Lab 7: Optimus Prime

Please **read this entire assignment**, **every word**, before you start working on the code.

This lab consists of multiple parts. **All Parts of this lab are August 3rd by midnight.** Submit a single gzipped tar file to **TEACH**. The single gzipped tar file should contain the all source files (C source [*.c and *.h] and the Makefile). **You must have a single Makefile to build this assignment. If your Makefile does not build a portion of the assignment, then a zero will be given for that portion of the assignment.** Do yourself a favor, write your Makefile first and keep it up to date.

Part 1: Optimus Prime (300 points)

Write a multi-threaded program (using PThreads on our Linux server os1) that will use the **Sieve of Eratosthenes** to find prime numbers. Here are a couple examples or animations of using the Sieve of Eratosthenes for find prime numbers.

https://www.youtube.com/watch?v=2SD2ILFj4h0 (soothing music too)
https://www.youtube.com/watch?v=V08g_lkKj6Q (this is tough to listen to)
http://www.hbmeyer.de/eratosiv.htm
https://www.geeksforgeeks.org/sieve-of-eratosthenes/

You must use a "bit-array" to keep track of the values you eliminate/identify as non-prime numbers. There is a fantastic write up on bit-arrays on this web site (you need to read and understand this web site):

http://www.mathcs.emory.edu/~cheung/Courses/255/Syllabus/1-C-intro/bit-array.html

Do not just assume that you should ignore the bit-array requirement and use an array of integers to keep track of the prime numbers. You will run out of memory. An array of int will take many times more memory. Though an array of char would work for a while, you'll still run out of memory. **The bit-array is much more memory efficient**. If the value in the bit array is set, it is not a prime number. Look at the web page listed above. Spend some time thinking about it and playing with some code. Read through the example on the Wikipedia page. You will start to see how the code will come together.

The description of building bit array in C shows 3 implementations of the various operations: Step-by-step, "Professional", and C macro. My recommendation is that you avoid the macro and even keep away from the "Professional." The Step-by-step is perfectly fine. If you really are the *Macro-mancer* you can research how you would expand the given macros to include requisite locking and unlocking.

The program must be named, primesMT. If you don't call it primesMT, the script for testing it will quickly fail. If I can't use the script to test your code, it's a zero for this portion of the assignment.

Since you will be using PThreads, you will also **have** to use one or more mutexes to protect shared resources (aka, portions of the bit-array). How and where you put your mutexes will affect the speedup you realize with your code. Do you want one mutex for the entire bit-array (**No**, you don't), one for each bit (**No**, you most certainly do not!!!), or something in between (**head nods vigorously**)? I chose to use a structure that looks a lot like this:

```
typedef struct BitBlock_s {
     uint32_t bits;
     pthread_mutex_t mutex;
} BitBlock_t;
```

It allowed me to use fine level locks, but not so fine that all the time was spent in getting locks. The locks were not at such a coarse level that the threads became effectively serialized waiting on all the other threads to complete. It is interesting to see how the performance of the program is affected by changing from a `uint32_t` to a `uint8_t`, `uint16_t`, or `uint64_t` for the `bits` data member of the structure. If you are feeling really adventurous, `gcc` does have an unsigned 128 bit integer type, `__uint128_t` (yes, the leading underscores are required). A little `Makefile` magic makes it easy to switch between the various unsigned `int` types in the structure. As for using signed or unsigned, don't use signed. BTW, the `uint32_t` (and other sized types) can be found in the `stdint.h` (I recommend including it in your code).

When you are done writing your program, run it varying the number of threads to get an idea where the speedup starts to flatten out or even decline as the number of threads is increased. If you don't see a speedup by changing the number of threads from 1-30, you are doing something wrong. You will also want to vary the upper bound on the prime numbers you generate to find out where creation of the threads startes to become helpful. If your code does not show any speed up when varied across multiple threads, spend the time getting your code to work better.

Your program must accept the following command line options. My code accepts more options than this, but these are the ones your code must accept and correctly handle.

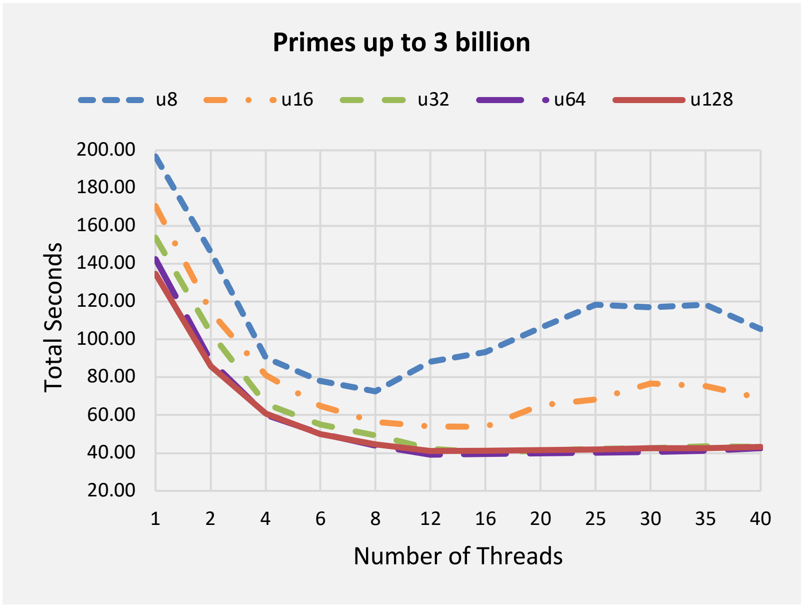
Option	Description
`-t #`	This is the number of threads that will be used to compute the number of prime numbers. If this value is not given on the command line, the default number of threads is 1.
`-u #`	This is the upper bound on how large a prime number you will calculate. You should compute all the prime numbers between 2 and the value given here. If this number is not given on the command line, compute the prime numbers up to 10,240. **Note that is not the how many prime numbers to generate, but the largest value to test for primality**.
`-h`	Show the help text and exit. Have something reasonable for this, like a listing of all the command line options. You can look at my program for an example.
`-v`	Verbose processing. This is really to help you follow what your code is doing. You need to accept this switch, have your code emit some diagnostics with this set. **All the messages from the verbose output should be sent to `stderr`, NOT `stdout`**.

I have placed a couple files containing prime numbers in my `Lab7` directory. **All the prime number output from your program should go to `stdout`, with a single prime number per line.** That will make it easy for you to compare your output to the sample prime number files I have provided. You can/should test your output against those files. I also have a `bash` script you will be able to use to check your code for correctness. It is called `testPrimes.bash` and is in the `Lab7` directory. I've used the script extensively against my code and it works well. If your code is so broken that is breaks the script, don't expect a reward of a good grade. You might enjoy looking through the script. If you have ideas for additions or corrections, please let me know. The script will provide you with feedback while it is running. Testing mine takes about 2 minutes.

Don't forget to use `-phtread` command line option with `gcc`, both for the compilation step and linking.

I ***LOVE*** this assignment. I have spent a lot of time (*waaaay* too much) playing around with just about every aspect of this code. Don't think that yours should perform the same as mine; and that is not how you will be graded. Make sure yours works and shows speedup when run with multiple threads. The biggest places students struggle on this assignment are:

1. Just wrapping their head around concurrently running threads
2. How to manage the locking and unlocking of mutexes.
3. The data structure that holds the bits.
4. Partitioning the calculation of prime numbers to distribute the workload to the threads.



My code is a twisted gnarly tangle of various `#ifdef` blocks from various experiments.

Run and test your code thoroughly using `valgrind` to identify and eliminate all memory leaks before submitting the assignment. Unaccounted memory leaks will be a 20% deduction per leak (the PThread library does have some leaks you cannot avoid). The false reports about leaks that come from PThreads look like this:

```
==46711== 21 bytes in 1 blocks are still reachable in loss record 1 of 4
==46711==    at 0x4C29C23: malloc (vg_replace_malloc.c:299)
==46711==    by 0x4019F39: strdup (strdup.c:42)
==46711==    by 0x4017183: _dl_load_cache_lookup (dl-cache.c:305)
==46711==    by 0x4008A69: _dl_map_object (dl-load.c:2258)
==46711==    by 0x40143E3: dl_open_worker (dl-open.c:232)
==46711==    by 0x400F913: _dl_catch_error (dl-error.c:177)
==46711==    by 0x4013CCA: _dl_open (dl-open.c:650)
==46711==    by 0x5491081: do_dlopen (dl-libc.c:87)
==46711==    by 0x400F913: _dl_catch_error (dl-error.c:177)
==46711==    by 0x5491141: dlerror_run (dl-libc.c:46)
==46711==    by 0x5491141: __libc_dlopen_mode (dl-libc.c:163)
==46711==    by 0x5147E52: pthread_cancel_init (unwind-forcedunwind.c:54)
==46711==    by 0x514801B: _maybe_pthread_cancel_init (unwind-forcedunwind.c:104)
==46711==    by 0x514801B: _Unwind_ForcedUnwind (unwind-forcedunwind.c:138)
```

Notice that they come from the `pthread_cancel_init` call. These are safe to ignore, unless you want to completely re-write PThreads (**Don't!**).

Finally, this is not the fastest way to generate prime numbers. There are many number theoretic methods that are faster, better, and scarier. Don't consider this the best, just a really fun example of generating a bunch of prime numbers with a multi-threaded program.

Steps to success with Optimus Prime:

1. Work out what you will use as a data structure. I described the `BitBlock_t` structure above and it works well, but it is not a requirement.
2. Process all the command line options with `getopt()`. All of them, just as you have been doing.
3. Allocate the data structure you will be using to hold the bit vector and the mutexes for the calculations. You'll need to decide if a bit set to 1 means the value is prime or is not prime. I used a 1 to indicate that the value identified is not prime.
4. Initialize all the values in the big data structure to their initial values. Set all the bits in the bit vector to 1 or 0 (however you decided in step 3). Initialized all mutexes.
5. Allocate space to store all the `tid` identifiers returned from `pthread_create()`. You'll need these when you call `pthread_join()`.
6. Decide what data you are going to pass to the thread function when you call `pthread_create()`. Remember, you get 1 (single, solitary, lonely) pointer to pass something, make it good.
7. Spin up the threads. The number of threads to start is based on the value passed with the -t command line option.
8. Calculate all the prime numbers up to the upper bound value (either the default or the one passed in with the -u command line option).
9. Call `pthread_join()` on all the threads.
10. Output all the prime numbers you generated. It is worth mentioning, with one exception, all prime numbers are odd. I just grant you the one even prime number and you can start determining primality with all odd values after that.
11. De-allocate any memory you allocated. You should destroy all the mutexes you created before de-allocating them.
12. Run the `testPrimes.bash` script.
13. Celebrate your success.

Part 3 – The `Makefile` (25 points)

You must have a single `Makefile` that compiles all the programs. If you do not have a `Makefile` that builds all programs, it will put a major dent in your grade for the assignment (think zero for each part not compiled). Your code must compile without any errors or warnings from `gcc`. Do not adorn your calls to `gcc` with any `-std=`… options. **Be sure your code does not generate any errors or warnings when compiled**. Hunt down and fix all warnings in your code. I will deduct 20% for each warning your code produces when compiled.

You must use the following `gcc` command line options in your `Makefile` when compiling your code (make your life easier, use variables).

```
-Wall
-Wshadow
-Wunreachable-code
-Wredundant-decls
-Wmissing-declarations
-Wold-style-definition
-Wmissing-prototypes
-Wdeclaration-after-statement
-Wno-return-local-addr
-Wuninitialized
```

Your `Makefile` must include the following targets:

- `all` – should build all out-of-date programs and prerequisites.
- `clean` – clean up the compiled files and editor chaff.
- `primeMT` – linking the part 2 program, rebuilding the `primeMT.o` file, if necessary
- `primeMT.o` – build the `primtMT.o` file, if necessary
- if your `primeMT` is built from multiple C files, you'll need to have additional targets to support the additional C files.

Final note

The labs in this course are intended to give you basic skills. In later labs, we will ***assume*** that you have mastered the skills introduced in earlier labs. **If you don't understand, ask questions.**