Understanding Makefiles

COMP 2400, Fall 2008

Prof. Chris GauthierDickey

Why Makefiles?

- Unix has been around for a long, long time
- Makefiles have been around about as long (1977)
- Makefiles are the defacto standard when it comes to Unix compilation
  - Many modern projects just let the IDE do dependency tracking
  - IDEs are great for that, but have trouble with general non-language build instructions

More on make

- Various versions have floated around over the years
- Most Unix systems have GNUmake, which we'll use, or BSDmake
    - It has some non-portable (to other make) things, but that's usually not a problem
- Most open-source projects use make, usually generated by the auto-tools

What do makefiles do?

- A makefile is a set of instructions basically telling the 'make' program how to do something
  - It's that general--it doesn't have to be compiling source, but it mostly is
  - It's a program that looks at dependencies and 'does something' when those dependencies have been modified

Make for compilation

- The most common use of make is for compilation
  - Doing anything system admin related will often force you to do this more than you can imagine
    - Packages, projects, etc, especially open-source projects, are often compiled from scratch
    - Kernels are also recompiled, so understanding make is good for that too (even though today it's mostly automated)

Make syntax

- Easy stuff first:
  - # begins a comment, regardless of where it's placed
    - It doesn't line-wrap
    - It can begin in the middle of a line

Rules and Targets

- Rules tell make how to execute a series of commands
- target should be the result of the make
- dependencies are what make checks to determine if target should be remade

```
target:  dependencies ...
    commands
    ...
```

Minor Issues

- Makefiles are particular about spacing
    - Your set of commands each occur on one line
    - You MUST put a tab character before each command
- The commands can be any Unix program found in your path

More on Targets

- Make works by looking at the modification date of the target and dependencies
  - If the target doesn't exist, it executes the associated commands
  - If the target date is older than any of the dependencies, the commands are executed

An Aside on C

- As a Unix user/admin, you may have to compile projects to add functionality
  - For example, you may want to add a feature for your specific hardware to the kernel
- Many of these are written in C, so it's useful to understand what C is and looks like

C Compilation

- C supports separate compilation, meaning that any C file can be turned into code that can be combined
- Each C file is turned into an object file by the compiler:
    - gcc -c file.c
        - Here, gcc is the compiler, the flag -c says to turn it into an object file, by default, it will be called file.o

C Compilation...

- Once you have a collection of object files, they are combined into an executable file
  - gcc -o myprog file.o foo.o bar.o
    - This results in a program called myprog that can be executed
- Every C program links with the C library
- Every Unix system is based on the C library, so it's always there

An Example

```
# example taken from the GNU make manual

edit : main.o kbd.o command.o display.o \
            insert.o
             cc -o edit main.o kbd.o command.o display.o \
                        insert.o

     main.o : main.c defs.h
             cc -c main.c
     kbd.o : kbd.c defs.h command.h
             cc -c kbd.c
     command.o : command.c defs.h command.h
             cc -c command.c
     display.o : display.c defs.h buffer.h
             cc -c display.c
     insert.o : insert.c defs.h buffer.h
             cc -c insert.c
     clean :
             rm edit main.o kbd.o command.o display.o \
                insert.o search.o files.o utils.o
```

Things to Note

- Your target, dependencies and commands can only run a single line--no newlines
  - You continue a line with a \, which escapes the newline
- Dependencies can refer to targets in the Makefile also
- We have a special target called 'clean', which removes all the old object files

Variables

- make can use variables to simplify the Makefile
- We can use them to make sure we don't miss things, like adding all the dependencies

```
objects = main.o kbd.o \
  command.o display.o \
  insert.o

edit : $(objects)
  cc -o edit $(objects)
```

Understanding the 'clean' rule

- By default, make runs down the list of targets and executes them if they're not up to date
- The 'clean' rule doesn't have dependencies and never exists
- If everything is up to date, clean can be run automatically (not what you want!)
- Instead, insert a .PHONY target ahead of it!

The clean rule in action

- clean can now only be run if you type 'make clean'
- The preceding - prevents an echo to the screen when it's executing

```
.PHONY :
clean :
     -rm edit $(objects)
```

Name of your Makefile

- GNU make tries the following:
  - GNUmakefile, makefile, Makefile
  - Usually, you should use the last two
  - Most source files are lower-case, therefore Makefile appears at the top of a sorted list of files

- If you pass -f, you can specify the makefile name