Lab #2 Kernel Module Concurrent Memory Use

Tara Renduchintala: trenduchintala@wustl.edu
Anderson Gonzalez: agonzalez@wustl.edu

Design Implementation

Init Function

In the init function, we had to allocate memory for all of the shared memory regions we were going to be accessing throughout the program. It also needed to initialize all the variables and check if the module parameters had valid values.

1) All of the memory regions were allocated using kmalloc with GFP_KERNEL.

- Allocation of the Number Array:

The number array had to be able to hold an array of integers whose length was one less than the upperbound. Therefore, the amount of memory allocated was the size of an int * (upper bound – 1). If the number array was allocated properly, we printed a statement indicating the allocation of the number array was complete.

If the number array was NOT allocated properly, in which case it would return NULL, then we have to ensure that all the values are reset (number of threads, upper bound) and make the pointer to the number array be 0. We then would return a negative non-zero number that indicates that the array was not allocated properly.

- Allocation of the Counter Array:

The counter array had to hold an array of integers that is the length of the number of threads (as these would hold the count of crossed off numbers for each corresponding thread). Therefore, the amount of memory allocated was similar to that of the number array, except we multiplied the size of an int by the number of threads [size of an int * number of threads]. If the allocation was successful, we printed a statement indicating the allocation of the counter array was complete.

If the counter array was NOT allocated properly, we once again reset all of the values to 0 (number of threads, upper bound) and we made sure the counter array pointer was set to 0. We also had to free the previously allocated number array using kfree. Then, set the number array pointer back to 0 as well. Then we returned the same negative non-zero number indicating that the array was not allocated properly.

- Allocation of thread pointers

The array of task_struct pointers was called threads. We had to allocate the size of a task_struct pointer for each thread that was going to be run. Therefore the size we needed to allocate became (the size of a task_struct *) * number of threads).

2) We the pointers to the arrays to be 0 at the very beginning so they could be properly allocated later on.

3) We also set the atomic variables in the very beginning. We set the first and second barrier synchronization atomic variables to be 0 such that they were unlocked at the beginning. We set the finished atomic variable (which indicated that all the threads had finished) to be 1 while we were allocating memory. Just before we spawned the threads, we set that back to 0 so it could officially start.

4) At the very beginning, we got the time that the module was initialized by using the function getnstimeofday(). We chose to use this timestamps as based on our research, it seems as though it has better granularity than current_kernel_time(). It also has better precision than do_gettimeofday(). Therefore, since we are trying to measure performance, we opted for this time function in order to be as precise as possible.

5) Once the threads were initialized, we passed the thread function to all of them, as well as a reference to each thread's corresponding crossed number count. We had to cast the crossed number count as a void pointer as the thread function had to intake a void pointer (according to the compiler warnings we were getting). We then woke up each thread process.

Thread Function and Barrier Functions

The thread function is responsible for the barrier synchronization and the manipulation of the array by passing the counter variable into the prime function and having the thread run the prime function.

Once the thread enters the thread function, they wait at the first barrier. Until the first barrier sees the correct number of thread, the threads are stalled. Each time a new thread arrives at the barrier, indicating that it has spawned and started running, we add 1 to our atomic variable "barrier_1". This variable needs to be atomic as it cannot be manipulated by more than one process at a time. If that were to occur, there would be a race condition which could mean that the processes start running the function to early, or not at all.

Once all of the threads arrive at the first barrier and the number of threads at the barrier matches the total number of threads, the processes then proceed on to the prime function where they manipulate the array.

After each thread finishes manipulating the array, they return back to the thread function which then sends them to the second barrier. There, the processes wait at the second barrier until all of the threads arrive. The second barrier works similarly to the first in terms of having its own atomic variable to keep track of how many threads there are.

NOTE: The barrier functions are identical except in the fact that the first barrier uses the atomic variable named barrier_1 while the second uses barrier_2. We initially attempted to create one general function and pass in the atomic variable, however, for some reason the function kept stalling after the first thread was at the barrier. Once we separated it into two functions it proceeded to work. We are unsure why this was the case and even after attending some office hours it was not clear. Therefore, we left the two functions separated even though it may not seem like the best coding practice.

We also got the time at each barrier function so that we could evaluate the processing time between when the functions entered the prime function to when they left. The entering of the prime function is signified from when they leave the first barrier, and the exiting of the prime function is signified when all of the threads arrive at the second barrier.

Once the second barrier has been left, the thread function sets the finished atomic variable to Finished indicating that the threads have finished processing and that the module can exit.

Prime Function and Locking

The prime function manipulated the thread. It took in a pointer to the crossed count number that was associated with the thread that was currently using the function.

The actual functionality of the prime function had to be executed in a look that didn't terminate until it hit a specific condition. Therefore, we had an integer that was always set to 1 as our while condition, making the while loop spin continuously until a condition was met. These conditions were:

1. The current prime number index (the global prime number index) was equal to upper bound - 1 (ie. the length of the array).
2. The current index (the local prime number index) was greater than the length of the array

In order to not have race conditions, we used spin locks (as putting the process to sleep for a period of time did not make sense to us) to lock the prime function. The spin locks were implemented by ourselves, similar to how we implemented them in our studios. Once this part of the prime function was locked (this was the first lock in the prime function) we set:

- the local current prime number index to the global current prime number index
- a local value variable to the value at the index of the local current prime number

We then incremented the global current prime number index. We then checked condition 1 above. If condition 1 was NOT met, we then incremented the global index until we found the next non-zero value in the array. After we finished incrementing the index, we unlocked this part of the function. This way another process could come and get it's local version of the next prime number index.
The process then goes on to check condition 2 above. If condition 2 is NOT met, it enters a loop that checks all of the values that are multiples of the current value. The iterator increases by the amount of "value" as adding "value+value" gives you the next multiple of value. Inside the loop, we lock the function with our second lock as we are now manipulating the number array. If the value that we are incrementing by is 0, we want to break out of the loop as we don't want to be stuck in an infinite loop as the iterator is not increasing. We then set the value at the index indicated by the iterator in the number array to 0. Then, increase the count. After that, we unlock the second lock so the next process can continue to manipulate the array.

This prime function is repeated by each loop until one of the two conditions above are met. When they have been met, this indicates that the entire function has been combed through, leaving only the prime numbers.

Exit Function

The exit function first checks to see if all the threads have finished. If they have not finished, the exit function safely stops the threads and outputs a message the processes didn't finish but the threads are stopping. If the threads do finish, it then prints out all the prime numbers.

After the printing has finished, it then prints out the initialization time (the point of initialization till when the first barrier is reached) and the process time (the time from the first barrier to the second barrier).

After the time has been printed, it safely frees all of the allocated memory (the number array and the count of crossed numbers array) using kfree(). Once it has finished, it prints out a message saying that the module has unloaded.

Atomic Changes

When we made the number array to be atomic integers, we did not have to have a second lock. Therefore, in the exit function we only had to lock the current prime number critical region. This was because the atomic integer array was able to ensure, through the atomic function, that there was no race condition when accessing the array and changing numbers. This way, multiple threads could be working on the array WITHOUT accessing the same index and causing a race condition.

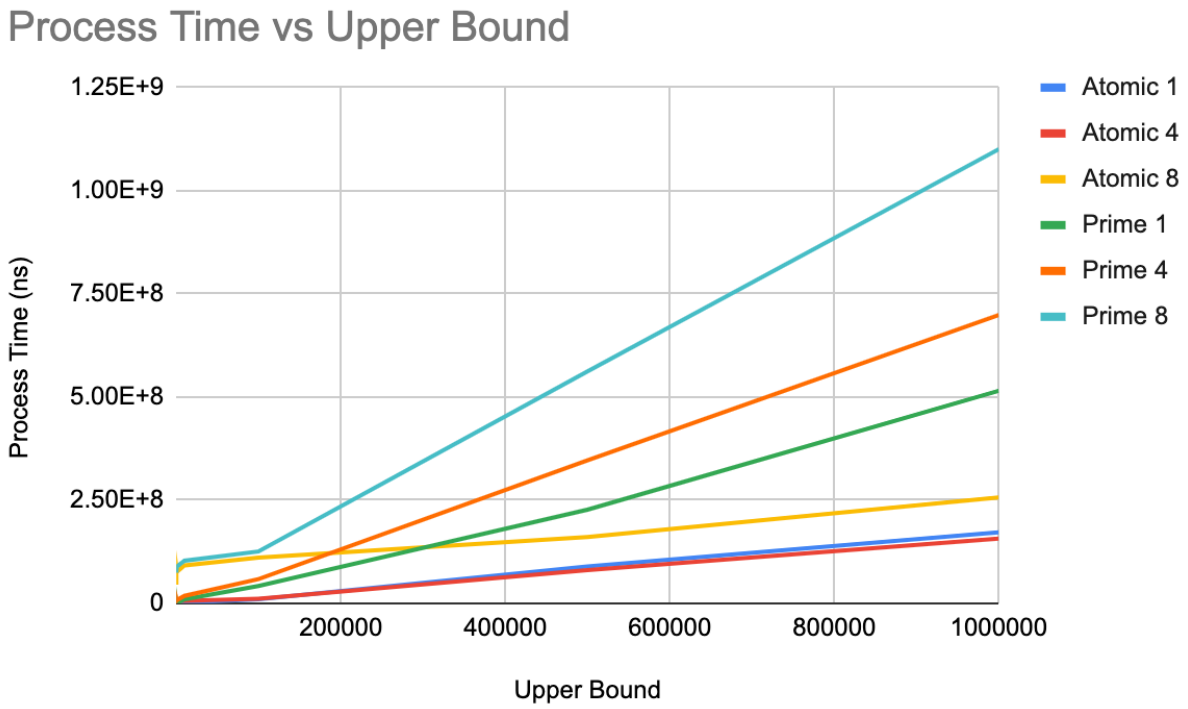
We had to update all of the times we access the number array by using the atomic set and read. We had to initialize the array in the init function using set. Any time we had to change the value to a 0, we also had to use set. If we wanted to check the value, we had to use read.

Module Performance

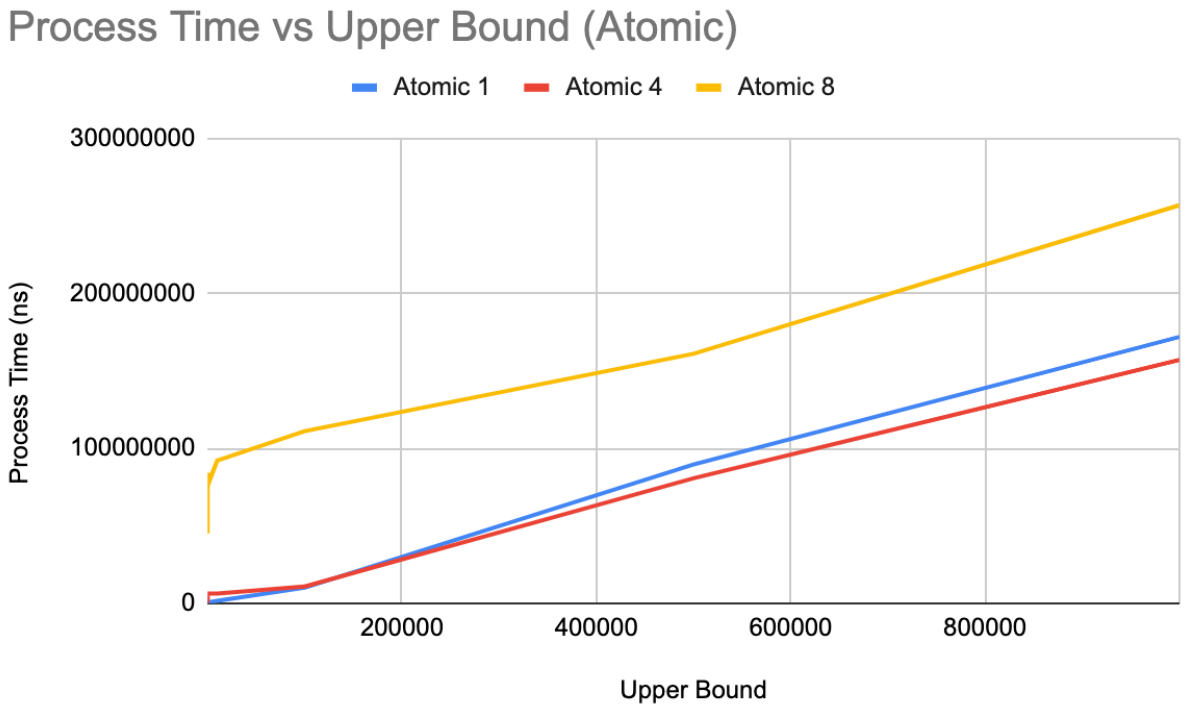
Upper Bound vs Process Time

The total file of our data is attached to our submission, but these are the means of our data collection. We tested the process time with 1, 4, and 8 threads. We also had a variety of upperbounds.

Upper Bound	Atomic 1	Atomic 4	Atomic 8	Prime 1	Prime 4	Prime 8
10	3497655	4757268	70928246	495140	8173562	78217113
20	527042	4909388	76680754	509940	11912186	78619753
50	881843	218297	45103668	1387911	9788248	74553308
100	417309	4205737	72587965	673325	10556643	75575876
500	318776	5755908	84201392	821280	7079989	86718575
1000	661134	6241154	77196793	614210	8185304	89548363
10000	1646464	6236500	92193181	10048723	18038127	103523637
100000	10096240	10834251	111107427	41875275	58700947	125694108
500000	89586809	80712887	160958725	226843581	346859664	562143434
1000000	171949747	157054842	256873140	515048728	698740684	1100724723



It is evident from the chart above that the Atomic number array GREATLY reduced the time of processing, and thereby reducing the overall process time. This is most likely due to the fact that it bypasses a spin lock entirely, thus there was no idle spinning. Furthermore, more than one thread could work on the array at a time, so long as they were both not trying to manipulate the same index.

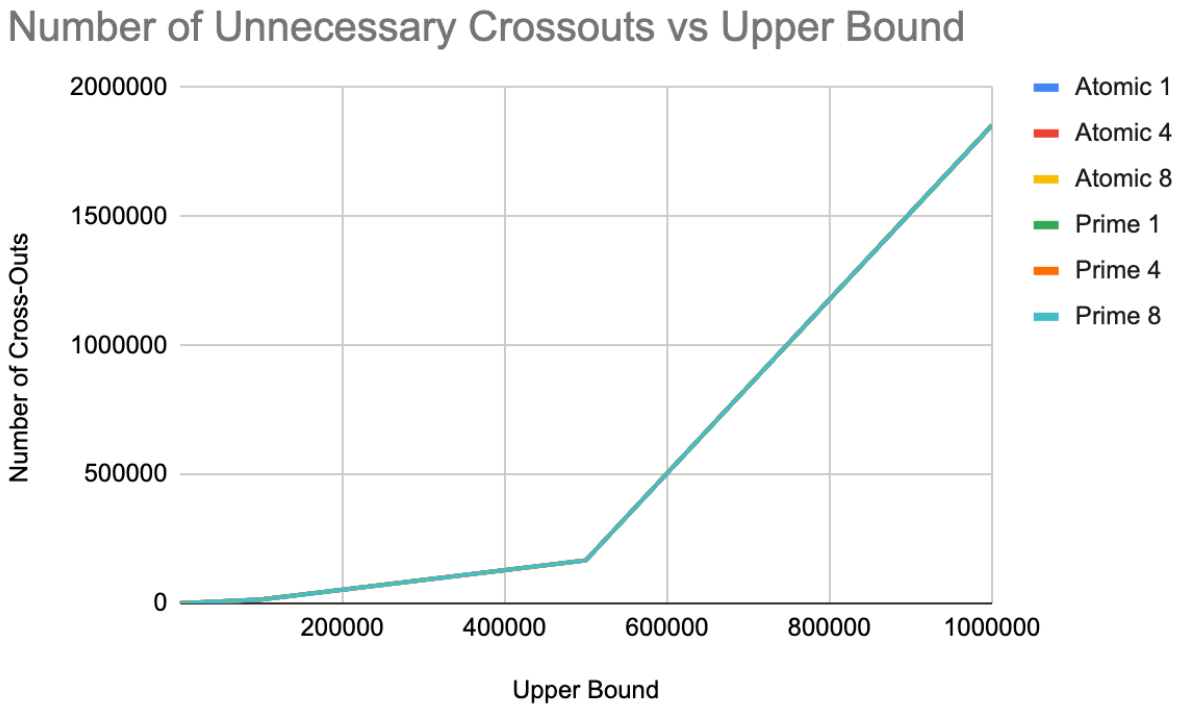


It is also evident that from the graph above, the process time for more threads does not exactly imply that the time will be better. When there were 8 threads, the initialization time overhead was so great that the 8 threads performed worse than one thread or 4 threads. This is most likely due to the fact that our Raspberry Pi has 4 cores and thereby, the 8 threads would need to be consistently scheduled on different cores. However, it is evident that in the atomic case, 4 threads performed better than one thread. This is probably due to the fact that the Pi could have had one thread per core. However, this 4 threads only started performing better after a certain bound. This is probably due to the fact that the overhead of initialization time was great and was overshadowing the prime computation time and the lower bounds. Once prime computation time came to be greater than the initialization time, it became clear the 4 thread atomic was out performing the one thread.

These are the Number of Unnecessary Cross-Outs in regards to the upper bound and different number of threads.

Upper Bound	Atomic 1	Atomic 4	Atomic 8	Prime 1	Prime 4	Prime 8
10	2	2	2	2	2	2
20	7	7	7	7	7	7

50	28	28	28	28	28	28
100	72	72	72	72	72	72
500	170	170	170	170	170	170
1000	509	509	509	509	509	509
10000	1127	1127	1127	1127	1127	1127
100000	14301	14301	14301	14301	14301	14301
500000	166401	166401	166401	166401	166401	166401
1000000	1853709	1853709	1853709	1853709	1853709	1853709



As can be seen by the data and graph above, the number of threads and type of threads had no effect on the number of cross-outs that occurred. However, as the upper bound increased, the number of cross-outs increased significantly. This is due to the arrays getting larger and the method being inefficient. Therefore, the number of cross-outs is definitely proportional to the upper bound, but is not related to the number of threads. This is due to the fact that the same computation is occurred on the array regardless of the number of threads that are running.

File Names:
Prime module: sieve_final.c
Atomic module: sieve_atomic_final.c

Data: Lab2 Data

Compilation:

The generic kernel module Makefile used in CSE422.

Development Effort

This lab took us about 25 hours combined.

Tara – 15 hours
Anderson – 10 hours