Introduction to Linux: Day 1 lab

July 15, 2024

1 Introduction

In this lab we will learn the basics of compiling C programs (using the command line) with multiple translation units and we will learn to automate this process using Makefiles. We will also build libraries (static and dynamic) and link them to our final binary. Finally, we will learn the basics of analysing our binary (both statically and dynamically) and we'll have a look at gdb (GNU Debugger) and how to debug programs.

2 Building a simple C program

2.1 GNU Compiler Collection (gcc)

To start, we will first compile a simple C program and run it. Got to directory **1**. you will find the following file: main.c which is a simple hello world program.

```
#include <stdio.h>
int main() {
    // printf() displays the string inside quotation
    printf("Hello, World!");
    return 0;
}
```

To compile this program, run the following command:

```
$ gcc main.c -o main
```

This command invokes the gcc compiler and tells it to compile the **main.c** program into a **main** binary. running file on the output produces the following:
$ file main
main: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=740c79556db86b61b66075d38279b70d88c8b596, for GNU/Linux 3.2.0, not stripped

The **file** command provides information about files. Running **file** on elf files can give us information about the architecture of the binary, and whether or not it has dependencies to shared libraries.

2.2 Adding a library

Switch to directory 2. you will find the same main.c file, and another file called math.c. this is a c file that has 3 functions: add, multiply and factorial. We can start by compiling this the conventional way as part of the program. run the following command to do so:

```
$ gcc main.c math.c -o math
```

The output will be a file called math. Run the following command and check the result.

```
$ readelf -s math
```

The **readelf -s** command enables us to see the symbol table of the binary. Symbols are basically function calls and variables. the compiler would use these symbols to associate the names to the memory addresses.
In our program we used the multiply function to print out the result of multiplying 5 by 6. Suppose another program wanted to use the add function to print out the result of adding 5 and 6. They can either implement their own add function, or copy our code and build it with their program as we did. A third option is for us to build our math.c code into a library. That way, other programs can use our library without the need for the source code.

2.2.1 Static Libraries

To build a static library, we will need to invoke the ar utility which is used to create and manipulate archives (which is what a static library is). Run the following command to build.

```
$ gcc -c math.c
$ ar rcs libmath.a math.o

$ gcc main.c -o main -L. -lmath
```

the -L. option specifies the path where the linker would look for libraries (which is the current directory, thus the "."). the -lmath is telling the compiler to look for libraries called libmath.a so. Every library name should start with the **lib** prefix.

2.2.2 Dynamic/shared libraries

building a shared library involves more steps than the static library. These are the commands you'll need.

```
$ gcc -c -fPIC math.c
$ gcc -shared -o libmath.so math.o
$ gcc main.c -o main -L. -lmath
```

The -fPIC option stands for position Independent code. Which enables the same code to be shared amongst multiple processes.
the -shared option is used to create a shared object so that it can be dynamically linked with other executables at runtime.
If we now execute the main program, it will give us this error.

./main: error while loading shared libraries: libmath.so: cannot open shared object file: No such file or directory

This error is due to the fact that the os could not find the shared library that we depend on (libmath.so) and that is because the os has default paths where it looks for shared libraries which we can find using this command:

```
$ sudo ldconfig -v 2>/dev/null | grep -v ^$ | grep -E "^/|:" | awk -F':' '{print $1}'
```

To make our library visible to the operating system we need to change an environment variable (LD_LIBRARY_PATH) to include the directory where our library exists.

```
$ export LD_LIBRARY_PATH=$(pwd)
```

Running the program now yields the desired result.

2.2.3 Makefile

Using the command line to build binaries can become a tedious task, especially when project get bigger and include multiple files.
To automate this process we will use a Makefile. This is a file that the gnu make utility uses to execute certain commands based on the targets provided.
go to directory 3. you'll find the same source code files along with a Makefile.

```
CC=gcc
LIBS_DYN= -lmath_dyn
LIBS_STATIC= -lmath
LDFLAGS_DYN=-L. $(LIBS_DYN)
LDFLAGS_STATIC= -L. $(LIBS_STATIC)
AR=ar



all: main_static main_dynamic

main_dynamic: main.o libmath_dyn.so
	$(CC) -o $@ $< $(LDFLAGS_DYN)
main_static: main.o libmath.a
	$(CC) -o $@ $< $(LDFLAGS_STATIC)

%.o: %.c
	$(CC) -c -o $@ $<
%_dyn.o: %.c
	$(CC) -c -fPIC -o $@ $<
lib%_dyn.so: %_dyn.o
	$(CC) -shared -o $@ $<
lib%.a: %.o
	$(AR) rcs $@ $<
clean:
	rm -f *.a *.so *.o main
```

To build the main program with both static and dynamic libraries, simply run:

```
$ make
```

3 Debugging

now that we had a brief idea about how programs are made and how processes work. We will turn to debugging programs that have software bugs.

3.1 Valgrind

go to directory 4. You will find a source code file leak.c. build it using gcc and add -g3 to include debug symbols and then run it. building leak.c will not result in any compilation errors yet upon inspecting the source code it is clear that a memory leak is present (malloc without free).

Run valgrind on this program by invoking the following command :

```
$ valgrind --leak-check=full ./leak
```

Valgrind tells us that there is a memory leak in the main function in leak.c line 9. This is the call for malloc.

3.2 Core dumps

Now let's try another program, this time, it is a program that dereferences a NULL pointer, which will result in a segmentation fault.
compile it with the same -g3 option and then run it. We get this error:
segmentation fault (core dumped) ./segfault Segmentation faults are very common especially in low level languages like C and C++. a good way to debug these kinds of error is using the core dump which is a snapshot of the program the moment it crashed. This is produced automatically by

the operating system when a program crashes.
By default core dumps are not enabled. to enable them run the following commands:

```
$ ulimit -c unlimited
$ sudo sysctl -w kernel.core_pattern='%e.core'
```

Rerun the program and this time you will notice a file segfault.core.xxxxx next to our executable. To examine the core dump, run gdb on your binary with the core dump file like so:

```
$ gdb ./segfault segfault.core.xxxxxx
```

gdb will now bring you to the exact line where the program crashed. you can then use the different gdb commands to examine what is happening with your binary.

3.3 strace

Up until now we have only debugged programs in C. It is not always the case that we get to debug programs written in a language we're familiar with or that we have the source code to. This example is a python script that tries to open a file that does not exist. we will try to debug it using strace which is a tool that enables us to view all the system calls by a certain program and their results. Since every program in our operating system uses system calls to talk to the kernel regardless of the programming language it's written in. It is always helpful to use strace to figure out what's happening exactly.

Run the following commmand on the python script

```
$ strace python3 non_existing_file.py
```

This will show us the list of system calls the program executes including:
openat(AT_FDCWD, "nonexistent_file.txt", O_RDONLY—O_CLOEXEC) = -1 ENOENT (No such file or directory)

This indicates that the openat system calls failed and the error number is ENONT (No such file or directory).

Notice that we were able to figure out the problem without having to check out the code. This is particularly useful when we don't know where the problem is coming from and we would use strace and other tools to tell what is happening so that we can go back to the code and try to solve the problem.

4 Good luck!

Finally, I hope you found this lab useful and I wish you good luck in your professional careers :) !