

# 第四章 程序的链接与加载执行

目标文件格式

符号解析与重定位

共享库与动态链接

库打桩机制

可执行文件的加载执行

# 程序的链接与加载执行

---

- 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。
  - 简单了解可执行文件的加载执行过程
- 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
  - 共享（动态）库链接、库打桩机制
  - 可执行文件的加载执行过程

# 程序的链接与加载执行

---

- 分以下四个部分介绍
  - 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
  - 第三讲：动态链接和库打桩机制
    - 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、位置无关代码的生成、库打桩机制
  - 第四讲：可执行文件的加载和执行
    - 可执行文件的加载
    - 程序和指令的执行过程

# 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");
```

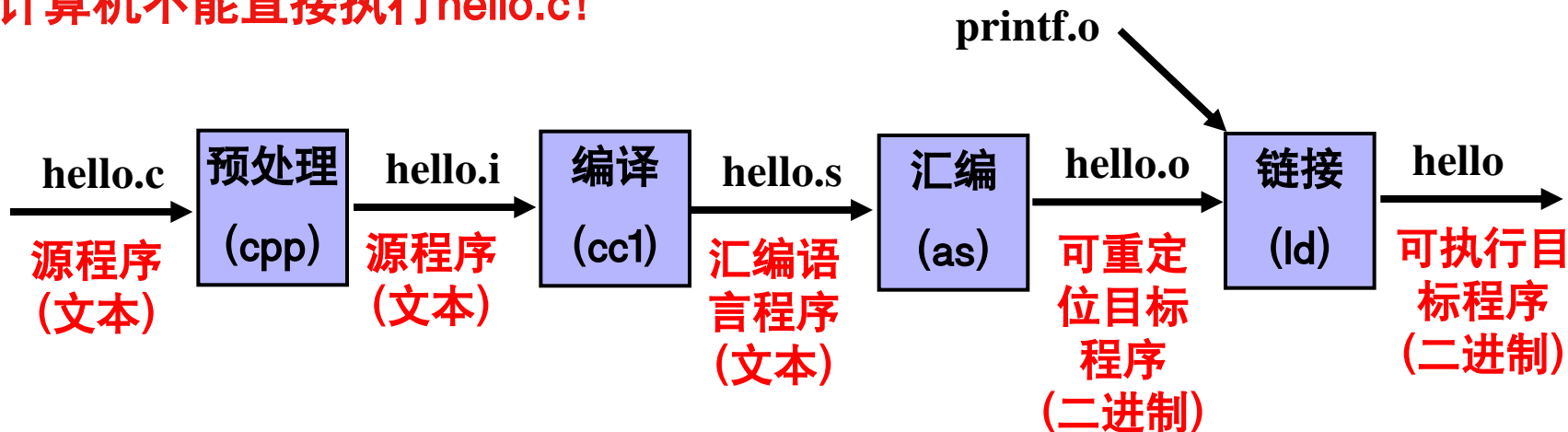
```
}
```

## hello.c的ASCII文本表示

```
# i n c l u d e < 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!



# 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
- ...

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

add B  
jmp

L0

L0

sub C

B: .....

C: .....

- 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I/O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**符号的定义**？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

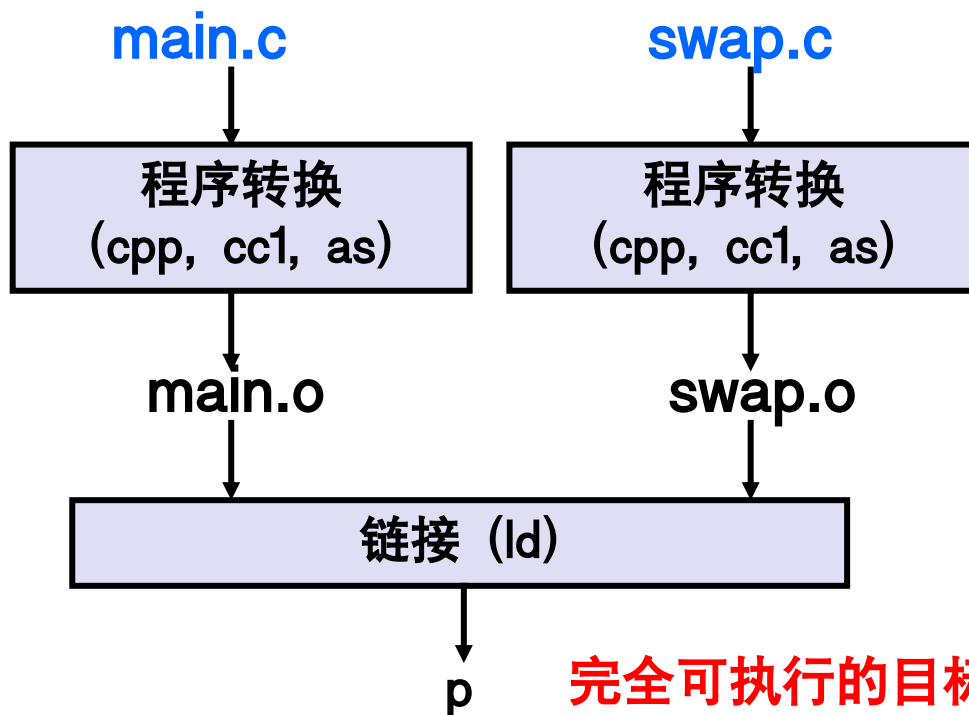
– \$ ./p

–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



源程序文件

分别转换（预处理、编译、汇编）为可重定位目标文件

完全可执行的目标文件

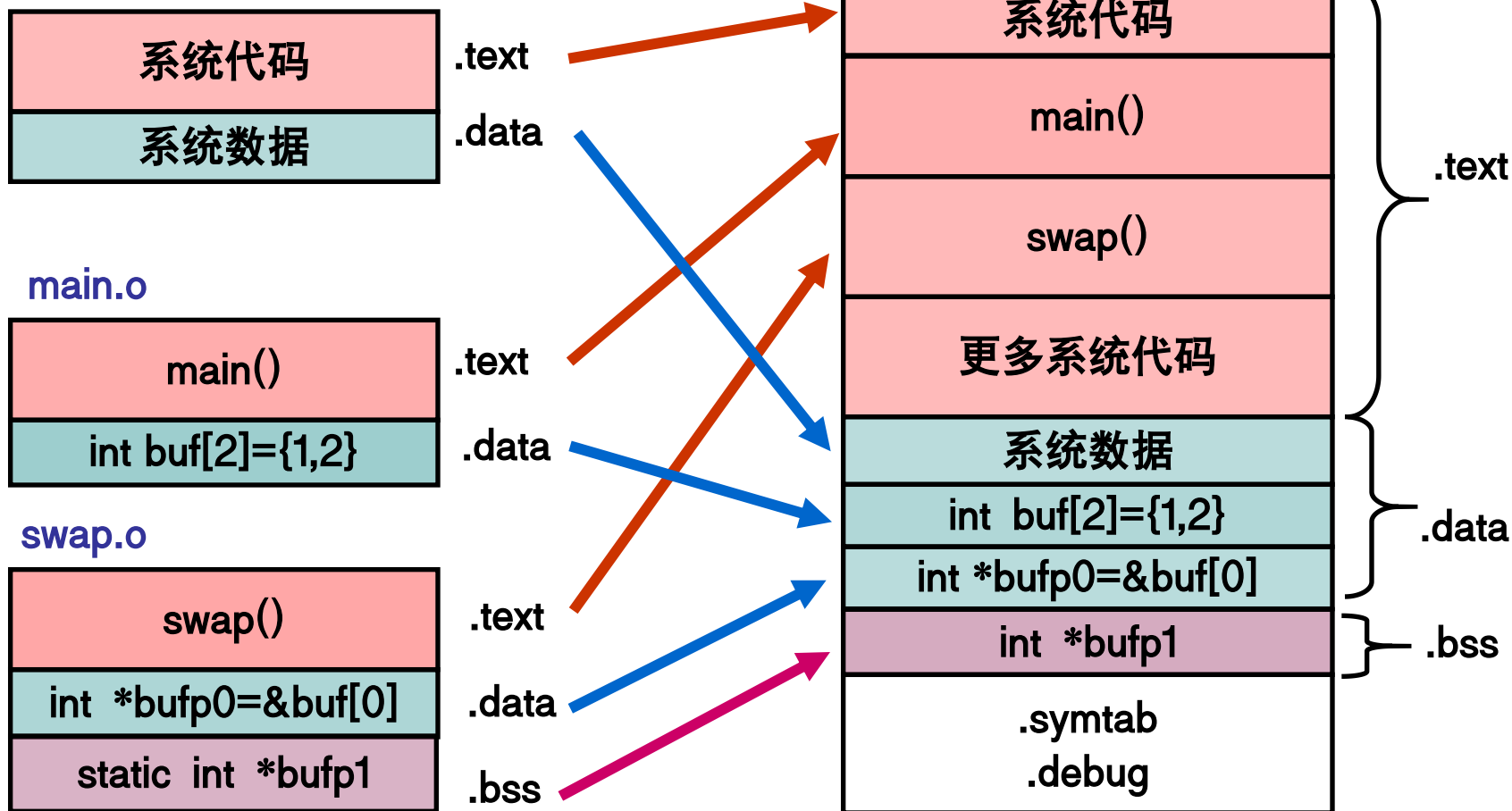


# 链接过程的本质

链接本质：合并相同的“节”

可执行目标文件

可重定位目标文件



# 目标文件

在LA64中将以下程序编译生成可重定位文件test.o和可执行文件test

```
/* 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;
}
```

可重定位文件由单个模块生成，可执行文件由多个模块组合而成。对于前者，代码总是从0开始，对于后者，代码的地址是虚拟地址空间中的地址。

**objdump -d test.o**

0000000000000000 <add>:

0:	00101484	add.w	\$a0,\$a0,\$a1
4:	4c000020	jirl	\$r0,\$ra,0

**objdump -d test**

0000000120000680 <add>:

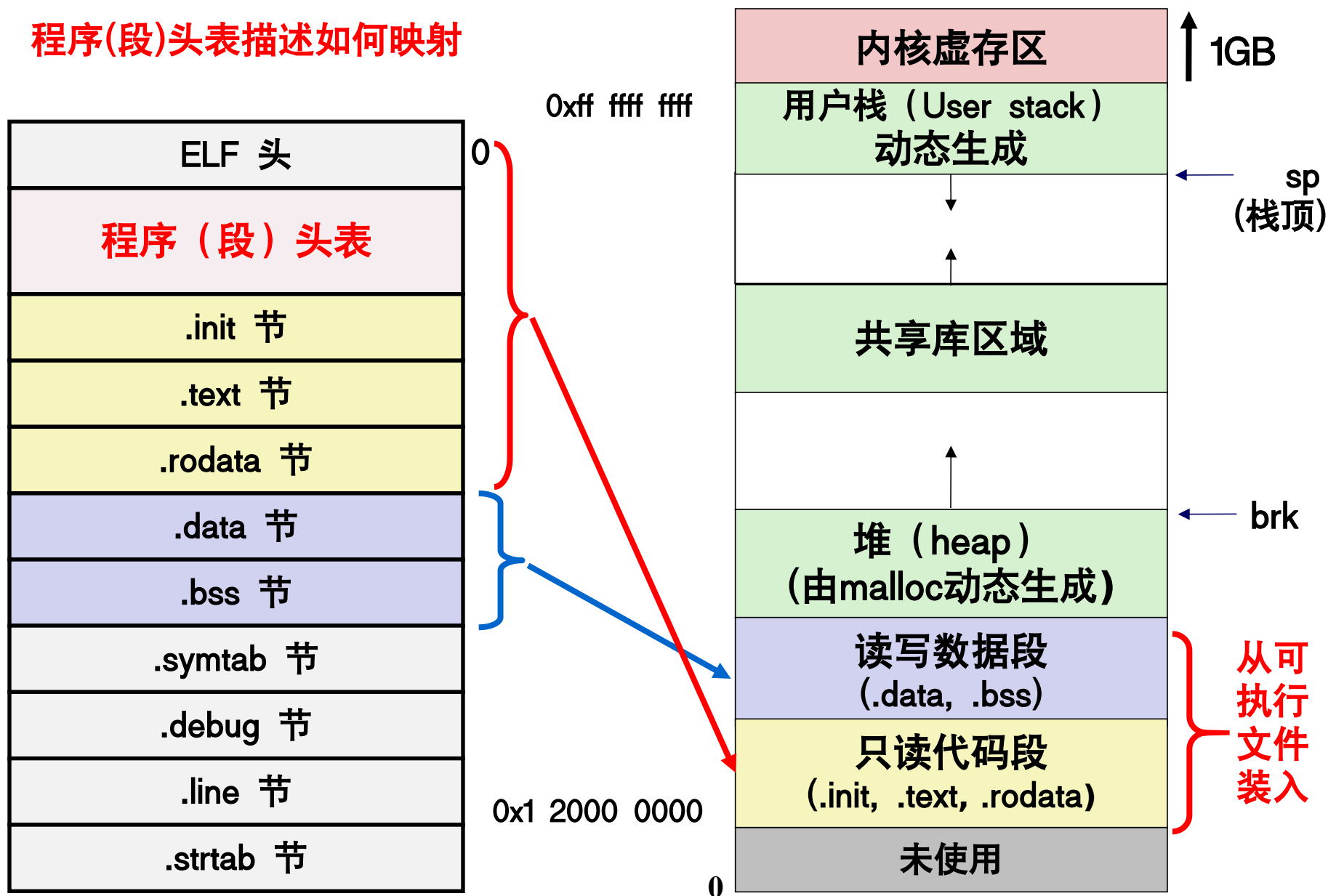
120000680:	00101484	add.w	\$a0,\$a0,\$a1
120000684:	4c000020	jirl	\$r0,\$ra,0

通过objdump命令输出的结果包括指令的地址、指令机器代码和反汇编得到的汇编指令代码。

可重定位文件test.o中add模块代码起始地址为0；在可执行文件test中add的起始地址为0x1 2000 0680

# LA64可执行文件的存储器映像

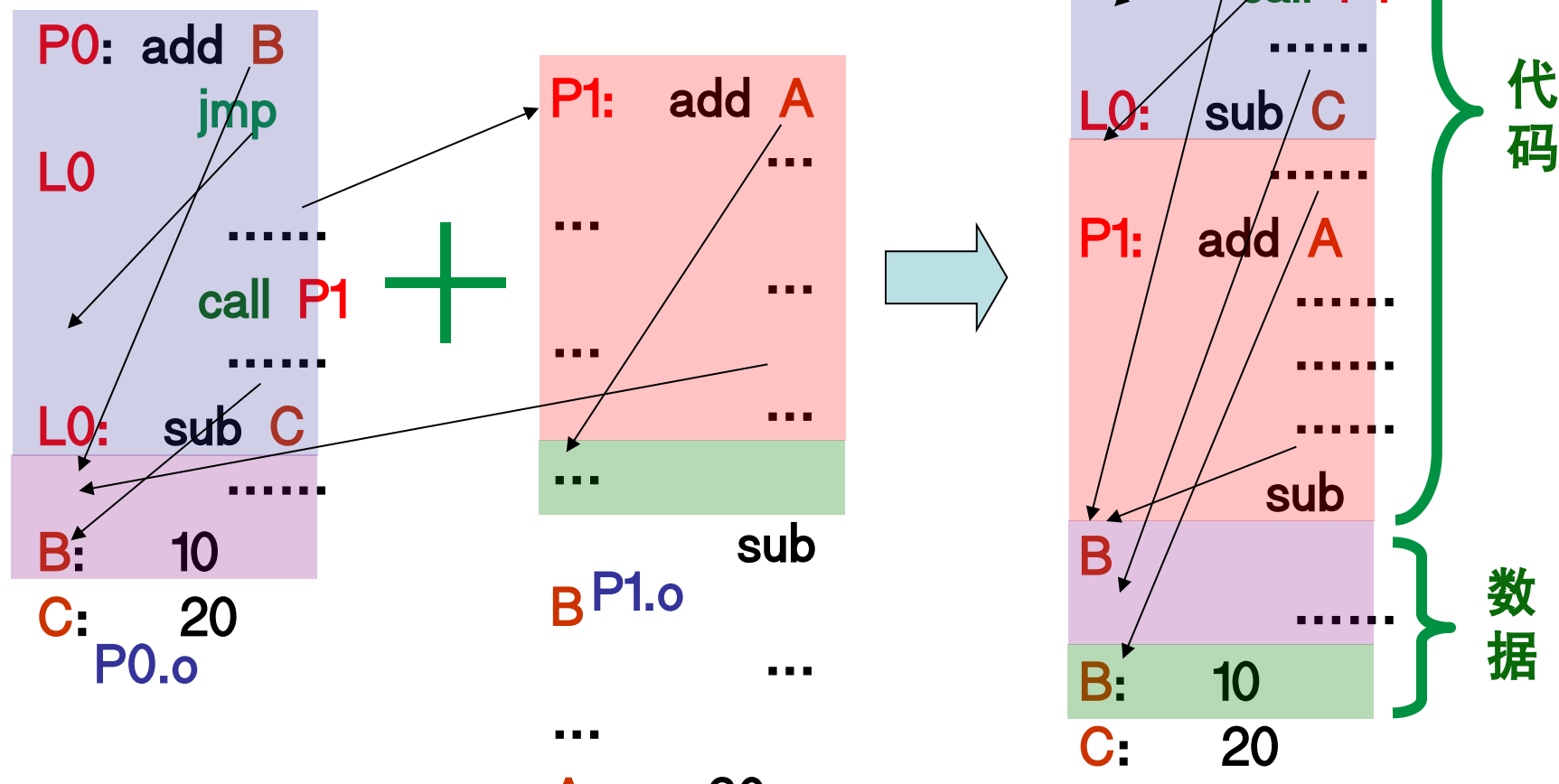
程序(段)头表描述如何映射



# 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

add B  
jmp

L0

- 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 (值) 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

L0

sub C

# 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

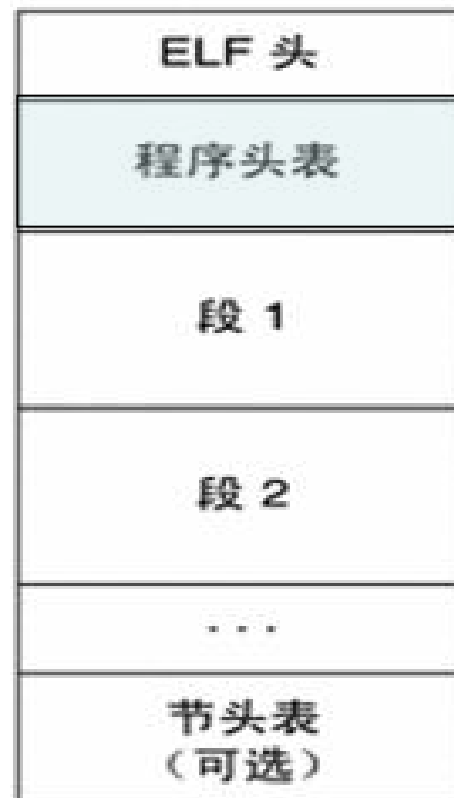
节 (section) 是 ELF 文件中具有相同特征的最小可处理单位

.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

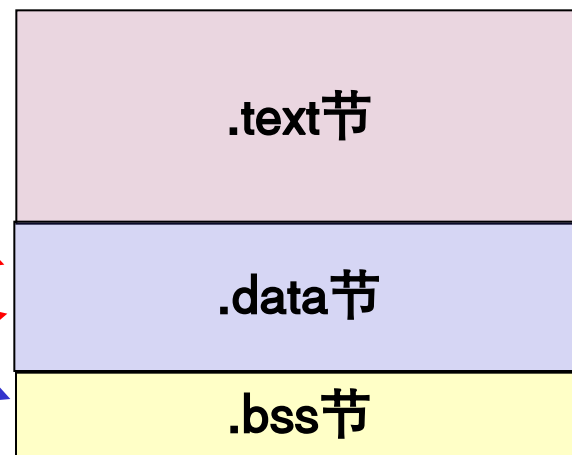


# 链接视图—可重定位目标文件

- 可被链接（合并）生成可执行文件或**共享目标文件**
- 若干个可重定位目标文件可组成**静态链接库文件**
- 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

0

ELF 头

- ü 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、节头表的起始位置和长度等

.text 节

- ü 编译汇编后的代码部分

.rodata 节

- ü 只读数据, 如 printf 格式串、switch 跳转表等

.data 节

- ü 已初始化且初值不为0的全局/静态变量

.bss 节

- ü 未初始化或初值为0的全局/静态变量, 仅是占位符, 不占任何实际磁盘空间

.rela.text节、.rela.data节

- ü .text节和.data节相关的重定位信息

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

# LA32中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;
```

```
}
```

a在10和17之间

```
        addi.w    $r12, r4, -10(0xff6)
```

```
        addi.w    $r13, r0, 7(0x7)
```

```
        bltu      $r13, r12, 52(0x34)
```

```
        slli.w    $r12, $r12, 0x2
```

```
        pcaddu12i $r13, 101(0x65)
```

```
        addi.w    $r13, $r13, -1936(0x870)
```

```
        add.w     $r12, $r13, $r12
```

```
        ld.w      $r12, $r12, 0
```

```
        jirl      $r0, $r12, 0
```

```
    .L1
```

```
        andi      $r6, $r5, 0xf
```

```
    .L2
```

```
        addi.w    $r4, $r6, 50(0x32)
```

```
        jirl      $r0, $r1, 0
```

```
    .L3
```

```
        addi.w    $r4, $r5, 50(0x32)
```

```
        jirl      $r0, $r1, 0
```

```
    .L4
```

```
        move      $r4, $r5
```

```
    .L5
```

```
        jirl      $r0, $r1, 0
```

R[r12]=a-10=i

if (a-10)>7 转 L5

R[r12]←i\*4

转.L6+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 2	a=
.L6	
.word	.L2 10
.word	.L5 11
.word	.L3 12
.word	.L5 13
.word	.L4 14
.word	.L1 15
.word	.L5 16
.word	.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
```

```
    Elf32_Half
```

```
    Elf32_Half
```

```
    Elf32_Word
```

```
    Elf32_Addr
```

```
    Elf32_Off
```

```
    Elf32_Off
```

```
    Elf32_Word
```

```
    Elf32_Half
```

```
    Elf32_Half
```

```
    Elf32_Half
```

```
    Elf32_Half
```

```
    Elf32_Half
```

```
    Elf32_Half
```

```
} Elf32_Ehdr;
```

```
    e_ident[EI_NIDENT];
```

```
    e_type;
```

```
    e_machine;
```

```
    e_version;
```

```
    e_entry;
```

```
    e_phoff;
```

```
    e_shoff;
```

```
    e_flags;
```

```
    e_ehsize;
```

```
    e_phentsize;
```

```
    e_phnum;
```

```
    e_shentsize;
```

```
    e_shnum;
```

```
    e_shstrndx;
```

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      可重定位目标文件的ELF头

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  
Data: 2's complement, little endian  
Version: 1 (current)  
OS/ABI: UNIX – System V  
ABI Version: 0      ABI: 应用程序二进制接口

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

没有程序头表

15x40B

.strtab在节头表中的索引

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)

# 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 和 sh\_info 用于与链接相关的节（如

```
    Elf32_Word
```

sh\_info 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)

PROGBITS节表示所含信息由程序定义; NOBITS类型表示不占盘空间; SYMTAB类型节为符号表; STRTAB表示节中存放的是字符串表, 这些字符串可能是函数名、变量名、节名等。

0

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)



# 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个节分配 (A) 空间

.text: 可执行

.data和.bss: 可读写

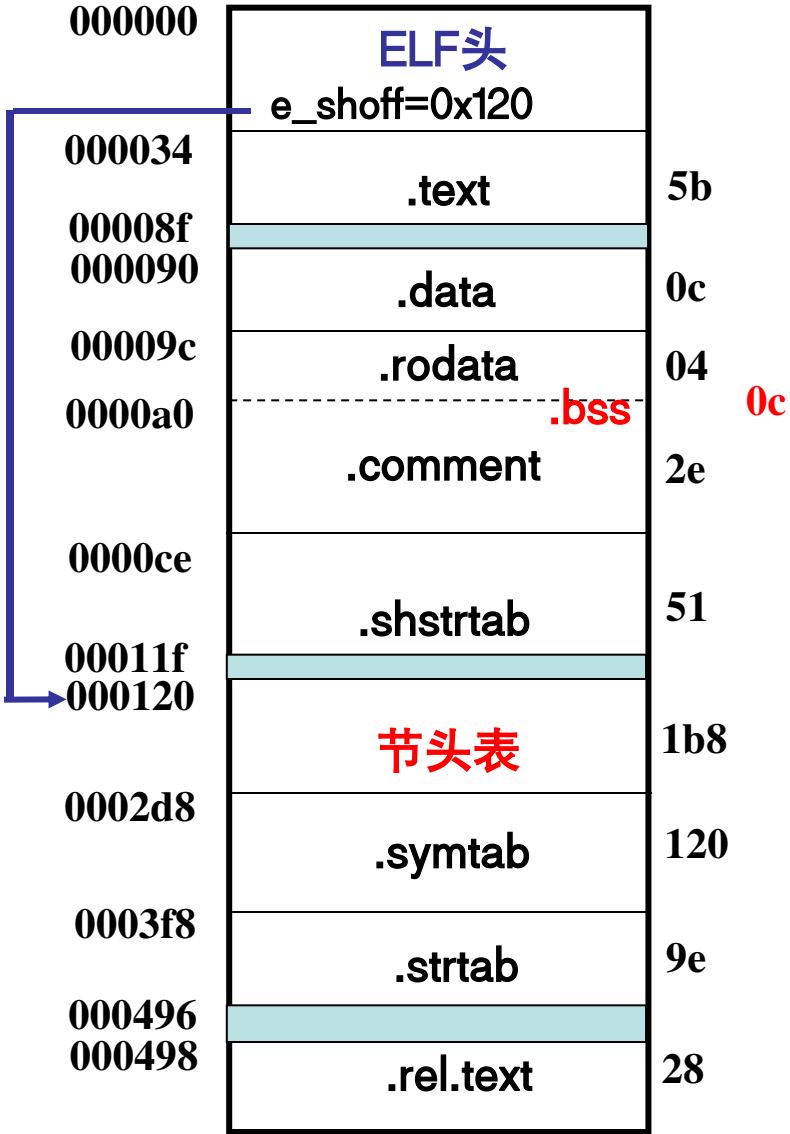
.rodata: 只读

shstrtab: 节名表

strtab: 符号名表

symtab: 符号表

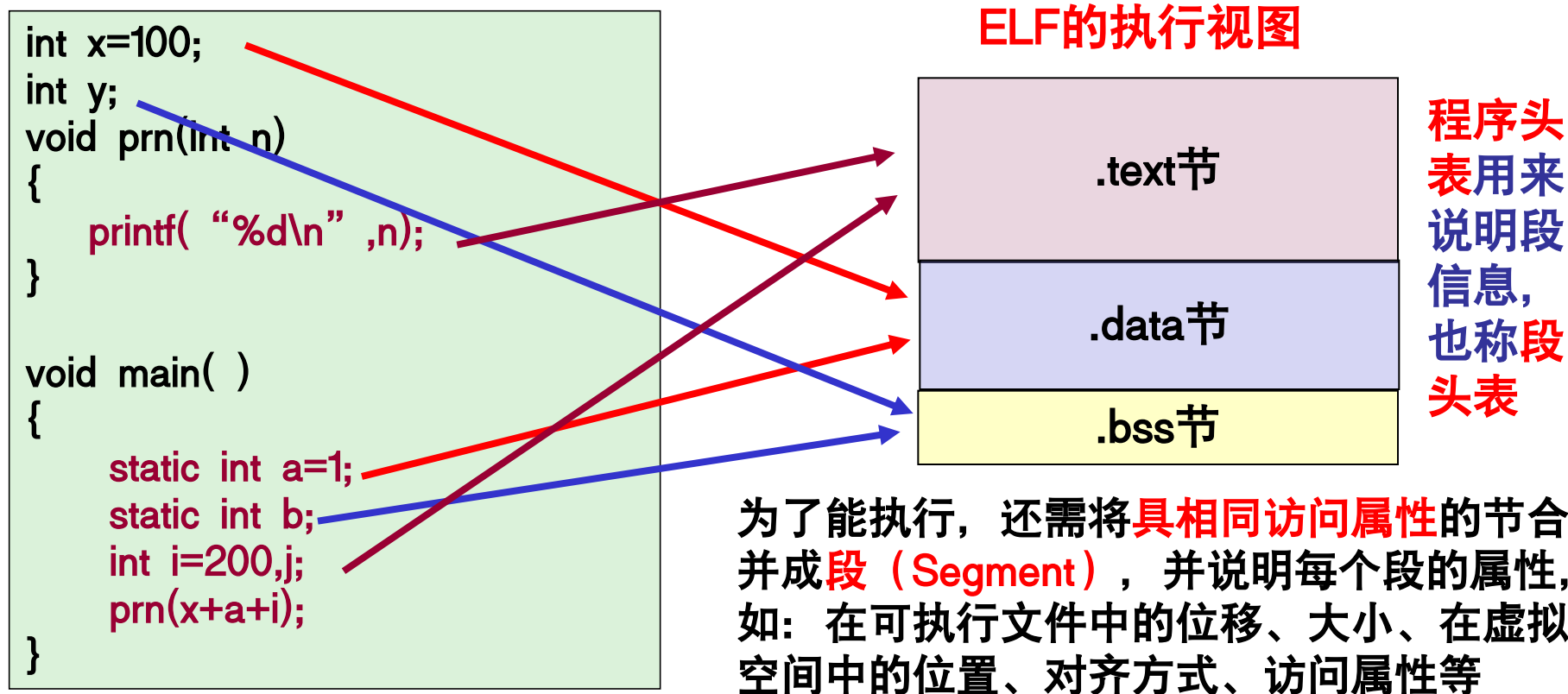
可重定位目标文件test.o的结构





# 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**

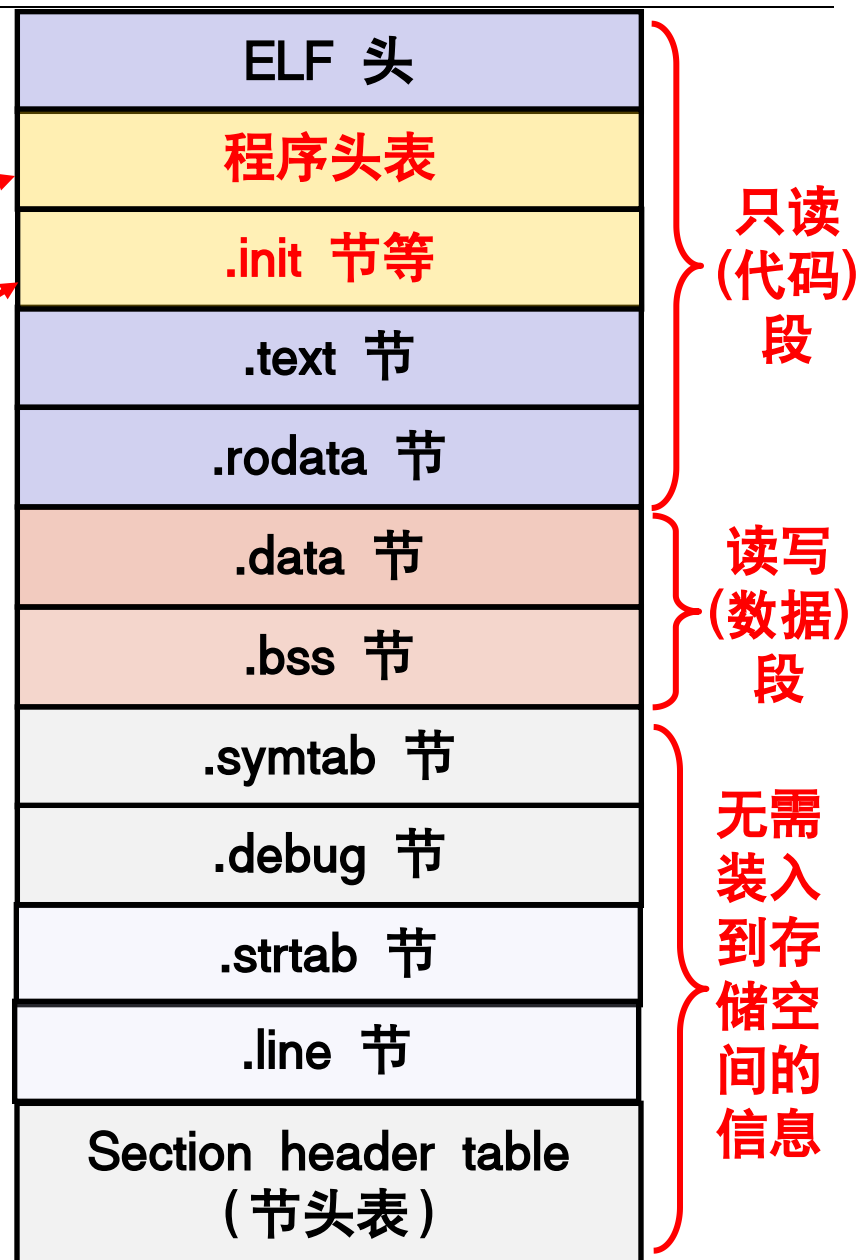


# 可执行目标文件格式

- 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
- 通常有.init节和.fini节, 用于定义 \_init函数等, 用于可执行文件开始执行时的初始化工作, 并生成终止时要执行的指令代码
- 少了.rela.text和.rela.data等重定位信息节。(无需重定位)

ELF头、程序头表、.init节、.fini节、.text节和.rodata节等合起来可构成一个只读代码段; .data节和.bss节合起来可构成一个可读写数据段



# 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

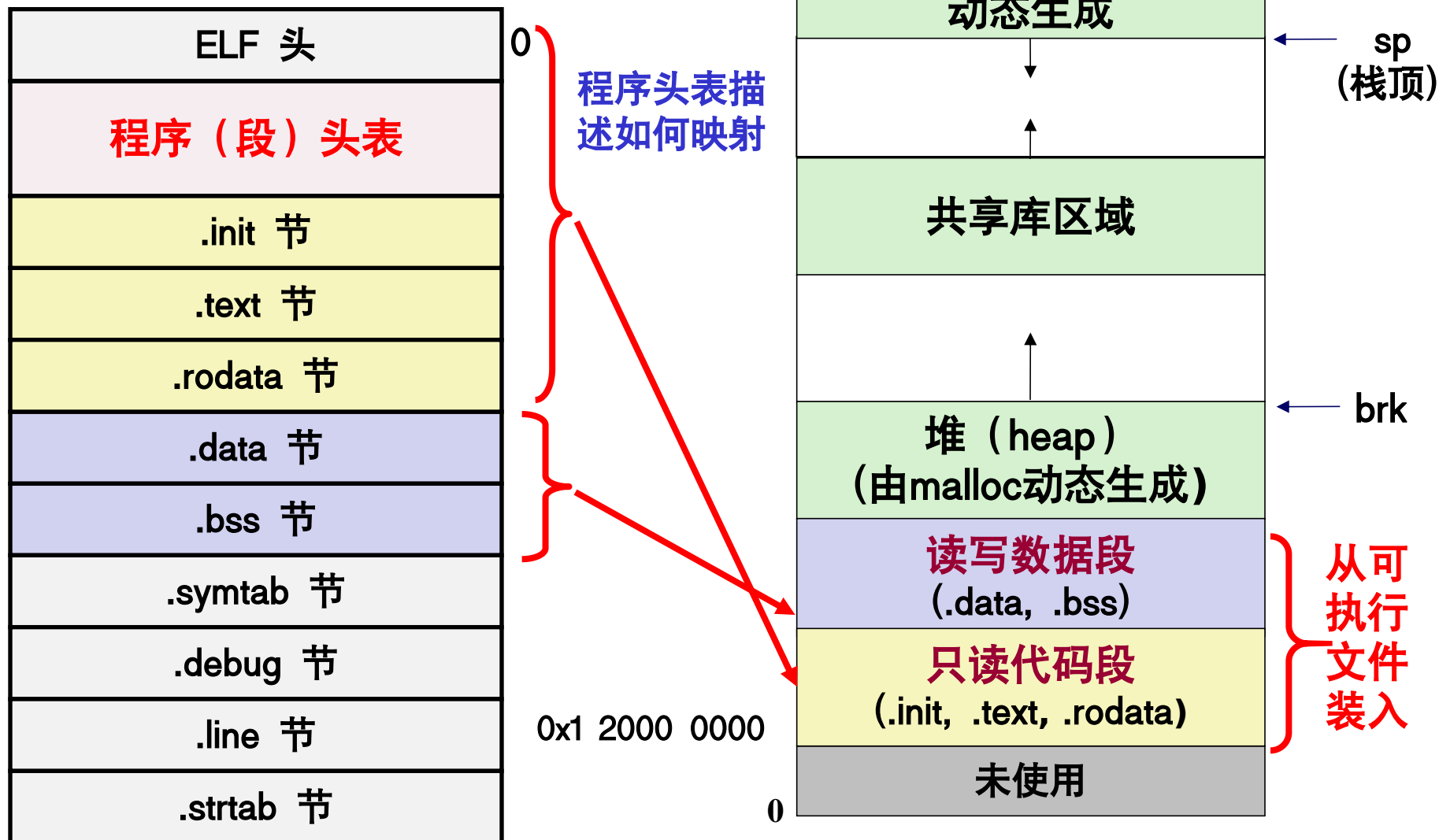
ELF 头	
00 00 00	程序头表
	.init 节
	.text 节
	.rodata 节
	.data 节
	.bss 节
	.symtab 节
	.debug 节
	.strtab 节
	.line 节
	Section header table (节头表)

8x32B

29x40B

# LA64 ABI规定的存储器映像

可执行文件与虚拟地址空间间的  
存储器映像（memory  
mapping）由ABI规范定义



# 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)

Program Headers: \$ readelf -l main

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:

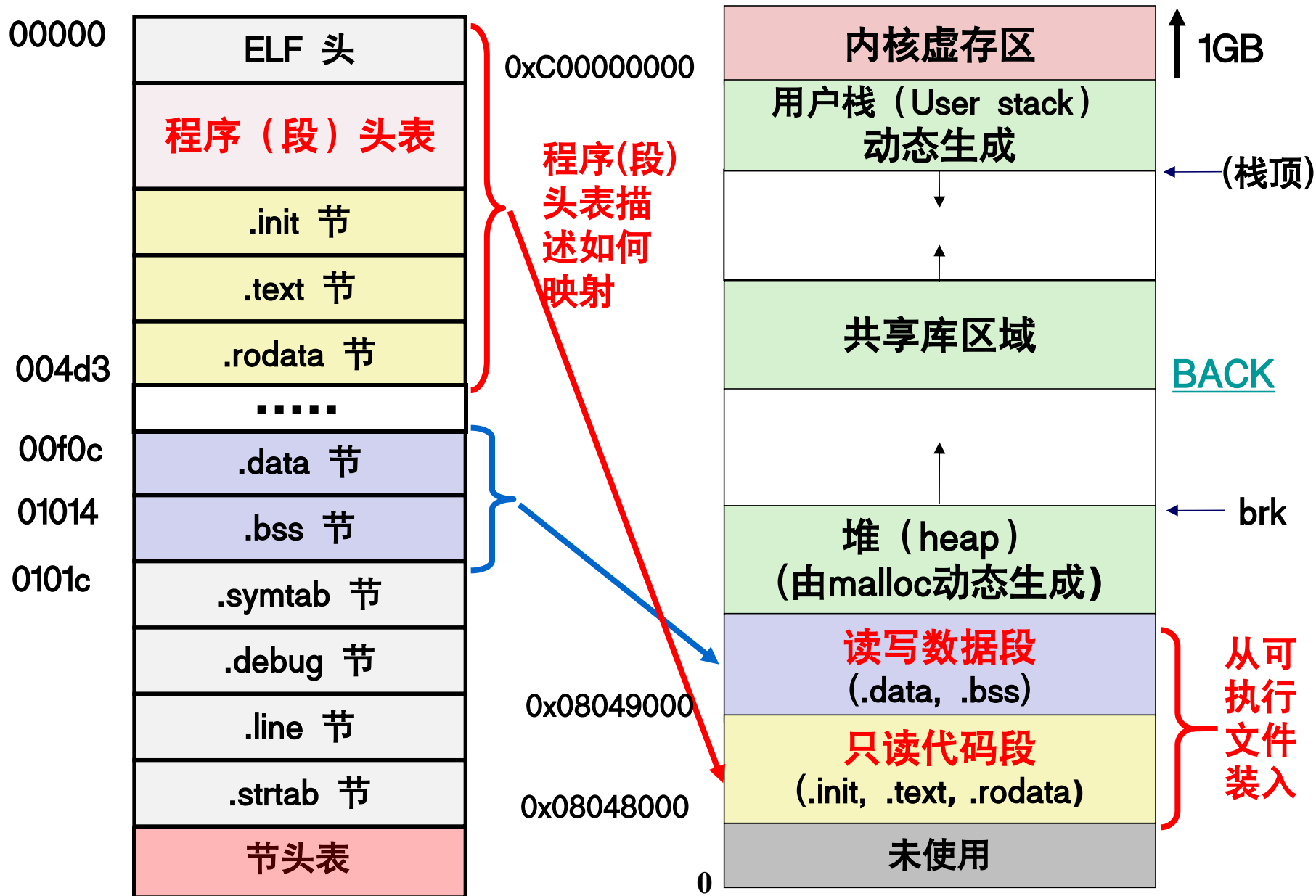
Type	Offset	VirtAddr	PhysAddr	FileSiz	MemSiz	Flg	Align
PHDR	0x000034	0x08048034	0x08048034	0x00100	0x00100	R E	0x4
INTERP	0x000134	0x08048134	0x08048134	0x00013	0x00013	R	0x1
[Requesting program interpreter: /lib/ld-linux.so.2]							
LOAD	0x000000	0x08048000	0x08048000	0x004d4	0x004d4	R E	0x1000
LOAD	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<sup>12</sup>=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



# 可执行文件中的程序头表

用“`readelf -l test`”命令显示LA64中可执行文件test的程序头表部分信息

Elf file type is EXEC (Executable file)

Entry point 0x1200004e0

There are 9 program headers, starting at offset 64

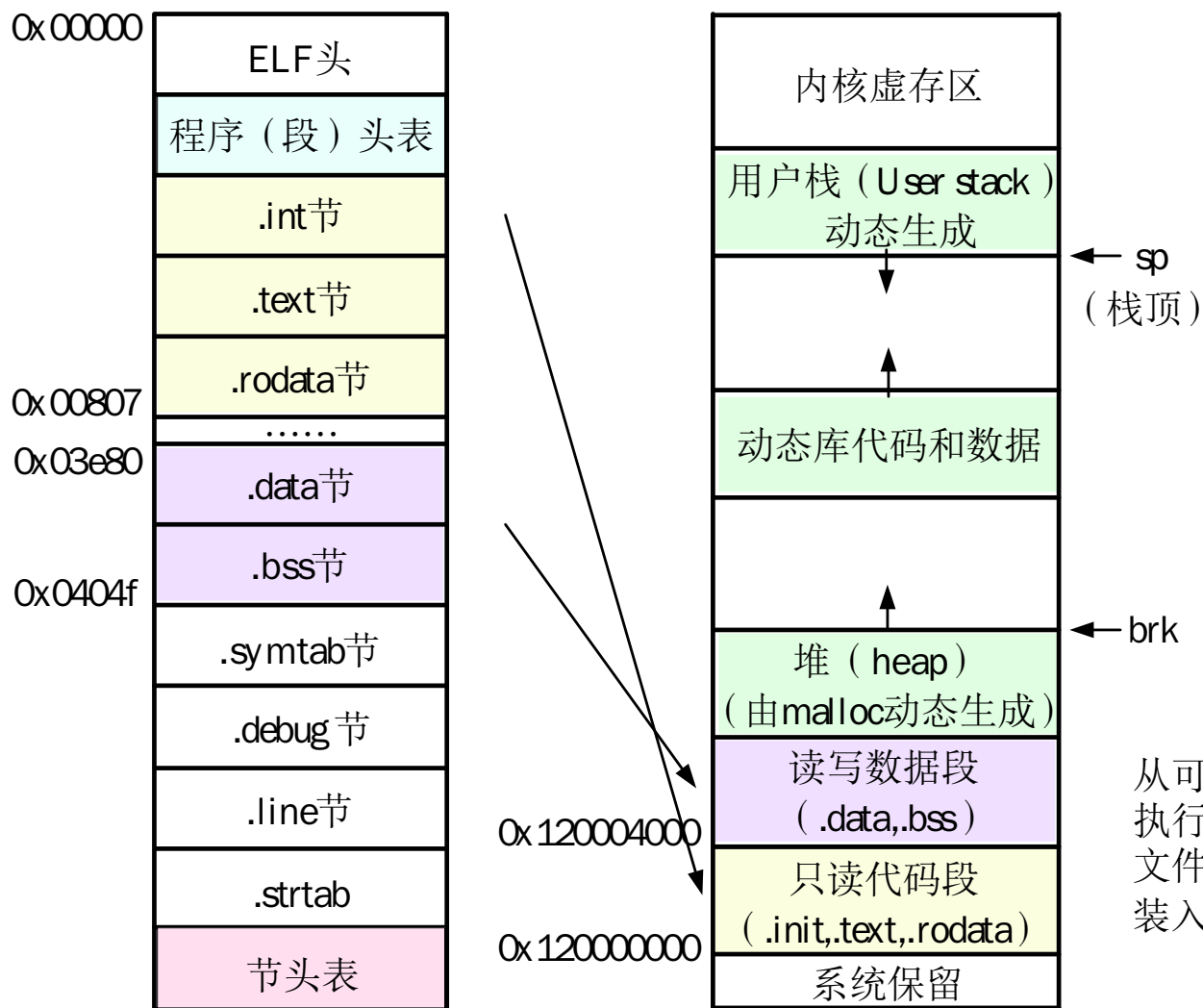
Program Headers:

Type	Offset	VirtAddr	PhysAddr	FileSiz	MemSiz	Flags	Align
PHDR	0x00040	0x0120000040	0x0120000040	0x001f8	0x01f8	R	0x8
INTERP	0x00238	0x0120000238	0x0120000238	0x0000f	0x0000f	R	0x1
[Requesting program interpreter: /lib64/ld.so.1]							
LOAD	0x00000	0x120000000	0x120000000	0x0808	0x0808	R E	0x4000
LOAD	0x03e80	0x120007e80	0x120007e80	0x01d0	0x01e0	RW	0x4000
DYNAMIC	0x03e90	0x120007e90	0x120007e90	0x0170	0x0170	RW	0x8
NOTE	0x00248	0x120000248	0x120000248	0x0044	0x0044	R	0x4
GNU_EH_FRAME	0x00780	0x120000780	0x120000780	0x001c	0x001c	R	0x4
GNU_STACK	0x00000	0x000000000	0x000000000	0x0000	0x0000	RW	0x10
GNU_RELRO	0x03e80	0x120007e80	0x120007e80	0x0180	0x0180	R	0x1



# 可执行文件的存储器映像

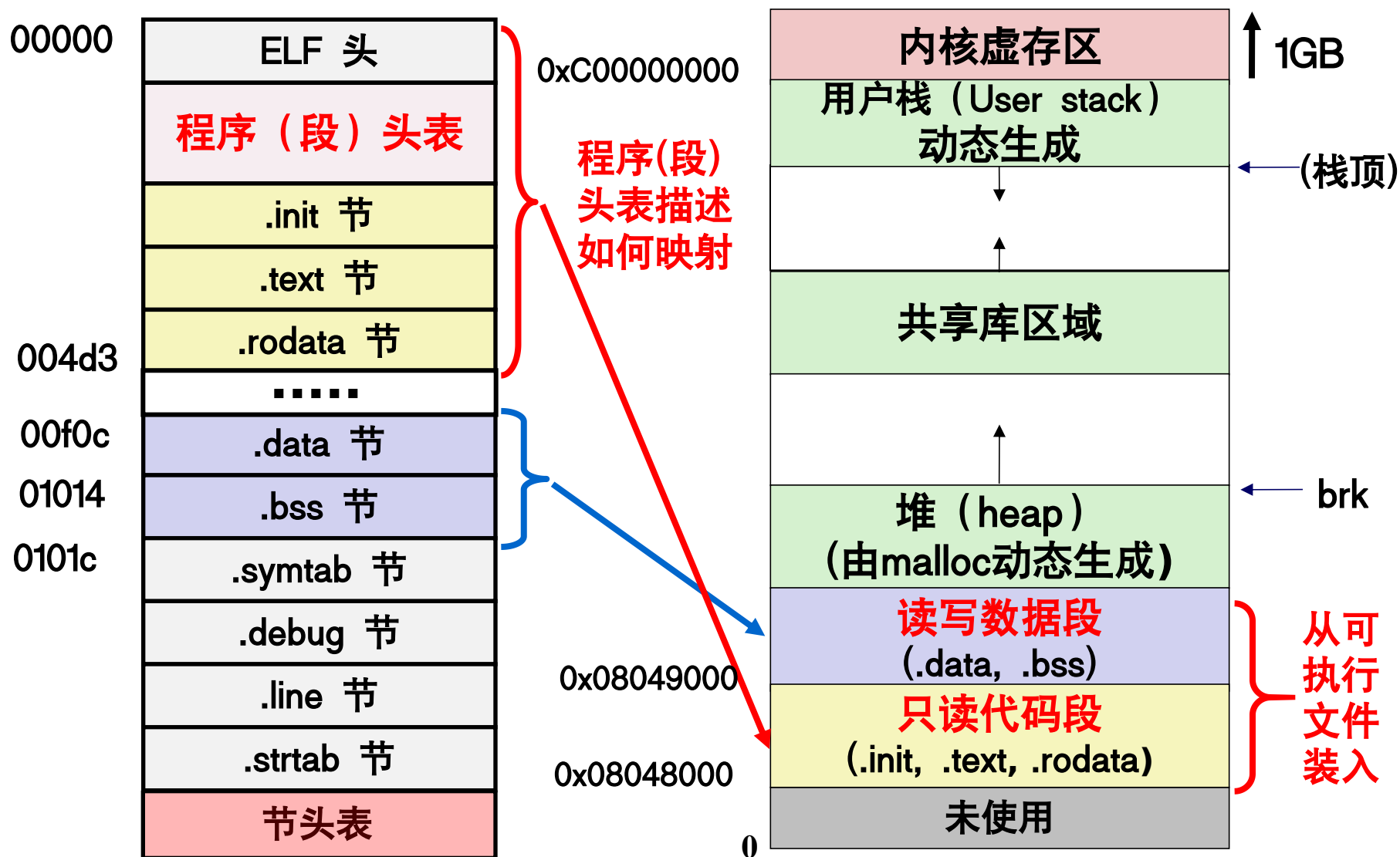
根据上页可执行文件test的程序头表信息画出其存储器映像如下



test中最开始的0x00808字节映射到虚拟地址0x1 2000 0000开始的只读代码段，按0x4000=16KB对齐，只读代码段只有0x808B，因此，读写数据段从0x1 2000 4000开始。为提升存储访问性能，test中0x03e80到0x0404f之间的可装入段映射到虚拟地址0x1 2000 4000+0x03e80=0x12000 7e80开始的位置，该装入段包含.data节和.bss节，在虚拟地址空间中需给.bss节中定义的变量分配空间，且初始值为0。.bss节的起始虚拟地址为0x12000 4000 + 0x03e80+0x1d0 = 0x1 2000 8050。

# 要求思考的问题

- 你会实现自己的readelf (-h/-S/-l) 吗? objdump呢?



# 程序的链接与加载执行

---

- 分以下四个部分介绍
  - 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
  - 第三讲：动态链接和库打桩机制
    - 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、位置无关代码的生成、库打桩机制
  - 第四讲：可执行文件的加载和执行
    - 可执行文件的加载
    - 程序和指令的执行过程

# 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 (全局符号)

- 由模块m定义并能被其他模块引用的符号，包括非static 的函数名和非static的全局变量名（指不带static的全局变量）

如，main.c 中的全局变量名buf

- External symbols (外部符号)

- 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 (本地符号、局部符号)

- 由模块m定义和引用的带static的函数名和变量名。因其生存期为整个程序运行过程，故并不分配在栈中，而是分配在静态数据区，即在.data节或.bss节中分配空间。

如，swap.c 中的static变量名bufp1

# 符号和符号解析

main.c

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

swap.c

```
extern int buf[];  
  
int *bufp0 = &buf[0];  
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？哪些是本地(局部)符号？

# 目标文件中的符号表

函数名在text节中

变量名在data节或  
bss节中

.symtab 节记录符号表信息，是一个结构数组

```
typedef struct {
```

```
    Elf32_Word  st_name;      /*符号对应字符串在strtab节中的偏移量*/
```

```
    Elf32_Addr  st_value;     /*在对应节中的偏移量，可执行文件中是虚拟地址
```

```
*/
```

```
    Elf32_Word  st_size;      /*符号对应目标所占字节数*/
```

```
    unsigned char st_info;    /*符号对应目标的类型和绑定属性*/
```

```
    unsigned char st_other;   /*符号的可见性*/
```

```
    Elf32_Half  st_shndx;     /*符号所在节在节头表中的索引*/
```

```
} Elf32_Sym;
```

符号类型可以是未指定 (NOTYPE)、变量 (OBJECT)、函数 (FUNC)、节等。为“节”时，其表项用于重定位。绑定属性可以是本地 (LOCAL)、全局 (GLOBAL)、弱 (WEAK) 等。其中，本地符号外部模块不可见，名称相同的本地符号可存在于多个文件中；全局符号对于所有被合并的目标文件都可见；弱符号是通过GCC扩展的属性指示符\_\_attribute\_\_((weak))指定的符号，它与全局符号一样，对于所有被合并目标文件都可见。

函数大小或变量长度

在节头表中无索引的节称为伪节：ABS表示不该被重定位；UND表示未定义；COM表示未初始化变量，称为COMMON符号，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
9:	0	8	Object	Global	0	3	buf
10:	0	44	Func	Global	0	1	main
11:	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44B；

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
9:	0	8	Object	Global	0	3	bufp0
10:	0	0	Notype	Global	0	UND	buf
11:	8	8	Object	Global	0	COM	bufp1
12:	0	124	Func	Global	0	1	swap

bufp1是未分配地址且未初始化的全局变量(Ndx=COM)，按8B对齐且占8B



# 目标文件中的符号表

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

```
int x=100;
```

```
void main()
```

```
{  static int x;  
    printf("x=%d\n",x);  
}
```

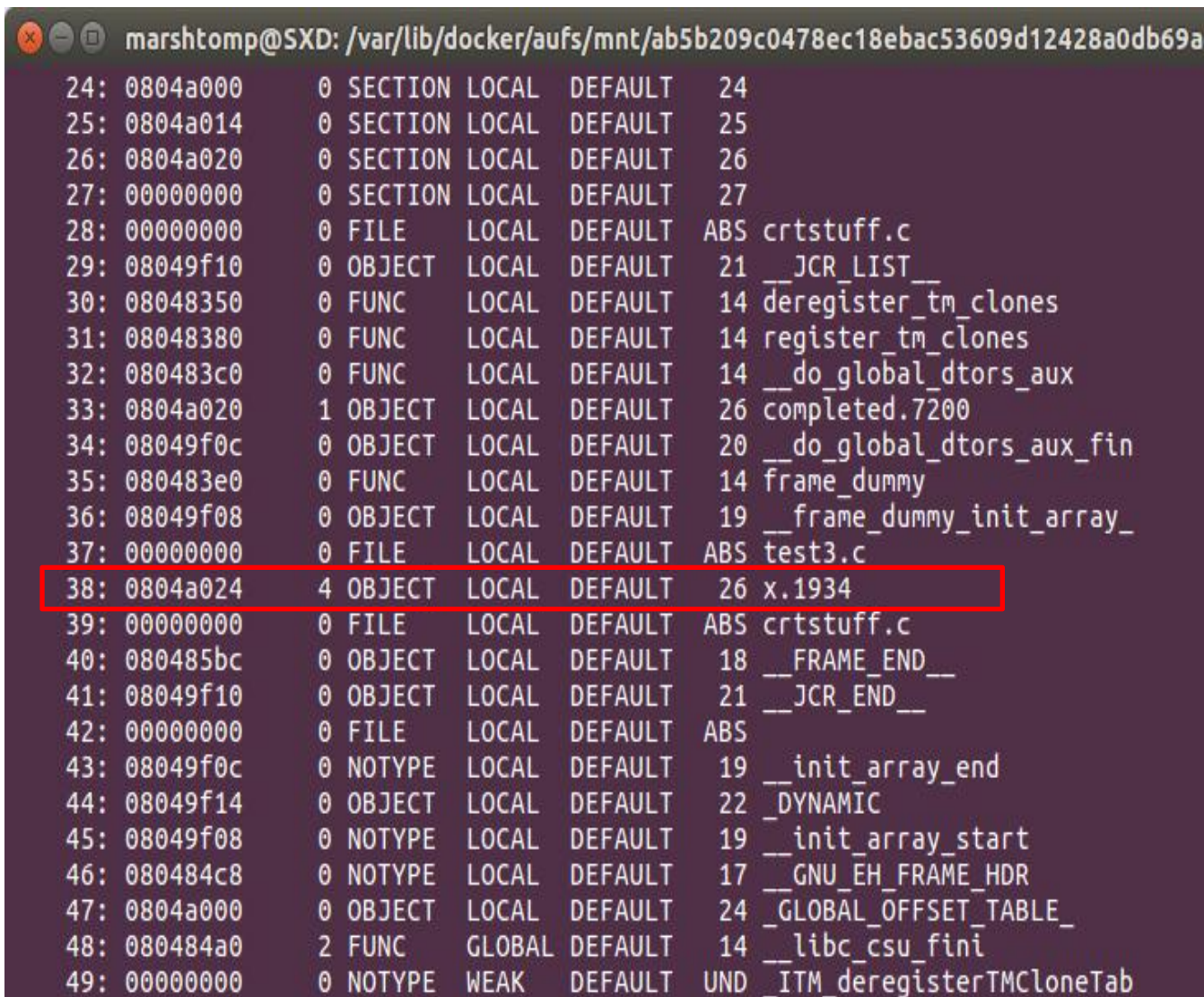
根据反汇编代码可知:

静态变量x存储在  
0x804a024处, 对应.bss  
节所在位置。

调用printf函数时, 该位置  
上的值作为参数传递给了  
printf函数。

存储在.bss节的static变量  
被自动初始化为0, 因此  
输出结果为0。

用readelf -a 查看符号表如下:



24:	0804a000	0	SECTION	LOCAL	DEFAULT	24	
25:	0804a014	0	SECTION	LOCAL	DEFAULT	25	
26:	0804a020	0	SECTION	LOCAL	DEFAULT	26	
27:	00000000	0	SECTION	LOCAL	DEFAULT	27	
28:	00000000	0	FILE	LOCAL	DEFAULT	ABS	crtstuff.c
29:	08049f10	0	OBJECT	LOCAL	DEFAULT	21	__JCR_LIST__
30:	08048350	0	FUNC	LOCAL	DEFAULT	14	deregister_tm_clones
31:	08048380	0	FUNC	LOCAL	DEFAULT	14	register_tm_clones
32:	080483c0	0	FUNC	LOCAL	DEFAULT	14	__do_global_ctors_aux
33:	0804a020	1	OBJECT	LOCAL	DEFAULT	26	completed.7200
34:	08049f0c	0	OBJECT	LOCAL	DEFAULT	20	__do_global_ctors_aux_fin
35:	080483e0	0	FUNC	LOCAL	DEFAULT	14	frame_dummy
36:	08049f08	0	OBJECT	LOCAL	DEFAULT	19	__frame_dummy_init_array_
37:	00000000	0	FILE	LOCAL	DEFAULT	ABS	test3.c
38:	0804a024	4	OBJECT	LOCAL	DEFAULT	26	x.1934
39:	00000000	0	FILE	LOCAL	DEFAULT	ABS	crtstuff.c
40:	080485bc	0	OBJECT	LOCAL	DEFAULT	18	__FRAME_END__
41:	08049f10	0	OBJECT	LOCAL	DEFAULT	21	__JCR_END__
42:	00000000	0	FILE	LOCAL	DEFAULT	ABS	
43:	08049f0c	0	NOTYPE	LOCAL	DEFAULT	19	__init_array_end
44:	08049f14	0	OBJECT	LOCAL	DEFAULT	22	__DYNAMIC
45:	08049f08	0	NOTYPE	LOCAL	DEFAULT	19	__init_array_start
46:	080484c8	0	NOTYPE	LOCAL	DEFAULT	17	__GNU_EH_FRAME_HDR
47:	0804a000	0	OBJECT	LOCAL	DEFAULT	24	__GLOBAL_OFFSET_TABLE__
48:	080484a0	2	FUNC	GLOBAL	DEFAULT	14	__libc_csu_fini
49:	00000000	0	NOTYPE	WEAK	DEFAULT	UND	__ITM_deregisterTMCloneTab



# 目标文件中的符号表

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

```
int x=100;
```

```
void main()
```

```
{  
    printf("x=%d\n",x);  
}
```

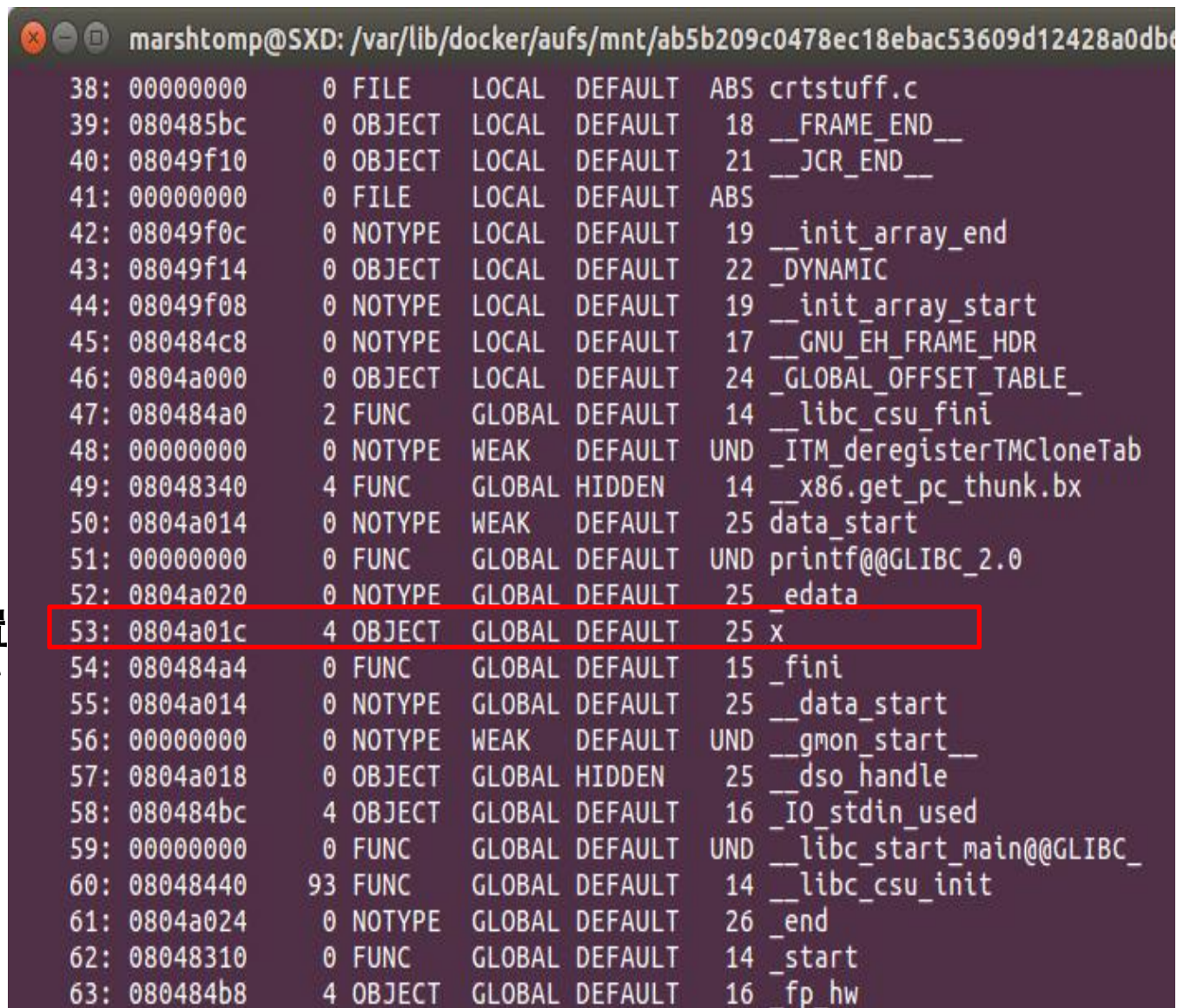
根据反汇编代码可知:

全局变量x存储在  
0x804a01c处, 对应.data  
节所在位置。

调用printf函数时, 该位置  
上的值作为参数传递给了  
printf函数。

因此输出结果为100。

用readelf -a 查看符号表如下:



38:	00000000	0	FILE	LOCAL	DEFAULT	ABS	crtstuff.c
39:	080485bc	0	OBJECT	LOCAL	DEFAULT	18	__FRAME_END__
40:	08049f10	0	OBJECT	LOCAL	DEFAULT	21	__JCR_END__
41:	00000000	0	FILE	LOCAL	DEFAULT	ABS	
42:	08049f0c	0	NOTYPE	LOCAL	DEFAULT	19	__init_array_end
43:	08049f14	0	OBJECT	LOCAL	DEFAULT	22	__DYNAMIC
44:	08049f08	0	NOTYPE	LOCAL	DEFAULT	19	__init_array_start
45:	080484c8	0	NOTYPE	LOCAL	DEFAULT	17	__GNU_EH_FRAME_HDR
46:	0804a000	0	OBJECT	LOCAL	DEFAULT	24	__GLOBAL_OFFSET_TABLE__
47:	080484a0	2	FUNC	GLOBAL	DEFAULT	14	__libc_csu_fini
48:	00000000	0	NOTYPE	WEAK	DEFAULT	UND	_ITM_deregisterTMCloneTab
49:	08048340	4	FUNC	GLOBAL	HIDDEN	14	__x86.get_pc_thunk.bx
50:	0804a014	0	NOTYPE	WEAK	DEFAULT	25	data_start
51:	00000000	0	FUNC	GLOBAL	DEFAULT	UND	printf@@GLIBC_2.0
52:	0804a020	0	NOTYPE	GLOBAL	DEFAULT	25	edata
53:	0804a01c	4	OBJECT	GLOBAL	DEFAULT	25	x
54:	080484a4	0	FUNC	GLOBAL	DEFAULT	15	_fini
55:	0804a014	0	NOTYPE	GLOBAL	DEFAULT	25	__data_start
56:	00000000	0	NOTYPE	WEAK	DEFAULT	UND	__gmon_start__
57:	0804a018	0	OBJECT	GLOBAL	HIDDEN	25	__dso_handle
58:	080484bc	4	OBJECT	GLOBAL	DEFAULT	16	_IO_stdin_used
59:	00000000	0	FUNC	GLOBAL	DEFAULT	UND	__libc_start_main@@GLIBC_2.3.4
60:	08048440	93	FUNC	GLOBAL	DEFAULT	14	__libc_csu_init
61:	0804a024	0	NOTYPE	GLOBAL	DEFAULT	26	_end
62:	08048310	0	FUNC	GLOBAL	DEFAULT	14	_start
63:	080484b8	4	OBJECT	GLOBAL	DEFAULT	16	_fp_hw

# 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（局部）符号**在本模块定义并引用，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**（包括外部和内部定义的）解析涉及多个模块，故较复杂



符号解析也称**符号绑定**

“符号的定义”其实是什么？

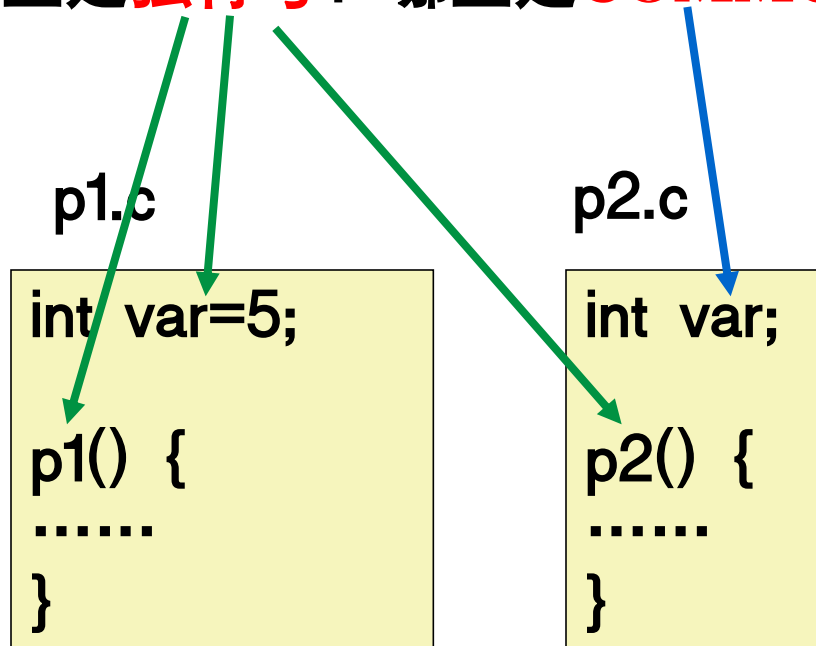
指被分配了存储空间。为函数名时，指代码所在区；为变量名时，指所占的静态数据区。

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

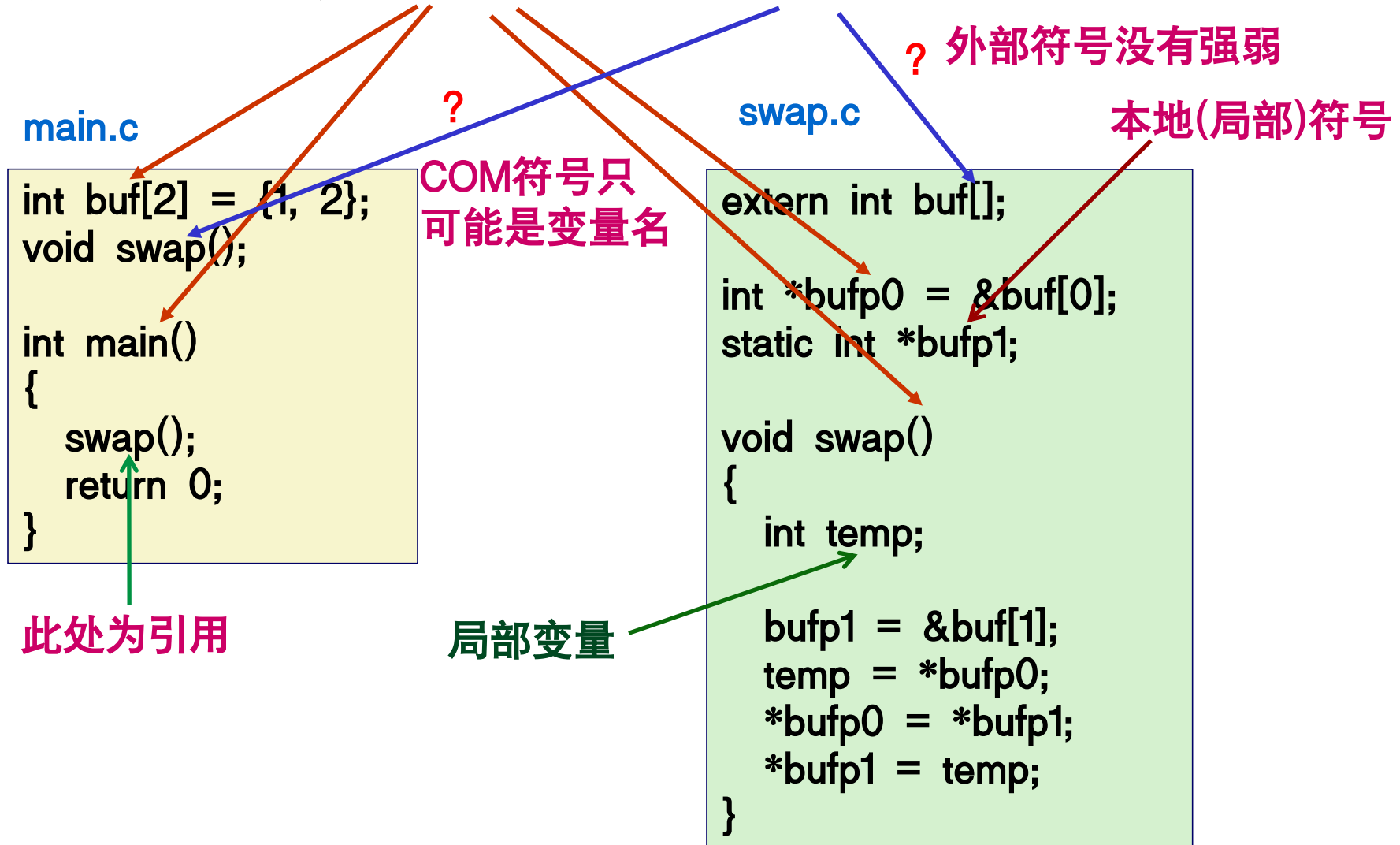
- 全局符号的特性
  - 强符号**：函数、.data节中具有特定初始值的全局变量名、.bss节中被初始化为0的全局变量名
  - 弱符号**：绑定属性为WEAK的符号
  - COMMON符号**：COMMON伪节中的未初始化全局变量名

以下符号哪些是**强符号**？ 哪些是**COMMON符号**？



# 全局符号的符号解析

以下符号哪些是**强符号**? 哪些是**COMMON**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: 强符号只能定义一次，否则链接错误

Rule 2: 若出现一次强符号定义和多次COMMON符号或弱符号定义，则按强符号定义为准

Rule 3: 若同时出现COMMON符号和弱符号定义，则按COMMON符号定义为准

Rule 4: 若一个COMMON符号出现多次，则以占空间最大的一个为准。

Rule 5: 若使用命令 `gcc -fno-common` 链接，则不考虑COMMON符号，相当于将COMMON符号作为强符号处理。

# 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有一次强定义

p1有一次强定义

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COMMON定义  
z两次COMMON定义（以占空间更大的main模块中的z为准）

p1一次强定义

main一次强定义

```
int y;
```

```
short 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？

main.c

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

**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.c

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

**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 0111111111 0...0B

=3FF0 0000 0000 0000H



# 多重定义符号的解析举例

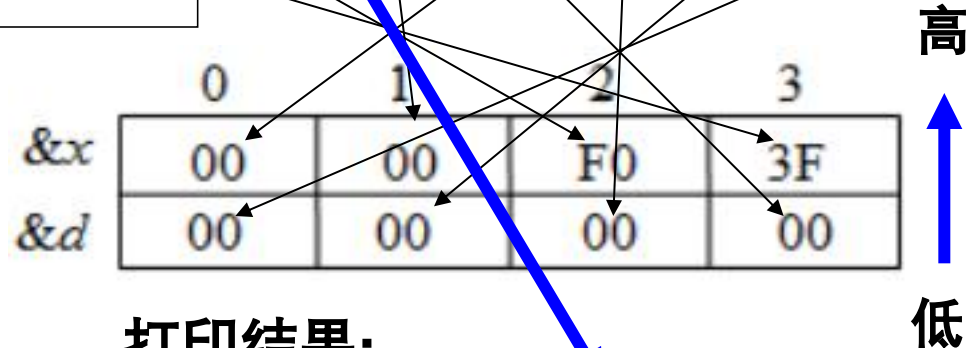
main.c

```
.....  
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

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

$$=1024 \times 1024 \times 1023$$

$$=1\ 072\ 693\ 248$$

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
- 使用选项命令-fno-common，告诉链接器在遇到多重定义的全局符号时，触发一个错误，或者使用-Werror选项命令，将所有警告变为错误。

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？

---

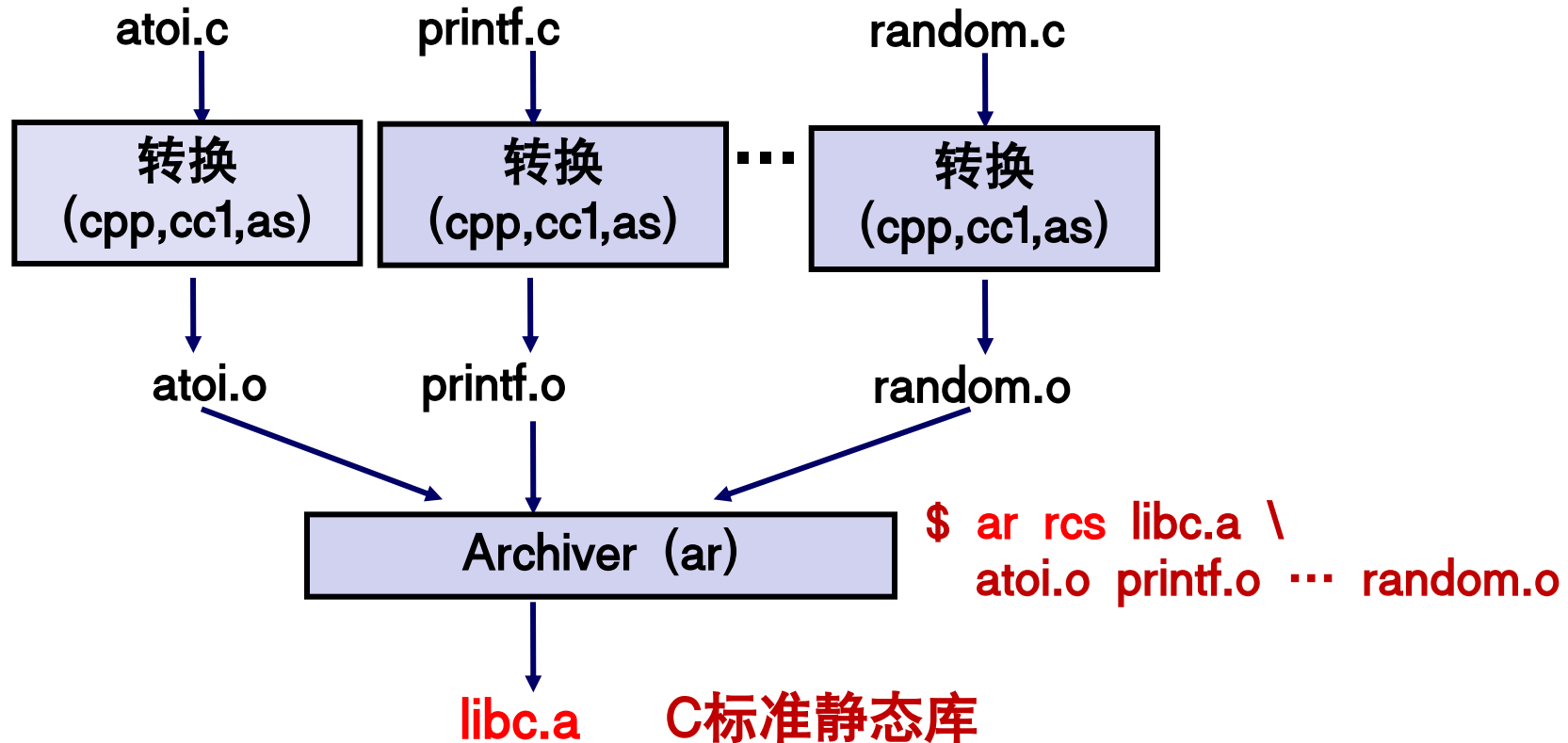
- 许多函数无需自己写，可使用共享库函数
  - 如数学库，输入/输出库，存储管理库，字符串处理等
- 避免以下两种极端做法
  - 将所有函数都放在一个源文件中
    - 修改一个函数需要对所有函数重新编译
    - 时间和空间两方面的效率都不高
  - 一个源文件中仅包含一个函数
    - 需要程序员显式地进行链接
    - 效率高，但模块太多，故太繁琐

# 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。

在gcc命令行中无需明显指定C标准库libc.a(默认库)

# 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      libc.a无需明显指出!
$ 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？

# 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
```

调用关系: main→myfunc1→printf

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myfunc1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。

main.c

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

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，D中符号唯一。



# 链接器中符号解析的全过程

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

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

main→myfunc1→printf

main.c

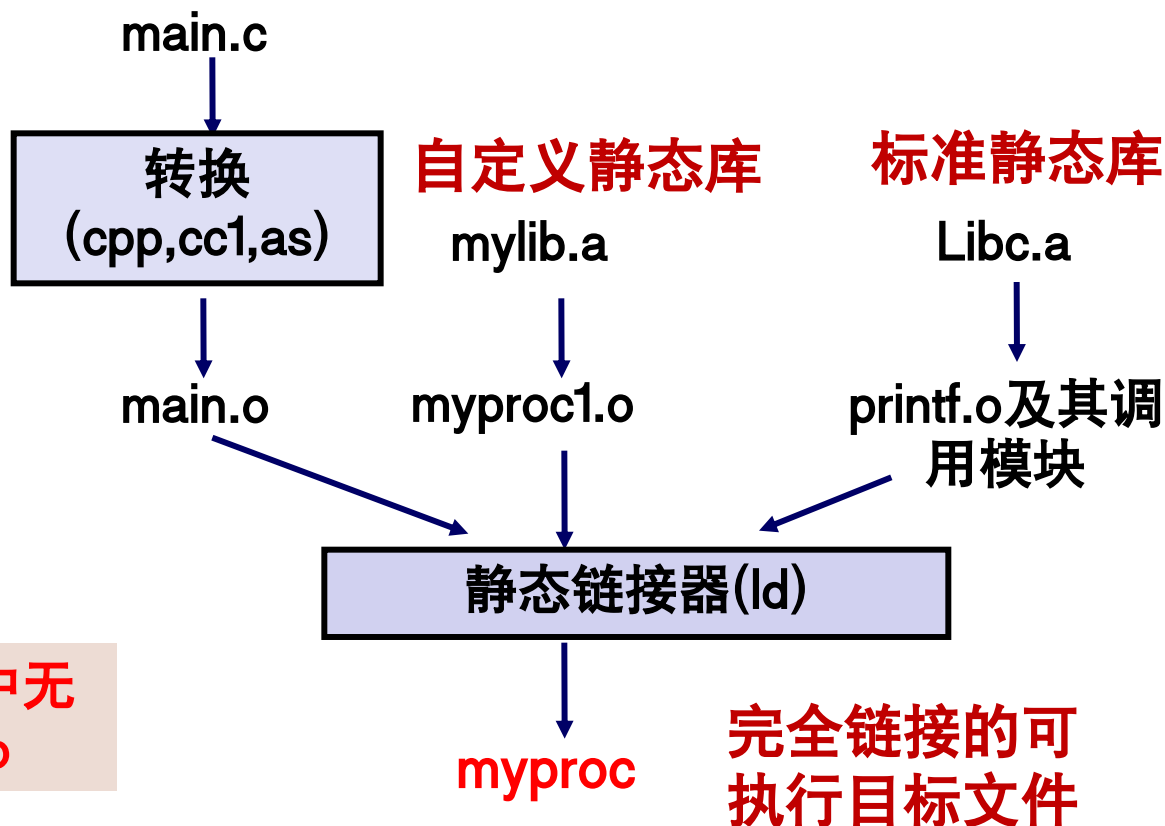
```
void myfunc1(viod);  
int main()  
{  
    myfunc1();  
    return 0;  
}
```

解析结果:

注意: E中无  
myproc2.o

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U中一开始为空, 所以mylib中的myproc1.o和myproc2.o都被丢弃。

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
```

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

```
$ gcc -L. -lmine libtest.o
```

```
libtest.o: In function `main':
```

```
libtest.o(.text+0x4): undefined reference to `libfun'
```

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！

# 链接顺序问题举例

---

- 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:

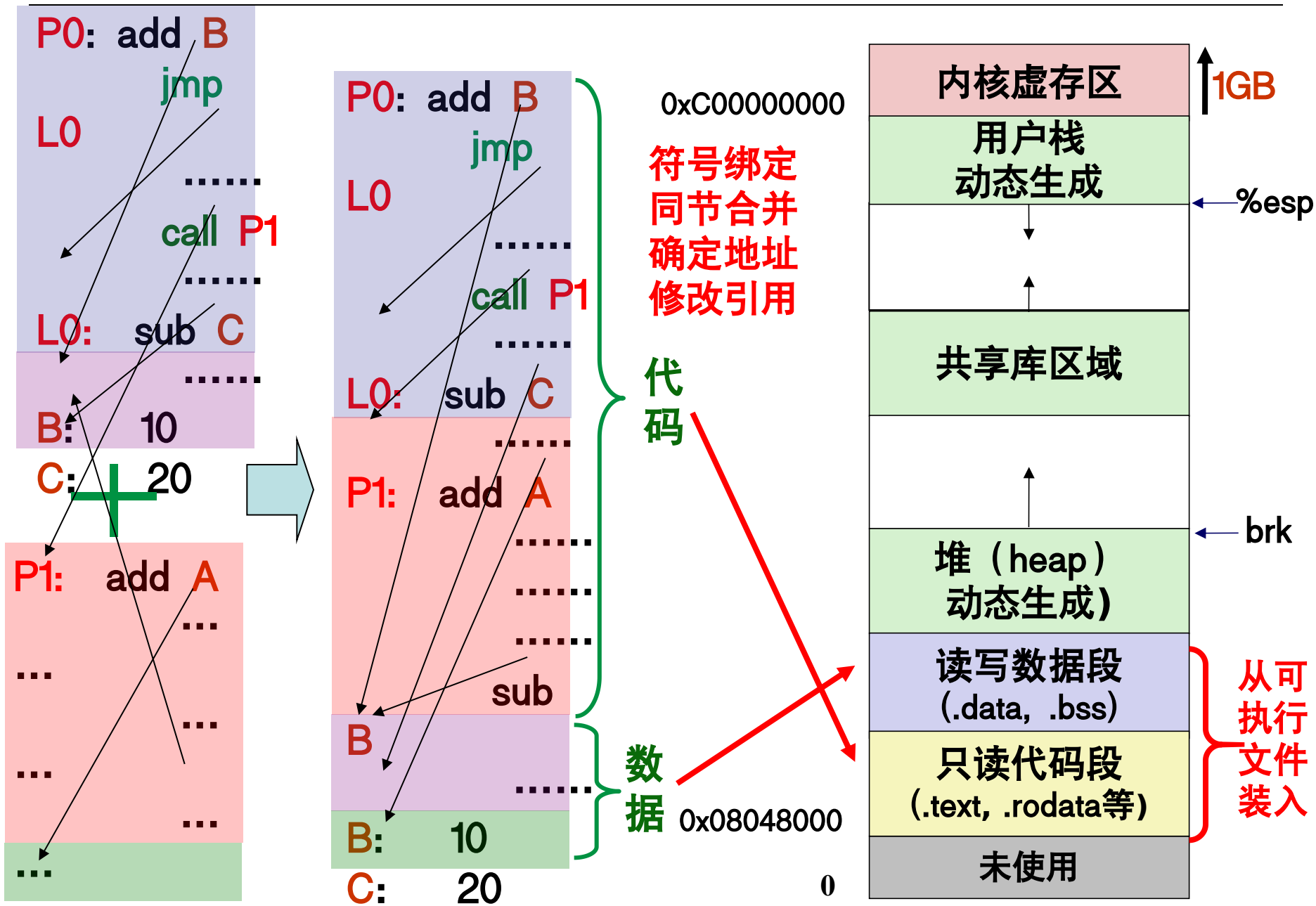
func.o → libx.a 和 liby.a 中的函数

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

# 链接操作的步骤



# 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.rela.data和.rela.text节中保存的重定位信息

# IA-32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
- IA-32中重定位条目格式如下：

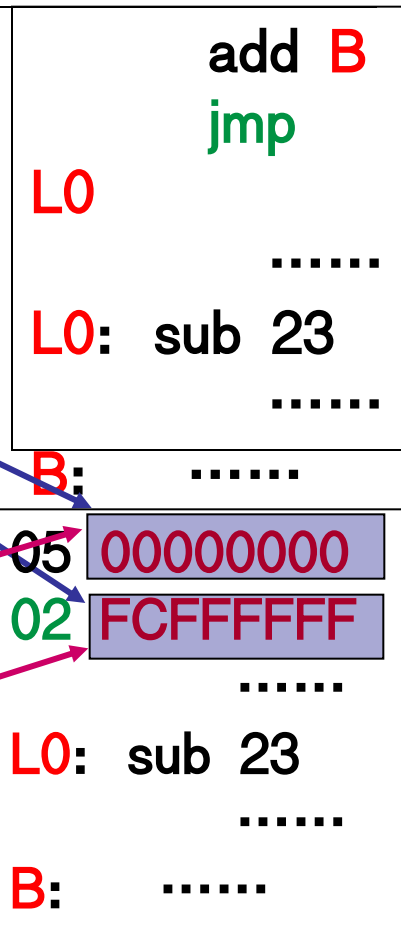
```
typedef struct {  
    int offset;                /*节内偏移*/  
    int symbol:24, /*所绑定符号*/  
        type: 8;             /*重  
定位类型*/  
} Elf32_Rel;
```

- 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

重定位表的信息可以用命令“readelf -r”来显示

例如，在rel\_text节中有重定位条目如下

offset: 0x1	offset: 0x6
symbol: B	symbol: L0
type: R_386_32	type: R_386_PC32



问题：重定位条目和汇编后的  
机器代码在何种目标文件中？

在可重定位目标  
(.o) 文件中！

# 重定位信息

- 重定位条目格式有两种，一种是不带加数的**Rel类型**，另一种是带加数的**Rela类型**。
- **64位**系统中表项的数据结构如下。

```
typedef struct {  
    Elf64_Addr  r_offset;    /*节内偏移*/  
    uint64_t    r_info;      /*引用符号和重定位类型*/  
} Elf64_Rel;
```

```
typedef struct {  
    Elf64_Addr  r_offset;    /*节内偏移*/  
    uint64_t    r_info;      /*引用符号和重定位类型*/  
    int64_t     r_addend;  
} Elf64_Rela;
```

- Rela类型中的r\_addend给出一个加数，用于计算重定位后的符号引用地址
- **IA-32**只使用Rel类型（采用上一页中的**Elf32\_Rel**格式）
- **LoongArch (LA)**、x86-64、SPARC、RISC-V等只使用Rela类型
- 根据LoongArch相关ABI规范，其重定位类型有**100种**左右  
（LA架构中重定位过程较复杂，下面简介IA-32中的重定位过程）

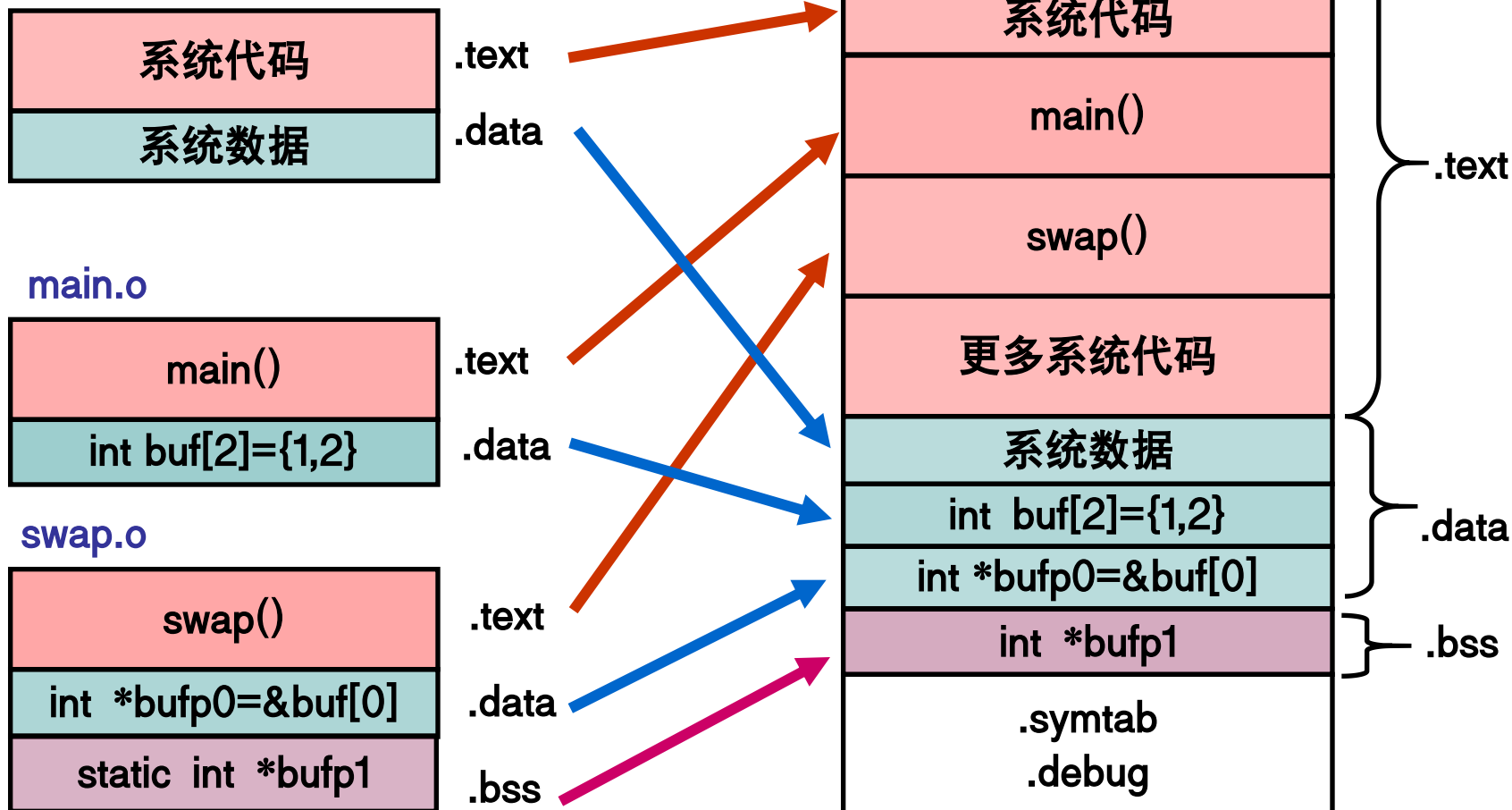


# 重定位过程

链接本质：合并相同的“节”

可执行目标文件

可重定位目标文件



合并后的可执行文件中，需要对引用符号进行重定位。

# 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

# IA-32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;
}
```

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）

# 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。

## 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。

## 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
```

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	Object	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,  $PC = 0x8048380 + 0x07 - \text{init}$
- $PC = 0x8048380 + 0x07 - (-4) = 0x804838b$
- 重定位值=转移目标地址-PC= $0x8048394 - 0x804838b = 0x9$
- call指令的机器代码为 “e8 09 00 00 00”

PC相对地址方式下, 重定位值计算公式为:

$$\text{ADDR}(r\_sym) - ( ( \text{ADDR}(.text) + r\_offset ) - \text{init} )$$

引用目标处

call指令下条指令地址

即当前PC的值

Disassembly of section .text:

00000000 <main>:

.....

6: e8 fc ff ff ff

call 7 <main+0x7>

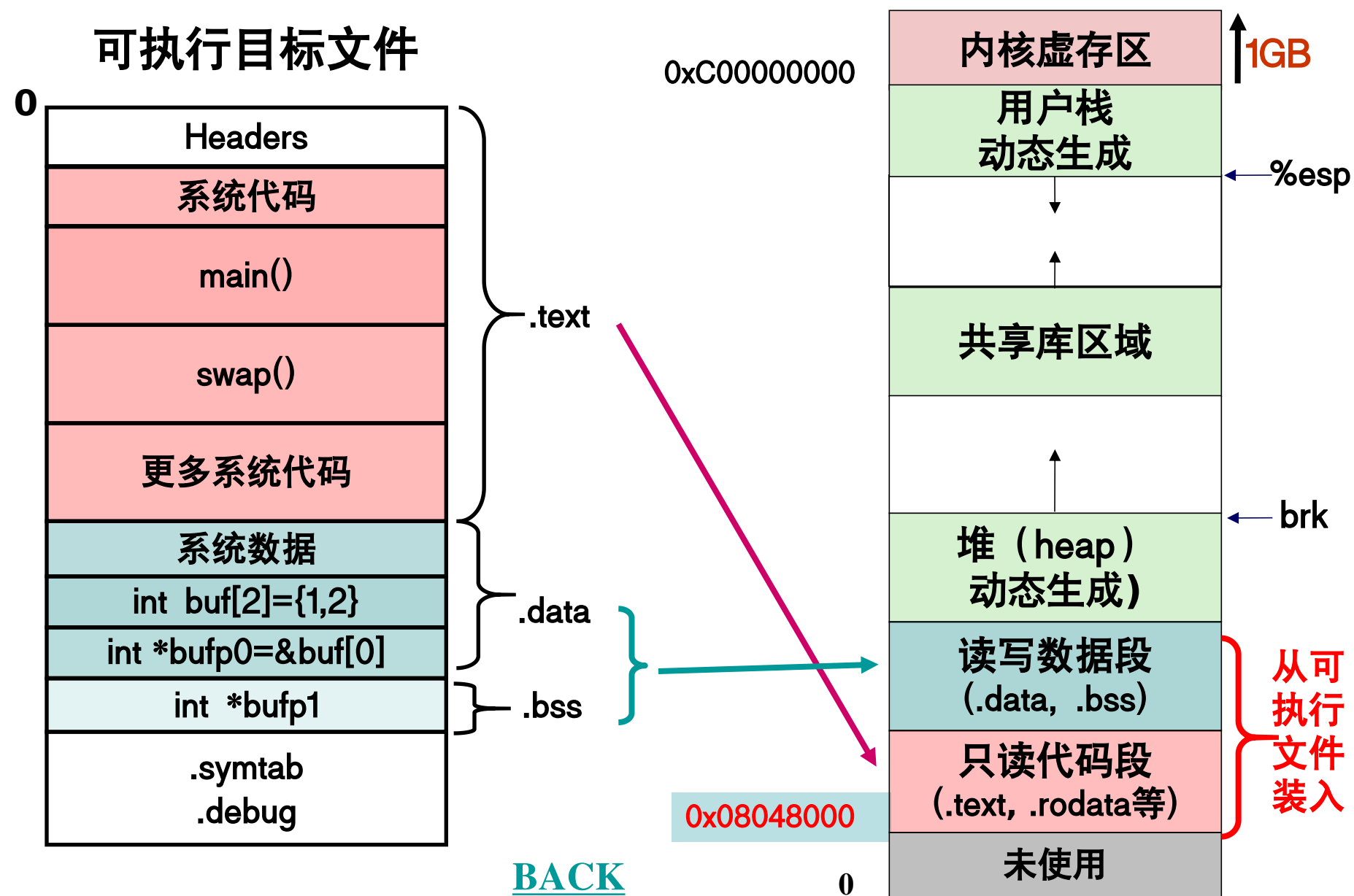
7: R\_386\_PC32 swap

重定位值

值为-4

SKIP

# 确定定义符号的地址



# 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.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

swap.c

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

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	Object	Global	0	3	bufp0
9:	0	0	Notype	Global	0	UND	buf
10:	0	36	Func	Global	0	1	swap
11:	4	4	Object	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:
  - 合并后buf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:	<u>c7 05 00 00 00 00 04</u>	movl	\$0x4,0x0
d:	<u>00 00 00</u>		
		8:	R_386_32 .bss
		c:	R_386_32 buf
10:	<u>a1 00 00 00 00</u>	mov	0x0,%eax
		11:	R_386_32 bufp0
15:	8b 00	mov	(%eax),%eax
17:	89 45 fc	mov	%eax,-0x4(%ebp)
1a:	<u>a1 00 00 00 00</u>	mov	0x0,%eax
		1b:	R_386_32 bufp0
1f:	<u>8b 15 00 00 00 00</u>	mov	0x0,%edx
		21:	R_386_32 .bss
25:	8b 12	mov	(%edx),%edx
27:	89 10	mov	%edx,(%eax)
29:	<u>a1 00 00 00 00</u>	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:	<u>c7 05 00 00 00 00 04</u>	movl \$0x4,0x0	
d:	<u>00 00 00</u>		
		8: R_386_32	.bss
		c: R_386_32	buf
10:	<u>a1 00 00 00 00</u>	mov 0x0,%eax	
		11: R_386_32	bufp0
15:	8b 00	mov (%eax),%eax	
17:	89 45 fc	mov %eax,-0x4(%ebp)	
1a:	<u>a1 00 00 00 00</u>	mov 0x0,%eax	
		1b: R_386_32	bufp0
1f:	<u>8b 15 00 00 00 00</u>	mov 0x0,%edx	
		21: R_386_32	.bss
25:	8b 12	mov (%edx),%edx	
27:	89 10	mov %edx,(%eax)	
29:	<u>a1 00 00 00 00</u>	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

你能写出该call指令的功能描述吗？

08048380 <main>:

```
8048380: 55          push %ebp
8048381: 89 e5       mov %esp,%ebp
8048383: 83 e4 f0    and $0xffffffff,%esp
8048386: e8 09 00 00 00 call 8048394 <swap>
804838b: b8 00 00 00 00 mov $0x0,%eax
```

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
```

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

SKIP

# 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,  $PC = 0x8048380 + 0x07 - \text{init}$
- $PC = 0x8048380 + 0x07 - (-4) = 0x804838b$
- 重定位值=转移目标地址-PC= $0x8048394 - 0x804838b = 0x9$

- call指令的机器代码为 “e8 09 00 00 00”

BACK

PC相对地址方式下, 重定位值计算公式为:

$$\text{ADDR}(r\_sym) - ( ( \text{ADDR}(.text) + r\_offset ) - \text{init} )$$

引用目标处

call指令下条指令地址

即当前PC的值

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

# 程序的链接与加载执行

---

- 分以下四个部分介绍
  - 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
  - 第三讲：动态链接和库打桩机制
    - 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、位置无关代码的生成、库打桩机制
  - 第四讲：可执行文件的加载和执行
    - 可执行文件的加载
    - 程序和指令的执行过程



# 动态链接的共享库 (Shared Libraries)

---

- 静态库有一些缺点:
  - 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，造成磁盘空间的极大浪费
  - 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
  - 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 (调用), 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
- 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

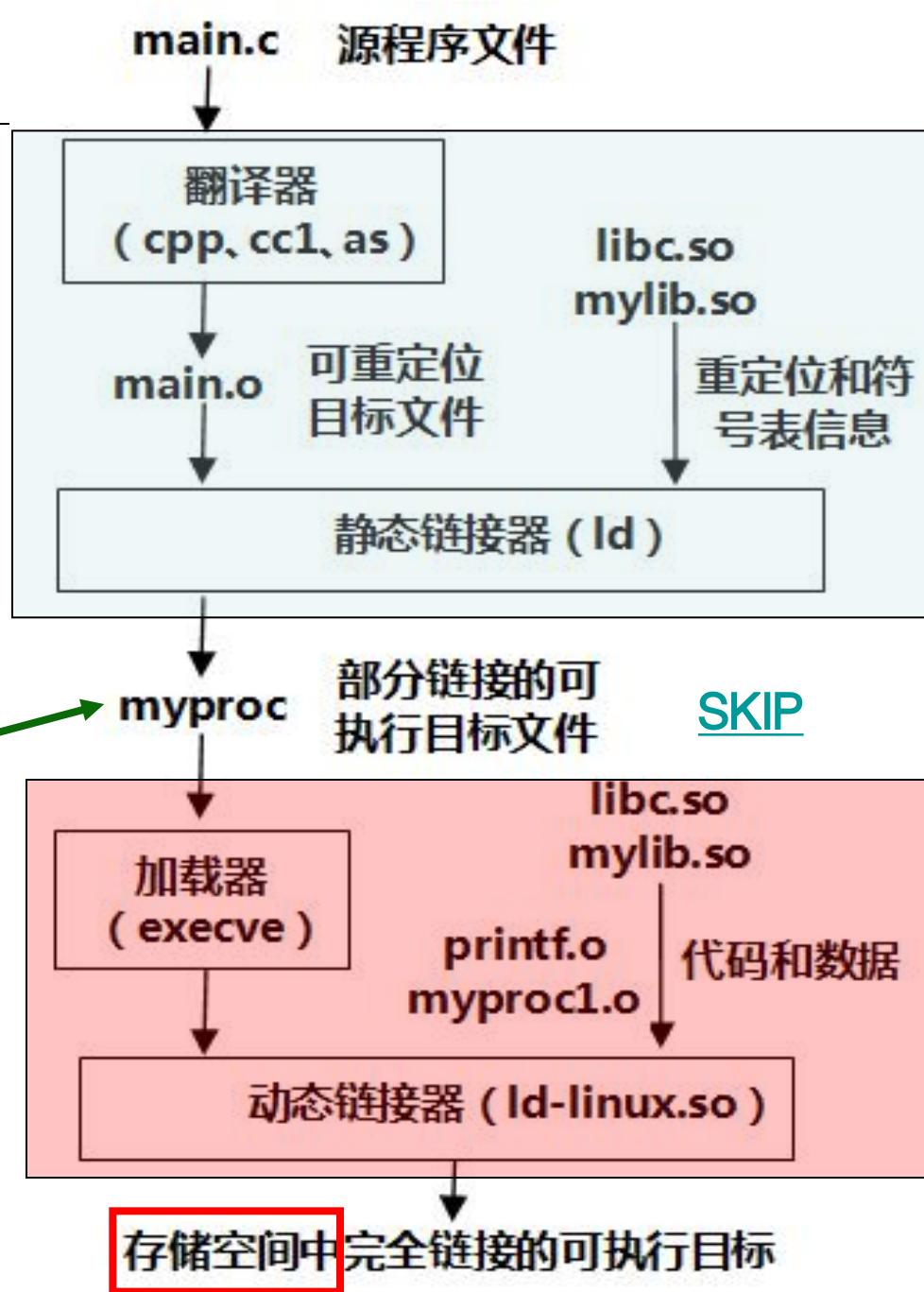
# 加载时动态链接

gcc -c main.c    **libc.so**无需明显指出  
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

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

通过-fPIC得到的PIC汇编代码及其跳转表如下

```
void switch_test(int x, int *ptr) {
```

```
    switch(x) {
```

```
        .....
```

```
    default:
```

```
        .....
```

```
    }
```

```
    *ptr += x;
```

```
}
```

```
    addi.w $t0,$a0,3(0x3)
```

```
    addi.w $t1,$zero,6(0x6)
```

```
    bltu    $t1,$t0,56(0x38) # .L3
```

```
    pcaddu12i $t1,0
```

```
    addi.d $t1,$t1,316(0x13c)
```

```
    als!l.d $t0,$t0,$t1,0x3
```

```
    ldptr.d $t0,$t0,0
```

```
    add.d $t1,$t1,$t0
```

```
    jirl    $zero,$t1,0
```

```
.section .rodata
.align 3
.align 2
.L7:
.dword .L2-.L7
.dword .L3-.L7
.dword .L4-.L7
.dword .L5-.L7
.dword .L3-.L7
.dword .L5-.L7
.dword .L6-.L7
```

第8行指令使得对跳转表的访问采用基于当前指令地址的PC相对寻址方式实现  
跳转表中是相对地址而非原来的绝对地址

同一模块内指代码节.text和跳转表所在.rodata节相对位置固定, 不管只读代码段映射到地址空间何处, 通过第8行指令都能访问到相应跳转表项。

# 回顾：LA64中switch-case语句举例

C语言函数switch\_test()的部分代码及其LA64部分汇编代码和跳转表如下

```
int sw_test(int x, int *ptr)
{
    switch(x) {
        .....
    default:
        .....
    }
    *ptr+=x;
}
```

addi.w	\$t0,\$a0,3(0x3)
addi.w	\$t1,\$zero,6(0x6)
bltu	\$t1,\$t0,52(0x34) # .L3
pcaddu12i	\$t1,87(0x57)
addi.d	\$t1,\$t1,-1116(0xba4)
alsl.d	\$t0,\$t0,\$t1,0x3
ldptr.d	\$t0,\$t0,0
jirl	\$zero,\$t0,0

.section	.rodata
.align 3	
.L7:	
.dword	.L2
.dword	.L3
.dword	.L4
.dword	.L5
.dword	.L3
.dword	.L5
.dword	.L6

switch\_test()函数的switch语句中共有几个case分支？case取值各是什么？各对应跳转表中哪个标号？

当 $x+3>6$ 时为default情况，对应标号.L3。bltu指令按无符号整数比较，故仅在 $0\leq x+3\leq 6$ （ $x$ 在-3~3之间）才不满足bltu条件，从而需执行jirl指令通过跳转表进行指令跳转。

alsl.d指令实现 $R[t0] \leftarrow R[t1] + (x+3)*8$ ，即t0中为表项地址

ldptr.d指令用于将表项内容送t0.

switch语句中共有6个分支，对应的x取值分别-3（.L2）、-1（.L4）、0（.L5）、2（.L5）、3（.L6）和default（.L3）



# 位置无关代码 (PIC)

- 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
  - 引入PIC的目的
    - 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

要实现动态链接, 必须生成PIC代码。以下通过该示例说明在上述4种情况下, 如何生成位置无关代码的共享库文件, 假设以下例子编译出的共享库文件为mylib.so。

# (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
- 不管mylib.so中的代码加载到哪里，bl指令中的偏移量不变

```
0000000000000067c <bar>:
```

```
.....
```

```
6bc: 4c000020 jirl $zero,$ra,0
```

```
000000000000006c0 <foo>:
```

```
6c0: 02ffc063 addi.d $sp,$sp,-16(0xff0)
```

```
.....
```

```
6d0: 57ffa000 bl -84(0xfffffac) # 67c <bar>
```

```
.....
```

根据引用符号bar的起始位置与bl指令之间的位移量算出偏移值，得到bl立即数字段offs26，这里偏移值为 $0x067c - 0x06d0 = 0xfffffac = -84$

```
static int a=0;
extern int b;
int c=0;
extern void ext();
static void bar() {
    a=1;
    b=c;
}

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

假设bar加载到0x1 2000 067c，则foo起始位置为0x1 2000 06c0，BI指令目标地址为：

$0x1\ 2000\ 06d0 +$   
 $0xf\ fff\ fac(-84) =$   
 $0x1\ 2000\ 067c$

## (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，其偏移量可在静态链接时的重定位阶段确定

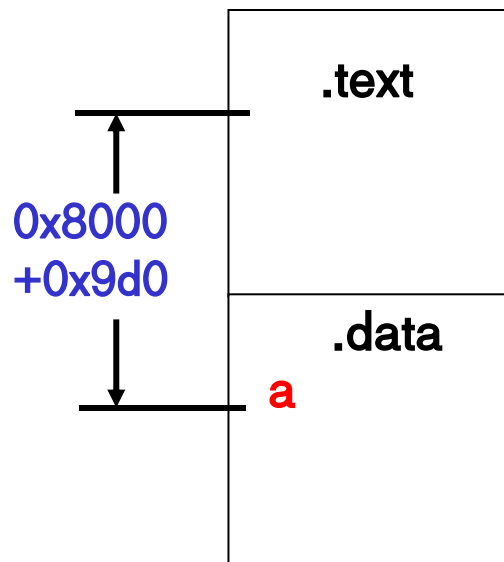
以下是赋值语句“a=1;”的编译结果，为生成PIC，编译器对语句“a=1;”生成了多条指令。

688:	1c00010c	pcaddu12i	\$t0,8(0x8)
68c:	02e7418c	addi.d	\$t0,\$t0,-1584(0x9d0)
690:	0280040d	addi.w	\$t1,\$zero,1(0x1)
694:	2500018d	stptr.w	\$t1,\$t0,0

开始两条指令可计算出变量a的地址，为  $0x0688 + 0x8000 + 0xf9d0 = 0x8058$ ，在指令stptr.w中通过对R[t0]所指存储单元的访问实现对a的引用  
这里编译器通过pcaddu12i指令将当前PC作为变量a的地址计算过程中的基准地址实现代码的浮动

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

多用了4条指令



共享库模块

### (3) 模块间数据的引用

- 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**），动态链接时填入地址，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

以下是赋值语句“b=c;”的编译结果，为生成PIC，编译器对语句“b=c;”生成了多条指令。

```
698: 1c00010c    pcaddu12i    $t0,8(0x8)
69c: 28e6618c    ld.d         $t0,$t0,-1640(0x998)
6a0: 2400018d    ldptr.w      $t1,$t0,0
6a4: 1c00010c    pcaddu12i    $t0,8(0x8)
6a8: 28e6118c    ld.d         $t0,$t0,-1660(0x984)
6ac: 2500018d    stptr.w      $t1,$t0,0
```

```
static int a=0;
extern int b;
int c=0;
extern void ext();
static void bar() {
    a=1;
    b=c;
}
....
```

0x8000  
+0xf984

.text

&b GOT  
&c .data  
c

共享库模块

.text

b .data

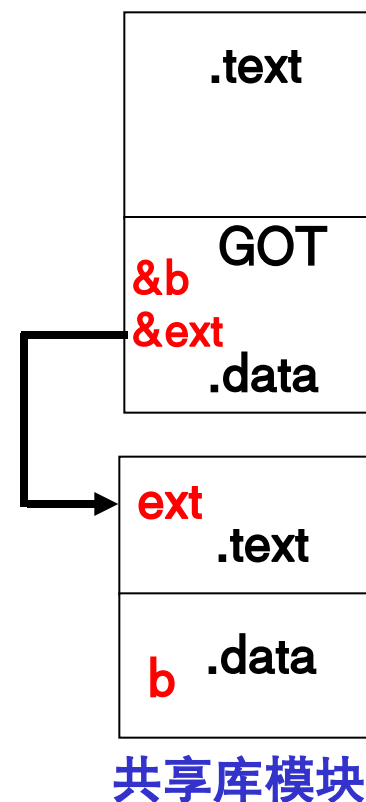
存放&b的表项地址为 $0x06a4 + 0x8000 + 0xf984 = 0x8028$ ，&c的表项地址为 $0x0698 + 0x8000 + 0xf998 = 0x8030$ ，stptr.w指令通过t0和t1分别引用变量b和c

若可执行文件和mylib.so都定义了全局变量c，ABI规定可执行文件中的定义优先级高，为此，需在GOT中加&c表项，动态链接时填入的可能是可执行文件中c符号定义值，而不是mylib.so中定义的c

## (4) 模块间过程调用和跳转

- **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），**程序加载时**，通过重定位填入目标函数的首地址
  - 这种方法存在有两方面问题：
    - 加载时，需对每个外部函数进行重定位并填GOT表，使得**程序加载过程变慢**，而且程序中有一些外部函数可能因为某些条件不满足而不会被执行，因而实际上无须对其进行重定位并填写GOT表
    - 有些系统中，每次调用所引用的外部函数时，都需多条指令实现外部函数的调用
- 可用“**延迟绑定（lazy binding）**”技术加快加载过程并减少指令执行条数：不在加载时重定位，而是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）

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



# (4) 模块间函数调用

## • 方法二：采用延时绑定技术

GOT[0]为动态链接器延迟绑定函数的入口地址。所有外部函数在GOT中都有对应表项，如GOT[2]就是ext()对应表项

PLT[0]项占32B，其余表项占16B，含4条指令。除PLT[0]外，其余表项各自对应一个共享库函数，如PLT[1]对应ext()函数

00000000000000530 <.plt>:

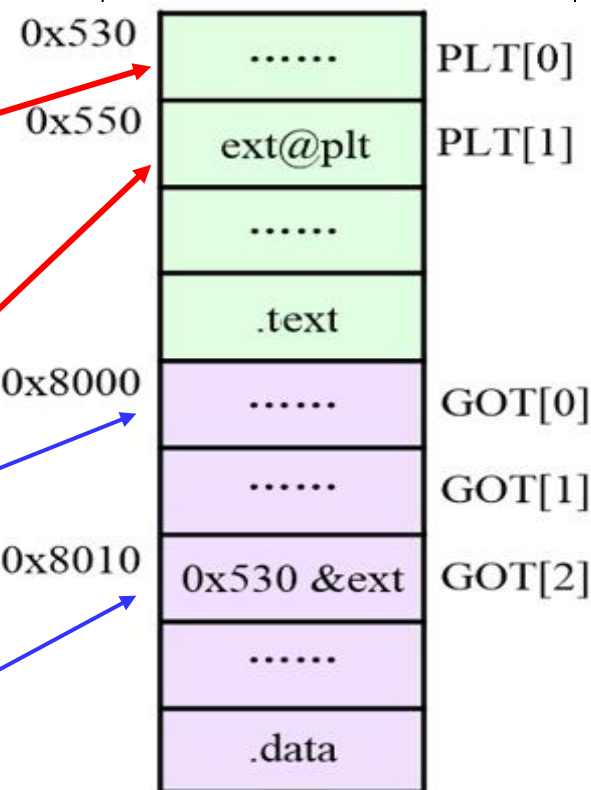
```
530: pcaddu12i $r14,8(0x8)
534: sub.d $r13,$r13,$r15
538: ld.d $r15,$r14,-1328(0xad0)
53c: addi.d $r13,$r13,-40(0xfd8)
540: addi.d $r12,$r14,-1328(0xad0)
544: srli.d $r13,$r13,0x1
548: ld.d $r12,$r12,8(0x8)
54c: jirl $r0,$r15,0
```

00000000000000550 <ext@plt>:

```
550: pcaddu12i $r15,8(0x8)
554: ld.d $r15,$r15,-1344(0xac0)
558: pcaddu12i $r13,0
55c: jirl $r0,$r15,0
```

第一次执行bl指令时，通过jirl指令跳转到GOT[2]所指向的0x530处执行PLT[0]的代码，再通过GOT[0]跳转到动态链接器延迟绑定函数执行，对ext重定位，解析其GOT偏移量，在GOT[2]中填入真正的ext首地址，并转ext执行。以后每次直接转GOT[2]所指代码执行

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



可执行文件中的 PLT 和 GOT

调用ext()的指令如下:

6d4: bl -388(0xffffe7c) # 550 <ext@plt>

# 位置无关可执行文件

---

- 为防范恶意程序对已知地址攻击，常采用**地址空间布局随机化**（Address Space Layout Randomization, **ASLR**），使每次加载的位置不同，即支持**位置无关可执行文件**（Position-Independent Executable, **PIE**）
- PIE和PIC类似，需保证**可执行文件代码与地址无关**，使得无论将其加载到何处都能正确执行
- 可通过**编译选项-fPIE**和**链接选项-pie**，指示编辑器和链接器生成PIE。其中，**-fPIE选项**指示编译器生成位置无关目标文件，**-pie选项**指示链接器把通过-fPIE选项编译出的位置无关目标文件链接成PIE
- 有些新版本GCC默认采用上述选项，能直接生成PIE



# 位置无关可执行文件

- 目前LA64架构下GCC默认是-no-pie选项，对于以下源程序，将其用编译选项-fPIE和链接选项-pie得到的PIE进行反汇编后的LA64代码如下

**/\* main.c \*/**

```
int add(int, int);
int main() {
    return add(20, 13);
}
```

**PIE代码:**

```
000000000000007f4 <add>:
      7f4: 00101484      add.w    $a0,$a0,$a1
      7f8: 4c000020      jirl     $r0,$ra,0
```

**非PIE代码:**

**/\* test.c \*/**

```
int add(int i, int j) {
    int x = i + j;
    return x;
}
```

```
00000000120000680 <add>:
      120000680: 00101484      add.w    $a0,$a0,$a1
      120000684: 4c000020      jirl     $r0,$ra,0
```

在add()函数中添加语句“printf(“&add=%p, &x=%p\n “, add, &x);”后重新编译为PIE，连续三次执行test的结果如下:

&add=0xaaaab58864, &x=0xffbdb0efc

&add=0xaaae214864, &x=0xffba3ff7c

&add=0xaaac8f4864, &x=0xffbd5b35c

可见: 每次运行test时, 操作系统都将其加载到不同位置, 可有效防止恶意程序的攻击



# 目标文件

在LA64中将以下程序编译生成可重定位文件test.o和可执行文件test

```
/* 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;
}
```

可重定位文件由单个模块生成，可执行文件由多个模块组合而成。对于前者，代码总是从0开始，对于后者，代码的地址是虚拟地址空间中的地址。

**objdump -d test.o**

0000000000000000 <add>:

0:	00101484	add.w	\$a0,\$a0,\$a1
4:	4c000020	jirl	\$r0,\$ra,0

**objdump -d test**

0000000120000680 <add>:

120000680:	00101484	add.w	\$a0,\$a0,\$a1
120000684:	4c000020	jirl	\$r0,\$ra,0

通过objdump命令输出的结果包括指令的地址、指令机器代码和反汇编得到的汇编指令代码。

可重定位文件test.o中add模块代码起始地址为0；在可执行文件test中add的起始地址为0x1 2000 0680

# 库打桩机制

---

- 库打桩（library interpositioning）：截获对共享库函数的调用，转而替代调用程序员自己编写的函数
- 被截获的共享库函数称为目标函数（target function），程序员自己编写的替代函数称为封装函数（wrapper function）
- 封装函数和目标函数的原型应完全相同
- 利用库打桩可以进行如下操作：
  - 跟踪共享库函数的调用次数和每次的入口参数及返回值
  - 将目标函数替换成与其完全不同的功能实现
  - 将包含恶意代码的封装函数预先生成动态链接库并借助运行时打桩机制设置软件后门
- 有三种库打桩方式
  - 编译时打桩、链接时打桩、运行时打桩

# 编译时打桩

- ① 在当前目录生成一个头文件，其中使用#define将目标函数的调用替换为对封装函数的调用，并给出封装函数的原型声明
- ② 编写封装函数源程序，用#include将头文件嵌入源程序中
- ③ 在GCC命令行中使用-l.参数，以设定预处理程序最先查找并使用当前目录中的头文件，从而在程序编译过程中实现函数的替换调用

**myabs.h:**

```
#define abs(x) myabs(x)
int myabs(int x);
```

**abs.c:**

```
#include <stdio.h>
#include <stdlib.h>
#include <myabs.h>
int main() {
    int y=abs(-10);
    return 0;
}
```

**myabs.c:**

```
#ifdef COMPILE_INTERPOSITION
#include <stdio.h>
#include <stdlib.h>
#include <myabs.h>
/* abs wrapper function */
int myabs(int x) {
    int y=abs(x);
    printf("abs(%d)=%d\n",x,y);
    return 0;
}
#endif
```

执行以下命令后，生成可执行文件myabs

```
gcc -DCOMPILE_INTERPOSITION -c myabs.c
```

```
gcc -l. -o myabs abs.c myabs.o
```

myabs的执行结果如下：

```
abs(-10)=10
```

# 链接时打桩

- GCC链接器可使用`-Wl,--wrap,func` 或 `-Wl,--wrap=func`参数进行打桩
- 该参数指示链接器按如下方式对符号`func`进行符号解析：若当前模块未定义符号`func`，就将`func`的引用解析成符号`__wrap_func`，若当前模块未定义符号`__real_func`，就将`__real_func`的引用解析成符号`func`。

**main.c:**

```
#include <stdio.h>
void __wrap_test(){
    printf("File: %s, Function: %s\n",__FILE__,__FUNCTION__);
}
void foo1(){
    test();
}
int main(){
    test();
    foo1();
    foo2();
    return 0;
}
```

**GCC命令如下:**

```
gcc -Wl,--wrap=test -o test test.c main.c
```

**test.c:**

```
#include <stdio.h>
void __real_test();
/* test wrapper function */
void test(){
    printf("File: %s, Function: %s\n",__FILE__,__FUNCTION__);
}
void foo2(){
    __real_test();
}
```

**执行可执行文件test的结果如下:**

```
File: main.c, Function: __wrap_test
File: main.c, Function: __wrap_test
File: test.c, Function: test
```

# 运行时打桩

- 通过LD\_PRELOAD环境变量设置共享目标库路径名的一个列表
- 动态链接器符号解析时，将优先搜索在LD\_PRELOAD中设置的共享目标库
- 将封装函数定义在使用LD\_PRELOAD环境变量设置的共享库中
- 动态链接器对未定义引用进行符号解析时，先解析成封装函数中定义的符号，而不会解析成C标准函数库中目标函数定义的符号

**mygets.c:**

```
#define _GNU_SOURCE
#include <stdio.h>
#include <dlfcn.h>
/* gets wrapper function */
char *gets(char *str) {
    char>(*getsp)(char*);
    char *error;
    printf("wrapper func gets str: %s\n",str);
    getsp=dlsym(RTLD_NEXT,"gets");
    if ((error = dlerror()) != NULL) {
        fprintf(stderr, "%s\n", error);
        exit(1);
    }
    getsp(str); //调用目标gets
    return ptr
}
```

**main.c:**

```
#include <stdio.h>
int main(){
    char str[10]="\0";
    printf("Input:\n",);
    gets(str);
    return 0;
}
```

**执行结果（假定输入01234）：**

**Input:**

**wrapper func gets str:**

**（在键盘上输入）01234**

**GCC命令和bash shell中设置环境变量并执行：**

**gcc -shared -fPIC -ldl -o mygets.so mygets.c**

**gcc -o test main.c**

**LD\_PRELOAD="./mygets.so" ./test**

# 程序的链接与加载执行

---

- 分以下四个部分介绍
  - 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
  - 第三讲：动态链接和库打桩机制
    - 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、位置无关代码的生成、库打桩机制
  - 第四讲：可执行文件的加载和执行
    - 可执行文件的加载
    - 程序和指令的执行过程

# 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。

程序被启动

如 `$ ./P`

调用fork()

以构造的argv和envp为参数调用execve()

execve()调用加载器进行可执行文件加载，并最终转去执行main



# 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

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

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



# 程序及指令的执行过程

---

- 程序和指令的关系
  - 程序由一条一条指令组成，指令按顺序存放在连续存储单元
- 程序的执行：周而复始地执行一条一条指令
  - 正常情况下，指令按其存放顺序执行
  - 遇到需改变程序执行流程时，用相应的转移指令（包括无条件转移指令、条件转移指令、调用指令和返回指令等）来改变程序执行流程
- 程序的执行流的控制
  - 将要执行的指令所在存储单元的地址由程序计数器PC给出，通过改变PC的值来控制执行顺序
- 指令周期：CPU取出并执行一条指令的时间

# 程序及指令的执行过程

在内存存放的指令实际上是机器代码（0/1序列）

08048394 <add>:

```
8048394: 55          push  %ebp
1 8048395: 89 e5      mov  %esp, %ebp
2 8048397: 8b 45 0c   mov  0xc(%ebp),
%3ax
4 804839a: 03 45 08   add  0x8(%ebp),
%5ax
6 804839d: 5d         pop  %ebp
804839e: c3        ret
```

° 对于add函数

程序执行需要解决的问题:

如何判定每条指令有多长?  
如何判定操作类型、寄存器  
编号、立即数等? 如何区分  
第2行和第3行mov指令的不  
同? 如何确定操作数是在寄  
存器中还是在存储器中? 一  
条指令执行结束后如何正确  
读取到下一条指令?

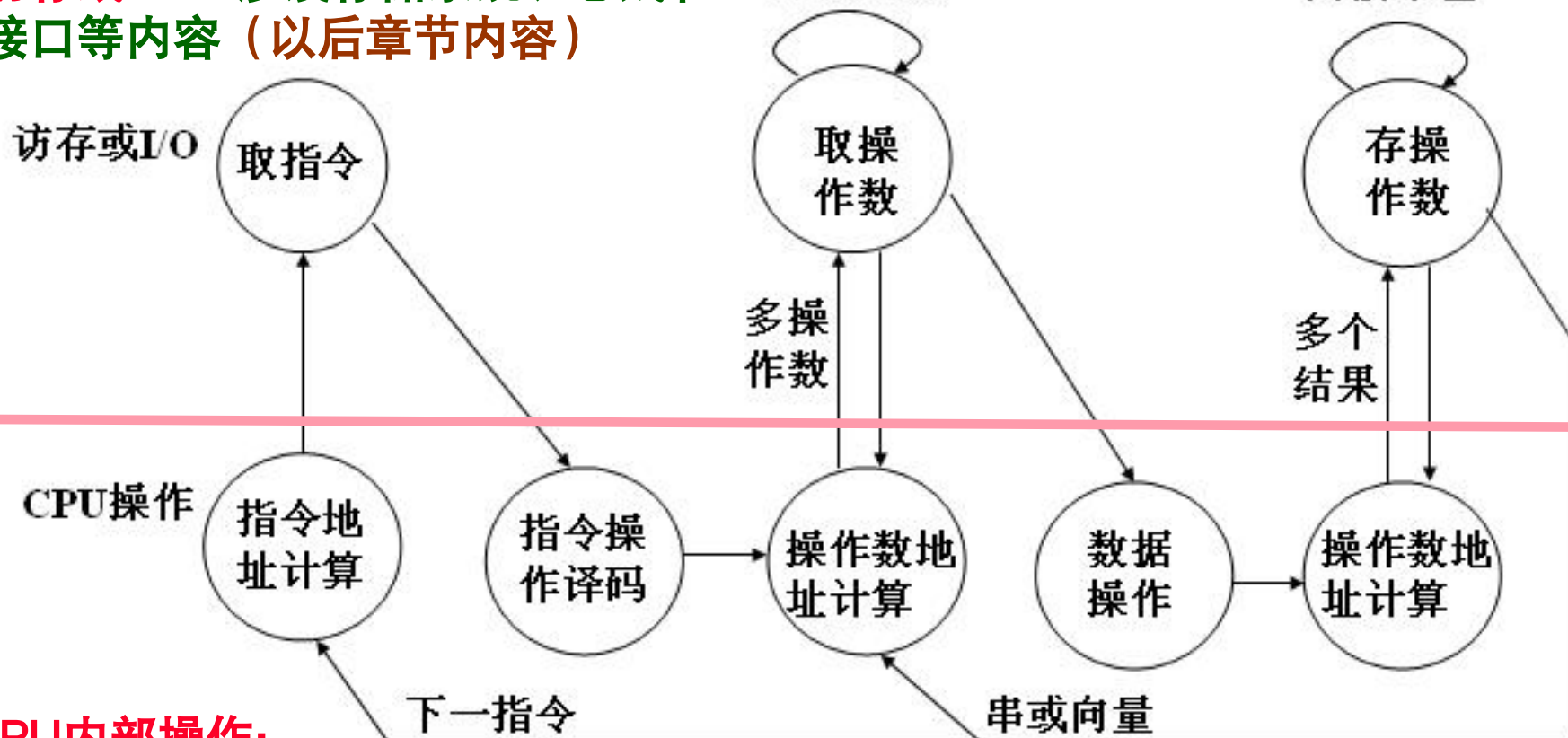
ü 指令按顺序存放在0x08048394开始的存储空间。

ü 各指令长度可能不同，如push、pop和ret指令各占一个字节，第2行mov指令占两个字节，第3行mov指令和第4行add指令各占3字节。

ü 各指令对应的0/1序列含义有不同的规定，如“push %ebp”指令为01010101B，其中01010为push指令操作码，101为%ebp的编号，“pop %ebp”为01011101B，其中01011为pop指令的操作码。

# 程序及指令的执行过程

**访存或I/O:** 涉及存储系统、总线和I/O间接寻址接口等内容（以后章节内容）



**CPU内部操作:**  
涉及CPU内部数据通路（本章节内容）

CPU运行程序的过程就是执行一条一条指令的过程

CPU执行指令的过程中，包含CPU操作、访问内存或I/O端口的操作两类

# 机器指令的执行过程

## ° CPU执行指令的过程

- 取指令
- PC+ “1”
- 指令译码
- 进行主存地址运算
- 取操作数
- 进行算术 / 逻辑运算
- 存结果
- 以上每步都需检测 “异常”
- 若有异常，则自动切换到异常处理程序
- 检测是否有 “中断” 请求，有则转中断处理

取指阶段

“1”：指一条指令的长度，定长指令字每次都一样；变长指令字每次可能不同

执行阶段

指令执行过程

定长指令字通常在译码前做，变长指令字在译码后做！

## 问题：

“取指令”一定在最开始做吗？PC+ “1”一定在译码之前做吗？

“译码”须在指令执行前做吗？

你能说出几种 “异常” 事件？“异常” 和 “中断” 的差别是什么？

异常是在CPU内部发生的，中断是由外部事件引起的

# 机器指令的执行过程

---

- 每条指令的功能总是由以下四种基本操作来实现:

读取某一主存单元的内容, 并将其装入某个寄存器 (取指, 取数)

把一个数据从某个寄存器存入给定的主存单元中 (存结果)

把一个数据从某寄存器送到另一寄存器或者ALU (取数, 存结果)

进行算术或逻辑运算 ( $PC+1$ , 计算地址, 运算)

指令执行过程中查询各种异常情况, 并在发现异常时转异常处理

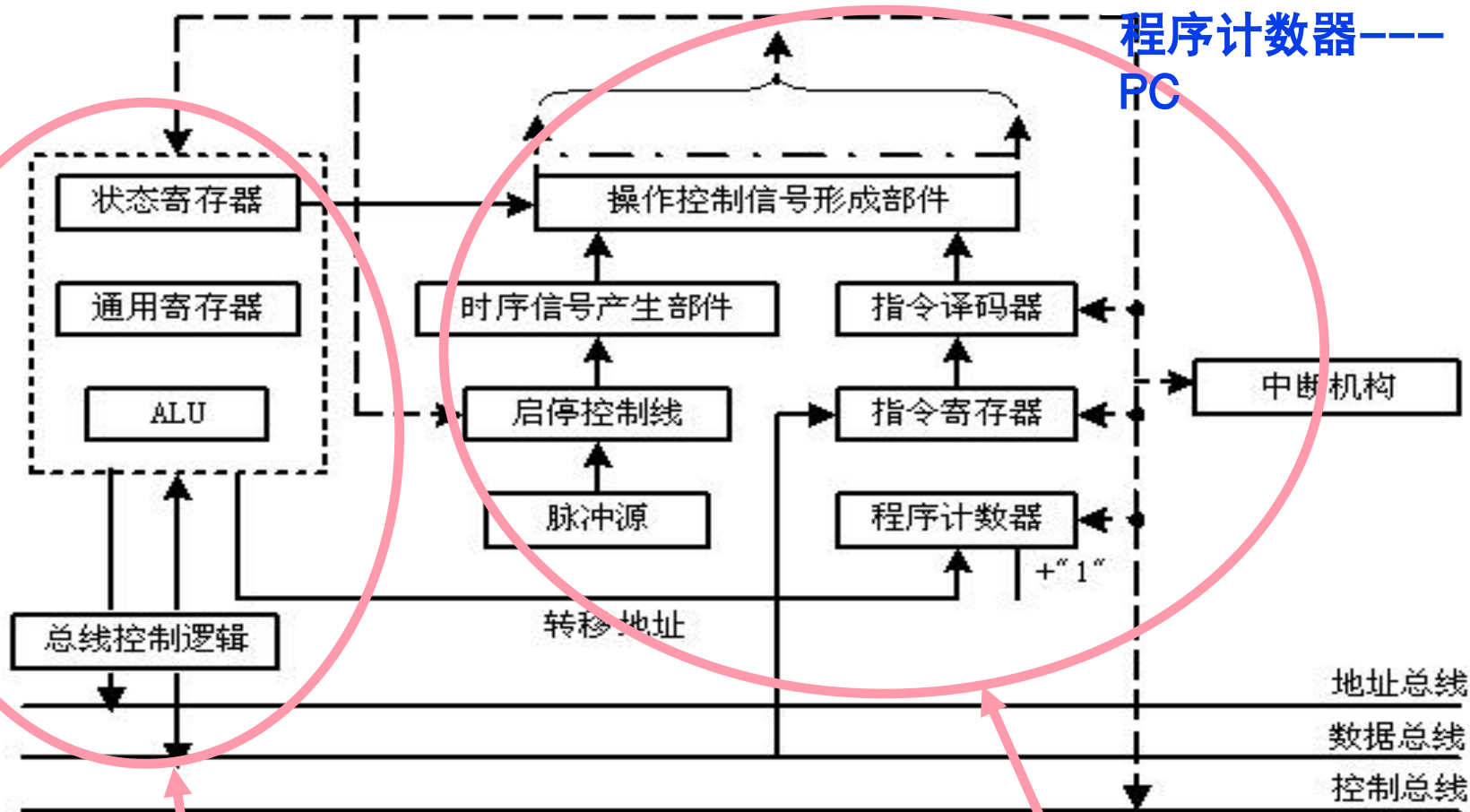
指令执行结束时查询中断请求, 并在发现中断请求时响应中断

- 操作功能可形式化描述

描述语言称为寄存器传送语言RTL (Register Transfer Language)

# CPU基本组成原理图

指令寄存器----  
IR  
程序计数器---  
PC



执行部件

控制部件

控制器主要由 指令译码器 和 控制信号形成部件 组成

CPU 由 执行部件 和 控制部件 组成

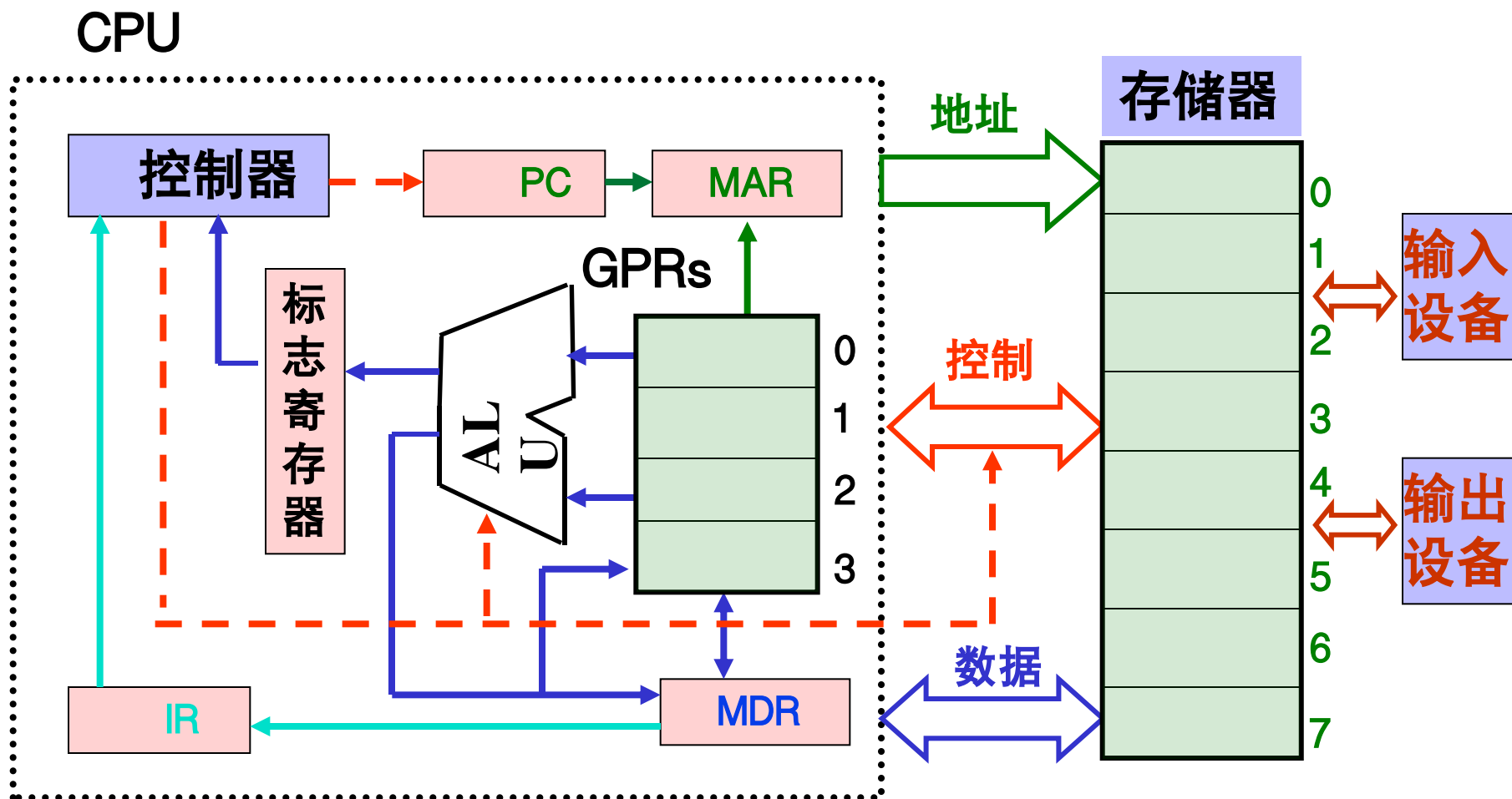
CPU 包含 数据通路(执行部件) 和 控制器 (控制部件) 两种基本部件

# 回顾：冯.诺依曼结构模型机

妈妈会做的菜和厨师会做的菜不一样，同一个菜谱的做法也可能不同

如同

不同架构支持的指令集不同，同一种指令的实现方式和功能也可能不同



# 回顾：IA-32的体系结构是怎样的呢？

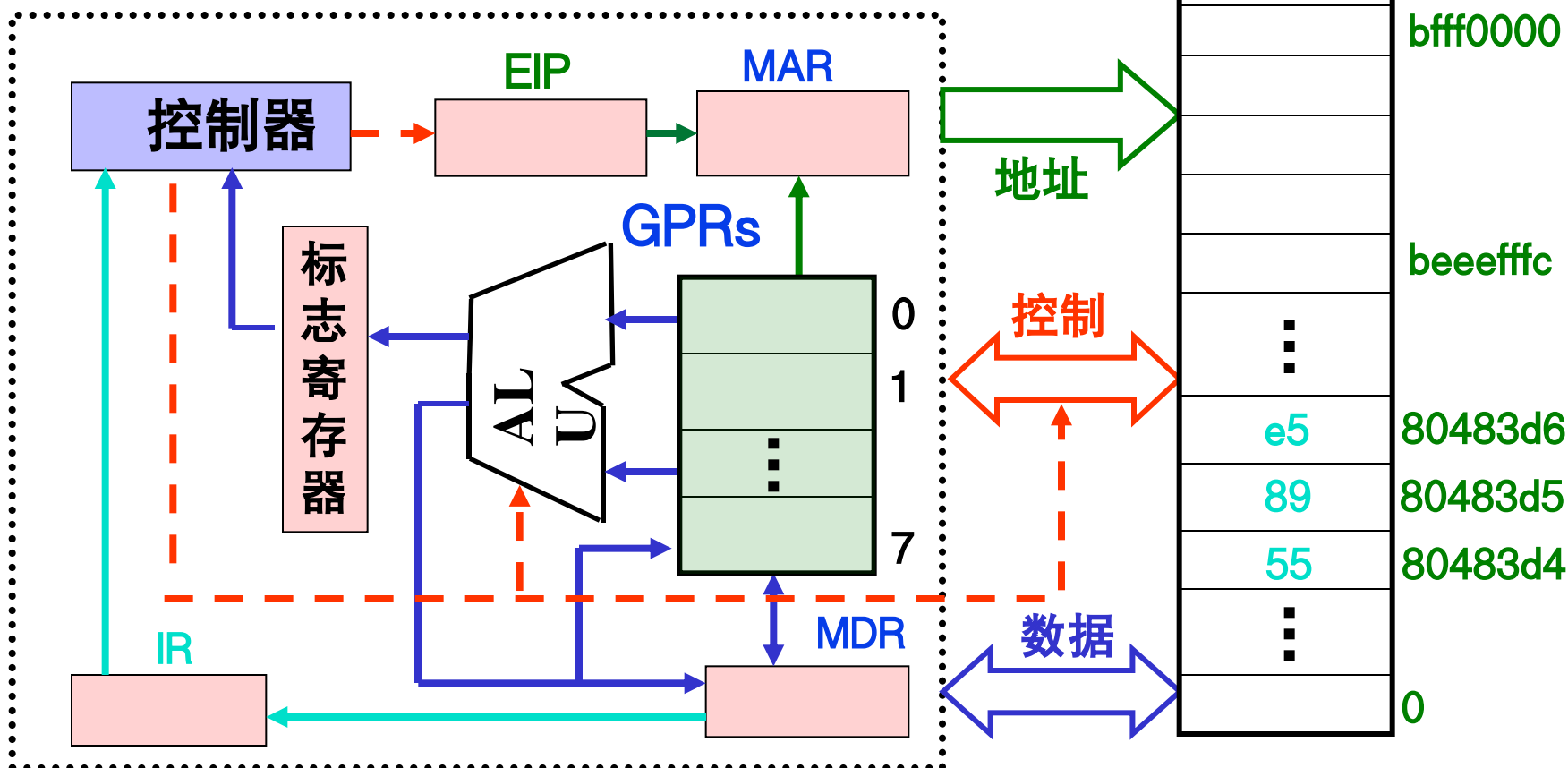
8个GPR（0~7），一个EFLAGS，PC为EIP

可寻址空间4GB（编号为0~0xFFFFFFFF）

指令格式变长，操作码变长

由若干字段（OP、Mod、SIB等）组成

存储器（假定主存  
地址即虚存地址）





# 程序由指令序列组成

```
1 // test.c
2 #include <stdio.h>
3 int add(int i, int j )
4 {
5     int x = i +j;
6     return x;
7 }
```

若  $i = 2147483647$ ,  $j = 2$ ,  
则程序执行结果是什么?  
每一步如何执行?

想想妈妈怎么做菜的?

“objdump -d test” 显示的add函数结果

080483d4 <add>:       $PC \leftarrow 0x80483d4$

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	

取并  
执行  
指令

根据PC取指令  
指令译码  
计算下指地址  
计算操作数地址  
取操作数  
执行计算  
指令回写结果  
修改PC的值

假定0x80484d4等是内存  
地址，实际上它们是虚存  
地址，需要虚-实地址转换

代码执行从80483d4开始!

OP

起始PC=?

功能:  $R[esp] \leftarrow R[esp] - 4$ ,  $M[R[esp]] \leftarrow R[ebp]$

080483d4 <add>:

→ 80483d4: 55 push %ebp  
80483d5: 89 e5 mov %esp,  
%ebp

S1:取指令 S2:指令译码

S3:指令执行

EBP

bfff0020 5

ESP

bfff0000 4

EIP

80483d4

80483d4

80483d4

地址

控制器

55

标志寄存器

GPRs

MAR

0

1

7

Rd

IR

MDR

5589e583

5589e583

5589e583

控制

Rd

数据

5589e583

bfff0020

bfff0000

bffeffc

e5 80483d6

89 80483d5

55 80483d4

0

功能:  $R[esp] \leftarrow R[esp] - 4$ ,  $M[R[esp]] \leftarrow R[ebp]$

080483d4 <add>:

80483d4: 55 push %ebp  
80483d5: 89 e5 mov %esp,

%ebp

S1:取指令 S2:指令译码

S3:指令执行

EBP

bfff0020

5

ESP

bffeffc

4

EIP

80483d4

80483d4

控制器

地址

55

标志寄存器

GPRs

MAR

ALU

0

1

7

控制

IR

MDR

数据

bfff0020

bfff0000

bffeffc

e5 80483d6

89 80483d5

55 80483d4

0

功能:  $R[esp] \leftarrow R[esp] - 4$ ,  $M[R[esp]] \leftarrow R[ebp]$

080483d4 <add>:

80483d4: 55 push %ebp  
80483d5: 89 e5 mov %esp,

%ebp

S1:取指令 S2:指令译码

S3:指令执行

EBP

bfff0020

5

ESP

bffeffc

4

EIP

80483d4

bffeffc

控制器

地址

55

标志寄存器

GPRs

MAR

ALU

0

1

7

控制

IR

MDR

数据

bfff0020

bfff0000

bffeffc

e5 80483d6

89 80483d5

55 80483d4

0

功能:  $R[esp] \leftarrow R[esp] - 4$ ,  $M[R[esp]] \leftarrow R[ebp]$

080483d4 <add>:

80483d4: 55 push %ebp  
80483d5: 89 e5 mov %esp,

%ebp

S1:取指令 S2:指令译码

S3:指令执行

EBP

bfff0020

5

ESP

bffefffc

4

EIP

80483d4

bffefffc

控制器

55

标志寄存器

GPRs

MAR

ALU

0

1

...

7

Wr

IR

MDR

bfff0020

bffefffc

地址

控制

Wr

数据

bfff0020

bfff0020

bfff0000

bffefffc

80483d6

80483d5

80483d4

0

功能:  $R[esp] \leftarrow R[esp] - 4$ ,  $M[R[esp]] \leftarrow R[ebp]$

080483d4 <add>:

80483d4: 55 push %ebp  
80483d5: 89 e5 mov %esp,

%ebp

S1:取指令 S2:指令译码

S3:指令执行

EBP

bfff0020 5

ESP

bffefffc 4

EIP

80483d4 bffefffc

控制器

55

标志寄存器

GPRs

MAR

ALU

0  
1  
:  
7

Wr

IR

MDR

bfff0020

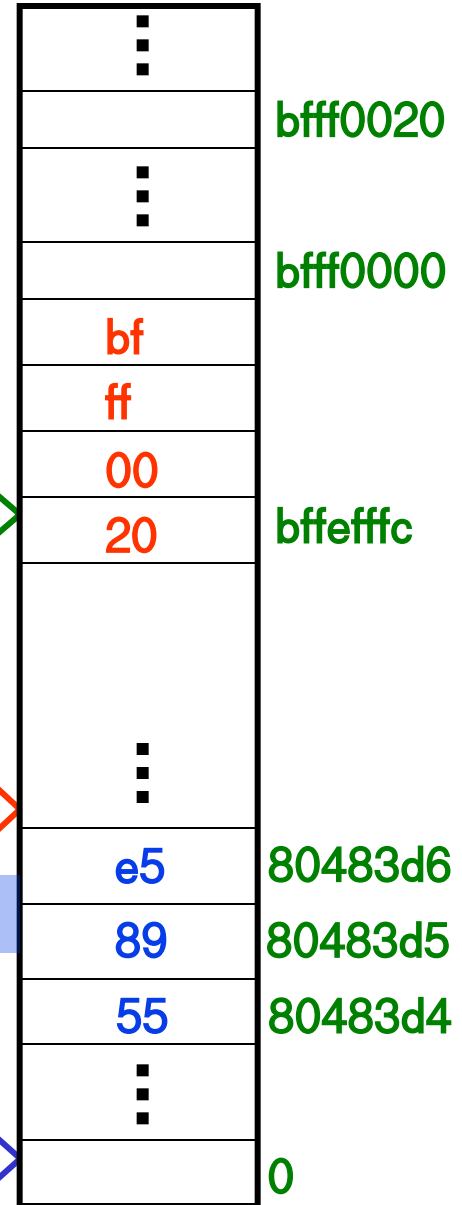
地址

控制

Wr

数据

bfff0020



# 开始执行下一条指令

080483d4 <add>:

80483d4: 55 push %ebp  
80483d5: 89 e5 mov %esp,

→  
%ebp

S1:取指令 S2:指令译码

S3:指令执行、EIP增量

EBP

bfff0020

5

ESP

bffefffc

4

EIP

80483d5

bffefffc

控制器

55

标志寄存器

GPRs

MAR

ALU

0

1

...

7

Wr

IR

MDR

bfff0020

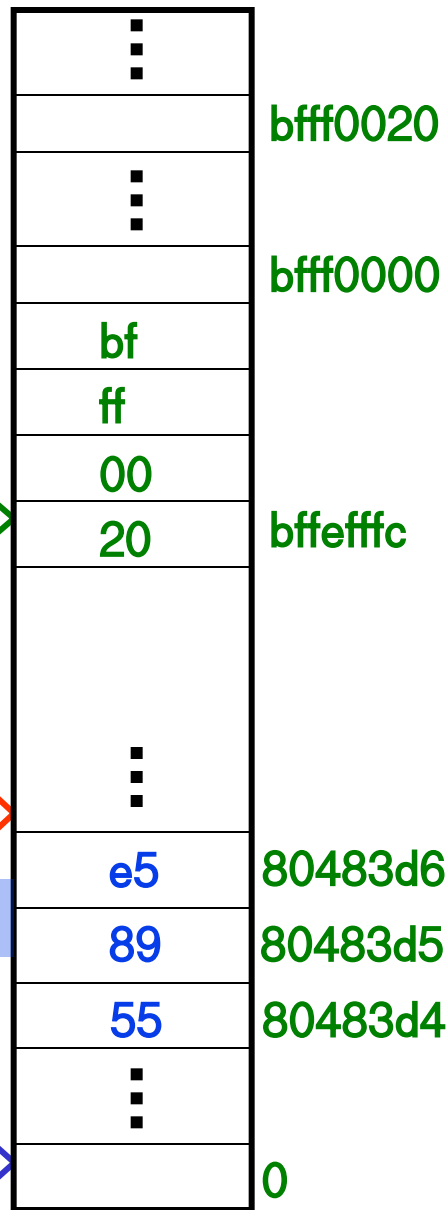
地址

控制

Wr

数据

bfff0020



# 打断程序正常执行的事件

---

- 程序执行过程中CPU会遇到一些特殊情况，使正在执行的程序被“中断”
  - CPU中止原来正在执行的程序，转到处理异常情况或特殊事件的程序去执行，结束后再返回到原被中止的程序处（断点）继续执行。
- 程序执行被“中断”的事件（在硬件层面）有两类
  - 内部“异常”：在CPU内部发生的意外事件或特殊事件  
按发生原因分为硬故障中断和程序性中断两类  
硬故障中断：硬件线路故障等  
程序性中断：执行某条指令时发生的“例外(Exception)”事件，如溢出、缺页、越界、越权、越级、非法指令、除数为0、堆/栈溢出、访问超时、断点设置、单步、系统调用等
  - 外部“中断”：在CPU外部发生的特殊事件，通过“中断请求”信号向CPU请求处理。如实时钟、控制台、打印机缺纸、外设准备好、采样计时到、DMA传输结束等。