Between Linearizability and Quiescent Consistency

Radha Jagadeesan  James Riely

DePaul University
Chicago, USA

ICALP 2014
Linearizability  (Herlihy/Wing 1990)

- “Each method call should appear to take effect instantaneously at some moment between its invocation and response.” (Herlihy/Shavit 2008)
- I.e., for every invocation, exists a linearization point such that
  - linearization point is between call and return
  - real-time order corresponds to some sequential execution

**Specification**

**Implementation**

- Compositional  (Herlihy/Wing 1990)
  Composition of the histories of two non-interfering linearizable objects is linearizable

- Intrinsically inefficient  (Dwork/Herlihy/Waarts 1997)
  Trade-off between high contention and using many variables

*Data Structures in the Multicore Age* (Shavit 2011, CACM)
Linearizability (Herlihy/Wing 1990)

- “Each method call should appear to take effect instantaneously at some moment between its invocation and response.” (Herlihy/Shavit 2008)

- I.e., for every invocation, exists a *linearization point* such that
  - linearization point is between call and return
  - real-time order corresponds to some sequential execution

---

Compositional (Herlihy/Wing 1990)
Composition of the histories of two non-interfering linearizable objects is linearizable

Intrinsically inefficient (Dwork/Herlihy/Waarts 1997)
Trade-off between high contention and using many variables
*Data Structures in the Multicore Age* (Shavit 2011, CACM)
Linearizability (Herlihy/Wing 1990)

- “Each method call should appear to take effect instantaneously at some moment between its invocation and response.” (Herlihy/Shavit 2008)
- I.e., for every invocation, exists a linearization point such that
  - linearization point is between call and return
  - real-time order corresponds to some sequential execution

- Compositional (Herlihy/Wing 1990)
  Composition of the histories of two non-interfering linearizable objects is linearizable

- Intrinsically inefficient (Dwork/Herlihy/Waarts 1997)
  Trade-off between high contention and using many variables

Data Structures in the Multicore Age (Shavit 2011, CACM)
Linearizability (Herlihy/Wing 1990)

- “Each method call should appear to take effect instantaneously at some moment between its invocation and response.” (Herlihy/Shavit 2008)

- I.e., for every invocation, exists a *linearization point* such that
  - linearization point is between call and return
  - real-time order corresponds to some sequential execution

Compositional (Herlihy/Wing 1990)
Composition of the histories of two non-interfering linearizable objects is linearizable

Intrinsically inefficient (Dwork/Herlihy/Waarts 1997)
Trade-off between high contention and using many variables

*Data Structures in the Multicore Age* (Shavit 2011, CACM)
**Linearizability** (Herlihy/Wing 1990)

- “Each method call should appear to take effect instantaneously at some moment between its invocation and response.” (Herlihy/Shavit 2008)
- I.e., for every invocation, exists a *linearization point* such that
  - linearization point is between call and return
  - real-time order corresponds to some sequential execution

![Diagram of Linearizability](attachment:image.png)

- **Compositional** (Herlihy/Wing 1990)
  Composition of the histories of two non-interfering linearizable objects is linearizable

- **Intrinsically inefficient** (Dwork/Herlihy/Waarts 1997)
  Trade-off between high contention and using many variables

*Data Structures in the Multicore Age* (Shavit 2011, CACM)
Quiescent Consistency  (Aspnes/Herlihy/Shavit 1991)

- Weaker than Linearizability (Lin \(\Rightarrow\) QC)
- Compositional
- “Method calls separated by a period of quiescence should appear to take effect in their real-time order.” (Herlihy/Shavit 2008)

Aspnes/Herlihy/Shavit (1991) actually prove other things

- Step property (weaker than QC)
  - Concretely: When quiescent, state is “very sensible”
  - Abstractly: *If at any point accessed sequentially, behaves sequentially*

- Gap property (morally “stronger” than QC)
  - Concretely: Even when not quiescent, state is “pretty sensible”
  - Abstractly: ??? *This paper*
Quiescent Consistency  (Aspnes/Herlihy/Shavit 1991)

- Weaker than Linearizability (Lin ⇒ QC)
- Compositional
- “Method calls separated by a period of quiescence should appear to take effect in their real-time order.” (Herlihy/Shavit 2008)

Aspnes/Herlihy/Shavit (1991) actually prove other things
- Step property (weaker than QC)
  Concretely: When quiescent, state is “very sensible”
  Abstractly: If at any point accessed sequentially, behaves sequentially
- Gap property (morally “stronger” than QC)
  Concretely: Even when not quiescent, state is “pretty sensible”
  Abstractly: ??? This paper
Quiescent Consistency (Aspnes/Herlihy/Shavit 1991)

- Weaker than Linearizability (Lin ⇒ QC)
- Compositional
- “Method calls separated by a period of quiescence should appear to take effect in their real-time order.” (Herlihy/Shavit 2008)

Aspnes/Herlihy/Shavit (1991) actually prove other things
- Step property (weaker than QC)
  Concretely: When quiescent, state is “very sensible”
  Abstractly: *If at any point accessed sequentially, behaves sequentially*
- Gap property (morally “stronger” than QC)
  Concretely: Even when not quiescent, state is “pretty sensible”
  Abstractly: ??? *This paper*
Quiescent Consistency  (Aspnes/Herlihy/Shavit 1991)

- Weaker than Linearizability (Lin ⇒ QC)
- Compositional
- “Method calls separated by a period of quiescence should appear to take effect in their real-time order.” (Herlihy/Shavit 2008)

Aspnes/Herlihy/Shavit (1991) actually prove other things

- Step property (weaker than QC)
  Concretely: When quiescent, state is “very sensible”
  Abstractly:  If at any point accessed sequentially, behaves sequentially

- Gap property (morally “stronger” than QC)
  Concretely: Even when not quiescent, state is “pretty sensible”
  Abstractly:  ??? This paper
Quiescent Consistency (Aspnes/Herlihy/Shavit 1991)

- Weaker than Linearizability (Lin ⇒ QC)
- Compositional
- “Method calls separated by a period of quiescence should appear to take effect in their real-time order.” (Herlihy/Shavit 2008)

Aspnes/Herlihy/Shavit (1991) actually prove other things

- Step property (weaker than QC)
  - Concretely: When quiescent, state is “very sensible”
  - Abstractly: If at any point accessed sequentially, behaves sequentially
- Gap property (morally “stronger” than QC)
  - Concretely: Even when not quiescent, state is “pretty sensible”
  - Abstractly: ??? This paper

is a quiescent point
Quiescent Consistency (Aspnes/Herlihy/Shavit 1991)

- Weaker than Linearizability (Lin $\Rightarrow$ QC)
- Compositional
- “Method calls separated by a period of quiescence should appear to take effect in their real-time order.” (Herlihy/Shavit 2008)

Aspnes/Herlihy/Shavit (1991) actually prove other things
- Step property (weaker than QC)
  - Concretely: When quiescent, state is “very sensible”
  - Abstractly: If at any point accessed sequentially, behaves sequentially
- Gap property (morally “stronger” than QC)
  - Concretely: Even when not quiescent, state is “pretty sensible”
  - Abstractly: ??? This paper
Quiescent Consistency (Aspnes/Herlihy/Shavit 1991)

- Weaker than Linearizability (Lin $\Rightarrow$ QC)
- Compositional
- “Method calls separated by a period of quiescence should appear to take effect in their real-time order.” (Herlihy/Shavit 2008)

Aspnes/Herlihy/Shavit (1991) actually prove other things

- Step property (weaker than QC)
  Concretely: When quiescent, state is “very sensible”
  Abstractly: *If at any point accessed sequentially, behaves sequentially*

- Gap property (morally “stronger” than QC)
  Concretely: Even when not quiescent, state is “pretty sensible”
  Abstractly: ??? *This paper*
But first. . . Weak Quiescent Consistency

- Abstract view of “step property”
- If at any point accessed sequentially, behaves sequentially

Spec [ ] ( ) ( ) ← →

Exec ← ( ) ( ) ( ) ( ) ( ) ( ) ( ) ( ) →

- No comment about periods of concurrency
  QC requires permutation
  Weak QC does not (may be no spec trace with same set of events)
But first... Weak Quiescent Consistency

- Abstract view of “step property”
- If at any point accessed sequentially, behaves sequentially

No comment about periods of concurrency
QC requires *permutation*
Weak QC does not (may be no spec trace with same set of events)
But first... Weak Quiescent Consistency

- Abstract view of “step property”
- If at any point accessed sequentially, behaves sequentially

Spec

Exec

- No comment about periods of concurrency
  QC requires *permutation*
  Weak QC does not (may be no spec trace with same set of events)
This paper... Quantitative Quiescent Consistency

- Between Linearizability and QC (Lin $\Rightarrow$ QQC $\Rightarrow$ QC)
- Compositional
- “Nonlinearizable behavior proportional to number of *early concurrent* calls”
This paper... Quantitative Quiescent Consistency

- Between Linearizability and QC (Lin $\Rightarrow$ QQC $\Rightarrow$ QC)
- Compositional
- “Nonlinearizable behavior proportional to number of early concurrent calls”
Between Linearizability and QC (Lin $\Rightarrow$ QQC $\Rightarrow$ QC)

Compositional

“Nonlinearizable behavior proportional to number of early concurrent calls”
Between Linearizability and QC (Lin $\Rightarrow$ QQC $\Rightarrow$ QC)

- Compositional
- “Nonlinearizable behavior proportional to number of early concurrent calls”

Called early
This paper... Quantitative Quiescent Consistency

- Between Linearizability and QC (Lin $\Rightarrow$ QQC $\Rightarrow$ QC)
- Compositional
- “Nonlinearizable behavior proportional to number of early concurrent calls”

![Diagram showing the relationship between Spec and Impl, with arrows indicating the flow of calls and return actions.]

- Called early
- Returns late
This paper... Quantitative Quiescent Consistency

- Between Linearizability and QC (Lin $\Rightarrow$ QQC $\Rightarrow$ QC)
- Compositional
- “Nonlinearizable behavior proportional to number of early concurrent calls”

- Early concurrent calls enable out-of-order behavior

Called early

Returns late
Definitions

- Number the call/return pairs of the specification

\[ [1,1] (2,2) \{3,3\} (4,4) \{5,5\} [6,6] \ldots \]

- Linearizability: If \( i \stackrel{\text{precedes}}{\longrightarrow} j \) then \( i < j \) (Herlihy/Wing 1990)

- QC: If \( i \stackrel{\text{quiescent}}{\longrightarrow} j \) then \( i < j \) (Definition 3.1)

- Linearizability: If \( i < j \) then \( [i, i \stackrel{\text{precedes}}{\longrightarrow} ] j \) (Theorem 2.2)

- Linearizability: \( \{i \mid i \stackrel{\text{precedes}}{\longrightarrow} j\} \supseteq \{1, \ldots, j\} \) (Calculation)

- QQC: \( |\{i \mid i \stackrel{\text{precedes}}{\longrightarrow} j\}| \geq |\{1, \ldots, j\}| = j \) (Theorem 4.3)
Definitions

- Number the call/return pairs of the specification

![Spec diagram]

- **Linearizability:** If \( i \) precedes \( j \) then \( i < j \)  
  (Herlihy/Wing 1990)

- **QC:** If \( i \) quiescent precedes \( j \) then \( i < j \)  
  (Definition 3.1)

- **Linearizability:** If \( i < j \) then \( [i \to j] \)  
  (Theorem 2.2)

- **Linearizability:** \( \{ i \mid [i \to ] \} \supseteq \{1, \ldots, j\} \)  
  (Calculation)

- **QQC:** \( |\{ i \mid [i \to ] \}| \geq |\{1, \ldots, j\}| = j \)  
  (Theorem 4.3)
Definitions

- Number the call/return pairs of the specification

Spec: [ ] ( ) [ ]

- Linearizability: If \( i \) precedes \( j \) then \( i < j \) (Herlihy/Wing 1990)
- QC: If \( i \) quiescent \( j \) then \( i < j \) (Definition 3.1)
- Linearizability: If \( i < j \) then \( i \) precedes \( j \) (Theorem 2.2)
- Linearizability: \( \{ i \mid i \) precedes \( j \} \supseteq \{ 1, \ldots, j \} \) (Calculation)
- QQC: \( \left| \{ i \mid i \) precedes \( j \} \right| \geq \left| \{ 1, \ldots, j \} \right| = j \) (Theorem 4.3)
Definitions

- Number the call/return pairs of the specification

\[ \text{Spec} \]

- Linearizability: If \( i \xrightarrow{\text{precedes}} j \) then \( i < j \) (Return-to-call)

- QC: If \( i \xrightarrow{\text{quiescent}} j \) then \( i < j \) (Definition 3.1)

- Linearizability: If \( i < j \) then \( [i \xrightarrow{\text{precedes}} j] \) (Call-to-return)

- Linearizability: \( \{ i \mid [i \xrightarrow{\text{precedes}} j] \} \supseteq \{1, \ldots, j\} \) (Calculation)

- QQC: \( |\{ i \mid [i \xrightarrow{\text{precedes}} j] \}| \geq |\{1, \ldots, j\}| = j \) (Theorem 4.3)
Definitions

- Number the call/return pairs of the specification

\[ \text{Spec} \]

- **Linearizability:** If \( i \xrightarrow{\text{precedes}} j \) then \( i < j \) (Herlihy/Wing 1990)

- **QC:** If \( i \xrightarrow{\text{quiescent}} j \) then \( i < j \) (Definition 3.1)

- **Linearizability:** If \( i < j \) then \( i \xrightarrow{\text{precedes}} j \) (Theorem 2.2)

- **Linearizability:** \( \{ i \mid i \xrightarrow{\text{precedes}} j \} \supseteq \{1, \ldots, j\} \) (Calculation)

- **QQC:** \( \left| \{ i \mid i \xrightarrow{\text{precedes}} j \} \right| \geq \left| \{1, \ldots, j\} \right| = j \) (Theorem 4.3)
Definitions

- Number the call/return pairs of the specification

- **Linearizability**: If \( i \) precedes \( j \) then \( i < j \)  
  (Herlihy/Wing 1990)

- **QC**: If \( i \) quiescent precedes \( j \) then \( i < j \)  
  (Definition 3.1)

- **Linearizability**: If \( i < j \) then \( [i] \) precedes \( j \)  
  (Theorem 2.2)

- **Linearizability**:  
  \( \{ i \mid [i] \) precedes \( j \} \supseteq \{1, \ldots, j\} \)  
  (Calculation)

- **QQC**:  
  \( \mid \{ i \mid [i] \) precedes \( j \} \mid \geq \mid \{1, \ldots, j\} \mid = j \)  
  (Theorem 4.3)
Definitions

- Number the call/return pairs of the specification

**Spec**

- **Linearizability:** If \( i \xleftarrow{\text{precedes}} j \) then \( i < j \)
- **QC:** If \( i \xrightarrow{\text{quiescent}} j \) then \( i < j \)
- **Linearizability:** If \( i < j \) then \( \{i \leftarrow j\} \supseteq \{1, \ldots, j\} \)
- **Linearizability:** \( \left| \{i \leftarrow j\} \right| \geq \left| \{1, \ldots, j\} \right| = j \)
- **QQC:**
Interesting examples

- Enabling early call can be used repeatedly

- Enablers can accumulate

- Enablers can themselves be out-of-order
Interesting examples

- Enabling early call can be used repeatedly

- Enablers can accumulate

- Enablers can themselves be out-of-order
Interesting examples

- Enabling early call can be used repeatedly

- Enablers can accumulate

- Enablers can themselves be out-of-order
Quiescently Consistent Data Structures

- Counting networks
  - Bitonic Networks (Aspnes/Herlihy/Shavit 1991)
  - Diffracting Trees (Shavit/Zemach 1994)
  - Decrement/increment (Shavit/Touitou 1995)
    (Aiello/Busch/Herlihy/Mavronicolas/Shavit/Touitou 1999)

- Stacks and Bags (aka, Pools)
  - Elimination Arrays/Trees (Shavit/Touitou 1995)

- “Almost” Linearizable
  - Experimental results
  - Theory involving max/min times (Lynch/Shavit/Shvartsman/Touitou 1996)

- *The Art of Multiprocessor Programming* (Herlihy/Shavit 2008)
**N-counter** (simplified from Aspnes/Herlihy/Shavit 1991)

```java
class Counter<N:Int> {
    field b:[0..N-1] = 0; // 1 balancer
    field c:Int[] = [0, 1, ..., N-1]; // N counters
    method getAndIncrement():Int {
        val i:[0..N-1];
        atomic { i = b; b++; }
        atomic { val v = c[i]; c[i] += N; return v; } }
}
```

\[ \langle b = 0, c = [0, 1] \rangle \xrightarrow{inc} \langle b = 1, c = [0, 1] \rangle \xrightarrow{inc} \langle b = 1, c = [2, 1] \rangle \]

\[ \langle b = 0, c = [2, 1] \rangle \xrightarrow{inc} \langle b = 0, c = [2, 1] \rangle \xrightarrow{inc} \langle b = 0, c = [2, 3] \rangle \]

\[ \langle b = 1, c = [2, 1] \rangle \xrightarrow{inc} \langle b = 1, c = [2, 3] \rangle \xrightarrow{inc} \langle b = 1, c = [4, 3] \rangle \]

Behaves sequentially ☺️
$N$-counter (simplified from Aspnes/Herlihy/Shavit 1991)

```java
class Counter<N:Int> {  
  field b:[0..N-1] = 0;  // 1 balancer
  field c:Int[] = [0, 1, ..., N-1];  // N counters
  method getAndIncrement():Int {  
    val i:[0..N-1];  
    atomic { i = b; b++; }  
    atomic { val v = c[i]; c[i] += N; return v; } } }
```

\[
\langle b = 0, c = [0, 1]\rangle \xrightarrow{\text{inc}} \langle b = 1, c = [0, 1]\rangle \xrightarrow{\text{inc}} \langle b = 1, c = [2, 1]\rangle
\]

\[
\langle b = 0, c = [2, 1]\rangle \xrightarrow{\text{inc}} \langle b = 0, c = [2, 3]\rangle
\]

\[
\langle b = 1, c = [2, 3]\rangle \xrightarrow{\text{inc}} \langle b = 1, c = [4, 3]\rangle
\]

Behaves sequentially 😊
\[ N \text{-counter} \] (simplified from Aspnes/Herlihy/Shavit 1991)

```java
class Counter<N:Int> {
    field b:[0..N-1] = 0; // 1 balancer
    field c:Int[] = [0, 1, ..., N-1]; // N counters
    method getAndIncrement():Int {
        val i:[0..N-1];
        atomic { i = b; b++; }
        atomic { val v = c[i]; c[i] += N; return v; } } }
```

\[ \langle b = 0, c = [0, 1] \rangle \xrightarrow{inc} \langle b = 1, c = [0, 1] \rangle \xrightarrow{inc} \langle b = 1, c = [2, 1] \rangle \]

\[ \langle b = 0, c = [2, 1] \rangle \xrightarrow{inc} \langle b = 0, c = [2, 3] \rangle \]

\[ \langle b = 1, c = [2, 1] \rangle \xrightarrow{inc} \langle b = 1, c = [2, 3] \rangle \xrightarrow{inc} \langle b = 1, c = [4, 3] \rangle \]

Behaves sequentially ☺️
**$N$-counter** (simplified from Aspnes/Herlihy/Shavit 1991)

```java
class Counter<N:Int> {
    field b:[0..N-1] = 0; // 1 balancer
    field c:[Int[]] = [0, 1, ..., N-1]; // N counters
    method getAndIncrement():Int {
        val i:[0..N-1];
        atomic { i = b; b++; } // For illustrative purposes, we can consider atomic operations.
        atomic { val v = c[i]; c[i] += N; return v; } // Ensure consistency in operations.
    }
}
```

\[
\langle b = 0, c = [0, 1] \rangle \xrightarrow{\text{inc}^0} \langle b = 1, c = [0, 1] \rangle \xrightarrow{\text{inc}^1} \langle b = 1, c = [2, 1] \rangle
\]

\[
\langle b = 0, c = [2, 1] \rangle \xrightarrow{\text{inc}^1} \langle b = 0, c = [2, 1] \rangle \xrightarrow{\text{inc}^2} \langle b = 0, c = [2, 3] \rangle
\]

\[
\langle b = 1, c = [2, 3] \rangle \xrightarrow{\text{inc}^2} \langle b = 1, c = [4, 3] \rangle
\]

Behaves sequentially 😊
**$N$-counter** (simplified from Aspnes/Herlihy/Shavit 1991)

```java
class Counter<N: Int> {
    field b: [0..N-1] = 0; // 1 balancer
    field c : Int[] = [0, 1, ..., N-1]; // N counters
    method getAndIncrement() : Int {
        val i: [0..N-1];
        atomic { i = b; b++; }
        atomic { val v = c[i]; c[i] += N; return v; } } }
```

\[
\langle b = 0, c = [0, 1] \rangle \xrightarrow{\text{inc}} \langle b = 1, c = [0, 1] \rangle \xrightarrow{\text{inc}} \langle b = 1, c = [2, 1] \rangle \\
\langle b = 0, c = [2, 1] \rangle \xrightarrow{\text{inc}} \langle b = 0, c = [2, 3] \rangle \\
\langle b = 1, c = [2, 1] \rangle \xrightarrow{\text{inc}} \langle b = 1, c = [2, 3] \rangle \xrightarrow{\text{inc}} \langle b = 1, c = [4, 3] \rangle
\]

Behaves sequentially 😊
N-counter (simplified from Aspnes/Herlihy/Shavit 1991)

```scala
class Counter<N: Int> {
    field b: [0..N-1] = 0; // 1 balancer
    field c: Int[] = [0, 1, ..., N-1]; // N counters
    method getAndIncrement(): Int {
        val i: [0..N-1];
        atomic { i = b; b++; }
        atomic { val v = c[i]; c[i] += N; return v; } }
}
```

Behaves sequentially 🌼

\[
\langle b = 0, c = [0, 1]\rangle \xrightarrow{\text{inc}_0} \langle b = 1, c = [0, 1]\rangle \xrightarrow{\text{inc}_2} \langle b = 1, c = [2, 1]\rangle \\
\langle b = 0, c = [0, 1]\rangle \xrightarrow{\text{inc}_1} \langle b = 1, c = [2, 1]\rangle \xrightarrow{\text{inc}_4} \langle b = 1, c = [4, 3]\rangle \\
\langle b = 0, c = [2, 3]\rangle \xrightarrow{\text{inc}_0} \langle b = 1, c = [2, 3]\rangle \xrightarrow{\text{inc}_1} \langle b = 1, c = [4, 3]\rangle
\]
$N$-counter (simplified from Aspnes/Herlihy/Shavit 1991)

```java
class Counter<N:Int> {
    field b:[0..N-1] = 0; // 1 balancer
    field c:Int[] = [0, 1, ... , N-1]; // N counters
    method getAndIncrement():Int {
        val i:[0..N-1];
        atomic { i = b; b++; }
        atomic { val v = c[i]; c[i] += N; return v; }
    }
}
```

\[
\langle b = 0, c = [0, 1] \rangle \xrightarrow{\text{inc}_0} \langle b = 1, c = [0, 1] \rangle \xrightarrow{\text{inc}_0} \langle b = 1, c = [2, 1] \rangle
\]

\[
\langle b = 0, c = [2, 1] \rangle \xrightarrow{\text{inc}_1} \langle b = 0, c = [2, 3] \rangle
\]

\[
\langle b = 1, c = [2, 3] \rangle \xrightarrow{\text{inc}_2} \langle b = 1, c = [4, 3] \rangle
\]

Behaves sequentially 😊
class Counter<N: Int> {
    field b: [0..N-1] = 0;  // 1 balancer
    field c: Int[] = [0, 1, ..., N-1];  // N counters
    method getAndIncrement(): Int {
        val i: [0..N-1];
        atomic { i = b; b++; }
        atomic { val v = c[i]; c[i] += N; return v; } } }

\langle b = 0, c = [0, 1] \rangle \xrightarrow{inc} \langle b = 1, c = [0, 1] \rangle
\langle b = 0, c = [0, 1] \rangle \xrightarrow{inc} \langle b = 0, c = [0, 3] \rangle
\langle b = 0, c = [0, 1] \rangle \xrightarrow{inc} \langle b = 1, c = [0, 3] \rangle
\langle b = 1, c = [0, 3] \rangle \xrightarrow{inc} \langle b = 1, c = [2, 3] \rangle
\langle b = 1, c = [2, 3] \rangle \xrightarrow{inc} \langle b = 1, c = [4, 3] \rangle

\langle b = 0, c = [0, 1] \rangle \xrightarrow{inc} \langle b = 1, c = [0, 1] \rangle
\langle b = 1, c = [0, 1] \rangle \xrightarrow{inc} \langle b = 0, c = [0, 3] \rangle
\langle b = 0, c = [0, 3] \rangle \xrightarrow{inc} \langle b = 1, c = [0, 3] \rangle
\langle b = 1, c = [0, 3] \rangle \xrightarrow{inc} \langle b = 1, c = [2, 3] \rangle
\langle b = 1, c = [2, 3] \rangle \xrightarrow{inc} \langle b = 1, c = [4, 3] \rangle

b=0
\begin{array}{c}
/ \\
| \\
/ \\
\end{array}
\begin{array}{c}
c[0]=0 \\
c[1]=1
\end{array}

Not Linearizable 😞, but QQK 🎉
class Counter<N: Int> {
    field b: [0..N-1] = 0; // 1 balancer
    field c: Int[] = [0, 1, ..., N-1]; // N counters
    method getAndIncrement(): Int {
        val i: [0..N-1];
        atomic { i = b; b++; } // inc
        atomic { val v = c[i]; c[i] += N; return v; } // inc
    }
}
\(N\)-counter — Execution 2

class Counter\(<N: Int>\) {
    field \(b: [0..N-1] = 0;\) // 1 balancer
    field \(c: \text{Int}[] = [0, 1, \ldots, N-1];\) // \(N\) counters
    method getAndIncrement(): \text{Int} {
        val \(i: [0..N-1];\)
        atomic \{ i = b; b++; \}
        atomic \{ val \(v = c[i]; c[i] += N; \text{return } v; \}\}
    }
}

\[
\begin{align*}
\langle b = 0, c = [0, 1] \rangle &\xrightarrow{\text{inc}} \langle b = 1, c = [0, 1] \rangle \\
\langle b = 0, c = [0, 1] \rangle &\xrightarrow{\text{inc}} \langle b = 0, c = [0, 3] \rangle \\
\langle b = 0, c = [0, 3] \rangle &\xrightarrow{\text{inc}} \langle b = 1, c = [2, 3] \rangle \\
\langle b = 1, c = [2, 3] \rangle &\xrightarrow{\text{inc}} \langle b = 1, c = [4, 3] \rangle
\end{align*}
\]

Not Linearizable ☹, but QQC ☺
$N$-counter — Execution 2

class Counter<N:Int> {
    field b:[0..N-1] = 0; // 1 balancer
    field c:Int[] = [0, 1, ..., N-1]; // N counters
    method getAndIncrement():Int {
        val i:[0..N-1];
        atomic { i = b; b++; }
        atomic { val v = c[i]; c[i] += N; return v; } } }
\(N\)-counter — Execution 2

```kotlin
class Counter<N:Int> {
    field b:[0..N-1] = 0;  // 1 balancer
    field c:Int[] = [0, 1, ..., N-1];  // N counters
    method getAndIncrement():Int {
        val i:[0..N-1];
        atomic { i = b; b++; }
        atomic { val v = c[i]; c[i] += N; return v; } } } 
```

\[
\langle b = 0, c = [0, 1]\rangle \xrightarrow{\text{inc}} \langle b = 1, c = [0, 1]\rangle \\
\langle b = 0, c = [0, 1]\rangle \xrightarrow{\text{inc}} \langle b = 0, c = [0, 3]\rangle \\
\langle b = 1, c = [0, 3]\rangle \xrightarrow{\text{inc}} \langle b = 1, c = [2, 3]\rangle \\
\langle b = 1, c = [2, 3]\rangle \xrightarrow{\text{inc}} \langle b = 1, c = [4, 3]\rangle \\
\]

Not Linearizable ☹️, but QQKC 😊
$N$-counter — Execution 2

```java
class Counter<N:Int> {
    field b: [0..N-1] = 0; // 1 balancer
    field c: Int[] = [0, 1, ..., N-1]; // N counters
    method getAndIncrement(): Int {
        val i: [0..N-1];
        atomic { i = b; b++; }
        atomic { val v = c[i]; c[i] += N; return v; } }
}
```

\[
\langle b = 0, c = [0, 1] \rangle \xrightarrow{\text{inc}} \langle b = 1, c = [0, 1] \rangle
\]

\[
\langle b = 0, c = [0, 1] \rangle \xrightarrow{\text{inc}} \langle b = 0, c = [0, 3] \rangle
\]

\[
\langle b = 0, c = [0, 3] \rangle \xrightarrow{\text{inc}} \langle b = 0, c = [2, 3] \rangle
\]

\[
\langle b = 0, c = [2, 3] \rangle \xrightarrow{\text{inc}} \langle b = 0, c = [4, 3] \rangle
\]

Not Linearizable 😞, but QQC 😊
$N$-counter — Execution 2

```kotlin
class Counter<N:Int> {
    field b:[0..N-1] = 0; // 1 balancer
    field c:Int[] = [0, 1, ..., N-1]; // N counters
    method getAndIncrement():Int {
        val i:[0..N-1];
        atomic { i = b; b++; }
        atomic { val v = c[i]; c[i] += N; return v; } } }
```

\[
\langle b = 0, c = [0, 1]\rangle \xrightarrow{\text{inc}} \langle b = 1, c = [0, 1]\rangle
\]

\[
\langle b = 0, c = [0, 1]\rangle \xrightarrow{\text{inc}} \langle b = 0, c = [0, 3]\rangle
\]

\[
\langle b = 1, c = [0, 3]\rangle \xrightarrow{\text{inc}} \langle b = 1, c = [2, 3]\rangle
\]

\[
\langle b = 1, c = [4, 3]\rangle
\]

Not Linearizable 😞, but QQC ☺️
Increment/Decrement counter

```kotlin
class Counter<N: Int> {
    field b: [0..N-1] = 0;       // 1 balancer
    field c: Int[] = [0, 1, ..., N-1]; // N counters
    method getAndIncrement(): Int {
        val i: [0..N-1];
        atomic { i = b; b++; }
        atomic { val v = c[i]; c[i] += N; return v; } }
    method decrementAndGet(): Int {
        val i: [0..N-1];
        atomic { i = b-1; b--; }
        atomic { c[i] -= N; return c[i]; }
    }
}
```

\[\langle b = 0, c = [0, 1] \rangle \xrightarrow{\text{inc}} \langle b = 1, c = [0, 1] \rangle \xrightarrow{\text{inc}} \langle b = 0, c = [0, 1] \rangle \]

\[\langle b = 0, c = [0, 1] \rangle \xrightarrow{\text{dec}} \langle b = 1, c = [0, 1] \rangle \xrightarrow{\text{dec}} \langle b = 0, c = [0, 1] \rangle \]

\[\langle b = 0, c = [-2, 1] \rangle \xrightarrow{\text{inc}} \langle b = 0, c = [0, 1] \rangle \]

\[\langle b = 0, c = [0, 3] \rangle \xrightarrow{\text{inc}} \langle b = 0, c = [0, 1] \rangle \]

\[\langle b = 0, c = [0, 1] \rangle \xrightarrow{\text{dec}} \langle b = 0, c = [0, 1] \rangle \]

\[\langle b = 0, c = [0, 1] \rangle \xrightarrow{\text{dec}} \langle b = 0, c = [0, 1] \rangle \]

Only weak QC 😊

not a permutation of any spec trace!
Increment/Decrement counter

class Counter<N:Int> {
    field b:[0..N-1] = 0; // 1 balancer
    field c:Int[] = [0, 1, ..., N-1]; // N counters
    method getAndIncrement():Int {
        val i:[0..N-1];
        atomic { i = b; b++; }
        atomic { val v = c[i]; c[i] += N; return v; } }
    method decrementAndGet():Int {
        val i:[0..N-1];
        atomic { i = b-1; b--; }
        atomic { c[i] -= N; return c[i]; } }
}

⟨
  b = 0, c = [0, 1]
⟩ \xrightarrow{\text{inc}} ⟨b = 1, c = [0, 1]⟩ \xrightarrow{\text{inc}} ⟨b = 0, c = [0, 1]⟩

⟨b = 0, c = [0, 1]⟩ \xrightarrow{\text{dec}} ⟨b = 1, c = [0, 1]⟩ \xrightarrow{\text{dec}} ⟨b = 0, c = [0, 1]⟩

⟨b = 0, c = [0, 1]⟩ \xrightarrow{\text{inc}} ⟨b = 0, c = [0, 3]⟩ \xrightarrow{\text{dec}} ⟨b = 0, c = [0, 1]⟩

Only weak QC 😊

Not a permutation of any spec trace!

\[ \text{dec} \quad \text{inc} \quad \text{inc} \quad \text{inc} \quad \text{dec} \]

\[ -2 \quad -2 \quad 1 \quad 1 \]
Increment/Decrement counter

```java
class Counter<N:Int> {
    field b:[0..N-1] = 0; // 1 balancer
    field c:Int[] = [0, 1, ..., N-1]; // N counters
    method getAndIncrement():Int {
        val i:[0..N-1];
        atomic { i = b; b++; }
        atomic { val v = c[i]; c[i] += N; return v; } }
    method decrementAndGet():Int {
        val i:[0..N-1];
        atomic { i = b-1; b--; }
        atomic { c[i] -= N; return c[i]; } }
}
```

Only weak QC 
not a permutation of any spec trace!
class Counter<N: Int> {
    field b: [0..N-1] = 0; // 1 balancer
    field c: Int[] = [0, 1, ..., N-1]; // N counters
    method getAndIncrement(): Int {
      val i: [0..N-1];
      atomic { i = b; b++; }
      atomic { val v = c[i]; c[i] += N; return v; } }
    method decrementAndGet(): Int {
      val i: [0..N-1];
      atomic { i = b-1; b--; }
      atomic { c[i] -= N; return c[i]; } }
  }

\<b = 0, c = [0, 1]\> \xrightarrow{\text{inc}} \langle b = 1, c = [0, 1]\rangle \xrightarrow{\text{inc}} \langle b = 0, c = [0, 1]\rangle
\langle b = 0, c = [0, 1]\rangle \xrightarrow{\text{dec}} \langle b = 1, c = [0, 1]\rangle \xrightarrow{\text{dec}} \langle b = 0, c = [0, 1]\rangle
\langle b = 0, c = [0, 1]\rangle \xrightarrow{\text{inc}} \langle b = 0, c = [0, 3]\rangle \xrightarrow{\text{dec}} \langle b = 0, c = [0, 1]\rangle
\langle b = 0, c = [0, 1]\rangle

Only weak QC 😊
not a permutation of any spec trace!
Increment/Decrement counter

```java
class Counter<N:Int> {
    field b:[0..N-1] = 0; // 1 balancer
    field c:Int[] = [0, 1, ..., N-1]; // N counters
    method getAndIncrement():Int {
        val i:[0..N-1];
        atomic { i = b; b++; }
        atomic { val v = c[i]; c[i] += N; return v; } }
    method decrementAndGet():Int {
        val i:[0..N-1];
        atomic { i = b-1; b--; }
        atomic { c[i] -= N; return c[i]; } }
}
```

The counter operates on two fields, `b` and `c`, where `b` is a single counter and `c` is an array of counters. The `getAndIncrement()` method increments `b` and returns the value from the corresponding `c[i]` field. The `decrementAndGet()` method decrements `b` and updates `c[i]` accordingly.

![Sequence diagram of counter operations]

Only weak QC ☹️

Not a permutation of any spec trace!
Increment/Decrement counter

```java
class Counter<N:Int> {
    field b:[0..N-1] = 0;  // 1 balancer
    field c:Int[] = [0, 1, ..., N-1];  // N counters
    method getAndIncrement():Int {
        val i:[0..N-1];
        atomic { i = b; b++; }
        atomic { val v = c[i]; c[i] += N; return v; } } } } 
```

```
\[
\begin{array}{c}
\langle b = 0, c = [0, 1]\rangle \xrightarrow{\text{inc}} \langle b = 1, c = [0, 1]\rangle \xrightarrow{\text{inc}} \langle b = 0, c = [0, 1]\rangle \\
\langle b = 1, c = [0, 1]\rangle \xrightarrow{\text{dec}} \langle b = 0, c = [0, 1]\rangle \xrightarrow{\text{dec}} \langle b = 0, c = [0, 1]\rangle \\
\langle b = 0, c = [-2, 1]\rangle \xrightarrow{\text{dec}} \langle b = 0, c = [0, 1]\rangle \xrightarrow{\text{inc}} \langle b = 0, c = [0, 1]\rangle \\
\langle b = 0, c = [0, 3]\rangle \xrightarrow{\text{dec}} \langle b = 0, c = [0, 1]\rangle \xrightarrow{\text{dec}} \langle b = 0, c = [0, 1]\rangle \\
\end{array}
\]
```

Only weak QC 😞 not a permutation of any spec trace!
Increment/Decrement counter

```java
class Counter<N: Int> {
    field b: [0..N-1] = 0; // 1 balancer
    field c: Int[] = [0, 1, ..., N-1]; // N counters
    method getAndIncrement(): Int {
        val i: [0..N-1];
        atomic { i = b; b++; }
        atomic { val v = c[i]; c[i] += N; return v; }
    }
    method decrementAndGet(): Int {
        val i: [0..N-1];
        atomic { i = b-1; b--; }
        atomic { c[i] -= N; return c[i]; }
    }
}
```

\[
\langle b = 0, c = [0, 1]\rangle \xrightarrow{\text{inc}} \langle b = 1, c = [0, 1]\rangle \xrightarrow{\text{inc}} \langle b = 0, c = [0, 1]\rangle
\]
\[
\xrightarrow{\text{dec}} \langle b = 1, c = [0, 1]\rangle \xrightarrow{\text{dec}} \langle b = 0, c = [0, 1]\rangle
\]
\[
\xrightarrow{\text{dec}_{-2}} \langle b = 0, c = [-2, 1]\rangle \xrightarrow{\text{inc}_{-2}} \langle b = 0, c = [0, 1]\rangle
\]
\[
\xrightarrow{\text{inc}} \langle b = 0, c = [0, 3]\rangle \xrightarrow{\text{dec}} \langle b = 0, c = [0, 1]\rangle
\]

Only weak QC 😞

not a permutation of any spec trace!
Increment/Decrement counter

class Counter<N:Int> {
    field b:[0..N-1] = 0; // 1 balancer
    field c:Int[] = [0, 1, ..., N-1]; // N counters
    method getAndIncrement():Int {
        val i:[0..N-1];
        atomic { i = b; b++; }
        atomic { val v = c[i]; c[i] += N; return v; } }
    method decrementAndGet():Int {
        val i:[0..N-1];
        atomic { i = b-1; b--; }
        atomic { c[i] -= N; return c[i]; } }
}

\begin{align*}
\langle b = 0, c = [0, 1] \rangle & \xrightarrow{\text{inc}} \langle b = 1, c = [0, 1] \rangle \\
& \xrightarrow{\text{dec}} \langle b = 1, c = [0, 1] \rangle \\
& \xrightarrow{\text{inc}} \langle b = 0, c = [0, 1] \rangle \\
& \xrightarrow{\text{dec}} \langle b = 0, c = [0, 1] \rangle \\
& \xrightarrow{\text{inc}} \langle b = 0, c = [0, 3] \rangle \\
& \xrightarrow{\text{dec}} \langle b = 0, c = [0, 1] \rangle \\
\end{align*}

Only weak QC 😊

Not a permutation of any spec trace!
Increment/Decrement counter

class Counter<N:Int> {
    field b:[0..N-1] = 0; // 1 balancer
    field c:Int[] = [0, 1, ..., N-1]; // N counters
    method getAndIncrement():Int {
        val i:[0..N-1];
        atomic { i = b; b++; }
        atomic { val v = c[i]; c[i] += N; return v; } }
    method decrementAndGet():Int {
        val i:[0..N-1];
        atomic { i = b-1; b--; }
        atomic { c[i] -= N; return c[i]; } }
}

\[
\langle b = 0, c = [0, 1] \rangle \xrightarrow{\text{inc}} \langle b = 1, c = [0, 1] \rangle \xrightarrow{\text{inc}} \langle b = 0, c = [0, 1] \rangle
\]

Only weak QC 😃
not a permutation of any spec trace!
class Stack<N:Int> {
  field b:[0..N-1] = 0;  // 1 balancer
  field s:Stack[] = [[], [], ..., []];  // N stacks of values
  method push(x:Object):Unit {
    val i:[0..N-1];
    atomic { i = b; b++; }
    atomic { val v = s[i].push(x); return v; } }
  method pop():Object {
    val i:[0..N-1];
    atomic { i = b-1; b--; }
    atomic { val v = s[i].pop(); return v; } }
}

\[\langle b = 0, s = [[]], [[]] \rangle \]
\[
\begin{align*}
\text{psh}_a &\rightarrow \langle b = 1, s = [[]], [[]] \rangle \\
\text{psh}_b &\rightarrow \langle b = 0, s = [[]], [[]] \rangle \\
\text{pop} &\rightarrow \langle b = 1, s = [[]], [[]] \rangle \\
\text{pop}_\text{fail} &\rightarrow \langle b = 0, s = [[]], [[]] \rangle \\
\text{psh} &\rightarrow \langle b = 0, s = [[a]], [[]] \rangle \\
\text{psh} &\rightarrow \langle b = 0, s = [[a], [b]] \rangle \\
\text{pop}_b &\rightarrow \langle b = 0, s = [[a], [[]] \rangle \\
\end{align*}
\]

Linearizable 😊
# Stack

```java
class Stack<N: Int> {
    field b: [0..N-1] = 0; // 1 balancer
    field s: Stack[] = [[]], [], ..., []]; // N stacks of values
    method push(x: Object): Unit {
        val i: [0..N-1];
        atomic { i = b; b++; }
        atomic { val v = s[i].push(x); return v; } }
    method pop(): Object {
        val i: [0..N-1];
        atomic { i = b-1; b--; }
        atomic { val v = s[i].pop(); return v; } }
}
```

\[
\begin{align*}
\langle b = 0, s = [[]], [] \rangle & \xrightarrow{[\text{psh}]} \langle b = 1, s = [[]], [] \rangle \xrightarrow{[\text{psh}]} \langle b = 0, s = [[]], [] \rangle \\
\langle b = 1, s = [[]], [] \rangle & \xrightarrow{[\text{pop}]} \langle b = 0, s = [[]], [] \rangle \\
\langle b = 0, s = [[]], [] \rangle & \xrightarrow{[\text{pop}]} \langle b = 0, s = [[]], [] \rangle \\
\langle b = 0, s = [[]], [] \rangle & \xrightarrow{[\text{pop}]} \langle b = 0, s = [[]], [] \rangle \\
\langle b = 0, s = [[a]], [] \rangle & \xrightarrow{[\text{psh}]} \langle b = 0, s = [[a]], [] \rangle \\
\langle b = 0, s = [[a]], [] \rangle & \xrightarrow{[\text{psh}]} \langle b = 0, s = [[a]], [] \rangle \\
\langle b = 0, s = [[a]], [b] \rangle & \xrightarrow{[\text{psh}]} \langle b = 0, s = [[a]], [b] \rangle \\
\langle b = 0, s = [[a]], [b] \rangle & \xrightarrow{[\text{psh}]} \langle b = 0, s = [[a]], [b] \rangle \\
\langle b = 0, s = [[a]], [b] \rangle & \xrightarrow{[\text{psh}]} \langle b = 0, s = [[a]], [b] \rangle \\
\langle b = 0, s = [[a]], [b] \rangle & \xrightarrow{[\text{psh}]} \langle b = 0, s = [[a]], [b] \rangle \\
\langle b = 0, s = [[a]], [b] \rangle & \xrightarrow{[\text{psh}]} \langle b = 0, s = [[a]], [b] \rangle \\
\end{align*}
\]
Stack

```java
class Stack<N:Int> {
    field b:[0..N-1] = 0; // 1 balancer
    field s:Stack[] = [[]], [[], ...], [[]]; // N stacks of values
    method push(x:Object):Unit {
        val i:[0..N-1];
        atomic { i = b; b++; }
        atomic { val v = s[i].push(x); return v; } }
    method pop():Object {
        val i:[0..N-1];
        atomic { i = b-1; b--; }
        atomic { val v = s[i].pop(); return v; } }
}
```

\[ \langle b = 0, s = [[]], [[]] \rangle \xrightarrow{[psh]_a} \langle b = 1, s = [[]], [[]] \rangle \xrightarrow{[psh]_b} \langle b = 0, s = [[]], [[]] \rangle \]

Between Linearizability and Quiescent Consistency
Stack

class Stack<N: Int> {
    field b: [0..N-1] = 0;  // 1 balancer
    field s: Stack[] = [[]], [], ..., []];  // N stacks of values

    method push(x: Object): Unit {
        val i: [0..N-1];
        atomic { i = b; b++; }
        atomic { val v = s[i].push(x); return v; } }

    method pop(): Object {
        val i: [0..N-1];
        atomic { i = b-1; b--; }
        atomic { val v = s[i].pop(); return v; } }
    }

    ⟨b = 0, s = [[]], []] ⟷[psh] b = 1, s = [[]], []] ⟷[psh] b = 0, s = [[]], []]

    ⟨b = 0, s = [[]], []] ⟷[pop] b = 1, s = [[]], []] ⟷[pop] b = 0, s = [[]], []]

    ⟨b = 0, s = [[]], []] ⟷[pop] b = 1, s = [[]], []] ⟷[pop] b = 0, s = [[]], []]

    ⟨b = 0, s = [[]], []] ⟷[pop] b = 1, s = [[]], []] ⟷[pop] b = 0, s = [[]], []]

    ⟨b = 0, s = [[]], []] ⟷[pop] b = 1, s = [[]], []] ⟷[pop] b = 0, s = [[]], []]

    ⟨b = 0, s = [[]], []] ⟷[pop] b = 1, s = [[]], []] ⟷[pop] b = 0, s = [[]], []]

    ⟨b = 0, s = [[]], []] ⟷[pop] b = 1, s = [[]], []] ⟷[pop] b = 0, s = [[]], []]

    ⟨b = 0, s = [[]], []] ⟷[pop] b = 1, s = [[]], []] ⟷[pop] b = 0, s = [[]], []]

    ⟨b = 0, s = [[]], []] ⟷[pop] b = 1, s = [[]], []] ⟷[pop] b = 0, s = [[]], []]

    ⟨b = 0, s = [[]], []] ⟷[pop] b = 1, s = [[]], []] ⟷[pop] b = 0, s = [[]], []]

    ⟨b = 0, s = [[]], []] ⟷[pop] b = 1, s = [[]], []] ⟷[pop] b = 0, s = [[]], []]
Stack

```java
class Stack<N:Int> {
  field b:[0..N-1] = 0; // 1 balancer
  field s:Stack[] = [[], [], ..., []]; // N stacks of values
  method push(x:Object):Unit {
    val i:[0..N-1];
    atomic { i = b; b++; }
    atomic { val v = s[i].push(x); return v; } }
  method pop():Object {
    val i:[0..N-1];
    atomic { i = b-1; b--; }
    atomic { val v = s[i].pop(); return v; } }
}
```

\[
\langle b = 0, s = [[], []] \rangle \xrightarrow{psh} \langle b = 1, s = [[], []] \rangle \xrightarrow{psh} \langle b = 0, s = [[], []] \rangle \\
\langle b = 0, s = [[], []] \rangle \xrightarrow{pop} \langle b = 1, s = [[], []] \rangle \xrightarrow{pop} \langle b = 0, s = [[], []] \rangle \\
\langle b = 0, s = [[], []] \rangle \xrightarrow{pop} \langle b = 0, s = [[], []] \rangle \xrightarrow{psh} \langle b = 0, s = [[a], []] \rangle \\
\langle b = 0, s = [[a], [b]] \rangle \xrightarrow{psh} \langle b = 0, s = [[a], [b]] \rangle \xrightarrow{pop} \langle b = 0, s = [[a], []] \rangle \\
\]

Linearizable 😊
class Stack<N:Int> {
    field b:[0..N-1] = 0;  // 1 balancer
    field s:Stack[] = [[], [], ..., []];  // N stacks of values
    method push(x:Object):Unit {
        val i:[0..N-1];
        atomic { i = b; b++; }
        atomic { val v = s[i].push(x); return v; } }
    method pop():Object {
        val i:[0..N-1];
        atomic { i = b-1; b--; }
        atomic { val v = s[i].pop(); return v; } }
}

\langle b = 0, s = [[], []] \rangle \xrightarrow{psh_a} \langle b = 1, s = [[], []] \rangle \xrightarrow{psh_b} \langle b = 0, s = [[], []] \rangle
\langle b = 1, s = [[], []] \rangle \xrightarrow{pop} \langle b = 0, s = [[], []] \rangle
\langle b = 0, s = [[], []] \rangle \xrightarrow{fail} \langle b = 0, s = [[], []] \rangle
\langle b = 0, s = [[], []] \rangle \xrightarrow{pop_a} \langle b = 0, s = [[a], []] \rangle
\langle b = 0, s = [[a], [b]] \rangle \xrightarrow{psh_b} \langle b = 0, s = [[a], [b]] \rangle
\langle b = 0, s = [[a], [b]] \rangle \xrightarrow{pop_b} \langle b = 0, s = [[a], []] \rangle

Linearizable 😊
class Stack<N: Int> {
    field b: [0..N-1] = 0; // 1 balancer
    field s: Stack[] = [[]], [], ..., []]; // N stacks of values
    method push(x: Object): Unit {
        val i: [0..N-1];
        atomic { i = b; b++; }
        atomic { val v = s[i].push(x); return v; } }
    method pop(): Object {
        val i: [0..N-1];
        atomic { i = b-1; b--; }
        atomic { val v = s[i].pop(); return v; } }
}

\[
\langle b = 0, s = [[]], [] \rangle \xrightarrow{\text{psh}_a} \langle b = 1, s = [[]], [] \rangle \xrightarrow{\text{psh}_b} \langle b = 0, s = [[]], [] \rangle
\]

\[
\langle b = 1, s = [[]], [] \rangle \xrightarrow{\text{pop}} \langle b = 0, s = [[]], [] \rangle \xrightarrow{\text{pop}} \langle b = 0, s = [[]], [] \rangle
\]

\[
\langle b = 0, s = [[]], [] \rangle \xrightarrow{\text{pop}} \langle b = 0, s = [[a]], [] \rangle \xrightarrow{\text{psh}_a} \langle b = 0, s = [[a]], [] \rangle
\]

\[
\langle b = 0, s = [[a], [b]] \rangle \xrightarrow{\text{pop}_b} \langle b = 0, s = [[a], []] \rangle
\]

Linearizable 🤪
Stack

```java
class Stack<N:Int> {
    field b:[0..N-1] = 0; // 1 balancer
    field s:Stack[] = [[], [], ..., []]; // N stacks of values

    method push(x:Object):Unit {
        val i:[0..N-1];
        atomic { i = b; b++; }
        atomic { val v = s[i].push(x); return v; } }

    method pop():Object {
        val i:[0..N-1];
        atomic { i = b-1; b--; }
        atomic { val v = s[i].pop(); return v; } }
}
```

Between Linearizability and Quiescent Consistency

Linearizable 😊
Stack

class Stack<N:Int> {
    field b:[0..N-1] = 0; // 1 balancer
    field s:Stack[] = [[], [], ..., []]; // N stacks of values
    method push(x:Object):Unit {
        val i:[0..N-1];
        atomic { i = b; b++; }
        atomic { val v = s[i].push(x); return v; } }
    method pop():Object {
        val i:[0..N-1];
        atomic { i = b-1; b--; }
        atomic { val v = s[i].pop(); return v; } }
}

⟨b = 0, s = [[]], []⟩ \xrightarrow{\text{psh}_a} ⟨b = 1, s = [[]], []⟩ \xrightarrow{\text{psh}_b} ⟨b = 0, s = [[]], []⟩
\xrightarrow{\text{pop}} ⟨b = 1, s = [[]], []⟩ \xrightarrow{\text{pop}} ⟨b = 0, s = [[]], []⟩
\xrightarrow{\text{fail}} ⟨b = 0, s = [[]], []⟩ \xrightarrow{\text{psh}} ⟨b = 0, s = [[a]], []⟩
\xrightarrow{\text{psh}} ⟨b = 0, s = [[a], [b]]⟩ \xrightarrow{\text{pop}_b} ⟨b = 0, s = [[a]], []⟩

b=0
/ \ 
[] a s[0]=a s[1]=[]
\xrightarrow{\text{psh}_a} \xrightarrow{\text{pop}_b} \xrightarrow{\text{psh}_b} \xrightarrow{\text{pop}} \xrightarrow{\text{fail}} \xrightarrow{\text{psh}} \xrightarrow{\text{pop}_b}

Linearizable 😊
Stack — Execution 2

```kotlin
class Stack<N:Int> {
    field b:[0..N-1] = 0; // 1 balancer
    field s:Stack[] = [[], [], ..., []]; // N stacks of values
    method push(x:Object):Unit {
        val i:[0..N-1];
        atomic { i = b; b++; }
        atomic { val v = s[i].push(x); return v; } } } // N stacks

⟨b = 0, s = [[], []]⟩ ↘↓ psh a ↘↓ psh b ↘↓ psh c ↘↓ pop a ↘↓ pop b ↘↓ psh d ↘↓ (Not even quiescent consistent 😊)
```

Not even quiescent consistent 😊

 vợ chồng nên tập trung vào vấn đề của mình sau khi kết thúc việc làm.
Stack — Execution 2

class Stack<N:Int> {
    field b:[0..N-1] = 0; // 1 balancer
    field s:Stack[] = [[]], [], ..., []]; // N stacks of values
    method push(x:Object):Unit {
        val i:[0..N-1];
        atomic { i = b; b++; }
        atomic { val v = s[i].push(x); return v; } }
    method pop():Object {
        val i:[0..N-1];
        atomic { i = b-1; b--; }
        atomic { val v = s[i].pop(); return v; } }
}

⟨b = 0, s = [[]], [[]]⟩ \xrightarrow{psh} ⟨b = 1, s = [[]], [[]]⟩ \xrightarrow{psh} ⟨b = 1, s = [[a], [[]]]⟩

b=1
/ \ 
\begin{array}{c}
\text{s[0]} = \text{[psh} \text{a} \\
\text{s[1]} = \text{[psh} \text{c} \\
\end{array}

Not even quiescent consistent 😊

\longleftrightarrow \text{should pop from } \longleftrightarrow \text{ or } \longleftrightarrow, \text{ but not } \longleftrightarrow
class Stack<N:Int> {
    field b:[0..N-1] = 0; // 1 balancer
    field s:Stack[] = [[], [], ..., []]; // N stacks of values
    method push(x:Object):Unit {
        val i:[0..N-1];
        atomic { i = b; b++; }
        atomic { val v = s[i].push(x); return v; } }
    method pop():Object {
        val i:[0..N-1];
        atomic { i = b-1; b--; }
        atomic { val v = s[i].pop(); return v; } }
}

⟨
b = 0, s = [[]], []⟩ \xrightarrow{\text{psh}^a} ⟨b = 1, s = [[]], []⟩ \xrightarrow{\text{psh}^j} ⟨b = 1, s = [[a]], []⟩

\begin{align*}
&\xrightarrow{\{\text{psh}^b\}} ⟨b = 0, s = [[a]], []⟩ \xrightarrow{\text{psh}^j} ⟨b = 0, s = [[a], [b]]⟩ \\
&\xrightarrow{\{\text{psh}^c\}} ⟨b = 1, s = [[a], [b]]⟩ \xrightarrow{\text{pop}^g} ⟨b = 0, s = [[a], [b]]⟩ \\
&\xrightarrow{\text{pop}^a} ⟨b = 0, s = [[], [b]]⟩ \xrightarrow{\text{psh}^j} ⟨b = 0, s = [[c], [b]]⟩
\end{align*}

Not even quiescent consistent 😊

\text{←→ should pop from } ←→ \text{ or } ←→, \text{ but not } ←→
Stack — Execution 2

```java
class Stack<N:Int> {
  field b:[0..N-1] = 0; // 1 balancer
  field s:Stack[] = [[], [], ..., []]; // N stacks of values
  method push(x:Object):Unit {
    val i:[0..N-1];
    atomic { i = b; b++; }
    atomic { val v = s[i].push(x); return v; } }
  method pop():Object {
    val i:[0..N-1];
    atomic { i = b-1; b--; }
    atomic { val v = s[i].pop(); return v; } }
}
```

\[
\begin{align*}
\langle b = 0, s = [[]], [] \rangle & \xrightarrow{psh_a} \langle b = 1, s = [[]], [] \rangle \xrightarrow{psh}\langle b = 1, s = [a], [] \rangle \\
\langle b = 0, s = [a], [] \rangle & \xrightarrow{psh_b} \langle b = 0, s = [a], [a] \rangle \\
\langle b = 1, s = [a], [b] \rangle & \xrightarrow{psh}\langle b = 0, s = [a], [b] \rangle \\
\langle b = 1, s = [a], [b] \rangle & \xrightarrow{pop}\langle b = 0, s = [a], [b] \rangle \\
\langle b = 0, s = [\_], [b] \rangle & \xrightarrow{psh}\langle b = 0, s = [c], [b] \rangle \\
\end{align*}
\]

Not even quiescent consistent 😐

Should pop from ← or →, but not ←
class Stack<N:Int> {
    field b:[0..N-1] = 0; // 1 balancer
    field s:Stack[] = [[], [], ..., []]; // N stacks of values
    method push(x:Object):Unit {
        val i:[0..N-1];
        atomic { i = b; b++; }
        atomic { val v = s[i].push(x); return v; } }
    method pop():Object {
        val i:[0..N-1];
        atomic { i = b-1; b--; }
        atomic { val v = s[i].pop(); return v; } }
}

\[
\begin{align*}
  \langle b = 0, s = [[], []] \rangle & \xrightarrow{psh} \langle b = 1, s = [[], []] \rangle \\
  \langle b = 0, s = [[], []] \rangle & \xrightarrow{\{psh\}} \langle b = 1, s = [[a], []] \rangle \\
  \langle b = 0, s = [[a], []] \rangle & \xrightarrow{psh} \langle b = 0, s = [[a], [b]] \rangle \\
  \langle b = 1, s = [[a], [b]] \rangle & \xrightarrow{\langle pop \rangle} \langle b = 0, s = [[a], [b]] \rangle \\
  \langle b = 0, s = [[a], [b]] \rangle & \xrightarrow{\langle pop \rangle} \langle b = 0, s = [[c], [b]] \rangle \\
\end{align*}
\]

Not even quiescent consistent 😞

← should pop from ← or →, but not ←
Stack — Execution 2

```java
class Stack<N: Int> {
    field b: [0..N-1] = 0;  // 1 balancer
    field s: Stack[] = [[]] // N stacks of values
    method push(x: Object): Unit {
        val i: [0..N-1];
        atomic { i = b; b++; }
        atomic { val v = s[i].push(x); return v; }
    }
    method pop(): Object {
        val i: [0..N-1];
        atomic { i = b-1; b--; }
        atomic { val v = s[i].pop(); return v; }
    }

    ⟨b = 0, s = [[]]⟩ →[psh] ⟨b = 1, s = [[]]⟩ →[psh] ⟨b = 1, s = [[a]]⟩
    \langle b = 0, s = [[]] \rangle \xrightarrow{psh}[a] \langle b = 1, s = [[a]] \rangle \xrightarrow{psh} \langle b = 1, s = [[a], [a]] \rangle
    \langle b = 0, s = [[]] \rangle \xrightarrow{psh}[b] \langle b = 1, s = [[a], [a]] \rangle \xrightarrow{psh} \langle b = 1, s = [[a], [b]] \rangle
    \langle b = 0, s = [[]] \rangle \xrightarrow{psh}[c] \langle b = 1, s = [[a], [b]] \rangle \xrightarrow{pop} \langle b = 0, s = [[a], [b]] \rangle
    \langle b = 0, s = [[]] \rangle \xrightarrow{pop}[a] \langle b = 0, s = [__, [b]] \rangle \xrightarrow{psh} \langle b = 0, s = [__, [b]] \rangle
    \langle b = 0, s = [[]] \rangle \xrightarrow{pop}[b] \langle b = 0, s = [__, [b]] \rangle \xrightarrow{psh} \langle b = 0, s = [__, [b]] \rangle
    \langle b = 0, s = [[]] \rangle \xrightarrow{pop}[c] \langle b = 0, s = [__, [b]] \rangle \xrightarrow{psh} \langle b = 0, s = [__, [b]] \rangle

    Not even quiescent consistent 😊

    Should pop from [→→] or [→→], but not [←←]
}
```

Not even quiescent consistent 😊

Should pop from [→→] or [→→], but not [←←]
Stack — Execution 2

class Stack<N: Int> {
    field b: [0..N-1] = 0; // 1 balancer
    field s: Stack[] = [[], [], ..., []]; // N stacks of values
    method push(x: Object): Unit {
        val i: [0..N-1];
        atomic { i = b; b++; }
        atomic { val v = s[i].push(x); return v; }
    }
    method pop(): Object {
        val i: [0..N-1];
        atomic { i = b-1; b--; }
        atomic { val v = s[i].pop(); return v; }
    }
}

⟨ b = 0, s = [[], []] ⟩ \xrightarrow{[a]} ⟨ b = 1, s = [[], []] ⟩ \xrightarrow{[psh]} ⟨ b = 1, s = [[a], []] ⟩
\xrightarrow{[b]} ⟨ b = 0, s = [[a], []] ⟩ \xrightarrow{[psh]} ⟨ b = 0, s = [[a], [b]] ⟩
\xrightarrow{[c]} ⟨ b = 1, s = [[a], [b]] ⟩ \xrightarrow{[psh]} ⟨ b = 0, s = [[a], [b]] ⟩
\xrightarrow{[a]} ⟨ b = 0, s = [[a], [b]] ⟩ \xrightarrow{[psh]} ⟨ b = 0, s = [[c], [b]] ⟩
Not even quiescent consistent 😊

\xrightarrow{←} should pop from \xrightarrow{←} or \xrightarrow{←}, but not \xrightarrow{←}
Stack — Execution 2

class Stack<N:Int> {
    field b:[0..N-1] = 0;  // 1 balancer
    field s:Stack[] = [[], [], ..., []];  // N stacks of values
    method push(x:Object):Unit {
        val i:[0..N-1];
        atomic { i = b; b++; }
        atomic { val v = s[i].push(x); return v; } }
    method pop():Object {
        val i:[0..N-1];
        atomic { i = b-1; b--; }
        atomic { val v = s[i].pop(); return v; } } }

\[
\begin{align*}
\langle b = 0, s = [[], []] \rangle & \xrightarrow{psh_a} \langle b = 1, s = [[], []] \rangle & \xrightarrow{psh_a} \langle b = 1, s = [[a], []] \rangle \\
\langle b = 0, s = [[a], []] \rangle & \xrightarrow{psh_b} \langle b = 0, s = [[a], [b]] \rangle \\
\langle b = 1, s = [[a], [b]] \rangle & \xrightarrow{pop_c} \langle b = 0, s = [[a], [b]] \rangle \\
\langle b = 0, s = [[a], [b]] \rangle & \xrightarrow{psh_b} \langle b = 0, s = [[c], [b]] \rangle \\
\end{align*}
\]

Not even quiescent consistent 😐

← should pop from → or ←, but not →
class Stack<N:Int> {
    field b:[0..N-1] = 0;  // 1 balancer
    field s:Stack[] = [[]], [[], ...], [[]];  // N stacks of values
    method push(x:Object):Unit {
        val i:[0..N-1];
        atomic { i = b; b++; }
        atomic { val v = s[i].push(x); return v; } }
    method pop():Object {
        val i:[0..N-1];
        atomic { i = b-1; b--; }
        atomic { val v = s[i].pop(); return v; } }
}

⟨b = 0, s = [[]], [[]]⟩ \xrightarrow{\text{psh}} ⟨b = 1, s = [[]], [[]]⟩ \xrightarrow{\text{psh}} ⟨b = 1, s = [[a]], [[]]⟩
\xrightarrow{\text{psh}} ⟨b = 0, s = [[a]], [[]]⟩ \xrightarrow{\text{psh}} ⟨b = 0, s = [[a], [b]]⟩
\xrightarrow{\text{psh}} ⟨b = 0, s = [[a], [b]]⟩ \xrightarrow{\text{pop}} ⟨b = 0, s = [[a], [b]]⟩
\xrightarrow{\text{pop}} ⟨b = 0, s = [[b]], [b]]⟩ \xrightarrow{\text{psh}} ⟨b = 0, s = [[], [b]]⟩

Not even quiescent consistent ☹

\[\text{\longleftrightarrow should pop from \{} or \text{\longleftrightarrow}, but not \text{\{}\]}\]
Results

Three characterizations of QQC

- Call-to-return  
- Return-to-call  (à la Herlihy/Wing)
- Proxy for sequential implementation  (flat combiner + speculation)
  - Single thread accesses sequential structure
  - Upon receiving actual call, speculatively execute any method with any args
  - Only return when speculative call matches actual call

Proof of compositionality

- Global constraints that are solvable because of “flow” properties

Proofs and counterexamples for tree-based structures

- Increment/decrement $N$-counter  (weak QC)
- Increment-only $N$-counter  (QQC)
- General $N$-stack  (QC)
- “Properly popped” $N$-stack  (QQC)
  - Pop must wait for concurrent push on same underlying stack
  - Not sufficient for pop to wait on empty stack
  - Proof uses instrumented $N$-stack that emits a QQC specification trace

- Tree of $N$-stacks  (same as single $N$-stack)
Results

- Three characterizations of QQC
  - Call-to-return (given earlier)
  - Return-to-call (à la Herlihy/Wing)
  - Proxy for sequential implementation (flat combiner + speculation)
    - Single thread accesses sequential structure
    - Upon receiving actual call, speculatively execute any method with any args
    - Only return when speculative call matches actual call

- Proof of compositionality
  - Global constraints that are solvable because of “flow” properties

- Proofs and counterexamples for tree-based structures
  - Increment/decrement $N$-counter (weak QC)
  - Increment-only $N$-counter (QQC)
  - General $N$-stack (QC)
  - “Properly popped” $N$-stack (QQC)
    - Pop must wait for concurrent push on same underlying stack
    - Not sufficient for pop to wait on empty stack
    - Proof uses instrumented $N$-stack that emits a QQC specification trace

- Tree of $N$-stacks (same as single $N$-stack)
Return-to-call characterization

Linearizability:
\[ \forall \text{prefix/ suffix } = \text{exec} \]
\[ \forall \text{ret } \in \text{prefix} \]
\[ \forall \text{call } \in \text{suffix} \]
\[ \text{ret } \xrightarrow{\text{exec}} \text{call} \quad \text{implies} \quad \text{ret } \xrightarrow{\text{spec}} \text{call} \]
Return-to-call characterization

Quiescent consistency:

∀ prefix/suffix = exec

if prefix has 0 open calls, then

∀ ret ∈ prefix

∀ call ∈ suffix

\[ \text{ret } \xrightarrow{\text{exec}} \text{call} \quad \text{implies} \quad \text{ret } \xrightarrow{\text{spec}} \text{call} \]
Return-to-call characterization

QQC:
∀prefix/suffix = exec
if prefix has \( k \) open/early calls, then there exists \( \left| \text{ignoredCalls} \right| \leq k \)
∀ret ∈ prefix
∀call ∈ suffix − ignoredCalls

\[
ret \xrightarrow{\text{exec}} call \quad \text{implies} \quad ret \xrightarrow{\text{spec}} call
\]
Results

- Three characterizations of QQC
  - Call-to-return (given earlier)
  - Return-to-call (à la Herlihy/Wing)
  - Proxy for sequential implementation (flat combiner + speculation)
    - Single thread accesses sequential structure
    - Upon receiving actual call, speculatively execute any method with any args
    - Only return when speculative call matches actual call

- Proof of compositionality
  - Global constraints that are solvable because of “flow” properties

- Proofs and counterexamples for tree-based structures
  - Increment/decrement $N$-counter (weak QC)
  - Increment-only $N$-counter (QQC)
  - General $N$-stack (QC)
  - “Properly popped” $N$-stack (QQC)
    - Pop must wait for concurrent push on same underlying stack
    - Not sufficient for pop to wait on empty stack
    - Proof uses instrumented $N$-stack that emits a QQC specification trace
  - Tree of $N$-stacks (same as single $N$-stack)
Results

- Three characterizations of QQC
  - Call-to-return 
  - Return-to-call 
  - Proxy for sequential implementation 
    - Single thread accesses sequential structure 
    - Upon receiving actual call, speculatively execute any method with any args 
    - Only return when speculative call matches actual call 

- Proof of compositionality
  - Global constraints that are solvable because of “flow” properties 

- Proofs and counterexamples for tree-based structures
  - Increment/decrement \( N \)-counter (weak QC) 
  - Increment-only \( N \)-counter (QQC) 
  - General \( N \)-stack (QC) 
  - “Properly popped” \( N \)-stack (QQC) 
    - Pop must wait for concurrent push on same underlying stack 
    - Not sufficient for pop to wait on empty stack 
    - Proof uses instrumented \( N \)-stack that emits a QQC specification trace 

- Tree of \( N \)-stacks (same as single \( N \)-stack)
Results

- Three characterizations of QQC
  - Call-to-return (given earlier)
  - Return-to-call (à la Herlihy/Wing)
  - Proxy for sequential implementation (flat combiner + speculation)
    - Single thread accesses sequential structure
    - Upon receiving actual call, speculatively execute any method with any args
    - Only return when speculative call matches actual call

- Proof of compositionality
  - Global constraints that are solvable because of “flow” properties

- Proofs and counterexamples for tree-based structures
  - Increment/decrement $N$-counter (weak QC)
  - Increment-only $N$-counter (QQC)
  - General $N$-stack (QC)
  - “Properly popped” $N$-stack (QQC)
    - Pop must wait for concurrent push on same underlying stack
    - Not sufficient for pop to wait on empty stack
    - Proof uses instrumented $N$-stack that emits a QQC specification trace
  - Tree of $N$-stacks (same as single $N$-stack)
Results

- Three characterizations of QQC
  - Call-to-return (given earlier)
  - Return-to-call (à la Herlihy/Wing)
  - Proxy for sequential implementation (flat combiner + speculation)
    - Single thread accesses sequential structure
    - Upon receiving actual call, speculatively execute any method with any args
    - Only return when speculative call matches actual call

- Proof of compositionality
  - Global constraints that are solvable because of “flow” properties

- Proofs and counterexamples for tree-based structures
  - Increment/decrement \( N \)-counter (weak QC)
  - Increment-only \( N \)-counter (QQC)
  - General \( N \)-stack (QC)
  - “Properly popped” \( N \)-stack (QQC)
    - Pop must wait for concurrent push on same underlying stack
    - Not sufficient for pop to wait on empty stack
    - Proof uses instrumented \( N \)-stack that emits a QQC specification trace

- Tree of \( N \)-stacks (same as single \( N \)-stack)
Results

- Three characterizations of QQC
  - Call-to-return (given earlier)
  - Return-to-call (à la Herlihy/Wing)
  - Proxy for sequential implementation (flat combiner + speculation)
    - Single thread accesses sequential structure
    - Upon receiving actual call, speculatively execute any method with any args
    - Only return when speculative call matches actual call

- Proof of compositionality
  - Global constraints that are solvable because of “flow” properties

- Proofs and counterexamples for tree-based structures
  - Increment/decrement $N$-counter (weak QC)
  - Increment-only $N$-counter (QQC)
  - General $N$-stack (QC)
  - “Properly popped” $N$-stack (QQC)
    - Pop must wait for concurrent push on same underlying stack
    - Not sufficient for pop to wait on empty stack
    - Proof uses instrumented $N$-stack that emits a QQC specification trace

- Tree of $N$-stacks (same as single $N$-stack)
Results

- Three characterizations of QQC
  - Call-to-return (given earlier)
  - Return-to-call (à la Herlihy/Wing)
  - Proxy for sequential implementation (flat combiner + speculation)
    - Single thread accesses sequential structure
    - Upon receiving actual call, speculatively execute any method with any args
    - Only return when speculative call matches actual call

- Proof of compositionality
  - Global constraints that are solvable because of “flow” properties

- Proofs and counterexamples for tree-based structures
  - Increment/decrement $N$-counter (weak QC)
  - Increment-only $N$-counter (QQC)
  - General $N$-stack (QC)
  - “Properly popped” $N$-stack (QQC)
    - Pop must wait for concurrent push on same underlying stack
    - Not sufficient for pop to wait on empty stack
    - Proof uses instrumented $N$-stack that emits a QQC specification trace

- Tree of $N$-stacks (same as single $N$-stack)
Results

- Three characterizations of QQC
  - Call-to-return
  - Return-to-call
  - Proxy for sequential implementation
    - Single thread accesses sequential structure
    - Upon receiving actual call, speculatively execute any method with any args
    - Only return when speculative call matches actual call

- Proof of compositionality
  - Global constraints that are solvable because of “flow” properties

- Proofs and counterexamples for tree-based structures
  - Increment/decrement $N$-counter
  - Increment-only $N$-counter
  - General $N$-stack
  - “Properly popped” $N$-stack
    - Pop must wait for concurrent push on same underlying stack
    - Not sufficient for pop to wait on empty stack
    - Proof uses instrumented $N$-stack that emits a QQC specification trace

- Tree of $N$-stacks
Results

- Three characterizations of QQC
  - Call-to-return (given earlier)
  - Return-to-call (à la Herlihy/Wing)
  - Proxy for sequential implementation (flat combiner + speculation)
    - Single thread accesses sequential structure
    - Upon receiving actual call, speculatively execute any method with any args
    - Only return when speculative call matches actual call

- Proof of compositionality
  - Global constraints that are solvable because of “flow” properties

- Proofs and counterexamples for tree-based structures
  - Increment/decrement $N$-counter (weak QC)
  - Increment-only $N$-counter (QQC)
  - General $N$-stack (QC)
  - “Properly popped” $N$-stack (QQC)
    - Pop must wait for concurrent push on same underlying stack
    - Not sufficient for pop to wait on empty stack
    - Proof uses instrumented $N$-stack that emits a QQC specification trace
  - Tree of $N$-stacks (same as single $N$-stack)
Related work

- **Quantitative Relaxation of Concurrent Data Structures**
  (Henzinger/Kirsch/Payer/Sezgin/Sokolova 2013)

- Incomparable

  - Stack that is 1-out-of-order but not QQC:
    $$(\text{psh} \ [a] \ \text{psh} \ [b] \ \text{psh} \ < \text{pop} \ >_a)$$
    However,
    $$(\text{psh} \ [a] \ \text{psh} \ [b] \ \text{psh} \ < \text{pop} \ >_a \ \text{psh} \ [c])$$
    is QQC w.r.t. the stack spec
    $$(\text{psh} \ [a] \ \text{psh} \ [b] \ \text{psh} \ < \text{pop} \ >_a \ (\text{psh} \ [c]))$$

- For stacks, it may be that QQC is finer that $n$-out-of-order (arbitrary $n$)

- Queue that is QQC but not $(n - 1)$-out-of-order:
  $$(\text{psh} \ [a_1] \ \text{psh} \ [a_1] \ \text{psh} \ \ldots \ \text{psh} \ [a_n] \ \text{psh} \ < \text{pop} \ >_c \ \text{psh} \ [c])$$
  This is QQC w.r.t.
  $$(\text{psh} \ [a_1] \ \text{psh} \ [a_1] \ \text{psh} \ \ldots \ \text{psh} \ [a_n] \ \text{psh} \ < \text{pop} \ >_c \ \text{psh} \ [a])$$
Related work

- **Quantitative Relaxation of Concurrent Data Structures**
  (Henzinger/Kirsch/Payer/Sezgin/Sokolova 2013)

- Incomparable
  - Stack that is 1-out-of-order but not QQC:
    \[
    (\text{psh}_c [\text{psh}_a] \text{psh} \{\text{psh}_b\} \text{psh} < \text{pop}_a > \text{pop}_a)
    \]
    However,
    \[
    (\text{psh}_c [\text{psh}_a] \text{psh} \{\text{psh}_b\} \text{psh} < \text{pop}_a > \text{pop}_a) \text{psh}
    \]
    is QQC w.r.t. the stack spec
    \[
    \{\text{psh}_b\} \text{psh} [\text{psh}_a] \text{psh} < \text{pop}_a > \text{pop}_a (\text{psh}_c) \text{psh}
    \]

- For stacks, it may be that QQC is finer that \(n\)-out-of-order (arbitrary \(n\))

- Queue that is QQC but not \((n - 1)\)-out-of-order:
  \[
  (\text{psh}_a [\text{psh}_{b_1}] \text{psh} [\text{psh}_{b_1}] \text{psh} \ldots [\text{psh}_{b_n}] \text{psh} \{\text{psh}_c\} \text{psh} < \text{pop}_c > \text{pop}_c) \text{psh}
  \]
  This is QQC w.r.t.
  \[
  \{\text{psh}_c\} \text{psh} [\text{psh}_{b_1}] \text{psh} [\text{psh}_{b_1}] \text{psh} \ldots [\text{psh}_{b_n}] \text{psh} < \text{pop}_c > \text{pop}_c (\text{psh}_a) \text{psh}
  \]
Related work

- **Quantitative Relaxation of Concurrent Data Structures**
  (Henzinger/Kirsch/Payer/Sezgin/Sokolova 2013)
  (Examples from Sezgin)

- **Incomparable**
  - Stack that is 1-out-of-order but not QQ:
    \[
    (\text{psh}_c \ [\text{psh}_a \ ]\text{psh} \ \{\text{psh}_b \} \text{psh} < \text{pop} > \text{pop}_a)
    \]
  
  However,
  \[
  (\text{psh}_c \ [\text{psh}_a \ ]\text{psh} \ \{\text{psh}_b \} \text{psh} < \text{pop} > \text{pop}_a \) \text{psh}
  \]
  is QQ w.r.t. the stack spec
  \[
  \{\text{psh}_b \} \text{psh} [\text{psh}_a \] \text{psh} < \text{pop} > \text{pop}_a \ (\text{psh}_c \) \text{psh}
  \]

- For stacks, it may be that QQ is finer than $n$-out-of-order (arbitrary $n$)
- Queue that is QQ but not $(n - 1)$-out-of-order:
  \[
  (\text{psh}_a \ [\text{psh}_{b_1} \ ]\text{psh} [\text{psh}_{b_1} \ ]\text{psh} \ldots [\text{psh}_{b_n} \ ]\text{psh} \ \{\text{psh}_c \} \text{psh} < \text{pop} > \text{pop}_c \) \text{psh}
  \]
  This is QQ w.r.t.
  \[
  \{\text{psh}_c \} \text{psh} [\text{psh}_{b_1} \ ]\text{psh} [\text{psh}_{b_1} \ ]\text{psh} \ldots [\text{psh}_{b_n} \ ]\text{psh} < \text{pop} > \text{pop}_c \ (\text{psh}_a \) \text{psh}
  \]
Related work

- **Quantitative Relaxation of Concurrent Data Structures**
  (Henzinger/Kirsch/Payer/Sezgin/Sokolova 2013)
  (Examples from Sezgin)

- Incomparable
  Stack that is 1-out-of-order but not QQ:
  $$\langle psh_c \ [psh_a \ ]psh \ {psh_b \ }psh \ psh < pop > pop_a \rangle$$

  However, $$\langle psh_c \ [psh_a \ ]psh \ {psh_b \ }psh < pop > pop_a \ psh \rangle$$ is QQ w.r.t. the stack spec

  $$\{ psh_b \ psh \ [psh_a \ ]psh < pop > pop_a \ (psh_c \ psh) \rangle$$

- For stacks, it may be that QQ is finer than n-out-of-order (arbitrary n)

- Queue that is QQ but not (n – 1)-out-of-order:
  $$\langle psh_a \ [psh_{b_1} \ ]psh \ [psh_{b_1} \ ]psh \ … \ [psh_{b_n} \ ]psh \ {psh_c \ psh < pop > pop_c \ psh \rangle$$

  This is QQ w.r.t.

  $$\{ psh_c \ psh \ [psh_{b_1} \ ]psh \ [psh_{b_1} \ ]psh \ … \ [psh_{b_n} \ ]psh < pop > pop_c \ (psh_a \ psh \rangle$$
Proxy characterization code

interface Object {
    method run(i: Invocation): Response;
    method predict(): Invocation; }

class QQCProxy<o:Object> {
    field called: ThreadSafeMultiMap<Invocation, Semaphore> = [];
    field returned: ThreadSafeMap <Semaphore, Response> = [];

    method run(i: Invocation): Response {
        // proxy for external access to o
        val m: Semaphore = [];
        called.add(i, m);
        m.wait();
        return returned.remove(m); }

    thread {
        // single thread to interact with o
        val received: MultiMap<Invocation, Semaphore> = [];
        val executed: MultiMap<Invocation, Response> = [];
        repeatedly choose {
            choice if called.notEmpty() {
                received.add(called.removeAny());
                val i: Invocation = o.predict();
                val r: Response = o.run(i);
                executed.add(i, r); } }

            choice if exists i in received.keys() intersect executed.keys() {
                val m: Semaphore = received.remove(i);
                val r: Response = executed.remove(i);
                returned.add(m, r);
                m.signal(); } } } }