Lab 3

Objective:

- Using **`makefile`** and compiler optimization
- Introducing the assembly language x86-64
- Introducing the disassembler
- Introducing function call protocol in x86-64
- Debugging assembly code
  - Our textbook lists GDB commands in Figure 3.39 on page 280.

We do not need to hand-in anything in this lab session!

Part 1: Compiling and executing a C program using **`makefile`** and optimization

In the first part of this lab, we shall use the files we downloaded for Lab 1, namely, **`main.c`**, **`times.c`** and **`makefile`**. Use the **`makefile`** we improved in Part 3 of Lab 1.

- Copy them onto our **`sfuhome/CMPT295/Lab3`** directory.
- Compile the code files using **`make`** and have a look at the content of our directory. Amongst the files produced by the compilation, we should see the following:
  - **`main.s`** and **`times.s`**. These files contain the assembly code version of the C program located in the respective files. Have a look at each file using our favourite text editor. Lines that begin with a period are assembler directives: they are not x86-64 instructions, but they help guide the assembler when it creates the object file.

    NOTE: Since we have just started learning x86-64 assembly language, it is quite normal that at this point we may not understand all the assembly instructions we see in these files right now.
  - **`main.o`** and **`times.o`**. These files are the object files created by the assembler. Objects files are binary files: they contain the machine code (0's and 1's) version of the C program located in the respective files. These are not text files, so our favourite text editor may have a difficult time displaying their content. It may either display this binary content as text and in this case, we get very interesting display of all sorts of characters, or it may actually be capable of displaying the 0s and 1s as hexadecimal numbers. Try it and see what happens!
- To know what the machine code version of our C program looks like (i.e., the actual 0's and 1's or hexadecimal numbers expressing these 0's and 1's), we can use the disassembler. Try the following commands:

objdump -d main.o

objdump -d times.o

By the way, if we are wandering why the flag **-d** is used, here is what the **man** page for the command **objdump** says (https://man7.org/linux/man-pages/man1/objdump.1.html):

```
-d
--disassemble
--disassemble=symbol
    Display the assembler mnemonics for the machine instructions
    from the input file.  This option only disassembles those
    sections which are expected to contain instructions.  If the
    optional symbol argument is given, then display the assembler
    mnemonics starting at symbol.  If symbol is a function name
    then disassembly will stop at the end of the function,
    otherwise it will stop when the next symbol is encountered.
    If there are no matches for symbol then nothing will be
    displayed.
```

- As we can see, the output of these commands can be lengthy so we may wish to pipe them (i.e., redirect their output) to files. Try these commands:

  objdump -d main.o > main.objdump

  objdump -d times.o > times.objdump

  Note that the extension given to these files (**.objdump**) is arbitrary which means that we can give any extension we wish. Let's remember that using a descriptive extension is always very helpful. However, do not use extensions already used by **gcc** such as "**.s**", "**.o**" and "**.c**".

- Let's open **main.objdump** in our favourite text editor. We see three columns below the label **main**:
  - the rightmost column lists the assembly instructions representing our **main** function in **main.c**,
  - the middle column is the machine code representing these assembly instructions (this machine code is expressed as hexadecimal numbers), and
  - the leftmost column (the one with the colon ":") lists the memory address **offsets** (expressed as hexadecimal numbers) corresponding to each of these machine code instructions (i.e., assembly instructions). More on this later.
- Compare the content of **main.objdump** with the content of **main.s**. Their assembly instructions should be very similar. However, since the code in the object file **main.o**

(which was used to create this **`main.objdump`**) does not contain comments nor directives, the resulting **`main.objdump`** does not either. Furthermore, there are at least four main differences (aside from the lack of comments and directives) between the assembly instructions in **`main.s`** and **`main.objdump`**. For example, one of these main differences is that in **`main.objdump`** all immediate values (**Imm**) are expressed in hexadecimal numbers (as opposed to being expressed as decimal numbers in **`main.s`**). Here is an example where the decimal value 16 is used in **`main.s`**, but its hexadecimal representation, i.e., 0x10, is used in **`main.objdump`** instead:

```
main.s: movq 16(%rbx),%rdi => main.objdump: mov 0x10(%rbx),%rdi
```

Can we spot the other three main differences?

- Now compare the content of **`times.objdump`** with the content of **`times.s`**. Do we notice similar differences between the assembly instructions listed in these two files?
- Have a look at the leftmost column of **`times.objdump`**. As stated above, these hexadecimal numbers do not represent memory addresses (yet), but **offsets**, i.e., how far away *in memory* is each instruction from the first instruction of our function **`times`**, or more specifically: how many bytes away *in memory* is each instruction from the first instruction of our function **`times`**. (Remember: to be executed, our program must first be loaded, from its executable file - stored most likely on some hard disk - onto memory.) Let's verify this:
  - Start at byte 0, which is the offset of the byte containing the hexadecimal value **`f3`**, i.e., the first byte of our function **`times`**, then move rightward to the next byte containing **`0f`** counting 1 (i.e., this byte is located at offset 1), then move rightward again to the next byte (containing **1e**) counting 2 (i.e., this byte is located at offset 2), then move rightward once again to the next byte (containing **`fa`**) counting 3 (i.e., this byte is located at offset 3). This signifies that the first assembly instruction of our function **`times`**, namely **`endbr64`**, is expressed as a machine instruction of 4 bytes: **`f3 0f 1e fa`**. Therefore, the first byte of the next assembly instruction should be at offset 4.
  - Indeed, looking at the offset on line 9, we see **`4`**. This second machine instruction (**`mov %edi, %eax`**) is 2-byte long. Therefore, the offset of the third machine instruction is 4+2 = 6, and indeed it is.
  - Verify the offset of the last machine instruction of our function **`times`**.
- Also, how many bytes does the code of the function **`times`** occupy in memory? How could we answer this question without counting each byte?

The reason why these disassembled object files do not contain memory addresses in the leftmost column is because they have not yet been linked together and their instructions, and various other things (referred to as **tokens**), given addresses by the compiler (more specifically, the linker). Once we link both of our object files and produce our executable **`mul`**, we are then able to see the memory address of each instruction of our program when we disassemble it. Let's give this a try!

- Enter the following command:

  `objdump -d mul > mul.objdump`

- Open **`mul.objdump`** and notice its length. It is rather long. This is because it contains all the code needed to execute our program. This is to say that aside from containing the code for our **`main`** and our **`times`** functions, it also contains the code implementing all the C functions our C code calls such as **`atoi(…)`** and **`printf(…)`**.
- Now, have a look at the leftmost column. These hexadecimal numbers no longer represent **offsets**, but actual memory addresses. More specifically, each of them represents the memory address of the first byte of each instruction of the functions contained in our executable **`mul`**. For example, the second instruction of our **`times`** function, a **`mov`** instruction, has two bytes and its memory address is **`118d`** (at least on my computer ☺, it may be different on yours), which is the memory address of the **`mov`**'s first byte, i.e., the byte containing **`89`**.
  - As stated earlier in this lab, the middle column of **`mul.objdump`** lists the machine code representing the assembly instructions of the function (expressed as hexadecimal numbers) and the rightmost column, its assembly instructions.

Let's now investigate the effect of the **compiler optimization.**

- Currently our **`makefile`** uses the optimization level **`-Og`**. What if we did not use the optimizing capability of **`gcc`**? What kind of assembly program and machine code would we get? Well, let's try!

  Recall: The **`–Og`** flag requests the compiler to perform optimization that promotes the use of the debugger. More on the subject of using the debugger with assembly code later on in this lab.

- Let's remove the optimization flag **`–Og`** from the SFLAGS in our **`makefile`**.

  Notice that using macros such as SFLAGS in our **`makefile`** allows us to easily make modifications to the **`makefile`**. Here, we only need to make this one change (remove **`–Og`** from the macro) only once and it is automatically reflected throughout the **`makefile`**.

If we were not using macros in our **`makefile`**, we would need to make the change a few times throughout the **`makefile`** and this could be time consuming and very error-prone (what if we were to miss making one of the changes?).

- Let's also remove **`main.s`** and **`times.s`** in order for **`gcc`** to remake these files:

  ```
  rm main.s times.s
  ```

- Finally, let's clean our directory by using the command: **`make clean`**, then remake our executable.

- Have a look at the content of the new **`times.s`**. It is much longer than the previous one, i.e., the one we obtained using the optimization flag **`-Og`**, and its code seems more complicated. Remember, the previous **`times.s`** had four instructions so it seems more efficient as it performs the same task with fewer instructions.

- Now that we have done our investigation, let's put our optimization flag **`–Og`** back into our **`makefile`** (as part of the SFLAGS variable), let's remove **`main.s`** and **`times.s`** and let's clean our directory using **`make clean`**. Finally, let's rebuild our executable using **`make`** in order to be ready for Part 2 of our Lab 3.

We are now ready to get started with the assembly language x86-64.

Part 2: Getting started with the assembly language x86-64

One of the virtues of programming in assembly language is that software developers 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. Also, systems programmers often write routines initially in C, and then examine the generated assembly language code for opportunities to optimize it.

In this part of our lab, we will implement a new version of **`times(x,y)`**, which we will save in **`times.s`** instead of having the compiler creating **`times.s`** for us by compiling the C program **`times.c`**. In other words, we shall write our own assembly code version of **`times.s`**.

- Let's save our executable **`mul`** into a file called **`mul_Og`**.
- Open **`times.s`** and replace its ***entire content*** with the following:

```
    .globl times
times:
    xorl   %eax,  %eax
    movl   %edi,  %ecx
    movl   %esi,  %edx
    imull  %edx,  %ecx
    movl   %edx,  %eax
    ret
```

- Tip:
  - Labels such as **times** should usually appear flush left, followed by a colon.
  - Assembly instructions and directives are prepended by white space, usually **a single tab**.
  - Including **an empty line at the end of our file** prevents the assembler warning "end of file not at end of a line: newline inserted".
- Note:
  - The directive **.globl** is for the linker. It allows the linker to link (using memory addresses) the code of the called function (for example, **times**) to the calling function(s) (for example, **main**). Here is an illustration:

    In **mul.objdump**:

    ```
    0000000000001189 <times>:
        1189: f3 0f 1e fa              endbr64
    ```

    where **1189** is the memory address of the first byte of the first instruction of **times**.

    ```
    0000000000001193 <main>:
    …
    11d5:     e8 af ff ff ff      callq  1189 <times>
    ```

    where **callq 1189** is how the function **main** calls the function **times**.

    Note: The memory addresses you see in your **mul.objdump** may be different from the ones displayed in this lab.

- We may have noticed that this new implementation of **`times`** is dubious. If we have not, no problem! We shall discover how dubious it is in the next part of this lab.
- Make sure the SFLAGS in our **`makefile`** is back to **`-S -Og`**.
- Important: Also, comment out the following two lines in our **`makefile`**:
  If we don't, the compiler will overwrite our **`times.s`** with its own version.

```
times.s: times.c
     gcc $(SFLAGS) times.c
```

  becomes:

```
# times.s: times.c
#    gcc $(SFLAGS) times.c
```

- Let's **`make`** our executable.
- Finally, run the new executable as follows: **`./mul 1 5`**. Compare it with our initial executable: **`./mul_Og 1 5.`** Both should produce the same correct result:

  `times(1, 5) produces 5 as a result.`

- Let's try another test case: **`./mul 5 6`**. Oops! The result (**`6`**) is incorrect. Let's try this test case with the initial executable: **`./mul_Og 5 6.`** With our initial executable, we do obtain the correct result (i.e., **`30`**) and this confirms the fact that our new implementation of the function **`times`** is indeed buggy!
- Let's investigate why the last result is incorrect by using the debugger **`gdb`**.

Part 3: Function call protocol in x86-64, debugging assembly code and a challenge

Before we proceed, we need to learn a little about **function call protocol** in x86-64. In this lab, the **`main`** function calls the function **`times`**. Therefore, **`main`** is seen as the ***caller*** and **`times`**, as the ***callee***. In x86-64, the **function call protocol** describes how ***caller*** and ***callee*** functions should behave. Also, because this is a protocol, it is expected that all C compilers executing on our ***target machine*** will implement this protocol.

Part of the protocol deals with parameter passing and return value. As a matter of efficiency, these are passed in registers whenever possible. Therefore, the x86-64 **function call protocol** states 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 parameters are required, the stack is used for the 7th, 8th, etc.

We shall cover this protocol in more details over the next few lectures, but for now, the above gives us enough information about this **x86-64 function call protocol** for us to understand the rest of our Lab 3 and to grasp what is happening with our new **`times.s`**.

We shall now investigate what is happening with our new **`times.s`** using the **`gdb`** debugger.

Of course, we do not need to use the **`gdb`** debugger to figure out how this new implementation of our **`times`** function works and why it is dubious. We could simply hand trace it with a few test cases. But using the **`gdb`** debugger in this part of our lab will allow us to gain "debugging assembly programs" skills and these skills will come in very handy when we write our own assembly programs in this course. So, let's give it a go!

- **`gdb mul`**

  We can use the commands we have already used in the previous lab, namely, **`run`**, **`list`**, **`break`**, **`continue`**, **`display`**, **`print`**, etc… and proceed exploring **`times.s`** on our own. Or we can follow the steps below:

  - Set a breakpoint for **`main`**, then **`run`** the program as follows: **`run 1 5`**.
    - A note about this **test case**: in terms of terminology, its ***test data*** is **`1 5`** and its ***expected result*** is 1 x 5 = 5.
  - Do a **`list`** to see where we are in the execution of our program. We are about to execute the first instruction of **`main`**, i.e., **`endbr64`** (which we will cover in a few lectures from now).
  - Display the relevant registers.

    To print the value of a register, name the register with **$** (as opposed to **%**).
    - For example, to print the value of **`eax`**, use **`print $eax`**
    - For example, to print the value of **`eax`** in hex, use **`print /x $eax`**

    To display the value of a register automatically after every step, type **`display $eax`** or type **`display /x $eax`**.

    Determine which registers we need to display.

    Note that if we want to display the state of all registers using one command, type **`info registers`**. However, we will have to repeat this command at every step we take executing our program.
  - Set a breakpoint for **`times`**.
  - **`Continue`**. The function **`main`** executes and calls **`times`**. According to the **x86-64 function call protocol**, **`times`**' first parameter (here: **`1`**) is stored into the

register **%edi** and **times**' second parameter (here: **5**) is stored into the register **%esi**. Is this the case?

- Then **list** to figure out where we are in the program. We will then see the content of **times.s**. Notice the numbers on the left. As we saw in Lab 2, we can use these line numbers to set breakpoints as well.
- Step (command: **step** or **s**) into **times** 😉 and notice the values of the displayed registers changing.
  - After issuing our first **step** command, the following is displayed on the monitor screen:

    ```
    4           movl %edi, %ecx
    1: $eax = 0
    2: $edi = 1
    3: $esi = 5
    4: $edx = -5
    5: $ecx = 0
    ```

    It lists the content of the registers **%eax**, **%edi**, **%esi**, **%edx**, and **%ecx** once the instruction **xorl %eax, %eax** has executed. Notice **%eax** now contains **0**.

    Note that the displayed instruction **movl %edi, %ecx** above has not yet been executed. It is the instruction at which the execution flow has stopped. It will be executed next time we **step**.
  - After our second **step**: **movl %edi, %ecx** has executed and **$ecx = 1**
  - After our third **step**: **movl %esi, %edx** has executed and **$edx = 5**
  - After our fourth **step**: **imull %edx, %ecx** has executed and **$ecx = 5**
  - After the fifth **step**: **movl %edx, %eax** has executed and **$eax = 5**
- **Continue** once more time to complete the execution of our program. We should be seeing the output of our program on the monitor screen.
- Now, let's try the other test case: **run 5 6**.
  - Our **break** points should still be set as well as the **display** commands we issued earlier.
- Repeat the above instructions (the ones we follow for our first test case), i.e., **Continue** the execution to **times**, then **step** into **times** 5 times.
  - At every step, let's keep an eye on the instruction being executed during this step and the values stored in our registers.

    Do the registers contain the value we are expecting them to contain at every step?

    Can we see which instruction is incorrect, i.e., which one does not produce the result we are expecting?

Can we fix it? If so, let's fix it, recompile and see whether we have indeed solved the problem!

- That's it, everyone! I hope we have found this lab useful! 😊
