Introduction To Unix **Make**

CS 471/Math 471

Brian T. Smith, UNM
Andrew C. Pineda, UNM

What Is The **Make** Utility?

- A Unix utility program to help maintain a collection of files, making up a complete program or document
- Developed in the earlier 70s by Stuart Feldman in response to the need to maintain the files making up the first portable Fortran 77 compiler
- Recommend the tutorial, Ch. 5, IHPSC text

The Essence of **Make**

- A formal mechanism for specifying the dependencies of input files on a final file or result and for specifying the actions required to update the final file when the input files are modified.
- Definition of "Dependency"
  - file A depends on file B means that A is generated from B

An Example Of Dependencies And The **Make** Concept

- Suppose
  - file FINAL depends on files A and B
  - file B depends on file E
  - file A depends on files M and N.
  - file E is changed.
- Then, to update FINAL,
  - process file E
  - process file B using the updated E
  - recreate FINAL from updated B and old A

Specifying Dependencies

- File dependencies are specified in a file
- Usually the file is called Makefile or makefile
- The form of the dependencies is:

```
<output>: <list of files representing dependencies>
<tab>   <execute command 1>
<tab>   <execute command 2>
        … produces file <output>
```

- There may be no commands in a rule

The Simplest (Non-Empty) **Make** File

- Suppose you wanted to compile a program myprog.f to produce the executable myprog
- The file myprog depends on myprog.f
- Create a **make** file called Makefile with the following contents:

```
myprog:  myprog.f
     f90 -O -o myprog myprog.f
myprog.out: myprog
     myprog > myprog.out
```

The **Make** Command

– The general form of the make command is:

make <make_options> <target>

– <target> is the file to generated; the **make** file is scanned looking for a rule to generate <target>

– type:

man make

- to read about the many make options
  - a commonly used option is -f <name> which specifies a name other than Makefile or makefile for the **make** file

Executing **Make** With A **Make** File

- To create myprog.out, type:

  make myprog.out

  – it compiles myprog.f (if myprog does not exist or myprog.f has been modified) to generate myprog and then executes myprog to create the file myprog.out
- When no target is specified in the **make** command, it executes the first dependency

No **Make** File In The Current Directory -- Default Rules

- Consider typing with no **make** file present:

  make myprog

  – uses default implicit rules to generate myprog
  - looks around for a file myprog.o in the current directory so that the loader/linker can generate myprog with default options
  - if no file myprog.o, it looks around in the current directory for a file myprog.f (or myprog.c, …) and uses the appropriate compiler with default options to generate myprog.o

When The **Make** Command Fails

– Designed to fail gracefully. For example:

- an unknown target

  make <unknown_target>
- response

  Make: Don't know how to make <unknown_target>. Stop
- an up-to-date target

  make <uptodate_target>
- response

  <uptodate_target> is up to date.

Failure Of Target Commands

- Target commands satisfy the syntax of the Bourne Shell (/bin/sh)
- If a command fails (say f90 compilation error), command diagnoses the fault and **make** executes a stop command.

More Advanced **Make** Files

- Example

```
myprog: myprog.o sub1.o sub2.o mytime.o
# Example of a comment and a continuation (backslash)
        f90 -o myprog myprog.o sub1.o sub2.o \
                mytime.o
#Compiler does  no optimization here
myprog.o: myprog.f
        f90 -c myprog.f
# Optimization used for sub1, sub2, and mytime
sub1.o: sub1.f
        f90 -c -O sub1.f
```

Macros

- To avoid repetitious aspects of **make** files
  - predefined macros $<, $@, $?, $*
    - behave like variables that are instantiated from a context, taking on values of targets or dependencies
    - their values are set at the beginning of each rule
  - user-defined macros
    - are defined in assignments near the beginning of the **make** file and available for use below in the **make** file

$@ Dynamic Macro

- $@ is a macro that is given the value of (instantiated with) the name of the target that invokes the rule
- When used in the dependency list, it must be preceded by a $

```
myprog: $$@.o sub1.o sub2.o mytime.o
        f90 -o $@ $@.o sub1.o sub2.o mytime.o
```


Implicit Rules

- An implicit rule is a rule appearing in a **make** file that implicitly specifies a target and a dependency
  - they may override a default implicit rule
  - implicit rules are of the form:

```
.x.y:
    &lt;commands to generate &lt;name&gt;.y from &lt;name&gt;.x
```

- Example of an implicit rule:

```
.f.o:
    f90 -O -c $<
```


How An Implicit Rule Is Invoked

- .x.y means that you invoke this implicit rule when you want to generate a file named <name>.y and you can find a file named <name>.x in the current directory
- <name>.x will then be used in the commands in this rule to generate the file <name>.y

$< Dynamic Macro

- $< macro is set to the name of the target file that invokes an implicit rule
  - Example:

```
.f.o:
        f90 -O -c $<
```

    - $< is set to the name that invokes this implicit rule, for example, myprog.f
    - the command f90… generates the file myprog.o from the file myprog.f using the value of $<

$? Dynamic Macro

- $? is a list of file names that **make** uses to create a particular target
- One of its important uses is to print the list of dependencies in complicated **make** files

```
create: $(OBJECTS)
        echo "The dependencies are" $?
```

`$*` Dynamic Macro

- `$*` is used only in implicit rules and is the root name of the file that initiates the implicit rule
- One of its important uses is to print the root name that initiated an implicit rule in complicated **make** files

```
.x.y:
        echo "The root name for implicit rule .x.y is" $*
```


A User-Defined Macro

- A string of characters
- Defined in an assignment-like statement
- The form of the definition is:

```
<name> = <string_of_characters>
```

- Convention is to use capital letters for the name

```
OBJECTS = myprog.o sub1.o sub2.o mytime.o
```

- A reference to the string is `$(<name>)`

```
myprog: $(OBJECTS)
    f90 -o $@ $(OBJECTS)
```


An Example

```
COMPILER = f90     # Can change this to xlf90
FFLAGS = -c -O     # Can change to specify diff. opts
OBJECTS = myprog.o sub1.o sub2.o mytime.o

myprog: $(OBJECTS)
        $(COMPILER) -o $@ $(OBJECTS)
.f.o:
        $(COMPILER) $(FFLAGS) $<
```

- To override a macro, you can type:

```
make myprog "FFLAGS=-c -O3" "COMPILER=xlf90"
```


Silent **Make** Options

- The -s option does not echo the commands of the rules as they are executed -- silent mode
- Insert @ before command to not echo the command
- Insert the dependency:

```
.SILENT:
```

  - in the **make** file causes the **make** file to be always run in silent mode

A Help Document For Your **Make** file

- You can create a help document for your **make** file by using @echo command after the target help

```
help:
    @echo "This make file supports the following actions"
    @echo "make run.out  -- creates output from current data"
    @echo "make run -- create the executable file"
```

- The command

```
make help
```

  - lists the above lines of help information

Other **Make** Options

- Option -n lists all commands that would be executed (but are not) to make the target
- Option -t touches all targets that would be (re-) created to make the specified target
  - useful when only comments are modified
- Option -d prints debugging information from **make** indicating what **make** is doing
- Option -f <name> specifies a specific make file to be used in place of the defaults (Makefile)

A Complete Make File

```
# An example make file for Fortran programs
#        Establish some useful macros
SOURCES = myprog.f sub1.f sub2.f mytime.f
OBJECTS = myprog.o sub1.o sub2.o mytime.o
FFLAGS = -O -c
FCOMPILER = f90
#        My help document for this make file
help:
      @echo “make help gives the following output”
      @echo “make myprog creates the executable”
      @echo “ … lines of help deleted here”
```


Complete Example Continued

```
#        To execute my program
myprog.out: myprog mydata
      myprog < mydata > myprog.out
#        To create the executable file  myprog
myprog: $(OBJECTS)
      $(FCOMPILER) -o myprog $(OBJECTS)
#        To compile the Fortran components
.f.o:
      $(FCOMPILER) $(FFLAGS) $<
#        To compile the Fortran with debugging options set
debug: $(SOURCES)
      $(FCOMPILER) -g -o debug $(SOURCES)
```


Example **Make** File Continued

```
#        To print the source files
printsrc:
      enscript -2rG $(SOURCES)
#           To print the output file
print.output: myprog.out
      lpr myprog.out
#           Clean up intermediate file
clean:
      /bin/rm core debug *.o
```