Tutorial 2: Pthread Tutorial: Parallel Prime Number Generation with pthread_t, pthread_mutex_t, pthread_mutex_cond (v0.10)

Note: This tutorial must be completed on a computer with a minimum of two CPU cores. Four to eight CPU cores are recommended. It is recommended to close the web browser while running the program. If the browser cannot be closed, it is recommended to close all tabs except for the assignment PDF and Canvas.

Note: This tutorial uses an entirely different prime number generation algorithm than assignment 2. The algorithm in tutorial 2 assumes the use of a fixed number of pthreads (e.g. n=4 or n=8), whereas in assignment 2 the algorithm assumes the use of an infinite number of threads (n) and is based on recursion. The implementation of the algorithm in tutorial 2 is more efficient in practice as a result of the inefficiency of implementing recursion using separate pthreads in C.

The purpose of this tutorial is to review C pthreads, mutex locks, and condition variables to support assignments in TCSS422.

Complete this tutorial using your Ubuntu Virtual Machine, or another Linux system equipped with gcc.

Tutorial Submission

This tutorial is accompanied by a Canvas quiz. After completing the activities described below, log into Canvas and complete the Tutorial #2 Quiz. Tutorial #2 is scored in the "Tutorials/Quizzes/In-class Activities" category (15%) of TCSS 422.

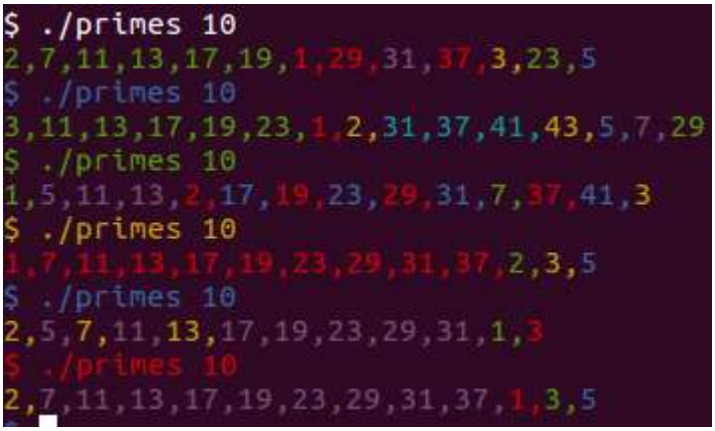
1. **Download primes starter code.**

To start the tutorial, download the tar.gz archive containing the "primes" starter code:
http://faculty.washington.edu/wlloyd/courses/tcss422/tutorials/primes_starter.tar.gz

This program generates prime numbers with up to 8 pthreads. Inspect the code carefully. Try to understand how it operates. Spend ~5-10 minutes studying the code. Then compile the project with make:

```
$ make
gcc -pthread -I. -Wall -Wno-int-conversion -D_GNU_SOURCE -lreadline -
I/usr/local/include -L/usr/local/lib counter.c primes.c -o primes
```

Now run the "primes" program. Specify "10" as a command line argument to restrict output to 10 prime numbers (***see figure***):



This program generates prime numbers. Note that each thread prints the prime numbers it generates using a different color.

However, out of the box we notice there are some *issues*.

First, we have asked the program to generate 10 prime numbers. When we run the program, however, we seem to get a different number of prime numbers each time !

Your first task is to correct this problem.

Solution: Introduce a state variable to track when any of the threads creates the n^{th} prime number. This is the number of primes the user has requested to produce. Also, introduce a mutex_lock and condition variable to signal the parent thread when the n^{th} prime number has been generated. The main program will then terminate the program.

2. **Generate correct number of primes**

Before the first function in primes.c (about line #23), add three new global variables to the project:

```
int bdone = 0;
pthread_mutex_t donelock;
pthread_cond_t donecond;
```

The donelock is a lock variable that protects changes made to the bdone variable. The donecond variable is used to signal when the program is done.

Next, we need to modify the findPrime() function so that it only generates prime numbers before the program is "done". The program is done when it generates "int genprimes" prime numbers.

Immediately before the "#if OUTPUT" tag, acquire the donelock, and add the "if statement" as:

```
    pthread_mutex_lock(&donelock);
    if (bdone==0)
    {
```

Then, locate the "#endif" tag, and close the if statement:

```
    }
```

Note that our condition code to display prime numbers is entirely inside this if block. The #IF block should not change the fundamental structure of the code. We have to be careful not to lock/unlock inside an #IF block.

Next, on about line 69, we need to replace the simple inc_counter() function call that increments the number of prime numbers we've created with a more sophisticated block of code:

Original:

```
  inc_counter(&primescnt);
```

New:

```
    if (inc_counter(&primescnt) == genprimes)
    {
      bdone=1;
      pthread_cond_signal(&donecond);
    }
  pthread_mutex_unlock(&donelock);
```

With the new version, we first check if incrementing the prime number counter (primescnt) allows us to reach

"genprimes". When we've created the last prime, we set the bdone flag to true (1), and fire a signal. We then proceed to release the donelock.

Next, we need to initialize the donelock and donecond variables inside int main().
On the 4th line of int main() immediately after initializing the counters, add the following to initialize the lock and condition variables:

```
pthread_mutex_init(&donelock,NULL);
pthread_cond_init(&donecond,NULL);
```

Next, we will modify the end of int main(). The original version uses pthread_join() to wait for each of the threads to finish. In the new version, comment out all of the calls to pthread_join():

```
//pthread_join(p1, NULL);
//pthread_join(p2, NULL);
//pthread_join(p3, NULL);
//pthread_join(p4, NULL);
//pthread_join(p5, NULL);
//pthread_join(p6, NULL);
//pthread_join(p7, NULL);
//pthread_join(p8, NULL);
```

Next, add the following lines to wait for any of the threads to signal the parent that prime number generation is complete. Do this before the printf() statement:

```
pthread_mutex_lock(&donelock);
while (bdone==0)
  pthread_cond_wait(&donecond, &donelock);
```

Only one thread needs to signal the parent to inform it that the last prime number has been generated.
Any threads that may be running are terminated when the program exits.
Now, save your code, and rebuild with make, and run the program to generate 10 prime numbers:

```
$ make
gcc -pthread -I. -Wall -Wno-int-conversion -D_GNU_SOURCE -lreadline
 -I/usr/local/include -L/usr/local/lib counter.c primes.c -o primes
$ ./primes 10
1,11,13,17,19,23,29,31,2,41
$ ./primes 10
1,7,11,13,17,19,23,29,31,2
$ ./primes 10
1,7,11,13,2,17,19,23,29,31
```

Now, as you can see each time the program is run, only 10 prime numbers are generated.

3. **Performance test the program**

For this step, please adjust the number of pthreads in your program to equal the number of CPU cores of your Ubuntu environment. The number of CPU cores can be checked with the following command:

```
$ lscpu | grep "^CPU(s):"
```

Please comment out pthread_create statements for any extra threads.

Additionally, extra pthread variables can be commented out.

Now, notice on line 18 in your program a compiler constant called OUTPUT is defined and set to 1.
Compiler constants in C allow for segments of the program to be selectively enabled or disabled depending on the value of the variable.

Now open a second terminal window.
In the terminal, run the command:

```
$ top -d .2
```

Use top to follow how the percentage of CPU utilization (%CPU) that the primes program obtains.

Time the generation of 50000 prime numbers with output displayed to the screen:

```
$ time ./primes 50000
```

Note, this will print 50,000 prime numbers to the screen.

Write down the real, user, and system time. (capture the run time with output enabled)

Next, record the time to generate 50000 prime numbers while redirecting the output to **`/dev/null`**.
"**`/dev/null`**" in Linux is a virtual device. You can think of this as the Windows recycle bin, or the macintosh recycle bin. Any data that goes to **`/dev/null`** is effectively gone.

```
$ time ./primes 50000 >/dev/null
```

Again, write down the real, user, and system time, and note the difference between the two.
(capture times with no output)

Now, edit primes.c and change the OUTPUT compiler constant on line 18 **to 0**, and rebuild your program.

Now time the program with and without output. The time should be similar:

```
$ time ./primes 50000
$ time ./primes 50000 >/dev/null
```

On some computers you may observe that turning off the output has actually ***slowed down*** the program.
(disclaimer: *this behavior may not occur on all systems*)

Why could this happen?
When observed, the extra I/O (input/output) code that prints to the screen ***should*** make the program slower, but it may actually make it faster.

What is happening is that the extra work of printing to the screen *spaces out the operations of the program* to

effectively **reduce the lock contention**. Having small pauses between each thread's attempts to acquire the lock reduces race conditions between the threads. These random-like delays help the program achieve better concurrency and less waiting.

4. **Correct prime number sequencing**

Problem: Random CPU context switches occur at any time causing threads to be interrupted between when they initially grab a number to check if it's prime, and when prime numbers are finally reported by printing their value to the screen.

Solution: To enforce a strict ordering of prime number generation, a lock variable can be introduced to force: (1) finding a prime number, and (2) reporting it to the screen, to be a **single atomic operation**.

The first 10 prime numbers should be: ***1, 2, 3, 5, 7, 11, 13, 17, 19, 23***
However, the program is having trouble here:

2 and 3 are generated, but they appear in the wrong place. 17 is missed altogether and instead we get 29.

Why is this happening?

Notice that each time there is an issue, output is displayed in a different color.
What happens is that a thread may acquire a number from the counter, but it gets preempted before it can display its output.
For the sequence above there are at least 3 threads: **red**, **yellow**, and **green**.
Red acquires 1 from the counter, then yellow acquires 2, and green acquires 3, but red obtains access to the display to print 1, 5, 7, 11, 13, before yellow can print 2, and green 3.

In our program the act of acquiring a number to check if it is prime, and displaying a confirmed prime number are not atomic. Presently these operations are not synchronized, and they can happen at any time without coordination.

A huge benefit here is that the code can scream through prime generation using this non-optimal algorithm fairly quickly if we only care about generating any primes, and not necessarily the order they are generated.

(note: a better algorithm for prime number generation is Sieve of Eratosthenes, but this algorithm doesn't exercise the counter data structure from Chapter 30, that is used in Assignment 2.)

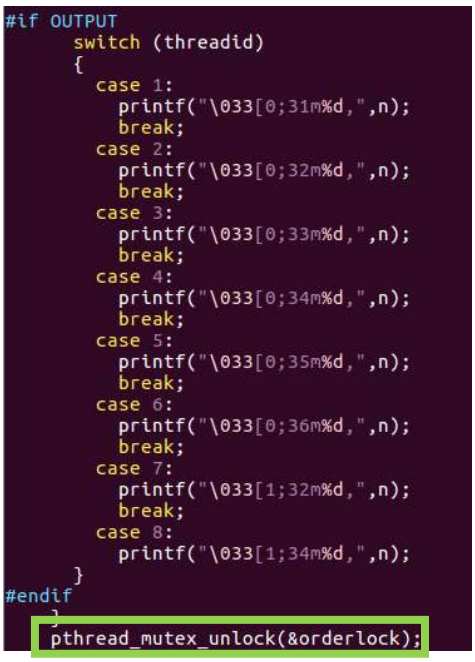
Using mutex-locking we can fix the program so that it generates 50,000 primes in the correct order.

At the beginning of the program after the declaration of the donelock and donecond, add a lock variable called "orderlock". The order lock will ensure that prime numbers are generated in numerical order.

```
pthread_mutex_t orderlock;
```

Now, inside the findPrime() function, on the very first line of code acquire the orderlock:

```
pthread_mutex_lock(&orderlock);
```



After the output block, release the lock as in the figure. →

Next, at the top of int main(), add a line to initialize this global lock variable:

```
pthread_mutex_init(&orderlock,NULL);
```

Now, compile and run the program.

What happens?

OOPS- we have just introduced ***deadlock*** in the program !

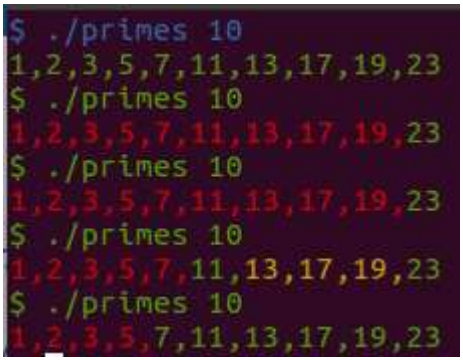
We are acquiring the lock, but we only release it when a prime number is found.

The findPrime() function exits whenever either of the following happens:

- A number is tested and found to not be prime – FALSE is returned
- A number is tested and found to be prime – TRUE is returned

Now, add a call to pthread_unlock() where the number is found to not be prime, before returning false.

Recompile and test the program.



Now, each time the program is run, it generates correctly the first 10 prime numbers in order.

Let's check the cost of correctness in terms of performance.

In the second terminal window, continue to monitor primes CPU utilization using "top":

```
$ top -d .2
```

Time how long it now takes to generate 50,000 prime numbers.

```
$ time ./primes 50000 >/dev/null
```

Make a note of how long this took, as well as the CPU utilization.

We have exposed a trade-off in this program between **performance** and **correctness**. With the prime number generation algorithm used, we can't obtain both speed and proper sequencing.

Without changing the prime number generation algorithm, contemplate ways to improve the performance of the program. What could be done?

To complete this tutorial, to obtain credit, log in to Canvas, and complete the **Tutorial 2 Quiz**.