Project 2
COM S 362
Spring 2022

1. Introduction

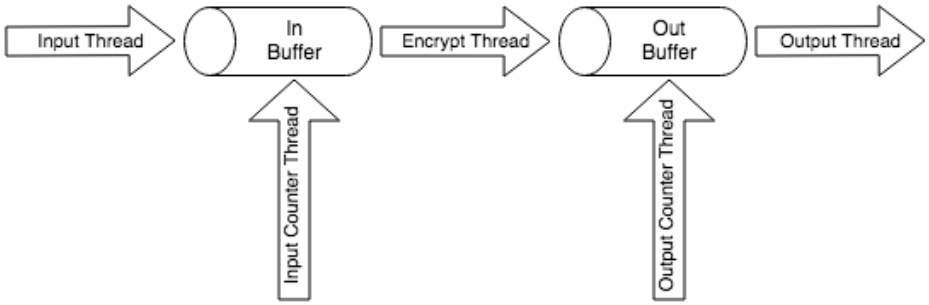
The purpose of this project is to implement a multi-threaded text file encryptor. Conceptually, the function of the program is simple: to read characters from an input source, encrypt the text, and write the encrypted characters to an output. Also, the encryption program counts the number of occurrences of each letter in the input and output files. All I/O and encryption operations are performed by a module (**`encrypt-module.c`**) which is supplied by the project. You are making a driver (in a file named **`encrypt-driver.c`**) that uses the encrypt module to provide the required functionality. Here is an attempt at a single threaded version of the solution (the full code is provided to you in **`encrypt-driver-simple.c`**).

```
void reset_requested() {
	log_counts();
}

void reset_finished() {
	// does nothing in simple version
}

int main(int argc, char *argv[]) {
	init("in.txt", "out.txt", "log.txt");
	char c;
	while ((c = read_input()) != EOF) {
		count_input(c);
		c = encrypt(c);
		count_output(c);
		write_output(c);
	}
	printf("End of file reached.\n");
	log_counts();
}
```

The major challenge of the project is that all processing must be performed in five concurrent threads and those threads must be kept *as busy as possible*. The threads are a reader, input counter, encryptor, output counter and writer. Also, the encrypt module can reset itself at any time, when it does so the character input and output counts must be logged in a consistent state (i.e., the number of characters written before the reset should be the same as the number of characters read). Since multiple threads will be accessing the same data structures, you will need to synchronize the threads to avoid race conditions (therein lies the difficulty).

Concurrency Requirement

All threads must be kept as busy as possible, if there is work available for a thread it must be doing that work.



Synchronization must be done pthread or semaphore constructs (i.e., your code cannot depend on spinlocks). For more information, consult the man pages for the following.

- `pthread_create`
- `pthread_join`
- `sem_init`
- `sem_wait`
- `sem_post`
- `sem_destroy`
- `pthread_mutex_init`
- `pthread_mutex_lock`
- `pthread_mutex_unlock`
- `pthread_mutex_destroy`
- `pthread_cond_init`
- `pthread_cond_wait`
- `pthread_cond_signal`
- `pthread_cond_destroy`

You have been provided with the files **`encrypt-module.h`** and `enrypt-module.c` which implement all the file I/O, encryption and counting operations you need. You **must** use these functions to perform I/O, encryption and counting. You should not make any assumptions about the timing of these functions, for example, **`count_input`** might return in 1 ms or in 1 second. The **`init()`** function must be called in **`main()`**. Portions of this project are graded using a testing framework, failure to follow these rules may result in a very low score. Here are the functions.

- `init - must be called from main()`
- `read_input - get the next input`
- `write_output - write an encrypted character`
- `encrypt - encrypt a character`
- `log_counts - logs the input and output character counts`

- `count_input – count an input(plaintext)character`
- `count_output – countanoutput(cyphertext)character`
- `get_input_count – get the input count`
- `get_output_count – get the output count`
- `reset_requested – this one you implement`
- `reset_finished – this one you implement`

2. Requirements

Main Thread (35 points)

All of you code including **`main()`** goes in a file called **`encrypt-driver.c`**. To get started copy **`encrypt-driver-simple.c`** to **`encrypt-driver.c`** and compile with:

```
gcc encrypt-driver.c encrypt-module.c -lpthread -o encrypt
```

Main does the following:

1. Obtain the input filename, output filename and log filename as **command line arguments**. If the number of command line arguments is incorrect, exit after displaying a message about correct usage.
2. Call **`init()`** with the file names.
3. Prompt the user for the input and output buffer sizes *N* and *M*. The buffers may be any size >1.
4. Initialize shared variables. This includes allocating appropriate data structures for the input and output buffers. You may use any data structure capable of holding exactly *N* and *M* characters for the input and output buffers respectively. A circular buffer is recommended.
5. Create the other threads (reader, input counter, encryptor, output counter and writer).
6. Wait for all threads to complete.
7. Log the character counts by calling **`log_counts()`**.

Reader Thread (10 points)

The reader thread is responsible for reading from the input file one character at a time and placing the characters in the input buffer. It must do so by calling the provided function **`read_input()`**. Each buffer item corresponds to a character. Note that the reader thread may need to block until other threads have consumed data from the input buffer. Specifically, a character in the input buffer cannot be overwritten until the encryptor thread and the input counter thread have processed the character. The reader continues until the end of the input file is reached.

Input Counter Thread (5 points)

The input counter thread simply counts occurrences of each letter in the input file by looking at each character in the input buffer. It must call the provided function **`count_input()`**. The input counter thread will need to block if no characters are available in the input buffer.

Encryption Thread (10 points)

The encryption thread consumes one character at a time from the input buffer, encrypts it, and places it in the output buffer. It must do so by calling the provided function **`encrypt()`**. The encryption thread may need to wait for an item to become available in the input buffer, and for a slot to become available in the output buffer. Note that a character in the output buffer cannot be overwritten until the writer thread and the output counter thread have processed the character. The encryption thread continues until all characters of the input file have been encrypted.

Output Counter Thread (5 points)

The output counter thread simply counts occurrences of each letter in the output file by looking at each character in the output buffer. It must call the provided function **`count_output()`**. The output counter thread will need to block if no characters are available in the output buffer.

Writer Thread (5 points)

The writer thread is responsible for writing the encrypted characters in the output buffer to the output file. It must do so by calling the provided function **`write_output()`**. The writer may need to block until an encrypted character is available in the buffer. The writer continues until it has written the last encrypted character. (Hint: the special character EOF (End of File) will never appear in the middle of an input file.)

Encryption Module Reset (15 points)

To change encryption keys, the encryption module occasionally needs to reset itself. When this happens, the input and output counts are reset to zero. The driver must also log the characters counts by calling **`log_counts()`** when a reset occurs. To reduce predictability, the module may decide to perform a reset at any time. Before it performs a reset, it will inform the driver code you are writing by calling **`reset_requested()`**. The encryption module is then blocked waiting from your code to return from **`reset_requested()`**, do not do so until it is safe for the reset to proceed. Consider this case, the input counter has counted 20 total characters and the output counter has only counted 10 total characters. If the driver allows the reset to immediately procced, the logged character counts will be inconsistent. However, you may not simply block **`reset_requested()`** until the entire file is read, you must use concurrency control mechanisms to ensure the reset can happen in a reasonable amount of time. Here are the two methods you are implementing.

```
void reset_requested() {
    // stop the reader thread from reading any more input
    // make sure it is safe to reset
```

```
    log_counts();
}

void rest_finished() {
    // resume the reader thread
}
```

Documentation (15 points)

You get 5 points for simply using a makefile. Name your source files whatever you like. Please make the name of your executable be **encrypt**. Be sure that "make" will build your executable on pyrite.

If you have more than one source file, then you must submit a **Readme** file containing a brief explanation of the functionality of each source file. Each source file must also be well-documented. There should be a paragraph or so at the beginning of each source file describing the code; functions and data structures should also be explained. Basically, another programmer should be able to understand your code from the comments.

Synchronization Requirement

Your program should achieve maximum concurrency. That is, you should allow different threads to operate on different buffer slots concurrently. For example, when the reader thread is placing a character in slot 5 of the input buffer, the input counter thread may process the character in slot 3 of the input buffer and the encryption thread may process the character in slot 2 of the input buffer.

Your program must not do the following: first let the reader thread put *N* characters in the input buffer, and then let the input counter thread and the encryption thread consume all the characters in the input buffer. This does not provide maximum concurrency.

3. Submission

You will submit your project on Canvas. Your program must compile and run without errors on `pyrite.cs.iastate.edu`.

Put all your source files (including the Makefile and the README file) in a folder. Then use command zip -r <your ISU Net-ID> <src_folder> to create a .zip file. For example, if your Net-ID is ksmith and project2 is the name of the folder that contains all your source files, then you will type zip -r ksmith project2 to create a file named ksmith.zip and then submit this file.