1 Lab 1

Date: Sep 2, 2021

This document first describes the aims of this lab. It then provides necessary background. It then describes the exercises which need to be performed.

In the listings which follow, comments are any text extending from a `#` character to end-of-line.

1.1 Aims

The aim of this lab is to introduce you to the use of `Makefile`'s under Unix. After completing this lab, you should be familiar with the following topics:

- The basic operation of `make` for building C programs.
- Common problems when using `make`.
- The use of `make` *variables*.
- The presence of *implicit commands* in `make`.
- Auto-generation of dependencies.

1.2 Background

A typical large program consists of multiple sub-systems and libraries. Each sub-system or library will contain multiple source files. Building the program entails compiling all sub-systems and libraries with the correct options and assembling them together. This can often be quite complex and time consuming. If any source file changes, it should be possible to rebuild the program while redoing as little work as possible. The `make` program allows the automation of such tasks. The operation of `make` is controlled by a file typically named `Makefile` in the directory where `make` is invoked.

Note that `make` is an example of a *build* tool. The `make` used in this lab is typical of that found in Unix systems. Microsoft's `nmake` is a similar program. Build tools like Java's `ant`, Ruby's `rake` and Python's `scons` have similar functionality.

1.2.1 Principles

A `Makefile` basically consists of a set of *rules*. Each rule describes the *prerequisite* files for building a *target* file and the *recipe* which needs to be carried out if any of the prerequisite files are newer than the target file.

```
target: prequisite ...
recipe
```

The *recipe* can consist of multiple Unix shell commands (this can include compilation commands), which must be run to make the *target* from the *prerequisite* files.

The *target* for one rule can be a *prequisite* for another rule. Hence the first rule will not be run until the prerequisite is made up-to-date by its rule. The `make` program (at least the GNU version) tracks these dependencies across any number of levels and executes all necessary recipes to bring the targets of all relevant rules up-to-date.

To build a particular target, `make` can be invoked with that `target` as its command-line argument. If invoked with no targets, it will attempt to build the target for the **first rule** in the `Makefile`.

Consider building an executable `hello` from 3 files: a `hello` module consisting of a specification header file `hello.h` and an implementation file `hello.c` and a `main.c` which includes the `hello.h` header file. This can be achieved using the following `Makefile`:

```
hello: main.o hello.o
#link main.o and hello.o to executable hello
gcc main.o hello.o -o hello

hello.o: hello.c hello.h
#compile hello.c to object file hello.o
gcc -std=c18 -g -Wall -c hello.c

main.o: main.c hello.h
#compile main.c to object file main.o
gcc -std=c18 -g -Wall -c main.c

clean:
rm -f *~ *.o hello
```

Note the last target `clean`. It does not have any prerequisites and hence will run its recipe whenever it is invoked (typically invoked explicitly as `make clean`). It's recipe runs the shell `rm` command which will remove all emacs backup files specified by the wildcard pattern `*~`, all object files specified by `*.o` as well as the built `hello` executable. The name `clean` is conventionally used for such targets which clean-up files built by `make` as well as any garbage files.

1.2.2 Variables in make

Note that in the previous example, both the `hello.o` and `main.o` use the compiler options `-g` to turn on debugging and `-Wall` to turn on reasonable warnings.

This is a violation of the *DRY principle*, since the same options were specified multiple times. Such violations can be avoided by the use of `make` variables.

A `make` variable is defined on a line which consists of an identifier *VAR* followed by an `=` character which may be preceeded/followed by linear whitespace (i.e. whitespace within the same line) followed by a definition. If the definition is spread across multiple lines, then the last character must be a \ on all except the last line of the definition.

The use of a variable *VAR* within a rule is indicated by `$(`*VAR*`)` and is replaced by its definition. If a `$` is to occur within a rule, then it must be quoted by repeating it.

Additionally, within each rule, the special `make` variable `$@` stands for the target and the special `make` variable `$<` stands for the first prerequisite and `$^` stands for all the prerequisites with spaces between them.

With the use of variables, the previous `Makefile` can become:

```
TARGET = hello
OBJS = \
  main.o \
  hello.o

CC = gcc
CFLAGS = -std=c18 -g -Wall
LDFLAGS =

$(TARGET): $(OBJS)
#link $(OBJS) to executable hello
$(CC) $(OBJS) $(LDFLAGS) -o $@

hello.o: hello.c hello.h
#compile hello.c to object file hello.o
$(CC) $(CFLAGS) -c $<

main.o: main.c hello.h
#compile main.c to object file main.o
$(CC) $(CFLAGS) -c $<

clean:
rm -f *~ *.o $(TARGET)
```

1.2.3 Implicit Rules

Note that in the previous example, the recipes for building both `hello.o` and `main.o` are absolutely identical. In fact, a little thought will reveal that this recipe can always be used for building a `.o` file from a `.c` file. So `make` contains a set of **implicit rules** similar to this. If there is no recipe given for building a prerequisite file, then `make` uses its implicit rules.

With the use of implicit rules, the `Makefile` can be simplified to:

```
TARGET = hello
OBJS = \
  main.o \
  hello.o

CC = gcc
CFLAGS = -std=c18 -g -Wall
LDFLAGS =

$(TARGET): $(OBJS)
#link $(OBJS) to executable hello
$(CC) $(OBJS) $(LDFLAGS) -o $@

clean:
rm -f *~ *.o $(TARGET)

hello.o: hello.c hello.h
main.o: main.c hello.h
```

Note that the rules specifying the dependencies for the `.o` file have been moved to the bottom of the `Makefile` as they are purely declarative (using the implicit rules). If it was not necessary to record the fact that both `hello.o` and `main.o` also depend on `hello.h`, then the last two lines too could be removed as `make` is capable of concluding that `hello.o` depends on `hello.c` and `main.o` depends on `main.c`.

1.2.4 Gotcha's

The `make` program evolved in the 1970's when many programming languages were line-oriented. Hence it has a line-oriented syntax with some very peculiar syntax rules which can result in extremely painful gotcha's for the unwary.

- The lines containing recipes **MUST BEGIN WITH A TAB CHARACTER**. Since most text editors do not distinguish between the display of tab and space characters, this is a very common problem (the `emacs` editor will warn you about *suspicious lines*).
- When `make` variable definitions or recipe commands extend over multiple lines, all but the last line must terminate with a \ character. There **CANNOT BE ANY SPACES** after the \ character.

- Each command in a recipe is run in a separate shell. Hence a command cannot affect the state of the shell for a subsequent command.

  For example, the following rule attempts to delete all `.o` files in directory `dir`:

  ```
        clean-dir:
         cd dir
  rm -f *.o
  ```

  This will not work. The first command runs in a separate shell and changes its current directory to `dir`, but then that shell terminates. The second command runs in a new shell and will delete all `*.o` in the current directory, not the `dir` directory.

  The fix for this is to run both commands within a single shell as follows:

  ```
        clean-dir:
         cd dir; \
  rm -f *.o
  ```

  By using the trailing \ after the first command, only a single shell is used to run the sequential shell command `cd dir; rm -f *.o` which has the desired effect.

- The `clean` target does not actually build a file called `clean`. In fact, it is a phony target and we want it to run whenever we type `make clean` irrespective of whether or not a file called `clean` actually exists. To do so, we can force execution of the cleanup command by specifying the `clean` target as `.PHONY`:

  ```
      .PHONY: clean
      clean:
  rm -f *~ *.o $(TARGET)
  ```

1.3 Exercises

Follow the *provided directions* for starting up this lab in a new git `lab1` branch and a new `submit/lab1` directory. You should have copied over the contents of `~/cs220/labs/lab1/exercises` over to your directory.

When the exercises mention a new Unix command you are unfamiliar with, it is a good idea to do a `man` or google lookup on that command to get an idea of its capabilities.

1.3.1 Exercise 1: Hello World

Change over to the 1-hello directory.

```
$ cd ~/i220?/submit/lab1/exercises/1-hello
$ ls -l
```

You should see that the directory contains a single `hello.c` file.

Perform the following steps To examine the different stages of compilation:

1. To quit the compilation after running the C preprocessor use the `-E` option:

   ```
   $ gcc -E hello.c -o hello.i
   ```

   Examine the `hello.i` file using a text editor. You will see the preprocessed output. Note the huge number of declarations sucked in by the `#include <stdio.h>`. In particular, you should see a **declaration** (not definition) for `printf()`.

2. To quit the compilation after running the compiler proper (`cc1`), use the `-S` option to produce a `.s` assembly language file.

   ```
   $ gcc -S hello.c
   ```

   Examine the generated `hello.s` file using a text editor. Since the format string provided to `printf()` is a simple string without any `%` escapes, the current version of gcc replaces the call to `printf()` with a call to puts(). Note that since `puts()` outputs a string followed by an extra newline, the compiler generates the `"hello world"` string without the `\n` which was present in the `hello.c` source.

3. To quit the compilation after assembling the `.s` file to a `.o` file, using the `-c` option:

   ```
   $ gcc -c hello.c
   ```

   The generated `hello.o` file is a binary file. You can look at it using your text editor. If your text editor does not choke on a binary file, you will see that the string `puts` is present in the file.

   Another way to look at the file is by using the `objdump` program.

   ```
   $ objdump --syms hello.o
   ```

   The `--syms` option will dump out the symbols in the file. You should see the `puts` symbol with attribute `*UND*` for undefined. So the `hello.o` file records the fact that it needs a definition for `puts`.

4. The definition for `puts` will be made available in the executable which can be produced using:

   ```
   $ gcc -static hello.c -o hello
   ```

   The `-static` forces the compiler to include definitions for all outstanding symbols within the output `hello`. This is referred to as **static linking**, as opposed to **dynamic linking** where the definitions are loaded only at runtime.

Run the executable:

```
$ ./hello
```

Examine the executable using `objdump -d hello >hello.dump`. This will disassemble (the opposite of *assemble*) `hello` into `hello.dump`. Use a text editor on `hello.dump` to search for `<main>`. You will see the dissembled code for `main`. Within that code you will see a call to a function `_IO_puts`; if you now search for `<_IO_puts>` you will see a definition for `puts()`.

5. In this step, we will build the program using the `make` program. First, get rid of the existing `hello` executable:

   ```
   $ rm hello
   ```

   Now type `make`. You should get an error message. However, now try `make hello`. You should see that `make` automatically builds a `hello` executable. Type `ls -l` to see the created file, use the command `file hello` to confirm that it is an executable, and type `./hello` to execute it. You should see the usual `hello world` message.

   How did `make` know how to build `hello` even though there is no `Makefile` in the directory? The answer is by using implicit rules.

   To see the list of `make`'s builtin implicit rules, type `make -p | less` The `less` command allows you to page back and forth through the output using the spacebar and the `b` key respectively). You will see that the set of rules is quite extensive. To see lines which are relevant to c programs, type `make -p | grep '\.c'` (the `grep` program filters out lines which do not match the pattern given by its argument). You will see lines relevant to compiling c programs but you will also see lines related to C++ and `YACC` programs (the latter is a parser-generator program).

1.3.2 Exercise 2: Makefile with Syntax Error

Change over to the 2-err directory and type `make`. You should get an error. Fix the error and retry. A greeting should be printed on your terminal.

Hint: *see gotchas*.

The `Makefile` contains a second target `count-bin` which will print out the number of files in the `/usr/bin` directory.

The provided recipe for count-bin is:

```
cd /usr/bin ;
ls | wc -l
```

which changes directory over to `/usr/bin` and uses `wc -l` to count the number of lines in the output of the `ls` command when run in that directory.

If you run the recipe manually by typing it on a terminal you will get a count of a few 1000 printed.

However, if you return to the 2-err directory and type `make count-bin`, you will get a much smaller count Fix the `Makefile` so that `make count-bin` prints a count consistent with what you got by running the recipe manually.

Hint: *see gotchas*.

1.3.3 Exercise 3: Multi-File Compilation

Change over to the 3-multifile directory and take a look at the files there. This directory contains a program to solve quadratic equations.

Note that this file contains a `README` file. It is always a very good idea to have a `README` file in the top-level directory of a project giving a rough idea of what that project is.

Perform the following steps:

1. This directory contains a `Makefile`, but unfortunately it contains an error.

   Simply type `make` in an attempt to build the top-level target. If you look at the resulting `make` trace you should see that `make` successfully built `quadr.o` and `main.o`, but got errors regarding a missing `sqrt()` function when attempting to link the object files to produce the executable.

   The linker needs to be told which libraries to search when linking. By default it always searches the standard C library (corresponding to the gcc linker option `-lc`) when linking (which is why you do not get link errors for functions like `printf()` which are defined there). However, the default `-lc` does not include the math library which is where the `sqrt()` function is defined. Hence to fix the error simply change the definition of the `LDFLAGS` `make` variable from empty to `-lm` to include the math library.

   Retype `make`. You will still get an error. The problem is that in the line `$(CC) $(LDFLAGS) quadr.o main.o -o $@`, the arguments are processed in the order they are mentioned; hence the math library specified by `LDFLAGS` is searched before the the linker realizes that `quadr.o` has an outstanding reference to the `sqrt()` function. The fix is to search the libraries after processing the object files; you can make that happen by moving the $(LDFLAGS) after the `quadr.o main.o` object file names.

   If you retry `make` after making the above change, you should get a successful build of the `quadr` executable. Test it by providing input lines with each line containing whitespace-separated coefficient triples (remember to terminate your input with a `control-D` character to indicate end-of-file on the terminal), or by redirecting standard input from `test.data`.

```
$ ./quadr < test.data
```

2. If you look at `main.c` and `quadr.c`, you will see that they both depend on `quadr.h`. Hence if `make` is doing its job, it should recompile all files if `quadr.h` changes.

   Try this. Simply `touch quadr.h`. Using `ls -l` you should see that it has a modification time newer than `quadr.o` and `main.o`. Now type `make`, you would expect it to recompile all the files. Instead it will simply output a message saying `quadr` is up-to-date. What went wrong?

   The problem is that though `make` has implicit knowledge that `main.o` and `quadr.o` depend on `main.c` and `quadr.c`, it does not know that `main.o` and `quadr.o` depend on `quadr.h`. One solution would be to provide the dependencies explicitly in the `Makefile` as in the example `Makefile`'s listed in the **Background** section. However, as the project evolves, it is very likely that these dependencies will no longer be correct. Fortunately, there is a better way.

   The C pre-processor used by gcc has a special switch `-MM` which will output all the non-system dependencies of its command-line arguments. Simply add the following lines to your `Makefile` (make sure that the recipe line starts with a tab character):

```
    depend:
$(CC) -MM $(CPPFLAGS) *.c
```

   Now type `make depend`. The dependencies for all your C files should be printed on your terminal. Cut and paste these dependencies at the end of your `Makefile`. Precede the dependencies by a suitable comment (starting with a # character) like

```
#automatically generated dependencies produced by make depend
```

   Now typing `make` should rebuild your entire project.

   [This procedure is obviously extremely clumsy. A subsequent exercise provide better alternatives.]

3. When you complete a project, you may want to create a single file archive which contains all the source files used for building the project. This is known as a *source distribution*. In this step, we will modify our `Makefile` to create a source distribution.

   First we need to list all the source files in the project. This can be done by defining a `make` variable `SRC_FILES` which contains a space-separated list of all the source files. Define this variable after the definition of the `PROJECT` variable in the `Makefile`.

It is a bad idea to use shell glob-patterns like `*.c` to match all `.c` files as advanced projects may generate `.c` and `.h` files automatically and those should not be included; instead list all files explicitly. You should include all *human-generated* files in the project, including the `README`, test files like `test.data` as well as the `Makefile`. It is probably a good idea to list one file per physical line and combine those physical lines into a single logical line by terminating all but the last one by a \.

Then add the following rule (before the automatically generated dependencies section which is conventionally always kept at the end):

```
    dist: $(PROJECT).tar.gz

    $(PROJECT).tar.gz: $(SRC_FILES)
      tar -cf $(PROJECT).tar $(SRC_FILES)
gzip -f $(PROJECT).tar
```

The first command in the recipe creates a `tar` archive of all the `SRC_FILES`. The second command compresses the archive.

Create a source distribution by typing `make dist`.

Now test your distribution. Create a temporary directory and `cd` to it. Then type `tar -xzf` *DISTR_PATH* where *DISTR_PATH* is the path from your temporary directory to the created `quadr.tar.gz` distribution file. You should then be able to `make` the project afresh in this temporary directory by simply typing `make`.

When you have completed this exercise, `cat Makefile` to your terminal so as to include it in the script you will show the TA.

1.3.4 Exercise 4: Makefile from Scratch

In this exercise, you will create a `Makefile` from scratch. Change over to the 4-from-scratch directory. There you will find a set of files which when built into an executable allows addition, lookup and deletion of key/value pairs from a table of key/value pairs, for alphanumeric keys and integer values. Look at the README for an example log.

Use the `Makefile` from the previous exercise as a template to create a `Makefile` for this directory. The top-level target should create an executable named `key-value`. You should use automatic dependency generation to record all dependencies. You should also provide a `dist` target which will build a source distribution.

When you have the `Makefile` working, `cat` it to your terminal so as to include it in the script you will show the TA.

1.3.5 Exercise 5: Auto-Dependencies

Change over to the `5-auto-dependencies` directory. The files provided are identical to those from the 3-multifile exercise except for the `Makefile`. Instead of explicitly listing the dependencies, the `Makefile` is set up to automatically generate them with help from the compiler.

Compile and run. Everything should work as before. Notice the creation of a `.deps` directory which contains dependency files for each `.c` file.

1.4 Winding Up

Wind up your lab by using the *provided directions* to terminate your log in a `lab1.LOG` file and merging your `lab1` branch into the `master` branch. Once you have the lab on your `master` branch, commit and push your changes to github. Be sure to include your `lab1.LOG` file as well as all the exercise directories.

1.5 References

GNU Make Manual.

Advanced Auto-Dependency Generation.

Robert Mecklenburg, *Managing Projects with GNU Make*, O'Reilly, 2004.