CSE 220: Systems Programming

Ethan Blanton

Department of Computer Science and Engineering
University at Buffalo

Make

Make is a general-purpose dependency resolver.

That's a fancy term that means:

- You provide it with a list of rules.
- The rules say "If you have A, this is how you get B".
- You say "I want Z".
- It figures out how to get Z from what you have.

The canonical use of Make is building software.

GNU Make

The flavor of Make we will use is GNU Make.

Some CS department machines are FreeBSD.

Be aware that, on those machines, GNU make is `gmake`.

Building Software

Make has several features for building software.

In particular, it can detect changed files.

Suppose that:

- `fileB` is built from `fileA`
- `fileA` has changed

If you ask Make to create `fileB`, it will rebuild it from `fileA`.

On the other hand, if no dependencies have changed, Make does nothing.

Why Use a Build Tool?

There are many reasons to use a build tool.

- Eliminate human error in builds
- Manage builds of thousands of files or more (Firefox, Chrome, the Linux kernel, …)
- Automate repetitive tasks
- Ensure that best practices are followed
- …

Other Build Tools

There are many tools that tackle this same job.

Some examples:

- Make
- ant
- Gradle
- Ninja-build
- Meson
- …

Clearly this is an important problem!

Setting Variables

Make can set variables that can be used later.

Make variables are actually rather complicated.

The simplest syntax is:

```
VARNAME := value
```

This sets the variable `VARNAME` to the value `value`.

Using Variables

Variables can be dereferenced after they are set.

Dereferences use the syntax `$(VARNAME)`

The value of the variable will be inserted where it is dereferenced.

It is helpful to think of most Make constructs as strings.

If you need an actual `$`, use `$$`.

If you forget the `()`, make will use only the first letter of your variable name

Make Rules

Make rules are what it uses to build dependency chains.

A rule expresses:

- The item to be created
- What it requires
- How to build it from what it requires

Rules are evaluated transitively:
A requires B which requires C;
I want A, so build C then B then A.

Syntax

Make is very fussy about syntax.

In particular, it ascribes meaning to the tab character.

A make rule is:

```
target: dependencies
        recipe
```

The blank space before `recipe` must be a tab.

Rule Syntax

```
target: dependencies
        recipe
```

This rule says “target is created from dependencies, and recipe is how you create it.”

- The target is typically a filename
- The dependencies are typically files
- The recipe is a Unix shell script

Predefined Variables

Make has many predefined variables.

These variables are created for you.

Examples:

- `$@` is the target of this rule
- `$<` is the first dependency of this rule
- `$^` is all dependencies of this rule

…their names are mostly terrible.

Example Rule

Let's see an example, from PA0:

```
CC := gcc

calc: calc.o
	$(CC) -o calc $^
```

This says:

- `calc` is built from `calc.o`
- The command to build `calc` is `gcc -o calc calc.o`

Note the `$^` to express "all of this rule's dependencies."
(Which here is only `calc.o`.)

Pattern Rules

Certain rules express patterns which match many files.

For example:

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

This says:

- Any file ending in `.o` can be created from a file of the same name ending in `.c`
- The command to build that file is `$(CC)` with some arguments

The stem of a rule (which matches the `%`) is in `$*`

Special Rules

Some rules are special:

- The first rule in the file will run by default
- The rule `.PHONY:` declares that its dependencies do not produce any output
- The rule `.SUFFIXES:` defines file type suffixes known to the build rules
- Older style pattern rules are of the form `.X.Y` where `.X` and `.Y` are filename suffixes
- Others, see the Make manual

You probably only need to worry about the default rule (and maybe `.PHONY:`).

Make for Automation

You can use make to automate any task conveniently expressed as dependencies.

Running tests is a great example!

```
TESTS := test_onething test_anotherthing

test_%: tests/%.o
        $(CC) -o $@ $< $(CFLAGS)

test: $(TESTS)
        ./test_onething
        ./test_anotherthing
```

Automation Pattern Rule

Let’s examine that in parts. First, the pattern rule:

```
test_%: tests/%.o
        $(CC) -o $@ $< $(CFLAGS)
```

- Build any file starting with `test_something` from a file `tests/something.o`
- Use `$(CC)` to build it
- Presumably there’s also a rule to build a `.o` file from C source!

Automation Rule

The rule for `test` is a phony rule.

It doesn't actually build anything.

```
test: $(TESTS)
        ./test_onething
        ./test_anotherthing
```

We know that `test_onething` and `test_anotherthing` exist before this rule runs.

If either command fails, `make` stops with an error.

Cleaning Up

Another common automation is cleaning up:

```
clean:
	rm -f *.o test_* submission.tar *~
```

If you keep this rule up-to-date, `make clean` will clean up your projects.

Summary

- Make is a dependency solver
- Software construction can be expressed as dependencies!
- Make rules run shell scripts
- You should learn make

References I

Optional Readings

[1] The Free Software Foundation. *The GNU Make Manual*. `info make`.

License