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.

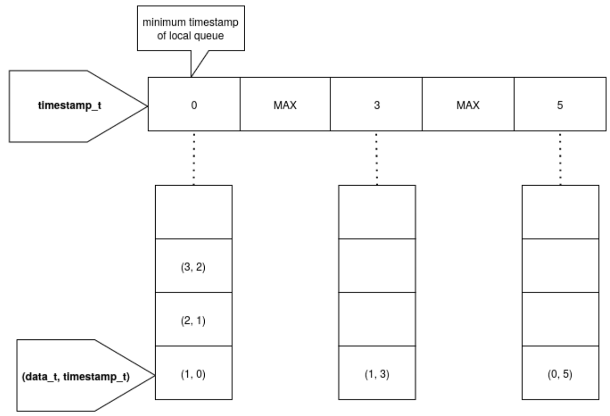

Figure 1: Basic structure of slot queue

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 = FAA(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. Linearizability of the local SPSC

In this section, we prove that the local SPSC is linearizable.

Lemma 4.1 *(Linearizability of `spsc_enqueue`)* The linearization point of `spsc_enqueue` is right after line 2 or right after line 4.

Lemma 4.2 *(Linearizability of `spsc_dequeue`)* The linearization point of `spsc_dequeue` is right after line 6 or right after line 8.

Lemma 4.3 *(Linearizability of `spsc_readFront`)* The linearization point `spsc_readFront` is right after line 11 or right after line 12.

Theorem 4.4 *(Linearizability of local SPSC)* The local SPSC is linearizable.

Proof This directly follows from Lemma 4.1, Lemma 4.2, Lemma 4.3. □

5. ABA problem

Noticeably, we use no scheme to avoid ABA problem in Slot-queue. In actuality, ABA problem does not adversely affect our algorithm's correctness, except in the extreme case that the 64-bit global counter overflows, which is unlikely.

5.1. ABA-safety

Not every ABA problem is unsafe. We formalize in this section which ABA problem is safe and which is not.

Definition 5.1.1 A **CAS-sequence** on a variable `v` is a sequence of instructions that:

- Starts with a load v_0 = `load(`v`)`.
- Ends with a `CAS(&`v`,`v_0`,`v_1`)`.

Definition 5.1.2 A **successful CAS-sequence** on a variable `v` is a **CAS-sequence** on `v` that ends with a successful CAS.

Definition 5.1.3 A **modification instruction** on a variable `v` is an atomic instruction that may change the value of `v` e.g. a store or a CAS.

Definition 5.1.4 A **successful modification instruction** on a variable `v` is an atomic instruction that changes the value of `v` e.g. a store or a successful CAS.

Definition 5.1.5 A **history** of successful **CAS-sequences** and **modification instructions** is a timeline of when any **CAS-sequences** start/end and when any modification instructions end.

We can define a strict partial order $<$ on the set of **CAS-sequences** and **modification instructions** such that:

- $A < B$ if A and B are both **CAS-sequences** and A ends before B starts.
- $A < B$ if A and B are **modifcation instructions** and A ends before B ends.
- $A < B$ if A is a **modification instruction**, B is a **CAS-sequence** and A ends before B starts.
- $B < A$ if A is a **modification instruction**, B is a **CAS-sequence** and A ends after B ends.

Definition 5.1.6 Consider a history of successful **CAS-sequences** and **modification instructions** on the same variable `v`. **ABA problem** is said to have occurred with `v` if there exists a **successful CAS-sequence** on `v`, during which there's some **successful modification instruction** on `v`.

Definition 5.1.7 Consider a history of successful **CAS-sequences** and **modification instructions** on the same variable `v`. A history is said to be **ABA-safe** with `v` if and only if:

- **ABA problem** does not occur with `v` in the history.
- We can reorder the **successful CAS-sequences** and **modification instructions** in the history such that:
  - No two successful CAS-sequences overlap with each other.
  - No modification instruction lies within another successful CAS-sequence.
  - The resulting history after reordering produces the same output as the original history.

5.2. Proof of ABA-safety

Notice that we only use `CAS` on:

- Line 13 of `refreshEnqueue` (Procedure 5), or an `enqueue` in general (Procedure 4).
- Line 42 of `refreshDequeue` (Procedure 8) or a `dequeue` in general (Procedure 6).

Both `CAS` target some slot in the `slots` array.

We apply some domain knowledge of our algorithm to the above formalism.

Definition 5.2.1 A **CAS-sequence** on a slot `s` of an `enqueue` that corresponds to `s` is the sequence of instructions from line 8 to line 13 of its `refreshEnqueue`.

Definition 5.2.2 A **slot-modification instruction** on a slot `s` of an `enqueue` that corresponds to `s` is line 13 of `refreshEnqueue`.

Definition 5.2.3 A **CAS-sequence** on a slot `s` of a `dequeue` that corresponds to `s` is the sequence of instructions from line 36 to line 42 of its `refreshDequeue`.

Definition 5.2.4 A **slot-modification instruction** on a slot `s` of a `dequeue` that corresponds to `s` is line 40 or line 42 of `refreshDequeue`.

Definition 5.2.5 A **CAS-sequence** of a `dequeue`/`enqueue` is said to **observes a slot value of s_0** if it loads s_0 at line 8 of `refreshEnqueue` or line 36 of `refreshDequeue`.

We can now turn to our interested problem in this section.

Lemma 5.2.1 *(Concurrent accesses on a local SPSC and a slot)* Only one dequeuer and one enqueuer can concurrently modify a local SPSC and a slot in the `slots` array.

Proof This is trivial to prove based on the algorithm's definition. □

Lemma 5.2.2 *(Monotonicity of local SPSC timestamps)* Each local SPSC in Slot-queue contains elements with increasing timestamps.

Proof Each `enqueue` would `FAA` the global counter (line 1 in Procedure 4) and enqueue into the local SPSC an item with the timestamp obtained from the counter. Applying Lemma 5.2.1, we know that items are enqueued one at a time into the SPSC. Therefore, later items are enqueued by later `enqueues`, which obtain increasing values by `FAA`-ing the shared counter. The theorem holds. □

Lemma 5.2.3 A `refreshEnqueue` (Procedure 5) can only changes a slot to a value other than `MAX`.

Proof For `refreshEnqueue` to change the slot's value, the condition on line 11 must be false. Then `new-timestamp` must equal to `ts`, which is not `MAX`. It's obvious that the `CAS` on line 13 changes the slot to a value other than `MAX`. □

Theorem 5.2.4 *(ABA safety of `dequeue`)* Assume that the 64-bit global counter never overflows, `dequeue` (Procedure 6) is ABA-safe.

Proof Consider a **successful CAS-sequence** on slot `s` by a `dequeue` d.

Denote t_d as the value this CAS-sequence observes.

Due to Lemma 5.2.1, there can only be at most one `enqueue` at one point in time within d.

If there's no **successful slot-modification instruction** on slot `s` by an `enqueue` e within d's **successful CAS-sequence**, then this `dequeue` is ABA-safe.

Suppose the `enqueue` e executes the *last* **successful slot-modification instruction** on slot `s` within d's **successful CAS-sequence**. Denote t_e to be the value that e sets `s`.

If $t_e \neq t_d$, this CAS-sequence of d cannot be successful, which is a contradiction.

Therefore, $t_e = t_d$.

Note that e can only set `s` to the timestamp of the item it enqueues. That means, e must have enqueued a value with timestamp t_d. However, by definition, t_d is read before e executes the CAS. This means another process (dequeuer/enqueuer) has seen the value e enqueued and CAS `s` for e before t_d. By Lemma 5.2.1, this "another process" must be another dequeuer d' that precedes d.

Because d' and d cannot overlap, while e overlaps with both d' and d, e must be the *first* `enqueue` on `s` that overlaps with d. Combining with Lemma 5.2.1 and the fact that e executes the *last* **successful slot-modification instruction** on slot `s` within d's **successful CAS-sequence**, e must be the only `enqueue` that executes a **successful slot-modification instruction** within d's **successful CAS-sequence**.

During the start of d's successful CAS-sequence till the end of e, `spsc_readFront` on the local SPSC must return the same element, because:

- There's no other `dequeues` running during this time.
- There's no `enqueue` other than e running.
- The `spsc_enqueue` of e must have completed before the start of d's successful CAS sequence, because a previous dequeuer d' can see its effect.

Therefore, if we were to move the starting time of d's successful CAS-sequence right after e has ended, we still retain the output of the program because:

- The CAS sequence only reads two shared values: `slots[rank]` and `spsc_readFront()`, but we have proven that these two values remain the same if we were to move the starting time of d's successful CAS-sequence this way.
- The CAS sequence does not modify any values except for the last CAS instruction, and the ending time of the CAS sequence is still the same.
- The CAS sequence modifies `slots[rank]` at the CAS but the target value is the same because inputs and shared values are the same in both cases.

We have proven that if we move d's successful CAS-sequence to start after the *last* **successful slot-modification instruction** on slot `s` within d's **successful CAS-sequence**, we still retain the program's output.

The theorem directly follows. □

Theorem 5.2.5 *(ABA safety of `enqueue`)* Assume that the 64-bit global counter never overflows, `enqueue` (Procedure 4) is ABA-safe.

Proof □

Theorem 5.2.6 *(ABA safety)* Assume that the 64-bit global counter never overflows, Slot-queue is ABA-safe.

Proof This follows from Theorem 5.2.5 and Theorem 5.2.4. □

6. Linearizability of Slot-queue

7. Wait-freedom

8. 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.