COMP 322: Fundamentals of Parallel Programming

Lecture 22: Actors (contd), Linearizability of Concurrent Objects

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
Worksheet #21: Interaction between finish and actors

What would happen if the end-finish operation from slide 16 was moved from line 13 to line 11 as shown below?

1. finish {
2.     int numThreads = 4;
3.     int numberOfHops = 10;
4.     ThreadRingActor[] ring = new ThreadRingActor[numThreads];
5.     for(int i=numThreads-1;i>=0; i--) {
6.         ring[i] = new ThreadRingActor(i);
7.         ring[i].start();
8.         if (i < numThreads - 1) {
9.             ring[i].nextActor(ring[i + 1]);
10.        }
11.    } // finish
12.    ring[numThreads-1].nextActor(ring[0]);
13.    ring[0].send(numberOfHops);

Deadlock: the end-finish operation in line 11 waits for all the actors created in line 7 to terminate, but the actors are waiting for the message sequence initiated in line 13 before they call exit()
Recap of Monitors and Actors

Monitors:
- A monitor is a passive object containing local variables (private data) and methods that operate on local data (monitor regions)
- Only one task can be active in a monitor at a time, executing some monitor region

Actors:
- An actor has mutable local state, a process() method to manipulate local state, and a thread of control to process incoming messages
- An actor may process messages, send messages, change local state, and create new actors
The Actor Model: Fundamentals

- An actor may:
  - process messages
  - send messages
  - change local state
  - create new actors
Actors - Simulating synchronous replies

• Actors are inherently asynchronous

• Synchronous replies require blocking operations --- async await can help

```java
class CountMessage {
    ... ddf = new DataDrivenFuture();
    int localCount = 0;

    static int getAndIncrement(
        CounterActor counterActor) {

        ... msg = new CountMessage();
        counterActor.send(msg);
        // use ddf to wait for response
        // THREAD-BLOCKING
        finish { async await(msg.ddf) { }}
        // return count from the message
        return msg.localCount;
    }
}

class CounterActor extends Actor {
    int counter = 0;
    void process(Object m) {
        if (m instanceof CountMessage){
            CountMessage msg = ... counter++;
            msg.localCount = counter;
            msg.ddf.put(true);
        } ... } }
```
Synchronous Reply using Async-Await

1. class SynchronousReplyActor1 extends Actor {
2.   void process(Message msg) {
3.     if (msg instanceof Ping) {
4.         finish {
5.             DataDrivenFuture<T> ddf = new DataDrivenFuture<T>();
6.             otherActor.send(ddf);
7.             async await(ddf) {
8.                 T synchronousReply = ddf.get();
9.                 // do some processing with synchronous reply
10.             }
11.         }
12.     } else if (msg instanceof ...) { ... } }
}
Actors – Global Consensus

- Global consensus is simple with barriers/phasers but can be complex with actors e.g.,
  - First send message from master actor to participant actors signaling intention
  - Wait for all participants to reply they are ready. Participants start ignoring messages sent to them apart from the master
  - Once master confirms all participants are ready, master sends the request to each participant and waits for reply from each
  - Master notifies participants that consensus has been reached, everyone can go back to normal functioning
Parallelizing Actors in HJ

• Two techniques:
  – Use finish construct to wrap asyncs in message processing body
    • Finish ensures all spawned asyncs complete before next message returning from process()
  – Allow escaping asyncs inside process() method
    • *WAIT!* Won't escaping asyncs violate the one-message-at-a-time rule in actors
    • Solution: Use pause and resume
Actors: pause and resume

- Paused state: actor will not process subsequent messages until it is resumed
- Pause an actor before returning from message processing body with escaping asyncs
- Resume actor when it is safe to process the next message
- Akin to Java’s wait/notify operations with locks
Synchronous Reply using Pause/Resume

1. class SynchronousReplyActor2 extends Actor {
2.     void process(Message msg) {
3.         if (msg instanceof Ping) {
4.             DataDrivenFuture<T> ddf = new DataDrivenFuture<T>();
5.             otherActor.send(ddf);
6.             async await(ddf) { // this async processes synchronous reply
7.                 T synchronousReply = ddf.get();
8.                 // do some processing with synchronous reply
9.                 resume(); // allow actor to process next message
10.             }
11.             pause(); // when paused, the actor doesn't process messages
12.         } else if (msg instanceof ...) { ... } }
}
Other uses of hybrid actor+task parallelism

- Can use finish to detect actor termination
- Event-driven tasks
- Stateless Actors
  - If an actor has no state, it can process multiple messages in parallelism
- Pipeline Parallelism
  - Actors represent pipeline stages
  - Use tasks to balance pipeline by parallelizing slower stages
Concurrent Objects

• A concurrent object is an object that can correctly handle methods invoked in parallel by different tasks or threads
  — Originated as monitors
  — Also 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 as in monitors (e.g., enclose each method in an object-based isolated statement) or concurrent (e.g., ConcurrentHashMap, ConcurrentLinkedQueue and CopyOnWriteArraySet)

• A desirable goal is to develop implementations that are concurrent while being as close to the semantics of the serial version as possible
Canonical Example of a Concurrent Object

• 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

- 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.
- A concurrent object is linearizable if all its executions are linearizable.
Example 1

```
q.enq(x)
```

Source: http://www.elsevierdirect.com/companions/9780123705914/Lecture%20Slides/03~Chapter_03.ppt
Example 1 (contd)

$q.enq(x)\quad q.enq(y)$

time

Source: http://www.elsevierdirect.com/companions/9780123705914/Lecture%20Slides/03~Chapter_03.ppt
Example 1 (contd)

\[ q.\text{enq}(x) \rightarrow q.\text{enq}(y) \rightarrow q.\text{deq}():x \]

time

Source: http://www.elsevierdirect.com/companions/9780123705914/Lecture%20Slides/03~Chapter_03.ppt
Example 1 (contd)

\[
\begin{align*}
&\text{q.enq(x)} \\
&\text{q.enq(y)} \\
&\text{q.deq():x} \\
&\text{q.deq(y)} \\
\end{align*}
\]

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)
Example 1 (contd)

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

not linearizable

Source: http://www.elsevierdirect.com/companions/9780123705914/Lecture%20Slides/03~Chapter_03.ppt
Example 3

Is this execution linearizable? How many possible linearizations does it have?

\[
\text{Example 3}
\]

Is this execution linearizable? How many possible linearizations does it have?
Example 4: 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>Invoke q.enq(y)</td>
</tr>
<tr>
<td>2</td>
<td>Work on q.enq(x)</td>
<td>Work on q.enq(y)</td>
</tr>
<tr>
<td>3</td>
<td>Return from q.enq(x)</td>
<td>Return from q.enq(y)</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>Invoke q.deq()</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>Return x from q.deq()</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Yes! Equivalent to “q.enq(x) ; q.enq(y) ; q.deq():x”
Example 5: 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”
Example 5: 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></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>Invoke q.deq()</td>
</tr>
<tr>
<td>5</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”
Example 6: yet another 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></td>
</tr>
<tr>
<td>1</td>
<td>Return from <code>q.enq(x)</code></td>
<td>Invoke <code>q.enq(y)</code></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>Work on <code>q.enq(y)</code></td>
</tr>
<tr>
<td>3</td>
<td>Invoke <code>q.deq()</code></td>
<td>Return from <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 <code>y</code> from <code>q.deq()</code></td>
<td></td>
</tr>
</tbody>
</table>

Let’s figure it out in Worksheet 22!
Linearizability of Concurrent Objects
(Summary)

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
Worksheet #22:
Linearizability of method calls on a concurrent object

Is this a linearizable execution for a FIFO queue, q?

<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>Return from q.enq(x)</td>
<td>Work on q.enq(y)</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>Return from q.enq(y)</td>
</tr>
<tr>
<td>3</td>
<td>Invoke q.deq()</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Work on q.deq()</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Return y from q.deq()</td>
<td></td>
</tr>
</tbody>
</table>