Lab 1

Objectives:

- Introducing Linux commands
- Compiling and executing a C program using the commands from the command line.
- Compiling and executing a C program using a makefile.

You do not need to hand-in anything in this lab!

Resources:

In this course, you need to be familiar with common Linux commands and be able to write basic C programs.

For a summary of basic Linux commands:

- https://ryanstutorials.net/linuxtutorial/cheatsheet.php
- https://www.guru99.com/must-know-linux-commands.html

Many more Linux command tutorials can be found online. Feel free to peruse.

For a review of C fundamentals, try the following introductory C tutorials:

- https://www.tutorialspoint.com/cprogramming/
- https://cvw.cac.cornell.edu/Cintro/
- http://www.w3schools.in/c/

Many more C tutorials can be found online. Feel free to peruse.

Also, have a look at the Resources web page on our course web site.

Part 1: Setup

1. If you are in CSIL, then log to one of the CSIL workstations. If you are not in CSIL, then remotely login to one of the CSIL machines in the Linux environment (Ubuntu). To do so:
    - Click on the link Remote Login to Linux on CSIL – new : VPN (found in the menu on the left on our course web site) and follow the instructions. You may want to connect to CSIL using the ***Remote Desktop***.
    - Use your ***SFU account name*** (a.k.a. *username* or *userID*) and campus password.
    - You can also work locally using a Linux Ubuntu VM on your computer. You will need to adjust some of the instructions below. For example, you may not have the **`sfuhome`** directory on your local drive.

- If you work locally on your computer, I **strongly** suggest that, once you have compiled and tested your code on your local computer, you compile it on the ***target machine*** (CSIL machine) since this is the environment we will be using in this course.

2. Start a browser, access the course web site through it and open **`Lab1`**.
3. Start a **`Terminal`** window.
    - At this point your current directory path should be **`/home/userid`** where **`userid`** is your *SFU username*.
    - To check, use the Linux command **`pwd`**.
4. Make **`sfuhome`** your working directory by using the command **`cd sfuhome`**.
    - This **`cd`** command is the ***change directory*** command.
    - Note: It is important that you complete all of your coursework within the **`sfuhome`** directory so that your work is on the network instead of being on one of the local machines. This way, your work can be accessed from any SFU and CSIL computer.
5. Within your **`sfuhome`** directory, create a **`CMPT295`** directory.
    - To create new directories (or folders), use the command **`mkdir`**.
    - Alternatively, you can use the **`Files`** application to navigate through your **`sfuhome`** directory and create the **`CMPT295`** directory.
6. Download **`Lab1.zip`** from our course web site, save and unpack it (using the command **`unzip`**) in this **`CMPT295`** directory.
    - Tip: If this command failed, make sure you have saved the file in the correct directory. You may either use the **`Files`** application or use the Linux command **`mv`** to move it to the correct location.
    - Alternatively, you can unpack compressed files using the **`Extract Here`** command of the **`Files`** application.
7. There should now be a subdirectory named **`Lab1`** inside your **`CMPT295`** directory.
    - Change directory using **`cd Lab1`**.
    - Alternatively, you can use the **`Files`** application to navigate to the **`Lab1`** directory.
8. Within the **`Lab1`** directory there should be three files: **`main.c`**, **`makefile`** and **`times.c`**.
    - Use **`ls -la`** to see a listing of your directory.

- Alternatively, you can see the directory listing using the **Files** application.

Note: The above describes the procedure you will need to follow at the start of each lab in order to set the appropriate lab for the week, with the exception of Step 5 which needs to be done only once.

Part 2: Compiling and Executing a C Program – Using the Linux command Line

In this Part 2, you are led through the sequence of steps involved in compilation.

Yes, it is possible to compile and create an executable file with a single Linux shell command, but in systems programming it is often important to perform each step separately in order to produce intermediary files such as files containing the assembly code version of the C program being compiled. We shall talk more about this "assembly code version of a C program" over the next few weeks.

1. Open each file found in the **Lab1** directory using your favourite editor.
2. Make sure you understand the C code before proceeding. Feel free to ask the TA if you have any questions.
3. Use the command:

   ```
   gcc -E main.c > main.i
   ```

   to invoke the compiler's preprocessor.
4. Open and have a look at the resulting file **main.i**.
   - Notice that the preprocessor strips out all comment statements and also replaces the compiler directives **#include <stdlib.h>** and **#include <stdio.h>** by the content of these two system's library header files. These header files contains their own comment statements and compiler directives that are also stripped and replaced respectively.
5. Preprocess **times.c** to produce **times.i** and have a look at it.
6. To convert the preprocessed **main.i** file into x86-64 assembly, use

   ```
   gcc -Og -S main.i
   ```

   - The flag **-Og**, a capital letter '**O**' followed by a lowercase letter '**g**', advises the compiler to perform some optimization of the code. Without it, a number of unnecessary x86-64 instructions would be generated.
7. Convert the preprocessed **times.i** file into the x86-64 assembly file **times.s**.
   - We shall have a closer look at this type of file (***.s**) in a future lab.

8. The next step is to translate the assembly into object code, i.e., a machine language program (or instructions). Use

   `gcc –g -c main.s`

   `gcc –g -c times.s`

   `OR  gcc –g -c main.s times.s`

   - The resulting files will be **`main.o`** and **`times.o`**.

9. The files **`main.o`** and **`times.o`** are binary files, not text files, so your favourite code editor may not be useful here.

10. To execute the machine language instructions, the files **`main.o`** and **`times.o`** must be linked together, along with the system library subprograms like **`printf()`**. To do the linking, use the command:

    `gcc -o mul times.o main.o`

    which generates an executable program called **`mul`**.

11. To run the program, use **`./mul`**.

    - Note: To execute a program, prepend its name with "**`./`**", which means "the current directory".
    - As you saw when you looked at **`main.c`**:

      ```
      main(int argc, char *argv[]) {
      …
        if (argc == 3) {
          a = atoi(argv[1]);
          b = atoi(argv[2]);
      ```

      you will need to provide two (2) integers on the command line in order to successfully execute this program. Here is the command's syntax:
      `./mul <integer1> <integer2>`.

12. In the final step of this Part 2, we shall remove the object files and the executable file created in this part of the lab by typing this command **`rm –f *.o mul`** at the command line. You'll see why we are doing so in a moment!

Part 3: Compiling and Executing a C Program – Using a makefile

In this Part 3, you are to discover how a simple *makefile* works.

1. First, have a read through this short yet informative article on makefiles: https://www.cprogramming.com/tutorial/makefiles.html

2. Now, have a look at the makefile found in the directory **Lab1**. Makefile is a text file and can be open in your favourite code editor. You will notice that it looks a little different from the makefiles shown in the article. What our makefile actually contains is an expanded version of the command associated with the **build** target shown in the article, namely **$(CC) –o $(OUT_EXE) $(FILES)**. This expanded version has been illustrated in Part 2 of this lab. Can you see a correlation between the content of our makefile and the commands listed in Part. 2 above?
3. Now, issue the command **make** at the command line and observe that each of the commands in the makefile are echoed on the screen as they are executing. Have a look at your directory listing to see which new files have now been created.
4. Issue the command **make** once again and observe what happens. As the article explained, only the code files that are newer than their object files are recompiled. Since none of them are newer than their object files, nothing needs to be recompiled hence the message `make: 'mul' is up to date.`

   To force a recompile, issue the command **make clean.** This will remove all the object files (*.o) that were created by the previous execution of **make**.
5. Now, let's modify the makefile by uncommenting the three (3) macros defined at the top. You will also need to figure out in which of the five (5) compile commands to put each of these macros. Make sure you use the syntax described in the article when putting a macro in a compile command and make sure you remove the flags that were initially in the compile command.
6. Once your makefile has been modified, save it and issue the command **make**. Observe the expanded commands being printed by **make** as it executes.
7. Finally, insure that the expected executable has been created properly by executing it.

Part 4: Demo Programs

Spend some time examining the Demo programs that have posted on our course web site under Lecture 4 and Lecture 5, and under Lecture 7 later on this week.

As C software developers, it is important for us to be familiar with the effect of fixed-size memory on our data when it is converted from one data type to another and when it is manipulated through arithmetic operations. This is what these Demo programs illustrate.

Here is a question for you:

1. Download the Demo code posted under Lecture 4, i.e., the code file **Lecture_4_Demo.c** and open it in your favourite code editor.

2. What happens when you change the data type of the iterating variable **`i`** from an **`int`** to an **`unsigned int`** in the function **`show_bytes(…)`**?
3. Can you explain why?

Part 5: Challenge

In this final part, the challenge is for you to find a Linux command that would allow you to answer the following question:

- Is the CSIL machine you are currently using a 32- or a 64-bit machine?

Thank you to Dr. Dixon for having inspired this lab.