COMP 322: Fundamentals of Parallel Programming

Lecture 21: Linearizability of Concurrent Objects

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

• Graded midterm exams can be picked up from Amanda Nokleby in Duncan Hall room 3137
• Homework 5 (written assignment) has been posted
  — Deadline: 5pm on Friday, March 18th
• Homework 6 (HJ programming assignment) will be given on March 18th
• Homework 7 (Concurrent Java programming assignment) will be given on April 1st (really!)
Acknowledgments for Today’s Lecture

- Lecture 21 handout
  - Optional text for COMP 322
Concurrent Objects

• A concurrent object is an object that can correctly handle methods invoked in parallel by different tasks or threads
  — Originally referred to as monitors
  — Also informally referred to as “thread-safe objects”

• For simplicity, it is usually assumed that the body of each method in a concurrent object is itself sequential
  — Assume that method does not create child async tasks

• Implementations of methods can be serial (e.g., enclose each method in an isolated statement like a critical section) or concurrent (e.g., ConcurrentHashMap, ConcurrentLinkedQueue and CopyOnWriteArraySet)

• A desirable goal is to develop method implementations that are concurrent while being as close to the semantics of the serial version as possible
The Big Question!

• Consider a simple FIFO (First In, First Out) queue as a canonical example of a concurrent object
  - Method `q.enq(o)` inserts object `o` at the tail of the queue
    - Assume that there is unbounded space available for all `enq()` operations to succeed
  - Method `q.deq()` removes and returns the item at the head of the queue.
    - Throws `EmptyException` if the queue is empty.

• What does it mean for a concurrent object like a FIFO queue to be correct?
  - What is a concurrent FIFO queue?
    - FIFO means strict temporal order
    - Concurrent means ambiguous temporal order
Describing the concurrent via the sequential

Behavior is “Sequential”

Source: http://www.elsevierdirect.com/companions/9780123705914/Lecture%20Slides/03~Chapter_03.ppt
Informal definition of Linearizability

1. A *linearizable execution* is one in which the semantics of a set of method calls performed in parallel on a concurrent object is equivalent to that of some legal linear sequence of those method calls.

2. A *linearizable concurrent object* is one for which all possible executions are linearizable.
### Table 1: Example execution of a monitor-based implementation of FIFO queue q

**Is this a linearizable execution?**

<table>
<thead>
<tr>
<th>Time</th>
<th>Task A</th>
<th>Task B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Invoke q.enq(x)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Work on q.enq(x)</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Work on q.enq(x)</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Return from q.enq(x)</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>Invoke q.enq(y)</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>Work on q.enq(y)</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>Work on q.enq(y)</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>Return from q.enq(y)</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>Invoke q.deq()</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>Return x from q.deq()</td>
</tr>
</tbody>
</table>

Yes! Equivalent to “q.enq(x); q.enq(y); q.deq():x”
Table 2: Example execution of method calls on a concurrent FIFO queue $q$

Is this a linearizable execution?

<table>
<thead>
<tr>
<th>Time</th>
<th>Task A</th>
<th>Task B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Invoke $q$.enq(x)</td>
<td>Invoke $q$.enq(y)</td>
</tr>
<tr>
<td>1</td>
<td>Work on $q$.enq(x)</td>
<td>Return from $q$.enq(y)</td>
</tr>
<tr>
<td>2</td>
<td>Work on $q$.enq(x)</td>
<td>Invoke $q$.deq()</td>
</tr>
<tr>
<td>3</td>
<td>Return from $q$.enq(x)</td>
<td>Return x from $q$.deq()</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Yes! Equivalent to “$q$.enq(x) ; $q$.enq(y) ; $q$.deq():x”

• Would the execution be linearizable if $q$.deq() returned $y$ instead of $x$?
Table 3: Example of a non-linearizable execution on a concurrent FIFO queue `q` 

Is this a linearizable execution?

<table>
<thead>
<tr>
<th>Time</th>
<th>Task A</th>
<th>Task B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Invoke <code>q.enq(x)</code></td>
<td>Invoke <code>q.enq(y)</code></td>
</tr>
<tr>
<td>1</td>
<td>Return from <code>q.enq(x)</code></td>
<td>Work on <code>q.enq(y)</code></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>Return from <code>q.enq(y)</code></td>
</tr>
<tr>
<td>3</td>
<td>Invoke <code>q.deq()</code></td>
<td>Work on <code>q.enq(y)</code></td>
</tr>
<tr>
<td>4</td>
<td>Work on <code>q.deq()</code></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Return y from <code>q.deq()</code></td>
<td></td>
</tr>
</tbody>
</table>

- No! `q.enq(x)` must precede `q.enq(y)` in all linear sequences of method calls invoked on `q`. It is illegal for the `q.deq()` operation to return `y`. 
Alternate definition of Linearizability

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

• Execution is linearizable if we can choose instantaneous points that are consistent with a sequential execution in which methods are executed at those points.
Table 2: Example execution of method calls on a concurrent FIFO queue q

Is this a linearizable execution?

<table>
<thead>
<tr>
<th>Time</th>
<th>Task A</th>
<th>Task B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Invoke q.enq(x)</td>
<td>Invoke q.enq(y)</td>
</tr>
<tr>
<td>1</td>
<td>Work on q.enq(x)</td>
<td>Return from q.enq(y)</td>
</tr>
<tr>
<td>2</td>
<td>Work on q.enq(x)</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Return from q.enq(x)</td>
<td>Invoke q.deq()</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>Return x from q.deq()</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Yes! Equivalent to “q.enq(x) ; q.enq(y) ; q.deq():x”

• Would the execution be linearizable if q.deq() returned y instead of x?
An Example

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

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

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

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

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

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

Source: [http://www.elsevierdirect.com/companions/9780123705914/Lecture%20Slides/03~Chapter_03.ppt](http://www.elsevierdirect.com/companions/9780123705914/Lecture%20Slides/03~Chapter_03.ppt)
Another Example (like Table 3)

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

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

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

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

Source: [http://www.elsevierdirect.com/companions/9780123705914/Lecture%20Slides/03~Chapter_03.ppt](http://www.elsevierdirect.com/companions/9780123705914/Lecture%20Slides/03~Chapter_03.ppt)
Another Example

Source: http://www.elsevierdirect.com/companions/9780123705914/Lecture%20Slides/03~Chapter_03.ppt
Figure 1: Computation Graph for monitor-based implementation of FIFO queue (Table 1)

Task A

```
  i-begin  →  q.enq(x)  →  i-end
```

Task B

```
  i-begin  →  q.enq(y)  →  i-end  →  i-begin  →  q.deq():x  →  i-end
```

- **Continue edge**
- **Serialization edge**
Figure 2: Creating a Reduced Graph to model Instantaneous Execution of Methods

Method \texttt{q.enq(x)}

\begin{itemize}
  \item \texttt{i-begin}
  \item \texttt{q.enq(x)}
  \item \texttt{i-end}
\end{itemize}

Method \texttt{q.enq(y)}

\begin{itemize}
  \item \texttt{i-begin}
  \item \texttt{q.enq(y)}
  \item \texttt{i-end}
\end{itemize}

Method \texttt{q.deq():x}

\begin{itemize}
  \item \texttt{i-begin}
  \item \texttt{q.deq():x}
  \item \texttt{i-end}
\end{itemize}

Method-level Reduced Graph

\begin{itemize}
  \item Method \texttt{q.enq(x)}
  \item Method \texttt{q.enq(y)}
  \item Method \texttt{q.deq():x}
\end{itemize}
Relating Linearizability to the Computation Graph model

- Given a reduced CG, a sufficient condition for linearizability is that the reduced CG is acyclic as in Figure 2.
- This means that if the reduced CG is acyclic, then the underlying execution must be linearizable.
- However, the converse is not necessarily true, as we will see.
Figure 3: example Computation Graph for concurrent implementation of FIFO queue (Table 2)
Figure 4: Reduced method-level graph for Computation Graph in Figure 3

- Example of linearizable execution graph for which reduced method-level graph is cyclic

- Approach to make cycle test more precise for linearizability
  - Decompose concurrent object method into a sequence of “try” steps followed by a sequence of “commit” steps
  - Assume that each “commit” step’s execution does not use any input from any prior “try” step

  ➔ Reduced graph can just reduce the “commit” steps to a single node instead of reducing the entire method to a single node