Libraries and Compilation Environment (I)

Computadors – Grau en Ciència i Enginyeria de Dades – 2019-2020 Q2

Facultat d'Informàtica de Barcelona – Departament d'Arquitectura de Computadors

Applications use to comprise multiple source code files that implement parts of the program following modular development approach. Thus, developers combine object files developed by themselves and libraries developed from third parties, that include most of the common functionalities that applications need. Libraries are part of the compilation environment, where the compiler has the main role to transform the source code provided onto a binary executable file that implements a specific application.

During this lab session, we will work with the creation of an executable and understand the components and the main parts of their internals. Among others, the structure of the binary files created by the compiler, the typical locations of the libraries and header files, and the way programs are built by the compiler. We will use a simple example program as a basis. Then, you will have to implement few codes and generate object files, a library, and executables, as well as use commands to analyse their internals.

The basis of source code for this session

The file FileS5.tar.gz comprises a source code that will be the basis of the exercises. The code requests to the user to introduce an option. Depending on that, few additional functions are executed. Actually the source code needs to be extended by you as follows:

Exercise 1

Implement few functions into two different moduls: one for simple arithmeticae operations (arith.cc and header file) and another for comparison based functions (compare.cc and header file). Please, take into account the aim of this session is not to enhance developing skills, but the understanding behind the use of object files. Thus, implement just two simple functions each.

Compilation steps

The most simple way to compile a program is:

```
$ g++  -o  <executable>   sources.cc  …
```

By default, if no `-o` option is given, a `a.out` file is generated: `$ g++   sources.cc …`

A common way of compiling applications is doing it in two steps:

Compilation phase:

```
$ g++ -c sources.cc …
```

Each source file generates a `.o` object file.

Linking step:

```
$ g++ -o <executable> sources.o …
```

Usually, the stdc++, math, and c libraries are automatically added into the linking process, so all symbols from them get automatically resolved.

Common compilation options

- `-c`, compile to object file
- `-o filename`, compile to generate the output called filename
- `-S`, compile to assembly file
- `-S -o filename.s`, compile to assembly file and give the output the name filename.s
- `-I<dir>`, while pre-processing, search for header files in <dir>
- `-L<dir>`, while linking, search for library files in <dir>
- `-l<name>`, link against the lib<name>.so or lib<name>.a library
    - `-lc   -lm   -lstdc++   -lX11   -lXaw` …

- See the full GCC compiler options using:
    - `g++ --help --verbose | less`

Exercise 2

Update the Makefile based on the new files in order to create three object files, that is, program.o, arith.o, and compare.o).

Format of Object File

Use the command "objdump" to check the contents of the object file as follows:

$ objdump <options> <file>

(hint: use the -h option to see the full list of sections, and their properties)

(hint: use the -s option to see the full contents of all sections)

(hint: use the -p option to see the private headers, and the libraries needed to complete the loading of these binary files)

(hint: use the -r option to see the rellocation information)

(hint: use the -t option to see the symbol table)

(hint: use the -C to demangle (decode) low-level names)

You can combine these data with the output of another command called "nm" to show information about the symbol table.

Exercise 3

Analyse the difference for both relocation information and symbol table sections among the three files you have previously created. Write your findings into a new file, called "answers.txt" you have to edit.

Basic system libraries

In our OS, there are several hundreds of library files, let's see a few of them:

- libc.so, system calls and basic C support functions (files, FILE, processes, sockets…)

- pthread.so, basic POSIX thread support
- libm.so, Math functions (sqrt, log, exp, trigonometric functions…)
- libstd++, basic C++ support functions (vector, list, deque, iostreams…)
- libX11.so, libXaw.so, graphics support (X-Windows…)
- libgomp.so, support for OpenMP parallelism, usually from GCC
- libmpi.so, support for Message Passing Interface (MPI)
- …

Libraries usually reside in the /lib or /usr/lib directories. When systems support both 32- and 64-bit applications, they can also be resident in /lib32, /usr/lib32, or /lib64, /usr/lib64. In modern OS versions they can also be under an specific "machine-architecture" directory:

/lib/x86_64-linux-gnu/	/usr/lib/x86_64-linux-gnu/
/lib/i386-linux-gnu/	/usr/lib/i386-linux-gnu/

For locally installed services and applications, libraries can also be in directories like /usr/local/lib or /opt/lib.

Libraries are binary files, and the compiler cannot understand them. Only the linker can use them to link with the symbols they contain. For this reason, we also need a set of "include" – or *header* - files, that are written in a source language (C, C++), and that the compiler understands. They provide the type definitions, constants, and external symbol definitions contained in the binary libraries. These header files usually are installed in the directory:

/usr/include

For header files there is a single version, supporting both 32- and 64-bit installations. The reason is that header files come prepared for both types of compilation environments, and sometimes more. The C/C++ preprocessors implements this feature:

```
#if defined(__x86_64__)  // __i386__   __powerpc__  __powerpc64__ __arm__ __arm64__
…
#endif
```

Exercise 4

Count the number of shared object files (.so) present in your system library directories:

/lib	*/usr/lib*
/lib64	*/usr/lib64*

(hint: use `wc -l` to count the number of lines in a text output)

(hint: use `find <directory> -name \*.so -print` to search for all files ending in .so in a directory hierarchy, including subdirectories)

Write your answers in the "answers.txt" file.

Exercise 5

Update the Makefile, if required, to perform the final linking step of the compilation pipeline, using the Makefile, to create the executable. This executable has to combine the object files previously created as well as the required libraries you are using. That is, at least the default library that is always linked. You don't need to include any additional flag to include the default library. However, you have to include "–lxxx" to link the library called "libxxx" if you are using non-default libraries.

Let's create two versions of the executable. One with default linking (i.e. dynamic linking) and another with static linking. The latter has to be created with the flag "- - shared" in the compilation command line. Name both executables in a different way to manage them independently.

Exercise 6

Analyse the differences of both executables and write your findings in the "answers.txt" file. You can focus on:

- *The size of the file*
- *Repeat the comparison of exercise 3 (relocation information and symbol table)*
- *Execute the command "ldd" (check the "man" for more information) that shows the shared libraries that use the programs*

Exercise 7

Modify the Makefile to include debugging information in object files and executable (hint: check the "–g" flag of the compiler). Rename the object files and executables in such a way you can compare afterwards the files with debug information against the files without debug information. Finally, analyse the contents with objdump, as well as the file size and write your findings in the "answers.txt".

Exercise 8

Use the command objdump with the "-d" flag to disassemble the code of the program. Redirect the ouput to a new file, such as "program.asm" and check it afterwards.

Upload the Deliverable

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

```
#tar czvf session5.tar.gz answers.txt Makefile *.cc *.h
```

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