Modified LTQueue without Load-Link/Store-Conditional

1. Original LTQueue

The original algorithm is given in [1] by Prasad Jayanti and Srdjan Petrovic in 2005. LTQueue is a wait-free MPSC algorithm, with logarithmic-time complexity for both enqueue and dequeue operations. The algorithm achieves good scalability by distributing the "global" queue over n queues that are local to each enqueuer. This helps avoid contention on the global queue among the enqueuers and allows multiple enqueuers to succeed at the same time. Furthermore, each enqueue and dequeue is efficient: they are wait-free and guaranteed to complete in $\theta(\log n)$ steps where n is the number of enqueuers. This is possible due to the novel tree structure proposed by the authors.

1.1. Local queue algorithm

Each local queue in LTQueue is an SPSC that allows an enqueuer and a dequeuer to concurrently access. Every enqueuer has one such local SPSC. Only the owner enqueuer and the dequeuer can access this local SPSC.

This section presents the SPSC data structure proposed in [1]. Beside the usual `enqueue` and `dequeue` procedures, this SPSC also supports the `readFront` procedure, which allows the enqueuer and dequeuer to retrieve the first item in the SPSC. Notice that enqueuer and dequeuer each has its own `readFront` method.

Types

> `data_t` = The type of data stored
>
> `node_t` = The type of the linked-list's nodes
>
> > **record**
> >
> > > `val`: `data_t`
> > >
> > > `next`: **pointer to** `node_t`
> >
> > **end**

Shared variables

> `First`: **pointer to** `node_t`
>
> `Last`: **pointer to** `node_t`
>
> `Announce`: **pointer to** `node_t`
>
> `FreeLater`: **pointer to** `node_t`
>
> `Help`: `data_t`

Initialization

```
First = Last = new Node()
FreeLater = new Node()
```

The SPSC always has a dummy node at the end.

The enqueuer's procedures are given as follows.

Procedure 1: `spsc_enqueue(v: data_t)`

```
1 newNode = new Node()
2 tmp = Last
3 tmp.val = v
4 tmp.next = newNode
5 Last = newNode
```

Procedure 2: `spsc_readFront`$_e$`()` **returns** `data_t`

```
6  tmp = First
7  if (tmp == Last) return ⊥
8  Announce = tmp
9  if (tmp != First)
10  | retval = Help
11 else retval = tmp.val
12 return retval
```

The dequeuer's procedures are given as follows.

Procedure 3: `spsc_dequeue()` **returns** `data_t`

```
13 tmp = First
14 if (tmp == Last) return ⊥
15 retval = tmp.val
16 Help = retval
17 First = tmp.next
18 if (tmp == Announce)
19 |  tmp' = FreeLater
20 |  FreeLater = tmp
21 |  free(tmp')
22 else free(tmp)
23 return retval
```

Procedure 4: `spsc_readFront`$_d$`()` **returns** `data_t`

```
24 tmp = First
25 if (tmp == Last)
26 |  return ⊥
27 return tmp.val
```

The treatment of linearizability, wait-freedom and memory safety is given in the original paper [1]. However, we can establish some intuition by looking at the procedures.

For wait-freedom, each procedure doesn't loop and doesn't wait for any other procedures to be able to complete on its own, therefore, they are all wait-free.

For memory safety, note that `spsc_enqueue` never accesses freed memory because the node pointed to by `Last` is never freed inside `spsc_dequeue`. `spsc_readFront`$_e$ tries to read `First` (line 6) and announces it not to be deleted to the dequeuer (line 8), it then checks if the pointer it read is still `First` (line 9), if it is, then it's safe to dereference the pointer (line 11) because the dequeuer would take note not to free it (line 18), otherwise, it returns `Help` (line 10), which is safely placed by the dequeuer (line 16). `spsc_dequeue` safely reclaims memory and does not leak memory (line 18-22). `spsc_readFront`$_d$ is safe because there's only one dequeuer running at a time.

For linearizability, the linearization point of `spsc_enqueue` is after line 5, `spsc_readFront`$_e$ is right after line 8, `spsc_dequeue` is right after line 17 and `spsc_readFront`$_d$ is right after line 25. Additionally, all the operations take constant time no matter the size of the queue.

1.2. LTQueue algorithm

LTQueue's idea is to maintain a tree structure as in Image 1. Each enqueuer is represented by the local SPSC node at the bottom of the tree. Every SPSC node in the local queue contains data and a timestamp indicating when it's enqueued into the SPSC. For consistency in node structure, we consider the leaf nodes of the tree to be the ones that are attached to the local SPSC of each enqueuer. Every internal node contains the minimum timestamp among its children's timestamps.

Types

> `data_t` = The type of the data to be stored in LTQueue
>
> `spsc_t` = The type of the local SPSC
>
> `tree_t` = The type of the tree constructed by LTQueue
>
> `node_t` = The node type of `tree_t`, containing a 64-bit timestamp value, packing a monotonic counter and the enqueuer's rank.

Shared variables

> `counter`: integer (`counter` supports LL and SC operations)
>
> `Q`: **array** `[1..n]` **of** `spsc_t`
>
> `T`: `tree_t`

Initialization

```
counter = 0
```

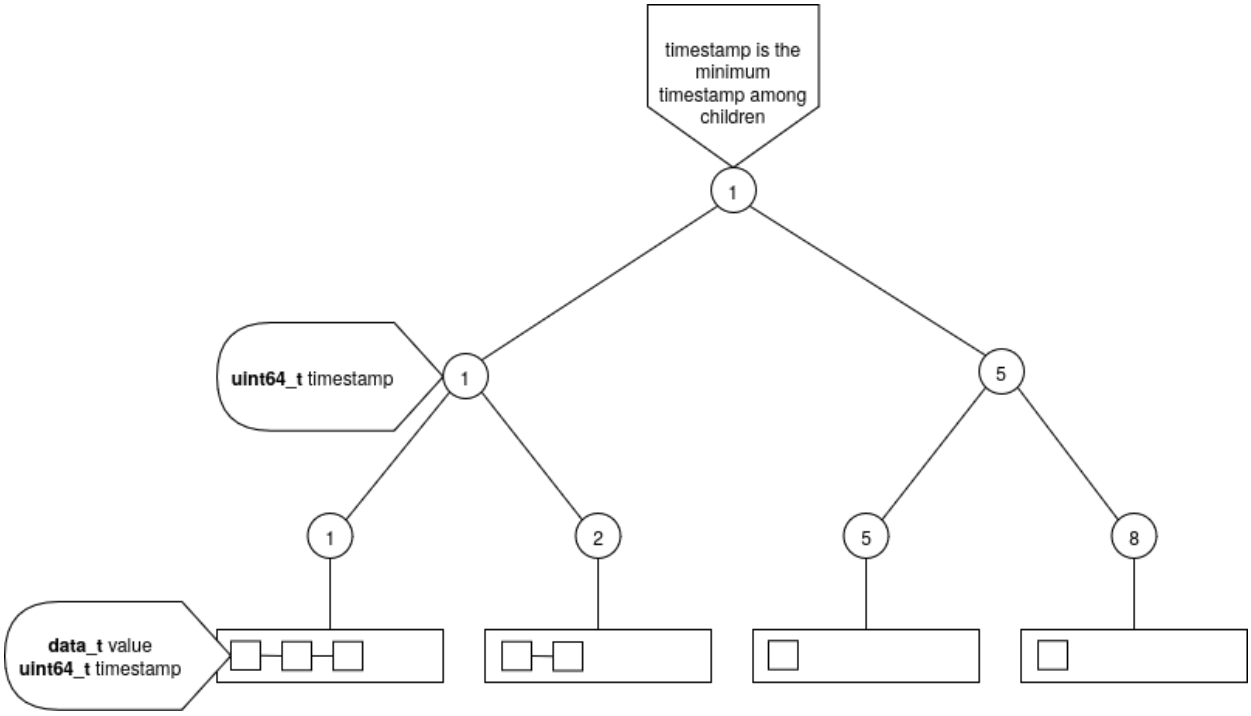

Image 1: LTQueue's structure

Procedure 5: `enqueue(rank: int, value: data_t)`

```
1 count = LL(counter)
2 SC(counter, count + 1)
3 timestamp = (count, rank)
4 spsc_enqueue(Q[rank], (value, timestamp))
5 propagate(Q[rank])
```

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

```
6 [count, rank] = read(root(T))
7 if (rank == ⊥) return ⊥
8 ret = spsc_dequeue(Q[rank])
9 propagate(Q[rank])
10 return ret.val
```

The followings are the timestamp propagation procedures.

Procedure 7: `propagate(spsc: spsc_t)`

```
11 if ¬refreshLeaf(spsc)
12 | refreshLeaf(spsc)
13 currentNode = leafNode(spsc)
14 repeat
15 | currentNode = parent(currentNode)
16 | if ¬refresh(currentNode)
17 | | refresh(currentNode)
18 until currentNode == root(T)
```

Procedure 8: `refresh(currentNode:` **pointer** to `node_t)`

```
19 LL(currentNode)
20 for childNode in children(currentNode)
21 | let minT be the minimum timestamp for every childNode
22 SC(currentNode, minT)
```

Procedure 9: `refreshLeaf(spsc: spsc_t)`

```
23 leafNode = leafNode(spsc)
24 LL(leafNode)
25 SC(leafNode, spsc_readFront(spsc))
```

Note that compare to the original paper [1], we have make some trivial modification on line 11-12 to handle the leaf node case, which was left unspecified in the original algorithm. In many ways, this modification is in the same light with the mechanism the algorithm is already using, so intuitively, it should not affect the algorithm's correctness or wait-freedom. Note that on line 25 of `refreshLeaf`, we omit which version of `spsc_readFront` it's calling, simply assume that the dequeuer and the enqueuer should call their corresponding version of `spsc_readFront`.

Similarly, the proofs of LTQueue's linearizability, wait-freedom, memory-safety and logarithmic-time complexity of enqueue and dequeue operations are given in [1]. One notable technique that allows LTQueue to be both correct and wait-free is the double-refresh trick during the propagation process on line 16-17.

The idea behind the `propagate` procedure is simple: Each time an SPSC queue is modified (inserted/deleted), the timestamp of a leaf has changed so the timestamps of all nodes on the path from that leaf to the root can potentially change. Therefore, we have to propagate the change towards the root, starting from the leaf (line 11-18).

The `refresh` procedure is by itself simple: we access all child nodes to determine the minimum timestamp in each child's subtree and try to set the current node's timestamp with the minimum timestamp using a pair of LL/SC. However, LL/SC can not always succeed so the current node's timestamp may not be updated by `refresh` at all. The key to fix this is to retry `refresh` on line 17 in case of the first `refresh`'s failure. Later, when we prove the correctness of the modified LTQueue, we provide a formal proof of why this works. Here, for intuition, we visualize in Image 2 the case where both `refresh` fails but correctness is still ensures.

2. Adaption of LTQueue without load-link/store-conditional

The SPSC data structure in the original LTQueue is kept in tact so one may refer to Procedure 1, Procedure 2, Procedure 3, Procedure 4 for the supported SPSC procedures.

The followings are the rewritten LTQueue's algorithm without LL/SC.

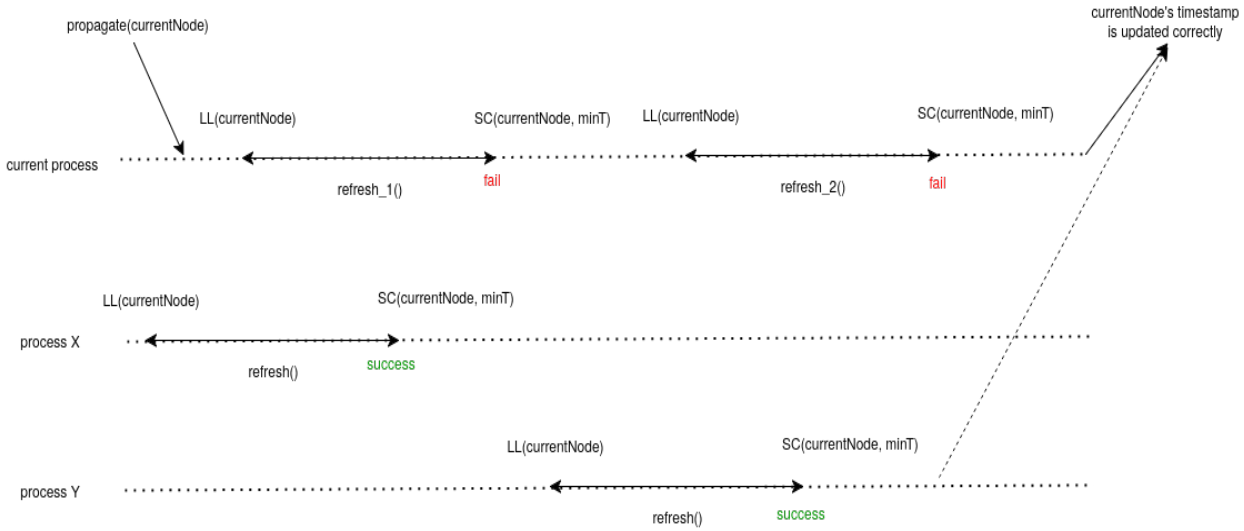

Image 2: Even though two `refresh`s fails, the `currentNode`'s timestamp is still updated correctly

The structure of LTQueue is modified as in Image 3. At the bottom **enqueuer nodes** (represented by the type `enqueuer_t`), besides the local SPSC, the minimum-timestamp among the elements in the SPSC is also stored. The **internal nodes** no longer store a timestamp but a rank of an enqueuer. This rank corresponds to the enqueuer with the minimum timestamp among the node's children's ranks. Note that if a local SPSC is empty, the minimum-timestamp of the corresponding enqueuer node is set to `MAX` and the corresponding leaf node's rank is set to a `DUMMY` rank.

Types

> `data_t` = The type of the data to be stored in LTQueue
>
> `spsc_t` = The type of the local SPSC
>
> `rank_t` = The rank of an enqueuer
>
> > **struct**
> >
> > > `value`: `uint32_t`
> > >
> > > `version`: `uint32_t`
> >
> > **end**
>
> `timestamp_t` =
>
> > **struct**
> >
> > > `value`: `uint32_t`
> > >
> > > `version`: `uint32_t`
> >
> > **end**
>
> `enqueuer_t` =
>
> > **struct**
> >
> > > `spsc`: `spsc_t`
> > >
> > > `min-timestamp`: `timestamp_t`
> >
> > **end**
>
> `node_t` = The node type of the tree constructed by LTQueue
>
> > **struct**
> >
> > > `rank`: `rank_t`
> >
> > **end**

Shared variables

> `counter`: `uint64_t`
>
> `root`: **pointer to** `node_t`
>
> `enqueuers`: **array** `[1..n]` **of** `enqueuer_t`

Initialization

> `counter = 0`
>
> construct the tree structure and set `root` to the root node

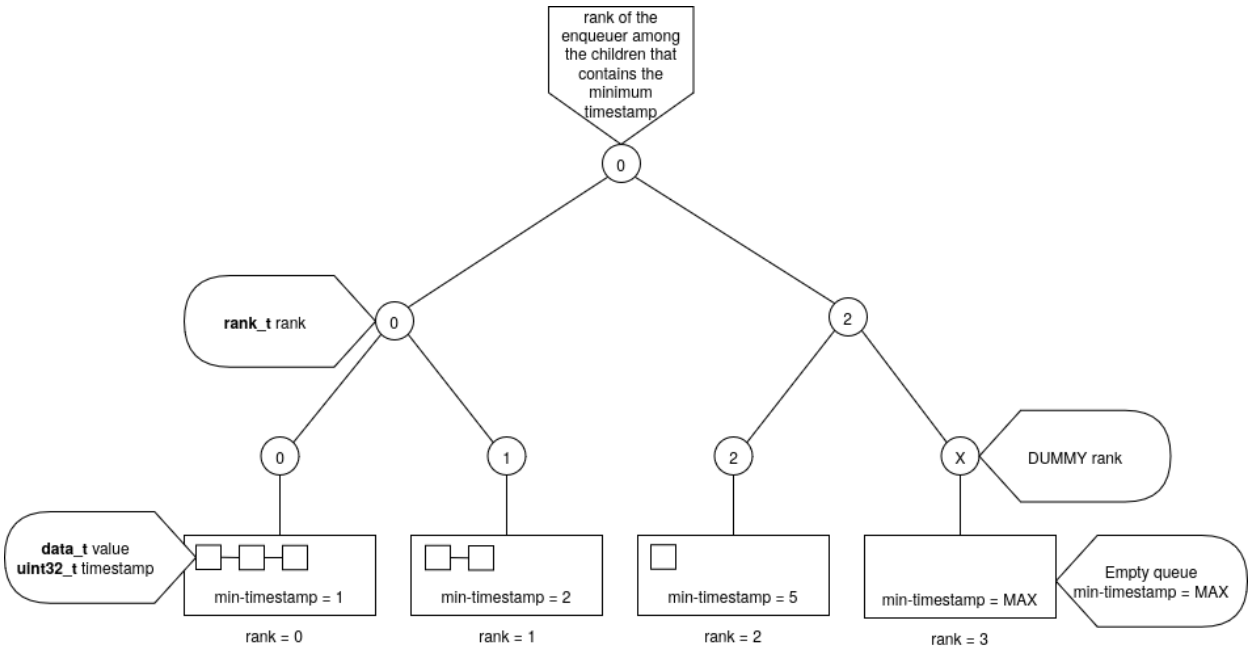

Image 3: Modified LTQueue's structure

initialize every node in the tree to contain `DUMMY` rank and version `0`

initialize every enqueuer's `timestamp` to `MAX` and version `0`

Procedure 10: `enqueue(rank: int, value: data_t)`

```
1 count = FAA(counter)
2 timestamp = (count, rank)
3 spsc_enqueue(enqueuers[rank].spsc, (value, timestamp))
4 propagate(rank)
```

Procedure 11: `dequeue()` **returns** `data_t`

```
5 [rank, version] = root->rank
6 if (rank == DUMMY) return ⊥
7 ret = spsc_dequeue(enqueuers[rank].spsc)
8 propagate(rank)
9 return ret.val
```

We omit the description of procedures `parent`, `leafNode`, `children`, leaving how the tree is constructed and children-parent relationship is determined to the implementor. The tree structure used by LTQueue is read-only so a wait-free implementation of these procedures is trivial.

Procedure 12: `propagate(rank: uint32_t)`

```
10 if ¬refreshTimestamp(rank)
11 | refreshTimestamp(rank)
12 if ¬refreshLeaf(rank)
13 | refreshLeaf(rank)
14 currentNode = leafNode(rank)
15 repeat
16 | currentNode = parent(currentNode)
17 | if ¬refresh(currentNode)
18 | | refresh(currentNode)
19 until currentNode == root
```

Procedure 13: `refresh(currentNode: pointer to node_t)`

```
20 [old-rank, old-version] = currentNode->rank
21 min-rank = DUMMY
22 min-timestamp = MAX
23 for childNode in children(currentNode)
24 | [child-rank, ...] = childNode->rank
25 | if (child-rank == DUMMY) continue
26 | child-timestamp = enqueuers[child-rank].min-timestamp
27 | if (child-timestamp < min-timestamp)
28 | | min-timestamp = child-timestamp
29 | | min-rank = child-rank
30 return CAS(&currentNode->rank, [old-rank, old-version], [min-rank, old-version + 1])
```

Procedure 14: `refreshTimestamp(rank: uint32_t)`

```
31 [old-timestamp, old-version] = enqueuers[rank].timestamp
32 front = spsc_readFront(enqueuers[rank].spsc)
33 if (front == ⊥)
34 | return CAS(&enqueuers[rank].timestamp, [old-timestamp, old-version], [MAX, old-version + 1])
35 else
36 | return CAS(&enqueuers[rank].timestamp, [old-timestamp, old-version], [front.timestamp, old-version + 1])
```

Procedure 15: `refreshLeaf(rank: uint32_t)`

```
37 leafNode = leafNode(spsc)
38 [old-rank, old-version] = leafNode->rank
39 [timestamp,                          ...]
   = enqueuers[rank].timestamp
   return CAS(&leafNode->rank, [old-rank,
40 old-version], [timestamp == MAX ? DUMMY :
   rank, old-version + 1])
```

Notice that we omit which version of `spsc_readFront` we're calling on line 32, simply assuming that the producer and each enqueuer are calling their respective version.

3. Proof of correctness

This section proves that the algorithm given in the last section is linearizable, memory-safe and wait-free.

3.1. Linearizability

Within the next two sections, we formalize what it means for an MPSC to be linearizable.

Definition of linearizability

The following discussion of linearizability is based on [2] by Herlihy and Shavit.

For a concurrent object `S`, we can call some methods on `S` concurrently. A method call on the object `S` is said to have an **invocation event** when it starts and a **response event** when it ends.

Definition 3.1.1 An **invocation event** is a triple $(S, t, args)$, where S is the object the method is invoked on, t is the timestamp of when the event happens and $args$ is the arguments passed to the method call.

Definition 3.1.2 A **response event** is a triple (S, t, res), where S is the object the method is invoked on, t is the timestamp of when the event happens and res is the results of the method call.

Definition 3.1.3 A **method call** is a tuple of (i, r) where i is an invocation event and r is a response event or the special value $\perp$ indicating that its response event hasn't happened yet. A well-formed **method call** should have a reponse event with a larger timestamp than its invocation event or the response event hasn't happened yet.

Definition 3.1.4 A **method call** is **pending** if its invocation event is $\perp$.

Definition 3.1.5 A **history** is a set of well-formed **method calls**.

Definition 3.1.6 An extension of **history** H is a **history** H' such that any pending method call is given a response event.

We can define a **strict partial order** on the set of well-formed method calls:

Definition 3.1.7 $\rightarrow$ is a relation on the set of well-formed method calls. With two method calls X and Y, we have $X \rightarrow Y \Leftrightarrow X$'s response event is not $\perp$ and its response timestamp is not greater than Y's invocation timestamp.

Definition 3.1.8 Given a **history** H, $\rightarrow_H$ is a relation on H such that for two method calls X and Y in H, $X \rightarrow_H Y \Leftrightarrow X \rightarrow Y$.

Definition 3.1.9 A **sequential history** H is a **history** such that $\rightarrow_H$ is a total order on H.

Now that we have formalized the way to describe the order of events via **histories**, we can now formalize the mechanism to determine if a **history** is valid. The easier case is for a **sequential history**:

Definition 3.1.10 For a concurrent object S, a **sequential specification** of S is a function that either returns `true` (valid) or `false` (invalid) for a **sequential history** H.

The harder case is handled via the notion of **linearizable**:

Definition 3.1.11 A history H on a concurrent object S is **linearizable** if it has an extension H' and there exists a *sequential history* H_S such that:

1. The **sequential specification** of S accepts H_S.
2. There exists a one-to-one mapping M of a method call $(i, r) \in H'$ to a method call $(i_S, r_S) \in H_S$ with the properties that:
   - i must be the same as i_S except for the timestamp.
   - r must be the same r_S except for the timestamp or r.
3. For any two method calls X and Y in H', $X \rightarrow_{H'} Y \Rightarrow M(X) \rightarrow_{H_S} M(Y)$.

We consider a history to be valid if it's linearizable.

Definition of linearizable MPSC

An MPSC supports 2 **methods**:
- `enqueue` which accepts a value and returns nothing
- `dequeue` which doesn't accept anything and returns a value

An MPSC has the same **sequential specification** as a FIFO:
- `dequeue` returns values in the same order as they was `enqueued`.
- An `enqueue` can only by `dequeued` once.
- An item can only be `dequeued` after it's `enqueued`.
- If the queue is empty, `dequeue` returns nothing.

An MPSC places a special constraint on **the set of histories** it can produce: Any history H must not have overlapping `dequeue` method calls.

Definition 3.1.12 An MPSC is **linearizable** if and only if any history produced from the MPSC that does not have overlapping `dequeue` method calls is *linearizable* according to the *FIFO sequential specification.*

Proof of linearizability

Definition 3.1.13 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 3.1.14 An `enqueue` operation e is said to be **unmatched** if no `dequeue` operation **matches** it.

Definition 3.1.15 A `dequeue` operation d is said to be **unmatched** if no `enqueue` operation **matches** it, in other word, d returns $\perp$.

Theorem 3.1.1 Only the dequeuer and one enqueuer can operate on an enqueuer node.

Proof This is trivial. □

We immediately obtain the following result.

Corollary 3.1.2 Only one `dequeue` operation and one `enqueue` operation can operate concurrently on an enqueuer node.

Proof This is trivial. □

Theorem 3.1.3 The SPSC at an enqueuer node contains items with increasing timestamps.

Proof Each `enqueue` would `FAA` the shared counter (line 1 in Procedure 10) and enqueue into the local SPSC an item with the timestamp obtained from the counter. Applying Corollary 3.1.2, 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. □

Definition 3.1.16 For a tree node n, the enqueuer rank stored in n at time t is denoted as $rank(n, t)$.

Definition 3.1.17 For an `enqueue` or a `dequeue` `op`, the rank of the enqueuer it affects is denoted as $rank(\texttt{op})$.

Definition 3.1.18 For an enqueuer whose rank is r, the `min-timestamp` value stored in its enqueuer node at time t is denoted as $min\text{-}ts(r, t)$. If r is `DUMMY`, $min\text{-}ts(r, t)$ is `MAX`.

Definition 3.1.19 For an enqueuer with rank r, the minimum timestamp among the elements between `First` and `Last` in the local SPSC at time t is denoted as $min\text{-}spsc\text{-}ts(r, t)$. If r is dummy, $min\text{-}spsc\text{-}ts(r, t)$ is `MAX`.

Definition 3.1.20 For an `enqueue` or a `dequeue` `op`, the set of nodes that it calls `refresh` or `refreshLeaf` on is denoted as $path(\texttt{op})$.

Definition 3.1.21 For an `enqueue` or a `dequeue`, **timestamp-refresh phase** refer to its execution of line 10-11 in `propagate` (Procedure 12).

Definition 3.1.22 For an `enqueue` or a `dequeue` `op`, and a node $n \in path(\texttt{op})$, **node-n-refresh phase** refer to its execution of line 12-13 (if n is a leaf node) or line 17-18 (if n is a non-leaf node) to refresh n's rank in `propagate` (Procedure 12).

Definition 3.1.23 `refreshTimestamp` is said to start its **CAS-sequence** if it finishes line 31 in Procedure 14. `refreshTimestamp` is said to end its **CAS-sequence** if it finishes line 34 or line 36 in Procedure 14.

Definition 3.1.24 `refresh` is said to start its **CAS-sequence** if it finishes line 20 in Procedure 13. `refresh` is said to end its **CAS-sequence** if it finishes line 30 in Procedure 13.

Definition 3.1.25 `refreshLeaf` is said to start its **CAS-sequence** if it finishes line 38 in Procedure 15. `refreshLeaf` is said to end its **CAS-sequence** if it finishes line 40 in Procedure 15.

Theorem 3.1.4 For an `enqueue` or a `dequeue` `op`, if `op` modifies an enqueuer node and this enqueuer node is attached to a leaf node l, then $path(\texttt{op})$ is the set of nodes lying on the path from l to the root node.

Proof This is trivial considering how `propagate` (Procedure 12) works. □

Theorem 3.1.5 For any time t and a node n, $rank(n, t)$ can only be `DUMMY` or the rank of one of the enqueuer nodes in the subtree rooted at n.

Proof This is trivial considering how `refresh` and `refreshLeaf` works. □

Theorem 3.1.6 If an `enqueue` or a `dequeue` `op` begins its **timestamp-refresh phase** at t_0 and finishes at time t_1, there's always at least one successful `refreshTimestamp` on $rank(\texttt{op})$ starting and ending its **CAS-sequence** between t_0 and t_1.

Proof If one of the two `refreshTimestamps` succeeds, then the theorem obviously holds.

Consider the case where both fail.

The first `refreshTimestamp` fails because there's another `refreshTimestamp` ending its **CAS-sequence** successfully after t_0 but before the end of the first `refreshTimestamp`'s **CAS-sequence**.

The second `refreshTimestamp` fails because there's another `refreshTimestamp` ending its **CAS-sequence** successfully after t_0 but before the end of the second `refreshTimestamp`'s **CAS-sequence**. This another `refreshTimestamp` must start its **CAS-sequence** after the end of the first successful `refreshTimestamp`, else it overlaps with the **CAS-sequence** of the first successful `refreshTimestamp` but successful **CAS-sequences** cannot overlap. In other words, this another `refreshTimestamp` starts and successfully ends its **CAS-sequence** between t_0 and t_1.

We have proved the theorem. □

Theorem 3.1.7 If an `enqueue` or a `dequeue` begins its **node-n-refresh phase** at t_0 and finishes at t_1, there's always at least one successful `refresh` or `refreshLeaf` on n starting and ending its **CAS-sequence** between t_0 and t_1.

Proof This is similar to the above proof. □

Theorem 3.1.8 For any node n, $min\text{-}ts\big(rank(n, t_x), t_y\big)$ is monotonically decreasing for $t_x, t_y \in [t_0, t_1]$ if within t_0 and t_1, any `dequeue` d where $n \in path(d)$ has finished its **node-n-refresh phase**.

Proof We have the assumption that within t_0 and t_1, all `dequeue` where $n \in path(d)$ has finished its **node-n-refresh phase**. Notice that if n satisfies this assumption, any child of n also satisfies this assumption.

We will prove a stronger version of this theorem: Given a node n, time t_0 and t_1 such that within

$[t_0, t_1]$, any `dequeue` d where $n \in path(d)$ has finished its **node-n-refresh phase**. Consider the last `dequeue`'s **node-n-refresh phase** before t_0 (there maybe none). Take $t_s(n)$ and $t_e(n)$ to be the starting and ending time of the CAS-sequence of the last successful **n-refresh call** during this phase, or if there is none, $t_s(n) = t_e(n) = 0$. Then, $min\text{-}ts\big(rank(n, t_x), t_y\big)$ is monotonically decreasing for $t_x, t_y \in [t_e(n), t_1]$.

Consider any enqueuer node of rank r that's attached to a satisfied leaf node. For any n' that is a descendant of n, during $t_s(n')$ and t_1, there's no call to `spsc_dequeue`. Because:

- If an `spsc_dequeue` starts between t_0 and t_1, the `dequeue` that calls it hasn't finished its **node-n'-refresh phase**.
- If an `spsc_dequeue` starts between $t_s(n')$ and t_0, then a `dequeue`'s **node-n'-refresh phase** must start after $t_s(n')$ and before t_0, but this violates our assumption of $t_s(n')$.

Therefore, there can only be calls to `spsc_enqueue` during $t_s(n')$ and t_1. Thus, $min\text{-}spsc\text{-}ts(r, t_x)$ can only decrease from `MAX` to some timestamp and remain constant for $t_x \in [t_s(n'), t_1]$. (1)

Similarly, there can be no `dequeue` that hasn't finished its **timestamp-refresh phase** during $t_s(n')$ and t_1. Therefore, $min\text{-}ts(r, t_x)$ can only decrease from `MAX` to some timestamp and remain constant for $t_x \in [t_s(n'), t_1]$. (2)

Consider any satisfied leaf node n_0. There can be no `dequeue` that hasn't finished its **node-n_0-refresh phase** during $t_e(n_0)$ and t_1. Therefore, any successful `refreshLeaf` during $[t_e(n_0), t_1]$ must be called by an `enqueue`. Because there's no `spsc-dequeue`, this `refreshLeaf` can only set $rank(n_0, t_x)$ from `DUMMY` to r and remains r until t_1, which is the rank of the enqueuer node its attached to. Therefore, combining with (1), $min\text{-}ts\big(rank(n_0, t_x), t_y\big)$ is monotonically decreasing for $t_x, t_y \in [t_e(n_0), t_1]$. (3)

Consider any satisfied non-leaf node n' that is a descendant of n. Suppose during $[t_e(n'), t_1]$, we have a sequence of successful **n'-refresh calls** that start their CAS-sequences at $t_{start\text{-}0} < t_{start\text{-}1} < t_{start\text{-}2} < ... < t_{start\text{-}k}$ and end them at $t_{end\text{-}0} < t_{end\text{-}1} < t_{end\text{-}2} < ... < t_{end\text{-}k}$. By definition, $t_{end\text{-}0} = t_e(n')$ and $t_{start\text{-}0} = t_s(n')$. We can prove that $t_{end\text{-}i} < t_{start\text{-}(i+1)}$ because successful CAS-sequences cannot overlap.

Due to how `refresh` is defined, for any $k \geq i \geq 1$:

- Suppose $t_{rank\text{-}i}(c)$ is the time `refresh` reads the rank stored in the child node c, so $t_{start\text{-}i} \leq t_{rank\text{-}i}(c) \leq t_{end\text{-}i}$.
- Suppose $t_{ts\text{-}i}(c)$ is the time `refresh` reads the timestamp stored in the enqueuer with the rank read previously, so $t_{start\text{-}i} \leq t_{ts\text{-}i}(c) \leq t_{end\text{-}i}$.
- There exists a child c_i such that $rank(n', t_{end\text{-}i}) = rank(c_i, t_{rank\text{-}i}(c_i))$. (4)
- For every child c of n', $min\text{-}ts(rank(n', t_{end\text{-}i}), t_{ts\text{-}i}(c_i)) \leq min\text{-}ts(rank(c, t_{rank\text{-}i}(c)), t_{ts\text{-}i}(c))$. (5)

Suppose the stronger theorem already holds for every child c of n'. (6)

For any $i \geq 1$, we have $t_e(c) \leq t_s(n') \leq t_{start\text{-}(i-1)} \leq t_{rank\text{-}(i-1)}(c) \leq t_{end\text{-}(i-1)} \leq t_{start\text{-}i} \leq t_{rank\text{-}i}(c) \leq t_1$. Combining with (5), (6), we have for any $k \geq i \geq 1$,
$min\text{-}ts(rank(n', t_{end\text{-}i}), t_{ts\text{-}i}(c_i))$
$\leq min\text{-}ts(rank(c, t_{rank\text{-}i}(c)), t_{ts\text{-}i}(c))$
$\leq min\text{-}ts\Big(rank\Big(c, t_{rank\text{-}(i-1)}(c)\Big), t_{ts\text{-}i}(c)\Big)$.

Choose $c = c_{i-1}$ as in (4). We have for any $k \geq i \geq 1$,
$min\text{-}ts(rank(n', t_{end\text{-}i}), t_{ts\text{-}i}(c_i))$
$\leq min\text{-}ts(rank\Big(c_{i-1}, t_{rank\text{-}(i-1)}(c_{i-1})\Big),$
$t_{ts\text{-}i}(c_{i-1}))$
$= min\text{-}ts(rank\Big(n', t_{end\text{-}(i-1)}\Big), t_{ts\text{-}i}(c_{i-1})$.

Because $t_{ts\text{-}i}(c_i) \leq t_{end\text{-}i}$ and $t_{ts\text{-}i}(c_{i-1}) \geq t_{end\text{-}(i-1)}$ and (2), we have for any $k \geq i \geq 1$,
$min\text{-}ts(rank(n', t_{end\text{-}i}), t_{end\text{-}i})$
$\leq min\text{-}ts\Big(rank\Big(n', t_{end\text{-}(i-1)}\Big), t_{end\text{-}(i-1)}\Big)$. $(*)$

$rank(n', t_x)$ can only change after each successful `refresh`, therefore, the sequence of its value is $rank(n', t_{end\text{-}0})$, $rank(n', t_{end\text{-}1})$, ..., $rank(n', t_{end\text{-}k})$. $(**)$

Note that if `refresh` observes that an enqueuer has a `min-timestamp` of `MAX`, it would never try to CAS n''s rank to the rank of that enqueuer (line 22 and line 27 of Procedure 13). So, if `refresh` actually set the rank of n' to some non-`DUMMY` value, the corresponding enqueuer must actually has a non-`MAX` `min-timestamp` *at some point*. Due to (2), this is constant up until t_1. Therefore, $min\text{-}ts(rank(n', t_{end\text{-}i}), t))$ is constant for any $t \geq t_{end\text{-}i}$ and $k \geq i \geq 1$. $min\text{-}ts(rank(n', t_{end\text{-}0}), t))$ is constant for any $t \geq t_{end\text{-}0}$ if there's a `refresh` before t_0. If there's no `refresh` before t_0, it is constant `MAX`. So, $min\text{-}ts(rank(n', t_{end\text{-}i}), t))$ is constant for any $t \geq t_{end\text{-}i}$ and $k \geq i \geq 0$. $(***)$

Combining $(*)$, $(**)$, $(***)$, we obtain the stronger version of the theorem. □

Theorem 3.1.9 If an `enqueue` e obtains a timestamp c and finishes at time t_0 and is still **unmatched** at time t_1, then for any subrange T of $[t_0, t_1]$ that does not overlap with a `dequeue`, $min\text{-}ts(rank(root, t_r), t_s) \leq c$ for any $t_r, t_s \in T$.

Proof We will prove a stronger version of this theorem: Suppose an `enqueue` e obtains a timestamp c and finishes at time t_0 and is still **unmatched** at time t_1. For every $n_i \in path(e)$, n_0 is the leaf node and n_i is the parent of n_{i-1}, $i \geq 1$. If e starts and finishes its **node-n_i-refresh phase** at $t_{start\text{-}i}$ and $t_{end\text{-}i}$ then for any subrange T of $[t_{end\text{-}i}, t_1]$ that does not overlap with a `dequeue` d where $n_i \in path(d)$ and d hasn't finished its **node n_i refresh phase**, $min\text{-}ts(rank(n_i, t_r), t_s) \leq c$ for any $t_r, t_s \in T$.

If $t_1 < t_0$ then the theorem holds.

Take r_e to be the rank of the enqueuer that performs e.

Suppose e enqueues an item with the timestamp c into the local SPSC at time $t_{enqueue}$. Because it's still unmatched up until t_1, c is always in the local SPSC during $t_{enqueue}$ to t_1. Therefore, $min\text{-}spsc\text{-}ts(r_e, t) \leq c$ for any $t \in \left[t_{enqueue}, t_1\right]$. (1)

Suppose e finishes its **timestamp refresh phase** at $t_{r\text{-}ts}$. Because $t_{r\text{-}ts} \geq t_{enqueue}$, due to (1), $min\text{-}ts(r_e, t) \leq c$ for every $t \in [t_{r\text{-}ts}, t_1]$. (2)

Consider the leaf node $n_0 \in path(e)$. Due to (2), $rank(n_0, t)$ is always r_e for any $t \in [t_{end\text{-}0}, t_1]$. Also due to (2), $min\text{-}ts(rank(n_0, t_r), t_s) \leq c$ for any $t_r, t_s \in [t_{end\text{-}0}, t_1]$.

Consider any non-leaf node $n_i \in path(e)$. We can extend any subrange T to the left until we either:

- Reach a `dequeue` d such that $n_i \in path(d)$ and d has just finished its **node-n_i-refresh phase**.
- Reach $t_{end\text{-}i}$.

Consider one such subrange T_i.

Notice that T_i always starts right after a **node-n_i-refresh phase**. Due to Theorem 3.1.7, there's always at least one successful `refresh` in this **node-n_i-refresh phase**.

Suppose the stronger version of the theorem already holds for n_{i-1}. That is, if e starts and finishes its **node-n_{i-1}-refresh phase** at $t_{start\text{-}(i-1)}$ and $t_{end\text{-}(i-1)}$ then for any subrange T of $\left[t_{end\text{-}(i-1)}, t_1\right]$ that does not overlap with a `dequeue` d where $n_i \in path(d)$ and d hasn't finished its **node n_{i-1} refresh phase**, $min\text{-}ts(rank(n_i, t_r), t_s) \leq c$ for any $t_r, t_s \in T$.

Extend T_i to the left until we either:

- Reach a `dequeue` d such that $n_i \in path(d)$ and d has just finished its **node-n_{i-1}-refresh phase**.
- Reach $t_{end\text{-}(i-1)}$.

Take the resulting range to be T_{i-1}. Obviously, $T_i \subseteq T_{i-1}$.

T_{i-1} satisifies both criteria:

- It's a subrange of $\left[t_{end\text{-}(i-1)}, t_1\right]$.
- It does not overlap with a `dequeue` d where $n_i \in path(d)$ and d hasn't finished its **node-n_{i-1}-refresh phase**.

Therefore, $min\text{-}ts(rank(n_{i-1}, t_r), t_s) \leq c$ for any $t_r, t_s \in T_{i-1}$.

Consider the last successful `refresh` on n_i ending not after T_i, take $t_{s'}$ and $t_{e'}$ to be the start and

end time of this `refresh`'s CAS-sequence. Because right at the start of T_i, a **node-n_i-refresh phase** just ends, this `refresh` must be within this **node-n_i-refresh phase**. (4)

This `refresh`'s CAS-sequence must be within T_{i-1}. This is because right at the start of T_{i-1}, a **node-n_{i-1}-refresh phase** just ends and $T_{i-1} \supseteq T_i$, T_{i-1} must cover the **node-n_i-refresh phase** whose end T_i starts from. Combining with (4), $t_{s'} \in T_{i-1}$ and $t_{e'} \in T_i$. (5)

Due to how `refresh` is defined and the fact that n_{i-1} is a child of n_i:

- t_{rank} is the time `refresh` reads the rank stored in n_{i-1}, so that $t_{s'} \le t_{rank} \le t_{e'}$. Combining with (5), $t_{rank} \in T_{i-1}$.
- t_{ts} is the time `refresh` reads the timestamp from that rank $t_{s'} \le t_{ts} \le t_{e'}$. Combining with (5), $t_{ts} \in T_{i-1}$.
- There exists a time t', $t_{s'} \le t' \le t_{e'}$, $\textit{min-ts}(\textit{rank}(n_i, t_{e'}), t') \le \textit{min-ts}(\textit{rank}(n_{i-1}, t_{rank}), t_{ts})$. (6)

From (6) and the fact that $t_{rank} \in T_{i-1}$ and $t_{ts} \in T_{i-1}$, $\textit{min-ts}(\textit{rank}(n_i, t_{e'}), t') \le c$.

There shall be no `spsc_dequeue` starting within $t_{s'}$ till the end of T_i because:

- If there's an `spsc_dequeue` starting within T_i, then T_i's assumption is violated.
- If there's an `spsc_dequeue` starting after $t_{s'}$ but before T_i, its `dequeue` must finish its **node-n_i-refresh phase** after $t_{s'}$ and before T_i. However, then $t_{e'}$ is no longer the end of the last successful `refresh` on n_i not after T_i.

Because there's no `spsc_dequeue` starting in this timespan, $\textit{min-ts}(\textit{rank}(n_i, t_{e'}), t_{e'}) \le \textit{min-ts}(\textit{rank}(n_i, t_{e'}), t') \le c$.

If there's no `dequeue` between $t_{e'}$ and the end of T_i whose **node-n_i-refresh phase** hasn't finished, then by Theorem 3.1.8, $\textit{min-ts}(\textit{rank}(n_i, t_r), t_s)$ is monotonically decreasing for any t_r, t_s starting from $t_{e'}$ till the end of T_i. Therefore, $\textit{min-ts}(\textit{rank}(n_i, t_r), t_s) \le c$ for any $t_r, t_s \in T_i$.

Suppose there's a `dequeue` whose **node-n_i-refresh phase** is in progress some time between $t_{e'}$ and the end of T_i. By definition, this `dequeue` must finish it before T_i. Because $t_{e'}$ is the time of the last successful `refresh` on n_i before T_i, $t_{e'}$ must be within the **node-n_i-refresh phase** of this `dequeue` and there should be no `dequeue` after that. By the way $t_{e'}$ is defined, technically, this `dequeue` has finished its **node-n_i-refresh phase** right at $t_{e'}$. Therefore, similarly, we can apply Theorem 3.1.8, $\textit{min-ts}(\textit{rank}(n_i, t_r), t_s) \le c$ for any $t_r, t_s \in T_i$.

By induction, we have proved the stronger version of the theorem. □

Corollary 3.1.10 If an `enqueue` e obtains a timestamp c and finishes at time t_0 and is still **unmatched** at time t_1, then for any subrange T of $[t_0, t_1]$ that does not overlap with a `dequeue`, $\textit{min-spsc-ts}(\textit{rank}(\textit{root}, t_r), t_s) \le c$ for any $t_r, t_s \in T$.

Proof Call t_{start} and t_{end} to be the start and end time of T.

Applying Theorem 3.1.9, we have that $\textit{min-ts}(\textit{rank}(\textit{root}, t_r), t_s) \le c$ for any $t_r, t_s \in T$.

Fix t_r so that $\textit{rank}(\textit{root}, t_r) = r$. We have that $\textit{min-ts}(r, t) \le c$ for any $t \in T$.

$\textit{min-ts}(r, t)$ can only change due to a successful `refreshTimestamp` on the enqueuer node with rank r. Consider the last successful `refreshTimestamp` on the enqueuer node with rank r not after T. Suppose that `refreshTimestamp` reads out the minimum timestamp of the local SPSC at $t' \le t_{start}$.

Therefore, $\textit{min-ts}(r, t_{start}) = \textit{min-spsc-ts}(r, t') \le c$.

We will prove that after t' until t_{end}, there's no `spsc_dequeue` on r running.

Suppose the contrary, then this `spsc_dequeue` must be part of a `dequeue`. By definition, this `dequeue` must start and end before t_{start}, else it violates the assumption of T. If this `spsc_dequeue`

starts after t', then its `refreshTimestamp` must finish after t' and before t_{start}. But this violates the assumption that the last `refreshTimestamp` not after t_{start} reads out the minimum timestamp at t'.

Therefore, there's no `spsc_dequeue` on r running during $[t', t_{end}]$. Therefore, $min\text{-}spsc\text{-}ts(r, t)$ remains constant during $[t', t_{end}]$ because it's not `MAX`.

In conclusion, $min\text{-}spsc\text{-}ts(r, t) \leq c$ for $t \in [t', t_{end}]$.

We have proved the theorem. □

Theorem 3.1.11 Given a rank r. If within $[t_0, t_1]$, there's no uncompleted `enqueues` on rank r and all matching `dequeues` for any completed `enqueues` on rank r has finished, then $rank(n, t) \neq r$ for every node n and $t \in [t_0, t_1]$.

Proof If n doesn't lie on the path from root to the leaf node that's attached to the enqueuer node with rank r, the theorem obviously holds.

Due to Corollary 3.1.2, there can only one `enqueue` and one `dequeue` at a time at an enqueuer node with rank r. Therefore, there is a sequential ordering within the `enqueues` and a sequential ordering within the `dequeues`. Therefore, it's sensible to talk about the last `enqueue` before t_0 and the last matching `dequeue` d before t_0.

Since all of these `dequeues` and `enqueues` work on the same local SPSC and the SPSC is linearizable, d must match the last `enqueue`. After this `dequeue` d, the local SPSC is empty.

When d finishes its **timestamp-refresh phase** at $t_{ts} \leq t_0$, due to Theorem 3.1.6, there's at least one successful `refreshTimestamp` call in this phase. Because the last `enqueue` has been matched, $min\text{-}ts(r, t) =$ `MAX` for any $t \in [t_{ts}, t_1]$.

Similarly, for a leaf node n_0, suppose d finishes its **node-n_0-refresh phase** at $t_{r\text{-}0} \geq t_{ts}$, then $rank(n_0, t) =$ `DUMMY` for any $t \in [t_{r\text{-}0}, t_1]$. (1)

For any non-leaf node $n_i \in path(d)$, when d finishes its **node-n_i-refresh phase** at $t_{r\text{-}i}$, there's at least one successful `refresh` call during this phase. Suppose this `refresh` call starts and ends at $t_{start\text{-}i}$ and $t_{end\text{-}i}$. Suppose $rank(n_{i-1}, t) \neq r$ for $t \in \left[t_{r\text{-}(i-1)}, t_1\right]$. By the way `refresh` is defined after this `refresh` call, n_i will store some rank other than r. Because of (1), after this up until t_1, r never has a chance to be visible to a `refresh` on node n_i during $[n_{i-1}, t]$. In other words, $rank(n_i, t) \neq r$ for $t \in [t_{r\text{-}i}, t_1]$.

By induction, we obtain the theorem. □

Theorem 3.1.12 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 If d doesn't match anything, the theorem holds. If d matches e, the theorem also holds. Suppose d matches e', $e' \neq e$.

If e matches d' and d' precedes d, the theorem also holds. Suppose e matches d' such that d precedes d' or is unmatched. (1)

Suppose e obtains a timestamp of c and e' obtains a timestamp of c'.

Because e precedes d and because an MPSC does not allow multiple `dequeues`, from the start of d at t_0 until after line 5 of `dequeue` (Procedure 11) at t_1, e has finished and there's no `dequeue` running that has *actually performed* `spsc_dequeue`. Also by t_0 and t_1, e is still unmatched due to (1).

Applying Corollary 3.1.10, $min\text{-}spsc\text{-}ts\big(rank(root, t_x), t_y\big) \leq c$ for $t_x, t_y \in [t_0, t_1]$. Therefore, d reads out a rank r such that $min\text{-}spsc\text{-}ts(r, t) \leq c$ for $t \in [t_0, t_1]$. Consequently, d dequeues out a value with a timestamp not greater than c. Because d matches e', $c' \leq c$. However, $e' \neq e$ so $c' < c$.

This means that e cannot precede e', because if so, $c < c'$.

Therefore, e' precedes e or overlaps with e. □

Lemma 3.1.13 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. □

Theorem 3.1.14 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 If d isn't matched, the theorem holds.

Suppose d matches e'. Applying Lemma 3.1.13, e' must precede or overlap with d. In other words, d cannot precede e'.

If e precedes or is e', then d must precede e', which is contradictory.

Therefore, e' must precede e or overlap with e. □

Theorem 3.1.15 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 If both e_0 and e_1 aren't matched, the theorem holds.

Suppose e_1 matches d_1. By Lemma 3.1.13, either e_1 precedes or overlaps with d_1.

If e_0 precedes d_1, applying Theorem 3.1.12 for d_1 and e_0:
- d_1 isn't matched, contradictory.
- d_1 matches e_0, contradictory.
- e_0 matches d_0 and d_0 precedes d_1, the theorem holds.
- d_1 matches e_1 and e_1 precedes e_0, contradictory.
- d_1 matches e_1 and e_1 overlaps with e_0, contradictory.

If d_1 precedes e_0, applying Theorem 3.1.14 for d_1 and e_0:
- d_1 isn't matched, contradictory.
- d_1 matches e_1 and e_1 precedes or overlaps with e_0, contradictory.

Consider that d_1 overlaps with e_0, then d_1 must also overlap with e_1. Call r_1 the rank of the enqueuer that performs e_1. Call t to be the time d_1 atomically reads the root's rank on line 5 of `dequeue` (Procedure 11). Because d_1 matches e_1, d_1 must read out r_1 at t_1.

If e_1 is the first `enqueue` of rank r_1, then t must be after e_1 has started, because otherwise, due to Theorem 3.1.11, r_1 would not be in *root* before e_1.

If e_1 is not the first `enqueue` of rank r_1, then t must also be after e_1 has started. Suppose the contrary, t is before e_1 has started:
- If there's no uncompleted `enqueue` of rank r_1 at t and they are all matched by the time t, due to Theorem 3.1.11, r_1 would not be in *root* at t. Therefore, d_1 cannot read out r_1, which is contradictory.
- If there's some unmatched `enqueue` of rank r_1 at t, d_1 will match one of these `enqueues` instead because:
  - There's only one `dequeue` at a time, so unmatched `enqueues` at t remain unmatched until d_1 performs an `spsc_dequeue`.
  - Due to Corollary 3.1.2, all the `enqueues` of rank r_1 must finish before another starts. Therefore, there's some unmatched `enqueue` of rank r_1 finishing before e_1.
  - The local SPSC of the enqueuer node of rank r_1 is serializable, so $d_1$will favor one of these `enqueues` over e_1.

Therefore, t must happen after e_1 has started. Right at t, no dequeue is actually modifying the LTQueue state and e_0 has finished. If e_0 has been matched at t then the theorem holds. If e_0 hasn't been matched at t, applying Theorem 3.1.9, d_1 will favor e_0 over e_1, which is a contradiction.

We have proved the theorem. □

Theorem 3.1.16 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 If d_0 isn't matched or d_1 isn't matched, the theorem holds.

Suppose d_0 matches e_0 and d_1 matches e_1.

Suppose the contrary, e_1 precedes e_0. Applying Theorem 3.1.12:
- Both e_0 and e_1 aren't matched, which is contradictory.
- e_1 is matched but e_0 is not matched, which contradictory.
- e_1 matches d_1 and e_0 matches d_0 such that d_1 precedes d_0, which is contradictory.

Therefore, the theorem holds. □

Theorem 3.1.17 The modified LTQueue algorithm is linearizable.

Proof Suppose some history H produced from the modified LTqueue algorithm.

If H contains some pending method calls, we can just wait for them to complete (because the algorithm is wait-free, which we will prove later). Therefore, now we consider all H to contain only completed method calls. So, we know that if a dequeue or an enqueue in H is matched or not.

If there are some unmatched enqueues, we can append dequeues sequentially to the end of H until there's no unmatched enqueues. Consider one such H'.

We already have a strict partial order $\rightarrow_{H'}$ on H'. Because the queue is MPSC, there's already a total order among the dequeues.

We will extend $\rightarrow_{H'}$ to a strict total order $\Rightarrow_{H'}$ on H' as follows:
- If $X \rightarrow_{H'} Y$ then $X \Rightarrow_{H'} Y$. (1)
- If a dequeue d matches e then $e \Rightarrow_{H'} d$. (2)
- If a dequeue d_0 matches e_0 and another dequeue matches e_1 such that $d_0 \Rightarrow_{H'} d_1$ then $e_0 \Rightarrow_{H'} e_1$. (3)
- If a dequeue d overlaps with an enqueue e but does not match e, $d \Rightarrow_{H'} e$. (4)

We will prove that $\Rightarrow_{H'}$ is a strict total order on H'. That is, for every pair of different method calls X and Y, either exactly one of these is true $X \Rightarrow_{H'} Y$ or $Y \Rightarrow_{H'} X$ and for any X, $X \not\Rightarrow_{H'} X$.

It's obvious that $X \not\Rightarrow_{H'} X$.

If X and Y are dequeues, because there's a total order among the dequeues, either exactly one of these is true: $X \rightarrow_{H'} Y$ or $Y \rightarrow_{H'} X$. Then due to (1), either $X \Rightarrow_{H'} Y$ or $Y \Rightarrow_{H'} X$. Notice that we cannot obtain $X \Rightarrow_{H'} Y$ or $Y \Rightarrow_{H'} X$ from (2), (3), or (4).
Therefore, exactly one of $X \Rightarrow_{H'} Y$ or $Y \Rightarrow_{H'} X$ is true. (∗)

If X is dequeue and Y is enqueue, in this case (3) cannot help us obtain either $X \Rightarrow_{H'} Y$ or $Y \Rightarrow_{H'} X$, so we can disregard it.

- If $X \rightarrow_{H'} Y$, then due to (1), $X \Rightarrow_{H'} Y$. By definition, X precedes Y, so (4) cannot apply. Applying Theorem 3.1.14, either
  - X isn't matched, (2) cannot apply. Therefore, $Y \not\Rightarrow_{H'} X$.
  - X matches e' and $e' \neq Y$. Therefore, X does not match Y, or (2) cannot apply. Therefore, $Y \not\Rightarrow_{H'} X$.

  Therefore, in this case, $X \Rightarrow_{H'} Y$ and $Y \not\Rightarrow_{H'} X$.
- If $Y \rightarrow_{H'} X$, then due to (1), $Y \Rightarrow_{H'} X$. By definition, Y precedes X, so (4) cannot apply. Even if (2) applies, it can only help us obtain $Y \Rightarrow_{H'} X$. Therefore, in this case, $Y \Rightarrow_{H'} X$ and $X \not\Rightarrow_{H'} Y$.
- If X overlaps with Y:
  - If X matches Y, then due to (2), $Y \Rightarrow_{H'} X$. Because X matches Y, (4) cannot apply. Therefore, in this case $Y \Rightarrow_{H'} X$ but $X \not\Rightarrow_{H'} Y$.
  - If X does not match Y, then due to (4), $X \Rightarrow_{H'} Y$. Because X doesn't match Y, (2) cannot apply. Therefore, in this case $X \Rightarrow_{H'} Y$ but $Y \not\Rightarrow_{H'} X$.

Therefore, exactly one of $X \Rightarrow_{H'} Y$ or $Y \Rightarrow_{H'} X$ is true. $(**)$

If X is `enqueue` and Y is `enqueue`, in this case (2) and (4) are irrelevant:

- If $X \rightarrow_{H'} Y$, then due to (1), $X \Rightarrow_{H'} Y$. By definition, X precedes Y. Applying Theorem 3.1.15,
  - Both X and Y aren't matched, then (3) cannot apply. Therefore, in this case, $Y \not\Rightarrow_{H'} X$.
  - X is matched but Y is not matched, then (3) cannot apply. Therefore, in this case, $Y \not\Rightarrow_{H'} X$.
  - X matches d_x and Y matches d_y such that d_x precedes d_y, then (3) applies and we obtain $X \Rightarrow_{H'} Y$.

  Therefore, in this case, $X \Rightarrow_{H'} Y$ but $Y \not\Rightarrow_{H'} X$.
- If $Y \rightarrow_{H'} X$, this case is symmetric to the first case. We obtain $Y \Rightarrow_{H'} X$ but $X \not\Rightarrow_{H'} Y$.
- If X overlaps with Y, because in H', all `enqueues` are matched, then, X matches d_x and d_y. Because d_x either precedes or succeeds d_y, Applying (3), we obtain either $X \Rightarrow_{H'} Y$ or $Y \Rightarrow_{H'} X$ and there's no way to obtain the other.

Therefore, exactly one of $X \Rightarrow_{H'} Y$ or $Y \Rightarrow_{H'} X$ is true. $(***)$

From $(*)$, $(**)$, $(***)$, we have proved that $\Rightarrow_{H'}$ is a strict total ordering that is consistent with $\rightarrow_{H'}$. In other words, we can order method calls in H' in a sequential manner. We will prove that this sequential order is consistent with FIFO semantics:

- An item can only be `dequeued` once: This is trivial as a `dequeue` can only match one `enqueue`.
- Items are `dequeued` in the order they are `enqueued`: Suppose there are two `enqueues` e_1, e_2 such that $e_1 \Rightarrow_{H'} e_2$ and suppose they match d_1 and d_2. Then we have obtained $e_1 \Rightarrow_{H'} e_2$ either because:
  - (3) applies, in this case $d_1 \Rightarrow_{H'} d_2$ is a condition for it to apply.
  - (1) applies, then e_1 precedes e_2, by Theorem 3.1.15, d_1 must precede d_2, thus $d_1 \Rightarrow_{H'} d_2$.

  Therefore, if $e_1 \Rightarrow_{H'} e_2$ then $d_1 \Rightarrow_{H'} d_2$.
- An item can only be `dequeued` after it's `enqueued`: Suppose there is an `enqueue` e matched by d. By (2), obviously $e \Rightarrow_{H'} d$.
- If the queue is empty, `dequeues` return nothing. Suppose a dequeue d such that any $e \Rightarrow_{H'} d$ is all matched by some d' and $d' \Rightarrow_{H'} d$, we will prove that d is unmatched. By Lemma 3.1.13, d can only match an `enqueue` e_0 that precedes or overlaps with d.
  - If e_0 precedes d, by our assumption, it's already matched by another `dequeue`.
  - If e_0 overlaps with d, by our assumption, $d \Rightarrow_{H'} e_0$ because if $e_0 \Rightarrow_{H'} d$, e_0 is already matched by another d'. Then, we can only obtain this because (4) applies, but then d does not match e_0.

  Therefore, d is unmatched.

In conclusion, $\Rightarrow_{H'}$ is a way we can order method calls in H' sequentially that conforms to FIFO semantics. Therefore, we can also order method calls in H sequentially that conforms to FIFO semantics as we only append `dequeues` sequentially to the end of H to obtain H'.

We have proved the theorem. □

3.2. ABA problem

ABA problem is unlikely to occur, because any CAS usage increases the value of a monotonic tag.

3.3. Memory safety

Memory allocation and deallocation are performed only in the SPSC data structure. Since [1] has proved that it's memory-safe, the modified algorithm is also memory-safe.

3.4. Wait-freedom

All the procedures `enqueue`, `dequeue`, `refresh`, `refreshTimestamp`, `refreshLeaf`, `propagate` are wait-free because there's no possibility of inifinite loops and a process waiting for another process.

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.

[2] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming, Revised Reprint*. Morgan Kaufmann, 2012.