Intro to Make

CS255 – Systems Programming Lab

John Malliotakis – jmal@csd.uoc.gr

Department of Computer Science, University of Crete, Heraklion, Greece

UNIVERSITY
OF CRETE

What is Make ?

- Free and open-source tool
- Allows build (and task) automation
- Only requires a formatted input text file

According to GNU

> *GNU Make is a tool which controls the generation of executables and other non-source files of a program from the program's source files.*

Ok, but what does all this mean?

Background

What is its purpose?

”Why should I use it? I can just use gcc”

- Automate the process, write *make* and be done with it
- Avoid typing out huge compilation commands
- You do not have to know the entire build process

More benefits

- Can be used for *any* programming language
- Or any *task*
- You just specify the commands to be run
- Can evaluate necessary steps to take and their order, e.g.:
  - Program B needs program A, compile A before compiling B
  - Program C needs A and B, A is changed but B is not, only recompile A and C

How does it work?

Make uses an input text file with a specific syntax

1. Usually called *makefile* or *Makefile*
2. Makefile contains *variables, targets, and rules* (more on those later)
3. Commands in Makefile executed one by one, user notified which command is being executed
4. Make stops when all commands executed, or on error

Makefiles

Makefile structure

A Makefile is a collection of rules

- A rule describes the steps needed to build a target from a set of dependencies
- The steps are just shell commands
- Dependencies can be either files or other targets
- A target usually (but not necessarily) refers to a final produced file (like an executable)

Rule Format (Note the tab!)

```
target [target...] : [dependency...]
        [command...]
```

Rule Example

```
translate: translate.c
        gcc -ansi -Wall -pedantic translate.c -o translate
```

To compile this you would:

- Run `make translate`
- Or just run `make`
  - Without a target specified, the first target in the makefile is executed
- `Make` will only compile *translate.c* if:
  1. The target (executable) does not exist

  Or:

  2. The target has an older modification date than its dependency (C source file)

Some useful Makefile features

- Lines starting with # are comments

```
# This is a comment
translate: translate.c
	gcc -ansi -Wall -pedantic translate.c -o translate

clean:
	-ls | grep "\.o"
	@rm -f *.o *.out translate
```

Some useful Makefile features

- Lines starting with # are comments
- Commands starting with @ will not be echoed by Make

```
# This is a comment
translate: translate.c
	gcc -ansi -Wall -pedantic translate.c -o translate

clean:
	-ls | grep "\.o"
	@rm -f *.o *.out translate
```

Some useful Makefile features

- Lines starting with `#` are comments
- Commands starting with `@` will not be echoed by `Make`
- Commands starting with `-` instruct `Make` not to stop execution if the command fails (ignores errors)

```
# This is a comment
translate: translate.c
	gcc -ansi -Wall -pedantic translate.c -o translate

clean:
	-ls | grep "\.o"
	@rm -f *.o *.out translate
```

Variables

Suppose you want to change the compiler flags for all your rules

- Tedious
- Error prone (might miss something)

Solution: ?

Variables

Suppose you want to change the compiler flags for all your rules

- Tedious
- Error prone (might miss something)

Solution: *Variables*

Variables (Cont'd)

- Define variables using `=` (or `:`)
- To use the variable value: Start with `$` and enclose with `(...)` or `{...}`

```
CFLAGS=-ansi -Wall -pedantic

# This is a comment
translate: translate.c
	gcc $(CFLAGS) translate.c -o translate

clean:
	@rm -f *.o *.out translate
```

Variables(Cont'd)

Automatic Variables

- Provided as a utility by `Make`
- Dynamically defined on a per-rule basis
- Useful to write less, be more efficient

Variables(Cont'd)

Some useful automatic variables

- `$@` The target filename
- `$*` The target filename with no extension
- `$<` The first dependency filename
- `$^` All dependency filenames, space-separated, no duplicates
- `$+` Like `$^`, but with duplicates
- `$?` All dependencies newer than target, space-separated

Variables(Cont'd)

Some useful automatic variables

- `$@` The target filename
- `$*` The target filename with no extension
- `$<` The first dependency filename
- `$^` All dependency filenames, space-separated, no duplicates
- `$+` Like `$^`, but with duplicates
- `$?` All dependencies newer than target, space-separated

```
CFLAGS=-ansi -Wall -pedantic

# This is a comment
translate: translate.c
	gcc $(CFLAGS) $< -o $@

clean:
	@rm -f *.o *.out translate
```

Patterns

Make also provides patterns

- With patterns, one can group together rules with common actions
- For instance, all *.c* files should be compiled with `gcc` using our defined `CFLAGS`
- Use `%` to match any string of characters (0 or more)

```
CFLAGS=-ansi -pedantic -Wall

%.o: %.c lib.h
        gcc $(CFLAGS) -c $< -o $@
```

A complete example (courtesy of Foivos Zakkak)

```
CFLAGS=-ansi -pedantic -Wall

test3: test.o
        gcc $(CFLAGS) test.o -o $@

all: test1 test2 test3

test1: main.o lib1.o
        gcc $(CFLAGS) main.o lib1.o -o test1

test2: main.o lib2.o
        gcc $(CFLAGS) $^ -o $@

%.o: %.c lib.h
        gcc $(CFLAGS) -c $< -o $@

clean:
        -rm *.o
```

A note about commands

- Each command in a rule is executed in a separate shell
- Format actions as one shell command if needed, using `;`, `\`

```
lsdir_wrong :
        cd dir
        ls -l
```

This does not work

```
lsdir_right :
        cd dir;\
        ls -l
```

This works

Invoking Make

`make` Look for M/makefile, execute first target

`make <target>` Default Makefile used, execute specified target

`make -f <file>` Use specified file as Makefile

Some Links

man make Manual on Linux

GNU Make homepage https://www.gnu.org/software/make/

Makefile tutorial https://cs.colby.edu/maxwell/courses/tutorials/maketutor/

Any questions?