Nathaniel Fenoglio

1001634050

Lemuel Rivera

1000842851

**General Process Outline for Project 2 Including Problems and Solutions Encountered Along the Way**

Assignment 1

The goal for this assignment was to write a program to utilize multiple threads to find out the number of matches of a smaller substring in a larger string. We started from the example code that was provided as a template and we modified it to utilize multithreading.

We removed the characters '\r' or '\n' from the strings, inserting '\0' when we found the1st occurrence of one of these characters to get an accurate string length. By checking for '\r' or '\n', we were able to successfully handle and identify whether the input file contained a new line after the 2nd string.

We wanted to be able to handle strings of all lengths and found that the solution was to determine the section length for a thread through dividing the 1st string length by the number of threads, then calculating the remainder and dividing the remaining values that needed to be checked amongst the threads, as such:

*section\_length = section\_length + ((n1 % NUM\_THREADS) / NUM\_THREADS) + 1;*

There was a bit of confusion with what was happening when we attempted to pass the array offset starting points to the num\_substring function because we did not notice at first that the offset value was being overridden at the same time as it was being sent into the function. This was corrected by instead saving the array offset values in separate indexes of an array to pass to the num\_substring function.

Because we needed to count the successes of the multiple threads operating on the array at the same time, we needed to lock the mutex when one of the threads was updating the global count variable, then unlock the mutex so that other threads could do the same once done with the update.

The various threads were then able to operate on their respective sections of the array and update the count variable giving accurate results.

Assignment 2

We used a mutex lock and 2 semaphores to avoid buffer reading and writing errors.

The mutex was used to lock the critical region of when either the producer was writing to the buffer or the consumer was printing from it.

The semaphore “empty” was used by the producer thread to wait on and the consumer thread to post to. This allowed the producer thread to be woken up when the consumer thread had left the critical region and the buffer was no longer full. Same process for the semaphore “full”, except that in the case of “full”, the buffer was no longer empty.

It took some trial and error to figure out how to keep the producer and consumer in sync with the correct indexes in the buffer array to write or read to from. The solution that we found was to let the travel through the buffer be circular. As long as the reading or writing did not go further than the declared buffer size, then nothing would be overwritten before read, or vice versa.

The buffer size was incremented on every write and decremented on every read to keep track of the current position and to be checked if the buffer was full or empty.

Assignment 3

…talk about perhaps not figuring out the benchmark execution maybe you’ll figure it out still butt….

While trying to figure out a solution for Assignment 3, we were completely stumped on how to implement the benchmark execution. Despite the article given, we were still having a bit of trouble understanding how to adapt the code that we were given for the context switch program. At the time of writing this, we are sure that the program to be submitted will not be a proper benchmark program. Instead, we will try our best to write the most competent program that we can.

…talk about what you learned along the way by researching the situation and what you think that you should’ve seen…

Through research and attempted troubleshooting we did gain insight into the context switching costs involving multiple threads. Threads have lower context switching costs in comparison to processes due to the memory that is shared by threads. Threads exist within 1 process with the same code, data, heap while different processes exist in entirely different address spaces. This causes the context switching cost for threads to be notably lower than that of processes.

We expected to measure much lower context switching costs when we measured the difference between the thread version of measureSingle and measureSwitch. Instead, we received an output showing smaller output for measureSwitch than measureSingle. We had expected to see larger times for the measureSwitch output to be able to calculate the direct and indirect context switching costs with the equations:

*total cost of context switch = time2 - time1 (microseconds)*

*indirect cost of context switch = total cost - direct cost*

We tried using pthread\_yield(), sched\_yield(), or mutexes and semaphores to create the situation of context switching like the example program used the pipe communication technique to achieve. When debugging, it was noted that the 2 processes were alternating in succession successfully.

We included the processor output information from running the cat /proc/cpuinfo > cpuinfo.txt command showing that 4 virtual processors are configured in the virtual machine.

The cache size for each processor shows to be 6144 KB. The indirect cost of the context switches would be the greatest when the size of the array is greater than the available memory in the cache. This would cause most all of the cache memory to need to be replaced. We kept this in mind when attempting to test and calculated that 6144000 / 8 = 768000 would be the size of the array that we would need in order to have the cache need to be repopulated due to a context switch. We tested with a stride size of 8 or more to avoid prefetching by the operating system.

We suspect that there may be a problem in the time calculation as when we worked with larger array sizes the program took noticeably longer to complete, but we still received similar output.

Things that we tried for setting the threads to run on the same processor:

We tried using sched\_setaffinity(0, len, &new\_mask); because we learned that “0” as the 1st argument would select the current process and both threads would reside in this same process.

We tried using the pthread\_setaffinity\_np() method that we found while researching the situation. It was noted that the method should return “0” on success and instead was returning “3” when attempting to set the processor for each thread to the same processor. We are unsure what the cause of this error is.

The files are included for our attempt so that they are able to be analyzed in hopes that a solution will be able to be determined and we are able to gain insight into the cause of the program not producing the expected output. Commented out code is included, although messy, showing various paths that were tried.