

Introduction to `make`

Marius Nita

CS Tutors – Portland State University

May 3, 2005

1 Introduction

`make` is a UNIX tool that facilitates working with projects which are composed of multiple files. It can be instructed to execute a sequence of commands and resolve dependencies between files. Once the project is compiled, `make` is smart enough to only recompile those files which have been changed in the mean time, thus saving time.

This document gives a very brief introduction to `make`, discussing enough material to enable a student completely unfamiliar with the tool to get started with using it in coding projects.

2 Example

Let's assume that we have a project composed of files `one.c`, `two.c` and `inc.h` in a project directory. The files `one.c` and `two.c` include the file `inc.h` with an `#include` statement. Our directory looks like this:

```
$ ls
inc.h  one.c  two.c
```

We need a way to manage the compilation process so that we don't have to recompile the whole project every time we change one file, and we want to compile everything with one quick command. We create a file named `Makefile` in the current directory, with the following contents¹: (The meaning of this file will be explained below.)

```
proj: one.o two.o
      gcc -o proj one.o two.o
one.o: inc.h one.c
      gcc -c one.c
two.o: inc.h two.c
      gcc -c two.c
clean:
      rm -f *.o core proj
```

¹The indented lines have to be indented by exactly one tab. They cannot be indented by 8 spaces. This is a hard rule that cannot be circumvented.

Now that our `Makefile` is in the current directory, we type `make`:

```
$ make
gcc -c one.c
gcc -c two.c
gcc -o proj one.o two.o
$ ls
Makefile      one.c      proj*      two.o
inc.h         one.o     two.c
```

A binary by the name of `proj` has been created; this is our program. So, with one simple command, we are able to compile the entire project without worrying about getting the compiler arguments right, the order in which we compile, etc. Now, say that we need to make a change to `one.c`. We change the file, save it, and then type `make` again:

```
$ make
gcc -c one.c
gcc -o proj one.o two.o
```

Note that this time, the file `two.c` has not been recompiled. Its old object file, `two.o`, was reused in the linking process, since it is still valid; its source file has not changed. Now, say we want to clean up the project directory to where we only have source files, so we can tar it up and submit it. We type `make clean`:

```
$ make clean
rm -f *.o core proj
$ ls
Makefile      inc.h      one.c      two.c
```

And our project is ready to be submitted.

3 How does it work?

When executed, the `make` program looks for a special file named `Makefile` in the current directory, typically provided by the author of the project being built. It then reads this `Makefile` and builds the project according to its contents. A typical `Makefile` is composed of a series of rules. The following is a rule:

```
one.o: inc.h one.c
      gcc -c one.c
```

3.1 Rule components

A rule has three main components: a *target*, a list of *dependencies* and a *command*. The three are represented in a rule as follows:

TARGET: DEPENDENCIES
COMMAND

The **target** is typically the name of a file which you want to create. In the example above, `one.o` is our target.

The **dependency list** is typically a list of files that our *target* needs in order to be built. Therefore, before it attempts to build our target, **make** will make sure that these files either already exist, or can be created from other rules which have them as targets. In our example, **make** will make sure that `inc.h` and `one.c` exist before it attempts to build the target. If these files are missing, and there are no rules which have them as targets, **make** will fail. Furthermore, if any of the files in the dependency list have been changed since the last build, the target will be rebuilt. A well designed dependency graph will keep a project sane and will prevent errors which result from linking against old code. The dependency list is optional.

The **command** is typically a shell command that **make** executes in order to build the target. In our example, `gcc -c one.c` will be executed in order to create our target, `one.o`. The command portion of the rule is optional.

3.2 Selecting rules

Typically, **make** takes a target name as an argument. For example, if all we want to do is build `one.o`, we would type `make one.o`:

```
$ make one.o  
gcc -c one.c
```

And **make** will build that target and exit. When we type **make** without an argument, **make** will attempt to build the target named `all`. If a rule for `all` doesn't exist, **make** will build the topmost target in your **Makefile**. Typically, you will see a rule for `all` that looks like this:

```
all: proj
```

This simply says that the implied target is `proj` and is useful when your main target is not at the top of the file, or simply for readability.

3.3 The clean target

Notice that our **Makefile** from the previous section has a peculiar target named `clean` with no dependency list. The `clean` target is a widespread convention for providing a rule which cleans up the directory and revert it to its source-only state, before any builds were performed. The command for this rule is usually a `rm` command which removes any object files, binaries, core dumps, temporary files, etc. This allows us to simply type `make clean` to start fresh.

4 Further reading

After reading the above, you should be able to write your own Makefiles for your small-to-medium size school projects. `make`, however, is a complex tool with a wealth of features, which can be used to manage projects with thousands of files. To find out more, read the GNU make manual: <http://www.gnu.org/manual/make-3.80/make.html>

On our UNIX systems, the default `make` command is different than GNU `make`. Though the two are very similar, GNU `make` has many more features. GNU `make` is available as `gmake` by adding the GNU package with the `addpkg` command. The examples in this tutorial work with both versions.

5 Further help

Email your questions to CS tutors at tutors@cs.pdx.edu, or join our IRC channel, `#cschat` on irc.cat.pdx.edu.