Project - wfqueue

Implementation of a wait free queue in C

- Ashish Jindal

Overview

This project is about implementing a wait-free queue in an environment where garbage collector is not present for e.g. in our case we will be giving the implementation in C language. We will use the algorithm described in [1] for implementing a multi enqueuer multi de-queuer wait free queue.

Motivation

1. Currently there is no known implementation of a wait-free garbage collector, so a wait-free data structure implemented in a garbage collector based environment can't have a true wait-free implementation .
2. Wait Freedom provides stronger guarantees for starvation freedom in the system.
3. Every step in a wait-free algorithm has a bound on the number of steps it will take to complete, and such a property is very crucial in real time systems.

Specifications

1. Wait Free Queue algorithm:
    a. Algorithm is based on a helping scheme [1] in which faster threads help slower peers to complete their pending operations.
    b. Wait freedom is achieved by assigning each operation a dynamic age based priority and making threads with younger operations to help the ones with older operations [1].
2. Blocking Queue:
    a. We Implemented a blocking queue supporting multiple enqueue and dequeue operations simultaneously.
    b. The above implementation will then be used to compare the performance of our wait-free queue implementation.
3. Hazard Pointer:

a. Because of the absence of a garbage collector we need another mechanism to free unreferenced memory in our wait free queue.
b. We plan to use Hazard pointers mechanism [3] to manage unreferenced memory and free that memory in a safe way.

Algorithm[1]

1. Wait-free Queue implementation is based on the underlying singly-linked list and holds two references to the head and tail of the list, respectively called head and tail.
2. Every thread ti starting an operation on the queue chooses a phase number, which is higher than phases of threads that have previously chosen phase numbers for their operations.
3. Then it records this number, along with some additional information on the operation it is going to execute on the queue, in a special state array.
4. Next, ti traverses the state array and looks for threads with entries containing a phase number that is smaller or equal than the one chosen by ti.
5. Once such a thread tj is found (it can be also ti itself), ti tries to help it execute its operation on the queue, i.e., to insert tj ’s node into the queue (enqueue) or to remove the first node from the queue on behalf of tj (dequeue).
6. The thread ti learns the details on tj ’s operation from the state array. Finally, when ti has tried to help all other threads with a phase number not larger than it has, it can safely return the execution to the caller of the (dequeue or enqueue) operation. That is, ti can be confident that its own operation on the queue has been completed either by ti or by some other concurrently running and helping thread.

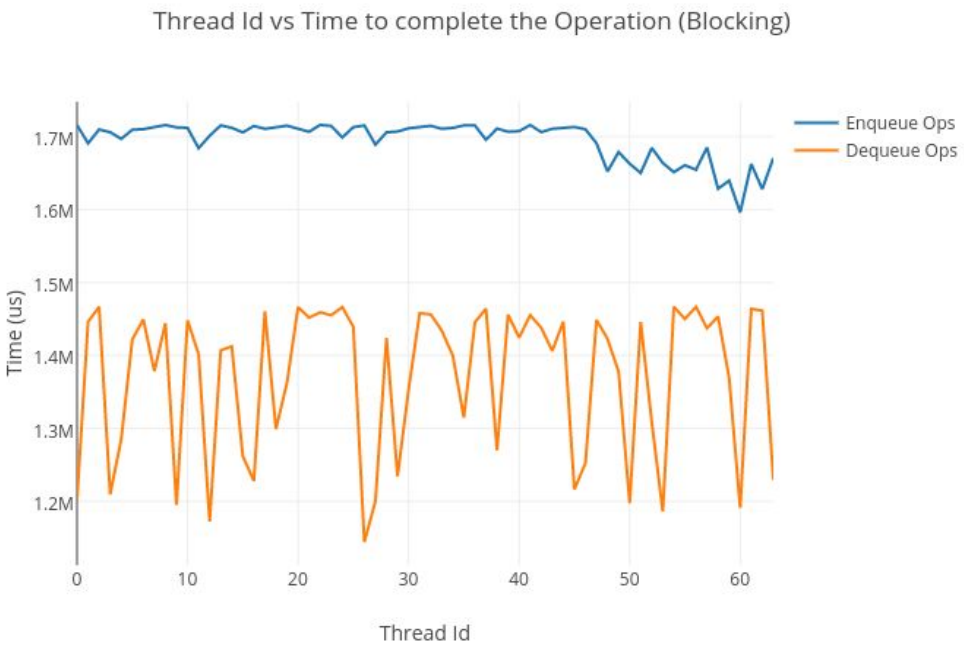
Evaluation

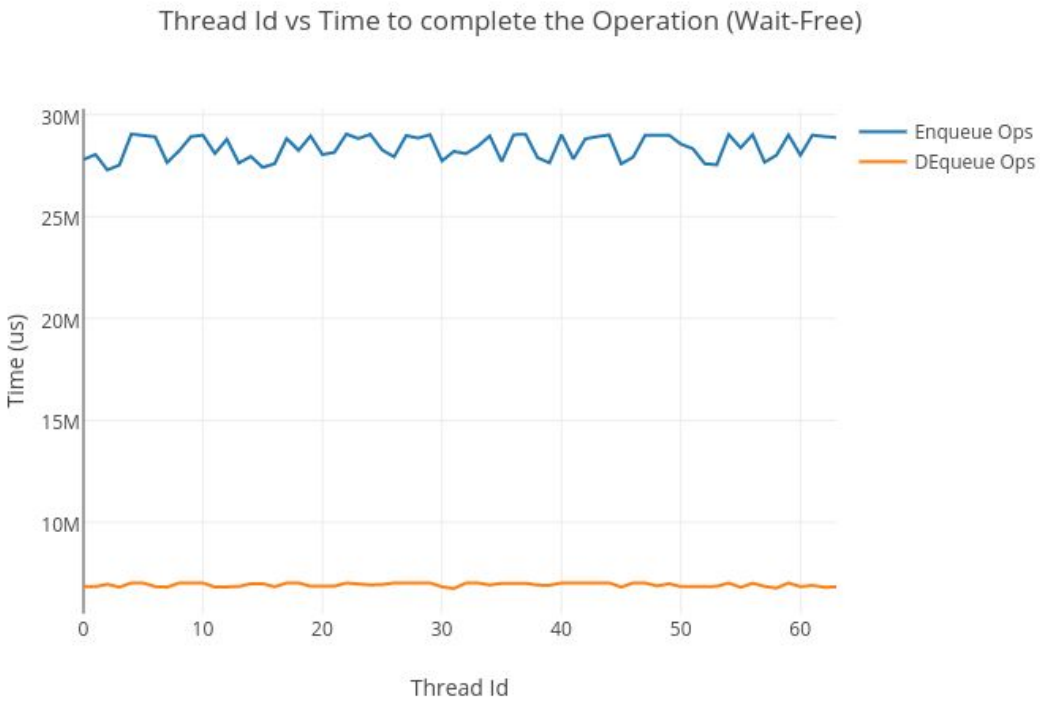
1. In the paper [1], they evaluated the performance of the wait free queue by comparing it with a lock-free implementation [2] of queue, which is one of the most efficient dynamically allocated queue algorithm. We compared our wait queues performance with a blocking queue which has a comparable performance.
2. The evaluation was done on a ubuntu machine on Amazon EC2, with up to 64 cores, and number of threads varying between 1 to 64.

3. Following benchmark is considered for the performance analysis -
    a. N enqueue operation are first performed concurrently by all the threads, which is then followed by same number of concurrent dequeue operations.
4. We measure the completion time of each of the algorithms as a function of the number of concurrent threads. Each thread performs up to1, 000, 000 iterations. Thus, in the above described benchmark, given the number of threads k, the number of operations is 2000000·k, divided equally between enqueue and dequeue..
5. Wait-Free algorithm will be slower than its lock free and blocking counterpart, but the idea is can we sacrifice a bit of performance to gain stronger guarantees.

Results

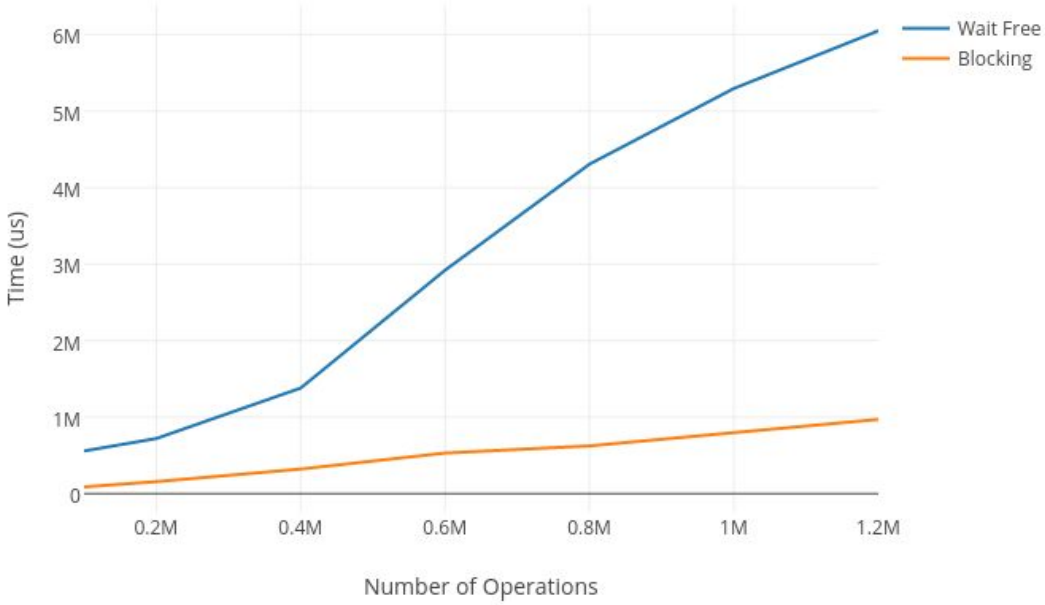
Because of a bug in our implementation related to volatile keyword, the implementation doesn't work if we use any compiler optimizations. So following results are a bit biased towards blocking queue implementation, we plan to fix the bug in future to get correct result-



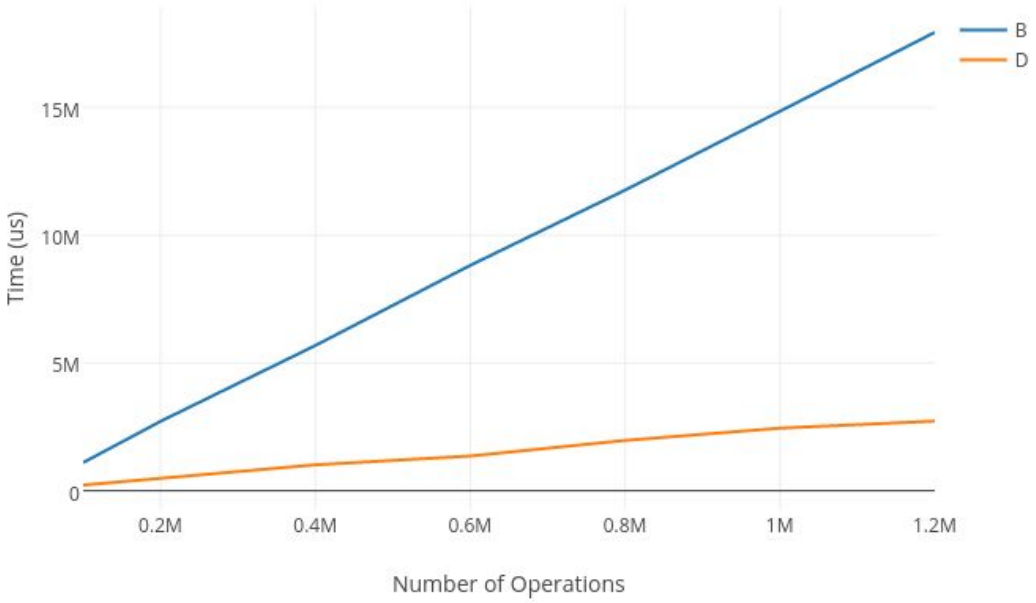


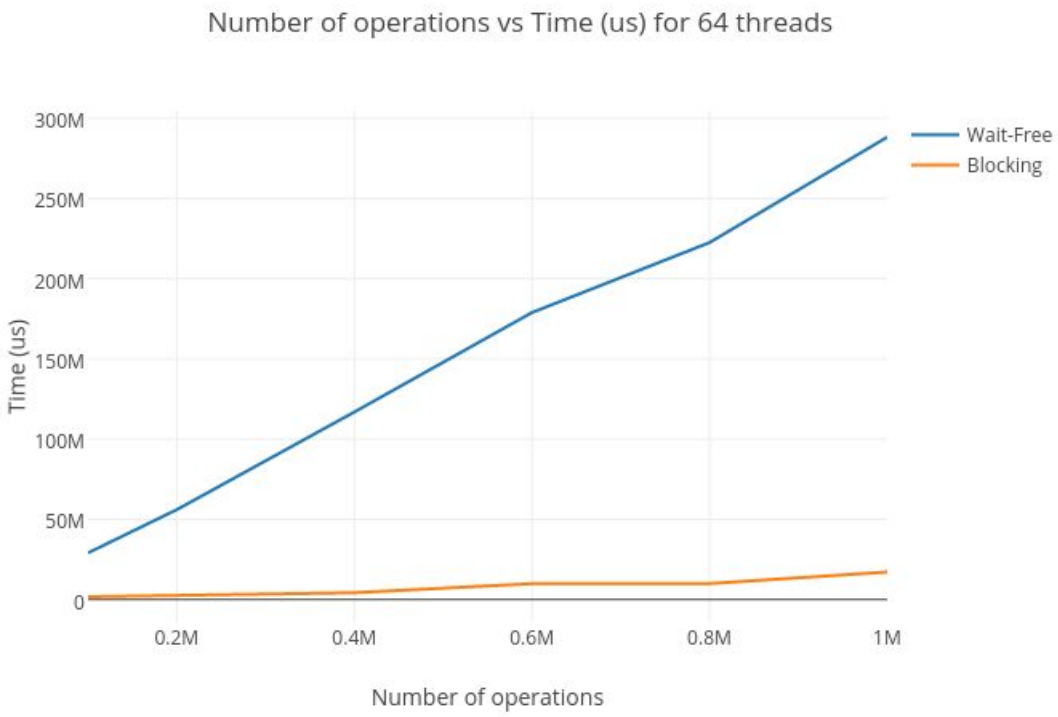
Expected result from above plots was a uniform variance and lesser standard deviation in the wait free implementation. Though the processing time is more than blocking counterpart, but each thread should complete its work in ome near constant time.

Number of operations vs Time (us) for 4 threads



Number of operations vs Time (us) for 8 threads





Conclusion

1. Wait Freedom not for performance centric systems
2. Real time systems with hard timing constraints are a good application for Wait Free structures.
3. Given Wait free algorithm doesn't scale well with increasing number of threads.
4. Complexity Vs Progress Guarantees tradeoff while choosing between Blocking/Lock Free/Wait Free

References

1. Alex Kogan and Erez Petrank .Wait-Free Queues With Multiple Enqueuers and Dequeuers. In ACM symposium on Principles and Practices of Parallel Programming (PPoPP), 2011.

2. M. M. Michael and M. L. Scott. Simple, fast, and practical nonblocking and blocking concurrent queue algorithms. In Proc. ACM Symposium on Principles of Distributed Computing (PODC), pages 267–275, 1996.

3. M. M. Michael. Hazard pointers: Safe memory reclamation for lockfree objects. IEEE Trans. Parallel Distrib. Syst., 15(6):491–504, 2004.