Makefile

From OSDev Wiki

A Makefile is a file which controls the 'make' command. Make is available on virtually every platform, and is used to control the build process of a project. Once a Makefile has been written for a project, make can easily and efficiently build your project and create the required output file(s).

Difficulty level

Beginner

Make reads dependency information from your Makefile, figures out which output files need (re-)building (because they are either missing or older than their corresponding input files), executes the necessary commands to actually do the (re-)building. This compares favourably to "build batchfiles" that always rebuild the whole project.

In doing this, make is not limited to any specific set of languages. Any tool that takes some input file and generates an output file can be controlled via make.

A Makefile can offer several different "targets", which can be invoked by 'make <target>'. A frequently-used target is 'make clean', which removes any temporary files, or 'make install', which installs the project in its target location - but you can also give a specific output file as a target, and have make process only that file. Invoking make without a target parameter builds the default target (which is usually to build the whole project).

But 'make' requires a well-written Makefile to do these things efficiently and reliably.

Contents

Makefile tutorial

The 'make' manual will tell you about the hundreds of things you can do with a Makefile, but it doesn't give you an example for a good Makefile. The following examples are mostly shaped after the real-life PDCLib (http://pdclib.e43.eu) Makefile I used at the time, and shows some of the "tricks" used therein that may not be that obvious to the make beginner.

The Makefile creates only one project-wide linker library, but it should be easy to expand it for multiple binaries/libraries.

(Note: There are several different flavours of make around, and POSIX defines a common denominator for them. This tutorial specifically targets GNU make. See the discussion page for further info.)

Basics

It is best practice to name your Makefile `Makefile` (without an extension), because when make is executed without any parameters, it will by default look for that file in the current directory. It also makes the file show up prominently on top of directory listings, at least on Unix machines.

Generally speaking, a Makefile consists of definitions and rules.

A definition declares a variable, and assigns a value to it. Its general syntax is `VARIABLE := Value`.

Note: Frequently, you will see definitions using "=" instead of ":=". Such a definition will result in the assignment being evaluated every time the variable is used, while ":=" evaluates only once at the startup of make, which is usually what you want. Don't go changing other people's Makefiles, though - ":=" is a GNU extension to the make syntax.

A rule defines a target, 0..n dependencies, and 0..n commands. The general idea is that make checks if the target (file) is there; if it isn't, or any of the dependencies is newer than the target, the commands are executed. The general syntax is:

```
target: dependency
        command
```

Both dependency and command are optional. There might be more than one command line, in which case they are executed in sequence.

Note that commands must be tab-indented. If your editor environment is set to replace tabs with spaces, you have to undo that setting while editing a Makefile.

What makes Makefiles so hard to read, for the beginner, is that it is not procedural in syntax (i.e., executed top-down), but functional: 'make' reads the whole Makefile, building a dependency tree, and then resolves the dependencies by hopping from rule to rule as necessary until the target you gave it on the command line has successfully been resolved.

But let's not go into internal details of 'make'. This is a tutorial, not a man page, so it will show you how a real-life Makefile could be built, and the ideas behind each line.

Automated Testing

As the Makefile presented in this tutorial takes care of automated testing in a special way, this approach will be explained up front, so the related parts of the tutorial will make sense to you.

As stated above, this tutorial is mainly derived from work done on the PDCLib project - which builds a single linker library. In that project, there is strictly one library function per source file. In each such source file, there is a test driver attached for conditional compilation, like this:

```
#ifdef TEST
int main()
{
    /* Test function code here */
    return NUMBER_OF_TEST_ERRORS;
}
#endif
```

Thus, when that source is compiled with `gcc -c`, it results in an object file with the library function code; when compiled with `gcc -DTEST`, it gives a test driver executable for that function. Returning the number of errors allows to do a grand total of errors encountered (see below).

File Lists

First, various "file lists" are assembled which are needed later in the Makefile.

Non-Source Files

```
# This is a list of all non-source files that are part of the di
AUXFILES := Makefile Readme.txt
```

Further down we will have a target "dist", which packages all required files into a tarball for distribution. Lists of the sources and headers are created anyway. But there are commonly some auxiliary files, which are not referenced anywhere else in the Makefile but should still end up in the tarball. These are listed here.

Project Directories

```
PROJDIRS := functions includes internals
```

Those are subdirectories holding the actual sources. (Or rather, searched for source files automatically, see below.) These could be subprojects, or whatever. We could simply search for source files starting with the current working directory, but if you like to have temporary subdirectories in your project (for testing, keeping reference sources etc.), that wouldn't work.

Note that this is not a recursive approach; there is no Makefile in those subdirectories. Dependencies are hard enough to get right if you use one Makefile. At the bottom of this article is a link to a very good paper on the subject titled "recursive make considered harmful"; not only is a single Makefile easier to maintain (once you learned a couple of tricks), it can also be more efficient!

Sources and Headers

```
SRCFILES := $(shell find $(PROJDIRS) -type f -name "\*.c")
HDRFILES := $(shell find $(PROJDIRS) -type f -name "\*.h")
```

It should be obvious to see what these two lines do. We now have a list of all source and header files in our project directories.

Object Files and Test Driver Executables

```
OBJFILES := $(patsubst %.c,%.o,$(SRCFILES))
TSTFILES := $(patsubst %.c,%_t,$(SRCFILES))
```

OBJFILES should be clear - a list of the source files, with *.c exchanged for *.o. TSTFILES does the same for the filename suffix *_t, which we will use for our test driver executables.

> Note: This tutorial initially used *.t instead of *_t here; this, however, kept us from handling dependencies for test driver executables seperately from those for library object files, which was necessary. See next section.

Dependencies, pt. 1

Many people edit their Makefile every time they add/change an #include somewhere in their code. (Or forget to do so, resulting in some scratching of heads.)

But most compilers - including GCC - can do this automatically for you! Although the approach looks a little backward. For each source file, GCC will create a dependency file (which is canonically made to end in *.d), which contains a Makefile dependency rule listing that source file's includes. (And more, but see below.)

We need two seperate sets of dependency files; one for the library objects, and one for the test driver executables (which usually have additional includes, and thus dependencies, not needed for the OBJFILES).

```
DEPFILES    := $(patsubst %.c,%.d,$(SRCFILES))
TSTDEPFILES := $(patsubst %,%.d,$(TSTFILES))
```

Distribution Files

The last list is the one with all sources, headers, and auxiliary files that should end up in a distribution tarball.

```
ALLFILES := $(SRCFILES) $(HDRFILES) $(AUXFILES)
```

.PHONY

The next one can take you by surprise. When you write a rule for make clean, and there happens to be a file named clean in your working directory, you might be surprised to find that make does nothing, because the "target" of the rule clean already exists. To avoid this, declare such rules as phony, i.e. disable the checking for a file of that name. These will be executed every time:

```
.PHONY: all clean dist check testdrivers todolist
```

CFLAGS

If you thought -Wall does tell you everything, you'll be in for a rude surprise now. If you don't even use -Wall, shame on you. ;)

```
WARNINGS := -Wall -Wextra -pedantic -Wshadow -Wpointer-arith -Wc
            -Wwrite-strings -Wmissing-prototypes -Wmissing-decla
            -Wredundant-decls -Wnested-externs -Winline -Wno-lon
            -Wuninitialized -Wconversion -Wstrict-prototypes
CFLAGS := -g -std=c99 $(WARNINGS)
```

It is suggested to add new warning options to your project one at a time instead of all at once, to avoid getting swamped in warnings. ;) These flags are merely recommendations for C work. If you use C++, you need different ones. Check out the GCC manual; each major compiler update changes the list of available warning options.

Rules

Now come the rules, in their typical backward manner (top-level rule first). The topmost rule is the default one (if 'make' is called without an explicit target). It is common practice to have the first rule called "all".

Top-Level Targets

```
all: pdclib.a

pdclib.a: $(OBJFILES)
        @ar r pdclib.a $?
```

```
clean:
        -@$(RM) $(wildcard $(OBJFILES) $(DEPFILES) $(TSTFILES) p

dist:
        @tar czf pdclib.tgz $(ALLFILES)
```

The @ at the beginning of a line tells make to be quiet, i.e. not to echo the executed commands on the console prior to executing them. The Unix credo is "no news is good news". You don't get a list of processed files with cp or tar, either, so it's completely beyond me why developers chose to have their Makefiles churn out endless lists of useless garbage. One very practical advantage of shutting up make is that you actually get to see those compiler warnings, instead of having them drowned out by noise.

The - at the beginning of a line tells make to continue even in case an error is encountered (default behaviour is to terminate the whole build).

The $(RM) in the clean rule is the platform-independent command to remove a file.

The $? in the pdclib.a rule is an internal variable, which make expands to list all dependencies to the rule which are newer than the target.

Automated Testing, pt. 2

```
check: testdrivers
        -@rc=0; count=0; \
        for file in $(TSTFILES); do \
            echo " TST     $$file"; ./$$file; \
            rc=`expr $$rc + $$?`; count=`expr $$count + 1`; \
        done; \
        echo; echo "Tests executed: $$count  Tests failed: $$rc"

testdrivers: $(TSTFILES)
```

Call it crude, but it works beautifully for me. The leading - means that 'make' will not abort when encountering an error, but continue with the loop.

The echo " TST $$file" in there is useful in case you get a SEGFAULT or something like that from one of your test drivers. (Without echoing the drivers as they are executed, you would be at a loss as to which one crashed.)

Dependencies, pt. 2

```
-include $(DEPFILES) $(TSTDEPFILES)
```

Further below, you will see how dependency files are generated. Here, we include all of them, i.e. make the dependencies listed in them part of our Makefile. Never mind that they might not even exist when we run our Makefile the first time - the leading "-" again suppresses the errors.

Extracting TODO Statements

```
todolist:
	-@for file in $(ALLFILES:Makefile=); do fgrep -H -e TODO
```

Taking all files in your project, with exception of the Makefile itself, this will grep all those TODO and FIXME comments from your files, and display them in the terminal. It is nice to be remembered of what is still missing before you do a release. To add another keyword, just insert another `-e keyword`.

> **Note:** $(ALLFILES:Makefile=) is a list of everything in $(ALLFILES), except for the string "Makefile" which is replaced with nothing (i.e., removed from the list). This avoids a self-referring match, where the string "TODO" in the grep command would find itself. (Of course, it also avoids finding any *real* TODO's in the Makefile, but there's always a downside. ;-))

Dependencies, pt. 3

Now comes the dependency magic I talked about earlier. Note that this needs GCC 3.3 or newer.

```
%.o: %.c Makefile
	@$(CC) $(CFLAGS) -MMD -MP -c $< -o $@
```

Isn't it a beauty? ;-)

The -MMD flag generates the dependency file (%.d), which will hold (in Makefile syntax) rules making the generated file (%.o in this case) depend on the source file and any non-system headers it includes. That means the object file gets recreated automatically whenever relevant sources are touched. If you want to also depend on system headers (i.e., checking them for updates on each compile), use -MD instead. The -MP option adds empty dummy rules, which avoid errors should header files be removed from the filesystem.

Compiling the object file actually looks like a side effect. ;-)

Note that the dependency list of the rule includes the Makefile itself. If you changed e.g. the CFLAGS in the Makefile, you want them to be applied, don't you? Using the $< macro ("first dependency") in the command makes sure we do not attempt to compile the Makefile as C source.

Of course we also need a rule for generating the test driver executables (and their dependency files):

```
%_t: %.c Makefile pdclib.a
	@$(CC) $(CFLAGS) -MMD -MP -DTEST $< pdclib.a -o $@
```

Here you can see why test driver executables get a *_t suffix instead of a *.t extension: The -MMD option uses the basename (i.e., filename without extension) of the compiled file as basis for the dependency file. If we would compile the sources into abc.o and abc.t, the dependency files would both be named abc.d, overwriting each other.

Other compilers differ a bit in their support for this, so look up their specifics when using something else than GCC.

Backups

Backing up your files is a perfect candidate of a task that should be automated. However, this is generally achieved through revision control.

Below is an example of a backup target, which creates a dated 7-Zip archive of the directory where the Makefile resides.

```
THISDIR := $(shell basename `pwd`)
TODAY := $(shell date +%Y-%m-%d)
BACKUPDIR := projectBackups/$(TODAY)

backup: clean
        @tar cf - ../$(THISDIR) | 7za a -si ../$(BACKUPDIR)/$(TH
```

Summary

A well-written Makefile can make maintaining a code base much easier, as it can wrap complex command sequences into simple 'make' invocations. Especially for large projects, it also cuts back on compilation times when compared to a dumb "build.sh" script. And, once written, modifications to a well-written Makefile are seldom necessary.

Advanced Techniques

Some of the stuff we did above already was pretty advanced, and no mistake. But we needed those features for the basic yet convenient setup. Below you will find some even trickier stuff, which might not be for everybody but is immensely helpful if you need it.

Conditional Evaluation

Sometimes it is useful to react on the existence or content of certain environment variables. For example, you might have to rely on the path to a certain framework being passed in FRAMEWORK_PATH. Perhaps the error message given by the compiler in case the variable is not set is not helpful, or it takes long until 'make' gets to the point where it actually detects the error. You want to fail early, and with a meaningful error message.

Luckily, 'make' allows for conditional evaluation and manual error reporting, quite akin to the C preprocessor:

```
ifndef FRAMEWORK_PATH
    $(error FRAMEWORK_PATH is not set. Please set to the path wh
endif

ifneq ($(FRAMEWORK_PATH),/usr/lib/framework)
```

```
    $(warning FRAMEWORK_PATH is set to $(FRAMEWORK_PATH), not /u
endif
```

Multi-Target

Preliminary work has been done to write an improved Makefile, which allows generating multiple binaries with individual settings while still being easy to configure and use. While this is not yet in "tutorial" format, commented source can be found here.

See Also

Articles

- User:Solar/Makefile, A more complex example capable of building multiple executables and libraries from a single Makefile.

External Links

- JAWS (http://www.rootdirectory.de/wiki/JAWS) , a pre-configured CMake (http://www.cmake.org) setup which, while not geared toward OS development, is a definite step forward from "naked" Makefiles.
- Recursive Make Considered Harmful (http://aegis.sourceforge.net/auug97.pdf) by Peter Miller
  A paper detailing why the traditional approach of recursive make invocations harms performance and reliability.
- Implementing non-recursive make (http://www.xs4all.nl/~evbergen/nonrecursive-make.html) by Emile van Bergen
  Further input on the subject of non-recursive, low-maintenance Makefile creation.
- Manual for GNU make (http://www.gnu.org/software/make/manual/)
- Managing Projects with GNU Make (https://www.goodreads.com/book/show/583690.Managing_Projects_with_GNU_Make) , the O'Reilly book on GNU Make

---