1. CSE 422S: Lab 2 Kernel Module Concurrent Memory Use Report

2. Names and Emails

Jesse Huang: jessehuang@wustl.edu
Zhengliang Liu: zhengliang@wustl.edu
Elaine Cole: elainemcole@wustl.edu

3. Significant Sources of Influence

LKD by Robert Love

4. A Module Design and Implementation

Overview

The most noticeable design choice was the addition of a struct task_struct ** , threads, that stored each thread's respective task_struct*. We chose to create this pointer to allow greater control of each thread. Specifically, the pointer allowed us to, in the exit function, call k_thread_stop, instead of relying on having it called implicitly.

We also chose to separate out our prime computation function into two, so that our locking and unlocking calls could be more easily seen.

We also created an additional variable, num_threads_created, which was incremented each time kthread_create was successful. Most thread iterations used num_threads_created instead of num_threads, as we wanted to account for the (admittedly rare) case where kthread_create fails.

We also chose to use mutex locks instead of spin locks, to allow for sleeping.

Barrier Synchronization

Our barrier synchronization did not use locks. Instead, we relied on 2 atomic variables initialized to zero- one for each call to barrier_sync. In barrier_sync, which would be called for each thread, we then incremented the appropriate atomic variable. Once the atomic variable's value was equal to the number of threads that were created, the barrier was released, and all threads could resume execution.

The storage of one atomic variable for each barrier_sync call allowed us to avoid worrying about resetting them, which was a significant challenge that we had faced early on.

Locking vs Atomic Operations

Because of how the prime computation was modularized, switching to atomic variables were fairly simple. After deleting the locks, it was a matter of replacing assignment operators with atomic_set, and switching from array indexing to pointer arithmetic.

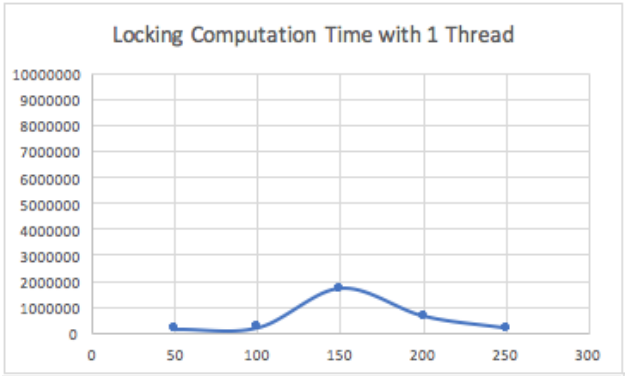
The locking module is found in sieve.c, and the atomic module is found in sieve_atomic.c

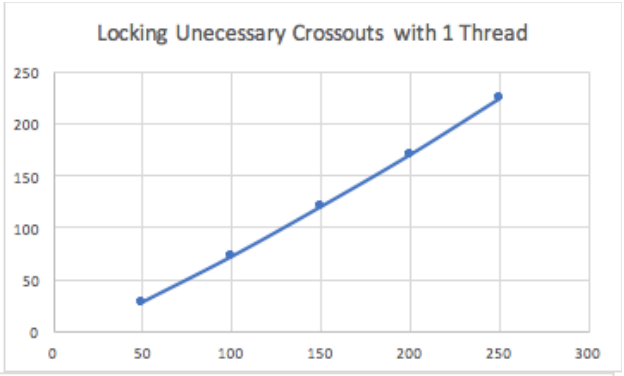
5. Module Performance

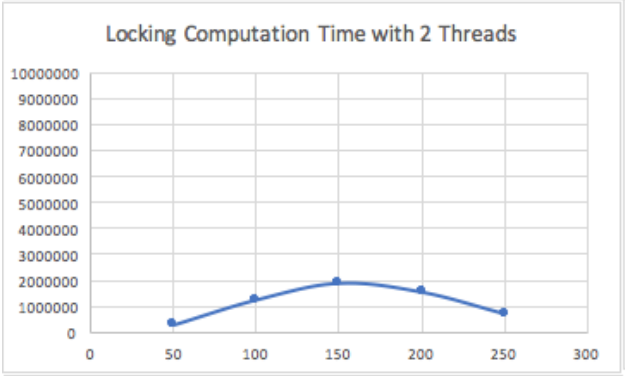
Performance Specifications

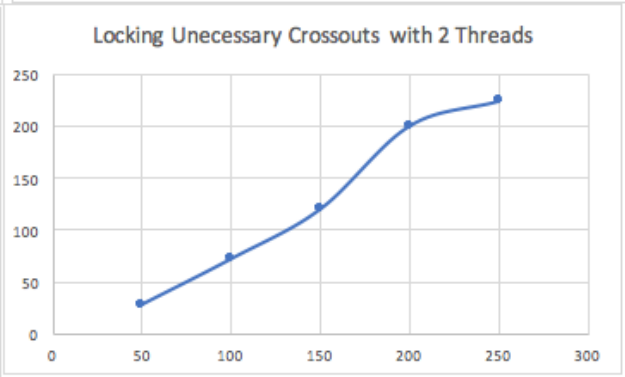
It is important to recognize that all measures of runtime are in nanoseconds. As such, variations in runtime at that granularity are somewhat erratic and not entirely representative of the efficiency of the modules.
Trials were run for num_threads = {1, 2, 3}, and upper_bound = {50, 100, 150, 200, 250}.

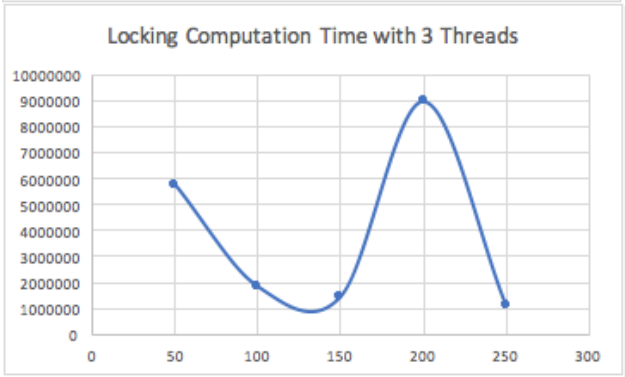
For the locking module, it can be observed that for all three threads, after an initial increase in runtime as a function of upper_bound, the runtime decreases. However, the additional overhead for running 3 threads is far greater than that of running only 1 or 2, and so running the locking module with one or two threads is preferable. Among threads, the number of unnecessary crossouts performed was more predictable as a positive linear function of upper_bound. However, for num_threads=2, there is a perturbation in this line. This irregularity suggests that data races which contribute to the algorithm's efficiency, while quite rare, sometimes do occur.

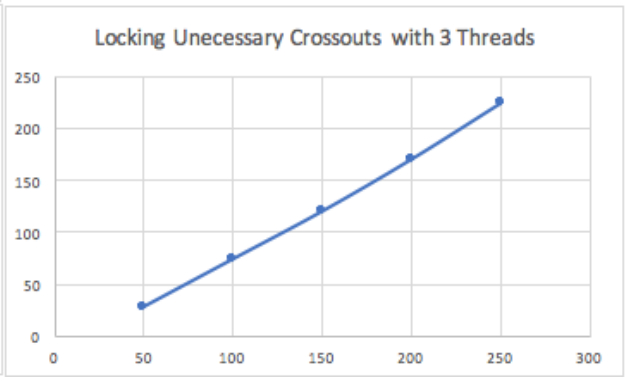




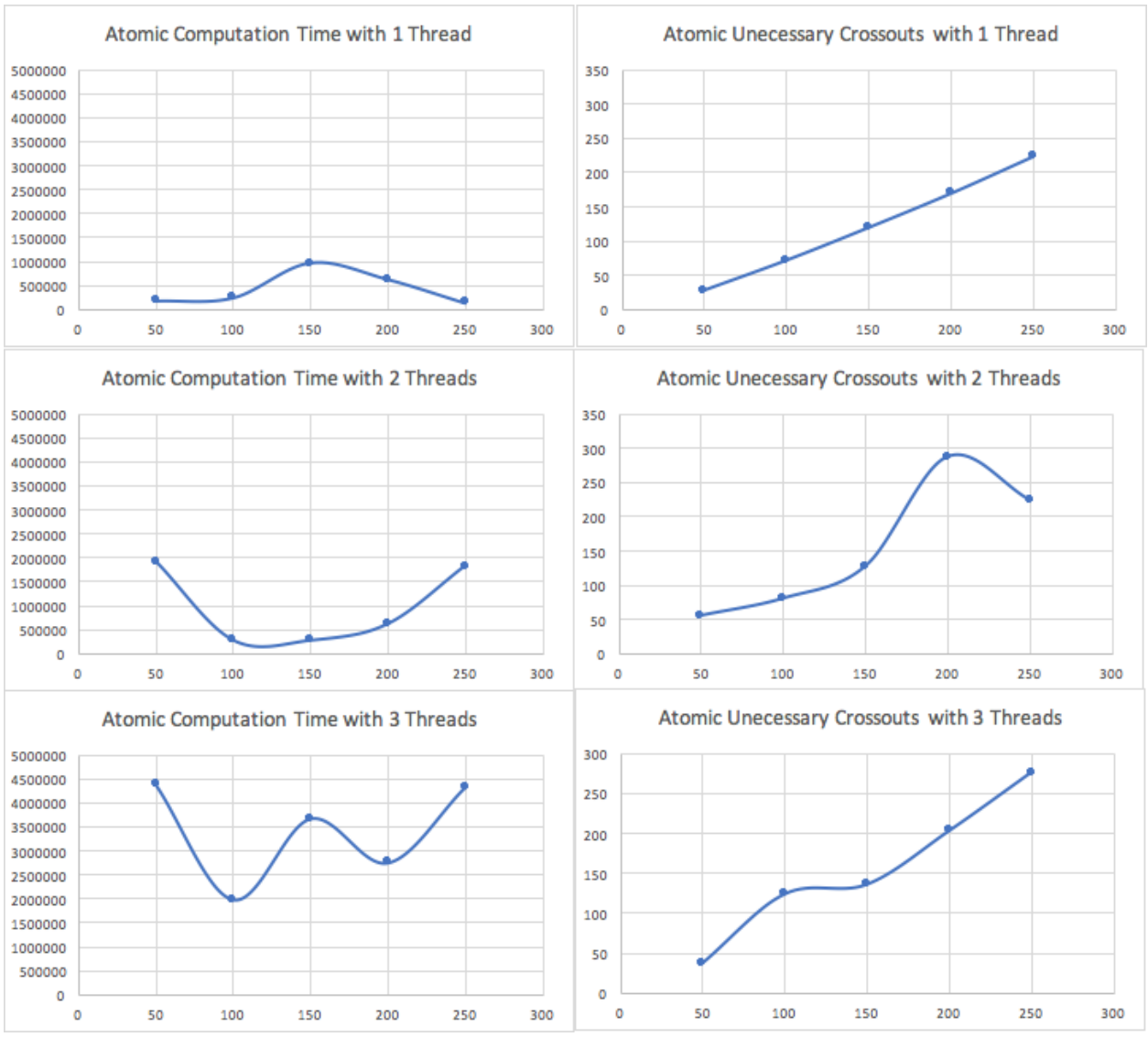








For the atomic locking module, the runtime is similarly much greater for num_threads=3 than for num_threads=1 or 2. However, the overall runtime is far lower for each number of threads. This improvement suggests that the native implementations of locking embedded in atomic variables have much less overhead than manually called mutex locks. The unnecessary cross outs also follows a generally positive trend with respect to upper_bound, although data races seem to be slightly more common.

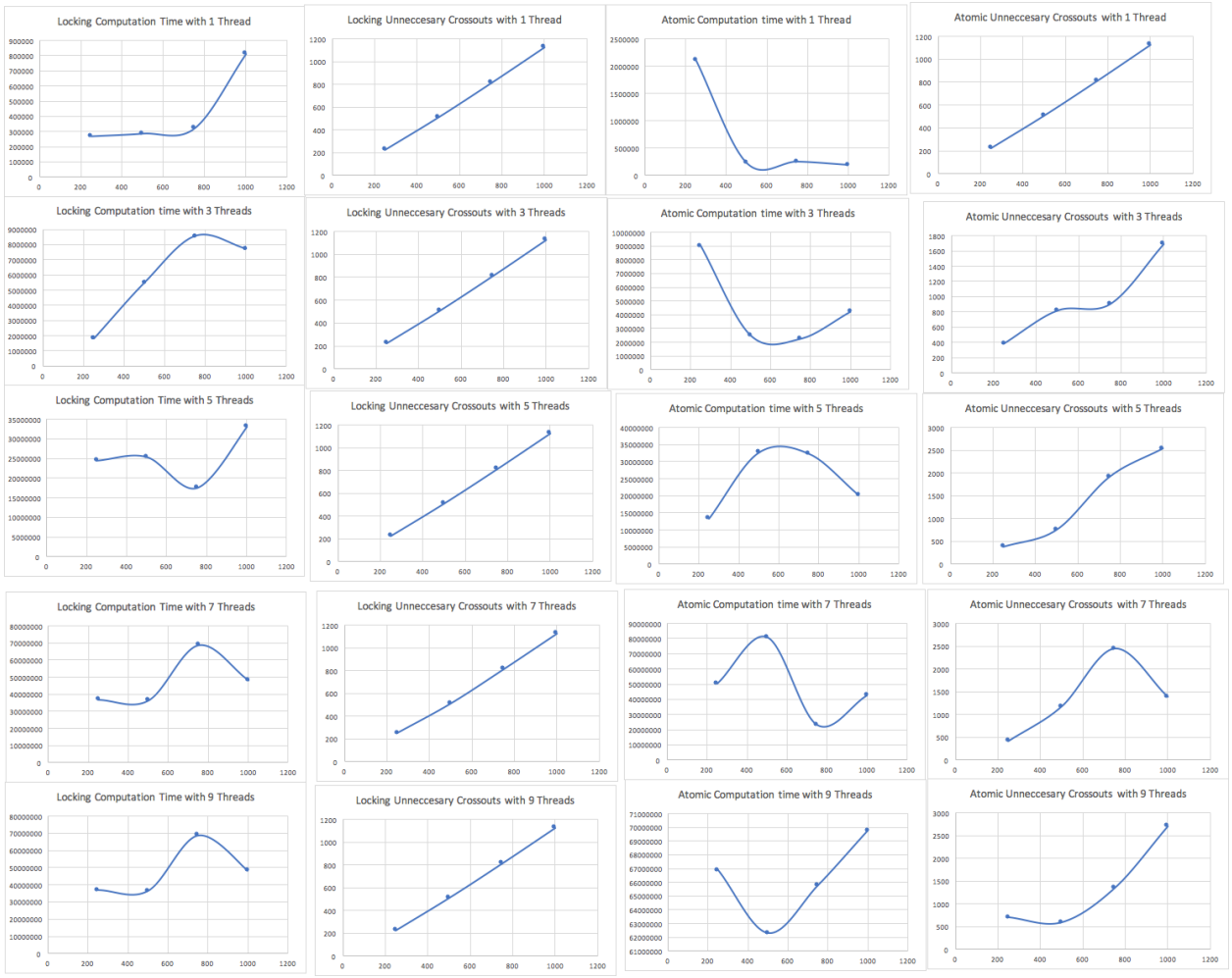


6. Chart Names

Each plot is found in the /charts folder. They follow the naming convention moduletype_metricNumthreads.png. For example, atomic_crossout2.png is the chart that measures the number of unnecessary crossouts performed by sieve_atomic.ko, with num_threads=2.

7. Names of any other files with interesting screenshots or traces you may have collected along with how you generated them and a brief discussion of why you find their results interesting.

Trials were also run with num_threads= {1,3,5,7,9} and upper_bound={250, 500, 750, 1000}. These charts are found in the /charts/additional_charts folder. We can see that these graphs reflect more erratic module performance as the number of threads increases beyond what the Raspberry pi can usefully take advantage of.



8. Any insights or questions you may have had while completing this assignment.

Throughout the module development process, the errors that we encountered were nondeterministic and seemed a bit dangerous. For example, we encountered a lot of “unable to handle kernel paging request at virtual address” errors, which would inconsistently lead to corruption of the pi, such as the command “ls” triggering a segmentation fault. Could any of these errors be persistent across reboot? Additionally, were the specific trends that were expected to be observed in the Module Performance section?

9. Assignment Improvement Suggestions

It would have been helpful if, on the lap page, there was a section for how to properly debug kernel Oops, and potential causes of kernel oops that may result from the limitations of the raspberry pi.

10. Approximate time spent on this lab:
30 hours