Essentials for Scientific Computing: Source Code, Compilation and Libraries Day 8

Ershaad Ahamed
TUE-CMS, JNCASR

May 2012

1 Introduction

In the first session we discussed instructions that the CPU processes in order to perform some function. These instruction are encoded in a binary format that is specific to the processor and is called *machine language*. Writing even simple programs in machine language can be extremely difficult and error prone. Most programs are written in a high level language like Java, C, C++, FORTRAN, among many others. These programs are then processed by a program called a *compiler* which, in several stages, converts it into hardware specific machine language. This process is called *compilation*. High level languages have a structure (syntax) that is more easily comprehended by people. High level languages are also independent of the specific hardware on which the final program will run. This frees programmers from the constraint of writing programs for particular hardware, and allows them to concentrate on the actual functionality or logic of the program. There are on some occasions, good reason to program in a hardware specific language. This may be to take advantage of some special capability of the hardware or to optimise the performance of some section of a program. This is usually done in *assembly language*, which is text based and specific to the hardware family, and can be converted into machine language by a program called an *assembler*. The use of assembly language by a programmer is usually limited to small, performance critical parts of programs that are written in a high level language.

2 Source Code to Executable

The code written by the programmer in a high level language like C or FORTRAN is called the *source code*. At the end of compilation, the resulting program is called an *executable*. The executable contains machine language instructions in addition to operating system specific code.

The compilation of a source code into an executable takes place in 4 stages, Preprocessing, Compilation, Assembly and linking.

Let us consider the following C source code example before we discuss concepts further.

```
#include <stdio.h>

#define AVALUE 5

int main()
{
        float a = AVALUE;
        float b = 25;

        // calculate and print arithmetic operations
        printf("%f\n", a * b);
        printf("%f\n", a + b);

        return 0;
}
```

To compile this program into an executable we simply run the GNU C Compiler (GCC) as follows.

```
gcc -o test test.c
```

This tells GCC to compile the program and to name the output executable `test`. The output is in an executable format called Executable and Linkable Format (ELF), which is the standard executable file format for Linux. Running the program with

```
./test
```

produces the output.

```
125.000000
30.000000
-20.000000
```

In the next sections, we will follow each stage of the compilation process.

2.1 Preprocessing

In the source code of our program above, the lines that begin with a `#` are preprocessor directives. Preprocessor directives are processed by the *preprocessor* and are never seen by the compiler. The preprocessor performs text substitution, macro expansion, comment removal and file inclusion. The first preprocessor directive in our program above is `#include <stdio.h>` which instructs the preprocessor to include the header file `stdio.h` into our source. The next preprocessor directive `#define AVALUE 5`, defines a text substitution. Once the preprocessor has completed processing our source code, every occurrence of the string `AVALUE` is substituted by the string `5`.

We can instruct GCC to stop processing the source code after preprocessing and to send the preprocessed file to stdout.

```
gcc -E test.c > test.txt
```

The output `test.txt` has 860 lines. This is because the preprocessor includes the contents of the header file `stdio.h`, which in turn includes several other header files. When it does this it adds lines which contain line numbers and source information, which makes it possible to track where a line in the preprocessed output originated from. The part of the output, we are interested in is towards the end and looks like.

```
int main()
{
        float a = 5;
        float b = 25;

        printf("%f\n", a * b);
        printf("%f\n", a + b);
        printf("%f\n", a - b);
        return 0;
}
```

Notice that comments have been removed and text substitution has been done.

2.2 Compilation to Assembly Language

This part of the compilation process converts the preprocessed source code into assembly language. We can see the output using.

```
gcc -S test.c
```

Below is a snippet of the output `test.s`.

```
movl $.LC2, %eax
movq %rax, %rdi
movl $1, %eax
call printf
movss -4(%rbp), %xmm0
addss -8(%rbp), %xmm0
unpcklps %xmm0, %xmm0
cvtps2pd %xmm0, %xmm0
movl $.LC2, %eax
movq %rax, %rdi
movl $1, %eax
call printf
movss -4(%rbp), %xmm0
subss -8(%rbp), %xmm0
unpcklps %xmm0, %xmm0
cvtps2pd %xmm0, %xmm0
movl $.LC2, %eax
movq %rax, %rdi
movl $1, %eax
call printf
movl $0, %eax
leave
ret
```

2.3 Assembly

This stage of the compilation process assembles the assembly language program generated into a machine language object file.

2.4 Linking

In the linking stage the object file is made into an executable after resolving dependencies with other object files and after adding startup and termination code. When writing a program, the source code for various parts or functions of the program may be defined in separate source files. The source code in one file may call functions which are defined in other files. Since each source file is compiled into a separate object file, the object file produced from one source file may have a dependency on another object file. The function of the linker is to merge these object files so that any such dependencies are resolved.

3 Libraries

3.1 Static Libraries

Since one object file can call functions which are defined in another object file, this provides a convenient way to group common code. For example, consider that a source file is created which contains the definition for some mathematical function that we need. An object file is created from this source. We can now write several separate programs that call this mathematical function when they require it but dont need define the function themselves. After each of these programs is compiled into an object file, we can resolve the dependency they have on the said mathematical function, by linking it with the object file that we created in the beginning and which has the definition of the mathematical function. In this way we can maintain a set of commonly used functions as object files that are ready to be linked with any new programs that needs them. This also eases the compilation and distribution of programs, since the common object files need to be compiled only once and the source files for these common objects need not be distributed along with programs that utilize them. Several such object files can be archived together into *static libraries*.

We will create a small static library as a example to illustrate the linking process. First our main program `test_link.c`.

```
#include <stdio.h>

float cubed(float a);
float powerfour(float a);

int main()
{
        float x = 23.1415;
        float y = cubed(x);
        printf("%f\n", y);
        y = powerfour(x);
        printf("%f\n", y);
```

```
        return 0;
}
```

Notice that the function `cubed` and `powerfour()` are declared but not defined. The definition of `cubed()` and `powerfour()` will be placed in the next two files `libcubed.c` and `libpowerfour.c`. Below is the listing of `libcubed.c`.

```
float cubed(float a)
{
        return a*a*a;
}
```

And `libpowerfour.c`.

```
float powerfour(float a)
{
        return a*a*a*a;
}
```

To compile `libcubed.c` and `libpowerfour.c` as an object files, pass the `-c` option to gcc. This stops GCC from linking the object file into an executable and produces the object files `libcubed.o` and `libpowerfour.o`.

```
gcc -c libcubed.c libpowerfour.c
```

We will now combine `libcubed.o` and `libpowerfour.o` into a single static library using the `ar` command.

```
ar rs libmymath.a libcubed.o libpowerfour.o
```

Let us try compiling the main program `test_link.c` without specifying the library object.

```
gcc -o test_link test_link.c
```

The output error will be.

```
/tmp/ccuJraJ2.o: In function ‘main’:
test_link.c:(.text+0x16): undefined reference to ‘cubed’
test_link.c:(.text+0x3f): undefined reference to ‘powerfour’
collect2: ld returned 1 exit status
```

Now include the library file.

```
gcc -o test_link test_link.c libmymath.a
```

We could also list the individual object files during compilation. But could get inconvenient when the number or object file increases.

```
gcc -o test_link test_link.c libcubed.o libpowerfour.o
```

This gives us a working executable which can be run using

```
./test_link
```

3.2 Dynamic Libraries

As we described earlier, including static libraries while compiling programs actually merges the object files during the linking stage to produce a complete executable. There can be disadvantages to this. If there are several library files or they are very large, every executable that is generated by linking with these libraries will have a copy of the library code embedded in the executable. This makes the executable large. And when an executable is run, the executable file is actually mapped into main memory before it is executed. This means it also occupies an unnecessarily large portion of memory. The solution to this is *dynamic libraries*.

When a program containing calls to undefined functions is compiled, references to the called functions are made. Unlike static libraries, where these references are resolved during the linking stage, in dynamic libraries, these references are resolved during run time. This means that the dynamic libraries must actually be present during program execution. This makes the executables smaller since they only contain references to and not the actual code of the library functions. Also, only one copy of the code of a dynamic library needs to be maintained in main memory, and the operating system makes sure that every running process that refers to this library has its references resolved at run time.

The process of compiling the functions in the example above as a dynamic library consists of two steps. First, generate "position independent code" object files with.

```
gcc -fPIC -c libcubed.c libpowerfour.c
```

Then combine the object files into a dynamic library.

```
gcc -shared -Wl,-soname,libmymath.so -o libmymath.so libcubed.o \
libpowerfour.o
```

The `-shared` option tells gcc to produce a shared object file.
`-Wl,-soname,libmymath.so` is a linker option that sets the internal name of the library to `libmymath.so`.
And `-o libmymath.so` sets the name of the output dynamic library file to `libmymath.so`.

Now, to link our main program to the dynamic libraries, we must compile it with.

```
gcc -o test_link test_link.c libmymath.so
```

On running the executable, instead of getting the expected output, we get the error.

```
./test_link: error while loading shared libraries: libmymath.so:
cannot open shared object file: No such file or directory
```

When we run an executable that is dynamically linked, the Linux dynamic linker resolves dependencies before executing the program. We get the error above because the dynamic linker is unable to find the dynamic library `libmymath.so` in order to resolve dependencies. The dynamic linker searches through several predefined directories in order to find libraries during the dynamic linking stage.

We can fix the problem by telling the dynamic linker to look for libraries in the directory where we have stored `libmymath.so`. We do this by adding it to the environment variable `LD_LIBRARY_PATH` which is a colon separated list of directores to search for libaries in addition to the predefined ones. Since we are running the executable from the current working directory, we simply add the `.` directory to the environment variable with.

```
export LD_LIBRARY_PATH=.:$LD_LIBRARY_PATH
```

This prepends `.` to `LD_LIBRARY_PATH`, if it exists. If not `$LD_LIBRARY_PATH` expands to the empty string `""`. Now running the executable with `./test_link` works as expected. It is better to specify an absolute pathname in `LD_LIBRARY_PATH`, like `\home\ershaad\libs` for instance, since relative pathnames will stop working when we change working directories.

3.3 Using `nm` to Resolve Dependencies

During compilation of a program's source, we may get `undefined reference` errors (as seen earlier) due to libraries not being included during the compilation step. When a large number of dynamic libraries are involved, it may be difficult to determine which library file to include in order to resolve a dependency on a particular function. The `nm` command lists the names of symbols present in an object file. If we had a number of object files, and needed to determine which file defines a particular function `funcname`, we would use.

```
nm --print-file-name *.so | grep funcname
```

or

```
nm --print-file-name *.a | grep funcname
```

and would give output similar to

```
libmymath.so:000000000000058c T cubed
```