Makefiles for Dummies

Luca Abeni
luca.abeni@unitn.it

March 3, 2008

Abstract

These short notes describe how to easily write makefiles for compiling C/C++ projects composed by multiple files. ... [to be continued]

1 Introduction

To manage the complexity of the code, the sources of large software projects are often organised in different files (a single file often corresponds to a software module). Groups of source files can be compiled independently, and the resulting objects are then linked together obtaining the final executable. In particular, C projects are composed by `.c` files (containing source code) and `.h` files (describing software interfaces): each `.c` file is compiled (together with the used headers) to obtain `.o` object files, and the `.o` files are linked together with some libraries to obtain the *executable file*. The standard `make` program is a tool designed to automate this build process, keeping track of the dependencies between source files, object files, and executables, and recompiling files only when really needed.

The behaviour of the `make` program is controlled through a `makefile`, containing a description of all the dependencies and building rules needed to compile the final executable (informally speaking, a makefile is a collection of instructions that should be followed to compile your program). The big difference between using make respect to a shell script is that when some source files are modified the "make" command is able to compile only the needed files (instead of recompiling all the sources, as a shell script). In other word, the program will be recompiled using as few compilation commands as possible. To achieve this goal, you need to supply the rules for compiling various files and file types, and the list of dependencies between files (in the form of relationships like "if file `A` was changed, then files `B` and `C` also need to be re-compiled", or similar). Writing a `makefile` containing this kind of information is generally quite simple, but there is the risk to end up with long lists of dependencies and rules which can look difficult to maintain.

The GNU make comes with a set of predefined rules which help in reducing the size and complexity of makefiles, and should be used to write effective and simple makefiles. [to be continued]

```
<target>: <prerequisites>
        <command>
```

Figure 1: A Makefile rule

```
test: a.o b.o c.o
        gcc -o test a.o b.o c.o
a.o: a.c
        gcc -Wall -g -c a.c
b.o: b.c
        gcc -Wall -g -c b.c
c.o: c.c
        gcc -Wall -g -c c.c
clean:
        rm -f test a.o b.o c.o
```

Figure 2: An example of simple Makefile

2 Makefiles Structure

As already observed, makefile is mainly a collection of *rules* describing dependencies between *prerequisites* and *targets*, and the commands needed to generate a target from its prerequisites. Figure 2 describes a generic rule, where:

- `<target>` is a name for the action executed by the rule, or (more frequently!) the name of a file generated by the rule. Example of targets are `.o` object files, executable files, etc..., but also `clean`, `install`, etc...
- `<prerequisites>` is a list of files used to build the target
- `<command>` is a description of the action executed by the rule (sometimes, more than one command).

Note that each command line begins with the `<tab>` character.

When considering a rule, the `make` program checks is the prerequisites are newer than the target: in such case, the target is rebuilt by executing the command. If the rules has no prerequisites, the target is always rebuilt. If the rule has some prerequisites, make checks if the have to be rebuilt in a recursive way, by checking the rules that have them as targets.

When the `make` command is executed with no arguments, it starts by considering the first rule in the makefile (sometimes known as *default rule*) and trying to build it. To do so, it consults the rules for building all the prerequisites, and so on... The user can select an alternative target to build instead of the default one by passing such target as a parameter to `make`.

An example makefile, building an executable "`test`" from the source files "`a.c`", "`b.c`", and "`c.c`" is shown in Figure 2: the default rule (with target

```
test: a.o b.o c.o
	gcc -o test a.o b.o c.o
a.o: a.c b.h c.h
	gcc -Wall -g -c a.c
b.o: b.c b.h
	gcc -Wall -g -c b.c
c.o: c.c c.h
	gcc -Wall -g -c c.c
clean:
	rm -f test a.o b.o c.o
```

Figure 3: A more correct Makefile example

`test`) shows how to build the executable from the single `.o` object files, and the following 3 rules show how to build the object files from the sources. Finally, the last rule (with target `clean`) permits to remove all the generated files.

Now, it is important to note that the rules generating the objects files are not fully correct: for example, if module `a.o` uses some functionalities from the `b.o` module, it must include the `b.h` header file, but this dependency is not modelled in the makefile. As a result, if `b.h` is modified `a.o` is not rebuilt (to understand why this is a problem, consider the case when `b.h` contains something like `#define MAX_VALUE 10`, and `a.c` contains something like `int values[MAX_VALUE]`... What happens if `MAX_VALUE` is changed from `10` to `100`?). In this case, a more correct makefile can be the one shown in Figure 2 (assuming that `a.c` uses `b.c` and `c.c`).

Note that tracking all the dependencies in a makefile is not easy, and requires a lot of maintenance work (for example, a modification to the program during the development or debugging can change the dependencies).

3 Writing Simpler Makefiles

Fortunately, GNU make provides some ways to simplify the makefiles, and to make them more manageable. Such simplifications are mainly based on two techniques: *makefile variables* and `implicit rules`.

3.1 Makefile Variables

The concept of variables is not a GNU-only feature, but is supported by every POSIX compliant `make`. In general, a variable can contain a single value or a list of values (file names, compiler options, etc...), is assigned using a statement like `<variable name> = ...`, and is dereferenced by prepending its name with the `$` symbol. See Figure 3.1 for an example (note that if the variable name is longer than 1 character, the it must be enclosed in parenthesys when dereferencing the variable).

```
VAR=test

print:
	echo $(VAR)
```

Figure 4: Using variables in a makefile

```
CFLAGS = -Wall -g
LDLIBS = -lefence
test: test.o b.o c.o
clean:
	rm -f test a.o b.o c.o
```

Figure 5: An example of simple Makefile

3.2 Implicit Rules

To avoid the need to repeat similar rules in all the makefiles, GNU Make provides some implicit rules, which automatically implement standard techniques for building some targets. For example, there are implicit rules for building executables files from object files, or to compile `.c` source files into `.o` objects.

Implicit rules use some default makefile variables so that, by changing the values of the variables, it is possible to change the way the implicit rule works. The most important of such variables are:

- `CPPFLAGS`: the parameters to be passed to the C preprocessor (for example, `-I <path>`, `-D <symbol>`, etc.)
- `CFLAGS`: the parameters to be passed to the C compiler (for example, `-Wall`, `-g`, etc)
- `CXXFLAGS`: the parameters to be passed to the C++ compiler
- `LDFLAGS`: the parameters to be passed to the linker (for example, `-L <path>`, etc)
- `LDLIBS`: the libraries that have to be linked into the executable (for example, `-lm`, etc)

Thanks to the implicit rules, GNU make knows how to generate a `.o` file from a `.c` file, a `.cc` (or `.cpp`, or...) file, a `.s` file, etc, so there is no need to write targets like the `a.o: ...` target in Figures 2 and 2.

GNU Make also knows how to generate executable files from `.o` files, when the first object file in the prerequisites is `<executable name>.o` (you just need to specify the prerequisites for the target executable). So, a makefile for generating the `test` program from `test.o`, `b.o`, and `c.o` can look like the one listed in Figure 3.2 (the dependencies on `.h` header files are not specifies yet).

```
%.d: %.c
	$(CC) -MM -MF $@ $<
```

Figure 6: Generating dependencies from a C file

It is worth noting some important points:

- **Never** set `-I` or `-D` in `CFLAGS`. They belong to `CPPFLAGS`
- If you need to set `-I .` in `CPPFLAGS`, you probably wrote wrong `#include` directives in yoyr code
- **Always** set `-Wall` in `CFLAGS`. Compiler warnings are very useful to detect bugs or anomalies in your programs
- When you compile C++ programs, you must use `CXXFLAGS`, not `CFLAGS`
- Setting `-g` in `CFLAGS` is generally a good idea. The generated code will not be affected, but the object and executable files will contain useful debug information, making the `gdb` output less cryptic
- When compiling programs that use pthreads, you probably need to set `-pthread` in `CFLAGS` and `LDFLAGS` (in this case, you do not need to set `-lpthread` in `LDLIBS`). If `-pthread` is not supported by your compiler (this probably means that you are on windows), you need to set `-lpthread` in `LDLIBS`, and some `-D` options (probably `-DREENTRANT` and some others, depending on your C library) in `CPPFLAGS`

Finally, note that in some cases you do not even have to write a `makefile`: for example, if you want to compile the C file `test.c` generating an executable file named `test`, you can just type `make test`. Of course you want to properly set `CFLAGS` (and maybe other options), so you must use something like `make CFLAGS="-Wall -g" test`.

4 Automatic Dependencies Generation

The dependencies of `.o` object files from `.h` header files are generally quite complex, and difficult to keep up-to-date. Fortunately, they can be automatically generated by instructing `gcc` to look at the `#include` directives. This is done by using the `-MM` gcc option (plus the `-MF` option to specify an output file); see the gcc manual (or type `man gcc` in a linux shell) for more details.

In few words, the command line `gcc -MM -MF <filename>.d <filename>.c` reads the `<filename>.c` file and creates a `<filename>.d` file containing all the dependencies on header files included with `#include "..."` (headers included with `#include <...>` are system files and do not need to be tracked). The corresponding makefile rule is shown in Figure 4: note that `%.d: %.c` is a

```
CFLAGS = -Wall -g
LDLIBS = -lefence

OBJS = test.o b.o c.o

test: $(OBJS)

clean:
        rm -f *.o *.d test

%.d: %.c
        $(CC) -MM -MF $@ $<
-include $(OBJS:.o=.d)
```

Figure 7: Generating dependencies from a C file

```
obj-m = test_module.o
test_module-objs = file1.o file2.o test.o
```

Figure 8: A simple makefile for compiling a Linux kernel module

way to say "a .d file is generate from the corresponding .c file", `$(CC)` is a predefined variable containing the name of the C compiler, `$@` is a variable containing the target, and `$<` is a variable containing the prerequisites. As a result, a complete example of simple makefile is listed in Figure 4 (note that `include $(OBJS:.o=.d)` is some black magic needed to include all the `.d` files corresponding to the `.o` files in `$OBJS`).

5 Compiling Linux Kernel Modules

Compiling Linux kernel modules is much more complex than compiling regular user-space applications: for example, `CFLAGS` and `CPPFLAGS` must be set with some proper options (depending on the Linux kernel, and even on its configurations), and some special post-linking operations have to be performed.

Fortunately, the Linux kernel already provides a set of proper makefiles and scripts (known as `KBuild` system) that can be used to compile generic modules.

The simplest way to use KBuild to compile a Linux kernel module is to write a makefile similar to the one in Figure 5 (showing how to build a module named `test_module.ko` from the source files `file1.c`, `file2.c`, and `test.c`), and to type `make -C <path to configured kernel sources> M=<path to the module source> modules` (note that to use KBuild you need some configured Linux sources somewhere).

For more information about KBuild, see the Documentation/kbuild direc-

tory inside the kernel source.

6 References

This document does not pretend to provide an exhaustive description of the `make` program. For more information, consult the make manual (`http://www.gnu.org/software/make/manual`), or type `info make` in a linux shell.