CSE 422S Lab 2 – Kernel Module Concurrent Memory Use

Anand Nambakam anambakam@wustl.edu

Resources Used: LKD textbook, Course Material, Linux Man Pages (Bootlin)

Module Design and Implementation:

Time Function:

The time function converts the time of invocation to nanoseconds and returns this value. This function is called in the init function and barrier functions to determine the module initialization, runtime, and completion time.

Init Function:

This function is invoked upon installation of the kernel module and is used to instantiate and allocate module memory. First, the module "safely initializes" the integer and counter arrays, and then performs validation checks upon user inputted num_threads and upper_bound variables. Upon success of the validation checks, memory is allocated for the integer and counter arrays, and kernel threads are created respective to num_threads using kthread_run. The atomic variable used for checking thread completion is updated to signify that the threads are not completed.

Per-Thread "run" Function:

This function is run by all spawned threads. The first barrier function is invoked to cause all running threads to synchronize at the barrier point. The "sieve" function, or prime calculation function is invoked, followed by another barrier synchronization function. The atomic variable used for checking thread completion is then updated to signify per-thread completed status.

Barrier Functions:

To synchronize threads prior and ensuing "critical" prime computation, I implemented two distinct barrier functions. Upon thread arrival, a spin lock is set, and an atomic variable counter is incremented per-thread. When the atomic counter equals the number of threads, the threads are "synchronized" at the barrier, the timestamp is updated (as the condition is set upon arrival of the "last" thread) and the spin lock is released. I chose to use spin locks due to busy waiting – if I used a mutex, threads may have been put to sleep, incurring delay.

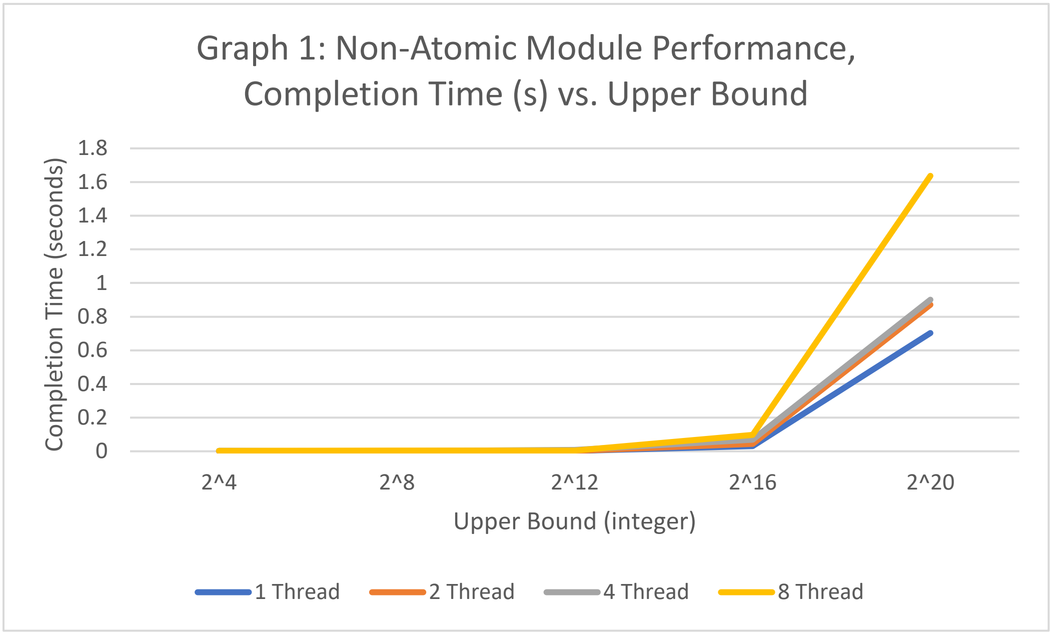
Sieve Function:

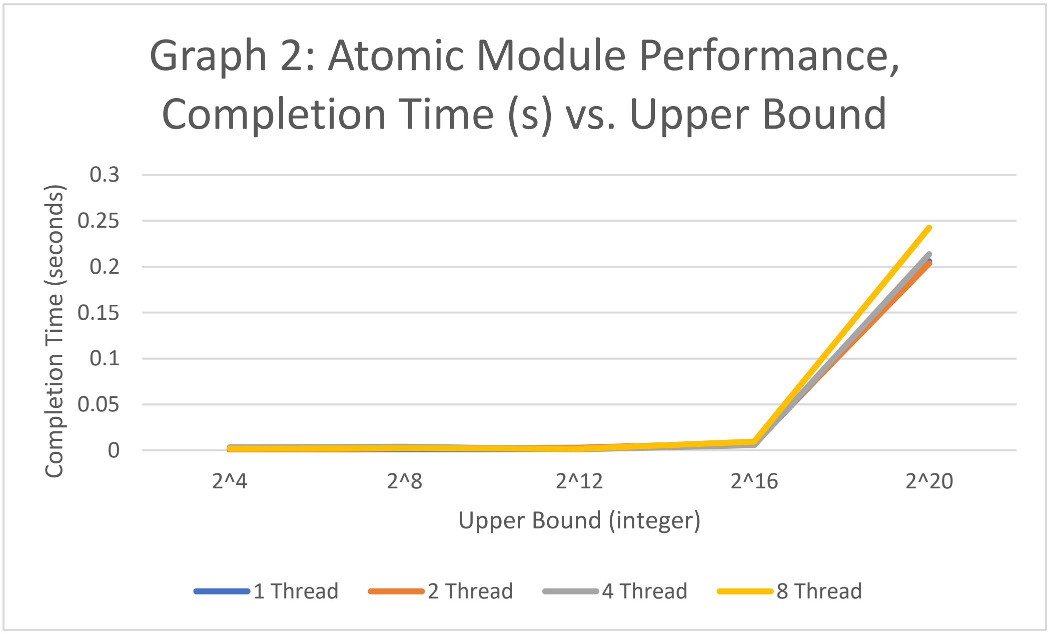
The prime calculation function employs two respective mutex locks. The first lock protects storing of the current position in a local variable, while the second mutex protects iterative "crossing out" of non-primes via array index, within the allocated integer array. The "cross out" component iterates, and modifies, array indices that are multiples of the current position. This function runs repeatedly per-thread using an infinite while loop, until all non-primes are “crossed-out”, constrained by upper_bound. To reduce occurrences of threads “crossing out” the same array element, a global position variable is updated and incremented upon function iteration. Inherent data race resulted in excess “cross outs”, but by incrementing the position index upon iteration, the number of excess “cross outs” was reduced. Because the Raspberry Pi is limited to 4 cores, schedule() was used to allow *n* > 4 threads to perform work concurrently. Without schedule(), I noticed that only 4 of the *n* threads ran.

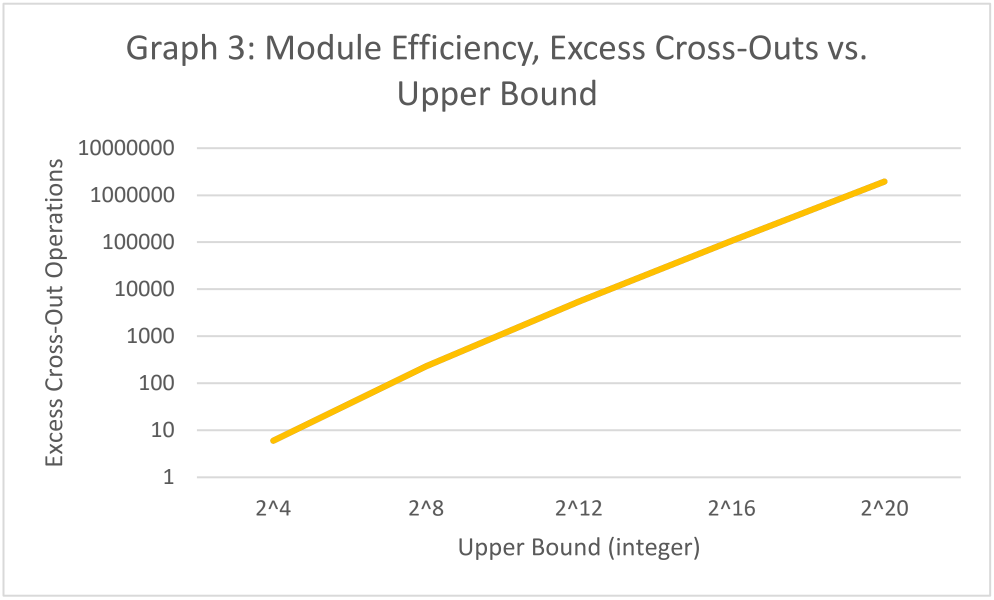
Exit Function:
This function is invoked upon module unloading. It checks the atomic variable used for checking thread completion, to ensure threads are not still computing primes. Additional checks are made to verify allocation of integer and counter arrays. The integer array contents, calculated statistics (cross_outs, excess cross_outs, primes, nonprimes), and module timing calculations are output to console via a helper “print_stats” function. Additionally, the memory allocated to the integer and counter arrays are freed.

Atomic Module:
To implement the atomic version of the module, I changed the integer array variable to type atomic_t. Instead of using the mutex lock as in the non-atomic version, atomic_set was used to atomically write/cross-out array elements in the Sieve function. Atomic_read was used in place of referencing the integer array, while incrementing the global position variable, and in reading the array to print contained primes.

Module Performance:







Observations:

For representative trials, num_threads = 1,2,4,8 were chosen because the Raspberry Pi has four cores. Upper_bound = 2^4, 2^8, 2^{12}, 2^{16}, and 2^{20} was used.

Module Performance Atomic vs. Non-Atomic

Comparing Graph 1 and Graph 2, the completion time appears to vary linearly with the upper bound. The atomic module completed markedly faster than the non-atomic module, especially observable as upper bound increased. As noted in Graph 1, the non-atomic module completion time remained constant among different numbers of threads, until differentiation at upper bound=2^{16}.

Completion time was markedly worse for the 8-thread configuration at this high upper bound, as compared to the 1,2, and 4 thread configurations. Additionally, while the thread number did not drastically impact performance, at high upper bounds, the 1 thread performed better than 2 and 4 threads, which performed markedly better than 8 threads. This may be caused by time incurred as 8 threads are allocated to 4 cores, as well as the time incurred by spinlock blocking mechanism in the non-atomic module.

For the non-atomic module, there was no marked differentiation among different thread configurations, as upper bound was increased. The atomic module does not incur this delay due to using atomic operations. As the upper bound increased to over 2^{16}, completion time increased linearly.

Module Efficiency

The number of excess cross-outs increased exponentially as the upper bound increased – the y-axis (excess cross-outs) was scaled to log10 to demonstrate this exponential trend. There was no differentiation between the number of excess cross-outs between the atomic and non-atomic modules. The non-atomic module used a mutex to protect per-thread “cross-out” operations, taking more time than atomic operations. Functionally, the atomic and non-atomic modules both protect per-thread cross-out operations, which does not impact the number of excess cross-outs. In both modules, the global position variable was incremented upon thread arrival. Because both modules have an inherent data race in “crossing out multiples”, irrespective of protection mechanism employed, it is expected that the number of excess cross-outs was the same between modules and per thread configuration.

Time Spent

~30 hours