IC221 Lab: Makefiles and Debugging

Spring, AY2022. 95 points total

**Intro**

Two of the most important skills of the systems programmer are:

- Managing multi-file programming projects with dependencies

- Debugging compiled C code

In the first part of this lab, we practice simplifying the management of multi-file dependencies through the creation of Makefiles. In the second part, we practice finding infromation about C programs while they’re running using the debugger, GDB.

**Submission (submit all three files at once)**

Makefile\_Simple

Makefile\_Multi

worksheet.docx

**Part 1: Makefiles** (45 points)

Simple Compilation

A Makefile is a small program that describes a compilation process. There are three main elements of a Makefile:

*Targets*: This is the goal of a compilation process, such as an executable or object file

*Dependencies*: Files which the target depends on, such as the source files

*Commands*: What should be run to actually compile a file to produce a target.

Once the Makefile is in place, in that directory, you run the make command, which looks for a Makefile and attempts to ‘make’ a specific target according to the recipe.

Let's look at a very simple example. Here's how we would use a makefile to compile a helloworld.c program:

all: helloworld

helloworld: helloworld.c

gcc helloworld.c -o helloworld

clean:

rm -f helloworld

All *targets* are set to the left, and followed by ":". So, the targets in this Makefile are all, helloworld, and clean. *Dependencies* are found to the right of the targets. For example, the all target depends on generating the helloworld target, which in turn, depends on the helloworld.c source file. Finally, needed *commands* are on lines below each target and *must be tabbed in using the Tab key* (very important).

Reading the Makefile, the key thing is to follow the targets through their dependencies to the commands needed to do the execution. For example, when we type make, the all target is executed by default. The all target depends on producing the helloworld target, which depends on helloworld.c. Now, the file helloworld.c is not a target, it's a file, and by listing it as a dependency, we are saying "this target is not met whenever the file changes," like when we edit the source code. Assuming the helloworld.c source had changed, thus the helloworld target is not met, then the command is executed, which (re)compiles helloworld.c to produce the helloworld executable.

The last target, clean, does not have any dependencies. Instead, it just has the shell command to remove the executable. It is good practice to have a clean target in your Makefiles. You will often need to clean up the source by removing extraneous files, and the Makefile is a fast and convenient way to do this.

When we use a Makefile, we can just type make, which will compile all the targets associate with the all target. Or, we can type make <target>, which will just execute the commands to reach the given target. For example, to execute the clean target, we type make clean.

Task 1 (15 points)

In the folder, you will find a program called compileme.c.

* Write a Makefile called Makefile\_Simple that will compile compileme.c by typing make Makefile\_Simple and also will clean up any stray executables by typing make clean.
* Test your makefile by typing make -f Makefile\_Simple, then execute the program. Note the output. Execute the program, then type make -f Makefile\_Simple again and note the output.
* To test your makefile dependencies, add an additional printf statement to the compileme.c source, and type make -f Makefile\_Simple again. If compileme.c recompiles, you've done this right.
* Finally, add some options to the compilation statement, so that you compile compileme.c with the debug flag (-g) and the warning all flag (-Wall).

Multipart Compilation

Let's review the compilation process. When we have source broken across multiple files, we first have to compile those files to object code, an intermediate compilation stage.

gcc -c source.c -o source.o

**Important**: Use the -c flag with gcc when generating object files (.o files). Do **not** use the -c flag when generating final executables.

Next we can compile multiple object files to assemble an executable.

gcc source.o main.o -o executable

If you look at the above compilation command, you can see the *target* and *dependencies*. The *target* is executable. The *dependencies* are the object files (source.o and main.o), which each in turn have their own dependencies -- their associated source code (and possibly header files). Let's translate that into a Makefile.

all: executable

executable: source.o main.o

gcc source.o main.o -o executable

source.o: source.c source.h

gcc -c source.c -o source.o

main.o: main.c

gcc -c main.c -o main.o

Tracing the dependencies and the commands starting with all, we can see that to reach the compilation command for executable, first source.c and main.c must be compiled to object files, source.o and main.o. You will also notice the header file, source.h, is listed as a dependency for source.o, which is common, so that recompilation will occur whenever the header file changes.

Task 2 (30 points)

In the folder you will find four source files and two header files (runme.c, runme\_too.c, library.c, library.h, fun.c and fun.h). Two of the source files have a main() function.

* Write a Makefile called Makefile\_Multi to compile (as the target) the binary executable called runme. You will need to also compile the dependencies, and inspect the source file and the associated headers to determine what that might be. When you test, be sure to specify the filename, as in: make -f Makefile\_Multi
* Add another target to the all target so that it now compiles two executables, runme and runme\_too.
* Include a clean target to remove all object files (those that end in .o) and executables, e.g., runme and runme\_too.

**Part 2: Intro to Debugging** (50 points)

The program string has been compiled with debugging symbols included. For example:

gcc -g -o string string.c

There is a secret string embedded in the program, but its value is only “correct” (understandable) mid-program, and it never is sent to standard out. The string is obfuscated at the beginning and end of the program, but not in the middle. By debugging and examining the program, you'll be able to figure out the hidden string value.

- Open or print for reference a copy of the gdb reference card.

- Set up gdb to use Intel syntax:

echo "set disassembly-flavor intel" >> ~/.gdbinit

- Launch gdb, with the executable as an argument.

gdb -q string

- Set a breakpoint in the main() function:

(gdb) b main

- View the disassembly of the main() function:

(gdb) disass main

Although you won't understand all the x86 functionality, you may be able to identify some things that are going on:

- A loop is often identified with an increment of a variable value by one (0x1) using the add instruction. There are two loops in the program. Identify an example of this kind of increment-by-one instruction in each loop, and copy the examples into your worksheet. (Hint: hit <return> as needed to scroll down).

- Local variables are stored on the stack below the base pointer, contained in the register 'rbp'. Since the stack grows down, local variables are below the base pointer in memory, such as [rbp-0x50]. Give one example of an assembly instruction from main() that assigns a local variable its value using the mov instruction, which copies a value to memory. Write your answer in your worksheet.

- Run the program, which will proceed until the breakpoint at main():

(gdb) r

- Get info about the local variables:

(gdb) info locals

- What are the local variable names? Copy the variable names and their current values into your worksheet.

- List the contents of the C source code. Look for the line that starts the second loop.

(gdb) list 1,20

- Identify the source code line number that starts the second loop, and execute up to that point, with something like:

(gdb) until 14

- Now that the first loop is complete, let's look at local variable values again:

(gdb) info locals

Copy the output into your worksheet.

- Now that s1 is not obfuscated, what is its secret value? Write your answer in your worksheet.

- Looking at the source code, describe the means by which the string is obfuscated. Write your answer in your worksheet.

- Quit the debugger

(gdb) q