COMP 273

Assignment 3
School of Computer Science
McGill University
Available On: February 26th, 2019
Due Date: March 14th, 2019. 11:59pm.
Submit your solution in electronic form using MyCourses
Read the submission instructions at the end of the document
(late policy: 10 marks off per day late, up to 2 days late. 0 After that.)
Sharing code is strictly prohibited.
YOU WILL BE GRADED ON WHAT YOU
SUBMIT IN YOUR ASSIGNMENT FOLDER AND
IT IS YOUR RESPONSIBILITY TO CHECK THAT
THE FILES YOU INTENDED TO SUBMIT ARE
ACTUALLY IN THAT FOLDER!
Make sure to follow format instructions carefully!

Things to consider

- Welcome to Assignment 3! We will be going through some fun image manipulation exercises in this one, developing some tools for your image processing toolbox. You may notice that FILE I/O will receive no partial marks. The reason for this is that every following question depends on FILE I/O functioning.
- Make sure to follow conventions! beyond just readability and consistency between developers, the conventions also help developers avoid hours of debugging. If you have questions about conventions/debugging, feel free to ask any of the TAs.
- Even if your code does not produce the right image, make it produce something!
- You will receive a 0 on the question if the code does not compile.
- We will be using our own input files during grading, but the general structure is pretty straight forward.
- **Do not change any labels in the templates**. You may add whatever extra labels you may need, but use the ones provided as intended.
- Make sure you read all the questions first. Though some things you can get away with by hard coding, this isn't the case for all questions. Certain things (like moving through a 2D matrix) are necessary in multiple questions, for example, and avoiding hard coding might save time in the long run!
- File I/O is the only part of the assignment that requires error checking.
  -MARS uses relative paths to find images. This means, if you have Mars in the same folder as your .txt and scripts, then the paths specified in the templates will work accordingly.
- GIMP is an image editing software. Though you can use anything you'd like, its software that can be used to view the .pgm file after generating the image is complete.
  https://www.gimp.org/
- headers for .pgm files have a specific format.
  $P2$(which is grayscale)

width height
(max value)
http://netpbm.sourceforge.net/doc/pgm.html contains more information about proper .pgm formatting. We're concerned with PLAIN pgm, so we will assume that there are no comment lines in the pgm file.

- You may use as many sub routines as you like! I.E. They must not be called from the main function.
- You may add any extra .data you like, but make sure its after the specified line. The data specified in the templates should be used accordingly.

1 FILE I/O (15 marks) No partial marks

(10 marks) The template for this question is provided in fileio.asm

(a) Using the template *fileIO.asm* as a starting point, write a MIPS procedure *readfile* that takes as its argument the address of a string that contains a valid filename, and then uses appropriate syscalls to read from that input file and then simply prints the content of that file to the screen. In the template there are two input files called test1.txt and test2.txt which you should test. To accomplish this task we allow you to create a large buffer, i.e., one that is larger than the expected number of ASCII characters (bytes) in the input file. The body of your code should work by calling the procedure you have written. The procedure should open the file, read its content as ASCII characters, store the content in the buffer, print the content to the screen, and then close the file. For your reference a more complete set of MIPS syscalls implemented in MARS, along with clear documentation on how to use each, is here: http://courses.missouristate.edu/kenvollmar/mars/help/syscallhelp.html.

(b) We shall now build on the above example. Following the process of reading the file *test1.txt*, after which the ASCII characters have been read into the buffer, we shall call a second MIPS procedure called *writefile*. This procedure should open a file called *copy.pgm*, should then write the following information to that file:
P2
24 7
15
Then it should write out the content that was read into the buffer. It should then close the file.

***(5 marks)Error statements should be printed if there are any errors in opening the file.

If things are working properly when you view *copy.pgm* with a suitable image viewer, e.g., gimp, you should see something interesting.

2 Flip Image (15 marks)

1. Template: flipper.asm
2. Read the data in *test1.txt* into the specified buffer.
3. The data read into the buffer should now be converted to consecutive integers and then stored in a 2D array of length 24×7, i.e., one that has 24 columns and 7 rows. However that 2D array will actually be represented as a 1D array*. Finally, take care to convert the entries in the buffer (which are in ASCII) to their numerical values in base 10.
4. Write a procedure named $flip$.
5. Argument structure outlined in template.

* Normally, in a language like Java, we would simply specify the respective array positions of i and j. That is, we would let i represent the row we are currently at, and j represent the column we are currently at. In MIPS, however, our 2D array is stored as values in a 1D array. It is clear to see that for any position [i,j] in our 2D array, we can retrieve this position by simple computing

(i * width) + j. Since i represents rows, whenever we add a width (for this assignment, width is 24) we are essentially going to the next row in our conceptual 2D array. j simply represents which column we are currently looking at.

3 Transpose Image (15 marks)

1. The template for this question is provided in *transpose.asm*.
2. Read the data in *test1.txt* into the specified buffer.
3. The data read into the buffer should now be converted to consecutive integers and then stored in a 2D array of length 24×7, i.e., one that has 24 columns and 7 rows. However that 2D array will actually be represented as a 1D array*. Finally, take care to convert the entries in the buffer (which are in ASCII) to their numerical values in base 10.
4. Write a procedure named *transpose*. This routine will take your image buffer, and apply the matrix operation transpose to it (where position (i,j) in a matrix is now at position (j,i).
5. The content of the output 2D array should then be written into a file named *transposed.pgm*.
6. **The header will be changed! update your writefile routine to work accordingly.**
   (a) THIS CAN ONLY BE VIEWED if you have properly implemented (a). This means that you are expected to write the information as you did in 1(b) and THEN write the completed output 2D array to *transposed.pgm*.
   (b) TESTING: You can assume that all .txt files we may use for this question use the same header as specified in b).

4 Crop Image (30 marks)

1. Template: *cropper.asm*
2. You are given $x1, y1, x2, y2$, and a *test.txt* file.
3. Read the data in *test1.txt* into the specified buffer.
4. Generate a new .pgm named *cropped.pgm* that contains a cropped version of the image information contained in *test.txt*, i.e., reading it into an array and then cropping the rectangular portion bounded by $x1, y1, x2, y2$.
5. **Make sure that the cropped.pgm has the correct header!** Otherwise, the cropped image won't be displayed properly.
6. Make sure to use P2 for "plain pgm" and 15 for the max intensity (last term from the header in part 1 b). This means the only values that change are those specifying the dimensions of your image. Therefore, the header must not be hard coded for this question. This extension should be added to your write function.
7. TESTING: you can expect that all test.txt files that are tested will start with the same header as part 1, and that x1,y1,x2,y2 will be valid inputs, i.e., they will be within the bounds of the original array.

5 Add Image Border (25 marks)

1. Template: border.asm
2. Read the data in *test1.txt* into the specified buffer.
3. In a routine called *bord*, given *borderWidth* (a pixel width), take your *test.txt* image and produce a new *borded.pgm* image with a borderwidth equal to the pixels.
4. The border will be white (=15). You can assume P2 and 15 here as well.

5. TESTING: you can expect that all test.txt files that are tested will start with the same header as part 1.

6 Assignment Submission Instructions

1. the **edited** template files: *fileio.asm ,flipper.asm, transposer.asm, cropper.asm* and *border.asm* must contain a main function that produces the images *copy.pgm, flipped.pgm, transposed.pgm, cropped.pgm and borded.pgm* (as specified in the appropriate sections). Do **NOT** include the images, as we will be running the code with our own .txt files.

2. Submit your solution to myCourses before the due date.

3. Highlight the FOUR template files (which you have completed) and zip them. The zipped file should be named <studentID>.zip. If my student ID is 123456789, then the zip file to submit will be named *123456789.zip*.

4. If you have special comments about your code, feel free to include a *confessions.txt* file in your zip containing your specific comments. Otherwise, simply comment your code as you normally would. Partial marks are easier to give (though no certainty that any will be given) if we see that you understand where your code is going wrong.

5. Your code *must* run and assemble, even if the final solution is not quite working. Part marks will be awarded for correct high-level control flow and use of conventions. If something is not working, comment-out the broken parts of code and write a comment or two about what you expect to happen, what is happening, and what the problem may be. **You are expected to follow conventions. Doing so will save you time and a headache during debugging, so its great for everyone!** *If your code does not assemble you will receive 0 points for that question.*
penalties will be given if the grader is required to change more than just the input values (This means your main method must run, producing whatever it can produce, without having to rewrite the main function.)