Proving Linearizability Using Partial Orders

Artem Khyzha   Mike Dodds
Alexey Gotsman   Matthew Parkinson
In this talk

• Algorithms posing challenges for the linearization points method:
  
  • the Herlihy-Wing queue
  
  • the Optimistic Set / Lazy List
  
  • the Timestamp Queue
In this talk

• Algorithms posing challenges for the linearization points method:
  • the Herlihy-Wing queue
  • the Optimistic Set / Lazy List
  • the Timestamp Queue
Linearizability

for every concrete history

\[ t_1 \quad e(\text{`a'}) \quad d(?) \]
\[ t_2 \quad e(\text{`b'}) \quad e(\text{`d'}) \]
\[ t_3 \quad e(\text{`c'}) \]

find a linearization

\[ e(\text{`b'}) \quad e(\text{`a'}) \quad e(\text{`c'}) \quad d(?) \quad e(\text{`d'}) \]
Linearizability

for every concrete history

\[ t_1 \quad e(\text{'a'}) \quad d(?) \]

\[ t_2 \quad e(\text{'b'}) \quad e(\text{'d'}) \]

\[ t_3 \quad e(\text{'c'}) \]

find a linearization

\[ e(\text{'b'}) \quad e(\text{'a'}) \quad e(\text{'c'}) \quad d(?) \quad e(\text{'d'}) \]

Standard proof technique: linearization points
for every concrete history

\[ t_1 \quad \text{e(‘a’) \quad d(?) \quad e(‘d’) \quad e(‘c’) \quad e(‘b’) \quad e(‘a’) \quad e(‘c’) \quad d(?) \quad e(‘d’) \quad \text{e(‘b’)} \quad \text{? = ‘b’} \]

find a linearization

Linearization points
Linearization points

by the moment (*) we still may not know where the linearization points of e('a') and e('b') are
The TS Queue

- each thread has its own pool
- pool is an abstract sequence
- (single producer multiple consumers)
The TS Queue

<table>
<thead>
<tr>
<th>t₁</th>
<th>t₂</th>
<th>t₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘a’</td>
<td>‘b’</td>
<td>‘c’</td>
</tr>
<tr>
<td>‘d’</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ (3,4) \]

\[ (1,1) \]

\[ (2,3) \]

\[ (2,4) \]

\[ t₁ \text{ e(‘a’)} \]

\[ t₂ \quad \text{e(‘b’)} \]

\[ t₃ \quad \text{e(‘c’)} \]

\[ d(?) \]

\[ \text{e(‘d’)} \]
The TS Queue

enqueue(Val v) {
    ts := newTimestamp();
    insert(this_thread, v, ts);
}

---

t1
  'a'
  ts_a

  'b'
  ts_b

  'd'
  ts_d

  
  
  t2
  'b'
  ts_b

  'd'
  ts_d

  t3
  'c'
  ts_c

  
  
  enqueue(Val v) {
    ts := newTimestamp();
    insert(this_thread, v, ts);
  }
The TS Queue

Val dequeue() {
    do {
        for each pool do {
            get the front node;
            update the candidate for removal;
        }
        if (there is a candidate)
            try removing it and returning its value;
    } while (true);
}
The TS Queue

Val dequeue() {
    do {
        for each pool do {
            get the front node;
            update the candidate for removal;
        }
        if (there is a candidate)
            try removing it and returning its value;
    } while (true);
}
Val dequeue() {
    do {
        for each pool do {
            get the front node;
            update the candidate
            for removal;
        }
        if (there is a candidate)
            try removing it and
            returning its value;
    } while (true);
}

by choosing the
smallest timestamp

<table>
<thead>
<tr>
<th>Pool</th>
<th>Timestamp</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>t₁</td>
<td>(t_{sa})</td>
<td>'a'</td>
</tr>
<tr>
<td>t₂</td>
<td>(t_{sb})</td>
<td>'b'</td>
</tr>
<tr>
<td>t₃</td>
<td>(t_{sc})</td>
<td>'c'</td>
</tr>
<tr>
<td></td>
<td>(t_{sd})</td>
<td>'d'</td>
</tr>
</tbody>
</table>
• the connection with the real-time order

• \( e_1 \longrightarrow_{rt} e_2 \implies \text{timestamp}(e_1) < \text{timestamp}(e_2) \)
Val dequeue() {
    do {
        for each pool do {
            get the front node;
            update the candidate for removal;
        }
        if (there is a candidate)
            try removing it and returning its value;
    } while (true);
}

the smallest timestamp belongs to the "oldest" element
- problem: the order on enqueues may be impossible to determine till they are dequeued
Our technique

• We alter the standard proof technique to support late choice via partial orders

• Our proof technique is implemented in a logic, should be sound

• Examples: the TS queue, the Herlihy-Wing queue, the Optimistic Set
Abstract histories

- For each history of a data structure we construct a matching abstract history (instead of a linearization)

- An “abstract” history (E, R):
  - E — a set of events [eid: (tid, op, arg, rval)]
  - R — a partial order
Abstract histories

- For each history of a data structure we construct a **matching** abstract history (instead of a linearization)

1. the abstract history extends the real-time order
2. all linearizations meet the sequential spec
Abstract histories

• We prove a number of invariant properties including:

  • all linearizations meet the sequential spec

  • for all enqueues e, e’ with values in the data structure:

    • $e \rightarrow e’ \implies \text{timestamp}(e) < \text{timestamp}(e’)$

    • …
Commitment points

- Abstract histories are constructed:
  - by adding new events
  - by adding more edges into the partial order
  - by assigning a return value to an event
Commitment points

- At the beginning of each operation, a fresh event and real-time order edges are added to the abstract history.

\[ t_1 \quad e('a') \quad d('b') \]

\[ t_2 \quad e('b') \]

\[ t_3 \quad e('c') \]
Commitment points

- At the beginning of each operation, a fresh event and real-time order edges are added to the abstract history.

\[ \text{t}_1 \quad \text{e(‘a’)} \quad \text{d(‘b’)} \quad \text{e(‘a’)} \quad \text{t}\_1, \text{enq, ‘a’, TODO} \]

\[ \text{t}_2 \quad \text{e(‘b’)} \quad \text{e(‘c’)} \]

\[ \text{t}_3 \quad \text{e(‘c’)} \]
Commitment points

- At the beginning of each operation, a fresh event and real-time order edges are added to the abstract history.
Commitment points

- At the beginning of each operation, a fresh event and real-time order edges are added to the abstract history.

- Events get completed by the end of the operations.
Commitment points

- At the beginning of each operation, a fresh event and real-time order edges are added to the abstract history.

- Edges from the completed events are added.
Enqueue’s commitment point

enqueue(Val v) {
    ts := newTimestamp();
    atomic {
        insert(this_thread, v, ts);
        E(this_event).rval := DONE;
        G[this_event] := ts;
    }
}

• Ghost state G is a map from events and timestamps

• Helps to establish a bijection between events and elements of the data structure
Commitment points

- The late choice is resolved by a dequeue with the FIFO policy in mind
Commitment points

- The late choice is resolved by a dequeue with the FIFO policy in mind

1. e('b') e('a') d('b') e('c')
2. e('b') e('a') e('c') d('b')
3. e('a') e('b') d('b') e('c')
4. e('a') e('b') e('c') d('b')
Commitment points

- The late choice is resolved by a dequeue with the FIFO policy in mind

1. e('b') e('a') d('b') e('c')
2. e('b') e('a') e('c') d('b')
3. e('a') e('b') d('b') e('c')
4. e('a') e('b') e('c') d('b')

![Diagram showing the sequence of events]
Commitment points

• The late choice is resolved by a dequeue with the FIFO policy in mind

1. e(‘b’) e(‘a’) d(‘b’) e(‘c’)
2. e(‘b’) e(‘a’) e(‘c’) d(‘b’)

[Diagram showing the sequence of events]
Commitment points

• The late choice is resolved by a dequeue with the FIFO policy in mind
Dequeue’s commitment point

if (there is a candidate enq) {
    res := try removing enq;
    if (res != FAIL) {
        E(this_event).rval := res;
        R := (R
            U {(enq, this_event)}
            U {(enq, e’) | the value of e’ is in queue}
            U {(deq, d’) | d’ is an uncompleted dequeue})+;
    }
}
Acyclicity

• Need to prove that ordering by commitment points is possible without breaking acyclicity

  • Loop invariant — implies that the candidate for removal is the minimal in the partial order

  • Invariant properties (what holds by construction of the partial order)
The proof

- For each history of a data structure we construct a matching abstract history (instead of a linearization)
  - preserving the real-time order
  - ensuring that all linearizations meet the sequential specification
  - constructed with commitment points
  - Partiality of the order enables delaying the choice of a linearization order
Conclusions

• The technique for proving linearizability of algorithms that are challenging with the linearization points method

• Examples: the TS queue, the Herlihy-Wing queue, the Optimistic Set

• Does not work for the TS stack