Creating and Using Software Libraries

1 Introduction

These notes summarize how to create and use static and shared libraries in a UNIX environment. They are designed to be tutorial and elementary. For a more advanced explanation about creating and using library files, I strongly recommend that you read David Wheeler's Program Library HOWTO.

These notes begin by explaining a bit about software libraries in general, then proceed to describe the differences between static and shared libraries. After this conceptual material, they describe the how-to's about creating and using both types of libraries using the tools available in a GNU-based UNIX system such as Linux. The discussion here is limited to executables and libraries in the *Executable and Link Format* (ELF), which is the format used by Linux and most UNIX systems at the time of this writing. If you do not know what this means or why it might be important, that is fine; you may safely ignore this.

If you think you do not need the conceptual discussions, you can just "cut to the chase" and jump directly to the appropriate section below, either §5 and §6 for static libraries or §7 and §8 for shared libraries.

2 About Libraries

A *software library*, also called a *program library*, is a file containing compiled code and possibly data that can be used by other programs. Libraries are not stand-alone executables – you can not "run" a library. They contain things like functions, type definitions, and useful constants that other programs can use. You have been using software libraries since your very first "Hello World" program, whether you knew it or not. Whatever function that you used to print those words on the screen was contained in a library, most likely either the C standard I/O library (if you used `printf`, for instance), or the C++ iostreams library (if you used the insertion operator of the `cout` ostream object.)

Perhaps you might have reached the point where you realize that you are writing useful code, code that you might want to use in more than one project, and that while you could continue to copy those functions into each new project, perhaps you would like to reuse that code in a more efficient way by creating a library file that contains it. If so, read on.

3 Static Versus Shared Libraries in UNIX

In UNIX, there are two kinds of library files, *static* and *shared*. The term "static library" is short for "statically linked library." A *static library* is a library that can be linked to the program statically, after the program is compiled, as part of the program executable file. In other words, it is incorporated into the program executable file as part of the build of that executable. A *shared library* is a library that is linked dynamically, either at *loadtime* or at *runtime*, depending on the particular system. Loadtime is when the program is loaded into memory in order for it to execute. Runtime is the interval of time during which it is actually running. If linking is delayed until runtime, then a symbol such as a function in the library is linked to the program only when the program calls that function or otherwise references that symbol. The fact that a shared library is a dynamically linked library is not to be confused with the use of that term by Microsoft in what they call a DLL. While "DLL" is short for "dynamically linked library", DLLs are different from UNIX shared libraries. In these notes, I use the term in the more general sense of a library that is linked to a program either at loadtime or at runtime.

Static linking, which was the original form of linking, resolves references to externally-defined symbols such as functions, by copying the library code directly into the executable file when the executable (file) is built. The *linkage editor*, also called the *link editor*, or just the *linker*, performs static linking. The term "linker" is a bit ambiguous, so I will avoid using it. The primary advantage of static linking, perhaps now the only advantage, is that the executable is self-contained and can run on multiple platforms. For example, a program might use a graphical toolkit such as GTK that may not be present on all systems. With the toolkit's libraries statically linked into the executable, the executable can run on other systems (with the same machine architecture) without requiring the users on those systems to install those library files. Once upon a time, static linking resulted in faster code as well, but the gain is negligible today.

Dynamic linking can be done either when the program is loaded into memory, or while it is running and references an unresolved symbol. In the former case, the start-up time of the program is slightly longer than if it had been statically linked, since the libraries have to be located in memory (and possibly loaded into memory if they were not already there) and then linked to the program before it can actually begin execution. In the latter case, the program will experience slightly longer running time, because whenever an unresolved symbol is found and must be resolved, there is a bit of overhead in locating the library and linking to it. This latter approach is the more common approach because it only links symbols that are actually used. For example, if a function from a shared library is not called during execution, it will not be linked to the library at all, saving time.

There are several advantages of linking dynamically over linking statically. One is that, because the executable program file does not contain the code of the libraries that must be linked to it, the executable file is smaller. This means that it loads into memory faster and that it uses less space on disk. Another advantage is that it makes possible the sharing of memory resources. Instead of multiple copies of a library being physically incorporated into multiple programs, a single memory-resident copy of the library can be linked to each program, provided that it is a shared library. Shared libraries are dynamically-linked libraries that are designed so that they are not modified when a process uses them. This is why they have the extension, `".so"` – short for shared object.

Another advantage of linking to shared libraries is that this makes it possible to update the libraries without recompiling the programs which use them, provided the interfaces to the libraries do not change. If bugs are discovered and fixed in these libraries, all that is necessary is to obtain the modified libraries. If they were statically linked, then all programs that use them would have to be recompiled.

Still other advantages are related to security issues. Hackers often try to attack applications through knowledge of specific addresses in the executable code. Methods of deterring such types of attacks involve randomizing the locations of various relocatable segments in the code. With statically linked executables, only the stack and heap address can be randomized: all instructions have a fixed address in all invocations. With dynamically linked executables, the kernel has the ability to load the libraries at arbitrary addresses, independent of each other. This makes such attacks much harder.

4 Identifying Libraries

Static libraries can be recognized by their ending: they end in `".a"`. Shared libraries have a ".so" extension, possibly with a version number following, such as `librt.so.1`. Both types of libraries start with the prefix `"lib"` and then have a unique name that identifies that library. So, for example, the standard C++ static library is `libstdc++.a`, and the shared real-time library is `librt.so.1`. The `"rt"` in the name is short for real-time.

5 Creating a Static Library

The steps to create a static library are fairly simple. Suppose that you have one or more source code files containing useful functions or perhaps definitions of useful types. For the sake of precision, suppose that `timestuff.c` and `errors.c` are two such files.

1. Create a header file that contains the prototypes of the functions defined in `timestuff.c` and `errors.c`. Suppose that file is called `utils.h`.

2. Compile the C source files into object files using the command

   ```
   gcc -c timestuff.c gcc -c errors.c
   ```

   This will create the two files, `timestuff.o` and `errors.o`.

3. Run the GNU archiver, `ar`, to create a new archive and insert the two object files into it:

   ```
   ar rcs libutils.a timestuff.o errors.o
   ```

   The "rcs" following the command name consists of a one-letter operation code followed by two modifiers. The "`r`" is the operation code that tells `ar` to insert the object files into the archive. The "`c`" and "`s`" are modifiers; `c` means "create the archive if it did not exist" and `s` means "create an index," like a table of contents, in the archive file. The name of the archive is given after the options but before the list of files to insert in the archive. In this case, our library will be named `libutils.a`.

   This same command can be used to add new object files to the library, so if you later decide to add the file `datestuff.o` to your library, you would use the command

   ```
   ar rcs libutils.a datestuff.o
   ```

4. Install the library into some appropriate directory, and put the header file into an appropriate directory as well. I use the principle of "most-closely enclosing ancestral directory" for installing my custom libraries. For example, a library that will be used only for programs that I write for my UNIX System Programming class will be in a directory under the directory containing all of those programs, such as:

   ```
   ~/unix_demos/lib/libutils.a
   ```

   and its header will be

   ```
   ~/unix_demos/include/utils.h
   ```

   If I have a library, say `libgoodstuff.a`, that is generally useful to me for any programming task, I will put it in my `~/lib` directory:

   ```
   ~/lib/libgoodstuff.a
   ```

   with its header in my `~/include` directory:

   ```
   ~/include/goodstuff.h
   ```

5. Make sure that your `LIBRARY_PATH` environment variable contains paths to all of the directories in which you might put your ***static*** library files. Your `.bashrc` file should have lines of the form[1]:

   ```
   LIBRARY_PATH="$LIBRARY_PATH:~/lib:"
   export LIBRARY_PATH
   ```

   so that `gcc` will know "where to look" for your custom static libraries. If you want your libraries to be searched before the standard ones, then reverse the order:

   ```
   LIBRARY_PATH="~/lib:$LIBRARY_PATH"
   export LIBRARY_PATH
   ```

[1] This is not the best way to do this. I use a shell function called pathmunge() for modifying paths. You can find examples of pathmunge in web searches.

6. Make sure that your `CPATH` or `C_INCLUDE_PATH` (or if using C++, your `CPLUS_INCLUDE_PATH`) contains the path to the directory in which you put the header file. My `.bashrc` file has the lines

```
CPATH="~/include"
export CPATH
```

Note. Do not put your static libraries into the same directories as your shared libraries. Keep them separate. There is a good reason for this, which will become clear later.

6 Using (Linking to) a Static Library

To use the library in a program, (1) you have to tell the compiler to include its interface, i.e., its header file, and (2) you have to tell the linkage editor to link to the library itself. The first task is accomplished by putting an include directive in the program. The second task is achieved by using the `-l` option to `gcc` to specify the *name* of the library. Remember that the name is everything between `"lib"` and the `"."`. *The* `-l` *option must follow the list of files that refer to that library.* For example, to link to the `libutils.a` library you would do two things:

1. In the program you would include the header file for the library:

```
#include "utils.h"
```

2. To build the executable, you would issue the command

```
gcc -o myprogram myprogram.c -lutils
```

   or the following if you did not modify your `CPATH`:

```
gcc -o myprogram myprogram.c -lutils -I~/unix_demos/include
```

   but in either case, only if you are certain that there is not a shared library with the same name in a directory that will be searched ahead of the one in which `libutils.a` is located, or in the same directory as `libutils.a`. This is because `gcc`, by default, will always choose to link to a shared library of the same name rather than a static library of that name. This is one reason why you should not put static libraries in the same directory as shared libraries.

   If you get the error message

```
/usr/bin/ld: cannot find -lutils
collect2: ld returned 1 exit status
```

   it means that you did not set up the `LIBRARY_PATH` properly. (Did you export it? Did you type it correctly?)

   If you want to be safe, you can use the `-L`*dir* option to the compiler. This option adds *dir* to the list of directories that will be searched when looking for libraries specified with the `-l` option, as in

```
gcc -o myprogram myprogram.c -L~/unix_demos/lib -lutils
```

   Directories specified with `-L` will be searched before those contained in the `LIBRARY_PATH` environment variable.

If you do a web search on this topic, you may see instructions for building your program of the form

```
gcc -static myprogram.c  -o myprogram -lutils
```

This will probably fail with the error message

```
/usr/bin/ld: cannot find -lc
collect2: ld returned 1 exit status
```

because the `-static` option tells `gcc` to statically link `myprogram.c` to all libraries, not just `libutils.a`. Since the C standard library no longer ships as a static library with most operating systems, the link editor, `ld`, will not find `libc.a` anywhere. Do not try to use the `-static option`. Follow my instructions instead.

7 Creating a Shared Library

The `ar` command does not build shared libraries. You need to use `gcc` for that purpose. Before diving into the details though, you need to understand a few things about shared libraries in UNIX to make sense out of the options to be passed to `gcc` to create the library.

Shared Library Names

Every shared library has a special name called its *"soname"*. The *soname* is constructed from the prefix "`lib`", followed by the name of the library, then the string `".so"`, and finally, a period and a version number that is incremented whenever the interface changes. So, for example, the soname of the math library, `m`, might be `libm.so.1`.

Every shared library also has a "*real name*", which is the name of the actual file in which the library resides. The real name is longer than the soname; it must be formed by appending to the soname a period, and a minor number, and optionally, another period and a release number. The minor number and release number are used for configuration control.

Lastly, the library has a name that is used by the compiler, which is the soname without the version number.

Example 1

The `utils` library will have three names:

`libutils.so.1` – This will be its soname.

`libutils.so.1.0.1` – This will be the name of the file. I will use a minor number of 0 and a release number 1

`libutils.so` – This is the name the compiler will use, which we will called the *linker name*.

Example 2

If you look in the `/lib` directory, you will see that links are created in a specific way; for each shared library there are often at least three entries, such as

```
lrwxrwxrwx 1 root root 11 Aug 12 18:52 libacl.so -> libacl.so.1
lrwxrwxrwx 1 root root 15 Aug 12 18:51 libacl.so.1 -> libacl.so.1.1.0
-rwxr-xr-x 1 root root 31380 Aug 3 18:42 libacl.so.1.1.0
```

Notice that the compiler's name (without the version number) is a soft link to the soname, which is a soft link to the actual library file. When we set up our `libutils` library, we need to do the same thing. Every library will have three files in the directory where it is placed: the soname will be a soft link to the actual library file, and a soft link to the soname file named with the linker name.

Steps to Create the Library

1. For each source code file that you intend to put into a shared library, say `stuff.c`, compile it with *position independent code* using the following command:

   ```
   gcc -fPIC -g -Wall -c stuff.c
   ```

   This will produce an object file, `stuff.o`, with debugging information included (the `-g` option), with all warnings enabled (the -Wall option), which is always a safe thing to do. The `"-fPIC"` option is what generates the *position independent code* (hence `PIC`). Position independent code is code that can be executed regardless of where it is placed in memory. This is not the same thing as relocatable code. Relocatable code is code that can be placed anywhere into memory with help from a linkage editor or loader. Instructions such as those that specify addresses relative to the program counter are position independent.

2. Suppose that `stuff.o` and `tools.o` are two object files generated in accordance with the first step. To create a shared library containing just those files with soname `libgoodstuff.so.1`, and real file name `libgoodstuff.so.1.0.1`, use the following command:

   ```
   gcc -shared -Wl,-soname,libgoodstuff.so.1 -o libgoodstuff.so.1.0.1  stuff.o  tools.o
   ```

   This will create the file `libgoodstuff.so.1.0.1` with the soname `libgoodstuff.so.1` stored internally. ***Note that there cannot be any white space before or after the commas***. The `-Wl` option tells `gcc` to pass the remaining comma-separated list to the link editor as options. You might be advised by someone else to use `"-fpic"` instead of `"-fPIC"` because it generates faster code. Do not do so. It is not guaranteed to work in all cases. `-fPIC` generates bigger code but it never fails to work.

3. It is time to install the library in the appropriate place. Unless you have superuser privileges, you will not be able to install your nifty library in a standard location such as `/usr/local/lib`. Instead, you will most likely put it in your own `lib` directory, such as `~/lib`. Just copy the file into the directory.

4. After you copy the file into the directory, you should run `ldconfig` on that directory, with a `-n` option, e.g.

   ```
   ldconfig -n ~/lib
   ```

   `ldconfig`, with the `-n` option, creates the necessary links and cache to the most recent shared libraries found in the given directory. In particular, it will create a symbolic link from a file named with the soname to the actual library file. If there are multiple minor versions or releases, `ldconfig` will link the soname file to the highest-numbered minor version and release combination. The `-n` option tells `ldconfig` not to make any changes to the standard set of library directories. After `ldconfig` runs in our example, we would have the link

   ```
   libgoodstuff.so.1 -> libgoodstuff.so.1.0.1
   ```

   After running `ldconfig`, you should manually create a link from a file with the linker name to the highest-numbered soname link. In our example, we would type

   ```
   ln -s libgoodstuff.so.1 libgoodstuff.so
   ```

   to create the link

   ```
   libgoodstuff.so -> libgoodstuff.so.1
   ```

5. If at some future time, you revise the `goodstuff` library, you would increment either the minor version number or the release number, or perhaps even the major version number, if the interface to the library changed. If you just change an algorithm internally or fixed a few bugs, you would not change the major number, only the minor one or the release number. Suppose that you create a new release, `libgoodstuff.so.1.0.2`, with soname `libgoodstuff.1`. You would copy the file into the same directory as the older release and run `ldconfig` again. `ldconfig` would change the link from the soname to the later release. A listing of that directory would then look like

```
libutils.so -> libutils.so.1
libutils.so.1 -> libutils.so.0.2
libutils.so.1.0.1
libutils.so.1.0.2
```

8 Using a Shared Library

What you need to understand about how to use shared libraries is that it is a two-step linking process. In the first step, the linkage editor will create some static information in your executable file that will be used later by the dynamic linker at runtime. So both the linkage editor and a dynamic linker participate in creating a working executable.

You link your program to a shared library in the same way that you link it to a static library, using the `-l` option to `gcc`, to name the library to which you want your program linked, and using the `-L`*dir* option to tell it which directory it is in if it is not in a standard location. For example:

```
gcc -o myprogram myprogram.c -L~/lib  -lgoodstuff
```

will create the executable `myprogram`, to be linked dynamically to the library `~/lib/libgoodstuff.so`. We can also write

```
gcc -o myprogram myprogram.c ~/lib/libgoodstuff.so
```

skipping the options `-l` and `-L`. The two methods are equivalent. If `~/lib` is in the `LIBRARY_PATH` environment variable, then you can also write

```
gcc -o myprogram myprogram.c   -lgoodstuff
```

and this will be equivalent as well. All of the above assume that the directory containing the header file is in your `CPATH` or is in a standard location. Otherwise remember to add the option `-I`*includedir* to this command.

This is just the first step. Your executable will not run correctly unless the dynamic linker can find you shared library file. One way to tell whether it will run correctly is with the `ldd` command. The `ldd` command prints shared dependencies in a file. Translation: it displays a list of shared libraries upon which your program depends. If `ldd` does not display the path to `~/lib/libgoodstuff.so`, then `myprogram` will fail to find the file and will not run. If the dynamic linker will be able to find my library, the output of `ldd` would look something like:

```
linux-gate.so.1 =>  (0x00a31000)
libgoodstuff.so.1 => ~/lib/libgoodstuff.so.1 (0x00caa000)
libc.so.6 => /lib/libc.so.6 (0x00110000)
/lib/ld-linux.so.2 (0x00bd5000)
```

If it will not be able to find it, I will see

```
linux-gate.so.1 =>  (0x00a31000)
libgoodstuff.so.1 => not found
libc.so.6 => /lib/libc.so.6 (0x00110000)
/lib/ld-linux.so.2 (0x00bd5000)
```

If you had the means to put your shared library file in a standard directory, this problem would be solved easily. Unfortunately, with just user privileges and not superuser privileges, you cannot do this. The easiest solution to this problem is one that is not recommended for various reasons: you can modify the environment variable `LD_LIBRARY_PATH`, which the dynamic linker uses at loadtime and runtime to locate shared libraries. To be precise, the dynamic linker searches the directories in this variable before any in the standard locations. Therefore, you can put the line

```
LD_LIBRARY_PATH="~/lib"
export LD_LIBRARY_PATH
```

in your `.bashrc` file to have the dynamic linker search that directory at run time. The alternative is to modify the variable every time you run the program, which is a nuisance I think, or to hard code the path to the libraries into the executable using the `-rpath` option to the linkage editor (which is described in the `ld` man page.)

There is one other option. You can define the `LD_RUN_PATH` variable to contain the directory in which you put your libraries, in your `.bashrc file`:

```
LD_RUN_PATH="~/lib"
export LD_RUN_PATH
```

If this variable is defined when you compile the executable, then the run path will be hard-coded into the executable and the dynamic linker will find your libraries at run time.

9 Displaying the Contents of a Library

This section is a bit more advanced and can be skipped if all you want to do is create your own libraries. It is also not very thorough. Its purpose is to give you some "leads" in case you want to understand more about the structure of libraries and code in general. There are several utilities that can be used to examine the contents of library files, with varying degrees of information provided and ease of use.

The least amount of information is obtained with the GNU archiver, `ar`, which will display a list of the names of the `.o` files contained in a static library file. Use the `t` operation code. For example, if you list the `.o` files in the GNU Standard C++ library, you would first find it, usually in a directory such as `/usr/lib/gcc/`<*machine_architecture*>`/`<*version*>. For example, on my system it is in `/usr/lib/gcc/i686-redhat-linux/4.4.5`. Then the command

```
ar t libstdc++.a
```

will produce a very long list containing each of the object files that has been incorporated into the library. A partial list would include

```
atomic.o
codecvt.o
compatibility.o
complex_io.o
ctype.o
debug.o
hash.o
globals_io.o
hashtable.o
ios.o
...
```

This doesn't tell you very much, and `ar` does not work on shared libraries. You can use the `nm -s` command to get information about shared and static libraries. For static libraries, you would type

```
nm -s <libraryname>
```

For shared libraries, you need the `--dynamic` option:

```
nm -s --dynamic <libraryname>
```

In either case, unless you know how to interpret the output, this will not be very useful. But if all you want to do is see if a particular function name is actually in the library, you can `grep` for the name in the output. If you do not find it, it is not in that library.

The `readelf` utility is a command designed to display information about ELF files in general[2]. ELF stands for "Executable and Linkable Format". ELF is a standard format for executable files, object files, and libraries. It replaces the older "a.out" and COFF formats. It was developed by UNIX System Laboratories and has been adopted by almost all UNIX vendors. It will be even more difficult to understand the output of `readelf` unless you spend some time learning about the structure of ELF files and the output of the `readelf` command itself. But if all you want to do is check what functions or other symbols are in an executable, you can type

```
readelf -s <elf-file>
```

and you will see a large amount of output that you can pass through a filter. For example, if I run `readelf` on a program, say `myprogram`, that was linked to my `libutils.so` shared library,

```
readelf -s myprogram
```

a portion of the output looks like this:

```
Symbol table '.dynsym' contains 17 entries:
Num:    Value Size Type     Bind     Vis      Ndx Name
  0: 00000000    0 NOTYPE   LOCAL    DEFAULT  UND
  1: 00000000    0 FUNC     GLOBAL   DEFAULT  UND show_time
  2: 00000000    0 NOTYPE   WEAK     DEFAULT  UND __gmon_start__
```

The fact that `show_time` has a value of 0 means that it is not yet bound to an address. This is to be expected, because the actual binding will not take place until runtime. To learn more, read the man page for `ELF` and then for `readelf`. You can also download the specification of ELF at various websites.

[2]On some systems such as Solaris, there is no readelf; use elfdump instead