Compiling Programs

- In this chapter, we will look at how to build programs by compiling source code.
- The availability of source code is the essential freedom that makes Linux possible.
- The entire ecosystem of Linux development relies on free exchange between developers.
- We will introduce a new command:
  - make – Utility to maintain programs

What is Compiling?

- compiling is the process of translating source code (the human-readable description of a program written by a programmer) into the native language of the computer’s processor.
- The computer’s processor (or CPU) works at an elemental level, executing programs in what is called machine language.
- This is a numeric code that describes extremely small operations, such as “add this byte,” “point to this location in memory,” or “copy this byte.” Each of these instructions is expressed in binary (ones and zeros).

Are All Programs Compiled?

- No. As we have seen, there are programs such as shell scripts that do not require compiling.
- They are executed directly. These are written in what are known as scripting or interpreted languages.
- These languages have grown in popularity in recent years and include Perl, Python, PHP, Ruby, and many others.
- Scripted languages are executed by a special program called an interpreter.
- An interpreter inputs the program file and reads and executes each instruction contained within it.
- In general, interpreted programs execute much more slowly than compiled programs. This is because each source code instruction in an interpreted program is translated every time it is carried out, whereas with a compiled program, a source code instruction is only translated once, and this translation is permanently recorded in the final executable file.

Compiling a C Program

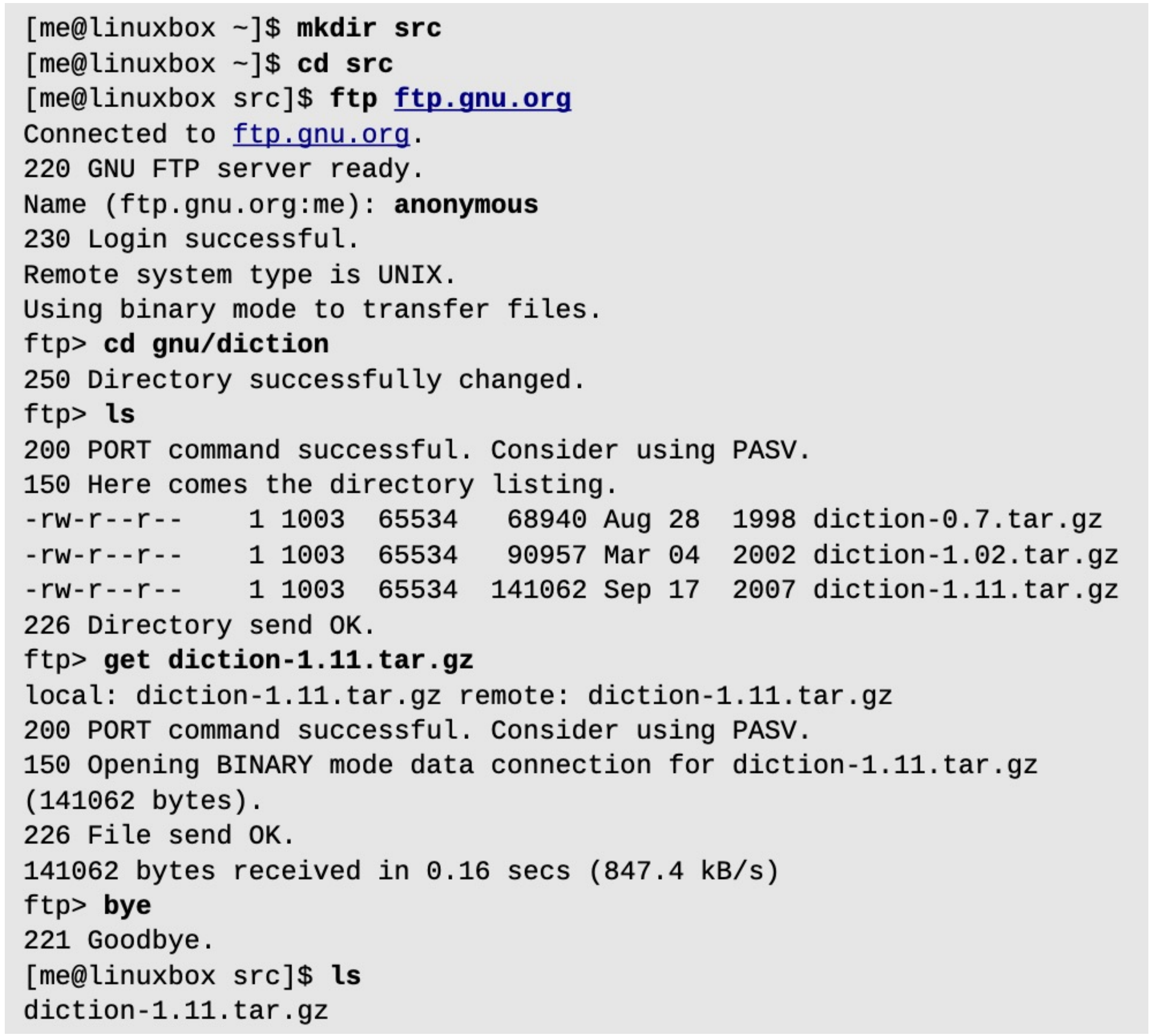
- Let’s compile something. Before we do that, however, we’re going to need some tools like the compiler, the linker, and make. The C compiler used almost universally in the Linux environment is called gcc (GNU C Compiler), originally written by Richard Stallman.
- Most distributions do not install gcc by default. We can check to see whether the compiler is present like this:

```
[me@linuxbox ~]$ which gcc
/usr/bin/gcc
```

- The results in this example indicate that the compiler is installed.

Obtaining the Source Code

- For our compiling exercise, we are going to compile a program from the GNU Project called diction. This handy little program checks text files for writing quality and style.
- As programs go, it is fairly small and easy to build.
- Following convention, we're first going to create a directory for our source code named src and then download the source code into it using ftp.



```
[me@linuxbox ~]$ mkdir src
[me@linuxbox ~]$ cd src
[me@linuxbox src]$ ftp ftp.gnu.org
Connected to ftp.gnu.org.
220 GNU FTP server ready.
Name (ftp.gnu.org:me): anonymous
230 Login successful.
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> cd gnu/diction
250 Directory successfully changed.
ftp> ls
200 PORT command successful. Consider using PASV.
150 Here comes the directory listing.
-rw-r--r--    1 1003   65534    68940 Aug 28  1998 diction-0.7.tar.gz
-rw-r--r--    1 1003   65534    90957 Mar 04  2002 diction-1.02.tar.gz
-rw-r--r--    1 1003   65534   141062 Sep 17  2007 diction-1.11.tar.gz
226 Directory send OK.
ftp> get diction-1.11.tar.gz
local: diction-1.11.tar.gz remote: diction-1.11.tar.gz
200 PORT command successful. Consider using PASV.
150 Opening BINARY mode data connection for diction-1.11.tar.gz
(141062 bytes).
226 File send OK.
141062 bytes received in 0.16 secs (847.4 kB/s)
ftp> bye
221 Goodbye.
[me@linuxbox src]$ ls
diction-1.11.tar.gz
```

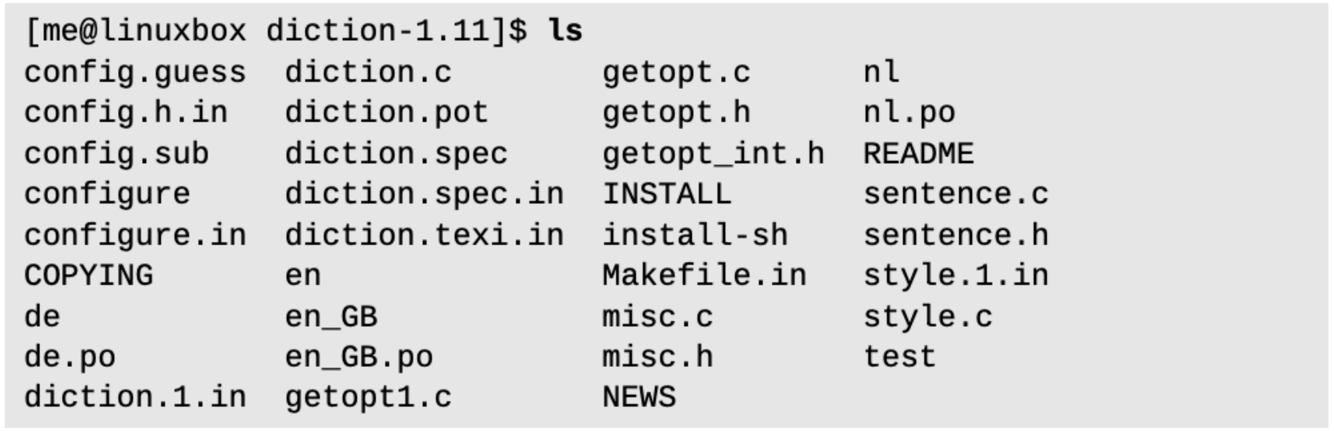
- While we used ftp in the previous example, which is traditional, there are other ways of downloading source code. For example, the GNU Project also supports downloading using HTTPS. We can download the diction source code using the curl program.

```
[me@linuxbox ~]$ curl -O https://ftp.gnu.org/gnu/diction/diction-1.11.tar.gz
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  
Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  137k  100  137k    0     0   652k      0 --:--:-- --:--:-- --:--:-- 652k
```

- As we can see, source code is usually supplied in the form of a compressed tar file. Sometimes called a tarball, this file contains the source tree, or hierarchy of directories and files that comprise the source code.
- After arriving at the ftp site, we examine the list of tar files available and select the newest version for download.
- Using the get command within ftp, we copy the file from the ftp server to the local machine. Once the tar file is downloaded, it must be unpacked. This is done with the tar program.

Examining the Source Tree

- Unpacking the tar file results in the creation of a new directory, named diction-1.11.
- This directory contains the source tree. Let’s look inside.
- `$ cd diction-1.11`
- In it, we see a number of files. Programs belonging to the GNU Project, as well as manyothers, will supply the documentation files README, INSTALL, NEWS, and COPYING.
- These files contain the description of the program, information on how to build and install it, and its licensing terms. It is always a good idea to carefully read the README and INSTALL files before attempting to build the program.



```
[me@linuxbox diction-1.11]$ ls
config.guess  diction.c         getopt.c       nl
config.h.in   diction.pot       getopt.h       nl.po
config.sub    diction.spec      getopt_int.h   README
configure     diction.spec.in   INSTALL        sentence.c
configure.in  diction.texi.in   install-sh     sentence.h
COPYING       en                Makefile.in    style.1.in
de            en_GB             misc.c         style.c
de.po         en_GB.po          misc.h         test
diction.1.in  getopt1.c         NEWS
```

- The other interesting files in this directory are the ones ending with .c and .h.

```
[me@linuxbox diction-1.11]$ ls *.c
diction.c  getopt1.c  getopt.c  misc.c  sentence.c  style.c
[me@linuxbox diction-1.11]$ ls *.h
getopt.h  getopt_int.h  misc.h  sentence.h
```

- The .c files contain the two C programs supplied by the package (style and diction), divided into modules. It is common practice for large programs to be broken into smaller, easier-to-manage pieces. The source code files are ordinary text and can be examined with less.

```
[me@linuxbox diction-1.11]$ less diction.c
```

- The .h files are known as header files. These, too, are ordinary text. Header files contain descriptions of the routines included in a source code file or library. For the compiler to connect the modules, it must receive a description of all the modules needed to complete the entire program. Near the beginning of the diction.c file, we see this line:

```
#include "getopt.h"
```

- This instructs the compiler to read the file getopt.h as it reads the source code indiction.c to “know” what’s in getopt.c. The getopt.c file supplies routines that are shared by both the style and diction programs.

- Before the include statement for getopt.h, we see some other include statements such as these:

```
#include <regex.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
```

- These also refer to header files, but they refer to header files that live outside the currentsource tree. They are supplied by the system to support the compilation of every program.
- If we look in /usr/include, we can see them.
- The header files in this directory were installed when we installed the compiler.

```
[me@linuxbox diction-1.11]$ ls /usr/include
```

Building the Program

- Most programs build with a simple, two-command sequence.

```
./configure
make
```

- The configure program is a shell script that is supplied with the source tree. Its job is to analyze the build environment. Most source code is designed to be portable.
- That is, it is designed to build on more than one kind of Unix-like system. But to do that, the source code may need to undergo slight adjustments during the build to accommodate differences between systems. configure also checks to see that necessary external tools and components are installed.
- Let’s run configure. Since configure is not located where the shell normally expects programs to be located, we must explicitly tell the shell its location by prefixing the command with ./ to indicate that the program is located in the current working directory.

```
[me@linuxbox diction-1.11]$ ./configure
```

- configure will output a lot of messages as it tests and configures the build. When it finishes, it will look something like this:

```
checking libintl.h presence... yes
checking for libintl.h... yes
checking for library containing gettext... none required
configure: creating ./config.status
config.status: creating Makefile
config.status: creating diction.1
config.status: creating diction.texi
config.status: creating diction.spec
config.status: creating style.1
config.status: creating test/rundiction
config.status: creating config.h
[me@linuxbox diction-1.11]$
```

- What’s important here is that there are no error messages. If there were, the configuration
- failed, and the program will not build until the errors are corrected.
- We see configure created several new files in our source directory. The most important one is the makefile.
- The makefile is a configuration file that instructs the make program exactly how to build the program. Without it, make will refuse to run. The makefile is an ordinary text file, so we can view it:

```
[me@linuxbox diction-1.11]$ less Makefile
```

- The make program takes as input a makefile (which is normally named Makefile), which describes the relationships and dependencies among the components that comprise the finished program.
- The first part of the makefile defines variables that are substituted in later sections of the makefile. For example we see the following line:

```
CC=             gcc
```

- That defines the C compiler to be gcc. Later in the makefile, we see one instance where it gets used.

```
diction:        diction.o sentence.o misc.o getopt.o getopt1.o
                $(CC) -o $@ $(LDFLAGS) diction.o sentence.o misc.o \
                getopt.o getopt1.o $(LIBS)
```

- A substitution is performed here, and the value $(CC) is replaced by gcc at runtime.
- Most of the makefile consists of lines, which define a target, in this case the executablefile diction and the files on which it is dependent. The remaining lines describe the commands needed to create the target from its components.

- We see in this example that the executable file diction (one of the end products) depends on the existence of diction.o, sentence.o, misc.o, getopt.o, and getopt1.o. Later, in the makefile, we see definitions of each of these as targets.

```
diction.o:      diction.c config.h getopt.h misc.h sentence.h
getopt.o:       getopt.c getopt.h getopt_int.h
getopt1.o:      getopt1.c getopt.h getopt_int.h
misc.o:         misc.c config.h misc.h
sentence.o:     sentence.c config.h misc.h sentence.h
style.o:        style.c config.h getopt.h misc.h sentence.h
```

- However, we don’t see any command specified for them. This is handled by a general target, earlier in the file, that describes the command used to compile any .c file into a .o file.

```
.c.o:
                $(CC) -c $(CPPFLAGS) $(CFLAGS) $<
```

- This all seems very complicated. Why not simply list all the steps to compile the parts and be done with it? The answer to this will become clear in a moment. In the meantime, let’s run make and build our programs.

```
[me@linuxbox diction-1.11]$ make
```

- The make program will run, using the contents of Makefile to guide its actions. It will produce a lot of messages.
- When it finishes, we will see that all the targets are now present in our directory.

```
[me@linuxbox diction-1.11]$ ls
```

```
config.guess   de.po            en            install-sh   sentence.c
config.h       diction          en_GB         Makefile     sentence.h
config.h.in    diction.1        en_GB.mo      Makefile.in  sentence.o
config.log     diction.1.in     en_GB.po      misc.c       style
config.status  diction.c        getopt1.c     misc.h       style.1
config.sub     diction.o        getopt1.o     misc.o       style.1.in
configure      diction.pot      getopt.c      NEWS         style.c
configure.in   diction.spec     getopt.h      nl           style.o
COPYING        diction.spec.in  getopt_int.h  nl.mo        test
de             diction.texi     getopt.o      nl.po
de.mo          diction.texi.in  INSTALL       README
```

- Among the files, we see diction and style, the programs that we set out to build.
- Congratulations are in order! We just compiled our first programs from source code!

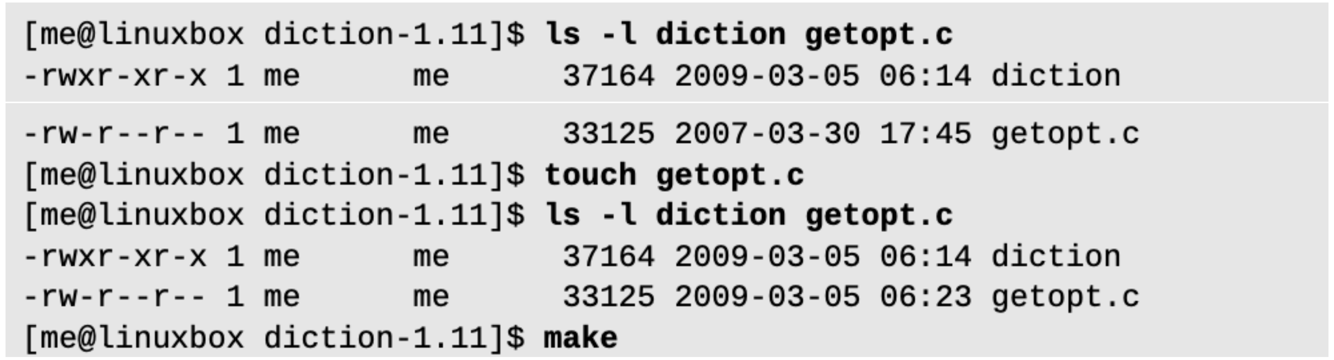
- let’s run make again.

```
[me@linuxbox diction-1.11]$ make
make: Nothing to be done for `all'.
```

- It only produces this strange message. What’s going on? Why didn’t it build the program again? Ah, this is the magic of make.
- Rather than simply building everything again, make only builds what needs building. With all of the targets present, make determined that there was nothing to do.
- We can demonstrate this by deleting one of the targets and running make again to see what it does. Let’s get rid of one of the intermediate targets.

```
[me@linuxbox diction-1.11]$ rm getopt.o
[me@linuxbox diction-1.11]$ make
```

- We see that make rebuilds it and re-links the diction and style programs, since they depend on the missing module.
- This behavior also points out another important feature of make: it keeps targets up-to-date. make insists that targets be newer than their dependencies.
- This makes perfect sense, because a programmer will often update a bit of source code and then use make to build a new version of the finished product. make ensures that everything that needs building based on the updated code is built.
- If we use the touch program to “update” one of the source code files, we can see this happen:



```
[me@linuxbox diction-1.11]$ ls -l diction getopt.c
-rwxr-xr-x 1 me      me      37164 2009-03-05 06:14 diction
-rw-r--r-- 1 me      me      33125 2007-03-30 17:45 getopt.c
[me@linuxbox diction-1.11]$ touch getopt.c
[me@linuxbox diction-1.11]$ ls -l diction getopt.c
-rwxr-xr-x 1 me      me      37164 2009-03-05 06:14 diction
-rw-r--r-- 1 me      me      33125 2009-03-05 06:23 getopt.c
[me@linuxbox diction-1.11]$ make
```

- After make runs, we see that it has restored the target to being newer than the dependency:

```
[me@linuxbox diction-1.11]$ ls -l diction getopt.c
-rwxr-xr-x 1 me      me      37164 2009-03-05 06:24 diction
-rw-r--r-- 1 me      me      33125 2009-03-05 06:23 getopt.c
```

- The ability of make to intelligently build only what needs building is a great benefit to programmers.
- While the time savings may not be apparent with our small project, it is very significant with larger projects. Remember, the Linux kernel (a program that undergoes continuous modification and improvement) contains several million lines of code.

Installing the Program

- Well-packaged source code will often include a special make target called install.
- This target will install the final product in a system directory for use. Usually, this directory is /usr/local/bin, the traditional location for locally built software.
- However, this directory is not normally writable by ordinary users, so we must become the superuser to perform the installation.

```
[me@linuxbox diction-1.11]$ sudo make install
```

- After we perform the installation, we can check that the program is ready to go.

```
[me@linuxbox diction-1.11]$ which diction
/usr/local/bin/diction
[me@linuxbox diction-1.11]$ man diction
```