Programming Assignment Multi-Threading 1 and Debugging 1

Due Date:  **Friday, May 18 @ 11:59pm**

**Overview:**
The purpose of this mini-assignment is to briefly introduce you to parallel programming and multi-threading; covering aspects of implementation, benefits and the tradeoffs involved. Provided with skeleton code, you will fill in the functionality where necessary to initialize an array with decoded values, calculate the sum of the squares, min, and max of the values in the array, and calculate the square root of the values, min, and max of the values in the array -- first with sequential algorithms and then using OpenMP directives to parallelize portions of the algorithms. The program will also record and display the execution time of the initializations and calculations, allowing you to see the relative performance of performing these algorithms sequentially and parallelized.

NOTE: Due to the CPU-intensive nature of this assignment, we are developing and running the multi-threaded programming assignment (PAMT1) on the workstations in the labs or ieng6 only! See the section on Getting the Code for more details on how to set up the pamt1 project on the workstations.

You will also be given a chance to practice your debugging skills. We've written a program that can encrypt and decrypt files, using two different kinds of ciphers. We think it's close to working, but we didn't have time to debug it. It's your job to help us track down the bugs and fix it up so we can get it to properly encrypt/decrypt files. **The debugging exercise will be completed on pi-cluster.**

**Important Points:**
1) **PAMT1** will be developed and executed and turned in with your cs30x account on the workstations in the lab (preferred) or ieng6.ucsd.edu remotely.

2) **Debug1** will be debugged and executed and turned in with your cs30x account on pi-cluster.ucsd.edu.

**Grading**

**PAMT1**
- **README:** 10 points - See PAMT1 README File section
- **Compiling:** 10 points - Using our Makefile; no warnings. If what you turn in does not compile with the given Makefile, you will receive 0 points for this assignment. **NO EXCEPTIONS!**
- **Correctness:** 40 points

**Debug1**
- **README:** 30 points - See DEBUG README File section
- **Correctness:** 10 points

NOTE: If what you turn in does not compile with given Makefile, you will receive 0 points for this assignment.
PAMT1 Overview

For this assignment will we explore using OpenMP, an API that supports multi-platform shared memory multi-threading with minimal effort. Serial code can be easily made parallel by the addition of added directives (OpenMP pragmas) that will instruct the compiler to generate parallelized code. This program will use a 12-byte encoded pattern that needs to be decoded to initialize an array of a user-specified length. Once the array is initialized correctly, the program will perform two modified sum calculations and find the min and max values in the array. For each step of the project, we will perform it two different ways: 1) using a serial implementation and 2) with an OpenMP parallelized version. We will then compare the runtime of each version.

You will write three algorithms: `initData`, `squaredSumMinMax`, and `sqrtSumMinMax`. For each of these algorithms, you will implement a serial version and an OpenMP parallelized version. Each of the parallelized versions will be identical to the corresponding serial version, but with added OpenMP pragmas. The project's `main()` is already written, the only modification that is required is commenting out five return statements (marked with TODOs) as you implement each of the algorithms. Each of these functions have already been stubbed out and function header comments provided, but the body must be added for correct functionality. You will also need to provide your personal info in the headers.

References
==========

https://en.wikipedia.org/wiki/OpenMP
- In particular, under "OpenMP clause" - pretty good intro to default sharing.
- Under "Data Sharing attribute clauses" - the explanation for "shared" and "private" are pretty good. Farther down the "Reduction" header (below "Data copying") is helpful.

- First page/middle column - Parallel Loop shows syntax for #pragma omp parallel for [clause] and Simple Parallel Loop Example.
- Second page under "Clauses" - reduction table of operators (in our case: +, min, and max).

http://openmp.org/

Detailed Overview

The function prototypes for the C functions

```c
int main( int argc, char* argv[] );
void initData( unsigned char a[], size_t arraySize,
               unsigned char data[], size_t dataSize );
void parallel_initData( unsigned char a[], size_t arraySize,
                       unsigned char data[], size_t dataSize );
struct result squaredSumMinMax( unsigned char a[], size_t arraySize );
struct result parallel_squaredSumMinMax( unsigned char a[], size_t arraySize );
struct result1 sqrtSumMinMax( unsigned char a[], size_t arraySize );
struct result1 parallel_sqrtSumMinMax( unsigned char a[], size_t arraySize );
```

The definitions for `struct result` and `struct result1` are in the `pamt1.h` header file.
Getting the Code for PAMT1

Create a pamt1 directory in your cs30x class account on a workstation in the basement or on ieng6 (NOT in your cs30x class account while ssh'ed into pi-cluster!).

```
[cs30xzzz@ieng6-203]:~:420$ mkdir pamt1
```

Copy all the files in ~/.public/pamt1StarterFiles to your pamt1 directory.

```
[cs30xzzz@ieng6-203]:~:421$ cp ~/.public/pamt1StarterFiles/* ~/pamt1/
```

The files in your pamt1 directory should be:

- Makefile
- pamt1.h
- parallel_squaredSumMinMax.c
- initData.c
- parallel_initData.c
- sqrtSumMinMax.c
- main.c
- parallel_sqrtSumMinMax.c
- squaredSumMinMax.c

1. `initData.c`

```c
void initData( unsigned char a[], size_t arraySize, 
               unsigned char data[], size_t dataSize );
```

This function will fill the array "a" of arraySize elements using the decoding algorithm listed below (and in the comments in the initData.c file) and using the values in the "data" array of dataSize elements.

Steps:

1. Implement the decoding algorithm.
   a) Be sure to use the constants defined in the pamt1.h header file to avoid use of magic numbers (BITS_IN_A_NIBBLE, BIT_7, BITS_IN_A_BYTE, etc.).

   **Decoding Algorithm:**
   1) Define 4 unsigned char variables b1, b2, b3, b4;
   2) For size_t i from zero thru all the elements in array a (where is of type size_t)
      3a) Set a[i] to data[i modulus dataSize]
      3b) Set b4 to zero
      3c) For int j from zero up to BITS_IN_A_NIBBLE (exclusive)
         4a) Set b1 to a[i] ANDed with (BIT_7 right shifted by j) bits
         4b) Set b1 to b1 right shifted by (BITS_IN_A_BYTE - 1 - j) bits
         4c) Set b2 to a[i] ANDed with (BIT_0 left shifted by j) bits
         4d) Set b2 to b2 right shifted by j bits
         4e) Set b3 to b1 XORed with b2
         4f) Set b3 to b3 left shifted by j bits
         4g) Set b4 to b4 ORed with b3
   3d) Set a[i] to (a[i] ANDed with UPPER_NIBBLE) ORed with b4

2. Run 'make' in your pamt1 directory.
3. Run the program using './pamt1' followed by an array size (you can try multiple sizes to compare run-times).
   a) The program will check that the first 12 decoded bytes in the "a" array match the correct pattern.
4. Once the function works correctly comment out the "return 0;" line with the TODO labeled "After initData (serial)" in main.c.
2. parallel_initData.c

```c
void parallel_initData( unsigned char a[], size_t arraySize,
                        unsigned char data[], size_t dataSize );
```

This function works identically to the serial version, but uses OpenMP pragmas to execute in parallel.

Steps:
1. Copy your working algorithm/code from the serial version of `initData()` (initData.c) into parallel_initData.c.
2. Add an OpenMP pragma to parallelize the outer for loop (iterating over the arraySize).
   a) Adding OpenMP pragma.
      In OpenMP every pragma begins with \\
      By default, all variables in the work sharing region are \\
      In our case, the outer for loop block is our work sharing region.
3. Try re-compiling (make) and run the program. What happens?
   a) You may notice that you are not getting the correct results for the parallel `initData`. This is most likely 
      because of the scope of the unsigned char variables b1, b2, b3 and b4. As stated in the wikipedia article on 
      OpenMP (https://en.wikipedia.org/wiki/OpenMP): "By default, all variables in the work sharing region are 
      shared except the loop iteration counter." In our case, the outer for loop block is our work sharing region.
4. Set the data scope attribute of the local variables.
   a) In OpenMP, if a variable is defined inside a parallel region, then a unique copy of the variable will be used 
      for each thread; this variable would be considered "private" to each thread. However, if a variable is defined 
      outside of the region, it will be shared among all of the threads; this variable would be considered "shared".
   b) The local variables b1, b2, b3, and b4 are defined outside/above the outer for loop we are trying to 
      parallelize, so by default they are shared between all the threads. The parallel threads will continuously 
      overwrite each other's shared copies of b1, b2, b3, and b4. Not good and not what we want.
   c) The solution (keeping the local variables defined at the top of the function above the for loop) is to add a 
      "data scope attribute clause" to specify the scope. In our case we will add the "private" clause (e.g. "private(b1, 
      b2, b3, b4)"). This indicates we want separate private copies of these variables for each thread of execution vs. 
      having them all shared among all the threads.
5. Your OpenMP pragma should now look like:
#pragma omp parallel for private(b1, b2, b3, b4)

Note: If you defined the inner loop counter "j" as a local variable above the outer for loop, you will need to add "j" to the list of variables in the private clause. If you defined it as int j = 0 in the inner for loop construct, then by default it is private in each outer loop work sharing region and you are fine.

6. Try re-compiling (make) and run the program. You should get a correct output for the parallel version that matches the serial version now. Use different array sizes to see if you can get the initData Speed-up to exceed 1.0, indicating that the parallel version ran faster than the serial version. You should see similar speed-up results to the example output, but they will probably not be exactly the same due to many factors such as other users/processes contending for time on the cores on the system you are running on.

7. Once the function works correctly comment out the "return 0;" line with the TODO labeled "After initData (parallel)" in main.c.

3. squaredSumMinMax.c

struct result squaredSumMinMax( unsigned char a[], size_t size );

This function will sum the squares of each element (square an element in the array and then add that value to an accumulating sum) in "a" and find the min and max values in the array.

The function returns a struct result. This struct has three members, unsigned long long int sum, unsigned int min, and unsigned int max. See pamt1.h.

Steps:
1. Create three local variables: sum, min and max with the same type as their correspondingly named struct result members. Use these during your calculations and set/assign them into a struct result right before returning.

2. Initialize the local variables using the first element of the array (initialize sum to a[0] * a[0], min and max to a[0]).

3. Iterate over all of the remaining elements in the array (starting with the second element).
   a) Add the squared value of each element into the sum. Make sure to manually compute the square (instead of using the "pow" function, which returns a double).
   b) Check each element to see if it is a new min or max and update the min and max variable accordingly.

4. Set the elements of a local struct result variable using the local variables from your calculations and return the struct.

5. Try re-compiling (make) and run the program. Check your output against the provided example outputs for correctness.

6. Once the function works correctly comment out the "return 0;" line with the TODO labeled "After squaredSumMinMax (serial)" in main.c.
4. parallel_squaredSumMinMax.c

struct result parallel_squaredSumMinMax( unsigned char a[], size_t size );

This function works identically to the serial version, but uses OpenMP pragmas to execute in parallel.

Steps:
1. Copy your working algorithm/code from the serial version of squaredSumMinMax() in squaredSumMinMax.c into parallel_squaredSumMinMax.c.

2. Add an OpenMP pragma to parallelize the for loop.
   a) Adding "parallel for" constructs.
      Once again we will use the "parallel" and "for" constructs to parallelize the loop. Recompile the program and run. Compare the values of Squared Sum, Min, and Max between the sequential version and the parallel version of squaredSumMinMax. Hey, the Squared Sum values are different! Remember - by default, these local variables are being shared (except for the loop iteration counter variable).
   b) Adding the "reduction" clause.
      Now how will we define the data scope attribute of the sum, min and max variables? We do not want each thread overwriting the same shared variables; resulting in incorrect output. We also do not want to use a critical section (or atomics) to have all of the threads correctly share the variables because this will dramatically slow the program. We also do not want these variables to be strictly private to each thread (like we did for the local variables in parallel_initData.c) because in the end we want to combine each thread's local sum, min and max into a single overall sum, min, and max (this is called a reduction).
      What we want is for each thread to have a private copy of each local variable during execution in each thread and then combine all of the threads' individual results in a shared value once they complete. To do this we will use the "reduction" clause. This clause will create a private copy of a variable for each thread and then combine the values in a shared variable using a specified operator once the threads finish (note that the private variables are initialized to a specific value based on the operator). Make sure that you are using local variables in the reduction clause, it will not work with struct members (e.g. "reduction(+:result.sum)" will not work).
      1. For the "sum" we want to use "reduction(+:sum)".
      2. For the "min" we want to use "reduction(min:min)".
      3. For the "max" we want to use "reduction(max:max)".

3. Your OpenMP pragma should now look like:

   #pragma omp parallel for reduction(+:sum) reduction(min:min) reduction(max:max)

   or to put them on separate lines, use the backslash to escape the newline:

   #pragma omp parallel for reduction(+:sum) \ 
       reduction(min:min) \ 
       reduction(max:max)

4. Try re-compiling and run the program. Make sure the parallel version's sum, min, and max values match those of the sequential version. Use different array sizes to see if you can get the squaredSumMinMax Speed-up to exceed 1.0, indicating that the parallel version ran faster. You should see similar speed-up results to the example output.
5. Once the function works correctly comment out the "return 0;" line with the TODO labeled "After squaredSumMinMax (parallel)" in main.c.

5. **sqrtSumMinMax.c**

```c
struct result1 sqrtSumMinMax( unsigned char a[], size_t size );
```

This function will sum the square root of each element in "a" (call sqrt() on each element in the array and then add that value to an accumulating sum) and find the min and max values.

We will use the math library's "sqrt()" function for our calculations. See "man sqrt" for more information. Since this function returns a double, a separate "struct result1" with a double "sum" member is used as the return type.

Steps:
1. This function behaves almost exactly like squaredSumMinMax(). The only changes required are changing the data type of the "sum" variable and using sqrt() instead of squaring the array elements.
2. Try re-compiling and run the program. Check your output against the provided example outputs for correctness.
3. Once the function works correctly comment out the "return 0;" line with the TODO labeled "After sqrtSumMinMax (serial)" in main.c.
5. \texttt{parallel\_sqrtSumMinMax.c}

\begin{verbatim}
struct result1 parallel_sqrtSumMinMax( unsigned char a[], size_t size );
\end{verbatim}

This function works identically to the serial version, but uses OpenMP pragmas to execute in parallel.

Steps:
1. Using what you have learned about OpenMP, try adding the required pragma to parallelize this function.
2. Try re-compiling and run the program. Note that we are summing up double precision floating point values. Addition of floating point values is not associate, so we most likely will get different rounding errors between the sequential version and the parallel version - especially for large array sizes. But they should match that of the sample \texttt{pamt1test}. Use different array sizes to see if you can get the speed-up to exceed 1.0, indicating that the parallel version ran faster. You should see similar speed-up results to the example output.

There is a sample executable called \texttt{pamt1test} in \texttt{~/../public} you can run. For example:

\begin{verbatim}
~/.public/pamt1test 40000000
\end{verbatim}

\textbf{Example Output}

\begin{verbatim}
[cs30xyz@ieng6]:pamt1:1$ ~/.public/pamt1 40000
Using array size = 40000
Initializing array with values using sequential initData()
    [be patient with large values]
Checking that sequential initData produced the values as expected
Checking the first 12 bytes only
Should print out the string: "CSE30 Rocks!"
CSE30 Rocks!
Sequential initData time = 0.001554

Initializing array with same values using parallel initData()
    [be less patient with large values]
Num of threads = 8
Checking that parallel initData produced the values as expected
Checking the first 12 bytes only
Should print out the string: "CSE30 Rocks!"
CSE30 Rocks!
Parallel initData time = 0.004342

*** initData Speed-up: 0.357939 ***

Sequential squared sum, min, max [be patient]
Sequential squaredSumMinMax time = 0.000191

Squared Sum is: 255583181
Min value is: 32
Max value is: 115
Completed in 0.000191 sec

Parallel squared sum, min, max [don't need to be as patient]
\end{verbatim}
Parallel squaredSumMinMax time = 0.003774

Squared Sum is: 255583181
Min value is: 32
Max value is: 115
Completed in 0.003774 sec

*** squaredSumMinMax Speed-up: 0.050535 ***

Sequential sqrt sum, min, max [be patient]
Sequential sqrtSumMinMax time = 0.000590

Sqrt Sum is: 338939.708947
Min value is: 32
Max value is: 115
Completed in 0.000590 sec

Parallel sqrt sum, min, max [don't need to be as patient]
Parallel sqrtSumMinMax time = 0.002722

Sqrt Sum is: 338939.708947
Min value is: 32
Max value is: 115
Completed in 0.002722 sec

*** sqrtSumMinMax Speed-up: 0.216885 ***

------------------------------------------------------------------
[cs30xyz@ieng6]:pamt1:2$ ./pamt1 40000000
Using array size = 40000000
Initializing array with values using sequential initData()
    [be patient with large values]

Checking that sequential initData produced the values as expected
Checking the first 12 bytes only
Should print out the string: "CSE30 Rocks!"
CSE30 Rocks!

Sequential initData time = 1.508953

Initializing array with same values using parallel initData()
    [be less patient with large values]
Num of threads = 8

Checking that parallel initData produced the values as expected
Checking the first 12 bytes only
Should print out the string: "CSE30 Rocks!"
CSE30 Rocks!

Parallel initData time = 0.295013

*** initData Speed-up: 5.114865 ***

Sequential squared sum, min, max [be patient]
Sequential squaredSumMinMax time = 0.173058
Squared Sum is: 255589993181
Min value is: 32
Max value is: 115
Completed in 0.173058 sec

Parallel squared sum, min, max [don't need to be as patient]
Parallel squaredSumMinMax time = 0.041531

Squared Sum is: 255589993181
Min value is: 32
Max value is: 115
Completed in 0.041531 sec

*** squaredSumMinMax Speed-up: 4.166964 ***

Sequential sqrt sum, min, max [be patient]
Sequential sqrtSumMinMax time = 0.586355

Sqrt Sum is: 338940858.017723
Min value is: 32
Max value is: 115
Completed in 0.586355 sec

Parallel sqrt sum, min, max [don't need to be as patient]
Parallel sqrtSumMinMax time = 0.117479

Sqrt Sum is: 338940858.042362
Min value is: 32
Max value is: 115
Completed in 0.117479 sec

*** sqrtSumMinMax Speed-up: 4.991132 ***

------------------------------------------------------------------

[cs30xyz@ieng6]:pamt1:3$
./pamt1 400000000
Using array size = 400000000
Initializing array with values using sequential initData()
    [be patient with large values]
Checking that sequential initData produced the values as expected
Checking the first 12 bytes only
Should print out the string: "CSE30 Rocks!"
CSE30 Rocks!

Sequential initData time = 15.239569

Initializing array with same values using parallel initData()
    [be less patient with large values]
Num of threads = 8
Checking that parallel initData produced the values as expected
Checking the first 12 bytes only
Should print out the string: "CSE30 Rocks!"
CSE30 Rocks!
Parallel initData time = 2.649712

*** initData Speed-up: 5.751406 ***

Sequential squared sum, min, max [be patient]
Sequential squaredSumMinMax time = 1.733613

Squared Sum is: 2555899993181
Min value is: 32
Max value is: 115
Completed in 1.733613 sec

Parallel squared sum, min, max [don't need to be as patient]
Parallel squaredSumMinMax time = 0.365531

Squared Sum is: 2555899993181
Min value is: 32
Max value is: 115
Completed in 0.365531 sec

*** squaredSumMinMax Speed-up: 4.742720 ***

Sequential sqrt sum, min, max [be patient]
Sequential sqrtSumMinMax time = 5.892237

Sqrt Sum is: 3389408590.849287
Min value is: 32
Max value is: 115
Completed in 5.892237 sec

Parallel sqrt sum, min, max [don't need to be as patient]
Parallel sqrtSumMinMax time = 1.100185

Sqrt Sum is: 3389408590.272765
Min value is: 32
Max value is: 115
Completed in 1.100185 sec

*** sqrtSumMinMax Speed-up: 5.355679 ***

**PAMT 1 README File**

Along with your source code, you will be turning in a README (use all caps and no file extension for example, the file is README and not README.txt) file with every assignment. Use vi/vim to edit this file!

Your README file for this and all assignments should contain:
- Header with your name, cs30x login
- High level description of what your program does
- How to compile it (usually just typing "make")
- How to run it (give an example)
- An example of normal output and where that normal output goes (stdout or a file or ???)
- An example of abnormal/error output and where that error output goes (stderr usually)
- How you tested your program
- Anything else that you would want/need to communicate with someone who has not read the writeup
Questions to Answer for the README

The command `lscpu` on the workstations or ieng6 displays information about the CPU(s) on that system. Running lscpu on one of the workstations in B240 shows there is a single socket (chip) with 4 cores per socket and 2 threads per core for a total of 8 logical CPUs. The 2 threads per core indicates the cores are hyper-threaded - sharing 2 threads of execution on a single physical CPU. Running lscpu on ieng6.ucsd.edu (say, ieng6-201.ucsd.edu) shows they are configured with 8 sockets (chips) with 1 core per socket and 1 thread per core for a total of 8 logical CPUs. These are not hyper-threaded cores, so each core does not share its CPU resources with multiple threads.

Run `pamt1` with a large `array_size` (say, 400000000) on both ieng6 and one of the workstations in B240. Note: Your speed-ups will vary, be sure to run the program multiple times in each environment.

1. Which system (ieng6 or B240 workstation) shows the larger speed-up due to parallelization of the 3 algorithms?

2. Which system, ieng6 or B240 workstation, runs faster (the different sequential and parallel completed times are smaller)?

3. Why do you think this is the case?

Note: Please answer the following questions for both ieng6 and the B240 workstation.

4. About what `array_size` values will result in the `initData` Speed-up to exceed 1.0? What about exceeding 2.0?

5. About what `array_size` values will result in the `squaredSumMinMax` Speed-up to exceed 1.0? What about exceeding 2.0?

6. About what `array_size` values will result in the `sqrtSumMinMax` Speed-up to exceed 1.0? What about exceeding 2.0?

Add an OpenMP parallel for pragma to also parallelize the inner for loop in `parallel_initData.c`.

7. How does this change the parallel speed-up? Does it make it faster or slower than not parallelizing the inner for loop? You can always comment out the omp parallel for pragma above the inner for loop to recompile and test with and without the inner for loop being parallelized.

8. Why do you think it is faster or slower?

BE SURE TO REMOVE OR COMMENT OUT THIS "EXPERIMENT" OF PARALLELIZING THE INNER LOOP BEFORE TURNING IN YOUR PAMT1!!!
Debugging Exercise Overview

Do this debugging exercise on pi-cluster.ucsd.edu
Note: The purpose of this assignment is to give you practice in identifying and developing solutions to bugs. Because of this, the tutors have been instructed not to help you find the bugs in this assignment. Tickets asking for help identifying or fixing bugs will be removed from the queue.

The purpose of this program is to encrypt or decrypt an input stream, stdin, with a specified cipher algorithm, print out the resulting encryption or decryption to stdout, and finally print the number of bytes written to stdout to stderr. The main() function is written in C, and it will encrypt or decrypt an input stream with help from several assembly functions.

We provide all of the code to you, but the code doesn’t quite compile or work as it should. It is up to you to track down the bugs and fix them. There are a total of 7 bugs in the source code. You are required to record ALL of the bugs and the solution for each bug. See the section on Debug README File for more details.

C routines
int main( int argc, char * argv[] );
char getKeyByte( byteArray_t * key, int idx );
void vigenereDec( byteArray_t * out, byteArray_t * key, byteArray_t * in );
void xorDec( byteArray_t * out, byteArray_t * key, byteArray_t * in );

Assembly routines
char myRand( void );
void vigenereEnc( byteArray_t * out, byteArray_t * key, byteArray_t * in );
void xorEnc( byteArray_t * out, byteArray_t * key, byteArray_t * in );

Getting the Code for Debugging Exercise
For debug1 (the debugging exercise), you will develop on pi-cluster as you do for most programming assignments. However, we will provide you with the buggy code. Simply go to your home directory and copy the whole folder to your home directory:

$ cd ~
$ cp -r ~/../public/debug1 .

This will provide you with all of the buggy source code. All you have to do is fix the code and detail in a README file exactly what fixes you had to make to get this code to work properly. Include:

- Effects of the bug. What signaled to you that this bug exists? This can be error messages for compiler errors, weird behavior for runtime errors, etc.
- The line number(s) of the fix(es). (These may change as you fix more bugs, so it's fine if this isn't exact)
- What the line(s) looked like before and after your fix(es).
- A short explanation (1-2 sentences is fine) of your reasoning behind each fix and how you debugged the problem.

Keep in mind that each bug fix should only modify at most two (2) contiguous lines of code. If you find yourself thinking of a fix spanning more than two lines, either the bug is not in those lines, or your fix can be written more concisely.
**Debugging Exercise Example Output**

This program will read in from stdin and write out an encrypted or decrypted version of the input, depending on the user-specified command-line argument, to stdout. A key string will be used to encrypt or decrypt, if the user specifies one. If not, a "random" (but really it's the same sequence of bytes on each run, since the same seed is being used) key will be used. There are two Cipher algorithms: Vigenere (0) and XOR (1). The details of how these algorithms work can be found below in the function descriptions in the next section.

The program takes several arguments from the command line:

```
$ ./crypt [--help] {d|e} {0,1} [key]
  --help   (optional) Display the usage string.
  {d,e}   (required) Decipher(d) or encipher(e).
  {0,1}   (required) Cipher type:
           0 - Vigenere
           1 - XOR
  key     (optional) String used to encrypt data.
```

Since the program is reading from stdin, if you do not redirect a file to stdin, the program will "hang", because it is waiting for input. So the following example inputs and outputs will include only tests that redirect files to stdin (all of our tests will be written this way anyway). We have provided you with input files to test with in the files directory. movie is the original file, movie_0.enc is the resulting encrypted file using the 0 Cipher (Vigenère), movie_0_secretkey.enc is the resulting encrypted file using the 0 Cipher with key "secretkey", etc.

You should test your output with the reference executable, ~/../public/debug1test, in case you have any doubts.

Below are a few examples of error input/output (bold indicates user input):

```
[cs30xyz@pi-cluster-042]:debug1$ ./crypt x 0 < files/movie > encfile
   Not a valid encrypt or decrypt flag.
[cs30xyz@pi-cluster-042]:debug1$

[cs30xyz@pi-cluster-042]:debug1$ ./crypt e 1a < files/movie > encfile
   Cipher_Index: Invalid int '1a'
[cs30xyz@pi-cluster-042]:debug1$
```

Encrypting a file and decrypting the resulting encrypted file sfould produce a file with the same contents as the original file. You can use the command `cmp (man cmp)` to check that the file contents are equal. To illustrate:

```
[cs30xyz@pi-cluster-042]:debug1$ ./crypt e 0 < files/movie > encfile
   Wrote 49474 bytes to stdout.
[cs30xyz@pi-cluster-042]:debug1$ ./crypt d 0 < encfile > output
```
Wrote 49474 bytes to stdout.

[cs30xyz@pi-cluster-042]:debug1$  cmp files/movie output
[cs30xyz@pi-cluster-042]:debug1$

[cs30xyz@pi-cluster-042]:debug1$  ./crypt e 1 < files/movie > encfile

Wrote 49474 bytes to stdout.

[cs30xyz@pi-cluster-042]:debug1$  cmp files/movie_1.enc encfile

[cs30xyz@pi-cluster-042]:debug1$

---

**Debugging C Modules**

**main.c**

```c
int main( int argc, char *argv[] );
```

The driver of this program. Parses command line arguments and encrypts/decrypts based on the supplied arguments. Produces error messages if the supplied arguments are invalid. If arguments are valid, will take input from stdin and write the encrypted output to stdout. Finally, it will print the number of bytes written to stdout to stderr.

**Return Value:** EXIT_SUCCESS on success, EXIT_FAILURE on failure.

**getKeyByte.c**

```c
char getKeyByte( byteArray_t * key, int idx );
```

This function tries to get a byte from the key array, at index idx. If the index is past the end of the array, the function will wrap the index to be within the array size. If the byte at idx is nul ('\0'), `getKeyByte()` will return a random byte instead.

Note that this function is given to you for reference, and should not be modified.

**Return Value:** The byte in key at idx, or a random byte if the key’s byte is '\0'.

**vigenereDec.c**

```c
void vigenereDec( byteArray_t * out, byteArray_t * key, byteArray_t * in );
```

This function is the inverse of `vigenereEnc()` (see in the next section). It decrypts the in parameter using the key passed to it, and writes the result to the out parameter. Decryption is done by subtracting each key byte from each byte of the input, since this is the inverse operation from encrypting with a Vigenère cipher. In other words, we decrypt using the following formula for every position $i$ in the input (to get original message $M$, from key $K$, and from encrypted input $C$):

$$M[i] = C[i] - K[i]$$
Note that this function is given to you for reference, and should not be modified.

**Return Value:** None.

---

**xorDec.c**

```c
void xorDec( byteArray_t * out, byteArray_t * key, byteArray_t * in );
```

This function is the inverse of `xorEnc()` (see in the next section). It decrypts the `in` parameter using the `key` passed to it, and writes the original message to the `out` parameter. Decryption is done by XOR-ing each key byte with each byte of the input, since this is the opposite operation from encrypting in a XOR cipher. In other words, we decrypt using the following formula for every position `i` in the input (to get original message `M`, from key `K`, and from encrypted input `C`):

```
M[i] = C[i] ^ K[i]
```

Note that this function is given to you for reference, and should not be modified.

**Return Value:** None.

---

### Debugging Assembly Modules

---

**myRand.s**

```c
char myRand( void );
```

This function generates a random byte value, using a simple algorithm involving bitwise shifts and OR operations. It makes use of a seed (`RNG_SEED`) to deterministically generate the same sequence random values. The module also defines a static variable which it uses to preserve the state of the random number generation, so that multiple calls to this function will yield different return values.

**Return Value:** A random byte value.

---

**vigenereEnc.s**

```c
void vigenereEnc( byteArray_t * out, byteArray_t * key, byteArray_t * in );
```

This function encrypts the `in` parameter using the `key` by adding key values to the input, pair by pair. This is because the general idea of a Vigenère cipher is that we encrypt by the following formula (for output `C`, key `K`, and input `M`):

```
C[i] = K[i] + M[i]
```

Note that if the key is shorter than the message, we repeat the key as many times as necessary. If while iterating, the byte in key is zero (`\0`), we instead choose a random byte to add. This logic is implemented in the `getKeyByte()` function.

**Return Value:** None.

---

**xorEnc.s**

```c
void xorEnc( byteArray_t * out, byteArray_t * key, byteArray_t * in );
```
This function encrypts the `in` parameter using the `key` by XOR-ing key values to the input, pair by pair. This is because the general idea of an XOR cipher is that we encrypt by the following formula (for output `C`, key `K`, and input `M`):

\[ C[i] = K[i] \oplus M[i] \]

Note that if the key is shorter than the message, we repeat the key as many times as necessary. If while iterating, the byte in key is zero (`\'\0\'`), we instead choose a random byte to add. This logic is implemented in the `getKeyByte()` function.

Return Value: None.

---

**isInRange.s**

```c
long isInRange( long num, long min, long max );
```

This is just your `isInRange()` from PA1.

Reasons for error:
- `minRange` is greater than or equal to `maxRange` → return -1

Note that this function is given to you for reference, and should not be modified.

Return Value: -1 if an error occurred. Otherwise, return 1 if value is in range and 0 if not in range.

---

**Debugging 1 README File**

For the debugging assignment only, you do not have to include the usual README sections like high level description, how tested, etc. You will, however, have to document each of the compilation errors you encountered, as well as runtime errors. Some runtime errors will be obvious (for example, `Segmentation fault`), but some will involve a little more testing and debugging.

Again, for each bug fix, include:
- **Effects** of the bug. What signaled to you that this bug exists? This can be error messages for compiler errors, weird behavior for runtime errors, etc.
- The **line number(s)** of the fix(es). (These may change as you fix more bugs, so it's fine if this isn't exact)
- **What the line(s) looked like** before and after your fix(es).
- A **short explanation** (1-2 sentences is fine) of your reasoning behind each fix and how you debugged the problem.

As a guideline, there should be 3 compilation errors and 4 runtime/functionality problems. Make sure you locate all of them! (Note: When we say there are 3 compilation errors, we mean that there are three fixes you'll have to make, not that there are exactly three errors that are printed to the screen).

**Turnin Instructions**

Complete Turnin - due Friday night, May 18 @ 11:59 pm

Once you have checked your output, compiled, executed your code, and finished your README file (see above), you are ready to turn it in. Before you turn in your assignment, you should do `make clean` in order to remove all the object files, core dumps, and executables.
How to Turn in an Assignment
First, you need to have all the relevant files in a subdirectory of your home directory.

PAMT1
`~/.public/bin/cse30turnin pamt1`
In your cs30x account on the lab workstations or ieng6. You will not be able to turn-in pamt1 on pi-cluster.

<table>
<thead>
<tr>
<th>Files required for PAMT1 Final Turn-in:</th>
</tr>
</thead>
<tbody>
<tr>
<td>initData.c</td>
</tr>
<tr>
<td>main.c</td>
</tr>
<tr>
<td>pamt1.h</td>
</tr>
</tbody>
</table>

DEBUG1
`~/.public/bin/cse30turnin debug1`
In your cs30x account on pi-cluster. You will not be able to turn-in debug1 on ieng6.

<table>
<thead>
<tr>
<th>Files required for Debug1 Final Turn-in:</th>
</tr>
</thead>
<tbody>
<tr>
<td>debug1Globals.c</td>
</tr>
<tr>
<td>debug1.h</td>
</tr>
<tr>
<td>debug1Strings.h</td>
</tr>
<tr>
<td>getKeyByte.c</td>
</tr>
</tbody>
</table>