Fairness vs. Linearizability in a Concurrent FIFO Queue

Mike Dodds, Andreas Haas, Christoph M. Kirsch

We are interested in the design and implementations as well as correctness conditions of efficient and multicore-scalable concurrent data-structures. In particular, we are exploring tradeoffs between their performance, scalability, and semantics. In this paper we discuss the relation between linearizability and what we call fairness of concurrent FIFO queues.

Intuitively, in a correct FIFO queue an element which gets enqueued first also gets dequeued first. However, in a concurrent environment the order in which elements get inserted and removed is not clear as some operations may execute faster than others. For example, a dequeue operation may start before but return after another dequeue operation. Thereby an element in the queue may get returned after elements which were inserted later but get removed by faster dequeue operations. We say that a FIFO queue implementation is fair if any element is returned by its dequeue operation before any other element is returned which was inserted later. An element b is inserted later than an element a if the insert operation of a returns before the invocation of the insert operation of b according to a global time. A similar kind of fairness is discussed in [HKLP12].

The correctness condition linearizability [HW90] tolerates unfair behavior as it only requires that there exists a linearization where the queue shows correct behavior independent of the times dequeue operations actually return. In this paper we presents an optimization of a recently designed concurrent FIFO queue which, although incorrect according to linearizability, is still quiescently consistent [AHS94] and moreover provides more fairness and performance than a second optimization which is linearizable.

The key idea of the underlying FIFO queue is that elements are timestamped before they are enqueued into an unordered buffer. Dequeue operations search through the buffer and try to remove the element with the earliest timestamp. If a global atomic counter is used to create unique and strictly increasing timestamps, then an optimization described in [HPS13] is possible: Assume there exist two elements a and b in the queue with timestamps 4 and 5, respectively, and assume that two dequeue operations d_1 and d_2 are executed concurrently. If d_2 knows that there exists a second dequeue operation d_1, then d_2 can remove b immediately knowing that the concurrent operation d_1 will remove a. Intuitively, d_2 considers the element a as already removed.

The optimization can be applied naively on our FIFO queue as follows: In addition to timestamping elements, also dequeue operations get timestamped. Each dequeue operation removes the element with the same timestamp as itself. In the following we call this optimization the linearizable optimization. The linearizable optimization has several drawbacks: If an enqueue operation gets interrupted between acquiring a timestamp and inserting the element into the buffer, then the matching dequeue operation would have to wait for the enqueue operation to complete or it would have to make sure that a later dequeue operation will remove the element. If the dequeue operation gets interrupted after acquiring the timestamp, then the queue shows unfair behavior.

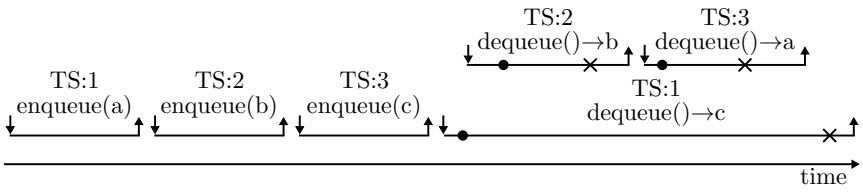

Figure 1: Example where the optimization is fair but unsound according to linearizability.

Both problems are solved by what we call the fair optimization: A dequeue operation still acquires a timestamp, but different to the linearizable optimization it does not try to find and remove the element with the same timestamp but any element with the same or an earlier timestamp. If no such element is found, then the dequeue operation removes the element with the earliest timestamp it encounter in the buffer.

The queue with the fair optimization is quiescently consistent because if a dequeue operation is executed in a quiescent state, then it will return the element with the same timestamp. However, the optimization is incorrect according linearizability. Consider the execution in Figure 1 where horizontal lines represent the execution times of operations, $\downarrow$ and $\uparrow$ mark the invocation and response of operations, respectively, and $\bullet$ and $\times$ mark the time when a dequeue operation acquires a timestamp and removes an element, respectively. First three elements a, b, and c are enqueued sequentially with timestamps 1, 2, and 3. Then a dequeue operation acquires a timestamp 1 but gets interrupted afterwards. A second dequeue operation acquires a timestamp 2 and removes b. After the response of the second dequeue operation a third dequeue operation starts, acquires a timestamp 3, and removes the element a because the timestamp of a is lower than the timestamp of the dequeue operation.

In this execution, the element a is inserted before but removed after b which clearly violates the sequential specification of a queue. Had the dequeue operation with timestamp 1 returned a, the behavior would have been linearizable, but a would have been treated even more unfairly than in our example execution as both b and c would have been returned before a then.

The example shows that linearizability does not imply fairness, and that fairness does not require linearizability. Preliminary measurements showed that the optimized queue outperforms most state-of-the-art concurrent FIFO queues in both execution time and fairness.

References

[AHS94] J. Aspnes, M. Herlihy, and N. Shavit. Counting networks. *Journal of the ACM*, 41, 1994.

[HKLP12] A. Haas, C.M. Kirsch, M. Lippautz, and H. Payer. How FIFO is your concurrent FIFO queue? In *RACES*. ACM, 2012.

[HPS13] T.A. Henzinger, H. Payer, and A. Sezgin. Replacing competition with cooperation to achieve scalable lock-free FIFO queues. Technical Report IST-2013-124-v1+1, IST Austria, 2013.

[HW90] M.P. Herlihy and J.M. Wing. Linearizability: a correctness condition for concurrent objects. *TOPLAS*, 12(3), 1990.