Development Tools 6: Advanced Development

1. Intro
2. Text Editors
3. IDE's
4. Compiling
5. Debugging and Tracing
6. Advanced Development
   - **advanced development tools**
   - **ELF executables**
   - **libraries**

Advanced Development Tools

Large projects, such as an open source software project, will typically require the use of additional development-related tools.

For example, the GNU **Autotools** suite (**Autoconf**, **Automake**, etc.) helps to support the distribution of cross-platform UNIX software.

Autoconf produces the standard **configure** scripts that most open source projects provide, which when run examine the capabilities of a platform and produce appropriate *makefiles*.

So these tools are the basis for the standard command sequence for building open source software:
`./configure`, `make`, `make install`.

Advanced Development Tools (contd.)

Projects that involve multiple programmers will generally require use of **version control** (**revision control**) software.

This type of software provides capabilities such as controlling who can "check out" software modules, allowing developers to work in parallel and merge modifications, keeping track of who was responsible for "committing" each modification, and so forth.

Among the better known and/or more popular systems for open source projects are: RCS, CVS, Subversion/SVN, and Git.

Some editors (e.g., vi and Emacs) and many IDE's support one or more of the popular version control systems.

Process Address Spaces

The address space of any Linux process consists of several different **segments**:

text – machine code/instructions, generally read-only

data – *initialized* global and `static` variables

bss (or **BSS**) – *uninitialized* global/`static` variables (automatically zeroed/`NULL`'d at startup)

heap – dynamically allocated memory

memory mapping – shared libraries and shared memory IPC

thread stacks – stacks for addition Pthreads of process

stack – the program/execution stack (userspace)

These segments are placed in the address space in the order given: text segment at lowest addresses and stack segment at highest addresses.

Process Address Spaces (contd.)

The stack *bottom* is at the *top* of the (user's) virtual address range and *grows down* in address as stack frames are pushed on.

The heap sits at the *top* of the data segment and *grows up* in address if it must be expanded (e.g., by C runtime).

The top of the heap (end of program plus variables/data) is called the **program break**.

The virtual address range between the break and the stack top is used for **dynamic libraries**, **shared memory**, and **thread stacks**.

ELF Executables

The text, data, and bss segments come directly from the contents of the **executable file** being run.

Linux uses the popular executable format **ELF** (Executable and Linkable Format).

ELF files consist of the **ELF header** describing various **sections**, plus **program headers** that describe how to map sections to *segments*.

Four key sections are: .text, .data, .rodata, and .bss.

The .text and .rodata sections form the text process segment with read and execute rights.

The .data and .bss sections are put into process segments with read and write rights.

ELF Executables (contd.)

See "`man elf`" for further info.

There are a number of commands that are useful for working with executable and object files:

- `readelf` – view ELF file contents:
  - `-S` option – display sections
  - `-l` option – display segments (section to segment mapping)
- `objdump` – display information from object files
- `nm` – list symbols from object files
- `ar` – create, modify, and extract from archives
- `strings` – print the strings of printable characters in files

Libraries

A **library** file contains precompiled (**binary/object code**) versions of a set of functions.

Library files make it easy for the set of functions to be used by other programs.

Note that a library file is *not* an **executable**—i.e., it cannot be directly executed.

Instead, as part of the compilation process, a library's functions must be **linked** with the program that calls its functions, to create an executable.

Depending on the type of the library, the library file may also need to be available at execution time.

Libraries: Static vs. Shared

There are two types of library files:

- **static libraries** – `.a` extension
- **shared libraries** – `.so` ("shared object") extension

Linux shared libraries may also be referred to as **dynamic libraries**.

Technically, they are **dynamically linked shared object libraries**.

Being **dynamically linked** means that function addresses are resolved at *execution/load time* rather than at compile time.

With a *static library*, the code for referenced functions gets included in the created executable.

Libraries: Static vs. Shared (contd.)

With a *shared library*, references to the required functions will simply be left in the executable, to be resolved by the linker/loader `ld` at execution time.

Static libraries:

- advantage: the resulting executables are complete; they does not require further access to the library
- disadvantage: executables will be larger since they contain library code
- disadvantage: code for popular libraries will be stored many times via multiple executables
- disadvantage: each process will have to load its own copy of library routines into RAM to execute
- disadvantage: bugfixes to library will require recompiling all executables using library

Libraries: Static vs. Shared (contd.)

Shared libraries:

- advantage: executables will be smaller since they won't contain library code
- advantage: code for popular libraries will be stored just once (in library file)
- advantage: all processes using the same library routine will share one copy in RAM
- disadvantage: executables are incomplete; they cannot be executed without access to the library files
- disadvantage: furthermore, must have access to *compatible* library versions (see *"DLL hell"* for more info!)

Libraries: Static vs. Shared (contd.)

Linux and most other modern OSs primarily make use of shared libraries.

Shared libraries had significant advantages when harddrives were small and RAM was limited.

These days, though, the need to have compatible libraries installed on evey system you want to run an executable on can be a major annoyance.

In fact, a significant new concept is **software containers**.

While containers incorporate security mechanisms like **sandboxing**, a key aspect of their popularity is that they are guaranteed to run anywhere, since they contain all required program resources.

This is a property that executables based on static libraries share.

Using Libraries

Whether using static or shared libraries, the corresponding library file must be available when the executable is being produced.

Furthermore, `gcc` must be told which library files (other than `libc`) to have `ld` look in to create an executable.

There are three ways to inform `gcc` command about libraries:

- option `-lNAME` causes library file `libNAME.*` to be searched for referenced functions
- option `-LDIRECTORY_PATH` causes `DIRECTORY_PATH` to be searched for a required library file, instead of only the standard library locations
- including a library file's *path* causes it to be searched for referenced functions

Using Librariess (contd.)

Example `gcc` calls using libraries:

- Use library `libcrypt.*`:
  `gcc -omyprog myprog.c -lcrypt`
- Use library `libcrypt.*` from non-standard directory:
  `gcc -omyprog myprog.c -L~/Downloads/libcrypt -lcrypt`
- Use specific `libcrypt` library file:
  `gcc -omyprog myprog.c ~/Downloads/libcrypt/libcrypt-2.18.so`

The standard directories for libraries are typically `/lib` or `/usr/lib`.

64-bit versions of libraries are generally installed in `/lib64` or `/usr/lib64`.

Using Libraries (contd.)

One confusing aspect of working with *shared libraries* is that even if you tell `gcc` about required libraries *at compile time*, `ld` must know how to find them again *at execution time*.

`ld` searches directories that are typically specified in `/etc/ld.so.conf` and/or in files in the directory `/etc/ld.so.conf.d`.

If one has installed libraries in other directories, those directories can be added to one of the above locations.

Note that after doing this, the linker cache must then be updated by running the command "`ldconfig`."

A temporary way to add a directory to search for a library is to do the following before execution:

`export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:`*dir_to_search*

Declarations vs. Definitions

When creating libraries, it is important to understand the distinction in C between a **declaration** and a **definition**:

- A *declaration* informs the compiler about characteristics of a function or variable so that the compiler can correctly process references to it.
- A declaration does not allocate storage for variables nor detail how functions are implemented.
- A *definition* is what allocates storage for a variable or gives the implementation of a function.

Recall that the C compiler needs to see at least a *declaration* of a variable/function prior to encountering a reference to the variable/function.

Header Files

For functions defined within the same file, we include **function prototypes** prior to the definitions of main and other functions.

We also place *global variable* definitions near the top of the file.

With libraries, the library author needs to provide a **header file** that will be `#include`'d in any files that call the library's functions.

The header file must containing prototypes for all functions that are part of the library (along with other needed information).

So a library really consists of two files: (1) a header file, and (2) the library (archive) file.

Header Files

What should go into each file?

Header files are to include:

- *declarations* for:
  - functions
  - global variables
- *definitions* for:
  - (global) types
  - macros

Library (archive) files are to include:

- compiled *definitions* for:
  - functions
  - global variables

Creating Static Libraries

A static library is simply a collection of **object files** representing compiled functions.

The `ar` (achiver) command is used to combined one or more object files into a library file.

The first step in creating a library is to use `gcc` to create the *object files* from your C source code files, e.g.:

```
gcc -c libfoo1.c libfoo2.c
```

This will create the object files `libfoo1.o` and `libfoo2.o`.

To create the static library file `libfoo.a` from these two object files, do the following:

```
ar -cr libfoo.a file1.o file2.o
```

Creating Static Libraries (contd.)

One can list the object files included in a library like:

```
ar -t libfoo.a
```

Symbols can be seen using the `nm` command:

```
$ nm  /lib64/libc.so.6
0002efc0 T abort
001b7920 b abortfunc
001b71a4 B __abort_msg
00030eb0 T abs
000f9020 W accept
000f98c0 W accept4
000e51b0 W access
...
000b86c0 T alarm
...
```

Shared Library sonames

A key goal with shared libraries is to be able to have multiple different versions of each library installed, and have executables link to a compatible version when run.

This is accomplished by adhering to particular naming schemes, along with the use of *symbolic links*.

Each shared library file will have both its *"real name"* and its **soname**.

An *soname* consists of the following components concatenated together:

- `lib`
- the name of the library (e.g., `foo`)
- `.so.`
- a version number

Shared Library sonames (contd.)

E.g., `libfoo.so.1`

The version numbers must get incremented whenever there are incompatible changes to the library's interface (i.e., function prototypes change).

So how do these sonames relate to real names—i.e., the actual filenames of the library files?

Real names add to the soname additional version information: a dot and a *minor number*, and *optionally*, another dot and a *release number*.

E.g., `libfoo.so.1.3` or `libfoo.so.1.3.2`

While the actual library file will be, say, `libfoo.so.1.3.2`, a *symbolic link* to that file will be created, with the soname `libfoo.so.1`.

Shared Library sonames (contd.)

For example, the bzip2 library entries on a Linux system:

```
lrwxrwxrwx 1 root root    15 Jan 24  2015 /usr/lib64/libbz2.so -> libbz2.so.1.0.6
lrwxrwxrwx 1 root root    15 Jan 24  2015 /usr/lib64/libbz2.so.1 -> libbz2.so.1.0.6
-rwxr-xr-x 1 root root 67936 Oct 17  2013 /usr/lib64/libbz2.so.1.0.6
```

For example, the wireshark library entries on a Linux system:

```
lrwxrwxrwx 1 root root       22 Jun 28  2015 /usr/lib64/libwireshark.so.3 ->
                                                 libwireshark.so.3.1.14
-rwxr-xr-x 1 root root 60989088 May 13  2015 /usr/lib64/libwireshark.so.3.1.14
lrwxrwxrwx 1 root root       21 Sep  1  2015 /usr/lib64/libwireshark.so.5 ->
                                                 libwireshark.so.5.0.7
-rwxr-xr-x 1 root root 66689336 Aug 13  2015 /usr/lib64/libwireshark.so.5.0.7
```

Creating Shared Libraries

Creating a shared library is similar to creating a static library (create object files, combine into library).

However, different settings and only `gcc` (with `ld`) are required.

The first difference is that when creating *object files* from your library source files, you must use `gcc` options that generate **position independent code**:

- `-fpic` – best, but may be too big for some architectures and fail, then use next:
- `-fPIC` – guaranteed to work but potentially less efficient

E.g., `gcc -c -fpic libfoo1.c`

Creating Shared Libraries (contd.)

The simplest way to create a shared library from object file(s) is to use gcc's `-shared` option and manually specify the output name:

```
gcc -shared -o libfoo.so.1 libfoo1.o libfoo2.o
```

You can have gcc automatically create the soname to real name:

```
gcc -shared -Wl,-soname,libfoo.so.1
    -o libfoo.so.1.1.0 libfoo1.o libfoo2.o
```

The `ldd` command allows you to view the shared library dependencies of an executable:

```
$ ldd myprog
  linux-vdso.so.1 (0x00007ffc21bfe000)
  librt.so.1 => /lib64/librt.so.1 (0x00007f15ee78e000)
  libpthread.so.0 => /lib64/libpthread.so.0 (0x00007f15ee571000)
  libc.so.6 => /lib64/libc.so.6 (0x00007f15ee1b9000)
  /lib64/ld-linux-x86-64.so.2 (0x00007f15ee996000)
```

Library Problems

If `gcc/ld` cannot find the shared library at compile time, you will see messages like: (assume `foo_func1` is a function in the library)

```
myprog.c: undefined reference to 'foo_func1'
collect2: error: ld returned 1 exit status
```

This will require that you inform `gcc` about the library using the `-l`, `-L`, or library path approaches shown earlier.

If `ld` cannot find the shared library at execution time, you will see messages like:

```
./myprog: error while loading shared libraries:
  libfoo.so.1: cannot open shared object file:
    No such file or directory
```

In that case, running `ldd` on the executable should show something like: `libfoo.so.1 => not found`

Library Problems (contd.)

This will require that you inform `ld` about additional directories to search for needed library files, or else add your library to a standard location.

Methods for adding search paths for `ld` were discussed earlier (and may require privileges).

The simplest approach to temporarily run your program if the library is in the CWD is:

```
LD_LIBRARY_PATH=. ./myprog
```

(here we are defining `LD_LIBRARY_PATH` just for this execution)