Makefile.*

Andrey Lukyanenko, CSE, Aalto University

Spring, 2015

1. Command **make** is one which operates on `Makefiles` reading them and processing instructions. Calling **make** without parameters will try to execute instructions in a file `Makefile` which is searched locally. There is possibility to give a `Makefile` explicitly, e.g., **make -f my_makefile**.

2. Default `Makefile` structure consists of set preambule and a set of `target` rules.

```
target1 ... : prerequisites1 ...
    rule1 ...  # there is a <tab> before rule 1.
    rule2
    ...

target2... : prerequisites2 ... ; rule1  # semicolon before rule 1
    rule2...  # there is a <tab> before rule 2.
    rule3
    ...
```

3. The first target which does not start from '.' is a default target.

4. **make** reads file from the beginning finds a default target, looks for dependencies mentioned in `prerequisites` and if there are rules with target specified for these `prerequisites`, then it reads the file further for these targets and repeat the procedure. The targets which are not default or does not meet in tree of `prerequisites` from default are not processed. To process a specific target makefile can be called specifying it explicitly.

5. An example of Makefile:

```
main : main.o algorithm.o \
       protocol.o tree.o                # main is dependant on four object files
       cc -o main main.o algorithm.o \
       protocol.o tree.o

main.o : main.c header.h
       cc -c main.c
algorithm.o : algorithm.c header.h protocol.h
       cc -c algorithm.c
protocol.o : protocol.c header.h protocol.h
       cc -c protocol.c
tree.o : tree.c header.h
       cc -c tree.c
clean :
       rm edit main.o algorithm.o protocol.o tree.o
```

   make will check for default target `main`, by checking current directory for this program, it sees dependencies and verify those also. First check if file `main.o` is older than `main.c`, `header.c`. If older, then the rule for compilation of `main.c` is invoked. The same applies to `algorithm.o`, `protocol.o` and `tree.o`. In result if any of the object files is newer than `main` then the compilation (linking) rule of the first target is invoked. Rule `clean` would not be called, to call it the command `make` should be called as **make clean**.

6. **make** can use variables.

```
objects = main.o algorithm.o \
       protocol.o tree.o

objects += another.o


main : $(objects)                # main is dependant on four object files
       cc -o main $(objects)
```

*Detailed documentation: `http://www.gnu.org/s/make/manual/make.html`

7. Multiple targets:

```
algorithm.o protocol.o : algorithm.c protocol.c header.h protocol.h
       cc -c algorithm.c
       cc -c protocol.c
```

8. Wildcards usage:

```
clean:
        rm -f *.o

# wrong usage
objects = *.o

foo : $(objects)
        cc -o foo $(CFLAGS) $(objects)

# special wildcard command defines wildcards expliclitly
$(wildcard *.c)

# ... and is only option in functions
objects := $(patsubst %.c,%.o,$(wildcard *.c))
```

9. Static patterns:

```
targets ...: target-pattern: prereq-patterns ...
        rule1
        ...

# Example:
objects = foo.o bar.o

all: $(objects)

$(objects): %.o: %.c
        $(CC) -c $(CFLAGS) $< -o $@
```

10. As in shell variables (and a lot of functionality may be used).

```
# Example 1:
foo = $(bar)
bar = $(ugh)
ugh = Huh?

all:;echo $(foo)


# Example 2:
x := foo
y := $(x) bar
x := later

# Example 3:
whoami    := $(shell whoami)

# Example 4:
x = y
y = z
z = u
a := $($($(x)))

# Example 5:
LIST = one two three
all:
    for i in $(LIST); do \
        echo $$i; \
    done

# Example 6:
ifeq ($(CC),gcc)
  libs=$(libs_for_gcc)
else
  libs=$(normal_libs)
endif
```

11. Recursion

```
subsystem:
    cd subdir && $(MAKE)
```

12. `VPATH` and `vpath`

    `VPATH` is a path to search for prerequisite files, it can extend the current directory. E.g., `"VPATH = src:../headers"`.

    `vpath` is a path that more specifically defined based on a pattern. E.g., `"vpath %.h ../headers"`.

13. Automatic variables:

$@	The file name of the target of the rule. If the target is an archive member, then '$@' is the name of the archive file. In a pattern rule that has multiple targets.
$%	The target member name, when the target is an archive member. For example, if the target is foo.a(bar.o) then '$%' is bar.o and '$@' is foo.a. '$%' is empty when the target is not an archive member.
$<	The name of the first prerequisite. If the target got its recipe from an implicit rule, this will be the first prerequisite added by the implicit rule.
$?	The names of all the prerequisites that are newer than the target, with spaces between them. For prerequisites which are archive members, only the named member is used.
$^	The names of all the prerequisites, with spaces between them. For prerequisites which are archive members, only the named member is used. A target has only one prerequisite on each other file it depends on, no matter how many times each file is listed as a prerequisite. So if you list a prerequisite more than once for a target, the value of $^ contains just one copy of the name. This list does not contain any of the order-only prerequisites; for those see the '$|' variable, below.
$+	This is like '$^', but prerequisites listed more than once are duplicated in the order they were listed in the makefile. This is primarily useful for use in linking commands where it is meaningful to repeat library file names in a particular order.
$|	The names of all the order-only prerequisites, with spaces between them.
$*	The stem with which an implicit rule matches. If the target is dir/a.foo.b and the target pattern is a.%.b then the stem is dir/foo.

'$(@D)'	The directory part of the file name of the target, with the trailing slash removed. If the value of '$@' is dir/foo.o then '$(@D)' is dir. This value is . if '$@' does not contain a slash.
'$(@F)'	The file-within-directory part of the file name of the target. If the value of '$@' is dir/foo.o then '$(@F)' is foo.o. '$(@F)' is equivalent to '$(notdir $@)'.
'$(*D)'** **'$(*F)'	The directory part and the file-within-directory part of the stem; dir and foo in this example.
'$(%D)'** **'$(%F)'	The directory part and the file-within-directory part of the target archive member name. This makes sense only for archive member targets of the form archive(member) and is useful only when member may contain a directory name.
'$(<D)'** **'$(<F)'	The directory part and the file-within-directory part of the first prerequisite.
'$(^D)'** **'$(^F)'	Lists of the directory parts and the file-within-directory parts of all prerequisites.
'$(+D)'** **'$(+F)'	Lists of the directory parts and the file-within-directory parts of all prerequisites, including multiple instances of duplicated prerequisites.
'$(?D)'** **'$(?F)'	Lists of the directory parts and the file-within-directory parts of all prerequisites that are newer than the target.

14. Functions:

`$(subst from,to,text)`	Performs a textual replacement on the text text: each occurrence of from is replaced by to. The result is substituted for the function call.
`$(patsubst pattern,replacement,text)`	Finds whitespace-separated words in text that match pattern and replaces them with replacement.
`$(strip string)`	Removes leading and trailing whitespace from string and replaces each internal sequence of one or more whitespace characters with a single space.
`$(findstring find,in)`	Searches in for an occurrence of find. If it occurs, the value is find; otherwise, the value is empty.
`$(filter pattern...,text)`	Returns all whitespace-separated words in text that do match any of the pattern words, removing any words that do not match.
`$(filter-out pattern...,text)`	Returns all whitespace-separated words in text that do not match any of the pattern words, removing the words that do match one or more. This is the exact opposite of the filter function.
`$(sort list)`	Sorts the words of list in lexical order, removing duplicate words. The output is a list of words separated by single spaces.
`$(word n,text)`	Returns the nth word of text. The legitimate values of n start from 1. If n is bigger than the number of words in text, the value is empty.
`$(wordlist s,e,text)`	Returns the list of words in text starting with word s and ending with word e (inclusive). The legitimate values of s start from 1; e may start from 0. If s is bigger than the number of words in text, the value is empty. If e is bigger than the number of words in text, words up to the end of text are returned. If s is greater than e, nothing is returned.
`$(words text)`	Returns the number of words in text. Thus, the last word of text is `$(word $(words text),text)`.
`$(firstword names...)`	The argument names is regarded as a series of names, separated by whitespace. The value is the first name in the series. The rest of the names are ignored.
`$(lastword names...)`	The argument names is regarded as a series of names, separated by whitespace. The value is the last name in the series.

$(dir names...)	Extracts the directory-part of each file name in names. The directory-part of the file name is everything up through (and including) the last slash in it.
$(notdir names...)	Extracts all but the directory-part of each file name in names. If the file name contains no slash, it is left unchanged. Otherwise, everything through the last slash is removed from it.
$(suffix names...)	Extracts the suffix of each file name in names. If the file name contains a period, the suffix is everything starting with the last period. Otherwise, the suffix is the empty string.
$(basename names...)	Extracts all but the suffix of each file name in names. If the file name contains a period, the basename is everything starting up to (and not including) the last period.
$(addsuffix suffix,names...)	The argument names is regarded as a series of names, separated by whitespace; suffix is used as a unit. The value of suffix is appended to the end of each individual name and the resulting larger names are concatenated with single spaces between them.
$(addprefix prefix,names...)	The argument names is regarded as a series of names, separated by whitespace; prefix is used as a unit. The value of prefix is prepended to the front of each individual name and the resulting larger names are concatenated with single spaces between them.
$(join list1,list2)	Concatenates the two arguments word by word: the two first words (one from each argument) concatenated form the first word of the result, the two second words form the second word of the result, and so on. So the nth word of the result comes from the nth word of each argument. If one argument has more words that the other, the extra words are copied unchanged into the result.
$(wildcard pattern)	The argument pattern is a file name pattern, typically containing wildcard characters (as in shell file name patterns). The result of wildcard is a space-separated list of the names of existing files that match the pattern.
$(realpath names...)	For each file name in names return the canonical absolute name. A canonical name does not contain any . or .. components, nor any repeated path separators (/) or symlinks. In case of a failure the empty string is returned.
$(abspath names...)	For each file name in names return an absolute name that does not contain any . or .. components, nor any repeated path separators (/). Note that, in contrast to realpath function, abspath does not resolve symlinks and does not require the file names to refer to an existing file or directory. Use the wildcard function to test for existence.
$(if condition,then-part[,else-part])	The if function provides support for conditional expansion in a functional context.
$(or condition1[,condition2[,condition3...]])	The or function provides a "short-circuiting" OR operation. Each argument is expanded, in order. If an argument expands to a non-empty string the processing stops and the result of the expansion is that string.
$(and condition1[,condition2[,condition3...]])	The and function provides a "short-circuiting" AND operation. Each argument is expanded, in order.
$(foreach var,list,text)	The first two arguments, var and list, are expanded before anything else is done; note that the last argument, text, is not expanded at the same time. Then for each word of the expanded value of list, the variable named by the expanded value of var is set to that word, and text is expanded. Presumably text contains references to that variable, so its expansion will be different each time.

Other functions worth consider: **call**, **value**, **eval**, **origin**, **flavor**, **shell**.

15. An example of `Makefile` suitable for an automatic Latex paper compiling:

```
PREFIX=/usr/local/teTeX/bin/

LATEX=latex
BIBTEX=bibtex
PDFLATEX=pdflatex
DOC=mypaper
SRC=mypaper.tex part1.tex part2.tex part3.tex

FIGS_EPS=

FIGS_PDF=

FIGS_PNG=


%.png: %.fig
fig2dev -L png $< $@
```

```
%.eps: %.fig
fig2eps $< $@

%.pdf: %.plt
gnuplot $<

%.pdf: %.eps
epstopdf $<


all: $(FIGS_EPS) $(FIGS_PDF) $(DOC).pdf

$(DOC).dvi: $(DOC).tex $(SRC) Makefile $(DOC).bib
-$(LATEX) $(DOC)
-$(BIBTEX)  -min-crossrefs=100 $(DOC)
-$(LATEX) $(DOC)
-$(LATEX) $(DOC)

$(DOC).ps: $(DOC).dvi
dvips -ta4 $(DOC).dvi -o $(DOC).ps


$(DOC).pdf: $(DOC).dvi $(FIGS_PDF)
dvipdf $(DOC).dvi $(DOC).pdf


clean:
rm -f *.aux *.dvi *.idx *.ilg *.ind *.log *.toc $(DOC).bbl $(DOC).blg temp.ps $(DOC).ps *.pdf *~ *.png *.bak
```