Lab 2 Report

1. **Name and number of the lab:** Lab 2 – Kernel module concurrent memory use
2. **Names of the people:** Sayali Patil (sayali.patil@wustl.edu)

3. **Attribution of sources for any materials that were used in (or strongly influenced) your solution:**

   i. LKD book
   ii. Barriers: https://stackoverflow.com/questions/26231727/what-is-the-correct-way-to-implement-thread-barrier-and-barrier-resetting-in-c
   iii. https://stackoverflow.com/questions/25319825/how-to-use-atomic-variables-in-c
   iv. For kernel data structure: https://www.tldp.org/LDP/tlk/ds/ds.html
   v. spinlock: https://docs.oracle.com/cd/E26502_01/html/E35303/ggecq.html
   vi. https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes
   vii. Kernel threads: https://lwn.net/Articles/65178/
   viii. Kernel threads: http://www.qnx.com/developers/docs/6.5.0/index.jsp?topic=%2Fcom.qnx.doc.neutrino_getting_started%2Fs1_procs.html
   ix. Struct timespec: https://www.gnu.org/software/libc/manual/html_node/Elapsed-Time.html

4. **Module Design and Implementation:**

Barrier synchronization function:

To implement two barriers, I defined two barrier functions inside which I kept a track of number of threads reached inside each barrier. If the number of threads reached is equal to the num_threads then I set the atomic counter associated with the barrier to 1. To protect the data and to avoid contention I used spinlock inside each on of the barrier functions. These barrier functions were then called inside the thread_do_work function.

Function to mark non-prime numbers:

I used spinlock here as well to protect the code section while updating the current position variable and array of integers. I wrote the function based on Sieve Eratosthenes algorithm where we mark the non-prime numbers in the array of integers. All the defined threads call this function until all the non-prime numbers are marked and we keep track of those cross-outs performed by every single thread. For normal version, I protected the integers array using a spinlock but for atomic version the protection was offered by converting the integer array into atomic type.

Thread_do_work function:

Inside this function I first called the barrier function for barrier 1 and then non-prime marker function and then barrier function for barrier two and saved this state in the atomic variable defined to check whether the prime processing has finished or not.

Module Initialization Function:

I first initialized all the module variables to zero and then tried allocating memory to all the defined variables. If at any point memory allocation was failed for any variable, then I freed the memory allocated to other variables earlier and set all the variables back to zero. Also, I set the counter variables to zero and the integer array from 2 to the upper bound. And then spawned as many numbers of threads as passed from the command line argument.

Module Exit function:

Inside this function, all the computation for the total number of prime and non-prime numbers, crossouts and unnecessary crossouts, initialization and prime processing time was calculated and then in the end all the memory allocated to module variables was freed using kfree() function.

5. **Module Performance:**

When I tried running the code it worked fine for single thread, but it gave an error for multithreaded module. For single thread, I observed that completion time required by the non-atomic module was less than the time required by atomic module. For upper bound ranging from 1000-5000, I observed that the completion time increased continuously till upper bound 4000 and then dropped again for bound 5000, in both the cases.

Number of unnecessary cross-outs remained the same in case of both atomic and non-atomic module for every upper bound and it increased with the increase in upper bound.

I am attaching the graphs for single threaded environment herewith the report as well a screenshot of error I was getting when I tried to increase the number of threads.

6. **Names of the files:**
   i. completion_time_primes.png
   ii. unnecessary_crossouts_primes.png
   iii. completion_time_primes_atomic.png
   iv. unnecessary_crossouts_primes_atomic.png

7. **Names of any other files:** None

8. **Insights/ Questions:**

Even though I couldn't resolve the issue and run the program for multiple threads, according to the theoretical understanding, I feel that module efficiency and performance

will not change much after increasing the number of threads beyond 4, as we have a constraint on our pi's cores that are restricted to 4.

9. **Any suggestion:**
   Can we use kernel shark to observe the performance of all the threads also how they preempt each other, if they do at any point.

10. **Approximate amount of time spent on the assignment:**

    I accidentally deleted my code twice and as I was working alone on this lab, it took me around 60-70 hours in total.