Exercise 2

1. *SYMBOL RESOLUTION*. The following program consists of two modules: `foo` and `bar`. The source code are shown below.

```
/** foo.c **/
#include <stdio.h>
void f(void);
short a = 0x1;
short b;
static short c = 0x3;
int main(void) {
        b = 0x2;
        short d = 0x4;
        static int e = 0x10;
        f();
        printf("a=0x%x b=0x%x c=0x%x d=0x%x e=0x%x\n", a, b, c, d, e);
        return 0;
}

/** bar.c **/
long a;
int d;
void f(void) {
        a = 0x0;
        d = 0x0;
        int e = 0x0;
}
```

(a) For each symbol in `foo.o`, please indicate whether it will have a symbol table entry in the `.symtab` section. If Yes, please fill the binding (GLOBAL, LOCAL); If No, fill with '–'.

Sym Name	Has a .symtab Entry?	Binding
a	Y	GLOBAL
b	Y	GLOBAL
c	Y	LOCAL
d	N	--
e	Y	LOCAL
f	Y	GLOBAL

(b) `foo.o` and `bar.o` are linked to `foobar`. The output of `readelf -s foobar` is provided below (Some entries are omitted). What is the output after running `./foobar`?

```
   Num:    Value          Size Type    Bind   Vis      Ndx Name
   142: 00000000006b90f2     2 OBJECT  LOCAL  DEFAULT   21 c
   143: 00000000006b90f4     4 OBJECT  LOCAL  DEFAULT   21 e.2256
   746: 00000000006bc3a0     2 OBJECT  GLOBAL DEFAULT   26 b
   834: 0000000000400baf    35 FUNC    GLOBAL DEFAULT    6 f
  1376: 0000000000400b4d    98 FUNC    GLOBAL DEFAULT    6 main
  1425: 00000000006bc3a4     4 OBJECT  GLOBAL DEFAULT   26 d
  1633: 00000000006b90f0     2 OBJECT  GLOBAL DEFAULT   21 a
```

`a=0x0 b=0x2 c=0x0 d=0x4 e=0x0`

2. *ELF INSIDE*. For a source file `a.c`, an ELF object file `a.o` is derived using `gcc -c a.c` (compiled and assembled, but not linked). Here is the source code and a disassembly of the `.text` section of `a.o`.

```
long seq[3] = {1, 2, 3};
long flag;

int main(int argc, char **argv)
{
        if (seq[2] > 0) {
                flag = 1;
        }
        else {
                flag = 0;
```

```
        }
        return 0;
}

Disassembly of section .text:
0000000000000000 <main>:
   0:   55                      push   %rbp
   1:   48 89 e5                mov    %rsp,%rbp
   4:   89 7d fc                mov    %edi,-0x4(%rbp)
   7:   48 89 75 f0             mov    %rsi,-0x10(%rbp)
   b:   48 8b 05 00 00 00 00    mov    0x0(%rip),%rax
  12:   48 85 c0                test   %rax,%rax
  15:   7e 0d                   jle    24 <main+0x24>
  17:   48 c7 05 00 00 00 00    movq   $0x1,0x0(%rip)
  1e:   01 00 00 00
  22:   eb 0b                   jmp    2f <main+0x2f>
  24:   48 c7 05 00 00 00 00    movq   $0x0,0x0(%rip)
  2b:   00 00 00 00
  2f:   b8 00 00 00 00          mov    $0x0,%eax
  34:   5d                      pop    %rbp
  35:   c3                      retq
```

(a) What are the symbol table entries for symbol `seq`, `flag` and `main`?

Sym Name	Section/Pseudosection	Type	Binding	Size
seq	.data	OBJ	GLOBAL	24
flag	COM	OBJ	GLOBAL	8
main	.text	FUNC	GLOBAL	54

(b) What is the relocation entry for the position at `0xe` after the `.text` section?

Offset	Sym	Type	Addend
0xe	seq	R_X86_64_PC32	0xc

(c) When linking, suppose the linker has decided that the address of `.text` is `0x400b4d`, the address of `seq` is `0x6b90f0`. What is the relocated form of the `mov 0x0(%rip),%rax` instruction?

`0x6b90f0 - (0x400b4d + 0xe) + 0xc = 0x2b85a1`. So in the resulting executable object file, the relocated form of this instruction is `48 8b 05 a1 85 2b 00`.

(d) Clark prepares to do some hacking to the ELF file `a.o`. He wants to manually manipulate the `Type` field of the relocation entry in question (b) from R_X86_64_PC32 to R_X86_64_32. Please try you best to give a guess of how he does it step-by-step. You don't need to worry about the details of file-related operations. For your information, it first opens `a.o`, then after `mmap`, the file data is "mapped" into the memory. Memory accesses to this region are all backed by the underlying file.

```
        if ((fd = open(file_name, O_RDWR)) < 0) {
                fprintf(stderr, "Cannot open file %s\n", file_name);
                return errno;
        }

        fstat(fd, &sb);
        file_mapped = mmap(NULL, sb.st_size, PROT_READ | PROT_WRITE,
                           MAP_SHARED, fd, 0);
        if (file_mapped == MAP_FAILED) {
                fprintf(stderr, "mmap\n");
                return errno;
        }

        ehdr = (Elf64_Ehdr *)file_mapped;
        shdrs = (Elf64_Shdr *)(file_mapped + ehdr->e_shoff);
        for (i = 0; i < ehdr->e_shnum; i++) {
                if (shdrs[i].sh_type == SHT_RELA)
                        break;
        }
```

```
        if (i >= ehdr->e_shnum) {
                fprintf(stderr, "rel not found\n");
                return -1;
        }

        rel = (Rel_ent *)(file_mapped + shdrs[i].sh_offset);

        rel[0].r_type = R_X86_64_32;

        munmap(file_mapped, sb.st_size);
        close(fd);
```

- `ehdr` represents the ELF header which is at the beginning of the ELF file.
- In ELF header, there is a field (`ehdr->e_shoff`) tells the byte offset of the section header table.
- The section header table is an array of section headers (`shdrs`).
- Each section header records the information about one section. The field about section type is searched for the relocation section (`SHT_RELA`).
- Once found, use the offset in the section header (`shdrs[i].sh_offset`) to locate the relocation section.
- The relocation section is an array of reloc entries. Change the first entry's type field to R_X86_64_32.

(e) After (d), what is the relocated form of the `mov 0x0(%rip),%rax` instruction? (Suppose the address of `.text` and `seq` is the same as question (c))

`0x6b90f0 + 0xc = 0x6b90fc`. So in the resulting executable object file, the relocated form of this instruction is `48 8b 05 fc 90 6b 00`.

3. *RELOCATION.* Two source files and the disassembly of their object files are given below.

```
/** foo.c **/
static int n = 2013;
int *p_n = &n;
int foo(int x) {
        if (x < n) return 1;
        return foo(x-1) * n;
}

/** bar.c **/
extern int foo(int n);
extern int *p_n;
int n = 2015;
int a[2048];
void bar(void) {
        *p_n = 2014;
        a[2] = foo(n);
}

/** foo.obj **/
0000000000000000 <foo>:
   0:   55                      push   %rbp
   1:   48 89 e5                mov    %rsp,%rbp
   4:   48 83 ec 10             sub    $0x10,%rsp
   8:   89 7d fc                mov    %edi,-0x4(%rbp)
   b:   8b 05 00 00 00 00       mov    0x0(%rip),%eax         @@
  11:   39 45 fc                cmp    %eax,-0x4(%rbp)
  14:   7d 07                   jge    1d <foo+0x1d>
  16:   b8 01 00 00 00          mov    $0x1,%eax
  1b:   eb 18                   jmp    35 <foo+0x35>
  1d:   8b 45 fc                mov    -0x4(%rbp),%eax
  20:   83 e8 01                sub    $0x1,%eax
  23:   89 c7                   mov    %eax,%edi
```

```
  25:   e8 00 00 00 00           callq  2a <foo+0x2a>         @@
  2a:   89 c2                    mov    %eax,%edx
  2c:   8b 05 00 00 00 00        mov    0x0(%rip),%eax
  32:   0f af c2                 imul   %edx,%eax
  35:   c9                       leaveq
  36:   c3                       retq

/** bar.obj **/
0000000000000000 <bar>:
   0:   55                       push   %rbp
   1:   48 89 e5                 mov    %rsp,%rbp
   4:   48 8b 05 00 00 00 00     mov    0x0(%rip),%rax        @@
   b:   c7 00 de 07 00 00        movl   $0x7de,(%rax)
  11:   8b 05 00 00 00 00        mov    0x0(%rip),%eax
  17:   89 c7                    mov    %eax,%edi
  19:   e8 00 00 00 00           callq  1e <bar+0x1e>
  1e:   89 05 00 00 00 00        mov    %eax,0x0(%rip)        @@
  24:   90                       nop
  25:   5d                       pop    %rbp
  26:   c3                       retq
```

(a) Fill in the symbol table of foo.o.

Type	Binding	Section/Pseudosection	Name
OBJ	LOCAL	.data	n
OBJ	GLOBAL	.data	p_n
FUNC	GLOBAL	.text	foo

(b) Fill in the symbol table of bar.o.

Type	Binding	Section/Pseudosection	Name
OBJ	GLOBAL	.data	n
OBJ	GLOBAL	COMM	a
OBJ	GLOBAL	UNDEF	p_n
FUNC	GLOBAL	UNDEF	foo

(c) Fill in the relocation entries of foo.o.

Section	Offset	Type	Addend
.text	0x0000000d	R_x86_64_PC32	-4
.text	0x00000026	R_x86_64_PC32	-4
.text	0x0000002e	R_x86_64_PC32	-4

(d) Fill in the relocation entries of bar.o.

Section	Offset	Type	Addend
.text	0x00000007	R_x86_64_PC32	-4
.text	0x00000013	R_x86_64_PC32	-4
.text	0x0000001a	R_x86_64_PC32	-4
.text	0x00000020	R_x86_64_PC32	+4

(e) After relocation and the program is built, what is the relocated form of the 4 instructions tagged with `@@`? Suppose the runtime address of some symbols are decided as below.

foo	0x4004fd
bar	0x4004d6
n (foo.o's .data)	0x601038
n (bar.o's .data)	0x601088
p_n	0x601040
a	0x601080

1. `2a 0b 20 00` (`0x601038 - 0x4004fd - 0x11 = 0x200b2a`)
2. `d6 ff ff ff` (`0x4004fd - (0x4004fd + 0x26) - 0x4 = 0xffffffd5`)
3. `5f 0b 20 00` (`0x601040 - 0x4004dd - 4 = 0x200b5f`)
4. `8e 0b 20 00` (`0x601080 - 0x4004f6 + 4 = 0x200b8e`)