Makefile Primer

How to use *make*
Course 1.12
Spring 1997
Based on 1.124 Makefile Primer

Introduction

make is a command generator which generates a sequence of commands for execution by the UNIX shell. These commands usually relate to the maintenance of a set of files in a software development project. We will use *make* to help us organize our C++ and C source code files during the compilation and linking process. In particular, *make* can be used to sort out the dependency relations among the various source files, object files and executables and to determine exactly how the object files and executables will be produced.

Invoking *make* from the command line

make may be invoked from the command line by typing

```
make -f makefilename program
```

Here, *program* is the name of the **target** i.e. the program to be made. *makefilename* is a **description file** which tells the *make* utility how to build the target *program* from its various **components**. Each of these components could be a target in itself. *make* would therefore have to build these targets, using information in the description file, before *program* can be made. *program* need not necessarily be the highest level target in the hierarchy, although in practice it often is.

It is not always necessary to specify the name of the description file when invoking *make*. For example,

```
make program
```

would cause *make* to look in the current directory for a default description file named *makefile* or *Makefile*, in that order.

Furthermore, it is not even necessary to specify the name of the final target. Simply typing

```
make
```

will build the first target found in the default description file, together with all of its components. On the other hand, it is also possible to specify multiple targets when invoking *make*.

There are a few other command line options which we will not discuss here. To find out more about these options, type *man make* at your *athena%* prompt.

make description files (makefiles)

Here is an example of a simple makefile:

```
program: main.o iodat.o
	cc -o program main.o iodat.o
main.o: main.c
	cc -c main.c
iodat.o: iodat.c
	cc -c iodat.c
```

Each entry consists of a **dependency line** containing a colon, and one or more **command lines** each starting with a tab. Dependency lines have one or more **targets** to the left of the colon. To the right of the colon are the **component files** on which the target(s) depend.

A command line will be executed if any target listed on the dependency line does not exist, or if any of the component files are more recent than a target.

Here are some points to remember:

- Comments start with a pound sign (#).
- Continuation of a line is denoted by a backslash (\).
- Lines containing equals signs (=) are macro definitions (see next section).
- Each command line is typically executed in a separate Bourne shell i.e. *sh*[1].

To execute more than one command line in the same shell, type them on the same line, separated by semicolons. Use a \ to continue the line if necessary. For example,

```
program: main.o iodat.o
	cd newdir; \
	cc -o program main.o iodat.o
```

would change to the directory *newdir* before invoking *cc*. (Note that executing the two commands in separate shells would not produce the required effect, since the *cd* command is only effective within the shell from which it was invoked.)

The Bourne shell's pattern matching characters maybe used in command lines, as well as to the right of the colon in dependency lines e.g.

```
program: *.c
	cc -o program *.c
```

[1]Type *man sh* to find out more about the Bourne shell.

Macros

Macro definitions in the description file

Macro definitions are of the form

name = string

Subsequent references to $(*name*) or ${*name*} are then interpreted as *string*. Macros are typically grouped together at the beginning of the description file. Macros which have no string to the right of the equals sign are assigned the null string. Macros may be included within macro definitions, regardless of the order in which they are defined.

Here is an example of a macro:

```
CC = /mit/cygnus/decmipsbin/g++
program: program.c
        ${CC} -o program program.c
```

Shell environment variables

Shell variables that were part of the environment before *make* was invoked are available as macros within *make*. Within a *make* description file, however, shell environment variables must be surrounded by parentheses or braces, unless they consist of a single character. For example, ${PWD} may be used in a description file to refer to the current working directory.

Command line macro definitions

Macros can be defined when invoking *make* e.g.

make program CC=/mit/cygnus/decmipsbin/g++

Internal macros

make has a few predefined macros:

1. $? evaluates to the list of components that are *younger* than the current target. Can only be used in description file command lines.

2. $@ evaluates to the current target name. Can only be used in description file command lines.

3. $$@ also evaluates to the current target name. However, it can only be used on dependency lines.

   Example:

```
PROGS = prog1 prog2 prog3
${PROGS}: $$@.c
        cc -o $@ $?
```

This will compile the three files *prog1.c*, *prog2.c* and *prog3.c*, unless any of them have already been compiled. During the compilation process, each of the programs becomes the current target in turn. In this particular example, the same effect would be obtained if we replaced the $? by $@.c.

Order of priority of macro assignments

The following is the order of priority of macro assignments, from least to greatest:

1. Internal (default) macro definitions.
2. Shell environment variables.
3. Description file macro definitions.
4. Command line macro definitions.

Items 2. and 3. can be interchanged by specifying the *-e* option to *make*.

Macro string substitution

String substitutions may be performed on all macros used in description file shell commands. However, substitutions occur only at the end of the macro, or immediately before white space. The following example illustrates this:

```
LETTERS = abcxyz xyzabc xyz
print:
        echo $(LETTERS:xyz=def)
```

This description file will produce the output

```
abcdef xyzabc def
```

Suffix rules

The existence of naming and compiling conventions makes it possible to considerably simplify description files. For example, the C compiler requires that C source files always have a *.c* suffix. Such naming conventions enable *make* to perform many tasks based on **suffix rules**. *make* provides a set of default suffix rules. In addition, new suffix rules can be defined by the user.

For example, the description file on page 2 can be simplified to

```
program: main.o iodat.o
        cc -o program main.o iodat.o
```

make will use the following default macros and suffix rules to determine how to build the components *main.o* and *iodat.o*.

```
CC = cc
CFLAGS = -O
.SUFFIXES: .o .c
.c.o:
        ${CC} ${CFLAGS} $<
```

The entries on the *.SUFFIXES* line represent the suffixes which *make* will consider **significant**. Thus, in building *iodat.o* from the above description file, *make* looks for a user-specified dependency line containing *iodat.o* as a target. Finding no such dependency, *make* notes that the *.o* suffix is significant and therefore it looks for another file in the current directory which can be used to make *iodat.o*. Such a file must

- have the same name (apart from the suffix) as *iodat.o*.
- have a significant suffix.
- be able to be used to make *iodat.o* according to an existing suffix rule.

make then applies the above suffix rule which specifies how to build a *.o* file from a *.c* file. The $< macro evaluates to the component that triggered the suffix rule i.e. *iodat.c*.

After the components *main.o* and *iodat.o* have been updated in this way (if necessary), the target *program* will be built according to the directions in the description file.

Internal macros in suffix rules

The following internal macros can only be used in suffix rules.

1. $< evaluates to the component that is being used to make the target.
2. $* evaluates to the filename part (without any suffix) of the component that is being used to make the target.

Note that the $? macro cannot occur in suffix rules. The $@ macro, however, can be used.

Null suffixes

Files with null suffixes (no suffix at all) can be made using a suffix rule which has only a single suffix e.g.

```
.c:
        ${CC} -o $@ $<
```

This suffix rule will be invoked to produce an executable called *program* from a source file *program.c*, if the description file contains a line of the form

```
program:
```

Note that in this particular situation it would be incorrect to specify that *program* depended on *program.c*, because *make* would then consider the command line to contain a null command and would therefore not invoke the suffix rule. This problem does not arise when building a *.o* file from a *.c* file using suffix rules. A *.o* file *can* be specified to depend on a *.c* file (and possibly some additional header files) because of the one-to-one relationship that exists betweeen *.o* and *.c* files.

Writing your own suffix rules

Suffix rules and the list of significant suffixes can be redefined. A line containing *.SUFFIXES* by itself will delete the current list of significant suffixes e.g.

```
.SUFFIXES:
.SUFFIXES: .o .c
.c.o:
        ${CC} -o $@ $<
```

References

[1] S. Talbot, ‘Managing Projects with Make’, O’Reilly & Associates, Inc.