CSE 333

Interlude - make and build tools

Hal Perkins

Department of Computer Science & Engineering

University of Washington

Administrivia

Exercise 5 due before class Monday

- Clean up a buggy program, split into appropriate source files, add a Makefile (more about that today)

Homework 1 due Thursday 11 pm

make

make is a classic program for controlling what gets (re) compiled and how. Many other such programs exist (e.g., ant, maven, “projects” in IDEs, ...)

make has tons of fancy features, but only two basic ideas:

1. Scripts for executing commands
2. Dependencies for avoiding unnecessary work

To avoid “just teaching make features” (boring and narrow), let’s focus more on the concepts...

Building software

Programmers spend a lot of time “building” (creating programs from source code)

Programs they write

Programs other people write

Programmers automate repetitive tasks. Trivial example:

```
gcc -Wall -g -std=gnu99 -o widget foo.c bar.c baz.c
```

If you:

Retype this every time: “shame, shame”

Use up-arrow or history: “shame” (retype after logout)

Have an alias or bash script: “good-thinkin”

Have a Makefile: you’re ahead of us

“Real” build process

On larger projects, you can’t or don’t want to have one big (set of) command(s) that redoes everything every time you change anything

1. If gcc didn’t combine steps behind your back, you’d need to preprocess and compile each file, then run the linker
2. If another program (e.g., sed) created some C files, you would need an “earlier” step
3. If you have other outputs for the same source files (e.g., javadoc), it’s unpleasant to type the source file names multiple times
4. If you want to distribute source code to be built by other users, you don’t want to explain the build logic to them
5. If you have 10^5 to 10^7 lines of source code, you don’t want to recompile them all every time you change something

A simple script handles 1–4 (use a variable for filenames for 3), but 5 is trickier

Recompilation management

The "theory" behind avoiding unnecessary compilation is a "dependency dag" (*d*irected, *a*cyclic *g*raph):

To create a target *t*, you need sources s_1,s_2, …,s_n and a command *c* (that directly or indirectly uses the sources)

If *t* is newer than every source (file-modification times), assume there is no reason to rebuild it

Recursive building: If some source s_i is itself a target for some other sources, see if it needs to be rebuilt…

Cycles "make no sense"

Theory applied to C

Compiling a .c creates a .o – the .o depends on the .c and all included files (.h files, recursively/transitively)

An archive (library, .a) depends on included .o files

Creating an executable ("linking") depends on .o files and archives (-L. -lfoo to get libfoo.a from current dir)

So if one .c file changes, just need to recreate one .o file, maybe a library, and relink

If a header file changes, may need to rebuild more

And there are many more possibilities

make basics

A makefile contains a bunch of triples

```
target: sources
        command
```

Example:

```
foo.o:  foo.c foo.h bar.h
TAB     gcc -Wall -o foo.o -c foo.c
```

Syntax gotchas:

The colon after the target is required

Command lines must start with a ***TAB NOT SPACES***

You can actually have multiple commands (executed in order); if one command spans lines you must end the previous line with \

Which shell-language interprets the commands? (Typically bash; to be sure, set the SHELL variable in your makefile.)

Using make

At the prompt:

```
prompt% make -f nameOfMakefile aTarget
```

Defaults:

If no -f specified, use a file named Makefile

If not target specified, use the first one in the file

Open source usage: You can download a tarball, extract it, type make (four characters) and everything should work

Actually, there's typically a "configure" step too, for finding things like "where is the compiler" that generates the Makefile (but we won't get into that)

- The mantra: ./configure; make; make install

Precise review

A Makefile has a bunch of these:

```
target: source_1 ... source_n
        shell_command
```

Running make target does this:

For each source, if it is a target in the Makefile, process it recursively

Then:

- If some source does not exist, error
- If some source is newer than the target (or target does not exist), run shell_command (presumably updates target, but that is up to you; shell_command can do anything)

make variables

You can define variables in a Makefile. Example:

```
CC = gcc
CFLAGS = -Wall -std=gnu99
foo.o: foo.c foo.h bar.h
        $(CC) $(CFLAGS) -c foo.c -o foo.o
```

Why?

- Easy to change things
- Can change on make command line (CFLAGS=g)

More variables

It's also common to use variables to hold list of filenames:

```
OBJFILES = foo.o bar.o baz.o
widget: $(OBJFILES)
        gcc -o widget $(OBJFILES)
clean:
        rm $(OBJFILES) widget
```

clean is a convention: remove any generated files, to "start over" and have just the source

It's "funny" because the target doesn't exist and there are no sources, but that's okay:

- If target doesn't exist, it must be "remade" so run the commands
- These "phony" targets have several uses, another is an "all" target....

“all” example

```
all: prog B.class someLib.a    # notice no commands this time
prog:   foo.o bar.o main.o
        gcc -o prog foo.o bar.o main.o
B.class: B.java
        javac B.java
someLib.a: foo.o baz.o
        ar r foo.o baz.o
foo.o: foo.c foo.h header1.h header2.h
        gcc -c -Wall foo.c

...(similar targets for bar.o, main.o, baz.o) ...
```

Revenge of the funny characters

Lots - see the documentation

- $@ for target
- $^ for all sources
- $< for left-most source
- …

Examples:

```
widget: foo.o bar.o
        $(CC) $(CFLAGS) -o $@ $^
foo.o: foo.c foo.h bar.h
        $(CC) $(CFLAGS) -c $<
```

And more…

There are a lot of "built-in" rules. E.g., make just "knows" to create foo.o by calling $(CC) $(CFLAGS) on foo.c. (Opinion: may be more confusing than helpful. YMMV)

There are "suffix" rules and "pattern" rules. Example:

```
%.class: %.java
        javac $<    # Note we need $< here
```

Remember you can put any shell command on the command-line, even whole scripts

You can repeat target names to add more dependencies (useful with automatic dependency generation)

Often this stuff is more useful for reading makefiles than writing your own (until some day…)