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

- **Maurice Herlihy and Nir Shavit. The art of multiprocessor programming. Morgan Kaufmann, 2008.**
  - **Optional text for COMP 322**
  - **Chapter 3 slides extracted from http://www.elsevierdirect.com/companion.jsp?ISBN=9780123705914**
- **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:

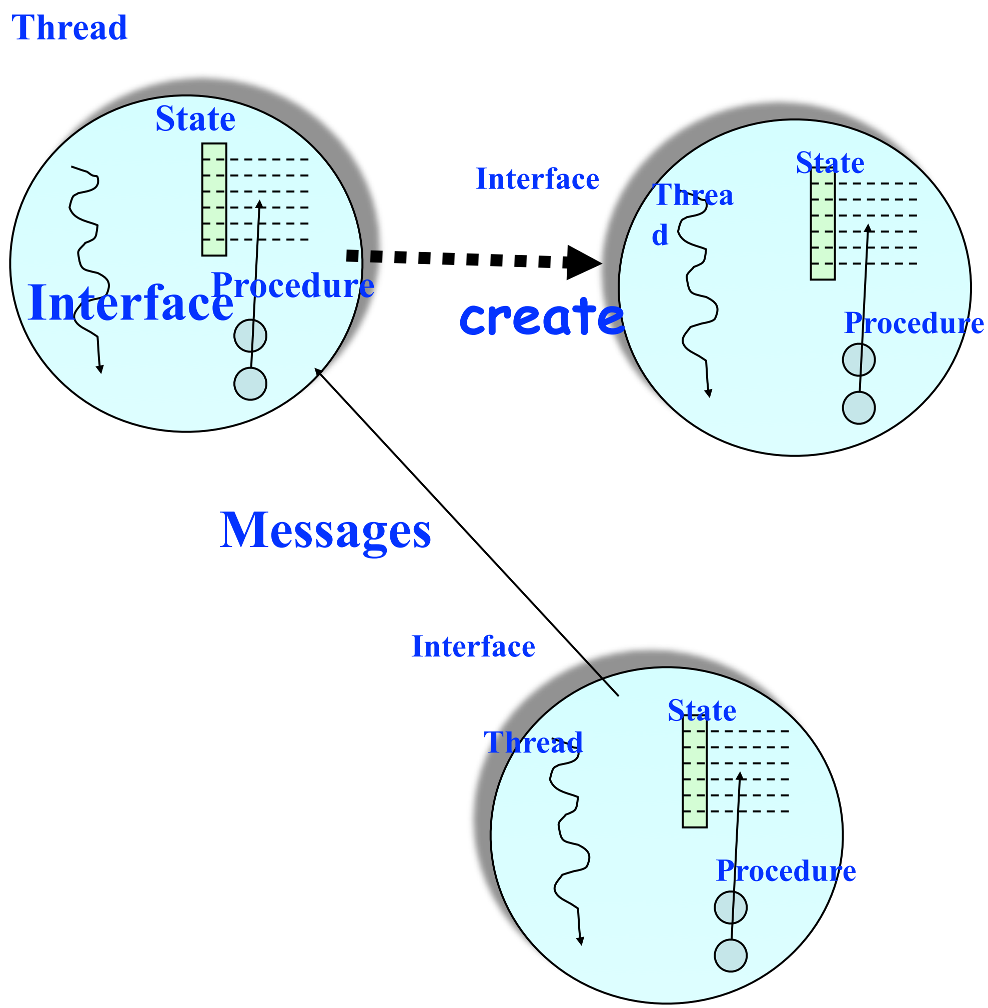
- A monitor is a passive object containing local variables (private data) and methods that operate on local data (monitor regions)
- Only one task can be active in a monitor at a time, executing some monitor region

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

```
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

```
class SynchronousReplyActor1 extends Actor {
void process(Message msg) {
   if (msg instanceof Ping) {
      finish {
         DataDrivenFuture<T> ddf = new DataDrivenFuture<T>();
         otherActor.send(ddf);
         async await(ddf) {
            T synchronousReply = ddf.get();
            // do some processing with synchronous reply
         }
      }
   } else if (msg instanceof ...) { ... } } }
```

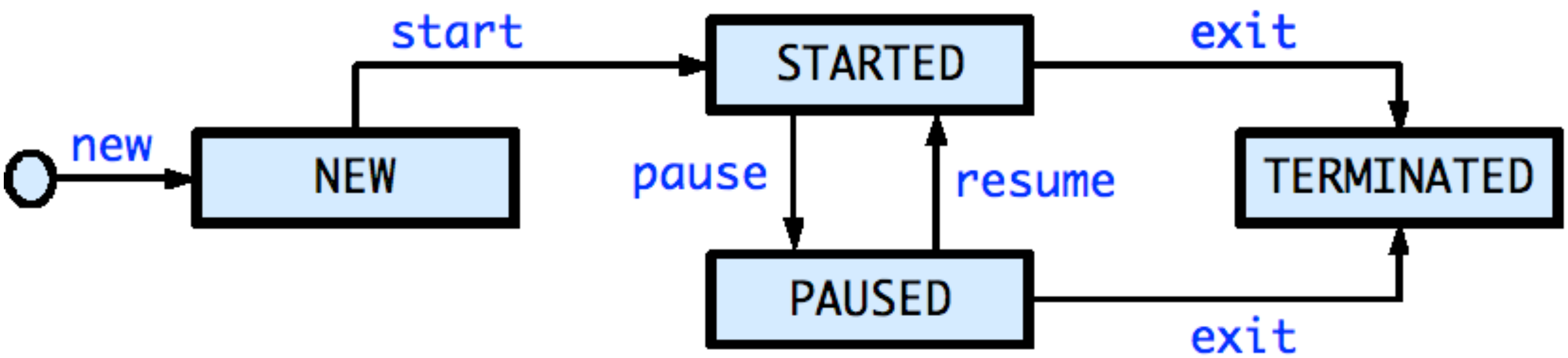
Actors – Global Consensus

- 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

```
class SynchronousReplyActor2 extends Actor {
  void process(Message msg) {
    if (msg instanceof Ping) {
      DataDrivenFuture<T> ddf = new DataDrivenFuture<T>();
      otherActor.send(ddf);
      async await(ddf) { // this async processes synchronous reply
         T synchronousReply = ddf.get();
         // do some processing with synchronous reply
         resume(); // allow actor to process next message
      }
      pause(); // when paused, the actor doesn't process messages
    } 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

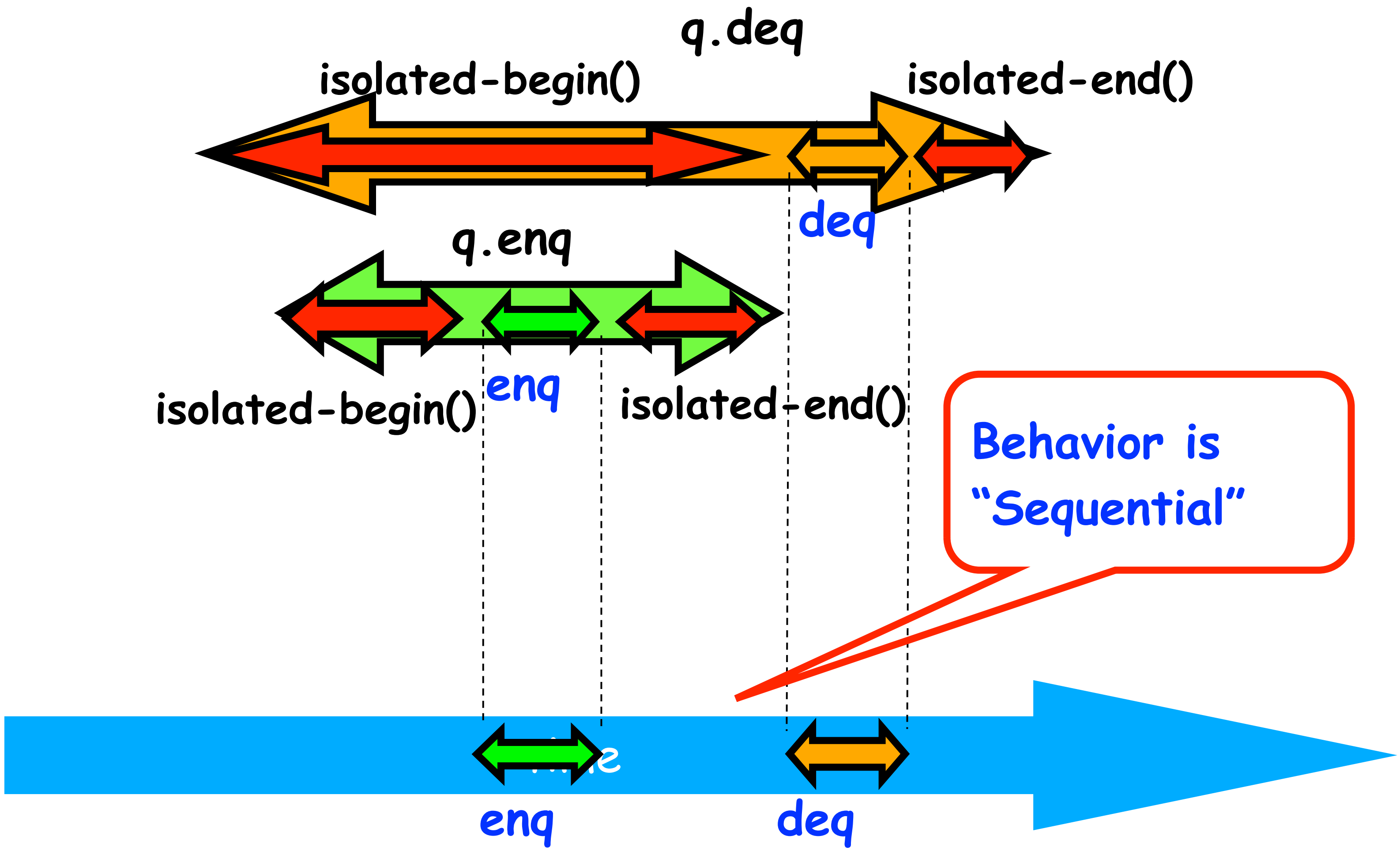
Concurrent Objects

- 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

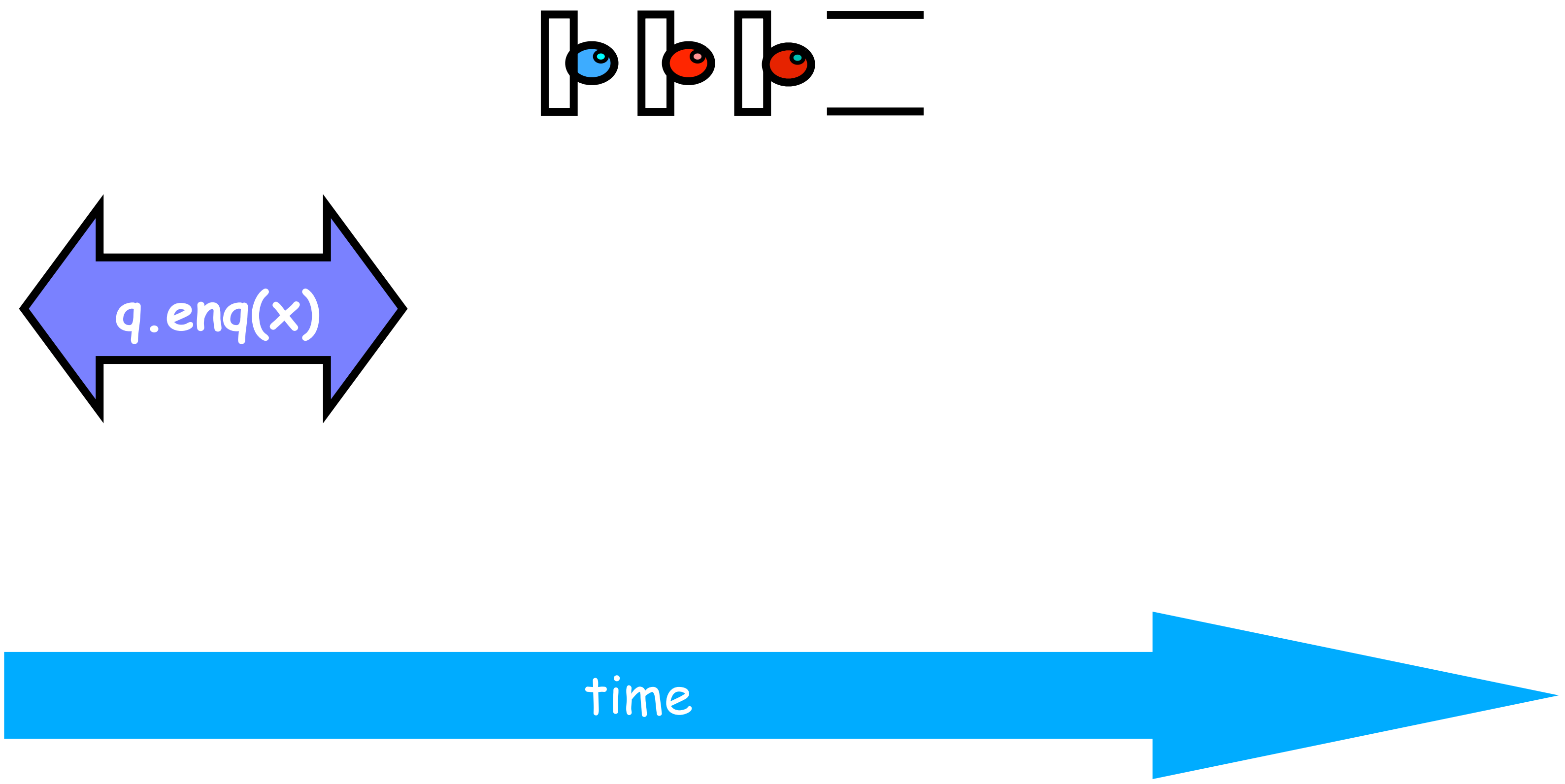


Source: http://www.elsevierdirect.com/companions/9780123705914/Lecture%20Slides/03~Chapter_03.ppt

Informal definition of Linearizability

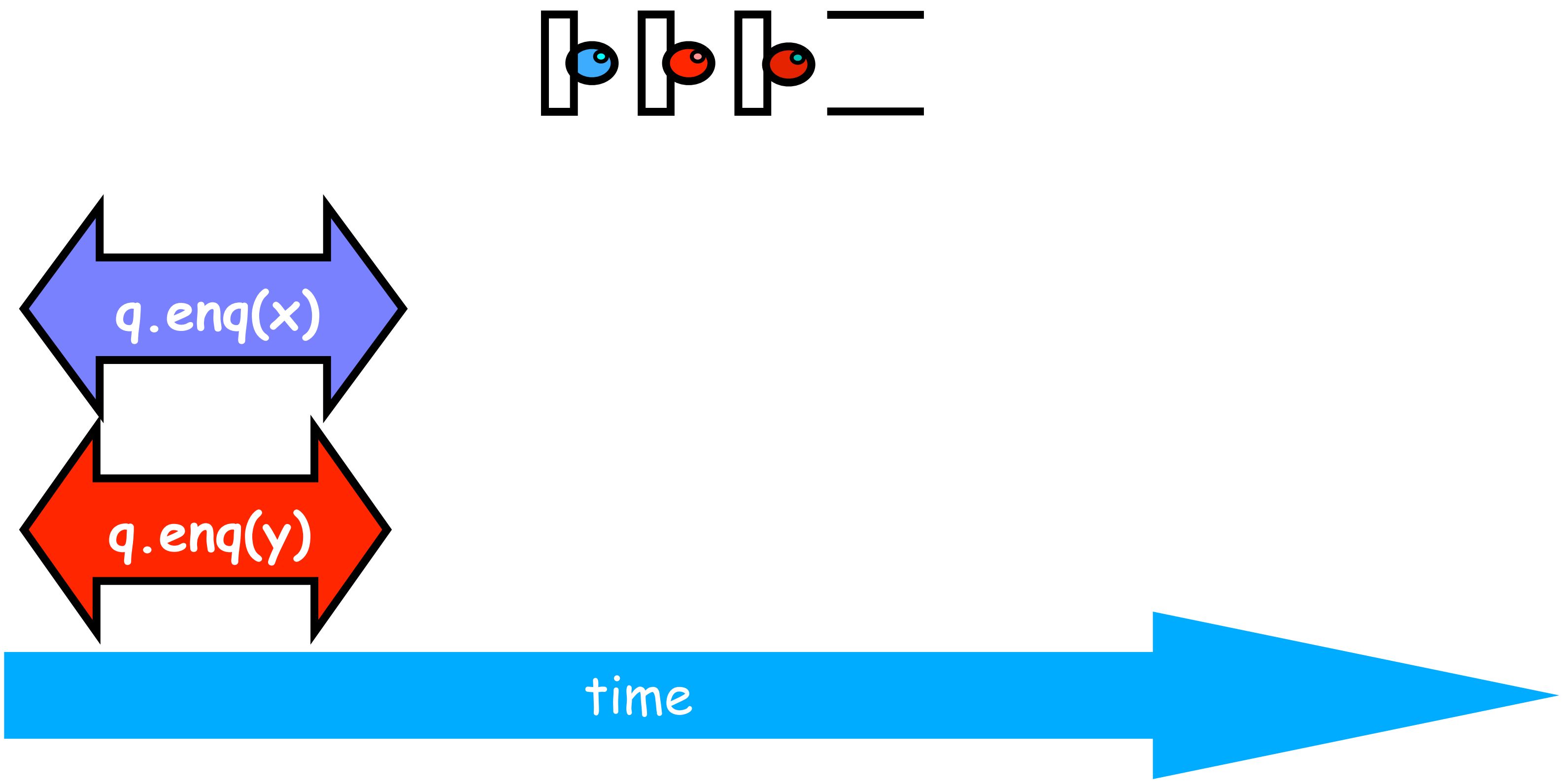
- Assume that each method call takes effect "instantaneously" at some *distinct* point in time between its invocation and return.
- An execution is linearizable if we can choose instantaneous points that are consistent with a sequential execution in which methods are executed at those points
- A concurrent object is linearizable if all its executions are linearizable.

Example 1



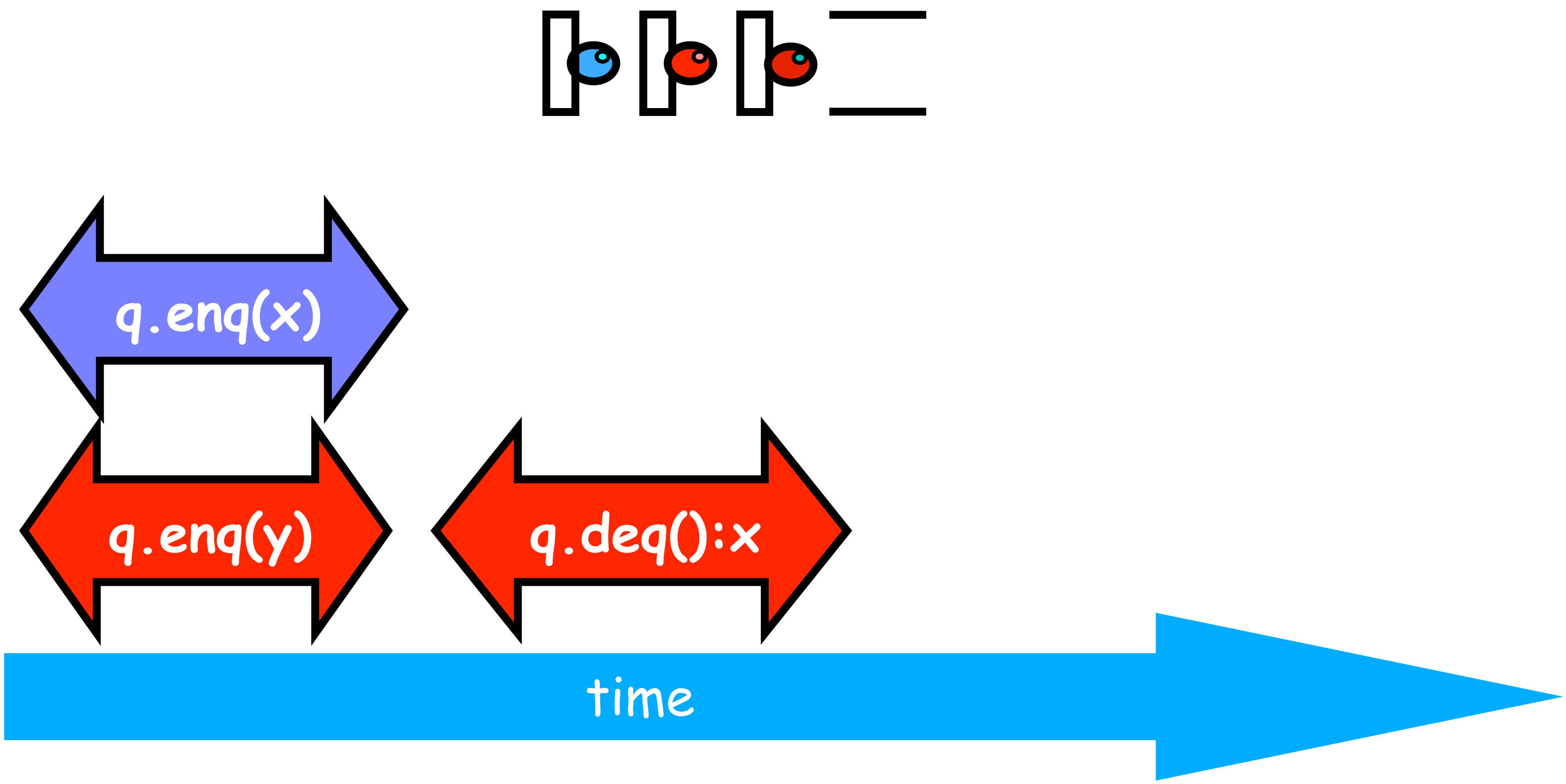
Source: http://www.elsevierdirect.com/companions/9780123705914/Lecture%20Slides/03~Chapter_03.ppt

Example 1 (contd)



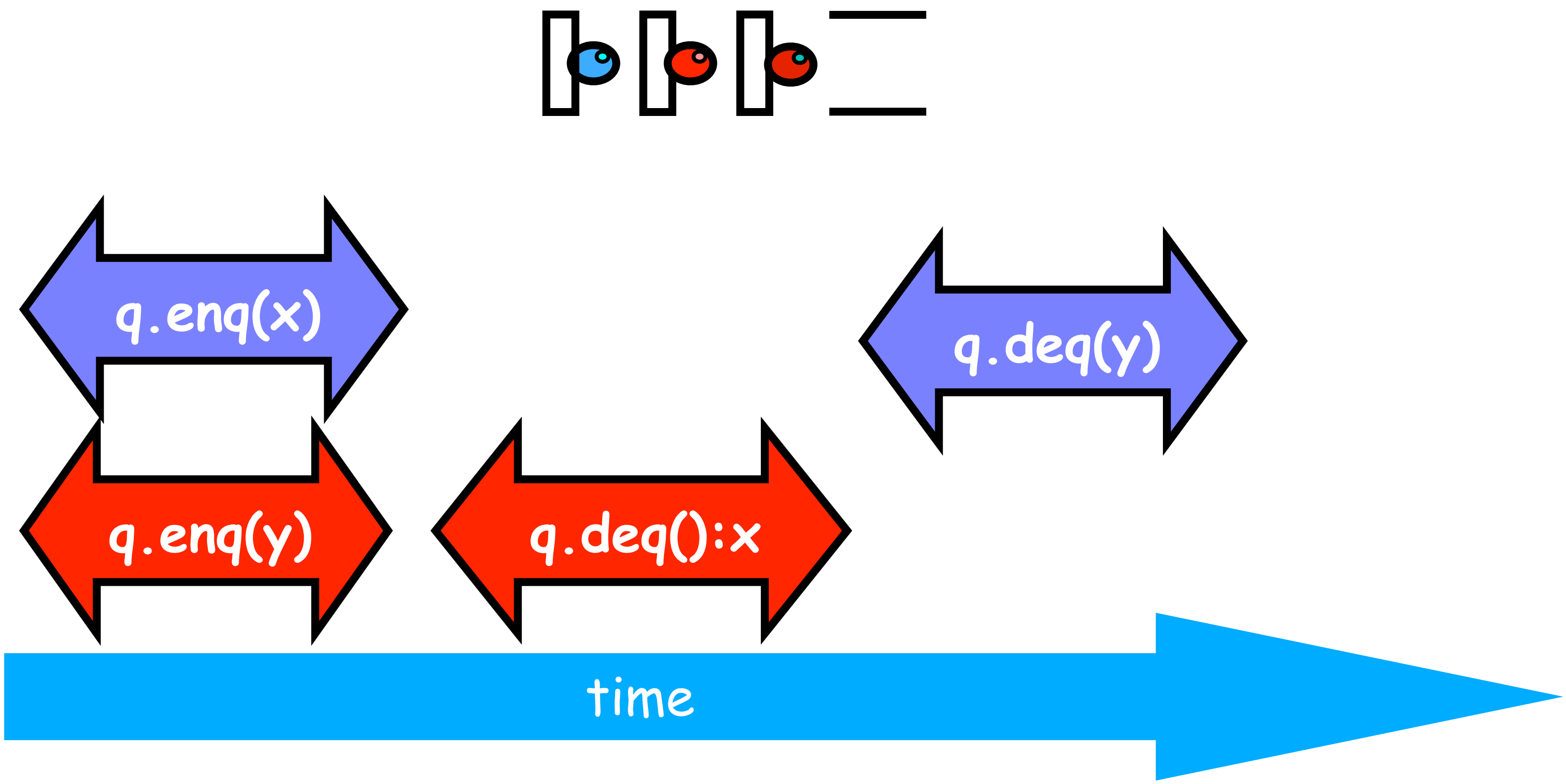
Source: http://www.elsevierdirect.com/companions/9780123705914/Lecture%20Slides/03~Chapter_03.ppt

Example 1 (contd)



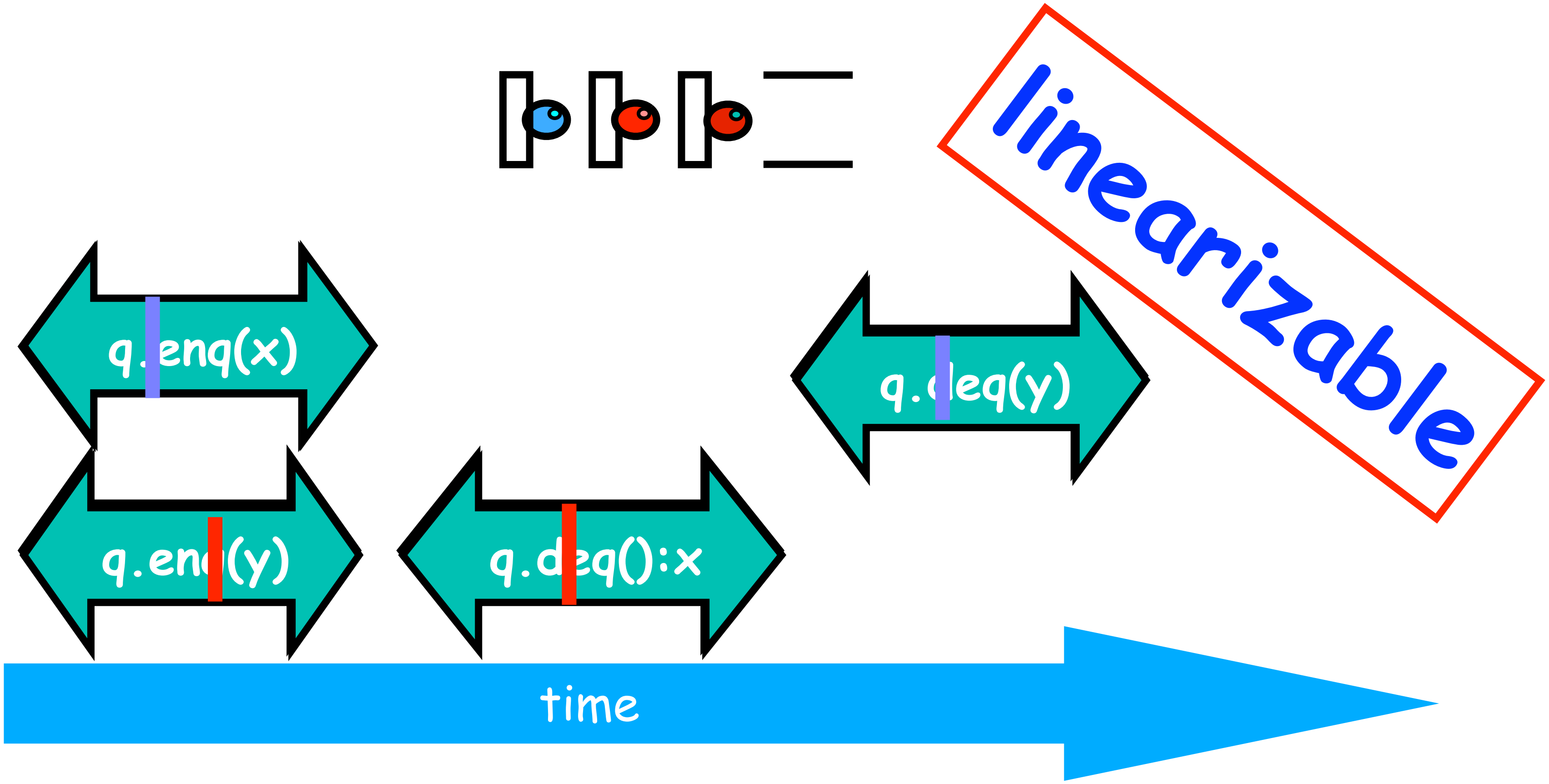
Source: http://www.elsevierdirect.com/companions/9780123705914/Lecture%20Slides/03~Chapter_03.ppt

Example 1 (contd)



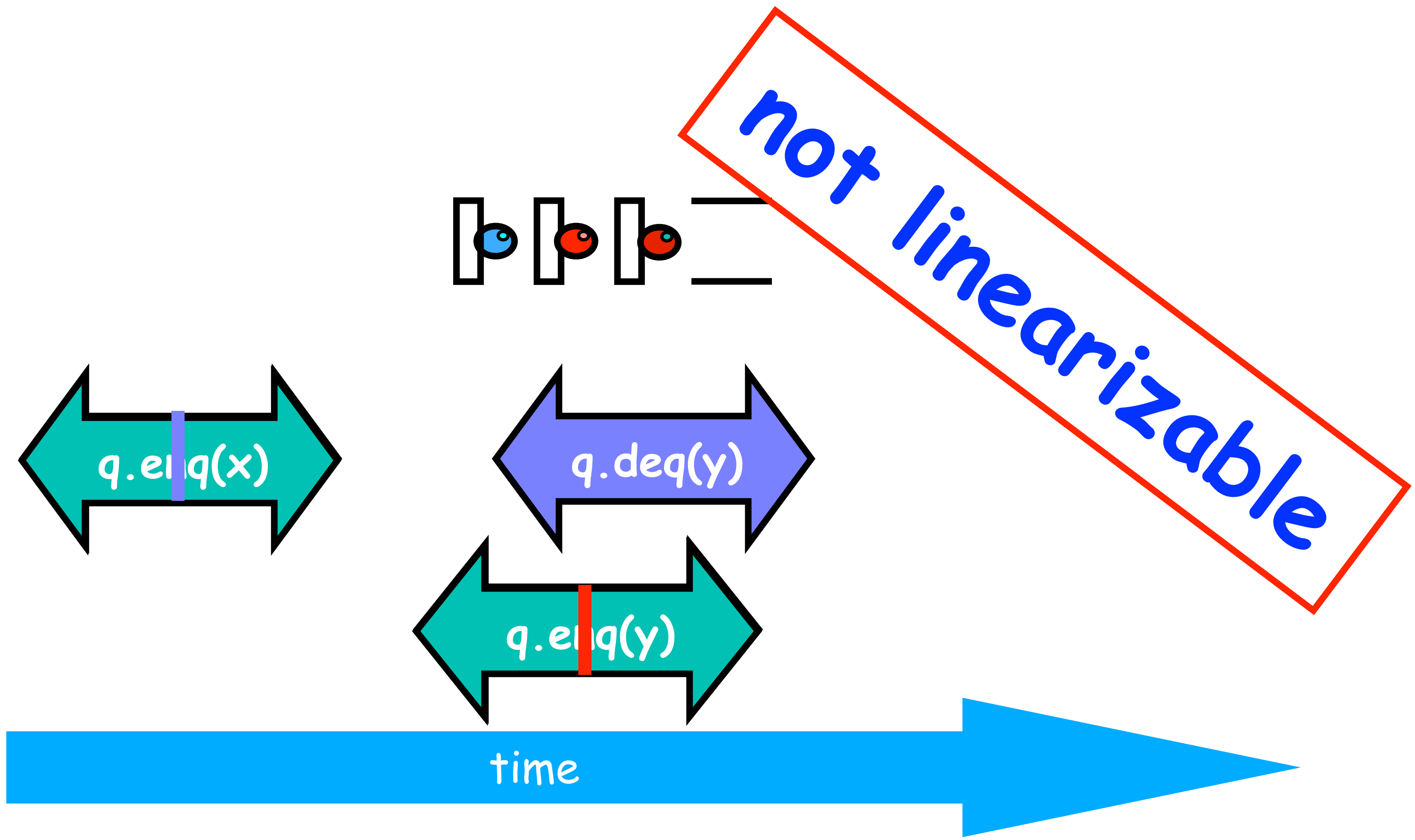
Source: http://www.elsevierdirect.com/companions/9780123705914/Lecture%20Slides/03~Chapter_03.ppt

Example 1 (contd)



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

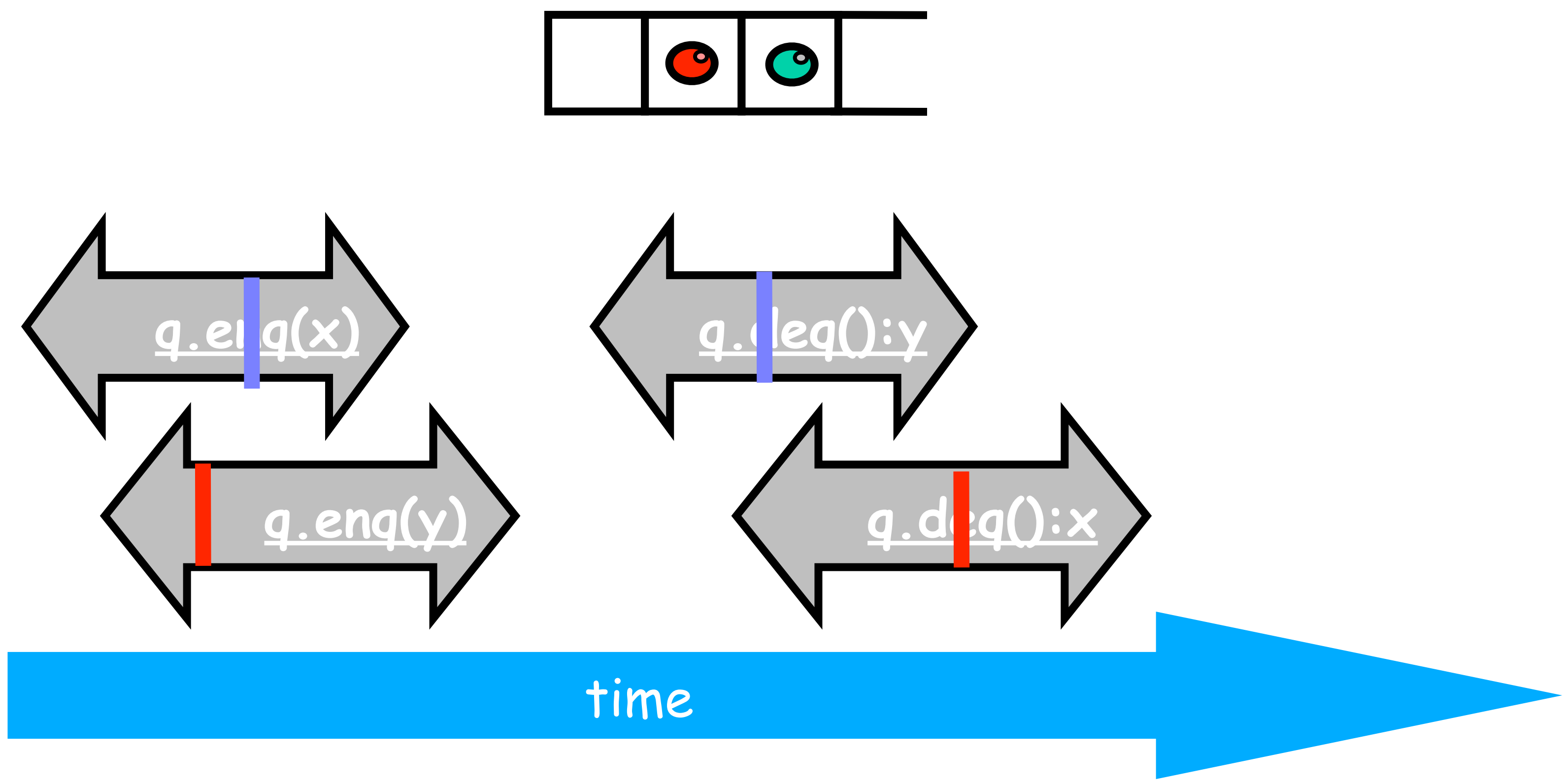
Example 2



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?



Example 4: execution of a monitor-based implementation of FIFO queue q

Is this a linearizable execution?

Time	Task *A*	Task *B*
0	Invoke `q.enq(x)`	
1	Work on `q.enq(x)`	
2	Work on `q.enq(x)`	
3	Return from `q.enq(x)`	
4		Invoke `q.enq(y)`
5		Work on `q.enq(y)`
6		Work on `q.enq(y)`
7		Return from `q.enq(y)`
8		Invoke `q.deq()`
9		Return `x` from `q.deq()`

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?

Time	Task A	Task B
0	Invoke `q.enq(x)`	
1	Work on `q.enq(x)`	Invoke `q.enq(y)`
2	Work on `q.enq(x)`	Return from `q.enq(y)`
3	Return from `q.enq(x)`	
4		Invoke `q.deq()`
5		Return `x` from `q.deq()`

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?

Time	Task A	Task B
0	Invoke `q.enq(x)`	
1	Work on `q.enq(x)`	Invoke `q.enq(y)`
2	Work on `q.enq(x)`	Return from `q.enq(y)`
3	Return from `q.enq(x)`	
4		Invoke `q.deq()`
5		Return `x` from `q.deq()`

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?

Time	Task A	Task B
0	Invoke `q.enq(x)`	
1	Return from `q.enq(x)`	
2		Invoke `q.enq(y)`
3	Invoke `q.deq()`	Work on `q.enq(y)`
4	Work on `q.deq()`	Return from `q.enq(y)`
5	Return y from `q.deq()`	

Let's figure it out in Worksheet 22!

Linearizability of Concurrent Objects (Summary)

<u>Concurrent object</u>

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

<u>Linearizability</u>

- **Assume that each method call takes effect "instantaneously" at some distinct point in time between its invocation and return.**
- **An <u>execution</u> is linearizable if we can choose instantaneous points that are consistent with a sequential execution in which methods are executed at those points**
- **An <u>object</u> is linearizable if all its possible executions are linearizable**

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

Name 1: ______________________ Name 2: ______________________

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

Time	Task A	Task B
0	Invoke `q.enq(x)`	
1	Return from `q.enq(x)`	
2		Invoke `q.enq(y)`
3	Invoke `q.deq()`	Work on `q.enq(y)`
4	Work on `q.deq()`	Return from `q.enq(y)`
5	Return y from `q.deq()`	