CS232L Operating Systems Lab
Lab 08: Static and Dynamic Libraries

CS Program
Habib University

Fall 2024

1 Introduction

In this lab you will learn how to:

1. How to build static and dynamic libraries in Linux

2 C Programming Workflow

The basic flow of writing and executing a C program is [2]. The linker part combines different object files and links their code to different libraries. Introduction to these libraries will be the focus of this lab [3].

Let's start with an example. Compile a simple program into an executable (let's say a.out) and then run the command `ldd` on it, i.e.:

```
# ldd a.out
```

The `ldd` command tells you which dynamic libraries this executable will link to during its execution. It will output a bunch of lines containing library names.

Libraries come in two flavours: `static` whose file extension is `.a`, and `dynamic` - with `.so` file extensions.

3 What is a Library

A library is nothing more than a bunch of function definitions compiled and grouped together in a single file in their binary form. Usually, along with the binary implementation, a library also provides a bunch of .h files which will contain the declarations for those functions. We include the .h files in our source files in which we call those library functions. At compile time we tell the compiler to go find the definitions of those called function in that library by providing the library binary path preceded by the correct compiler flags. The C standard library is linked with every program by default.

4 Static libraries

When linking with static libraries, when making the executable, the linker will copy the code for all the used library functions direclty inside the final executable. Static library files conventionally have .a file extensions.

Let us say we want to make a static library of arithmetic functions. The codes in listings 1, 2, 3, and 4 very simple codes for two functions `add()` and `sub()` that will be part of our library.

```
1 #ifndef ADD_H
2 #define ADD_H
3
4 int add (int a, int b);
```

```
5
6 #endif
```

Listing 1: add.h

```
1 #include "add.h"
2
3 int add (int a, int b){
4   return a+b;
5 }
```

Listing 2: add.c

```
1 #ifndef SUB_H
2 #define SUB_H
3
4 int sub (int a, int b);
5
6 #endif
```

Listing 3: sub.h

```
1 #include "sub.h"
2
3 int sub (int a, int b){
4   return a-b;
5 }
```

Listing 4: sub.c

The following commands will generate the `add.o` and `sub.o` object files from the respective source files:

```
# gcc -Wall -c add.c
# gcc -Wall -c sub.c
```

The `ar` command will archive, i.e. combine, the multiple object files into one static library file titled `libmylibrary.a`:

```
# ar rs libmylibrary.a add.o sub.o
```

Next, we write the `main.c` file which uses the library function as shown in Listing 5.

```
1 #include<stdio.h>
2 #include "add.h"
3 #include "sub.h"
4
5 int main (int argc, char* argv[]){
6
7   printf ("Sum of 2 and 5 is %d\n", add(2,5));
8   printf ("Diff of 2 and 5 is %d\n", sub(2,5));
9
10  return 0;
11 }
```

Listing 5: main.c

The command will generate the `main.o` object file:

```
# gcc -Wall -c main.c
```

Finally, we link our program, i.e. `main.o`, to the static library `libmylibrary.a`:

```
# gcc -Wall -o a.out main.o libmylibrary.a
```

Now, the executable `a.out` is created which contains the code for library functions `add()` and `sub()`. We can run it to verify that the library functions are actually being called.

An alternative way to write the last command would be:

```
# gcc -Wall -o a.out -L . main.o -lmylibrary
```

The `-L` flag tells the linker the paths where to look for the library files, the current directory in this case, and the `-l` flag gives the name of the library to search for.

5 Dynamic libraries

One issue with static libraries is that linking will copy the code for library functions inside the executable files. This causes two problems: the executable sizes are quite large and, secondly, the same code is duplicated many times in different executables. Dynamic libraries provide a solution to that.

With dynamic libaries, during the linking, the code for library functions is not copied inside the executable, rather a *pointer*[1] to that function is stored. When the program runs it will load the dynamic library in RAM and during execution the function calls will jump to the function code inside the library. Subsequent programs that use the same library, when run, will not load the library but rather detect that the library is already loaded in RAM and just link to the already loaded library, i.e., on calls to library function they will jump to the functions in the already loaded library in memory. This way the same library is shared between multiple programs. The dynamic libraries in Linux end in `.so` extensions, meaning that they are *shared objects*.

Let us redo the steps in the last section to create a dynamic library. To create a dynamic libraries, when creating the object files, we will pass the `-fPIC` flag to `gcc` to generate *position independent code*.

```
# gcc -Wall -fPIC -c add.c
# gcc -Wall -fPIC -c sub.c
```

Next we will create our shared library `mylib.so`:

```
# gcc -shared -o mylib.so add.o sub.o
```

As opposed to static libraries, just building dynamic libaries is not enough; we will aslo have to *install* it. The simplest way to do it will be to copy the library file to the `/usr/lib` folder and run the `ldconfig` command (both these operations require `sudo` access):

```
# cp mylib.so /usr/lib
# ldconfig
```

Now we can compile our `main.c` and link it to our dynamic library:

```
# gcc -Wall -c main.c
# gcc -Wall -o a.out main.o mylib.so
```

[2]

The `a.out` can be run now and it will link with `mylib.so` dynamically. We can also verify this by running the command:

```
# ldd a.out
```

You can refer to the link below [4] for more information about dynamic libraries on Linux.

6 On Windows

In Windows, the library file extensions and the process is slightly different but the main concepts remain the same. Google is your friend here.

7 Exercise

Convert the codes that you wrote for the linked list in lab04 into functions and expose them in the form of a dynamically linked library.

Submit the .h files, .c files, Makefile and the PDF.

---

[1] pointer here means enough information to locate the function inside the library file at program run time.

[2] An alternative syntax for the last command could be:

```
# gcc -Wall -o a.out main.o -lmylib
```

But to use with the `-l` option, the library should be named `libmylib.so`, i.e., the name should start with the prefix `lib`.

References

[1] Raymond Eric S. *"Basics of Unix Philosophy" The Art of Unix Programming.* Addison-Wesley, Professional.

[2] BTYANT, O'HALLARON. *Computer Systems, A Programmer's Perspective.* Pearson.

[3] https://cs-fundamentals.com/c-programming/static-and-dynamic-linking-in-c.

[4] https://tldp.org/HOWTO/Program-Library-HOWTO/shared-libraries.html.