Libraries and Compilation Environment (II)

Computadors – Grau en Ciència i Enginyeria de Dades – 2021-2022 Q2

Facultat d'Informàtica de Barcelona

Compilers are the tools we use to generate binary executables, as well as other code files like object files and libraries. Besides, executables can be diferent depending on the compiler optimizations, as discussed in the Theory lectures. We attach some source codes to perform the following exercises. Use the following link to download:

https://docencia.ac.upc.edu/FIB/GCED/COM/documents/Lab/S6/FilesS6.tar.gz

In this lab session we will get experience with modular implementation, compilation steps, compare optimizations, learn how to deal wtih libraries, and statically and dynamically link executables.

Compiler command line options

Compilers usually have a large number of command line options. Let's see the ones that are most useful in the GCC C/C++ compilers:

Getting help

```
--help              # simple help
--help  --verbose   # help of the compiler driver and sub-processes. Most useful!!
```

Common options

```
-S          # generate assembly only ('S' with capital letter)
-c          # generate object file(s) only
-o <name>   # name the output file - can be assembly (-S), object (-c) or the executable
```

Code generation options *(https://eli.thegreenplace.net/2012/01/03/understanding-the-x64-code-models)*

```
-fpic       # generate position independent code – small mode.
            # needed to build shared libraries
-fPIC       # generate position independent code – large mode. For shared libraries
```

Optimization options *(https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html)*

```
-O          # equivalent to  -O1
-O0         # do not optimize. Sometimes useful for debugging
-O1         # optimization level 1: basic optimizations that do not take compilation time
            #    merges constants, moves loop invariants…
-O2         # adds some more expensive optimizations:
            #    code alignment, partial inlining, switch conversion, store merging…
-O3         # adds even more expensive optimizations:
            #    inline functions, loop vectorize, loop distribution, loop interchange…
```

Debug support options

-g	# generates debug information, useful for gdb, ddd, etc.

Linking options

-L<path>	# adds path to the list of directories where to find libraries for linking
-l<name>	# adds lib<name>.so for shared linking, and/or lib<name>.a for static linking
-shared	# generates a shared library, instead of a binary executable
-static	# generates a statically linked binary executable

Compiling programs and libraries

In the attached file (FilesS6.tar.gz) there are two folders. In the first set of exercises we are going to deal with “fibonacci” folder.

Exercise 1

Copy the “fibonacci-orig.c” to ***“fibonacci.c”****. Modify the code as you consider to put the “fibonacci(unsigned long)” function in an external file, such as* ***“fib.c”****, and include the respective* ***“fib.h”*** *header file. Finally, modify the* ***Makefile*** *to include the new rule (called “fibonacci”) to perform such new compilation.*

Exercise 2

Modify the ***Makefile*** *to include the new rules to create the object files (rule called “fibonacci.o”) and the assembly version (rule called “assembly”) of both, the original implementation as well as the modified version from Exercise 1. You should obtain output files with “.o” and “.s” extensions, respectively. Use a text editor to take a look at the assemble files (“*.s”) and briefly identify function names, variables, and the similarities with the example shown in the Theory slides. It is not necessary to fully understand the contents, just to briefly relate these contents with the ones shown in the slides.*

Exercise 3

There is a command called “nm” to present the symbol table of a compiled file (use the “man” command to see further details). Use a text editor to create and edit a file called ***“answers.txt”*** *to briefly explain your findings when you use “nm” with the object files and the executable. We strongly suggest to use the object files obtained in the previous exercises. In your findings, just focus on: 1) identify the symbols in the object filles that you are familiar with, from the source code file; 2) compare the symbol table with the one obtained from the executable; 3) understand the type of the symbol, according to the second column (just a letter) from the output of “nm” (use “man” to understand the meaning of that letter).*

Create Static library

Now we are going to create a static library (with the extension “.a”). This library will comprise the object file of fib.c. Remember that you can capsulate multiple object files into a single library. To do this you have to use the “ar” command. In this particular case you have to execute the following command line:

```
#>ar –csr libCOMstatic.a fib.o
```

You can also use this “ar” command to extract object files from a given library file. We suggest you read the “-x” and “-p” options of “ar” in the man pages.

This command line will create the static library library libCOMstatic.a. The flags indicate: (c) create the library; (s) create an index of the files inside the library; and (r) to add files to the library. In this particular case, we reuse the object files created in the previous exercise. Now let’s create a program with this library instead of referencing the object files.

Execute *“gcc -o **fib-liba.exe** fibonacci.c -L. -lCOMstatic”*
or *“gcc -o **fib-liba.exe** fibonacci.c libCOMstatic.a”*
If you execute the new executable should work as expected and thus it means the library has been correctly created and can be properly used.

Create a Shared Library

Now we are gonna create a shared library (with the extension “.so”). Remember that by default the compiler assumes to use shared libraries and, thus, dynamic linking. This library will comprise the object file of fib.c. To do this you have to use the “gcc” compiler with the following:

1) Create again the object file from the fib.c, but adding the “-fpic” flag (check the introduction of this flag at the beginning of this document).
2) Execute the command line “gcc -o libCOMdyn.so -shared fib.o”
3) Compile a new version of the program, called **fib-libso.exe**, but using the newly created library. In this case, you have to introduce the flag “-L.” to indicate the compiler the current working directory as the path to find the library. Remember to also introduce the flag “-lCOMdyn” to link this particular library **at the end** of the command line.

Do not execute **fib-libso.exe** yet. Let’s try to double check the shared objects the program depends on: execute the command “ldd” (***NOTE:*** *read the man of this command to find out how to use it*). It will display the libraries that are dynamically linked at run-time with fibdyn.exe. We suggest you to use the “-v” flag, to obtain also the information about the libraries dependences and versions needed. Unfortunately, you will see from the output, the new library “libCOMdyn.so” is not found. In fact, if you try to run **fib-libso.exe**, it doesn’t work. The problem is that the environment variable “LD_LIBRARY_PATH” (the one used to find shared libraries to be dynamically loaded in alternative directories in addition to the standard ones) does not include the path of our library. To solve this issue, you have to update this environment variable by adding “.” path into this variable (***NOTE:*** *check the notes of the second lab session – Linux Operating System Environment - to see the details on the environment variables*). Afterwards, execute again the binary and it should work.

Statically vs Dynamically linked

Compile again the binaries, but adding the “-static” flag in the compilation command line, include the static library, and add the suffix “static” at the end of the name of every new binary. Remember that, by default, binaries are compiled using dynamically linking.

Exercise 4

*Write in the **“answers.txt”** file the difference of the static versus non-static linked binaries. Focus the comparison on differences of: the file size (using “ls –l”) and the output of the symbol table using the “nm” command. In the latter, pay special attention on: 1) the symbol of the fibonacci function; 2) the symbols obtained from the static compiled binary (readdress the output to a given text file and do a brief analysis, specially about why there are functions that now appear.*

Optimization Flags

There is another folder in the attached file, called “matrix”. Inside, you will find a Makefile and a source code, named “matmul.c”. This code is an evolution of the “matinit.c” from the previous Lab session. In this case, we have included the required code to do a matrix multiplication. You can modify the size and the type of the matrix at the beginning of the source code, as we presented in the previous session. The Makefile builds four different executables, depending on the optimization flag that has been used.

Even though we have not explained the nomenclature of assembly (and it is beyond the scope of this course), if you are curious to see what are the differences in the code, you can use the “objdump” command, with the option “-d” to extract the assembly code from the program (redirect the output to a text file to be reviewed using a text editor).

Exercise 5

*Write in the **“answers.txt”** file the difference in size of the binaries (using “ls –l”) and the time shown in the execution of every single version of the matmul compilation. Relate the differences with the goal of every optimization level described in the short summary at the beginning of this document.*

Upload the Deliverable

When done with the lab session, to save the changes you can use the tar command as follows:

```
#tar  czvf  session6.tar.gz   answers.txt Makefile fibonacci.c fib.c fib.h
```

Now go to RACO and upload this recently created file to the corresponding session slot.