

CSIS0230A Principles of Operating Systems (Class A)

Notes for Tutorial 8

Program compilation, linking and execution

In the tutorial this week, we will use various methods to create an executable file from multiple source files. To do this it is important to understand the compilation process in Linux (or Unix in general), and the role of each step involved. This note introduces the process, and explain how you can observe the results of each of the steps.

1. The compilation process

Most novice programmers believe that a program is specified by one `.c` or `.cc` **source file** in C or C++, consisting of a `main()` function. This is not entirely true. Firstly, the source file can include other files, usually **header files** with `.h` file extensions¹, using the `#include` directive. Secondly, a program can be specified using many, not just one, source file. Most likely it consists of multiple source files, many of them within **library files**. One of them contains the `main()` function. Indeed, each function or global variable in the program should be contained in exactly one source file.

Each source file is processed separately into an object file, in a few stages. The first stage, called **pre-processing**, is performed by a program called `cpp`. It has nothing to do with machine code. In this stage, only superficial replacements are performed. These include removing all the comments, adding contents of header files as specified by `#include`, expanding all macros, and processing all the conditional compilation directives (as specified using `#ifdef`, `#endif`, etc). The result of this stage is a **pre-processed source file**, with file extension `.i`. However, normally the output is not written to a file. Instead, it is passed to the second stage of the compilation process through an unnamed pipe. If you want to see the output, you can ask the compiler to stop after preprocessing. For `gcc` this can be done using the `-E` flag.²

Once the file is pre-processed, the second stage, **compilation**, can take place. This stage is done by a program named `cc1` deeply hidden within a subdirectory under `/usr/lib/gcc-lib`. It converts the high-level language statements into machine-dependent, language independent **assembly language** mnemonics. In assembly language, functions are still in symbolic form rather than address form, so there is no address resolution problem. On the other hand, the compiler must know the prototype of functions and type of global variables to generate correct code, e.g., to know whether to preform automatic type conversion. That is why we need **header files** to provide them during pre-processing³. Assembly language files typically have file extension `.s`, although again the compiler normally will use unnamed pipes rather than files to give the file to the next stage. You can ask `gcc` to stop after compiling using the `-S` flag.

The assembly language files are then given to the **assembler**⁴, which turns the human-readable

¹File extensions, i.e., the ending of filenames, are just a convention used by user programs like `gcc` to decide how to process files. The OS kernel does not use it, and in general you can use any extension as long as no tools depend on it.

²Now is a good time to write a simple `helloworld` program and compile it with `-E` to see what happens. You should do similar experiment for all the remaining stages.

³Many students mistakenly that `.h` files are the libraries. This is not the truth. The `.h` files contain just the declarations. The definitions for standard functions are contained in a library `/usr/lib/libc.so`. Since this library is linked by default, you don't need to specify this library when invoking `gcc`. If you use a header file for any other library, you will have to specify the library to link into your program, as we did in tutorial 2/3 for the `crypt` library.

⁴Some compilers merge this step with the compilation step.

assembly language mnemonics into machine-readable code in **object files**. This is done by the program called `gas`, the GNU assembler. In an object file, all the symbols (i.e., function and global variable names) within the code are replaced by virtual addresses. However, most symbols cannot be resolved yet, because of two reasons. First, some symbols are simply not yet defined. They reside in a shared library or another object file yet to be produced or processed. Second, the address of the whole `.o` file depends on where the `.o` file is put in the memory, which cannot be determined yet because we don't know how much space the other `.o` files occupy. Conversely, the object file also provide some symbols that other object files can refer to. The object file thus contains a **symbol table** to record what symbols it defines and can be used by other files, or it needs to find from other files. It also contains a **relocation table** to record which locations of the file need to be modified by adding the starting address of the `.o` file, or need to be fixed up later using the address of symbols from other object files. Normally the `.o` object file is produced in the `/tmp` directory waiting for the next stage. The `-c` flag of `gcc` may be used to stop the compilation process after assembling (and place the object file in the current directory).

Once object files are produced, they can be **combined** to an executable file or a shared library file. This is done by the program `ld`, the GNU linker. The input to the linker is a set of object files and (static or shared) libraries. All object files are processed first to find all the symbols required by the object files, and resolve symbols provided by the object files. Then the libraries are added one by one, looking for symbols that are not yet defined. If one is found in a shared library, it is added to the list of dependent shared libraries, to be loaded when the resulting code is loaded into memory. If one is found in a static library, the object file that define the symbol is added to the program. At the end, the result is put into an **executable** file, or a **shared library** if the `-shared` flag is passed to `gcc` or `ld`.

When you run `gcc` without flags, all the above steps are done automatically for you. But sometimes we want to stop after some of the steps, especially when you write a program with multiple source files. At such time the flags mentioned above would be very useful.

2. Making a static library

When we write a large piece of software, we usually would like to separate it into a few programs, to make each resulting program simpler, and to avoid having to load unneeded components into memory. Typically, these programs share some object files, but not all. It would be tedious if we need to pick the needed object file for each program. Instead, we want to have a collection of object files, and let the compilation process selects them as needed. This is the idea of static libraries.

A static library is an archive of object files, with file extension `.a`. The `ar` command can be used to modify (`-r`), list (`-t`) and extract (`-x`) object files in such files. The library should also have a symbol table ("index", produced using `-s`), to ease the linker to find out whether a symbol is defined within the library, and which object file contains it. Note that a static library is purely a collection of object files, with no symbol resolution performed. The symbol resolution is done only when the linker extracts an object from the static library and link it to other object files.

To compile a program using the static library, one would simply write the full pathname of the library in the `gcc` command line. Alternatively, if you specify, say, `-lcrypt` when invoking `gcc`, the compiler looks for the static library named `libcrypt.a` in a few default places (like `/lib`, `/usr/lib`, etc), and if found (and the shared library is not found), the static library is used.

3. Making a shared library

Shared library is an alternative to static library. It is appropriate whenever the library is **frequently used** enough, by a **sufficiently large number of different programs**. The difference between shared library and static library is that, while a static library contains separated object files meant to be selected and combined by the linker, a shared library contains a **combined** set of object files—the individual object file cannot be extracted from the shared library. Like an executable file, a shared library has most of the internal addresses resolved (perhaps except a relocation to be done when the library is loaded), and the only unresolved addresses are those that are defined in a separate shared library. They are **ready to be directly mapped** into the process memory when a program is being executed. The benefit is that if multiple processes and programs use the same shared library, the same file will be used for mmap. The OS kernel can use copy-on-write for these mmaps. If the shared library does not modify the code segment (i.e., when `-FPIC` is specified), the same frames will be used for the code segment of the shared library by all these programs.

Usually, shared libraries are named like `libcrypt.so.2.3.2`, with a version number attached. A symbolic link with name like `libcrypt.so` is then created in the `/usr/lib` directory to point to the actual library, which normally resides also in `/usr/lib`. If all 2.3.x versions of `libcrypt.so` provide the same set of symbols, they are said to be compatible, and the system administrator should arrange a symbolic link `libcrypt.so.2.3` to point to the newest 2.3 version of `libcrypt`. This allows the **library to be upgraded without recompiling all the programs**. This works in the following way: when a program is compiled, it is marked with the full version number of the link target of the library, e.g., `libcrypt.so.2.3.2`. If that file cannot be found, e.g., it is replaced by a newer version, the run-time linker tries to remove the last version number, and try `libcrypt.so.2.3`. This would succeed in our case. If instead it is not found, then `libcrypt.so.2` will be tried, and if it still cannot be found an error is emitted, meaning that no compatible library version is usable. This mechanism has the benefit that the program would not accidentally use an incompatible version of the library.

As described above, the `-shared` flag of `gcc` allows us to create a shared library. Like a static library, a shared library can be used by adding the full path of the shared library when `gcc` is invoked, or by using the `-l` flag. In case when both a static and a shared library is available, preference is given to the shared library, unless you specify `-static` during the compilation.

4. Inspecting an object file

Once the assembly file is processed by `gas`, machine code is generated, which cannot be conveniently inspected using a standard editor. This is not a problem for programmers who just want to get their programs compiled and loaded into memory. However, for those who want to understand the linking process, and for those who have problems linking their object files to libraries, this can be a serious hindrance.

We thus need special tools to look at the content of these files. The `objdump` and `readelf` programs fit this gap. `objdump` has capability to look at various types of object files, including to read the **file header, program headers and section headers, symbol tables, relocation tables, debugging information and disassembly of code** stored in the object file. The `readelf` program can only read ELF-formatted files, the most usual format currently used in Linux. But it adds the capability to inspect ELF specific information like notes, version information, etc. For us, the most important benefit of `readelf` is that it display the header in a form more similar to the actual ELF file.

5. ELF: an overview

An ELF file consists of several parts:

- **File header:** It is a fixed-size header that describe the general feature and structure of the ELF file. This includes the type of ELF file (object file, shared library, executable, etc), the CPU and OS architecture for running the program, the ELF version that the file complies to, and the location and number of sections and program headers (see below). It also contains the entry point of the program, i.e., where the program should start executing.
- **Program headers:** This is the instruction given to the OS kernel about how to execute the program, by telling the kernel to map ranges of bytes within the file into ranges of addresses in the virtual address space for the process. This exists only for executable files and shared libraries. The latter range can be larger than the former, and in this case the latter part is initialized to 0 when the program loads. The file offsets are all within sections of the file, to be explained next. The entries also contain the permissions that the kernel should use for the mapping (whether it should be readable, writable and executable).
- **Section headers and the actual sections:** After compilation, the generated code and data of the program are placed into **sections**. For example, the code is placed into a section called `.text`, initialized data `.data`, uninitialized data `.bss`, symbol table `.symtab`, relocation table in `.rel.text`, string table in `.strtab`, etc. When multiple object files are linked together, those sections with the same name **merge** into one big section. This will make sure all code are put into a single contiguous piece of memory, there is one big uninitialized data section rather than many, etc. The final parts of the ELF file is a table of sections that the file contains, and then the actual content of the sections (which contain all code and data of the program).

When creating executable and shared libraries, the linker also generates some section by itself during the linking process, to support the run-time linker. They include dynamic symbol table, dynamic string table, dynamic relocation table, global offset table (GOT, for the run-time linker to put the result of relocations), and procedure linkage table (PLT, a list of code to call the functions at the offsets in the GOT).

The last two tables need a bit more explanation. Under the current implementation of the linker, the code emitted is actually like this:

```
        .section      .got                ;; The GOT
        ...
myfunc@GOT:
        .long         myfunc@FIXUP       ;; reserve 4 bytes, default to fixup
        ...
        .section      .plt                ;; The PLT
        ...
myfunc@PLT:
        jmp           *myfunc@GOT        ;; jump to the GOT entry
myfunc@FIXUP:
        push         myfunc@GOT - .got   ;; push the GOT location as argument
        jmp          *fixup@GOT          ;; jump to real fixup routine, which
                                           ;; change the GOT entry to point to
                                           ;; the real address of myfunc
```

A call to `myfunc()` is compiled to a call to `myfunc@PLT`, which jumps to the real `myfunc()` if it is fixed up before, or the fixup function if it hasn't been.