Compiling C and Assembling x86-64 Source Files in Linux

The purpose of this lab is to introduce the Linux commands and switches for compiling C into assembly programs, converting assembly language programs to object code, and linking object code to provide a runtime file.

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

For a summary of basic linux commands:

https://ubuntudanmark.dk/filer/fwunixref.pdf

http://vic.gedris.org/Manual-ShellIntro/1.2/ShellIntro.pd

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

https://cvw.cac.cornell.edu/Cintro/

http://www.w3schools.in/c/

Selecting the Proper CSIL Platform : Since you will be programming in C and x86-64 assembler, you will need to run your programs in the Linux environment.

1. logon to a CSIL Linux/Windows dual boot workstation in ASB 9840. Launch Ubunto Linux as follows:
   (a) If the computer is off, push the power button and go to step (d).
   (b) If the computer is on, with Ubunto Linux already running, go to step 2.
   (c) If the computer is on with Windows running then:
      i. Push Ctrl-Alt-Del
      ii. Left-click the cursor arrow on the red power button near the bottom-right corner of the screen
      iii. Left-click "Restart"
      iv. Wait till the OS select screen shows (this may take awhile)
   (d) With the up-arrow key, highlight "Ubuntu" (the default OS), select it with the "Enter" key, and go to step 3.
2. Login when the Ubuntu login screen is deiplayed.

   At this point your current directory path should be /home/<*userid*> where <*userid*> is your SFU user name. Check with the linux command:

   `pwd`
3. Make "sfuhome" your working directory:

   `cd sfuhome`

   NOTE: It is important that all your assignments be created in your `sfuhome` directory. You may create subdirectories (using the "`mkdir`" command) within `./sfuhome` if you wish.

Compiling and Executing a C Program : While it is possible to compile and create an executable file with a single linux command, in systems programming it is often important to perform each step of the compile and execute process separately. The following steps illustrate how to do this.

1. Using a text editor, define the following C source file, saving the text within `sfuhome` in a file called `testmain.c`:

```
#include <stdio.h>

/*Lab 1: Test program 1*/

void main(){
  int answer = 2 * 3;
  printf("Answer is %d \n",answer);
  return;
   }
```

2. "Preprocess" this C source file with the following command:

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

NOTE: The symbol ">" is a linux command line directive to send the output of the command to a file called `testmain.i`. It is important that the file extension be ".i ".

This step strips out comment statements and directives introduced by '#" and introduces directory paths to various routines that will be required to translate the C program to assembly and machine language. The file testmain.i is a rather lengthy text file that you can view with:

```
cat testmain.i
```

3. The preprocessed C source file, `testmain.i`, can now be converted to an x86-64 assembly language program with:

```
gcc -S -Og testmain.i
```

The flag "-Og" (the letter "oh", not the number zero) advises the assembler to perform some optimization of the code. Otherwise, if the assembler were to translate the C program without optimization, a number of unnecessary x86-64 instructions and "pseudo-ops" would be generated. The result of assembling the preprocessed C program is placed in the file testmain.s. It is a text file and its contents can be displayed with:

```
cat testmain.s
```

Lines of text that begin with a period are "pseudo operations (pseudo-ops)." They are not x86-64 instructions but rather directives that will be used by the assembler when an object file is created.

4. The next step is to translate the assembly language program into an object file; that is, a machine language program. This is achieved with the command line:

```
gcc  -c testmain.s
```

This produces an incomplete object file, "`testmain.o`." Since this is a binary file rather than a text file, it cannot be displayed meaningfully with the "`cat`" command. However the machine language code and the corresponding assembly language instructions can be displayed with:

```
objdump -d testmain.o
```

The "`objdump`" program is called a "disassembler." It translates a machine language program back into assembly language. The object code is shown on the left-hand side. The length of

instructions varies and each line shows the contents of each byte that collectively make up the instruction. Values stored in each byte are expressed in hexadecimal.

5. In order to execute the machine language program it must be linked with the machine language libary programs like 'printf." As well it must be loaded into an absolute address in external memory. This is achieved with the following command:

   ```
   gcc testmain.o -o run_testmain
   ```

   This command creates a run-time linked executable file called "run_testmain." The program is now ready for execution and can be exected with the command line statement:

   ```
   ./run_testmain
   ```

   NOTE that in order to execute the run time file, the name is preceded by " ./ ".

Linking an x86-64 assembly subprogram to a C main program :

1. Edit the C program stored in `testmain.c`, replacing the statement:

   ```
   int answer = 2 * 3
   ```

   by the statement:

   ```
   int answer = times(2, 3)
   ```

   You will also need to include the following statement prior to the "main" declaration statement and vollowing the "#include..." statement:

   ```
   int times(int, int)
   ```

   Your C source file, `testmain.c`, should look like this:

   ```
   #include <stdio.h>

   int times(int, int);

   void main(){
     int answer = times(2, 3);
     printf("Answer is %d \n",answer);
     return;
   }
   ```

2. Using your text editor define the following x86-64 assembly language source program, storing it in a file called "subprogram.s":

   ```
           .globl  times
           .text
   times: -c
           mov     %edi, %eax
           imul    %esi, %eax
           ret
   ```

3. Assemble this subprogram with the following command:

   ```
   as subprogram.s -o subprogram.o
   ```

4. Compile and assemble the revised version of `testmain.c` with the following statements:

   ```
   gcc -S -Og testmain.c
   gcc -c testmain.s
   ```

   Note that this command will skip the creation of the testmain.i file, replace the testmain.s file that contains the assembly language version of the compiled `testmain.c` file, and then replace the `testmain.o` file with the new object code.

5. Now construct a linked executable file called "`run_times`" with the command:

   ```
   gcc testmain.o subprogram.o -o run_times
   ```

6. Test the executable file, `run_times`, to see whether it produces the same result as `run_testmain`, the run time file created earlier:

   ```
   ./run_times
   ```

7. Subprograms written in C can also be compiled and linked in the same way. For example, enter the following C program in a file called "`times.c`":

   ```
   int times(int x, int y){
     int answer = x * y;
     return answer;
     }
   ```

8. The C main program defined previously can be compiled and linked with this subprogram instead of the x86-64 assembly language subprogram:

   ```
   gcc -Og -o runtimesv2 testmain.c times.c
   ```

9. Confirm that the executable file produces the same result as that obtained with the x86-64 subprogram:

   ```
   ./runtimesv2
   ```

10. You can generate the assembly language for the C function "`times`" by simply compiling the C subprogram source file, `times.c` with:

    ```
    gcc -Og -S times.c
    ```

11. Compare the contents of the assembly language file `times.s` that is created by the previous command with the assembly language file `subprogram.s` that is defined above in step 2. Ignoring all the pseudo-ops (lines beginning with a period), you should observe that the assembly language program generated by the assembly of the `times.c` file is much longer.

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