

# 程序的链接

# 为什么学习 编译、链接？

---

- **帮助构造大型程序**

未定义的符号、缺少模块、缺少库 .....

- **避免危险的编程错误**

定义多个同名的全局变量 .....

- **理解作用域规则是如何实现的**

语句级、函数级、类级、模块级、全局、  
静态（static）变量或函数

- **其他重要的系统概念**

加载和运行程序、虚拟内存、分页、内存映射

- **利用共享库**

动态链接库

# 学习内容

---

- 编译、汇编
- 目标文件格式
- 符号表和符号解析
- 重定位
- 静态链接
- 动态链接

# 程序编译、汇编

hello.c

```
#include <stdio.h>

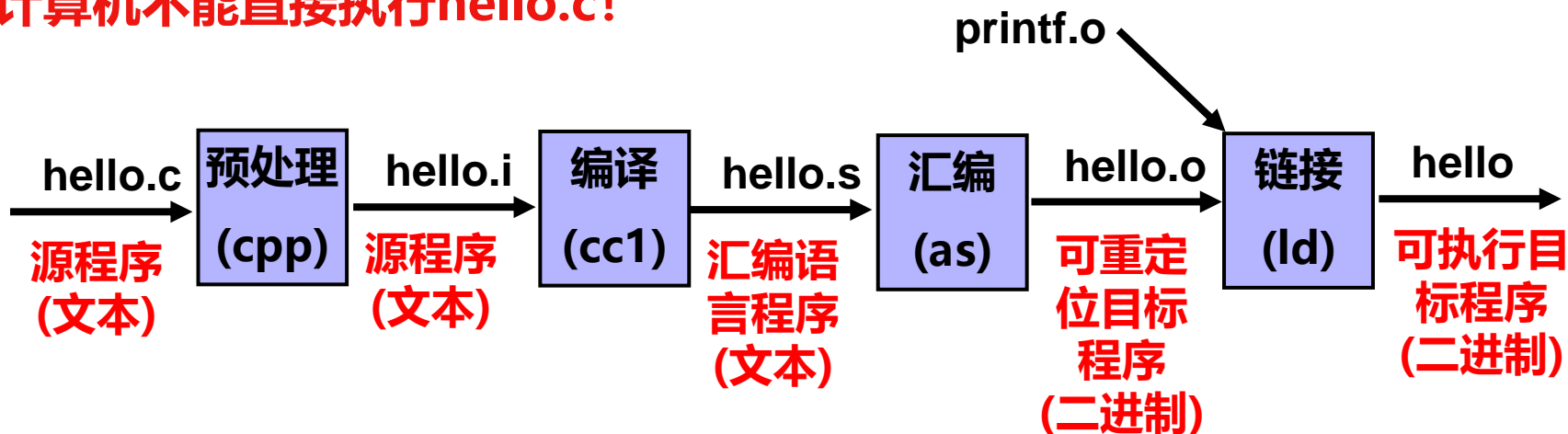
int main()
{
    printf("hello, world\n");
}
```

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!

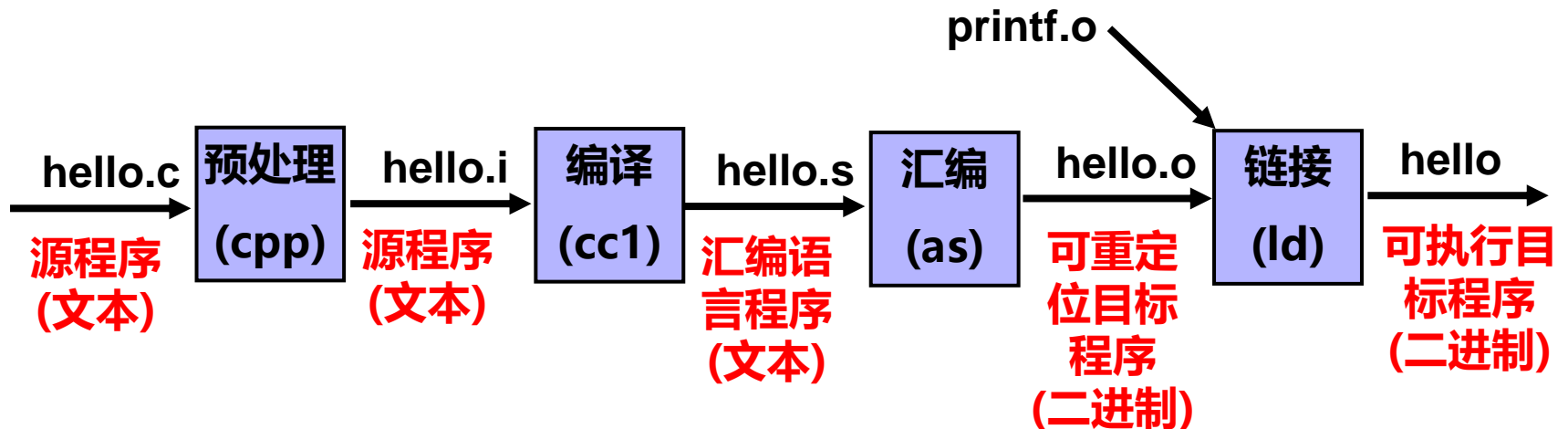


# 程序编译、汇编

预处理:     gcc -E hello.c -o hello.i  
或            cpp hello.c -o hello.i

编译:        gcc -S hello.c -o hello.s  
或            cc -S hello.i -o hello.s

汇编:        gcc -c hello.c -o hello.o  
或            as hello.s -o hello.o



# 程序编译、汇编

Q: 对一个文件汇编生成 .o 文件, 哪些信息可以确定, 哪些又不能确定?

## 内存映像

地址c1	代码段
	.....
地址cn	机器指令序列
	.....
地址d1	数据段
	.....
地址dm	存放 全局变量、非静态局部变量的 值
	.....
地址s1	堆栈段
	.....
地址su	?? 可变的内容
	.....

## 执行文件

代码段	.....
	机器指令序列
数据段	.....
	存放 全局变量、非静态局部变量的 值
	.....

# 程序编译、汇编

---

Q: 对一个文件汇编生成 .o 文件, 哪些信息可以确定, 哪些又不能确定?

```
mov %eax, %ebx  
sub $0x10, %esp
```

Q: 有变量名、参数名、函数名、标号名的语句如何处理?

**参数名 → 地址 n (%rbp)、n(%rsp)、Register**

**局部变量名 (非静态的) → 地址 n (%rbp)、n(%rsp)、R**

**标号 → 位移量, 与当前指令之间的字节距离**

不论函数放在执行文件 (或者内存) 何处,  
参数、非静态的局部变量、标号都是可以确定表达方式的!

# 程序编译、汇编

---

Q: 全局变量名、静态的局部变量名、函数名  
对应的地址能确定吗？

**需要重定位！**

**在可重定位的目标文件中，对它们要放什么一些什么信息？**

**指令代码中：**

**它们占用的地址空间要保留，可预先填成 00 ...00**

**要记录哪些位置的值要重写，并且要用什么符号对应的值来写**

**符号构成表、然后用符号的索引**



# 程序编译、汇编

---

Q: 全局变量名、静态的局部变量名、函数名  
对应的地址能确定吗? → 对应单元的内容能确定吗?

```
int g1=30;  
int *p = &g1;  
int (*q)(int ,int)=add;  
int add(int i,int j) { ... }
```

**在数据节中：要初始化相应单元的值，如何初始化？**

**它们占用的地址空间要保留，可预先填成 00 ...00**

**要记录哪些位置的值要重写，并且要用什么符号对应的值来写**

**符号构成表、然后用符号的索引**

# 链接

- 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 test.c
```

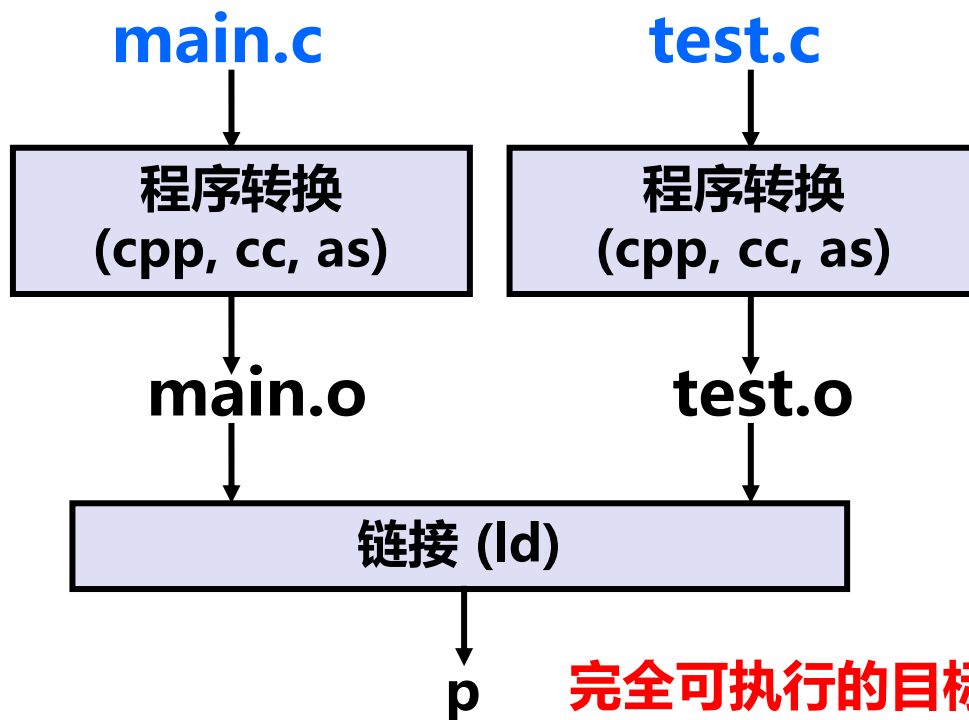
```
# ./p
```

-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

---

**链接 是 模块化程序设计的必然要求**

- 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**
- 3) 时间上， 各个源程序可分开编译**

**只需重新编译被修改的源程序文件，然后重新链接**

- 4) 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、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**

# 链接

```
#gcc main_link.c -o main_d
```

```
#gcc main_link.c /usr/lib/x86_64-linux-gnu/libc.so  
-o main_dll
```

```
#gcc -static main_link.c -o main_s
```

**C标准库的大部分函数在 libc.so 中，自动链接；对于其他的库（非标准库、第三方库等），需要手动添加。**

**生成的文件大小差别很大**

**动态链接库以.so作为后缀**

**静态链接库通常以.a作为后缀**

```
-rwxr-xr-x 1 root root 8408 May 7 18:36 main_d  
-rwxr-xr-x 1 root root 8408 May 7 23:31 main_dll  
-rw-r--r-- 1 root root 147 May 7 18:27 main_link.c  
-rwxr-xr-x 1 root root 958808 May 7 18:36 main_s
```

# 链接

```
// math_test.c
#include <stdio.h>
#include <math.h>
#define PI 3.14159265
int main ()
{ double param, result;
  param = 60.0;
  result = cos(param * PI / 180.0 );
  printf ("The cosine of %f  is %f.\n", param, result );
  return 0;
}
#gcc math_test.c -o math_test
```

```
math_test.c:(.text+0x33): undefined reference to `cos'
collect2: error: ld returned 1 exit status
```

```
#gcc math_test.c -o math_test -lm
```

数学库: /usr/lib/x86\_64-linux-gnu/libm.so / libm.a

# 链接

---

**链接：将多个可重定位的目标文件合成一个可执行文件。**

**Q：可重定位的目标文件中包含什么信息？如何合并？**

## 1、符号解析

**将符号的引用与一个确定的符号定义建立关联。**

**符号：全局变量名、函数名、静态的局部变量名**

**非静态的局部变量名不是符号、参数名不是符号。**

## 2、重定位

**在合并生成执行文件时，重新确定每条指令的地址、  
每个数据的地址、在指令中 确定所引用符号对应的地址。**

# 目标文件格式

---

# 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
- 几种标准的目标文件格式
  - **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 (Common Object File Format)
  - **Windows: PE格式 (COFF的变种)**，可移植可执行  
**Portable Executable**
  - **Linux等类UNIX: ELF格式 (COFF的变种)**，可执行可链接  
**Executable and Linkable Format**

# 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

ELF 头
程序头表 (可选)
节 1
...
节 n
...
...
节头表

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

ELF 头
程序头表
段 1
段 2
...
节头表 (可选)

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

# 目标文件格式

Q: 目标文件中会存储什么信息?

```
test.c                                     #gcc -c -g test.c -o test.o
int add(int i,int j) {
    int x=i+j;
    return x;
}                                           #od -Ax -tx1 test.o
```

```
root@LAPTOP-CJLSTBTI:/home/chapter4# od -Ax -tx1 test.o
000000 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00 00
000010 01 00 3e 00 01 00 00 00 00 00 00 00 00 00 00 00
000020 00 00 00 00 00 00 00 00 00 68 05 00 00 00 00 00 00
000030 00 00 00 00 40 00 00 00 00 00 00 40 00 13 00 12 00
000040 55 48 89 e5 89 7d ec 89 75 e8 8b 55 ec 8b 45 e8
000050 01 d0 89 45 fc 8b 45 fc 5d c3 77 00 00 00 04 00
000060 00 00 00 00 08 01 00 00 00 00 00 0c 00 00 00 00 00
```

od : octal dump

od [-abcdfhilovx] [-A<字码基数>] [-j<字符数目>]  
[-N<字符数目>] [-s<字符串字符数>] [-t<输出格式>]  
[-w<每列字符数>] [--help] [--version] [文件名]

# 目标文件格式

## 以二进制形式显示文件的另一种方式

```
hexdump [-bcCdovx] [-e fmt] [-f fmt_file]
        [-n length] [-s skip] [file ...]
```

```
#hexdump -C -n 512 test.o
```

-C 输出规范的十六进制和ASCII码

```
root@LAPTOP-CJLSTBT1:/home/chapter4# hexdump -C -n 512 test.o
00000000  7f 45 4c 46 02 01 01 00  00 00 00 00 00 00 00 00  |.ELF.....|
00000010  01 00 3e 00 01 00 00 00  00 00 00 00 00 00 00 00  |..>.....|
00000020  00 00 00 00 00 00 00 00  68 05 00 00 00 00 00 00  |.....h....|
00000030  00 00 00 00 40 00 00 00  00 00 40 00 13 00 12 00  |....@.....@....|
00000040  55 48 89 e5 89 7d ec 89  75 e8 8b 55 ec 8b 45 e8  |UH...}..u..U..E.|
00000050  01 d0 89 45 fc 8b 45 fc  5d c3 77 00 00 00 04 00  |...E..E.].w....|
00000060  00 00 00 00 08 01 00 00  00 00 0c 00 00 00 00 00  |.....|
```

# 目标文件格式

Q: 目标文件中会存储什么信息?

test.c	000000000000000000 <add>:
	int add(int i,int j){
	0: 55 push %rbp
int add(int i,int j) {	1: 48 89 e5 mov %rsp,%rbp
int x=i+j;	4: 89 7d ec mov %edi,-0x14(%rbp)
return x;	7: 89 75 e8 mov %esi,-0x18(%rbp)
}	int x=i+j;
	a: 8b 55 ec mov -0x14(%rbp),%edx
	d: 8b 45 e8 mov -0x18(%rbp),%eax
#gcc -c -g test.c -o test.o	10: 01 d0 add %edx,%eax
#objdump -d -S test.o	12: 89 45 fc mov %eax,-0x4(%rbp)
	return x;
	15: 8b 45 fc mov -0x4(%rbp),%eax
	}
	18: 5d pop %rbp
	19: c3 retq

目标文件中存储  
代码

Q: 参数i, j, 局部变量x 的地址, 在指令中是否明确?

# 目标文件格式

Q: 目标文件中会存储什么信息?

```
root@LAPTOP-CJLSTBTI:/home/chapter4# od -tx1 test.o
00000000 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
00000020 01 00 3e 00 01 00 00 00 00 00 00 00 00 00 00 00
00000040 00 00 00 00 00 00 00 00 00 68 05 00 00 00 00 00
00000060 00 00 00 00 40 00 00 00 00 00 40 00 13 00 12 00
00000100 55 48 89 e5 89 7d ec 89 75 e8 8b 55 ec 8b 45 e8
00000120 01 d0 89 45 fc 8b 45 fc 5d c3 77 00 00 00 04 00
00000140 00 00 00 00 08 01 00 00 00 00 00 0c 00 00 00 00
```

```
root@LAPTOP-CJLSTBTI:/home/chapter4# od -Ax -tx1 test.o
000000 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
000010 01 00 3e 00 01 00 00 00 00 00 00 00 00 00 00 00
000020 00 00 00 00 00 00 00 00 00 68 05 00 00 00 00 00
000030 00 00 00 00 40 00 00 00 00 00 40 00 13 00 12 00
000040 55 48 89 e5 89 7d ec 89 75 e8 8b 55 ec 8b 45 e8
000050 01 d0 89 45 fc 8b 45 fc 5d c3 77 00 00 00 04 00
```

最左边是地址，对地址编码用的进制分别为 8/16进制

代码节 .text

# 目标文件格式

Q: 对于“节”，相关的信息有哪些？

节的名称？ 节从文件的什么位置开始？ 节有多长？

节的类型？（代码或数据节 1，符号表 2，字符串表 8）

节的访问属性？（可写、可执行……）

节执行时对应的虚拟地址……

```
typedef struct{
    Elf64_Word  sh_name;    /* Section name (string tbl index) : 4字节*/
    Elf64_Word  sh_type;    /* Section type : 4 */
    Elf64_Xword sh_flags;   /* Section flags : 8字节*/
    Elf64_Addr  sh_addr;    /* Section virtual addr at execution : 8字节*/
    Elf64_Off   sh_offset;  /* Section file offset : 8字节*/
    Elf64_Xword sh_size;    /* Section size in bytes : 8 */
    Elf64_Word  sh_link;    /* Link to another section : 4 */
    Elf64_Word  sh_info;    /* Additional section information : 4 */
    Elf64_Xword sh_addralign; /* Section alignment : 8 */
    Elf64_Xword sh_entsize; /* Entry size if section holds table : 8 */
} Elf64_Shdr;           // 64 个字节
```

节头描述项    Elf64\_Shdr    Elf32\_Shdr

参见：            /usr/include/elf.h

# 目标文件格式

**Q:** 对“节”的描述，含有哪些信息？

```
typedef struct{
```

```
    Elf64_Word sh_name;
```

**节名字符串在.shstrtab中的偏移**

```
    Elf64_Word sh_type;
```

**节类型：无效/代码或数据/符号/字符串/...**

```
    Elf64_Xword sh_flags;
```

**节标志：该节在虚拟空间中的访问属性**

```
    Elf64_Addr sh_addr;
```

**虚拟地址：若可被加载，则对应虚拟地址**

```
    Elf64_Off  sh_offset;
```

**在文件中的偏移地址，对.bss节而言则无意义**

```
    Elf64_Xword sh_size;
```

**节在文件中所占的长度**

```
    Elf64_Word sh_link;
```

**sh\_link和sh\_info用于与链接相关的节（如**

```
    Elf64_Word sh_info;
```

**.rel.text节、.rel.data节、.symtab节等）**

```
    Elf64_Xword sh_addralign;
```

**节的对齐要求**

```
    Elf64_Xword sh_entsize;
```

```
} Elf64_Shdr;
```

Executable and Linkable Format

**节头描述项 Elf64\_Shdr**

**节头表：由若干节头表项组成**



# 目标文件格式

```
# readelf -W -S test.o
```

Section Headers:

[Nr]	Name	Type	Address	Off	Size	ES	Flg	Lk	Inf	Al
[ 0]		NULL	0000000000000000	000000	000000	00		0	0	0
[ 1]	.text	PROGBITS	0000000000000000	000040	00001a	00	AX	0	0	1
[ 2]	.data	PROGBITS	0000000000000000	00005a	000000	00	WA	0	0	1
[ 3]	.bss	NOBITS	0000000000000000	00005a	000000	00	WA	0	0	1
[ 4]	.debug_info	PROGBITS	0000000000000000	00005a	00007b	00		0	0	1
[ 5]	.rela.debug_info	RELA	0000000000000000	0003b8	0000a8	18	I	16	4	8
[ 6]	.debug_abbrev	PROGBITS	0000000000000000	0000d5	000059	00		0	0	1
[ 7]	.debug_aranges	PROGBITS	0000000000000000	00012e	000030	00		0	0	1
[ 8]	.rela.debug_aranges	RELA	0000000000000000	000460	000030	18	I	16	7	8
[ 9]	.debug_line	PROGBITS	0000000000000000	00015e	00003b	00		0	0	1
[10]	.rela.debug_line	RELA	0000000000000000	000490	000018	18	I	16	9	8
[11]	.debug_str	PROGBITS	0000000000000000	000199	00005d	01	MS	0	0	1
[12]	.comment	PROGBITS	0000000000000000	0001f6	00002a	01	MS	0	0	1
[13]	.note.GNU-stack	PROGBITS	0000000000000000	000220	000000	00		0	0	1
[14]	.eh_frame	PROGBITS	0000000000000000	000220	000038	00	A	0	0	8
[15]	.rela.eh_frame	RELA	0000000000000000	0004a8	000018	18	I	16	14	8
[16]	.symtab	SYMTAB	0000000000000000	000258	000150	18		17	13	8
[17]	.strtab	STRTAB	0000000000000000	0003a8	00000c	00		0	0	1
[18]	.shstrtab	STRTAB	0000000000000000	0004c0	0000a3	00		0	0	1

Key to Flags:

W (write), A (alloc), X (execute), M (merge), S (strings), I (info),  
L (link order), O (extra OS processing required), G (group), T (TLS),  
C (compressed), x (unknown), o (OS specific), E (exclude),  
l (large), p (processor specific)

# 节头描述项 Elf64\_Shdr 解读

---

**节名 sh\_name:** 占 4个字节

节名字符串在 .shstrtab 中的偏移

.text、.data、.shstrtab……

**节类型 sh\_type:** 4个字节

SHT\_NULL          无效节

SHT\_PROGBITS      代码或数据节

SHT\_SYMTAB        符号表节

SHT\_HASH          符号表的哈希表

SHT\_STRTAB        字符串节

SHT\_RELA          重定位表

SHT\_NOBITS        表示该节在文件中没有内容, 如 .bss 节

SHT\_DYNAMIC       动态链接信息

SHT\_DNYSYM        动态链接的符号表

SHT\_NOTE          提示性信息

# 节头描述项 Elf64\_Shdr 解读

---

## 节标志 sh\_flags:

该节在虚拟空间中的访问属性

SHF\_WRITE (W) 在进程空间中可写

SHF\_ALLOC (A) 在进程空间中需要分配空间

包含指示或控制信息的节不分配空间

SHF\_EXECINSTR 该节在进程空间中可以被执行

SHF\_MERGE 可以被合并

SHF\_STRINGS 字符串

SHF\_INFO\_LINK

SHF\_LINK\_ORDER

如果节的类型是和链接相关的，比如重定位表、符号表等，那么sh\_link和sh\_info两个成员有意义。对于其他段，这两个成员没有意义。

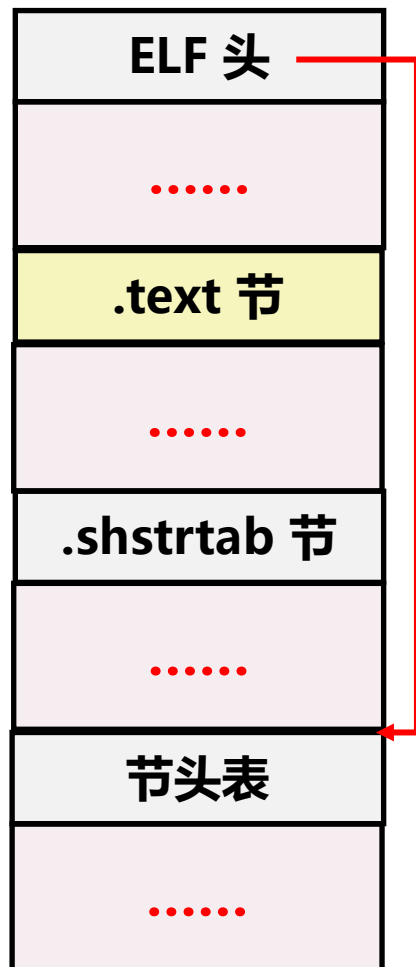
# 节头描述项 Elf64\_Shdr 解读

---

- **节名字符串索引：** 在.shstrtab中的偏移
- **节类型：** 无效/代码或数据/符号/字符串/...
- **节标志：** 该节在虚拟空间中的访问属性
- **虚拟地址：** 若可被加载，则对应虚拟地址
- **在文件中的偏移地址，** 对.bss节而言则无意义
- **节在文件中所占的长度**
- **节的链接信息**  
在本节是用于链接时（由节类型确定），才有效  
本节使用的符号表的在哪一个节中；用于对哪一个节的链接
- **节的对齐要求**

# 目标文件格式

Q: “节头表”又在文件的什么位置?



从文件头偏移 40个字节开始,  
有 一个 8个字节的偏移量,  
指明节头表在文件中的开始位置  
**// e\_shoff = 00 00 00 00 00 00 05 68**

```
root@LAPTOP-CJLSTBT1:/home/chapter4# od -Ax -txl test.o
000000 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
000010 01 00 3e 00 01 00 00 00 00 00 00 00 00 00 00 00
000020 00 00 00 00 00 00 00 00 00 68 05 00 00 00 00 00
000030 00 00 00 00 40 00 00 00 00 00 40 00 13 00 12 00
000040 55 48 89 e5 89 7d ec 89 75 e8 8b 55 ec 8b 45 e8
000050 01 d0 89 45 fc 8b 45 fc 5d c3 77 00 00 00 04 00
```

# 目标文件格式

## ELF 头在目标文件的起始位置

```
#define EI_NIDENT (16)
typedef struct{
    unsigned char  e_ident[EI_NIDENT]; /* Magic number and other info : 16*/
    Elf64_Half     e_type;              /* Object file type : 2 */
    Elf64_Half     e_machine;           /* Architecture : 2 */
    Elf64_Word     e_version;           /* Object file version : 4 */
    Elf64_Addr     e_entry;             /* Entry point virtual address :8 */
    Elf64_Off      e_phoff;             /* Program header table file offset :8 */
    Elf64_Off      e_shoff;             /* Section header table file offset : 8 */
    Elf64_Word     e_flags;             /* Processor-specific flags : 4 */
    Elf64_Half     e_ehsize;            /* ELF header size in bytes */
    Elf64_Half     e_phentsize;         /* Program header table entry size */
    Elf64_Half     e_phnum;            /* Program header table entry count */
    Elf64_Half     e_shentsize;         /* Section header table entry size */
    Elf64_Half     e_shnum;            /* Section header table entry count */
    Elf64_Half     e_shstrndx;         /* Section header string table index */
} Elf64_Ehdr; // 64 个字节
```

**ELF 头结构**

# ELF头 (ELF Header)

---

e\_ident : ELF魔数、版本、小端/大端、操作系统平台  
e\_type : 目标文件的类型  
e\_machine : 机器结构类型  
e\_version: 目标文件的版本  
e\_entry : 程序执行的入口地址  
e\_phoff : 程序头表（段头表）的起始位置  
e\_shoff : 节头表的起始位置  
e\_flags : 处理器标志  
e\_ehsize: ELF 头的大小  
e\_phentsize: 程序头表结构的大小（一个表项的长度）  
e\_phnum : 程序头表的条目数  
e\_shentsize : 节头表结构的大小  
e\_shnum : 节头表的条目数  
e\_shstrndx; 节头字符串表的索引

# 目标文件格式

## ELF 头

```
root@LAPTOP-CJLSTBTI:/home/chapter4# readelf -h test.o
ELF Header:
  Magic:      7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                                ELF64
  Data:                                2's complement, little endian
  Version:                                1 (current)
  OS/ABI:                                UNIX - System V
  ABI Version:                            0
  Type:                                REL (Relocatable file)
  Machine:                                Advanced Micro Devices X86-64
  Version:                                0x1
  Entry point address:                    0x0
  Start of program headers:                0 (bytes into file)
  Start of section headers:                1384 (bytes into file)
  Flags:                                0x0
  Size of this header:                    64 (bytes)
  Size of program headers:                0 (bytes)
  Number of program headers:                0
  Size of section headers:                64 (bytes)
  Number of section headers:                19
  Section header string table index:      18
```



# 目标文件格式

```
root@LAPTOP-CJLSTBTI:/home/chapter4# od -Ax -tx1 -v -j0x568 test.o
000568 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000578 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000588 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000598 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0005a8 1b 00 00 00 01 00 00 00 06 00 00 00 00 00 00 00 00
0005b8 00 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00 00
0005c8 1a 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0005d8 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0005e8 21 00 00 00 01 00 00 00 03 00 00 00 00 00 00 00 00
0005f8 00 00 00 00 00 00 00 00 5a 00 00 00 00 00 00 00 00
000608 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000618 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

0x 00000568 处开始节头表；每个节头表项 64个字节

## .text节的节头描述

0000001b：节的名字 在节名字字符串表（**.shstrtab**）的位置

00000001：节的类型，1 表示 为 代码或者数据节

000000000000000006：节在虚拟空间中的访问属性

000000000000000000：被加载时对应的虚拟地址

000000000000000040：在文件中的偏移地址

00000000000000001a：节的长度

# 目标文件格式

```
#readelf -W -S test.o
```

```
root@LAPTOP-CJLSTBTI:/home/chapter4# readelf -W -S test.o
There are 19 section headers, starting at offset 0x568:
```

```
Section Headers:
```

[Nr]	Name	Type	Address	Off	Size	ES	Flg	Lk	Inf	Al
[ 0]		NULL	0000000000000000	000000	000000	00		0	0	0
[ 1]	.text	PROGBITS	0000000000000000	000040	00001a	00	AX	0	0	1
[ 2]	.data	PROGBITS	0000000000000000	00005a	000000	00	WA	0	0	1
[ 3]	.bss	NOBITS	0000000000000000	00005a	000000	00	WA	0	0	1
[ 4]	.debug_info	PROGBITS	0000000000000000	00005a	00007b	00		0	0	1
[ 5]	.rela.debug_info	RELA	0000000000000000	0003b8	0000a8	18	I	16	4	8
[ 6]	.debug_abbrev	PROGBITS	0000000000000000	0000d5	000059	00		0	0	1
[ 7]	.debug_aranges	PROGBITS	0000000000000000	00012e	000030	00		0	0	1
[ 8]	.rela.debug_aranges	RELA	0000000000000000	000460	000030	18	I	16	7	8
[ 9]	.debug_line	PROGBITS	0000000000000000	00015e	00003b	00		0	0	1
[10]	.rela.debug_line	RELA	0000000000000000	000490	000018	18	I	16	9	8
[11]	.debug_str	PROGBITS	0000000000000000	000199	00005d	01	MS	0	0	1
[12]	.comment	PROGBITS	0000000000000000	0001f6	00002a	01	MS	0	0	1
[13]	.note.GNU-stack	PROGBITS	0000000000000000	000220	000000	00		0	0	1
[14]	.eh_frame	PROGBITS	0000000000000000	000220	000038	00	A	0	0	8
[15]	.rela.eh_frame	RELA	0000000000000000	0004a8	000018	18	I	16	14	8
[16]	.symtab	SYMTAB	0000000000000000	000258	000150	18		17	13	8
[17]	.strtab	STRTAB	0000000000000000	0003a8	00000c	00		0	0	1
[18]	.shstrtab	STRTAB	0000000000000000	0004c0	0000a3	00		0	0	1

第1行指明有 19个节，节头表 在 0x568处开始

# 目标文件格式

```
#readelf -S -W test.o
```

```
[18] .shstrtab          STRTAB          0000000000000000 0004c0 0000a3 00          0  0  1
```

```
#hexdump -C test.o
```

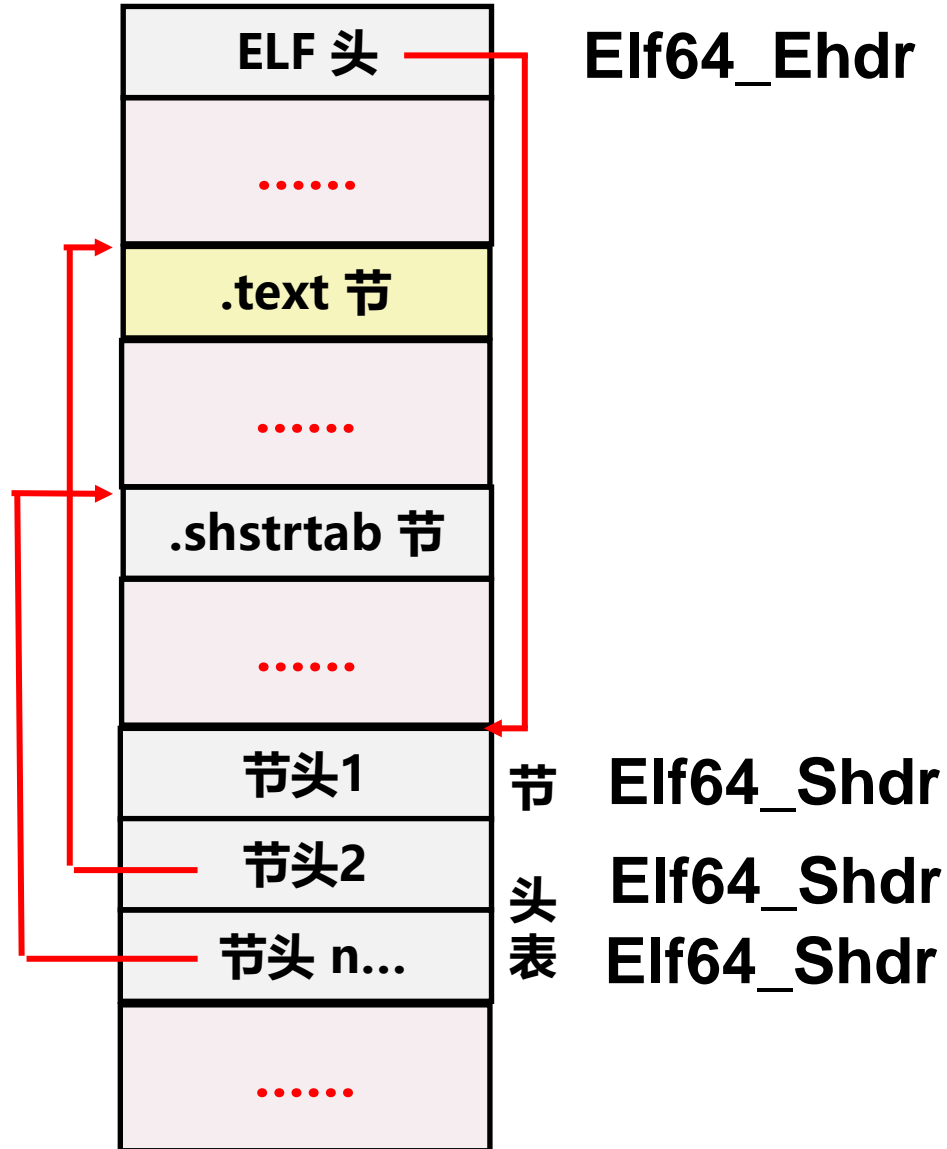
000004c0	00 2e 73 79 6d 74 61 62	00 2e 73 74 72 74 61 62	..symtab..strtab
000004d0	00 2e 73 68 73 74 72 74	61 62 00 2e 74 65 78 74	..shstrtab..text
000004e0	00 2e 64 61 74 61 00 2e	62 73 73 00 2e 72 65 6c	..data..bss..rel
000004f0	61 2e 64 65 62 75 67 5f	69 6e 66 6f 00 2e 64 65	a.debug_info..de
00000500	62 75 67 5f 61 62 62 72	65 76 00 2e 72 65 6c 61	bug_abbrev..rela
00000510	2e 64 65 62 75 67 5f 61	72 61 6e 67 65 73 00 2e	.debug_aranges..
00000520	72 65 6c 61 2e 64 65 62	75 67 5f 6c 69 6e 65 00	rela.debug_line.
00000530	2e 64 65 62 75 67 5f 73	74 72 00 2e 63 6f 6d 6d	.debug_str..comm
00000540	65 6e 74 00 2e 6e 6f 74	65 2e 47 4e 55 2d 73 74	ent..note.GNU-st
00000550	61 63 6b 00 2e 72 65 6c	61 2e 65 68 5f 66 72 61	ack..rela.eh_fra
00000560	6d 65 00 00 00 00 00 00	00 00 00 00 00 00 00 00	me.....

节名字符串表（.shstrtab）从 0x004c0 开始

节名字符串“.text”是从该表的 0x001b 处开始

节名字符串以 00 作为分隔符

# 目标文件格式



# 目标文件格式

**Q:** 有全局变量时，目标文件中会存储什么信息？

test\_v.c

```
extern int g2;  
int g1=30;  
int add(int i,int j) {  
    int x=i+j;  
    x+=g1;  
    x+=g2;  
    return x;  
}
```

```
#gcc -c -g test_v.c -o test_v.o
```

```
#gcc -c -g main.c -o main.o
```

```
#gcc -g main.o test_v.o -o main
```

main.c

```
int add(int,int);  
int g2=50;  
int main(){  
    return add(15,20);  
}
```

**Q:** test\_v.c 中含有全局变量，如何确定g1, g2 的地址？

# 目标文件格式

Q: 不同的组装顺序，全局变量的地址是否发生变化？

```
test_v.c
extern int g2;
Int g1=30;
int add(int i,int j) {
}
```

```
#gcc -g main.c test_v.c -o main_test
#gcc -g test_v.c main.c -o test_main
```

```
#gdb main_test
```

```
(gdb) print &g1
$2 = (int *) 0x8201014 <g1>
(gdb) print &g2
$3 = (int *) 0x8201010 <g2>
```

```
#gdb test_main
```

```
(gdb) print &g1
$1 = (int *) 0x8201010 <g1>
(gdb) print &g2
$2 = (int *) 0x8201014 <g2>
```

结论：不同的组装顺序，全局变量的地址会发生变化！  
这也表明，在 test\_v.o 中不能确定全局变量的地址！

# 目标文件格式

0000000000000000 <add>:

extern int g2;

Int g1=30;

int add(int i,int j){

```
0:      55          push  %rbp
1:      48 89 e5     mov   %rsp,%rbp
4:      89 7d ec     mov   %edi,-0x14(%rbp)
7:      89 75 e8     mov   %esi,-0x18(%rbp)
```

int x=i+j;

```
a:      8b 55 ec     mov   -0x14(%rbp),%edx
d:      8b 45 e8     mov   -0x18(%rbp),%eax
10:     01 d0       add   %edx,%eax
12:     89 45 fc     mov   %eax,-0x4(%rbp)
```

x+=g1;

```
15: 8b 05 00 00 00 00 mov  0x0(%rip),%eax
      # 1b <add+0x1b>
```

```
1b: 01 45 fc          add  %eax,-0x4(%rbp)
```

x+=g2;

```
1e: 8b 05 00 00 00 00 mov  0x0(%rip),%eax
      # 24 <add+0x24>
```

```
24: 01 45 fc          add  %eax,-0x4(%rbp)
```

.....

test\_v.o

在代码中，对全局变量，翻译出的只是 00 00 00 00。

该内容是要修改的。

这就需要记录：

代码节中的什么位置，要被替换成一个什么类型的值，该值来源于哪一个变量

# 代码节的重定位信息

0000000000000000 <add>:

```
int add(int i,int j){
```

```
.....
```

```
    x+=g1;
```

```
15: 8b 05 00 00 00 00    mov    0x0(%rip),%eax
```

```
1b: 01 45 fc              add    %eax,-0x4(%rbp)
```

```
    x+=g2;
```

```
1e: 8b 05 00 00 00 00    mov    0x0(%rip),%eax
```

```
24: 01 45 fc              add    %eax,-0x4(%rbp)
```

```
.....
```

```
}
```

```
# objdump -r test_v.o
```

```
.rela.text      代码节的重定位信息
```

RELOCATION RECORDS FOR [.text]:

OFFSET	TYPE	VALUE
0000000000000017	R_X86_64_PC32	g1-0x0000000000000004
0000000000000020	R_X86_64_PC32	g2-0x0000000000000004

.text节中，地址为 0x17、0x20 处分别要重定位为 g1、g2的地址



# 代码节的重定位信息

```
# readelf -S -W test_v.o
```

```
.rela.text  代码节的重定位信息
```

```
Section Headers:
```

[Nr]	Name	Type	Address	Off	Size	ES	Flg	Lk	Inf	Al
[ 0]		NULL	0000000000000000	000000	000000	00		0	0	0
[ 1]	.text	PROGBITS	0000000000000000	000040	00002c	00	AX	0	0	1
[ 2]	.rela.text	RELA	0000000000000000	000448	000030	18	I 17	1	8	

```
root@LAPTOP-CJLSTBTI:/home/chapter4# od -j0x448 -Ax -tx1 test_v.o
000448 17 00 00 00 00 00 00 00 02 00 00 00 0d 00 00 00
000458 fc ff ff ff ff ff ff 20 00 00 00 00 00 00 00
000468 02 00 00 00 0f 00 00 00 fc ff ff ff ff ff ff
```

```
Relocation section '.rela.text' at offset 0x448 contains 2 entries:
```

Offset	Info	Type	Symbol's Value	Symbol's Name + Addend
0000000000000017	0000000d00000002	R_X86_64_PC32	0000000000000000	g1 - 4
0000000000000020	0000000f00000002	R_X86_64_PC32	0000000000000000	g2 - 4

```
typedef struct{
    Elf64_Addr    r_offset; /* Address 要重位的位置 */
    Elf64_Xword   r_info;   /* Relocation type and symbol index */
                        /* 重定位的方式，及符号表的哪一项 */
    Elf64_Sxword  r_addend; /* Addend 常量加数*/
} Elf64_Rela; // 24 个字节    24* 2 = 48 = 0x0030
```

# 代码节的重定位信息

---

- 任何引用全局变量或者调用函数的指令都需要被修正;
- 引用局部变量的指令不需要重定位;
- 在可执行目标文件中不存在重定位信息。

例: `r_offset` : `0x0000000000000017`

`r_info` : `0x0000000d00000002`

分解出定位方式: `0x00000002`, 即 `R_X86_64_PC32`

符号表的第几项: `0x0000000d`, 即 符号表中的第13项

`r_addend` : `0xfffffffffffffffffc`, 即 `-4`

```
typedef struct{
    Elf64_Addr    r_offset; /* Address 要重定位的位置 */
    Elf64_Xword   r_info;   /* Relocation type and symbol index */
                        /* 重定位的方式, 及符号表的哪一项 */
    Elf64_Sxword  r_addend; /* Addend 常量加数 */
} Elf64_Rela; // 24 个字节 24* 2 = 48 = 0x0030
```

# 符号表节 .symtab

---

```
13: 0000000000000000      4 OBJECT  GLOBAL DEFAULT    3 g1
14: 0000000000000000     44 FUNC    GLOBAL DEFAULT    1 add
15: 0000000000000000      0 NOTYPE  GLOBAL DEFAULT   UND g2
```

- g1 定义在 第3节(.data), 地址为 0
- add 定义在 第1节 (.text), 地址为 0
- g2 : 未定义

# 符号表 解读

```
typedef struct{
    Elf64_Word  st_name;    /* Symbol name (string tbl index) */
    unsigned char st_info;   /* Symbol type and binding */
    // 符号可以是数据、函数. Binding : 本地、全局
    unsigned char st_other; /* Symbol visibility */
    Elf64_Section st_shndx; /* Section index 符号被分配到的节号*/
    Elf64_Addr st_value;    /* Symbol value */
    Elf64_Xword st_size;    /* Symbol size 符号存储空间的大小*/
} Elf64_Sym;      24个字节
```

```
[17] .symtab          SYMTAB          0000000000000000 0002b0 000180
```

```
14: 0000000000000000 44 FUNC GLOBAL DEFAULT 1 add
```

test\_v.o 中, 有 16个符号条目;  $16 * 24 = 384 = 0x180$

对符号 add的描述:  $0x2b0 + 0x150 (14*24) = 0x400$

名字从字符串表的第13个字符开始 ->可从字符串表取出名字 add;

type 为 2 (st\_info的低4位, STT\_FUNC =2),

Binding 为 1 (STB\_GLOBAL=1)

为 add分配的空间 是 0000000000000002c个字节

```
|000400 0d 00 00 00 12 00 01 00 00 00 00 00 00 00 00 00 00|
|000410 2c 00 00 00 00 00 00 00 00 11 00 00 00 10 00 00 00|
```

# 几个节的整体解读

```
int g1=10;
char c1='a';
char c2='b';
short s1=25;
int g2=20;
int g3=30;
```

```
int main()
{ int x;
  x=g1+g2+g3;
  return 0;
}
```

.symtab 节

节头表—关于.data 的描述

.data 节

.strtab节

13:	000000000000000000	4	OBJECT	GLOBAL	DEFAULT	3	g1
14:	000000000000000004	1	OBJECT	GLOBAL	DEFAULT	3	c1
15:	000000000000000005	1	OBJECT	GLOBAL	DEFAULT	3	c2
16:	000000000000000006	2	OBJECT	GLOBAL	DEFAULT	3	s1
17:	000000000000000008	4	OBJECT	GLOBAL	DEFAULT	3	g2
18:	00000000000000000c	4	OBJECT	GLOBAL	DEFAULT	3	g3
19:	000000000000000000	36	FUNC	GLOBAL	DEFAULT	1	main

.data	PROGBITS	000000000000000000	000064	000010
-------	----------	--------------------	--------	--------

```
root@LAPTOP-CJLSTBTI:/home/chapter4# od -j0x64 -tx1 -Ax global.o
000064 0a 00 00 00 61 62 19 00 14 00 00 00 1e 00 00 00
```

000004e0	00 67 6c 6f 62 61 6c 2e 63 00 67 31 00 63 31 00	.global.c.g1.c1.
000004f0	63 32 00 73 31 00 67 32 00 67 33 00 6d 61 69 6e	c2.s1.g2.g3.main

# 几个节的整体解读

0000000000000000 <main>:

char c2='b';

short s1=25;

int g2=20;

int g3=30;

int main()

```
{ 0: 55          push  %rbp
  1: 48 89 e5     mov   %rsp,%rbp
```

int x;

x=g1+g2+g3;

```
  4: 8b 15 00 00 00 00  mov   0x0(%rip),%edx
```

```
  a: 8b 05 00 00 00 00  mov   0x0(%rip),%eax
```

```
 10: 01 c2          add   %eax,%edx
```

```
 12: 8b 05 00 00 00 00  mov   0x0(%rip),%eax
```

```
 18: 01 d0          add   %edx,%eax
```

```
 1a: 89 45 fc       mov   %eax,-0x4(%rbp)
```

return 0;

```
 1d: b8 00 00 00 00  mov   $0x0,%eax
```

```
}
```

```
 22: 5d            pop   %rbp
```

```
 23: c3            retq
```

# .data节的重定位信息

---

- 在定义全局变量时，可以用其他全局变量或者函数初始化，因而也存在要重定位的问题。

```
// test_rela_data.c
int add(int,int);
extern int g2;
int g1=30;
int *p = &g1;
int (*q)(int ,int)=add;
int add(int i,int j)
{ int x=i+j;
  x+=g1;
  x+=g2;
  return x;
}
```

# .data节的重定位信息

```
root@LAPTOP-CJLSTBTI:/home/chapter4# readelf -S -W test_rela_data.o
There are 22 section headers, starting at offset 0x788:
```

Section Headers:

[Nr]	Name	Type	Address	Off	Size	ES	Flg	Lk	Inf	Al
[ 0]		NULL	0000000000000000	000000	000000	00		0	0	0
[ 1]	.text	PROGBITS	0000000000000000	000040	00002c	00	AX	0	0	1
[ 2]	.rela.text	RELA	0000000000000000	000518	000030	18	I	19	1	8
[ 3]	.data	PROGBITS	0000000000000000	00006c	000004	00	WA	0	0	4
[ 4]	.bss	NOBITS	0000000000000000	000070	000000	00	WA	0	0	1
[ 5]	.data.rel.local	PROGBITS	0000000000000000	000070	000010	00	WA	0	0	8
[ 6]	.rela.data.rel.local	RELA	0000000000000000	000548	000030	18	I	19	5	8
[ 7]	.debug_info	PROGBITS	0000000000000000	000080	0000db	00		0	0	1
[ 8]	.rela.debug_info	RELA	0000000000000000	000578	0000f0	18	I	19	7	8

Relocation section '.rela.data.rel.local' at offset 0x548 contains 2 entries:

Offset	Info	Type	Symbol's Value	Symbol's Name + Addend
0000000000000000	0000000e00000001	R_X86_64_64	0000000000000000	g1 + 0
0000000000000008	0000001100000001	R_X86_64_64	0000000000000000	add + 0

Relocation section '.rela.text' at offset 0x518 contains 2 entries:

Offset	Info	Type	Symbol's Value	Symbol's Name + Addend
0000000000000017	0000000e00000002	R_X86_64_PC32	0000000000000000	g1 - 4
0000000000000020	0000001200000002	R_X86_64_PC32	0000000000000000	g2 - 4

g1 是符号表的第 14项; g2 是第18项; add 是 17项

定位方式 : R\_X86\_64\_64 直接的 64位地址

R\_X86\_64\_PC32 // PC relative 32 bit signed



# .data节的重定位信息

## 符号表中的信息

- `readelf -s -W test_rela_data.o`
- `g1` 定义在 第3节 (`.data`), 地址为 0  
    `p`, `q`, 定义在 第5节 (`.data.rel.local`), 地址分别为 0, 8  
    `add` 定义在 第1节 (`.text`), 地址为 0  
    `g2` : 未定义

```
14: 0000000000000000      4 OBJECT GLOBAL DEFAULT   3 g1
15: 0000000000000000      8 OBJECT GLOBAL DEFAULT   5 p
16: 0000000000000008      8 OBJECT GLOBAL DEFAULT   5 q
17: 0000000000000000     44 FUNC    GLOBAL DEFAULT   1 add
18: 0000000000000000      0 NOTYPE  GLOBAL DEFAULT  UND g2
```

# 目标文件格式

---

**Q:** 字符串"hello %d\n", 放在何处?

**printf** 翻译成什么语句?

```
/* test_all.c */
#include <stdio.h>
extern int g2;
int g1=30;
int add(int i,int j)
{
    int x=i+j;
    x+=g1;
    x+=g2;
    printf("hello %d \n",i);
    printf("very good %d\n",x);
    return x;
}
```

# 目标文件格式

```
extern int g2;      int g1=30;
    x+=g1;
19:  8b 05 00 00 00 00      mov     0x0(%rip),%eax    # 1f <add+0x1f>
1f:  01 45 fc                add     %eax,-0x4(%rbp)
    x+=g2;
22:  8b 05 00 00 00 00      mov     0x0(%rip),%eax    # 28 <add+0x28>
28:  01 45 fc                add     %eax,-0x4(%rbp)
    printf("hello %d \n",i);
2b:  8b 45 ec                mov     -0x14(%rbp),%eax
2e:  89 c6                  mov     %eax,%esi
30:  48 8d 3d 00 00 00 00    lea     0x0(%rip),%rdi    # 37 <add+0x37>
37:  b8 00 00 00 00        mov     $0x0,%eax
3c:  e8 00 00 00 00        callq   41 <add+0x41>
    printf("very good %d\n",x);
41:  8b 45 fc                mov     -0x4(%rbp),%eax
44:  89 c6                  mov     %eax,%esi
46:  48 8d 3d 00 00 00 00    lea     0x0(%rip),%rdi    # 4d <add+0x4d>
4d:  b8 00 00 00 00        mov     $0x0,%eax
52:  e8 00 00 00 00        callq   57 <add+0x57>
```

**Q: 符号有哪些? 全局变量、函数、常量字符串**

# 目标文件格式

```
root@LAPTOP-CJLSTBTI:/home/chapter4# readelf -r -W test_all.o
```

```
Relocation section '.rela.text' at offset 0xb50 contains 6 entries:
```

Offset	Info	Type	Symbol's Value	Symbol's Name
000000000000001b	0000000e00000002	R_X86_64_PC32	0000000000000000	g1 - 4
0000000000000024	0000001000000002	R_X86_64_PC32	0000000000000000	g2 - 4
0000000000000033	0000000500000002	R_X86_64_PC32	0000000000000000	.rodata - 4
000000000000003d	0000001200000004	R_X86_64_PLT32	0000000000000000	printf - 4
0000000000000049	0000000500000002	R_X86_64_PC32	0000000000000000	.rodata + 7
0000000000000053	0000001200000004	R_X86_64_PLT32	0000000000000000	printf - 4

.text 中有 6项 需要重新定位

其中，有2个是函数地址，R\_X86\_64\_PLT32

有2个是常量字符串的首地址，在 .rodata 节中

有2个是全局变量

# 目标文件格式

```
root@LAPTOP-CJLSTBTI:/home/chapter4# readelf -s -W test_all.o
```

```
Symbol table '.symtab' contains 19 entries:
```

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
0:	0000000000000000	0	NOTYPE	LOCAL	DEFAULT	UND	
1:	0000000000000000	0	FILE	LOCAL	DEFAULT	ABS	test_all.c
2:	0000000000000000	0	SECTION	LOCAL	DEFAULT	1	
3:	0000000000000000	0	SECTION	LOCAL	DEFAULT	3	
4:	0000000000000000	0	SECTION	LOCAL	DEFAULT	4	
5:	0000000000000000	0	SECTION	LOCAL	DEFAULT	5	
6:	0000000000000000	0	SECTION	LOCAL	DEFAULT	6	
7:	0000000000000000	0	SECTION	LOCAL	DEFAULT	8	
8:	0000000000000000	0	SECTION	LOCAL	DEFAULT	9	
9:	0000000000000000	0	SECTION	LOCAL	DEFAULT	11	
10:	0000000000000000	0	SECTION	LOCAL	DEFAULT	13	
11:	0000000000000000	0	SECTION	LOCAL	DEFAULT	15	
12:	0000000000000000	0	SECTION	LOCAL	DEFAULT	16	
13:	0000000000000000	0	SECTION	LOCAL	DEFAULT	14	
14:	0000000000000000	4	OBJECT	GLOBAL	DEFAULT	3	g1
15:	0000000000000000	92	FUNC	GLOBAL	DEFAULT	1	add
16:	0000000000000000	0	NOTYPE	GLOBAL	DEFAULT	UND	g2
17:	0000000000000000	0	NOTYPE	GLOBAL	DEFAULT	UND	_GLOBAL_OFFSET_TABLE_
18:	0000000000000000	0	NOTYPE	GLOBAL	DEFAULT	UND	printf

# 目标文件格式

## .rodata 节

[ 1]	.text	PROGBITS	000000000000000000 000040	00005c	00	AX	0
[ 2]	.rela.text	RELA	000000000000000000 000b50	000090	18	I	18
[ 3]	.data	PROGBITS	000000000000000000 00009c	000004	00	WA	0
[ 4]	.bss	NOBITS	000000000000000000 0000a0	000000	00	WA	0
[ 5]	.rodata	PROGBITS	000000000000000000 0000a0	000019	00	A	0
[ 6]	.debug info	PROGBITS	000000000000000000 0000b9	000360	00		0

000000a0	68	65	6c	6c	6f	20	25	64	20	0a	00	76	65	72	79	20	hello %d ..very
000000b0	67	6f	6f	64	20	25	64	0a	00	5c	03	00	00	04	00	00	good %d.. \. ....

# 可重定位目标文件格式

0

## ELF 头

- ✓ 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

- ✓ 编译汇编后的代码部分

## .rodata 节

- ✓ 只读数据，如 printf 格式串、switch 跳转表等

## .data 节

- ✓ 已初始化的全局变量

## .bss 节

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

SKIP

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 table (节头表)

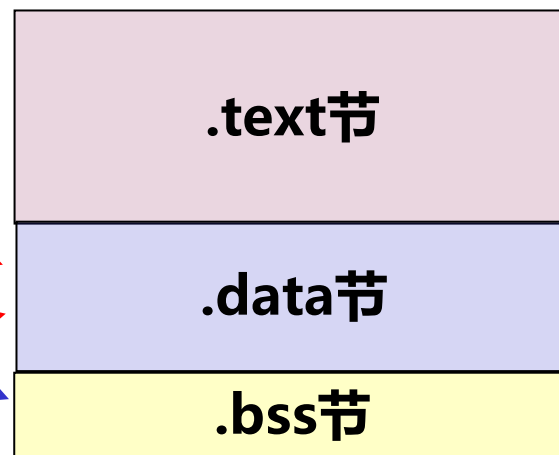
# 执行视图—可执行目标文件

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

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

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

## ELF的执行视图



**程序头表**用来说明段信息，也称**段头表**

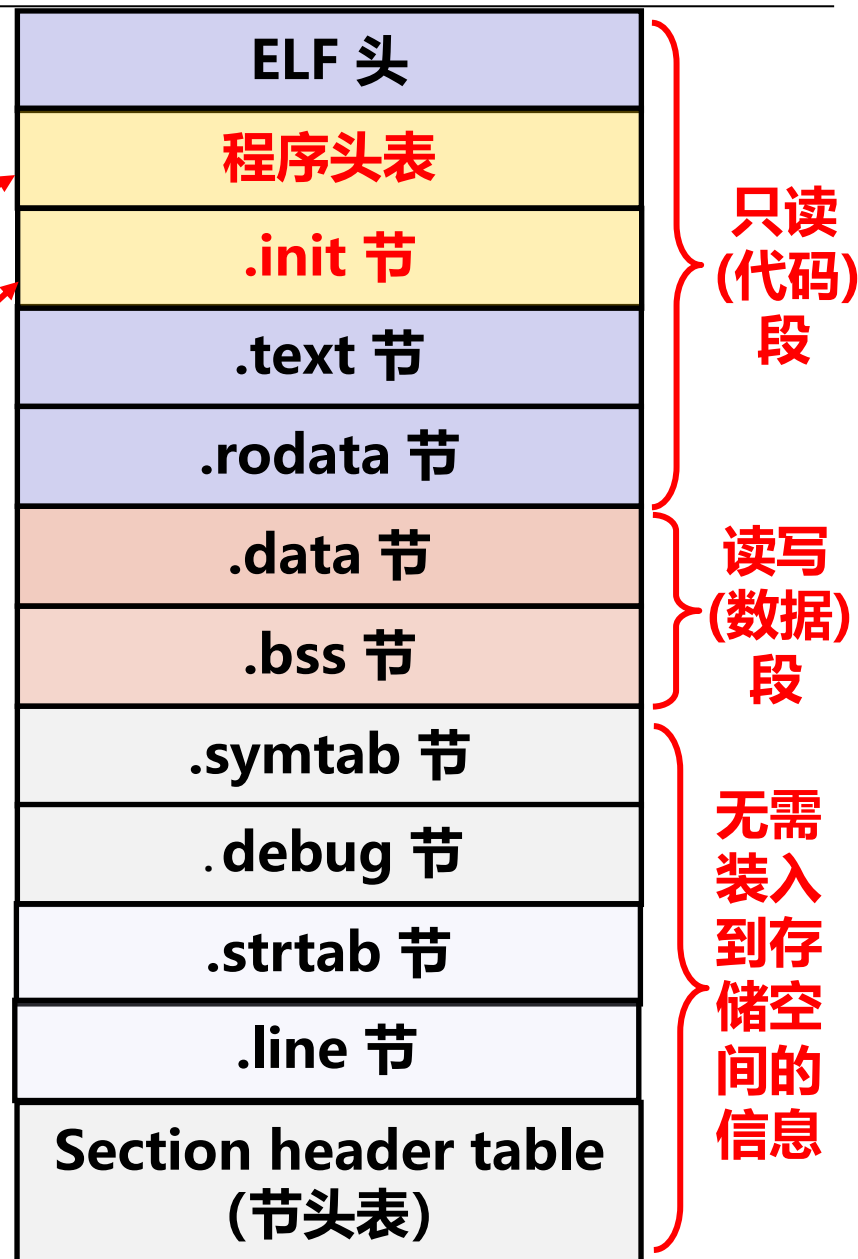
为了能执行，还需将**具相同访问属性**的节合并成**段 (Segment)**，并说明每个段的属性，如：在可执行文件中的位移、大小、在虚拟空间中的位置、对齐方式、访问属性等



# 可执行目标文件格式

- 与可重定位文件稍有不同:

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



# ELF头信息举例

**\$ readelf -h main**    可执行目标文件的ELF头

ELF Header:

Magic: 7f 45 4c 46 02 01 01 00 00 00 00 ...00 00

Class: ELF64

Data: 2's complement, little endian

Version: 1 (current)

OS/ABI: UNIX - System V

ABI Version: 0

Type: DYN (Shared object file)

Machine: Advanced Micro Devices X86-64

Version: 0x1

Entry point address: 0x4f0

Start of program headers: 64 (bytes into file)

Start of section headers: 7448 (bytes into file)

Flags: 0x0

Size of this header: 64 (bytes)

Size of program headers: 56 (bytes)

Number of program headers: 9

Size of section headers: 64 (bytes)

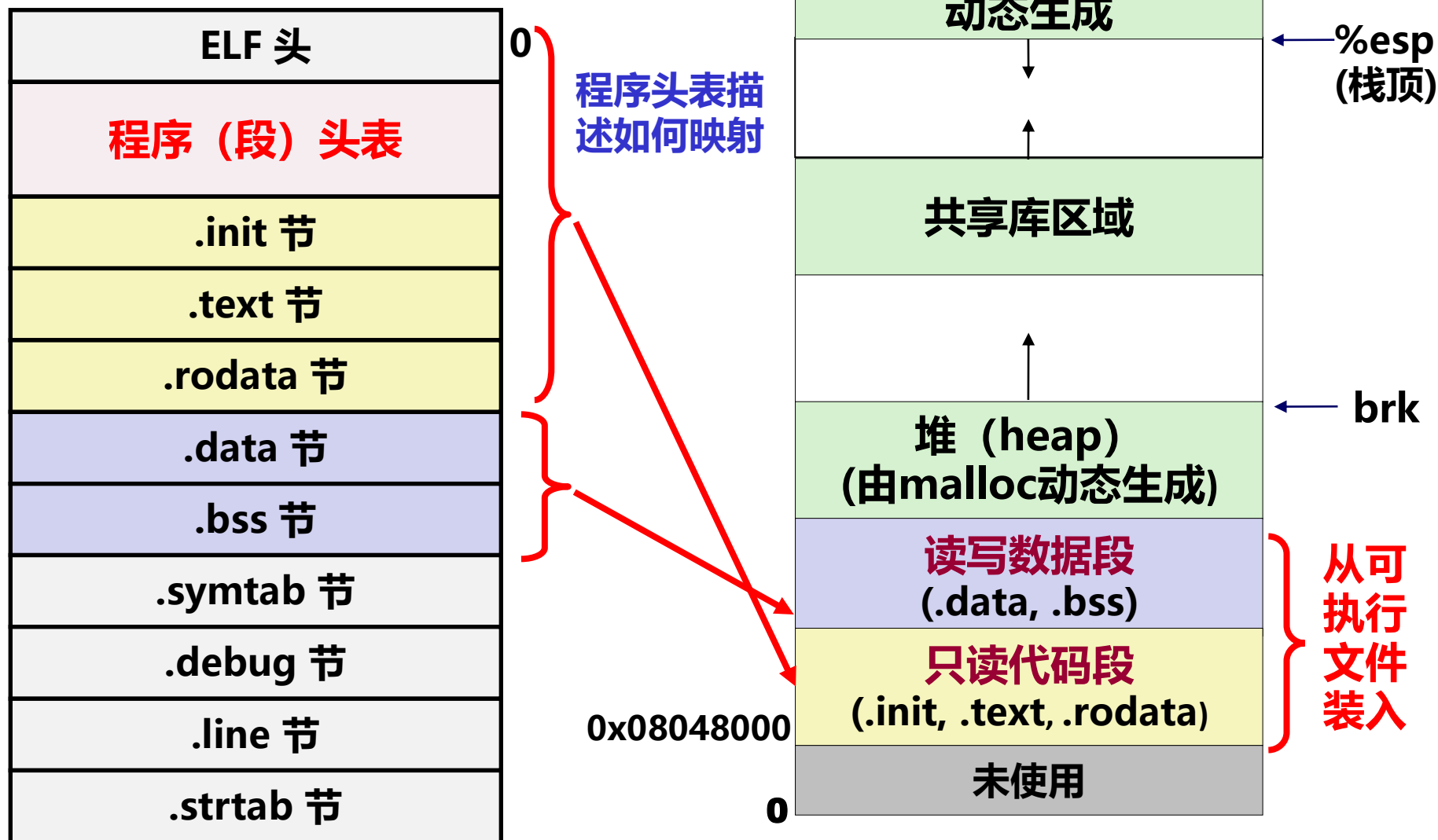
Number of section headers: 33

Section header string table index: 32

ELF 头
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)

# i386 System V 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)

**\$ readelf -l main**

Program Headers:

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

# 可执行文件中的程序头表

Program Headers:

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

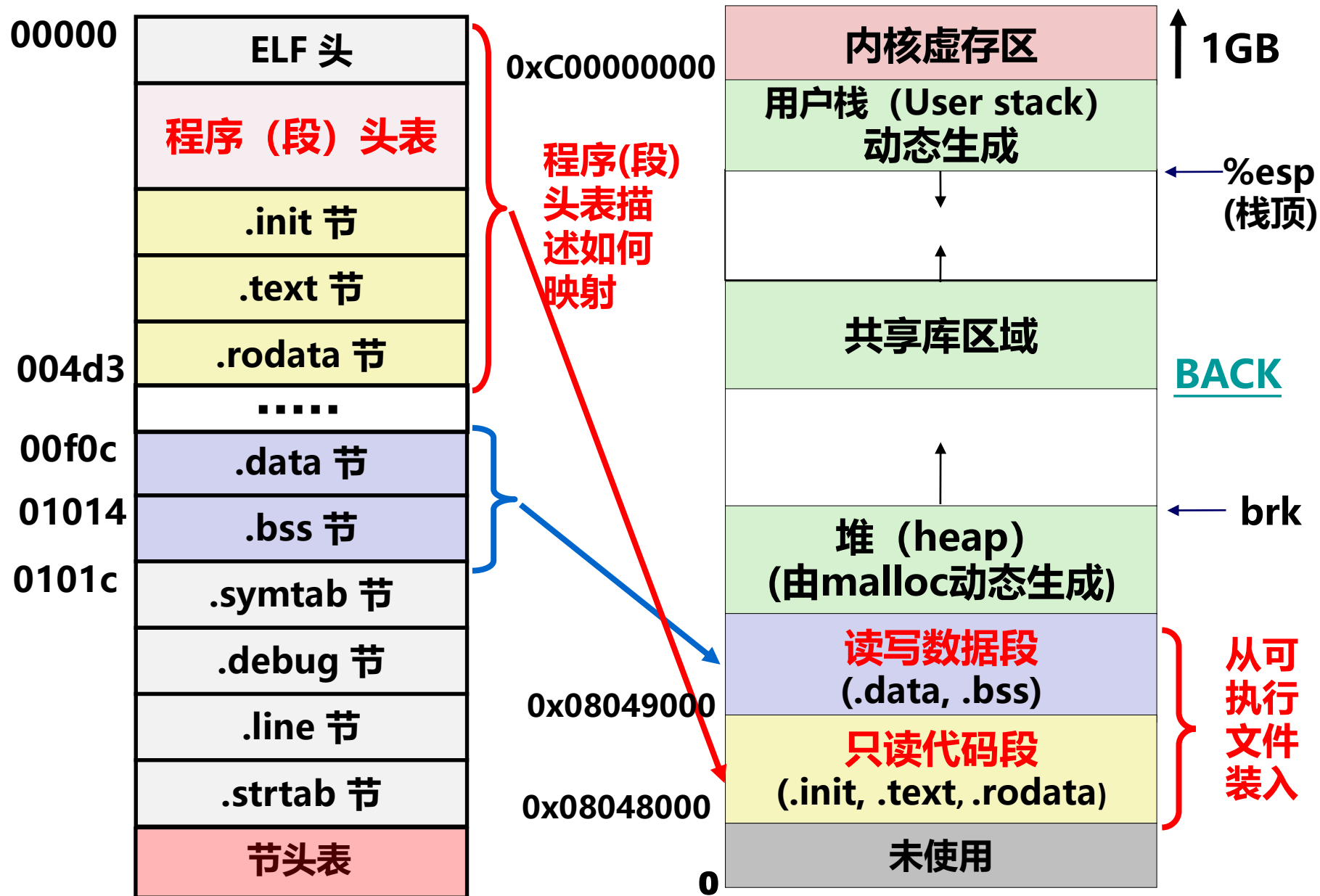
**SKIP**

**第一可装入段：**第0x00000~0x004d3字节（包括ELF头、程序头表、.init、.text和.rodata节），映射到虚拟地址0x8048000开始长度为0x4d4字节的区域，按0x1000=2<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字节的数据节，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。

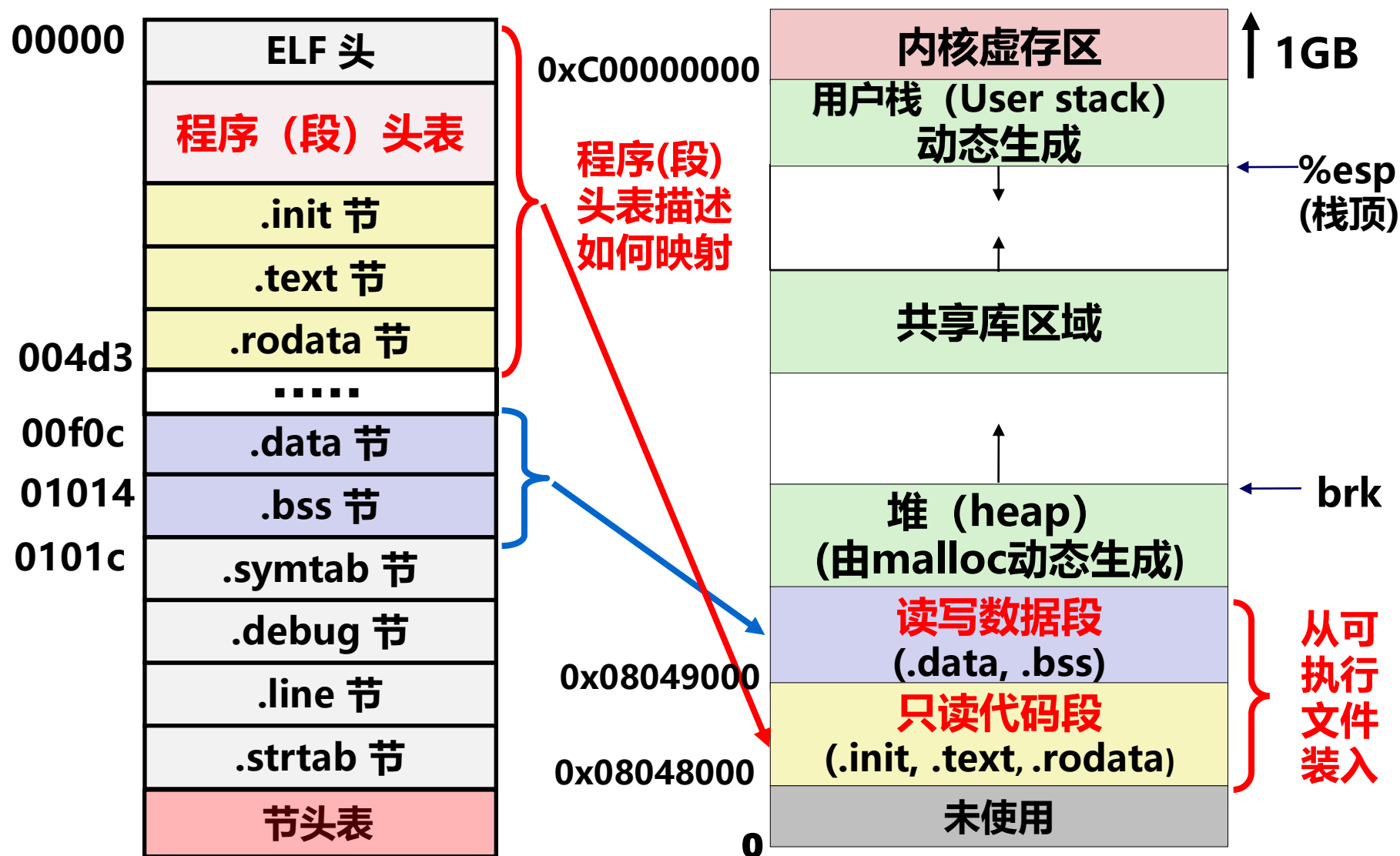


# 可执行文件的存储器映像



# 要求思考的问题

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



# 程序的链接

---

## 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



# 符号和符号解析

每个**可重定位目标模块m**都有一个符号表，它包含了在m中定义的符号。

有三种链接器符号：

- **Global symbols** (模块内部定义的**全局符号**)
  - 由模块m定义并能被其他模块引用的符号。例如，非static 函数和非static的全局变量（指不带static的全局变量）  
如，main.c 中的全局变量名buf
- **External symbols** (外部定义的**全局符号**)
  - 由其他模块定义并被模块m引用的全局符号  
如，main.c 中的函数名swap
- **Local symbols** (本模块的**局部符号**)
  - 仅由模块m定义和引用的本地符号。例如，在模块m中定义的带static的函数和全局变量  
如，swap.c 中的static变量名bufp1

链接器**局部符号**不是指程序中的**局部变量**（分配在栈中的临时性变量）

# 符号和符号解析

## main.c

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

## swap.c

```
extern int buf[];  
  
int *bufp0 = &buf[0];  
static int *bufp1;  
  
void swap()  
{  
    int temp;  
  
    bufp1 = &buf[1];  
    temp = *bufp0;  
    *bufp0 = *bufp1;  
    *bufp1 = temp;  
}
```

你能说出哪些是**全局符号**？哪些是**外部符号**？哪些是**局部符号**？

# 目标文件中的符号表

**.symtab** 节记录符号表信息，是一个结构数组

- 符号表 (**symtab**) 中每个条目的结构如下：

函数名在text节中

变量名在data节或  
bss节中

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

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

# 目标文件中的符号表

- main.o中的符号表中最后三个条目 (共10个)

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

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

main是第1节 (.text) 偏移为0的符号, 是全局函数, 占33B;

swap是main.o中未定义全局 (在其他模块定义) 符号, 类型和大小未知

- swap.o中的符号表中最后4个条目 (共11个)

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

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

# 符号解析 (Symbol Resolution)

- 目的：将每个模块中**引用的符号**与某个目标模块中的**定义符号**建立关联。
- 每个**定义符号**在代码段或数据段中都被分配了存储空间，将引用符号与定义符号建立关联后，就可在重定位时将引用符号的地址重定位为相关联的定义符号的地址。
- **局部（本地）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**（外部定义的、内部定义的）的解析涉及多个模块，故较复杂

add B  
jmp L0

.....  
L0: sub 23

.....  
B: .....

确定L0的地址，  
再在jmp指令中  
填入L0的地址

符号解析也称**符号绑定**

“符号的定义”其  
实质是什么？

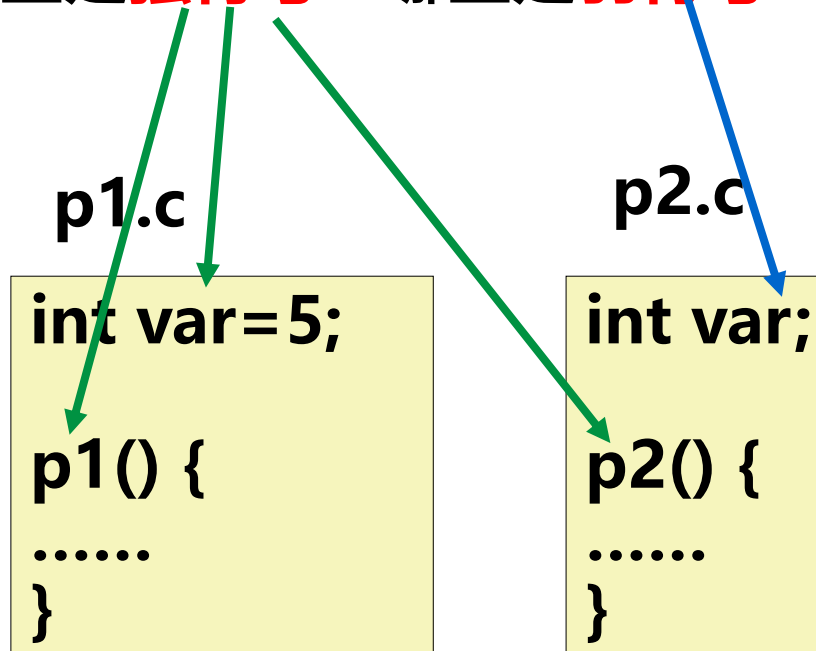
指被分配了存储空间。为函数名时，指代码所在区；为变量名时，指所占的静态数据区。

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

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

以下符号哪些是**强符号**？ 哪些是**弱符号**？



# 全局符号的符号解析

以下符号哪些是**强符号**? 哪些是**弱符号**?

? 外部符号没有强弱

main.c

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

此处为引用

?

弱符号只可能是变量名

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

局部变量

# 链接器对符号的解析规则

---

符号解析时只能有一个确定的定义（即每个符号仅占一处存储空间）

- **多重定义符号的处理规则**

**Rule 1: 强符号不能多次定义**

- 强符号只能被定义一次，否则链接错误

**Rule 2: 若一个符号被定义为一次强符号和多次弱符号，则按强定义为准**

- 对弱符号的引用被解析为其强定义符号

**Rule 3: 若有多个弱符号定义，则任选其中一个**

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



# 多重定义符号的解析举例

以下程序会发生链接出错吗？

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

main.c

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

p1.c

main只有一次强定义

p1有一次强定义，一次弱定义（但p1不是弱符号）

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？

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

main.c

p1.c

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

FLD1  
FSTPI &d

p1执行后d和x处内容是什么？

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

1.0: 0 01111111111 0...0B  
=3FF0 0000 0000 0000H

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！

# 多重定义全局符号的问题

---

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

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

大部分程序员并不了解链接器如何工作，因而养成良好的编程习惯是非常重要的。

# 如何划分模块？

---

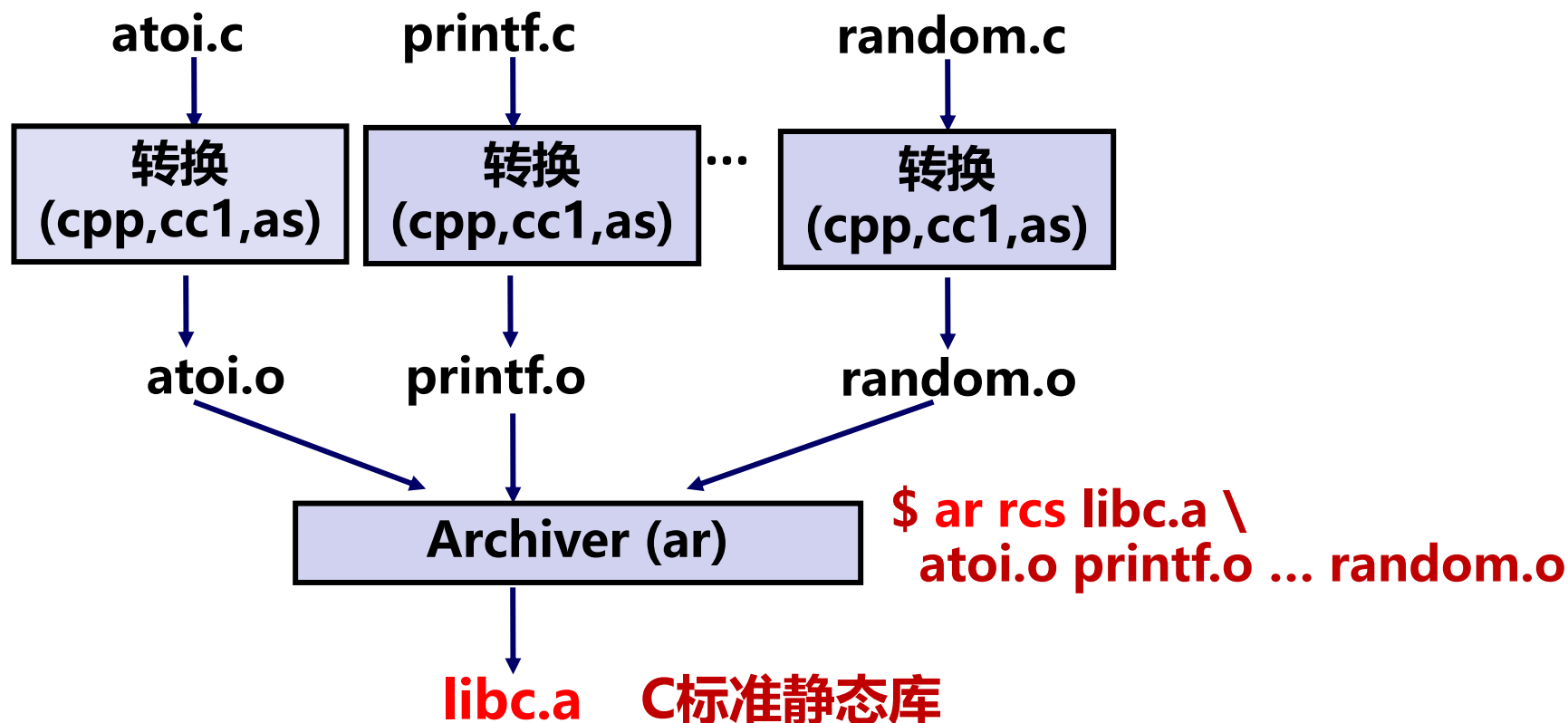
- 许多函数无需自己写，可使用共享库函数
  - 如数学库, 输入/输出库, 存储管理库, 字符串处理等
- 避免以下两种极端做法
  - 将所有函数都放在一个源文件中
    - 修改一个函数需要对所有函数重新编译
    - 时间和空间两方面的效率都不高
  - 一个源文件中仅包含一个函数
    - 需要程序员显式地进行链接
    - 效率高，但模块太多，故太繁琐

# 静态共享库

---

- **静态库 (.a archive files)**
  - 将所有相关的目标模块 (.o) 打包为一个单独的库文件 (.a) , 称为**静态库文件** , 也称**存档文件** (archive)
  - 增强了链接器功能, 使其能通过查找一个或多个库文件中的符号来解析符号
  - 在构建可执行文件时只需指定库文件名, 链接器会自动到库中寻找那些应用程序用到的目标模块, 并且**只把用到的模块从库中拷贝出来**
  - **在gcc命令中无需明显指定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/x86_64-linux-gnu/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_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**myfun1**加入**U**，**main**加入**D**。接着扫描到**mylib.a**，将**U**中所有符号（本例中为**myfunc1**）与**mylib.a**中所有目标模块（**myproc1.o**和**myproc2.o**）依次匹配，发现在**myproc1.o**中定义了**myfunc1**，故**myproc1.o**加入**E**，**myfunc1**从**U**转移到**D**。在**myproc1.o**中发现还有未解析符号**printf**，将其加到**U**。不断在**mylib.a**的各模块上进行迭代以匹配**U**中的符号，直到**U**、**D**都不再变化。此时**U**中只有一个未解析符号**printf**，而**D**中有**main**和**myfunc1**。因为模块**myproc2.o**没有被加入**E**中，因而它被丢弃。

**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

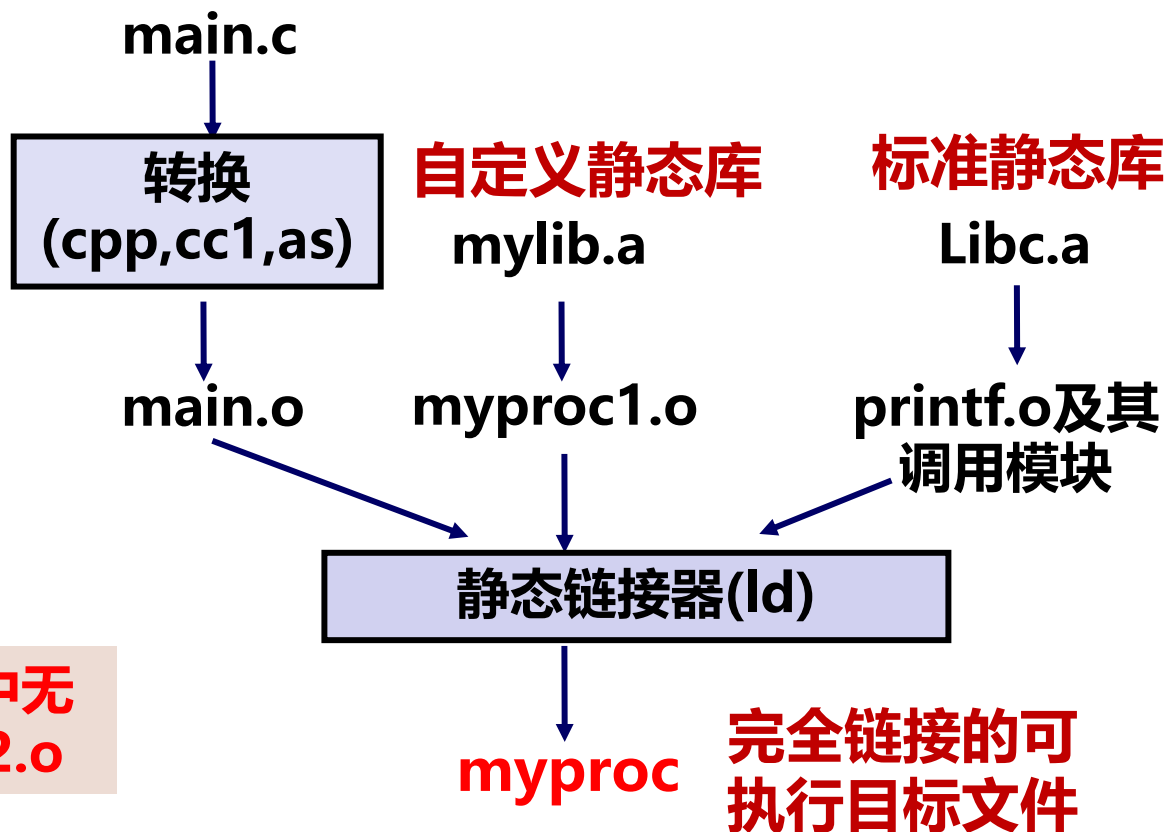
\$ ar rcs mylib.a myproc1.o myproc2.o

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

main→myfunc1→printf

main.c

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



注意：E中无  
myproc2.o

解析结果：

E中有main.o、myproc1.o、printf.o及其调用的模块

D中有main、myproc1、printf及其引用的符号

# 链接器中符号解析的全过程

main.c

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

main→myfunc1→printf

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

解析结果:

E中有main.o、myproc1.o、printf.o及其调用的模块

D中有main、myproc1、printf及其引用符号

被链接模块应按  
调用顺序指定!

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

首先, 扫描mylib, 因是静态库, 应根据其中是否存在U中未解析符号对应的定义符号来确定哪个.o被加入E。因为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  
\$ gcc -L. -lmine **libtest.o** libmine.a时, 用其定义的libfun来解析

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：

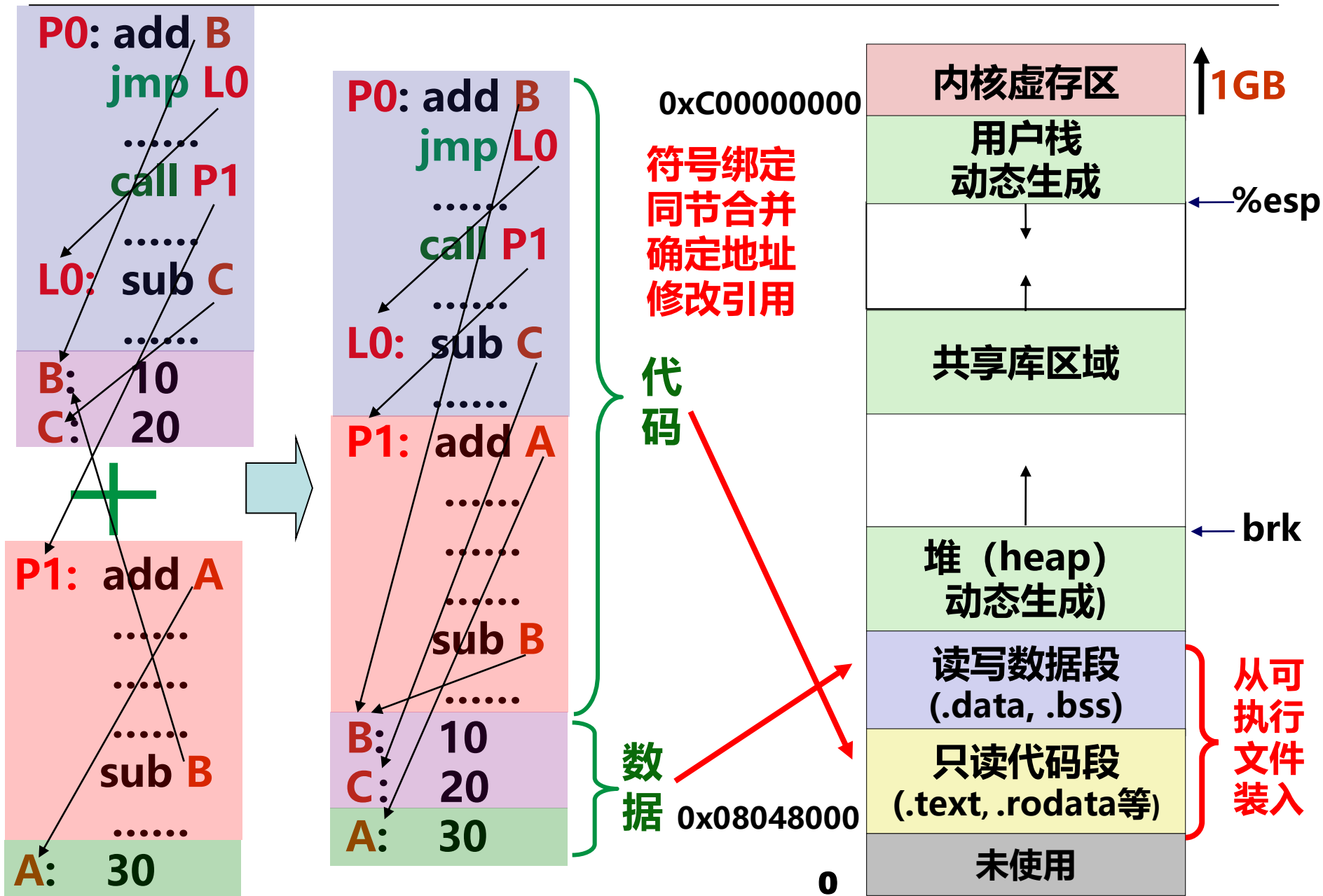
**func.o → libx.a 和 liby.a 中的函数**

**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.rel\_data和.rel\_text节中保存的重定位信息

# 重定位信息

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

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

- IA-32有两种最基本的重定位类型
  - R\_386\_32: 绝对地址
  - R\_386\_PC32: PC相对地址

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

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

例如，在rel\_text节中有重定位条目如下

offset: 0x1  
symbol: B  
type: R\_386\_32

offset: 0x6  
symbol: L0  
type: R\_386\_PC32

问题：重定位条目和汇编后的  
机器代码在何种目标文件中？

在可重定位目标  
(.o) 文件中！



# 重定位操作举例

## main.c

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

## swap.c

```
extern int buf[];  
int *bufp0 = &buf[0];  
static int *bufp1;  
void swap()  
{  
    int temp;  
    bufp1 = &buf[1];  
    temp = *bufp0;  
    *bufp0 = *bufp1;  
    *bufp1 = temp;  
}
```

你能说出哪些是**符号定义**？哪些是**符号的引用**？

局部变量temp分配在栈中，不会在过程外被引用，因此不是符号定义

# 重定位操作举例

## main.c

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

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

## swap.c

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

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

符号解析后的结果是什么？

E中有printf.o吗？

E中有main.o和swap.o两个模块！ D中有所有定义的符号！

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

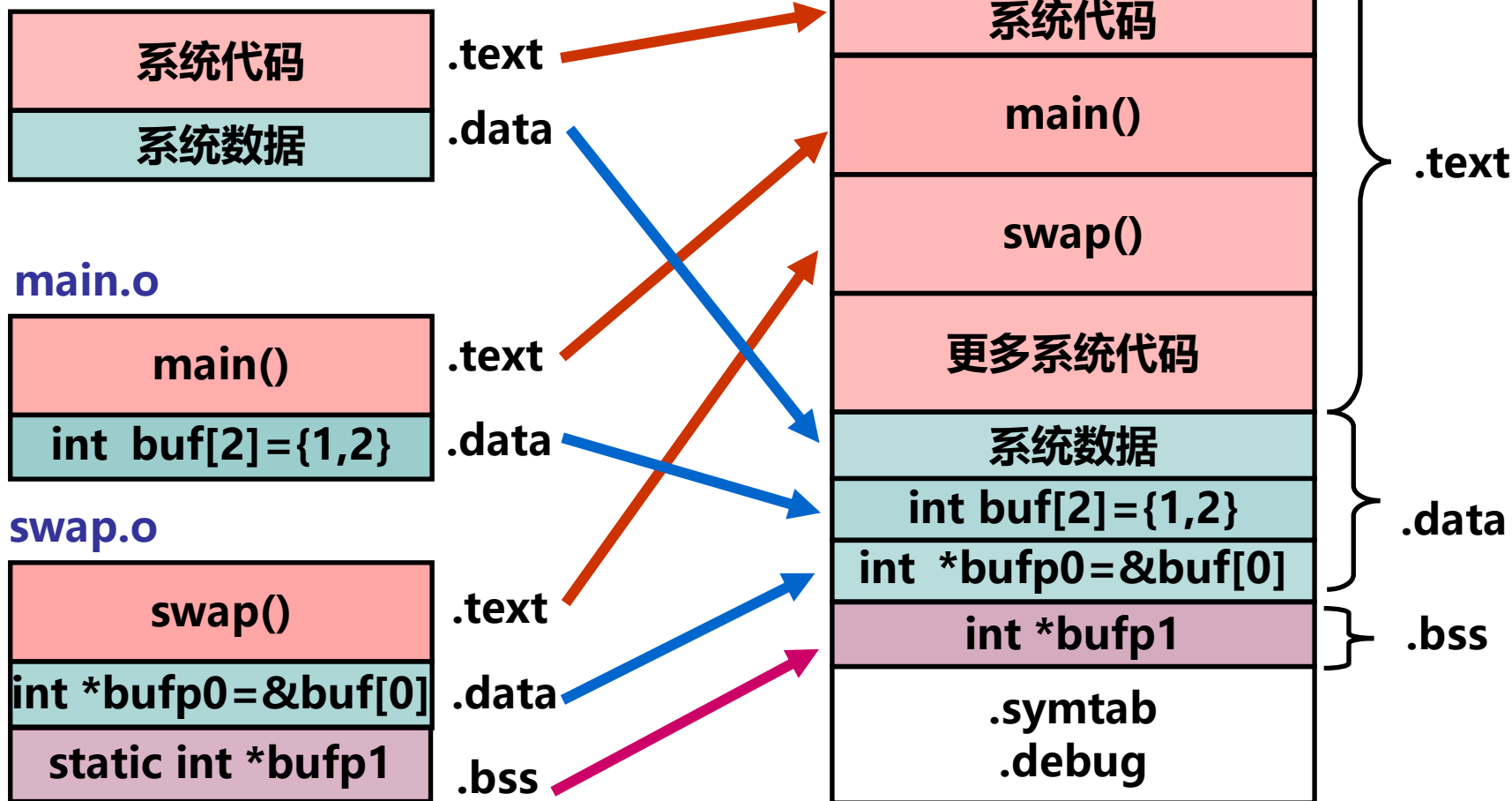
用命令**readelf -r main.o**可显示main.o中的重定位条目（表项）

# 符号引用的地址需要重定位

链接本质：合并相同的“节”

可执行目标文件

可重定位目标文件



虽然bufp1是swap的本地符号，也需在.bss节重定位

# main.o重定位前

## main.c

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

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

main的定义在.text节中偏移为0处开始, 占0x12B。

## 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 <b>fc ff ff ff</b>	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出来为 "7: R\_386\_PC32 swap"

r\_sym=10说明引用的是swap!

# main.o中的符号表

- main.o中的符号表中最后三个条目

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

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

在rel\_text节中的重定位条目为：

r\_offset=0x7, r\_sym=10,  
r\_type=R\_386\_PC32, dump出  
来后为 “7: R\_386\_PC32 swap”

r\_sym=10说明  
引用的是swap!

# R\_386\_PC32的重定位方式

- 假定:

- 可执行文件中main
- swap紧跟main后

- 则swap起始地址为:

- $0x8048380 + 0x12 = 0x8048392$
- 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:

$$\text{ADDR}(r \text{ sym}) - ( ( \text{ADDR}(.text) + r \text{ offset} ) - \text{init} )$$

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

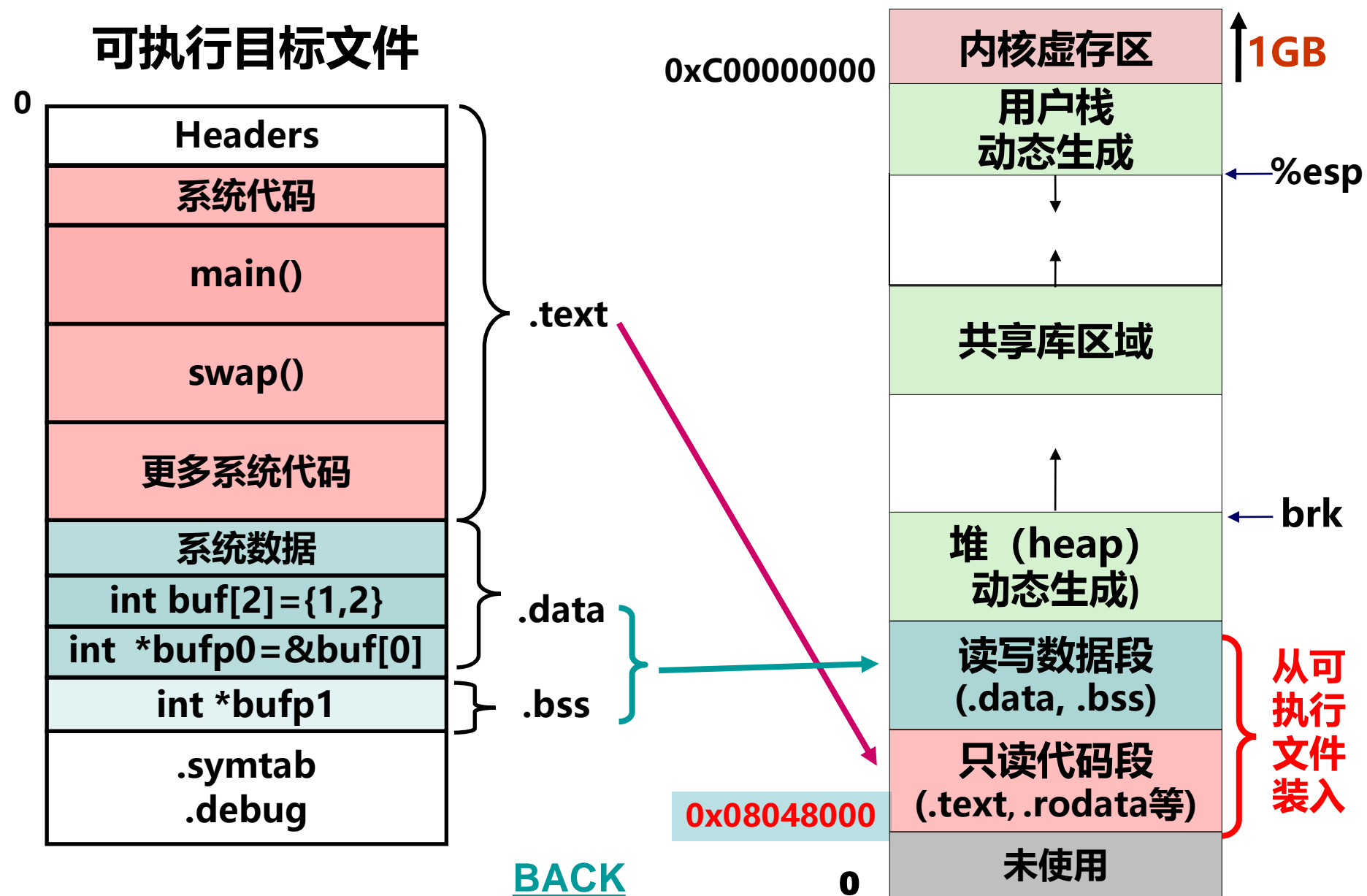
.....

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	Data	Global	0	3	bufp0
9:	0	0	Notype	Global	0	UND	buf
10:	0	36	Func	Global	0	1	swap
11:	4	4	Data	Local	0	COM	bufp1

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

重定位节.rel.data中有一个重定位表项：r\_offset=0x0, r\_sym=9, r\_type=R\_386\_32, OBJDUMP工具解释后显示为  
"0: R\_386\_32 buf"

r\_sym=9说明引用的是buf!

# R\_386\_32的重定位方式

- 假定：
  - 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:	c7 05 <u>00 00 00 00</u> 04	movl \$0x4,0x0
d:	<u>00 00 00</u>	
		8: R_386_32 .bss
		c: R_386_32 buf
10:	a1 <u>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:	a1 <u>00 00 00 00</u>	mov 0x0,%eax
1b:		R_386_32 bufp0
1f:	8b 15 <u>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:	a1 <u>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  $0xffffffff0,%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
- 2) M[R[esp]] ← R[eip]
- 3) 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:

- $0x8048380 + 0x12 = 0x8048392$
- 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:

$$\text{ADDR}(\text{r sym}) - ( (\text{ADDR}(\text{.text}) + \text{r offset}) - \text{init} )$$

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>

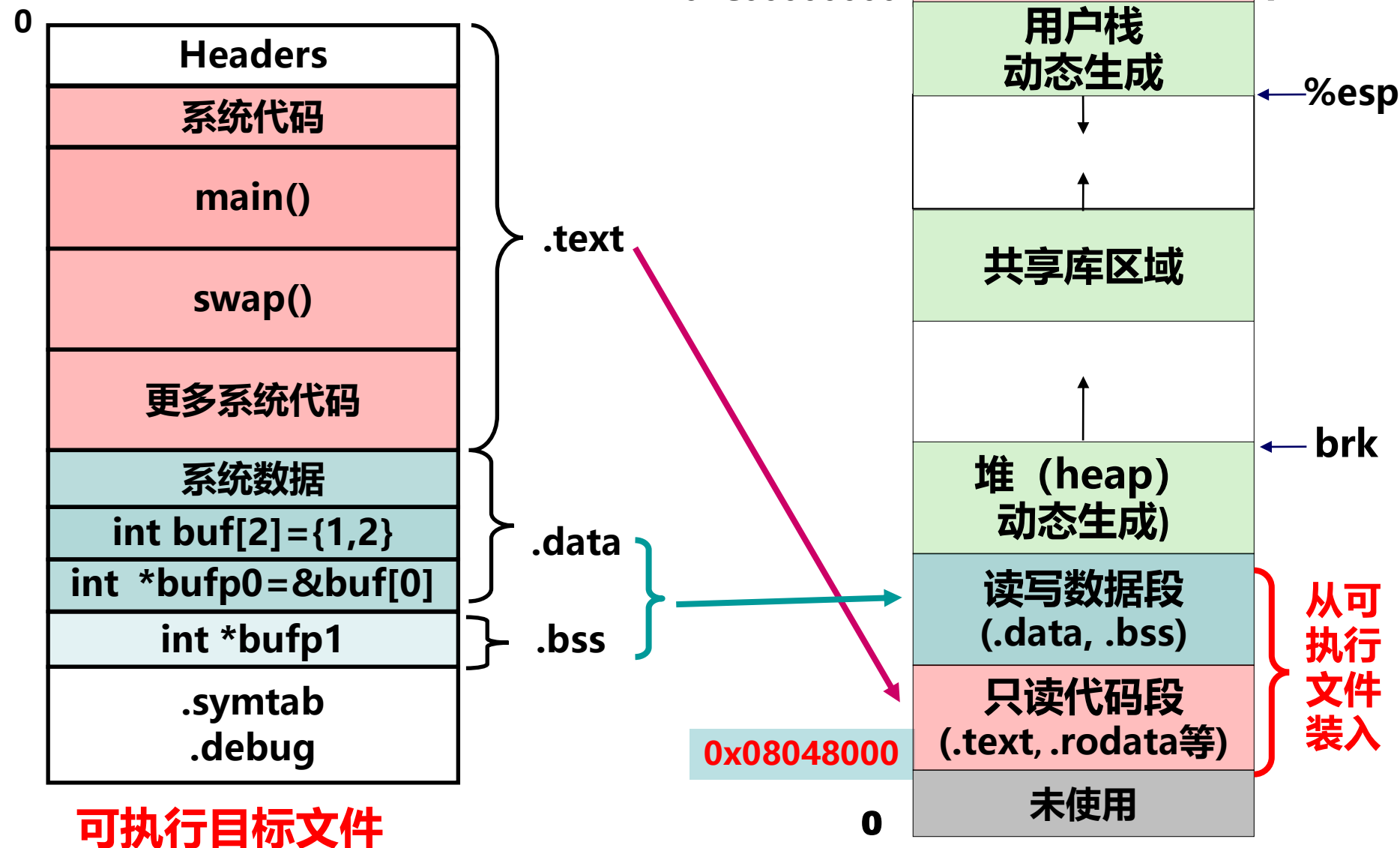
7: **R\_386\_PC32 swap**

.....

**重定位值**  
**值为-4**

# 可执行文件的存储器映像

程序(段)头表描述如何映射!



# 可执行文件的加载

- 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

\_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

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

# 程序的链接

---

## — 动态链接

- 动态链接的特性、程序加载时的动态链接、程序运行时的动态链接、动态链接举例

# 动态链接的共享库（Shared Libraries）

---

- 静态库有一些缺点：
  - 库函数（如printf）被包含在每个运行进程的代码段中，对于并发运行上百个进程的系统，造成极大的主存资源浪费
  - 库函数（如printf）被合并到可执行目标中，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
  - 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

**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**

# 加载时动态链接

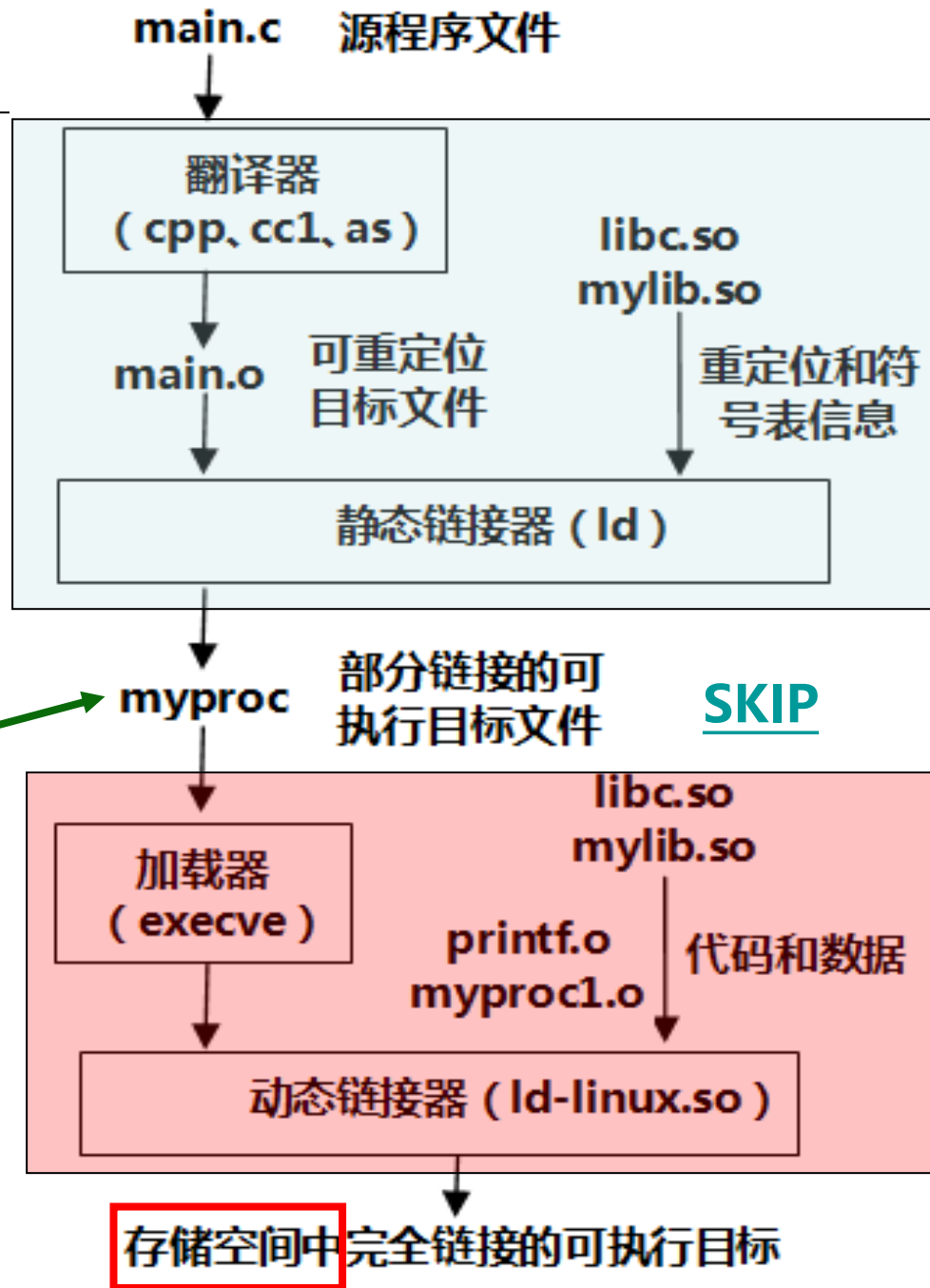
gcc -c main.c **libc.so**无需明显指出

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
- 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
  - 链接器无需修改代码即可将共享库加载到任意地址运行
- 所有引用情况
  - (1) 模块内的过程调用、跳转, 采用PC相对偏移寻址
  - (2) 模块内数据访问, 如模块内的全局变量和静态变量
  - (3) 模块外的过程调用、跳转
  - (4) 模块外的数据访问, 如外部变量的访问

要实现动态链接,  
必须生成PIC代码

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
- 不管.so中的代码加载到哪里，call指令中的偏移量不变

0000344 <bar>:

```
0000344: 55          pushl %ebp
0000345: 89 e5       movl %esp, %ebp
.....
0000352: c3         ret
0000353: 90         nop
```

0000354 <foo>:

```
0000354: 55          pushl %ebp
.....
0000364: e8 db ff ff ff  call 0000344 <bar>
0000369:
.....
```

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

```
void bar()
```

```
{
    a=1;
    b=2;
}
```

```
void foo()
```

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

call的目标地址为:

0x0000369+  
0xffffffffdb(-0x25)=  
0x0000344

JMP指令也可用相对寻址方式解决

## (2) 模块内部数据引用

- .data节与.text节之间的相对位置确定，任何引用局部符号的指令与该符号之间的距离是一个常数
- 不管代码加载到哪里，call指令中的偏移量不变

```
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

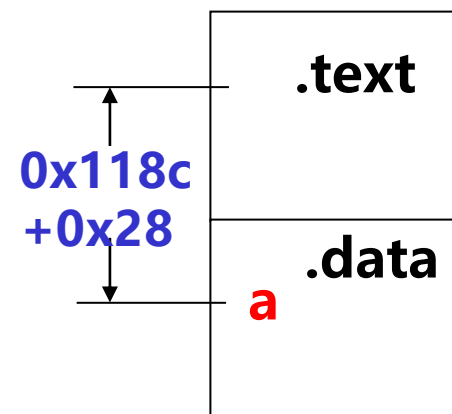
000039c <__get_pc>:
000039c: 8b 0c 24          movl (%esp), %ecx
000039f: c3               ret
```

```
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;  
}  
.....
```

0000344 <bar>:

0000344: 55

pushl %ebp

.....

0000357: e8 00 00 00 00

call 000035c

000035c: 5b

popl %ebx

000035d:

addl \$1180, %ebx

.....

movl (%ebx), %eax

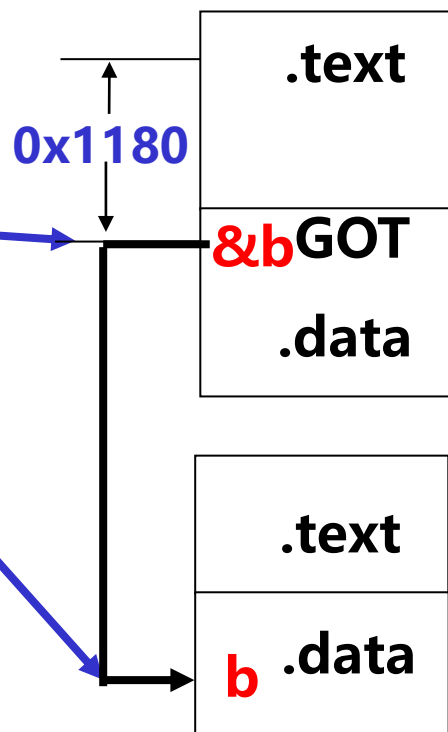
.....

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），并需多用一个寄存器（如EBX），易造成寄存器溢出



共享库模块

## (4) 模块间函数调用

- **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)

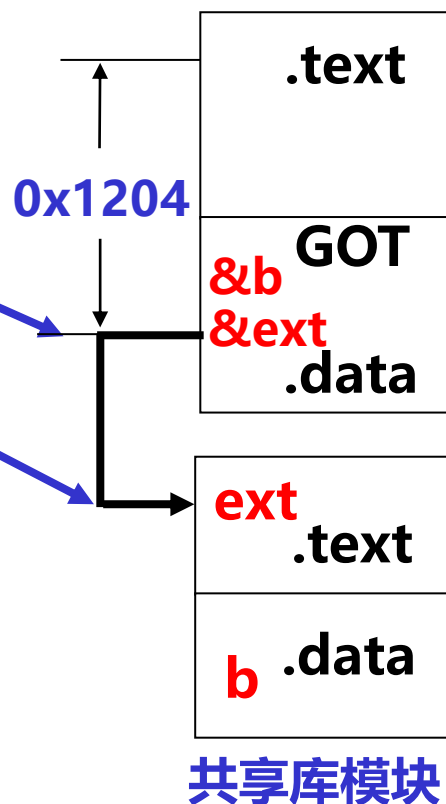
.....

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

- 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

```
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目标文件 **(PIC)** 。
- 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。

**介绍一本书：钱松林 赵海旭 《C++反汇编与逆向分析 技术揭秘》  
机械工业出版社**