Operating System
CS 370
Spring 2019

Project 5
Sophie Germain Prime Numbers

Due Date: 11:59 pm,27th March 2019
Must be submitted before deadline

Total Points: 100

Teaching Assistant:
Office Hours:
Graduate Assistant: Pradip Maharjan
Office Hours: MoWe 12 PM - 2 PM
Teaching Assistant: Luis Maya-Aranda
Office Hours: MoWe: 1 PM - 4 PM
TuTh: 10 AM - 12 PM

Course Web Page: http://osserver.cs.unlv.edu/moodle/

Objective: Become familiar with multi-threaded programming and synchronizing access to shared data with mutual exclusion. Explore different implementations of mutual exclusion and their performance.

Introduction:
In number theory, a prime number p is a Sophie Germain prime if 2p + 1 is also prime. The number 2p + 1 associated with a Sophie Germain prime is called a safe prime. For example, 11 is a Sophie Germain prime and $2 \times 11 + 1 = 23$ is its associated safe prime. Sophie Germain primes are named after French mathematician Sophie Germain, who used them in her investigations of Fermat's Last Theorem. Sophie Germain primes and safe primes have applications in public key cryptography and primality testing. It has been conjectured that there are infinitely many Sophie Germain primes, but this remains unproven*.

Submission
Your program should compile using following command:

```
gcc -std=c11 prime.c -o prime -pthread -lm
```

or

```
gcc-5 -std=c11 prime.c -o prime -pthread -lm
```

- Submit prime.c (not c++, just one file, not zipped) via the class web page (assignment submission link) by 11:59 pm, 27th March 2019.

We will check for code similarities using MOSS. Please do not download code from the internet or distribute solutions anywhere including Github, Bitbucket and any other public code publishing platforms.

Submissions received after the due date/time will not be accepted.

* https://en.wikipedia.org/wiki/Sophie_Germain_prime

Assignment

In this assignment, you will implement some solutions to the mutual exclusion problem and test their performance with a simple driver program. The driver program will find all Sophie Germain primes up to a given input value, **maxValue**, and save them to a shared (global) array. After all primes have been found, the program will print all the primes found as well as some statistics. The program will time how long it takes to test all the numbers using a C11 clock. The test program will take the following input:

./programName [threadCount] [maxValue] [lockType] [visualization]

Where **threadCount** is the number of threads to create, **maxValue** is the bound for the prime search (and should be an unsigned long), **visualization** is either 0 or 1 which will print a display of the lock's state and will only be used for Peterson's N-thread algorithm, and **lockType** is a number in {0, 1, 2, 3}:

$$lockType = \begin{cases} 0, & will\ not\ use\ Mutexes \\ 1, & will\ use\ Peterson's\ N-thread\ Filter\ Mutex \\ 2, & will\ use\ Test-And-Set\ Mutex \\ 3, & will\ be\ Wait\ Free \end{cases}$$

Threads and Thread Functions

This assignment will use the C pthread library to create threads that will run a function that you will write. Therefore, you must include the library **pthread.h** in your program. You can create a thread using the function pthread_create which has the prototype:

**int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
void *(*start_routine), void *arg);**

This will create a thread and store a reference to the thread in the location pointed to by the pthread_t ***thread** variable. The attributes variable may be set to define various properties of the thread, such as its scheduling policy, scheduling priority, stack size, stack address, etc. However, you can use the default attributes by passing **NULL** for this argument. The **start_routine** argument is the name of the function you want your thread to run. The **arg** argument is a single argument cast as type **void ***, which will be passed to the **start_routine** as the sole argument for that function. You may pass **NULL** if your start routine does not accept arguments. If you want the start routine to take more than one argument, you should create a struct to hold the arguments you wish to pass and then pass the struct through **arg**. An example function call is:

pthread_create(threadReference, NULL, threadFunction, (void *)myArg);

The start routine, also called the thread function, should always call **pthread_exit(void *retval)** once the function is complete which terminates the calling thread. The process that calls **pthread_create** should wait for the threads it has created to exit before exiting itself, by calling **pthread_join**. Joining with a thread means the caller will wait until that thread has terminated. The function **pthread_join** has the following function prototype:

int pthread_join(pthread_t thread, void **retval);

Where **thread** is the reference of the thread you wish to join with, and **retval** is a location of the return value for that function cast as a pointer to void.

C11 Clock and time.h Library

In previous classes, you may have captured the runtime of a program using the Linux **time** command. However, for this program, we only want to capture the time that it takes for the threads to find the prime numbers. Therefore, we will use the clock provided in the C **time.h** library. To do this, you can create a variable of type **clock_t** which can store the processor's clock time. Then, grab the clock time before creating any threads and after joining all of them using the **clock()** function which takes no arguments. Subtract the difference in clock times and you have the clock ticks it took to perform the task. The **time.h** library also provides a handy macro **CLOCKS_PER_SEC** which represents the number of processor clocks per second to convert ticks into seconds.

Sophie Germain Primes

A prime number, **p**, is defined as an integer with exactly two factors, namely 1 and p. A Sophie Germain prime is a special type of prime number, **p**, with the following properties:

1. **p is prime and**
2. **2p+1 is prime**

You will need to create a function to determine whether a given number is prime or not, such a test is called a primality test. You do not need to make your algorithm more efficient than $\boldsymbol{O(n)}$, but if you are interested, the following details may help you make an efficient primality test. There are two types of primality tests: deterministic and probabilistic. A deterministic primality test is one such that the function will determine if the input is prime or not, by testing all possible factors of the given input. It is fairly simple to create a deterministic primality test that runs in $\boldsymbol{O(\sqrt{n})}$ time, where **n** is the input to be tested. Several other optimizations exist, and the Wikipedia article for primality tests* is a good resource.

However, for large input, **n**, such as a potential prime with 1024 bits (n is on the order of 2^{1024}, so a deterministic test is about $O(\sqrt{2^{1024}} = 2^{512})$) which is used to generate a public key in **RSA-2048**, a deterministic test is still very computationally expensive, so probabilistic primality tests are used for practically all cryptography applications. A probabilistic primality test is one such that the function will determine if the input is probably prime or not, usually with a high degree of confidence. The Miller-Rabin primality test and Fermat primality tests are examples of a probabilistic primality tests. They are actually compositeness tests (recall that a number that is not prime or one is called a composite number), rather than true primality tests, so they test whether the input is composite. The Miller-Rabin test takes two inputs: the potential prime, **p**, and a parameter to determine the accuracy of the test, **k**, and it is commonly implemented as the primality test for many math and cryptography libraries because it has $\boldsymbol{O(k \log^3 p)}$ time complexity.

Parallelizing the Work

A naïve approach to parallelization is to split the numbers to be tested into **threadCount** sets with $\frac{maxValue}{threadCount}$ elements to test each. So, if the maximum value is 100 and there are 4 threads, it is tempting to split the work into [1,25], [26,50], [51,75], [76,100]. However, while each thread has the same number of numbers to check (25), since larger numbers take longer to check, the work is not split evenly.

* https://en.wikipedia.org/wiki/Primality_test

Instead, we will use a shared counter to coordinate the work. A thread will acquire the mutex, also known as a lock, for the counter, then get and increment the counter, and then unlock and check if the number is prime. If the number is prime, then the program will place the prime into the next element of the shared prime array. To do so, acquire a different mutex for the array and its index, add the new prime, and then unlock. In this way, each thread can get a new number to test as soon as it is finished with its previous number, and fast threads do not need to wait on slow threads.

Creating Mutexes using the C11 stdatomic Library

You will need at least gcc version 5 (the current version is gcc-8.2) for this assignment. The stock version of gcc that ships with Ubuntu 14.04 is gcc version 4.8.4, which added full support for C11 and the stdatomic library but forgot to include the stdatomic.h library in the install. More information can be found here: http://tuxamito.com/wiki/index.php/Installing_newer_GCC_versions_in_Ubuntu.

You can check your version by typing the command: **gcc -v**

You can install gcc version 5 by running the commands:

```
sudo add-apt-repository ppa:ubuntu-toolchain-r/test
sudo apt-get update
sudo apt-get install gcc-5
```

And then run gcc-5 by using the command:

```
gcc-5 -std=c11 prime.c -o prime -pthread -lm
```

You may recall from class that an atomic data type is one such that supported operations occur in one hardware step. Therefore, if multiple threads share access to an atomic data type, then supported operations on that type will not be subject to race conditions; each operation will occur in a sequential order. C11 provides a library, **stdatomic.h***, that gives atomic data types such as atomic integers, and access to stronger atomic instructions, which we will use in this assignment.

Firstly, an atomic type in C11 is a bool/char/short/int/long, signed or unsigned, that supports load (read), store (write), fetch-and-add, fetch-and-subtract, fetch-and-bitwise OR/exclusive OR/AND, and compare-and-exchange operations. The library reference* linked in the footnotes shows the operations which come in two types: explicit and not explicit. The explicit version of each operation allows you to specify the memory order for the instruction. The other version of each operation, **which is what we will use**, enforces a sequentially consistent memory order. The sequentially consistent memory order is the strongest memory order we can apply, and it forces each atomic operation to occur in the same order to all threads, whereas more relaxed conditions can allow for some reordering of instructions from different thread's perspectives of the cache which we do not want. If you are interested in this topic, take Dr. Datta's CS 789 course on Multicore Computing.

In class you discussed at least one algorithm for mutual exclusion. In this assignment we will use a generalization of Peterson's algorithm for mutual exclusion to create a mutex for n threads. First, we will discuss Peterson's original algorithm for 2 threads, and then we will show a generalization called Peterson's n-thread filter algorithm which can be used for n threads. The benefit of Peterson's algorithm is that it is implemented with only atomic variables equipped with load and store operations.

* *https://en.cppreference.com/w/c/atomic*

Peterson's 2-Thread Mutual Exclusion Algorithm

Peterson's 2-thread algorithm uses three atomic variables: two flags, written as an array and a victim variable. There are two threads, one with thread ID 0, thread 0, and the other with thread ID 1, thread 1. The algorithm implemented in C11 to initialize, acquire, and release the lock is as follows:

```
struct peterson2ThdLock{
  atomic_int flag[2]; //initialized to 0
  atomic_int victim;
}

void initializeLock(struct peterson2ThdLock* lock){
  //To be run when the lock is created:
  atomic_store(&flag[0], 0);
  atomic_store(&flag[1], 0);
  atomic_store(&victim, 0);
}

//threadID must be either 0 or 1
void lock(struct peterson2ThdLock* lock, int threadID){
  //Declare that we are interested in using the critical section
  atomic_store(&lock->flag[threadID], 1);
  //Set ourselves as the victim
  atomic_store(&lock->victim, threadID);
  //If the other thread is also interested and we are the victim, wait
  while(atomic_load(&flag[(threadID+1)%2]) == 1 && atomic_load(&victim) == threadID){
    //do nothing AKA spin AKA busy-waiting
  }
  //If we get here, we have acquired the lock because
  //Either the other thread is not interested, or they are the victim
}

void unlock(struct peterson2ThdLock* lock, int threadID){
  //Declare that we are no longer interested in the lock
  atomic_store(&lock->flag[threadID], 0);
}
```

The lock algorithm works by declaring interest in acquiring the lock and then yielding to any thread that has declared interest before us by becoming the victim. Then spin until either the other thread is not interested, or we are no longer the victim. We must declare our flag atomically because if the other thread is spinning at the while loop, it will be reading our flag and we need to ensure that the variable always contains the most up to date value. Similarly, for the victim variable. If the victim variable is not set atomically, another thread may not see our store operation due to delays and/or reordering in the underlying memory system or reordering and/or optimization by the compiler. The 2-thread algorithm provides starvation freedom and 2-bounded bypass which is not guaranteed in the filter algorithm.

Try to convince yourself that Peterson's 2-Thread algorithm is a good mutual exclusion algorithm. There are three properties that make a mutual exclusion algorithm good:

Safety (Correctness) Property:

1. **Mutual Exclusion**: Critical sections (CS) of different threads never occur concurrently. For two threads, thread 0 and thread 1, either CS_0 precedes CS_1 or CS_1 precedes CS_0.

Liveness (Progress) Properties:

2. **Freedom from Deadlock**: If some thread attempts to acquire the lock, then some thread will succeed in acquiring the lock. If thread A calls **lock()** but never acquires the lock, then other threads must be completing and infinite number of critical sections.
3. **Freedom from Starvation**: Every thread that attempts to acquire the lock will eventually succeed. That is, every call to **lock()** eventually returns.

Starvation freedom implies deadlock freedom, so if the algorithm has starvation freedom then it will have deadlock freedom. However, many mutual exclusion algorithms do not have starvation freedom, and instead make do on the idea that starvation is highly unlikely in practice. To reason about these algorithms, imagine both threads call **lock()**. They could stop at any point in the algorithm and begin again after any amount of delay. Therefore, the instructions can be reordered in any way between the two threads, but each thread must execute the algorithm's instructions in the order that they are written (called the **program order**). Try to answer the following questions for yourself:

1. Can two threads get into the critical section at the same time (breaking mutual exclusion)? What would the order of the two threads' code look like if they could?
2. Can both threads get stuck causing neither to ever get into the critical section (deadlock)? What if only one thread runs the lock algorithm, can it get stuck since it sets itself as the victim?
3. Can one thread get stuck forever while the other thread repeatedly completes the lock algorithm (starvation)?

Try to keep this way of reasoning about a lock algorithm in your mind when we discuss the next locks.

Peterson's n-Thread Filter Algorithm

In the 2-Thread algorithm, the victim variable is like a waiting room. If both threads are interested in acquiring the lock, whichever thread was last to set the victim variable must wait in the waiting room until the other thread completes the call. The n-thread generalization uses this idea. In Peterson's n-thread filter algorithm, there n-1 waiting rooms called *levels*. Threads must travel through all levels to acquire the lock. The levels have two properties:

1. At least one thread trying to enter level *k* succeeds.
2. If more than one thread is trying to enter level *k*, then at least one is blocked.

Intuitively, n threads can join level 0, n-1 in level 1, n-2 in level 2, n-k in level k, until 2 threads in level n-2 and 1 thread in level n-1 (critical section). Each level has one victim, so each level is a copy of the 2-thread algorithm. Each thread has its own entry in the level array to indicate its level. To unlock, just set the thread's entry in the level array to zero (just like Peterson's 2-thread algorithm).

The algorithm for Peterson's n-thread filter algorithm is:

```
Algorithm Peterson's n-thread Filter Lock
Inputs: thread ID
Outputs: none
me ← thread ID
for i = 1 to n do
  level[me] ← i
  victim[i] ← me
  while ($\exists k \neq me, (level[k] \geq i$
            $\&\& \; victim[i] = me)$);
end for
```

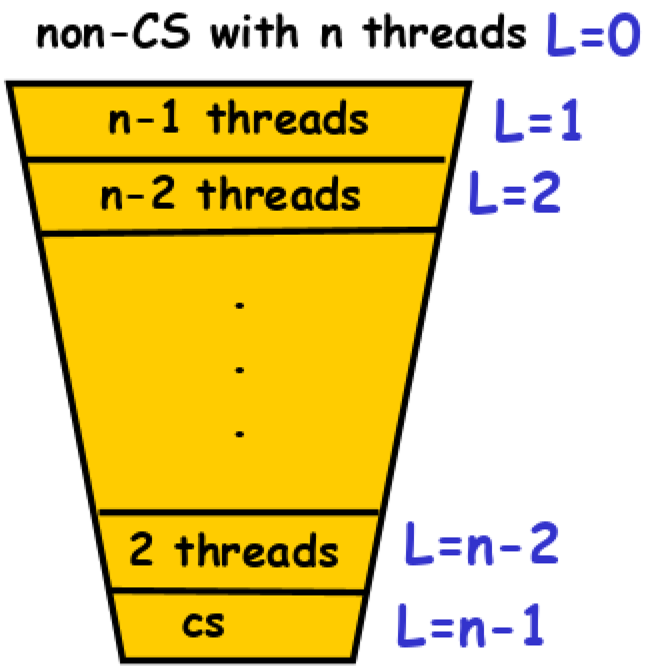
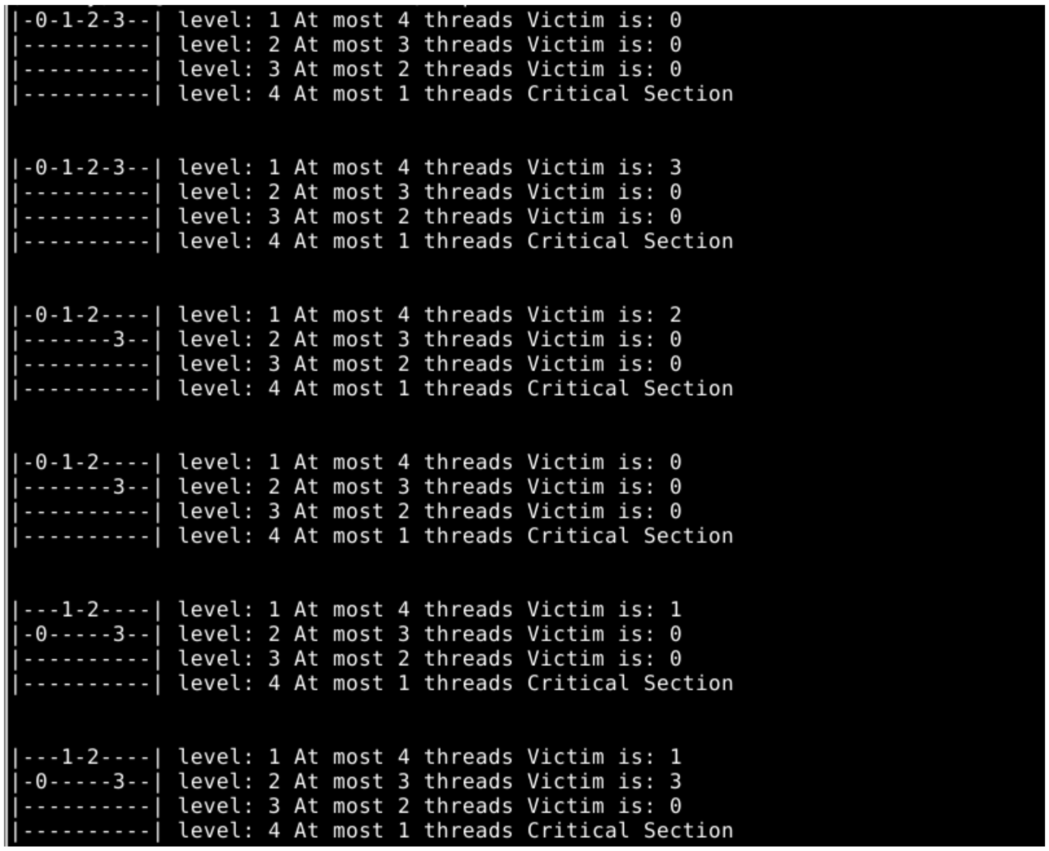


Figure 1(right): The filter algorithm has n-1 levels. Image from the Art of Multiprocessor Programming.

To help you visualize this lock algorithm, if the **visualization** input parameter is **1**, then your program will print the lock's state whenever a change in the level array occurs and print whenever a thread enters and leaves the critical section. After a write to the level array, you should make a **sleep(1)** call to slow the execution down. However, you will find that multiple threads may print at the same time, so in order to visualize the algorithm, use another lock to coordinate access to stdout (the terminal). The following is an example of the visualized output of this lock:



```
|-0-1-2-3--| level: 1 At most 4 threads Victim is: 0
|----------| level: 2 At most 3 threads Victim is: 0
|----------| level: 3 At most 2 threads Victim is: 0
|----------| level: 4 At most 1 threads Critical Section


|-0-1-2-3--| level: 1 At most 4 threads Victim is: 3
|----------| level: 2 At most 3 threads Victim is: 0
|----------| level: 3 At most 2 threads Victim is: 0
|----------| level: 4 At most 1 threads Critical Section


|-0-1-2----| level: 1 At most 4 threads Victim is: 2
|-------3--| level: 2 At most 3 threads Victim is: 0
|----------| level: 3 At most 2 threads Victim is: 0
|----------| level: 4 At most 1 threads Critical Section


|-0-1-2----| level: 1 At most 4 threads Victim is: 0
|-------3--| level: 2 At most 3 threads Victim is: 0
|----------| level: 3 At most 2 threads Victim is: 0
|----------| level: 4 At most 1 threads Critical Section


|---1-2----| level: 1 At most 4 threads Victim is: 1
|-0-----3--| level: 2 At most 3 threads Victim is: 0
|----------| level: 3 At most 2 threads Victim is: 0
|----------| level: 4 At most 1 threads Critical Section


|---1-2----| level: 1 At most 4 threads Victim is: 1
|-0-----3--| level: 2 At most 3 threads Victim is: 3
|----------| level: 3 At most 2 threads Victim is: 0
|----------| level: 4 At most 1 threads Critical Section
```

Test And Set Lock Algorithm

A Test-And-Set (TAS) lock uses a stronger primitive instruction than a Read-Write lock like Peterson's 2-thread or n-thread filter algorithm. In this way, it avoids having the O(n) space complexity of atomic variables used in a Read-Write lock. For the Test-And-Set lock, use a single atomic Boolean lock field and an **atomic_fetch_or()** instruction to write a **1** to the lock field. Busy wait, repeating the **atomic_fetch_or()** call as long as the call to **atomic_fetch_or()** returns **true**. If the call returns **false**, then the lock is acquired and your lock algorithm should no longer spin. To unlock, just atomically write a 0 to the lock field. This algorithm is very simple, clearly maintains mutual exclusion and deadlock freedom, but is it starvation free?

No, the Test-And-Set lock is **not** starvation free, since a thread can fail to be the successful writer to the lock infinitely many times if another thread succeeds infinitely often. However, in a fair scheduling environment, starvation is highly unlikely to occur. After implementing the Peterson n-thread algorithm and then being presented with an infinitely simpler and more efficient lock, you may wonder if the Test-And-Set lock has any drawbacks at all. Besides not being starvation free, the Test-And-Set lock also experiences **worse** performance as the number of threads using the lock increases. This is due to a cache coherency problem. You may also notice that the runtime of Peterson's n-thread algorithm becomes worse as the number of threads increases. This is due to the complexity of the algorithm as well as many atomic reads of (cache reads) that only increase as the number of threads increases since we need **2n** atomic variables to implement Peterson's n-thread filter lock. While the Test-And-Set lock does significantly improve on this, there are more complicated locks that succeed in reducing the cache coherency problem which are beyond the scope of this course, such as Queue Locks.

You do **not** need to implement a visualization for this lock.

Wait Free Algorithm

We now turn our attention to an algorithm with no locks **and** no race condition. A **wait free** algorithm is one such that there are no locks and no race conditions. This means that we maintain correctness, while also allowing every thread to make progress when it is scheduled by the operating system. This means that a wait free approach can make progress in the absence of a fair scheduler. Of course, there is a clear benefit to making a wait free algorithm.

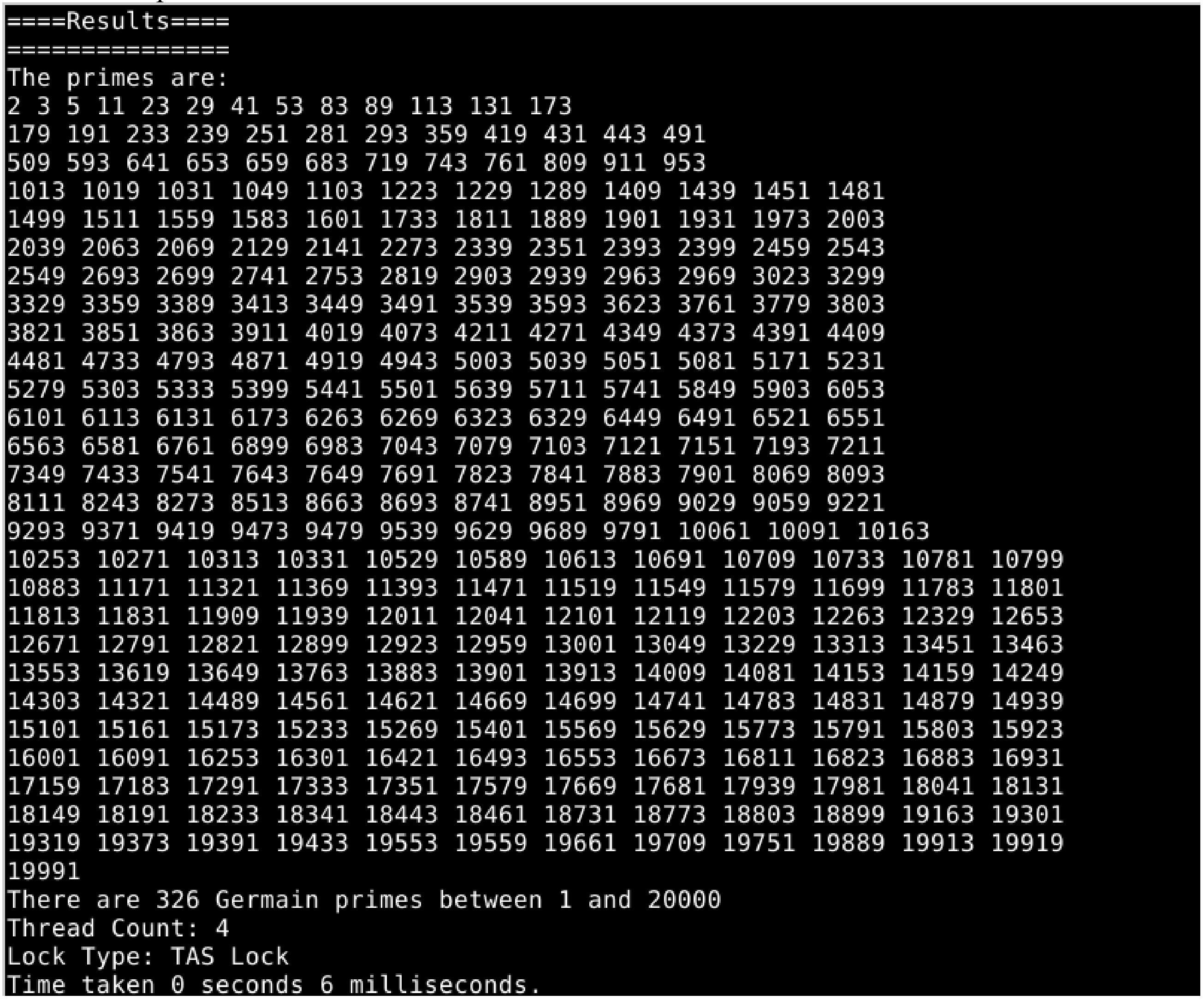
Making the thread function wait free relies on one key observation: if you can fetch and increment each counter in one atomic step, then the algorithm doesn't need locks. There are two counters, one for the number to check and another for the next available spot in the primes array. Make both counters atomic and use **fetch-and-add** instructions to access them for the wait free algorithm.

You do **not** need to implement a visualization for this approach.

Testing Your Program

To test your program, you can run the test script given on the class website. It will run each algorithm with a varying number of threads and your program will print the run time in each case. The following is an example output for your program using the Test-And-Set lock.

Command: ./primes 4 20000 2 0



```
====Results====
===============
The primes are:
2 3 5 11 23 29 41 53 83 89 113 131 173
179 191 233 239 251 281 293 359 419 431 443 491
509 593 641 653 659 683 719 743 761 809 911 953
1013 1019 1031 1049 1103 1223 1229 1289 1409 1439 1451 1481
1499 1511 1559 1583 1601 1733 1811 1889 1901 1931 1973 2003
2039 2063 2069 2129 2141 2273 2339 2351 2393 2399 2459 2543
2549 2693 2699 2741 2753 2819 2903 2939 2963 2969 3023 3299
3329 3359 3389 3413 3449 3491 3539 3593 3623 3761 3779 3803
3821 3851 3863 3911 4019 4073 4211 4271 4349 4373 4391 4409
4481 4733 4793 4871 4919 4943 5003 5039 5051 5081 5171 5231
5279 5303 5333 5399 5441 5501 5639 5711 5741 5849 5903 6053
6101 6113 6131 6173 6263 6269 6323 6329 6449 6491 6521 6551
6563 6581 6761 6899 6983 7043 7079 7103 7121 7151 7193 7211
7349 7433 7541 7643 7649 7691 7823 7841 7883 7901 8069 8093
8111 8243 8273 8513 8663 8693 8741 8951 8969 9029 9059 9221
9293 9371 9419 9473 9479 9539 9629 9689 9791 10061 10091 10163
10253 10271 10313 10331 10529 10589 10613 10691 10709 10733 10781 10799
10883 11171 11321 11369 11393 11471 11519 11549 11579 11699 11783 11801
11813 11831 11909 11939 12011 12041 12101 12119 12203 12263 12329 12653
12671 12791 12821 12899 12923 12959 13001 13049 13229 13313 13451 13463
13553 13619 13649 13763 13883 13901 13913 14009 14081 14153 14159 14249
14303 14321 14489 14561 14621 14669 14699 14741 14783 14831 14879 14939
15101 15161 15173 15233 15269 15401 15569 15629 15773 15791 15803 15923
16001 16091 16253 16301 16421 16493 16553 16673 16811 16823 16883 16931
17159 17183 17291 17333 17351 17579 17669 17681 17939 17981 18041 18131
18149 18191 18233 18341 18443 18461 18731 18773 18803 18899 19163 19301
19319 19373 19391 19433 19553 19559 19661 19709 19751 19889 19913 19919
19991
There are 326 Germain primes between 1 and 20000
Thread Count: 4
Lock Type: TAS Lock
Time taken 0 seconds 6 milliseconds.
```

Your program should also **handle input error checking** as appropriate.