Make Tutorial

Le Yan

User Services
High Performance Computing @ LSU/LONI
Outline

• What is make

• How to use make
  – How to write a makefile
  – How to use the “make” command
What is Make

• A tool that
  – Controls the generation of executable and other non-source files (libraries etc.)
  – Simplifies (a lot) the management of a program that has multiple source files
• Have many variants
  – GNU make (we will focus on it today)
  – BSD make
  – …
• Other utilities that do similar things
  – Cmake
  – Zmake
  – …
What is Make

• A tool that
  – Controls the generation of executable and other non-source files (libraries etc.)
  – Simplifies (a lot) the management of a program that has multiple source files
• Have many variants
  – GNU make (we will focus on it today)
  – BSD make
  – ...
• Other utilities that do similar things
  – Cmake
  – Zmake
  – ...
Why having multiple source files

• It is very important to keep different modules of functionalities in different source files, especially for a large program
  – Easier to edit and understand
  – Easier version control
  – Easier to share code with others
  – Allow to write a program with different languages
From source files to executable

• Two-step process
  – The compiler generates the object files from the source files
  – The linker generates the executable from the object files

• Most compilers do both steps by default
  – Use “-c” to suppress linking
Compiling multiple source files

- Compiling single source file is straightforward
  - `<compiler> <flags> <source file>`
- Compiling multiple source files
  - Need to analyze file dependencies to decide the order of compilation
  - Can be done with one command as well
    - `<compiler> <flags> <source file 1> <source file 2>...`
A “Hello world” example (1)

Source file | Purpose
---|---
Common.f90 | Declares a character variable to store the message
Hello.f90 | Prints the message to screen
Adjust.f90 | Modifies the message and prints it to screen
Main.f90 | Calls functions in hello.f90 and adjust.f90
A “Hello world” example (2)

[lyan1@eric2 make]$ ls
adjust.f90  common.f90  hello.f90  main.f90
[lyan1@eric2 make]$ ifort common.f90 hello.f90
adjust.f90  main.f90
[lyan1@eric2 make]$ ./a.out
Hello, world!

Hello, world!
Command line compilation

• Command line compilation works, but it is
  – Cumbersome
    • Does not work very well when one has a source tree with many source files in many sub-directories
  – Not flexible
    • What if different source files need to be compiled using different flags?

• Use Make instead!
How Make works

• Two parts
  – The Makefile
    • A text file that describes the dependency
  – The “make” command
    • Compile the program using the dependency provided by the Makefile

```bash
[lyan1@eric2 make]$ ls
adjust.f90  common.f90  hello.f90  main.f90  Makefile
[lyan1@eric2 make]$ make
ifort common.f90 hello.f90 adjust.f90 main.f90
[lyan1@eric2 make]$ ls
adjust.f90  a.out  common.f90  common.mod  hello.f90
main.f90  Makefile
```
A Makefile with only one rule

```makefile
all:
  ifort common.f90 hello.f90 adjust.f90 main.f90
```

- **Target**: all
- **Action**: shell commands that will be executed
- **Explicit rule**: A mandatory tab

```
[lryan1@eric2 make]$ cat Makefile
```

LONI Fortran Programming Workshop, LSU  Feb 13-16, 2012
Exercise 1

• Copy all files under /home/lyan1/traininglab/make to your own user space

• Check the Makefile and use it to build the executable
Makefile components

- Explicit rules
  - Purpose: create a target or re-create a target when any of prerequisites changes
  - Syntax:

```
target: prerequisites
  (tab) action
```

- Implicit rules
- Variable definition
- Directives
Explicit rules (1)

• Multiple rules can exist in the same Makefile
  – The “make” command builds the first target by default
  – To build other targets, one needs to specify the target name
    • make <target name>

• A single rule can have multiple targets separated by space

• An action (or recipe) can consist of multiple commands
  – They can be on multiple lines, or on the same line separated by semicolons
  – Wildcards can be used
  – By default all executed commands will be printed to screen
    • Can be suppressed by adding “@” before the commands
Explicit rules (2)

• How file dependencies are handled
  – Targets and prerequisites are often file names
  – A target is considered out-of-date if
    • It does not exist, or
    • It is older than any of the prerequisites
A Makefile with many rules

```
all: main.o adjust.o hello.o
    ifort main.o adjust.o hello.o
main.o: main.f90
    ifort -c main.f90
adjust.o: adjust.f90 common.mod
    ifort -c adjust.f90
hello.o: hello.f90 common.mod
    ifort -c hello.f90
common.mod: common.f90
    ifort -c common.f90
```
Exercise 2

• Write a Makefile using the template provided on the previous slide and “make”
• Run “make” again and see what happens
• Modify the message (common.f90) and “make” again
• Add a new rule “clean” which deletes all but the source and makefiles (the executable, object files and common.mod), and try “make clean”
Variables in Makefile (1)

• These kinds of duplication are error-prone
• One can solve this problem by using variables

```
all: main.o adjust.o hello.o
    ifort main.o adjust.o hello.o
main.o: main.f90
    ifort -c main.f90
adjust.o: adjust.f90 common.mod
    ifort -c adjust.f90
hello.o: hello.f90 common.mod
    ifort -c hello.f90
common.mod: common.f90
    ifort -c common.f90
```
Variables in Makefile (2)

- Similar to shell variables
  - Define once as a string and reuse later

Without variables

```
all: main.o adjust.o hello.o
  ifort main.o adjust.o hello.o
main.o: main.f90
  ifort –c main.f90
```

With variables

```
FC=ifort
OBJ=main.o adjust.o hello.o

all: $(OBJ)
  $(FC) $(OBJ)
main.o: main.f90
  $(FC) –c main.f90
```
Automatic variables

• The values of automatic variables change every time a rule is executed
• Automatic variables only have values within a rule
• Most frequently used ones
  – $@$: The name of the current target
  – $^$: The names of all the prerequisites
  – $??: The names of all the prerequisites that are newer than the target
  – $<$: The name of the first prerequisite
Implicit rules (1)

- Tells Make system how to build a certain type of targets
  - GNU make has a few built-in implicit rules
- Syntax is similar to an ordinary rule, except that “%” is used in the target
  - “%” stands for the same thing in the prerequisites as it does in the target
    - There can also be unvarying prerequisites
    - Automatic variables can be used here as well

```
%.o: %.c
    (tab) action
```
Implicit rules (2)

```
CC=icc
CFLAGS=-O3

%.o : %.c
    @$\{CC\} \$(CFLAGS) -c -o $@ $<

data.o: data.h
```

- In this example, any .o target has a corresponding .c file as an implied prerequisite
- If a target needs additional prerequisites, write a action-less rule with those prerequisites
Exercise 3

• Rewrite the Makefile from Exercise 2
  – Define an implicit rule so that no more than 3 explicit rules are necessary (excluding “clean”)
  – Use variables so that no file name appears in the action section of any rule
Directives

• Make directives are similar to the C preprocessor directives
  – E.g. include, define, conditionals
• Include directive
  – Read the contents of other Makefiles before proceeding within the current one
  – Often used to read
    • Top level and common definitions when there are multiple sub-directories and makefiles
Command line options of make (1)

- **-f <file name>**
  - Specify the name of the file to be used as the makefile
  - Default is GNUmakefile, makefile and Makefile (in that order)
  - Multiple makefiles may be useful for compilation on multiple platforms

- **-s**
  - Turn on silent mode (as if all commands start with an “@”)

2/14/2012
LONI Fortran Programming Workshop, LSU
Feb 13-16, 2012
Command line options of make (2)

• `-j <number of jobs>`
  – Build multiple targets in parallel
• `-i`
  – Ignore all errors
  – A warning message will be printed out for each error
• `-k`
  – Continue as much as possible after an error.
Exercise 4

• Take a look at a real life makefile
  – /home/lyan1/traininglab/valgrind/Makefile
  – Makefile for a memory profiler Valgrind
Questions?