CSE 220: Systems Programming

Lab 02: Introduction to Make

Introduction

This exercise will help you become familiar with Make. Make is a tool for managing the build of complex software projects, although it can be used for many other things as well. At its heart, it is a *dependency solver* that creates a *dependency graph* with software sources at the "leaves" and processed or built components at the nodes. Make then uses this information to determine whether the dependencies for a node in the graph have been met, and if so, executes user-defined instructions to create that node.

Make uses filesystem timestamps to determine that portions of the software are "stale" with respect to their sources, and then executes user-supplied commands to bring them up-to-date. In this manner a very large application can be re-compiled after minor changes very quickly, by re-compiling only those portions of the application that are affected by the changes.

As you will not be building particularly large applications in this course, the primary benefit that you will receive from using Make is automation. It should also reduce errors in builds by ensuring repeatability of the build process.

We will be using GNU Make, although most of the syntax and features that you will learn are the same as other popular Make implementations, such as the original AT&T Make and BSD Make.

1 Getting Started

Follow the GitHub Classroom link that has been provided to you in Piazza to create the starter repository for this assignment. Check it out following the same steps you used for the first lab of the semester. It will include a README, Makefile, and two C source files. You should not modify the C sources for this assignment, but you may wish to inspect them.

1.1 A Brief Introduction to Make

Your TAs will provide you with a more thorough introduction to Make. However, you may find this section useful for reference.

Make is invoked via the `make` command on most UNIX systems. It will read a file called `Makefile` from the current directory (unless otherwise specified), and then execute the first *rule* it finds in that file. Make rules are described below, and will be covered in more detail by your TAs. You can also provide a specific rule to be executed on the command line, by simply typing it after `make` on the command line; for example, `make hello` will instruct Make to consult the Makefile for a rule that will build the file `hello`, and then execute it.

Note that the order in which rules appear in the Makefile is not important *except for the first rule*, which is the rule that will be executed if no rule is specified on the command line. There is an idiomatic broad ordering of rules that many or most developers will follow, however, with the "biggest" and most important rules toward the top of the file, followed by more specific rules, followed by "phony" rules, or rules that don't actually build anything. (There is an example phony rule named `clean` in the handout code that removes the built files. There are additional subtleties to phony rules, and this one is incompletely specified for simplicity, but it will work.)

For this lab you will need to use Make *variables*, create dependency *rules*, and assign *recipes* to those rules. The following Makefile demonstrates these things, and will be described below:

```
VARIABLE := value

var:
        echo VARIABLE is $(VARIABLE)

chain: var
        echo The var rule should have run before this.
```

The first line in this file sets the variable named `VARIABLE` to the value `value`. Note the `:=` syntax used for this assignment. Make will also accept `=`, although the meaning is slightly different. Once this variable is created, it can be referenced with `$(VARIABLE)`. You may also see `${VARIABLE}` used (with curly brackets instead of parenthesis), which has the same meaning.

The next pair of lines defines a *rule* for the *target* `var`, with a *recipe* to create it. If `make` is invoked with no arguments, this rule (being the first in the file) is the rule that will be run by default.

The syntax for a rule is as follows:

```
TARGET: DEPENDENCIES
        RECIPE
```

The syntax for Makefiles is picky; in particular, the whitespace before the recipe **must be** an ASCII Tab character. If you use spaces, Make will (if you are lucky) emit an error like one of these:

```
example-Makefile:4: *** missing separator (did you mean TAB instead of 8 spaces?).
   Stop.

example-Makefile:4: *** missing separator.  Stop.
```

It may, however, emit an even more cryptic message.

This example recipe declares that building the target named `TARGET` requires the dependencies `DEPENDENCIES`, and that producing `TARGET` from `DEPENDENCIES` is accomplished by running the UNIX commands described by `RECIPE`. The dependencies specified in `DEPENDENCIES` may be either other Make targets or filenames, separated by whitespace. Make targets with no associated file will always be considered out-of-date and run *once* each time Make is invoked. If they are filenames, `RECIPE` will be run only if the file `TARGET` is older than each of the files described in `DEPENDENCIES`. In the above example Makefile, note that neither `var` nor `chain` creates a file; therefore, both of these rules will be run any time they are a dependency of any requested target.

Each *line* of `RECIPE` is run as a UNIX shell script. Any UNIX command or combination of commands may be included. If any command in a recipe fails, the recipe will fail and Make will stop with an error. This will happen if, for example, your recipe is compiling C source code and you have an error in your sources. Multi-line UNIX scripts must be written as a single logical line by using semicolons to separate commands and escaping actual newlines with a backslash.

The example Makefile rule for `chain` therefore declares that the rule `var` must be run before `chain`, and that the target described by `chain` is satisfied by echoing a sentence to the terminal. Try creating a `Makefile` like the example above and then running `make`, `make var`, and `make chain` to see how this works.

1.2 The Make Manual

The manual for GNU Make is available online, but *does not use the `man` command that we have prevously seen*. Instead, it uses GNU Info, which works somewhat differently. You can consult it by running `info make` at the prompt. You can press `H` while in the Info viewer for a summary of usage, or see `info info` for more detail. Some useful keys to know are `/`, which initiates a search, `u`, which moves to "up" to the next higher level of the document structure, spacebar to page down, and `q` to quit. The arrow keys will work as expected, and the enter key will jump to a hyperlink under the cursor.

Emacs users can press `C-h i` or `M-x info` to open an info viewer.

2 printf

You will need the `printf()` function for the final requirement of this lab. You can find the `printf()` documentation in `man 3 printf`, and it is discussed in The C Programming Language, but here is a brief overview.

The `printf()` function takes one or more arguments. The first argument is a C string to be printed, the *format string*, optionally containing *format specifiers*. Format specifiers are a percent sign (`%`) followed by characters indicating what is being formatted. For this project, you will need the specifier `%s`. This specifier indicates that the next argument in the `printf()` argument list is a C string, which should be inserted into the format string at the position of the format specifier `%s`. For example:

```
printf("My favorite editor is %s\n", "Emacs");
```

This line would print "My favorite editor is Emacs", followed by a newline.

3 Requirements

You must create a Makefile to compile the sources in the handout repository into an executable named `hello`, which is one of the more complicated "Hello, World" programs ever devised. The handout repository includes a starter `Makefile`, but it is missing some variable definitions and rules. You must:

- Define the variable `CFLAGS`.

  It should hold the value `-g -Wall -Werror`. This variable is traditionally used to hold options passed to the C compiler.

- Fix the rule for the target `main.o`.

  This rule must pass the extra option `-DMAIN` to the C compiler.

- Create a rule to compile `dohello.o` from `dohello.c` using `gcc`.

  This rule must include `$(CFLAGS)` on the command line, and must define the preprocessor symbol `DOHELLO` to have the value `do_hello` using the option `-DDOHELLO=do_hello` passed to the C compiler.

- Create a C source file named `name.c`, containing a program that prints "Hello, <name>", followed by a newline, where <name> is provided as its first argument. Write a Makefile rule to compile this file into a binary called `name`. For example:

  ```
  [elb@westruun]~$ ./name Ethan
  Hello, Ethan
  [elb@westruun~]~$
  ```

  Look at the `hello` and `dohello.o` rules for this.

Once these requirements are complete, typing `make` with no options must cause `hello` to be built, and running `hello` must produce the string "Hello, World!" at the terminal. *You should not modify any C source to accomplish this!*

4 The C Preprocessor

We have not seen the C preprocessor in any detail before this lab; however, two of your rules require that you use it! The C preprocessor does exactly what it says on the label: it pre-processes your C sources *before the C compiler sees them*. The details of how it works will be covered later, but for now, here's what you need to know:

If you pass `-D`*SYMBOL* to the preprocessor, then in your C code, the preprocessor symbol called *SYMBOL* will be defined. What exactly this means is out of scope for this lab, but note how it is used in `main.c`. This can be used to *conditionally remove code*, which you may find helpful for, *e.g.*, including debugging statements only when you are testing, and not submitting them to the Autograder (where they will probably lose you points!).

You can also *set a symbol to a value* with the argument `-D`*SYMBOL*=*value* to the compiler. This causes every instance of the text *SYMBOL* in your source code to be replaced with the text *value*. You can use this to, for example, rename functions or variables (as you are asked to do in this lab). Once again, this can be very helpful for debugging, among other things.

The preprocessor is very powerful, but somewhat difficult to use correctly. We will cover it in more detail later, but the techniques you see in this lab will probably be helpful for you in testing and developing your projects in this course.

5 Submission and Grading

Run `make submission` to create the file `submission.tar`, and turn this file in to Autograder.