Linklab实验报告

张睿恒 2024302182001

一、摘要

本总结共分文五个部分：摘要、实验前对编译的理解、实验内容、实验后对链接的理解、总结。其中实验内容是本次实验的主体：通过编译链接不同的C语言源文件对链接过程进行探讨；借助objdump与readelf阅读可执行文件的不同内容；了解了几种经典的链接时错误。

二、实验前对链接的理解

在实验开始前，我对链接的理解主要是这样的：链接是高级语言程序编译成为可执行文件的重要部分之一，主要负责将不同的.o文件和库组合成一个可执行文件。主要任务是符号解析和重定位。其中符号解析是确定每个符号（包括函数名、变量名甚至文件名）的含义，重定位是调整代码段和数据段在内存中的地址。符号又分为强符号和弱符号，其中强符号是指以初始化的全局变量或函数名；弱符号则是未初始化的全局符号。在目前编译器默认的-fno-common编译选项中，多个同名弱符号被视作是warning。而重定位中一个重要的部分就是寻址。在目前的编译器中，默认打开了编译选项-fpic，用来生成“位置无关代码“。

三、实验内容

1. 构建prog

```
# 首先是编译阶段，利用gcc命令
gcc -Og -o main.o main.c sum.c # 将main.c和sum.c编译链接成main.o文件
gcc -Og -o sum.o sum.c main.c
gcc -Og -S main.s main.c # 将main.c 编译成机器码(汇编)
# gcc -Wall -Og -o prog sum.o main.o
# 这条指令原本出现在实验文档上，但操作中遇到了报错（如图1-1所示，无法将.o文件作为链接的输入）
gcc -Wall -Og -o prog main.c sum.c # 因此，我直接将.c文件进行编译链接来构建prog
# 下面是反编译阶段，利用objdump命令
objdump -dx main.o      > main-relo.d # 查看main函数的依赖。-dx是-d 和 -x 的简写，下同
objdump -dx -j .data main.o > maindata-relo.d # -j 是主要针对某个section进行输出，比如这条命令是只查看.data字段的部分
readelf -s main.o          > mainsym.d # readelf命令是用来查看elf文件的一些细节内容。elf文件就是可执行文件的一种通用格式
# -s 命令选项是用来读取符号表的
objdump -dx sum.o      > sum-relo.d
objdump -dx -j .data sum.o  > sumdata-relo.d
objdump -dx prog       > prog-exe.d
objdump -dx -j .data prog  > progdata-exe.d
```

下面两幅图是实验过程中构建的具体步骤。其中显示了链接.o文件的报错以及prog的构建过程

```
zrheng@zrheng-ThinkPad-E495:~/Desktop/link$ gcc -Og -o main.o main.c sum.c
zrheng@zrheng-ThinkPad-E495:~/Desktop/link$ gcc -Og -o sum.o sum.c main.c
zrheng@zrheng-ThinkPad-E495:~/Desktop/link$ gcc -Og -o prog sum.o main.o
/usr/bin/ld: cannot use executable file 'sum.o' as input to a link
collect2: error: ld returned 1 exit status
zrheng@zrheng-ThinkPad-E495:~/Desktop/link$ gcc -Og -o prog main.o sum.o
/usr/bin/ld: cannot use executable file 'main.o' as input to a link
collect2: error: ld returned 1 exit status
zrheng@zrheng-ThinkPad-E495:~/Desktop/link$ gcc -Wall -Og -o prog main.o sum.o
/usr/bin/ld: cannot use executable file 'main.o' as input to a link
collect2: error: ld returned 1 exit status
zrheng@zrheng-ThinkPad-E495:~/Desktop/link$ gcc -Wall -Og -o prog main.c sum.c
zrheng@zrheng-ThinkPad-E495:~/Desktop/link$ gcc -Wall -Og -S main.s
zrheng@zrheng-ThinkPad-E495:~/Desktop/link$ gcc -Wall -Og -S main.c
```

```
zrheng@zrheng-ThinkPad-E495:~/Desktop/link$ objdump -dx main.o > main-relo.d
zrheng@zrheng-ThinkPad-E495:~/Desktop/link$ objdump -dx sum.o > sum-relo.d
zrheng@zrheng-ThinkPad-E495:~/Desktop/link$ readelf -s main.o > mainsym.d
zrheng@zrheng-ThinkPad-E495:~/Desktop/link$ objdump -dx -j .data main.o > maindata-relo.d
zrheng@zrheng-ThinkPad-E495:~/Desktop/link$ objdump -dx -j .data sum.o > sumdata-relo.d
zrheng@zrheng-ThinkPad-E495:~/Desktop/link$ objdump -dx prog > prog-exe.d
zrheng@zrheng-ThinkPad-E495:~/Desktop/link$ objdump -dx -j .data prog > progdata-exe.d
zrheng@zrheng-ThinkPad-E495:~/Desktop/link$ ls
addvec.c      foo2.c       linkerror.c      m.c              sum-relo.d
bar1.c        foo3.c       main2.c          multvec.c        swap2.c
bar2.c        foo4.c       main.c           prog             swap.c
bar3.c        foo5.c       maindata-relo.d  progdata-exe.d   t1.c
bar4.c        foo6.c       main.o           prog-exe.d       t2.c
bar5.c        fragments.c  main-relo.d      share.c          vector.h
bar6.c        hello.c      main.s           static.c
dll.c         interpose    mainsym.d        sum.c
elfstructs.c  libc.so.6    Makefile         sumdata-relo.d
foo1.c        libvector.a  map.c            sum.o
```

2.在编译时构建静态链接库

```
gcc -Wall -Og -c main2.c addvec.c multvec.c # 仅生成编译后的文件，而不进行链接
ar rcs libvector.a addvec.o multvec.o
gcc -Wall -Og -static -o prog2c main2.c libvector.a # 值得注意的，在文档中好像把字母o打成了数字0。
gcc -Wall -Og -static -o prog2c main2.c -L. -lvector
```

在这里解释一下ar命令的后面3个options（因为太长注释写不下）：r代表如果这个lib中已经有内容了，那么用后面的新的文件来代替之前的老的；c代表着创建一个新的lib（如果没有已经存在的的话）；s代表着给这个lib添加一个索引，这样后续编译的时候会快。因此，rcs可以看作是replace,create,sort的缩写。值得注意的是，虽然看起来似乎用了ar这个和压缩包相关的命令，但其实rcs options更多的是在用makefile之类的文件进行编译时用的。

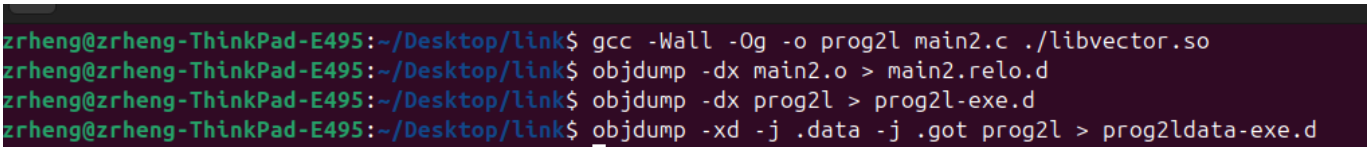
下图是实验过程中构建静态链接库的具体步骤。

```
zrheng@zrheng-ThinkPad-E495:~/Desktop/link$ gcc -Wall -Og -c main2.c addvec.c multvec.c
zrheng@zrheng-ThinkPad-E495:~/Desktop/link$ ar rcs libvector.a addvec.o multvec.o
zrheng@zrheng-ThinkPad-E495:~/Desktop/link$ gcc -Wall -Og -static -o prog2c main2.0 libvec.a

/usr/bin/ld: cannot find main2.0: No such file or directory
/usr/bin/ld: cannot find libvec.a: No such file or directory
collect2: error: ld returned 1 exit status
zrheng@zrheng-ThinkPad-E495:~/Desktop/link$ gcc -Wall -Og -static -o prog2c main2.c libvector.a

zrheng@zrheng-ThinkPad-E495:~/Desktop/link$ gcc -Wall -Og -static -o prog2c main2.c -L. -lvector
```

3.在加载时构建共享链接库

```
gcc -Wall -Og -o prog2l main2.c ./libvector.so # 在第一次完成的时候，报错“没有找到libvector.so库”
# 后来发现，确实需要自己构建libvector.so这个库。具体构建方法可以参考第四个part
objdump -dx main2.o      > main2-relo.d # 这些同理，不再赘述
objdump -dx prog2l > prog2l-exe.d
objdump -xs -j .data -j .got prog2l > prog2ldata-exe.d # 这块用到了-s这个命令，是用来输出多个section的
```

下面给出实验过程中的贴图。



4. 在运行时构建共享链接库

```
gcc -Wall -Og -rdynamic -o prog2r dll.c -ldl # 关于rdynamic 这个编译选项在下面有详细的即使
gcc -Wall -Og -shared -fpic -o libvector.so addvec.c multvec.c
objdump -xd libvector.so > libvector-relo.d
objdump -xRr -j .data -j .got.plt -j dela.dtn -j rela.plt libvector.so > libvectordata-relo.d
```

在这里首先解释一下gcc编译选项rdynamic是什么意思：rdynamic是--export-dynamic的简写，主要作用是用来链接共享库。当创建一个动态链接程序的时候，这个编译选项会将所有的符号加入到符号表里面。

再解释一下构建libvector.so的过程：加入-fpic选项，意味着构建了一个不依赖特定地址而“位置无关”（position independent）的机器码。以jump为例

```
100:cmp reg1,reg2
101:jmpe current+10

111:NOP
```

这段代码不论在地址为100还是1000都会正常工作。这对于构建一个共享库来说是很好的，因为每次链接时，他都会被重定位到内存的不同地址。采用这种“内存独立”的编译方式便于每次链接。

下面给出实验过程中的贴图

```
zrheng@zrheng-ThinkPad-E495:~/Desktop/link$ gcc -Wall -Og -rdynamic -o prog2r dll.c -ldl
zrheng@zrheng-ThinkPad-E495:~/Desktop/link$ gcc -Wall -Og -shared -fpic -o libvector.so addvec.c multvec.c
zrheng@zrheng-ThinkPad-E495:~/Desktop/link$ objdump -xd libvector.so > libvector-relo.d
zrheng@zrheng-ThinkPad-E495:~/Desktop/link$ objdump -xRr -j .data -j .got.plt -j rela.dtn -j rela.plt libvector.so > libvectordata-relo.d
```

5.查看符号表的练习

P1

先看programme header table，可以看到是一个“位置无关”的可执行文件（“位置无关”（position-independent）在上一个part中有描述），有13个header。后面给出了每个header的地址。

```
zrheng@zrheng-ThinkPad-E495:~/Desktop/link$ readelf -l p1

Elf file type is DYN (Position-Independent Executable file)
Entry point 0x1040
There are 13 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000000040 0x0000000000000040
                 0x00000000000002d8 0x00000000000002d8  R      0x8
  INTERP         0x0000000000000318 0x0000000000000318 0x0000000000000318
                 0x000000000000001c 0x000000000000001c  R      0x1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000608 0x0000000000000608  R      0x1000
  LOAD           0x0000000000001000 0x0000000000001000 0x0000000000001000
                 0x0000000000000181 0x0000000000000181  R E    0x1000
  LOAD           0x0000000000002000 0x0000000000002000 0x0000000000002000
                 0x00000000000000d8 0x00000000000000d8  R      0x1000
  LOAD           0x0000000000002df0 0x0000000000003df0 0x0000000000003df0
                 0x0000000000000230 0x0000000000000240  RW     0x1000
  DYNAMIC        0x0000000000002e00 0x0000000000003e00 0x0000000000003e00
                 0x00000000000001c0 0x00000000000001c0  RW     0x8
  NOTE           0x0000000000000338 0x0000000000000338 0x0000000000000338
                 0x0000000000000030 0x0000000000000030  R      0x8
```

然后查看一下符号表，一共有40个符号。其中从0-18号是链接器内部产生的局部变量。符号表中包括变量的大小、类型（对象、函数、文件等）、名字等。

```
Symbol table '.symtab' contains 40 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS Scrt1.o
     2: 000000000000038c    32 OBJECT  LOCAL  DEFAULT    4 __abi_tag
     3: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS crtstuff.c
     4: 0000000000001070     0 FUNC    LOCAL  DEFAULT   14 deregister_tm_clones
     5: 00000000000010a0     0 FUNC    LOCAL  DEFAULT   14 register_tm_clones
     6: 00000000000010e0     0 FUNC    LOCAL  DEFAULT   14 __do_global_dtors_aux
     7: 0000000000004020     1 OBJECT  LOCAL  DEFAULT   24 completed.0
     8: 0000000000003df8     0 OBJECT  LOCAL  DEFAULT   20 __do_global_dtor[...]
     9: 0000000000001120     0 FUNC    LOCAL  DEFAULT   14 frame_dummy
    10: 0000000000003df0     0 OBJECT  LOCAL  DEFAULT   19 __frame_dummy_in[...]
    11: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS m.c
    12: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS swap.c
    13: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS crtstuff.c
    14: 00000000000020d4     0 OBJECT  LOCAL  DEFAULT   18 __FRAME_END__
    15: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS
    16: 0000000000003e00     0 OBJECT  LOCAL  DEFAULT   21 _DYNAMIC
    17: 0000000000002004     0 NOTYPE  LOCAL  DEFAULT   17 __GNU_EH_FRAME_HDR
    18: 0000000000003fc0     0 OBJECT  LOCAL  DEFAULT   22 _GLOBAL_OFFSET_TABLE_
    19: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __libc_start_mai[...]
    20: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_deregisterT[...]
    21: 0000000000004000     0 NOTYPE  WEAK   DEFAULT   23 data_start
    22: 0000000000004020     0 NOTYPE  GLOBAL DEFAULT   23 _edata
    23: 0000000000001174     0 FUNC    GLOBAL HIDDEN    15 _fini
    24: 0000000000004000     0 NOTYPE  GLOBAL DEFAULT   23 __data_start
    25: 0000000000004018     8 OBJECT  GLOBAL DEFAULT   23 bufp0
    26: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
    27: 0000000000004008     0 OBJECT  GLOBAL HIDDEN    23 __dso_handle
    28: 0000000000002000     4 OBJECT  GLOBAL DEFAULT   16 _IO_stdin_used
    29: 0000000000004030     0 NOTYPE  GLOBAL DEFAULT   24 _end
    30: 0000000000001040    38 FUNC    GLOBAL DEFAULT   14 _start
    31: 0000000000004010     8 OBJECT  GLOBAL DEFAULT   23 buf
    32: 0000000000004020     0 NOTYPE  GLOBAL DEFAULT   24 __bss_start
    33: 0000000000001129    28 FUNC    GLOBAL DEFAULT   14 main
    34: 0000000000004020     0 OBJECT  GLOBAL HIDDEN    23 __TMC_END__
    35: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_registerTMC[...]
    36: 0000000000001145    45 FUNC    GLOBAL DEFAULT   14 swap
    37: 0000000000000000     0 FUNC    WEAK   DEFAULT  UND __cxa_finalize@G[...]
    38: 0000000000001000     0 FUNC    GLOBAL HIDDEN    11 _init
    39: 0000000000004028     8 OBJECT  GLOBAL DEFAULT   24 bufp1
```

接下来是节表。p1一共有29个节。

```
zrheng@zrheng-ThinkPad-E495:~/Desktop/link$ readelf -S p1
There are 29 section headers, starting at offset 0x3700:

Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .interp           PROGBITS         0000000000000318  00000318
       000000000000001c  0000000000000000   A       0     0     1
  [ 2] .note.gnu.pr[...] NOTE             0000000000000338  00000338
       0000000000000030  0000000000000000   A       0     0     8
  [ 3] .note.gnu.bu[...] NOTE             0000000000000368  00000368
       0000000000000024  0000000000000000   A       0     0     4
  [ 4] .note.ABI-tag     NOTE             000000000000038c  0000038c
       0000000000000020  0000000000000000   A       0     0     4
  [ 5] .gnu.hash         GNU_HASH         00000000000003b0  000003b0
       0000000000000024  0000000000000000   A       6     0     8
  [ 6] .dynsym           DYNSYM           00000000000003d8  000003d8
       0000000000000090  0000000000000018   A       7     1     8
  [ 7] .dynstr           STRTAB           0000000000000468  00000468
       0000000000000088  0000000000000000   A       0     0     1
  [ 8] .gnu.version      VERSYM           00000000000004f0  000004f0
       000000000000000c  0000000000000002   A       6     0     2
  [ 9] .gnu.version_r    VERNEED          0000000000000500  00000500
       0000000000000030  0000000000000000   A       7     1     8
  [10] .rela.dyn         RELA             0000000000000530  00000530
       00000000000000d8  0000000000000018   A       6     0     8
  [11] .init             PROGBITS         0000000000001000  00001000
       000000000000001b  0000000000000000  AX       0     0     4
  [12] .plt              PROGBITS         0000000000001020  00001020
       0000000000000010  0000000000000010  AX       0     0     16
  [13] .plt.got          PROGBITS         0000000000001030  00001030
       0000000000000010  0000000000000010  AX       0     0     16
  [14] .text             PROGBITS         0000000000001040  00001040
       0000000000000132  0000000000000000  AX       0     0     16
  [15] .fini             PROGBITS         0000000000001174  00001174
       000000000000000d  0000000000000000  AX       0     0     4
  [16] .rodata           PROGBITS         0000000000002000  00002000
       0000000000000004  0000000000000004  AM       0     0     4
  [17] .eh_frame_hdr     PROGBITS         0000000000002004  00002004
       0000000000000034  0000000000000000   A       0     0     4
```

P3

先看programme header table，p3也包含13个header，入口是在0x1040位置的地址。

然后查看一下符号表。可以看到，有42个不同的符号。

```
Symbol table '.symtab' contains 42 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS Scrt1.o
     2: 000000000000038c    32 OBJECT  LOCAL  DEFAULT    4 __abi_tag
     3: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS crtstuff.c
     4: 0000000000001070     0 FUNC    LOCAL  DEFAULT   14 deregister_tm_clones
     5: 00000000000010a0     0 FUNC    LOCAL  DEFAULT   14 register_tm_clones
     6: 00000000000010e0     0 FUNC    LOCAL  DEFAULT   14 __do_global_dtors_aux
     7: 0000000000004020     1 OBJECT  LOCAL  DEFAULT   24 completed.0
     8: 0000000000003df8     0 OBJECT  LOCAL  DEFAULT   20 __do_global_dtor[...]
     9: 0000000000001120     0 FUNC    LOCAL  DEFAULT   14 frame_dummy
    10: 0000000000003df0     0 OBJECT  LOCAL  DEFAULT   19 __frame_dummy_in[...]
    11: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS m.c
    12: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS swap2.c
    13: 0000000000001145     8 FUNC    LOCAL  DEFAULT   14 incr
    14: 0000000000004028     4 OBJECT  LOCAL  DEFAULT   24 count.0
    15: 0000000000004030     8 OBJECT  LOCAL  DEFAULT   24 bufp1
    16: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS crtstuff.c
    17: 00000000000020f0     0 OBJECT  LOCAL  DEFAULT   18 __FRAME_END__
    18: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS
    19: 0000000000003e00     0 OBJECT  LOCAL  DEFAULT   21 _DYNAMIC
    20: 0000000000002004     0 NOTYPE  LOCAL  DEFAULT   17 __GNU_EH_FRAME_HDR
    21: 0000000000003fc0     0 OBJECT  LOCAL  DEFAULT   22 _GLOBAL_OFFSET_TABLE_
    22: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __libc_start_mai[...]
    23: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_deregisterT[...]
    24: 0000000000004000     0 NOTYPE  WEAK   DEFAULT   23 data_start
    25: 0000000000004020     0 NOTYPE  GLOBAL DEFAULT   23 _edata
    26: 0000000000001184     0 FUNC    GLOBAL HIDDEN    15 _fini
    27: 0000000000004000     0 NOTYPE  GLOBAL DEFAULT   23 __data_start
    28: 0000000000004018     8 OBJECT  GLOBAL DEFAULT   23 bufp0
    29: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
    30: 0000000000004008     0 OBJECT  GLOBAL HIDDEN    23 __dso_handle
    31: 0000000000002000     4 OBJECT  GLOBAL DEFAULT   16 _IO_stdin_used
    32: 0000000000004038     0 NOTYPE  GLOBAL DEFAULT   24 _end
    33: 0000000000001040    38 FUNC    GLOBAL DEFAULT   14 _start
    34: 0000000000004010     8 OBJECT  GLOBAL DEFAULT   23 buf
    35: 0000000000004020     0 NOTYPE  GLOBAL DEFAULT   24 __bss_start
    36: 0000000000001129    28 FUNC    GLOBAL DEFAULT   14 main
    37: 0000000000004020     0 OBJECT  GLOBAL HIDDEN    23 __TMC_END__
    38: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_registerTMC[...]
    39: 000000000000114d    52 FUNC    GLOBAL DEFAULT   14 swap
    40: 0000000000000000     0 FUNC    WEAK   DEFAULT  UND __cxa_finalize@G[...]
    41: 0000000000001000     0 FUNC    GLOBAL HIDDEN    11 _init
```

最后是节表，一共29个节。

```
There are 29 section headers, starting at offset 0x3740:

Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .interp           PROGBITS         0000000000000318  00000318
       000000000000001c  0000000000000000   A       0     0     1
  [ 2] .note.gnu.pr[...] NOTE             0000000000000338  00000338
       0000000000000030  0000000000000000   A       0     0     8
  [ 3] .note.gnu.bu[...] NOTE             0000000000000368  00000368
       0000000000000024  0000000000000000   A       0     0     4
  [ 4] .note.ABI-tag     NOTE             000000000000038c  0000038c
       0000000000000020  0000000000000000   A       0     0     4
  [ 5] .gnu.hash         GNU_HASH         00000000000003b0  000003b0
       0000000000000024  0000000000000000   A       6     0     8
  [ 6] .dynsym           DYNSYM           00000000000003d8  000003d8
       0000000000000090  0000000000000018   A       7     1     8
  [ 7] .dynstr           STRTAB           0000000000000468  00000468
       0000000000000088  0000000000000000   A       0     0     1
  [ 8] .gnu.version      VERSYM           00000000000004f0  000004f0
       000000000000000c  0000000000000002   A       6     0     2
  [ 9] .gnu.version_r    VERNEED          0000000000000500  00000500
       0000000000000030  0000000000000000   A       7     1     8
  [10] .rela.dyn         RELA             0000000000000530  00000530
       00000000000000d8  0000000000000018   A       6     0     8
  [11] .init             PROGBITS         0000000000001000  00001000
       000000000000001b  0000000000000000  AX       0     0     4
  [12] .plt              PROGBITS         0000000000001020  00001020
       0000000000000010  0000000000000010  AX       0     0     16
  [13] .plt.got          PROGBITS         0000000000001030  00001030
       0000000000000010  0000000000000010  AX       0     0     16
  [14] .text             PROGBITS         0000000000001040  00001040
       0000000000000141  0000000000000000  AX       0     0     16
  [15] .fini             PROGBITS         0000000000001184  00001184
       000000000000000d  0000000000000000  AX       0     0     4
  [16] .rodata           PROGBITS         0000000000002000  00002000
       0000000000000004  0000000000000004  AM       0     0     4
  [17] .eh_frame_hdr     PROGBITS         0000000000002004  00002004
       000000000000003c  0000000000000000   A       0     0     4
  [18] .eh_frame         PROGBITS         0000000000002040  00002040
       00000000000000b4  0000000000000000   A       0     0     8
  [19] .init_array       INIT_ARRAY       0000000000003df0  00002df0
```

foobar3

由于长度关系，后续的部分大同小异，就不贴图了，仅以语言叙述。

foo3的错误原因是"x“被定义了两次。multiple definition of `x'; /tmp/cc9TdmbJ.o:(.data+0x0): first defined here。

foobar4

错误原因是一样的：/usr/bin/ld: /tmp/ccnNjysb.o:(.bss+0x0): multiple definition of `x'; /tmp/ccMqo6Ul.o:(.bss+0x0): first defined here。弱符号x是被重定义了两边

foobar5

这个部分很有意思，x的地址的值（原谅我采用这个抽象的表述）被意外的重写了。在foo里面，x被定义成了整形的变量而在bar中又定义了一个double的x。由于bar中的定义是弱符号（没有初始化）而foo中的定义是强符号，导致链接器在解析的时候直接将bar中的double的x“当成”的在foo中定义的int的x。这就导致了在bar中对double的x进行赋值的时候，错误的赋写了int的x的地址，并同时影响了其相邻的4个字节的y的值。

6.链接失败

linkerror

这个错误原因是main函数用到了函数foo但是没有给出定义（存在未解析的符号）。/usr/bin/ld: /tmp/ccb1c0ZY.o: in function `main': linkerror.c:(.text+0x9): undefined reference to `foo'

foobar1

这个错误原因是main被定义了两次（两个强符号撞了）。/usr/bin/ld: /tmp/cc2pr75R.o: in function `main': bar1.c:(.text+0x0): multiple definition of `main'; /tmp/cc57iA0H.o:foo1.c:(.text+0x0): first defined here

foobar2

这个错误原因是x被定义了两次（同样是两个强符号撞了）不过这次是两个已经初始化了的全局变量。/usr/bin/ld: /tmp/ccPDdUq7.o:(.data+0x0): multiple definition of `x'; /tmp/ccB3MOUG.o:(.data+0x0): first defined here

四、实验后对链接的理解

本次实验之后，我对之前被我忽略的调库过程有了新的认识。

静态链接库的构建与使用：在实验中，我们通过 ar 工具创建了静态链接库，并将其链接到可执行文件中。这一过程让我明白了静态库的本质：它实际上是一个包含多个目标文件的归档文件，在链接时会被提取出来参与符号解析。

动态链接的灵活性：动态链接库在运行时加载的特性使其非常适合于需要频繁更新的场景。实验中对比了静态链接和动态链接的行为差异，让我认识到动态链接的优势，例如减少可执行文件的体积和提高库的复用性。

符号冲突的处理：通过分析 foo1.o 和 bar1.o 等案例，我了解到当多个目标文件中存在同名符号时，链接器会根据符号的强弱属性进行决策。如果出现两个强符号冲突，则会导致链接失败。

五、实验总结

本次实验通过一系列具体的操作，让我从理论到实践全面了解了链接的过程及其实现细节。我熟悉了 ELF 文件的各个组成部分，包括头部信息、程序头部表、节区表、符号表等。理解了静态链接和动态链接的区别：静态链接适合于独立性强、不需要频繁更新的应用，而动态链接则更适合于需要模块化设计和资源节约的场景。实验中对比两者的优缺点，使我能够根据实际需求选择合适的链接方式。同时，我在亲自动手操作中理解了程序从高级语言一步步变成可执行文件的过程。这对我以后高效的写多文件程序打下了一个很好的理论基础。最后，我了解了不同的编译链接选项，使我对程序的编译、链接、运行机制有了更好的了解。