Operating Systems, Fall 2017
Instructions for Databar Exercise 2: Makefiles and debugging C programs

September 8, 2017

1 Introduction

In this second databar exercise you will work with makefiles but also invoke compilers and other tools directly. After that you will continue to develop your tool and C skills by debugging a few programs with gdb.

You do not report on this exercise anywhere. However, we will use the command line in later exercises and you will write reports about those exercises.

Read through this document in entirely before starting working on the exercises!

2 Learning objectives

During this assignment you will be working towards the following learning objectives:

- You can explain how the compiler, assembler and linker are used to create executables.
- You can apply standard programming methodologies and tools such as test-driven development, build systems, debuggers.
- You can explain the role of each component of a compilation toolchain used in system programming and how the components interact.

3 Reference material

During the work on the assignment you will program in C. Before starting working read the following document up until and including C pointers:

- ANSI C for Programmers on UNIX Systems by T. Love, `http://www-h.eng.cam.ac.uk/help/tpl/languages/C/teaching_C/`

  An introduction to C for programmers already knowing other languages, e.g. java.

You might find the following literature helpful as a reference:

- UNIX Tutorial for Beginners, `http://www.ee.surrey.ac.uk/Teaching/Unix/`

  Simple UNIX tutorial. Follow this if you need further reference to and guidance with UNIX command line tools.

During the work on the assignment you will use several tools. You might find the following manuals helpful and you should browse them:

- GNU Binutils `http://www.gnu.org/software/binutils/`

  This is a large collection of tools for manipulating different executable binary files.

- GCC `http://gcc.gnu.org/`

  GCC is a collection of compilers for many languages including Java, C and C++.

- GNU Make `http://www.gnu.org/software/make/manual/`

  make is a tool that can help you coordinate compilations of larger programs.

- GDB `https://sourceware.org/gdb/current/onlinedocs/gdb/`

  A debugger allowing you to step through the code as you are executing and print out values.

4 Setting up the environment

You use the VirtualBox image you set up last week.

5 Using the command line and creating a simple makefile

Start up the virtual machine you set up and use the command line skills you have acquired to download the exercise source code from CampusNet and extract it to a newly created folder. tar.gz files can be extracted with:

```
tar zxvf <filename>
```

while standing in the folder to which to extract.

First, we will simply use the GNU compiler (gcc) from the command line in the virtual machine to build a “Hello World” example and run it. Then, we’ll place these commands into a basic makefile and run the makefile. Later, we’ll use built-in and user-defined variables.

5.1 Build and run “Hello World” from the command line

To compile app.c, type the following command:

```
$ gcc -g -c app.c -o app.o
```

```
  gcc = GNU C compiler, command
   -g = symbolic debug, compiler option
   -c = see below
app.c = file to compile
   -o = output filename is next, compiler option
app.o = output file, the \target"
```

In the above gcc command, name the target, dependency and command.

```
Target = _______________
Dependency = ________________
Command = __________________
```

In your virtual machine use the man command to look up gcc.

To find the parameters for any standard C functions or Linux commands, you can use the “man”, short for “manual”, command. Let’s try it on gcc:

```
$ man gcc
```

What does the -c option, from the previous step, tell the compiler to do?

To quit the man page, type q.

5.2 Link the object file and produce the final executable

Next, link the object file, app.o, to create the executable app:

```
$ gcc -g app.o -o app
```

Now run the executable:

```
$ ./app
```

You should see “Hello World” displayed in the command window.

Note: For those of you who know Linux well, you can skip this explanation. For the rest . . .

./ before the name of an executable tells Linux to look for the program in the current directory. We use this as it is the proper way to specify the path of the file to be run.

5.3 “Clean” the existing executable and intermediate (.o) files

Type the following to remove the files generated by the gcc commands you executed:

```
$ rm -rf app
$ rm -rf app.o
```

This removal of files mirrors what a “clean” macro or rule might do. We’ll actually add a rule shortly to accomplish this in our makefile.

5.4 Examine “starter” Makefile

The current makefile simply contains comments and placeholders for the code you will write. Using your favorite editor, open the Makefile. For example:

```
$ emacs Makefile
```

NB: You can use whatever editor you like!

5.5 Create rules for app and app.o in your makefile

Remember, a rule is made up of a target, dependency(ies) and command(s). For example:

```
target : dependency
         CMD
```

Also note that the commands are tabbed over, at least one tab. Create the rule for app.o in the area of the makefile with the header comments specifying the intermediate (.o) rule, as shown below. We’ll help you with the rule for app.o, but app is up to you. For app.o, type in the following rule. We will use the absolute path of gcc for now and later turn it into a variable:

```
# ---------------------------------------------------
# ------ intermediate object files rule (.o) -------
# ---------------------------------------------------
app.o : app.c
        gcc -g -c app.c -o app.o
```

In the above:

```
            app.o = target
            app.c = dependency
gcc -g ... = command
```

5.6 Type in the rule for the executable

Next, type in the rule for app above the rule for app.o in the area specified for the executable rule. Make sure you use the -g compiler option in the rule.

5.7 Test your makefile

Close makefile and type the following:

```
$ make
```

After running make, list the current directory. Do you see a new app executable? Run it.

Do you see "Hello World"? If so, your rules work. Next, let's add a few more rules…

5.8 Open makefile in a new Linux process

Stop. Before you open makefile again, try opening it in a different Linux process by typing in the following:

```
$ emacs Makefile &
```

NB: You can use whatever editor you like!

The "&" tells Linux to open the makefile in a separate process. When you edit a file, you can simply save it, then click inside the terminal window and run it without having to re-open the makefile. Handy – and could save you some time.

5.9 Create a "clean" rule in your makefile

Whenever you run make, it will search and note the timestamps of the source files and executables and won't run if everything is up to date. So, it is common to create a "clean" rule that removes the intermediate and executable files prior to the next build.

In the makefile, underneath the comment header for "clean all", add the following .PHONY rule for "clean". These are the same commands you used earlier on the command line:

```
.PHONY : clean
clean :
        rm -rf ___
        rm -rf _______
```

.PHONY tells make to NOT search for a file named "clean" because this is a phony target, i.e., it is not a file that needs to be searched for or created. In a large and complex makefile, this actually saves some compile time. Plus, it is just good practice to use .PHONY when the target is not an actual file. The two files are the final executable and the intermediate object file.

5.10 Create an "all" rule in your makefile

When make runs without any rules specified, i.e., you just type "make" on the command line, it will make, by default, the first rule in the makefile. Therefore, it is common to create an "all" rule that is placed first in the makefile. Our example only has one final target (app), so "all" doesn't make as much sense now.

In the makefile, under the comment header for "make all", add the following .PHONY rule for "all":

```
.PHONY : all
all : app
```

Close makefile and let's run it. . .

5.11 Run make to create the executable app

On the command line, type in the following:

```
$ make
```

make will probably tell you that the files are "up to date" and there is nothing to do. So, you must run "clean" before you build again. Type:

```
$ make clean
```

and then:

```
$ make
```

make runs the first rule in the makefile which is the "all" rule. This should successfully build the app executable.

Note: make assumes the name of the make file is makefile or Makefile. make also looks for the first makefile it finds. So, to be safe, you might want to capitalize Makefile because capital "M" comes before lower-case "m" alphabetically. You can also use a different name for the makefile – e.g. my_makefile.mak. In this case, you need to use the following command to "force" the use of a different make file name:

```
$ make -f my_makefile.mak
```

Run app.

You should see "Hello World" again. Ok, now that we have the simple makefile done, let's turn it up a few notches...

5.12 Review the different ways to run make

As a review, you can run make in several ways:

```
make                makes the first rule in the make file named makefile or Makefile
make <rule>         makes the rule specified with <rule>, e.g. \make clean")
make -f my_makefile forces the use of a make file named my_makefile
```

6 Using built-in and user-defined variables

We will now add some user-defined variables and built-in variables to simplify and help the makefile more readable. You will also have a chance to build a "test' rule to help debug your makefile.

6.1 Add CC as a user-defined variable to your makefile

Right now, our makefile is "hard coded". Over the next few steps, we'll attempt to make it more generic. Variables make your code more readable and maintainable over time. With a large, complex makefile, you will only want to change variables in one spot vs. changing them everywhere in the code.

Add the following variable in the section of your makefile labeled "user-defined vars":

```
CC := gcc
```

CC specifies the path and name of the compiler being used. Whenever you use a variable (like CC) in a rule, you must place it inside `$( )` for make to recognize it – for example, `$(CC)`. After adding this variable, use it in the two rules.

Test your makefile: clean, make and then run the executable.

6.2 Add CFLAGS and LINKER_FLAGS variables to your makefile

Add the following variables in the section of your makefile labeled "user-defined vars":

```
CFLAGS := -g
LINKER_FLAGS :=
```

CFLAGS specifies the compiler options – in this case, -g. LINKER_FLAGS will tell the linker to include libraries during build. Right now we do not use any libraries and so this variable is empty.

Use these new variables in the rules in your makefile and test it.

6.3 Add built-in variables to your rules

Make contains some built in variables for targets `$@`, dependencies `$^` or `$<` and wildcards `%`. Modify the rules to use these built-in variables.

Because we only have one dependency, use the `$<` to indicate the first dependency only. Later on, if we add more dependencies, we might have to change this built-in symbol. `%` is a special type of make substitution for targets and dependencies. The `%.o` rule will not run unless a “filename.o” is a dependency to another rule and, in our case, app.o is a dependency to the app rule – so it works.

Comments can be printed to standard I/O by using the echo command. In the app rule, add a second command line as follows:

```
@echo; echo $@ successfully created; echo
```

The `@echo` command tells make to echo “nothing” and don’t echo the word “echo”. So, effectively, this is a line return just like the echo at the end of the line. Because built-in variables are valid for the entire rule, we can use the `$@` to indicate the target name.

Test makefile and observe the echo commands. Did they work? As usual, you might need to run “make clean” before “make” so that make builds the executable.

6.4 Add “test” rule to help debug your makefile

Near the bottom of makefile, you’ll see a commented area named “basic debug for makefile”. Add the following .PHONY rule beneath the comments:

```
.PHONY : test
test:
        @echo CC = $(CC)
```

This will echo the path and name of the compiler used. Try it. Does it work? You can also add other echo statements for CFLAGS and LINKER_FLAGS. This is a handy method to debug your makefile.

Close your makefile when finished.

7 Debugging C programs.

For this part of the exercise we will a part of a lab from UNSW in Australia. We will be using gdb which is a standard debugger used for operating system development and many kinds of embedded systems.

Several parts of the lab does not apply. For example, none of the discussion about marks, demo or blogs apply to this course. You will also have to treat the lab as if you did it at "home" and download files from the URL in the lab text. Due to legal rules I cannot copy or duplicate the lab instead I am pointing you directly to the original lab text.

Perform exercises "Setting up" up to and including "Debugging the Queue". If you wish you can show your debugged programs to a TA for review.

You find the lab at the following URL: see `https://webcms3.cse.unsw.edu.au/COMP2521/17s2/resources/10979`.

8 License