Make

Warwick RSE

Overview

- For simple programs just issue build command when needed
  - gcc test1.c test2.c test.c -o test
- What happens as the list gets longer?
- What happens if only some of the files need recompiling?
- What about building in parallel?

Build scripts

- Just write a script that executes the compiler lines that you need
- Can be any scripting language that can execute the compiler
- Typically shell script (bash, csh, etc.)
- Solves the problem of typing long compile lines
- Nothing else

Makefiles

- The most widely used build tool is “make”
  - Originated in 1976 after someone wasted a morning debugging a program that he just hadn’t compiled correctly
- Several different variants, but there is a single underlying standard they all support

Makefiles

- You start make just by typing “make”
- It then looks for a file called either “Makefile” or “makefile” that contains instructions on how to build the code
- If you’ve built large codes from source you might expect a “./configure” stage
- That’s another tool called “GNU Autotools” that builds a makefile for your system. Much more sophisticated
- Can write makefiles by hand

Makefiles

- Make is very powerful
- At core, simple idea - Makefile is just a list of rules
    - Target - what is to be made
    - Prerequisites - what needs to be made before this target can be. Can be files or other targets
    - Recipe - The command to build this target

Makefile whitespace

- Makefiles indent lines using TAB characters
    - This is not optional, spaces will not do

Makefile recipes

- Rules look like

```
Target : prerequisites

      recipe
```

- The first rule is special and is the default rule. It is built when you just type "`make`"
- Can make any rule using "`make {target}`"

Example makefile

```
code:  test.o
        cc -ocode test.o
test.o:test.c
        cc -c test.c
```

- First rule is default, depends on test.o
- Second rule says how to build test.o

Variables in makefiles

- Can set variables in makefiles
    - `variable = value`
- Variables are referenced by name using
    - `$(variable)`

Variables in makefiles

- There's a slight wrinkle if you set a variable equal to another variable

  -
    ```
    var1 = test
    var2 = $(var1)
    var1 = test2
    ```
- Leaves var2 with the value "test2" because it's only evaluated when used
- You have to use the ":=" operator rather than "=" to assign *here and now*

  -
    ```
    var1 = test
    var2 := $(var1)
    var1 = test2
    ```
- Leaves var2 with the value "test"

Automatic variables

- As well as variables that you've set make itself sets some automatic variables
  - `$@` - target of the current rule
  - `$<` - first prerequisite in the list
  - `$^` - list of all prerequisites separated by spaces
- Lots more (`https://www.gnu.org/software/make/manual/html_node/Automatic-Variables.html`)

Implicit rules

- Most recipes are very similar, so can automate them using implicit rules
- Any rule having the form of an implicit rule can be specified without a recipe
  - Recipe for implicit rule used instead
- Simplest example is

  -
    ```
    %.o:%.c
          cc -c $<
    ```
- https://www.gnu.org/software/make/manual/html_node/Pattern-Rules.html

Other bits of make

- Make can run any command that you want
- Can use "phony" targets that don't map to a file to cause make to do anything that you want
  - There is a ".PHONY" command to help with this
- Common example is "`make clean`"
  - Typically used to remove intermediate files

A simple makefile

```
cc = cc

.PHONY: clean

code: test.o
        $(cc) -ocode test.o
%.o:%.c
        $(cc) -c $<
test.o:test.c

clean:
        @rm -rf code
        @rm -rf *.o
```

Another advantage of make

- Because the makefile contains dependencies make now knows in which order to build files
- In fact, it can work out multiple possible paths if it needs to
- Parallel build
    - `make -j {number_of_processors}`
- Will only use as many processors as it can

Preprocesor Defines

- EPOCH uses several pre-processor defines
- Files that are preprocessed have the capital-F90 extension
- These are controlled via all commented out lines like

```
#DEFINES += $(D)PARTICLE_ID
```

- $(D) is a Makefile variable adding the correct characters (usually -D) for a define
- These add onto a Makefile variable DEFINES
- DEFINES is then used in the compile step

Make Options

- EPOCH also has arguments controlling the Makefile behaviour
- `make COMPILER=gfortran` for instance

```
ifeq ($(strip $(COMPILER)),gfortran)
   ...
endif
```

- first line strips spaces from `COMPILER=` and compares to string `gfortran`. In the body, set up flags etc for this compiler
- these aren't make rules
  - run in order before targets
  - don't usually run commands except to capture result e.g.

```
MACHINE := $(shell uname -n)
```