Memory-Friendly Lock-Free Bounded Queues

Vitaly Aksenov
ITMO University, St. Petersburg, Russia
aksenov.vitaly@gmail.com

Nikita Koval
JetBrains, St. Petersburg, Russia
ndkoval@ya.ru

Motivation

How to improve FIFO queues performance?

- Reduce the number of failed `CAS` (e.g., by replacing them with `FAA` operations)
- Make the algorithm more *memory-friendly*

How to achieve *memory-friendliness*?

- For unbounded queues: allocate memory in chunks (constructing a linked queue on them)
- For bounded queues: re-using a single pre-allocated array with the corresponding size
- Use the fewest descriptors or metadata possible

Is there a lock-free *bounded* queue that uses *O(1)* additional memory (no metadata or descriptors)?

Bounded Queue

Bounded queue is a FIFO queue which is limited in capacity:

- `offer(e)` inserts the element `e` and returns `true` if the queue is not full, returns `false` otherwise
- `poll()` retrieves and returns the first element if the queue is not empty, returns `null` otherwise

Algorithm Assumptions

- All inserting elements should be distinct (many software systems use queues for unique tasks or identifiers)
- We have an unlimited supply of versioned `null` values, so that we can use different `null`-s for different rounds (can be achieved by stealing one bit from values)

These assumptions guarantee the lack of the ABA problem on array cells updates.

Implementation

```
class BoundedQueue<T>(val CAPACITY: Int) {
  val a: T[] = Array(CAPACITY) // a[i] = ⊥_0
  var offers: Long = 0L
  var polls: Long = 0L
}

fun offer(e: T): Bool = while (true) {
  o := offers
  p := polls
  // is 'o' still the same?
  if o != offers: continue
  // is the queue full?
  if o == p + CAPACITY: return false
  // try to perform the offer
  i := o % CAPACITY
  round := o / CAPACITY
  success := CAS(&a[i], ⊥_round, e)
  // increment the counter
  CAS(&offers, o, o + 1)
  if success: return true
}
```

```
fun poll(): T? = while (true) {
  p := polls
  o := offers
  i := p % CAPACITY
  e := a[i]
  // is 'p' still the same?
  if p != polls: continue
  // is the queue empty?
  if p == o: return null
  // is the element already taken?
  nextRound = p / CAPACITY + 1
  if e == ⊥_nextRound {
    CAS(&polls, p, p + 1) // helping
    continue
  }
  // try to retrieve the element
  success := CAS(&a[i], e, ⊥_nextRound)
  // increment the counter
  CAS(&polls, p, p + 1)
  if success: return e
}
```

Theoretical Result

Def. An implementation is *value-preserving* if inserting values are subject only to reads, writes, and equality checks (including the ones during `CAS`-s). Thus, bit stealing is not allowed.

Def. Consider the arbitrary reachable state. Let $x_1, \ldots, x_n$ be the values that do not occur in the memory cells. Suppose we sequentially perform `offer`$(x_1), \ldots,$ `offer`(x_n) and reach the state M of the memory. An algorithm is *argument-independent* if after changing `offer`(x_i) to `offer`(v) all the memory cells with x_i in M now store v.

Theorem. If there are p processes working on value-preserving and argument-independent queue, any obstruction-free algorithm needs at least `CAPACITY` $+ O(p)$ memory cells.

Remarks

- The bound from the Theorem can be achieved with a lock-free algorithm on re-usable descriptors.
- As well as `null` values, elements should be distinct only between different rounds and can coincide during the same one.
- If the presented algorithm can be improved so that it supports indistinguishable either `null` or inserting values, it should be simple (but impossible) to solve the second problem.

Acknowledgements

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