Assignment 6

Objectives:

- Recursion in x86-64 assembly code
- Manipulating 2D arrays (matrices) in x86-64 assembly code
- Experimenting with Stack Randomization
- There are some extra problems at the end of this assignment!

Submission:

- Submit your document called **Assignment_6.pdf**, which must include your answers to all of the questions in Assignment 6.
    - Add your full name and student number at the top of the first page of your document **Assignment_6.pdf**.
    - You will also need to submit your source code file produced in Question 3 on CourSys. More submission details outlined in Question 3.
- Submit your assignment **Assignment_6.pdf** on CourSys.

Due:

- Friday, Nov. 5 at 23:59:59
- Late assignments will receive a grade of 0, but they will be marked (if they are submitted before the solutions are posted on Monday) in order to provide feedback to the student.

Marking scheme:

- Question 1 and Question 2 of this assignment will be marked for correctness.
- Question 3 of this assignment will be marked for completeness.
- The amount of marks for each question is indicated as part of the question.
- A solution will be posted on Monday after the due date.

---

1. [5 marks] Recursion in x86-64 assembly code

In this problem, you are asked to rewrite the `mul` function you wrote in Assisgnment 4. This time, instead of using a loop, you are to use recursion. In doing so, you must use the stack (so we can get some practice), either by pushing/popping or by getting a section of it (e.g., `subq $24, %rsp`) and releasing it (e.g., `addq $24, %rsp`) at the end of your program.

Yes, it is certainly possible to recursively implement `mul` without saving the values of the argument registers (edi and esi) onto the stack. However, for practice's sake, let's save them onto the stack as stated in the question.

Use your files from Assignment 4: `main.c`, `makefile` and `calculator.s`. Then, copy the following and paste it over (replace) your entire `mul` function in `calculator.s`:

```
mul: # performs integer multiplication - when both operands
are non-negative!
# x in edi, y in esi
# You can assume that both parameters are non-negative.
# Requirements:
# - cannot use imul* instruction
#   (or any kind of instruction that multiplies such as mul)
# - you must use recursion (no loop) and the stack
```

Implement this `mul` function satisfying its new requirements (above in green). You must also satisfy the requirements below.

Requirements:

- Your code must be commented such that others (i.e., TA's) can read your code and understand what each instruction does.
  - About comments:
    - Comment of Type 1: Here is an example of a useful comment:
      `cmpl %edx, %r8d # loop while j < N`
    - Comment of Type 2: Here is an example of a **not** so useful comment:
      `cmpl %edx, %r8d # compare %edx with %r8d`

    Do you see the difference? Make sure you write comments of Type 1.
- Make sure you update the header comment block in `calculator.s`.
- You must use the makefile provided in Assignment 4 when compiling your code. This makefile cannot be modified.
- You cannot modify the prototype of the function **`mul`**. The reason is that your code may be tested using a test driver built based on this function prototype.

- Your code **must** compile (using gcc) on our *target machine* and execute on our *target machine*.
- You must follow the x86-64 function call and register saving conventions described in class and in the textbook.
- Do not push/pop registers unless you make use of them in your function `mul` and their content needs to be preseved. Memory accesses are expensive! We'll soon see why!

Submission:

- Oce your new `calculator.s` compiles, executes and has been tested (i.e., it multiplies) on our *target machine*, copy and paste the entire content of your new `calculator.s` in this assignment (yes, it will still have the other functions you implemented in Assignment 4 in it – that is OK!).

---

2. [13 marks] Manipulating 2D arrays (matrices) in x86-64 assembly code

In linear algebra, a matrix is a rectangular grid of numbers. Its dimensions are specified by its number of rows and of columns. This question focuses on the representation and manipulation of square matrices, i.e., where the number of rows and the number of columns both equal **n**.

Here is an example of a square matrix where n = 4:

$$A = \begin{bmatrix} 1 & -2 & 3 & -4 \\ -5 & 6 & -7 & 8 \\ -1 & 2 & -3 & 4 \\ 5 & -6 & 7 & -8 \end{bmatrix} = \begin{bmatrix} A_{00} & A_{01} & A_{02} & A_{03} \\ A_{10} & A_{11} & A_{12} & A_{13} \\ A_{20} & A_{21} & A_{22} & A_{23} \\ A_{30} & A_{31} & A_{32} & A_{33} \end{bmatrix}$$

Note the notation A_{ij} refers to the matrix entry at the i^{th} row and the j^{th} column of A. Each row of the matrix A resembles a one dimensional array in the programming language C, with the value of j increasing for each element. The matrix A has i such rows.

Because of this resemblance, matrices can be represented (modeled) in our C programs using two dimensional arrays. One dimensional arrays are stored in contiguous memory space, where their element 0 is followed by their element 1 which is followed by their element 2, etc… Two-dimensional arrays follow a similar pattern when stored in memory: the one row following the other. In other words, the elements from row 0 are followed by the elements from row 1, which are followed by elements of row 2, and so on. Thus, a two dimensional array, representing a n x n matrix, has n^2 elements, and the base pointer A, contains the address of the first element of the array, i.e., the 0^{th} element of row 0.

Because of this regular pattern, accessing a two dimensional array element can be done in a random fashion, where the address of A_{ij} = A + L (i * n + j), where L is the size (in bytes) of each array element. For example, when L = 1, as it is for this assignment, then the element A_{32} can be found at address A + 1 (3 * 4 + 2) = A + 14.

In this question, you are asked to rotate a matrix 90 degrees clockwise. One way to do this is to first transpose the matrix then to reverse its columns.

Wikipedia says that, in linear algebra, the transpose of a matrix is an operator which flips a matrix over its diagonal, i.e., it switches the row and column indices of the matrix by producing another matrix denoted as A^T. Thank you, Wikipedia.

Here is an example where A^T is the transpose of matrix A (using the diagonal "1, 6, -3, -8"):

$$A = \begin{bmatrix} 1 & -2 & 3 & -4 \\ -5 & 6 & -7 & 8 \\ -1 & 2 & -3 & 4 \\ 5 & -6 & 7 & -8 \end{bmatrix} \qquad A^T = \begin{bmatrix} 1 & -5 & -1 & 5 \\ -2 & 6 & 2 & -6 \\ 3 & -7 & -3 & 7 \\ -4 & 8 & 4 & -8 \end{bmatrix}$$

We reverse the columns of the transpose matrix A^T, by swapping the last column with the first column, the penultimate column with the second column, etc...

Using the same example as above, here is what A^T looks like once it has been reversed. We call this final matrix A':

$$A^T = \begin{bmatrix} 1 & -5 & -1 & 5 \\ -2 & 6 & 2 & -6 \\ 3 & -7 & -3 & 7 \\ -4 & 8 & 4 & -8 \end{bmatrix} \qquad A' = \begin{bmatrix} 5 & -1 & -5 & 1 \\ -6 & 2 & 6 & -2 \\ 7 & -3 & -7 & 3 \\ -8 & 4 & 8 & -4 \end{bmatrix}$$

As you can see, A' is the rotated version of A (A has been rotated by 90 degrees clockwise).

Your task is to implement these two functions in x86-64 assembly code:

```
void transpose(void *, int );
void reverseColumns(void *, int n);
```

When they are called in this order, using a two dimensional array as their first argument, the effect will be to rotate this array by 90 degrees clockwise.

Download `Assn6-Files_Q3.zip`, expand it and open the files (`makefile`, `main.c` and an incomplete `matrix.s`.). Have a look at `main.c` and notice its content. Have a look at `matrix.s`. It contains functions manipulating matrices such as `copy`, `transpose` and

`reverseColumns`. You need to complete the implementation of the functions `transpose` and `reverseColumns`. The function `copy` has already been implemented for you. You may find hand tracing its code useful. You may also want to have a look at the video recording of Lecture 22 Part 1 posted on our course web site in which the code of the copy function `copy.s` is explained. Finally, you may want to "make" the given code and see what it does.

Requirements:

- Your code must be commented such that others (i.e., TA's) can read your code and understand what each instruction does.
  - About comments:
    - Comment of Type 1: Here is an example of a useful comment:
      ```
      cmpl   %edx, %r8d  # loop while j < N
      ```
    - Comment of Type 2: Here is an example of a **not** so useful comment:
      ```
      cmpl   %edx, %r8d  # compare %edx with %r8d
      ```

    Do you see the difference? Make sure you write comments of Type 1.
- You must add a header comment block to the file `matrix.s`. This header comment block must include the filename, the purpose/description of its functions, your name, your student number and the date.
- You must use the makefile provided when compiling your code. This makefile cannot be modified.
- You cannot modify the code that has been supplied to you in the zip file. This signifies that, amongst other things, you must not change the prototype of the functions given. The reason is that these functions may be tested using a test driver built based on these function prototypes.
- Your code **must** compile (using gcc) on our *target machine* and execute on our *target machine*. Your code **must** also solve the problem, i.e., rotate the array by 90 degrees clockwise.
- You must follow the x86-64 function call and register saving conventions described in class and in the textbook.
- Do not push/pop registers unless you make use of them in your functions and their content needs to be preseved.

Submission:

- Electronically submit your file `matrix.s` via CourSys.

- Also copy and paste the entire content of your `matrix.s` in this assignment. Make sure it is the same version of `matrix.s` that compiles and executes on our *target machine.*

---

3. [2 marks] Experimenting with Stack Randomization

We saw inn Lecture 22 that one of the ways to counter buffer overflow attacks was to use compilers (like `gcc` for Linux) that implemented "safety" mechanisms such as Stack Randomization. On way the compiler does this stack randomization is by assigning a different start address to the stack every time a program execxutes. This changing start address to the stack makes it difficult for hackers to predict the location of return addresses on the stack.

Before you proceed with this question, make sure you read Section 3.10.4 in our textbook and review our course lecture notes on this topic.

In this assignment you are asked to confirm or disprove the fact that `gcc` does implement Stack Randomization by running your own experiement as follows:

- Download from our course web site (see Lecture 22) the password program (`password.c`) we saw as part of our Lecture 22 demo.
- Add a statement that prints the memory address of the first byte of the array `password`.
- Execute your program 10 times and record the addresses at which this local variable array is stored on the stack.
- Then state your conclusion and support it with your experiential results.
- Finally, calculate the range of variance in the memory addresses you obtained (if any) over the 10 executions of your program.
- Copy the content of your `password.c` into this assignment (into this Assignment_6.pdf document).

Extra Problems – Not to be submitted – Not for grades, but for practice only!

A. Passing arguments and returning return value

Have a look at Slide 5 of our Lecture 18. As you hand trace the code, comment each line explaining what each instruction does and why?

Here is an example:

```
pushq %r13  # main function needs to use registers for temporary storage so it saves the
            original value of the callee saved register %r13 so it can make use of it.
```

B. Recursion in x86-64 assembly code

Have a look at Slide 6 of our Lecture 20.

1. Hand trace the C code of the function `countOnesR(…)` and figure out what it does. Feel free to use the second test case on Slide 13 (of our Lecture 20) as you are hand tracing the code.
2. Hand trace its assembly code and as you do so, draw its stack diagram on Slide 14. Feel free to use the Register Table, if you find it useful.
3. Create a third test case following the same format as the format used on Slide 13.
