Real *Wild* CS2002-World Secure Channels

This practical exposes system library tools for multi-process and concurrent programming. The introduction to modern encryption, and ability to generate provably secure messages, is just fortunate.

One Time Pad

The development of modern encryption techniques, arguably, begins with the *one time pad (OTP)*. It is the only known *provably* secure cryptographic system. Interestingly, it is also impractical[1].

The algorithm is simple, efficient, and characterised by the following relationship: $y = x \oplus k$, where $\oplus$ is the `xor` operation. The use of `xor` makes the process fully reversible, meaning that we use $x = y \oplus k$ to recover the original message!

Say we have a message file m, a keyfile k, and wish to produce a cipher (or encrypted version of the) message c. The message is encrypted on a byte-by-byte basis, such that the i^{th} byte is encrypted as:

$$c_i = m_i \oplus k_i. \tag{1}$$

The formulation means that each byte in the key is used only once. The implication is that keys must be at least as long as the message.

Part 1: Create a secure channel for multi-process communication.

Say two processes wish to share data, but are worried that the operating system is logging or monitoring all inter-process communication. What might be done in response? Your task is to create a (very) rudimentary encrypted channel of communication between processes.

Write a program that creates two child processes that use a pipe between them to communicate securely. One child will take and encrypt the message using a one time pad, as described above, before sending it over the pipe. The other child will output the encrypted message to a file. In particular,

- both children must be given a key (stored in a file) to encrypt or decrypt the message;
- the children must share a pipe between them;
- the ‘message source’ child must have an input (see command line arguments, below);
- the ‘message destination’ child must know where to save the encrypted message.

During execution, then, your program will consist of three separate processes. Also, the use of one time pad means that correctness can be ascertained by running the program a second time with the encrypted file as the input. A simple test suite of files is available alongside this spec sheet on `studres`.

[1] Short explanation: OTP shifts the challenges from encryption to key distribution, thus suffering from scale.

Program Execution

Write a program, `my_otp`, that takes the following arguments (optional args appear in brackets '[]'):

```
my_otp [-i infile] [-o outfile] -k keyfile
```

where arguments and their defaults (when arguments are omitted) are as follows:

-i Name of input file containing the cleartext, i.e. human-readable, message. *By default* the program must read from `stdin`.

-o Name of output file where the cipher, or encrypted text, will be written. *By default* the program must write to `stdout`.

-k Name of the 'secret' keyfile containing the encryption key. This argument is **mandatory**, since without a key there can be no encryption.

A '`usage:`' message should appear in response to invalid or absence of arguments.

NOTE: Sample source to process command-line arguments using `getopt` is given in the appendix at the end of this document.

Hints and Suggestions

Stage your progress! Start with a single-process version of the program so that potential issues pertaining to I/O and encryption are isolated from multi-processing and IPC. (Aside: This strategy becomes limited with scale in size and/or complexity – such matters are for concurrency modules in later years.)

Part 2: "My First Synchronised Structure!" (A Thread-safe Queue)

Far more than just being encrypted, secure channels must also be reliable and ensure data consistency.

A *message queue*, for example, is a high-level IPC abstraction that ensures data consistency by restricting reads and writes from multiple sources. As is suggested by the name, message queues enforce FIFO ordering on messages by design.

Your tasks are to (i) implement a *thread-safe* message queue, then (ii) write a test program to test the message queue, as follows:

i The message queue should be implemented in a file called `msg_queue.c`, and according to the struct definitions and function declarations in the `msg_queue.h` header file posted on `studres`. The queue itself must be a linked list; the messages themselves are restricted to integers.

ii Write a multi-threaded test program, `mq_test.c`, that uses and demonstrates correctness of the message queue.

Hints and Suggestions

See Part 1: Stage your progress! In this context, write both the message queue and the test program for a single thread of execution to ensure correctness of the queue, itself. Then use lecture notes, as well as `thread_count_mutex.c` from tutorials, to ensure thread-safety and appropriate testing.

Constraints

Pre-defined functions must adhere to their headers (declarations). Structure definitions and their members must also remain; however, additional members may (and at least in one manner should) be added. There are no additional constraints.

Requirements & Submission

A zipped archive is posted on `studres` with fixed directory structure and starting source code. Make sure to work within the given structure, otherwise automated testing will fail.

The final submission should be a zipfile submitted via MMS that contains the following:

- A short report in PDF format covering design choices, source files (if applicable), data structures (if applicable), implementation and testing, any problems encountered and lessons learned. Ensure that extensions are clearly labelled.
- Your source files as described above, organised according to the given structure; any additional `.c` and `.h` files are left to your design.
- `Makefiles` to build the executables, with a `clean` option to remove executables and object files.

(Possible) Extensions

The list of possible extensions includes, though by no means is limited to, the following. *Full functionality is a requirement*; extensions will be diminished in value or ignored entirely, otherwise.

- A message may be further safeguarded by distributing pieces to individual parties in such a way that the message may only be understood once the message is re-assembled. This idea suggests a multi-process version of `my_otp`, perhaps that takes a `-p` option to indicate the number of recipients.
- Parts 1 and 2, in combination, suggest an encrypted message queue (i.e.unlike previous suggestion, individual messages here are independent). Even so, there is a large gap between suggestion and execution.
- An inevitable question that each student will face during this practical is, "How can a 'job' (i.e. process/thread) know when the others have completed?" Rather than waits, cancellations, and kills, does something more intelligent await? Possibilities range from the naïve to the complex, determined by a range of (very) reasonable assumptions, and characterised by their trade-offs. We label such endeavours as *message protocols*. Note: It is safe to assume that no message can be lost in transmission.

There are many questions and considerations that ensue. Feel free to discuss these or others with tutors. Make sure to describe and detail these or other extensions in your report.

Assessment

Generally speaking, work is graded primarily along the following criteria:

Scope	The extent to which code implements the features required/specified
Correctness	The extent to which code is consistent with the specs and bug-free.
Design	The extent to which code is well written, ie. clearly, succinctly, elegantly, or logically.
Style	The extent to which code is readable, eg. comments, indentation, apt naming

More specifically, for this practical:

- 1-6: Very little evidence of work, software does not compile or run, or crashes before doing any useful work. (Please see or seek help from your tutor.)
- 7-10: Partial solutions with serious problems such as not compiling (at the low end), or lacking in robustness and maybe sometimes crashing during execution (at the high end).
- 11-13: Mostly correct implementations accompanied by a good report.
- 14-15: Fully correct solutions accompanied by a good report.
- 16-17: Fully correct pointer-based implementation demonstrating very good design and code quality, good functions and structures, and accompanied by a well-written report.
- 18-20: As above, with one or more extensions, accompanied by an excellent report. (Note: The existence of extensions is required, but by no means sufficient.)

Rubric

There is no fixed weighting between the different parts of the practical. So that students might be better prepared for subsequent practicals, general feedback will be offered according to guidelines published in the Student Handbook at,

https://info.cs.st-andrews.ac.uk/student-handbook/learning-teaching/feedback.html

Work must be submitted via MMS by the deadline. The standard lateness penalties apply to coursework submitted late, as indicated at

https://info.cs.st-andrews.ac.uk/student-handbook/learning-teaching/assessment.html#lateness-penalties

As a reminder, the relevant guidelines on good academic practice are outlined at http://www.st-andrews.ac.uk/students/rules/academicpractice/

Appendix: `getopt`

The `getopt` function can be used to easily retrieve command line options that are provided with the hyphen '-' character, eg."`ls -a`".

The third argument is an "options string" that identifies the options themselves, and if a corresponding argument is required. In the example provided, 'a' and 'c' are options (no arguments), while 'b' is an option that must be followed with an argument.

Each call to getopt() will return the next option in `opt` and the string representation of the argument in `optarg` (a global var linked in from library).

```
#include <stdio.h>
#include <unistd.h> /* POSIX requires getopt(). */

/***
 * A getopt() example. More info may be found at the man page:
 *      ~> man 3 getopt
 * Try this program with any valid or invalid arguments:
 *      ~> ./<your_executable_name> -a -b -c
 *      ~> ./<your_executable_name> -x -a -z
 *      ~> ./<your_executable_name> -a -b hello
 */

int main (int argc, char *argv[])
{
  int opt;

  while ((opt = getopt (argc, argv, "ab:c")) != -1)
  {
    switch (opt)
    {
      case 'a':
          printf("Option: '%c'\n", opt);
          break;
      case 'c':
          printf("Option: '%c'\n", opt);
          break;
      case 'b':
          printf("Option '%c' w/ mandatory Argument: \"%s\"\n",
                 opt, optarg);
          break;
      case ':':   /* Fall through is intentional */
      case '?':   /* Fall through is intentional */
      default:
          printf("Invalid option or missing argument: '-%c'.\n",
             optopt);
          break;
    }
  }
  return 0;
}
```