COMP 322: Fundamentals of Parallel Programming

Lecture 25: Linearizability (contd), Intro to Java Threads

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

https://wiki.rice.edu/confluence/display/PARPROG/COMP322

Solution to Worksheet #24: Linearizability of method calls on a concurrent object

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()`	

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.

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
  - If there is a choice of points that is inconsistent with a sequential execution that doesn't matter, so long as we can identify one choice of points that is consistent with a sequential execution
  - Innocent until proven guilty!
- An object is linearizable if all its possible executions are linearizable

Why is Linearizability important?

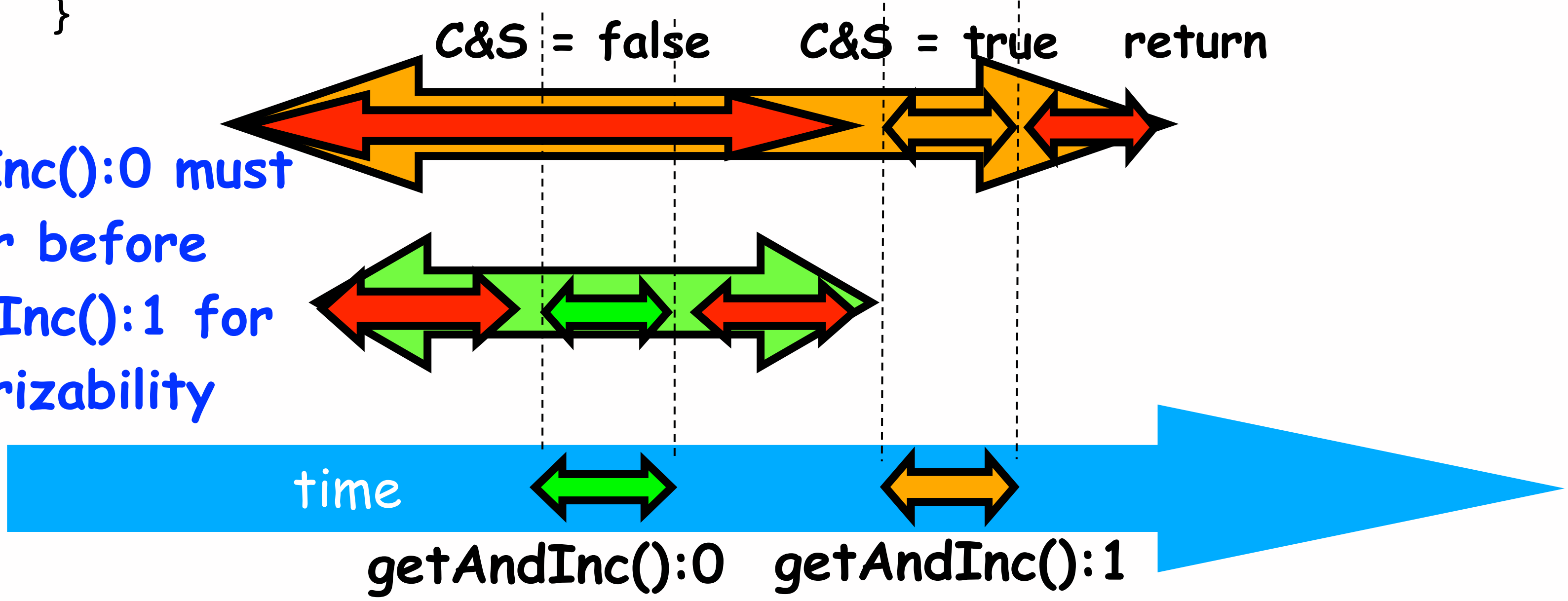
- **Linearizability is a correctness condition for concurrent objects**
- **For example, is the following implementation of AtomicInteger.getAndIncrement() linearizable?**
  - **Motivation: many processors provide hardware support for get() and compareAndSet(), but not for getAndAdd()**

```
public final int getAndIncrement() {
        int current = get();
        int next = current + 1;
        while (true) {
            if (compareAndSet(current, next))
                // success!
                return current;
        }
    }
```

A Linearizable Implementation of getAndIncrement()

```
public final int getAndIncrement() {
      while (true) {
            int current = get();
            int next = current + 1;
            if (compareAndSet(current, next))
                // success!
                return current;
      }
  }
```

getAndInc():0 must occur before getAndInc():1 for linearizability



Motivation for try-in-a-loop pattern

- Optimistic "nonblocking" synchronization
  - Pro: Resilient to failure or delay of any thread attempting synchronization
  - Con: "spin loop" may tie up a worker indefinitely
- *Try-in-a-loop* pattern for optimistic synchronization has the following structure

```
LOOP {
    1) Set-up (local operation invisible to other threads)
    2) Instantaneous effect e.g., CompareAndSet
        a) If successful break out of loop
        b) If unsuccessful continue loop
}
```

Another example of non-blocking synchronization: getAndAdd() as a generalization of getAndIncrement()

```
/** 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;
        }
  }
```

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

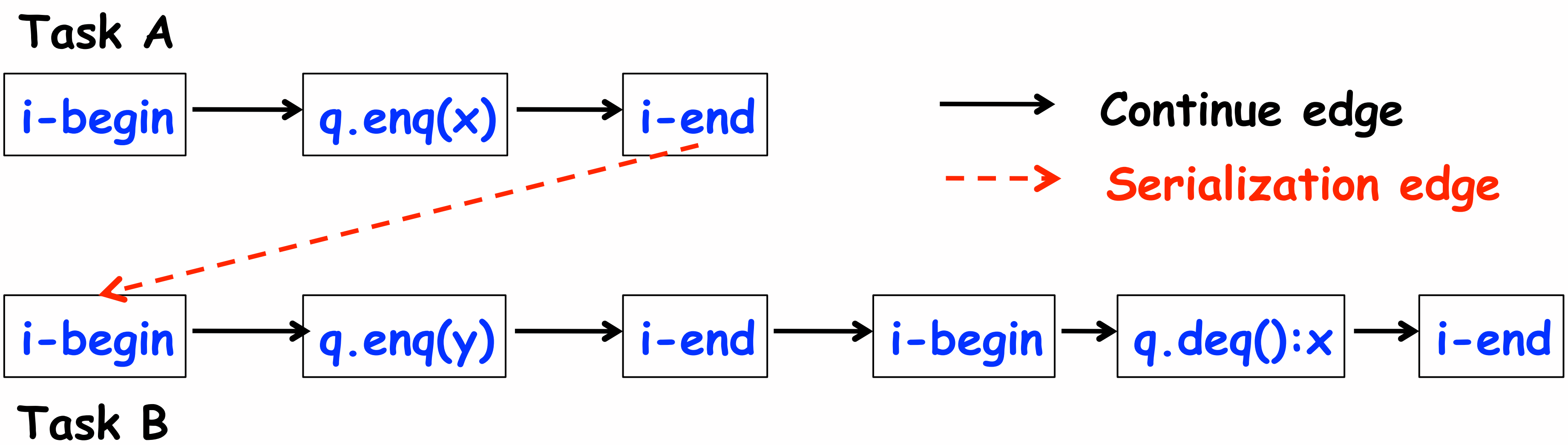
Example 4: execution of a monitor-based implementation of FIFO queue q (Recap)

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"

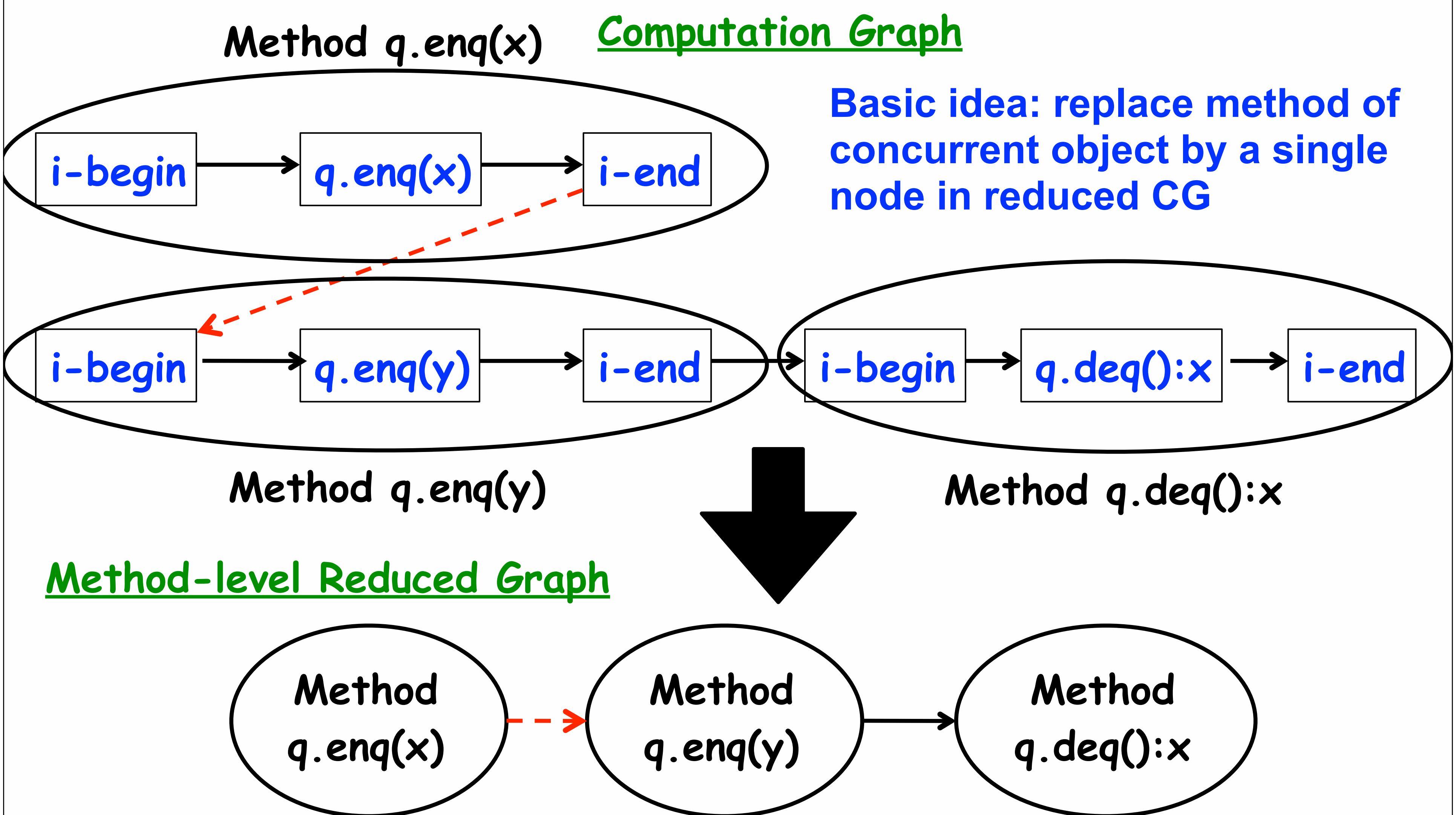
Computation Graph for previous execution (Example 4)



Monitor-based execution encloses each method call in an isolated statement, demarcated by isolated-begin (i-begin) and isolated-end (i-end) nodes

Creating a Reduced Computation Graph to model Instantaneous Execution of Methods in a Concurrent Object

Basic idea: replace method of concurrent object by a single node in reduced CG



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 the previous example.**
- **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.**
  - **We cannot use a cycle in the reduced CG as evidence of non-linearizability**

Example 5: Example execution of method calls on a concurrent FIFO queue q (Recap)

Is this a linearizable execution?

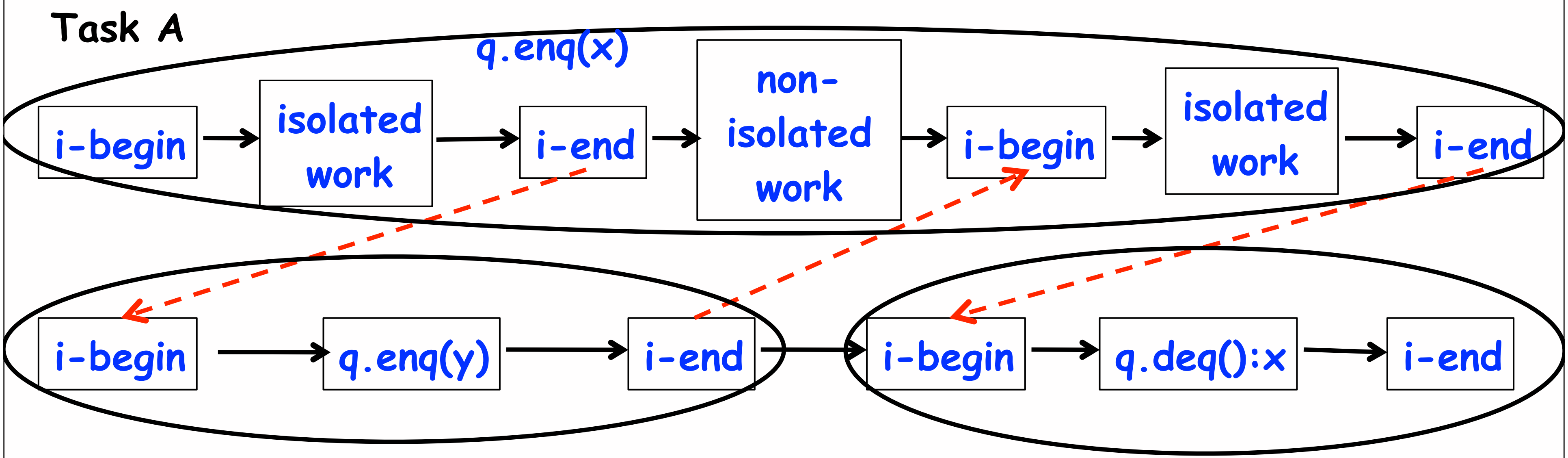
Time	Task *A*	Task *B*
0	Invoke `q.enq(x)`	
1	Work on `q.enq(x)`	Invoke `q.enq(y)`
2	Work on `q.enq(x)`	Return from `q.enq(y)`
3	Return from `q.enq(x)`	
4		Invoke `q.deq()`
5		Return `x` from `q.deq()`

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

Computation Graph for previous execution (Example 5)

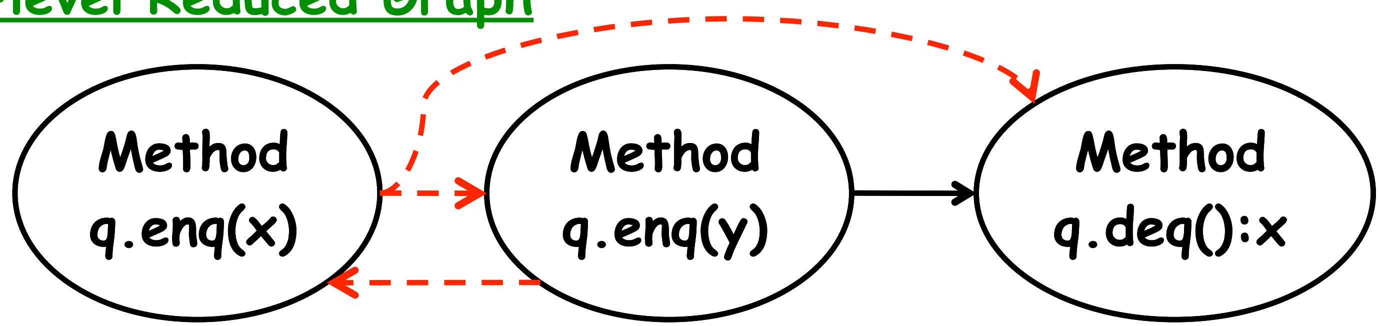
Computation Graph

Note: calls to get() & compareAndSet() are examples of isolated work



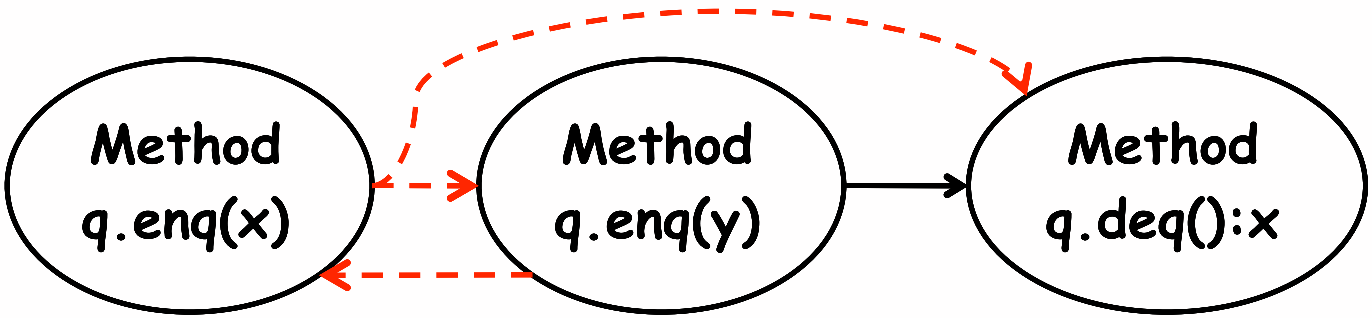
→ Continue edge - - -→ Serialization edge

Method-level Reduced Graph



Reduced Computation Graph for previous execution (Example 5)

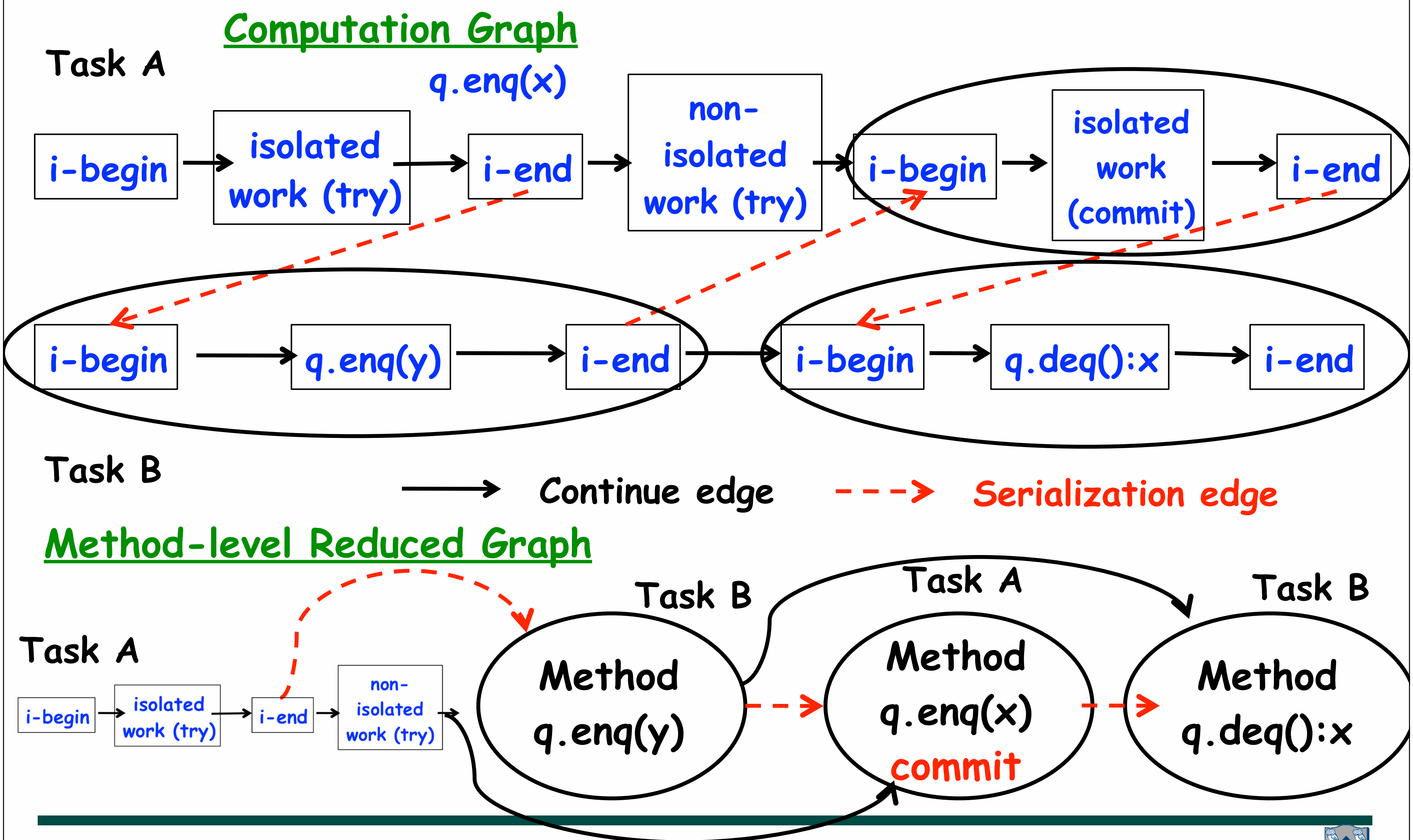
- **Example of linearizable execution graph for which reduced method-level graph is cyclic**



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

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

Computation Graph for Example 5 decomposed into try & commit portions



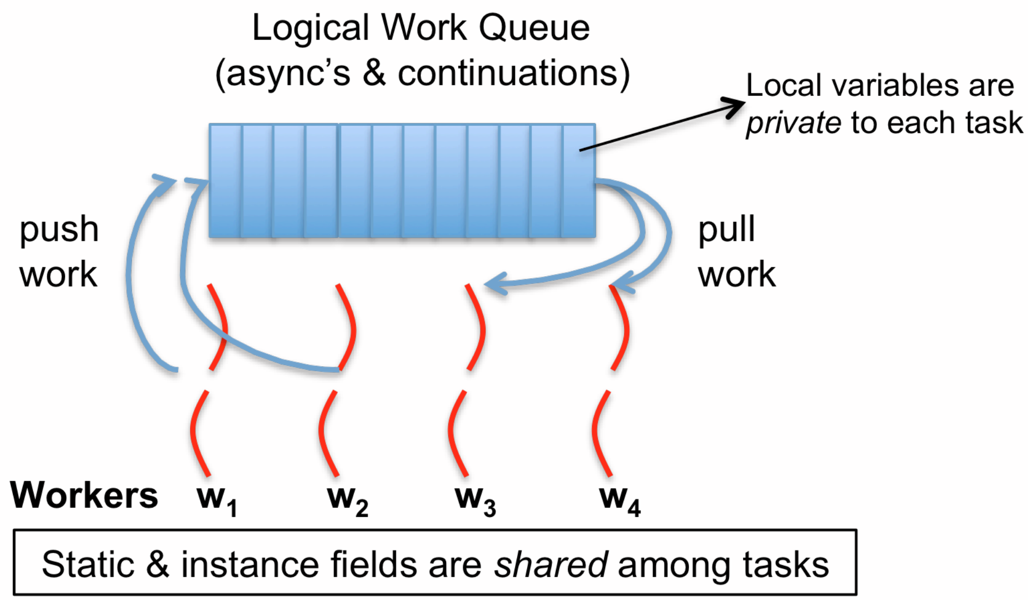
Introduction to Java threads: java.lang.Thread class

- Execution of a Java program begins with an instance of Thread created by the Java Virtual Machine (JVM) that executes the program's main() method.
- Parallelism can be introduced by creating additional instances of class Thread that execute as parallel threads.

```
public class Thread extends Object implements Runnable {
  Thread() { ... } // Creates a new Thread
  Thread(Runnable r) { ... } // Creates a new Thread with Runnable object r
  void  run() { ... } // Code to be executed by thr
   // Case 1: If this thread was cr
   //         then that object's run method
   // Case 2: If this class is subclassed, t
   //         in the subclass is called
  void start() { ... } // Causes this thread to sta
  void join() { ... } // Wait for this thread to die
  void  join(long m) // Wait at most m milliseconds for thread to die
  static Thread currentThread() // Returns currently executing thread
  . . .
}
```

A lambda can be passed as a Runnable

HJ runtime uses Java threads as workers …



- **HJ runtime creates a small number of worker threads, typically one per core**
- **Workers push async's/continuations into a logical work queue**
  - **when an async operation is performed**
  - **when an end-finish operation is reached**
- **Workers pull task/continuation work item when they are idle**

… because programming directly with Java threads can be expensive

k	$t_s(k)$	$t_1^{ws}(k)$	$t_1^{jt}(k)$
1	0.00550	1.67180	0.00264
2	0.00640	1.61984	0.64944
4	0.00752	1.67401	1.26081
8	0.00962	1.68423	5.39852
16	0.01117	1.71121	7.49290
32	0.01341	2.04591	8.14587
64	0.01962	2.07918	11.07557
128	0.02337	2.07780	12.03547
256	0.05199	2.13682	17.67796
512	0.07282	2.29679	28.28268
1024	0.14978	2.63632	51.30504
2048	0.31606	2.99007	90.20563
4096	0.57622	3.61543	175.49042
8192	0.75838	8.55980	333.09688
16384	**1.07625**	**9.50611**	**667.73758**

Fork-Join Microbenchmark Measurements (execution time in micro-seconds from Lecture 10)

start() and join() methods

- A Thread instance starts executing when its start() method is invoked
  - start() can be invoked at most once per Thread instance
    - Like actors, except that Java threads don't process messages
  - As with async, the parent thread can immediately move to the next statement after invoking t.start()
- A t.join() call forces the invoking thread to wait till thread t completes.
  - Lower-level primitive than finish since it only waits for a single thread rather than a collection of threads
  - No restriction on which thread performs a join on which thread, so it is possible to create a deadlock cycle using join()
    - Declaring thread references as final does not help because the new() and start() operations are separated for threads (unlike futures, where they are integrated)

Two-way Parallel Array Sum using Java Threads

```
// Start of main thread
sum1 = 0; sum2 = 0; // sum1 & sum2 are static fields
Thread t1 = new Thread(() -> {
    // Child task computes sum of lower half of array
    for(int i=0; i < X.length/2; i++) sum1 += X[i];
  });
t1.start();
// Parent task computes sum of upper half of array
for(int i=X.length/2; i < X.length; i++) sum2 += X[i];
// Parent task waits for child task to complete (join)
t1.join();
return sum1 + sum2;
```

Two-way Parallel Array Sum using HJ-Lib's finish & async API's

```
// Start of Task T0 (main program)
sum1 = 0; sum2 = 0; // sum1 & sum2 are static fields
finish(() -> {
  async(() -> {
    // Child task computes sum of lower half of array
    for(int i=0; i < X.length/2; i++) sum1 += X[i];
  });
  // Parent task computes sum of upper half of array
  for(int i=X.length/2; i < X.length; i++) sum2 += X[i];
});
// Parent task waits for child task to complete (join)
return sum1 + sum2;
```

Worksheet #25 (due by start of next lecture): Linearizability of method calls on a concurrent object

Name: ______________________ Netid: ______________________

Can you show an execution for which deq() results in an EmptyException in line 22 below? If so, that is a non-linearizable execution.

One Possible Attempt to Implement a Concurrent Queue

```
// Assume that no. of enq() operations is < Integer.MAX_VALUE
class Queue1 {
  AtomicInteger head = new AtomicInteger(0);
  AtomicInteger tail = new AtomicInteger(0);
  Object[] items = new Object[Integer.MAX_VALUE];
  public void enq(Object x) {
    int slot = tail.getAndIncrement(); // isolated(tail) ...
   items[slot] = x;
 } // enq
  public Object deq() throws EmptyException {
    int slot = head.getAndIncrement(); // isolated(head) ...
    Object value = items[slot];
    if (value == null) throw new EmptyException();
    return value;
  } // deq
} // Queue1

// Client code
finish {
  Queue1 q = new Queue1();
  async q.enq(new Integer(1));
  q.enq(newInteger(2));
  Integer x = (Integer) q.deq();
}
```