Lab #6: Multithreading

Introduction

In this lab we look at multithreading along with techniques to coordinate thread access to shared data.

Lab Activities

due 9 am Dec. 2nd

1. Locate the "`lab6n1.c`" file. It uses threads – an array of `pthread`s:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

void* workthread(void *arg) {
   int count = (int) arg;
   for (int i = 0; i < 20; i++)
       count++;
   sleep(1);
   printf( "Thread %d done\n", count );
   return (void*) count;
}

int main(int argc, char *argv[]) {
   int threads = 10;
   pthread_t p[threads];
   int       results[threads];
   printf("Starting %d threads\n", threads);

   for (int i=0; i<threads; i++){
      results[i] = 0;
      pthread_create(&(p[i]), NULL, workthread, (void*) (i+10) );
   }

   sleep(10);
   for (int i=0; i<threads; i++)
      printf("Final values   : %d\n", results[i]);
   return 0;
}
```

Compile using a command like: `gcc -o lab6n1 -g lab6n1.c -pthread`
The compiler may print warnings (but not errors); it should create a "`lab6n1`" executable file. Run the program and note its output.
Modify the program to comment out the call to `sleep` in the `main()` function but not the one in the thread function. Compile and run. How is the output of the program different from before?

Now modify the program so that after all of the threads have been created, the program waits – without calling the `sleep` function – for all of the child threads to finish and places the "return" value of each thread in the `results` array so that the values computed in each thread is printed by the `main()` function. Submit the modified program.

2. Locate the "`lab6n2.c`" file:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

pthread_mutex_t mutex;
int count = 0;

void* workthread(void *arg) {
   for (int i = 0; i < 20; i++) {
```

```
        pthread_mutex_lock( &mutex );
        count++;
      }
      return (void*) count;
}

int main(int argc, char *argv[]) {
   pthread_t p1, p2;
   printf("Starting 2 threads\n");

   pthread_create(&p1, NULL, workthread, NULL );
   pthread_create(&p2, NULL, workthread, NULL );

   pthread_join( p1, NULL);
   pthread_join( p2, NULL);
   return 0;
}
```

Compile and run it. Explain whether the threads that this program starts make any progress or not. If not, what is the problem?
Modify this program to unlock the threads' critical section – without changing the lock code. Submit the modified program.

3. Multithreaded crypto code breaking
   In this part we will implement a program that will try to break a coded message by guessing the key used to create the code.
   Rather than coming up with good ways to break codes, the objectives of this part are to learn how to:
   - use mutexes and condition variables effectively.
   - lock queues and other data structures safely.

   The program, **`pbreak`**, should try different keys to see if one of the keys, when used to decode the message, gives us something "interesting". We should be able to run the program as shown below:

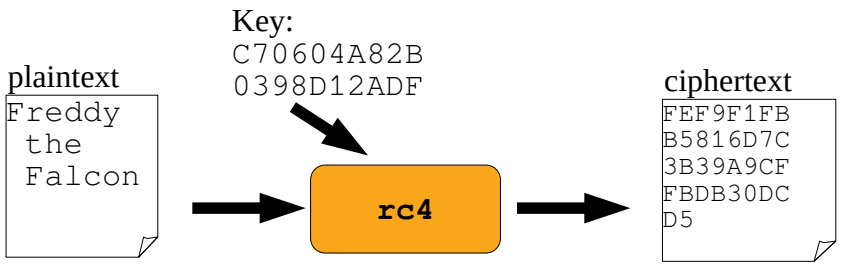
   `./pbreak filename n`

   The first argument, `filename`, should be the name of a file that contains a coded message – the "ciphertext" – in binary form and the second argument should be a positive integer for the number of threads to use to try to break the code.

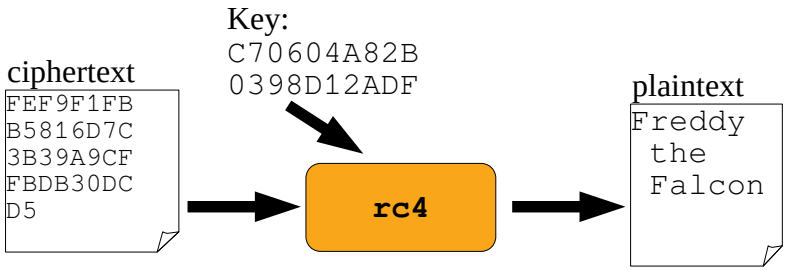
   For the encryption algorithm, we will use a simple symmetric stream cipher called RC4 which is still used in communication protocols. There are many ways to use (and misuse) RC4 and we will use it in a rather simple way: keys can only be up to 256 bytes long but messages can be any length.
   The rc4_demo.c program shows how to use our version of RC4:

```
$ ./rc4_demo
> Key:           0xC7 0x06 0x04 0xA8 0x2B 0x03 0x98 0xD1 0x2A 0xDF
> Plain text:    Freddy the Falcon
> Cipher text:   FEF9F1FBB5816D7C3B39A9CFFBDB30DCD5
> Decrypted:     Freddy the Falcon
$
```

   Note that the key in this case is 10 bytes. The starting (plaintext) message to be sent is "Freddy the Falcon" in ASCII. When we run this plaintext and a 10-byte key (shown as a hexadecimal number) through the **`rc4`** function, we get the encoded ciphertext (same length as the plaintext) shown hex:



The demo program also shows that we can convert the ciphertext back to plaintext (i.e. decrypt the ciphertext) if we know the encryption key:



Our **pbreak** program will be given just one input: a ciphertext file. The only assumptions are:

1. The encryption/decryption algorithm used was RC4 so **pbreak** can use code in the `rc4_demo.c` program.
2. The key is at most 256 bytes long.
3. The plaintext message is in English and has some common English language words including the word "Russia".
4. **pbreak** should only print the "correctly" decoded message and the key(s) used to decode.

In particular, **pbreak** is not given the key – **pbreak** has to "guess" what the key is by trying all possible keys (brute force) and checking if one of these attempts gives us the original "plaintext" Important: write the **pbreak** program in such a way that it should be easy to switch from using RC4 to some other encryption scheme .

Queues

Use a queue to keep a running list of all keys generated by the parent process (or, optionally, one or more threads). Write your own C code for the parent process (or thread(s)) to generate all possible keys using "brute force" and put each potential keys into an bounded queue using the enqueue operation. When all possible keys have been generated and put into a queue, the parent process should use a condition variable to signal that it is done generating keys.

Use the specified number of "worker" threads to try each key to decrypt the message. A single worker thread in this collection of "`n`" threads should get dequeue the next key in the queue and try to decode the coded message using this key. If the decoding gives us "something interesting" then we know that we have the "right" key.

Print the results (decrypted interesting messages) to either the screen or an output file while the other threads are still running; to prevent this multithreaded code from giving us messy output, "lock" the screen or file so that only one thread prints its result. Have each thread check the queue before dequeue-ing so that they don't get stuck waiting for more keys to show up even after all the work is done.

All the worker threads can return (i.e. stop running) after the parent has signalled that it is done generating keys and the queue is empty.

Implementation Hints

The `pbreak` program should use pthreads. Take an especially close look at `pthread_create`, `pthread_cond_wait`, `pthread_cond_signal`, `pthread_mutex_lock`, and `pthread_mutex_unlock`. An implementation that spins instead of using condition variables will end up receiving a poor grade (even if it passes the tests). Identify all critical sections and only lock what needs to be locked. You can also use semaphores if you wish.

Grading

Your implementation will be graded mainly on functionality and testing.
You are required to use at least 1 queue and as many threads as the user requests,

In your report:
List each critical section (show the C statements) in your code and explain why it is a critical section.
Explain how you tested your code:

- what input ciphertext you tested
- how you tested the creation of a set of threads and how you joined them
- how you tested your queue functions
- how you tested your critical sections to make sure they were locked
- how you tested for memory leaks

Divide your code into separate compilation units (.c files):

- `pbreak.c` with the main() function
- `thread.c` for the thread function to try a key
- `queue.h` and `queue.c` for the queue operations
- `.h` files and other `.c` files as required
- Testing code:
  - `testpbthreads.c` to test thread creation and joining
  - `testpbqueue.c` to test queue functions
  - `testpbcrsect.c` to test your critical sections
- Makefile should be set up with the correct dependencies so that only those files that need to be recompiled will be compiled. For example, if `queue.c` is modified, only `queue.c` should recompiled but if a `queue.h` file is modified, all .c files that include `queue.h` should be recompiled

The relative performance of your program is also important, so points for efficient code!

Lab 6 Submission

Make sure that your code runs correctly on a 64-bit Ubuntu VM. To ensure that I compile your C programs correctly, create a simple **`Makefile`** with your **`.c`** source files in the same directory. Do **not** submit any `.o` files.

Turn in as a group, using the Canvas dropbox:

- A report document listing all critical sections in your Part 3 code and explanations as far as why each one is a critical section.
- The report should also list the various kinds of inputs you used to test your submitted code
- All source files needed to build the programs
- A Makefile (for building all the C programs using a single **`make`** command)

as a single **`.tar.gz`** file.
Make sure to look up the **`man`** files for the **`tar`** and **`gzip`** commands to see how to create a **`.tar.gz`** file.