习题

1. 给出以下概念的解释说明。

链接	可重定位目标文件	可执行目标文件	符号解析
重定位	ELF 目标文件格式	ELF 头	节头表
程序头表（段头表）	只读代码段	可读写数据段	全局符号
外部符号	本地符号	强符号	弱符号
多重定义符号	静态库	符号的定义	符号的引用
未解析符号	重定位信息	运行时堆	用户栈
动态链接	共享库（目标）文件		

2. 简单回答下列问题。

（1）如何将多个 C 语言源程序模块组合起来生成一个可执行目标文件？简述从源程序到可执行机器代码的转换过程？

（2）引入链接的好处是什么？

（3）可重定位目标文件和可执行目标文件的主要差别是什么？

（4）静态链接方式下，静态链接器主要完成哪两方面的工作？

（5）可重定位目标文件的.text 节、.rodata 节、.data 节和.bss 节中分别主要包含什么信息？

（6）可执行目标文件中的.text 节、.rodata 节、.data 节和.bss 节中分别主要包含什么信息？

（7）可执行目标文件中有哪两种可装入段？哪些节组合成只读代码段？哪些节组合成可读写数据段？

（8）加载可执行目标文件时，加载器根据其中的哪个表的信息对可装入段进行映射？

（9）在可执行目标文件中，可装入段被映射到虚拟存储空间，这种做法有什么好处？

（10）静态链接和动态链接的差别是什么？

3. 假设一个 C 语言程序有两个源文件：main.c 和 test.c，它们的内容如图 4.23 所示。

```
1   /* main.c */
2   int sum();
3
4   int a[4]={1, 2, 3, 4};
5   extern int val;
6   int main( )
7   {
8       val=sum();
9       return val;
10  }
```

```
1   /* test.c */
2   extern int a[];
3   int val=0;
4   int sum()
5   {
6       int i;
7       for (i=0; i<4; i++)
8           val += a[i];
9       return val;
10  }
```

图 4.23 题 3 用图

对于编译生成的可重定位目标文件 test.o，填写下表中各符号的情况，说明每个符号是否出现在 test.o 的符号表（.symtab 节）中，如果是的话，定义该符号的模块是 main.o 还是 test.o、该符号的类型是全局、外部还是本地符号、该符号出现在 test.o 中的哪个节（.text、.data 或.bss）。

符号	在 test.o 的符号表中？	定义模块	符号类型	节
a				
val				
sum				
i				

参考答案：

符号	在 test.o 的符号表中？	定义模块	符号类型	节
a	在	main.o	extern	.data
val	在	test.o	global	.data
sum	在	test.o	global	.text
i	不在	--	--	--

4. 假设一个 C 语言程序有两个源文件：main.c 和 swap.c，其中，main.c 的内容如图 4-7a 所示，而 swap.c 的内容如下：

```
1   extern int buf[];
2   int *bufp0 = &buf[0];
3   static int *bufp1;
4
5   static void incr() {
6       static int count=0;
7       count++;
8   }
8
9   void swap() {
10      int temp;
11      incr();
12      bufp1=&bufp[1];
13      temp=*bufp0;
14      *bufp0=*bufp1;
15      *bufp1=temp;
16  }
```

对于编译生成的可重定位目标文件 swap.o，填写下表中各符号的情况，说明每个符号是否出现在 swap.o 的符号表（.symtab 节）中，如果是的话，定义该符号的模块是 main.o 还是 swap.o、该符号的类型是全局、外部还是本地符号、该符号出现在 swap.o 中的哪个节（.text、.data 或.bss）。

符号	在 swap.o 的符号表中？	定义模块	符号类型	节
buf				
bufp0				
bufp1				
incr				
count				
swap				
temp				

参考答案：

符号	在 swap.o 的符号表中？	定义模块	符号类型	节
buf	在	main.o	extern	.data
bufp0	在	swap.o	global	.data
bufp1	在	swap.o	local	.bss
incr	在	swap.o	local	.text
count	在	swap.o	local	.data
swap	在	swap.o	global	.text
temp	不在	--	--	--

5. 假设一个 C 语言程序有两个源文件：main.c 和 proc1.c，它们的内容如图 4.24 所示。

```
1   #include <stdio.h>
2   unsigned x=257;
3   short y, z=2;
4   void proc1(void);
5   void main()
6   {
7       proc1();
8       printf(“x=%u,z=%d\n”, x, z);
9       return 0;
10  }
```

a) main.c 文件

```
1   double x;
2
3   void proc1()
4   {
5       x=−1.5;
6   }
```

b) proc1.c 文件

图 4.24 题 5 用图

回答下列问题。

（1）在上述两个文件中出现的符号哪些是强符号？哪些是弱符号？

（2）程序执行后打印的结果是什么？请分别画出执行第 7 行的 proc1()函数调用前、后，在地址&x 和&z 中存放的内容。若 main.c 的第 3 行改为“short y=1, z=2;”，结果又会怎样？

（3）修改文件 proc1，使得 main.c 能输出正确的结果（即 x=257, z=2）。要求修改时不能改变任何变量的数据类型和名字。

参考答案：

（1）main.c 中强符号有 x、z、main，弱符号有 y 和 proc1；proc1.c 中的强符号有 proc1，弱符号有 x。根据多重定义符号处理规则 2（若一个符号被说明为一次强符号定义和多次弱符号定义，则按强符号定义为准），符号 x 的定义以 main.c 中的强符号 x 为准，即在 main.o 的.data 节中分配 x，占 4 个字节，随后是另一个强符号 z 占两个字节，x 和 z 都属于.data 节，随后是.bss 节，其中只有一个变量 y，按 4 字节对齐，因此，在 z 后面有两个字节空闲，再后面是变量 y 的空间。

（2）程序执行时，在调用 proc1()函数之前，&x 中存放的是 x 的机器数：00000101H，随后两个字节（地址为&z）存放 z，即 0002H，再后面两个字节空闲。如下左图所示：

	0	1	2	3
&z	02	00	---	---
&x	01	01	00	00

	0	1	2	3
&z	00	00	F8	BF
&x	00	00	00	00

在调用 proc1()函数以后，因为 proc1()中的符号 x 是弱符号，因此，x 的定义以 main 中的强符号 x 为准，执行 x=−1.5 后，便将“−1.5”的机器数 BFF80000 00000000H 存放到了&x 开始的 8 个字节中。即&x 中为其低 32 位的 00000000H，&z 中为高 32 位的 BFF80000H 中的低 16 位 0000H，z 后面的两个空闲字节中为高 16 位 BFF8H。如上右图所示。

因此，最终打印的结果如下：x=0, z=0。

若 main.c 的第 3 行改为“short y=1, z=2;”，则 x、y、z 都是强符号，都被分配在.data 节中，因此，x 占 4 个字节，随后是 y 占两个字节，z 占两个字节，proc1 函数执行前后的存储内容如下图所示，因此，最终打印的结果如下：x=0, z=−16392。

	0	1	2	3
&y	01	00	02	00
&x	01	01	00	00

	0	1	2	3
&y	00	00	F8	BF
&x	00	00	00	00

（3）只要将文件 proc1.c 中的第 1 行修改为“static double x;”就可以使得 proc1 中的 x 设定为本地变量，从而在 proc1.o 的.data 节中专门分配存放 x 的 8 个字节空间，而不会和 main 中的 x 共用同一个存储地址。因此，也就不会破坏 main 中 x 和 z 的值。

6. 以下每一小题给出了两个源程序文件，它们被分别编译生成可重定位目标模块 m1.o 和 m2.o。在模块 *mj* 中对符号 *x* 的任意引用与模块 *mi* 中定义的符号 *x* 关联记为 REF(*mj.x*)→DEF(*mi.x*)。请在下列空格处填写模块名和符号名以说明给出的引用符号所关联的定义符号，若发生链接错误则说明其原因；若从多个定义符号中任选则给出全部可能的定义符号，若是局部变量则说明不存在关联。

（1）
```
/* m1.c */
int p1(viod);
int main()
{
    int p1= p1();
    return p1;
}
```

```
/* m2.c */
static int main=1;
int p1()
{
    main++;
    return main;
}
```

① REF(m1.main)→DEF(_____._____)
② REF(m2.main)→DEF(_____._____)
③ REF(m1.p1)→DEF(_____._____)
④ REF(m2.p1)→DEF(_____._____)

（2）
```
/* m1.c */
int x=100;
int p1(viod);
int main()
{
    x=p1();
    return x;
}
```

```
/* m2.c */
float x=100.0;
int main=1;
int p1()
{
    main++;
    return main;
}
```

① REF(m1.main)→DEF(_____._____)
② REF(m2.main)→DEF(_____._____)
③ REF(m1.x)→DEF(_____._____)

（3）
```
/* m1.c */
int p1(viod);
int p1;
int main()
{
    int x=p1();
    return x;
}
```

```
/* m2.c */
int x=10;
int main;
int p1()
{
    main=1;
    return x;
}
```

① REF(m1.main)→DEF(_____._____)
② REF(m2.main)→DEF(_____._____)
③ REF(m1.p1)→DEF(_____._____)
④ REF(m1.x)→DEF(_____._____)
⑤ REF(m2.x)→DEF(_____._____)

（4）

```
/* m1.c */
int p1(viod);
int x, y;
int main()
{
    x=p1();
    return x;
}
```

```
/* m2.c */
double x=10;
int y;
int p1()
{
    y=1;
    return y;
}
```

① REF(m1.x)→DEF(_____._____)

② REF(m2.x)→DEF(_____._____)

③ REF(m1.y)→DEF(_____._____)

④ REF(m2.y)→DEF(_____._____)

参考答案：

（1） main 在 m1 中是强定义，在 m2 中是本地符号

① REF(m1.main)→在 m1 中不存在对 main 的引用

② REF(m2.main)→DEF(__m2__.__main__)

③ REF(m1.p1)→DEF(__m2__.__p1__)

④ REF(m2.p1)→在 m2 中不存在对 p1 的引用

（2）发生链接错误，因为全局变量 main 有两个强定义

（3）main 在 m1 中是强定义符号，在 m2 中是弱符号，因此链接器选择强定义

① REF(m1.main)→在 m1 中不存在对 main 的引用

② REF(m2.main)→DEF(__m1__.main__)

③ REF(m1.p1)→DEF(__m2__.__p1__)

④ REF(m1.x)→在 m1 中引用的 x 是局部变量，不存在关联

⑤ REF(m2.x)→DEF(__m2__.__x__)

（4）全局符号 x 在 m1 中是弱定义，在 m2 中是强定义，y 在两个模块中都是弱定义

① REF(m1.x)→DEF(__m2__.__x__)

② REF(m2.x)→在 m2 中不存在对 x 的引用

③ REF(m1.y)→在 m1 中不存在对 y 的引用

④ REF(m2.y)→DEF(__m1__.__y__) 或者 DEF(__m2__.__y__)

7. 以下由两个目标模块 m1 和 m2 组成的程序，经编译、链接后在计算机上执行，结果发现即使 p1 中没有对数组变量 main 进行初始化，最终也能打印出字符串“0x5589\n”。为什么？要求解释原因。

```
1     /* m1.c */
2     void p1(viod);
3
4     int main()
5     {
6         p1();
7         return 0;
8     }
```

```
1     /* m2.c */
2     #include <stdio.h>;
3     char main[2];
4
5     void p1()
6     {
7         printf(“0x%x%x\n”, main[0], main[1]);
8     }
```

参考答案：

全局符号 main 在 m1 中是强符号，在 m2 中是弱符号，因此，以 m1 中 main 的定义为准。在 m1 中全局符号 main 被定义在.text 节中，通常 main 函数开始两条指令如下：

```
1    Disassembly of section .text:
2    00000000 <main>:
```

```
3        0: 55                    push   %ebp
4        1: 89 e5                 mov    %esp,%ebp
```

其中，55H 是指令“push %ebp”的机器码，89E5H 是指令“mov %esp, %ebp”的机器码。因此，可以看出在 m2 中的 printf 语句中引用数组元素 main[0]和 main[1]时，main[0]=55H，main[1]=89H。

8. 图 4.25 中给出了用 OBJDUMP 显示的某个可执行目标文件的程序头表（段头表）的部分信息，其中，可读写数据段（Read/write data segment）的信息表明，该数据段对应虚拟存储空间中起始地址为 0x8049448、长度为 0x104 个字节的存储区，其数据来自可执行文件中偏移地址 0x448 开始的 0xe8 个字节。这里，可执行目标文件中的数据长度和虚拟地址空间中的存储区大小之间相差了 28 字节。请解释可能的原因。

```
Read-only code segment
  LOAD off    0x00000000 vaddr 0x08048000 paddr 0x08048000 align 2**12
              filesz 0x00000448 memsz 0x00000448 flags r-x

Read/write data segment
  LOAD off    0x00000448 vaddr 0x08049448 paddr 0x08049448 align 2**12
              filesz 0x000000e8 memsz 0x00000104 flags rw-
```

图 4.25 某可执行目标文件程序头表的部分内容

参考答案：

在可执行目标文件中描述的“可读写数据段”由所有可重定位目标文件中的.data 节合并生成的.data 节、所有可重定位目标文件中的.bss 节合并生成的.bss 节这两部分组成。.data 节由初始化的全局变量组成，因而其初始值必须记录在可执行文件中，而.bss 节由未初始化的全局变量组成，因而在可执行目标文件中无需记录其值，只要描述总的长度和每个变量的起始位置即可。

根据图 4.25 中的内容可知，.data 节中全局变量的初始值总的数据长度为 0xe8。因此，虚拟地址空间中长度为 0x104 字节的可读写数据段中，开始的 0xe8 个字节取自.data 节，后面的 28 字节是未初始化全局变量所在区域。

9. 假定 a 和 b 是可重定位目标文件或静态库文件，a→b 表示 b 中定义了一个被 a 引用的符号。对于以下每一小题出现的情况，给出一个最短命令行（含有最少数量的可重定位目标文件或静态库文件参数），使得链接器能够解析所有的符号引用。

（1）p.o→libx.a→liby.a→p.o

（2）p.o→libx.a→liby.a 同时 liby.a→libx.a

（3）p.o→libx.a→liby.a→libz.a 同时 liby.a→libx.a→libz.a

参考答案：

（1）gcc –static –o p p.o libx.a liby.a p.o

（2）gcc –static –o p p.o libx.a liby.a libx.a

（3）gcc –static –o p p.o libx.a liby.a libx.a libz.a

10. 图 4.15 给出了图 4.7a 所示的 main 源代码对应的 main.o 中.text 节和.rel.text 节的内容，图中显示其.text 节中有一处需重定位。假定链接后 main 函数代码起始地址是 0x8048386，紧跟在 main 后的是 swap 函数的代码，且首地址按 4 字节边界对齐。要求根据对图 4.15 的分析，指出 main.o

```
1   Disassembly of section .text:
2   00000000 <main>:
3      0:55                      push   %ebp
4      1:89 e5                   mov    %esp,%ebp
5      3:83 e4 f0                and    $0xfffffff0,%esp
6      6:e8 fc ff ff ff          call   7 <main+0x7>
7                7: R_386_PC32  swap
8      b:b8 00 00 00 00          mov    $0x0,%eax
9     10:c9                      leave
10    11:c3                      ret
```

图 4.15 main.o 中.text 节和.rel.text 节内容

的.text 节中需重定位的符号名、相对于.text 节起始位置的位移、所在指令行号、重定位类型、重定位前的内容、重定位后的内容，并给出重定位值的计算过程。

参考答案：

根据图 4.15 可知，main.o 的.text 节中只有一个符号需要重定位，它就是在 main.c 中被引用的全局符号 swap；需要重定位的是图 4.15 中第 6 行 call 指令中的偏移量字段，其位置相对于.text 节起始位置位移量为 7，按照 PC 相对地址方式（R_386_PC32）。

重定位前，在位移量 7、8、9、a 处的初始值 init 的内容分别为 fc ff ff ff，其机器数为 0xfffffffc，值为−4。重定位后，应该使 call 指令的目标转移地址指向 swap 函数的起始地址。

main 函数总共占 12H=18 字节的存储空间，其起始地址为 0x8048386，因此，main 函数最后一条指令地址为：0x8048386+0x12=0x8048398。因为 swap 函数代码紧跟在 main 后且首地址按 4 字节边界对齐，故 swap 的起始地址就是 0x8048398。

重定位值的计算公式为：

ADDR(r_sym) − ((ADDR(.text) + r_offset) − init)

= 0x8048398−((0x8048386+7)−(−4)) = 7

因此，重定位后，在位移量 7、8、9、a 处的 call 指令的偏移量字段为 07 00 00 00。

11．图 4.18 给出了图 4.7b 所示的 swap 源代码对应的 swap.o 文件中.text 节和.rel.text 节的内容，图中显示.text 节中共有 6 处需重定位。假定链接后生成的可执行目标文件中 buf 和 bufp0 的存储地址分别是 0x80495c8 和 0x80495d0，bufp1 的存储地址位于.bss 节的开始，为 0x8049620。根据对图 4.18 的分析，仿照例子填写下表，指出各个重定位的符号名、相对于.text 节起始位置的位移、所在指令行号、重定位类型、重定位前的内容、重定位后的内容。

序号	符号	位移	指令所在行号	重定位类型	重定位前内容	重定位后内容
1	bufp1（.bss）	0x8	6~7	R_386_32	0x00000000	0x8049620
2						
3						
4						
5						
6						

参考答案：

序号	符号	位移	指令所在行号	重定位类型	重定位前内容	重定位后内容
1	bufp1（.bss）	0x8	6~7	R_386_32	0x00000000	0x8049620
2	buf（.data）	0xc	6~7	R_386_32	0x00000004	0x80495cc
3	bufp0（.data）	0x11	10	R_386_32	0x00000000	0x80495d0
4	bufp0（.data）	0x1b	14	R_386_32	0x00000000	0x80495d0
5	bufp1（.bss）	0x21	17	R_386_32	0x00000000	0x8049620
6	bufp1（.bss）	0x2a	21	R_386_32	0x00000000	0x8049620

```
1   Disassembly of section .text:
2   00000000 <swap>:
3       0:55                        push   %ebp
4       1:89 e5                     mov    %esp,%ebp
5       3:83 ec 10                  sub    $0x10,%esp
6       6:c7 05 00 00 00 00 04      movl   $0x4,0x0
7       d:00 00 00                                          } bufp1=&buf[1]
8               8: R_386_32   .bss                          }
9               c: R_386_32   buf                           }
10      10:a1 00 00 00 00           mov    0x0,%eax                    }
11              11: R_386_32  bufp0                                      }
12      15:8b 00                    mov    (%eax),%eax                 } temp=*bufp0
13      17:89 45 fc                 mov    %eax,-0x4(%ebp)             }
14      1a:a1 00 00 00 00           mov    0x0,%eax                    }
15              1b: R_386_32  bufp0                                      }
16      1f:8b 15 00 00 00 00        mov    0x0,%edx                    }
17              21: R_386_32  .bss                                       } *bufp0=*bufp1
18      25:8b 12                    mov    (%edx),%edx                 }
19      27:89 10                    mov    %edx,(%eax)                 }
20      29:a1 00 00 00 00           mov    0x0,%eax                    }
21              2a: R_386_32  .bss                                       }
22      2e:8b 55 fc                 mov    -0x4(%ebp),%edx             } *bufp1=temp
23      31:89 10                    mov    %edx,(%eax)                 }
24      33:c9                       leave
25      34:c3                       ret
```

图 4.18 swap.o 中.text 节和.rel.text 节内容