DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

CONCURRENT AND REAL TIME PROGRAMMING
[INQ0091623] AA 2021-22

Lab X

GNU make

Gabriele Manduchi <gabriele.manduchi@unipd.it>

Andrea Rigoni Garola <andrea.rigonigarola@unipd.it>

GNU Make

https://www.gnu.org/software/make/manual/make.html#toc-Overview-of-make

The **make** utility automatically determines which pieces of a large program need to be recompiled, and issues commands to recompile them.

GNU make has been implemented by Richard Stallman, Roland McGrath and Paul D. Smith.

To prepare to use make, you must write a file called the Makefile (or makefile) that describes the relationships among files in your program and provides **commands for updating each file**.

Why update? because make tracks the time of last modifications and perform actions only when updates are needed:

For example In a program the e**xecutable file is updated from object files**, which are in turn made by **compiling source files**

Makefiles: RULE them all !

You need a file called a makefile to tell make what to do. Most often, the makefile tells make how to compile and link a program.

A simple makefile consists of **“rules”** with the following shape:

```
target … : prerequisites …
	recipe
	…
	…
```

Those spaces are TABS !!

you need to put a tab character at the beginning of every recipe line! This is an obscurity that catches the unwary. If you prefer to prefix your recipes with a character other than tab, you can set the .RECIPEPREFIX

Example

```
objects = main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o

edit : $(objects)
        cc -o edit $(objects)
main.o : main.c defs.h
        cc -c main.c
kbd.o : kbd.c defs.h command.h
        cc -c kbd.c
command.o : command.c defs.h command.h
        cc -c command.c
display.o : display.c defs.h buffer.h
        cc -c display.c
insert.o : insert.c defs.h buffer.h
        cc -c insert.c
search.o : search.c defs.h buffer.h
        cc -c search.c
files.o : files.c defs.h buffer.h command.h
        cc -c files.c
utils.o : utils.c defs.h
        cc -c utils.c
clean :
        rm edit $(objects)
```

Variables

Makefile can define variables: Variables Make **Makefiles Simpler**

EXAMPLE:

```
objects = main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o

program : $(objects)
        cc -o program $(objects)

main.o : main.c defs.h
        cc -c main.c

kbd.o : kbd.c defs.h command.h
        cc -c kbd.c

command.o : command.c defs.h command.h
        cc -c command.c
```

To substitute a variable’s value, write a dollar sign followed by the name of the variable in parentheses or braces: either ‘**$(foo)**’ or ‘**${foo}**’ is a valid reference to the variable foo. This special significance of ‘$’ is why you must write ‘$$’ to have the effect of a single dollar sign in a file name or recipe.

The Two Flavors of Variables: 1 recursively expanded

Variable can be defined with the A = B construct.

when using this variable is recursively expanded, for example:

```
foo = $(bar)
bar = $(ugh)
ugh = Huh?

all:;echo $(foo)
```

will echo ‘Huh?’: ‘$(foo)’ expands to ‘$(bar)’ which expands to ‘$(ugh)’ which finally expands to ‘Huh?’.

ADVANTAGE: You can define variables wherever you like within Makefile

DISADVANTAGE: You cannot append something on the end of a variable

X
```
CFLAGS = $(CFLAGS) -O
```

NOTE:
In GNU Makefile there is no difference between parenthesis $() and curly bracket ${} syntax.

The Two Flavors of Variables: 2 simply expanded

Simply expanded variables are defined by lines using ‘:=’ or ‘::=’

The value of a simply expanded variable is scanned once and for all, expanding any references to other variables and functions, when the variable is defined.

```
x := foo
y := $(x) bar
x := later
```

is equivalent to

```
y := foo bar
x := later
```

When a simply expanded variable is referenced, its value is substituted verbatim.

Function Calls

In Makefiles there is the possibility to define functions:

A function call resembles a variable reference. It can appear anywhere a variable reference can appear, and it is expanded using the same rules as variable references.

A function call looks like this:

```
$(function arguments)
```

Here are some functions that operate on strings:

- `$(subst from,to,text)`
  Performs a textual replacement on the text text
- `$(patsubst pattern,replacement,text)`
  Finds whitespace-separated words in text that match pattern and replaces them with replacement.
- `$(var:pattern=replacement)`
  Equivalent of patsubst on variables
- `$(strip string)`
  Removes leading and trailing whitespace from string
- `$(findstring find,in)`
  Searches in for an occurrence of find. If it occurs, the value is find;
- **ETC …   see: https://www.gnu.org/software/make/manual/make.html#Using-Variables**

Appending More Text to Variables

Often it is useful to add more text to the value of a variable already defined. You do this with a line containing ‘+=’, like this:

```
objects += another.o
```

The appending construct allows to append something to a recursively expanded var:

```
variable   = value
variable += more
```

Or:

```
variable := value
variable := $(variable) more
```

Automatic Variables

Suppose you are writing a pattern rule to compile a '.c' file into a '.o' file: how do you write the 'cc' command so that it operates on the right source file name? You cannot write the name in the recipe, because the name is different each time the implicit rule is applied.

$@	The file name of the target of the rule.
$%	The target member name, when the target is an archive member.
$<	The name of the first prerequisite.
$?	The names of all the prerequisites that are newer than the target, with spaces between them.
$^	The names of all the prerequisites, with spaces between them.

See more at: https://www.gnu.org/software/make/manual/make.html#Automatic-Variables

Implicit Rules

If you write a rule with no recipe, or don't write a rule at all, then make will figure out which implicit rule to use based on which kind of source file exists or can be made.

```
foo : foo.o bar.o
        cc -o foo foo.o bar.o $(CFLAGS) $(LDFLAGS)
```

Because you mention foo.o but do not give a rule for it, make will automatically look for an implicit rule that tells how to update it (this happens whether or not the file foo.o currently exists).

Catalogue-of-Rules

Compiling C programs

n.o is made automatically from **n.c** with a recipe of the form '**$(CC) $(CPPFLAGS) $(CFLAGS) -c**'.

Compiling C++ programs

n.o is made automatically from n.cc, n.cpp, or n.C with a recipe of the form '$(CXX) $(CPPFLAGS) $(CXXFLAGS) -c'. We encourage you to use the suffix '.cc' for C++ source files instead of '.C'.

Example: lab5

```
all: threads process

threads_example: threads_example.o
	$(LINK.C) $^ -o $@ -lpthread

process_example: process_example.o
	$(LINK.C) $^ -o $@

.SUFFIXES: .pix .png .csv
.pix.png:
	python from_pixmap.py $< $@
.csv.png:
	python benchmark.py $<

threads_benchmark: threads_benchmark.o
	$(LINK.C) $^ -o $@ -lpthread

MAX_THREADS = 20
threads_benchmark.csv:
	./threads_benchmark $(MAX_THREADS)

threads: threads_example threads_benchmark \
        threads_benchmark.png
```

```
CLEANFILES = threads_example \
            threads_benchmark \
            threads_benchmark.csv \
            threads_benchmark.png


process: process_example

clean:
	rm -rf *.o $(CLEANFILES)
```

Example: lab5

CODING EXAMPLE CODING EXAMPLE

```
andrea@HP:~/devel/unipd/crtp/src/lab5$make threads

cc threads_example.c -c
g++     threads_example.o -o threads_example -lpthread
cc threads_benchmark.c -c
g++     threads_benchmark.o -o threads_benchmark -lpthread
./threads_benchmark 20
Elapsed time(us): 1, 257424.649000, 290857.240000
Elapsed time(us): 2, 146827.569000, 267533.490000
Elapsed time(us): 3, 91046.573000, 252662.549000
Elapsed time(us): 4, 69873.186000, 285484.374000
Elapsed time(us): 5, 94856.254000, 376924.535000
Elapsed time(us): 6, 98053.288000, 603124.461000
Elapsed time(us): 7, 75306.083000, 417638.859000
Elapsed time(us): 8, 67736.100000, 449243.101000
Elapsed time(us): 9, 71132.930000, 448585.657000
Elapsed time(us): 10, 69848.667000, 423070.649000
Elapsed time(us): 11, 63406.762000, 424466.072000
Elapsed time(us): 12, 68060.672000, 431175.330000
Elapsed time(us): 13, 66806.866000, 440090.086000
Elapsed time(us): 14, 64653.446000, 452766.328000
Elapsed time(us): 15, 70393.598000, 440526.065000
Elapsed time(us): 16, 60709.902000, 454479.518000
Elapsed time(us): 17, 60321.442000, 448701.902000
Elapsed time(us): 18, 63353.505000, 456570.013000
Elapsed time(us): 19, 64392.365000, 454465.989000
Elapsed time(us): 20, 61376.107000, 458292.761000
python benchmark.py threads_benchmark.csv
```