Operating Systems 2INC0 Assignment 2

Zachary Kohnen (1655221) and Cansu Izat (1372416)

Eindhoven University of Technology

1 Main Implementation

We created an array to store all of the thread ids in order to access them later in execution. Then, in order to keep the threads as busy as possible, we went through each item in the array, checked if a thread exists, then if a thread did not exist, we spawned a new thread and saved its id in the thread ids array, and if a thread already existed, we waited for it to finish executing before doing the aforementioned steps. Once the end of this array is reached, we loop back to the start of the array and repeat all the steps until every thread that needs to be spawned is spawned. Then, we wait for all the threads to finish their tasks, and we print every item.

Inside of the thread, we retrieve what multiples we need to flip, from the arguments passed on the thread. Then, with a for loop, we loop through all of the multiples of the given number until we reach the number of pieces. We calculate which index of the array the piece is stored, and which bit the piece is stored in. Next, we lock the mutex, in order to have exclusive access to the buffer, and then we read the unsigned 128 bit integer from the index of the buffer. Then, we flip the bits corresponding to the specific piece, and we write the modified unsigned 128 bit integer back into the buffer at the same index. And then we unlock the mutex, as we no longer require exclusive access to the buffer.

2 Advanced Feature

For the optimized version, we chose to use a single mutex for each buffer index rather than choosing a single mutex for the entire buffer. We thought, since there would be a single mutex for each buffer index, which would allow multiple threads to be able to access separate parts of the buffer concurrently, preventing the waiting that is caused by the use of one mutex. Removing the waiting from the threads and allowing them to run concurrently would, we hoped, provide a net speed increase. We implemented such a process by creating a second array called mutexes which held one mutex per item in the pieces array. At the start of the program, since the amount of elements in the buffer array can change, we used a for loop to initialize every mutex in the array, rather than using the PTHREAD_MUTEX_INITIALIZER that we used when we had one mutex. When we needed to acquire a lock on the buffer, we would lock the corresponding mutex at the same index the item in the buffer array we were modifying.

2.1 Benchmarking

To determine the speed difference between our two implementations, we needed a way to calculate the time spent on each. Initially we chose to use the **clock()** function provided in the standard library, but this proved to be unhelpful since the times were reported in processor time, not real time. This meant that 1 second of real time, with 10 threads executing concurrently, would result in us calculating 10 seconds of processor time. We could divide this result by the NROF_THREADS constant, but since NROF_THREADS threads might not always be executing at the same time (NROF_THREADS > NROF_PIECES). Instead, we chose to use the **timespec_get()** function also provided in the standard library in order to get the unix timestamp in microseconds. We packaged this into a useful **micros()** function that will, using **timespec_get()**, return a 64 bit integer representing the unix timetamp in microseconds. Calling this function at the start of our program, and once at the end of our program, we can find the difference in the two, giving us the time our program took to execute in microseconds.

Running each of the test cases provided in the assignment description, we produced the following results:

Table 1. Benchmarking Data

One Mutex						A Mutex Per Array Index					
Trial #	1	2	3	4	5	Trial #	1	2	3	4	5
1	0.20812	0.006906	0.002846	4.524179	35.83967	1	0.225654	0.006131	0.002877	2.752223	37.71092
2	0.184191	0.008796	0.003886	3.635718	41.87118	2	0.188805	0.006422	0.002946	3.453984	43.53026
3	0.181086	0.005339	0.003823	4.082306	37.2252	3	0.24938	0.007366	0.00332	2.640807	33.82147
4	0.178242	0.006173	0.002504	2.982273	36.04425	4	0.187944	0.006213	0.002772	2.942952	37.30113
5	0.170323	0.005491	0.002649	3.788837	35.77865	5	0.181602	0.007246	0.002847	3.358828	39.93522
6	0.18217	0.005562	0.003068	4.013266	36.65167	6	0.188847	0.007559	0.003732	3.803271	40.01626
Statistics						Statistics					
Average	0.184022	0.006378	0.003129	3.837763	37.2351	Average	0.203705	0.006823	0.003082	3.158678	38.71921
Min	0.170323	0.005339	0.002504	2.982273	35.77865	Min	0.181602	0.006131	0.002772	2.640807	33.82147
Max	0.20812	0.008796	0.003886	4.524179	41.87118	Max	0.24938	0.007559	0.003732	3.803271	43.53026
% Variance	21%	54%	44%	40%	16%	% Variance	33%	21%	31%	37%	25%
						Total Statistics					
Trial #	1	2	3	4	5	Avg. Diff	0.019683	0.000445	-4.7E-05	-0.67909	1.484108
Pieces	3000	30	30	30000	300000	Min Diff	0.011279	0.000792	0.000268	-0.34147	-1.95718
Threads	10	1	100	100	100	Max Diff	0.04126	-0.00124	-0.00015	-0.72091	1.659084
					- = faster	% Diff	11%	7%	-2%	-18%	4%

The results showed no meaningful increase in speed of our optimized version, although it would seem given the implementation, that there should be some difference. We are unable to identify the underlying cause of the lack of speed increase.

We have considered different approaches to implement the advanced feature other than a mutex. We think that, if we could have used an atomic function, such as the **atomic_fetch_xor** function, instead of using multiple mutexes, then our advanced feature could have shown a more meaningful increase in speed. This function first reads the value of the atomic object, then it performs a bitwise exclusive or, and then replaces the initial value of the atomic object with the result of the exclusive or.(cplusplus; Prinz & Crawford,2005). We thought that this function could increase the speed, as atomic operations cannot be interrupted, different threads cannot affect the value that is being handled.

3 References

1- *atomic::fetch_xor - C++ Reference. (n.d.-b).* www.cplusplus.com. Retrieved December 21, 2021, from
https://cplusplus.com/reference/atomic/atomic/fetch_xor/
2- Prinz, P., & Crawford, T. (2005). C in a Nutshell. " O'Reilly Media, Inc.".