CSCI 4061: Introduction to Operating Systems (Spring 2014)
Programming Assignment 4 (100 pts.)

Released: Mar 27, 2014 Due: Apr 11, 2014

Welcome to CSCI4061. Hope you will have fun cracking this assignment!

1 Pre-requisites

It is assumed that you have **good** knowledge of C. Defining good: you can write, compile, run and debug C code. The purpose of this homework is to get you started with thread programming, so familiarity with POSIX threads is expected.

2 Introduction

We are going to implement a data server, simple program that can be queried (by clients) to retrieve data. Data stored by server is encrypted to protect against unauthorized use. So when a client requests particular data, server decrypts it before sending.

Generally, servers serve multiple concurrent clients. Forking new processes is not a cheap operation, so multi-**threaded** servers are common. Actual client-server programs communicate over networks, but we do **NOT** consider network programming here. However, this assignment asks you to implement the server and **emulate** client requests.

3 Problem

Your task is to implement a program that is called from the terminal as follows:

```
./retrieve clients.txt output num_threads

The executable file must be named retrieve. Arguments:

   clients.txt (required): path of the input file. clients.txt is an example,
   name can be different. Assume that name provided will be valid. Each line
   in this file is a file path.

   output (required): path of output directory. All output files will be
   written in this directory. It can be assumed that this dir is already
   present.

   num_threads (Optional): Maximum number of threads the server should be able
     to handle concurrently. If not specified, value should default to 5.
```

Note:

1. At all places following this line, words *client* and *file path* are used interchangeably. This is because we are emulating clients, and for the purpose of assignment client is basically a file path. More details ahead.
2. Shared queue or queue are used interchangably and mean the same.

retrieve main thread should read a text file called *clients.txt* which contains one client per line. Each client is represented as a file path which contains data to be decrypted. *retrieve* should insert the clients in a **shared queue**. It should then create *num_threads* threads in addition to its main thread. Each child thread should read a client at a time from the queue and decrypt the client's file. The decryption algorithm is discussed in next section. After handling a client, a child thread should get another client to handle and so on until the queue is empty. The shared queue size is unbounded (unless you implement the extra credit), so all clients can be inserted in this queue once, regardless of the number of children threads. The *retrieve* main thread must wait for all child threads to finish processing before exiting. You need to use **synchronization** mechanisms to manage the access to this shared queue between clients.

The flow of execution is summarized below:

Main thread:

1. Read *clients.txt* line by line and put these clients (or file paths) into the shared queue.
2. Create *num_threads*.
3. Wait for all the threads to finish before exiting.

Child thread (this flow is for each child thread created by main thread):

1. Child thread should read a client from the shared queue. This client path along with thread id should be written to a log file named *log.txt* which should be created in the *output* directory. The format of the message should be: "file_path: thread_id" strictly. Please note that many threads will write to this file simultaneously so you should **synchronize** your writes. Client should then be deleted from the shared queue: you need to make sure one client is read by only one child.
2. File at the client path read should be decrypted and output should be written to a new file in *output* directory (specified at command-line). Path of this output file should be original_file_name.out. For example: If the input client path read is /home/user/d.txt then output path should be *output*/d.txt.out.
3. If decryption is done, repeat above steps (1 - 3) until shared queue is empty.
4. Exit.

3.1 Decryption Algorithm

Each character in the file should be replaced by a character which is greater by two in terms of ASCII value. Example:

```
Input: adyzL
Output: cfabN
```

Only English alphabets should be decrypted. This includes: [a-z] and [A-Z]. All other characters should be output as they are. Please note that *y, Y* and *z, Z* are special cases which are replaced/encrypted with *a, A* and *b, B* respectively.

3.2 Extra Credit

To get extra credit, *retrieve* should allow a second command line argument *queue_size* (the extra credit option **requires** the *num_threads* argument to be specified if *queue_size* is specified). The *retrieve* main thread should use this parameter as the maximum size of the shared queue. If the number of clients is larger than the size of the buffer, then the main thread must initially fill the queue with *queue_size* clients. Then, it will wait for an empty space in the queue and insert each remaining path into the queue as slots become free. Whenever the main thread must wait to add paths due to a full queue, it should print a message in *log.txt*: "INFO: Queue full. Waiting to add more clients."

The flow of Main thread will change to:

1. Read *clients.txt* line by line and put *queue_size* clients (or file paths) into the shared queue.
2. Create *num_threads*.
3. Read *clients.txt* line by line for remaining clients and keep putting them into the shared queue until there are no more clients to read. **Catch here is to synchronize this write operation with child threads which will simultaneously read from the queue**. Whenever the queue is full print the message as specified above and wait until space is available.
4. Wait for all the threads to finish before exiting.

In the bounded queue problem, it is difficult for child threads to know when to exit (queue can be empty because main thread is slow in putting clients). Please see section 16.6 in Text book for a solution to a similar problem. You will have to consider this case, else your thread(s) may wait infinite.

Example invocation:

```
./retrieve clients.txt output num_threads queue_size

All the above arguments are compulsary.
```

Note: If you are implementing extra credit, you do not have to separately implement unbounded queue functionality. Extra credit functionality if successfully implemented guarantees synchronization between child threads which is the motive of this assignment (without extra credit). If *num_threads* and *queue_size* arguments are not specified on terminal, they should default to 5 and 7 respectively.

Please make sure to make a note in README if you are implementing extra credit. Please see submission guidelines below for more details.

3.3 Examples

Sample *clients.txt* file:

```
client1.txt
client2.txt
```

Sample path input file(*client1.txt*) as defined in *clients.txt* above:

```
Rfgq gq npmepykkgle yqqgelkclr 4. Nfcu, hsqr mlc jcdr!
```

Sample output file *output/client1.txt.out* for above input file:

```
This is programming assignment 4. Phew, just one left!
```

Sample wrong output:

```
Output:This is    programming assignment 4. Phew, just one left        !\t
```

Reasons for failed output above:

1. Extra message/string as a prefix: "Output:".
2. Extra white space characters in between and at the end.

Note: These are just two ways in which you can go wrong, output should have same number of characters as input (with valid decryption).

3.4 Rules and Hints

- *num_threads* can be less or greater or equal to the number of paths in *clients.txt*, so make sure to handle all the cases.
- You need to use **synchronization** mechanisms to manage the access to the **shared queue** by threads. Mechanisms that can be used are limited to: mutex locks, semaphores and/or condition variables.
- You are allowed to use **pthread** library only.
- While writing decrypted output, please make sure there are no extra characters/strings/messages including whitespaces written to the output apart from what input file contains. If not complied, test cases will be marked as failed.
- Only English alphabets should be decrypted. This includes: [a-z] and [A-Z]. Other characters should be output as is in the output file.
- Decryption of an empty input text is simply an empty output text. Absence of an input file should be warned and ignored (i.e., no output file should be generated).
- Print error messages with prefix "ERROR:" and warnings with "WARN:".

3.5 Assumptions

- Files to be decrypted will contain only single line.
- Although we are going to provide you with sample input, the test cases may contain various different inputs.
- *clients.txt* will not contain empty lines.
- You CANNOT assume the number of input paths. That is, a *clients.txt* file may contain 1 client or 1 thousand clients.
- Do NOT make any assumptions about *queue_size*. It can be any value.
- A single line of input from file to be decrypted will be restricted to 500 characters.
- *num_threads* and *queue_size* can be assumed to be integers. We will make sure to not give input value that exceeds limit of an *int* data type on any machine.

4 Few Do's

Please abide by these rules.

Technical:

1. Program must be written in C.
2. Make sure your program compiles using gcc compiler on any of the CSELab ubuntu machines. We will use the same development Environment. No points will be given if program does not compile.

Non-Technical

1. Assignment can be done in group of at most two head counts. Please include your names at the top in README file.
2. It is expected that you do not take any unfair advantage of any ambiguity in the problem statement. In that case, please confirm with Instructor or Teaching Asistants before you proceed with any approach. It might not make sense coming up with a solution that does not help hone your skills in the expected area.
3. This assignment is a bit involved and you might want to start implementing right away.

5 Submission Guidelines

The code should be packaged and submitted as a tar.gz file strictly (command: *tar -zcf csci4061pa4.tar.gz csci4061pa4*). Name of the TAR file should be **csci4061pa4**. This gunzip-ed TAR should contain a directory named **csci4061pa4**. This directory should contain:

- All your source files (*.c and *.h) and Makefile.
- A file named exactly **README** (not README.txt nor any other name) with the following info. Change the commented code below to include your personal information (and that of your partner if you have one) in the same exact format:

/* CSci4061 S2013 PA4
section: one digit number
section: one digit number
date: mm/dd/yy
name: full name1, full name2 (for partner)
id: U id for first name, U id for second name (a 7 digit number, not your x500) * Extra credit: implemented or not implemented */

Any additional information about your program that you feel we should know (e.g. design, implementation, some comments related to compilation, running, some errors you have ...etc) should be included here.

6 Grading

Maximum you can get for this assignment is 110%. Point distribution is as follows:

1. Following Submission Instructions and a sensible README:	5 points
2. Documentation and style:	15 points
3. Functionality and Error Handling:	80 points
4. Extra Credit:	10 points

References

[1] White space, `http://en.wikipedia.org/wiki/Whitespace_character`