COMP 322: Fundamentals of Parallel Programming

Lecture 25: Linearizability (contd), Progress Guarantees in HJ programs

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
  - Slides and code examples 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
- "Introduction to Synchronization", Klara Nahrstedt, CS 241 Lecture 10, Spring 2007
  - www.cs.uiuc.edu/class/sp07/cs241/Lectures/10.sync.ppt
- "Programming Paradigms for Concurrency", Pavol Černý, Fall 2010, IST Austria
  - http://pub.ist.ac.at/courses/ppc10/slides/Linearizability.pptx

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
- Linearizability is a safety property for concurrent objects

Outline

- Review of formal definition of Linearizability
  - Safety property
- Progress guarantees in HJ programs
  - Liveness properties

Legality condition for a sequential history (Recap)

- A sequential history H is legal if:

  for each object x, H|x is in the sequential specification for x.
- for example: objects like queue, stack

Sequential Specifications

- If (precondition)
  - the object is in such-and-such a state, before you call the method,
- Then (postcondition)
  - the method will return a particular value, or throw a particular exception.
  - the object will be in some other state, when the method returns,

Example: Pre and PostConditions for a deq() operation on a FIFO Queue in a Sequential Program

Case 1:

- Precondition:
  - Queue is non-empty
- Postconditions:
  - Returns first item in queue
  - Removes first item in queue

Case 2:

- Precondition:
  - Queue is empty
- Postconditions:
  - Throws Empty exception
  - Queue state unchanged

Sequential vs Concurrent Executions

- **Sequential:**
  - Each method described in isolation
  - Method call as a single event
    - Start and end times do not impact its semantics
- **Concurrent**
  - Method call is an interval from invocation to response
  - Must characterize **all** possible interactions with concurrent calls
    - What if two enqs overlap?
    - Two deqs? enq and deq? ...

Formal definition of Linearizability (Recap)

History H is linearizable if

1) it can be transformed to history G such that G has no pending invocations,

- For each pending invocation in G, either remove it from H or append a response in H

2) there exits a legal sequential history S that is equivalent to G, and

- G and S are equivalent if for each thread A, G|A = S|A

3) if method call m0 precedes method call m1 in G, m0 must also precede m1 in S

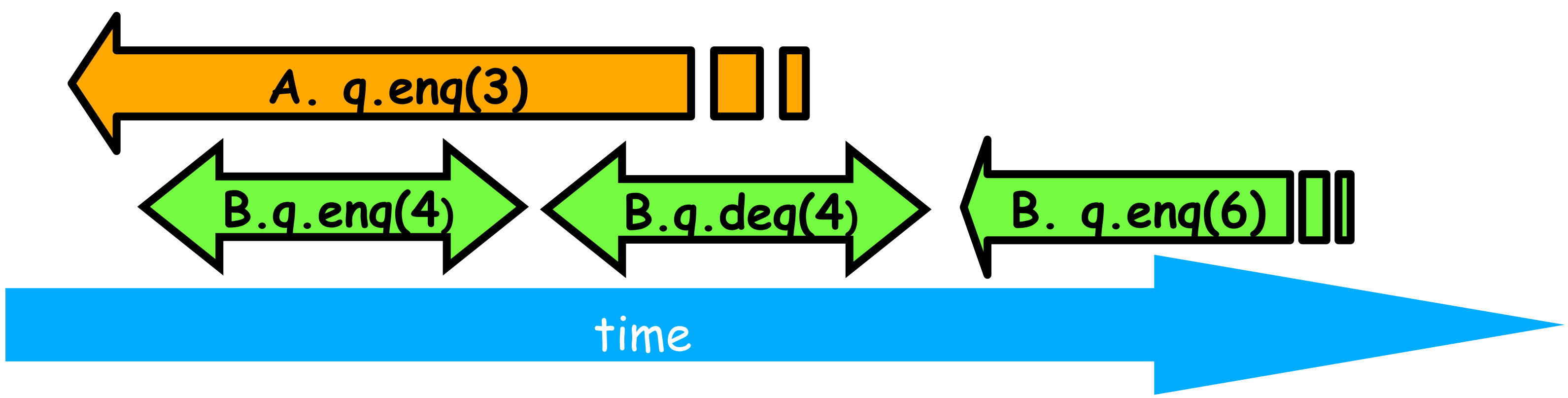
- Mathematically written as $\rightarrow_G \subset \rightarrow_S$

Example of history H (from last lecture)

If q.deq() returns 4, then q.enq(4) must take effect before q.enq(3)

A q.enq(3)
B q.enq(4)
B q:void
B q.deq()
B q:4
B q:enq(6)

Pending invocations: can be completed or discarded



Example (contd)

We (arbitrarily) decided to complete "A q.enq(3)", and discard "B q.enq(6)"

A q.enq(3)
B q.enq(4)
B q:void
B q.deq()
B q:4
A q:void

Two legal equivalent sequential histories

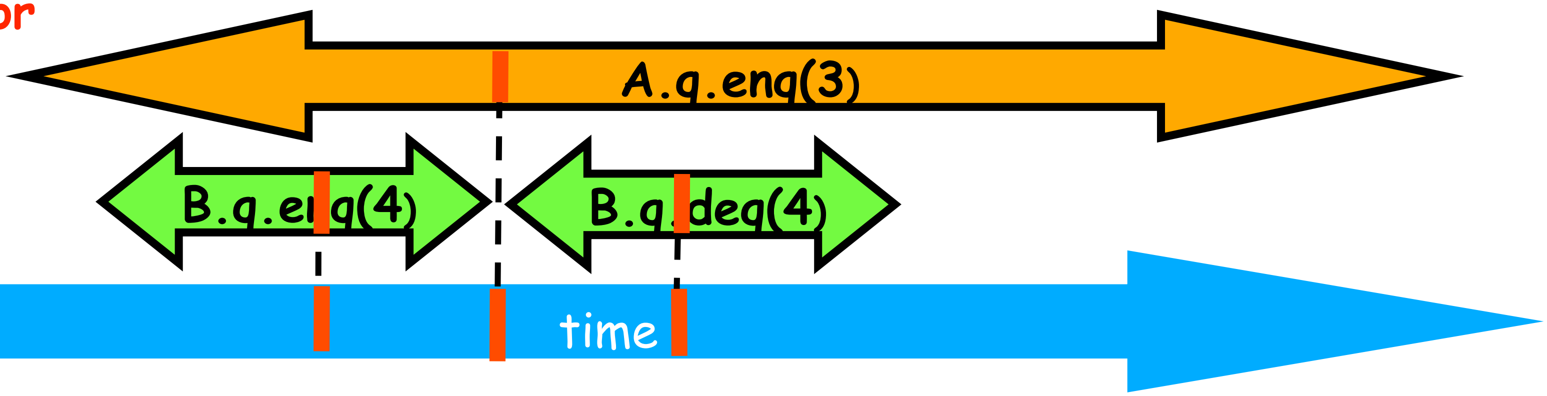
S1

B q.enq(4)
B q:void
A q.enq(3)
A q:void
B q.deq()
B q:4

S2

B q.enq(4)
B q:void
B q.deq()
B q:4
A q.enq(3)
A q:void

Time line for sequential history S1



Two Important Properties that follow from Linearizability

1) <u>Composability</u>

- History H is linearizable if and only if
  - For every object x
  - H|x is linearizable
- Why is composability important?
  - Modularity
  - Can prove linearizability of objects in isolation
  - Can compose independently-implemented objects

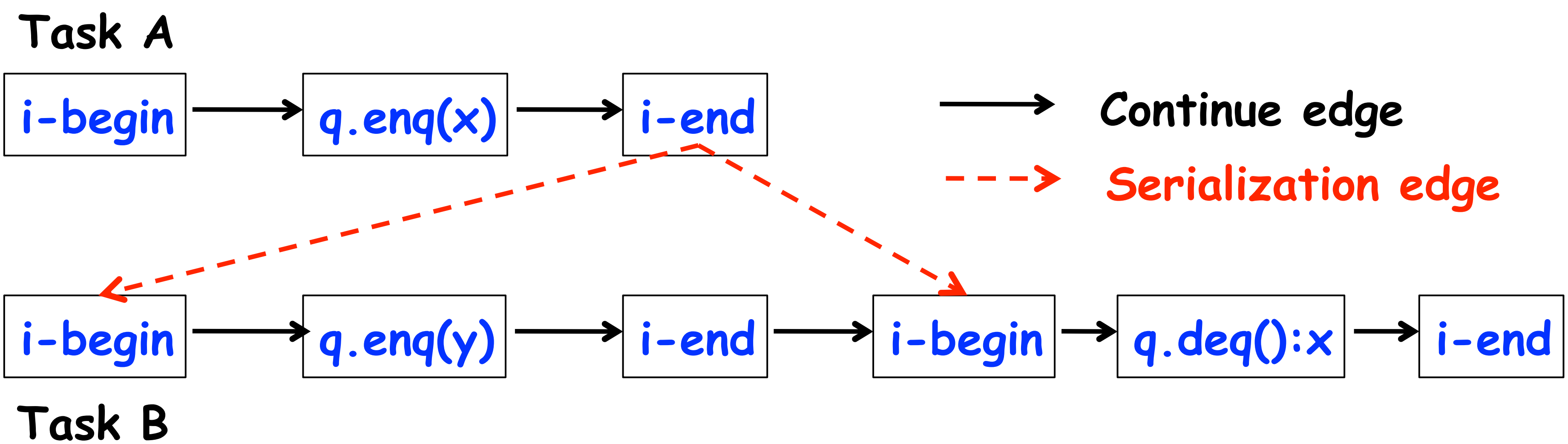
2) <u>Non-blocking</u>

- one method call is never forced to wait on another
- If method invocation "A q.inv(…)" is pending in history H, then there exists a response "A q:res(…)" such that "H + A q:res(…)" is linearizable

Relating Linearizability to the Computation Graph model (Lecture 23)

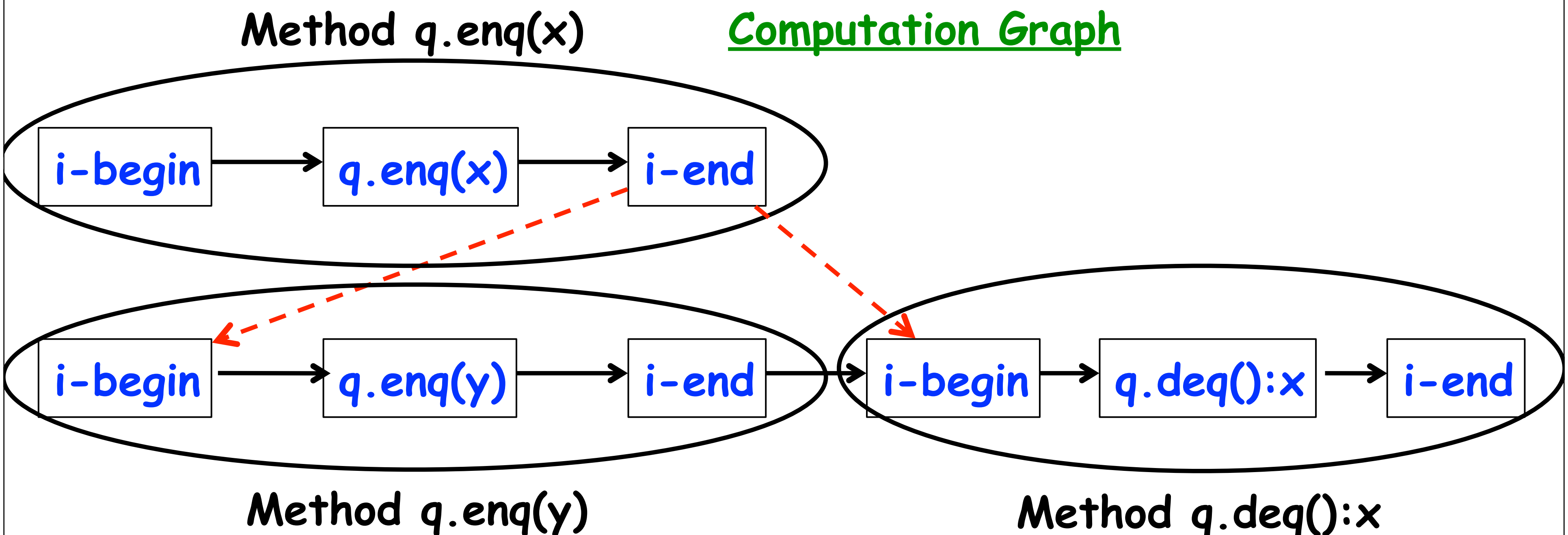
- Given a Computation Graph (CG), its reduced CG is obtained by collapsing also CG nodes belonging to teh same method call (on the concurrent object) to a single “macro-node”
- Given a reduced CG, a sufficient condition for linearizability is that the reduced CG is acyclic
  - This means that if the reduced CG is acyclic, then the underlying execution must be linearizable.
- However, the converse is not necessarily true

Computation Graph for monitor-based implementation of FIFO queue (Table 1)



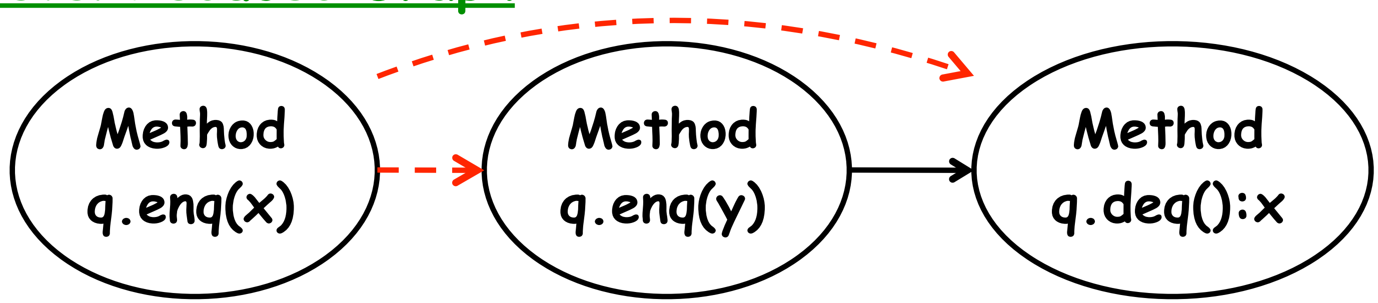
Creating a Reduced Graph to model Instantaneous Execution of Methods (Table 1)

<u>Computation Graph</u>

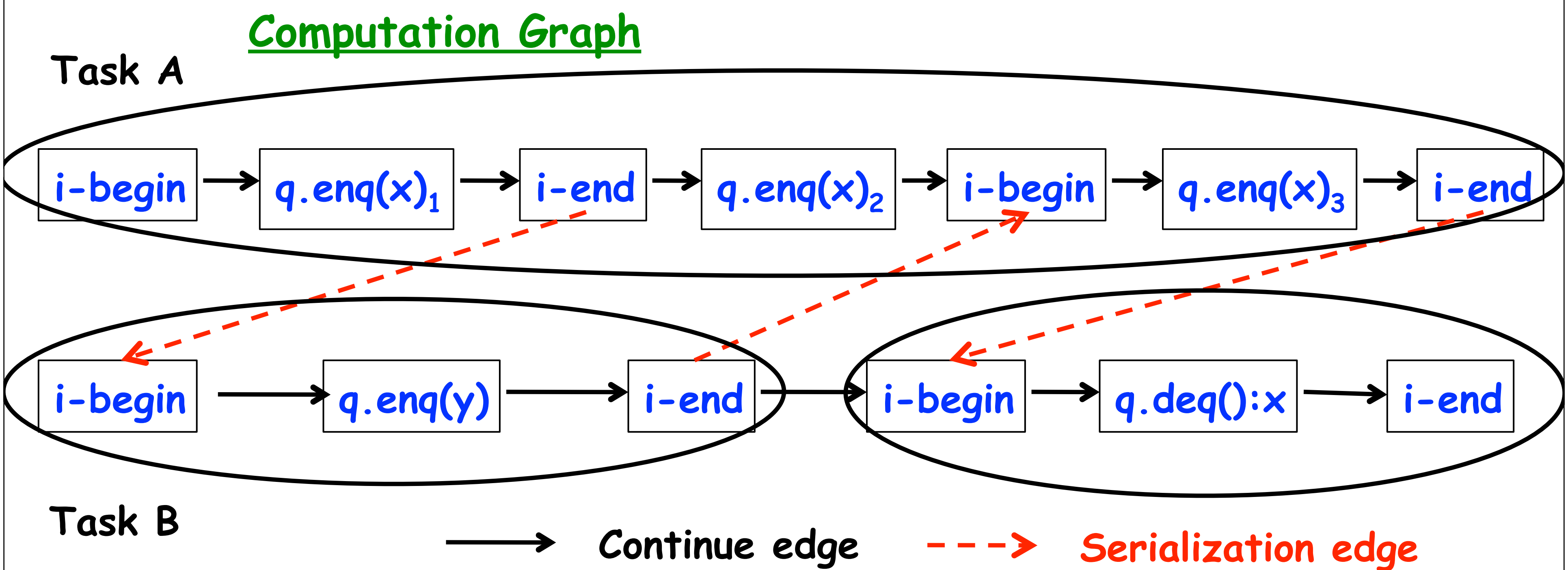


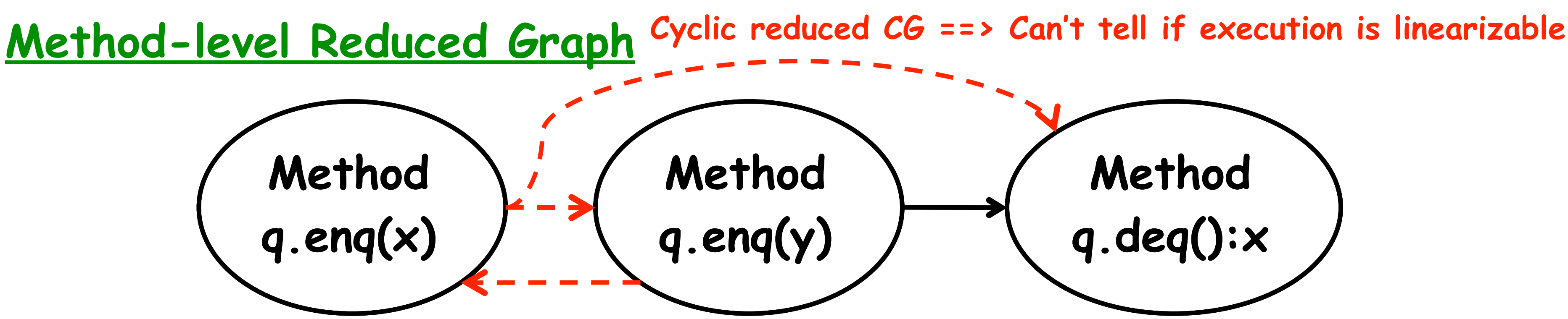
<u>Method-level Reduced Graph</u>

Acyclic reduced CG ==> Linearizable execution!



Computation Graph for concurrent implementation of FIFO queue (Table 2)





Making the cycle test more precise for linearizability

- Approach to make cycle test more precise for linearizability
  - Decompose concurrent object method into a sequence of pairs of "try" and "commit" steps
  - Assume that each "commit" step's execution does not use any input from any prior "try" step

  → Reduced graph can just reduce the "commit" steps to a single node instead of reducing the entire method to a single node

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;
      }
  }
```

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

Outline

- **Review of formal definition of Linearizability**
  - Safety property
- **Progress guarantees in HJ programs**
  - Liveness properties

Desirable Properties of Parallel Program Executions

- Data-race freedom
- Termination
  - But some applications are designed to be non-terminating
- 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
- Today's lecture discusses progress guarantees for HJ programs
  - We will revisit progress guarantees for Java concurrency later

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)

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 or 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)

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 have been non-terminating
- Classic source of starvation: "Priority Inversion" problem for OS threads (usually from different processes)
  - 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