**Kernel Module Concurrent Memory Use**

**1. Module Design and Implementation**

\*For simplicity, I will call the module with the lock implementation as Module 1 and the atomic version as Module 2.

To begin with, both modules take in two parameters, num\_threads and upper\_bound, and will perform Sieve’s Algorithm for the given range using the number of threads. Error cases regarding failing to successfully start the program has been handled appropriately by either exiting the program in the case of invalid parameter(s), or using the default values in the case of no input parameter(s). Once these parameters has been validated, I use a series of kmalloc to allocate enough space for the array of numbers, array of counters, and array of threads, in the given order. An important procedure here was to perform these kmallocs one by one and to check right after whether the allocation was successful via checking for NULL. For example, if in the case that our number array has been allocated but there isn’t enough space for the counter array, the program will free the number array, set params to 0, and simply return (exit). Once successfully allocated, I used a simple for-loop to create each threads, store them into the thread array, and woke them up so they could start processing. As we need to wait for all threads to arrive before entering the critical section, I created a global atomic\_t variable that will hold the count of threads that has arrived at the target function of kthread\_create using atomic\_add(). If the numbers don’t match, the thread will cycle through an empty while loop as it needs to wait. Once synchronized, the threads now call the main computation function with the pointer to the individual thread count address as its sole input. In this function, I initialized local variables that I will use to store my position index (declared globally) so I could use them safely without having them collide with other threads. A key point that I want to note is that in setting these local variables, I had to acquire a spinlock (declared globally) so that I could ensure that my global position value doesn’t get affected by other threads while we try to read and write from/to it. Once I release the first lock, I then iterate through the array of numbers to cross out non-prime values. As I need to write a 0 into the array if the value in that index isn’t prime, I had to acquire a second spinlock (also globally declared) within the for-loop and release it after each loop for Module 1. In the case of Module 2, the main difference is that I don’t use the lock, but rather the atomic\_set() function to cross out the non-primes in the number array. More details on the difference between the two modules are discussed in the next paragraph. Now, to ensure that all threads have finished their work, I used a second barrier in the thread target function so that all threads can synchronize once it has exited the Sieve computation function. The implementation is identical to the first barrier, where the atomic\_t variable has been globally declared and will wait in an empty while loop until it’s ready to continue. Now, once we have arrived at the exit function, the main aspect (other than all the analytic reportings) is the checking for whether there are any threads that are still processing and having to free our arrays from the allocated space once all computations are completed. The first part was done by the use of another global atomic\_t variable, where I simply marked it as 1 after going through the second barrier (instantiated in init function as 0 to indicate that it isn’t done). If the variable is still 0 when it reaches the exit function, the program will print an appropriate statement and return (exit). Although I wasn’t able to test this case for myself as the threads were never left hanging, I am confident that it handles the error case appropriately. For the second part, I simply called kfree() for the arrays after I have printed out all the necessary analytics and print out a statement indicating that the all processes have been completed and the allocated memories has been freed.

In Module 2, as mentioned earlier, one big difference is that it doesn’t use a second spin lock necessary to lock the array of numbers. As a result, it was necessary to declare the number array as an array of atomic\_t, instead of the familiar int array in Module 1. By doing this, I had to swap all the writes and reads to this array in the program to atomic\_read() and atomic\_set(). Once these changes were appropriately made, I now tested Module 1 and 2 to test their validity and performance.

**2. Module Design and Implementation**

I experimented with the modules using varying parameters to notice any trends with the result. For comparison, I tested each modules using the same set of parameters and graphed them using the same medium (excel). The data and graphs that I will be presenting in this analysis can be found in “data and graph.xlsx” file within the folder if interested. I will discuss the reasoning behind the choice of parameters and provide analysis separately below.

**Analysis of Initialization vs Computation time:**

In this part, I wanted to see the relationship between the initialization and computation time, so I chose various upper bounds and thread numbers that I determined would yield a data table that the viewer can easily comprehend the trend. As a result, I ran the following experiments for each modules using the combinations of Num\_threads [1,2,8] and upper\_bound [500000,900000] for each module.

**Module 1:**

|  |  |  |  |
| --- | --- | --- | --- |
| Num\_threads | Upper\_bound | Initialization Time(ns) | Completion Time(ns) |
| 1 | 500000 | 5536388 | 425308976 |
| 2 | 500000 | 6144980 | 334540470 |
| 8 | 500000 | 72163827 | 324325295 |
| 1 | 900000 | 9447105 | 717849688 |
| 2 | 900000 | 10287711 | 601761222 |
| 8 | 900000 | 66228234 | 655209112 |

**Module 2:**

|  |  |  |  |
| --- | --- | --- | --- |
| Num\_threads | Upper\_bound | Initialization Time(ns) | Completion Time(ns) |
| 1 | 500000 | 5832706 | 106968342 |
| 2 | 500000 | 5528799 | 86389537 |
| 8 | 500000 | 68792957 | 72781704 |
| 1 | 900000 | 9433058 | 161307863 |
| 2 | 900000 | 9616912 | 157624078 |
| 8 | 900000 | 87491543 | 150006090 |

From looking at these tables, we can see that for both modules, the initialization and completion **increases** as we introduce a larger upper bound, which makes sense as we would have more computations to go through. When comparing the effect of number of threads for a given bound, with the exception of the second data point in Module 2’s table, we can see that as we introduce more threads, the initialization time **increases** for each experiment. In general, this makes sense because there would be overhead caused when creating the threads, so we would expect the initialization time to be higher when we have more to initialize. In terms of completion time, we can see that there is a noticeable drop between the single thread and multi threaded implementation, but not as much of a difference between the multithreaded ones. This behavior is interesting because I expected the multithreaded version to have a much shorter completion time as the number of threads increased, but from pondering about the program’s behavior I came to conclusion that it was acceptable because even though we may have lots of threads, these threads blindly perform their computations in crossing out the non-prime numbers, so the work load isn’t actually divided into the number of threads (meaning that there would be overlapping tasks).

**Graph 1: Computation time analysis**

For these graphs, I wanted to observe how our computation time will be affected by varying the upper bound and number of threads. To do that, I tested experiments for upper bounds of 100000, 200000, 300000, 600000, and 800000, and number of threads of 1,2,3,4, and 8. By incrementing the two parameters so that they cover both low and large values, I was able to find an increasing trend for all threads as seen below:

|  |  |
| --- | --- |
|  |  |

First off, by analyzing the trend of the individual curves, which represents the number of threads, we can see that for both modules the completion time **increases** as the upper bound **increases.** This was to be expected because larger upper bound means that our number array would get larger and thus would need to compute more. What was interesting, however, was when I compared the curves within a graph with each other and when I compared the 2 graphs as a whole. When comparing the completion time amongst the threads, I noticed that simply having a high thread number **does not** yield a better completion time. Such effect can be clearly seen by comparing thread count 1 and 8 for both graphs - we can see that these two curves in general have the highest completion time. In other words, in general, thread counts of 2, 3, and 4 performed better than that of 1 and 8 in terms of completion time. Although the high completion time for thread count 1 was expected as it is running the code sequentially, from the observation of thread count 8 it was clear that there are noticeable overhead caused when creating multiple threads, and while there clearly are situations where this overhead can be less than the speedup from multithreading and make multithreading a viable option as seen in thread counts 2, 3, and 4, once the number of threads gets too large the module’s core computation time will stay similar, but the initiation time would increase due to the overhead. With this observation, as our hardware is limited to the raspberry pi, the low performance of the experiment with thread count 8 makes sense. Now, comparing the two graphs with each other, we can clearly see that the completion time for Module 2 is much shorter than that of Module 1 for all threads and bounds. This can be explained by the fact that while Module 1 uses a spin lock within the main critical section to handle the int array of numbers, Module 2 uses an atomic\_t array of numbers without locks in the critical section. By doing so, we can see that Module 2’s computation time has decreased substantially, as not only our read and writes to the number array are faster but also won’t have the overhead of creating and waiting for the locks. From analyzing these graphs, I learned two main things: 1. It is important to choose parameters that are suited for the algorithm and hardware as failing to do so would decrease performance while the output is still correct, and 2. Creating and handling locks and threads cause overhead, so be wise - use atomic variables if applicable to speed things up (in the context of locks) and don’t create extra resources unless necessary.

**Graph 2: Unnecessary Cross-out analysis**

|  |  |
| --- | --- |
|  |  |

For these graphs, I wanted to observe how the number of unnecessary crosses changes across different parameter values. In this experiment, I tested thread counts of 1 and 2, and upper bound values of 10000, 200000, 300000, 600000, 700000, and 800000. The reasoning behind only reporting thread counts of 1 and 2 is that the number of unnecessary crosses were identical to that of thread count 2 when num\_thread >= 3, so they yielded the same curves and thus were omitted. From analyzing the graphs, we can see that for all threads, as the number of upper bound **increases** the number of unnecessary crosses also **increase.** This is to be expected because as our array of number gets larger, we are ought to cross off unnecessary prime factors more occasionally as there are more multiples. In addition, we can clearly see that for both, the curve with thread count 2 has more unnecessary crosses than that with 1 thread. This also makes sense because as we have multiple threads interleaving the critical area, we are bound to cross off prime factors more frequently as they are individually crossing out values in a blind fashion.