A. p.o→libx.a→p.o

B. p.o→libx.a→liby.a 和 liby.a→libx.a

C. p.o→libx.a→liby.a→libz.a 和 liby.a→libx.a→libz.a

**7.11 图 7-14 中的程序头部表明数据段占用了内存中 0x230 个字节。然而，其中只有开始的 0x228 字节来自可执行文件的节。是什么引起了这种差异？

**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？

**7.13 完成下面的任务将帮助你更熟悉处理目标文件的各种工具。

A. 在你的系统上，lib.c 和 libm.a 的版本中包含多少目标文件？

B. gcc -Og 产生的可执行代码与 gcc -Og -g 产生的不同吗？

C. 在你的系统上，GCC 驱动程序使用的是什么共享库？

练习题答案

7.1 这道练习题的目的是帮助你理解链接器符号和 C 变量及函数之间的关系。注意 C 的局部变量 temp 没有符号表条目。

符号	.symtab条目?	符号类型	在哪个模块中定义	节
buf	是	外部	main.o	.data
bufp0	是	全局	swap.o	.data
bufp1	是	全局	swap.o	COMMON
swap	是	全局	swap.o	.text
temp	否	—	—	—

7.2 这是一个简单的练习，检查你对 Unix 链接器解析在一个以上模块中有定义的全局符号时所使用规则的理解。理解这些规则可以帮助你避免一些讨厌的编程错误。

A. 链接器选择定义在模块 1 中的强符号，而不是定义在模块 2 中的弱符号(规则 2)：

(a) REF(main.1) → DEF(main.1)

(b) REF(main.2) → DEF(main.1)

B. 这是一个错误，因为每个模块都定义了一个强符号 main(规则 1)。

C. 链接器选择定义在模块 2 中的强符号，而不是定义在模块 1 中的弱符号(规则 2)：

(a) REF(x.1) → DEF(x.2)

(b) REF(x.2) → DEF(x.2)

7.3 在命令行中以错误的顺序放置静态库是造成令许多程序员迷惑的链接器错误的常见原因。然而，一旦你理解了链接器是如何使用静态库来解析引用的，它就相当简单易懂了。这个小练习检查了你对这个概念的理解：

A. linux> gcc p.o libx.a

B. linux> gcc p.o libx.a liby.a

C. linux> gcc p.o libx.a liby.a libx.a

7.4 这道题涉及的是图 7-12a 中的反汇编列表。目的是让你练习阅读反汇编列表，并检查你对 PC 相对