Solution to Worksheet #24:
Ideal Parallelism in Actor Pipeline

Consider a three-stage pipeline of actors set up so that \( P_0.\text{nextStage} = P_1 \), \( P_1.\text{nextStage} = P_2 \), and \( P_2.\text{nextStage} = \text{null} \). The process() method for each actor is shown below. Assume that 100 non-null messages are sent to actor \( P_0 \) after all three actors are started, followed by a null message. What will the total WORK and CPL be for this execution? Recall that each actor has a sequential thread.

Solution: WORK = 300, CPL = 102

```
protected void process(final Object msg) {
    if (msg == null) {
        exit();
    } else {
        doWork(1); // unit work
    }
    if (nextStage != null) {
        nextStage.send(msg);
    }
}
```

Input sequence
\[ \cdots \; d_9d_8d_7d_6d_5d_4d_3d_2d_1d_0 \]

\( P_0 \) \( P_1 \) \( P_2 \)
Concurrent Objects

- A concurrent object is an object that can correctly handle methods invoked in parallel by different tasks or threads
  — 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 serialized (e.g., enclose each method in an actor or an object-based isolated statement) or can be concurrent (e.g., by using read-write modes in object-based isolation)

- A desirable goal is to develop implementations that are concurrent while being as close to the semantics of the serial version as possible

- Examples of concurrent objects: atomic variables, shared buffers, concurrent lists, concurrent hashmaps, ...
Example #1 of a Concurrent Object: Implementing an Unbounded Buffer using Actors

[Diagram showing a Master node connected to multiple Producer and Consumer nodes, with an unbounded buffer in the middle containing the numbers 2, 5, 7, 3, 6, 7]
Unbounded Buffer Actor Interaction Diagram

1. Determine new item required
2. Request Data from an idle producer
3. Produce Data
4. Send newly produced data
5. Store data item in buffer

Producer-P

Master
Unbounded Buffer Actor Interaction Diagram (contd)

1. Retrieve data item from buffer
2. Send Data to an idle consumer
3. Notify master of becoming idle
4. Store idle consumer locally for future use

Master

Consumer-C

3. Consume Data
Actor Responsibilities

• Master Actor
  — Receives Data Items from the producers
  — Stores data items in its unbounded buffer
  — Send data items to idle consumers
  — Receives notifications when consumers are idle

• Producer Actor
  — Receives requests to produce items
  — Sends data items to the Master

• Consumer Actor
  — Receives requests from Master to consume an item
  — Sends notification to Master when it becomes idle
Example #2 of a Concurrent Object: Implementing an Bounded Buffer using Actors

- Assume that $B > P$ to allow for the case where producer messages may be in flight
Bounded Buffer Actor Interaction Diagram

1. Space available in buffer
2. Select an idle producer
3. Request Data from selected producer
4. Produce Data
5. Send newly produced data
6. Store producer locally as idle

Processor-P

Master
Bounded Buffer Actor Interaction Diagram (contd)

1. Select an idle consumer
2. Retrieve data item from buffer
3. Select a producer
4. Request Data from selected producer (previous slide)

3. Send Data to an selected consumer
4. Consume Data
5. Notify master of becoming idle

4. Store idle consumer locally for future selection
Actor Responsibilities

- **Master Actor**
  - Sends requests to an *idle* producer when there is space in buffer
  - Receives Data Items from the producers
  - Stores data items in its bounded buffer
  - Send data items to *idle* consumers, thus making space in buffer
  - Receives notifications when consumers are idle

- **Producer Actor**
  - Receives requests from Master to produce items
  - Sends data items to the Master indirectly notifying it is now idle

- **Consumer Actor**
  - Receives requests from Master to consume an item
  - Sends notification to Master when it becomes idle
Correctness 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 implies a strict temporal order
  - Concurrent implies an ambiguous temporal order
Describing the concurrent via the sequential

```
<table>
<thead>
<tr>
<th>Task T1</th>
<th>Task T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>q.deq():x</td>
<td>deq() returns x</td>
</tr>
<tr>
<td>isolated-wait/begin</td>
<td>isolated-end</td>
</tr>
<tr>
<td>q.enq(x)</td>
<td>enq()</td>
</tr>
<tr>
<td>isolated-wait/begin</td>
<td>isolated-end</td>
</tr>
</tbody>
</table>
```

"Linearizability" -- sequence of enq() and deq() calls is consistent with sequential execution

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)
Informal definition of Linearizability

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

- An execution is linearizable if we can choose one set of instantaneous points that is consistent with a sequential execution in which methods are executed at those points.
  - It’s okay if some other set of instantaneous points is not linearizable.

- A concurrent object is linearizable if all its executions are linearizable.
  - Linearizability is a “black box” test based on the object’s behavior, not its internals.
Example 1

Task T1

$q.\text{enq}(x)$

time

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 1 (contd)

Task T1

\[ q \text{.enq}(x) \]

\[ q \text{.enq}(y) \]

Task T2

\[ q \text{.enq}(x) \]

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

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

Task T1

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

Task T2

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)

Task T1

1. q.enq(x)
2. q.enq(y)

Task T2

3. q.deq():x
4. q.deq():y

linearizable

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

Task T1

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

Task T2

- `q.enq(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)
Example 3

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
q.enq(x)
```
Example 4: execution of an isolated 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>Invoke $q.enq(y)$</td>
</tr>
<tr>
<td>1</td>
<td>Work on $q.enq(x)$</td>
<td>Work on $q.enq(y)$</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>
Example 5: execution of a concurrent implementation of a 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>
</tbody>
</table>
**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