7.12

原代码：

code/link/main.c

```
1  /* main.c */
2  void swap();
3
4  int buf[2] = {1, 2};
5
6  int main()
7  {
8      swap();
9      return 0;
10 }
```

code/link/main.c

a) main.c

code/link/swap.c

```
1  /* swap.c */
2  extern int buf[];
3
4  int *bufp0 = &buf[0];
5  static int *bufp1;
6
7  void swap()
8  {
9      int temp;
10
11     bufp1 = &buf[1];
12     temp = *bufp0;
13     *bufp0 = *bufp1;
14     *bufp1 = temp;
15 }
```

code/link/swap.c

b) swap.c

图 7-1 示例程序 1：这个示例程序由两个源文件组成，main.c 和 swap.c。main 函数初始化一个两元素的整数数组，然后调用 swap 函数来交换这一对数

可执行目标文件（已经被重定位过的）：

在得到的可执行目标文件中，引用有下面的重定位形式：

```
0804945c <bufp0>:
 804945c:  54 94 04 08                    Relocated!
```

总而言之，链接器决定在运行时变量 bufp0 将放置在存储器地址 0x804945c 处，并且被初始化为 0x8049454，这个值就是 buf 数组的运行时地址。

swap.o 模块中的 .text 节包含 5 个绝对引用，都以相似的方式进行重定位（参考练习题 7.12）。图 7-10 展示了最终的可执行目标文件中被重定位的 .text 和 .data 节。

code/link/p-exe.d

```
1  080483b4 <main>:
2   80483b4:  55                    push   %ebp
3   80483b5:  89 e5                 mov    %esp,%ebp
4   80483b7:  83 ec 08              sub    $0x8,%esp
5   80483ba:  e8 09 00 00 00        call   80483c8 <swap>          swap();
6   80483bf:  31 c0                 xor    %eax,%eax
7   80483c1:  89 ec                 mov    %ebp,%esp
8   80483c3:  5d                    pop    %ebp
9   80483c4:  c3                    ret
10  80483c5:  90                    nop
11  80483c6:  90                    nop
12  80483c7:  90                    nop

13  080483c8 <swap>:
14  80483c8:  55                    push   %ebp
15  80483c9:  8b 15 5c 94 04 08     mov    0x804945c,%edx          Get *bufp0
16  80483cf:  a1 58 94 04 08        mov    0x8049458,%eax          Get buf[1]
17  80483d4:  89 e5                 mov    %esp,%ebp
18  80483d6:  c7 05 48 95 04 08 58  movl   $0x8049458,0x8049548   bufp1 = &buf[1]
19  80483dd:  94 04 08
20  80483e0:  89 ec                 mov    %ebp,%esp
21  80483e2:  8b 0a                 mov    (%edx),%ecx
22  80483e4:  89 02                 mov    %eax,(%edx)
23  80483e6:  a1 48 95 04 08        mov    0x8049548,%eax          Get *bufp1
24  80483eb:  89 08                 mov    %ecx,(%eax)
25  80483ed:  5d                    pop    %ebp
26  80483ee:  c3                    ret
```

code/link/p-exe.d

a) 已重定位的 .text 节

code/link/pdata-exe.d

```
1  08049454 <buf>:
2   8049454:  01 00 00 00 02 00 00 00

3  0804945c <bufp0>:
4   804945c:  54 94 04 08                    Relocated!
```

code/link/pdata-exe.d

b) 已重定位的 .data 节

图 7-10 可执行文件 p 的已重定位的 .text 和 .data 节。原始的 C 代码在图 7-1 中

.o 文件原始代码（重定位前）：

*7.11 图 7-12 中的段头部表明数据段占用了存储器中 0x104 个字节。然而，只有开始的 0xe8 字节来自可执行文件的节。引起这种差异的原因是什么？

**7.12 图 7-10 中的 swap 程序包含 5 个重定位的引用。对于每个重定位的引用，给出它在图 7-10 中的行号、运行时存储器地址和值。swap.o 模块中的原始代码和重定位条目如图 7-19 所示。

```
1  00000000 <swap>:
2     0:  55                      push   %ebp
3     1:  8b 15 00 00 00 00       mov    0x0,%edx        Get *bufp0=&buf[0]
4                                 3: R_386_32   bufp0   Relocation entry
5     7:  a1 04 00 00 00          mov    0x4,%eax        Get buf[1]
6                                 8: R_386_32   buf     Relocation entry
7     c:  89 e5                   mov    %esp,%ebp
8     e:  c7 05 00 00 00 00 04    movl   $0x4,0x0        bufp1 = &buf[1];
9    15:  00 00 00
10                                10: R_386_32  bufp1   Relocation entry
11                                14: R_386_32  buf     Relocation entry
12    18:  89 ec                  mov    %ebp,%esp
13    1a:  8b 0a                  mov    (%edx),%ecx     temp = buf[0];
14    1c:  89 02                  mov    %eax,(%edx)     buf[0]=buf[1];
15    1e:  a1 00 00 00 00         mov    0x0,%eax        Get *bufp1=&buf[1]
16                                1f: R_386_32  bufp1   Relocation entry
17    23:  89 08                  mov    %ecx,(%eax)     buf[1]=temp;
18    25:  5d                     pop    %ebp
19    26:  c3                     ret
```

图 7-19 练习题 7.12 的代码和重定位条目

图 7-10 中的行号	地址	值
15	0x80483cb	0x80495ec
16	0x80483d0	0x8049448
18	0x80483d8	0x8049548
18	0x80483dc	0x8049458

7.13 考虑图 7-20 中的 C 代码和相应的可重定位目标模块。

7. 13

***7.13 考虑图 7-20 中的 C 代码和相应的可重定位目标模块。

A. 确定当模块被重定位时，链接器将修改 .text 中的哪些指令。对于每条这样的指令，列出它的重定位条目中的信息：节偏移、重定位类型和符号名字。

B. 确定当模块被重定位时，链接器将修改 .data 中的哪些数据目标。对于每条这样的指令，列出它的重定位条目中的信息：节偏移、重定位类型和符号名字。

可以随意使用诸如 OBJDUMP 之类的工具来帮助你解答这个题目。

```
1  extern int p3(void);
2  int x = 1;
3  int *xp = &x;
4
5  void p2(int y) {
6  }
7
8  void p1() {
9      p2(*xp + p3());
10 }
```

a) C 代码

```
1  00000000 <p2>:
2     0:  55                      push   %ebp
3     1:  89 e5                   mov    %esp,%ebp
4     3:  89 ec                   mov    %ebp,%esp
5     5:  5d                      pop    %ebp
6     6:  c3                      ret

7  00000008 <p1>:
8     8:  55                      push   %ebp
9     9:  89 e5                   mov    %esp,%ebp
10    b:  83 ec 08                sub    $0x8,%esp
11    e:  83 c4 f4                add    $0xfffffff4,%esp
12   11:  e8 fc ff ff ff          call   12 <p1+0xa>
13   16:  89 c2                   mov    %eax,%edx
14   18:  a1 00 00 00 00          mov    0x0,%eax
15   1d:  03 10                   add    (%eax),%edx
16   1f:  52                      push   %edx
17   20:  e8 fc ff ff ff          call   21 <p1+0x19>
18   25:  89 ec                   mov    %ebp,%esp
19   27:  5d                      pop    %ebp
20   28:  c3                      ret
```

b) 可重定位目标文件的 .text 节

```
1  00000000 <x>:
2     0:  01 00 00 00
3  00000004 <xp>:
4     4:  00 00 00 00
```

c) 可重定位目标文件的 .data 节

图 7-20 练习题 7.13 的示例代码

将代码保存为 t2.c 文件，使用 gcc -c t2.c 生成可重定位的目标文件，使

用 objdump -D t2.o > m.txt 反汇编并保存在 m.txt 内。查看.text 和.data。

```
Disassembly of section .text:

00000000 <p2>:
   0:   55                      push   %ebp
   1:   89 e5                   mov    %esp,%ebp
   3:   5d                      pop    %ebp
   4:   c3                      ret

00000005 <p1>:
   5:   55                      push   %ebp
   6:   89 e5                   mov    %esp,%ebp
   8:   53                      push   %ebx
   9:   83 ec 14                sub    $0x14,%esp
   c:   a1 00 00 00 00          mov    0x0,%eax
  11:   8b 18                   mov    (%eax),%ebx
  13:   e8 fc ff ff ff          call   14 <p1+0xf>
  18:   01 d8                   add    %ebx,%eax
  1a:   89 04 24                mov    %eax,(%esp)
  1d:   e8 fc ff ff ff          call   1e <p1+0x19>
  22:   83 c4 14                add    $0x14,%esp
  25:   5b                      pop    %ebx
  26:   5d                      pop    %ebp
  27:   c3                      ret

Disassembly of section .data:

00000000 <x>:
   0:   01 00                   add    %eax,(%eax)
        ...

00000004 <xp>:
   4:   00 00                   add    %al,(%eax)
        ...
```

使用 readelf -a t2.o 工具：

```
There are no section groups in this file.

There are no program headers in this file.

Relocation section '.rel.text' at offset 0x434 contains 3 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
0000000d  00000901 R_386_32          00000004   xp
00000014  00000c02 R_386_PC32        00000000   p3
0000001e  00000a02 R_386_PC32        00000000   p2

Relocation section '.rel.data' at offset 0x44c contains 1 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
00000004  00000801 R_386_32          00000000   x

Relocation section '.rel.eh_frame' at offset 0x454 contains 2 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
00000020  00000202 R_386_PC32        00000000   .text
00000040  00000202 R_386_PC32        00000000   .text

There are no unwind sections in this file.
```

由该工具可知：

A：

节偏移	重定位类型	符号名字
0x0d	绝对重定位	xp
0x14	相对重定位	p3
0x1e	相对重定位	p2

B:

节偏移	重定位类型	符号名字
0x04	绝对重定位	x

7.15

****7.15** 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，libc.a 和 libm.a 的版本中包含多少目标文件？

B. gcc-O2 产生的可执行代码与 gcc -O2 -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？

A:

首先找到两个文件的路径（使用 sudo 权限），然后进入该路径，使用 ar 命令列出库中所有目标文件，并使用管道传递给 wc 命令计算行数，得到目标文件数量。

1529 和 407。

```
guo@ubuntu:~$ sudo find / -name libc.a
/usr/lib/i386-linux-gnu/libc.a
/usr/lib/i386-linux-gnu/xen/libc.a
guo@ubuntu:~$ cd /usr/lib/i386-linux-gnu/
guo@ubuntu:/usr/lib/i386-linux-gnu$ ar t libc.a | wc -l
1529
guo@ubuntu:/usr/lib/i386-linux-gnu$ sudo find / -name libm.a
/usr/lib/i386-linux-gnu/xen/libm.a
/usr/lib/i386-linux-gnu/libm.a
guo@ubuntu:/usr/lib/i386-linux-gnu$ ar t libm.a | wc -l
407
guo@ubuntu:/usr/lib/i386-linux-gnu$
```

B:

-g：是一个调试标志。它告诉 GCC 在生成的可执行文件中**包含调试信息**。这些调试信息允许调试器（如 GDB）知道源代码中的变量名、函数名、行号等信息，从而可以更方便地调试程序。但是，由于调试信息需要额外的空间来存储，因此包含调试信息的可执行文件通常会比不包含调试信息的文件大。

C:

GCC 通常默认链接 C 共享库（libc.so），我们可以使用 ldd 命令检查生成的可执行文件依赖的共享库。这里使用 ldd hello 命令，hello 是一个简单的 C 程序，作用是使用 printf 输出一行 hello world，ldd 命令可以查看依赖的共享库，它的输出通常包含多行，每行对应一个共享库，格式为，库名称=>库路径。

```
guo@ubuntu:~/code$ ./hello
hello world!
guo@ubuntu:~/code$ ldd hello
        linux-gate.so.1 =>  (0xb77b5000)
        libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xb75f6000)
        /lib/ld-linux.so.2 (0xb77b6000)
```

这里输出了三行，第一行 linux-gate.so.1 是一个内核提供的接口，用于实现系统调用。第二行是我们想要知道的，libc.so.6，C 标准库的共享库。第三行 /lib/ld-linux.so.2 是动态链接器（dynamic linker）或称为解释器（interpreter）。当可执行文件启动时，它实际上是由这个动态链接器来加载的。动态链接器负责解析可执行文件依赖的共享库，并将它们加载到内存中，以便可执行文件能够运行。/lib/ld-linux.so.2 并不是 hello 可执行文件的直接

依赖，而是它用于加载其他共享库的工具。但是，由于 ldd 需要知道如何找到和加载其他共享库，所以它也会列出动态链接器本身。

我的 GCC 默认链接了 C 共享库（libc.so）。

8.19

**8.19 下面的函数打印多少行输出？给出一个答案为 n 的函数。假设 $n \geq 1$。

code/ecf/forkprob8.c

```
1  void foo(int n)
2  {
3      int i;
4
5      for (i = 0; i < n; i++)
6          Fork();
7      printf("hello\n");
8      exit(0);
9  }
```

code/ecf/forkprob8.c

2^n。每次循环都会生成一个子进程，打印一次。

8.21

**8.21 下面程序的可能的输出序列是什么？

code/ecf/waitprob3.c

```
1  int main()
2  {
3      if (fork() == 0) {
4          printf("a");
5          exit(0);
6      }
7      else {
8          printf("b");
9          waitpid(-1, NULL, 0);
10     }
11     printf("c");
12     exit(0);
13 }
```

code/ecf/waitprob3.c

在父进程中，先打印 b，然后使用 waitpid，等待子进程终止，在终止后打印 c；在子进程中，先打印 a，然后退出。

子进程和父进程都有可能先执行，可能输出 abc 或 bac

8.23

**8.23 你的一个同事想要使用信号来让一个父进程对发生在一个子进程中的事件计数。其思想是每次发生一个事件时，通过向父进程发送一个信号来通知它，并且让父进程的信号处理程序对一个全局变量 counter 加一，在子进程终止之后，父进程就可以检查这个变量。然而，当他在系统上运行图 8-41 中的测试程序时，发现当父进程调用 printf 时，counter 的值总是 2，即使子进程向父进程发送了 5 个信号。他很困惑，向你寻求帮助。你能解释这个程序有什么错误吗？

这个现象的出现与信号处理有关。对于信号处理，有一些基本知识：1.待处理信号会被阻塞，进程捕获了一个 SIGHT 信号正在处理，此时又有一个 SIGHT 信号，它会变成待处理信号，被阻塞。2.任意类型至多只有一个待处理信号，第二个信号会被简单丢弃，不会排队等待。

这道题中，父进程接收到了第一个 SIGUSR2 信号，进入信号处理例程。剩余 4 个该类信号 SIGUSR2 被阻塞，其中 1 个被视为待处理信号而被保存，其余的 3 个被丢弃。因此最终只有 2 个被处理，counter 最终结果为 2。