CS221 Lab21 – Multithreading again

Paul Haskell

INTRODUCTION

In the previous lab, we sped up our program by running multiple tasks in parallel. For this lab, we will use multithreading for a different reason: we will provide a responsive user interface while running some big and slow processing tasks in the background. The code will be more complicated than last time.

Prime numbers

You recall that a positive integer is prime if the only smaller integer that divides into it with no remainder is 1. It's easy to see that numbers like 13 and 17 are prime, while 15 and 18 are not. But how about 531426583489? Or 2305843009213693951?

Large prime numbers are very important for encryption, security, and cryptocurrency, and people have researched very sophisticated algorithms and computer architectures to find prime numbers quickly and efficiently. For this lab, we will use a really basic checker:

```
int found = 0;
for(int n = 2; n <= 1 + sqrt(NumberToCheck); n++) {
       if (NumberToCheck % n == 0) {
           printf("NOT PRIME!);
           found = 1;
           break;
       }
}
if (!found) { printf("PRIME!"); }
```

Please write a program called **primechecker.c** that repeatedly prompts the user to enter a number, reads the numbers from the keyboard, checks if they are prime, and prints out the results.

Details

We will work with some big numbers when testing today's program. Let's use the `unsigned long long` data type.

With big numbers, the basic algorithm above is slow: 1111111111111111111 is prime, but my computer needs 30 seconds to verify it with the above simple algorithm. It is not a good experience for the user if she enters a number and then doesn't see the program do anything for 30 seconds. Your program should prompt the user immediately every time after a number is entered.

- To do this, use different threads for reading user input and for checking whether input numbers are prime

- For speed, please create multiple "checking" threads, so your program can check multiple different numbers in parallel.

The threads in this program must coordinate with each other more closely than in **fastmerge.c** . The "main thread" passes values to the "checking threads" and the "checking threads" pass results back to the "main thread". Since all the threads run at the same time, there is a danger that the data passed between threads gets corrupted. We must use semaphores (mutexes) to prevent this.

- Create a queue of "jobs", with an entry for each number that must be checked for primeness. Your queue will need methods to add a number into the queue and to remove a number from the queue. (Remember your earlier Fifo software?)
- You must handle the special cases of an attempt to add to a full queue or to remove from an empty queue.
- Use a mutex so that only one thread at a time can change queue data.
- Your "main thread" should read numbers from the keyboard and add them to the job queue. Keep doing this until the keyboard input ends, then exit cleanly.
- Your "checking threads" should repeatedly check the input queue to see if it is nonempty. If so, the thread can retrieve a value from the queue, check it for primeness, and print out a result. Please print in the following format:
  `PRIME: <<the prime number>>`
  or
  `NOT prime: <<the number>> <<its divisor>>`
- If the input queue is empty, the "checking threads" can just `usleep()` for a fraction of a second. That frees up a CPU core to other useful work.

I said above that we also need to manage results—but we have the "checking threads" print out the results! We still need to manage results, for a subtle reason: we do not have a way to ask our "checking threads" whether they are busy checking a big number or if they are all done, sitting around waiting for a new input to arrive. We don't want our program to exit when the keyboard input finishes, if some thread is still doing a big calculation.

Couldn't we just wait for 30 seconds? Is 30 seconds enough? Does the user want to wait around for 30 seconds after the program is all done, "just in case"?

Here is a better approach:

```
struct ResultCount {
      sem_t resultLock;
      int resultCount;
}
```

Your "checking threads" increment `resultCount` every time they finish a job. At the end of the program, your "main thread" can wait until `resultCount` equals the number of jobs you submitted.

CONCLUSION:

This lab gave you experience with the "producer/consumer" multithreading model, and with how to synchronize thread operations safely. The previous lab used the "parallel processing" multithreading model. There are plenty of other more complicated models, but these two are a great start!

Please push your **primechecker.c** to GitHub before the project deadline. The deadline for completion of Lab21 is 11:59pm Monday April 21.

Task	Score, points
Makefile provided that successfully compiles **primechecker**	5
primechecker.c runs as specified, runs operations in parallel, and gives correct results with 5 test cases	25
Software and design quality, as judged by the grader	10