Session 3 Lab

Purpose

The purpose of this lab is to understand the compilation process and how different optimization levels affect the compilation of code. An additional goal of this lab is to understand how a makefile works and how you can create one.

If you have any questions about how different parts of the compilation process work, they will probably be covered throughout this course.

How to Access the Lab

Make sure to login to a shark machine before you begin. Follow the steps below to copy the lab into your private directory.

Navigate to a private directory where you want to complete the lab and type:

```
$ wget https://www.cs.cmu.edu/~213/bootcamps/lab3_handout.tar
```

To decompress the tar file, type:

```
$ tar -xf lab3_handout.tar
```

Part 1: Compilation Steps Walkthrough

Skim through the code in lab3.c and lab3_helper.c. There should be some unnecessary junk code placed throughout the files, ignore it for now.

1. Compile lab3.c using the **-E** flag. Place contents in a .txt file
    - `$ gcc -E lab3.c > pre_processor.txt`
    - Make a note of how the actual code for lab3.c changes (scroll to the bottom of the text file to see the code!)
    - If you wish to do this with the helper file, you can, but you will not notice many changes.
2. Compile lab3.c and lab3_helper.c separately using the **-S** flag
    - `$ gcc -S lab3.c`
    - `$ gcc -S lab3_helper.c`
    - Look through the generated assembly files. You do not need to understand what the code means.
3. Compile lab3.s and lab3_helper.s separately using the **-c** flag.
    - `$ gcc -c lab3.s`
    - `$ gcc -c lab3_helper.s`
    - Look through the code of the newly generated .o files using `xxd [file].o | less`. Can you read it?
    - View the actual machine code and place the contents in a .txt file to look at them later, using `$ objdump -s lab3.o > objdumplab3.txt`
        - You can also look at the machine code next to the assembly code by typing: `$ objdump -sdr lab3.o`
4. Compile lab3.o and lab3_helper.o together using the **-o** flag to rename the output file to lab3.
    - `$ gcc -o lab3 lab3_helper.o lab3.o`

- Look through the code of the newly generated executable, and run it to see what happens!

5. Compile lab3.c and lab3_helper.c together using the **-o** flag to rename the output file to lab3.
    - `$ gcc -o lab3 lab3_helper.c lab3.c`
    - You don't always have to compile the code in individual steps!

Visualizing Optimization Levels

Now, let's try to understand the differences between the optimization flags**: -O0, -O1, -O2, -O3,** and **-Os**. Optimization levels will show you different results and that some parts of the code will change, but you don't have to understand what is happening for now. Compile lab3.c and lab3_helper.c using each flag. It will be helpful to follow the command formatting below to create the different executable files.

Task 1:

Next, use `$ objdump -d [NameOfExecutable]` to find the assembly used to create the executable. It might be easiest to place the output of objdump into a .txt file. Note overarching observations about the differences in the assembly files. You can use: `$ diff [file1] [file2]` to print in the console the differences between the two files if you wish.

If you have time, feel free to try to hypothesize the reason for the differences. You do NOT need to understand what the assembly code is doing. For example, run these commands in the following order:

```
$  gcc -O0 lab3.c lab3_helper.c -o optimization0
$  objdump -d optimization0 > optimization0.txt
```

Remember to repeat these steps for the **-O1, -O2, -O3, -Os** flags!

Task 2:

Compare the difference between all of the optimization text files by utilizing the word count command: `$ wc optimizationX.txt`

Data Output: `LineCount WordCount Bytes FileName`

Make sure you determine file properties for when you compiled the lab using different optimization flags.

The last two line of the executable output has the following information:

```
Clock ticks: [NUM]
Time in seconds taken by CPU: [NUM]
```

This information represents different forms of timing of the execution. What do you notice about the timing for different optimization levels?

Part 2: Modify Makefile

As mentioned in the powerpoint, a Makefile is simply a recipe that contains a list of steps that the computer carries out. The goal of part 2 is to modify the Makefile. Read and complete through the // TODOs in the file. When you are done, run `$ make`. If you are trying to run a specific target, you can type `$ make [NameOfTarget]`.

The goal of this makefile is to see how you can use a Makefile to follow the same steps you did to compile your executable. There are comments describing what each rule does. Normally, you would not include all of the compilation steps in a Makefile, and you would just include the last target which compiles everything together!

HINT: Did you run into errors while running `$ make`. If so, make sure to address that unnecessary junk code placed throughout the files that we mentioned at the start of the lab.