Slot-queue - An optimized wait-free distributed MPSC

1. Motivation

A good example of a wait-free MPSC has been presented in [1]. In this paper, the authors propose a novel tree-structure and a min-timestamp scheme that allow both `enqueue` and `dequeue` to be wait-free and always complete in $\Theta(\log n)$ where n is the number of enqueuers.

We have tried to port this algorithm to distributed context using MPI. The most problematic issue was that the original algorithm uses load-link/ store-conditional (LL/SC). To adapt to MPI, we have to propose some modification to the original algorithm to make it use only compare-and-swap (CAS). Even though the resulting algorithm pretty much preserve the original algorithm's characteristic, that is wait-freedom and time complexity of $\Theta(\log n)$, we have to be aware that this is $\Theta(\log n)$ remote operations (and in reality, there are about $5 * \log n$ remote operations), which is very expensive.

Therefore, to be more suitable for distributed context, we propose a new algorithm that's inspired by LTQueue, in which both `enqueue` and `dequeue` only perform a constant number of remote operations, at the cost of `dequeue` having to perform $\Theta(n)$ local operations, where n is the number of enqueuers. Because remote operations are much more expensive, this might be a worthy tradeoff.

2. Structure

Each enqueue will have a local SPSC as in LTQueue [1] that supports `dequeue`, `enqueue` and `readFront`. There's a global queue whose entries store the minimum timestamp of the corresponding enqueuer's local SPSC.
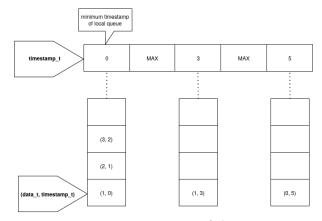

Figure 1: Basic structure of slot queue

3. Pseudocode

3.1. SPSC

The SPSC of [1] is kept in tact, except that we change it into a circular buffer implementation.

Types

```
data_t = The type of data stored
spsc_t = The type of the local SPSC
  record
    First: int
    Last: int
    Capacity: int
    Data: an array of data_t of capacity
    Capacity
  end
```

Shared variables

```
First: index of the first undequeued entry
Last: index of the first unenqueued entry
```

Initialization

```
First = Last = 0
Set Capacity and allocate array.
```

The procedures are given as follows.

Procedure 1: `spsc_enqueue(v:  data_t)` **returns** `bool`

---

1 **if** `(Last + 1 == First)`
2 | **return** `false`
3 `Data[Last] = v`
4 `Last = (Last + 1) % Capacity`
5 **return** `true`

Procedure 2: `spsc_dequeue()` **returns** `data_t`

---

6 **if** `(First == Last)` **return** ⊥
7 `res = Data[First]`
8 `First = (First + 1) % Capacity`
9 **return** `res`

Procedure 3: `spsc_readFront` **returns** `data_t`

---

10 **if** `(First == Last)`
11 | **return** ⊥
12 **return** `Data[First]`

3.2. Slot-queue

The slot-queue types and structures are given as follows:

Types
| `data_t` = The type of data stored
| `timestamp_t` = `uint64_t`
| `spsc_t` = The type of the local SPSC

Shared variables
| `slots`: An array of `timestamp_t` with the number of entries equal the number of enqueuers
| `spscs`: An array of `spsc_t` with the number of entries equal the number of enqueuers
| `counter`: `uint64_t`

Initialization
| Init all local SPSCs.

| Init `slots` entries to `MAX`.

The enqueue operations are given as follows:

Procedure 4: `enqueue(rank: int, v: data_t)` **returns** `bool`

---

1 `timestamp = FFA(counter)`
2 `value = (v, timestamp)`
3 `res = spsc_enqueue(spscs[rank], value)`
4 **if** `(res && spsc_readFront(spscs[rank]) == value)`
5 | `propagateEnqueue(rank, timestamp)`
6 **return** `res`

Procedure 5: `propagateEnqueue(rank: int, ts: timestamp_t)`

---

7 `slots[rank] = ts`

The dequeue operations are given as follows:

Procedure 6: `dequeue()` **returns** `data_t`

---

8 `rank = readMinimumRank()`
9 **if** `(rank == DUMMY || slots[rank] == MAX)`
10 | **return** ⊥
11 `res = spsc_dequeue(spscs[rank])`
12 **if** `res == ⊥` **return** ⊥
13 `propagateDequeue(rank)`
14 **return** `res`

Procedure 7: `readMinimumRank()` **returns** `int`

```
16  rank = length(slots)
17  min-timestamp = MAX
18  for index in 0..length(slots)
19  │  timestamp = slots[index]
20  │  if min-timestamp < timestamp
21  │  │  rank = index
22  │  │  min-timestamp = timestamp
23  old-rank = rank
24  for index in 0..old-rank
25  │  timestamp = slots[index]
26  │  if min-timestamp < timestamp
27  │  │  rank = index
28  │  │  min-timestamp = timestamp
29  return rank == length(slots) ? DUMMY :
       rank
```

Procedure 8: `propagateDequeue(rank: int)`

```
30  old-timestamp = slots[rank]
31  front = spsc_readFront(spscs[rank])
32  if (front != ⊥)
33  │  slots[rank] = front.timestamp
34  │  return
35  CAS(&slots[rank],      old-timestamp,
    front.timestamp)
```

---

4. Linearizability

5. Wait-freedom

6. Memory-safety

References

[1] P. Jayanti and S. Petrovic, "Logarithmic-time single deleter, multiple inserter wait-free queues and stacks," 2005, *Springer-Verlag*. doi: 10.1007/11590156_33.