GNU Make in Detail for Beginners

By **Sarath Lakshman** - June 9, 2012

Have you ever peeked into the source code of any of the applications you run every day? Ever used *make install* to install some application? You will see make in most projects. It enables developers to easily compile large and complex programs with many components. It's also used for writing maintenance scripts based on timestamps. This article shows you how to have fun with make.

Large projects can contain thousands of lines of code, distributed in multiple source files, written by many developers and arranged in several subdirectories. A project may contain several component divisions. These components may have complex inter-dependencies — for example, in order to compile component X, you have to first compile Y; in order to compile Y, you have to first compile Z; and so on. For a large project, when a few changes are made to the source, manually recompiling the entire project each time is tedious, error-prone and time-consuming.

Make is a solution to these problems. It can be used to specify dependencies between components, so that it will compile components in the order required to satisfy dependencies. An important feature is that when a project is recompiled after a few changes, it will recompile only the files which are changed, and any components that are dependent on it. This saves a lot of time. Make is, therefore, an essential tool for a large software project.

Each project needs a Makefile — a script that describes the project structure, namely, the source code files, the dependencies between them, compiler arguments, and how to produce the target output (normally, one or more executables). Whenever the make command is executed, the Makefile in the current working directory is interpreted, and the instructions executed to produce the target outputs. The Makefile contains a collection of rules, macros, variable assignments, etc. ('Makefile' or 'makefile' are both acceptable.)

Installing GNU Make

Most distributions don't ship make as part of the default installation. You have to install it, either using the package-management system, or by manually compiling from source. To compile and build from source, download the tarball, extract it, and go through the README file. (If you're running Ubuntu, you can install make as well as some other common packages required for building from source, by running: `sudo apt-get install build-essential` .)

A sample project

To acquaint ourselves with the basics of make, let's use a simple C "Hello world" project, and a Makefile that handles building of the target binary. We have three files (below): `module.h` , the header file that contains the declarations; `module.c` , which contains the definition of the function defined in `module.h` ; and the main file, `main.c` , in which we call

the `sample_func()` defined in `module.c`. Since `module.h` includes the required header files like `stdio.h`, we don't need to include `stdio.h` in every module; instead, we just include `module.h`. Here, `module.c` and `main.c` can be compiled as separate object modules, and can be linked by GCC to obtain the target binary.

module.h:

```
#include <stdio.h>
void sample_func();
```

module.c:

```
#include "module.h"
void sample_func()
{
    printf("Hello world!");
}
```

main.c:

```
#include "module.h"
void sample_func();
int main()
{
    sample_func();
    return 0;
}
```

The following are the manual steps to compile the project and produce the target binary:

```
slynux@freedom:~$ gcc -I . -c main.c # Obtain main.o
slynux@freedom:~$ gcc -I . -c module.c # Obtain module.o
slynux@freedom:~$ gcc main.o module.o -o target_bin #Obtain target binary
```

(`-I` is used to include the current directory (`.`) as a header file location.)

Writing a Makefile from scratch

By convention, all variable names used in a Makefile are in upper-case. A common variable assignment in a Makefile is `CC = gcc`, which can then be used later on as `${CC}` or `$(CC)`. Makefiles use `#` as the comment-start marker, just like in shell scripts.

The general syntax of a Makefile rule is as follows:

```
target: dependency1 dependency2 ...
[TAB] action1
[TAB] action2
    ...
```

Let's take a look at a simple Makefile for our sample project:

```
all: main.o module.o
    gcc main.o module.o -o target_bin
main.o: main.c module.h
    gcc -I . -c main.c
module.o: module.c module.h
    gcc -I . -c module.c
clean:
    rm -rf *.o
    rm target_bin
```

We have four targets in the Makefile:

- `all` is a special target that depends on `main.o` and `module.o`, and has the command (from the "manual" steps earlier) to make GCC link the two object files into the final executable binary.
- `main.o` is a filename target that depends on `main.c` and `module.h`, and has the command to compile `main.c` to produce `main.o`.
- `module.o` is a filename target that depends on `module.c` and `module.h`; it calls GCC to compile the `module.c` file to produce `module.o`.
- `clean` is a special target that has no dependencies, but specifies the commands to clean the compilation outputs from the project directories.

You may be wondering why the order of the make targets and commands in the Makefile are not the same as that of the manual compilation commands we ran earlier. The reason is so that the easiest invocation, by just calling the make command, will result in the most commonly desired output — the final executable. How does this work?

The make command accepts a target parameter (one of those defined in the Makefile), so the generic command line syntax is `make <target>`. However, make also works if you do not specify any target on the command line, saving you a little typing; in such a case, it defaults to the first target defined in the Makefile. In our Makefile, that is the target `all`, which results in the creation of the desired executable binary `target_bin`!

Makefile processing, in general

When the make command is executed, it looks for a file named `makefile` or `Makefile` in the current directory. It parses the found `Makefile`, and constructs a dependency tree. Based on the desired make target specified (or implied) on the command-line, make checks if the dependency files of that target exist. And (for filename targets — explained below) if they exist, whether they are newer than the target itself, by comparing file timestamps.

Before executing the action (commands) corresponding to the desired target, its dependencies must be met; when they are not met, the targets corresponding to the unmet dependencies are executed before the given make target, to supply the missing dependencies.

When a target is a filename, make compares the timestamps of the target file and its dependency files. If the dependency filename is another target in the Makefile, make then checks the timestamps of that target's dependencies. It thus winds up recursively checking all the way down the dependency tree, to the source code files, to see if any of the files in the dependency tree are newer than their target filenames. (Of course, if the dependency files don't exist, then make knows it must start executing the make targets from the "lowest" point in the dependency tree, to create them.)

If make finds that files in the dependency tree are newer than their target, then all the targets in the affected branch of the tree are executed, starting from the "lowest", to update the dependency files. When make finally returns from its recursive checking of the tree, it

completes the final comparison for the desired make target. If the dependency files are newer than the target (which is usually the case), it runs the command(s) for the desired make target.

This process is how make saves time, by executing only commands that need to be executed, based on which of the source files (listed as dependencies) have been updated, and have a newer timestamp than their target.

Now, when a target is not a filename (like all and clean in our Makefile, which we called "special targets"), make obviously cannot compare timestamps to check whether the target's dependencies are newer. Therefore, such a target is always executed, if specified (or implied) on the command line.

For the execution of each target, make prints the actions while executing them. Note that each of the actions (shell commands written on a line) are executed in a separate sub-shell. If an action changes the shell environment, such a change is restricted to the sub-shell for that action line only. For example, if one action line contains a command like `cd newdir`, the current directory will be changed only for that line/action; for the next line/action, the current directory will be unchanged.

Processing our Makefile

After understanding how make processes Makefiles, let's run make on our own Makefile, and see how it is processed to illustrate how it works. In the project directory, we run the following command:

```
slynux@freedom:~$ make
gcc -I . -c main.c
gcc -I . -c module.c
gcc main.o module.o -o target_bin
```

What has happened here?

When we ran make without specifying a target on the command line, it defaulted to the first target in our Makefile — that is, the target `all`. This target's dependencies are `module.o` and `main.o`. Since these files do not exist on our first run of make for this project, make notes that it must execute the targets `main.o` and `module.o`. These targets, in turn, produce the `main.o` and `module.o` files by executing the corresponding actions/commands. Finally, make executes the command for the target `all`. Thus, we obtain our desired output, `target_bin`.

If we immediately run make again, without changing any of the source files, we will see that only the command for the target `all` is executed:

```
slynux@freedom:~$ make
gcc main.o module.o -o target_bin
```

Though make checked the dependency tree, neither of the dependency targets (`module.o` and `main.o`) had their own dependency files bearing a later timestamp than the dependency target filename. Therefore, make rightly did not execute the commands for the

dependency targets. As we mentioned earlier, since the target `all` is not a filename, make cannot compare file timestamps, and thus executes the action/command for this target.

Now, we update `module.c` by adding a statement `printf("\nfirst update");` inside the `sample_func()` function. We then run make again:

```
slynux@freedom:~$ make
gcc -I . -c module.c
gcc main.o module.o -o target_bin
```

Since `module.c` in the dependency tree has changed (it now has a later timestamp than its target, `module.o`), make runs the action for the `module.o` target, which recompiles the changed source file. It then runs the action for the all target.

We can explicitly invoke the `clean` target to clean up all the generated `.o` files and `target_bin`:

```
$ make clean
rm -rf *.o
rm target_bin
```

More bytes on Makefiles

Make provides many interesting features that we can use in Makefiles. Let's look at the most essential ones.

Dealing with assignments

There are different ways of assigning variables in a Makefile. They are (type of assignment, followed by the operator in parentheses):

Simple assignment (:=)

We can assign values (RHS) to variables (LHS) with this operator, for example: `CC := gcc`. With simple assignment (`:=`), the value is expanded and stored to all occurrences in the Makefile when its first definition is found.

For example, when a `CC := ${GCC} ${FLAGS}` simple definition is first encountered, `CC` is set to `gcc -W` and wherever `${CC}` occurs in actions, it is replaced with `gcc -W`.

Recursive assignment (=)

Recursive assignment (the operator used is `=`) involves variables and values that are not evaluated immediately on encountering their definition, but are re-evaluated every time they are encountered in an action that is being executed. As an example, say we have:

```
GCC = gcc
FLAGS = -W
```

With the above lines, `CC = ${GCC} {FLAGS}` will be converted to `gcc -W` only when an action like `${CC} file.c` is executed somewhere in the Makefile. With recursive assignation, if the GCC variable is changed later (for example, `GCC = c++`), then when it is

next encountered in an action line that is being updated, it will be re-evaluated, and the new value will be used; `${CC}` will now expand to `c++ -W`.

We will also have an interesting and useful application further in the article, where this feature is used to deal with varying cases of filename extensions of image files.

Conditional assignment (?=)

Conditional assignment statements assign the given value to the variable only if the variable does not yet have a value.

Appending (+=)

The appending operation appends texts to an existing variable. For example:

```
CC = gcc
CC += -W
```

`CC` now holds the value `gcc -W`.

Though variable assignments can occur in any part of the Makefile, on a new line, most variable declarations are found at the beginning of the Makefile.

Using patterns and special variables

The `%` character can be used for wildcard pattern-matching, to provide generic targets. For example:

```
%.o: %.c
[TAB] actions
```

When `%` appears in the dependency list, it is replaced with the same string that was used to perform substitution in the target.

Inside actions, we can use special variables for matching filenames. Some of them are:

- `$@` (full target name of the current target)
- `$?` (returns the dependencies that are newer than the current target)
- `$*` (returns the text that corresponds to `%` in the target)
- `$<` (name of the first dependency)
- `$^` (name of all the dependencies with space as the delimiter)

Instead of writing each of the file names in the actions and the target, we can use shorthand notations based on the above, to write more generic Makefiles.

Action modifiers

We can change the behaviour of the actions we use by prefixing certain action modifiers to the actions. Two important action modifiers are:

- `-` (minus) — Prefixing this to any action causes any error that occurs while executing the action to be ignored. By default, execution of a Makefile stops when any command returns a non-zero (error) value. If an error occurs, a message is printed, with the status code of the command, and noting that the error has been ignored. Looking at the Makefile from our sample project: in the clean target, the `rm target_bin` command will produce an error if that file does not exist (this could happen if the project had never been compiled, or if `make clean` is run twice consecutively). To handle this, we can prefix the `rm` command with a minus, to ignore errors: `-rm target_bin` .
- `@` (at) suppresses the standard print-action-to-standard-output behaviour of make, for the action/command that is prefixed with `@` . For example, to echo a custom message to standard output, we want only the output of the echo command, and don't want to print the echo command line itself. `@echo Message` will print "Message" without the echo command line being printed.

Use PHONY to avoid file-target name conflicts

Remember the `all` and `clean` special targets in our Makefile? What happens when the project directory has files with the names `all` or `clean` ? The conflicts will cause errors. Use the `.PHONY` directive to specify which targets are not to be treated as files — for example: `.PHONY: all clean` .

Simulating make without actual execution

At times, maybe when developing the Makefile, we may want to trace the make execution (and view the logged messages) without actually running the actions, which is time consuming. Simply use `make -n` to do a "dry run".

Using the shell command output in a variable

Sometimes we need to use the output from one command/action in other places in the Makefile — for example, checking versions/locations of installed libraries, or other files required for compilation. We can obtain the shell output using the shell command. For example, to return a list of files in the current directory into a variable, we would run: `LS_OUT = $(shell ls)` .

Nested Makefiles

Nested Makefiles (which are Makefiles in one or more subdirectories that are also executed by running the make command in the parent directory) can be useful for building smaller projects as part of a larger project. To do this, we set up a target whose action changes directory to the subdirectory, and invokes make again:

```
subtargets:
    cd subdirectory && $(MAKE)
```

Instead of running the make command, we used `$(MAKE)` , an environment variable, to provide flexibility to include arguments. For example, if you were doing a "dry run" invocation: if we used the make command directly for the subdirectory, the simulation

option (`-n`) would not be passed, and the commands in the subdirectory's Makefile would actually be executed. To enable use of the `-n` argument, use the `$(MAKE)` variable.

Now let's improve our original Makefile using these advanced features:

```
CC = gcc # Compiler to use
OPTIONS = -O2 -g -Wall # -g for debug, -O2 for optimise and -Wall additional messages
INCLUDES = -I . # Directory for header file
OBJS = main.o module.o # List of objects to be build
.PHONY: all clean # To declare all, clean are not files

all: ${OBJS}
    @echo "Building.." # To print "Building.." message
    ${CC} ${OPTIONS} ${INCLUDES} ${OBJS} -o target_bin

%.o: %.c  # % pattern wildcard matching
    ${CC} ${OPTIONS} -c $*.c ${INCLUDES}
list:
    @echo $(shell ls) # To print output of command 'ls'

clean:
    @echo "Cleaning up.."
    -rm -rf *.o # - prefix for ignoring errors and continue execution
    -rm target_bin
```

Run make on the modified Makefile and test it; also run make with the new list target. Observe the output.

Make in non-compilation contexts

I hope you're now well informed about using make in a programming context. However, it's also useful in non-programming contexts, due to the basic behaviour of checking the modification timestamps of target files and dependencies, and running the specified actions when required. For example, let's write a Makefile that will manage an image store for us, doing thumbnailing when required. Our scenario is as follows:

- We have a directory with two subdirectories, `images` and `thumb` .
- The `images` subdirectory contains many large image files; `thumb` contains thumbnails of the images, as `.jpg` files, 100x100px in image size.
- When a new image is added to the images directory, creation of its thumbnail in the `thumb` directory should be automated. If an image is modified, its thumbnail should be updated.
- The thumbnailing process should only be done for new or updated images, and not images that have up-to-date thumbnails.

This problem can be solved easily by creating a Makefile in the top-level directory, as follows:

```
FILES = $(shell  find images -type f -iname "*.jpg" | sed 's/images/thumb/g')
CONVERT_CMD = convert -resize "100x100" $< $@
MSG = @echo "\nUpdating thumbnail" $@

all: ${FILES}
thumb/%.jpg: images/%.jpg
    $(MSG)
    $(CONVERT_CMD)
thumb/%.JPG: images/%.JPG
    $(MSG)
    $(CONVERT_CMD)
clean:
    @echo Cleaning up files..
    rm -rf thumb/*.jpg thumb/*.JPG
```

In the above Makefile, `FILES = $(shell  find images -type f -iname "*.jpg" | sed 's/images/thumb/g')` is used to generate a list of dependency filenames. JPEG files could have the extension `.jpg` or `.JPG` (that is, differing in case). The `-iname` parameter to find (`find images -type f -iname "*.jpg"`) will do a case-insensitive search on the names of files, and will return files with both lower-case and upper-case extensions — for example, `images/1.jpg` , `images/2.jpg` , `images/3.JPG` and so on. The `sed` command replaces the text "images" with "thumb", to get the dependency file path.

When make is invoked, the all target is executed first. Since FILES contains a list of thumbnail files for which to check the timestamp (or if they exist), make jumps down to the `thumb/%.jpg` wildcard target for each thumbnail image file name. (If the extension is upper-case, that is, `thumb/3.JPG` , then make will look for, and find, the second wildcard target, `thumb/%.JPG` .)

For each thumbnail file in the `thumb` directory, its dependency is the image file in the `images` directory. Hence, if any file (that's expected to be) in the `thumb` directory does not exist, or its timestamp is older than the dependency file in the `images` directory, the action (calling `$(CONVERT_CMD`) to create a thumbnail) is run.

Using the features we described earlier, `CONVERT_CMD` is defined before targets are specified, but it uses recursive assignment. Hence, the input and target filenames passed to the convert command are substituted from the first dependency (`$<`) and the target (`$@`) every time the action is invoked, and thus will work no matter from which action target (`thumb/%.JPG` or `thumb/%.jpg`) the action is invoked.

Naturally, the "Updating thumbnail" message is also defined using recursive assignment for the same reasons, ensuring that `$(MSG)` is re-evaluated every time the actions are executed, and thereby able to cope with variations in the case of the filename extension.

```
slynux@freedom:~$ make
Updating thumbnail 1.jpg
convert -resize "100x100" images/1.jpg thumb/1.jpg
   Updating thumbnail 4.jpg
convert -resize "100x100" images/4.jpg thumb/4.jpg
```

If I edit `4.jpg` in images and rerun make, since only `4.jpg` 's timestamp has changed, a thumbnail is generated for that image:

```
slynux@freedom:~$ make
Updating thumbnail 4.jpg
```

```
convert -resize "100x100" images/4.jpg thumb/4.jpg
```

Writing a script (shell script or Python, etc) to maintain image thumbnails by monitoring timestamps would have taken many lines of code. With make, we can do this in just 8 lines of Makefile. Isn't make awesome?

That's all about the basics of using the make utility. Happy hacking till we meet again!

This article was originally published in September 2010 issue of the print magazine.

Sarath Lakshman

A guy bitten by Linux bug! Free and Open Source enthusiast, author, contributor.