Make and makefiles

J. Tarus

CSC – Tieteen tietotekniikan keskus Oy
CSC – IT Center for Science Ltd.

GNU make

- The material presented here assumes the use of gnu make (some parts may not work with other implementations of make)
- For more information see
  - www.gnu.org/software/make/manual/

Make in general

- In practice codes are usually separated into several different files.
- When compiling the code, one should either compile everything, which can be too slow, or have booking to show what needs to be compiled.
- make makes things life easier by taking the role of bookkeeper.
- Makefile defines what rules for needed actions.

Basic form for a makefile

- Syntax for makefile

```
target: dependent1 dependent2
        command
```

- Target is usually the file that is produced by the command
- Target can also be a name for an action (for example *clean*)

Basic form for a makefile

- Dependent is a source that is used for making a target.
- Dependents are not always needed

```
clean:
	rm *.o
```

- Above mentioned command will be always executed as there there is no target *clean*

Basic form for a makefile

- *Command* is the function make executes (or to be precise it passes the commands to shell).
- NOTE: there is a tab character in front of the commands (not spaces)
- Without tab:

```
makefile:2: *** missing separator (did you mean TAB instead of 8 spaces?).  Stop.
```

Basic form for a makefile

- Make checks when the dependents were last changed. If any the dependents is newer than the target, the target will be rebuild according the commands.
- Dependents can be targets for other rules (for example object files). In that case make rebuilds dependents first.

Simple example

```
edit : main.o ali1.o ali2.o ali3.o
          f90 −o edit main.o ali1.o ali2.o ali3.o
main.o : main.f90
          f90 −c main.f90
ali1.o : ali1.f90
          f90 −c ali1.f90
ali2.o : a l i 2 . f90
          f90 −c ali2.f90
ali3.o : a l i 3 . f90
          f90 −c ali3.f90
clean :
          rm edit main.o ali1.o ali2.o ali3.o
```

Targets

- By, default make starts from the first target. *edit* in the previous example.
- You can also define the target from command line:

```
make clean

make main.o
```

Macros

- In the makefile example we had the line *main.o ali1.o ali2.o ali3.o* in several places. Instead of repeating the same line many times we can use macros. (Macros are sometimes called variables.)
- They are defined by

```
MACRO=value
```

- And referred by

```
$(MACRO)
```

Macros

- By convention they are usually all capital letters.
- In the previous makefile we could use a macro called OBJ.

Macros

```
OBJ=main.o ali1.o ali2.o ali3.o
edit : $(OBJ)
          f90 −o edit $(OBJ)
main.o : main.f90 type.mod
          f90 −c main.f90
ali1.o : ali1.f90 type.mod
          f90 −c ali1.f90
ali2.o : ali2 . f90 type.mod
          f90 −c ali2.f90
ali3.o : ali3 . f90 type.mod
          f90 −c ali3.f90
clean :
          rm edit $(OBJ)
```

Macros

- Some common macros
  - CC
  - CFLAGS
  - FC
  - FCFLAGS
  - LDFLAGS
  - OBJ
  - SRC
- These are also used in default/built-in rules that make uses.

Special macros

- $@
  - The name of the target

```
client: client.c
        $(CC) client.c -o $@
```

Special macros

- $<
  - The name of the first dependent

```
client: client.c
        $(CC) $< -o $@
```

Special macros

- $?
  - This stores the list of dependents more recent than the target

```
client: client.c
        $(CC) $? -o $@
```

Special macros

- $^
  - List of all the dependents. Duplicates are removed

```
# print the source to the screen
viewsource: client.c server.c
        less $^
```

- $+
  - Same as $^, but duplicates are not removed

Special macros

- $*
  - the prefix shared by target and dependent files

```
main.o: main.c
	$(CC) -c -o $*.o $*.c
```

Special characters

- / continues a line
- Everything after # is comment
- Make echos all the commands that are executed. This can be prevented by using @ sign at the beginning of the commad

```
@echo $(USER)
```

Special characters

- If there is an error in executing a command make stops. This can be prevented by a - sign

```
clean:
	-rm edit
	-rm $(OBJ)
```

SHELL variables

- You can also shell variables (HOME,USER,SHELL...) in the makefile

```
install:
	make
	mv edit $(HOME)/bin/.
	make clean
```

- You can also define your shell

```
SHELL=/bin/bash
```

IMPLICIT RULES

- You can use special characters to define implicit rules
- Quite often target and dependent share the name (different extensions). So one can define an implicit rule for compiling an object file from a f90 file.

```
%.o: %.f90
	$(F90) $(FFLAGS) -c -o $@ $<
```

Some GNU make's functions

- wildcard

```
SRC=$(wildcard *.f90)
```

- addprefix

```
SRCDIR=../src/
SRCM=$(addprefix $(SRCDIR),$(SRC))
```

- patsubst

```
OBJ=$(patsubst %.f90,%.o,$(SRC))
```

VPATH

- Sometimes the source files are distributed to different directories or your objects will be written to a different directory
- To locate the sources you can use VPATH.
  - VPAT=path1:path2:path2
- Note: There is no `VPATH' support specified in POSIX.  Many `make's have a form of `VPATH' support, but its implementation is not consistent amongst `make's.

Command line options

- -j parallel execution
- -n shows the commands, but will not execute them
- -p shows default rules and values for variables
- macro definitions from command line

```
make ”FFLAGS=−O” ”LDFLAGS=−s ” edit
```

- make -f makefile_old

Example

```
SRC=main.f90 ali1.f90 ali2.f90 ali3.f90
OBJ=$(patsubst %.f90,%.o,$(SRC))
F90=gfortran
FFLAGS=
VPATH=../src:../modules
.PHONY: clean

#Implicit rule fore compiling f90 files
%.o: %.f90
	$(F90) $(FFLAGS) -c -o $@ $<

edit: $(OBJ)
	$(F90) $(FFLAGS) -o $@ $(OBJ)
	mv edit ../.
clean:
	-rm *.o
	-rm ../edit
	-rm *.mod
#module dependencies
ali1.o:modules.o
ali2.o:modules.o
ali3.o:modules.o
```