Homework 5: Threads and shared memory

Due: Monday 3 June 2013, 23:30:00 Pacific USA time zone. That is the Monday of Dead Week.

Points on this assignment: 150 points with 15 bonus points available.

Work submitted late will be penalized as described in the course syllabus. You must submit your work twice for this and all other homework assignments in this class. Ecampus wants to archive your work through BlackBoard and EECS needs you to submit through TEACH to be graded. If you do not submit your assignment through TEACH, it cannot be graded (and you will be disappointed with your grade). Make sure you submit your work through TEACH. Submit your work for this assignment as a *single bzip file* through TEACH. The same single bzip file should also be submitted through BlackBoard.

Place all of the files you produce for this assignment in a single directory, maybe something called Homework5.

This assignment explores the use of threads, using the PThreads library, and shared memory. You want to make sure you have read the pthreads tutorial, TLPI chapters 29 and 30 (from week 6). TLPI chapters 53 and 54 will also be necessary (week 7 readings).

You have 3 weeks to complete this assignment. This is the last programming assignment in this class.[1]

1. **5 points**. When you are ready to submit your files for this assignment, make sure you submit a *single bzip file*. Review homework #1, problem #1 if you need a refresher on how to do this. If your file is not a single bzip file, you cannot receive points on this homework assignment.

2. **145 points**. For this project, you will be required to find all prime numbers within the range up to a 32 bit unsigned integer. Your solutions must be written in C and compile and run on the CS311 VM or `os-class` server. There are some specific requirements for the purposes of this project:

    1) You will be writing 2 slightly different versions of this project
        a) A **threaded** version with implicit memory sharing (the heap)
        b) A version using **multiple processes** and **shared memory**
    2) Call your threaded portion of the assignment primePThread.
    3) Call the multi-process portion of the assignment primeMProc.

[1] And there was much rejoicing.

4) For the threaded portion of the assignment, you are required to use the pthreads library.
5) The multi-process portion will fork() the appropriate number of child processes. The parent process will create a shared memory object to which the sub-processes will attach. You don't need to use pipes to communicate with child sub-processes.
6) In both versions, you are required to make use of a bit array (sometimes called a bitmap or bit vector), which will be used to store the results of the prime number checking.
   a) For the threaded version, the bit array will be in the heap.
   b) For multi-process/shared memory portion of the assignment, the bit array must be located in a shared memory object (use POSIX shared memory, TLPI chapter 54).
   c) The bit array allows you to store the collection of numbers in a much more compact form. You do not want to try to allocate an array of `UMAX_INT` integers.
7) By default, both applications should write all found prime numbers to stdout. The numbers should be written to stdout in order with the smallest number being written first.
8) You cannot brute force this algorithm. You must use a more sophisticated method for obtaining the list of prime numbers. The Sieve of Eratosthenes was given as the first assignment in this class and is acceptable for this assignment. A brute force algorithm just looks at ALL numbers up to the max. You need to have a better algorithm than that. There are a lot of algorithms out there, choose one and use it.

Both versions of your applications should allow 3 command line arguments:

1. **–q** -- Be quiet! In order to check the correctness of your applications, you'll need to generate the actual list of prime numbers. However, sending all that output to the terminal (or a file), will slow down your application. Use this option to create the numbers, but not output them. You can use this option to collect timing values (after you know it produces correct results).
2. **–m #** -- The maximum size of the prime number for which you check. If you do not specify the maximum, use the maximum value of an unsigned 32-bit integer (`UINT_MAX`).
3. **–c #** -- The concurrency (parallelism) to use in the application. If you do not use this option on the command line, the default level of concurrency (number of threads or processes) should be 1 (one). You should test your code with the concurrency ranging from 1 to 10 (threads/processes).

An example of how the command line options can be used is:

```
primePThread –q –m 500000000 –c 5
```

The above command would run the multi-thread portion of the assignment. It does not actually write the found prime numbers to stdout. It finds all prime numbers less than 500,000,000 and uses 5 threads.

If you want/need to have additional command line options (like –v for verbose debugging information), that is fine, but you must support the above 3 options.

These web pages can be used to help you find a list of prime numbers for testing the results of your code. Of course, you can also use the Python code from the assignment #2 to generate lists of prime numbers.

- http://www.mathsisfun.com/numbers/prime-number-lists.html
- http://primes.utm.edu/lists/small/millions/
- http://www.primos.mat.br/indexen.html

Working with bit arrays is also probably new to most of you, so here are a couple pretty nice descriptions (with code):

- http://www.mathcs.emory.edu/~cheung/Courses/255/Syllabus/1-C-intro/bit-array.html
- http://www.tek-tips.com/faqs.cfm?fid=3999

Provide timing numbers based on the maximum size of prime number (the –m # command line option) and the number of threads/sub-processes (the –c # command line option). Vary the number of threads/processes from 1 to 10 (more than 10 is okay, but don't go beyond 20). Vary the maximum size of the prime number up to `UINT_MAX` (increments of 500 million would be okay, but be careful that you don't overflow the int). These timings should be plotted in a fashion that makes sense. Use MS Excel or LibreOffice Calc to produce a spreadsheet of your timing results and create a graph showing time on the vertical axis and input size as the horizontal axis. 15 points (of the 145).

You should write-up how you partition the problem space so that each process or thread gets (nearly) the same amount of work. Factoring large numbers takes more time than factoring small numbers, so you don't want to give all the smaller numbers to one process/thread. 15 points (of the 145).

If you are collecting timing runs on a VM, you may see little to no speedup using multiple processes/threads. You'll need to run the applications on a system with multiple physical processors, such as `os-class`, for a true measure of speedup due to multi threading/processing. If you have your own beefy Linux system that has hyper-threaded CPUs, it will be interesting to see how the speedup varies for threaded and multi-process portions of the assignment. Timing can be collected using the built in shell command `time`, or you can place timing collection functions within your code.

For up to **15 points extra credit**, create a Bash shell script (`primeTest.bash`) that will test and validate your code. It should:

1. Build both portions of the assignment, `primePThread` and `primeMProc` (using your `Makefile`).
2. Validate the output of prime number against files containing prime numbers. You should validate against the first 1 million prime numbers as the minimum.
3. Run both portions of your assignment (without output, the –q option), varying the number of threads/sub-processes (the –c # command line option) from 1 to 10, and collect timing numbers. You should vary the –m option for at least 10 points for each application. Make sure to include `UMAX_INT` as a timing point.
4. Create a csv file containing all of the timing results.

The point distribution for this assignment is:

Task	**Number of points**
Just for your C code compiling (both portions must compile)	5
Makefile that compiles both portions of the assignment	10
Excel/LibraOffice spreadsheet with timing graph (as described above)	15
A write-up on how you partitioned the work to balance the factoring across all processes/threads (as described above)	15
The working threaded application	50
The working multi-process application using shared memory.	50
Total	**145 points**

Things to include with the assignment (in a single bzipped file):

1. C source code (.c and .h files) for the each portion of the assignment to the posed problems (all files).
2. A Makefile to build both portions of the assignment.
3. A file describing how you ran your tests an how you partitioned the problem space to evenly distribute work on the threads/processes.
4. An Excel/LibraOffice file showing the timing results.
5. The extra credit script (if you do that portion).

Please combine all of the above files into a single bzip file prior to submission.