CS 181E: Fundamentals of Parallel Programming

Instructor: Vivek Sarkar

Co-Instructor: Ran Libeskind-Hadas

http://www.cs.hmc.edu/courses/2012/fall/cs181e/

Recap of Lecture 9

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

Worksheet #9 solution: Interaction between finish and actors

What would happen if the end-finish operation from slide 29 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()

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

Outline

- Linearizability of Concurrent Executions and Concurrent Objects
- Liveness/progress guarantees
- Optimized Implementations of Isolated

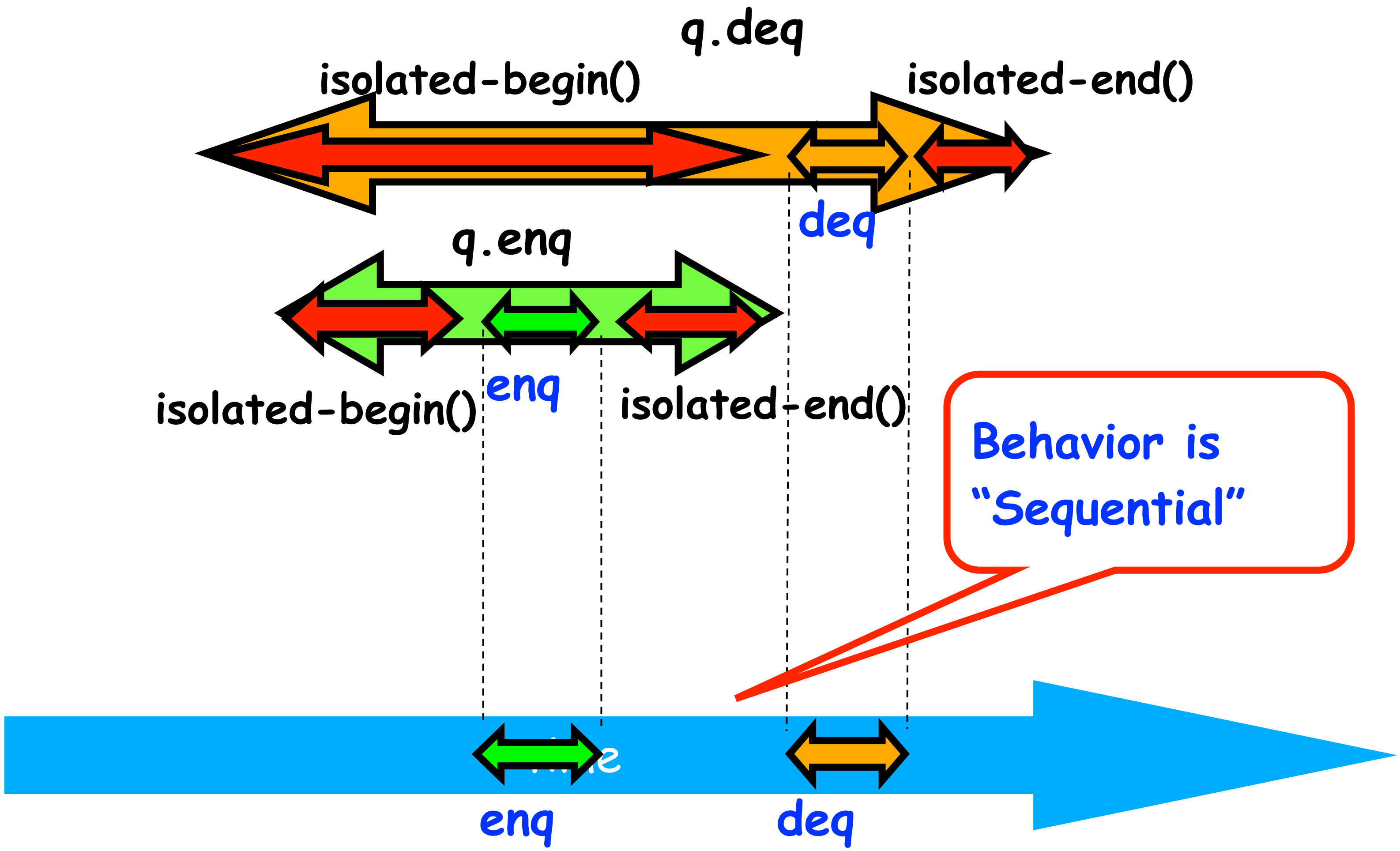
Concurrent Objects

- 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

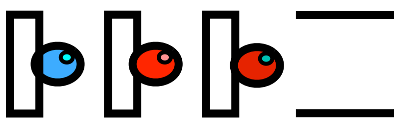


Source: http://www.elsevierdirect.com/companions/9780123705914/Lecture%20Slides/03~Chapter_03.ppt

Informal definition of Linearizability

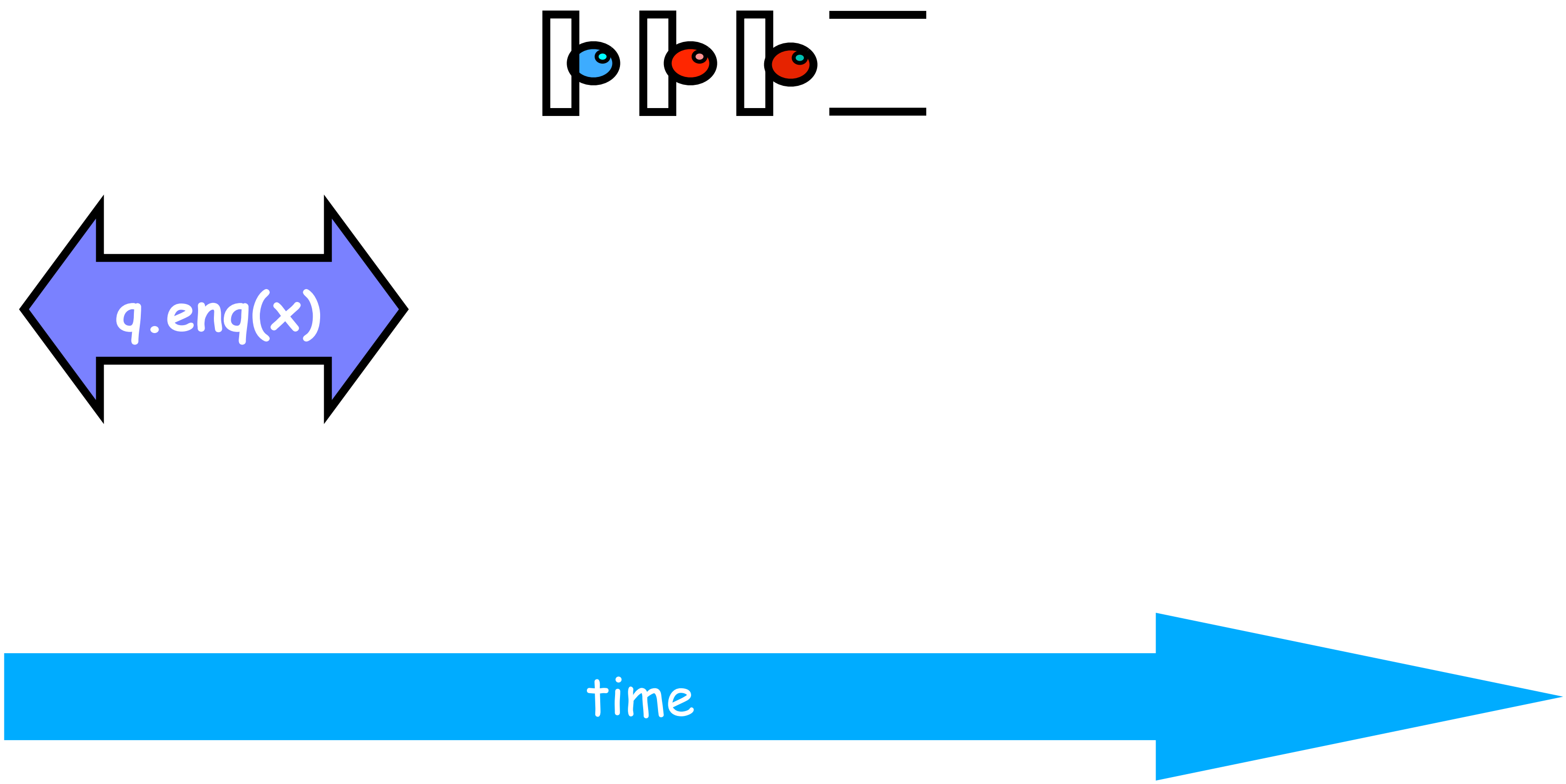
- Assume that each method call takes effect "instantaneously" at some distinct point in time between its invocation and return.
- An execution is linearizable if we can choose instantaneous points that are consistent with a sequential execution in which methods are executed at those points
- A concurrent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

Example 1



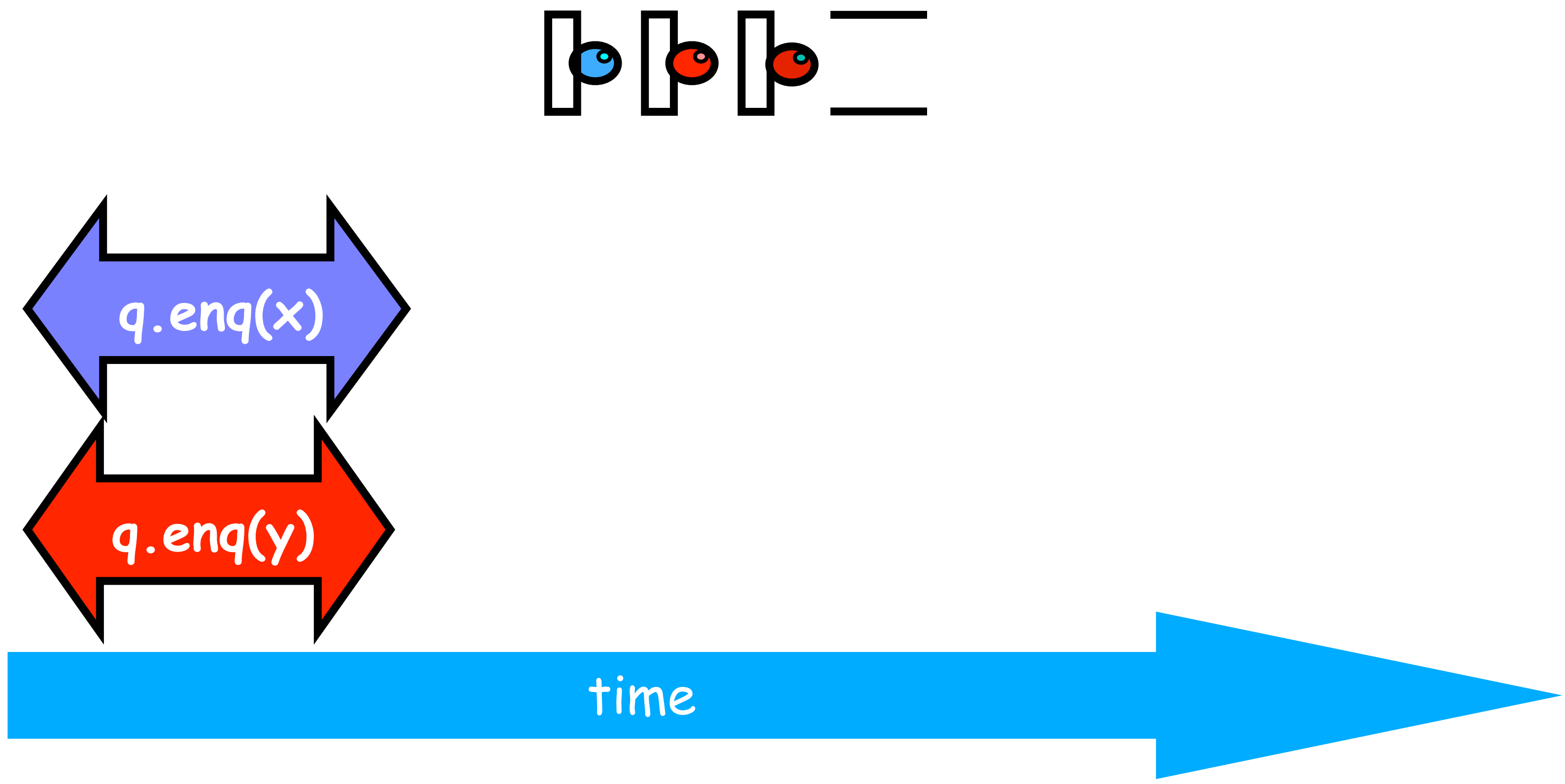
Source: http://www.elsevierdirect.com/companions/9780123705914/Lecture%20Slides/03~Chapter_03.ppt

Example 1 (contd)



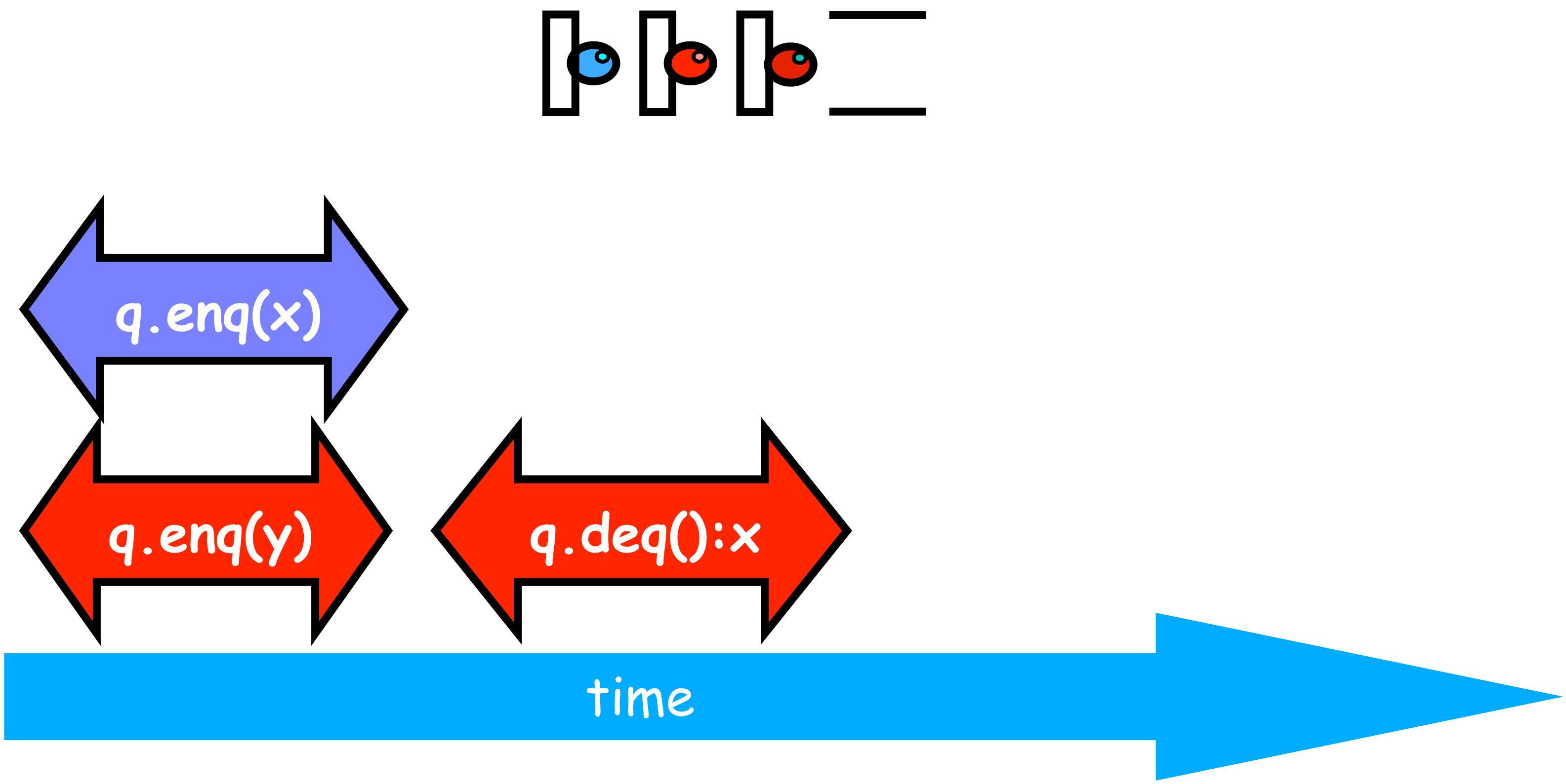
Source: http://www.elsevierdirect.com/companions/9780123705914/Lecture%20Slides/03~Chapter_03.ppt

Example 1 (contd)



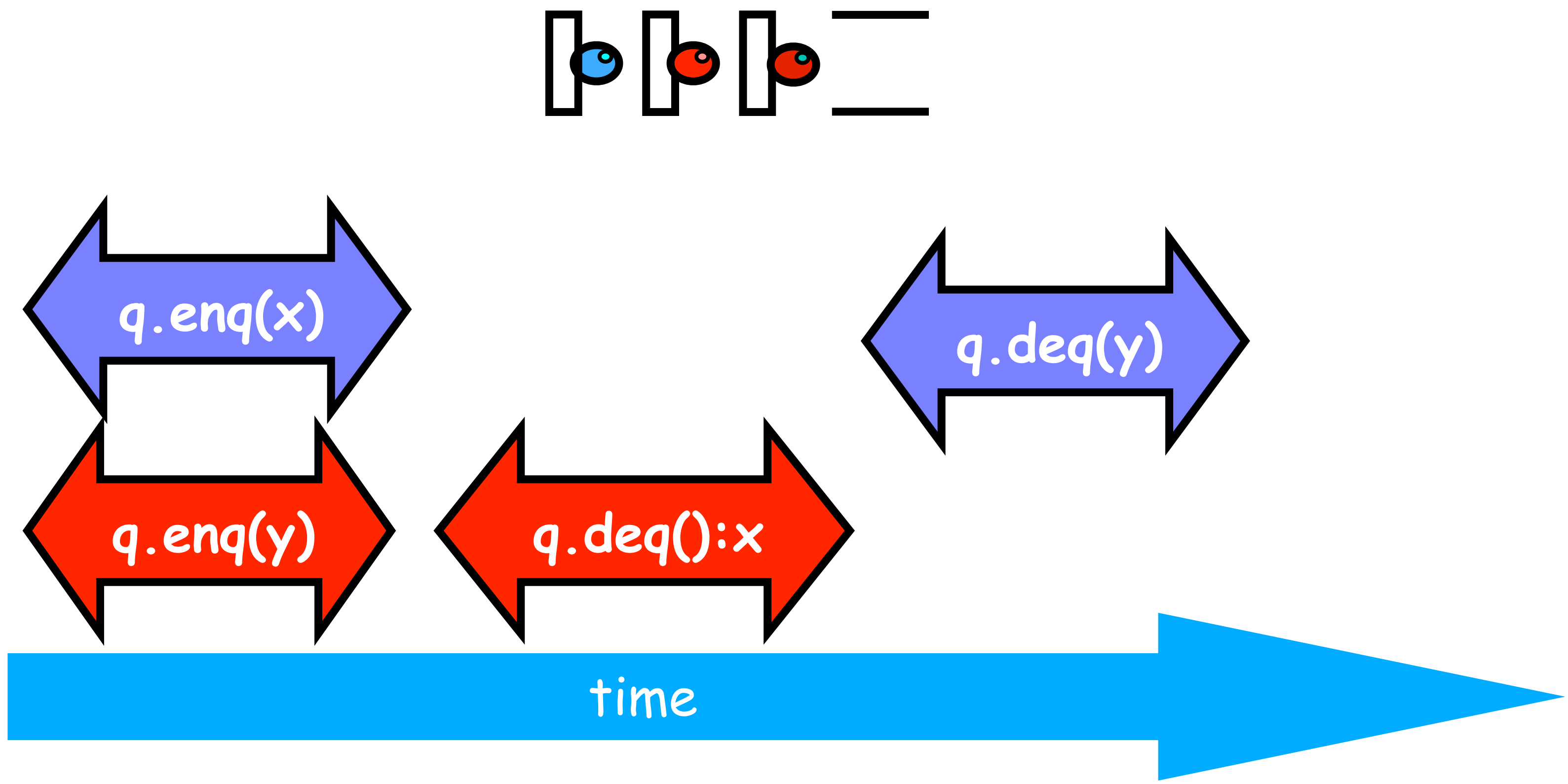
Source: http://www.elsevierdirect.com/companions/9780123705914/Lecture%20Slides/03~Chapter_03.ppt

Example 1 (contd)



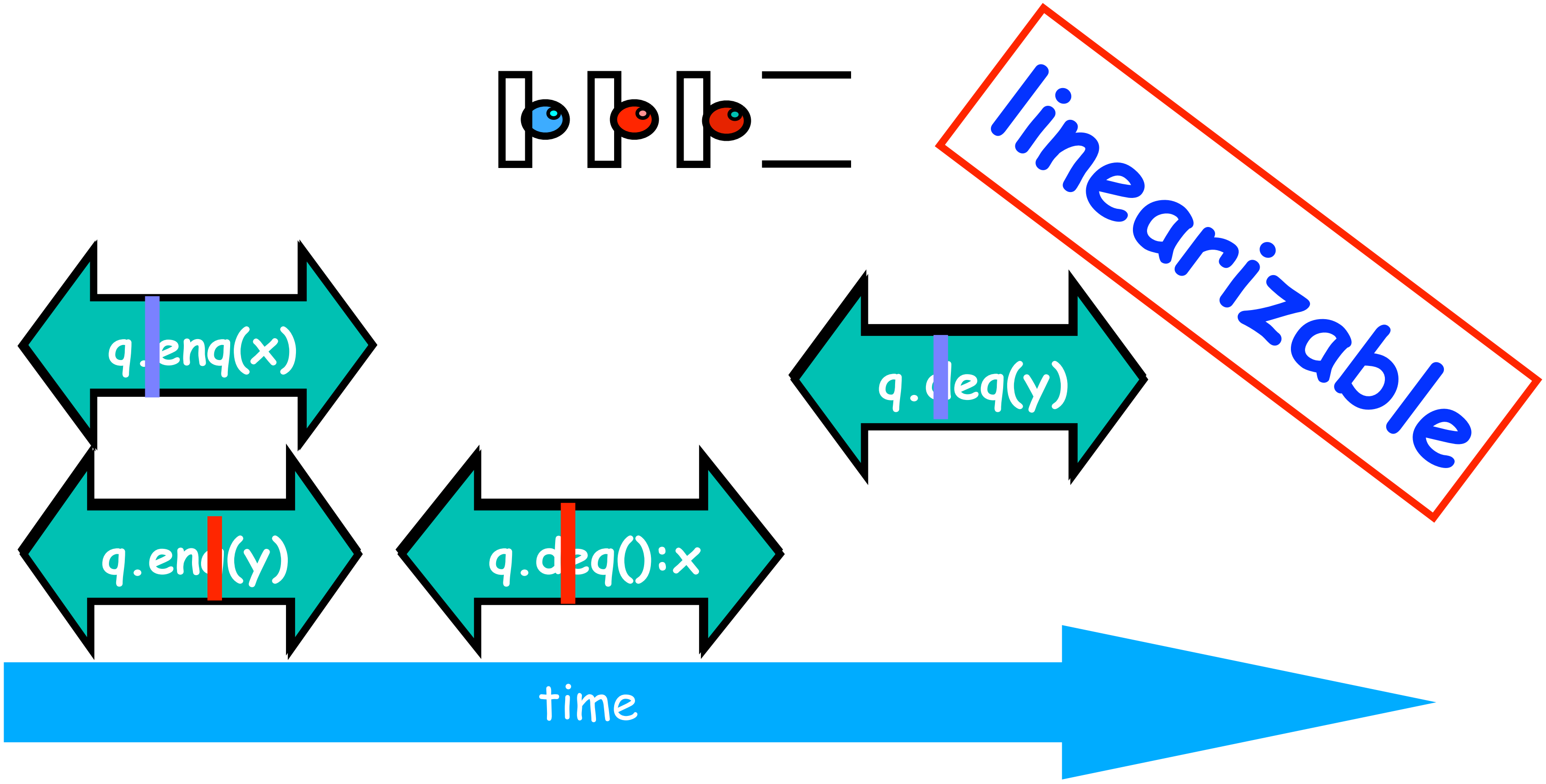
Source: http://www.elsevierdirect.com/companions/9780123705914/Lecture%20Slides/03~Chapter_03.ppt

Example 1 (contd)



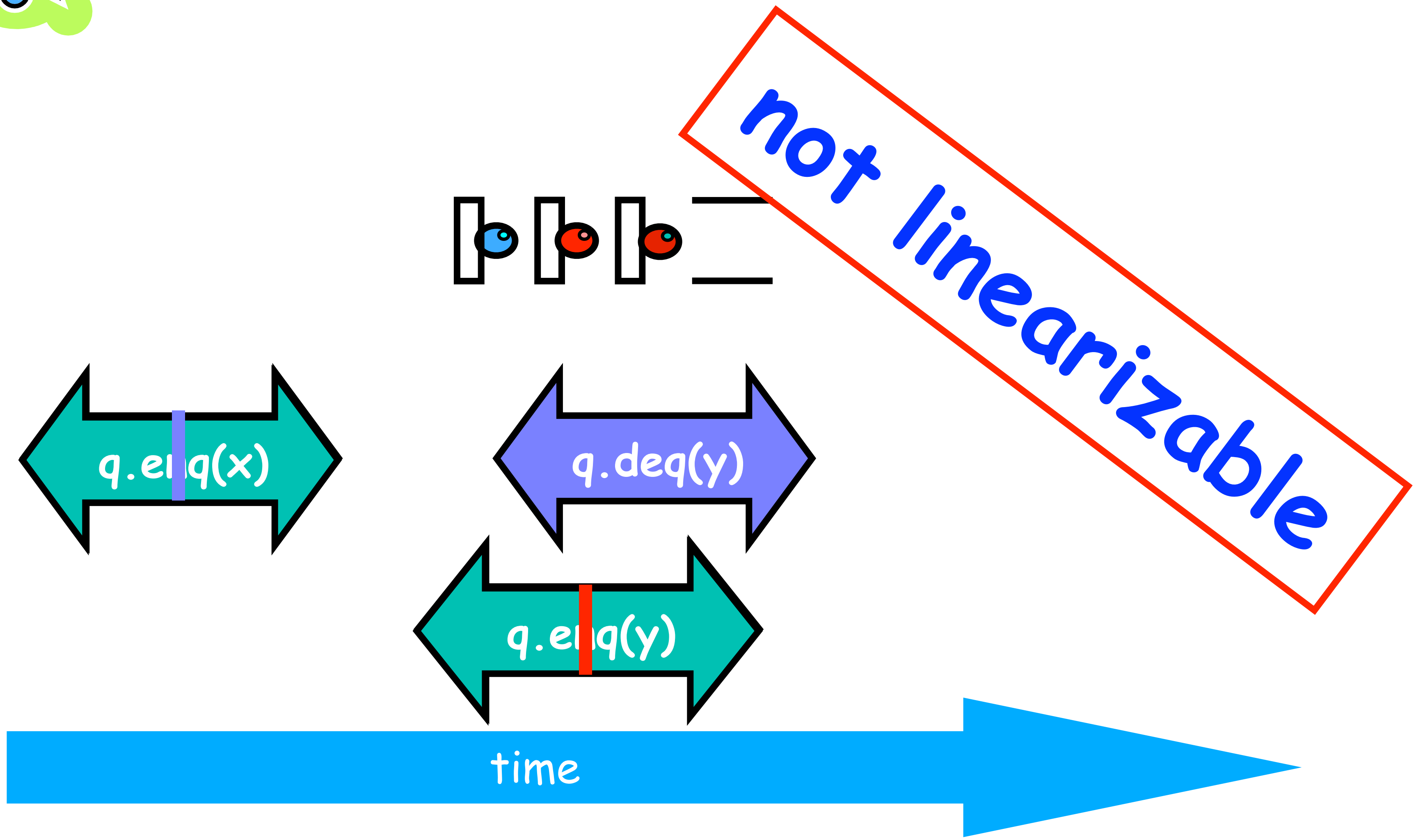
Source: http://www.elsevierdirect.com/companions/9780123705914/Lecture%20Slides/03~Chapter_03.ppt

Example 1 (contd)



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

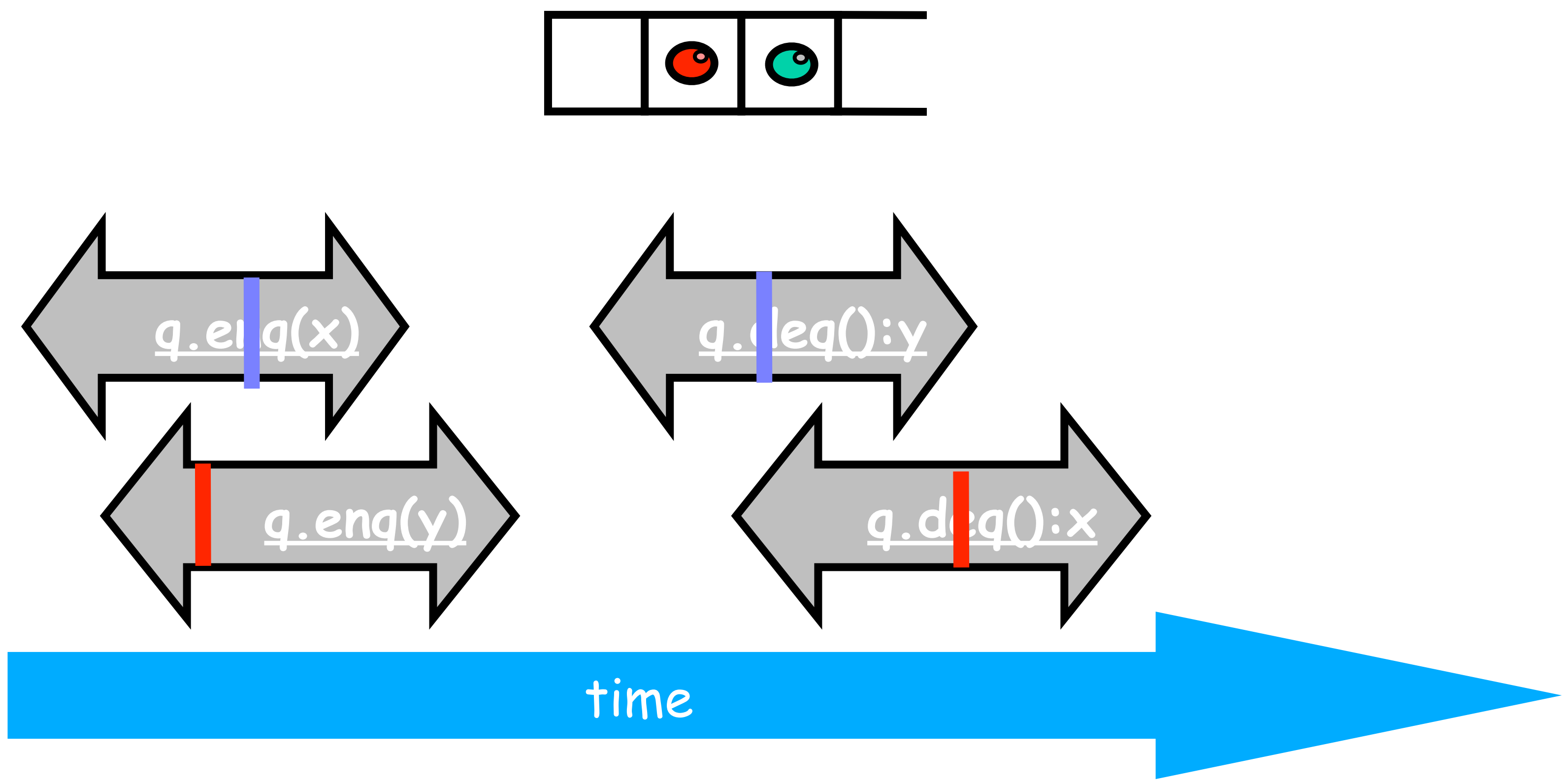
Example 2



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 10!

Linearizability of Concurrent Objects (Summary)

Concurrent object

- A concurrent object is an object that can correctly handle methods invoked in parallel bylin 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

Outline

- Linearizability of Concurrent Executions and Concurrent Objects
- <u>Liveness/progress guarantees</u>
- Optimized Implementations of Isolated

Safety vs. Liveness

- In a concurrent setting, we need to specify both the safety and the liveness properties of an object
- Need a way to define
  - Safety: when an implementation is correct
  - Liveness: the conditions under which it guarantees progress

- Data race freedom is a desirable safety property for most parallel programs
- Linearizability is a desirable safety property for most concurrent objects

Liveness Guarantees

- Liveness = a program's ability to make progress in a timely manner
- Different levels of liveness guarantees (from weaker to stronger)
  - Deadlock freedom
  - Livelock freedom
  - Starvation freedom
  - Bounded wait

Deadlock-Free Parallel Program Executions

- A parallel program execution is deadlock-free if no task's execution remains incomplete due to it being blocked awaiting some condition
- Example of a program with a deadlocking execution

```
DataDrivenFuture left = new DataDrivenFuture();
DataDrivenFuture right = new DataDrivenFuture();
finish {
  async await ( left ) right.put(rightBuilder()); // Task1
  async await ( right ) left.put(leftBuilder()); // Task2
}
```

- In this case, Task1 and Task2 are in a deadlock cycle.
  - Only two constructs can lead to deadlock in HJ: async await, finish + actors, explicit phaser wait (instead of next)
  - There are many mechanisms that can lead to deadlock cycles in other programming models (e.g., locks)

Livelock-Free Parallel Program Executions

- A parallel program execution exhibits livelock if two or more tasks repeat the same interactions without making any progress (special case of nontermination)
- Livelock example:

```
// Task 1
incrToTwo(AtomicInteger ai) {
  // increment ai till it reaches 2
  while (ai.incrementAndGet() < 2);
}
```

```
// Task 2
decrToNegativeTwo(AtomicInteger ai) {
  // decrement ai till it reaches -2
  while (a.decrementAndGet() > -2);
}
```

- Many well-intended approaches to avoid deadlock result in livelock instead
- Any data-race-free HJ program without isolated/atomic-variables/actors is guaranteed to be livelock-free (may be nonterminating in a single task, however)

Terminating Parallel Program Executions

- A parallel program execution is terminating if all sequential tasks in the program terminate
- Example of a nondeterministic data-race-free program with a nonterminating execution

```
p.x = false;
finish {
  async { // S1
     boolean b = false; do { isolated b = p.x; } while (! b);
  }
  isolated p.x = true; // S2
} // finish
```

- Some executions of this program may be terminating, and some not
- Cannot assume in general that statement S2 will ever get a chance to execute if async S1 is nonterminating e.g., consider case when program is run with one worker (-places 1:1)

Starvation-Free Parallel Program Executions

- A parallel program execution exhibits starvation if some task is repeatedly denied the opportunity to make progress
  - Starvation-freedom is sometimes referred to as "lock-out freedom"
  - Starvation is possible in HJ programs, since all tasks in the same program are assumed to be cooperating, rather than competing
    - If starvation occurs in a deadlock-free HJ program, the "equivalent" sequential program must be non-terminating
- Classic source of starvation: "Priority Inversion" problem for OS threads
  - Thread A is at high priority, waiting for result or resource from Thread C at low priority
  - Thread B at intermediate priority is CPU-bound
  - Thread C never runs, hence thread A never runs
  - Fix: when a high priority thread waits for a low priority thread, boost the priority of the low-priority thread

Bounded Wait

- **A parallel program execution exhibits bounded wait if each task requesting a resource should only have to wait for a bounded number of other tasks to "cut in line" i.e., to gain access to the resource after its request has been registered.**
- **If bound = 0, then the program execution is fair**


Bounded Wait
Mutual Exclusion
Progress
No cutting in!
Are there door locks?
Well, Did you see anybody go in?
Oversimplifying Assumptions

- Progress?
- Bounded Wait?

What's the difference?

- Progress?
  - If no process is waiting for a resource and several processes are requesting access to the resource, then access to the resource cannot be postponed indefinitely

- **Bounded Wait?**
  - A process requesting access to a resource should only have to wait for a bounded number of other processes to access the resource that requested access after it

Related Concepts: Progress Condition

- A resource is said to be obstruction-free if it is deadlock-free
- A resource is said to be lock-free if it is livelock-free and deadlock-free
- A resource is said to be wait-free if it is starvation-free, livelock-free, and deadlock-free

Example: Implementing AtomicInteger.getAndAdd() using compareAndSet()

```
/** Atomically adds delta to the current value.
 *
 * @param delta the value to add
 * @return the previous value
 */
public final int getAndAdd(int delta) {
    for (;;) { // try
        int current = get();
        int next = current + delta;
        if (compareAndSet(current, next))
            // commit
            return current;
    }
}
```

Is this implementation of getAndAdd() obstruction-free, lock-free or wait-free?

- **Source: http://gee.cs.oswego.edu/cgi-bin/viewcvs.cgi/jsr166/src/main/java/util/concurrent/atomic/AtomicInteger.java**

Outline

- Linearizability of Concurrent Executions and Concurrent Objects
- Liveness/progress guarantees
- <u>Optimized Implementations of Isolated</u>

Research Idea 1: Transactional Memory

- Execution of an isolated statement is treated as a transaction
  - In database systems, a transaction refers to a "unit of work" that has "all-or-nothing" semantics. Each unit of work must either complete in its entirety or have no visible effect.
- A TM system logs all read and write operations performed in a transaction and optimistically permits transactions to run in parallel, speculating that there won't be interference
- At the end of a transaction, a TM system checks if interference occurred with another transaction
  - If not, the transaction can be committed
  - If so, the transaction fails and has to be "retried"
- Both software and hardware implementations of TM have been explored extensively by the research community, but no implementation has proved suitable for mainstream use as yet
- Examples of Software TM system for Java: DSTM2, Deuce

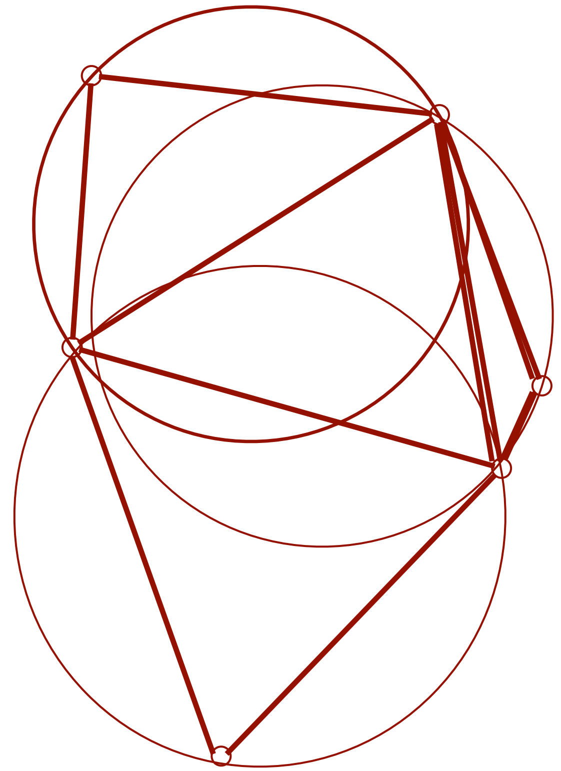
Research Idea 2: Delegated Isolation

- Challenge: scalable implementation of isolated without using a single global lock and without incurring transactional memory overheads
- Delegated isolation:
  - Restrict attention to "async isolated" case
    - replace non-async "isolated" by "finish async isolated"
  - Task dynamically acquires ownership of each object accessed in isolated block (optimistic parallelism)
    - Similar to transactional memory
  - On conflict, task A transfers all ownerships to worker executing conflicting task B and delegates execution of isolated block to B
    - Different from transactional memory
  - Deadlock-freedom and livelock-freedom guarantees

  - Reference: "Delegated Isolation", R. Lublinerman, J. Zhao, Z. Budimlic, S. Chaudhuri, V. Sarkar, OOPSLA 2011

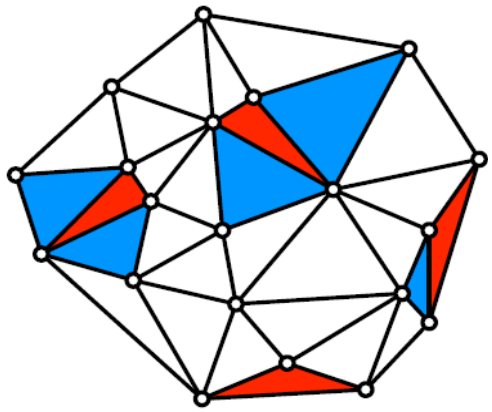
Example Algorithm: Delaunay Mesh Refinement

- **Input: a 2d triangle mesh that satisfies:**
  the Delaunay property: no point is contained in the circumcircle of a triangle
- **Output: a 2d triangle mesh that**
  —satisfies the Delaunay property
  —contains all points in the original mesh
  —satisfies an extra quality constraint
    - no triangle can have an angle $< 25°$
- **Algorithm (Ruppert's algorithm)**
  —iteratively select a triangle that violates the quality constraint and refine the mesh around it.

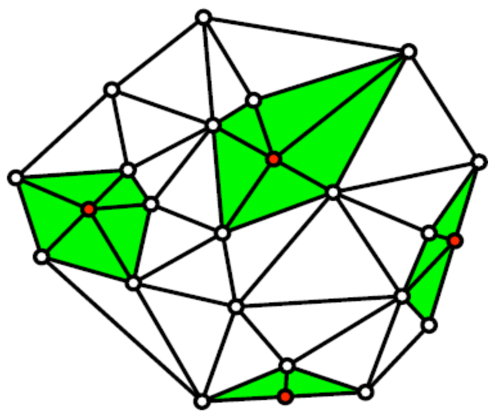


Delauney Mesh Refinement in Habanero-Java using Delegated Isolation

```
void doCavity(Triangle start) {
  async isolated {
    if (start.isActive()) {
      Cavity c = new Cavity(start);
      c.initialize(start);
      c.retriangulate();

      // launch retriagnulation on new bad triangles.
      Iterator bad = c.getBad().iterator();
      while (bad.hasNext()) {
        final Triangle b = (Triangle)bad.next();
        doCavity(b);
      }

      // if original bad triangle was NOT retriangulated,
      // launch its retriangulation again
      if (start.isActive())
        doCavity(start);
    }
  } // end isolated
}

void main() {
  mesh = ... ; // Load from file
  initialBadTriangles = mesh.badTriangles();
  Iterator it = initialBadTriangles.iterator();
  finish {
    while (it.hasNext()) {
      final Triangle t = (Triangle) it.next();
      if (t.isBad())
        Cavity.doCavity(t);
    }
  }
}
```



Before

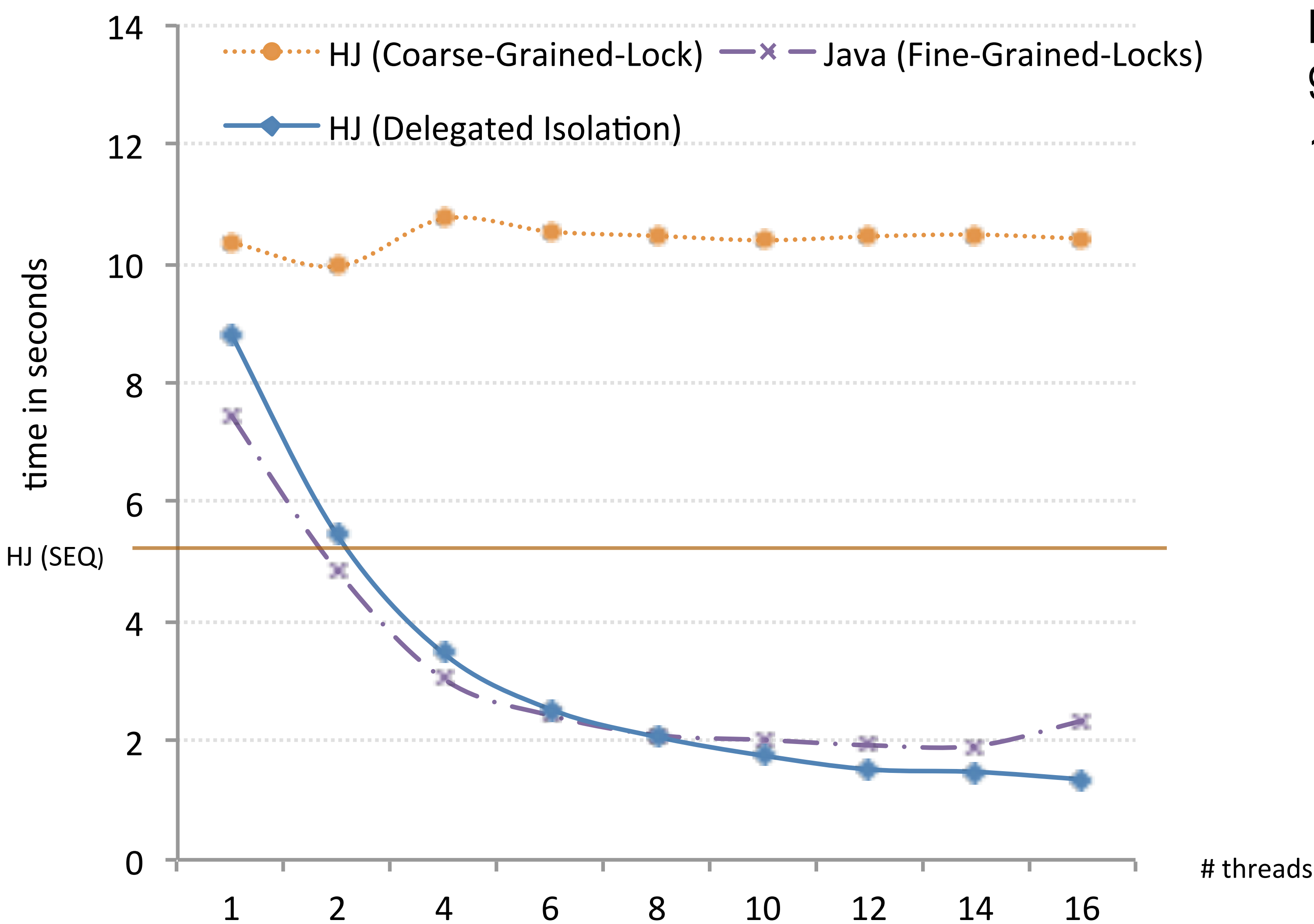


After

Figure source:
http://lcpc10.rice.edu/Keynote_Speakers_files/PingaliKeynote.pdf

Performance: DMR benchmark on 16-core Xeon SMP

(100,770 initial triangles of which 47,768 are "bad"; average # retriangulations is ~ 130,000)



DSTM2 performance:
962s w/ 1 thread
177s w/ 16 threads

Worksheet #10 (to be done individually or in pairs): Linearizability of method calls on a concurrent object

Name 1: ______________________ Name 2: ______________________

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()`	