L a b 7 : Linking

Date: Oct 21, 2021

This document first describes the aims of this lab. It then provides necessary background. It then describes the exercises which need to be performed.

0.1 Aims

The aim of this lab is to familiarize you with the use of linkers in C. After completing this lab, you should be familiar with the following topics:

- Recognizing linker error messages during the course of compilation.
- The process used by a linker to resolve symbols.
- The use of libraries and the differences between static and dynamic libraries.

0.2 Background

There are 3 possible types of object files:

Relocatable object file This type of file contains code and data but will need to be combined with other object files and libraries to create an executable object file. Usually, this kind of object file has extension `.o` and is produced using `gcc`'s `-c` option.

Executable object file Contains code and data for a complete program (except shared libraries) which is ready to be loaded into memory and executed.

Shared object file A type of relocatable object file which is loaded into memory and linked dynamically either at load time or at run time.

Recall that building an executable program involves the following steps:

1. Compiling all the `.c` files into `.o` files.
2. Linking the `.o` files along with any needed libraries to produce an executable file.

A library is a collection of object files. When a library is linked into a program, only those object files which define symbols required by the program are actually linked into the program.

There are two kinds of libraries:

Static Libraries All object modules needed from the library are linked into the executable at link time.

Dynamic Libraries The object modules needed from the library are not linked into the program until it is loaded into memory or even later. Also known as *shared libraries* as the library code can be shared simultaneously by multiple processes by being loaded (at possibly different virtual addresses) into the virtual address spaces of the processes.

0.3 Exercises

0.3.1 Starting up

Follow the *provided directions* for starting up this lab in a new git `lab7` branch and a new `submit/lab7` directory. You should have copied over the contents of `~/cs220/labs/lab7/exercises` over to your directory.

0.3.2 Exercise 1: Recognizing and Fixing a Linker Error

Change over to the log10 directory and look at the log10.c program. It contain code to read doubles from stdin and print out their base-10 logarithms. Once you have looked at the source code, build the `log10` executable by typing `make`.

You should receive a linker error saying there is an `undefined reference to log10`. The problem is that the `log10()` function is defined in the math library, but the provided `Makefile` does not link in the math library.

Do a `man` on the `log10` function. You will see a requirement that you link with the math library using `-lm`. When you get such a link error you should look at the man page for the problematic function to figure out which library you may be missing.

Now add the `-lm` flag to the `LDLIBS` `make` variable and re-`make`. This time it should build correctly. You can test it by simply typing `echo 10 1000 2 | ./log10`.

Notice that in addition to the `log10()` function, the program also uses the `printf()` and `scanf()` functions. The reason you did not get an error for these functions is that they are defined in the standard C library `libc` which is linked in by default.

Where are these libraries located? Traditionally, they were in `/usr/lib`. However, since modern systems may need to support multiple architectures (like 64-bit and 32-versions of x86), the version of Linux you are running has them in `/usr/lib/x86_64-linux-gnu` All the library file names start with the prefix `lib` and have extension either `.a` for a static library and `.so` (standing for **shared object**) for a dynamic library. Specifying the linker option `-lXXX` means to link with the library with name `libXXX`.

Do a `ls -l /usr/lib/x86_64-linux-gnu/libm.*` which will list all files in `/usr/lib/x86_64-linux-gnu/` which begin with the prefix `libm.`. You should see two: `libm.a` and `libm.so`; the former is a GNU linker script referring to the static math library and the latter is a script referring to the dynamic math library.

Look inside the `libm.a` script; you should see a reference to a specific version `LIBM_VERSION` of the actual `libm` library.

List out all the symbols from that specific version of the `libm` static library using `nm /usr/lib/x86_64-linux-gnu/LIBM_VERSION >libm.nm 2>/dev/null` where `LIBM_VERSION` is the `libm-N.NN.a` from the `libm.a` script (the `2>` redirects `stderr` to a bit-sink). Look at the generated output in `libm.nm` using a text editor. You should see a definition for `log10` listed under the object file `w_log10.o` (search for `w_log10.o` using your editor), with `__log10` defined in as a code symbol (aka **`text`** symbol) using `T`, and `log10` defined as a *weak symbol* using `W` (a *weak symbol* can be overridden by a non-weak definition). You should see the other symbols referenced by that object file but not defined within that object file as undefined `U`.

Do a `man` on `nm` (if it does not work on your local system pull a man page off the web). Using the information documenting the symbol types, look at the `nm` output in `libm.nm` to discover which object file defines the `__ieee754_log10` symbol referenced by the `w_log10.o` object file.

0.3.3 Exercise 2: Multiply-Defined Symbols

Change over to the directory multiple-symbols. Look at the files contained there; note that `sym` is defined differently in def1.c and def2.c. If you type `make`, you will get a multiply-defined symbol error for `sym`.

The linker classifies all external identifiers which have initializers as *strong symbols* and allows only a single strong definition in a program. A declaration without an initializer declares *weak symbols.* Multiple declarations for the same *weak symbol* are merged together; when weak symbols are linked with a strong symbol with the same spelling, the strong symbol definition wins.

You can fix the error by changing one of the definitions to be a *weak* symbol by removing the initializer.

0.3.4 Exercise 3: Multiple-Definition Bug

Change over to the directory multiple-defs. Observe that `x` is declared with inconsistent types in main.c and f.c. The definition in `main.c` is a strong definition and dominates over that in `f.c`.

Build the program by typing `make`. You may receive a link warning. If you ignore the warning and run the program you will see that the inconsistent types for `x` has caused a pernicious bug: `f()` changing `x` to `0.0` also happens to change the value of `y`!! Can you understand why? (Hint: a **double** occupies 8 bytes).

These kind of bugs can be avoided by putting the declaration of a symbol referenced by multiple program files into a **single** header file and `#include`'ing the header file into all files which reference or **define** the symbol.

0.3.5 Exercise 4: Dynamic versus Static Linking

Change over to the static-dynamic directory, the contained log10.c program is identical to that from the first exercise. However, the .<./exercises/static-dynamic/Makefile> Makefile is setup to build both a statically-linked and dynamically-linked executable. Build them by typing simply `make`. You should see `make` building a statically-linked `log10-static` executable and a dynamically-linked `log10-dynamic` executable.

Do a `ls -l`. You should see a dramatic difference in size between `log10-static` and `log10-dynamic`. That is because `log10-static` contains within it all the library code needed, whereas for `log10-dynamic`, the library code is linked in dynamically (at load-time or later).

Do a `nm` on both executables: `nm log10-static >log10-static.nm` and `nm log10-dynamic >log10-dynamic.nm`. Look at both output files using a text editor. In `log10-dynamic.nm` you should see that `log10` is undefined `U`, but in `log10-static.nm` you should see it is defined as a weak ('W') symbol.

0.3.6 Exercise 5: Building a Non-Standard Library

This exercise involves building static and dynamic versions of a custom library to add/multiply vectors (based on the example from the recommended text, ch. 7). Change over to the libvec directory and look at the two files addvec.c and multvec.c (which will be put into the library) and a test file testvec.c which will be linked with the library.

Then specifically look at the Makefile which is reproduced here with the lines numbered to facilitate discussion:

```
01 CFLAGS = -g -Wall -fPIC -std=c18
02
03 OBJS = \
04   addvec.o \
05   multvec.o
06
07 all: libvec.so  libvec.a testvec-static testvec-dynamic
08
```

```
09 libvec.so: $(OBJS)
10 $(CC) -shared $(OBJS) -o $@
11
12 libvec.a: $(OBJS)
13 ar rcs $@ $(OBJS)
14
15 testvec-static: testvec.o
16 $(CC)  -static testvec.o -L. -lvec -o $@
17
18 testvec-dynamic: testvec.o
19 $(CC) testvec.o -L. -lvec -o $@
20
21 .PHONY: clean
22 clean:
23 rm *.o *.so *.a testvec-*
24
```

Line 1 defines the options used for compilation. The one option which may not have been seen earlier is `-fPIC` which specifies generating Position-Independence Code. This is usually necessary when generating shared libraries which can be simultaneously loaded at different addresses in the virtual address spaces of multiple processes. `-g` turns on debugging, `-Wall` turns on reasonable warnings and `-std=c18` specifies the C dialect as C18.

Lines 3 - 5 specify the objects included in the libraries.

Line 7 lists all the targets to be built.

Lines 9 and 10 specify how to build the dynamic library `libvec.so`. The `-shared` option builds a shared library.

Line 12 and 13 specify how to build the static library `libvec.a`. The program `ar` is used to archive the object files together: `r` specifies inserting the object files into the archive with replacement, `c` creates the archive and `s` creates a symbol-table in the archive to facilitate searching (this can also be done using a special `ranlib` command).

Lines 15 and 16 specify how to build the test program using the static library. The `-static` option specifies that no dynamic libraries should be used, `-L .` says to add the current directory to the set of directories in which libraries are searched for, the `-lvec` option specifies the `libvec.a` library.

Lines 18 and 19 specify how to build the test program using the dynamic library. Since no `-static` option is used, it will use dynamic libraries and link with `libvec.so` in the current directory.

Build all the targets by typing `make`. Once again, do a `ls -l` listing to observe the significant difference in size between the statically-linked executable and the dynamically-linked executable.

Now run the statically-linked executable by typing a command like `./testvec-static 1 2 3` which should print out the sum and product of the vector `[1, 2, 3]` with itself.

Try the same thing with the dynamically-linked executable by typing a command like `./testvec-dynamic 1 2 3`. You will get an error saying that it cannot find the dynamic library `libvec.so`. This proves that this library is necessary to run the program.

You will need to tell the system to add the current directory to the set of directories which are searched for libraries when running the program. One way to do so is to add the current directory to the `LD_LIBRARY_PATH` environmental variable. If using a `sh`-based shell (your shell prompt will usually contain a `$` character), simply type `LD_LIBRARY_PATH=. ./testvec-dynamic 1 2 3` which should work. If using a `csh`-based shell (your shell prompt contains a `%` character) you will need 2 commands:

```
% setenv LD_LIBRARY_PATH .
% ./testvec-dynamic 1 2 3
```

You can list out the dynamic dependencies of a dynamically-linked executable by using the `ldd` command. Type `ldd testvec-dynamic` to see what libraries the `testvec-dynamic` executable depends on. Set up the `ldd` command so that it does not print `libvec.so` as `not found`.

0.3.7 Exercise 6: Symbol Definitions using nm

Stay in the same libvec directory as the previous exercise.

Produce a `nm` dump of the symbols in the statically-linked executable by running `nm testvec-static >testvec-static.nm`. Look at `testvec-static.nm` using a text editor and find the definitions for the symbols `main`, `addvec` and `multvec`. Then run `gdb` on that executable and print out the address of `main`, `addvec` and `multvec` using `p &main`, `p &addvec` and `p &multvec`. You should see that the addresses match the values in the `nm` dump.

Do the same thing with the dynamically-linked executable: specifically, produce a `nm` dump of `testvec-dynamic` using `nm testvec-dynamic >testvec-¬ dynamic.nm` and use a text editor to find the definitions for the symbols `main`, `addvec` and `multvec`. You should see that `main` is defined but `addvec` and `multvec` are not yet defined; they can only be defined once the dynamic library `libvec.so` has been loaded. Run `gdb` on the dynamic executable using `gdb testvec-dynamic`. Before you start the program print out the address of `main`, `addvec` and `multvec`. The address of `main` should print out fine, but the other two should print out only partial information.

Before you can run the program inside `gdb` for the dynamically-linked executable, you will need to set the `LD_LIBRARY_PATH` using `set env LD_LIBRARY¬`

`_PATH .` at the `gdb` prompt. Then put a breakpoint at `main()` using `b main` and run the program using `r 1 2 3`. When the program stops at `main()`, you should once again print out the addresses of `main`, `addvec` and `multvec`. You will now see that the latter two are defined and are in much higher memory than `main`, proving that they are in a different memory area (or segment) from the text segment containing `main`.

0.3.8 Exercise 7: Building Your Own Libraries

Change over to the libgeom directory. It contains a specification file geom.h for routines which calculate the perimeter and area of circles and rectangles and implementation files circ.c and rect.c. It also contains a crude interactive test program testgeom.c.

Create a `Makefile` which

1. Will build a static library `libgeom.a` containing the `circ.o` and `rect.o` object files.
2. Will build a dynamic library `libgeom.so` containing the `circ.o` and `rect.o` object files.
3. Will build a statically-linked executable `testgeom-static` which contains the test program linked with the static library.
4. Will build a dynamically-linked executable `testgeom-dynamic` which contains the test program linked with the dynamic library.

You can use the `libvec` Makefile as a starting point.

Build all of the above targets by running `make` on your `Makefile`. Test your static and dynamic executables.

0.4 References

Bryant and O'Halloran, recommended text, chapter 7.