Sergio Garcia Tapia
Computer Systems: A Programmer's Perspective, by Bryant and O'Hallaron
Chapter 7: Linking
May 8, 2024

Practice Problems

Exercise 7.1. This practice problem concerns the `m.o` and `swap.o` modules below:

```c
/* m.c */
void swap();

int buf[2] = {1, 2};

int main()
{
    swap();
    return 0;
}
```

```c
/* swap.c */
extern int buf[];

int *bufp0 = &buf[0];
int *bufp1;

void swap()
{
    int temp;

    bufp1 = &buf[1];
    temp = *bufp0;
    *bufp0 = *bufp1;
    *bufp1 = temp;
}
```

For each symbol defined or referenced in `swap.o`, indicate whether or not it will have a symbol table entry in the `.symtab` section in module `swap.o`. If so, indicate the module that defines the symbol (`swap.o` or `m.o`), the symbol type (local, global, or extern), and the section (`.text`, `.data`, `.bss`, or COMMON) it is assigned to in the module.

Solution: The `buf` symbol references a global symbol defined in `m.o`, so it will have a symbol table entry. The symbol will have an entry in `.symtab`, which has information about functions and global variables defined and referenced in the program. The variable is initialized, so it will be in the `.data` segment.

The `bufp0` symbol is defined in `swap.o`. It a global symbol that can be referenced in other modules because it does not use the `static` keyword. Since it is initialized, it belongs to the `.data` section, used for global and static C variables. It bears an entry on `.symtab`.

The `bufp1` symbol defines a global variable, so it will have entry in `.symtab`. Since it is uninitialized, it will be in the `COMMON` section, and not in the `.bss` section because it is not explicitly initialized to 0. The symbol is defined in `swap.o`.

The `swap` symbol is a nonstatic function so it is global; it bears a symbol on `.symtab`. Since it references a global variable, it will need to be modified later by the linker, so it is in the `.rel .text` section.

Finally, the `temp` variable is a nonstatic local variable managed by the stack, so it will not bear an entry on `.symtab`.

Symbol	`.symtab` entry?	Symbol type	Module where defined	Section
`buf`	Yes	External	`m.o`	`.data`
`bufp0`	Yes	Global	`swap.o`	`.data`
`bufp1`	Yes	Global	`swap.o`	`.bss`
`swap`	Yes	Global	`swap.o`	`.text`
`temp`	No	———	———	———

Exercise 7.2. In this problem, let `REF(x.i)` $\rightarrow$ `DEF(x.k)` denote that the linker will associate an arbitrary reference to symbol `x` in module `i` to the definition of `x` in module `k`. For each example that follows, use this notation to indicate how the linker would resolve references to the multiply-defined symbol in each module. If there is a link-time error (rule 1), write "ERROR". If the linker arbitrarily chooses one of the definitions (rule 3), write "UNKNOWN".

(a)

```c
/* Module 1 */
int main()
{
}

/* Module 2 */
int main;
int p2()
{
}
```

(a) `REF(main.1)` $\rightarrow$ `DEF(`_____`)`

(b) `REF(main.2)` $\rightarrow$ `DEF(`_____`)`

(b)

```c
/* Module 1 */
void main()
{
}

/* Module 2 */
int main = 1;
int p2()
```

```
{
}
```

(a) `REF(main.1)` → `DEF(_____)`

(b) `REF(main.2)` → `DEF(_____)`

(c)

```c
/* Module 1 */
int x;
void main()
{
}

/* Module 2 */
int main = 1;
int p2()
{
}
```

(a) `REF(x.1)` → `DEF(_____)`

(b) `REF(x.2)` → `DEF(_____)`

Solution:

(a) Since the `main` function is a global function, it is a strong symbol. Meanwhile, `main` in the second file is a weak symbol because it is an uninitialized global variable.

   (a) `REF(main.1)` → `DEF(main.1)`

   (b) `REF(main.2)` → `DEF(main.1)`

(b) The `main()` function in module 1 is a global function so it is a strong symbol. The `main` variable in module 2 is an initialized global variable, also a global symbol. Thus, there is a linker error.

(c) Both instances of the `x` symbol are global, but the one in module 1 is a weak symbol because it is uninitialized, whereas the one in module 2 is strong because it is initialized.

   (a) `REF(x.1)` → `DEF(x.2)`

   (b) `REF(x.2)` → `DEF(x.2)`

Exercise 7.3. Let `a` and `b` denote object modules or static libraries in the current directory, and let `a` → `b` denote that `a` depends on `b`, in the sense that `b` defines a symbol that is referenced by `a`. For each of the following scenarios, show the minimal command line (i.e., one with the least number of object file and library arguments) that allow the static linker to resolve all symbol references.

(a) `p.o` → `libx.a`

(b) `p.o` → `libx.a` → `liby.a`

(c) `p.o` → `libx.a` → `liby.a` *and* `liby.a` → `libx.a` → `p.o`

Solution:

(a) The linker always adds an object file to the set of file that will be merged into the executable. Since `p.o` depends on `libx.a`, it must precede it. The command is: `gcc p.o libx.a`

(b) This is similar to before: `gcc p.o libx.a liby.a`

(c) The seemingly circular dependency poses no problem. As explained in (a), when the linker adds any object file to the set of files that will be merged to form the executable. The chain dependency means that `p.o`, `libx.a`, and `lib.y` must follow in that order. The symbols used by `p.o` that are found in the object files concatenated in the `libx.a` static library will be added to the set of files that will be part of the executable object file. Since `liby.a` depends on `libx.a`, we must list `libx.a` again so that the object file containing the symbols referenced in `liby.a` also become part of the executable. We do not have to add `p.o` again because it is an object file, which is already saved in the set of object files by the linker.

The command is: `gcc p.o libx.a liby.a libx.a`

Exercise 7.4. This problem concerns the relocated program in Figure 7.12(a).

(a) What is the hex address of the relocated reference to `sum` in line 5?

(b) What is the hex value of the relocated reference to `sum` in line 5?

Solution:

(a) The instruction on line 5 has address `0x4004de`, which is the address and the instruction there is `e8 05 00 00 00`. As indicated in the annotation on the figure, this corresponds to the `callq 4004e8 <sum>` instruction, where `e8` is the opcode, and `05 00 00 00` is the 32-bit PC-relative address to the `sum` procedure. Since the 1-byte instruction `e8` is at `0x4004de`, the relative reference of `sum` is 1 byte later, at `0x4004df`.

(b) The hex value of the relocated reference to `sum` is `0x5`, which is relative to the instruction following the `callq` instruction.

Exercise 7.5. Consider the call to function `swap` in object file `m.o` (Figure 7.5)

```
9: e8 00 00 00 00     callq e <main+0xe>    swap()
```

with the following relocation entry:

```
r.offset = 0xa
r.symbol = swap
r.type = R_X86_64_PC32
r.addend = -4
```

Now suppose that the linker relocates `.text` in `m.o` to address `0x4004d0` and `swap` to address `0x4004e8`. Then what is the value of the relocated reference to `swap` in the `callq` instruction?

Solution: First we compute the address of the relocated reference:

```
ADDR(s) = ADDR(.text) = 0x4004d0
refaddr =  ADDR(s) + r.offset
       =   0x4004d0 + 0xa
       =   0x4004da
```

Then we update the value of the relocated reference:

```
*refptr = (unsigned) (ADDR(r.symbol) + r.addend - refaddr)
       = (unsigned) (0x4004e8 + (-4) - 0x4004da)
       = 0xa
```

Therefore after relocation the call would be:

```
0x4004d9: e8 0a 00 00 00:    callq 0x4004e8 <swap>
```

Exercise 7.6. This problem concerns the `m.o` module from Figure 7.5:

```
/* m.c */
void swap();

int buf[2] = {1, 2};

int main()
{
    swap();
    return 0;
}
```

and the following version of the `swap.c` function that counts the number of times it has been called:

```
extern int buf[];

int *bufp0 = &buf[0];
static int *bufp1;

static void incr()
{
    static int count = 0;

    count++;
}

void swap()
```

```
{
    int temp;

    incr();
    bufp1 = &buf[1];
    temp = *bufp0;
    *bufp0 = *bufp1;
    *bufp1 = temp;
}
```

For each symbol that is defined and referenced in `swap.o`, indicate if it will have a symbol table entry in the `.symtab` section in module `swap.o`. If so, indicate the module that defines the symbol (`swap.o` or `m.o`), the symbol type (local, global, or extern), and the section (`.text`, `.data`, or `.bss`) it occupies in that module.

Solution:

The `buf` symbol is an external global symbol referenced in the `swap.o` object module, so it will have a symbol in `.symtab`. It is defined in the `main.o` module as a global variable, and since it is initialized, it will occupy the `.data` section.

The `bufp0` symbol is a global symbol defined in the `swap.o` module, so it will have a symbol in `.symtab`. It is initialized, so it belongs to the `.data` section.

`bufp1` is a local symbol defined `swap.o` because it appears in the top-level scope with a `static` modifier. It does have an entry in `.symtab`. It is uninitialized, so it belongs to the `.bss` section.

`swap` is a global symbol defined in`swap.o`, and will have a `.symtab` entry. It is in the `.text` section.

`temp` does not have a symbol table entry because it is local to the `swap` function. Local variables are part of the runtime stack.

`incr` is a local symbol because it is defined at the outer scope with the `static` keyword. It is defined in `swap.o`, will occupy the `.text` section, and will have an entry in `.symtab`.

The `count` variable is declared with `static` inside the local `incr` procedure. Since it is initialized, it is placed in `.bss`, and will have a symbol in `.symtab`.

Symbol	`swap.o` `.symtab` entry?	Symbol type	Module where defined	Section
`buf`	Yes	External	`m.o`	`.data`
`bufp0`	Yes	Global	`swap.o`	`.data`
`bufp1`	Yes	Local	`swap.o`	`.bss`
`swap`	Yes	Global	`swap.o`	`.text`
`temp`	No	———	———	———
`incr`	Yes	Local	`swap.o`	`.text`
`count`	Yes	Local	`swap.o`	`.bss`

Exercise 7.7. Without changing any variable names, modify `bar5.c` on page 683 so that `foo5.c` prints the correct values of `x` and `y` (i.e., the hex representation of integers 15213 and 15212):

```c
/* foo5.c */
#include <stdio.h>
void f(void);

int y = 15212;
int x = 15213;

int main()
{
    f();
    printf("x = 0x%x y = 0x%x \n",
           x, y);
    return 0;
}
```

```c
/* bar5.c */
double x;

void f()
{
    x = -0.0;
}
```

Solution: The problem is that `double x` in `bar5.c` is a weak symbol, it being global and uninitialized, whereas `int x = 15213;` defines a strong symbol. When the linker encounters these, it will prefer the strong definition. Thus when `bar5.c` refers to `x`, it will use the one at the address defined by `foo5.c`. Since `x` and `y` are contiguous, assigning the 8-byte `double` with value `-0.0` to `x` causes the initial 4 bytes of `x` as well as the 4 bytes of `y` to be overwritten.

We cannot make `double x` a strong symbol ion `bar5.c` while it is strong symbol in `foo5.c`, because the linker does not allow this and would output an error. To prevent the problem, we can make `double x` local to `bar5.c` by using the `static` modifier:

```c
/* bar5.c, modified */
static double x;

void f()
{
    x = -0.0;
}
```

Exercise 7.8. In this problem, let `REF(x.y)` $\rightarrow$ `DEF(x.k)` denote that the linker will associate an arbitrary reference to symbol `x` in module `i` to the definition of `x` in module `k`. For each example below, use this notation to indicate how the linker would resolve references to the multiply-defined symbol in each module. If there is a link-time error (rule

1), write “ERROR”. If the linker arbitrarily chooses one of the definitions (rule 3), write “UNKNOWN”.

(a)

```c
/* Module 1 */
int main()
{
}
```

```c
/* Module 2 */
static int main = 1;
int p2()
{
}
```

(a) REF(main.1) → DEF(_____._____)

(b) REF(main.2) → DEF(_____._____)

(b)

```c
/* Module 1 */
int x;
int main()
{
}
```

```c
/* Module 2 */
double x;
int p2()
{
}
```

(a) REF(x.1) → DEF(_____._____)

(b) REF(x.2) → DEF(_____._____)

(c)

```c
/* Module 1 */
int x = 1;
int main()
{
}
```

```c
/* Module 2 */
double x = 1.0;
int p2()
{
}
```

(a) `REF(x.1)` → `DEF(_____._____)`

(b) `REF(x.2)` → `DEF(_____._____)`

Solution:

(a) The `main` symbol is global in module 1; in module 2, a local symbol `main` is defined as well. There is no conflict.

(a) `REF(main.1)` → `DEF(main.1)`

(b) `REF(main.2)` → `DEF(main.2)`

(b) Both modules defined the `x` symbol weakly, so the linker uses rule 3 to choose any of the symbols. It is not a compiler error, but it is ambiguous. (a) `REF(x.1)` → `DEF(UNKNOWN)`

(b) `REF(x.2)` → `DEF(UNKNOWN)`

(c) Both modules define `x` as a strong symbol, which leads to a compiler error.

Exercise 7.9. Consider the following program, which consists of two object modules:

```c
/* foo6.c */
void p2(void);

int main()
{
    p2();
    return 0;
}

/* bar6.c */
#include <stdio.h>

char main;

void p2()
{
    printf("0x%x\n", main);
}
```

When this program is compiled and executed on an x86-64 Linux system, it prints the string `0x48`
`n` and terminates normally, even though `p2` never initializes variable `main`. Can you explain this?

Solution: At compile time, the compiler exports the `main` global symbol in `foo6.o` to the assembler as a strong symbol because it is a function. Meanwhile, it exports the `main` global symbol in `bar6.o` as a weak global symbol because it is an uninitialized variable. By rule 2 on page 680, when `main` is referenced, the strong symbol is chosen. Therefore, even though the uninitialized variable `char main` would typically be assigned `0` (the value of the null byte) which has hex representation `0x0`, the reference in the `printf` in `bar6.c` actually uses the address of `main` from `foo6.c`.

Exercise 7.10. Let `a` and `b` denote object modules or static libraries in the current directory, and let `a` $\rightarrow$ `b` denote that `a` depends on `b`, in the sense that `b` defines a symbol that is referenced by `a`. For each of the following scenarios, show the minimal command line (i.e., one with the least number of object file and library arguments) that will allow the static linker to resolve all symbolic references:

(a) `p.o` $\rightarrow$ `libx.a` $\rightarrow$ `p.o`

(b) `p.o` $\rightarrow$ `libx.a` *and* `liby.a` $\rightarrow$ `libx.a`

(c) `p.o` $\rightarrow$ `libx.a` $\rightarrow$ `liby.a` *and* `libz.a` $\rightarrow$ `libx.a` $\rightarrow$ `libz.a`

Solution:

(a) The line is: `gcc p.o libx.a`. When the linker encounters `p.o`, it notices that it is an object file, and adds it to the set of relocatable object files that will be merged to form the executable. Since `p.o` depends on `libx.a`, we list it afterwards so that the linker can resolve these symbols. We do not have to list `p.o` again.

(b) The line is: `gcc p.o liby.a libx.a`. Since both `p.o` and `liby.a` depend on symbols from `libx.a`, they should precede it.

(c) A constraint is that `libx.a` must precede `liby.a` in the provided list of static libraries. We must list `libz.a` twice; once before `libx.a` so that it can reference symbols from `libx.a` that the linker will extract from `libx.a` when it reaches it during its scan. At that point, the symbols will be resolved, and `libx.a` will too will make references to symbols defined in `libz.a`, which as extracted from `libz.a` later when it is encountered during the scan. Thus the line is: `gcc p.o libz.a libx.a libz.a liby.a`.

Exercise 7.11. The program header in Figure 7.14 indicates that the data segment occupies `0x230` bytes in memory. However, only the first `0x228` bytes of these come from the executable file. What causes this discrepancy?

Solution: I think it's because `main.c`, the routine from which the program header table was created, has the declaration:

```c
int sum(int *a, int n);
```

which is a reference to an external symbol. It's a pointer to a function, and it is uninitialized, so it belongs to the `bss` segment. This does not occupy disk space in the object file. Once it has been resolved, this contains the absolute address of the procedure defined in the `sum.o` module.

Exercise 7.12. Consider the call to function `swap` in object file `m.o` (Problem 7.6).

```
9: e8 00 00 00 00 callq e <main+0xe> swap()
```

with the following relocation entry:

```
r.offset = 0xa
r.symbol = swap
r.type = R_X86_64_PC32
r.addend = -4
```

(a) Suppose that the linker relocates `.text` in `m.o` to address `0x4004e0` and `swap` to address `0x4004f8`. Then what is the value of the relocated reference to `swap` in the `callq` instruction?

(b) Suppose that the linker relocates `.text` in `m.o` to address `0x4004d0` and `swap` to `0x400500`. Then what is the value of the relocated reference to `swap` in the `callq` instruction?

Solution:

(a) First we need compute the address of the reference:

```
/* .text section address for m.o */
ADDR(s) =  ADDR(.text) = 0x4004e0
refaddr =  ADDR(s) + r.offset
        =  0x4004e0 + 0xa
        =  0x4004ea
```

Then we use this to update the reference in `m.o`:

```
/* Address of swap.o */
ADDR(r.symbol) = ADDR(swap) = 0x4004f8

*refptr =  (unsigned) (ADDR(r.symbol) + r.addend - refaddr)
        =  (unsigned) (0x4004e8     + (-4)     - 0x4004ea)
        =  (unsigned) 0x2
```

The value of the relocated referenced to `swap` in `callq` is thus `0x2`.

(b) Like before, we first compute the address of the reference:

```
    /* .text section address for m.o */
ADDR(s) =  ADDR(.text) = 0x4004d0
    refaddr =  ADDR(s) + r.offset
    =   0x4004d0 + 0xa
    =   0x4004da
```

Then we use this to update the reference in m.o:

```
/* Address of swap.o */
ADDR(r.symbol) = ADDR(swap) = 0x400500

*refptr =  (unsigned) (ADDR(r.symbol) + r.addend - refaddr)
       =  (unsigned) (0x400500     + (-4)     - 0x4004da)
       =  (unsigned) 0xa
```

The value of the relocated reference is 0xa.

Exercise 7.13. Performing the following tasks will help you become more familiar with the various tools for manipulating object files.

(a) How many object files are contained in the versions of `libc.a` and `libm.a` on your system?

(b) Does `gcc -Og` produce different executable code than `gcc -Og -g`?

(c) What shared libraries does the `gcc` driver on your system use?

Solution:

(a) The `ar` command creates static libraries, and inserts, deletes, lists, and extracts members. In this case, the members are the relocatable object files. Running `ar --help` reveals that the `-t` option displays the contents of the archive. To find `libc.a` on my system, I ran the `find` command, and I used the `-exec` option to run `ar -t`. I then piped it to `wc -l` which prints the number lines that it reads in `stdin`:

```
find /usr/lib -type f -name ``*libc.a'' -exec ar -t \{\} \textbackslash{};
    | wc -l
```

Notice there is no newline in the actual command. The result was 2055. The same approach did not immediately work for `libm.a`, which on my system is not an `ar` archive, but rather an ASCII text file with the following content:

```
/* GNU ld script
*/
OUTPUT\_FORMAT(elf64-x86-64)
GROUP ( /usr/lib/x86\_64-linux-gnu/libm-2.35.a
    /usr/lib/x86\_64-linux-gnu/libmvec.a )
```

Thus I instead ran `arr` on both of those files. `libm-2.35a` had 795 members, and `libmvec.a` had 548 members.

(b) Yes, it is different. The following information is in the `gcc` man page:

```
To tell GCC to emit extra information for use by a debugger, in almost all
    cases you need only to add -g
to your other options.

GCC allows you to use -g with -O. The shortcuts taken by optimized code may
    occasionally be surprising:
some variables you declared may not exist at all; flow of control may
    briefly move where you did not
expect it; some statements may not be executed because they compute
    constant results or their values are
already at hand; some statements may execute in different places because
    they have been moved out of
loops. Nevertheless it is possible to debug optimized output. This makes it
    reasonable to use the
optimizer for programs that might have bugs.

If you are not using some other optimization option, consider using -Og
    with -g. With no -O option at
all, some compiler passes that collect information useful for debugging do
    not run at all, so that -Og may
result in a better debugging experience.
```

(c) First I ran `whereis` to find the `gcc` executable on my system. Then I ran `ldd` to list shared libraries that it needs at runtime:

```
whereis gcc
# gcc: /usr/bin/gcc /usr/lib/gcc /usr/share/gcc /usr/share/man/man1/gcc.1.gz
ldd /usr/bin/gcc
```

The output was:

```
    linux-vdso.so.1 (0x00007ffff1578000)
libc.so.6 => /lib/x86\_64-linux-gnu/libc.so.6 (0x000074ec41000000)
/lib64/ld-linux-x86-64.so.2 (0x000074ec412c5000)
```

According to the man page *vdso(7)*, "The "vDSO" (virtual dynamic shared object) is a small shared library that the kernel automatically maps into the address space of all user-space applications." On the other hand, `libc.so.6` is the shared object for the standard C library, and `ld-linux-x86-64.so.2` is the shared object for the dynamic linker.