CSCI 2500: Computer Organization

Lab 9 – Exercises GNU Make / Digital Logic

A compiler compiles a C source file (*.c file) plus some header files (*.h files) into an object file (.o file). Make (`make`) is a tool that helps you to organize the build process, so that whenever a source file changes, the files which depend upon it get rebuilt automatically. A makefile is a special file, containing shell commands, that you create and name Makefile. While in the directory containing this makefile, you will type `make` and the commands in the makefile will be executed. If you create more than one makefile, be certain you are in the correct directory before typing make.

Make keeps track of the last time files (normally object files) were updated and only updates those files which are required (ones containing changes) to keep the source file up-to-date. If you have a large program with many source and/or header files, when you change a file on which others depend, you must recompile all the dependent files. Without a makefile, this is an extremely time-consuming task.

The makefile contains a list of *rules*. These rules tell the system what commands you want to be executed. Most times, these rules are commands to compile (or recompile) a series of files. The rules, which must begin in column 1, are in two parts. The first line is called a *dependency* line and the subsequent line(s) are called *actions* or *commands*. The action line(s) must be indented with a tab.

```
RULE: DEPENDENCY LINE
      [tab]ACTION LINE(S)
```

The dependency line is made of two parts. The first part (before the colon) are *target* files and the second part (after the colon) are called *source* files. It is called a dependency line because the first part depends on the second part. Multiple target files must be separated by a space. Multiple source files must also be separated by a space.

```
DEPENDENCY LINE:TARGET FILES:SOURCE FILES
```

After the makefile has been created, a program can be (re)compiled by typing `make` in the correct directory. *Make* then reads the makefile and creates a dependency tree and takes whatever action is necessary. It will not necessarily do all the rules in the makefile as all dependencies may not need updated. It will rebuild target files if they are missing or older than the dependency files.

In this lab you will learn how to apply the GNU make program to manage a simple project.

Exercises:

1. In a new directory, create a set of four files as follows:

main.c:
```
#include <stdio.h>
#include "pi.h"
#include "trig.h"

int main()
{
 double ang = 90.0 * pi/180.00; /* 90 Degrees in Radians */
 printf("cotan of 90 degrees: %0.4f\n", ctn(ang) );
 printf("cotan of 45 degrees: %0.4f\n", ctn(ang/2) );
}
```

trig.c:
```
#include <math.h>
#include "trig.h"

/* function definition */
double ctn(double x)
{
   return cos(x)/sin(x);
}
```

trig.h:
```
/* function prototype */
double ctn(double);
```

pi.h:
```
const double pi = 3.14159265358979323385;
```

Once these files are written and saved, compile them using the following command:
`gcc -g -Wall main.c trig.c`

2. Now we are going to write a simple Makefile to help us compile the code in the previous exercise. Create a text file called `Makefile` (Note: the name is case-sensitive) which contains the following code:

```
# Makefile for Lab 9
CC = gcc
CFLAGS = -g -Wall
LDFLAGS =

lab9: main.o trig.o
        ${CC} ${CFLAGS} main.o trig.o -o $@ ${LDFLAGS}

clean:
        rm lab9 *.o
```

Then, type into your Cygwin shell (or Linux shell) the following command:

```
make lab9
```

If you have trouble getting this to work, be sure that you put a TAB character in front of the "${CC}...". If you have cut and pasted the code from this lab exercise document, the TAB will not get carried over into your file; instead, a bunch of space characters (" ") will appear. You must replace all of the spaces by a single TAB.

Notice that the `make` utility calls `gcc` three times, the first two times it uses the "-c" option to create "object" code, main.o and trig.o, both of which are more or less "halfway to executable" : they need to be linked with each other and linked with the standard C libraries in order to fully complete the compilation process. That is the role of the third and final `gcc` call.

Run the "lab9" executable and check the results. Then then determine the effect of the following command:

```
make clean
```

3. In this part of the lab exercise, we consider “dependencies” in the `make` utility.

   First, get your program "lab9" rebuilt (you'll need to do this if you just typed "`make clean`") with the shell command:

   ```
   make lab9
   ```

   as in the previous exercise. Now type the same thing again. What happens?

   Next modify the file main.c. (either edit with a text editor or simply use the unix "`touch main.c`" command -- if you choose to edit the file, simply add an extra newline to the bottom and save the file)

   Run the `make` command again, using target lab9 as before. Note which source files are recompiled ("`gcc -c`").

   Now, edit the pi.h header file, doing something ridiculous like setting by pi=0.1415, save it and type "`make lab9`" again. Run the lab9 executable. Do the changes in the header get incorporated into the lab9 executable?

   We just learned that make tends to do a good job of understanding the dependency of object (.o) files upon C source (.c) files, but make has no built-in facility for determining, nor convenient way of expressing, the dependencies on header (.h) files.

We can fix this by adding the following lines at the end of your Makefile:

```
main.o: pi.h trig.h

trig.o: trig.h
```

Now type "`make lab9`" again. Take notice of which files are recompiled.

At this point, fix pi.h so that "pi" has its correct value. Then shut down your text editor so that you are no longer editing any of the source files involved in this exercise. Systematically "`touch`" all the files, *.c, *.h, and *.o, in groups to trick the `make` utility into thinking each group of files have been updated. Between each "`touch`" command, invoke the `make` utility (simply enter `make`) and note which files get recompiled.

The idea here is that when lots of source files are needed to make an executable program, `make` will efficiently recompile only code that has been updated more recently than the executable.

4. For the next part of this lab we will review and use a more generic Makefile. Rename the current `Makefile` in your directory to `Makefile_simple`. Download `Makefile` from LMS; Labs → Lab 9 (11/4/2015). Review the Makefile and note the differences between the simpler `Makefile` we created by hand earlier.

   As in exercise 3, systematically "`touch`" all the files, *.c, *.h, and *.o, (again in groups) to trick the `make` utility into thinking the files have been updated. Between each "`touch`", invoke the `make` utility (simply enter `make`) and note how each of the files get recompiled.

   Add a new target called 'test' to this Makefile that will run the `lab9` executable. This new target must check to make sure the executable exists (if not it must be automatically compiled) before executing the program.

5. For the final part of this lab you are to create and unit test a half adder and 1-bit full adder. Download lab_9.c from LMS and add code to simulate and unit test both adders. Hint: You may use the half-adder (and other gates) when constructing the 1-bit full adder. A truth table for the full adder is given below:

A	B	Carry-In	Sum	Carry-Out
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1

A	B	Carry-In	Sum	Carry-Out
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1