Homework 3

Learning Outcomes

After completion of this assignment, you should be able to:

- Manipulate 2D Arrays.
- Write Functions and manage the call stack.
- Perform low-level File I/O Operations.

Getting Started

To complete this homework assignment, you will need the MARS simulator. Download it from Blackboard. You can write your programs in the MARS editor itself. You can choose to use other text editors if you are not comfortable with the MARS editor. At any point, if you need to refer to instructions click on the *Help* tab in the MARS simulator.

Read the rest of the document carefully. This document describes everything that you will need to correctly implement the homework and submit the code for testing.

You should have already setup Git and configured it to work with SSH. If you haven't then do Homework 0 first!

The first thing you need to do is download or clone this repository to your local system. Use the following command:

```
$ git clone <ssh-link>
```

After you clone, you will see a directory of the form *cse220-hw3-username*, where *username* is your GitHub username.

In this directory, you will find *hw3.asm*. This file has function stubs that you will need to fill up. At the top of the file you will find hints to fill your full name, NetID, and SBU ID. Please fill them up accurately. This information will be used to collect your scores from GitHub. If you do not provide this information, your submission may not be graded. The directory also has a template test file ending with *hw3_test.asm*. Use the file for preliminary testing. You can change the data section or the text section in this files to test different cases for each part (described later). You may also create your own *_test.asm* files if necessary. Don't push these additional *_test.asm* files to the repository. The *tests* directory contain the test cases for this homework. You can use the test cases as specifications to guide your code. Your goal should be to pass all the tests. If you do so, then you are almost guaranteed to get full credit. The files in the *tests* directory should not be modified. If you do, you will receive no credit for the homework.

Note the hw3.asm file doent have a .data section. Do not add a .data section.

Don't forget to add you name and IDs at the top of hw3.asm. Follow the exact format, i.e, replace the hints with the correct information. You will be penalized if you do not follow the format.

Assembling and Running Your Program in MARS

To execute your MIPS programs in MARS, you will first have to assemble the program. Click on the *assemble* option in the *Run* tab at the top of the editor. If the instructions in your program are correctly specified, the MARS assembler will load the program into memory. You can then run the program by selecting the *Go* option in the same *Run* tab. To debug your program, add breakpoints. This is done after assembling the program. Select the *execute* tab, you will see the instructions in your program. Each instruction will have a checkbox associated with it. Clicking on the checkbox will add a breakpoint, that is, when the program is run, control will stop at that instruction allowing you to inspect the registers and memory up to that point. The execute tab will show you the memory layout in the bottom pane. The right hand pane shows the list of registers and their values.

Always assume that memory and registers will have garbage data. When using memory or registers, it is your responsibility to initialize it correctly before using it. You can enable the *Garbage Data* option under *Settings* in MARS to run your programs with garbage data in memory.

How to read the test cases

As mentioned previously, the *tests* folder contains the test file *Hw3Test.java*. Each test is a Java function with a name prefixed with *verify_*. In each of these functions, you will find an assert statement that needs to be true for the test to pass. These asserts compare the expected result with the actual result returned by the function under test. The tests assume that the expected results will be in certain registers or labels as explained in the later parts of this README. In each test, we will execute your program by calling the *run* function along with the necessary arguments required to test that particular feature. If a test fails, the name of the failing test will be reported with an error message. You can use the error message from the failing test to diagnose and fix your errors. Another way is to look at the inputs of the failed test case and plug them into appropriate labels in the *_test.asm* file/s and run it to debug your program.

Do not change any files in the tests directory. If you do then you won't receive any credit for this homework assignment.

File I/O in MIPS

To open a file in MIPS, we use the *syscall 13*. This syscall takes the address of a null-terminated string in *$a0*. This string indicates the filename. In *$a1*, we provide the mode in which the file will be opened. This should be 0 for reading and 1 for writing. The register *$a2* is ignored and should be 0. This syscall returns a file descriptor in the register *$v0*. The file descriptor is a pointer to the file we want to open. *$v0* is negative if an error occurs while opening the file. Once we have the file descriptor in *$v0*, we can now read/write the file depending on the flag we used to open it.

To read a file, we use the *syscall 14*. It takes the file descriptor in *$a0*, the address of an input buffer in *$a1*, and the maximum no. of characters to read in *$a2*. It returns the no. of characters read in *$v0*. It returns 0 in *$v0* if reading from end-of-file and a negative number if an error occurs.

To write to a file, we use the *syscall 15*. It takes the file descriptor in *$a0*, the address of an input buffer in *$a1*, and the no. of characters to write in *$a2*. It returns the no. of characters written in *$v0*. It returns a negative number if an error occurs.

See the *Help* section in MARS for more information.

Problem Specification

Suppose we are given a directory of files, which we need to process in a certain way. These files have integer matrices stored in them in the following format:

- The first line indicates the no. of rows in a 2D matrix. It must be an integer [1-9].
- The second line indicates the no. of columns in a 2D matrix. It must be an integer [1-9].
- The following lines represent the integers in the matrix. Each element in the matrix must be an integer [0-9].
- Each line represents a row in the matrix.

Here is an example file:

```
2
3
123
456
```

This file indicates a 2D matrix with 2 rows and 3 columns. Each line after the first two lines indicates a row in the matrix.

We will parse such files and store them in a data structure `Buffer`:

```
struct Buffer {
  int no_of_rows
  int no_of_cols
  int[81] matrix
}
```

We will use this data structure to perform certain operations as defined below.

Part 1 -- Initialize Data Structure

int initialize(char* filename, Buffer* buffer)

This function takes two arguments -- a string *filename* and the address of a data structure *buffer* (as defined above). The function will read the file content in the file *filename*, parse it and store the contents in *buffer*. The contents in *buffer* should be in the format defined for the data structure above. So, the first two elements should be the no. of rows and columns of a matrix and the remaining elements should be the integers of the matrix. One way to do this is to read the file one character at a time, and store them in the buffer as integers in appropriate positions.

If the file is read without any errors and the buffer is initialized properly, then the function should return 1 in *$v0*.

The function should return -1 in *$v0* if an error occurs during initialization. When this happens the *buffer* data structure should remain unchanged. You should assume the buffer contains all zeros before initialization. Apart from file reading errors, initialization errors can happen if the file is not in the defined format, i.e., the no. of rows and columns must be the characters [1-9] and the elements in the matrix must be the characters [0-9].

Note that this function must handle newline characters in both Windows and UNIX-based systems. On Windows, line endings are terminated with a combination of carriage return (|*r*) and newline (|*n*) characters, also referred to as CR/LF. On UNIX-based systems, line endings are just the newline character (|*n*).

Part 2 -- Write Buffer To File

void write_file(char* filename, Buffer* buffer)

This function takes two arguments -- a string filename and the address of the buffer data structure (as defined above). It should write the data in *buffer* to the file in *filename*. The format of the file is the same as defined previously. Here is an example:

```
2
3
123
456
```

for

```
struct Buffer {
  int no_of_rows = 2
  int no_of_cols = 3
  int[81] matrix = [1,2,3,4,5,6,...]
}
```

Remember to close the file after writing. Failure to close a file may lead to memory leaks, which degrades performance in the long run.

You may assume that the matrix space provided will be greater than or equal to the size defined, and that buffer's fields contain valid integers in range.

In the test files you may observe the matrix being given a much larger space than defined in no_of_rows/cols, this is fine and your method should only be concerned with the scope defined in `matrix[0] to matrix[ (no_of_rows x no_of_cols) ]`

The function returns nothing.

Part 3 -- Rotate Clockwise By 90

void rotate_clkws_90(Buffer* buffer, char* filename)

This function takes two arguments -- the address of the *buffer* data structure and a string *filename*. It rotates the matrix in *buffer* clockwise by 90 degrees and writes it to *filename*. For example, consider the following matrix with 2 rows and 3 columns:

```
1 2 3
4 5 6
```

Rotating this matrix clockwise by 90 degrees will result in the following matrix with 3 rows and 2 columns:

```
4 1
5 2
6 3
```

This new matrix after rotation should be written to the file *filename* as follows:

```
3
2
41
52
63
```

Notice how the first two lines in the file have no. of rows = 3 and no. of columns = 2 as the rotation switched the original no. of rows and columns.

Assume that buffer points to a valid Buffer struct.

The function returns nothing.

Part 4 -- Rotate Clockwise By 180

void rotate_clkws_180(Buffer* buffer, char* filename)

This function takes two arguments -- the address of the *buffer* data structure and a string *filename*. It rotates the matrix in *buffer* clockwise by 180 degrees and writes it to *filename*. For example, consider the following matrix with 2 rows and 3 columns:

```
1 2 3
4 5 6
```

Rotating this matrix clockwise by 180 degrees will result in the following matrix with 2 rows and 3 columns:

```
6 5 4
3 2 1
```

This new matrix after rotation should be written to the file *filename* as follows:

```
2
3
654
321
```

Notice how the first two lines in the file have the same no. of rows columns the original matrix.

Assume that buffer points to a valid Buffer struct.

The function returns nothing.

Part 5 -- Rotate Clockwise By 270

void rotate_clkws_270(Buffer* buffer, char* filename)

This function takes two arguments -- the address of the *buffer* data structure and a string *filename*. It rotates the matrix in *buffer* clockwise by 270 degrees and writes it to *filename*. For example, consider the following matrix with 2 rows and 3 columns:

```
1 2 3
4 5 6
```

Rotating this matrix clockwise by 270 degrees will result in the following matrix with 3 rows and 2 columns:

```
3 6
2 5
1 4
```

This new matrix after rotation should be written to the file *filename* as follows:

```
3
2
36
25
14
```

Similar to the 90 degree rotation, the no. of rows and columns will be switched.

Assume that buffer points to a valid Buffer struct.

The function returns nothing.

Part 6 -- Mirroring

void mirror(Buffer* buffer, char* filename)

This function takes two arguments -- the address of the *buffer* data structure and a string *filename*. It creates a mirror of the matrix in *buffer* writes it to *filename*. For example, consider the following matrix with 2 rows and 3 columns:

```
1 2 3
4 5 6
```

Mirroring the matrix will result in the following matrix with 3 rows and 2 columns:

```
3 2 1
6 5 4
```

This new matrix after mirroring should be written to the file *filename* as follows:

```
2
3
321
654
```

The no. of rows and columns (as indicated by the first two lines) will remain the same in the mirror file.

Assume that buffer points to a valid Buffer struct.

The function returns nothing.

Part 7 -- Duplicates

(int, int) duplicate(Buffer* buffer)

This function takes the data structure *buffer* as an argument. Assume that the matrix in *buffer* contains only binary values 0 and 1. The function checks to see if the matrix has any duplicate rows. If a duplicate row exists in the matrix then the function returns 1 in *$v0* and the index (starting at 1) of the first duplicate row in *$v1*. If the matrix has no duplicate rows then the function returns -1 in *$v0* and 0 in *$v1*

For example, consider the following matrix with 3 rows and 5 columns:

```
10010
01100
10010
```

The 3rd row in this matrix is a duplicate of the first row. Hence, the function should return 1 in *$v0* and 3 in *$v1*. See test cases for more clarity.

You may also assume that the row and col encoded in buffer are valid.

Submitting Code to GitHub

You can submit code to your GitHub repository as many times as you want till the deadline. After the deadline, any code you try to submit will be rejected. To submit a file to the remote repository, you first need to add it to the local git repository in your system, that is, directory where you cloned the remote repository initially. Use following commands from your terminal:

```
$ cd /path/to/cse220-hw3-<username>
```

(skip if you are already in this directory)

```
$ git add hw3.asm
```

To submit your work to the remote GitHub repository, you will need to commit the file (with a message) and push the file to the repository. Use the following commands:

```
$ git commit -m "<your-custom-message>"
```

```
$ git push
```

Every time you push code to the GitHub remote repository, the test cases in the *tests* folder will run and you will see either a green tick or a red cross in your repository just like you saw with homework0. Green tick indicates all tests passed. Red cross indicates some tests failed. Click on the red cross and open up the report to view which tests failed. Diagnose and fix the failed tests and push to the remote repository again. Repeat till all tests pass or you run out of time!

After you submit your code on GitHub. Enter your GitHub username in the Blackboard homework assignment and click on Submit. This will help us find your submission on GitHub.

Running Test Cases Locally

It may be convenient to run the test cases locally before pushing to the remote repository. To run a test locally use the following command:

```
$ java -jar munit.jar tests/Hw3Test.class hw3.asm
```

Remember to set java in your classpath. Your test cases may fail if you do not have the right setup. If you do not have the right setup it is most likely because you did not do homework 0 correctly. So, do homework 0 first and then come back here!