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)

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

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

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

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

(b)

```
/* 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` $\rightarrow$ `libx.a` $\rightarrow$ `liby.a`

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