Introduction to Make

Kurt Schmidt

Dept. of Computer Science, Drexel University

September 19, 2016

Intro

make

- Automates certain tasks
  - Usually simple command-line stuff
  - Compiling multi-file programs
  - Archving/extracting
  - Software installation
- Often used to manage builds
  - Compiles only as necessary
    - Uses file modification times to decide when it is necessary

Basic Make

Make Rules

- A basic makefile consists of *rules*

```
target : dependencies
TABcommand1
TAB[command2]
...
```

- The tab character precedes the rule
- The target is (usually) a file to be created
- Each command is executed in its own shell[1]

[1] We'll look at multi-line commands in a bit

Make Example

- E.g.

```
program : main.c
    gcc main.c -oprogram
```

- `main.c` should already exist
  - Or, there's another target that creates it
- `main.c` will only be compiled if:
  1. `program` doesn't exist, or
  2. `main.c` is newer than `program`

Dependency Recursion

- Dependencies are checked recursively down the tree:

```
program : main.o
   gcc main.o -oprogram

main.o : main.c
   gcc -c main.c
```

- Nothing happens if `program` is newer than `main.o`, and `main.o` is newer than `main.c`
- If `main.o` doesn't exist, or is older than `main.c`, it will be rebuilt, then `program` will be rebuilt
- If `program` doesn't exist, or is older than `main.o`, it will be rebuilt

Slightly More Involved Example

```
program : main.o service.o
   gcc main.o service.o -oprogram

service.o : service.c service.h
   gcc -c service.c

main.o : main.c service.h
   gcc -c main.c
```

- If `main.c` is updated, then `main.o` and `program` are rebuilt
- If `service.c` is updated, then `service.o` and `program` are rebuilt
- If `service.h` is updated, everybody is updated

Recipe Without Commands

A target may simply depend on other targets:

```
all : this that other

this : this.c
   gcc this.c -o this

that : that.c
   gcc that.c -o that

other : other.c
   gcc other.c -o other
```

Continuing Lines

Continuing Lines

- Use \ to continue a dependency list or a command `program` are rebuilt

```
program : main.o curses.o utils.o keyboard.o \
            deck.o suits.o
   gcc -oprogram main.o curses.o utils.o keyboard.o \
         deck.o suits.o

...
```

Multi-Line Commands

- So now you can pass more than one line to the shell
- Beware, the shell won't get any newlines (you escaaped them)
- So, use the shell's separator (most shells use ;)

```
input :
   f=‘mktemp‘ ;\
   i=1 ;\
   while [ $$i -le 10000 ] ;\
      do echo $$i >> "$$f" ;\
      i=‘expr $$i + 1‘ ;\
   done ;\
   shuf "$$f" >> input
   rm "$$f"
```

- Note, `make` uses Bourne (or, a minimal Bourne-compliant) shell by default

Command Prefixes

Command Prefixes

- Turn of make echo by preceding line with a `@`

```
blah :
    @echo "Don't say this line twice"
```

- If any command returns an unsuccessful status, `make` reports the error and exits
- Precede a line with a `-` to have `make` ignore the status
- Note, each of those `rm` statements happens in a separate shell

```
clean :
    -rm program # fails if program doesn't exist
    -rm *.o     # We want this to happen, regardless
```

Invocation

Specifying Input File

- Specify a makefile using the option `-f` option to `make`:

```
$ make -f someMakeFile
```

- If not specified, `make` looks in the current directory for:
  1. `makefile`
  2. `Makefile`

Specifying a Target

- Make allows you to specify target(s)

  `make [options] [target]`
- If no target is specified, `make` builds the first target it finds
- `-n` (dry run) is another handy option
  - Just print commands that would execute, w/out executing them

Phony Targets (Gnu only)

- Some targets exist for convenience
- We don't actually want to produce a file
- Commands won't run if a file of the same name exists
- We can declare targets as phony:

```
.PHONY : clean

clean :
   -rm program # fails if program doesn't exist
   -rm *.o     # We want this to happen, regardless
```

- No times are compared, commands run every time

Macros

Defining Macros in a Makefile

Macros can be defined in a makefile:

```
OBJS = main.o curses.o utils.o keyboard.o \
           deck.o suits.o
cc = gcc
CFLAGS =

program : $(OBJS)
   $(cc) $(CFLAGSO $(OBJS) -o program

main.o : main.c
   $(cc) -c $(CFLAGS) main.c

$(OBJS) : sysdefs.h

...
```

Macro Substitution

Evaluates the macro, after some substitutions.

```
SOURCE = main.c curses.c utils.c keyboard.c \
           deck.c suits.c

OBJS = ${SOURCE:.c=.o}

cc = gcc
CFLAGS =

...
```

Defined Macros

$@	Name of current target
$<	Name of first prerequisite
$^	All prerequisites
$?	All prerequisites newer than target

```
program : main.c service.h
   $(cc) $(CFLAGS) $< -o $@
...
```

Choosing a Different Shell

- If you want to use a different shell, say, `bash`, to interpret the commands
- Set the `SHELL` variable at the top to modify all commands:

```
SHELL := /bin/bash

...
```

- You can do this for individual targets:

```
program : SHELL:=/bin/bash
program : main.c service.h
   $(cc) $(CFLAGS) $< -o $@
...
```

Suffix Rules

- Some rules easy enough to be generalised
- If target has the same name as a dependency, but different suffix
- E.g., compile C files into object code

```
big.o : big.c this.h that.h other.h

%.o  :  %.c
   $(cc) -c $(CFLAGS) $<
```

- Other dependencies can be named
- Can also be specified this way:

```
.c.o :
   $(cc) -c $(CFLAGS) $<
```

Recap

Recap

- Make files can do anything you do at the command line
- Care has to be taken to make them portable
- We've looked at fairly simply makefiles
  - Still wildly useful
  - Makefile might call other makefiles
  - Macros can be defined in a separate file, used by several makefiles