CSE 2421 Lab 5

Learning Assembly, functions, and the debugger

Due: Thursday 13 July 2023 11:59 PM
Early: 2 bonus points if submitted by Tuesday 11 July 2023 11:59 PM
Late: Friday 14 July 2023 11:59 PM

We'll be using Carmen's timestamps to determine lateness or not. If you are not enrolled, you may send me an email with the zip file. If you are unable to submit by the deadline, you need to send me a screenshot of your finished files with the clock visible and then *still* submit to Carmen when you can.

This lab contains two parts. Make sure you do both of them!

Something you should do before getting started on the lab is creating your readme file. Simply create a file called "readme" and include your name at the top. As you progress through the lab, you may see different parts (labeled things like **Part 1**). When you see this, write it down in your readme. While in a numbered part of a lab, you'll come across various questions, prepended with a bolded Q and a number (such as **Q5**). When you see this, include the Q and the number, along with you answer to the question(s) in the file. A brief example:

> Name: Rob LaTour
>
> I certify that I completed all of the work myself with no aid from anyone aside from the instructor or the undergraduate graders.
>
> Part 1:
>
> Q1: Don't make fun of me, but my favorite color is gray
>
> Q2: The definitive answer is that Han did shoot first. The original version is the one that captured the heart of the public, and thus the director lost creative control. In this essay I will…

Part 1: Encryption

For this lab, we will be implementing a relatively simple (and limited) encryption/decryption scheme, just like in Lab 2! The first part of the lab, just like before, requires you to create a program that encrypts a plaintext file into another file. This time, however, you will be writing all the code in assembly.

If you aren't sure where to start, I'll make the same recommendation as last time: first just try to create something that can, through redirection, read an input file character by character and print those characters to an output file. Next, work towards creating a file that does the same work, but in batches of 8 characters. If the file has fewer than 8 characters, however, your program should still work! Finally, try tackling the encryption scheme pictured below. If you get stuck, consider checking out Part 2 to learn how to debug to help get you unstuck!

NOTE: This time around, I care a whole lot less about modularity and a whole lot more about efficiency. Whereas last time I wanted you guys to use header files to create code that was easily extensible, this time I want to encourage you to create code that's as quick as possible.

Our simple encryption scheme is pictured here:

I_1:	0	a_1	a_2	a_3	a_4	a_5	a_6	a_7
I_2:	0	b_1	b_2	b_3	b_4	b_5	b_6	b_7
I_3:	0	c_1	c_2	c_3	c_4	c_5	c_6	c_7
I_4:	0	d_1	d_2	d_3	d_4	d_5	d_6	d_7
I_5:	0	e_1	e_2	e_3	e_4	e_5	e_6	e_7
I_6:	0	f_1	f_2	f_3	f_4	f_5	f_6	f_7
I_7:	0	g_1	g_2	g_3	g_4	g_5	g_6	g_7
I_8:	0	h_1	h_2	h_3	h_4	h_5	h_6	h_7

Figure 1: Before Encryption

I_1:	h_7	a_1	a_2	a_3	a_4	a_5	a_6	a_7
I_2:	b_1	h_6	b_2	b_3	b_4	b_5	b_6	b_7
I_3:	c_1	c_2	h_5	c_3	c_4	c_5	c_6	c_7
I_4:	d_1	d_2	d_3	h_4	d_4	d_5	d_6	d_7
I_5:	e_1	e_2	e_3	e_4	h_3	e_5	e_6	e_7
I_6:	f_1	f_2	f_3	f_4	f_5	h_2	f_6	f_7
I_7:	g_1	g_2	g_3	g_4	g_5	g_6	h_1	g_7

Figure 2: After Encryption

In Figure 1, we have 8 characters (labeled I sub 1 through 8). Each of the lowercase letters in the boxes represents a bit (a 1 or a 0). At a high level, we are simply taking the bits that make up our 8th character and replacing the most significant bit in each of our other characters with that bit.

Your task is to create an executable file that will read in 8 characters at a time from an input file, encrypt those following this scheme, and then write the encrypted 7 characters to an output file. That process should be repeated until the end of the file is reached. If the number of characters in the file isn't a multiple of 8 (and it likely won't be), your program should simply

transcribe the characters without changing them. This task will heavily make use of bit shifting and bitwise operations!

Create a set of test files to make sure your program is producing reasonable output. You can choose exactly what you want your test files to look like, but you'll need to justify why you created them shortly. There are 2 files I will require you to create, however: One is an input file with exactly 8 characters in it. One is the corresponding "expected output" file. Do the encryption scheme by hand and write out this file. When you test your encryption scheme with the 8-character-long input file, you can then use the Linux command diff to ensure that the files are identical. When you're confident that your program is working, run each of your test files through your program and keep them around. They'll be useful in Part 2!

Q1: Before you look back at your old code for reference. Pull up the code you wrote for Part 3. Can you read and understand what it's supposed to do?

Q2: Based on your answer to Q1, do you think it's important to have sufficiently readable code? Why or why not?

Q3: Describe one key difference in the code that you had to write for Lab 2 vs the code you wrote for this lab.

Q4: Did you choose to create any functions to help with your encryption scheme? If yes, how did you choose which registers to use for local storage? If no, explain why you chose to avoid creating a helper function.

Q5: Run some tests and compare against your C program. Make sure that the output you're producing is the same as what your Lab 2 C program was able to produce!

Q6: Answer this question honestly: does it feel more natural to manipulate bits in C (as you did in Lab 2) or in Assembly (as you're doing now)? Why?

BONUS (5 points): Decryption

For the bonus part of this lab, you're going to create an executable that can decrypt your encrypted text back to plaintext. The procedure will essentially be the inverse of what was done above. The program should read in an input file 7 characters at a time, decrypt those, then repeat until the file ends. Once again, if there are fewer than 7 characters, your program should simply transcribe the characters as written.

If your program is able to read in a full 7 characters, then the most significant bit of each makes up the bits of the 8th character of plaintext. Consult the diagram above if you are confused! Once again, we will have to do a lot of bit shifting and bitwise operations.

You can use the output of Part 1 to test your implementation of this bonus part!

Part 2: Debugging

This part is intended to help you learn how to use gdb to debug assembly code. This will, however, be much more hands-off than the way that we explored the debugger in Lab 1. At this point, I expect you to be an expert in the typical uses of gdb!

As usual, you will need to compile your program with the -g flag in order to be able to run gdb on your executable.

Q1: Try setting a breakpoint on main. Did it work? Why do you think that is?

Q2: Try setting a breakpoint in a slightly different way. Namely, type "break [filename]:[linenumber]" where you replace the bracketed portions with the appropriate text.

Q3: Next, type "tui reg general." Describe what you see.

Q4: On this screen, are you still able to step through the program line-by-line?

Q5: Do the other commands for gdb still work while we're on this screen?