XOR Encryption Utility

# Requirements:

1. Program accepts a key file and number of threads in the following format

encryptUtil [-n #] [-k keyfile]

-n #        Number of

threads to create

-k keyfile    Path to file containing

key

1. Reads data from stdin.
2. Writes encrypted data to stdout.
3. Writes errors to stderr.
4. Input data can be large, far exceeding the available memory+swap space for the system.
5. Program allows the user to specify number of worker threads.
6. Program should be able to process multiple blocks of plaintext(input) in parallel.
7. Output cypher text should be the same regardless of the number of worker threads used.
8. The program should compile and run on a UNIX system.

# Design decisions:

1. PThreads : Decided to use pthreads to implement the parallel version of the encryption utility. PThreads provide a standard interface that should allow portability. But I realized that not all versions of UNIX based/ UNIX-like operating systems support Pthread barriers. So decided not to use that specific API.
2. Generate all keys ahead of time: Given that the input size can be very large, rotating the key for every XOR block operation can be expensive, so decided to generate all combinations of keys (key chain) ahead of time, and index into the key chain for every XOR block operation.

Read data in batches of thread\_count x block\_size : given that the input data can be very large, it is better to read the data in batches and process them. Partitioning the data also lends itself well if the solution ever needs to be modified to technologies like OpenCL to use multiple processing units.

1. The program validates the input parameters, and provides the user help over stderr.
2. Pipelining: Given that this program has two IO operations, 1> input, 2>output. It seemed like pipelining would be a good solution.

We read batches of data of size (thread\_count x block\_size) in the main thread, and we send it to a dispatcher which gets a LOCAL COPY of the input data which it can process. The main thread is freed to read more data, while the dispatcher processes the input plain text from the local copy.

The dispatcher spawns N threads which actually do the XOR operation, they indexing into the keychain and generate their partial outputs. When all the workers have generated their partial outputs, they signal the dispatcher, which writes the data to stdout.

All this happens while the main thread prepares the next batch of data to be processed.

The approach looked logical, but back fired ☹ , from a performance point of view (atleast for this specific problem), because the reading from stdin and stdout is fairly fast, we are not able to see the benefit of using the pipeline. It looks like the data movement and the synchronization overhead is causing the program to run slower than a single threaded program, which is also included as a solution I am submitting (also uses pre-generated keys).

# Source Code

Source code can be found under the “source” directory. The directory contains two sub-directories.

1. encryptUtil\_pthread : contains source files for the pthread – pipelining based solution
2. encryptUtil\_singleThread : single threaded solution
3. loadGen: load generation utility that repeatedly writes a test pattern to stdout

# Build Commands

encryptUtil\_pthread : gcc –pthread pt.c pthread\_encrypt.c -o encryptUtil

encryptUtil\_singleThread : gcc single\_thread\_encrypt.c –o encryptUtil\_1\_thread

Each individual folder also contains executable that were generated on a MAC.

# Test Cases

The test cases that were tried have been described in test\_cases.txt under the test\_cases directory. The same directory contains the input ( plaintext and the key files ) that were used for testing )