

Optimistic queue algorithm verification

Guillaume Claret

September 6, 2010

Abstract

We are trying to apply Ming Fu's [Historical Logic](#) to give a formal safety proof of an optimistic queue algorithm.

1 Choice of the algorithm

Multiple optimistic queue algorithms were proposed in the past. The [Michael and Scott algorithm](#) stands as a reference, but the paper only gives an informal proof.

They are two versions : one with a GC, simpler, and one without, more subtle and which requires to add a modification counter to each pointer. The GC version already has a formal proof in a [paper from Lindsay Groves](#), so we will focus on the more interesting version without GC.

There is also a [improved algorithm](#) by Ladan-Mozes and Shavit, more complex but with an interesting approach, which we may prove later.

2 Algorithm description

The queue is made of a simply linked list of nodes, with two pointers : *Head* to a dummy head node and *Tail* to the last or second last node. All the pointers have a modification counter field.

```
pointer = {
  node * ptr;
  int c;
}
node = {
  int value;
  pointer next;
}
queue = {
  pointer Head;
  pointer Tail;
}
```

Data structures

```
initialize() {
  node * node;
  node->next.ptr = NULL;
  Head = Tail = node;
}
```

Empty queue

```

void enqueue(int value) {
01:  node * node;
02:  node->value = value;
03:  node->next.ptr = NULL;
04:  while(true) {
05:      tail = Tail;
06:      next = tail.ptr->next;
07:      if(tail == Tail) {
08:          if(next.ptr == NULL) {
09:              if(CAS(&tail.ptr->next, next, (pointer)(node, next.c+1))) {
10:                  CAS(&Tail, tail, (pointer)(node, tail.c+1));
11:                  return;
12:              }
13:          } else
14:              CAS(&Tail, tail, (pointer)(next.ptr, tail.c+1));
15:      }
16:  }
}

```

Enqueue

```

bool dequeue(int * pvalue) {
01:  while(true) {
02:      head = Head;
03:      tail = Tail;
04:      next = head.ptr->next;
05:      if(head == Head) {
06:          if(head.ptr == tail.ptr) {
07:              if(next.ptr == NULL)
08:                  return FALSE;
09:              CAS(&Tail, tail, (pointer)(next.ptr, tail.c+1))
10:          } else {
11:              *pvalue = next.ptr->value;
12:              if(CAS(&Head, head, (pointer)(next.ptr, head.c+1))) {
13:                  free(head.ptr);
13:                  return TRUE;
14:              };
14:          }
15:      }
16:  }
}

```

Dequeue

Note that there are a lot of dangerous dereferencings, like l.6 in *enqueue*, which could fail if the memory region has been freed. So we need some specific hypothesis on the *malloc* / *free* functions (see the [ROP problem](#)).

3 Safety property

We will assume there are several threads indexed by tid_1, tid_2, \dots . Each will execute an *enqueue* or a *dequeue* on one common queue structure, described by *Head* and *Tail*.

Here are the predicate used to define a queue :

$$\begin{aligned}
\text{goodtriple}(L_l, L_v, L_p) &= \text{length}(L_l) = \text{length}(L_v) = \text{length}(L_p) \neq 0 \\
\text{list}([l], [v], [p]) &= l \mapsto (v, p) \wedge (p.\text{ptr} = \text{null}) \\
\text{list}(l : L_l, v : L_v, p : L_p) &= \text{goodtriple}(L_l, L_v, L_p) \wedge (p.\text{ptr} = L_l[0]) \wedge l \mapsto (v, p) * \text{list}(L_l, L_v, L_p) \\
\text{headtail}(L_l) &= \text{length}(L_l) \neq 0 \wedge (H = L_l[0]) \wedge \\
&\quad (T = L_l[-1] \vee (\text{length}(L_l) \geq 2 \wedge T = L_l[-2])) \\
\text{queue}(L_l, L_v, L_p) &= \text{headtail}(L_l) \wedge \text{list}(L_l, L_v, L_p)
\end{aligned}$$

We always suppose that L_l , L_v and L_p are not empty and have the same length. L_l are locations, L_v `int` values and L_p pointers with counter. $L[0]$ means the first element of a list, $L[-1]$ the last one, $L[-2]$ the second last, and $L[m..n]$ a sub-list from element m to n .

We define the atomic operation each thread can do on the queue :

$$\begin{aligned}
\text{enqueue}_{tid} &= \exists L_l, L_v, L_p, v, l, \\
&\quad \text{queue}(L_l, L_v, L_p) \times_{tid} \\
&\quad \text{queue}(L_l + [l], L_v + [v], L_p[0..-2] + [(l, L_p[-1]).c + 1], (\text{null}, -)) \\
\text{dequeue}_{tid} &= \exists l, v, p, L_l, L_v, L_p, \\
&\quad \text{queue}(l : L_l, v : L_v, p : L_p) \times_{tid} \\
&\quad \text{queue}(L_l, L_v, L_p) \\
\text{changetail}_{tid} &= \exists L_l, L_v, L_p, \text{queue}(L_l, L_v, L_p) \times_{tid} \text{queue}(L_l, L_v, L_p)
\end{aligned}$$

The last operation is not exactly *Id*, because the *Tail* pointer can change.

Some definitions about pointers :

$$\begin{aligned}
\text{keep}(x) &= \exists X, (x = X) \times (x = X) \\
\text{pointer}(p) &= \forall C, (\exists p.c = C) \Rightarrow (\text{keep}(p) \vee p.c = C + 1) \\
\text{pointers} &= \forall L_p, \text{queue}(-, -, L_p) \Rightarrow \\
&\quad (\text{pointer}(\text{Head}) \wedge \text{pointer}(\text{Tail}) \wedge (\forall p \in L_p, \text{pointer}(p)))
\end{aligned}$$

which means that all the pointer counters are strictly increasing during updates.

The *R*, *G*, *I* properties :

$$\begin{aligned}
I &= \exists L_l, L_v, L_p, \text{queue}(L_l, L_v, L_p) \\
G_{tid} &= (\text{enqueue}_{tid} \vee \text{dequeue}_{tid} \vee \text{changetail}_{tid}) \wedge (I \times_{tid} I) \wedge \text{pointers} \\
R_{tid} &= \bigvee_{tid' \neq tid} G_{tid'}
\end{aligned}$$

We omit invariants and properties about local variables of each process.

4 Safety proof

4.1 General properties

To simplify the proof, we will use some properties on the data structures which come from the invariants.

- If a pointer p has the same value twice, counter included, it means that it was a constant (here, P means a constant pointer value, φ a formula) :

$$\left. \begin{array}{l} \exists \text{ pointer}(p) \\ (p = P) \blacktriangleright \varphi(P) \\ p = P \end{array} \right\} \Rightarrow \diamond \varphi(p) \quad (1)$$

- If $Tail$ points to the end of the queue, and if the last pointer stays $null$, $Tail$ still points to the end :

$$\left. \begin{array}{l} \exists((R_{tid} \vee G_{tid}) \wedge I) \\ (Tail.ptr + 1 = L \wedge [L] = N) \gg [L] = N \\ N.ptr = null \wedge queue(L_l, -, -) \end{array} \right\} \Rightarrow Tail.ptr + 1 = L = L_l[-1] \quad (2)$$

- A non-null pointer of the queue is a constant :

$$\left. \begin{array}{l} \exists((R_{tid} \vee G_{tid}) \wedge I) \\ (queue(L_l, -, -) \wedge (L - 1) \in L_l \wedge [L] = P) \gg (queue(L'_l, -, -) \wedge (L - 1) \in L'_l) \\ P.ptr \neq null \end{array} \right\} \Rightarrow [L] = P \quad (3)$$

- If the $Head$ pointer is a constant and different from $Tail$, the queue continues to grow :

$$\left. \begin{array}{l} \exists((R_{tid} \vee G_{tid}) \wedge I) \\ H = Head \gg H.ptr \neq Tail.ptr \gg N = H.ptr \rightarrow next \gg H = Head \end{array} \right\} \Rightarrow \left\{ \begin{array}{l} Heap.ptr \neq Tail.ptr \\ N = H.ptr \rightarrow next \end{array} \right. \quad (4)$$

We now give a *Hoare* triple in sequential logic which can be used to prove each basic operation :

- *enqueue* :
 $\{queue(L_l, L_v, L_p) * l \mapsto (v, null, -) \wedge Tail.ptr = L_l[-1]\}$
 $Tail.ptr \rightarrow next = (pointer)(1, Tail.ptr \rightarrow next.c + 1);$
 $\{queue(L_l + [l], L_v + [v], L_p[0..-2] + [(l, L_p[-1].c + 1), (null, -)])\}$
- *dequeue* :
 $\{queue(l : L_l, v : L_v, p : L_p) \wedge Head.ptr \neq Tail.ptr\}$
 $Head = (pointer)(Head.ptr \rightarrow next.ptr, Head.c + 1);$
 $\{queue(L_l, L_v, L_p) * L_l[0] \mapsto (L_v[0], L_p[0].ptr, L_p[0].c)\}$
- *changetail* :
 $\{queue(L_l, L_v, L_p) \wedge Tail.ptr \rightarrow next.ptr \neq null\}$
 $Tail = (pointer)(Tail.ptr \rightarrow next.ptr, Tail.c + 1);$
 $\{queue(L_l, L_v, L_p)\}$

4.2 Proof

We will use the notation \boxed{P} for $(P * true) \wedge I$, *i.e.*, a property about the invariant data structure.

```
void enqueue(int value) {
   $\boxed{true}$ 
01: node * node;
02: node->value = value;
03: node->next.ptr = NULL;
   $\boxed{true} * node \mapsto (value, null, -)$ 
04: while(true) {
05:   tail = Tail;
   $\boxed{\diamond tail = Tail} * node \mapsto (value, null, -)$ 
06:   next = tail.ptr->next;
   $\boxed{(\diamond tail = Tail) \wedge (\diamond next = tail.ptr \rightarrow next)} * node \mapsto (value, null, -)$ 
07:   if(tail == Tail) {
    by (1), with  $p = Tail$  and  $P = tail$ , since  $Tail = tail$  :
```

```

    {  $P_{tail,next}$  } * node  $\mapsto$  (value, null, -)
with  $P_{tail,next} = \diamond(tail = Tail \wedge next = Tail.ptr \rightarrow next)$ .
08:   if(next.ptr == NULL) {
    {  $next.ptr = null \wedge P_{tail,next}$  } * node  $\mapsto$  (value, null, -)
09:     /* if(CAS(&tail.ptr->next, next, (pointer)(node, next.c+1))) */
09:     b = false;
09:     <if(tail.ptr->next == next) {
    {  $tail.ptr \rightarrow next = next \wedge next.ptr = null \wedge P_{tail,next}$  } * node  $\mapsto$  (value, null, -)
    by (2) and  $P_{tail,next}$ , with  $L = tail.ptr + 1$  and  $N = next$  :
    {  $tail.ptr = Tail.ptr = L[-1] \wedge tail.ptr \rightarrow next = next$  } * node  $\mapsto$  (value, null, -)
09:     tail.ptr->next = (pointer)(node, next.c+1);
    At this point we just achieved an enqueue operation and :
    {  $node \neq null \wedge tail.ptr \rightarrow next.ptr = node \wedge Tail.ptr = tail.ptr$  }
09:     b = true;
09:   }>
09:   if(b) {
    {  $node \neq null \wedge \diamond(tail.ptr \rightarrow next.ptr = node \wedge Tail.ptr = tail.ptr)$  }
10:     /* CAS(&Tail, tail, (pointer)(node, tail.c+1)); */
10:     <if(Tail == tail)
    {  $node \neq null \wedge Tail = tail \wedge \diamond(tail.ptr \rightarrow next.ptr = node \wedge Tail.ptr = tail.ptr)$  }
    then by (3), with  $L = tail.ptr + 1$  and  $P = (node, -)$ , since  $Tail = tail$  :
    {  $node \neq null \wedge Tail = tail \wedge Tail.ptr \rightarrow next.ptr = node$  }
10:     Tail = (pointer)(node, tail.c+1);
    Tail pointer has been advanced, which is the changetail operation.
10:     >
11:     return;
12:   }
13: } else
14:   /* CAS(&Tail, tail, (pointer)(next.ptr, tail.c+1)); */
15:   <if(Tail == tail)
    {  $Tail = tail \wedge next.ptr \neq null \wedge P_{tail,next}$  } * node  $\mapsto$  (value, null, -)
    by (3) with  $L = tail.ptr + 1$  and  $P = next$ , since  $Tail = tail$  :
    {  $Tail = tail \wedge next.ptr \neq null \wedge Tail.ptr \rightarrow next.ptr = next$  } * node  $\mapsto$  (value, null, -)
15:     Tail = (pointer)(next.ptr, tail.c+1);
    Tail pointer has been advanced, which is the changetail operation.
15:     >
16:   }
}

bool dequeue(int * pvalue) {
  { true }
01: while(true) {
02:   head = Head;
03:   tail = Tail;
04:   next = head.ptr->next;
  {  $(head = Head \blacktriangleright tail = Tail \blacktriangleright next = head.ptr \rightarrow next) \wedge (\diamond head = Head) \wedge (\diamond next = head.ptr \rightarrow next)$  }
05:   if(head == Head) {
    by (1), with  $p = Head$  and  $P = head$ , since  $Head = head$  :
    {  $(head = Head \blacktriangleright tail = Tail \blacktriangleright next = head.ptr \rightarrow next) \wedge P_{head,next}$  }
    with  $P_{head,next} = \diamond(head = Head \wedge next = head.ptr \rightarrow next)$ .
06:     if(head.ptr == tail.ptr) {
07:       if(next.ptr == NULL)

```

```

08:         return FALSE;
        {  $head.ptr = tail.ptr \wedge next.ptr \neq null \wedge P_{head,next}$  }
09:         /* CAS(&Tail, tail, (pointer)(next.ptr, tail.c+1)) */
09:         <if(Tail == tail) {
        {  $Tail = tail \wedge head.ptr = tail.ptr \wedge next.ptr \neq null \wedge P_{head,next}$  }
    by (1), with  $p = Tail$  and  $P = tail$ , since  $head.ptr = tail.ptr$  :
        {  $Tail = tail \wedge next.ptr \neq null \wedge P_{tail,next}$  }
    with  $P_{tail,next} = \diamond(tail = Tail \wedge next = Tail.ptr \rightarrow next)$ .
    Then, by (3) with  $L = tail.ptr + 1$  and  $P = next$  :
        {  $Tail = tail \wedge next.ptr \neq null \wedge next = Tail.ptr \rightarrow next$  }
09:         Tail = (pointer)(next.ptr, tail.c+1);
    Tail pointer has been advanced, which is the changetail operation.
09:         >
10:     } else {
11:         *pvalue = next.ptr->value;
        {  $(head = Head \blacktriangleright tail = Tail \blacktriangleright next = head.ptr \rightarrow next) \wedge head.ptr \neq tail.ptr$  }
12:         /* if(CAS(&Head, head, (pointer)(next.ptr, head.c+1))) { */
12:         b = false;
12:         <if(Head == head) {
    since  $head.ptr \neq tail.ptr$  :
        {  $(head = Head \gg head.ptr \neq Tail.ptr \gg next = head.ptr \rightarrow next \gg head = Head)$  }
    by (4), with  $H = head$  and  $N = next$  :
        {  $Head.ptr \neq Tail.ptr \wedge next = Heap.ptr \rightarrow next \wedge head = Head$  }
12:         Head = (pointer)(next.ptr, head.c+1);
    Hence we did a dequeue operation and :
        {  $true * head.ptr \mapsto (-, -, -)$  }
12:         b = true;
12:     }>
12:     if(b) {
        {  $true * head.ptr \mapsto (-, -, -)$  }
13:         free(head.ptr);
        {  $true$  }
13:         return TRUE;
14:     };
14: }
15: }
16: }
}

```

5 TODO

- Check real implementations :
 - <http://www.codeproject.com/KB/cpp/lockfreeeq.aspx> seems wrong (1.6 non-atomic in *enqueue*, dereferencing errors in C++ version, ...)
 - [Java version in ClassPath](#) seems OK
 - [original article implementation](#) seems OK, but dereferencing 1.6 in *enqueue* rely on a specific memory heap manager
 - [Iris library](#) seems OK, but they use a pointer counter modulo only 4
- Do the proof in Coq ?
- Reformulate proof not to use temporal operators in beginning of atomic block (with an imply)

- Assumptions made :
 - some potentially unsafe dereferencing : use an explicit free / malloc function instead
 - pointer.c increase only when updates : not exact l.9 in enqueue ; solution : explicit free / malloc