108字节的.data节，映射到虚拟地址0x8049f0c开始长度为0x110字节的存储区域，在0x110=272B存储区中，前0x108=264B用.data节内容初始化，后面272-264=8B对应.bss节，初始化为0，按0x1000=4KB对齐，具有可读可写权限（Flg=RW），是可读写数据段。

程序(段)头表描述如何映射

每个程序都有一个独立的统一逻辑地址空间



段页式存储管理



- ◆ 段页式存储管理方式先将用户程序分成若干个段，再把每个段分成若干个页（与物理内存块相对应），并为每一个段赋予一个段名。
- ◆ 段页式系统中，每个地址结构由**段号**、**段内页号**及**页内地址**三部分所组成，通过换算，**实现虚拟地址到物理地址的映射**。

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

二、符号和符号解析

- 符号包括：函数名、变量名
- 可重定位目标模块的符号表包含了模块中定义的所有符号的相关信息。
- 链接的第一步：符号解析

有三种类型的符号：

1) **Global symbols**:

- 指本模块内部定义的**全局符号**，**能被其他模块引用**。

例如，非static 函数和非static的全局变量

2) **Local symbols**:

- 本模块的**局部符号**，**仅由本模块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

3) **External symbols**:

- 指外部定义的**全局符号**，**由其他模块定义并被本模块引用**
的全局符号

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;  
}
```

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？

注：局部符号不是指程序中的局部变量（如temp）。

局部变量是分配在栈中的临时性变量,链接器不关心这种局部变量。

1. 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

符号表每个条目的结构如下：

```
typedef struct {
```

```
    Elf32_Word  st_name;
```

给出符号在字符串表(.strtab节)中的索引（字节偏移量），指向在字符串表中的一个以null结尾的字符串。

```
    Elf32_Addr  st_value;
```

给出符号的位置，在可重定位目标文件中，是指符号所在位置相对于所在节起始位值的字节偏移量。在可执行目标文件和共享目标文件中是符号所在的虚拟地址

```
    Elf32_Word  st_size;
```

给出符号所表示对象的字节个数，如果符号是函数名，则指函数所占字节个数；若符号是变量，则指变量所占字节个数；如果符号表示的内容没有大小或大小未知，则值为0

```
    char  type : 4,
```

符号类型：可以是变量(OBJECT)、函数(FUNC)、未指定(NOTYPE)、节(SECTION，用于重定位)

```
        binding : 4;
```

绑定属性：可以是本地(LOCAL，本模块定义，外部其他目标文件不可见)、全局(GLOBAL，对于合并的所有目标文件都可见)、弱(WEAK，弱符号)

```
    unsigned char  reserved;
```

未用

```
    Elf32_Half  st_shndx;
```

指出符号所在节在节头表中的索引。但有三种伪节，在节头表中没有相应的表项，故无法表示索引值，因而用以下特殊的索引值表示：ABS表示该符号不会由于重定位而发生值得改变，即不应该被重定位。UNDEF表示未定义符号，即在本模块引用而在其他模块定义的符号；COMMON表示还未被分配位置的未初始化的变量(.bss)，对于COMMON类型的符号，st_value字段给出的是对齐要求，st_size给出的是最小长度。

```
} Elf_Symbol;
```

函数名在text节中

变量名在data节或bss节中

例：对main.c/swap.c, **readelf -s main.o** 查看main.o中的符号表

1) 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	OBJECT	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, 是全球变量, 占8B;
- **main**是第1节 (.text) 偏移为0的符号, 是全球函数, 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。

2) swap.o中的符号表中最后4个条目 (共11个) **readelf -s swap.o**

Num:	value	Size	Type	Bind	Ot	Ndx	Name
8:	0	4	OBJECT	Global	0	3	bufp0
9:	0	0	NOTYPE	Global	0	UND	buf
10:	0	39	FUNC	Global	0	1	swap
11:	4	4	OBJECT	Local	0	COM	bufp1

buf: 未指定且无定义的全局符号

bufp1是未分配位置且未初始化的本地变量(ndx=COM), 按4B对齐, 至少占4B

2. 符号解析 (Symbol Resolution)

- 符号区分为**定义符号**和**引用符号**

- **定义符号**：在本模块中给出了定义的符号，为其相应分配
存储空间：对函数名而言指其代码所在区；
对变量名而言指其所占的静态数据区。
- **引用符号**：仅使用符号的名字，而不涉及其定义。
 - 但这些符号应在其他位置被定义（本地或外部）。

- **符号解析的目的就是**将每个模块中的引用符号与在某个目标模块中的定义符号建立关联，从而知道它的实际位置(地址)，也就能够访问它们的实际内容：函数体或变量的值。

符号解析：也称**符号绑定**，是将**引用符号**与**定义符号**建立关联的过程。

- 每个定义符号在代码段或数据段中都被分配了存储空间，有相应的地址。
- 有了这种关联，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**。

分两种情况：本地符号和全局符号

- 本地符号仅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。
- 全局符号的解析可能涉及多个模块，故较复杂。
- **这里主要关注全局符号的解析。**

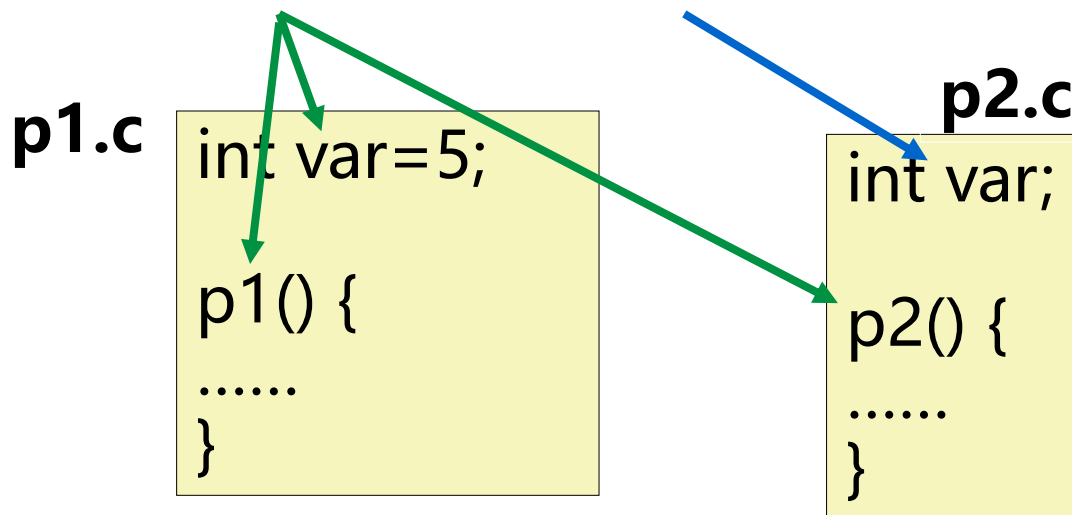
全局符号的符号解析

1. 全局符号的强/弱特性

编译器把符号定义分成**强符号定义**和**弱符号定义**：

- 函数定义时的函数名和已初始化的全局变量名是**强符号定义**。
- 未初始化的全局变量名是**弱符号定义**。
- 声明中出现函数名或全局变量名视为**弱符号**。

如，以下符号哪些是**强符号定义**？ 哪些是**弱符号定义**？



以下定义符号哪些是**强符号定义**？ 哪些是**弱符号定义**？

main.c

```
int buf[2] = {1, 2};  
void swap();  
  
int main()  
{  
    swap();  
    return 0;  
}
```

此处为引用

局部变量

注：局部变量
不在符号表里

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;  
}
```

本地局
部符号

注：本
地符号
没有强
弱之分

2.链接器对符号的解析规则

- 单一定义的符号，只要建立引用符号和定义符号的关联即可。
- 多重定义的符号如何处理？

如：

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

main.c

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

p1.c

- 符号解析时，一个符号不管有多少处定义，最终只能有一个确定的定义（即每个符号仅能对应一处唯一的存储空间）。

链接器对**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，则以强定义为准

- 此时，弱符号被解析为对其强定义符号的引用

Rule 3: 若有多个弱符号定义，则任选其中一个

多重定义符号的解析举例1

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

x有两次强定义，链接器
将输出一条出错信息。

```
----- Build: Debug in test (compiler: GNU GCC Compiler)-----  
mingw32-gcc.exe -Wall -g -c e:\test\f3.c -o obj\Debug\f3.o  
mingw32-g++.exe -o bin\Debug\test.exe obj\Debug\f3.o obj\Debug\test.o  
obj\Debug\test.o:test.c:(.data+0x0): multiple definition of `x'  
obj\Debug\f3.o:f3.c:(.data+0x0): first defined here  
collect2.exe: error: ld returned 1 exit status  
Process terminated with status 1 (0 minute(s), 2 second(s))  
1 error(s), 0 warning(s) (0 minute(s), 2 second(s))
```

多重定义符号的解析举例2

以下程序会发生链接出错吗？

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

```
int y=100;
```

```
int z;
```

```
void p1(void);
```

z的两次定义都是弱定义

强定义的符号y

弱定义的符号y

以main.o
强定义的
符号y为准

```
----- Build: Debug in test (compiler: GNU GCC Compiler)-----  
mingw32-gcc.exe -Wall -g -c e:\test\test.c -o obj\Debug\test.o  
mingw32-g++.exe -o bin\Debug\test.exe obj\Debug\test.o  
Output file is bin\Debug\test.exe with size 29.40 KB  
Process terminated with status 0 (0 minute(s), 1 second(s))  
0 error(s), 0 warning(s) (0 minute(s), 1 second(s))
```

没错！

main.c

p1.c

问题：打印结果是什么？

y=200, z=2000

该例说明：如果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！ 特别是有弱符号时。

多重定义符号的解析举例3

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
5 int
6 {
7 p
8 printf("d=%d,x=%d\n",d,x);
9 re
10 }
```

强定义的符号d

弱定义的符号d

p1.c

```
1 double d;
2
3 void p1()
4 {
```

类型不一，但不会报错，仍以main.o强定义的符号d为准

```
----- Build: Debug in test (compiler: GNU GCC Compiler)-----
mingw32-gcc.exe -Wall -g -c e:\test\test.c -o obj\Debug\test.o
mingw32-g++.exe -o bin\Debug\test.exe obj\Debug\test.o
Output file is bin\Debug\test.exe with size 29.40 KB
Process terminated with status 0 (0 minute(s), 2 second(s))
0 error(s), 0 warning(s) (0 minute(s), 2 second(s))
```

e:\test\bin\Debug\test.exe

d=0, x=1072693248

Process returned 0 (0x0) execution time : 0.658 s

Press any key to continue.

问题：

d=0,x=1 072 693 248

定义符，并有唯一的存储
p1.o里的d被解析
引向main.o中d的
存储位置。但在p1中，d的类型还是double的，赋值按double型处理：8字节。

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 }
```

存储位置在这里

p1执行后d和x的内容是什么?

IA-32是小端方式

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！

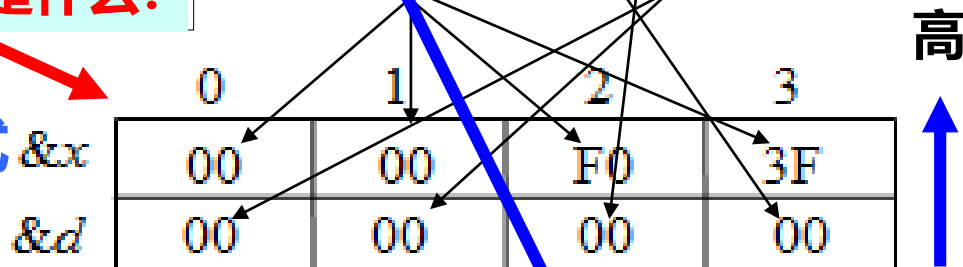
p1.c

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

按double赋值

double型数1.0对应的机器数

3FF0 0000 0000 0000H



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，本地化）
 - 全局变量要赋初值（强定义化）
 - 外部全局变量要使用**extern**（否则就成为弱类型定义符号）
 - 使用命令 `gcc -fno-common` 链接时，当链接器遇到多个弱定义的全局符号时输出一条警告信息，帮助程序员检查程序。
- 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。
- 了解链接器工作原理，**养成良好的编程习惯是非常重要的。**

3.静态共享库

静态库 (.a, archive files)

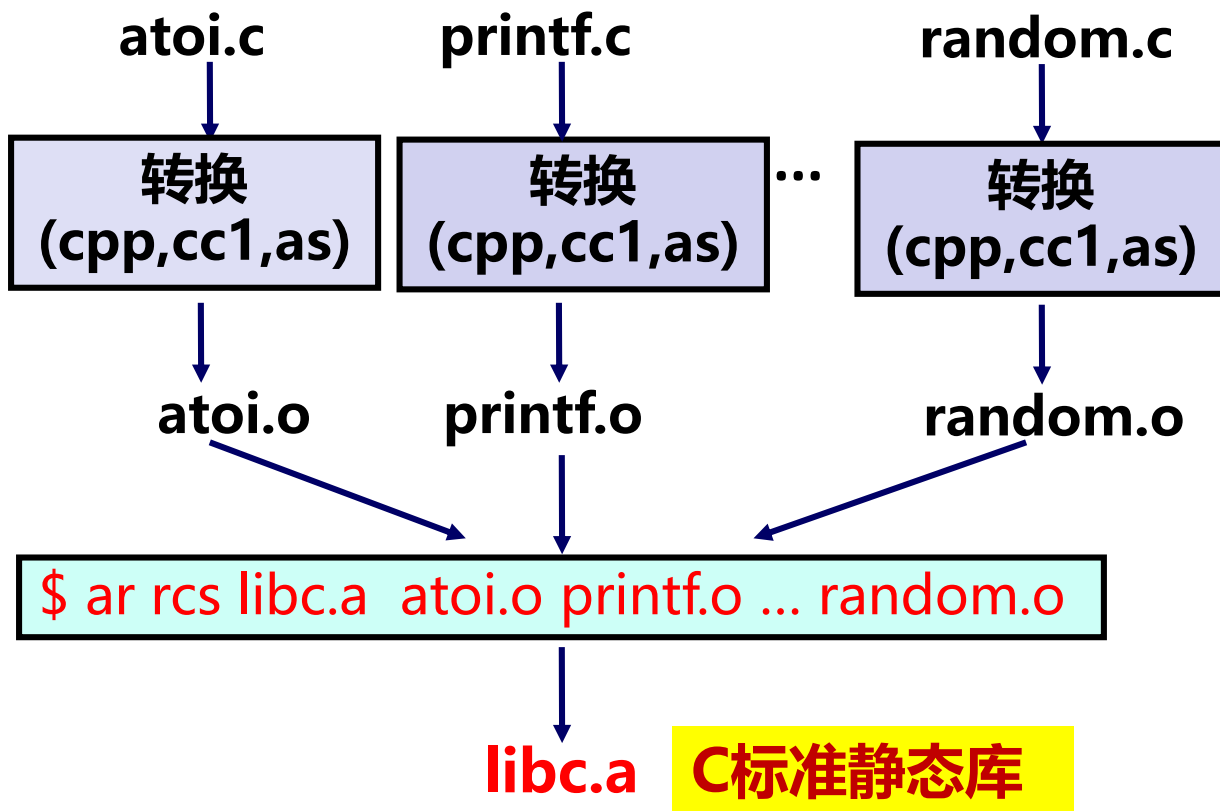
- 可以将一些相关的**目标模块** (.o) 打包为一个称之为**静态库文件**的文件 (.a文件) 也称**存档文件** (archive) 。
- 在构建可执行文件时，如果用到静态库文件中的模块时，只需在链接的时候指定**库文件名**，链接器会自动到库中寻找用到的这些目标模块，并且只把用到的模块从库中拷贝出来，再和其他模块一块链接。

如：

- C的标准函数库文件libc.a，包含了广泛使用的标准I/O函数、字符处理函数和整数处理函数。
- C的数学函数库文件libm.a，包含sin、cos、sqrt等数学函数。

使用AR工具创建静态库。

- 首先，用“ gcc -c ” 命令将目标模块源文件编译成可重定位目标文件 (.o文件)
- 然后，用ar命令打包.o文件生成.a文件



Archiver (归档器)
允许增量更新，只要
重新编译修改过的源
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

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

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/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

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

libc.a是默认用于静态链接的库文件，可以不用在链接命令中显式给出

main.c

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

\$ gcc -c main.c

\$ gcc -static -o myproc main.o ./mylib.a

调用关系：main→myfunc1→printf

问题：如何进行符号解析？

4. 链接器中符号解析的全过程

\$ gcc -c main.c **libc.a**无需明显指出!

\$ gcc -static -o myproc **main.o ./mylib.a**

函数调用关系: main→myfunc1→printf

维护三个集合:

- **E** 将被合并以组成可执行文件的所有目标模块集合
- **U** 当前所有未被解析的引用符号的集合
- **D** 当前所有定义符号的集合

处理步骤:

- 1) 开始E、U、D为空
- 2) 链接器首先扫描main.o, 把它加入E, 同时把main中的未解析符号myfunc1加入U, 而main加入D。同时**将libc.a加入到当前输入文件的末尾。**
- 3) 链接器接着扫描到mylib.a, 并将U中的所有符号 (本例中为myfunc1) 与mylib.a中所有目标模块 (myproc1.o和myproc2.o) 中定义的符号依次匹配。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。

main.c

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

4) 链接器继续在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。同时因为模块myproc2.o没有用到，所以没有被加入E中，被丢弃。

5) 再接着，链接器扫描默认的库文件libc.a。发现其中目标模块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（C中预定义的）加入D，而所有未解析符号加入U（这些未解析的符号都可以在标准库内其他模块中找到，所以）当链接器处理完libc.a时，U一定是空的。

6) 最后，链接器合并E中的目标模块并输出最后生成的可执行目标文件。

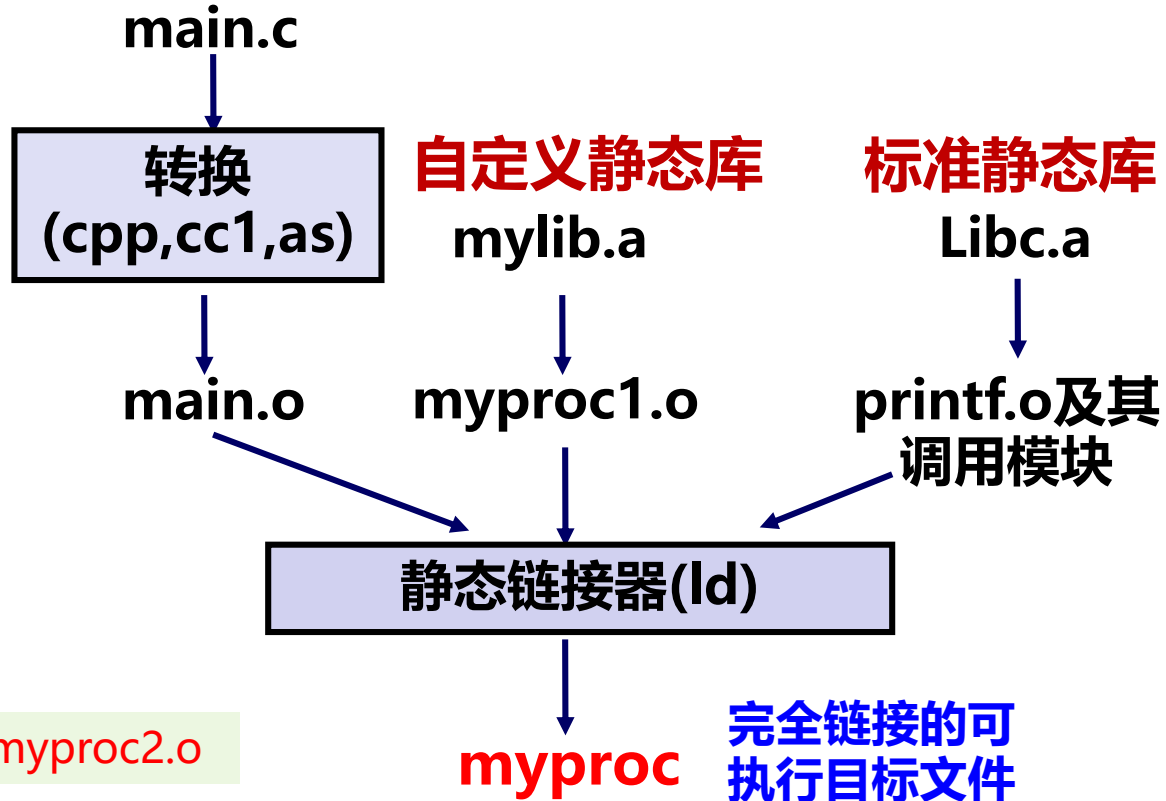
更详细的文字描述见P178

链接器中符号解析的全过程

\$ gcc -static -o myproc main.o ./mylib.a

main.c

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



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

5.27 内容回顾（第十一次课）

符号和符号解析

1. 符号：函数名、变量名

◆ 有三种类型的符号：Global symbols、Local symbols、External symbols

◆ 符号表记录了模块中每个符号的信息（.symtab 节）

符号表每个条目的结构如下：

```
typedef struct {
```

```
Elf32_Word st_name;
```

```
Elf32_Addr st_value;
```

```
Elf32_Word st_size;
```

```
char type : 4,
```

```
    binding : 4;
```

```
unsigned char reserved;
```

```
Elf32_Half st_shndx;
```

```
} Elf_Symbol;
```

函数名在text节中

变量名在data节或bss节中

给出符号在字符串
指向在字符串表中

给出符号的位置，在
位值的字节偏移量。

给出符号所表示对象的
，则指定其所占字节个

符号类型：可以是
节(SECTION)

绑定属性：可以是
局(GLOBAL, 对于

未用

指出符号所在节在
的表项，故无法表
符号不会由于重定
定义符号，即在本
未被分配位置的未
st_value字段给出的是对齐要求，st_size给出的是最小长度。

例：对main.c/swap.c, **readelf -s main.o** 查看main.o中的符号表

1) 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	OBJECT	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。

2. 符号解析

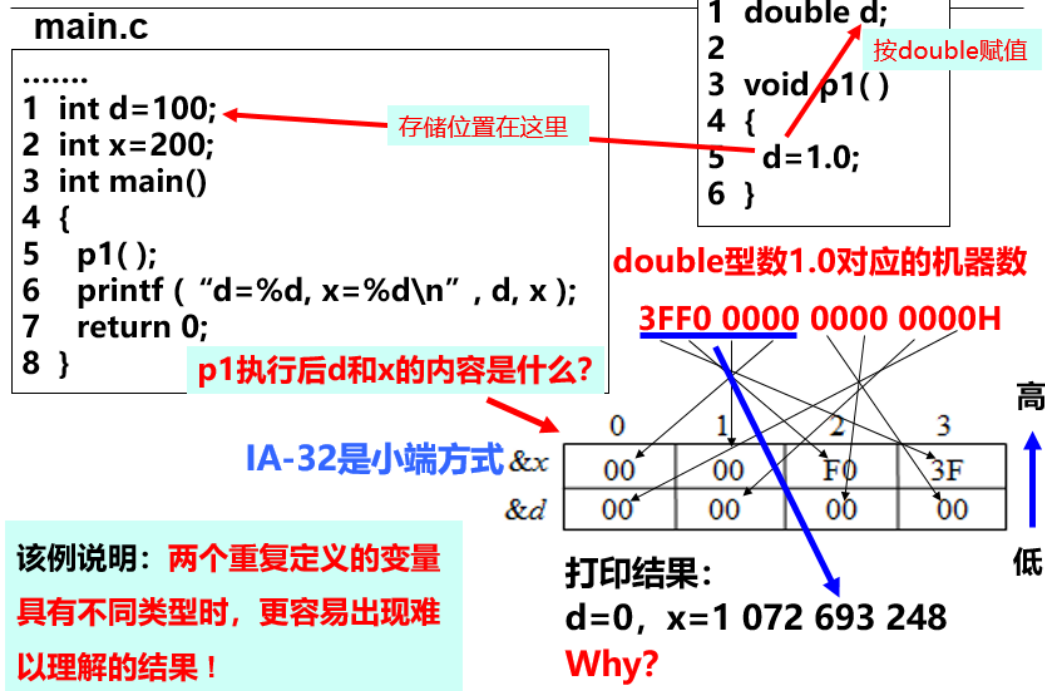
- ◆符号区分为定义符号和引用符号
- ◆符号解析就是将引用符号与定义符号建立关联的过程，从而知道引用符号的实体在哪里。
- ◆本地符号符号解析：本模块内部处理，比较简单
- ◆全局符号符号解析：跨模块处理，比较复杂
- ◆全局符号的强/弱特性：强符号定义和弱符号定义

链接器对符号的解析规则：

- ◆ 一个符号不管有多少处定义，最终只能有一个确定的定义。
- ◆ 单一定义的符号只要建立引用符号和定义符号的关联即可。
- ◆ 多重定义的符号：
 - Rule 1: 强符号不能多次定义
 - Rule 2: 若一个符号被定义为一次强符号和多次弱符号，
则以强定义为准
 - Rule 3: 若有多个弱符号定义，则任选其中一个

重复定义的变量容易出现难以理解的结果！

多重定义符号的解析举例



◆ 尽量避免使用全局变量

◆ 一定需要用的话，（1）尽量使用本地变量（static，本地化），（2）全局变量要赋初值（强定义化），（3）外部全局变量要使用extern（否则就成为弱类型定义符号）

3. 静态库:

◆存档文件, 一些可重定位目标文件的集合。

◆使用AR工具创建静态库:

```
ar rcs mylib.a myproc1.o myproc2.o
```

◆常用静态库:

libc.a (C标准库)、libm.a (C数学库)

4. 链接器中符号解析的全过程

◆维护三个集合：

- E 将被合并以组成可执行文件的所有目标模块集合
- U 当前所有未被解析的引用符号的集合
- D 当前所有定义符号的集合

◆符号解析的全过程

- 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
- 最后将E中的所有目标模块合并

注：符号解析成功与否与文件顺序有关

main.c

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

\$ 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！

main→myfunc1→printf

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。

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

链接顺序问题讨论

- 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

libx.a必须要在libz.a之前

- 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**

被链接的目标文件可以在命令行中重复出现

三、重定位

链接的第二步：**重定位**，又分三步

1) 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同一类型的新节

例如，所有.text节合并作为可执行文件中的.text节

2) 对**定义符号**进行重定位，**确定其地址**

- 根据每个新节在虚拟地址空间中的起始位置以及新节中每个定义符号的位置，为新节的每个定义符号确定其在统一虚拟地址空间中的存储地址。

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全局地址**

3) 对引用符号进行重定位

- 链接器对合并后新代码节(.text)和新数据节(.data)中的引用符号进行重定位，使其指向对应的定义符号起始处。

链接器怎么知道目标文件中有哪些引用符号需要重定位、所引用的又是哪个定义符号呢？

- 需要用到.o文件中.rel.data和.rel.text节中保存的重定位信息
- **编译器在对源文件进行编译时，会把每个全局符号（不管是定义符号还是引用符号）输出到汇编代码文件中，形成.symtab节。**
- **汇编器遇到引用时，为每个引用生成一个重定位条目记录重定位信息，并保存到目标文件的.rel.data和.rel.text节中**

重定位信息记录

- 汇编器为每一处引用生成一个**重定位条目**记录重定位信息

- **ELF中重定位条目格式如下:**

```
typedef struct {
```

```
    int offset;
```

```
    int symbol:24,
```

```
    type: 8;
```

```
} Elf32_Rel;
```

当前需重定位的位置相对于**其在目标文件所在节起始位置的字节偏移量**。

- 若重定位的是变量的位置，则所在节是.data节。
- 若重定位的是函数的位置，则所在节是.text节。

该引用符号**在符号表里的索引值**

重定位类型， R_386_32(绝对地址方式) 或 R_386_PC32(相对寻址方式)

- **重定位条目写在可重定位目标文件中。**

- **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**

内容就是上述结构的**结构数组**，每个元素对应一个需重定位的符号

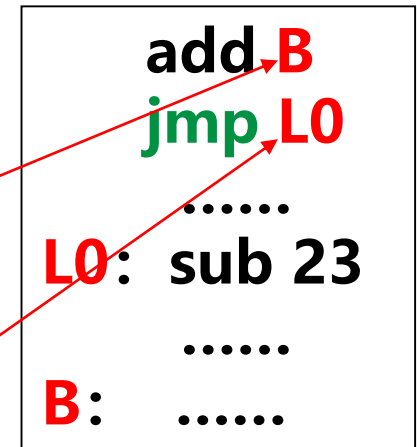
```
typedef struct {
    int offset;      /*节内偏移*/
    int symbol:24, /*所绑定符号*/
        type: 8;    /*重定位类型*/
} Elf32_Rel;
```

例如，在.rel.data节中有重定位条目如下

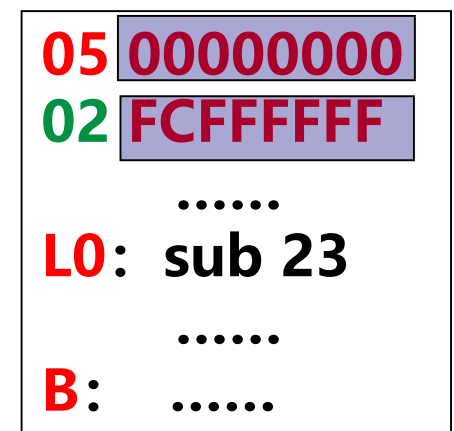
```
offset: 0x1
symbol: B
type: R_386_32 绝对地址方式
```

在.rel.text节中有重定位条目如下

```
offset: 0x6
symbol: L0
type: R_386_PC32 PC相对寻址方式
```



重定位后



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;  
}
```

符号定义

符号引用

- 局部变量temp分配在栈中，不会在过程外被引用，编译器不予考虑。

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[];

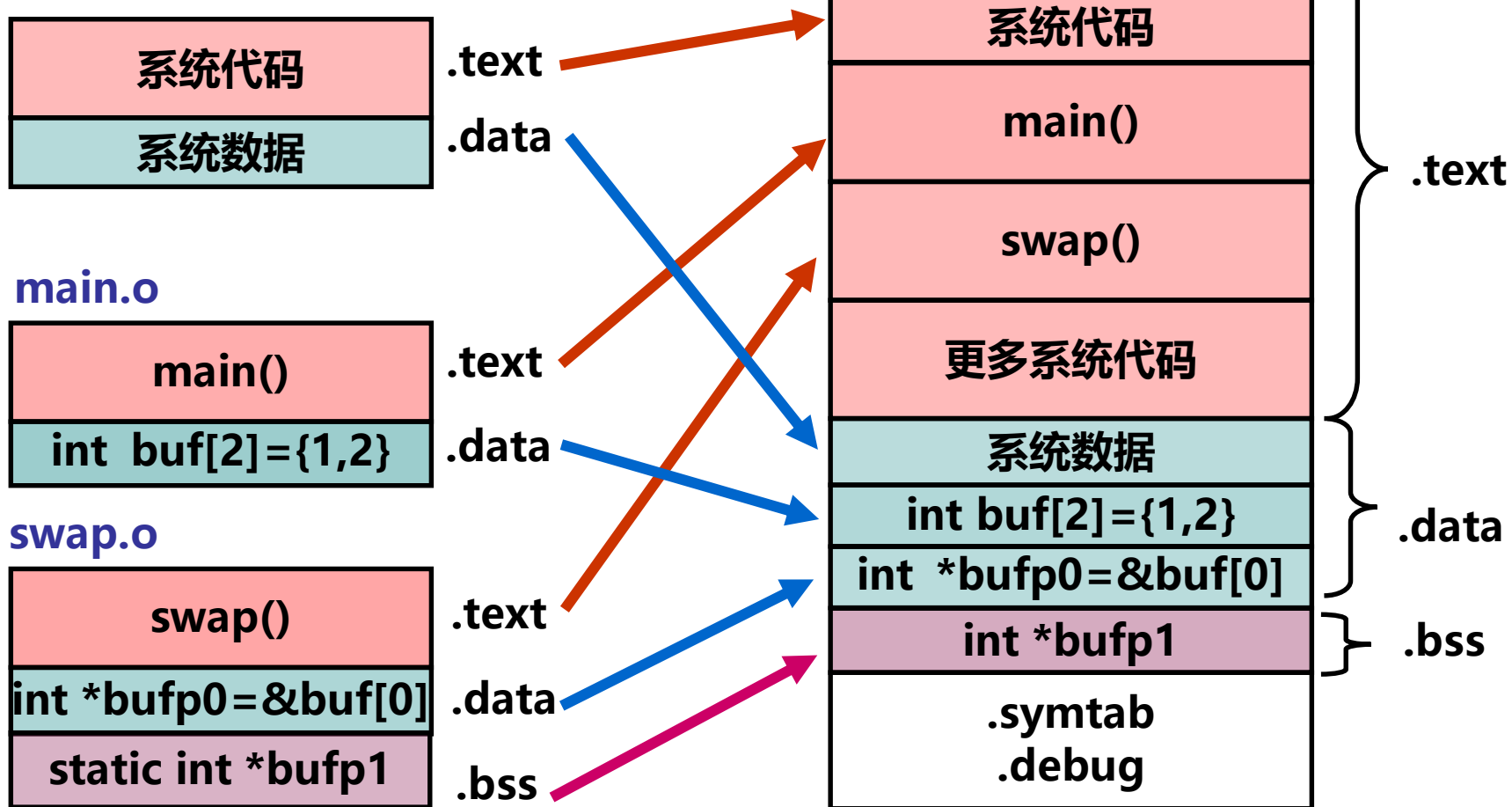
int *bufp0 = &buf[0];
static int *bufp1;

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

- 在main.o和swap.o的重定位节(.rel.text、.rel.data)中有重定位信息，给出每个需要重定位符号的引用位置、符号索引、重定位类型等信息。

可执行目标文件

可重定位目标文件



合并后，所有符号都有了全局地址

◆ 虽然bufp1是swap的本地符号，也需在.bss节重定位

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重定位前

main.o

Disassembly of section .text:

00000000 <main>:

0:	55	push	%ebp
1:	89 e5	mov	%esp,%ebp
3:	83 e4 f0	and	\$0xffffffff,%esp
6:	e8 fc ff ff ff	call	7 <main+0x7>
b:	b8 00 00 00 00	mov	\$0x0,%eax
10:	c9	leave	
11:	c3	ret	

7: R_386_PC32 swap

在main.o的rel_text节中有重定位条目:

r_offset=0x7,
r_sym=10
r_type=R_386_PC32

dump出
来的提
示信息

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。

r_sym=10说明引用的是swap!

main.o中的符号表

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

- main.o中的符号表中最后三个条目

readelf -s main.o 查看main.o中的符号表

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

 **r_sym=10**说明引用的是swap!

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

重定位类型：

- 重定位类型与特定的处理器有关。
- IA-32处理器的重定位类型有16种之多，最基本的是R_386_PC32和R_386_32

1) **R_386_PC32**：指明引用处采用的是**相对寻址**方式，即
有效地址 = PC的内容 + 重定位后的32位地址（偏移）

（PC的内容是下一条指令的地址）

（重定位后的32位地址简称为**重定位值**）

2) **R_386_32**：指明引用处采用**绝对地址**方式，即
有效地址 = 重定位后的32位地址

R_386_PC32的重定位方式举例：swap的重定位

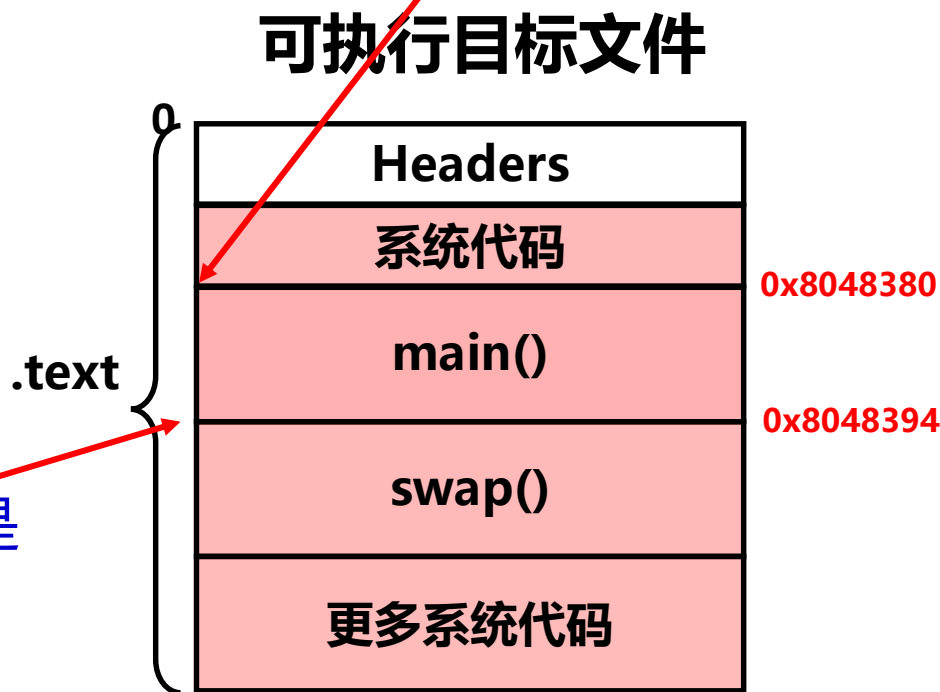
1) main.o + swap.o后，可执行文件的结构

假定：

- 可执行文件中，main函数对应的机器代码从**0x8048380**开始
- swap紧跟main后，其机器代码首地址按**4字节边界对齐**

则swap起始地址为多少？

- $0x8048380 + 0x12 = 0x8048392$
(main.o中.text节占12B)
- 在4字节边界对齐的情况下，
swap的机器代码的起始地址是
0x8048394



➤ 重定位后call指令的机器代码是什么？

分析：1) call命令处，PC的内容等于下一条指令的地址，所以

$$\text{PC} = 0x8048380 + 0x07 + 4 = 0x804838b$$

2) call命令执行时要转向swap，所以call命令的转移目标地址就是swap的地址0x8048394，又因为

$$\text{转移目标地址} = \text{PC} + \text{偏移地址 (重定位值)}$$

所以，重定位值

$$= \text{转移目标地址} - \text{PC}$$

$$= 0x8048394 - 0x804838b$$

$$= 0x9$$

main.o

Disassembly of section .text:

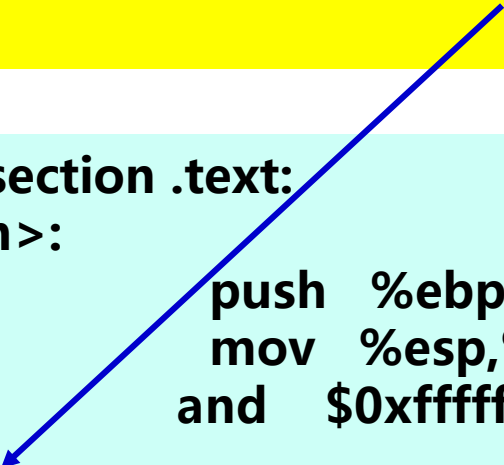
00000000 <main>:

0:	55	push	%ebp
1:	89 e5	mov	%esp,%ebp
3:	83 e4 f0	and	\$0xffffffff,%esp
6:	e8 <u>fc ff ff ff</u>	call	7 <main+0x7>
			7: R_386_PC32 swap
b:	b8 00 00 00 00	mov	\$0x0,%eax
10:	c9	leave	
11:	c3	ret	

swap的机器代码的起始地址是0x8048394

所以，重定位后，call指令的机器代码为 **"e8 09 00 00 00"**

Disassembly of section .text:
00000000 <main>:
0: 55 push %ebp
1: 89 e5 mov %esp,%ebp
3: 83 e4 f0 and \$0xffffffff0,%esp
6: e8 09 00 00 00 call %0x9
b: b8 00 00 00 00 mov \$0x0,%eax
10: c9 leave
11: c3 ret



swap的机器代码的起始地址是**0x8048394**

➤ 这个是什么？

- 小端格式的负数的补码
- 即：0x ff ff ff fc = -4
- 为什么是-4？

分析：

- 此值称为**重定位的初始值** (**init**)
- **含义**：此处重定位时，**PC**距离**需要重定位的地址**偏移为**4字节** (**负方向的4字节**)，取值：**-4**
- 即，**PC = ADDR(.text) + r_offset - init**

main.o

Disassembly of section .text:

00000000 <main>:

```
0:      55                push %ebp
1:      89 e5            mov %esp,%ebp
3:      83 e4 f0          and $0xffffffff0,%esp
6:      e8 fc 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
```

r_offset: 7
init: -4

综上所述，PC相对地址方式下，**重定位值计算公式**为：

$$\text{ADDR}(r_sym) - ((\text{ADDR}(.text) + r_offset) - \text{init})$$

符号 r_sym
的存储位置
(0x8048394)

.text节运行时的
起始存储地址
(0x8048380)

call指令下条指令地址
即当前PC的值
(0x804838b)

R_386_32的重定位方式举例:

main.c

```
int buf[2]={1,2};  
  
int main()  
.....
```

经编译转化为main.o

buf定义在.data节中偏移为0处, 占8B, 没有需重定位的符号。

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
```

swap.c

```
extern int buf[];  
  
int *bufp0 = &buf[0];  
static int *bufp1;  
  
void swap()  
.....
```

经编译转化为swap.o

bufp0定义在.data节中偏移为0处, 占4B, 初值为0x0, 内容需要重定位

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
```

swap.o中重定位节.rel.data中有重定位表项:

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

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

- **swap.o中的符号表中最后4个条目**

`readelf -s swap.o` 查看swap.o中的符号表

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

r_sym=9说明引用的是buf!

- 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。

R_386_32的重定位方式举例：buf的重定位

buf和bufp0同属于合并后得到**新的.data节**

- 假定：

- buf在运行时的存储地址ADDR(buf)=**0x8049620**

- 问：重定位后，bufp0的地址及内容变为什么？

- buf和bufp0都在.data中，且bufp0紧接在buf后
故，buf0的地址为 $0x8049620+8=$ **0x8049628**

- 因是**R_386_32方式(绝对地址)**，
故bufp0内容为buf的绝对地址
0x8049620。

即 “20 96 04 08”
(小端方式)

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

第2个元素



buf和bufp0的地址分别是0x8049620和0x8049628

bufp1的地址就是链接合并后.bss节的首地址，假定为0x8049700

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

数值型数据

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

6: c7 05 00 00 00 00 04 movl \$0x4,0x0

d: 00 00 00

8: R_386_32

c: R_386_32

.bss

buf

bufp1 = &buf[1];

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)

temp = *bufp0;

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

*bufp0 = *bufp1;

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

*bufp1 = temp;

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 \$0xffffffff,%esp
8048386:	<u>e8 09 00 00 00</u>	call 8048394 <swap>
804838b:	b8 00 00 00 00	mov \$0x0,%eax
8048390:	c9	leave
8048391:	c3	ret
8048392:	90	nop
8048393:	90	nop

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) $R[esp] \leftarrow R[esp] - 4$
- 2) $M[R[esp]] \leftarrow R[eip]$
- 3) $R[eip] \leftarrow R[eip] + 0x9$

返回地址入栈

构造跳转地址

重定位后

08048394 <swap>:

```
8048394: 55
8048395: 89 e5
8048397: 83 ec 10
804839a: c7 05 00 97 04 08 24
80483a1: 96 04 08
80483a4: a1 28 96 04 08
80483a9: 8b 00
80483ab: 89 45 fc
80483ae: a1 28 96 04 08
80483b3: 8b 15 00 97 04 08
80493b9: 8b 12
80493bb: 89 10
80493bd: a1 00 97 04 08
80493c2: 8b 55 fc
80493c5: 89 10
80493c7: c9
80493c8: c3
```

```
push %ebp
mov %esp,%ebp
sub $0x10,%esp
mov $0x8049624,0x8049700

mov 0x8049628,%eax
mov (%eax),%eax
mov %eax,-0x4(%ebp)
mov 0x8049628,%eax
mov 0x8049700,%edx
mov (%edx),%edx
mov %edx,(%eax)
mov 0x8049700,%eax
mov -0x4(%ebp),%edx
mov %edx,(%eax)
leave
ret
```

仔细观察全局变量和局部变量的访问方式的不同

思考：上述重定位后的地址都是用的绝对地址，难道每次运行都能将这些单元加载到固定的内存单元里吗？——真相：这些是可执行目标文件对应的逻辑地址空间里的虚拟地址。

回顾：可执行文件中的程序头表

- 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

```
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;
```

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。

- 以下是由OBJDUMP得到某可执行文件的段头部表内容

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）

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-

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

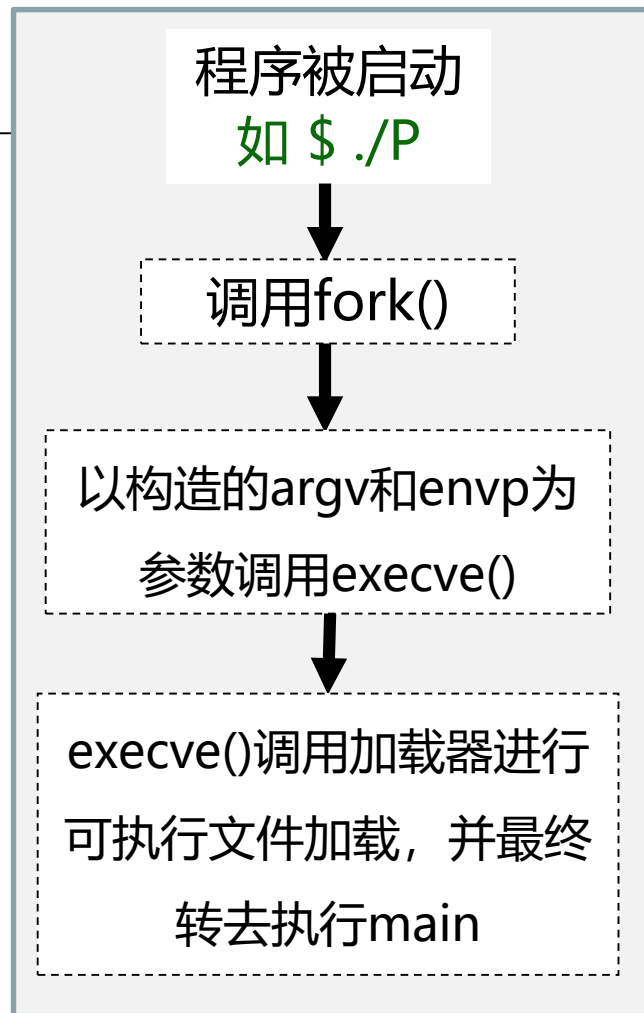
四、可执行文件的加载

当**启动一个可执行目标文件执行**时：

- 1) 通过**execve**系统调用函数来启动**加载器**;
(linux系统下)
- 2) 加载器(loader)根据可执行文件的**程序头表**中的信息, 将可执行文件的相关节的内容与虚拟地址空间中的只读代码段和可读写数据段建立映射 —— **代码和数据从磁盘执行文件中“拷贝”到存储器中, 但实际上不会真正拷贝, 仅建立一种映射**(这涉及到许多复杂的过程和一些重要概念, 将在后续课程学习)。

- 加载后, 将PC (EIP) 设定指向**Entry point** (即符号_start处)
- 然后执行main函数, 开始执行程序。

_start: **__libc_init_first** → **_init** → **atexit** → **main** → **_exit**



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

可执行目标文件入口点
第一条指令的地址

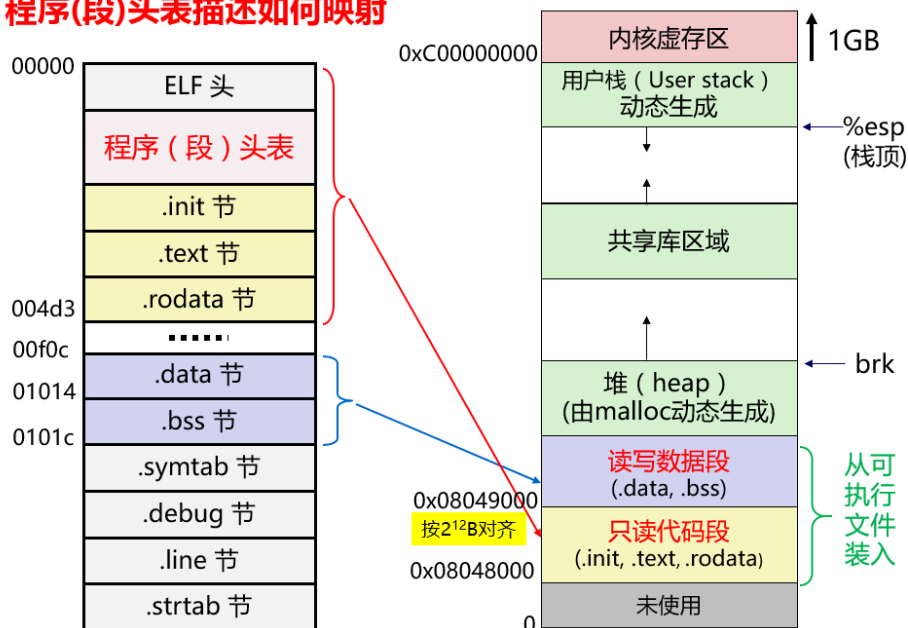
ELF头
程序头表
.init 节
.text 节
.rodata 节
.data 节
.bss 节
.symtab 节
.debug 节
.line 节
.strtab 节
节头表

关于ELF可执行目标文件到虚拟地址空间的映射

- 可执行目标文件的程序头表中记录了可执行文件中所有存储段的相关信息，如存储段类型、段在虚拟地址空间中的起始地址、长度、对齐方式、访问权限等。

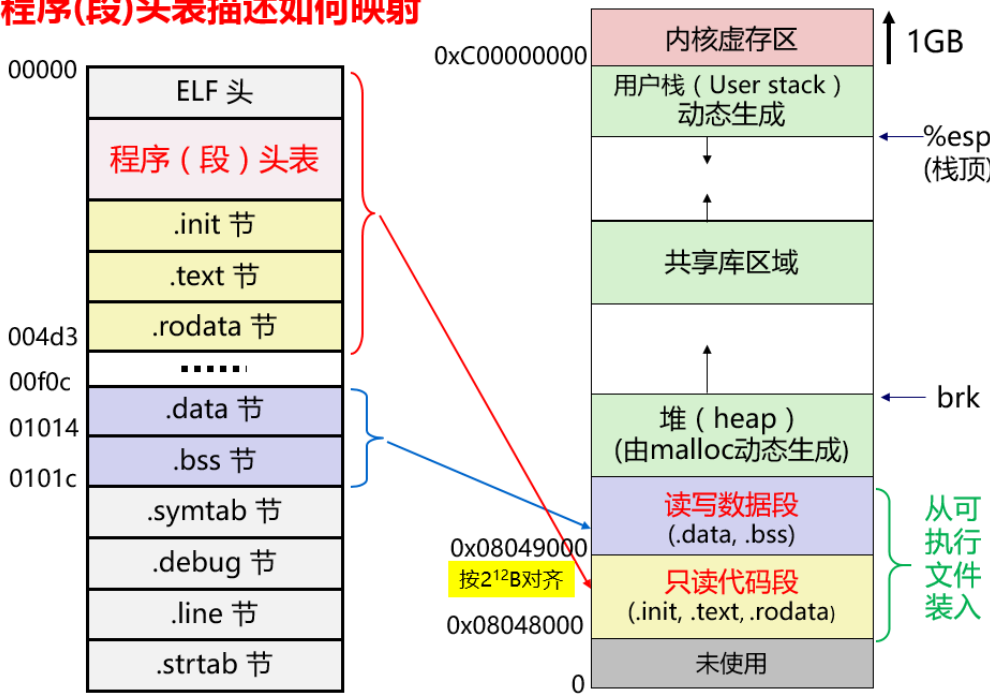
➤ 映射：

程序(段)头表描述如何映射



- ① 只读代码段(只读段)和可读写数据段(读/写段)为**可装入段**(PT_LOAD);
- ② 只读段 (包括.init节、.text节和.rodata节) 缺省映射到虚拟地址为**0x08048000**开始的一段区域;
- ③ 可读写数据段(包括.data节和.bss节) 映射到只读段后面按4KB对齐的高地址上。其中.bss节所在存储区在运行时被初始化为0;

程序(段)头表描述如何映射

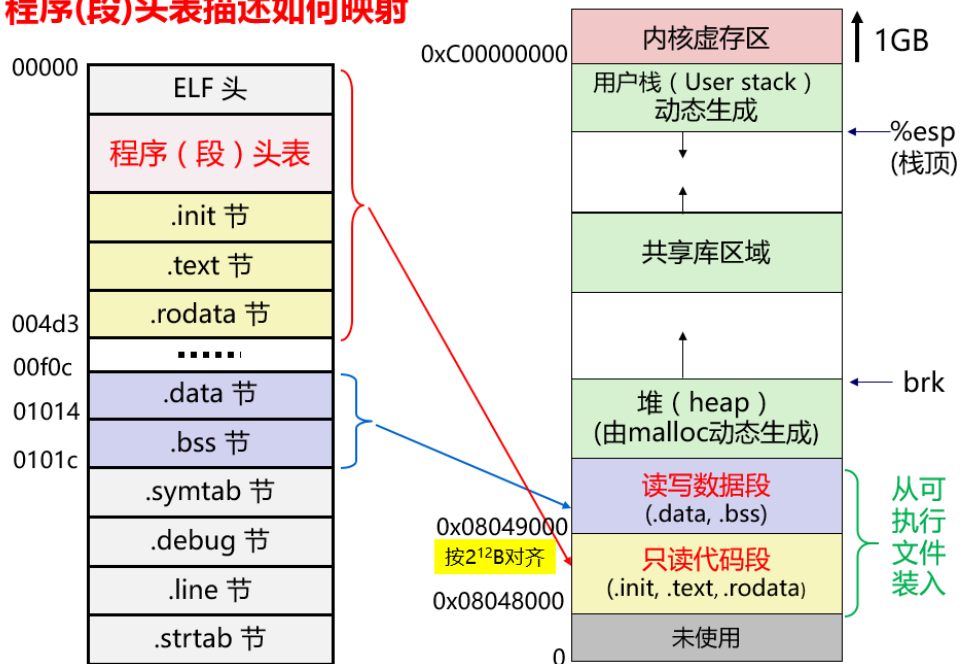


- ④ **运行时堆**(run-time heap)在可读写数据段后面4KB对齐的高地址处（通过malloc库函数动态申请的空间时向高地址进行空间分配）。
- ⑤ 运行时的**用户栈**（user stack）映射到**0xC0000000**下面向低地址空间增长的区域。
- ⑥ 堆区和栈区中间有**共享库代码**空间。
- ⑦ 栈区以上（0xC0000000 往上）的高地址区是**操作系统内核**的虚拟存储区，用户程序不可访问。

1) 引入虚拟存储管理大大简化了链接器的设计和实现

所有可执行目标文件都采用**一样的存储映像方式**，映射到一个**统一的虚拟地址空间**上。这样，链接器在重定位时可以完全按照统一的虚拟存储空间来确定每个符号的地址，而不用管其数据和代码将来存放在主存或磁盘的何处。

程序(段)头表描述如何映射

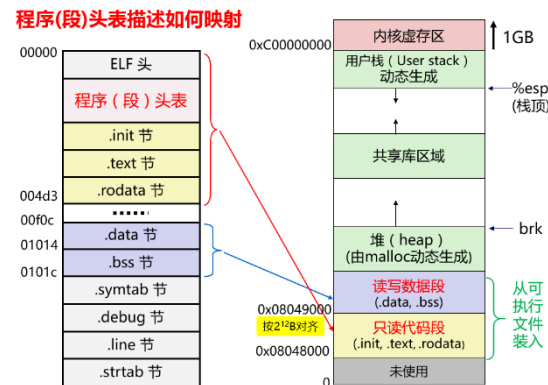


用户程序只考虑对虚拟地址空间的访问，虚拟地址到物理地址的转换由系统负责，用户程序不用关心。

2) 引入虚拟存储管理简化了程序的加载过程

➤ 在**统一的虚拟地址空间映像里**，每个可执行目标文件的只读代码段都被映射到**0x08048000**开始的一块连续区域，可读写数据段也映射到虚拟地址空间的一块连续区域。这样，加载器就可以非常容易地对这些区域进行分页，并初始化相应页表项的内容。

- 加载时，只读代码段和可读写数据段对应的页表的表项首先被初始化为“**未缓存页**”，并指向磁盘中可执行目标文件中的对应位置。
- 可见，在程序加载的过程中，程序代码和数据开始的时候并没有真正从磁盘加载到主存，而只是创建了只读代码和可读写数据段对应的页表项。
- 在执行过程中发生“缺页”异常时，再由操作系统负责把相应的页从磁盘调入内存。



关于虚拟存储管理、虚拟地址空间、页表、缺页异常等内容见后面章节或操作系统课程。

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

库文件 (Libraries)

■ 可以将一些可重定位目标文件包装成库文件 (.a文件)

- 特别是一些可为**多个程序使用**的可重定位目标文件，如C语言的标准函数（printf、scanf等），封装在库文件（libc.a）中，以便于其他程序使用其中的库函数（如printf），并在链接的时候将库函数的目标代码合并到可执行目标文件中。

■ 库中库函数的可重定位目标模块什么时候与程序的其他可重定位目标模块合并呢？

- 可以在link的时候，将库函数的目标代码“物理”加入程序的可执行代码中——静态链接；
- 也可以在link的时候不“物理”加入，而是在程序的可执行代码中引用这些库函数的地方做好**引用标识**，并将库函数的目标代码实际加入的时刻推迟到执行程序的时刻——动态链接。

- **静态库有一些缺点：**

- 库函数（如printf）的实体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，每个进程都有它们的副本，造成极大的**主存资源浪费**
- 库函数（如printf）被合并到可执行目标中，磁盘上存放着数千个可执行文件，每个文件里面也将包含同样多的库函数副本，造成**磁盘空间的极大浪费**
- 更新维护困难，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）**

- 也是一个目标文件，包含有代码和数据。只是将这些代码和数据从程序中分离出来，单独存放，在磁盘和内存中都**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：

- 在第一次运行加载进行 (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

■ 特点：

- 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: 位置无关代码
(Position Independent Code)

- 1) 使共享库代码的位置可以是不确定的
- 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

gcc -c myproc1.c myproc2.c 位置无关的共享代码库文件
gcc -shared -fPIC -o mylib.so myproc1.o myproc2.o

```
$ ar rcs libc.a atoi.o printf.o ... random.o
```

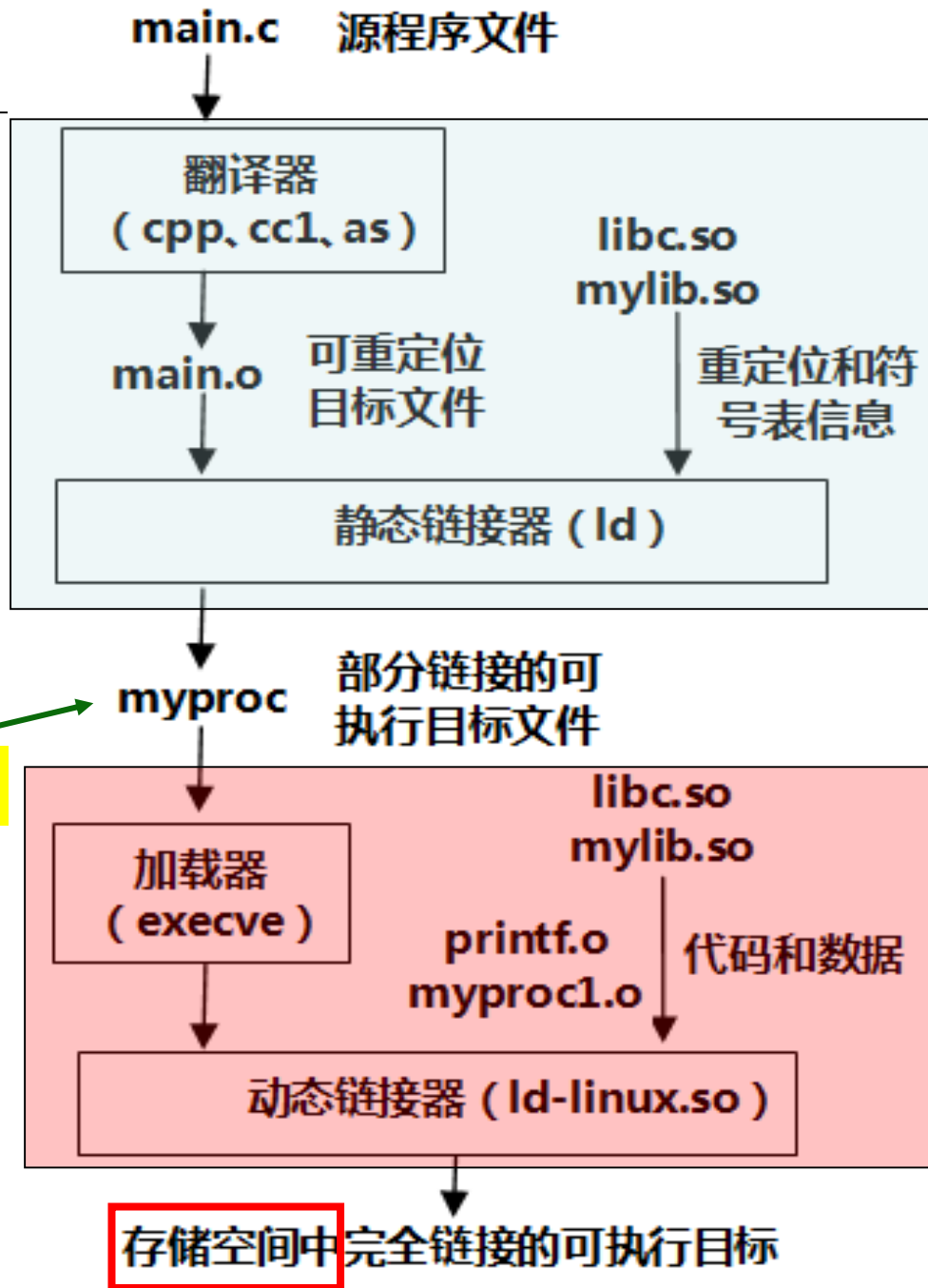
加载时动态链接

gcc -c main.c **libc.so**无需明显指出
gcc -o myproc main.o **./mylib.so**
main.c

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

调用关系: **main**→**myfunc1**→**printf**

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。



- 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

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

```
#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;
}
```

➤ 通过**动态链接器接口**提供的函数，程序可以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**

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**
 - 使得即使**共享库代码**的长度发生变化，也**不影响**调用它的程序
- **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
 - **要实现动态链接，必须生成PIC代码**
- 四种引用情况
 - (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:                  PC相对寻址
.....
```

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

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

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

JMP指令也可用相对寻址方式解决

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

(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

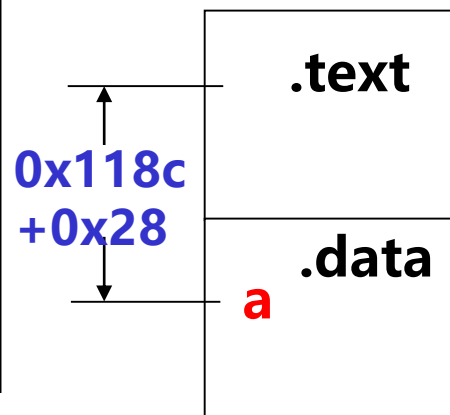
```
0000344 <bar>:
0000344: 55                pushl %ebp
0000345: 89 e5             movl  %esp, %ebp
0000347: e8 50 00 00 00    call 39c <__get_pc>
000034c: 81 c1 8c 11 00 00 addl  $0x118c, %ecx
0000352: c7 81 28 00 00 00 movl  $0x1, 0x28(%ecx)
.....
0000362: c3               ret

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

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

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

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

```
static int a;  
extern int b;  
extern void ext();  
  
void bar()  
{  
    a=1;  
    b=2;  
}  
.....
```

00000344 <bar>:

00000344: 55

pushl %ebp

.....

00000357: **e8 00 00 00 00** call 0000035c

0000035c: **5b**

popl %ebx

0000035d:

addl \$1180, %ebx

.....

movl (%ebx), %eax

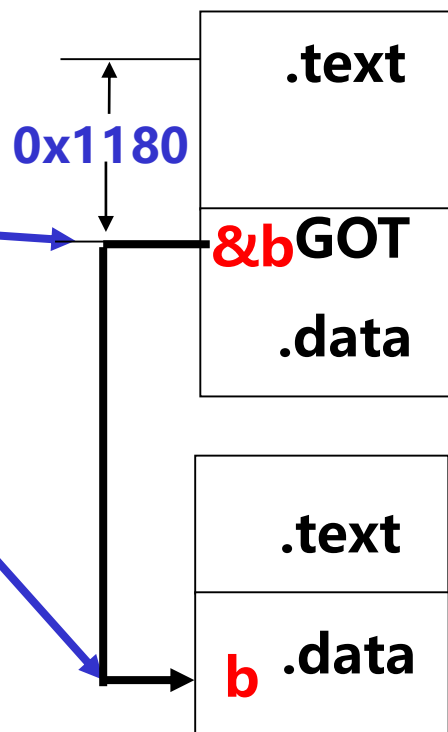
.....

movl \$2, (%eax)

.....

- 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



共享库模块

(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

0000050c <foo>:

0000050c: 55

pushl %ebp

.....

00000557: e8 00 00 00 00 call 0000055c

0000055c: 5b

popl %ebx

0000055d:

addl \$1204, %ebx

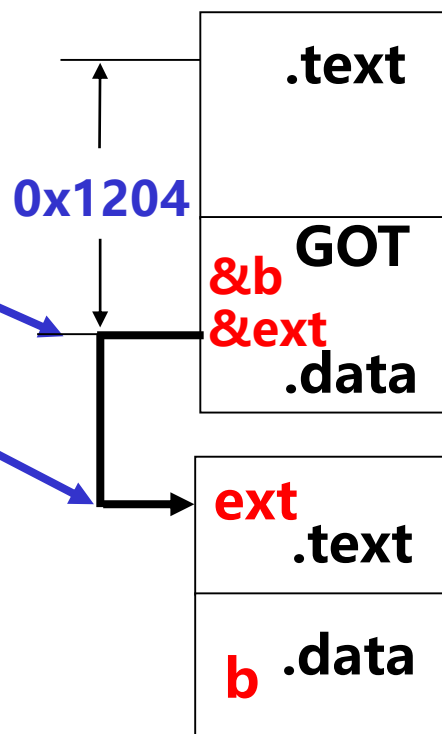
.....

call *(%ebx)

.....

***(%ebx)为间接地址: $R[eip] \leftarrow M[R[ebx]]$**

```
static int a;  
extern int b;  
extern void ext();  
  
void foo()  
{  
    bar();  
    ext();  
}  
.....
```



- **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）

共享库模块

(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 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>
```

804833c	:	PLT[0]
804834c	:	PLT[1]
	.text	
8049584	0804956c	GOT[0]
8049588	4000a9f8	GOT[1]
804958c	4000596f	GOT[2]
8049590	08048352	GOT[3]
	.data	

可执行文件foo

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