CSci 2021 Fall 2018 Lab 10 – Chapter 3 Follow-up

The purpose of this lab is to do a few applications of the Chapter 3 material. Specifically, you will write assembly code methods and run them from a C program. The easiest way to write assembly methods is to modify an existing assembly method.

Step 1. A Simple One-Parameter Method in Assembly.

Write and compile the following C method with: `gcc -S proc1.c`

```
int proc1(int x, int y) {
     return x;
}
```

This will give you a framework for an assembly method, and it should let you know where the parameters `x` and `y` are in the stack. Look at the `proc1.s` file generated when you compile it with:

```
gcc -S proc1.c
```

It should appear something like this:

```
        .section        __TEXT,__text,regular,pure_instructions
        .macosx_version_min 10, 13
        .globl  _proc1                  ## -- Begin function proc1
        .p2align        4, 0x90
_proc1:                                 ## @proc1
        .cfi_startproc
## %bb.0:
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset %rbp, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register %rbp
        movl    %edi, -4(%rbp)
        movl    %esi, -8(%rbp)
        movl    -4(%rbp), %eax
        popq    %rbp
        retq
        .cfi_endproc
                                        ## -- End function

.subsections_via_symbols
```

For fun, let's just say that we want to return the *sum* of `x` and `y`, instead of just `x`. In C, the body of `proc1` would become:

```
return x + y;
```

Instead of writing a C method to do this, we can modify the assembly code in `proc1.s`. Examining the above assembly code for `proc1`, we can see that `x` (passed in through `%edi`) is at `-4(rbp)` and `y` (passed in through `%esi`) is at `-8(rbp)`. Since our simple method above just returns `x` by placing it into `%eax`, we can add one line just before the `popq`:

```
addl    -8(%rbp), %eax
```

which will add `y` to `x` (already in `%eax`) and put the result (`x + y`) into `%eax`.

To test it, write a simple `main()` driver program like this:

```
#include <stdio.h>
extern int proc1(int x, int y);

int main() {
    int x = 3, y = 5;
    printf("\nValue of %d + %d is %d\n", x, y, proc1(x, y));
}
```

Note that the assembly procedure needs to be listed as "*`extern`*" since we cannot directly include the assembly code in our C program—but we can *call* the assembly method from within our C program if the linker knows that it is defined somewhere. To let the linker know where to look, we include the assembly file name on the compilation command:

```
gcc main.c proc1.s
```

Note that while our method `proc1()` is in a file called `proc1.c`, the name of the file does not have to agree with the name of the method(s) contained in it as long as the C program references the correct method name in the `extern` statement *and* in the call. Similarly, when compiling, the file name of the file containing the method to be used must match the name used on the `gcc` command.

Try it. If it works, you have created an assembly method to add two integers.
When done, have a TA check this step.

Step 2. Summing the Squares in Assembly.

Now, create a *new* file, `proc2.c` that contains a method called `sum_of_squares(int x, int y)`. This method will return the sum of $x^2 + y^2$. The easiest way to do this is to write the *same simple C method we started with in Step 1*, but this time, call it `sum_of_squares` and compile it with `gcc -S`. Then modify the assembly code produced to do the equivalent of this C statement:

```
return x * x + y * y;
```

Call your new assembly method to sum the squares from a `main()` test method using the same approach as in Step 1. Compare the assembly code you just wrote with the assembly code that is generated by the compiler from an equivalent C method.

Explain the differences to your TA when you have this step checked.

Step 3. One More Time.

Let's try one more. This time, write an assembly method to find the *larger* of the two parameters passed to it, and *return the larger*. Use an approach similar to what you did before--starting with the assembly code from the basic compiled C method and modifying it. As you write the assembly code, think about how we have seen `if` statements in assembly before.

Test your assembly method with a `main()` test driver like before. Once you have it working, compare the results with what the C compiler generates from an equivalent C method.

Let your TA know how your assembly code compares with that of the compiler when you have this step checked.

Step 4. A Byte of Hacking.

A computer science student, let's call him Alpha (not his real name), started writing a program in C which calls a simple method called `surprise_add` that takes two integer parameters and returns the sum of them. The program has a short character string in it that is to me used later. The beginnings of the program and method are shown below.

```
#include<stdio.h>

int main() {
    char c[5];
    c[0] = 'a';
    c[1] = 'b';
    c[2] = 'c';
    c[3] = 'd';
    c[4] = '\0';
    int x, y;
    printf("\nYour string c is: %s\n", c);
    printf("\nThe sum of 4 and 5 is: %d\n", surprise_add(4, 5));
    printf("\n");
}

int surprise_add(int x, int y) {
    return x;
}
```

On April 1, a friend of Alpha suggested using an assembly version of the method `surprise_add` saying that it was such a simple method that he could write one that is faster and better than what the C compiler produces. The friend even offered to supply the method. So, Alpha agreed, and he added an `extern` line to his program and compiled by linking in the assembly version of `surprise_add` supplied by his friend:

```
gcc main.c surprise_add.s
```

Everything appeared good. When running the program, the expected output was produced:

```
Your string c is: abcd

The sum of 4 and 5 is: 9
```

Later on, when Alpha started to develop the program and was using the string, c, he found that `c[0]` did not contain `'a'` even though he never changed `c[0]`. So, he decided to print out the string again after the call to `surprise_add`. Here is the modified code and output:

```
#include <stdio.h>
extern int surprise_add(int x, int y);

int main() {
    char c[5];
    c[0] = 'a';
    c[1] = 'b';
    c[2] = 'c';
    c[3] = 'd';
    c[4] = '\0';
    int x, y;
    printf("\nYour string c is: %s\n", c);
    printf("\nThe sum of 4 and 5 is: %d\n", surprise_add(4, 5));
    printf("\nYour string c is: %s\n", c);
    printf("\n");
}
```

The output:

```
Your string c is: abcd

The sum of 4 and 5 is: 9

Your string c is: haha
```

Suspecting a practical joke, Alpha decided to look at the assembly code his friend had provided him. This is what he found:

```
 .section    __TEXT,__text,regular,pure_instructions
 .macosx_version_min 10, 13
 .globl      _surprise_add                ## -- Begin function surprise
 .p2align    4, 0x90
_surprise_add:                               ## @surprise_add
 .cfi_startproc
## %bb.0:
 pushq %rbp
 .cfi_def_cfa_offset 16
 .cfi_offset %rbp, -16
 movq  %rsp, %rbp
 .cfi_def_cfa_register %rbp
 movl  %edi, -4(%rbp)
 movl  %esi, -8(%rbp)
       movq (%rbp), %rax
       movl $1634230632, -5(%rax)
 movl  -5(%rbp), %eax
       movl %esi, %eax
```

```
        addl -4(%rbp), %eax
    popq  %rbp
    retq
    .cfi_endproc
                                        ## -- End function

   .subsections_via_symbols
```

From viewing the assembly code, what is happening that changes the string, `c`, in Alpha's `main` method? Explain how you would modify the assembly method to fix the problem when you have this step checked by a TA.

That's all for this week. Have a great rest of the week!