Family Name:_____

Given Names:_____

ID#:_____

CMPT 295 Lab 1 (0.5%)

Thank you to Dr. Dixon for this lab.

The purpose of this lab is to introduce the Linux commands and tools to compile C into assembly code, edit the assembly code, convert the assembly into object code, and link the object code into an executable.

Note: To do some of the Labs and Assignments, you will need to be familiar with common Linux commands and to be able to write basic C programs.

For a summary of basic Linux commands:

https://ubuntudanmark.dk/filer/fwunixref.pdf
http://www.karlin.mff.cuni.cz/ hron/NMNV532/ShellIntro.pdf

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

https://cvw.cac.cornell.edu/Cintro/
http://www.w3schools.in/c/

Part 1: Setup

- Login to the CSIL machines in the Linux environment. Use your SFU account name and campus password.

  Note: If the machine is in Windows mode, use `Ctrl-Alt-Del` to restart the machine and select "Ubuntu" from the dual-boot menu.

- Start a web browser and download the care package for Lab 1 from the course web site:

  `www.cs.sfu.ca/CC/295/bbart/`

  Tip: Most Labs and Assignments will start you off with some base code and other provided files within its corresponding *care package*.

- Save the care package within your `sfuhome` directory.

- Start a Terminal window. At this point your current directory path should be `/home/userid` where `userid` is your SFU user name. To check, use the Linux command `pwd`.

- Make `sfuhome` your working directory by the command `cd sfuhome`.

  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 on just one of the *local machines*. You may create new directories within `sfuhome` by using the `mkdir` command.

- Unpack the care package using `unzip care-l1`.

  Tip: If this command failed, it is because you saved the care package in the wrong directory. You may either use the GUI or use the Linux command `mv` to move it to the correct location.

- There should now be a directory named `lab1`. Change directories using `cd lab1`.

- Within the `lab1` directory there should be two files. Use `ls` for a directory listing.

Part 2: Compiling and Executing a C Program

The next phase of this lab will take you through the sequence of steps involved in compilation. Yes, it is possible to compile and create an executable file with a single Linux command, but in systems programming it is often important to perform each step separately.

You should see two files: `main.c` and `times.c`. The former calls a subroutine `times(x,y)` that multiplies two numbers and returns the result. You should review and understand the C code before proceeding. View each file using your favourite editor, or by the Linux command `cat filename`.

- Use the command: `gcc -E main.c > main.i` to invoke the compiler's *preprocessor*. This step strips out all comment statements and also replaces the compiler directive `#include <stdio.h>` by the system's library header file. The header file itself contains its own comment statements and compiler directives that are also stripped and replaced recursively. The resulting file, `main.i`, is a rather lengthy text file that you can peruse within your favourite text editor or by using `cat main.i`.

  Before you proceed, preprocess `times.c` to produce `times.i`.

- To convert the preprocessed `main.i` file into x86-64 assembly, use `gcc -S -Og main.i`. (A similar command will convert `times.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. The flag `-Og` asks to generate assembly that most closely resembles that of the original C code.

  The resulting files are `main.s` and `times.s`. It may be worth it to have a look at each file using your favourite text editor or `cat`. Lines that begin with a period are *pseudo-ops*: they are not x86-64 instructions, but rather directives to help guide the assembler when it creates the object code.

- The next step is to translate the assembly into *object code*, i.e., a machine language program. Use `gcc -c main.s`, etc.; the resulting files will be `main.o` and `times.o`. These are binary files, not text files, so your favourite text editor or `cat` will not be useful here. Instead, you can use the *disassembler*: `objdump -d main.o`. Each instruction of object code is shown on the left (in hex), and its corresponding assembly is shown on the right. You may wish to compare with `main.s`: they should be synonymous. The object code will contain no comments and no directives, however.

- To execute the machine language, the files `main.o`, `times.o` must be *linked* together, along with the system library subprograms like `printf()`. Use the command `gcc -o mul times.o main.o`, which will generate an executable program called `mul`. To run the program, use `./mul`.

  Note: To execute a program, prepend its name with "`./`".

Part 3: Writing Assembly

One of the virtues of programming in assembly language is that you can often write much shorter, faster assembly language programs than those generated by the compiler, even with some of the higher levels of optimization enabled. Systems programmers often write routines initially in C, and then examine the generated assembly language code for opportunities to optimize it.

For example, in Part 2, the optimized assembly generated for `times` was 3 instructions long. If you generated an *un-optimized* version, it would have been 10 instructions. In this part, you will create a new version of `times(x,y)`.

- Edit a new file, call it `newtimes.s`, and enter the following code:

```
        .globl times
times:  leal    (%edi, %esi, 4), %eax
        ret
```

  Tip: As a matter of style, labels should usually appear flush left, followed by a colon. Assembly commands and directives are prepended by white space, usually a single `tab`.

  Note: The directive `.globl` is for the linker so that the mainline program (in `main.c`) will be able to find the subroutine.

- Assemble the subprogram in a different way than before: `as newtimes.s -o newtimes.o`.

- Re-link the object code: `gcc -o newmul newtimes.o main.o`, and run the new executable.

- This is the last step of Lab 1. Call the TA to your terminal to verify your work. The TA will then collect this sheet.

over . . .

Thinking Ahead

The *function call protocol* describes how functions and their caller routines should behave. Also, because it's a protocol, it is expected that all implementations across all compilers should be compatible with one another.

Part of the protocol discusses parameters and return values. As a matter of efficiency, these are passed in registers whenever possible. The standard includes the following rules:

- `%rax` holds the return value.
- `%rdi` is the first parameter.
- `%rsi` is the second parameter.
- Parameters 3-6 go in `%rdx`, `%rcx`, `%r8` and `%r9`, in that order. (If more than 6 are required, the *stack* is used for the 7^{th}, 8^{th}, etc.)

As an extra exercise, attempt the following questions:

1. In `newtimes.s`, identify which register is used for variables `x`, `y` and `result`.
2. What do you think the instruction `leal (%edi, %esi, 4), %eax` does? Test your hypothesis by trying different starting values for the variables `a` and `b`.