

**Dalton Driscoll**

**Shem Louisy**

**Abumere Okhihan**

Wednesday, March 5<sup>th</sup>, 2025

EECE 4811/5811 Operating Systems Spring 2025

Professor Tseng

#### Homework 4

### **What is the ABA Problem:**

**[1]** The ABA problem occurs in concurrent programming when a thread reads a value from a shared memory location, and then later, it assumes that no other thread has modified that value. However, in the time between the read and the action taken based on that value, another thread might change the value and then revert it back to its original state. The issue arises because the value ends up being the same as it was initially, so the first thread assumes no change has occurred. This creates a problem because the thread may proceed based on an incorrect assumption, leading to errors or inconsistencies in the program.

### **Why our Enqueue Didn't work:**

**[2]** In the enqueue function, the ABA problem affects the Tail pointer of the queue. The function checks the Tail pointer to see if it's unchanged before performing a compare-and-swap (CAS) operation. **[4]** However, between the time it reads the Tail and attempts to update it, another thread might modify the Tail (e.g., by changing it temporarily to a different node and then restoring it to the original one). Since the Tail pointer eventually reverts back to its initial value, the CAS operation incorrectly assumes that no change has happened, even though the queue structure might have been altered in between. This leads the function to incorrectly proceed with its operation, causing the enqueue to fail or behave in an unexpected way.

## Comparing Queue algorithms

The program benchmarks the performance of two concurrent queue implementations: the Michael and Scott queue (MS queue) and a lock-free queue (LF queue). **[4]** The MS queue uses a mutex lock while the LF queue uses atomic operations such as CAS and atomic loading. The `createQueues` function initializes both queues, allocating memory and calling their respective initialization functions. The program then defines two functions, `thread_enqueue` and `thread_dequeue`, which enqueue and dequeue a fixed number of items (`Item_Count`) into/from the chosen queue type. Each function records execution time using `clock_gettime` to measure the duration of enqueueing and dequeuing operations.

The main function creates and launches multiple threads to perform enqueue and dequeue operations concurrently, measuring the performance of each queue implementation. The goal here is to increase race conditions. In main, the program first checks if an argument is provided to set `Item_Count`, determining how many elements will be enqueued and dequeued. It then spawns and joins threads for both queue types, testing their enqueue and dequeue times in a multi-threaded environment. The first two threads test enqueueing into the MS queue, followed by two threads testing enqueueing into the LF queue. The same approach is used for dequeuing. After execution, the program cleans up by deleting both queue structures.

## Results

The results show that the MS queue and LF queue have similar enqueue times for lower item counts, but as the number of items increases, the \*\*LF queue

slows down during enqueue operations\*\*, while the MS queue maintains relatively consistent times. This happens because, in the MS queue, only one thread can hold the lock at a time, and if multiple threads attempt to enqueue simultaneously, the thread that acquires the lock first determines the execution order, causing delays. However, the LF queue experiences increased enqueue times as more nodes are added, likely due to the overhead of managing atomic operations and memory consistency in a lock-free environment. Additionally, the LF queue encounters a segmentation fault at high item counts, possibly due to an unchecked memory allocation or improper handling of atomic operations. In dequeue operations, the LF queue is significantly faster than the MS queue, as it avoids locks and instead relies on atomic operations, allowing multiple threads to dequeue in parallel. The MS queue, despite being slower at dequeuing, remains more stable because it ensures safe memory access through locks, reducing the risk of crashes but introducing contention as multiple threads wait for access.


```
[shem@Shems-MacBook-Pro-2 HW4 % ./main 300
Testing Michael and Scott queue enqueue times for 300 items:
Thread Execution time = 0.000023 seconds
Thread Execution time = 0.000014 seconds
Testing Lock Free queue enqueue times for 300 items:
Thread Execution time = 0.000088 seconds
Thread Execution time = 0.000088 seconds
Testing Michael and Scott queue dequeue times for 300 items:
Thread Execution time = 0.000030 seconds
Thread Execution time = 0.000048 seconds
Testing Lock Free queue dequeue times for 300 items:
Thread Execution time = 0.000018 seconds
Thread Execution time = 0.000018 seconds
shem@Shems-MacBook-Pro-2 HW4 %
```

## Instructions

Compile using GCC in a Mac (*preferred*) , Linux terminal or Windows command prompt.

Run the compiled file with the node count parameter.

```
[shem@Shems-MacBook-Pro-2 HW4 % gcc -o main main.c lf_queue.c ms_queue.c  
[shem@Shems-MacBook-Pro-2 HW4 % ./main 300
```



Node Count

## References:

[1] T. Hoefler, "Concurrency Theory," ETH Zurich, 2020. <https://spcl.inf.ethz.ch/Teaching/2020-pp/lectures/PP-l21-ConcurrencyTheory.pdf>. Accessed: 03-Mar-2025.

*Torsten Hoefler, who owns the lecture, is a computer science professor at ETH Zurich. ETH Zurich is regarded as one of the most prestigious universities in the world, therefore leading the team to believe this source is very credible.*

[2] M. L. Scott, "Nonblocking Algorithms and the ABA Problem," in *Proceedings of the 1996 ACM Symposium on Principles of Distributed Computing (PODC)*, Philadelphia, PA, USA, 1996, pp. 71–80. [https://www.cs.rochester.edu/~scott/papers/1996\\_PODC\\_queues.pdf](https://www.cs.rochester.edu/~scott/papers/1996_PODC_queues.pdf). Accessed: 03-Mar-2025.

*Michael Scott, an author of this research paper, is a computer science professor at University of Rochester. He has many publications and is widely recognized within the realm of concurrent programming. This article has also been cited over 1,000 times further proving its credibility.*

[3] S. H. Shanor, "An Optimistic Approach to Lock-Free FIFO Queues" MIT CSAIL, 2008. [https://people.csail.mit.edu/shanir/publications/FIFO\\_Queues.pdf](https://people.csail.mit.edu/shanir/publications/FIFO_Queues.pdf). Accessed: 03-Mar-2025.

*This is a credible resource because it is written by Nir Shavit and Edya Ladan-Mozes, a computer science professor and student, from MIT. This paper also has many citations and is proven to be credible.*

[4] M. M. Michael and M. L. Scott, "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms," *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, Philadelphia, PA, USA, 1996, pp. 267-275. [https://www.cs.rochester.edu/~scott/papers/1996\\_PODC\\_queues.pdf](https://www.cs.rochester.edu/~scott/papers/1996_PODC_queues.pdf).

*The authors Michael M. Michael and Michael L. Scott are well-known researchers in concurrent and parallel computing. Michael L. Scott, in particular, is a professor at the University of Rochester and has contributed significantly to research in synchronization, concurrent data structures, and parallel processing. His work is frequently cited in the field.*

