POSTER: Restricted Memory-Friendly Lock-Free Bounded Queues

Nikita Koval
JetBrains
nikita.koval@jetbrains.com

Vitaly Aksenov
ITMO University
aksenov@itmo.ru

1 Introduction

Multi-producer multi-consumer FIFO queue is one of the fundamental concurrent data structures used in software systems. A lot of progress has been done on designing concurrent *bounded* and *unbounded* queues [1–10]. As previous works show, it is extremely hard to come up with an *efficient* algorithm. There are two orthogonal ways to improve the performance of fair concurrent queues: reducing the number of `compare-and-swap` (CAS) calls, and making queues more *memory-friendly* by reducing the number of allocations. The most up-to-date efficient algorithms choose the first path and use more scalable `fetch-and-add` (FAA) instead of `CAS` [3, 4, 10]. For the second path, the standard way to design *memory-friendly* versions is to implement queues on top of arrays [2–4, 10]. For unbounded queues it is reasonable to allocate memory in chunks, constructing a linked queue on them; this approach significantly improves the performance. The bounded queues are more memory-friendly by design: they are represented as a fixed-sized array of elements even in theory. However, most of the bounded queue implementations still have issues with memory allocations — typically, they either use descriptors [5, 8] or store some additional meta-information along with the elements [1, 6, 7, 9].

The arising question is whether it is possible to design a lock-free bounded queue that uses only $O(1)$ (independent of the capacity) additional memory. Surprisingly, we found only one paper [7] that partially answers this question. However, their algorithm is subject to ABA problem even if all the inserted elements are distinct: the algorithm uses only two different `null` elements; thus, if one thread becomes asleep for two “rounds” (i.e., the pointers `head` and `tail` made two traversals through the whole array of the queue), it can incorrectly place the element into the queue after the wake-up. In our algorithm, we fix this problem by using an infinite supply of `null` values. Moreover, our algorithm is much simpler and clear for understanding.

Our contribution This paper presents a new lock-free bounded queue algorithm that uses only $O(1)$ additional memory and has two practical constraints. At first, it requires all the elements to be distinct; thus, avoiding the ABA problem on storing and retrieving. We find this constraint reasonable since many software systems use queues for tasks or identifiers, which are usually unique, for example, maintained by a garbage collector. The second constraint is that we are provided with an unlimited supply of versioned `null` values, so that we can use different `null`-s for different rounds. This condition is also practical and can be achieved by stealing one bit from addresses (values) to mark them as `null` values and use the rest of the address (value) for storing versions. We believe that the proposed algorithm is the first step towards indeed memory-friendly queues and, further, memory-friendly variants of other data structures.

2 Algorithm Description

Specification. We define the bounded queue as a standard FIFO queue with the limited capacity so that the number of stored in the queue elements cannot exceed it. The following operations are supported:

- `offer(e)` inserts the element `e` and returns `true` if the queue is not full, returns `false` otherwise;
- `poll` returns the oldest element, or returns `null` if the queue is empty.

Initialization. The structure of our queue is presented in listing below. All elements are stored in array `a` (line 2) of the queue capacity `size`; it is initially filled with nulls of round 0 ($\perp_0$). To know the target positions of next `offer` and `poll`, we maintain two counters, `offers` and `polls` — the total numbers of completed `offer` and `poll` invocations; taken by modulo `CAPACITY` these counters indicate the proper slots. The queue is empty when these counters coincide (`polls == offers`), and is full if their difference `offers - polls` is equal to the capacity.

```
1 class BoundedQueue<T>(val CAPACITY: Int) {
2   val a: T[] = Array(CAPACITY) // a[i] = ⊥₀
3   var offers: Long = 0L
4   var polls: Long = 0L
5 }
```

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

PPoPP '20, February 22–26, 2020, San Diego, CA, USA
© 2020 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-6818-6/20/02.
https://doi.org/10.1145/3332466.3374508

`Offer` algorithm. The pseudo-code is presented in the listing below. At first, the algorithm atomically snapshots the monotonously increasing `offers` and `polls` counters using the *double-collect* technique (lines 3–6).

After that, it checks whether the queue is full (line 8). Note, that there can be a concurrent `poll` invocation that already retrieved the element but has not increased `polls` counter yet — we can linearize the fullness detection before this `poll`.

At the next step, the algorithm tries to put the element into the slot using CAS from `null` value to the element (line 12); this CAS synchronizes parallel producers so that only one of them succeeds at this slot. It is possible for `offer` to suspend and skip its round: another element can be inserted at this slot and further retrieved. In this case, we need a mechanism to detect that the round is missed, and fail the element insertion CAS. For this purpose, we use different `null` values for each round; they are usually implemented via stealing the highest bit from element addresses or values.

After the element insertion attempt, we guarantee that either the current operation or a concurrent one has been succeeded. Therefore, the algorithm increments the number of completed `offer` invocations (line 14), and returns `true` if the algorithm successfully inserted the element, retrying the whole operation otherwise.

```
1  // All inputs should be different
2  fun offer(e: T): Bool = while (true) {
3    o := offers
4    p := polls
5    // is `o` still the same?
6    if o != offers: continue
7    // is the queue full?
8    if o == p + CAPACITY: return false
9    // try to perform the offer
10   i := o % CAPACITY
11   round := o / CAPACITY
12   success := CAS(&a[i], $\bot_{round}$, e)
13   // increment the counter
14   CAS(&offers, o, o + 1)
15   if success: return true
16 }
```

`Poll` algorithm. The pseudo-code is presented in the listing below. Roughly, the algorithm reads the counters and the element to be retrieved (lines 2–7), checks whether the queue is not empty (line 9), exchanges the element with the null value for the next round (line 17), and increments the number of successful polls at the end (line 19). Considering the constraint that all elements are different, CAS for retrieving an element can succeed only if the slot has not been changed.

Similarly to the `offer` operation, we use the double-collect technique to get an atomic snapshot (lines 2–7).

When the algorithm checks whether the queue is empty (line 9), it can get into a situation, where the counters coincide, but the array contains one element — there can be a concurrent `offer` which successfully inserted its element, but has not updated the counter yet; we linearize the emptiness detection before this `offer`.

Since our `poll` algorithm increments the corresponding counter at the end, the element can be already taken at the point of getting the snapshot while the counter is still not updated. The algorithm checks whether the element from the snapshot is `null` for the next round, helping to increment the counter and retrying the operation in this case (lines 12–15).

```
1  fun poll(): T? = while (true) {
2    p := polls
3    o := offers
4    i := p % CAPACITY
5    e := a[i]
6    // is `p` still the same?
7    if p != polls: continue
8    // is the queue empty?
9    if p == o: return null
10   // is the element already taken?
11   nextRound = p / CAPACITY + 1
12   if e == $\bot_{nextRound}$ {
13     CAS(&polls, p, p + 1) // helping
14     continue
15   }
16   // try to retrieve the element
17   success := CAS(&a[i], e, $\bot_{nextRound}$)
18   // increment the counter
19   CAS(&polls, p, p + 1)
20   if success: return e
21 }
```

Acknowledgments. This research is partially supported by the Government of Russian Federation (Grant 08-08).

References

[1] Steven Feldman and Damian Dechev. A wait-free multi-producer multi-consumer ring buffer. *ACM SIGAPP Applied Computing Review*, 15(3):59–71, 2015.

[2] Anders Gidenstam, Håkan Sundell, and Philippas Tsigas. Cache-aware lock-free queues for multiple producers/consumers and weak memory consistency. In *PODC*, pages 302–317, 2010.

[3] Adam Morrison and Yehuda Afek. Fast concurrent queues for x86 processors. In *ACM SIGPLAN Notices*, volume 48, pages 103–112, 2013.

[4] Ruslan Nikolaev. A scalable, portable, and memory-efficient lock-free fifo queue. In *DISC*, 2019.

[5] Peter Pirkelbauer, Reed Milewicz, and Juan Felipe Gonzalez. A portable lock-free bounded queue. In *ICAPP*, pages 55–73, 2016.

[6] Niloufar Shafiei. Non-blocking array-based algorithms for stacks and queues. In *ICDCN*, pages 55–66, 2009.

[7] Philippas Tsigas and Yi Zhang. A simple, fast and scalable non-blocking concurrent fifo queue for shared memory multiprocessor systems. In *SPAA*, pages 134–143, 2001.

[8] John D Valois. Implementing lock-free queues. In *ICPADS*, pages 64–69, 1994.

[9] Dmitry Vyukov. Bounded MPMC queue. http://www.1024cores.net/home/lock-free-algorithms/queues/bounded-mpmc-queue.

[10] Chaoran Yang and John Mellor-Crummey. A wait-free queue as fast as fetch-and-add. *ACM SIGPLAN Notices*, 51(8):16, 2016.