TITLE HERE

Mathias Pedersen, 201808137

Master's Thesis, Computer Science
March 2024
Advisor: Amin Timany

AARHUS
UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Abstract

►in English…◄

Resumé

►in Danish. . . ◄

Acknowledgments

►...◄

Mathias Pedersen
Aarhus, March 2024.

Contents

Chapter 1

Introduction

►motivate and explain the problem to be addressed◄
►example of a citation: [1]◄ ►get your bibtex entries from `https://dblp.org/`◄

Chapter 2

The Great Ideas

The Two-Lock Michael Scott Queue

I present here he an implementation of the Two-lock MS-Queue in HeapLang. This implementation differs slightly from the original, presented in [1], but most changes simply reflect the differences in the two languages.

Preliminaries

The underlying data structure making up the queue is a singly-linked list. The linked-list will always contain at least one element, called the *sentinel* node, marking the beginning of the queue. Note that the sentinel node is itself not part of the queue, but all nodes following it are. The queue keeps a head pointer (ℓ_{head}) which always points to the sentinel, and a tail pointer (ℓ_{tail}) which points to some node in the linked list.

In my implementation, a node can be thought of as a triple $(\ell_{i\_\text{in}}, v_i, \ell_{i\_\text{out}})$. The location ℓ_{i_in} points to the pair $(v_i, \ell_{i\_\text{out}})$, where v_i is the value of the node, and ℓ_{i_out} either points to None which represents the null pointer, or to the next node in the linked list. When we say that a location ℓ points to a node $(\ell_{i\_\text{in}}, v_i, \ell_{i\_\text{out}})$, we mean that $\ell \mapsto \ell_{i\_\text{in}}$. Hence, if we have two adjacent nodes $(\ell_{i\_\text{in}}, v_i, \ell_{i\_\text{out}})$, $(\ell_{i+1\_\text{in}}, v_{i+1}, \ell_{i+1\_\text{out}})$ in the linked list, then we have the following structure: $\ell_{i\_\text{in}} \mapsto (v_i, \ell_{i\_\text{out}})$, $\ell_{i\_\text{out}} \mapsto \ell_{i+1\_\text{in}}$, and $\ell_{i+1\_\text{in}} \mapsto v_{i+1}, \ell_{i+1\_\text{out}}$.

The reader may wonder why there is an extra, intermediary "in" pointer, between the pairs of the linked list, and why the "out" pointer couldn't point directly to the next pair. In the original implementation [1], nodes are allocated on the heap. To simulate this in HeapLang, when creating a new node, we create a pointer to a pair making up the node. Now, in the C-like language used in the original specification, an assignment operator is available which is not present in HeapLang. So in order to mimic this behaviour, we model variables as pointers. In this way, we can model a variable x as a location ℓ_x, and the value stored at ℓ_x is the current value of x. This means that the variable ℓ_{i_out} (called "next" in the original) becomes a location ℓ_{head}, and the value stored at the location is what head is currently assigned to. Since ℓ_{i_out} is supposed to be a variable containing a pointer, then the value saved at that location will also be a pointer.

The queue consists of 3 functions: initialize, enqueue, and dequeue which I now present in turn.

initialize

initialize will first create a single node – the sentinel – marking the start of the linked list. It then creates two locks, H_lock and T_lock, protecting the head and tail pointers, respectively. Finally, it creates the head and tail pointers, both pointing to the sentinel. The queue is then a pointer to a structure containing the head, the tail, and the two locks.
Figure 2.1 illustrates the structure of the queue after initialisation. Note that one of the pointers is coloured blue. This represents a *persistent* pointer; a pointer that will never be updated again. All "in" pointers ℓ_{i_in}, are persistent, meaning that they will always point to $(v_i, \ell_{i\_\text{out}})$. We shall use the notation $\ell \mapsto \Box v$ (introduced in [2]) to mean that ℓ points persistently to v.

Note that in the original specification, a queue is a pointer to a 4-tuple $(\ell_{head}, \ell_{tail}, H\_lock, T\_lock)$. Since HeapLang doesn't support 4-tuples, we instead represent the queue as a pointer to a pair of pairs: $((\ell_{head}, \ell_{tail}), (H\_lock, T\_lock))$.

enqueue

To enqueue a value, we must create a new node, append it to the underlying linked-list, and swing the tail pointer to this new node. These three operations are depicted in figure 2.2.

enqueue takes as argument the value to be enqueued and creates a new node containing this value (corresponding to figure 2.2a). This creation doesn't interact with the underlying queue data-structure, hence why we don't acquire the T_lock first. After creating the new node, we must make the last node in the linked list point to it. Since this operation interacts with the queue, we first acquire the T_lock. Once we obtain the lock, we make the last node in the linked list point to our new node (figure 2.2b). Following this, we swing ℓ_{tail} to the new last node in the linked list (figure 2.2c).

Figure 2.2 also illustrates when pointers become persistent; once the previous last node is updated to point to the newly inserted node, that pointer will never be updated again, hence becoming persistent.

dequeue

It is of course only possible to dequeue an element from the queue if the queue contains at least one element. Hence, the first thing dequeue does is check if the queue is empty. We can detect an empty queue by checking if the sentinel is the last node in the linked list. Being the last node in the linked list corresponds to having the "out" node be None. If this is the case, then the queue is empty and the code returns None. Otherwise, there is a node just after the sentinel, which is the first node of the queue. To dequeue it, we first read the associated value, and next we swing the head to it, making it the new sentinel. Finally, we return the value we read.

Since all of these operations interact with the queue, we shall only perform them after having acquired H_lock.

Figure 2.3 illustrates running dequeue on a non-empty queue. Note that the only change is that the head pointer is swung to the next node in the linked list; the old sentinel is not deleted, it just become unreachable from the heap pointer. In this way, the linked list only ever grows.

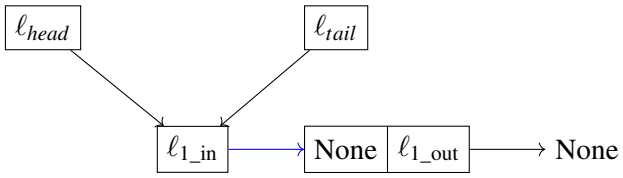
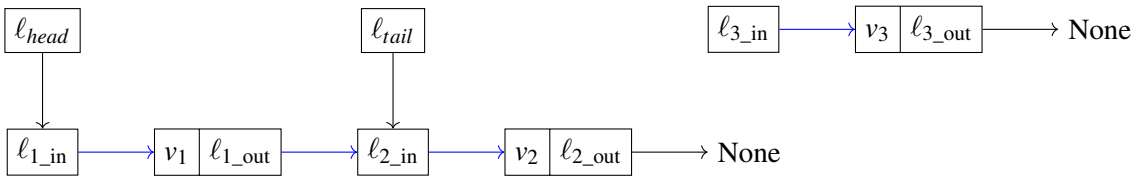
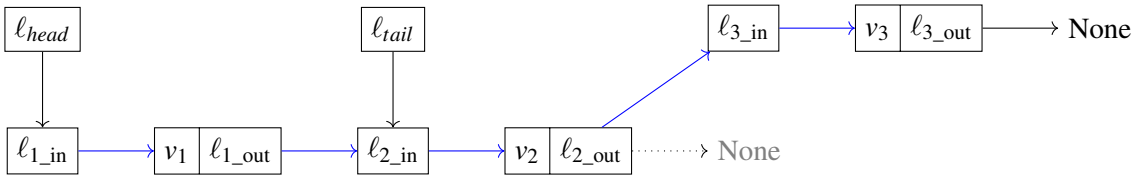


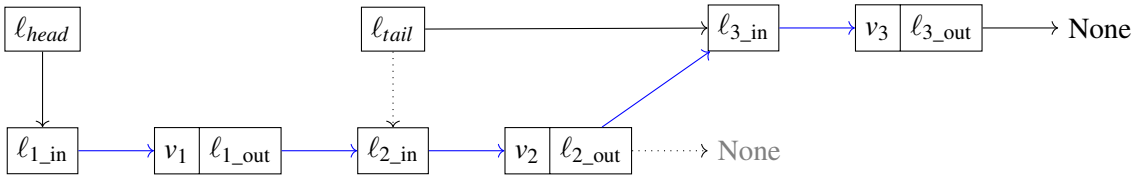
Figure 2.1: Queue after initialisation



(a) Queue after creating the new node $(\ell_{3\_in}, v_3, \ell_{3\_out})$ to be added to the queue.



(b) Queue after adding the new node to linked list.



(c) Queue after swinging tail pointer to the new node.

Figure 2.2: Enqueuing an element to a queue with one element.

$$
\begin{aligned}
&\textbf{let}\,\text{initialize} := \\
&\qquad \textbf{let}\, node = \textbf{ref}\,((\text{None}, \textbf{ref}\,(\text{None})))\,\textbf{in} \\
&\qquad \textbf{let}\, H\_lock = newlock()\,\textbf{in} \\
&\qquad \textbf{let}\, T\_lock = newlock()\,\textbf{in} \\
&\qquad \textbf{ref}\,((\textbf{ref}\,(node), \textbf{ref}\,(node)), (H\_lock, T\_lock))
\end{aligned}
$$

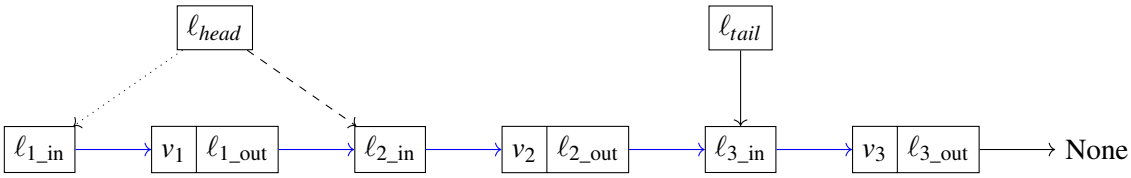


Figure 2.3: Dequeueing an element (v_2) from a queue with two elements (v_2, v_3). The dotted line represents the state before the dequeue, and the dashed line is the state after dequeuing.

```
let enqueue Q value :=
                    let node = ref ((Some value, ref (None))) in
                    acquire(snd(snd(!Q)));
                    snd(!(!(snd(fst(!Q))))) ← node;
                    snd(fst(!Q)) ← node;
                    release(snd(snd(!Q)))
```

```
let dequeue Q :=
                      acquire(fst(snd(!Q)));
                      let node = !(fst(fst(!Q))) in
                      let new_head = !(snd(!node)) in
                      if new_head = None then
                          release(fst(snd(!Q)));
                          None
                      else
                          let value = fst(!new_head) in
                          fst(fst(!Q)) ← new_head;
                          release(fst(snd(!Q)));
                          value
```

Sequential Specification
Key insights

1. Head always points to the first node in the queue.
2. Tail always point to either the last or second last node in the queue.
3. All but the last pointer in the queue (the pointer to null) never change

Insight 2 is true, as it holds initially, and every time a new node is enqueued, the tail pointer is updated to point to the last node, and no other nodes can be enqueue before the update takes place. However, just after the new node has been linked to the queue,

the tail queue will be pointing to the second last node in the queue, until it is updated in the next line.

Insight 3 means that we can mark all pointers in the queue (except the pointer to the null node) as persistent.

Concurrent Specification

Key insights (concurrent)

1. All the same as before
2. The tail can lag one node behind Head
3. At any given time, the queue is in one of four states:
   (a) No threads are interacting with the queue (**Static**)
   (b) A thread is enqueueing (**Enqueue**)
   (c) A thread is dequeuing (**Dequeue**)
   (d) A thread is enqueueing and a thread is dequeuing (**Both**)

Insight 2 might seem a little surprising, and indeed it stands in contrast to property 5 in [1], which states that the tail never lags behind head. I also didn't realise this possibility until a proof attempt using a model that "forgot" old nodes lead to an unprovable case (see section 2). The situation can occur when the queue is empty, and a thread performs an incomplete enqueue; it attaches the new node to the end, but before it can swing the tail to this new node, another thread performs a dequeue, which dequeues this new node, swinging the head to it. Now the tail is lagging a node behind the head.

It is not possible for the tail to point more than one node behind the head, as in order for this to happen, more nodes must be enqueued, but this can't happen before the current enqueue finishes, which will update the tail and bring it up to speed with the head.

Fortunately, this isn't an issue for safety, but a consequence of this possibility is that when modelling the queue, we must remember at least one "old" node (i.e. a dequeued node), as the tail might be pointing to this node. For the sake of simplicity in the model, the choice is made to remember an arbitrary amount of old nodes, which is represented by the list xs_{old}.

With these insights, we can now define the queue invariant.

Definition 2.0.1 (Two-Lock M&S-Queue Invariant) ►*Write it in*◄

The specification for the two-lock Michael Scott Queue *can* be proven using the queue invariant 2.0.1. However, during the proof, a simpler (but arguably less intuitive) queue invariant was discovered This simpler invariant is equivalent to 2.0.1 and has the added benefit of being easier to work with in the proofs. Thus, we shall be using this version of the queue invariant during the proofs.

Definition 2.0.2 (Simplified Two-Lock M&S-Queue Invariant) ►*Write it in*◄

Discussing the need for xs_{old}

As mentioned in the insights, it is possible for the tail to lag one node behind the head. This insight lead to including the old nodes of the queue in the queue invariant. This addition manifests in the end of the proof of dequeue. When we open the invariant to swing the head to the new node, we get that the entire queue is xs. After performing the write, we can then close the invariant with the same xs that we opened the queue to (just written differently to signify that x_{head} is now "old"). Because of this, we can supply the same predicate concerning the $tail$ (the or) that we opened the queue with, since these only mention xs, which remains the same.

Had we not used an xs_{old} and essentially just "forgotten" old nodes in the queue, we couldn't have done this. Say that we defined xs as $xs = x_{head} :: xs_{rest}$ instead. Then, once we have to close the invariant, we cannot supply the xs, which we got when we opened the invariant. Our only choice (due to the fact that $head$ must point to x_{n_head}) is to close the invariant with $xs' = xs_{rest} = x_{n_head} :: xs''_{n_rest}$. However, clearly $xs' \neq xs$, so we cannot supply the same predicate concerning the $tail$ (the or) that we got when opening the invariant, since this predicate talks about xs, not xs'. Now, if we opened the invariant in the Dequeue case, then we could assert that $lastxs = lastxs'$, and hence still be able close the invariant. However, if we opened the invariant in the Both case, then we would need to assert that $2lastxs = 2lastxs'$. This is however not provable, since it might be the case that xs''_{n_rest} is empty, and hence $2lastxs'$ is $None$, whereas $2lastxs = x_{n_head}$.

Chapter 3

Conclusion

►conclude on the problem statement from the introduction◄

Bibliography

[1] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In James E. Burns and Yoram Moses, editors, *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing, Philadelphia, Pennsylvania, USA, May 23-26, 1996*, pages 267–275. ACM, 1996.

[2] Simon Friis Vindum and Lars Birkedal. Contextual refinement of the michael-scott queue (proof pearl). In Catalin Hritcu and Andrei Popescu, editors, *CPP '21: 10th ACM SIGPLAN International Conference on Certified Programs and Proofs, Virtual Event, Denmark, January 17-19, 2021*, pages 76–90. ACM, 2021.

Appendix A

The Technical Details

►…◄