Chapter 5

Compiling Software

The process for adding software in Linux is quite different from people coming from a windows environment are used to. Specifically, it contains two extra steps: configuring and compiling. These extra steps take the program from source code to a compiled binary. Its not that these steps don't exist in Windows, its just that for the most part, they are done by the software developer before distribution.

The reason these steps come up is because compiled binaries are architecture dependent. That is, something compiled for the x86 architecture (Intel & AMD) won't run on a different architecture like PowerPC (Mac). The reason Windows people don't have to deal with this is because the entire Windows operating system is a pre-compiled binary that is built only to work on the x86 or x86_64 (64-bit) architectures. Thus, Windows software developers can pre-compile their code and distribute binary packages that only require installation.

Linux, however, is on open source system that can be compiled to work on any architecture (x86, x86_64, PowerPC, SPARC, ARM, MIPS, any more). Therefore, in order for software developers to make their code usable to people on any architecture, they release the source code and have the user go through the compilation process.

We will discuss how to to take software from the distributed source code to an installed program (often called the `configure`, `make`, `make install` process) using the **Apache** web server as an example.

5.1 Getting the Source

So we want to set up an Apache web server. The logical place to start is the Apache website: `http://apache.org/`. The Apache Software Foundation works on a lot of software products, but we are interested in the HTTP server. The downloads page offers several sources:

```
Unix Source: httpd-2.0.59.tar.gz [PGP] [MD5]
Unix Source: httpd-2.0.59.tar.bz2 [PGP] [MD5]
```

You will find that most of the time, program source is distributed in compressed tar archives. You will also find that most of the time an MD5 checksum for each archive is also released. This can be used after download to verify that the source file was not corrupted during download. First we get the files with the `wget` command line download program (ignore the line breaks, they were inserted so the command would fit on the page):

```
aoaks@blight:~/src$ wget http://apache.mirrors.tds.net/httpd/
httpd-2.0.59.tar.gz http://www.apache.org/dist/httpd/
httpd-2.0.59.tar.gz.md5
```

Next, if possible, we verify the archive using `md5sum`:

```
aoaks@blight:~/src$ md5sum httpd-2.0.59.tar.gz
35a4cebaa6b4548f9a48375ea9629c8f  httpd-2.0.59.tar.gz
aoaks@blight:~/src$ cat httpd-2.0.59.tar.gz.md5
35a4cebaa6b4548f9a48375ea9629c8f  httpd-2.0.59.tar.gz
```

or, simply:

```
aoaks@blight:~/src$ md5sum -cv httpd-2.0.59.tar.gz.md5
httpd-2.0.59.tar.gz OK
```

Finally, now that we have confirmed the source has not been altered, we extract the source from the archive. If your version of `tar` supports it, you can decompress and extract in one command:

```
aoaks@blight:~/src$ tar zxvf httpd-2.0.59.tar.gz
```

otherwise, you will have to decompress first, then extract:

```
aoaks@blight:~/src$ gunzip httpd-2.0.59.tar.gz
aoaks@blight:~/src$ tar xvf httpd-2.0.59.tar
```

5.2 Configuring

The **configure** step is used for two reasons. First, this is when you specify your setup options for the program you will eventually create. Second, this step is used for determining information about your system architecture and the paths to the relevant libraries and development files that the application will need. Fortunately, most of this work has been taken care of by the software developer. Most, if not all source packages will come with a `configure` script in the top directory of the package source. Lets look at the top directory of our Apache source:

```
aoaks@blight:~/src/httpd-2.0.59$ ls
ABOUT_APACHE      CHANGES        InstallBin.dsp  os/
acconfig.h        config.layout  LAYOUT          README
acinclude.m4      configure*     libhttpd.dsp    README.platforms
```

```
Apache.dsp         configure.in   LICENSE         server/
Apache.dsw         docs/          Makefile.in     srclib/
apachenw.mcp.zip   emacs-style    Makefile.win    support/
build/             httpd.spec     modules/        test/
BuildBin.dsp       include/       NOTICE          VERSIONING
buildconf*         INSTALL        NWGNUmakefile
```

As you can see, Apache came with a `configure` script. To configure the software with all the default options, you need only run the `configure` command with no arguments:

```
aoaks@blight:~/src/httpd-2.0.59$ ./configure
```

However, most of the time you will want to specify at least some custom build parameters. Most `configure` scripts are set up so that the `-h` or `--help` will give you all the possible build options:

```
aoaks@blight:~/src/httpd-2.0.59$ ./configure -h | less
```

This will give you a list of all the possible build options, usually with their default values noted. You specify the build options as command line options to `configure`. This process is similar to the Windows interactive installers. This is where you specify things like installation directory, which features you want enabled/disabled, and paths to needed libraries if they are in a non-standard location.

We won't do much customization of the sample Apache server, but this is the point most customizations would be done. Apache actually has a feature that allows additional modules to be built after the fact and loaded in without recompiling, but that won't usually be the case. Lets just specify an installation directory (notice the trailing `\` at the end of the first line. This escapes the `newline` character inserted when the `Enter` key is pressed, allowing commands to span multiple lines):

```
aoaks@blight:~/src/httpd-2.0.59$ ./configure \
--prefix=/opt/httpd/apache-2.0.59
```

This will start generating a massive amount of output as it checks your system configuration:

```
checking for getpwnam_r... yes
checking for getpwuid_r... yes
checking for getgrnam_r... yes
checking for getgrgid_r... yes

Checking for Shared Memory Support...
checking for library containing shm_open... -lrt
checking sys/mman.h usability... yes
checking sys/mman.h presence... yes
```

After a lot of output, it will eventually finish. If it finishes with no errors, you are pretty much golden. It is very unlikely you will have problems during the next steps (though not impossible, mind you).

The configure step is where most of your problems will come up. This is where you will find out if you are missing files or programs needed to build the software in question. For example, the `configure` script will probably look for the presence and location of the a `C` compiler. If it can't find one, it will error out, and you won't be able to continue building your software until that dependency has been resolved. You may also run into problems with missing development files. For example, if you were compiling **PHP** and you wanted to build it with **MySQL** support, you will need the MySQL client source files. If these files aren't in the standard location and you don't specify their location to PHP's `configure` script, it will generate an error. Again, you won't be able to build the software in question until the development file dependencies have been resolved.

Once your `configure` script finished without errors, you are ready to move on to the next step.

5.3 Building

Actually compiling the software will take the longer time of this process. There are often hundreds of source files that must be compiled and linked together to form the final binary program. Fortunately, all of the compile orders and linking instructions are taken care of by the `make` program.

The `make` program is a simple utility that takes a **Makefile** and an optional **target** and does whatever the `Makefile` says must be done to satisfy the build target. The `Makefile` itself consists of long lists of dependency listings that basically say: "This file depends on these other files. In order to generate this file from these files, run this command". When `make` is run, it finds the top level file and looks at its dependencies. If the file doesn't exist, it attempts to generate it using the command specified. However, before trying to make the top file, it looks at the dependencies to see if any of those files are specifies in the `Makefile`. If some of them are, it hold off on generating the top level file until its dependencies have been created. This process continues down the dependency tree until there are no more dependent files that must be generated and then works backwards, generating specified files from their dependencies.

This is a very nice facility, but you may wonder where this `Makefile` comes from. Well, this is the result of the `configure` script. While it was examining your system configuration, it was generating the `Makefiles` using the information it gathered and the options you specified. As a result, the entire build process is simplified to running `make`:

```
aoaks@blight:~/src/httpd-2.0.59$ make
```

This will begin compiling the final binaries and will output the commands it is running as they are run. For example:

```
/bin/sh /home/a/ao/aoaks/src/httpd-2.0.59/srclib/apr/libtool --si
lent --mode=compile gcc -g -O2 -pthread   -DHAVE_CONFIG_H -DLINUX
=2 -D_REENTRANT -D_GNU_SOURCE   -I../../include -I../../include/a
rch/unix -I../../include/arch/unix  -c copy.c && touch copy.lo

/bin/sh /home/a/ao/aoaks/src/httpd-2.0.59/srclib/apr/libtool --si
lent --mode=compile gcc -g -O2 -pthread   -DHAVE_CONFIG_H -DLINUX
=2 -D_REENTRANT -D_GNU_SOURCE   -I../../include -I../../include/a
rch/unix -I../../include/arch/unix  -c dir.c && touch dir.lo
```

As long as your `configure` process finished without errors, it is very unlikely that you will run into problems here, however, it is not outside the realm of possibility. If you do run into errors, it is usually relatively easy to determine what happened from the error messages and how to fix the problem.

One of the nice features of `make` is that it checks the modification time of files and dependencies before doing work. Thus, if a file is specified in the `Makefile` and has dependencies, but those dependencies all have modification times earlier than the file in question, `make` knows it doesn't have to regenerate the file and moves on to the next. Thus, if the `make` process was interrupted for some reason, this handy feature allows you to pick up right were you left off.

5.4 Installing

This is where the Linux software build process meets up with Windows and Mac type installers. The program has now been compiled and is ready to be installed into the system. Fortunately, all of the installation instructions were given during the `configure` process. Most of the time, installing simply means giving `make` the **install** target. There are often other targets available, such as **test**, which will test the compiled program with predefined tests, or **clean**, which will remove compiled files and configuration information so a different built process can be done.

While the other steps could be done as a normal user, the install process usually requires extra privileges. This depends on the prefix that `make` is expecting to install in. If you specified some place where you have permission, like `$HOME/apache/`, then you can do it as yourself. However, the default location will often be somewhere in the system directories like `/usr/local/` or `/opt/local/` (in our example, `/opt/httpd/apache-2.0.59`). Therefore, in order to continue, either become root or use sudo (if you have it) to run the install command:

```
aoaks@blight:~/src/httpd-2.0.59$ sudo make install
```

Again, this will output a lot of information as it moves around the source tree and places things in the correct place in the filesystem. You really shouldn't get any errors here. If you do, it is almost certainly related to a lack of permission on certain directories. If this finishes without errors, that it. You're done!

5.5 The Finished Product

Now that we have finished the install process, lets look at what we got:

```
aoaks@blight:/opt/httpd/apache-2.0.59$ ls
bin/    cgi-bin/  error/   icons/    lib/   man/     modules/
build/  conf/     htdocs/  include/  logs/  manual/
```

We can see intuitively where things are stored. The compiled binaries and control scripts are in `bin`, logs are stored in `logs`, web documents are in `htdocs`, additional modules are in `modules`, etc. The main configuration file is `conf/httpd.conf`. The actual server binary is `bin/httpd` and the control script is `bin/apachectl`. Note that in the default Apache server configuration, the daemon will attempt to bind to port 80. By convention, the first 1024 ports are considered privileged and can only be bound to by a process that is started by `root`. If you try to start apache as a normal user on the privileged port 80, you will get errors:

```
aoaks@blight:/opt/httpd/apache-2.0.59$ ./bin/apachectl start
httpd: Could not determine the server's fully qualified domain
name, using 127.0.1.1 for ServerName
(13)Permission denied: make_sock: could not bind to address
[::]:80
no listening sockets available, shutting down
Unable to open logs
```

You can either change the `httpd.conf` file so that the daemon will bind to a non-privileged port, or start the server as `root`:

```
aoaks@blight:/opt/httpd/apache-2.0.59$ sudo ./bin/apachectl start
Password:
httpd: Could not determine the server's fully qualified domain
name, using 127.0.1.1 for ServerName
```

Note that this time, no permission errors were generated, only a warning about the server's domain name was issued.
If you open a web browser and point it to the server, you will get a simple page saying "If you can see this, it means that the installation of the Apache web server software on this system was successful. You may now add content to this directory and replace this page."

5.6 Package Managers

As you have seen, building software from source can be time consuming and involve a lot of troubleshooting if the build process doesn't go as planned. This process can be greatly simplified through the use of **packages**. Many Linux distributions maintain a set of **repositories** that contain pre-compiled binary packages for the popular architectures like x86. Since these packages are already compiled, all that's left is to install them. You can download these packages and install them with the appropriate package handler. Keep in mind that this does not save you from having to satisfy dependencies. If you download a package that depends on other packages, you must install them first. This often leads to a situation know as **dependency hell**.

This process can be simplified even further through the use of **package managers**. These programs provide ways of quickly searching the package repositories for desired software. After you find what you want, the package manager will look at its dependencies, find those packages in the repository, continuing until it finds all required packages. You can then tell it to go ahead and install them and it will proceed to download all the necessary packages and unpack them to the proper locations.

One such system is the **APT** package management system for Debian Linux and its derivatives. This is possibly the most powerful package management system available, providing a powerful command line interface, in addition to well made GUIs that can run both in a terminal (**aptitude**) and in a graphical system (**synaptic**).