University of Windsor

School of Computer Science
Faculty of Science
COMP-2560: System Programming (Fall 2024)

Lab#	Date	Title	Due Date	Grade Release Date
Lab03	Week 03	**C and Assembly**[1]	One-Week Lab Oct. 01, Tuesday Midnight EDT	Oct. 07

The main objective of the second lab will be to learn how to **build**[2] a C program and make it a process in a UNIX-based/like operating system (OS). It includes 1) writing a program in C, 2) compiles it into assembly codes, 3) assembles it into opcodes, 4) linking it to library routines, and 4) make it a process (run it.)

Step 1. Simple C Program

Let's write a simple program that defines an integer variable with initial value 1 and quits:

```
hfani@alpha:~$ vi main.c
void main(void){
        int a = 1;
        return;
}
```

To make a run out of it, we have to translate it into the assembly codes by C compiler, `cc`, using `-S` (capital S):

```
hfani@alpha:~$ cc main.c -S
```

This translates `main.c` into assembly codes in a new file `main.s`:

```
hfani@alpha:~$ ls
Desktop  Documents  Downloads  eclipse-workspace  main.c  main.s  Music  Pictures  Public  Templates  Videos
```

Let's have a look at the assembly codes:

```
hfani@alpha:~$ vi main.s
        .file   "main.c"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    $1, -4(%rbp)
        nop
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
        .ident  "GCC: (Debian 10.2.1-6) 10.2.1 20210110"
        .section        .note.GNU-stack,"",@progbits
```

Although 4 lines of C program becomes more than 10 lines of assembly, you can find the assembly translation for some C lines, such as the initialization of the variable `a` to `1`:

```
        movl    $1, -4(%rbp)
```

[1] This lab is inspired by Lecture 27: C and Assembly by Ananda D. Gunawardena, Princeton University.
[2] Compiling, assembling, and linking all together is also called building.

Now, we have to translate assembly codes in main.s into the machine's opcodes (binary sequence of instructions for the processor). An assembler can do this for your machine. Fortunately, the C compiler, cc, has it built-in by the -c option (small c):

```
hfani@alpha:~$ cc main.s -c
```

This translates main.s into opcodes in a new file main.o, also known as **object file**:

```
hfani@alpha:~$ ls
Desktop  Documents  Downloads  eclipse-workspace  main.c  main.o  main.s  Music  Pictures  Public  Templ
```

Let's have a look at the opcodes in main.o:

```
hfani@alpha:~$ vi main.o
?ELF^B^A^A^@^@^@^@^@^@^@^@^@^A^@>^@^A^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^H^B^@^@^@
@UH<89>åÇEü^A^@^@^@<90>]Ã^@GCC: (Debian 10.2.1-6) 10.2.1 20210110^@^@^@^T^@^@^@^@^@
@^@^@^@^N^@^@^@^@A^N^P<86>^BC^M^FI^L^G^H^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^N^
@^@^@^@^@^@^@^C^@^A^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^C^@^B^@^@^@^@^@^@^@^@
^@^@^@^@^@^@^@^@^@^@^C^@^E^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^C^@^F^@^@^@^@^
@^@^@^@^@^@^@^@^@^@^@^H^@^@^@^R^@^A^@^@^@^@^@^@^@^@^@^N^@^@^@^@^@^@^@^@main.c^@main
@^@^@^@.symtab^@.strtab^@.shstrtab^@.text^@.data^@.bss^@.comment^@.note.GNU-stack^@.re
^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^
@^@^@^@^@^@^@@^@^@^@^@^@^@^@^N^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^A^@^@^@^@^@^@^@^@^@^
^@^@N^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^A^@^@^@^@^@^@^@^@^@^@^@^@^@'^
^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^A^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@,^@^@^@^A^@
(^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^A^@^@^@^@^@^@^@^A^@^@^@^@^@^@^@5^@^@^@^A^@^@^@^@^@^@^
^@^@^@^@^@^@^@^@^@^@^@^A^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@J^@^@^@^A^@^@^@^B^@^@^@^@^@
^@^@^@^@^@^@^H^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@E^@^@^@^D^@^@^@@^@^@^@^@^@^@^@^@^@^@^
@^@^@^H^@^@^@^@^@^@^@^X^@^@^@^@^@^@^@^A^@^@^@^B^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@
@^@^@^@^@^@^X^@^@^@^@^@^@^@  ^@^@^@^C^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@<88>^A^@^@
^@^@^@^@^@^@^@^@^@^@^@^Q^@^@^@^C^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@°^A^@^@^@^@^@^@
```

Surprisingly, you cannot read the opcodes as main.o is in binary format (not a text file.) You have to open the file by another C program and print out the binary numbers to see the opcodes. This program is already available and called hexdump:

```
hfani@alpha:~$ hexdump main.o
0000000 457f 464c 0102 0001 0000 0000 0000 0000
0000010 0001 003e 0001 0000 0000 0000 0000 0000
0000020 0000 0000 0000 0000 0208 0000 0000 0000
0000030 0000 0000 0040 0000 0000 0040 000b 000a
0000040 4855 e589 45c7 01fc 0000 9000 c35d 4700
0000050 4343 203a 4428 6265 6169 206e 3031 322e
0000060 312e 362d 2029 3031 322e 312e 3220 3230
0000070 3031 3131 0030 0000 0014 0000 0000 0000
0000080 7a01 0052 7801 0110 0c1b 0807 0190 0000
0000090 001c 0000 001c 0000 0000 0000 000e 0000
00000a0 4100 100e 0286 0d43 4906 070c 0008 0000
00000b0 0000 0000 0000 0000 0000 0000 0000 0000
00000c0 0000 0000 0000 0000 0001 0000 0004 fff1
00000d0 0000 0000 0000 0000 0000 0000 0000 0000
00000e0 0000 0000 0003 0001 0000 0000 0000 0000
00000f0 0000 0000 0000 0000 0000 0000 0003 0002
0000100 0000 0000 0000 0000 0000 0000 0000 0000
0000110 0000 0000 0003 0003 0000 0000 0000 0000
0000120 0000 0000 0000 0000 0000 0000 0003 0005
0000130 0000 0000 0000 0000 0000 0000 0000 0000
```

Now, we're ready to ask the shell to *bootstrap* our program and make it a *process*:

```
hfani@alpha:~$ ./main.o
-bash: ./main.o: Permission denied
```

But as you see, an error is raised from the shell telling the file cannot be executed. This may happen because in UNIX-based/like OS, an executable file should be marked by +x:

```
hfani@alpha:~$ chmod +x main.o
hfani@alpha:~$ ls
Desktop  Documents  Downloads  eclipse-workspace  main.c  main.o  main.s  Music  Pictures
```

As seen, the file is indicated by a different color, meaning that it is an executable file (containing opcodes.) Now, let's run it:

```
hfani@alpha:~$ ./main.o
-bash: ./main.o: cannot execute binary file: Exec format error
```

Surprisingly, you see another error that says the executable file has an incorrect format. This is because the object file has to be linked into the final executable (*ideally, there should be no need for this step. But that's how it is!*) To do so, cc can be called with -o option (small o) followed by a name for the executable file main:

```
hfani@alpha:~$ cc main.o -o main
```

```
hfani@alpha:~$ ls
Desktop  Documents  Downloads  eclipse-workspace  main  main.c  main.o  main.s  Music  Pictures
```

Now, let's run the generated executable file main:

```
hfani@alpha:~$ ./main
hfani@alpha:~$
```

Hooray*!* We could successfully run the opcodes of our program. Since there is no output in our program, it just finishes with no message.

Step 2. Assembly Language

Programming in assembly language (aka. machine language) requires understanding the Instruction Set (IS) of a processor and hardware level (low level) details of how a machine executes a set of instructions, fetch-execute cycle, among other things. Nowadays, most programmers don't deal directly with assembly language unless the task requires direct interfacing with hardware, e.g., to write a device driver or optimize part of a program.

This section will briefly show how optimization can be done via assembly hack to your C program. Let's say we want to increment the variable a by 1 unit in our C program. So,

```c
void main(void){
        int a = 1;
        a = a + 1;
        return;
}
```

And compile it into assembly by cc:

```
hfani@alpha:~$ cc main.c -S
hfani@alpha:~$ vi main.s
```

And see the assembly codes:

```
hfani@alpha:~$ vi main.s
```

```
        .file   "main.c"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    $1, -4(%rbp)
        addl    $1, -4(%rbp)
        nop
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
        .ident  "GCC: (Debian 10.2.1-6) 10.2.1 20210110"
        .section        .note.GNU-stack,"",@progbits
```

As seen, we have the assembly codes for the assignment int `a = 1` and `a = a + 1` as shown below:

```
        movl    $1, -4(%rbp)
        addl    $1, -4(%rbp)
```

Here an optimization can happen. Addition needs to be done in the Arithmetic Logic Unit (ALU) of a processor, and there is no difference between adding with `1` or a large number like `99999`. Usually, it takes 2-3 pulses for a processor to bring in the values in ALU and push the result back. However, almost all processors have an instruction called `incl` (increment long integer) that simply increments a memory unit or a register in-place, i.e., without involving ALU, at 1 pulse*!*

Let's edit the assembly code and replace `addl` with `incl`:

```
        movl    $1, -4(%rbp)
        incl    -4(%rbp)
```

And the remaining steps are the same as before; assembling into opcodes:

```
hfani@alpha:~$ cc main.s -c
main.s: Assembler messages:
main.s:14: Warning: no instruction mnemonic suffix given and no register operands; using default f
```

And make it executable:

```
hfani@alpha:~$ cc main.o -o main
hfani@alpha:~$ ./main
hfani@alpha:~$
```

Step 3. Opcodes (Object File)

Now that we can *hack* assembly codes of our C program, one exciting question will be whether we can hack *opcodes*. Yes, why not! But what is the benefit? We leave it to other courses such as *Advanced System Programming or Computer Architecture*.

Step 4. Linker (Library Routines)

Putting all parts of a program inside one file reduces the legibility and is hard to debug. So, there should be a way to split our code into different files with meaningful names and structures. Also, we should not write codes for doing those tasks that are already written by another programmer. For example, if Hossein already built a program that does matrix multiplication correctly, why not use his program without even knowing his C program.

Let's say we want to write a program that accepts a number and increment it by 1 and print out the result. As seen, there are two distinct tasks: 1) input of a number and output the result (I/O task), 2) increment. We can separate these two tasks into two C files:

The main file accepts an input number from the user and outputs the final result:

```
hfani@alpha:~$ vi main.c
```

```
#include <stdio.h>
int increment(int a);
void main(void){
        int a;
        scanf("%i", &a);
        a = increment(a);
        printf("%i \n", a);
        return;
}
```

Please pay attention to line#2 where we just put function's name increment, input arguments int a, and return type int without the body! This is called function's prototype and gives a hint to the C compiler that although we use this function in line#6, the actual body of the function is somewhere else. Without this line, the compiler raises a warning. Try removing line#2 and compile the file to see the compiler warning.

Next is the file that includes the C lines for incrementing a given number:

```
hfani@alpha:~$ vi increment.c
int increment(int a){
        int b = a;
        b = b + 1;
        return b;
}
```

Both files should be compiled into assembly codes:

```
hfani@alpha:~$ cc main.c -S
hfani@alpha:~$ cc increment.c -S
hfani@alpha:~$ ls
Desktop  Documents  Downloads  eclipse-workspace  increment.c  increment.s  main.c  main.s  Mus
```

Followed by translation to opcodes (object files) by the assembler:

```
hfani@alpha:~$ cc main.s -c
hfani@alpha:~$ cc increment.s -c
hfani@alpha:~$ ls
Desktop    Downloads          increment.c  increment.s  main.o  Music     Public
Documents  eclipse-workspace  increment.o  main.c       main.s  Pictures  Templates
```

Finally, we have to link the two object files into a single executable file:

```
hfani@alpha:~$ cc main.o increment.o -o main
hfani@alpha:~$ ./main
3
4
hfani@alpha:~$
```

As seen, the program waits for an integer. We put 3 and it returns 4. Our program, now, can increment an input integer faster than usual!

Wait*!* Is this linking *static* or *dynamic*? How do you know? Where scanf and printf functions in stdio.h are linked?!

Step 5. Lab Assignment

You should optimize increment.c in Step 4 by changing its assembly code similar to the way we did in Step 2. Then execute it to make sure it outputs the desired output. The sample code without optimization has been attached in a zip file named lab03_hfani.zip.

1.1. Download files *from* Server and Upload files *to* Server

As said above, to download files from the server (e.g., your code and executable file) or to upload a file (e.g., the sample code to the Server), you should use an application that implements a file transfer protocol such as SCP, PSCP, WinSCP, Cyberduck, etc. In case you use SCP, the command is as follows. *Note that the* scp *should be run from your local computer!*

```
scp source:file target:file
```

As seen, SCP can copy from any source to any target. In case of **download**, the source is the server and the target is our local computer:

```
scp hfani@cs.uwindsor.ca:hello.c C:/Users/Administrator/Desktop/hello.c
```

In case of **upload**, the source is our local computer and the target is the server:

```
scp C:/Users/Administrator/Desktop/hello.c hfani@cs.uwindsor.ca:hello.c
```

1.2. Deliverables

You will prepare and submit the program in one single zip file lab03_uwinid.zip containing the following items:

(90%) lab03_uwinid.zip
- (10%) main.c => must be compiled with no error.
- (10%) main.s => must be assembled with no error.
- (10%) main.o
- (10%) increment.c => must be compiled with no error.
- (20%) increment.s => must be optimized and assembled with no error.
- (20%) increment.o => disassembly should show the optimization*!*
- (10%) results.pdf/jpg/png => the image snapshot of the output
- (Optional) readme.txt

(10%) Files Naming and Formats

Please follow the naming convention as you lose marks otherwise. Instead of uwinid, use your own account name, e.g., mine is hfani@uwindsor.ca, so, lab03_hfani.zip