Lecture 24: Linearizability of Concurrent Objects (contd)

Vivek Sarkar
Department of Computer Science, Rice University
vsarkar@rice.edu

https://wiki.rice.edu/confluence/display/PARPROG/COMP322
Acknowledgments for Today’s Lecture

  — Optional text for COMP 322

• Lecture on “Linearizability” by Mila Oren
  — http://www.cs.tau.ac.il/~afek/Mila.Linearizability.ppt
Linearizability of Concurrent Objects (Recap)

Concurrent object

- A concurrent object is an object that can correctly handle methods invoked in parallel by different tasks or threads.
  
  - Examples: concurrent queue, AtomicInteger

Linearizability

- Assume that each method call takes effect “instantaneously” at some distinct point in time between its invocation and return.

- An execution is linearizable if we can choose instantaneous points that are consistent with a sequential execution in which methods are executed at those points.

- An object is linearizable if all its possible executions are linearizable.
Example 1

Is this execution linearizable?

Source: http://www.elsevierdirect.com/companions/9780123705914/Lecture%20Slides/03~Chapter_03.ppt
Example 2

Is this execution linearizable?

```
q.enq(x)
q.enq(y)
q.deq():y
q.deq():x
```

(time)
One Possible Attempt to Implement a Concurrent Queue

1. // Assume that no. of enq() operations is < Integer.MAX_VALUE
2. class Queue1 {
3.    AtomicInteger head = new AtomicInteger(0);
4.    AtomicInteger tail = new AtomicInteger(0);
5.    Object[] items = new Object[Integer.MAX_VALUE];
6.    public void enq(Object x) {
7.        int slot = tail.getAndIncrement(); // isolated(tail) ...
8.        items[slot] = x;
9.    } // enq
10.   public Object deq() throws EmptyException {
11.        int slot = head.getAndIncrement(); // isolated(head) ...
12.        Object value = items[slot];
13.        if (value == null) throw new EmptyException();
14.        return value;
15.    } // deq
16. } // Queue1

17. // Client code
18. finish {
19.    Queue1 q = new Queue1();
20.    async q.enq(new Integer(1));
21.    q.enq(new Integer(2));
22.    Integer x = (Integer) q.deq();
23. }
We split a method call into two events:

- **Invocation**: method names + args
  - `q.enq(x)`

- **Response**: result or exception
  - `q.enq(x)` returns void
  - `q.deq()` returns `x` or throws `emptyException`
Notations for invocations and responses

• **Invocation notation:** A q.enq(x)
  • A – thread
  • q – object
  • enq – method
  • x – arg

• **Response notation:** A q: void , A q: empty()
  • A – thread
  • q – object
  • void – result, exception
Execution History

A sequence of invocations and responses. It describes an execution.

\[ H = \]

A q.enq(3)
A q:void

A q.enq(5)
B p.enq(4)
B p:void
B q.deq()
B q:3
Some Definitions

- Invocation and Response match if: thread names and object names agree.

- A pending invocation is an invocation that has no matching response.

- Complete history: history without pending invocations.
Sequential History

- **Sequential history**: A sequence of matches, can end with pending invocation.

```
A q.enq(3)
A q:void
B p.enq(4)
B p:void
B q.deq()
B q:3
A q:enq(5)
```
Object and Thread Projections of Histories

- **Object projection:**

  \[ H \mid q = \]
  
  A q.enq(3)
  A q: void
  A q.enq(5)
  B q.deq()
  B q: 3

- **Thread projection:**

  \[ H \mid A = \]
  
  A q.enq(3)
  A q: void
  A q.enq(5)
Well-formed and Equivalent Histories

- **Well-formed history**: for each thread A, \( H|A \) is sequential.
- **Equivalent histories**: \( H \) and \( G \) are equivalent if for each thread A: \( H|A = G|A \)

\[
H = \begin{align*}
A & \text{ q.enq(3)} \\
B & \text{ p.enq(4)} \\
B & \text{ p:void} \\
B & \text{ q.deq()} \\
A & \text{ q:void} \\
B & \text{ q:3}
\end{align*}
\]

\[
G = \begin{align*}
A & \text{ q.enq(3)} \\
A & \text{ q:void} \\
B & \text{ p.enq(4)} \\
B & \text{ p:void} \\
B & \text{ q.deq()} \\
B & \text{ q:3}
\end{align*}
\]
Precedes relation on method calls

- Method call \( m_0 \) precedes method call \( m_1 \) in history \( H \) if \( m_0 \)'s response event precedes \( m_1 \)'s invocation event in \( H \)

\[
\begin{align*}
A & \quad \text{q.enq(3)} \\
B & \quad \text{p.enq(4)} \\
B & \quad \text{p.void} \\
A & \quad \text{q:void} \\
B & \quad \text{q.deq()} \\
B & \quad \text{q:3}
\end{align*}
\]

Notation:
\[
m_0 \Rightarrow^H m_1
\]

\( m_0 \) precedes \( m_1 \)
Non-Precedence

A q.enq(3)
B p.enq(4)
B p.void
B q.deq()
A q:void
B q:3

Some method calls overlap one another

Method call
Method call
Legality condition for a sequential history

- A sequential history $H$ is legal if:
  - for each object $x$, $H|x$ is in the sequential specification for $x$.
- for example: objects like queue, stack
Formal definition of Linearizability

- History $H$ is **linearizable** if it can be extended to history $G$ so that $G$ is equivalent to legal sequential history $S$ where $\models G \subseteq \models S$.
  - if $m_0$ precedes $m_1$ in $G$, $m_0$ must also precede $m_1$ in $S$
  - $G$ is the same as $H$ but without pending invocations
    - append responses to pending invocations that “took effect”
    - discard pending invocations that “don’t matter”
What is meant by $\Rightarrow_G \subset \Rightarrow_S$

$\Rightarrow_G = \{a \rightarrow c, b \rightarrow c\}$

$\Rightarrow_S = \{a \rightarrow b, a \rightarrow c, b \rightarrow c\}$

A limitation on the Choice of $S$!
Remarks

• Some pending invocations
  - Took effect, so keep them
  - Discard the rest

• Condition $\Rightarrow_G \subseteq \Rightarrow_S$
  - Means that $S$ respects “real-time order” of $G$
Example

A \texttt{q.enq(3)}
B \texttt{q.enq(4)}
B \texttt{q: void}
B \texttt{q.deq()}
B \texttt{q: 4}
B \texttt{q.enq(6)}
Example

A. q.enq(3)
B. q.enq(4)
B. q:void
B. q.deq()
B. q:4
B. q:enq(6)

Complete this pending invocation

A. q.enq(3)
B. q.enq(4)
B. q.deq(3)
Example

Complete this pending invocation

A.q.enq(3)
B.q.enq(4)
B.q: void
B.q.deq()
B.q: 4
B.q.enq(6)
A.q: void
Example

A q.enq(3)
B q.enq(4)
B q:void
B q.deq()
B q:4
B q:enq(6)
A q:void

discard this one

B.q.enq(3)
B.q.enq(4)
B.q.deq(4)
B.q.enq(6)
Example

A q.enq(3)
B q.enq(4)
B q: void
B q.deq()
B q: 4
A q: void
discard this one

B q.enq(3)
B q.enq(4)
B q.deq(4)
time
Example

A q.enq(3)
B q.enq(4)
B q:void
B q.deq()
B q:4
A q:void
Example

A q.enq(3)
B q.enq(4)
B q:void
B q.deq()
B q:4
A q:void

B q.enq(4)
B q:void
A q.enq(3)
A q:void
B q.deq()
B q:4

B.q.enq(3)
B.q.enq(4)
B.q.deq(4)
Example

Equivalent sequential history

A q.enq(3)
B q.enq(4)
B q: void
B q.deq()
B q:4
A q: void

B q.enq(4)
B q: void
A q.enq(3)
A q: void
B q.deq()
B q:4

B.q.enq(3)
B.q.enq(4)
B.q.deq(4)

time
Two Important Properties that follow from Linearizability

1) **Composability**
   - History $H$ is linearizable if and only if
     - For every object $x$
     - $H|x$ is linearizable
   - Why is composability important?
     - Modularity
     - Can prove linearizability of objects in isolation
     - Can compose independently-implemented objects

2) **Non-blocking**
   - One method call is never forced to wait on another
   - If method invocation “A q.inv(…)” is pending in history $H$, then there exists a response “A q:res(…)” such that “$H + A q:res(…)$” is linearizable