

# 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 enqueueers.

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 enqueueers. 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 enqueueer'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  $\perp$ 
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  $\perp$ 
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 enqueueers  
 spscs: An array of `spsc_t` with the number of entries equal the number of enqueueers  
 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`

---

```

1 timestamp = FAA(counter)
2 value = (v, timestamp)
3 res = spsc_enqueue(spsc_s[rank], value)
4 if (!res) return false
5 if (!refreshEnqueue(rank, timestamp))
6   | refreshEnqueue(rank, timestamp)
7 return res

```

---



---

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

---

```

8 old-timestamp = slots[rank]
9 front = spsc_readFront(spsc_s[rank])
10 new-timestamp = front ==  $\perp$  ? MAX : front.timestamp
11 if (new-timestamp != ts)
12   | return true
13 return CAS(&slots[rank], old-timestamp, new-timestamp)

```

---

The dequeue operations are given as follows:

---

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

---

```

14 rank = readMinimumRank()
15 if (rank == DUMMY || slots[rank] == MAX)
16   | return  $\perp$ 
17 res = spsc_dequeue(spsc_s[rank])
18 if (res ==  $\perp$ ) return  $\perp$ 
19 if (!refreshDequeue(rank))
20   | refreshDequeue(rank)
21 return res

```

---

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


---

```

22 rank = length(slots)
23 min-timestamp = MAX
24 for index in 0..length(slots)
25   timestamp = slots[index]
26   if (min-timestamp < timestamp)
27     rank = index
28     min-timestamp = timestamp
29 old-rank = rank
30 for index in 0..old-rank
31   timestamp = slots[index]
32   if (min-timestamp < timestamp)
33     rank = index
34     min-timestamp = timestamp
35 return rank == length(slots) ? DUMMY :
   rank

```

---

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


---

```

36 old-timestamp = slots[rank]
37 front = spsc_readFront(spsc[rank])
38 new-timestamp = front ==  $\perp$  ? MAX :
   front.timestamp
39 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  $\text{spsc\_enqueue}$* ) The linearization point of  $\text{spsc\_enqueue}$  is right after line 2 or right after line 4.

**Lemma 4.2** (*Linearizability of  $\text{spsc\_dequeue}$* ) The linearization point of  $\text{spsc\_dequeue}$  is right after line 6 or right after line 8.

**Lemma 4.3** (*Linearizability of  $\text{spsc\_readFront}$* ) The linearization point  $\text{spsc\_readFront}$  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.  $\square$

## 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 **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 5.1.2** 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 5.1.3** A **CAS-sequence** on a variable  $v$  is a sequence of instructions that:

- The first instruction is a load  $v_0 = \text{load}(v)$ .
- The last instruction is a CAS( $\&v, v_0, v_1$ ).
- There's no modification instruction on  $v$  between the first and the last instruction.

**Definition 5.1.4** A **successful CAS-sequence** on a variable  $v$  is a **CAS-sequence** on  $v$  that ends with a successful CAS.

**Definition 5.1.5** A **history** of successful **CAS-sequences** is a timeline of when any of the **CAS-sequences** starts and ends.

We can define a strict total order  $<$  on the set of **CAS-sequences** such that  $A < B$  if  $A$ 's **modification instruction** ends before  $B$ 's.

**Definition 5.1.6** Consider an operation  $op$ . Consider a history of successful **CAS-sequences** on the same variable  $v$ . **ABA problem** is said to occur with  $v$  in  $op$  if there exists one  $op$ 's **successful CAS-sequence** on  $v$ , during which there's some **successful modification instruction** on  $v$ .

**Definition 5.1.7** Consider an operation  $op$ . Consider a history of successful **CAS-sequences** on the same variable  $v$ .  $op$  is said to be **ABA-safe** with respect to  $v$  if and only if either:

- **ABA problem** does not occur with  $v$  in  $op$  in the history.
- We can reorder  $op$ 's **successful CAS-sequences** in the history such that:
  - The strict total order  $<$  on the history must be preserved.
  - The resulting history after reordering produces the same output as the original history.
  - **ABA problem** does not occur with  $v$  in  $op$  in the resulting 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 enqueue 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.  $\square$

**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.  $\square$

**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.  $\square$

**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 dequeuer  $d$ .

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 dequeuer 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/enqueueer) 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 it overlaps with  $e$ .

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** on  $s$  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. Furthermore, we only move the starting time of a CAS-sequence, therefore, the strict total order defined above is not changed.

If we apply the reordering for every dequeuer, the theorem directly follows.  $\square$

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

**Proof** Consider a **successful CAS-sequence** on slot  $s$  by an enqueue  $e$ .

Denote  $t_e$  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  $e$ .

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

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

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

Therefore,  $t_d = t_e$ .

If  $t_d = t_e = \text{MAX}$ , this means  $e$  observes a value of MAX before  $d$  even sets  $s$  to MAX. If this MAX value is the initialized value of  $s$ , it's a contradiction, as  $s$  must be non-MAX at some point for a dequeue such as  $d$  to run. If this MAX value is set by an enqueue, it's also a contradiction, as refreshEnqueue cannot set a slot to MAX. Therefore, this MAX value is set by a dequeue  $d'$ . If  $d' \neq d$  then it's a contradiction, because between  $d'$  and  $d$ ,  $s$  must be set to be a non-MAX value before  $d$  can be run. Therefore,  $d' = d$ . But, this means  $e$  observes a value set by  $d$ , which violates our assumption.

Therefore  $t_d = t_e = t' \neq \text{MAX}$ .  $e$  cannot observe the value  $t'$  set by  $d$  due to our assumption. Suppose  $e$  observes the value  $t'$  from  $s$  set by another enqueue/dequeue call other than  $d$ .

If this "another call" is a dequeue  $d'$  other than  $d$ ,  $d'$  precedes  $d$ . By Lemma 5.2.2, after each dequeue, the front element's timestamp will be increasing, therefore,  $d'$  must have set  $s$  to a timestamp smaller than  $t_d$ . However,  $e$  observes  $t_e = t_d$ . This is a contradiction.

Therefore, this "another call" is an enqueue  $e'$  other than  $e$  and  $e'$  precedes  $e$ . We know that an enqueue only sets  $s$  to the timestamp it obtains.

Suppose  $e'$  does not overlap with  $d$ .  $e'$  can only set  $s$  to  $t'$  if  $e'$  sees that the local SPSC has the front element as the element it enqueues. Due to Lemma 5.2.1, this means  $e'$  must observe a local SPSC with only the element it enqueues. Then,

when  $d$  executes readFront, the item  $e'$  enqueues must have been dequeued out already, thus,  $d$  cannot set  $s$  to  $t'$ . This is a contradiction.

Therefore,  $e'$  overlaps with  $d$ .

For  $e'$  to set  $s$  to the same value as  $d$ ,  $e'$ 's spsc\_readFront must serialize after  $d$ 's spsc\_dequeue.

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

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

- There's no other enqueues running during this time.
- There's no dequeue other than  $d$  running.
- The spsc\_dequeue of  $d$  must have completed before the start of  $e$ 's successful CAS sequence, because a previous enqueue  $e'$  can see its effect.

Therefore, if we were to move the starting time of  $e$ 's successful CAS-sequence right after  $d$  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  $e$ 's successful CAS-sequence this way.
- The CAS sequence does not modify any values except for the last CAS/store 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  $e$ 's successful CAS-sequence to start after the *last successful slot-modification instruction* on slot  $s$  within  $e$ 's **successful CAS-sequence**, we still retain the program's output. Furthermore, we only move the starting time of a CAS-sequence, therefore, the strict total order defined above is not changed.

If we apply the reordering for every enqueue, the theorem directly follows.  $\square$

**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.  $\square$

## 6. Linearizability of Slot-queue

**Definition 6.1** For an enqueue or dequeue  $op$ ,  $rank(op)$  is the rank of the enqueuer whose local SPSC is affected by  $op$ .

**Definition 6.2** For an enqueuer whose rank is  $r$ , the value stored in its corresponding slot at time  $t$  is denoted as  $slot(r, t)$ .

**Definition 6.3** For an enqueuer with rank  $r$ , the minimum timestamp among the elements between First and Last in its local SPSC at time  $t$  is denoted as  $min-spsc-ts(r, t)$ .

**Definition 6.4** For an enqueue, **slot-refresh phase** refer to its execution of line 5-6 of Procedure 4.

**Definition 6.5** For a dequeue, **slot-refresh phase** refer to its execution of line 19-20 of Procedure 6.

**Definition 6.6** For a dequeue, **slot-scan phase** refer to its execution of line 24-34 of Procedure 7.

**Definition 6.7** An enqueue operation  $e$  is said to **match** a dequeue operation  $d$  if  $d$  returns a timestamp that  $e$  enqueues. Similarly,  $d$  is said to **match**  $e$ . In this case, both  $e$  and  $d$  are said to be **matched**.

**Definition 6.8** An enqueue operation  $e$  is said to be **unmatched** if no dequeue operation **matches** it.

**Definition 6.9** A dequeue operation  $d$  is said to be **unmatched** if no enqueue operation **matches** it, in other word,  $d$  returns  $\perp$ .

We prove some algorithm-specific results first, which will form the basis for the more fundamental results.

**Lemma 6.1** If an enqueue  $e$  begins its **slot-refresh phase** at time  $t_0$  and finishes at time  $t_1$ , there's always at least one successful refreshEnqueue or refreshDequeue on  $rank(e)$  starting and ending its **CAS-sequence** between  $t_0$  and  $t_1$ .

**Proof** If one of the two refreshEnqueues succeeds, then the lemma obviously holds.

Consider the case where both fail.

The first refreshEnqueue fails because there's another refreshDequeue executing its **slot-modification instruction** successfully after  $t_0$  but before the end of the first refreshEnqueue's **CAS-sequence**.

The second refreshEnqueue fails because there's another refreshDequeue executing its **slot-modification instruction** successfully after  $t_0$  but before the end of the second refreshEnqueue's **CAS-sequence**. This another refreshDequeue must start its **CAS-sequence** after the end of the first successful refreshDequeue, due to Lemma 5.2.1. In other words, this another refreshDequeue starts and successfully ends its **CAS-sequence** between  $t_0$  and  $t_1$ .

We have proved the theorem.  $\square$

**Lemma 6.2** If a dequeue  $d$  begins its **slot-refresh phase** at time  $t_0$  and finishes at time  $t_1$ , there's always at least one successful refreshEnqueue or refreshDequeue on  $rank(d)$  starting and ending its **CAS-sequence** between  $t_0$  and  $t_1$ .

**Proof** This is similar to the above lemma.  $\square$

**Lemma 6.3** Given a rank  $r$  and a dequeue  $d$  that begins its **slot-scan phase** at time  $t_0$  and finishes at time  $t_1$ . If  $d$  finds that  $\text{slot}(r, t') = s_0 \neq \text{MAX}$  for some time  $t'$  such that  $t_0 \leq t' \leq t_1$ , then  $\text{slot}(r, t) = s_0 \neq \text{MAX}$  for any  $t$  such that  $t' \leq t \leq t_1$ .

**Proof** Denote  $s_r$  as the slot of rank  $r$ .

$\text{slot}(r, t') = s_0 \neq \text{MAX}$  because some processes have executed a successful slot-modification instruction on  $s_r$  to set it to  $s_0$ .

Take  $\text{op}$  to be the enqueue/dequeue that executes the last successful slot-modification instruction on  $s_r$  before  $t'$ . By definition,  $\text{op}$  set  $s_r$  to  $s_0$ .

Any dequeue before  $d$  would have finished before  $t_0$ , and thus its **slot-fresh phase**. By Lemma 6.2, for each dequeue before  $d$ , there must be some successful refresh call whose `spsc_readFront` observes the state of the local SPSC after  $d$ 's `spsc_dequeue`. By definition,  $\text{op}$ 's refresh call ended after all of these successful refresh call. By Theorem 5.2.6, the effect is as if  $\text{op}$  starts after all of these successful refresh call. Therefore,  $\text{op}$  can be treated as if it has seen the local SPSC after any of the previous dequeues' `spsc_dequeue` calls. In other words,  $\text{op}$  has set  $s_r$  to the front element's timestamp after it has observed all previous `spsc_dequeue` before  $d$ . During  $t_0$  to  $t_1$ , there's no `spsc_dequeue`. Therefore, from after  $\text{op}$ 's successful refresh call until  $t_1$ , there is no new `spsc_dequeue` that can be observed. Any refresh calls after  $\text{op}$  until  $t_1$  can only observe new `spsc_enqueues`, but because  $\text{op}$  set  $s_r$  to a non-MAX value, their corresponding `refreshEnqueues` cannot affect  $s_r$ . Therefore, the lemma holds.  $\square$

**Lemma 6.4** Given a rank  $r$  and a dequeue  $d$  that begins its **slot-scan phase** at time  $t_0$  and finishes at time  $t_1$ . If  $d$  finds that  $\text{slot}(r, t') = \text{MAX}$  for some time  $t'$  such that  $t_0 \leq t' \leq t_1$ , then  $\text{slot}(r, t) \neq \text{MAX}$  for any  $t$  such that  $t_0 \leq t \leq t'$ .

**Proof** Because during  $d$ 's **slot-scan phase**, no other dequeue can run and enqueues can only set a slot to non-MAX, if  $d$  finds that  $\text{slot}(r, t') =$

MAX for some time  $t'$  such that  $t_0 \leq t' \leq t_1$ , then  $\text{slot}(r, t) \neq \text{MAX}$  for any  $t$  such that  $t_0 \leq t \leq t'$ .

The theorem holds.  $\square$

We now look at the more fundamental results.

**Lemma 6.5** If  $d$  matches  $e$ , then either  $e$  precedes or overlaps with  $d$ .

**Proof** If  $d$  precedes  $e$ , none of the local SPSCs can contain an item with the timestamp of  $e$ . Therefore,  $d$  cannot return an item with a timestamp of  $e$ . Thus  $d$  cannot match  $e$ .

Therefore,  $e$  either precedes or overlaps with  $d$ .  $\square$

**Theorem 6.6** If an enqueue  $e$  precedes another dequeue  $d$ , then either:

- $d$  isn't matched.
- $d$  matches  $e$ .
- $e$  matches  $d'$  and  $d'$  precedes  $d$ .
- $d$  matches  $e'$  and  $e'$  precedes  $e$ .
- $d$  matches  $e'$  and  $e'$  overlaps with  $e$ .

**Proof**

$\square$

**Lemma 6.7** If  $d$  matches  $e$ , then either  $e$  precedes or overlaps with  $d$ .

**Proof**

$\square$

**Theorem 6.8** If a dequeue  $d$  precedes another enqueue  $e$ , then either:

- $d$  isn't matched.
- $d$  matches  $e'$  such that  $e'$  precedes or overlaps with  $e$  and  $e' \neq e$ .

**Proof**

$\square$

**Theorem 6.9** If an enqueue  $e_0$  precedes another enqueue  $e_1$ , then either:

- Both  $e_0$  and  $e_1$  aren't matched.
- $e_0$  is matched but  $e_1$  is not matched.
- $e_0$  matches  $d_0$  and  $e_1$  matches  $d_1$  such that  $d_0$  precedes  $d_1$ .

**Proof**

$\square$

**Theorem 6.10** If a dequeue  $d_0$  precedes another dequeue  $d_1$ , then either:



- $d_0$  isn't matched.
- $d_1$  isn't matched.
- $d_0$  matches  $e_0$  and  $d_1$  matches  $e_1$  such that  $e_0$  precedes or overlaps with  $e_1$ .

**Proof**

□

**Theorem 6.11** (*Linearizability of Slot-queue*) Slot-queue is linearizable.

## 7. Wait-freedom

The algorithm is trivially wait-free as there is no possibility of infinite loops.

## 8. Memory-safety

The algorithm is memory-safe: No memory deallocation happens and accesses are only made on allocated memory.

## 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.