FFQ: A Fast Single-Producer/Multiple-Consumer Concurrent FIFO Queue

Erratum

Sergei Arnautov, Christof Fetzer, Bohdan Trach
TU Dresden, Germany — first.last@tu-dresden.de

Pascal Felber
U. Neuchâtel, Switzerland — first.last@unine.ch

Algorithm 1 — FFQ^s: single-producer FIFO queue

```
1:  Type cell is:                                   ▷ Cell for holding data
2:      data ← NULL                                 ▷ Actual data (initially empty)
3:      rank ← 0                                    ▷ Rank of item (or 0 if cell unused)
4:      gap ← 0                                     ▷ Gap in numbering (skipped item)
5:  Variables:
6:      cells[N] ← array of cell                    ▷ Bounded array of cells (N ≥ 1)
7:      tail ← N                                    ▷ Tail counter (monotonically increasing)
8:      head ← N                                    ▷ Head counter (monotonically increasing)
9:  function FFQ_ENQ(data)                          ▷ Enqueue (single-producer)
10:     success ← FALSE
11:     while ¬success do                           ▷ Find empty cell...
12:         c ← cells[tail(mod N)]                  ▷ Try next cell
13:         if c.rank ≥ N then                      ▷ Cell used?
14:             c.gap ← tail                        ▷ Yes: skip it (gap in rank)
15:         else
16:             c.data ← data                       ▷ No: grab it
17:             c.rank ← tail                       ▷ Remember rank
18:             success ← TRUE
19:         tail ← tail+1                           ▷ Move to next cell
20: function FFQ_DEQ                                ▷ Dequeue (multi-consumers)
21:     rank ← fetch-and-inc(head)                  ▷ Get rank of next item
22:     c ← cells[rank(mod N)]                      ▷ Check associated cell
23:     success ← FALSE
24:     while ¬success do                           ▷ Find next used cell...
25:         if c.rank = rank then                   ▷ Cell used for rank?
26:             data ← c.data                       ▷ Yes: get item
27:             c.rank ← 0                          ▷ Recycle cell
28:             success ← TRUE
29:         else if c.gap ≥ rank ∧ |c.rank| ≠ rank then
30:             rank ← fetch-and-inc(head)          ▷ Cell skipped: ...
31:             c ← cells[rank(mod N)]              ▷ ...move to next cell
32:         else wait()                             ▷ Back off (producer still writing cell)
33:     return data                                 ▷ Return item
```

Algorithm 2 — FFQ^m: multi-producer FIFO queue

```
1:  function FFQ_ENQ(data)                          ▷ Enqueue (multi-producer)
2:      success ← FALSE
3:      while ¬success do                           ▷ Find empty cell...
4:          rank ← fetch-and-inc(tail)              ▷ Get next rank...
5:          c ← cells[rank(mod N)]                  ▷ ...and associated cell
6:          while (g ← c.gap) < rank do             ▷ Unless overtaken...
7:              if (r ← c.rank) ≠ 0 then            ▷ Cell used?
8:                  double-compare-and-set          ▷ Yes: skip it
        .....(⟨c.rank,c.gap⟩,⟨r,g⟩,⟨r,rank⟩)        ▷ ⇒ Set gap
9:              else if double-compare-and-set      ▷ No: use it
        .....(⟨c.rank,c.gap⟩,⟨0,g⟩,⟨−rank,g⟩) then  ▷ ⇒ Set rank
10:                 c.data ← data                   ▷ Store data
11:                 c.rank ← rank                   ▷ Remember rank
12:                 success ← TRUE
```

The algorithm now uses both positive and negative rank numbers to handle synchronization between producers and consumers. To denote ranks, we only use numbers such that $|rank| \geq N$ so that we can reserve lower values to indicate special cell states. We currently only use value 0 for a special purpose, hence the size N of the array must be at least 1. Note that this numbering will not change the behavior of the algorithm as positions are computed modulo N.

We now use constant 0 to denote empty cells and the absence of gaps in Algorithm 1 (Lines 3, 4, 13, and 27) and in Algorithm 2 (Lines 7 and 9).

The $tail$ and $head$ variables are initialized to N in Algorithm 1 (Lines 8 and 7). In the FFQ_DEQ() algorithm, the test at Line 29 in the original version [1] only verified that $c.rank \neq rank$ to handle the case of a slow consumer that might fail to dequeue an item because a fast producer would have created a gap since the time the consumer has done the check at Line 25. We slightly change this condition to handle multi-producer FFQ_ENQ() operations, because we will use negative ranks for producers to indicate that they are in the process of inserting an item. Now, if we find a negative rank whose absolute value is equal to the expected rank, we wait until this producer completes its insertion.

In the FFQ_ENQ() algorithm, when a producer enqueues an item, it first sets the rank to a special value in Algorithm 2 (Line 9) before storing data and updating the rank to its final value. Instead of having all producers use the same special value [1], which could lead to a subtle race condition where one fast producer would catch up a slow producer and both will deadlock trying to insert data in the same cell, we now use negative rank values to indicate that a producer is in the process of inserting an element. Since no two producers can get the same rank, this guarantees that these negative values will be distinct for two different producers.

Acknowledgements

We are grateful to Andreia Craveiro Ramalhete for pointing out the race condition in the original paper and helping with the development of the fix.

References

[1] Arnautov, S., Felber, P., Fetzer, C., and Trach, B. FFQ: A fast single-producer/multiple-consumer concurrent fifo queue. In *31st IEEE International Parallel & Distributed Processing Symposium* (May 2017).