Lab 06 - Linking and Loading

Out: October 20-26, 2014

1 Introduction

Up until now, Makefiles have been to you like dogs are to babies: they pet them and like them but they don't know how they work. In this lab, you will learn how to write makefiles and reinforce the concepts of linking and loading. Linking is the process by which an executable file is created from object files and libraries. The linker is the program that does this. The job of the linker is to arrange the program's code and data in address space, and maybe modify references as necessary.

2 Makefiles

2.1 Intro

Makefiles are files that hold detailed instructions to be executed by the system. You can execute instructions in a makefile by running `make` in the same directory as the makefile. If you have several makefiles, then you can execute them with the command: `make -f MyMakefile`. For more info, type `man make` in a department machine.

2.2 Writing a Makefile

2.2.1 Basic Makefile

A basic makefile is composed of:

```
target: dependencies
[tab] system command
```

NOTE: You *must* use tabs here. In other words, if you are using a text editor such as vim and choose to record your tabs as spaces, then the Makefile will not work correctly.

To compile by hand in terminal, you would usually type

```
gcc -Wall -Wunused -Wextra -std=c99 life.c -o life
```

Typing this every time you want to recompile can be tedious. Also, as projects get larger and involve more files, makefiles become very helpful.

To run the above command, you would create a file named *Makefile* and write

```
all:
	gcc -Wall -Wunused -Wextra -std=c99 life.c -o life
```

On this first example we see that our target is called `all`. This is the default target for makefiles. The make utility will execute this target if no other targer is specified. We also see that there are no dependencies for target `all`, so `make` safely executes the system commands specified. Finally, `make` compiles the program according to the command line we gave it.

2.2.2 Using Dependencies

Sometimes, it is useful to have different targets. These different targets can build different parts of your project, or build it in different ways, or do something else entirely. Generally, Makefiles contain a "clean" target that removes temporary files and the output files of the other targets. Example:

```
all: life hello

life: life.c
        gcc -Wall -Wunused -Wextra -std=c99 life.c -o life

hello: hello.c
        gcc -Wall -Wunused -Wextra -std=c99 hello.c -o hello

clean:
        rm -f hello life
```

Now we see that the target `all` has only dependencies, but no system commands. This target will, in turn, call the targets `hello` and `life`. Each of the dependencies are searched through all the targets available and executed if found. In this example we see a target called `clean`. It is useful to have such target if you want to have a fast way to get rid of all the object files and executables.

2.2.3 Variables and Comments

You can also use variables and comments when writing Makefiles:

```
#This is a comment.
#CC, CFLAGS and EXECS are variables.
CC = gcc
CFLAGS = -Wall -Wunused -Wextra -std=c99
EXECS = life hello
all: $(EXECS)

life: life.c
        $(CC)  life.c $(CFLAGS) -o life

hello: hello.c
        $(CC) $(CFLAGS) hello.c -o hello

clean:
        rm -f $(EXECS)
```

2.2.4 Automatic Variables

Finally, we will briefly discuss automatic variables. These are variables that are defined by each target rule. Some helpful automatic variables are listed below. For a full list, view make documentation.

- $@: The file name of the target of the rule.
- $<: The name of the first prerequisite.
- $^: The names of all the prerequisites, with spaces between them.

Example:

```
#This is a comment.
CC = gcc
CFLAGS= -Wall -Wunused -Wextra -std=c99
EXECS = life hello
all: $(EXECS)

life: life.c
        $(CC) $< $(CFLAGS) -o $@

hello: hello.c world.c
        $(CC) $(CFLAGS) $^ -o $@

clean:
        rm -f $(EXECS)
```

3 Linking and Loading

Several things happen between source code and a running program:

1. Preprocessor: handles preprocessor directives like define and include
2. Compiler: compiles the resulting .c files into a .o files
3. Linker: combines the multiple .o files into a single executable, resolve references and relocate modules as necessary
4. Loader: loads the program into memory and resolve references to shared libraries

As you might have guessed, linking and loading are parts 3 and 4.

3.1 Static Linking

This is the type of linking performed on static libraries (.a). Upon linking, the object code of these libraries is copied into your program executable, and references are resolved.

3.2 Dynamic Linking

This is the type of linking performed on dynamic shared object libraries (.so). In this case, the linker simply places a reference to the library inside your executable during compilation. At runtime, your executable and the shared libraries are placed in memory and the loader resolves and maps references appropriately.

This lab will give you some first-hand experience with these concepts. The lecture slides have more information on these, so check them out!

4 Assignment

You have been put in charge of overseeing the events of the IBR (International Baby Race). Baby Evan and Baby Advik, officials of the IBR, wrote some code to perform various administrative tasks, but while they're being babies and napping, you've realized that there is more to be done!

This assignment has three sections: warmup, versioning, and interpositioning. The stencil contains a directory for each section.

4.1 Warmup

Baby Cody and Baby Dylan were neck and neck in the most recent event. The judges think Baby Cody came first and Baby Dylan came second, but they're just not sure. As is common in the world of Baby Racing, they agreed to use photo-finish technology. As it turns out, Dylan made it across the line just ahead!

Thankfully, Evan and Advik wrote some neat code to swap the first two elements of an array. What you need to do is compile them together, and explain to the judges (your TAs) what you've done based on the instructions below.

4.1.1 Description

This part of the handout contains the following files:

- *main.c*: a C file which calls our `swap()` function.
- *swap.c*: a C file defining our `swap()` function.

4.1.2 Task

Here's what you need to do:

1. Take a quick look at *main.c* and *swap.c* and understand what's happening. The comments should help.
2. Compile *main.c* and *swap.c* separately into their object files, using `gcc -c main.c` and `gcc -c swap.c`

3. Dissassemble *main.o* using `objdump -d main.o`. Note down anything interesting you see, and try to reason why this might be happening. **Hint:** Your exprerience with byte representation of assembly code from Buffer will be useful here. **Biggger Hint:** Does `main()` know anything about `swap()` yet?
4. Dissassemble *swap.o* similarly. Again, note down anything interesting and try to justify why this might be happening. **Hint:** Does `swap()` really know anything about `buf`?
5. Compile your object files together into a single executable using (exactly) `gcc -o myprog main.o swap.o`. **Note:** This calls the linker and loader internally.
6. Dissassemble *myprog* using `objdump -d myprog`. Compare this result with your observations from previous parts. Is this what you expected? As usual, try to find a justification for this. **Warning:** This dissassembly is much longer than the two above, but all you need to worry about are the dissassembled sections of `main()` and `swap()`.

4.1.3 Checkpoint

Once you think you understand what's happening, call a TA over and talk to them about it! Hopefully, this process helped clarify some simple concepts of static linking.

4.2 Versioning

Another program involved delivering messages to babies with the location of the upcoming race. However, each race has a different location. Instead of writing a new program for each new race, we'll use versioning.

This can be accomplished by loading different versions of a library to the file system. When a program is linked using the corresponding library, different version of that program can coexist.

4.2.1 Description

This part of the handout contains the following files:

- *delivermsg.c*: a C file which calls our `deliver_message()` function.
- *message.h*: a header file declaring our `deliver_message()` function.
- *message1.c*: a C file defining the first version of `deliver_message()` function.
- *message2.c*: a C file defining the second version of `deliver_message()` function.
- *Makefile*: an empty Makefile where you will define rules to make this versioning possible

In this part of the lab, you will write a Makefile that will build two different libraries from provided header and c files and link them to compile two versions of the same program.
Specifically, your Makefile must contain at least the following rules:

- `all`: This should create *delivermsg1*, *delivermsg2*, *libmessage.so*, *libmessage.so.1*, and *libmessage.so.2* in the current directory.
- `clean`: This should remove all the files that were made my the Makefile.

This task can be done using only two rules, but those rules will have a lot of overlapping commands. You should write other rules for these common commands.

4.2.2 System Calls

As we saw on lecture, typing the following on terminal will effectively create the two versions of the program and library:

```
$ gcc -fPIC -c myputs1.c
$ ld -shared -soname libmyputs.so.1 -o libmyputs.so.1 myputs1.o
$ ln -s libmyputs.so.1 libmyputs.so
$ gcc -o prog1 prog.c -L. -lmyputs -Wl,-rpath .
$ gcc -fPIC -c myputs2.c
$ ld -shared -soname libmyputs.so.2 -o libmyputs.so.2 myputs2.o
$ rm -f libmyputs.so
$ ln -s libmyputs.so.2 libmyputs.so
$ gcc -o prog2 prog.c -L. -lmyputs -Wl,-rpath .
```

Explanation:

- The `ld` command invokes the loader directly, rather than through `gcc`. The `-soname` flag tells the loader to include in the shared object its name. which is the string following the flag *libmyputs.so.1* in the first call to `ld`. The `-lmyputs` flag tells the compiler to look for functions inside a library called `libmyputs.so`.
- The `ln -s` command creates a new name (its last argument) in the file system that refers to the same file as that referred to by ln's next-to-last argument.
- The `-Wl,-rpath` flag tells the compiler where to look for the library. The example tells the compiler to look in ”`.`”, the current working directory.”
- The call to `rm` removes the name *libmyputs.so* (but not the file it refers to, which is still referred to by *libmyputs.so.1.*
- And then we make it again, but associate it to *libmyputs.so.2*, the second version of library.

4.2.3 Testing

Run `./delivermsg1` and `./delivermsg2` from your current directory. Both messages should be different.

4.3 Interpositioning

Recently, it has been discovered that some contestants in the tournament have been entering negative numbers for their positions, which is clearly not a good thing. Unfortunately, it is too much effort to rewrite and recompile the systems, so you are tasked with wrapping the `atoi` function to take care of this problem.

In this section, your assignment is to write a function that wraps the standard library function `atoi`. If the input string contains non-digit characters, or would result in a negative number being returned, then your code should print a message to `stderr` indicating such before returning the result of `atoi`.

4.3.1 Testing

Stencil code is located in *atoi.c*. To compile this, you would run the command `gcc -Wl,--wrap=atoi atoi.c -o atoi` , which would generate an executable named `atoi`. You need to write the Makefile and compile running `make`.

5 Getting Checked Off

Once you've completed the lab, go to lab hours and call a TA over to get checked off. You can also get labs checked off at TA hours on Thursday, Fridays, Saturdays, and Sundays until 5pm. Lab questions will only be taken on hours Thursday-Saturday each week. Remember to read the course missive for information about course requirements and policies regarding labs and assignments.