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, which is very expensive. We have estimated that for an `enqueue` or a `dequeue` operation in our initial LTQueue version, there are about $2 * \log n$ to $10 * \log n$ remote operations, depending on data placements and the current state of the LTQueue.

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.

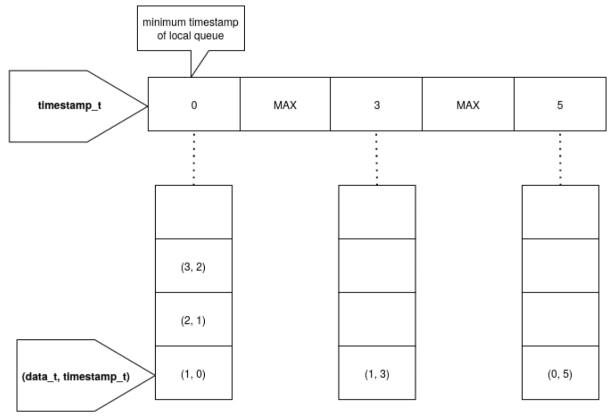

Figure 1: Basic structure of slot queue

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`

```
if (Last + 1 == First)
  return false
Data[Last] = v
Last = (Last + 1) % Capacity
return true
```

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

```
if (First == Last) return ⊥
res = Data[First]
First = (First + 1) % Capacity
return res
```

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

```
if (First == Last)
  return ⊥
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

Initialize all local SPSCs.

Initialize `slots` entries to `MAX`.

The enqueue operations are given as follows:

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

```
timestamp = FFA(counter)
value = (v, timestamp)
res = spsc_enqueue(spscs[rank], value)
if (!res) return false
if (!refreshEnqueue(rank, timestamp))
  refreshEnqueue(rank, timestamp)
return res
```

Procedure 5: `refreshEnqueue(rank: int, ts: timestamp_t)` **returns** `bool`

```
old-timestamp = slots[rank]
front = spsc_readFront(spscs[rank])
new-timestamp = front == ⊥ ? MAX : front.timestamp
if (new-timestamp != ts)
  return true
return CAS(&slots[rank], old-timestamp, new-timestamp)
```

The dequeue operations are given as follows:

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

```
rank = readMinimumRank()
if (rank == DUMMY || slots[rank] == MAX)
  return ⊥
res = spsc_dequeue(spscs[rank])
if (res == ⊥) return ⊥
if (!refreshDequeue(rank))
  refreshDequeue(rank)
return res
```

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

```
rank = length(slots)
min-timestamp = MAX
for index in 0..length(slots)
    timestamp = slots[index]
    if (min-timestamp < timestamp)
        rank = index
        min-timestamp = timestamp
old-rank = rank
for index in 0..old-rank
    timestamp = slots[index]
    if (min-timestamp < timestamp)
        rank = index
        min-timestamp = timestamp
return rank == length(slots) ? DUMMY : rank
```

Procedure 8: `refreshDequeue(rank: int)` **returns** `bool`

```
old-timestamp = slots[rank]
front = spsc_readFront(spscs[rank])
new-timestamp = front == ⊥ ? MAX : front.timestamp
if (front != ⊥)
    slots[rank] = new-timestamp
    return true
return CAS(&slots[rank], old-timestamp, new-timestamp)
```

4. ABA problem

Noticeably, we use no scheme to avoid ABA problem in Slot-queue. In actuality, ABA problem cannot happen in our algorithm, except in the extreme case that the 64-bit global counter overflows, which is unlikely.

5. Linearizability

6. Wait-freedom

7. 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.