Lab 06 - Linking and Loading

Out: Oct. 26 - Nov. 1, 2015

1 Introduction

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 and Targets

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. As projects get larger and involve more files, makefiles become very helpful because they allow you to use a single command to build an entire project. A basic makefile is composed of:

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

A *target* is a label that denotes a specific task or set of commands to run. The target and set of commands is sometimes referred to as a *rule*. To run a specific target, you run the command `make <target name>` in a shell.

To create a very basic Makefile for our `gcc` command above, you would create a file named *Makefile* and write

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

In 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 target is specified. Therefore, running `make` in a shell would create an executable called `life` using the `gcc` command given.

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 (e.g. executables) of the other targets.

NOTE: You *must* use tabs for indentation in your Makefiles. 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.

2.2.2 Using Dependencies

A *dependency* (sometimes called a prerequisite) is either a file name or the name of another target upon which this target depends. If it is a file name, `make` will only execute the target's commands if the file has changed since the last `make`. If it is a target name, `make` will run the dependency target first and then run the commands in this target. Any target can have multiple dependencies. Furthermore, if a dependency does not exist, then `make` will raise an error.

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 life hello
```

We see that the target `all` has only dependencies, but no commands. This target will, in turn, call the targets `life` and `hello`. The `clean` target removes the output (i.e. the executables `life` and `hello`) of the `life` and `hello` targets.

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`'s documentation.

- $@: The name of the target.
- $<: The name of the first dependency.
- $^: The names of all the dependencies, 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 .o files
3. Linker: combines the multiple .o files into a single executable, resolves references and relocates modules as necessary
4. Loader: loads the program into memory and resolves references to shared libraries

As you might have guessed, linking and loading are parts 3 and 4. To clarify, a *library* is a bundle of object code containing reusable functions that can be incorporated into multiple software systems. For example, when you include the `string.h` header in one of your C files, you are implicitly asking for the `glibc` string library to be included in your program's binary.

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.

4 Assignment

You have been put in charge of overseeing the Li'l Sebastian memorial foot race at this year's Harvest Fest. Jerry wrote some code to perform various administrative tasks, but while he's been busy making art (why does he even try), 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

Chris and Andy were neck and neck in the most recent race. The judges think Chris came first and Andy came second, but they're just not sure. They agreed to use photo-finish technology. Which requires some C array manipulation.

Thankfully, Jerry 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. Disassemble *main.o* using `objdump -d main.o`. Note down anything interesting you see, and try to reason why this might be happening. **Hint:** Your experience with byte representation of assembly code from Buffer will be useful here. **Biggger Hint:** Does `main()` know anything about `swap()` yet?
4. Disassemble *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 `gcc -o myprog main.o swap.o`. **Note:** This calls the linker and loader internally.
6. Disassemble *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 disassembled 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 Ann 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.

When you have a large software project (like the Linux kernel), shared functionality is separated into distinct libraries. This allows a library to be updated without altering the entire project. We can use the linker to build multiple versions of the project by including different versions of the library.

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 our `deliver_message()` function.
- *message2.c*: a C file defining the second version of our `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 by 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

Typing the following **example commands** on a terminal would effectively create two versions of a hypothetical `prog` program and `myputs` 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 `-fPIC` flag tells the compiler to generate "position independent code". This means that the object that `gcc` outputs can be loaded into memory at an arbitrary location and it should still work! Such objects don't make use of absolute addresses for procedure calls and data because such addresses will only be valid if the object were loaded at exactly the right memory location every time. PIC gets around this by making use of the offsets between data items/functions (to get an exact description of how this all works check out the textbook's description of the "procedure linkage table" and the "global offset table").
- The `-shared` flag tells the linker that it should generate a shared object/library. These are the files with .so extensions and the cool thing about them is that they can be loaded into memory either at load time (you'll have to have compiled a program with the shared object beforehand) or at runtime. What this means is that the logic of your C program can decide whether to load a particular library with functions like `dlopen` (for more info on these run `man dlopen` in the shell). After doing so, the program will have access to new functions/data that it didn't have access to before.
- The `-Wl,` flag tells `gcc` that the option immediately following the comma should be passed to the linker (which is basically `ld` called for your convenience by `gcc`!). The `-rpath` flag tells the linker where to look for the library – in this case, ".", 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. The 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.

Interpositioning is when we "wrap" a function (usually a system function) with another function that adds functionality and invokes the original, "real" function. This is particularly useful for

performing error checking in a uniform way. Instead of placing an if statement which checks the return value of a function and prints an error if necessary all over your code, we can wrap the function once with some error-handling code.

When you tell `gcc` to wrap a function (using the `-Wl,--wrap=<function name>` flag), it will generate two functions called `__real_<function name>` and `__wrap_<function name>`. The `__real` function calls the original function you want to wrap. For example, if we are wrapping `atoi`, `__real_atoi` will call the standard library implementation of `atoi`. You must define the `__wrap` function, which must call the `__real` function somewhere in its execution. Again, if we are wrapping `atoi`, you must define the function `__wrap_atoi`, which should both invoke `__real_atoi` and perform whatever additional functionality is desired.

4.3.1 Your Task

In this section, your assignment is to write a function that error checks the standard library function `atoi`. We will use interpositioning to accomplish this. Specifically, 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`.

4.3.2 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. Remember to read the course missive for information about course requirements and policies regarding labs and assignments.