

*7.8 在此题中, $REF(x, i) \rightarrow DEF(x, k)$ 表示链接器将任意对模块 i 中符号 x 的引用与模块 k 中符号 x 的定义相关联。在下面每个例子中, 用这种符号来说明链接器是如何解析在每个模块中有多重定义的引用的。如果出现链接时错误(规则 1), 写“错误”。如果链接器从定义中任意选择一个(规则 3), 那么写“未知”。

A.

```
/* Module 1 */
int main()
{
}

/* Module 2 */
static int main=1[
int p2()
{
}
```

(a) $REF(main.1) \rightarrow DEF(main, 1)$

(b) $REF(main.2) \rightarrow DEF(main, 2)$

B.

```
/* Module 1 */
int x;
void main()
{
}

/* Module 2 */
double x;
int p2()
{
}
```

(a) $REF(x.1) \rightarrow DEF(\text{未知})$

(b) $REF(x.2) \rightarrow DEF(\text{未知})$

C.

```
/* Module 1 */
int x=1;
void main()
{
}

/* Module 2 */
double x=1.0;
int p2()
{
}
```

(a) $REF(x.1) \rightarrow DEF(\text{错误})$

(b) $REF(x.2) \rightarrow DEF(\text{错误})$

**7.12 考虑目标文件 $m.o$ 中对函数 $swap$ 的调用(作业题 7.6)。

9: e8 00 00 00 00 callq e <main+0xe> swap()

具有如下重定位条目:

$r.offset = 0xa$
 $r.symbol = swap$
 $r.type = R_X86_64_PC32$
 $r.addend = -4$

A. 假设链接器将 $m.o$ 中的 $.text$ 重定位到地址 $0x4004e0$, 把 $swap$ 重定位到地址 $0x4004f8$ 。那么 $callq$ 指令中对 $swap$ 的重定位引用的值应该是什么?

B. 假设链接器将 $m.o$ 中的 $.text$ 重定位到地址 $0x4004d0$, 把 $swap$ 重定位到地址 $0x400500$ 。那么 $callq$ 指令中对 $swap$ 的重定位引用的值应该是什么?

A. $ADDR(s) = ADDR(.text) = 0x4004e0$

$ADDR(r.symbol) = ADDR(swap) = 0x4004f8$

$refaddr = ADDR(s) + r.offset = 0x4004e0 + 0xa = 0x4004ea$

$*retptr = (unsigned)(ADDR(r.symbol) + r.addend - refaddr)$
 $= (unsigned)(0x4004f8 + (-4) - 0x4004ea)$

$= 0xa$

B. $ADDR(s) = ADDR(.text) = 0x4004d0$

$ADDR(r.symbol) = ADDR(swap) = 0x400500$

$refaddr = ADDR(s) + r.offset = 0x4004d0 + 0xa = 0x4004da$

$*retptr = (unsigned)(ADDR(r.symbol) + r.addend - refaddr)$
 $= 0x22$

链接章节课后作业

一、课本练习题 7.8、7.12

二、判断题

/* 编译系统 */

1. (✓) c 语言的编译步骤依次是预处理、编译、汇编、链接。其中，预处理阶段主要完成的两件事情是头文件包含和宏展开。

2. (✓) 假设当前目录下已有可重定位模块 main.o 和 sum.o，为了链接得到可执行文件 prog，可以使用指令 ld -o prog main.o sum.o

/* 静态链接 */

3. (✗) 链接时，链接器会拷贝静态库(.a)中的所有模块(.o)。

4. (✓) 链接时，如果所有的输入文件都是.o 或.c 文件，那么任意交换输入文件的顺序都不会影响链接是否成功。

5. (✗) c 程序中的全局变量不会被编译器识别成局部符号。

/* 动态链接 */

6. (✓) 动态链接可以在加载时或者运行时完成，并且由于可执行文件中不包含动态链接库的函数代码，使得它比静态库更节省磁盘上的储存空间。

7. (✓) 动态库可以不编译成位置无关代码。

8. (✓) 通过代码段的全局偏移量表 GOT 和数据段的过程链接表 PLT，动态链接器可以完成延迟绑定 (lazy binding)。

/* 加载 */

9. (✓) _start 函数是程序的入口点。

10. (✗) ASLR 不会影响代码段和数据段间的相对偏移，这样位置无关代码才能正确使用。

/* static 和 extern 关键字 */

11. (✓) 函数内的被 static 修饰的变量将分配到静态存储区，其跨过程调用值仍然保持。

12. (✓) 变量声明默认不带 extern 属性，但函数原型声明默认带 extern 属性。

三、有下面两个程序。将它们先分别编译为.o 文件，再链接为可执行文件。

```
// m.c
#include <stdio.h>

void foo(int *);

int buf[2] = {1,2};
int main() {
    foo(buf);
    printf("%d %d", buf[0],buf[1]);
    return 0;
}
```

```
// foo.c
extern int buf[];
int *bufp0 = &buf[0];
int *bufp1;

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

Part A. 请填写 foo.o 模块的符号表。如果某个变量不在符号表中，那么在名字那一栏打×；如果它在符号表中的名字含有随机数字，那么请用不同的四位数字区分多个不同的符号。对于局部符号，不需要填强符号一栏。

变量名	符号表中的名字	局部符号?	强符号?	所在 section
buf	buf	X	X	UND
bufp0	bufp0	X	✓	.data
bufp1	bufp1	X	X	COM
temp	X			
count	count.1797	✓		.bss

Part B. 使用 gcc foo.c m.c 生成 a.out。其节头部表部分信息如下。已知符号表中 Size 列是十进制，且 Ndx 和 Nr 都是指节索引。请补充空缺的内容。

Section Headers:					
[Nr]	Name	Type	Address	Offset	Size
[1]	.interp	PROGBITS	00000000000002a8	000002a8	000000000000001c
[14]	.text	PROGBITS	0000000000001050	00001050	0000000000000205
[16]	.rodata	PROGBITS	0000000000002000	00002000	000000000000000a
[23]	.data	PROGBITS	0000000000004000	00003000	0000000000000020
[24]	.bss	NOBITS	0000000000004020	00003020	0000000000000010

Symbol Table:						
Num:	Value	Size	Type	Bind	Ndx	Name
35:	0000000000004024	4	OBJECT	LOCAL	24	count.1797
54:	0000000000004010	8	OBJECT	GLOBAL	23	bufp0
59:	000000000000115a	78	FUNC	GLOBAL	14	foo
62:	0000000000004018	8	OBJECT	GLOBAL	23	buf
64:	00000000000011a8	54	FUNC	GLOBAL	14	main
68:	0000000000004028	8	OBJECT	GLOBAL	24	bufp1
51:	0000000000000000	0	FUNC	GLOBAL	UND	printf@@GLIBC_2.2.5

Part C. 接 Part B 回答以下问题。

a) 读取 .interp 节，发现是一个可读字符串 /lib64/ld-linux-x86-64.so.2。

b) .bss 节存储时占用空间为 8 字节，运行时占用的空间为 16 字节。

Part D. 接 Part B, 通过 objdump -dx m.o 我们看到如下重定位信息。

```

0000000000000000 <main>:
 0: 55                                push %rbp
...
10: 8b 15 00 00 00 00    mov 0x0(%rip),%edx # 16 <main+0x16>
                        12: R_X86_64_PC32      buf
...
1e: 48 8d 3d 00 00 00 00 lea 0x0(%rip),%rdi # 25 <main+0x25>
                        21: R_X86_64_PC32      .rodata-0x4
...
2a: e8 00 00 00 00      callq 2f <main+0x2f>
                        2b: R_X86_64_PLT32      printf-0x4
...

```

假设链接器生成 a.out 时已经确定：m.o 的 .text 节在 a.out 中的起始地址

为 ADDR(.text)=0x11a8。请写出重定位后的对应于 main+0x10 位置的代码。

11b8: 8b 15 ~~2a 5e 00 00~~ mov 0x~~5e 2a~~(%rip),%edx

而 main+0x1e 处的指令变成:

11c6: 48 8d 3d 37 0e 00 00 lea 0xe37(%rip),%rdi

可见字符串 “%d %d” 在 a.out 中的起始地址是 0x 2021。

Part E. 使用 objdump -d a.out 可以看到如下 .plt 节的代码。

Disassembly of section .plt:

00000000000001020 <.plt>:

```
1020: ff 35 9a 2f 00 00    pushq  0x2f9a(%rip)
      # 3fc0 <_GLOBAL_OFFSET_TABLE_+0x8>
1026: ff 25 9c 2f 00 00    jmpq   *0x2f9c(%rip)
      # 3fc8 <_GLOBAL_OFFSET_TABLE_+0x10>
102c: 0f 1f 40 00          nopl   0x0(%rax)
```

00000000000001030 <printf@plt>:

```
1030: ff 25 9a 2f 00 00    jmpq   *0x2f9a(%rip)
      # 3fd0 <printf@GLIBC_2.2.5>
1036: 68 00 00 00 00      pushq  $0x0
103b: e9 e0 ff ff ff      jmpq   1020 <.plt>
```

a) 完成 main+0x2a 处的重定位。

11d2: e8 ~~59 fe # #~~ callq <printf@plt>

b) printf 的 PLT 表条目是 PLT[1], GOT 表条目是 GOT[3] (填写数字)。

c) 使用 gdb 对 a.out 进行调试。某次运行时 main 的起始地址为 0x555555551a8, 那么当加载器载入内存而尚未重定位 printf 地址前, printf 的 GOT 表项的内容是 0x 55555555036。