CISC372 – Parallel Programming

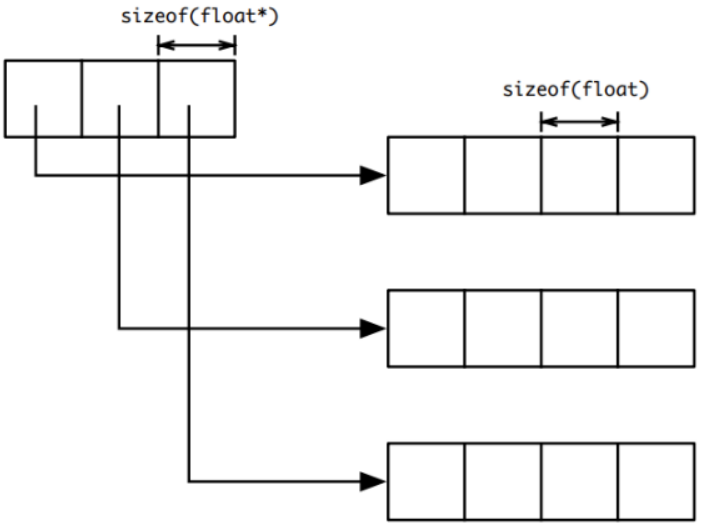
Assignment 1: Remembering c

Overview:

In this assignment, we will be working with the c language to write a serial (not parallel) program to re-familiarize ourselves with c programs, and compiling them via the unix operating system using gcc. You are to build a simple C program that implements a matrix (2d array) in two different ways, and then prints the matrix transposed (i.e. rows become columns, columns become rows). You should implement this as 2 functions (1 for each storage method) with a main function that simply calls the two functions that will do the work.

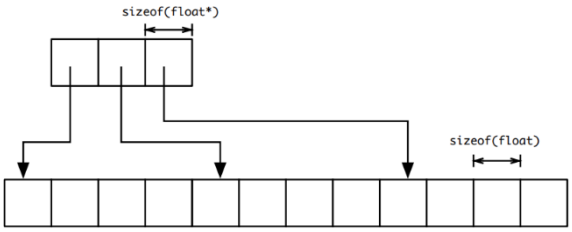
Details:

1) Construct a function void matrixArrayOfArrays(int n, int m); which constructs an n x m 2d array in c implemented as an array of pointers to arrays. The array of pointers of type float* as well as the sub-arrays of floats of type float should be dynamically allocated using malloc.



a. Here you will have an array of float* variables that you allocated with malloc of size m* sizeof(float*), then for each of these pointers, you will allocate an array of size n * sizeof(float)
b. Fill the array with the numbers 1-n*m so that the array is filled across the first row 1,2,3,4,n and the second row n+1,n+2,…,2n etc.
c. Print the array out so that it looks like a matrix on the screen (you can use \t as a tab character in your printf statements to line things up nicely).
d. Print the array out transposed (i.e. first column then second column, etc.
e. Deallocate all memory allocated with malloc so there are no memory leaks
f. Return from the function.

2) Construct a function void matrixOneBigArray(int n, int m) which constructs an n x m 2d array in c implemented as an array of pointers, and one large contiguous block of memory to store the array elements. The array of pointers of type float* as well as the large array of type float should be dynamically allocated using malloc.



a. Here you will have an array of float* variables that you allocated with malloc of size m* sizeof(float*), and a second array of type float allocated as n * m * sizeof(float).

b. Next you will look through the array of pointers setting each pointer to the beginning of a row in the array (i.e. first pointer to the beginning of large array of floats, second pointer to the n+1 element of the array, etc.

c. Fill the array with the numbers 1-n*m so that the array is filled across the first row 1,2,3,4,n and the second row n+1,n+2,…,2n etc.

d. Print the array out so that it looks like a matrix on the screen (you can use \t as a tab character in your printf statements to line things up nicely).

e. Print the array out transposed (i.e. first column then second column, etc.

f. Deallocate all memory allocated with malloc so there are no memory leaks

g. Return from the function.

3) Use the following main program to access and call your functions:

```
#define N 5
#define M 6

int main(int argc,char** argv){
   matrixArrayOfArrays(N,M);
   matrixOneBigArray(N,M);
   return 0;
}
```

4) Compile and test your program on a unix system using gcc. Verify that your program is not leaking memory. You should consider using valgrind (https://www.valgrind.org/docs/) to catch and help debug memory leaks in your program. Valgrind is installed on silber-courses.cis.udel.edu as well as cisc372.cis.udel.edu for your use and you can install it on any system you have root access to using apt.

5) Answer the following:

a. What are the pros and cons of each method of allocation?

b. Can you think of a situation where one of these would be better than the other?

NOTE: Here is an example of what your printouts should look like from each function:

```
1.000000        2.000000        3.000000        4.000000        5.000000
6.000000        7.000000        8.000000        9.000000        10.000000
11.000000       12.000000       13.000000       14.000000       15.000000
16.000000       17.000000       18.000000       19.000000       20.000000
21.000000       22.000000       23.000000       24.000000       25.000000
26.000000       27.000000       28.000000       29.000000       30.000000
```

```
1.000000        6.000000        11.000000       16.000000       21.000000       26.000000
2.000000        7.000000        12.000000       17.000000       22.000000       27.000000
3.000000        8.000000        13.000000       18.000000       23.000000       28.000000
4.000000        9.000000        14.000000       19.000000       24.000000       29.000000
5.000000        10.000000       15.000000       20.000000       25.000000       30.000000
```