CS2002 Practical 5 - World Secure Channels

By: 150013565 Tutor: Juliana Bowles

Date: 26th April 2018

Abstract

This practical exposes system library tools for multi-process and concurrent programming.

This practical was split up into two parts; processes and threads. The process part of the practical was to write a program that creates two child processes that use a pipe between them to communicate securely. The thread part of the practical was to write and test and thread safe queue.

Part 1 - Multi-Process Encrpytion

Design

My design for Part 1 separates the various aspects of the practical into three separate files. Since this is a multi-process application, I imagined it as running three separate programs all from one program and implemented program by program. The first program is *process1.c* which does the encryption stage. The second program is *process2.c* which writes to the file. The third program is *my_otp.c* which calls the previously mentioned programs and executes them.

Implementation

I wrote the process independent of each other first before combining it all.

process1.c includes a header file which contains the library functions needed and one function, *pcipher*. This stands for the cipher process. This function checks whether the user gives a file or writes to stdin as reading both is slightly different. When given an input file it the function uses *fgetc* on the file pointer to the input file. When given input from stdin, it uses *fgetc* on standard in. There is definitely ways to combine this, but to make it clear and obvious as to what the function I have chosen to separate them. Once a character is read, it gets ciphered with the keyfile and writes to the pipe.

process2.c includes a header file which contains the library functions needed and one function, *pexport*. This stands for the export process. This function continuously reads from the pipe until it is empty. Once it reads from the pipe, it either prints to stdout or writes to the output file specified by the user. Once again, these two methods could be combined, but for clarity again, I have chosen to separate them.

my_otp.c includes a header file which contains the three functions. Two of which are error checking functions. One of them makes sure that the fork has forked successfully and the other checks that the input paths the user specified are readable/createable. The other function prototype in this header file is *process*. This function forks the the process so that the output from *process1.c* is the input for *process2.c*.

Testing

To run the code, run *make all*, this will produce all the necessary files. Then run the program according to the specification - provide a key file - and it will run. If the user is giving input, after the string is entered, press CTRL + D to submit the input. I ran my code with the provided *testOTP.sh* file and it passes. I have also produced my own tests to verify that it works. The table below shows the results of my testing.

Test No.	Input	Expected Output	Actual Output
1	No input file	Encrypted input	Encrypted input
2	No output file	Encrypted characters printed to console	Encrypted characters printed to console
3	no output input file	Encrypted characters printed to console	Encrypted characters printed to console
4	no key file	Usage message	Usage message
5	key file specified user continually presses enter	Nothing program waits for input	Nothing program waits for input

Part 2 - Thread-Safe Queue

Design

My design for the queue is taken from the provided code. I have added a mutex to the MQueue structure to ensure that only one thread can access the methods of the instance of MQueue at any one time. I have also added a method *clearMQueue* which empties the contents of the queue. This helps with testing the queue. My design for testing the queue consists of fourteen test methods and three general methods to aid with testing. The test methods break down to ten single thread tests and four multi-threaded tests. There is also a test object in the header file for my test code, this allows me to pass in arguments to multi-threaded methods. One of the three general test methods are *sumcheck* which goes through the queue and adds all the elements in the queue. If this addition equals what I expect then the queue is correct. The other two methods are to aid with multithreading operations on the queue.

Implementation

message_queue.c is implemented much like a traditional queue. The only difference is that every single method bar *initMQueue* locks the mutex and once its done, unlocking the mutex. This locking and unlocking prevents other threads from accessing the queue and performing operations on it at the same time.

mq_test.c contains all the various tests used to test the queue. Most of the tests are detailed below but one part of the testing that I am particularly proud about is testing the *printMQueue* method. To test this, I create a new process to run *printMQueue*. A pipe is also created to get the output of *printMQueue*. This pipe is then read back in the parent process and a verification check is done to make sure that *printMQueue* behaves as expected. The process run in these processes are in separate *.c* files. This is done to give them separate execution without interfering with the original test queue. All tests have been written some kind of assertion to check that the program does not run into errors. Hence if the method returns without raising a SIGABRT then it has passed the test.

Testing

These tests can be run and using the Makefile. By running *make all* and then *.mq_test*. The tests are detailed in the table below:

Test No.	Name	Reason for Test	Result of Test
1	test_queue_init	To check that the MQueue struct is successfully created	Pass
2	test_clear_queue	To check that the method to clear elements in the MQueue works	Pass
3	test_clear_empty_queue	To check that the method to clear elements does not fault when queue is already empty	Pass
4	test_print_queue	To check that the method to print elements in the MQueue works	Pass
5	test_print_empty_queue	To check that the method to print elements does not fault when queue is empty	Pass
6	test_enqueue	To check that the method to enqueue elements works	Pass
7	test_five_enqueue	To check that the method to enqueue multiple elements works	Pass
8	test_dequeue	To check that the method to dequeue elements works	Pass

9	test_dequeue_five	To check that the method can dequeue multiple elements	Pass
10	test_invalid_dequeue	To check that the program does not fail when there is nothing to dequeue	Pass
11	test_five_enqueue_mt	To check that the ADT can handle multiple threads enqueuing	Pass
12	test_dequeue_five_mt	To check that the ADT can handle multiple threads dequeing	Pass
13	test_enqueue_and_dequeue_five_mt	To check that the ADT can handle multiple threads enqueing and dequeing concurrently	Pass
14	test_print_queue_mt	To check that the ADT can print the result of multiple threads enqueing concurrently	Pass

Evaluation

Part 1

I am quite happy with my implementation for Part 1. It is easy to understand and it works as intended. The only limitation with Part 1 lies with the keyfile as messages can only be as long as the keyfile.

Part 2

Part 2 was quite tricky as it is quite difficult to prove my implementation for a thread safe queue is indeed thread safe. Ideally I would have liked to include more multi-threaded tests but I could not think of any. I tried looking online for more inspiration for concurrency testing but that didn't really help.

Conclusion

Overall this practical was very interesting. The ability to create processes and threads to split large jobs into smaller jobs has benefits for the end user. However, coding this is challenging as there are consequences such as data racing and deadlocks. Learning about these problems and how to minimize the probability that these problems occur has been very interesting.

Extension can be found on the next page.

Extensions

Sending Encrypted Messages Over the Network

My extension for this practical is sending encrypted messages over the network. I do this by combining part one and part two of the original specification.

There are three assumptions made with this extension. The first assumption is that both server and client must have the same encryption key file otherwise it will not be possible for the client or the server to encrypt and decrypt respectively. Possible ways around this are mentioned later in the report. Another assumption is that port 1238 is open on the network. This port was arbitrarily chosen and has no significance, any port could have been used. The last assumption is that the input string is no larger than 1024 characters long. Ideally I would like to send the encrypted chars character by character but I ran into trouble server side reading the message character by character. Hence this limit is imposed to send a string of max 1024 characters long. It was a lot simpler implementing sending one string rather than multiple characters.

The server is run first and waits for a client to connect. Once a client has connected, the client can either send a message using *stdin* or pass in an input file. The client then encrypts the message using a forked child process much like *process1.c* mentioned earlier. The output is then piped back to the parent where it is enqueued into the ADT created from *msg_queue.c*. The original objective here was to implement multiple threads reading from the pipe to speed up the sending process but I ran into problems with reading from the pipe with multiple threads. Such problems included threads reading the same element from the pipe rather reading different elements from the pipe and not joining the elements in the original order the message was written in. Once elements are added into the queue, all elements of the queue are dequeued and sent over the network as one string. Again, the idea here would have been to have multiple threads dequeuing and joining the dequeued element to the string to be sent over the network, but I ran out of time to implement this.

On the server side, the server receives the encrypted string and prints the encrypted string to the console. This shows that the network connection worked and that the message was sent over the network. One problem however, is that the console does not print out the entire encrypted message. I am not sure why this is the case, a reasonable explanation might be due to terminal not understanding the characters or the printf function not printing out the entire string. Once received, the server then decrpyts the message using the keyfile and prints it to console. The message has successfully been encrypted and sent over the network!

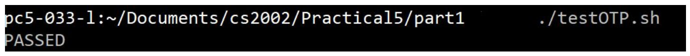
To run the program, run *make all* to make all the needed files. Once that has run, run *server* with a -k flag for the keyfile. After the server is running, run the *client* with the -k flag with or without an input file. If the user is giving input, after the string is entered, press CTRL + D to submit the input. The submitted input will be encrypted and be sent across to the server where it will print it out and decrypt it.

Test No.	Input	Expected Output	Actual Output
1	server no keyfile	Usage error	Usage error
2	client no keyfile	Usage error	Usage error
3	client no ip	Usage error	Usage error
4	client gives input file	Input file contents is encrypted and sent over the network	File contents is encrypted and sent over the network server receives it and prints encrypted and decrypted versions
5	client no input file	User string is encrypted and sent over the network	User string is encrypted and sent over the network server receives it and prints encrypted and decrypted versions
6	no input file and continually clicks enter	Exit message	Exit message

One limitation of this extension is the fact that both server and client require the keyfile. As Marwan mentions in the specification, using One Time Pad removes the challenge from encryption to key distribution and there is no safe way to distribute keys. An alternative solution to the OTP is Pretty Good Privacy (PGP). This encrpytion works by generating a public-private key pair. The public key can be sent out to any body and they can use this to encrypt the message they want to send you. After they have encrpyted it using your public key, only you can decrpyt it using the private key. This solution removes the focus from distributing the key file altogether as you can give anyone your public key as only you can decrpyt it.

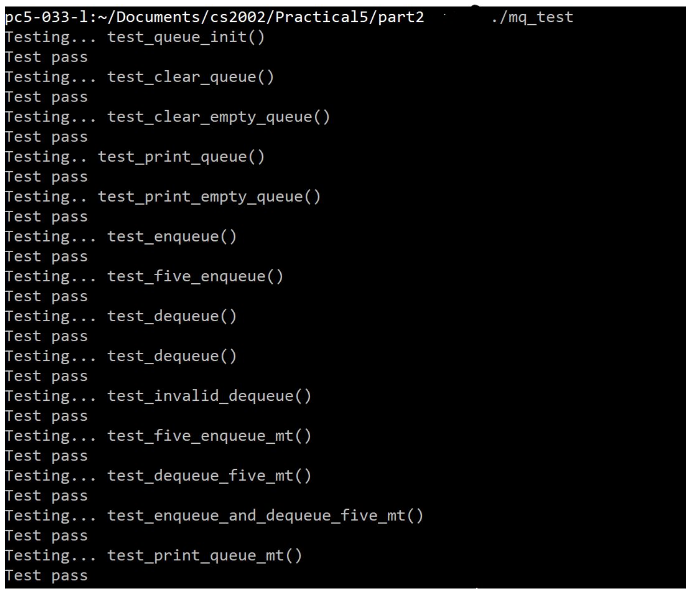
Overall I am quite happy with this extension. Learning about sockets in C sending encrpyted messages over the network and then decrpyting the message to get the original message is quite satisfying.

Below are some images of the tests conducted above as evidence



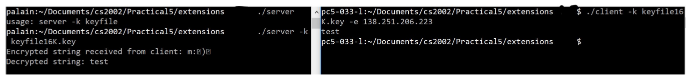
```
pc5-033-l:~/Documents/cs2002/Practical5/part1       ./testOTP.sh
PASSED
```

Figure 1: TestOTP script passing for Part 1



```
pc5-033-l:~/Documents/cs2002/Practical5/part2 [redacted] ./mq_test
Testing... test_queue_init()
Test pass
Testing... test_clear_queue()
Test pass
Testing... test_clear_empty_queue()
Test pass
Testing.. test_print_queue()
Test pass
Testing.. test_print_empty_queue()
Test pass
Testing... test_enqueue()
Test pass
Testing... test_five_enqueue()
Test pass
Testing... test_dequeue()
Test pass
Testing... test_dequeue()
Test pass
Testing... test_invalid_dequeue()
Test pass
Testing... test_five_enqueue_mt()
Test pass
Testing... test_dequeue_five_mt()
Test pass
Testing... test_enqueue_and_dequeue_five_mt()
Test pass
Testing... test_print_queue_mt()
Test pass
```

Figure 2: Test for Part 2



```
palain:~/Documents/cs2002/Practical5/extensions [redacted] ./server
usage: server -k keyfile
palain:~/Documents/cs2002/Practical5/extensions [redacted] ./server -k
 keyfile16K.key
Encrypted string received from client: m:�)�
Decrypted string: test
```

```
pc5-033-l:~/Documents/cs2002/Practical5/extensions [redacted] $ ./client -k keyfile16
K.key -e 138.251.206.223
test
pc5-033-l:~/Documents/cs2002/Practical5/extensions [redacted] $
```

Figure 3: An example of the server receiving a string from client