Concurrent Libraries

Correctness Criteria, Verification

Verification Ingredients

- Specifying a Library: φ
- Implementing a Library: $\mathbb{L}$
- Verifying a Library implementation: $\mathbb{L} \models \varphi$

The History of an Object

Object Specification

- How can we specify an object? (Library)
  - Objects API
  - Use cases
  - Pre and Post Conditions?

java.util

Class Stack<E>

java.lang.Object
  java.util.AbstractCollection<E>
    java.util.AbstractList<E>
      java.util.Vector<E>
        java.util.Stack<E>

Method Summary

Methods

Modifier and Type	Method and Description
`boolean`	`empty()` Tests if this stack is empty.
`E`	`peek()` Looks at the object at the top of this stack without removing it from the stack.
`E`	`pop()` Removes the object at the top of this stack and returns that object as the value of this function.
`E`	`push(E item)` Pushes an item onto the top of this stack.
`int`	`search(Object o)` Returns the 1-based position where an object is on this stack.

Methods inherited from class java.util.Vector

`add, add, addAll, addAll, addElement, capacity, clear, clone, contains, containsAll,`

Object Specification

- How can we specify an object? (Library)
  - Objects API
  - Use cases
  - Pre and Post Conditions?
- What are the behaviors of a client using the library?

java.util

Class Stack<E>

java.lang.Object
  java.util.AbstractCollection<E>
    java.util.AbstractList<E>
      java.util.Vector<E>
        java.util.Stack<E>

Method Summary

Methods

Modifier and Type	Method and Description
`boolean`	`empty()` Tests if this stack is empty.
`E`	`peek()` Looks at the object at the top of this stack without removing it from the stack.
`E`	`pop()` Removes the object at the top of this stack and returns that object as the value of this function.
`E`	`push(E item)` Pushes an item onto the top of this stack.
`int`	`search(Object o)` Returns the 1-based position where an object is on this stack.

Methods inherited from class java.util.Vector

`add, add, addAll, addAll, addElement, capacity, clear, clone, contains, containsAll,`

Object Specification

- How can we specify an object? (Library)
  - Objects API
  - Use cases
  - Pre and Post Conditions?
- What are the behaviors of a client using the library?

💡 for any client making library calls record the inputs and outputs of each call

java.util

Class Stack<E>

java.lang.Object
    java.util.AbstractCollection<E>
        java.util.AbstractList<E>
            java.util.Vector<E>
                java.util.Stack<E>

Method Summary

Methods

Modifier and Type	Method and Description
`boolean`	`empty()` Tests if this stack is empty.
`E`	`peek()` Looks at the object at the top of this stack without removing it from the stack.
`E`	`pop()` Removes the object at the top of this stack and returns that object as the value of this function.
`E`	`push(E item)` Pushes an item onto the top of this stack.
`int`	`search(Object o)` Returns the 1-based position where an object is on this stack.

Methods inherited from class java.util.Vector

`add, add, addAll, addAll, addElement, capacity, clear, clone, contains, containsAll,`
`copyInto, elementAt, elements, ensureCapacity, equals, firstElement, get, hashCode`

Implementation of a Queue

```
class LockBasedQueue<T> {
 int head, tail;
 Lock lock;
 T[] items;
 public LockBasedQueue(int capacity) {
    head = 0; tail = 0;
    lock = new ReentrantLock();
    items = (T[]) new Object[capacity];
}
```

```
public T deq() throws EmptyEx {
 lock.lock();
 try {
  if(tail==head)throw new EmptyEx();
   T x = items[head % items.length];
   head++;
   return x;
  } finally {
   lock.unlock();
  }
}

public T enq()throws
FullEx {…}
```

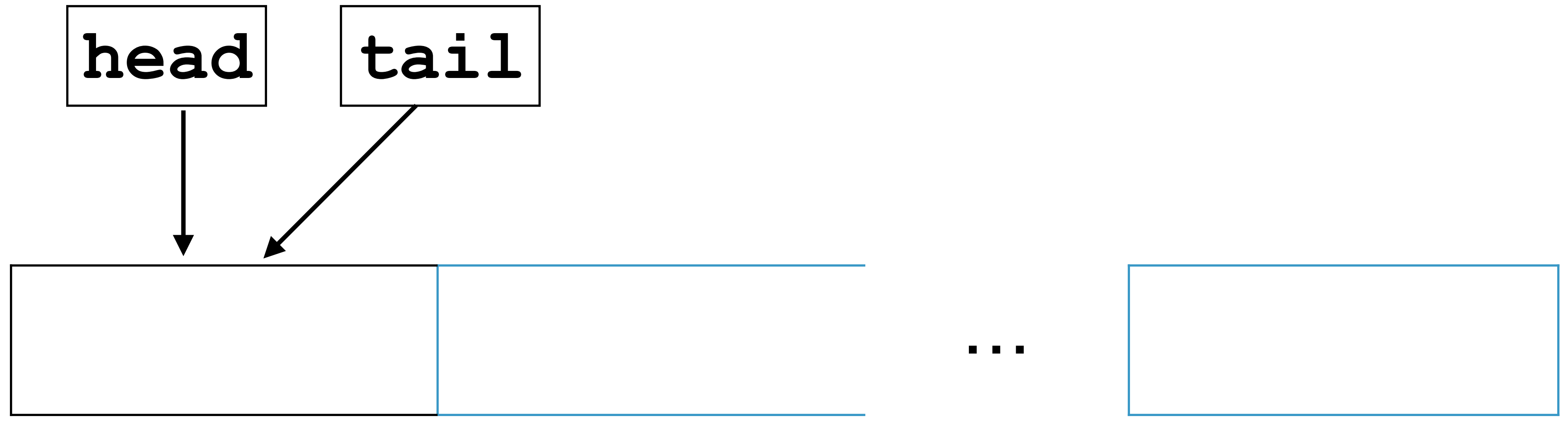
From: *The Art of Multiprocessor Programming* [Herlihy & Shavit 2012]

Implementation of a Queue

```
class LockBasedQueue<T> {
 int head, tail;
 Lock lock;
 T[] items;
 public LockBasedQueue(int capacity) {
    head = 0; tail = 0;
    lock = new ReentrantLock();
    items = (T[]) new Object[capacity];
}
```

```
public T deq() throws EmptyEx {
 lock.lock();
 try {
  if(tail==head)throw new EmptyEx();
   T x = items[head % items.length];
   head++;
   return x;
  } finally {
   lock.unlock();
  }
}

public T enq()throws
FullEx {…}
```



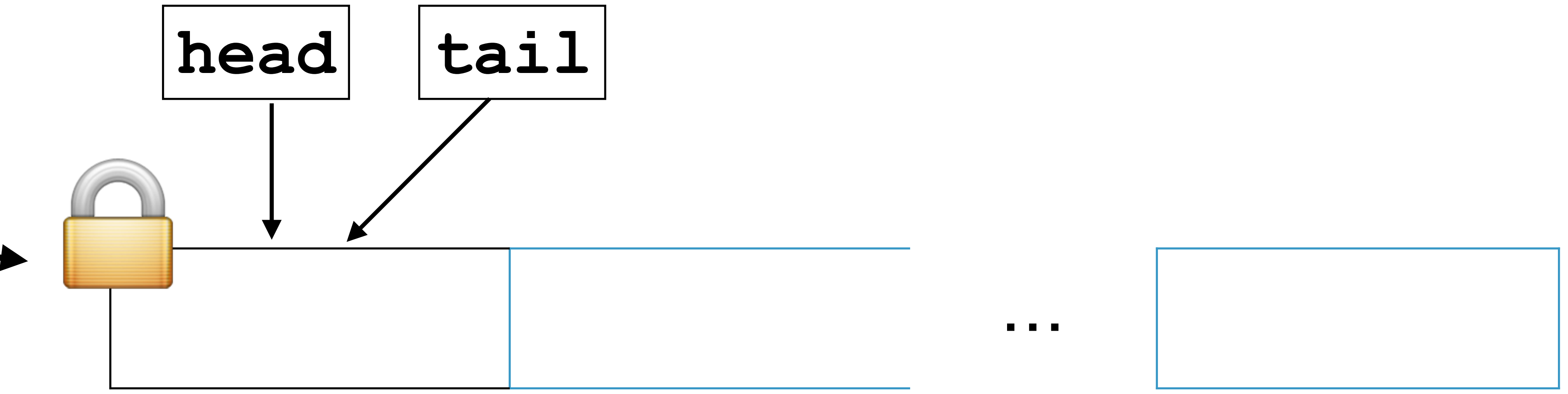
From: *The Art of Multiprocessor Programming* [Herlihy & Shavit 2012]

Implementation of a Queue

```
class LockBasedQueue<T> {
 int head, tail;
 Lock lock;
 T[] items;
 public LockBasedQueue(int capacity) {
    head = 0; tail = 0;
    lock = new ReentrantLock();
    items = (T[]) new Object[capacity];
}
```

```
public T deq() throws EmptyEx {
 lock.lock();
  try {
   if(tail==head)throw new EmptyEx();
    T x = items[head % items.length];
    head++;
    return x;
   } finally {
    lock.unlock();
   }
}

public T enq()throws
FullEx {…}
```



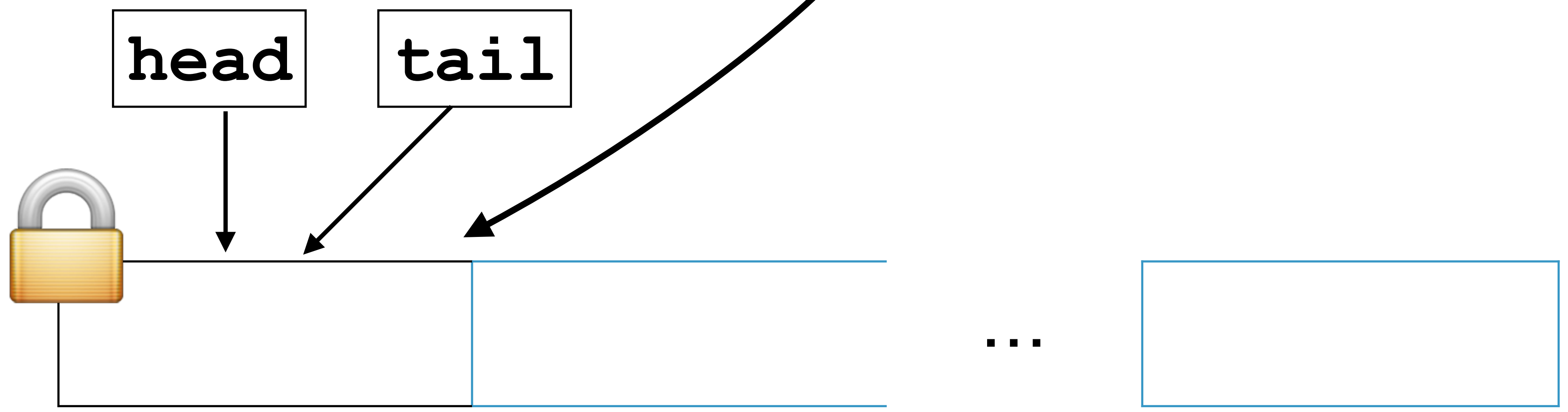
From: *The Art of Multiprocessor Programming* [Herlihy & Shavit 2012]

Implementation of a Queue

```
class LockBasedQueue<T> {
 int head, tail;
 Lock lock;
 T[] items;
 public LockBasedQueue(int capacity) {
    head = 0; tail = 0;
    lock = new ReentrantLock();
    items = (T[]) new Object[capacity];
}
```

```
public T deq() throws EmptyEx {
 lock.lock();
 try {
  if(tail==head)throw new EmptyEx();
   T x = items[head % items.length];
   head++;
   return x;
  } finally {
   lock.unlock();
  }
}

public T enq()throws
FullEx {…}
```



From: *The Art of Multiprocessor Programming* [Herlihy & Shavit 2012]

What is a client?

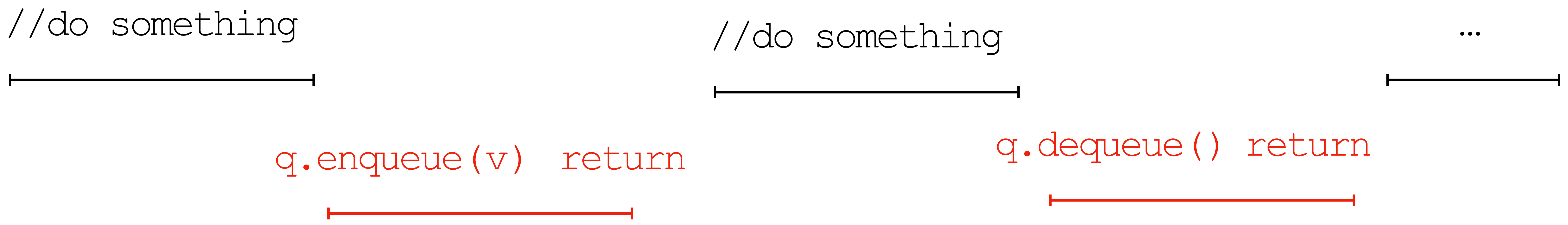
- What is a client of the Library?
  - Program that issues calls to a library instance

```
// do something
q.enque(v)
// do something
x = q.dequeue()
// …
```

What is a client?

- What is a client of the Library?
  - Program that issues calls to a library instance

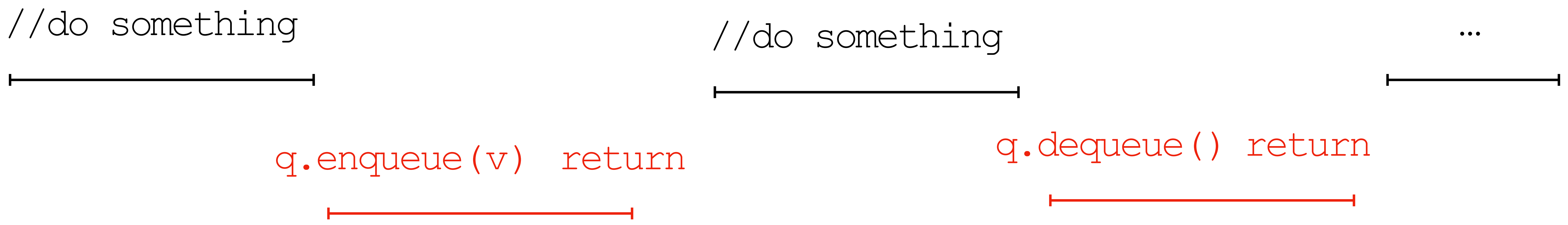
```
// do something
q.enque(v)
// do something
x = q.dequeue()
// …
```



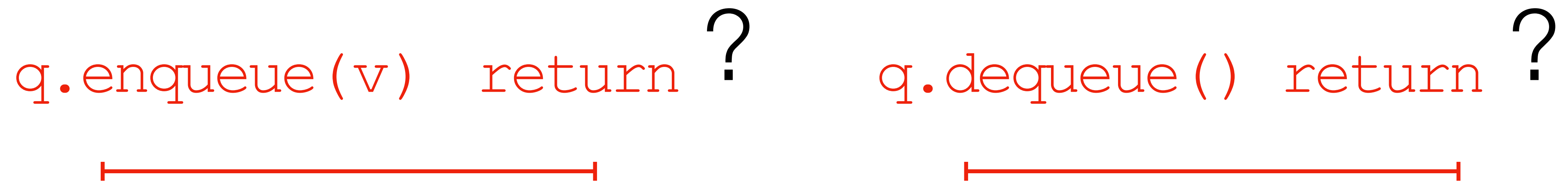
What is a client?

- What is a client of the Library?
  - Program that issues calls to a library instance

```
// do something
q.enque(v)
// do something
x = q.dequeue()
// …
```



- How do we specify a Data Structure (DS) generically?
  - Histories of calls and returns
  - Constraint possible return values

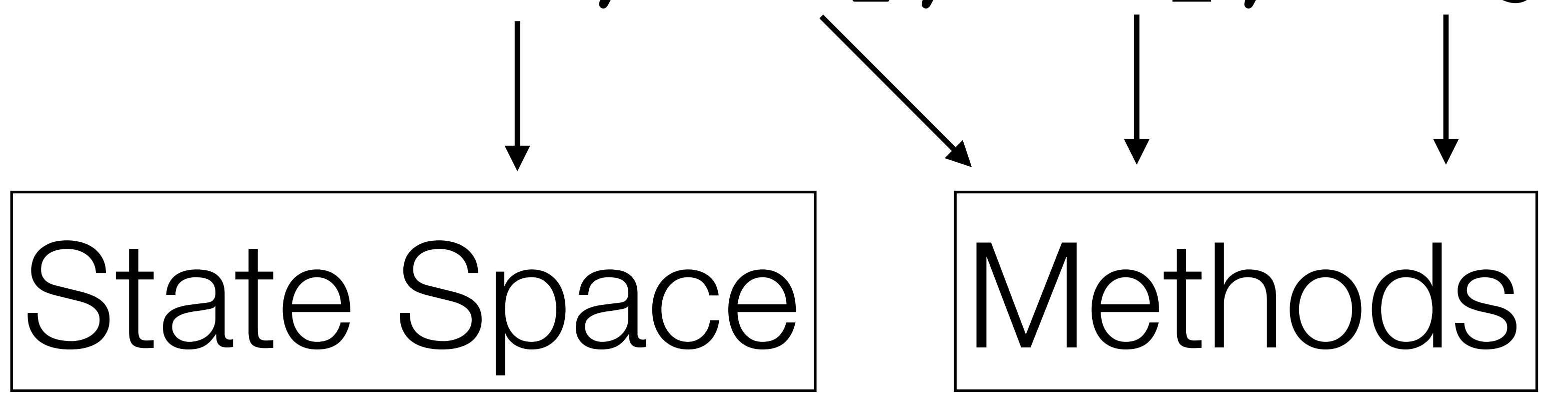


Well Encapsulated Objects

- *Global* object state:
  - Possibly *local* thread state
- A set of *operations* or *methods*
  - Input and output types
  - Methods are the only way to operate on the state

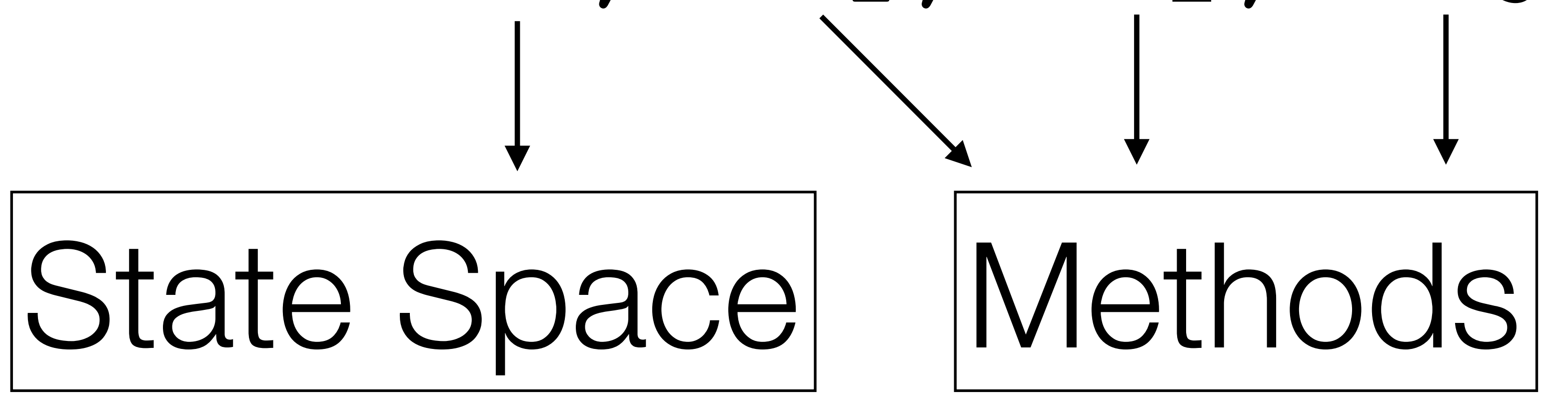
Sequential Object Specifications

- Library $\mathtt{L = <\Sigma,\ m_1,\ m_2,\ m_3>}$



Sequential Object Specifications

- Library `L = <Σ, m1, m2, m3>` ($L = \langle \Sigma, m_1, m_2, m_3 \rangle$)



- Client C: Issues calls to the library methods
  - (Sequential) Most General Client [SMGC]

Sequential Object Specifications

- Library `L = <Σ, m1, m2, m3>`

  Σ → State Space; m_1, m_2, m_3 → Methods

- Client C: Issues calls to the library methods
  - (Sequential) Most General Client [SMGC]

```
SMGC(L):
  while true do
    m_i = choseMethodFrom(L);
    args = choseInputsFor(m);
    m_i(args);
  od
```

Sequential Object Specifications

- Library $L = \langle \Sigma, m_1, m_2, m_3 \rangle$

  Σ → State Space; m_1, m_2, m_3 → Methods

- Client C: Issues calls to the library methods
  - (Sequential) Most General Client [SMGC]

```
SMGC(L):
  while true do
    mi = choseMethodFrom(L);
    args = choseInputsFor(m);
    mi(args);
  od
```

- We will talk about histories of calls with values
  - ϵ denotes the empty sequence,
  - o denotes an operation (eg. <pop(),v>), and
  - δ denotes a sequence of operations

Specifying a Register

- Inductive histories of a Stack:

Specifying a Register

- Inductive histories of a Stack:

1. ϵ is a Register History (RH)

Specifying a Register

- Inductive histories of a Stack:

1. ϵ is a Register History (RH)
2. <read(),0>* is a Register History

Specifying a Register

- Inductive histories of a Stack:

1. ϵ is a Register History (RH)
2. `<read(),0>*` is a Register History
3. if δ is a RH, then so is $\delta \cdot$`<write(v),_>`

Specifying a Register

- Inductive histories of a Stack:

1. ϵ is a Register History (RH)
2. <read(),0>* is a Register History
3. if δ is a RH, then so is δ·<write(v),_>
4. if δ·<write(v),_> is a RH, then so it is δ·<read(),v>*

Specifying a Register

- Inductive histories of a Stack:

1. ϵ is a Register History (RH)
2. `<read(),0>*` is a Register History
3. if δ is a RH, then so is $\delta \cdot$`<write(v),_>`
4. if $\delta \cdot$`<write(v),_>` is a RH, then so it is $\delta \cdot$`<read(),v>*`

Some examples on the board

Specifying a Stack

- Inductive histories of a Stack:

Specifying a Stack

- Inductive histories of a Stack:

1. ϵ is a Stack History (SH)

Specifying a Stack

- Inductive histories of a Stack:
  1. `ϵ` is a Stack History (SH)
  2. if `δ·<pop(),v>` is a SH, then so is `<push(w),_>·δ`

Specifying a Stack

- Inductive histories of a Stack:
  1. ϵ is a Stack History (SH)
  2. if δ ·<pop(),v> is a SH, then so is <push(w),_>·δ
  3. if δ is a SH, and $|\{\text{<pop(),v>}:\delta | v \neq \bot\}| = |\text{push(v),\_>}:\delta\}|$, then so it is δ ·<pop(),⊥>*

Specifying a Stack

- Inductive histories of a Stack:

1. ϵ is a Stack History (SH)
2. if δ ·<pop(),v> is a SH, then so is <push(w),_>·δ
3. if δ is a SH, and |{<pop(),v>:δ|v ≠ ⊥}| = |push(v),_>:δ}|, then so it is δ ·<pop(),⊥>*
4. same conditions as 4, and <pop(),⊥> does not occur in δ then, <push(w),⊥>·δ·<pop(),w> is a SH

Specifying a Stack

- Inductive histories of a Stack:

1. ϵ is a Stack History (SH)
2. if `δ·<pop(),v>` is a SH, then so is `<push(w),_>·δ`
3. if δ is a SH, and `|{<pop(),v>:δ|v ≠ ⊥}| = |push(v),_>:δ}|`, then so it is `δ·<pop(),⊥>*`
4. same conditions as 4, and `<pop(),⊥>` does not occur in δ then, `<push(w),⊥>·δ·<pop(),w>` is a SH
5. if $\delta_0 \cdot$`<pop(),⊥>` is a SH, and δ_1 is a SH, then $\delta_0 \cdot \delta_1$ is SH

Specifying a Stack

- Inductive histories of a Stack:

1. ϵ is a Stack History (SH)
2. if $\delta \cdot <\mathtt{pop}(), v>$ is a SH, then so is $<\mathtt{push}(w), \_> \cdot \delta$
3. if δ is a SH, and $|\{<\mathtt{pop}(), v> : \delta | v \neq \bot\}| = |\mathtt{push}(v), \_> : \delta\}|$,
   then so it is $\delta \cdot <\mathtt{pop}(), \bot>^*$
4. same conditions as 4, and $<\mathtt{pop}(), \bot>$ does not occur in δ
   then, $<\mathtt{push}(w), \bot> \cdot \delta \cdot <\mathtt{pop}(), w>$ is a SH
5. if $\delta_0 \cdot <\mathtt{pop}(), \bot>$ is a SH, and δ_1 is a SH, then $\delta_0 \cdot \delta_1$ is SH

Some examples on the board

Specifying a Queue

Specifying a Queue

Exercise

What about Concurrency?

```
while true do
 mi = choseMethodFrom(L);
 args = choseInputsFor(m);
 mi(args);
od
```

$\|$

```
while true do
 mi = choseMethodFrom(L);
 args = choseInputsFor(m);
 mi(args);
od
```

What about Concurrency?

```
while true do
  m_i = choseMethodFrom(L);
  args = choseInputsFor(m);
  m_i(args);
od
```

||

```
while true do
  m_i = choseMethodFrom(L);
  args = choseInputsFor(m);
  m_i(args);
od
```

s.push(v) return

s.pop() return v

What about Concurrency?

```
while true do
  m_i = choseMethodFrom(L);
  args = choseInputsFor(m);
  m_i(args);
od
```

$\|$

```
while true do
  m_i = choseMethodFrom(L);
  args = choseInputsFor(m);
  m_i(args);
od
```

s.push(v) return

s.pop() return ⊥

s.pop() return v

s.push(w) return

What about Concurrency?

```
while true do
  mᵢ = choseMethodFrom(L);
  args = choseInputsFor(m);
  mᵢ(args);
od
```

||

```
while true do
  mᵢ = choseMethodFrom(L);
  args = choseInputsFor(m);
  mᵢ(args);
od
```

s.push(v) return s.pop() return v ?

s.pop() return ⊥ ? s.push(w) return

Should this be legal?

What about Concurrency?

```
while true do
  mi = choseMethodFrom(L);
  args = choseInputsFor(m);
  mi(args);
od
```

||

```
while true do
  mi = choseMethodFrom(L);
  args = choseInputsFor(m);
  mi(args);
od
```

s.push(v) return s.pop() return v ?

Concurrent Consistency Criteria

Should this be legal?

Concurrent Clients

- Most General Client (seq)

- Most General Client (concurrent n threads)

```
CMGCn(L):
 SMGL(L) || SMGL(L) … || SMGL(L)
 |_____________________________|
               n
```

- Concurrent Library Verification w.r.t. $\texttt{CMGC}_n\texttt{(L)}$ for any n

Concurrent Clients

- Most General Client (seq)

```
SMGC(L):
 while true do
  mi = choseMethodFrom(L);
  args = choseInputsFor(m);
  mi(args);
 od
```

- Most General Client (concurrent n threads)

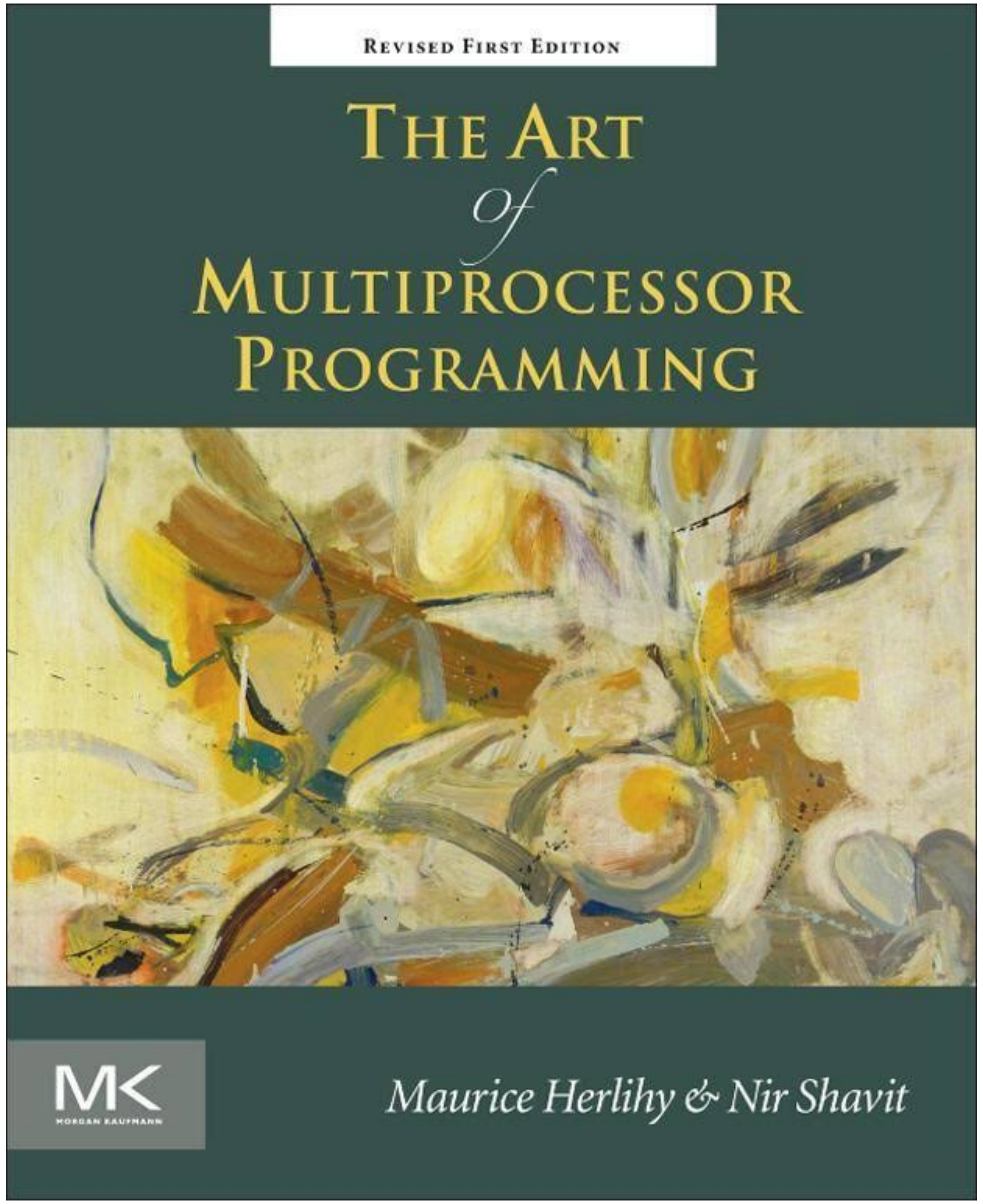
```
CMGCn(L):
 SMGL(L) || SMGL(L) … || SMGL(L)
 |___________________________________|
                  n
```

- Concurrent Library Verification w.r.t. $\texttt{CMGC}_n\texttt{(L)}$ for any n

Concurrent Consistency Criteria

- Quiescence Consistency
- Sequential Consistency
  - Serializability
    - Conflict Serializability
    - Strict Serializability
- Linearizability

We will work with Registers to exemplify the definitions



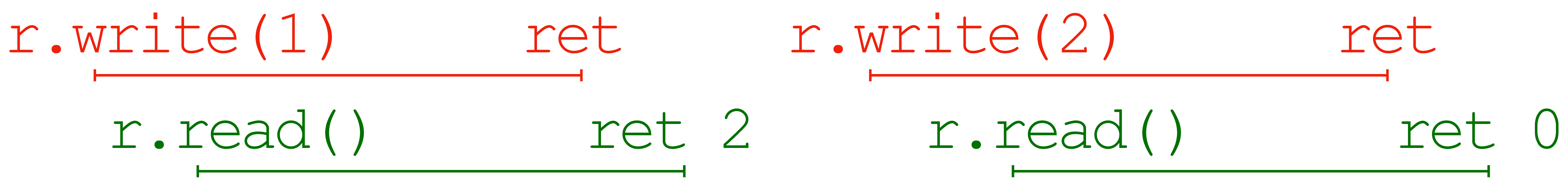
Quiescent Consistency

- Method calls should appear to happen one-at-a-time, sequential order

Quiescent Consistency

- Method calls should appear to happen one-at-a-time, sequential order

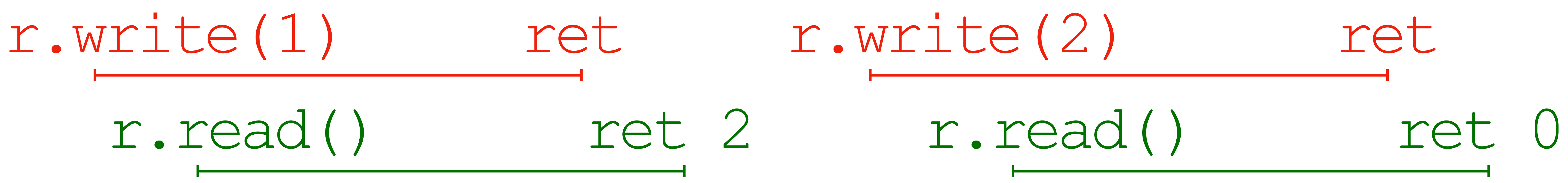
```
r.write(1); || r.read();
r.write(2); || r.read();
```

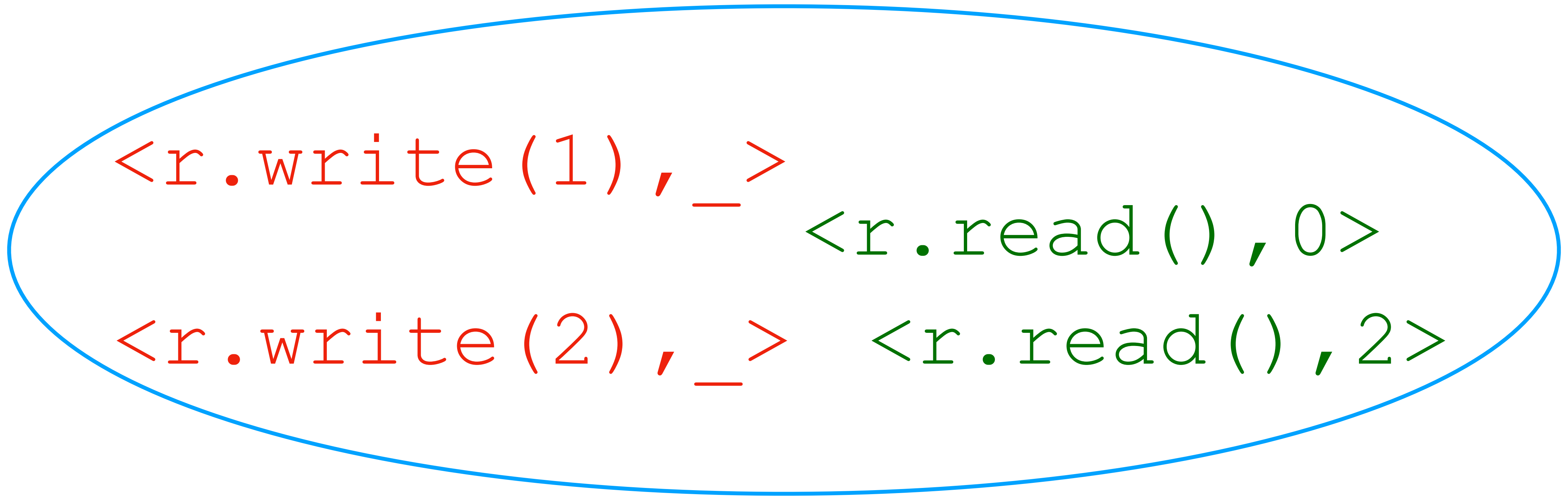


Quiescent Consistency

- Method calls should appear to happen one-at-a-time, sequential order

```
r.write(1); || r.read();
r.write(2); || r.read();
```

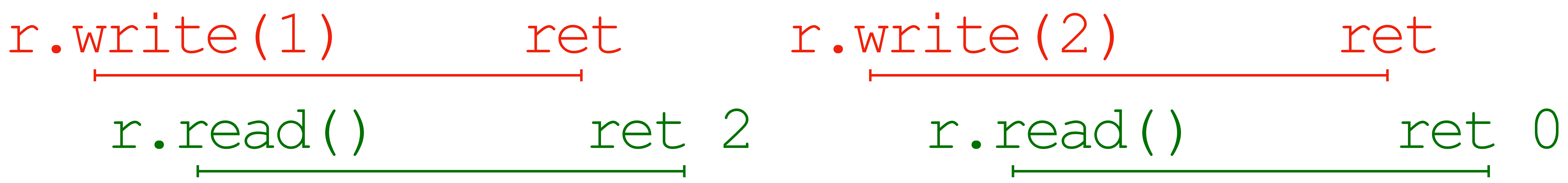




Quiescent Consistency

- Method calls should appear to happen one-at-a-time, sequential order

```
r.write(1); || r.read();
r.write(2); || r.read();
```



<r.write(1),_>
<r.read(),0>
<r.write(2),_> <r.read(),2>

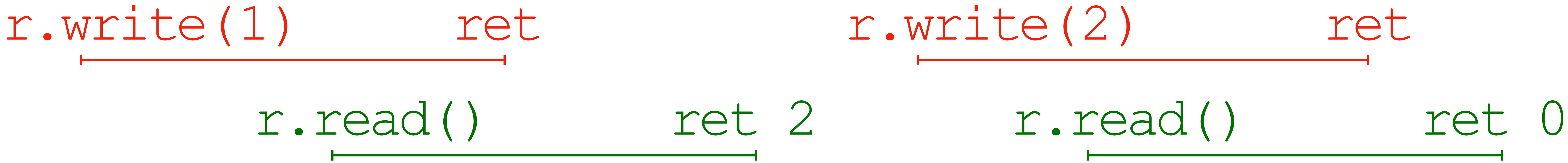
<r.read(),0> <r.write(1),_> <r.write(2),_> <r.read(),2>

Quiescent Consistency

- Method calls should appear to happen one-at-a-time, sequential order
- Method calls separated by a period of quiescence should appear to take effect in their real time order

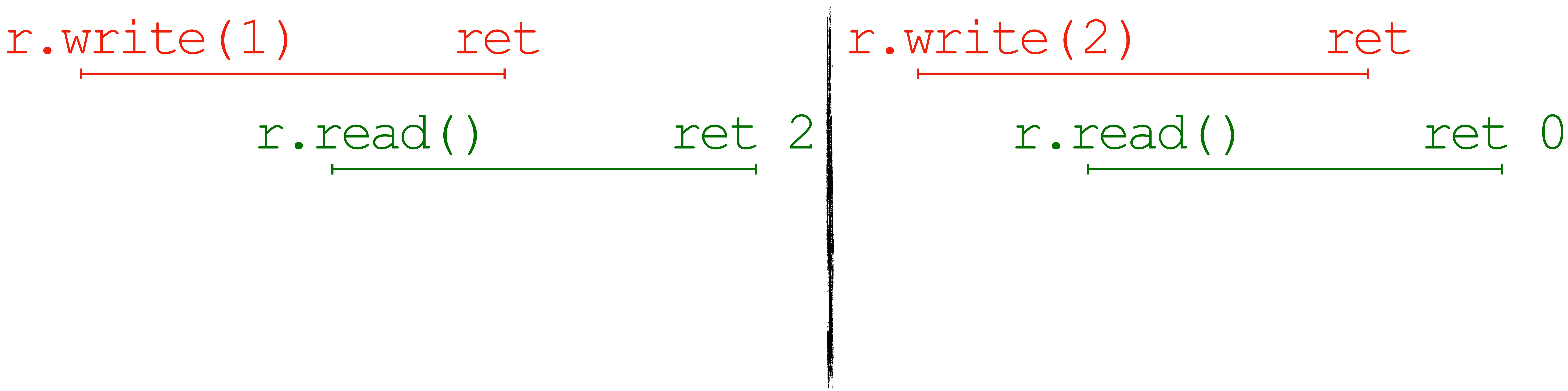
Quiescent Consistency

- Method calls should appear to happen one-at-a-time, sequential order
- Method calls separated by a period of quiescence should appear to take effect in their real time order



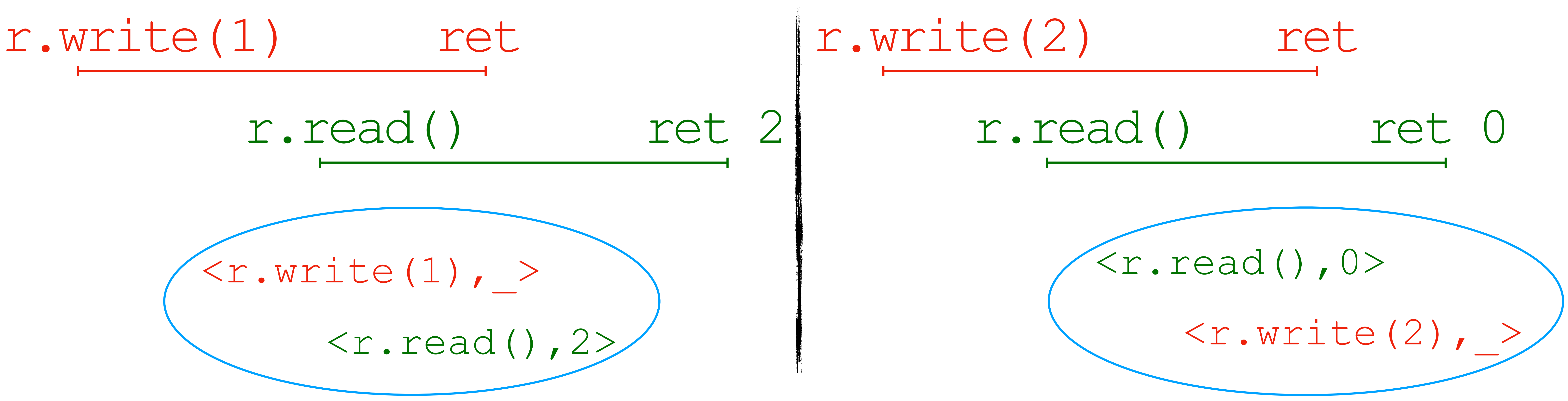
Quiescent Consistency

- Method calls should appear to happen one-at-a-time, sequential order
- Method calls separated by a period of quiescence should appear to take effect in their real time order



Quiescent Consistency

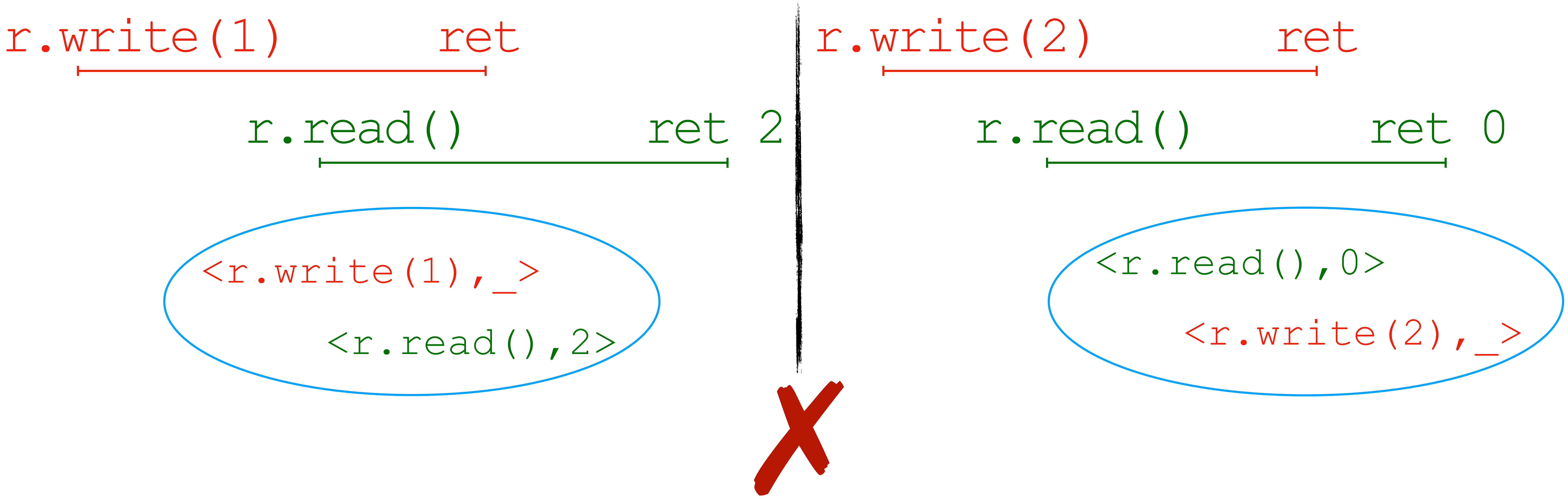
- Method calls should appear to happen one-at-a-time, sequential order
- Method calls separated by a period of quiescence should appear to take effect in their real time order



Quiescent Consistency

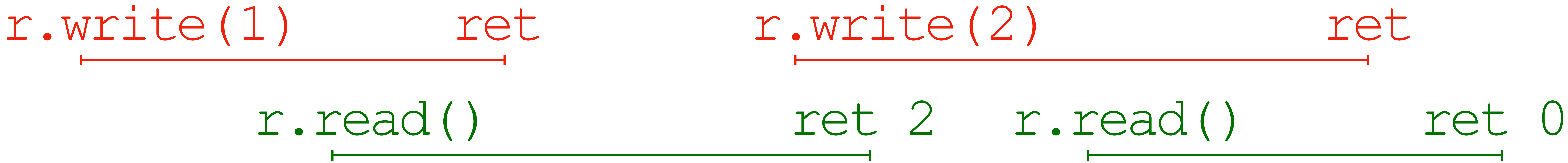
- Method calls should appear to happen one-at-a-time, sequential order

- Method calls separated by a period of quiescence should appear to take effect in their real time order



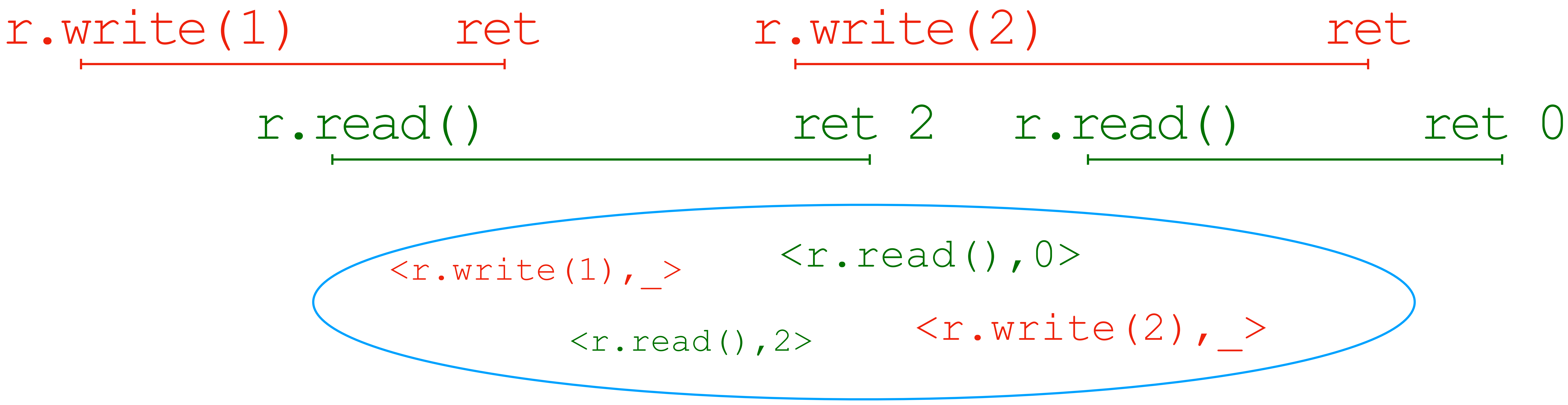
Quiescent Consistency

- Method calls should appear to happen one-at-a-time, sequential order
- Method calls separated by a period of quiescence should appear to take effect in their real time order



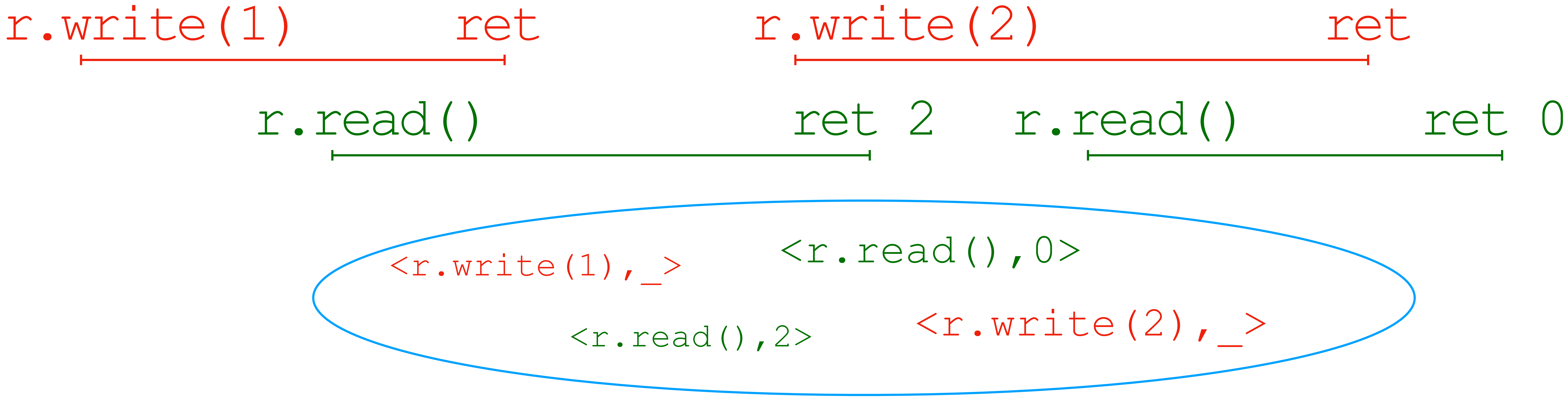
Quiescent Consistency

- Method calls should appear to happen one-at-a-time, sequential order
- Method calls separated by a period of quiescence should appear to take effect in their real time order



Quiescent Consistency

- Method calls should appear to happen one-at-a-time, sequential order
- Method calls separated by a period of quiescence should appear to take effect in their real time order



`<r.read(),0> <r.write(1),_> <r.write(2),_> <r.read(),2>`

Sequential Consistency

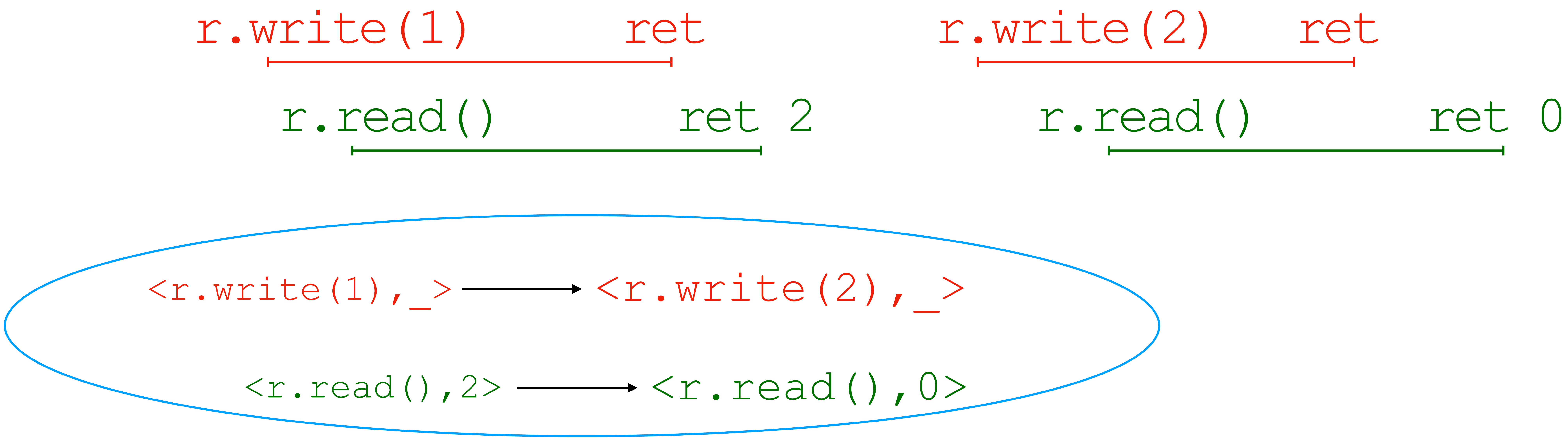
- *How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Computer Programs* [Lamport'79]
  - Each process issues operations in the order specified by its program.
  - Operations from all processors issued to a single object are serviced from a single FIFO queue. Issuing an operation consists in entering a request on this queue.

Sequential Consistency

```
r.write(1); ║ r.read();
r.write(2); ║ r.read();
```

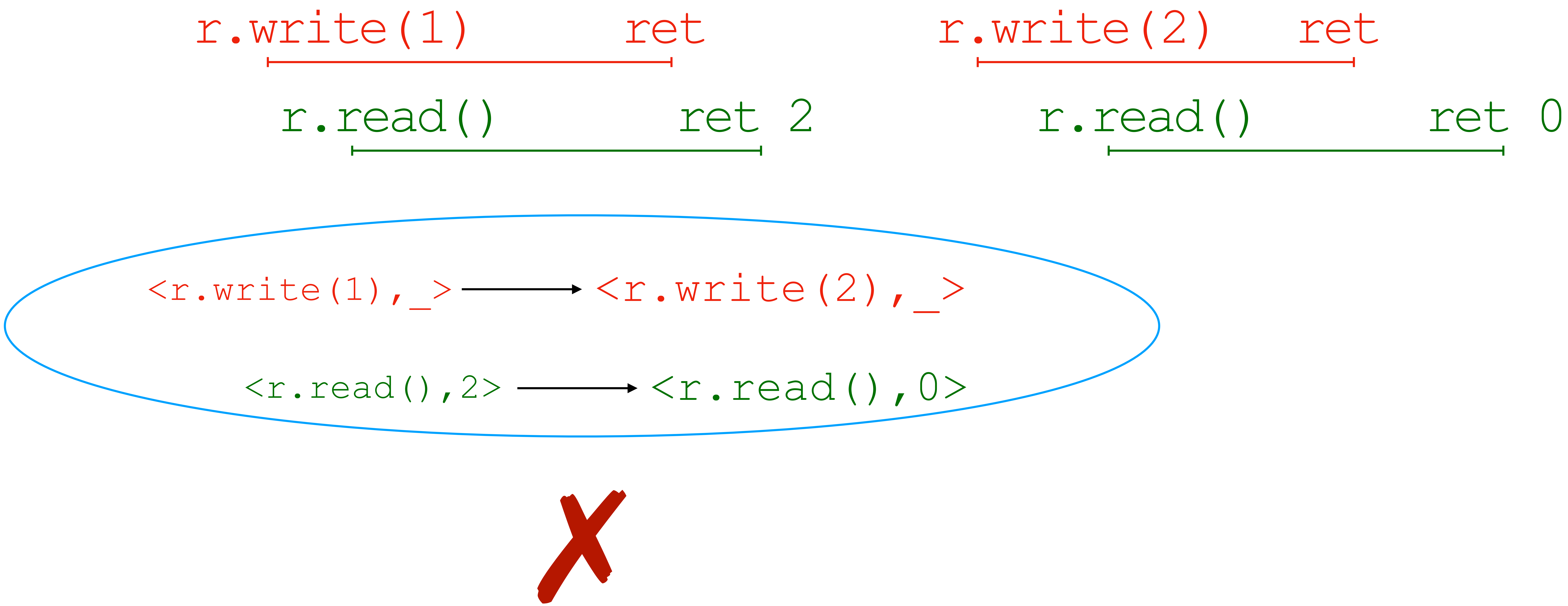
Sequential Consistency

```
r.write(1); ‖ r.read();
r.write(2); ‖ r.read();
```



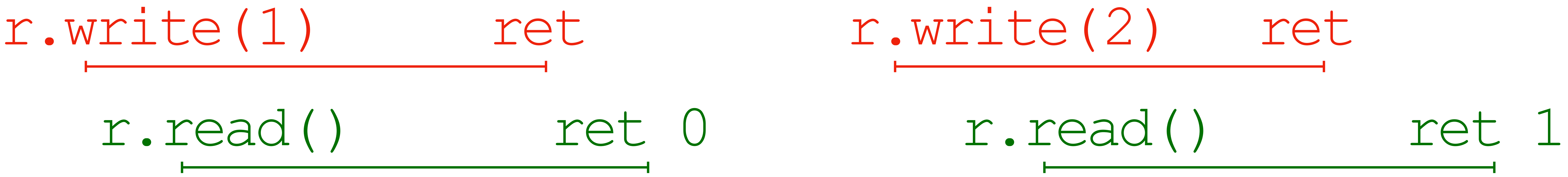
Sequential Consistency

```
r.write(1); ‖ r.read();
r.write(2); ‖ r.read();
```



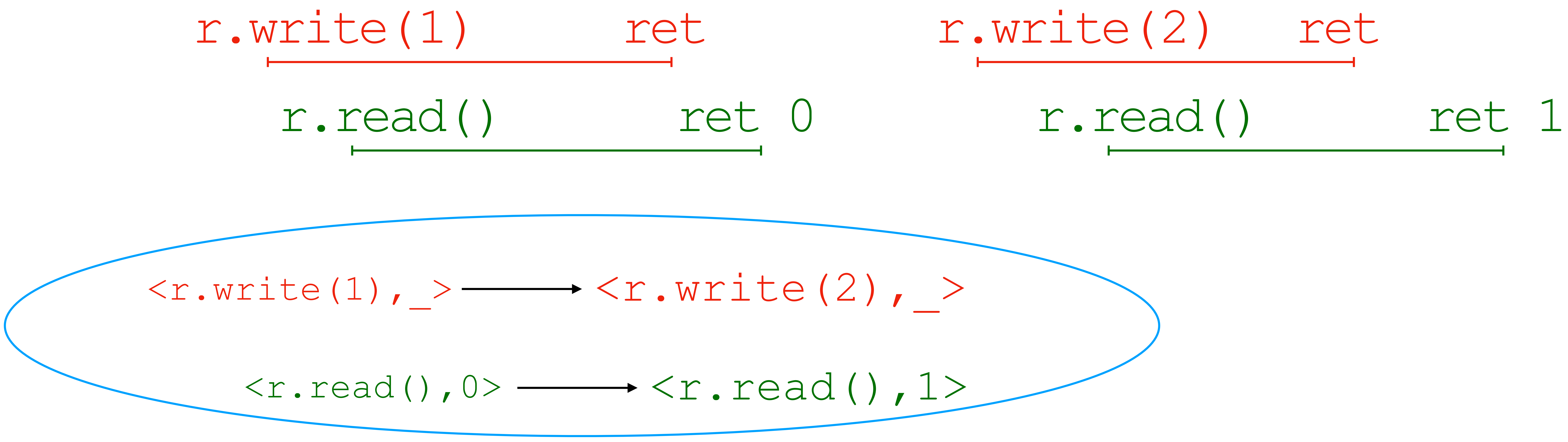
Sequential Consistency

```
r.write(1); ‖ r.read();
r.write(2); ‖ r.read();
```



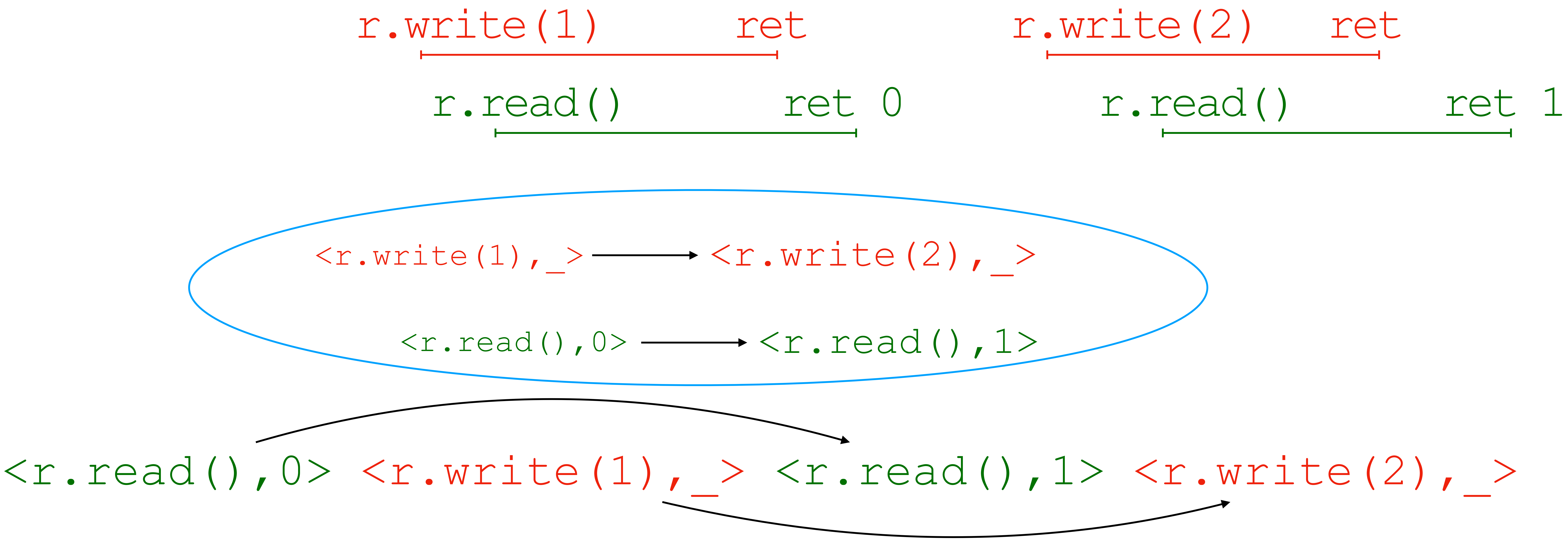
Sequential Consistency

```
r.write(1); ‖ r.read();
r.write(2); ‖ r.read();
```



Sequential Consistency

```
r.write(1); || r.read();
r.write(2); || r.read();
```



Sequential Consistency

- Quiescent Consistency +
- Method calls should appear to take effect in Program Order

program order ↓

```
r.write(1); || r.read();
r.write(2); || r.read();
```

↓ program order

Sequential Consistency

- Quiescent Consistency +
- Method calls should appear to take effect in Program Order

program order ↓

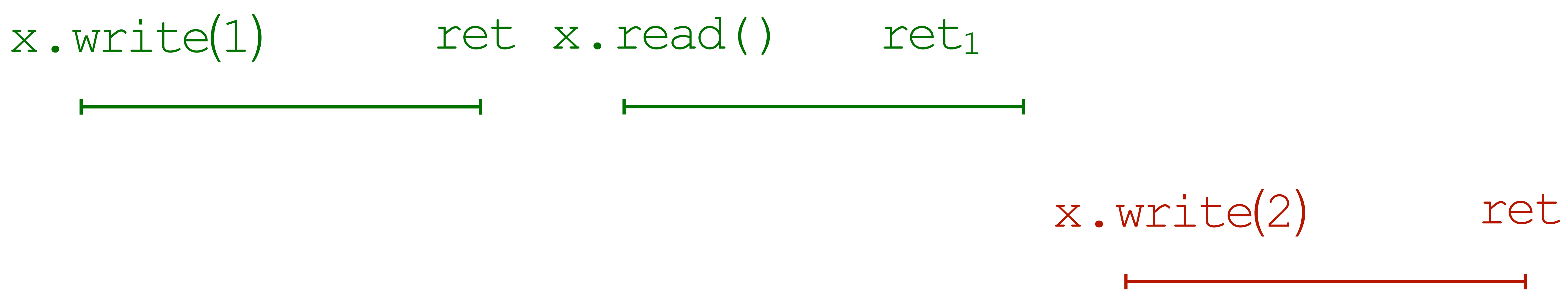
```
r.write(1); || r.read();
r.write(2); || r.read();
```

↓ program order

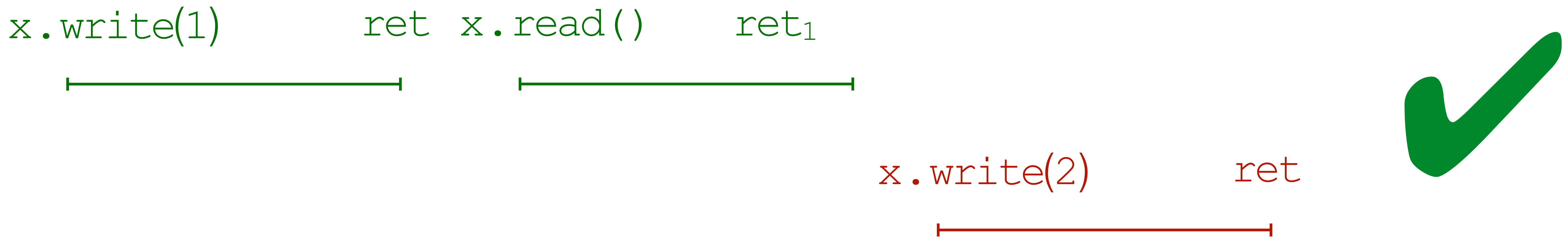
- Each history δ induces a per-thread total order of operations
  - $o_1 <_\delta o_2$ iff o_1 and o_2 are on the same thread, and o_1 occurs before o_2 in δ
- A history δ is Sequentially Consistent if there exists an equivalent *Sequential* history δ' (i.e. same operations), and
  - $o1 <_\delta o2$ implies $o1 <_{\delta'} o2$

Sequential Consistency

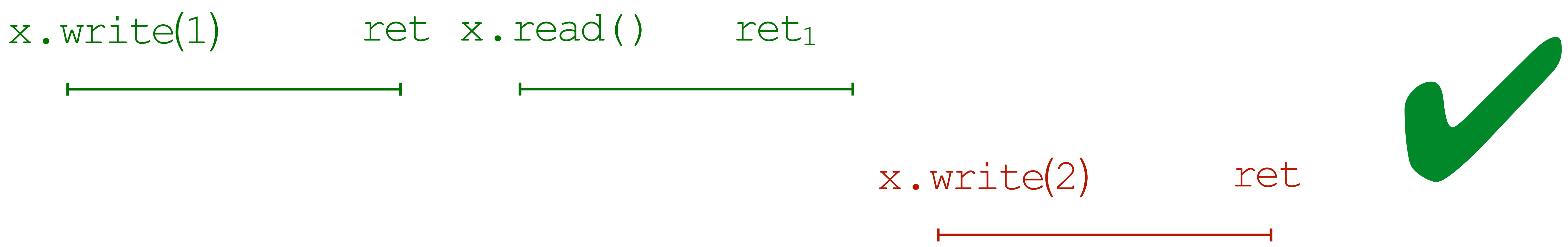
Sequential Consistency



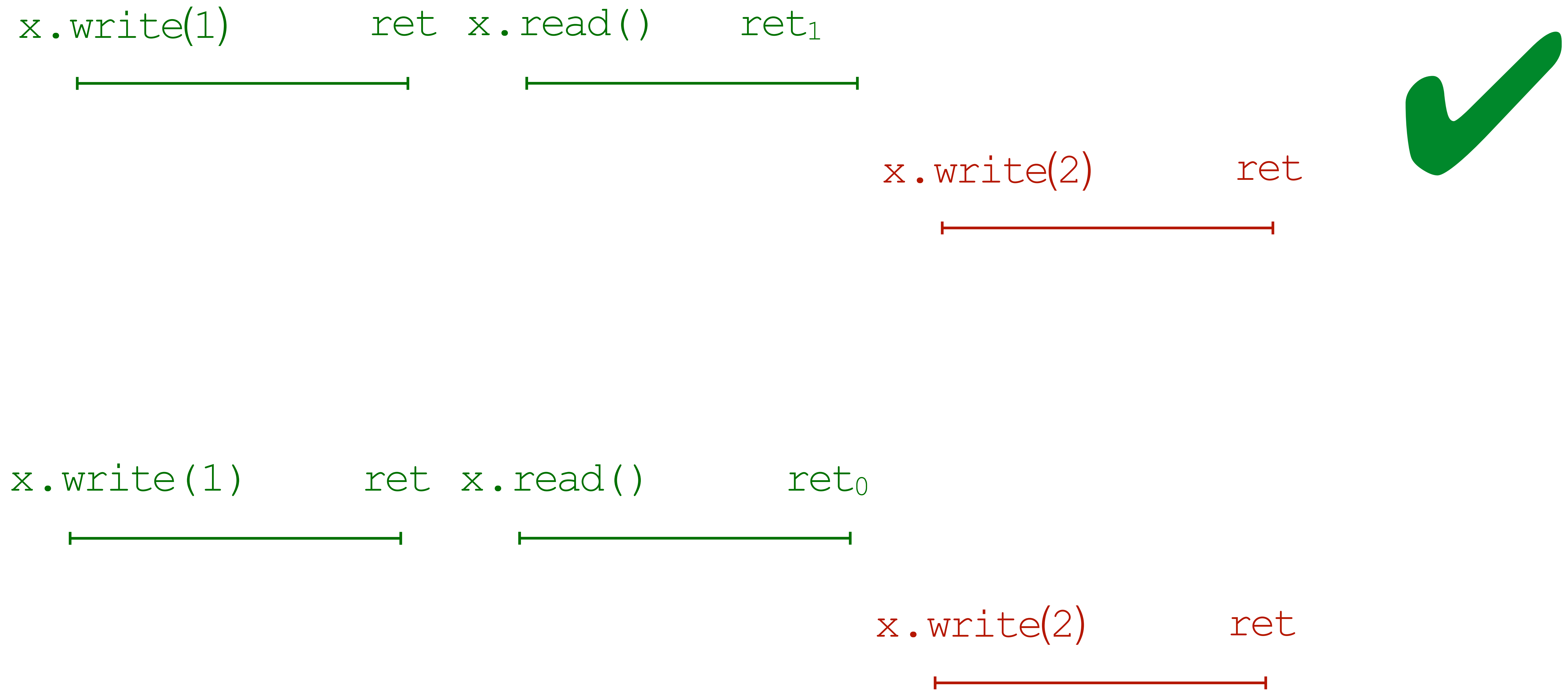
Sequential Consistency



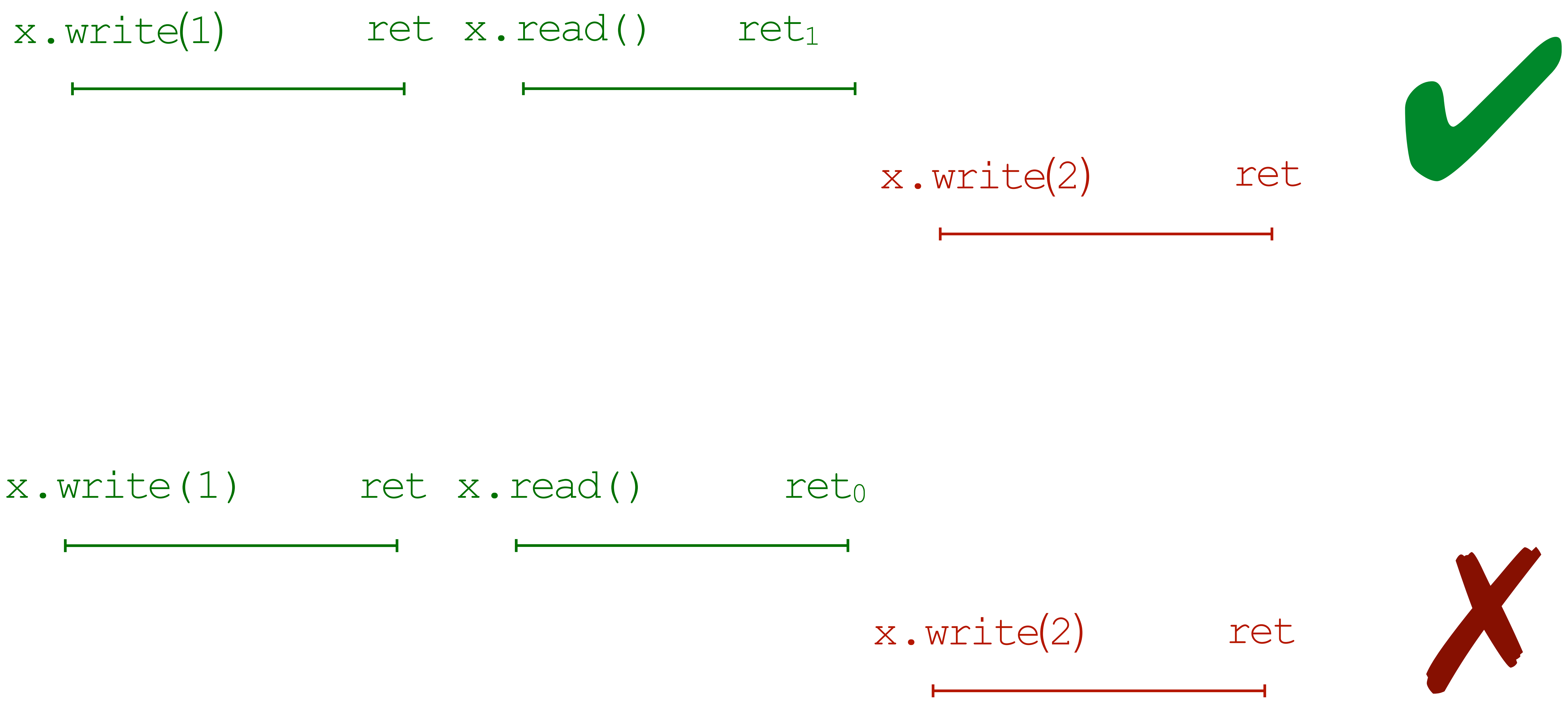
Sequential Consistency



Sequential Consistency



Sequential Consistency



Serializability (DB transactions)

- The read and write steps of transactions can be reordered until the read and writes of each transaction are together without affecting the values read by transactions. (c.f. `[Eswaran et al.'76]`)
- A set of transactions is serializable if the set produces the same result as some arbitrary *serial* execution of those same transactions for arbitrary input (c.f. `[Papadimitriou'79]`)
- Equivalent to Sequential Consistency for a library of transactions

Conflict Serializability (DB transactions)

- We need to inspect the implementation of the library
  - In a transaction these are writes and reads to different registers
- Specification Histories:
  - Call : `beginTx`
  - Return : `commitTx`
- Implementation Histories:
  - Call : `beginTx`
  - Return : `commitTx`
  - Write : $\texttt{wr}_{\texttt{p,v}}$
  - Read : $\texttt{rd}_{\texttt{p,v}}$
  - RMW : $\texttt{cas}_{\texttt{p,v,w}}$
- Sometimes we need to mention the thread: (t, $wr_{p,v}$)

Conflict Serializability (DB transactions)

- We define a conflict relation `#` between operations:
  - $\mathtt{wr}_{\mathtt{p,v}} \,\#\, \mathtt{rd}_{\mathtt{p,w}}$
  - $\mathtt{wr}_{\mathtt{p,v}} \,\#\, \mathtt{wr}_{\mathtt{p,w}}$
  - $\mathtt{rd}_{\mathtt{p,w}} \,\#\, \mathtt{wr}_{\mathtt{p,v}}$
- Conflict Equivalence:
  - Minimal equivalence on histories $\backsim$, such that if not $\mathtt{o}_1 \,\#\, \mathtt{o}_2$, then

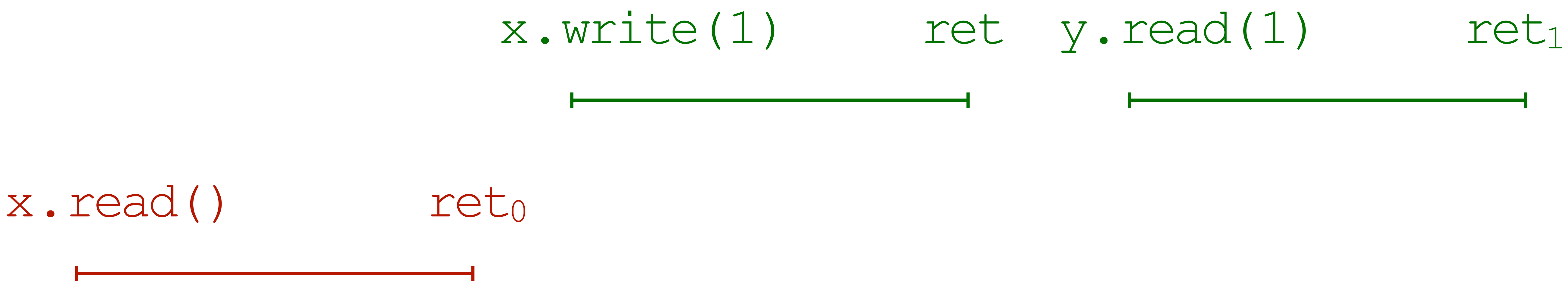
$$\delta_0 \cdot \mathtt{o}_1 \cdot \mathtt{o}_2 \cdot \delta_1 \backsim \delta_0 \cdot \mathtt{o}_2 \cdot \mathtt{o}_1 \cdot \delta_1$$

  - In a nutshell, reordering non-conflicting events renders equivalent histories

A Theory of Database Concurrency Control
[Papadimitriou 1986]

Conflict Serializability

Conflict Serializability



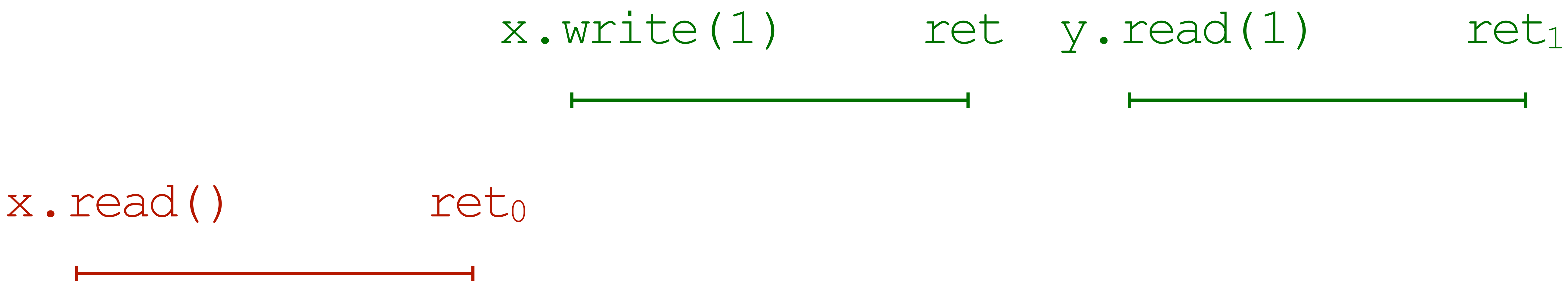
Conflict Serializability

x.write(1) ret y.read(1) ret_1

x.read() ret_0

Serializable ✔

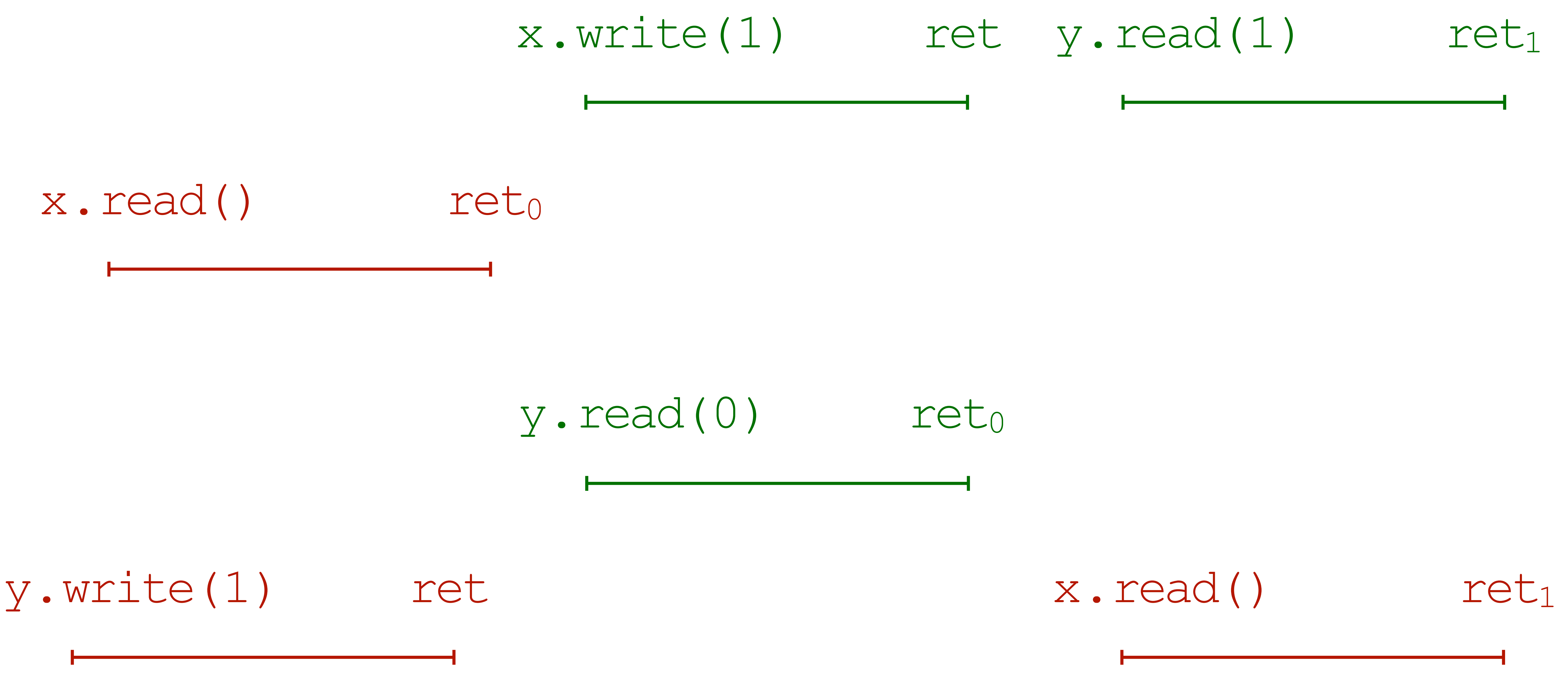
Conflict Serializability



Serializable ✔

Not Conf. Serializable ✗

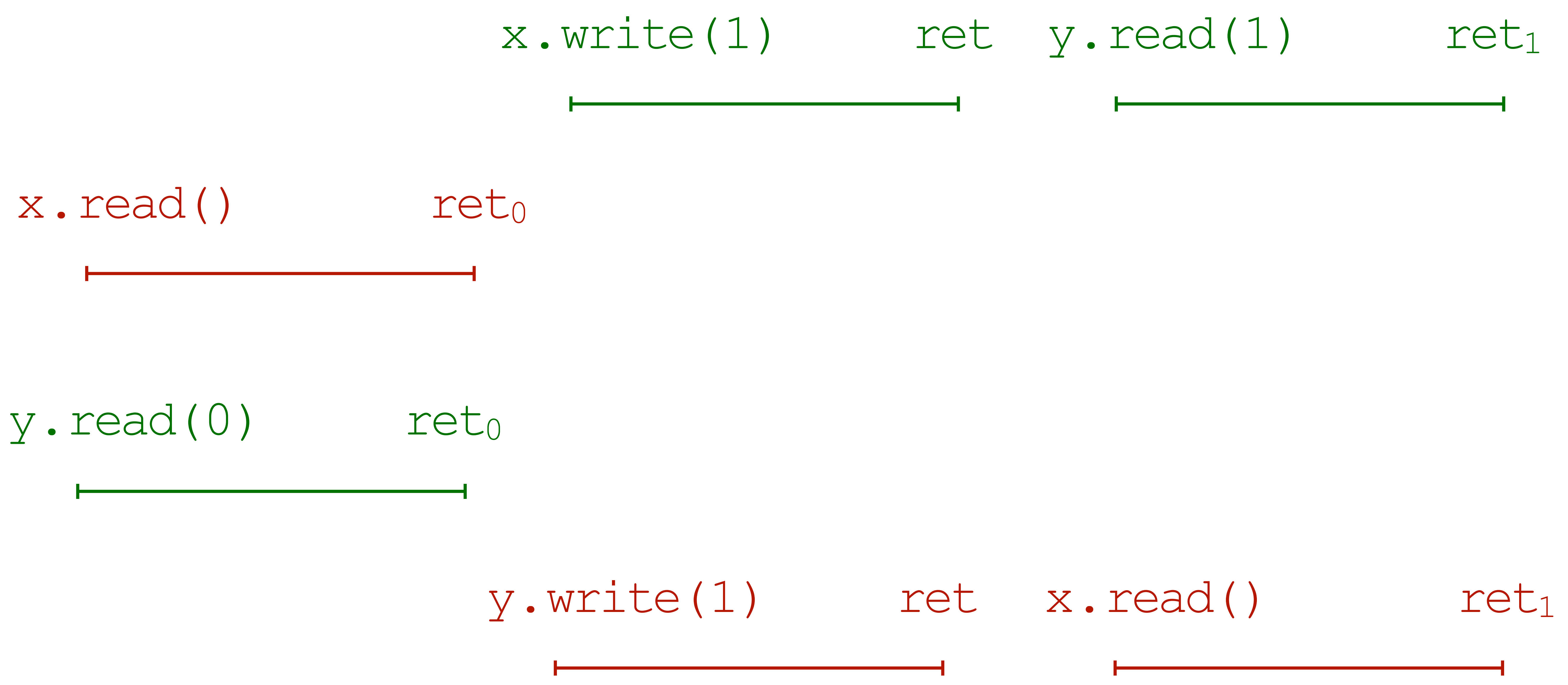
Conflict Serializability



Serializable ✔

Not Conf. Serializable ✗

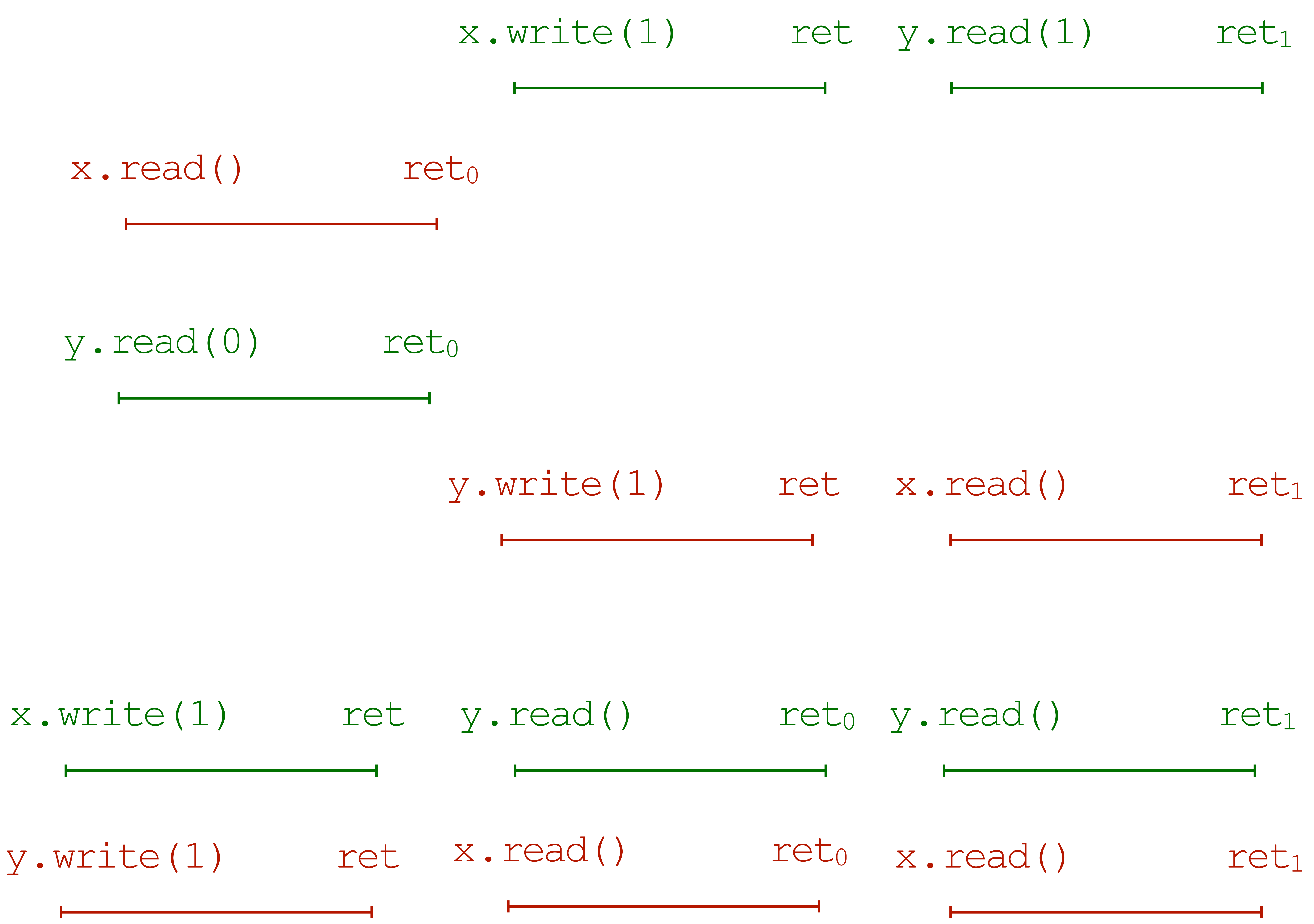
Conflict Serializability



Serializable ✔

Not Conf. Serializable ✗

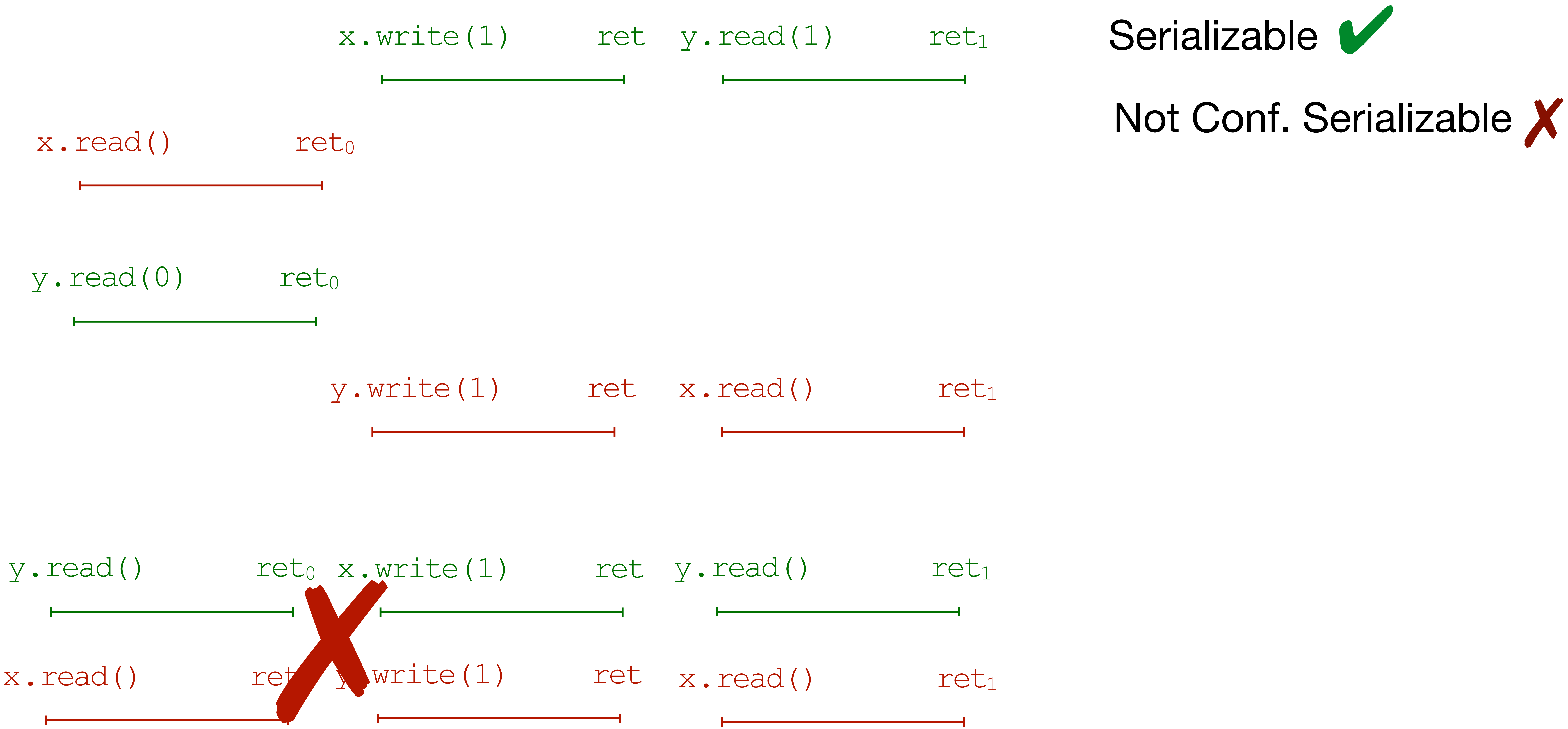
Conflict Serializability



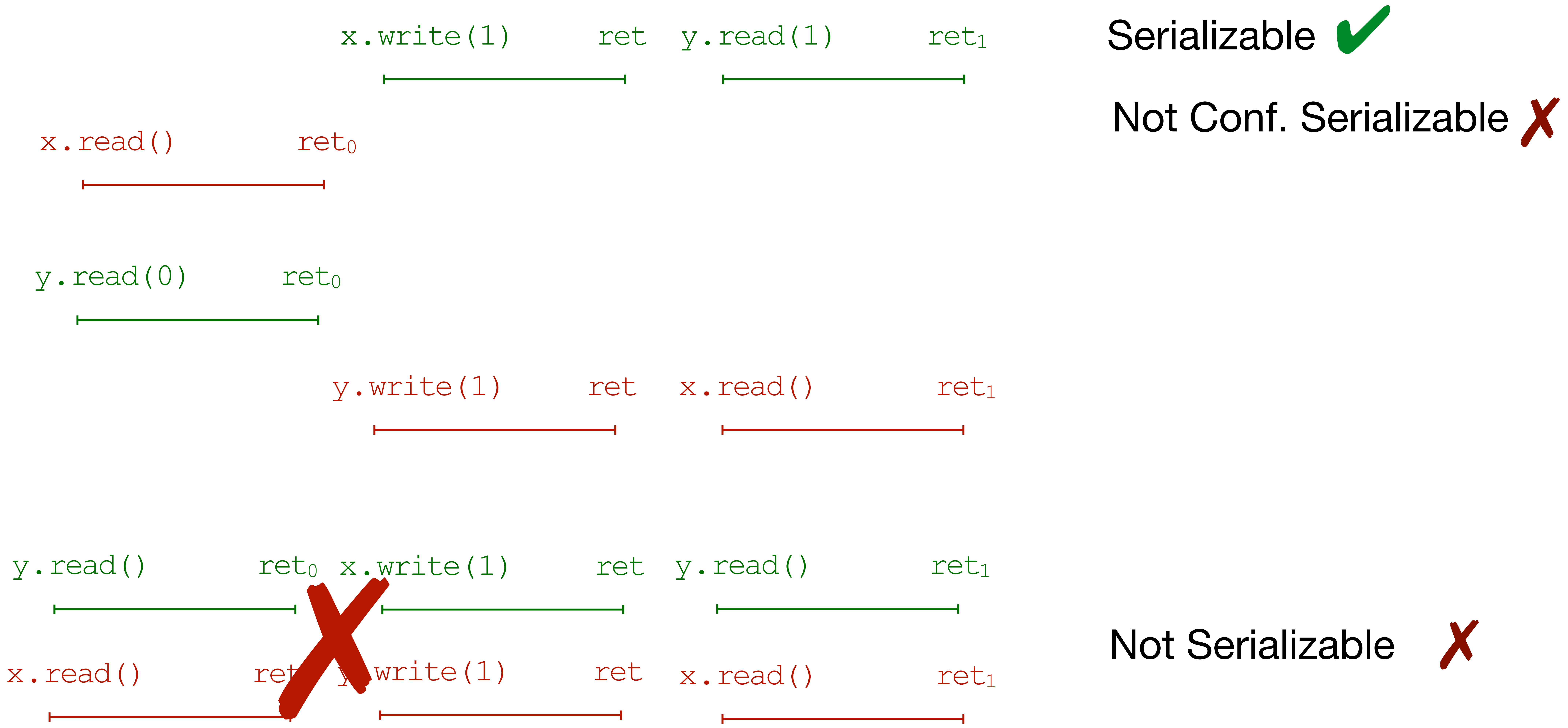
Serializable ✔

Not Conf. Serializable ✗

Conflict Serializability



Conflict Serializability



Strict Serializability (DB transactions)

- Transactions that are already in serial order in a history must remain in the same relative order. More precisely, if transaction T writes before T’ reads, then T must be serialized before T’ `[Sethi’82]`

- We need a Real Time Order (“T writes before T’ reads”)

Linearizability

Linearizability

- Same conditions as Sequential Consistency +
- Each method call should appear to take effect instantaneously at some moment between its invocation (call) and response (return)
- That is: we can pretend that the execution of each method is uninterrupted by other calls to the object
- De-facto standard for Concurrent Object Correctness (eg. `java.util.concurrent`)

Linearizability: A Correctness Condition for Concurrent Objects
[Herlihy and Wing ‘90]

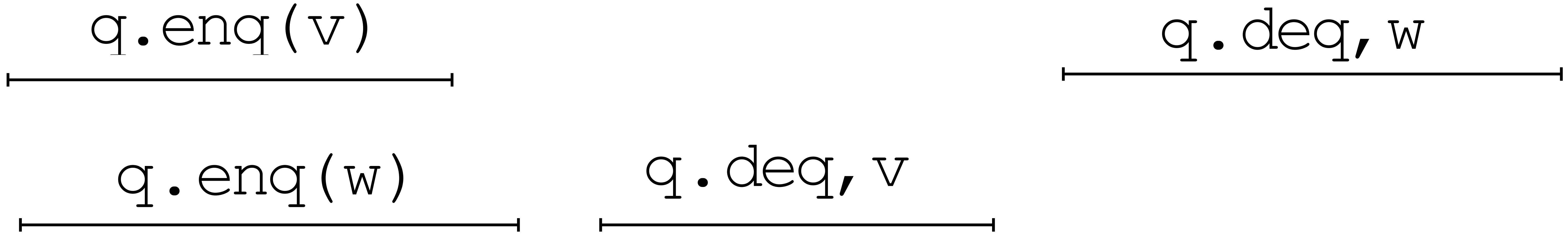
Linearizability

Linearizability

- Each history δ induces a partial order on operations such that
  - $o_1 \sqsubset_\delta o_2$ iff `ret` o_1 occurs before `call` o_2 in δ
- A history δ is Linearizable if there exists an equivalent *Sequential* history δ' (i.e. same operations), and
  - $o_1 \sqsubset_\delta o_2$ implies $o_1 \sqsubset_{\delta'} o_2$
- Ignoring uncompleted operations
- Strictly stronger than Sequential Consistency

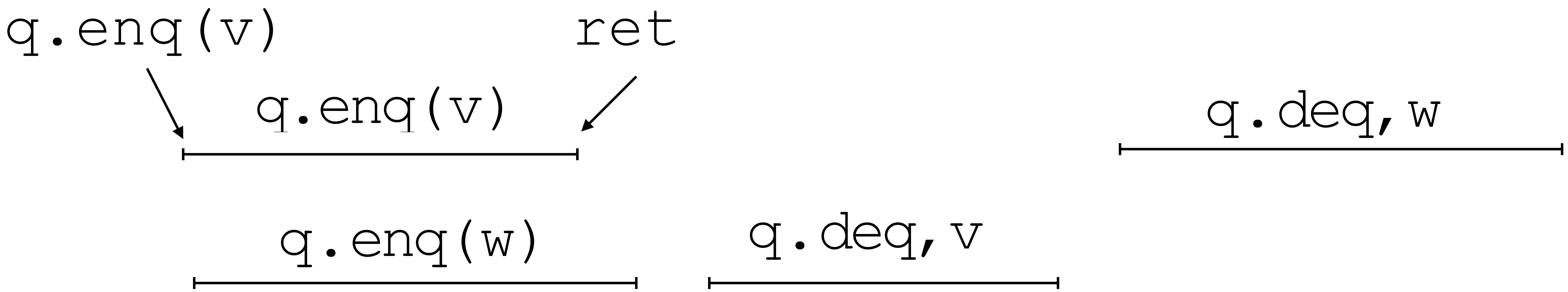
Linearizability

- Each operation takes place atomically within its call/return



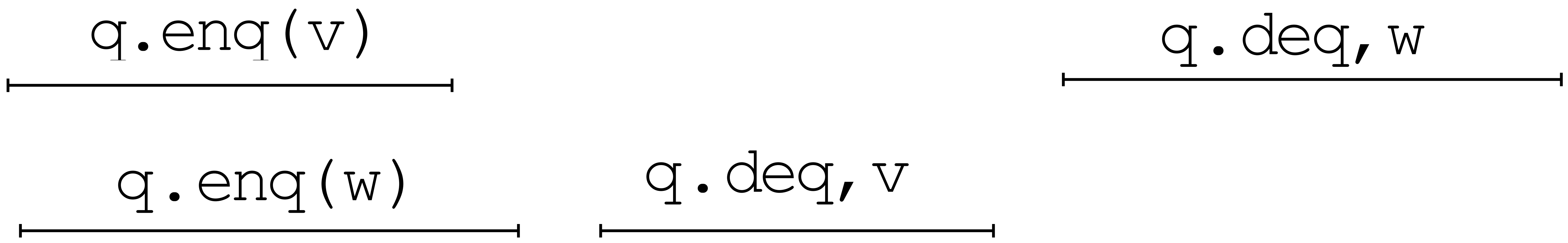
Linearizability

- Each operation takes place atomically within its call/return



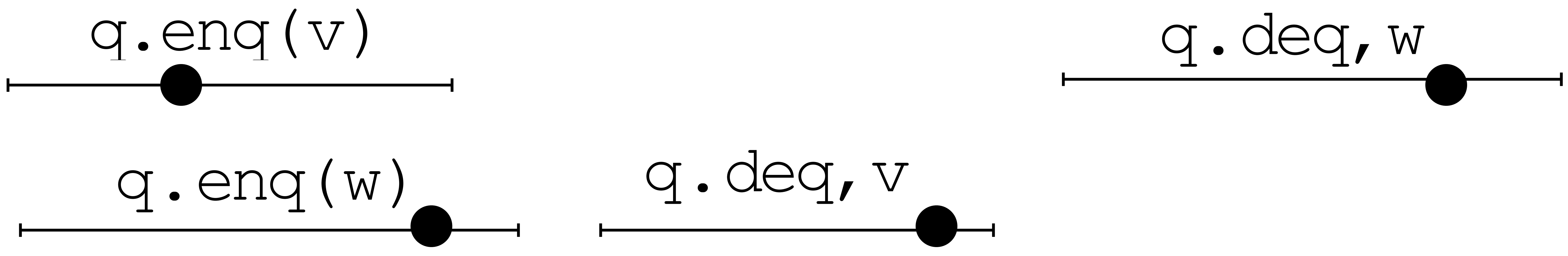
Linearizability

- Each operation takes place atomically within its call/return



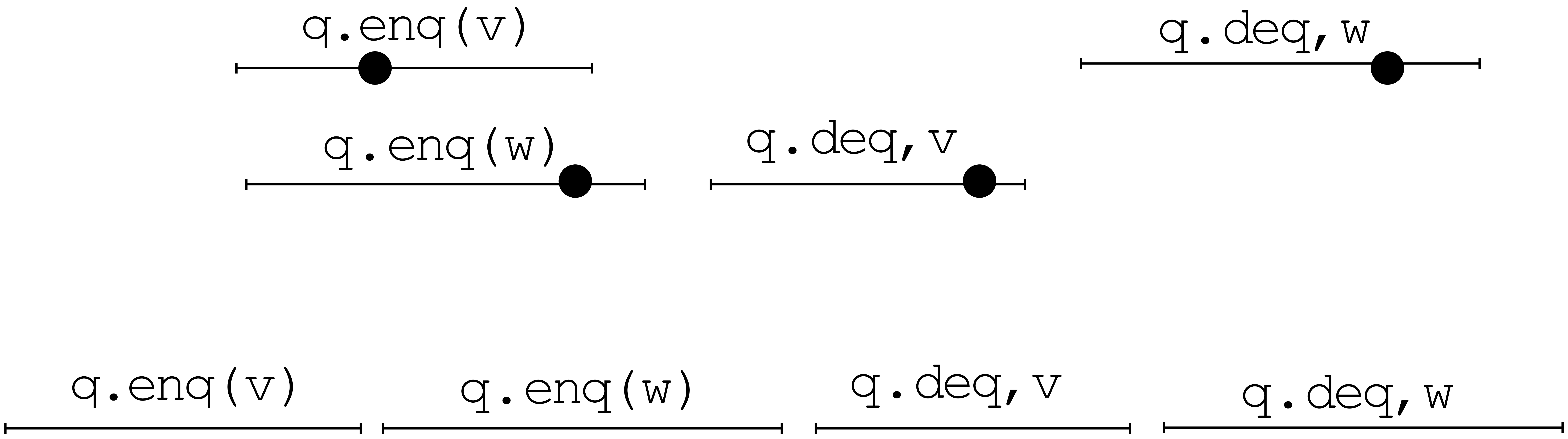
Linearizability

- Each operation takes place atomically within its call/return



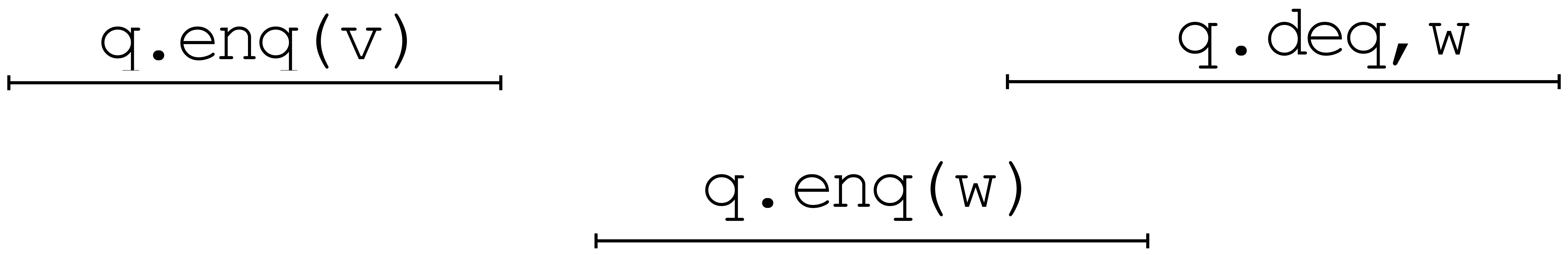
Linearizability

- Each operation takes place atomically within its call/return



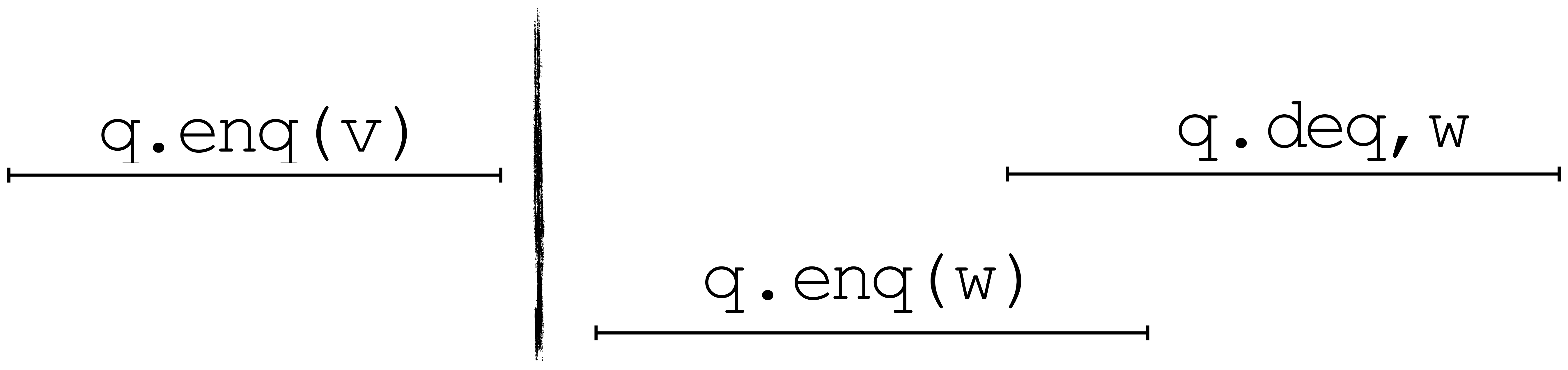
Linearizability

- Each operation takes place atomically within its call/return



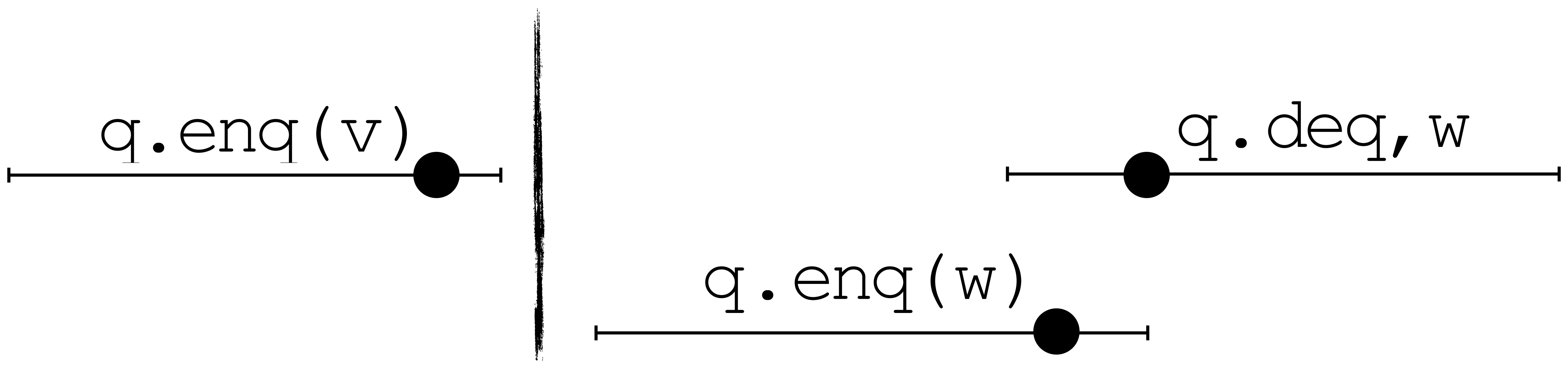
Linearizability

- Each operation takes place atomically within its call/return



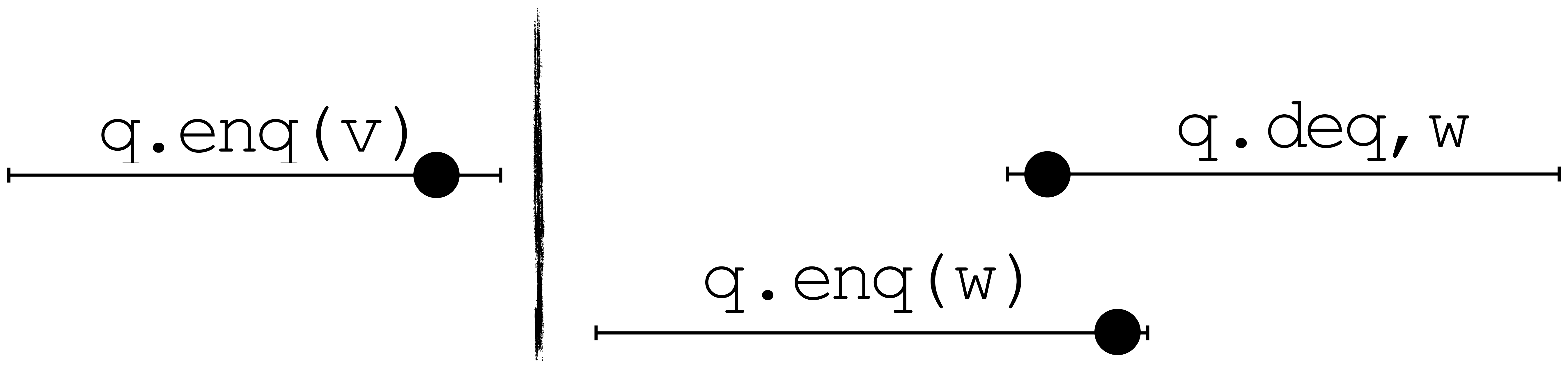
Linearizability

- Each operation takes place atomically within its call/return



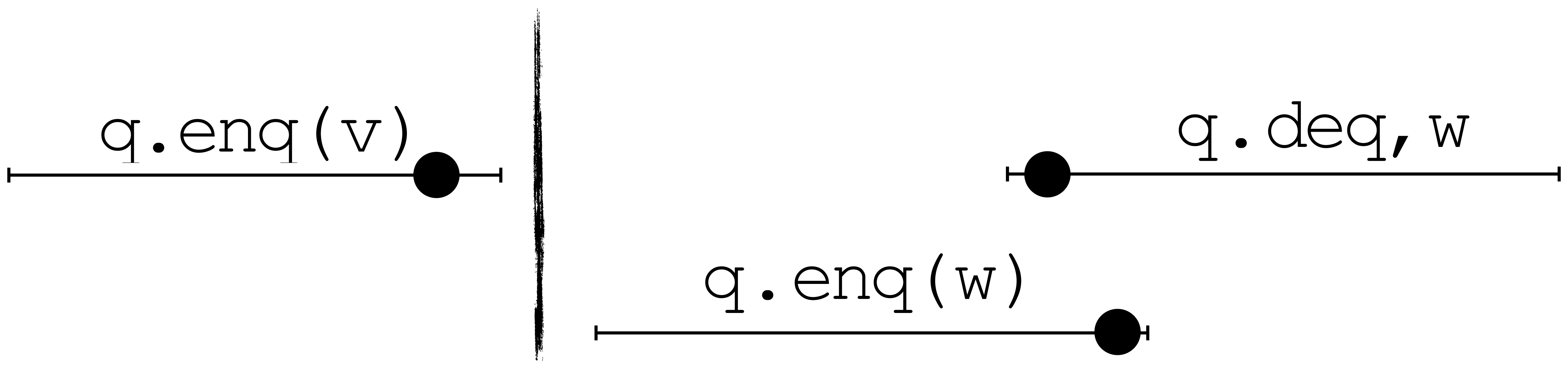
Linearizability

- Each operation takes place atomically within its call/return



Linearizability

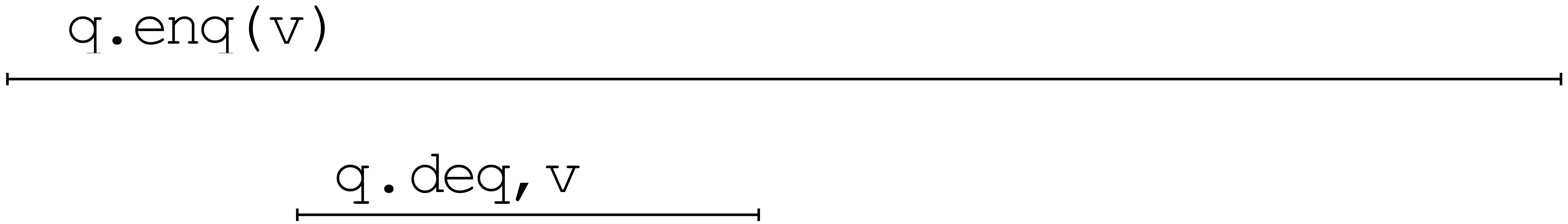
- Each operation takes place atomically within its call/return



Not Linearizable

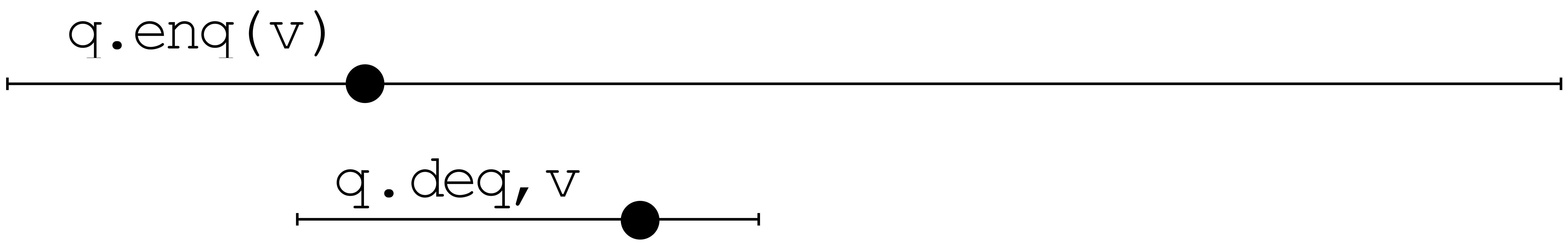
Linearizability

- Each operation takes place atomically within its call/return



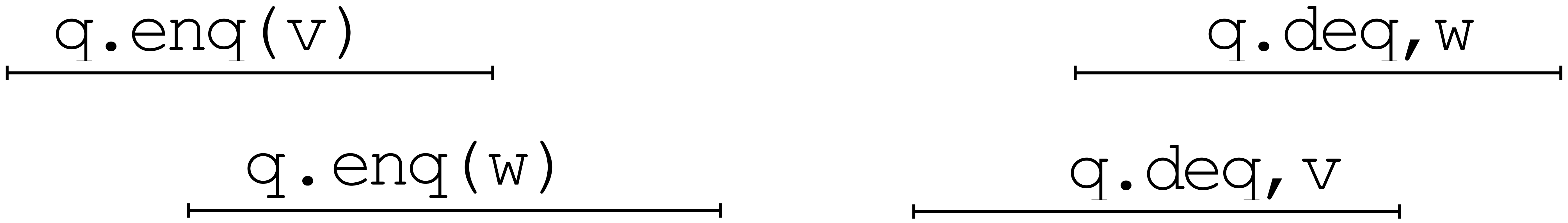
Linearizability

- Each operation takes place atomically within its call/return



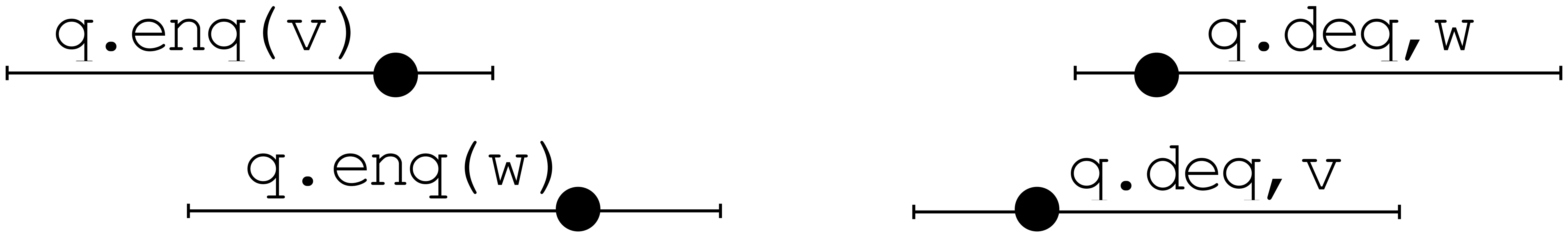
Linearizability

- Each operation takes place atomically within its call/return

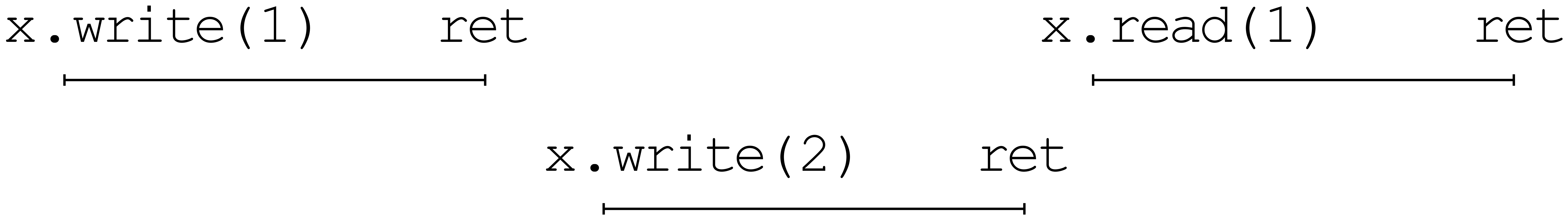


Linearizability

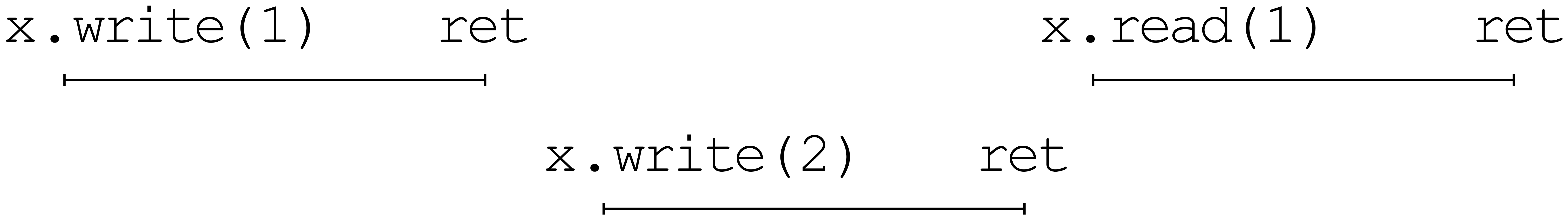
- Each operation takes place atomically within its call/return



Linearizability vs. Sequential Consistency



Linearizability vs. Sequential Consistency



Not linearizable to begin with!

Linearizability: Compositionallity

- Theorem: A history δ is linearizable if and only if for each object o in δ, δ_o is linearizable
  Proof: Simple induction on the number of operations appearing in δ
- Corollary: It is enough to show that each Library is linearizable to know that the system is

Linearizability: Proof Technique

- For each implementation method of a library:
  - Identify a *syntactic linearization point*
  - Check that for each successful execution of a method there is *exactly one linearization* point
  - Check that the *input/output corresponds to the sequential spec.* of the object

Some Object Implementations

Most code taken from *CAVE [Vafeiadis]*
https://people.mpi-sws.org/~viktor/cave/

To Lock or not to Lock?

The Art of Multiprocessor Programming
[Herlihy, Shavit'12]

To Lock or not to Lock?

```
class Queue implements Que {
    int head; // next item to dequeue
    int size; // number of items in queue
    Object[] items; // queue contents
    public Queue(int capacity) {
        head = 0; size = 0;
        items = new Object[capacity];
    }
    public synchronized void enq(Object x) {
        while (size == items.length)
            this.wait(); // wait until not full
        int tail = (head + size) % items.length;
        items[tail] = x;
        size = size + 1;
        this.notify();
    }
    public synchronized Object deq() {
        while (size == 0)
            this.wait(); // wait until non-empty
        Object x = items[head];
        size = size - 1;
        head = (head + 1) % items.length;
        this.notify();
        return x;
    }
}
```

The Art of Multiprocessor Programming
[Herlihy, Shavit'12]

To Lock or not to Lock?

```
class Queue implements Que {
    int head; // next item to dequeue
    int size; // number of items in queue
    Object[] items; // queue contents
    public Queue(int capacity) {
        head = 0; size = 0;
        items = new Object[capacity];
    }
    public synchronized void enq(Object x) {
        while (size == items.length)
            this.wait(); // wait until not full
        int tail = (head + size) % items.length;
        items[tail] = x;
        size = size + 1;
        this.notify();
    }
    public synchronized Object deq() {
        while (size == 0)
            this.wait(); // wait until non-empty
        Object x = items[head];
        size = size - 1;
        head = (head + 1) % items.length;
        this.notify();
        return x;
    }
}
```

```
class LockFreeQueue implements Que {
    int head = 0; // next item to dequeue
    int tail = 0; // next empty slot
    Object[] items; // queue contents

    public LockFreeQueue(int capacity) {
        head = 0; tail = 0;
        items = new Object[capacity];
    }

    public void enq(Object x) {
        while (tail - head == items.length);
        items[tail % items.length] = x;
        tail++;
    }

    public Object deq() {
        while (tail == head) {
            Thread.yield();
        };
        Object x = items[head % items.length];
        head++;
        return x;
    }
}
```

The Art of Multiprocessor Programming
[Herlihy, Shavit'12]

To Lock or not to Lock?

```
class Queue implements Que {
    int head; // next item to dequeue
    int size; // number of items in queue
    Object[] items; // queue contents
    public Queue(in
        head = 0; si
        items = new
    }
    public synchron
        while (size
            this.wai
        int tail =
        items[tail]
        size = size
        this.notify
    }
    public synchron
        while (size
            this.wai
        Object x = i
        size = size
        head = (heac
        this.notify
        return x;
    }
}
```

QueueStressTest.java   LockFreeQueue.java   Queue.java

```
            }
        };
    }

    public static void runTest(Que q) throws InterruptedException {
        ArrayList<Thread> thrd = new ArrayList<Thread>();
        long time0 = System.currentTimeMillis();
        for (int i=0; i < THREADS; i=i+2) {
            Thread t0 = create(ITEMS, q, i);
            t
            T
            t
            t
            t
        }
        for (Thread t : thrd) t.join();
        System.out.println("Ended with " + q.getClass() +
                           " in " + (System.currentTimeMillis() - time0)/360 +
                           " seconds");
    }

    public static void main(String[] args) throws InterruptedException {
        runTest(new Queue((int) MAXSIZE));
        runTest(new LockFreeQueue((int) MAXSIZE));
    }
}
```

Problems   Javadoc   Declaration   Console

Lets see about that

```
class LockFreeQueue implements Que {
    int head = 0; // next item to dequeue
    int tail = 0; // next empty slot
    Object[] items; // queue contents
                    Queue(int capacity) {
                    ail = 0;
                     Object[capacity];

                    (Object x) {
                    - head == items.length);
                    % items.length] = x;

                    eq() {
                     == head) {
                    ield();

                    items[head % items.length];

}
```

The Art of Multiprocessor Programming
[Herlihy, Shavit'12]

Spin Lock

```
int Lock = 0;
TID owner = null;

void lock(){
  bool l;
  do {
    while(Lock == 1);
    l = cas(Lock, 0, 1);
  until (l);
  owner = getTID();
  return;
}
```

```
void unlock(){
  owner = null;
  Lock = 0;
  return;
}
```

Counter

```
class IntPtr {
  int val;
}
IntPtr COU;
```

```
void inc(int v){
  int n;
  while(true) {
   n = COU->val;
   if (cas(COU->val, n, n+v))
     break;
  }
  return;
}
```

```
void dec(int v) {
  int n;
  while(true) {
    n = COU->val;
    if (cas(COU-val, n, n-v))
      break;
  }
  return;
}
```

```
int read() {
  return COU->val;
}
```

Stack Implementations

DCAS Stack

```
class Node {          class NodePtr {
  Node tl;              Node val;
  int val;            } TOP;
}
```

```
void push(int e) {
  Node y, n;
  y = new();
  y->val = e;
  while(true) {
    n = TOP->val;
    y->tl = n;
    if (cas(TOP->val, n, y))
      break;
  }
}
```

```
int pop() {
  Node y,z;
  while(true){
    y = TOP->val;
    if (y==0)
      return EMPTY;
    else {
      z = y->tl;
     if (dcas(TOP->val,y,y->tl,z, z)
       break;
    }
  }
  return y->val;
}
```

Treiber Stack

```
class Node {
  Node tl;
  int val;
}
```

```
class NodePtr {
  Node val;
} TOP;
```

```
void push(int e) {
  Node y, n;
  y = new();
  y->val = e;
  while(true) {
    n = TOP->val;
    y->tl = n;
    if (cas(TOP->val, n, y))
      break;
  }
}
```

```
int pop() {
  Node y,z;
  while(true) {
    y = TOP->val;
    if (y==0) return EMPTY;
    z = y->tl;
    if (cas(TOP->val, y, z))
      break;
  }
  return y->val;
}
```


[Treiber'86]

HSY Elimination Stack

Extremely simplified version: 1 collision

```
class Node {
  Node tl;
  int val;
}

class NodePtr {
  Node val;
} TOP;

class TidPtr {
  int val;
} clash;
```

```
void push(int e) {
  Node y, n;
  TID hisId;
  y = new();
  y->val = e;

  while (true) {
    n = TOP->val;
    y->tl = n;
    if (cas(TOP->val, n, y))
      return;
    //elimination scheme
   TidPtr t = new TidPrt();
   t->val = e;
    if (cas(clash,null,t)){
      wait(DELAY);
      //not eliminated
      if (cas(clash,t,null))
          continue;
      else break; //eliminated
    }
   }
}
```

```
int pop() {
  Node y,z;
  int t;
  TID hisId;
  while (true) {
    y = TOP->val;
    if (y == 0)
      return EMPTY;
    z = y->tl;
    t = y->val;
    if (cas(TOP->val, y, z)
      return t;
    //elimination scheme
    pusher = clash;
    while (pusher!=null){
     if (cas(clash,pusher,null)
      //eliminated push
      return pusher->val;
  }
}
```

[Hendler et al.'04]

Queue Implementations

Two Locks Queue

```
class Node {
  int val;
  Node tl;
}

class Queue {
  Node head;
  Node tail;
  thread_id hlock;
  thread_id tlock;
} Q;
```

```
void enqueue(int v) {
  Node n, t;
  n = new();
  n->val = v;
  n->tl = NULL;
  lock (&Q->tlock);
  temp = Q->tail;
  temp->tl = node;
  Q->tail = node;
  unlock (&Q->tlock);
}
```

```
int dequeue() {
  Node n, new_h;
  int v;
  lock (&Q->hlock);
  node = Q->head;
  new_h = n->tl;}
  if (new_h == NULL) {
    unlock (&Q->hlock);
    return EMPTY;
  } else {
    value = new_head->val;
    Q->head = new_head;
    unlock (&Q->hlock);
    //dispose(n);
    return v;
  }
}
```

Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms
[Michael,Scott'96]

Michael and Scott Queue

```
class Node {
  int val;
  Node tl;
}

class Queue {
  Node head;
  Node tail;
} Q;
```

```
void enqueue(int v) {
  Node nd, nxt, tl;
  int b1;
  nd = new();
  nd->val = v;
  nd->tl = NULL;
  while(true) {
    tl = Q->tail;
    nxt = tl->tl
    if (Q->tail == tl) b1 = 1;
    else b1 = 0;
    if (b1!=0)
      if (nxt == 0)
        if (cas(tl->tl,nxt,nd))
          break;
      else cas(Q->tail,tl, nxt);
  }
  cas(Q->tail, tl, nd);
}
```

```
int dequeue() {
  Node nxt, hd, tl;
  int pval;
  while(true) {
    hd = Q->head;
    tl = Q->tail;
    nxt = hd->tl;
    if (Q->head != hd) continue;
    if (hd == tl) {
      if (nxt == NULL)
        return EMPTY;
      cas(Q->tail, tl, nxt);
    } else {
      pval = next->val;
      if (cas(Q->head, hd, nxt))
        return pval;
    }
  }
}
```

[Michael,Scott'96]

Herlihy Wing Queue

```
class Node {
  int val; // -1 NAN
  Node tl;
  thread_id alloc;
}

class Queue {
  Node head;
  Node tail;
} Q;
```

```
void enqueue(int value) {
  Node nd, tl;
  nd = new();
  nd->alloc = TID;
  nd->val = -1;
  nd->tl = NULL;
  atomic {
    tl = Q->tail;
    tl->tl = nd;
    Q->tail = nd;
  } // end of slot reservation;
  nd->val = value;//value written;
}
```

```
int dequeue() {
  Node curr, tail;
  int pval;
  while (true) {
    curr = Q->head;
    tail = Q->tail;
    while (curr != tail) {
      atomic { //atomic swap
        pval = curr->val;
        curr->val = -1;
        if (pval != -1)
          return pval;
        curr = curr->tl;
      }
    }
  }
}
```

Linearizability: A Correctness Condition for Concurrent Objects
[Herlihy and Wing '90]

Set Implementations

Lock Coupling Set

```
class Nd {
  thread_id lk;
  int val;
  Node tl;
} head, tail;
```

```
(Nd, Nd) locate(int k)
{
  Node p, c, t2;
  int t;
  p = head;
  lock (&p->lk);
  c = p->tl;
  t = c->val;
  while(t < k) {
    lock (&c->lk);
    unlock (&p->lk);
    p = c;
    c = p->tl;
    t = c->val;
  }
  return (p, c);
}
```

```
bool add(int key){
  Node p, c, t2;
  (p, c) = locate(key);
  if (c->val > key) {
    lock (&c->lk);
    t2 = new();
    t2->lk = 0;
    t2->val = key;
    t2->tl = c;
    p->tl = t2;
    unlock (&p->lk);
    unlock (&c->lk);
    return true;
  } else {
    unlock (&p->lk);
    return false;
  }
}
```

```
bool remove(int key){
  Node p, c, t2;
  (p,c) = locate(key);
  if (c->val ==key ) {
    lock (&c->lk);
    t2 = c->tl;
    p->tl = t2;
    dispose(c);
    unlock (&p->lk);
    return true;
  } else {
    unlock (&p->lk);
    return false;
  }
}
```

[Vafeiadis'?]

Lock Coupling Set (complete)

```
class Nd {
  thread_id lk;
  int val;
  Node tl;
}

Nd head, tail;
```

```
(Nd, Nd) locate(int k)
{
  Node p, c, t2;
  int t;
  p = head;
  lock (&p->lk);
  c = p->tl;
  t = c->val;
  while(t < k) {
    lock (&c->lk);
    unlock (&p->lk);
    p = c;
    c = p->tl;
    t = c->val;
  }
  return (p, c);
}
```

```
bool add(int key){
  Node p, c, t2;
  (p, c) = locate(key);
  if (c->val > key) {
    lock (&c->lk);
    t2 = new();
    t2->lk = 0;
    t2->val = key;
    t2->tl = c;
    p->tl = t2;
    unlock (&p->lk);
    unlock (&c->lk);
    return true;
  } else {
    unlock (&p->lk);
    return false;
  }
}
```

```
bool remove(int key){
  Node p, c, t2;
  (p,c) = locate(key);
  if (c->val ==key ) {
    lock (&c->lk);
    t2 = c->tl;
    p->tl = t2;
    dispose(c);
    unlock (&p->lk);
    return true;
  } else {
    unlock (&p->lk);
    return false;
  }
}
```

```
bool contains(int key){
  Node p, c;
  (p, c) = locate(key);
  if (c-> val == key) {
    unlock (&p->lk);
    return true;
  } else {
    unlock (&p->lk);
    return false;
  }
}
```

[Vafeiadis'?]

Hindsight Set

```
class Node {
  int val;
  bool marked;
  Node next;
} Head, Tail;
```

```
bool contains(int key) {
  Node pr, cr;
  int k;

  pr = Head;
  cr = Head->next;
  k = cr->val;
  while (k < key) {
    pr = cr;
    cr = cr->next;
    k = cr->val;
  }
  return (k == key);
}
```

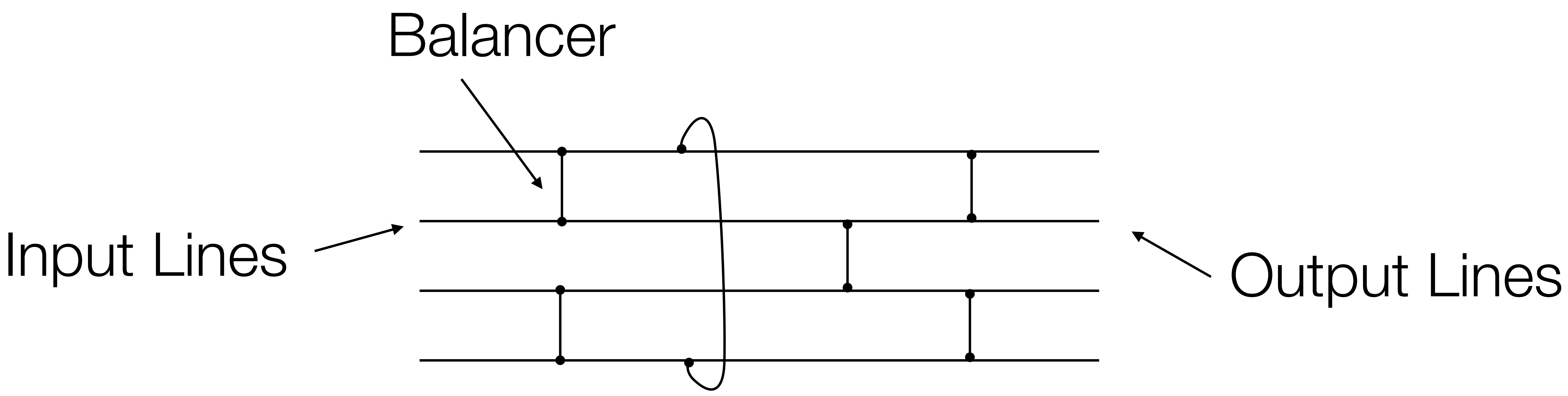
```
bool add(int key) {
  Node pr, cr, nw;
  int k;
  while (true) {
    pr = Head;
    curr = Head->next;
    k = curr->val;
    while (k < key) {
      pr = cr;
      cr = cr->next;
      k = cr->val;
    }
    if (k == key)
      return false;
    nw = new();
    nw->val = key;
    nw->marked = false;
    nw->next = curr;
    if (cas(pr->next, cr, nw))
      if (!pr->marked)
        return true;
  }
}
```

```
bool remove(int key) {
  Node pr, cr, nxt;
  int k;
  while (true) {
    pr = Head;
    curr = Head->next;
    k = curr->val;
    while (k < key) {
      pr = cr;
      cr = cr->next;
      k = cr->val;
    }
    if (k > key)
      return false;
    atomic {
      if (pr->next == cr && !pr->marked) {
        nxt = cr->next;
        cr->marked = true;
        pr->next = next;
        return true;
      }
    }
  }
}
```

Verifying Linearizability with Hindsight
[O'Hearn et al.'10]

Quiescent Consistent Objects

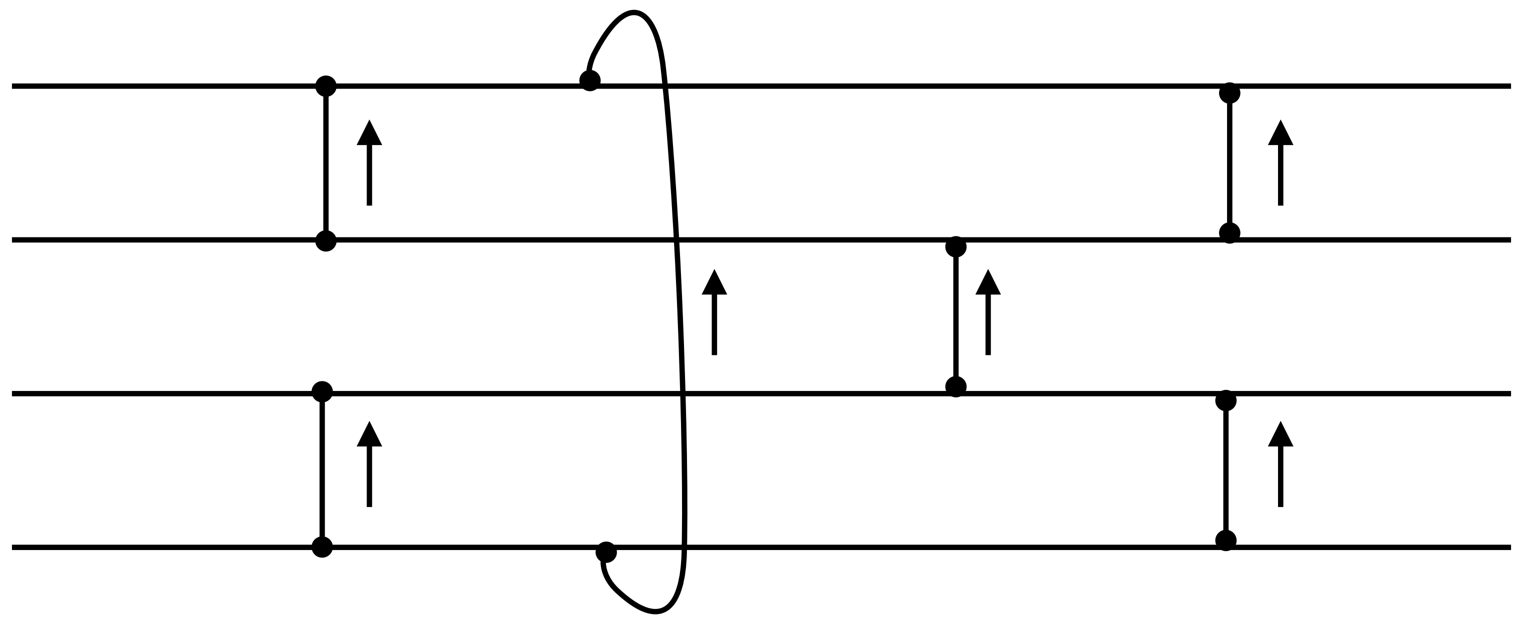
Quiescent Consistency: Counting Networks



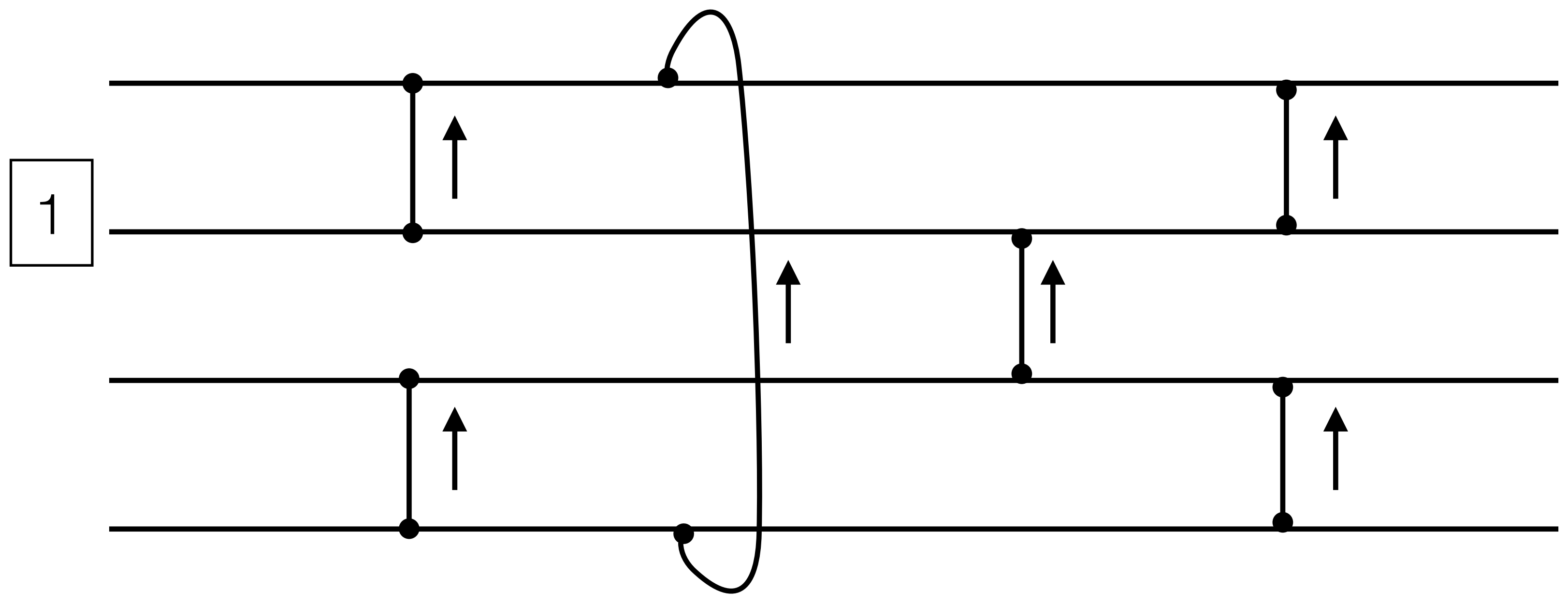

```
balancer = [toggle: boolean, next: array [0..1] of ptr to balancer]
traverse(b: balancer)
    loop until leaf(b)
        i := rmw(b.toggle := ¬ b.toggle)
        b := b.next[i]
        end loop
    end traverse
```

Counting Networks
[Aspnes et al.'94]

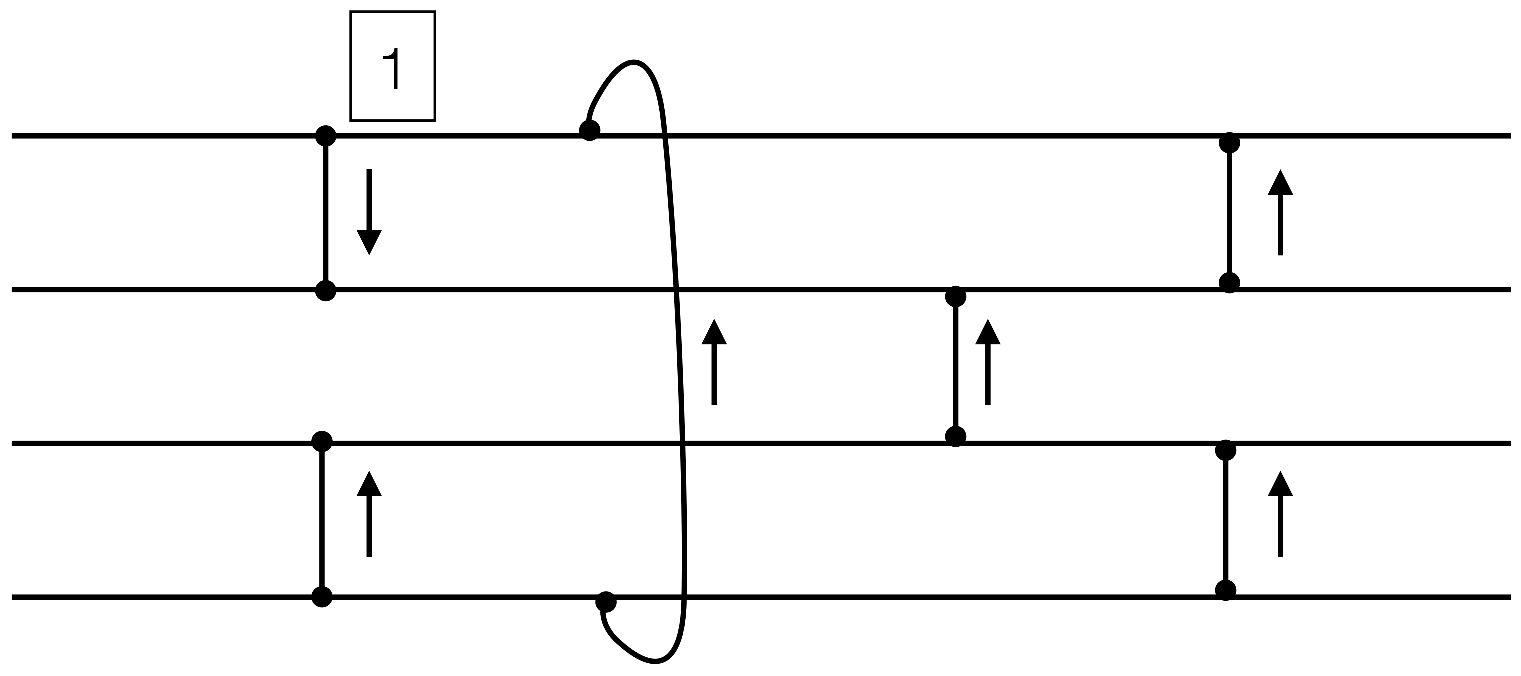
Quiescent Consistency: Counting Networks



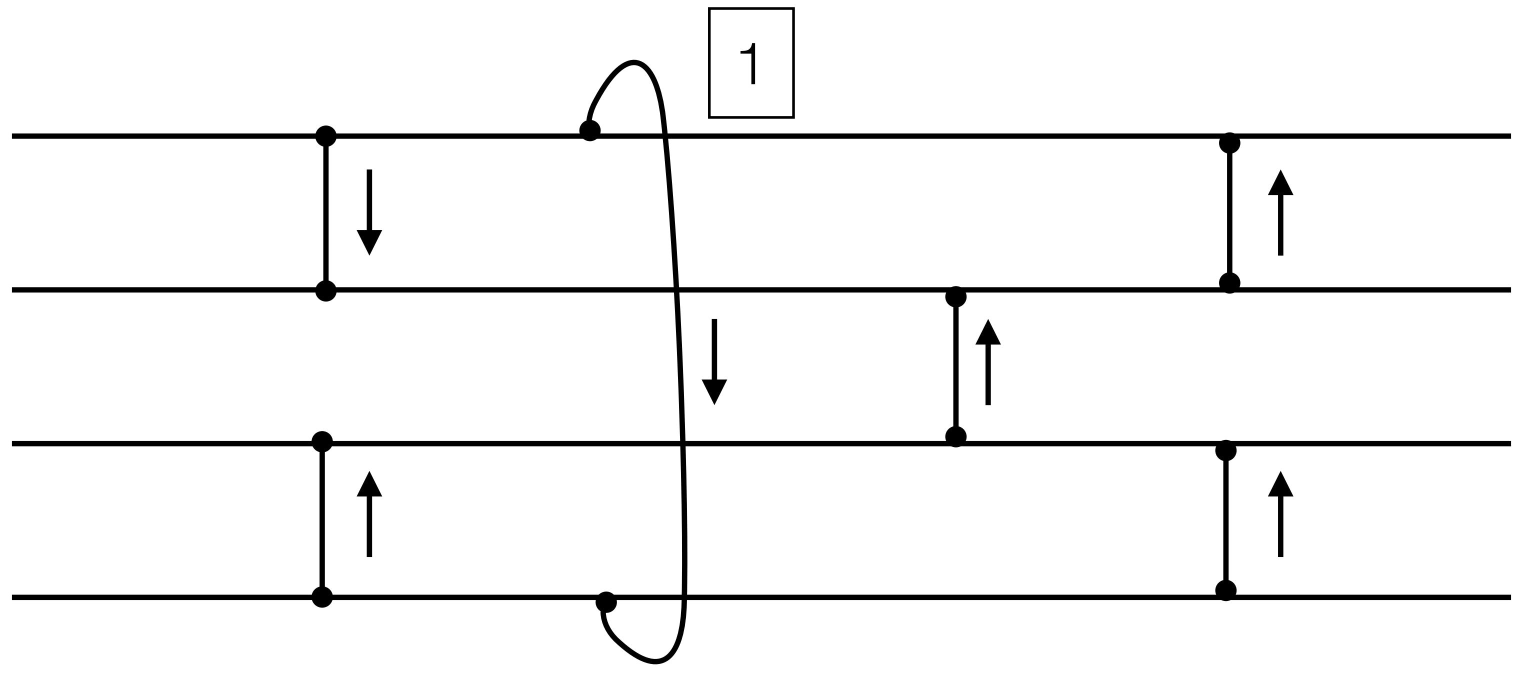
Quiescent Consistency: Counting Networks



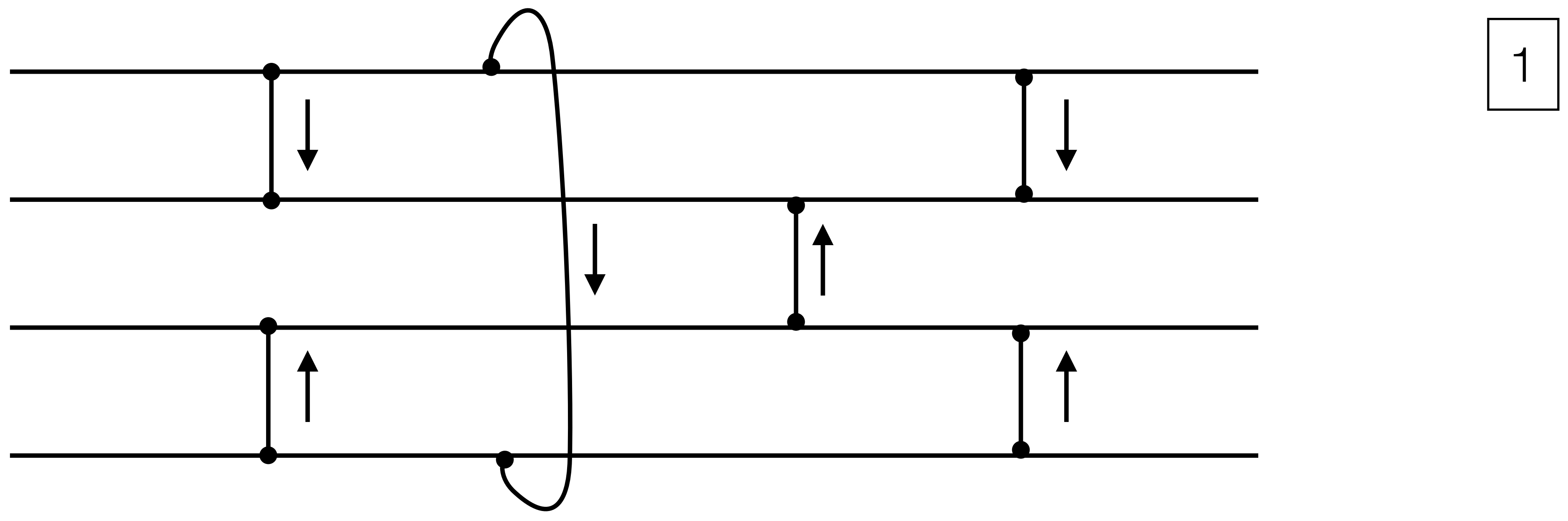
Quiescent Consistency: Counting Networks



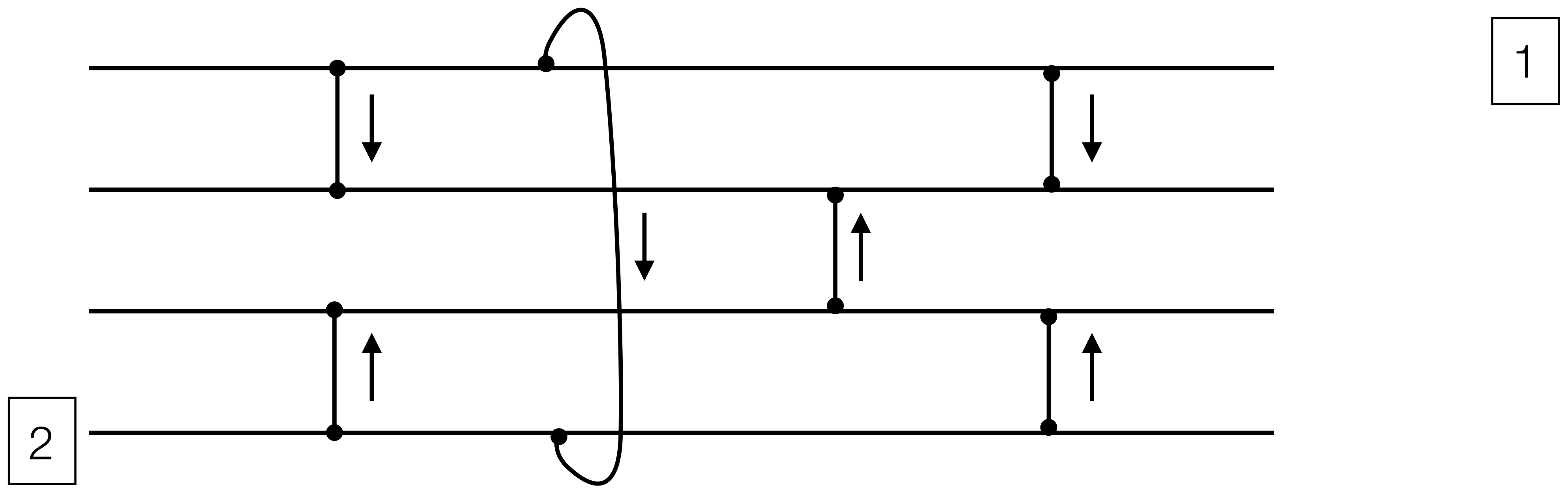
Quiescent Consistency: Counting Networks



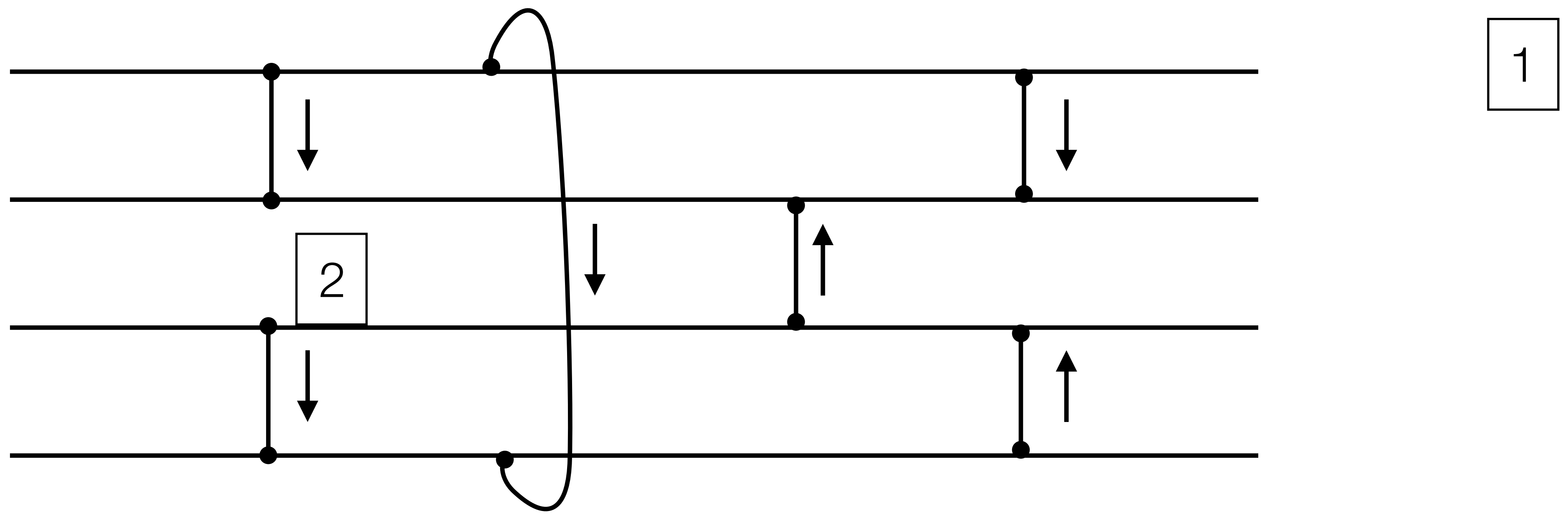
Quiescent Consistency: Counting Networks



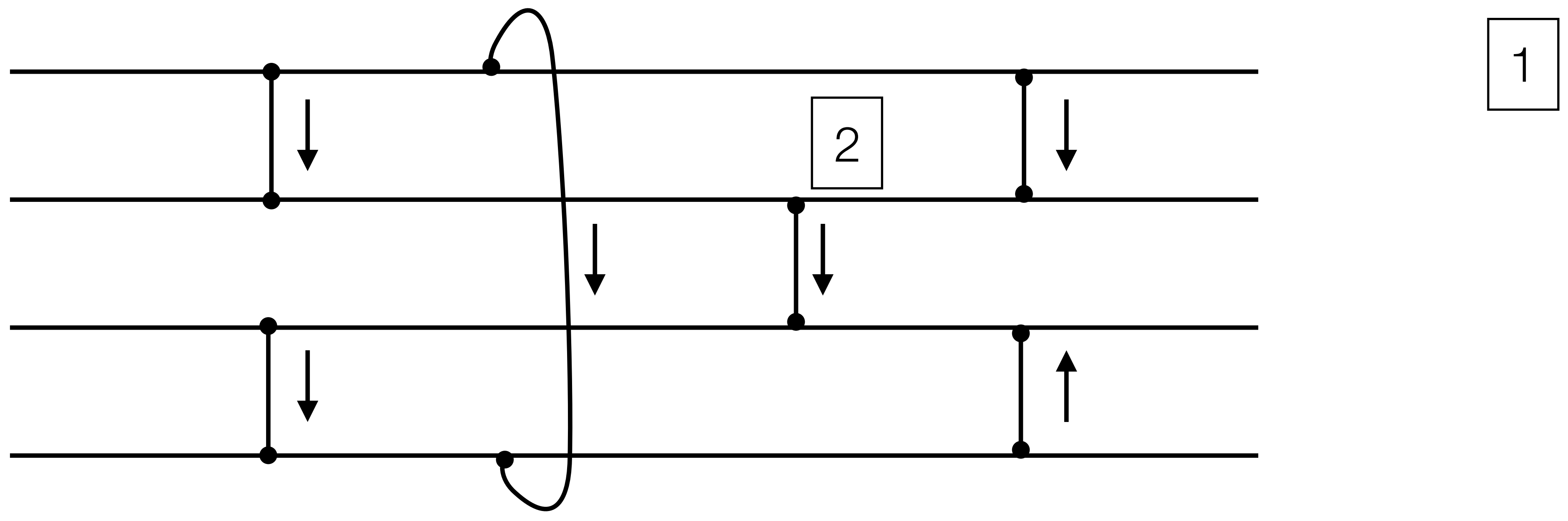
Quiescent Consistency: Counting Networks



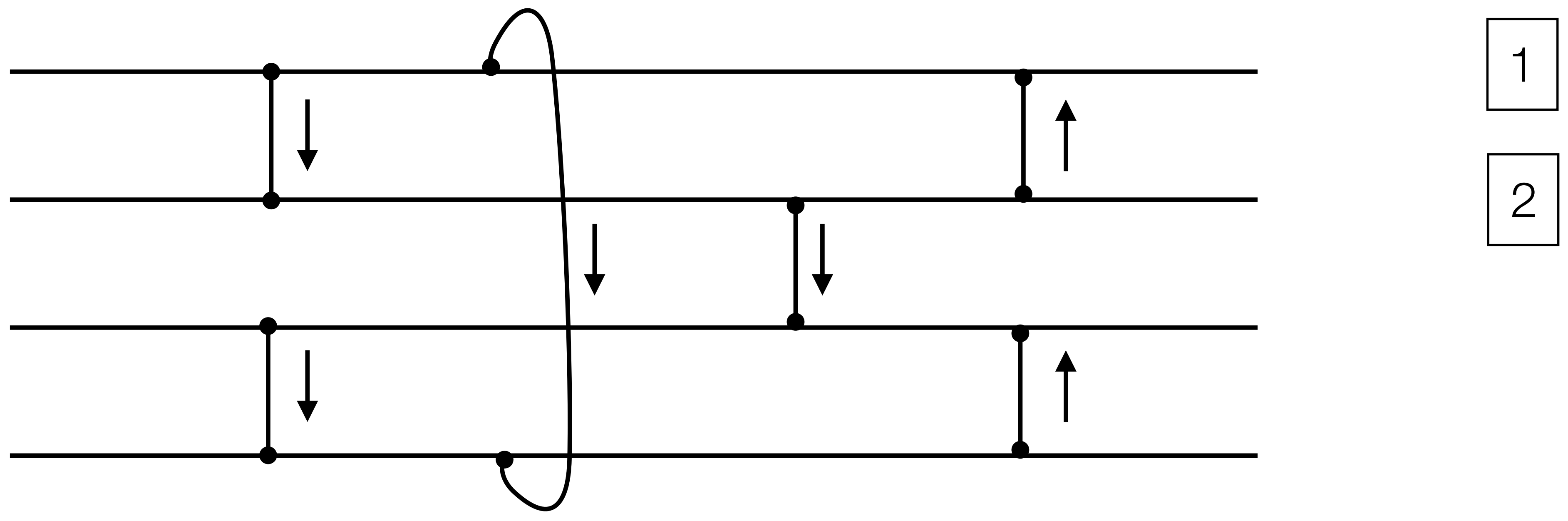
Quiescent Consistency: Counting Networks



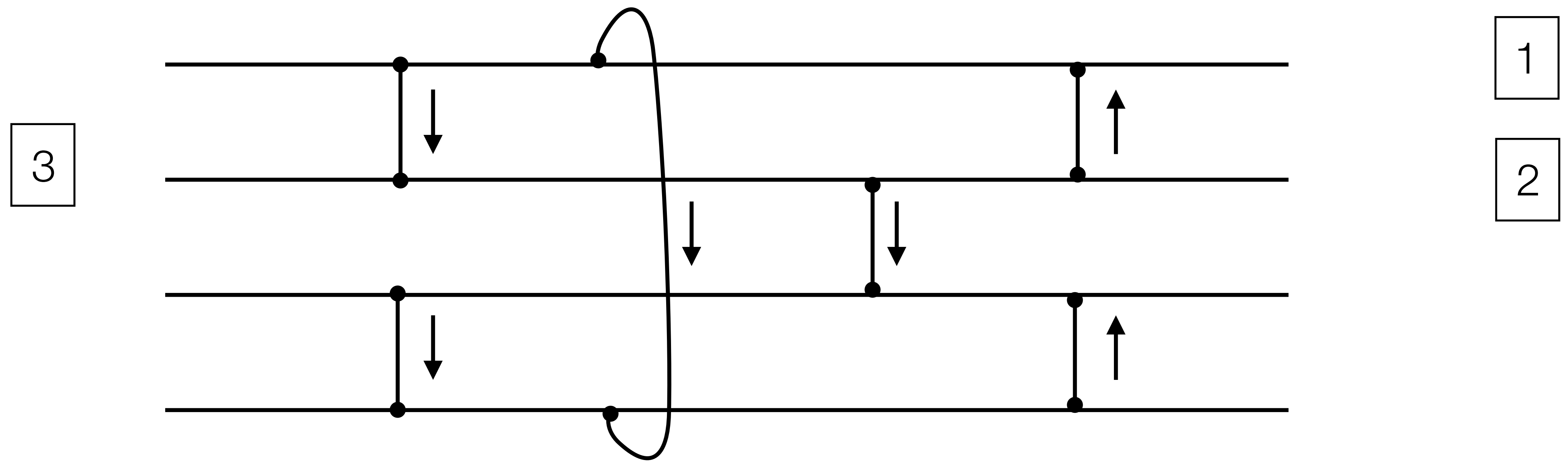
Quiescent Consistency: Counting Networks



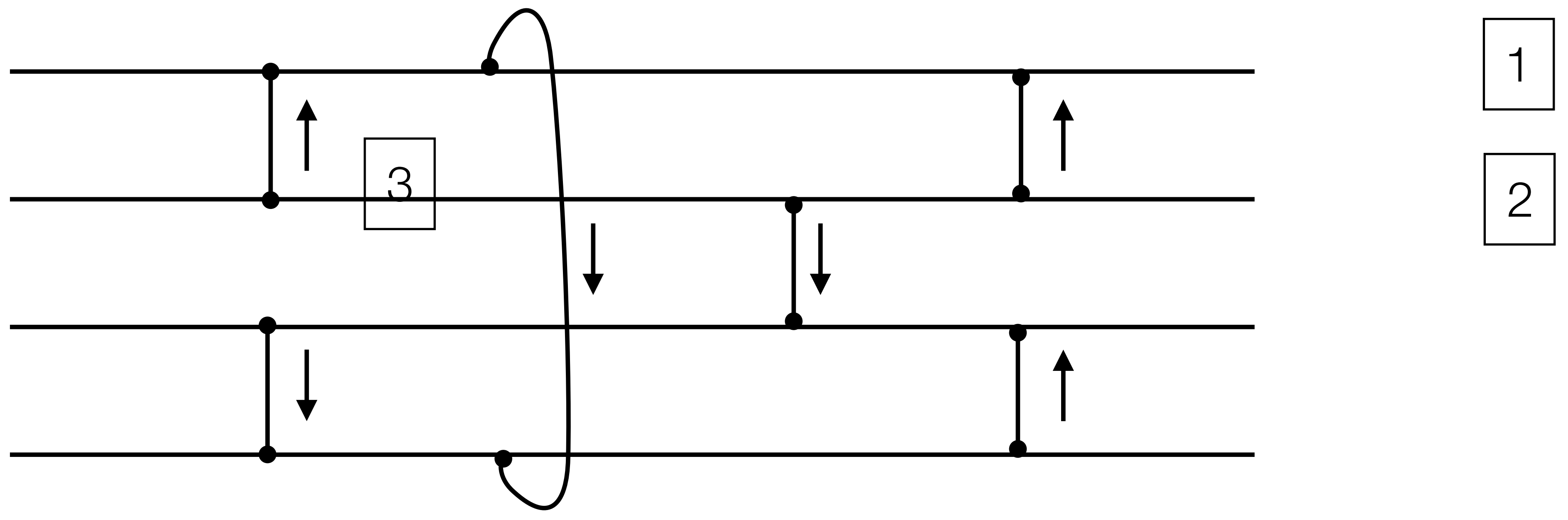
Quiescent Consistency: Counting Networks



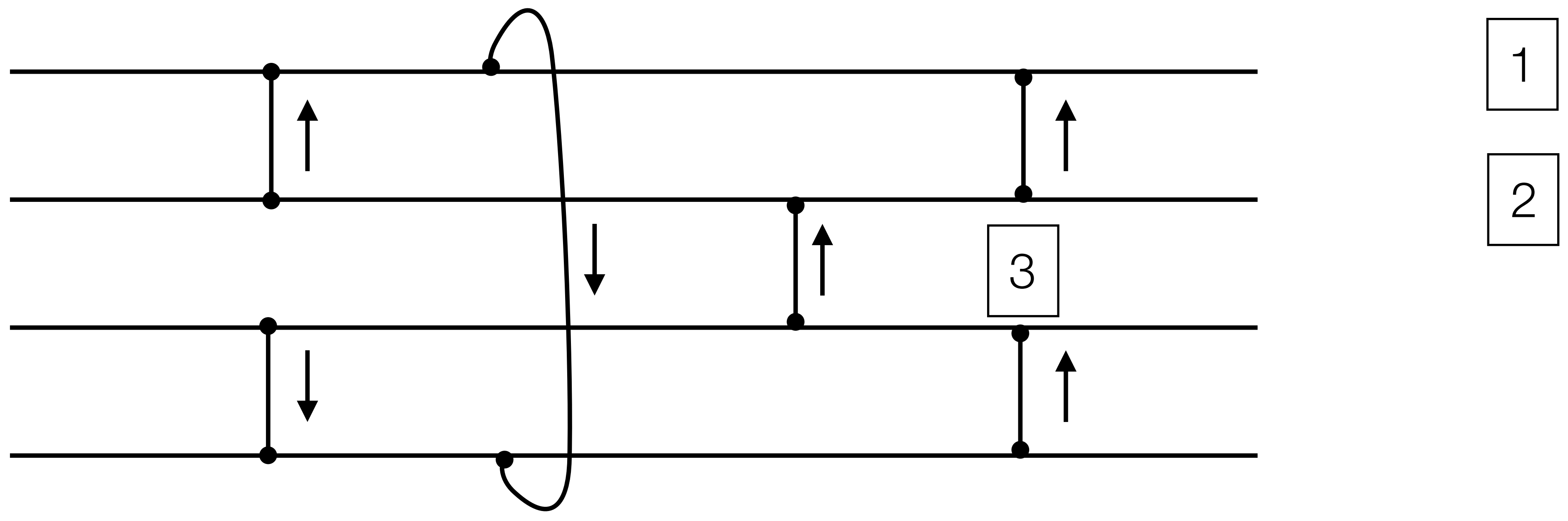
Quiescent Consistency: Counting Networks



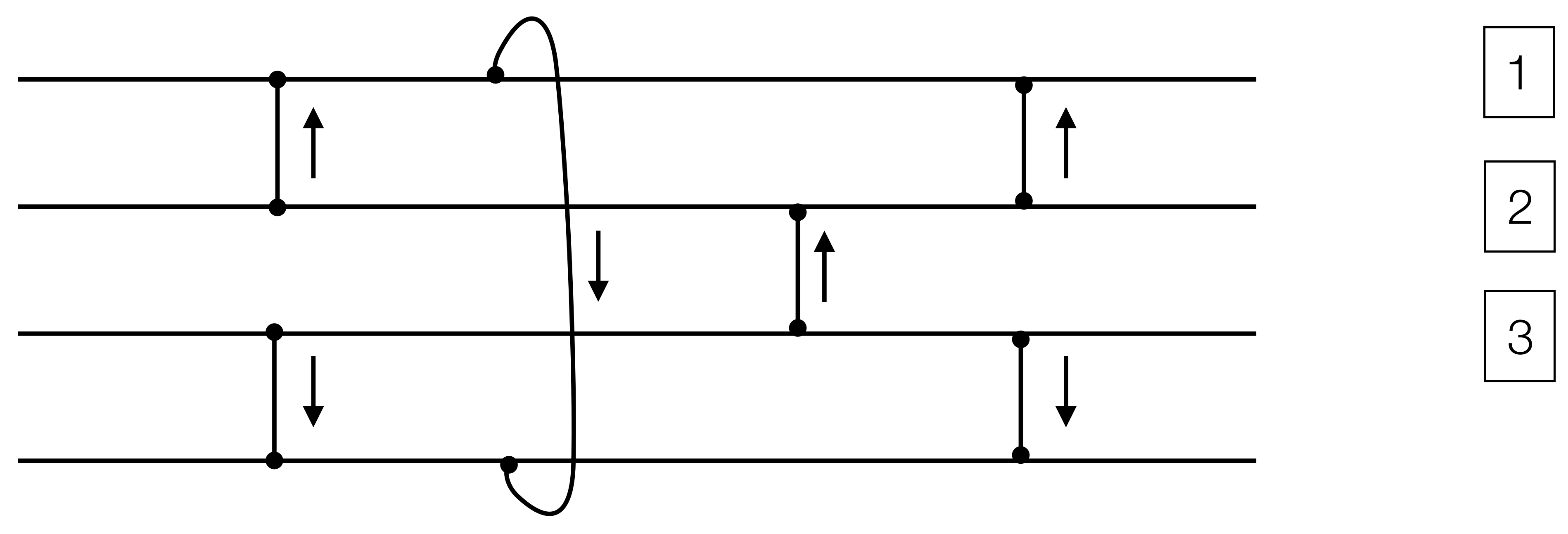
Quiescent Consistency: Counting Networks



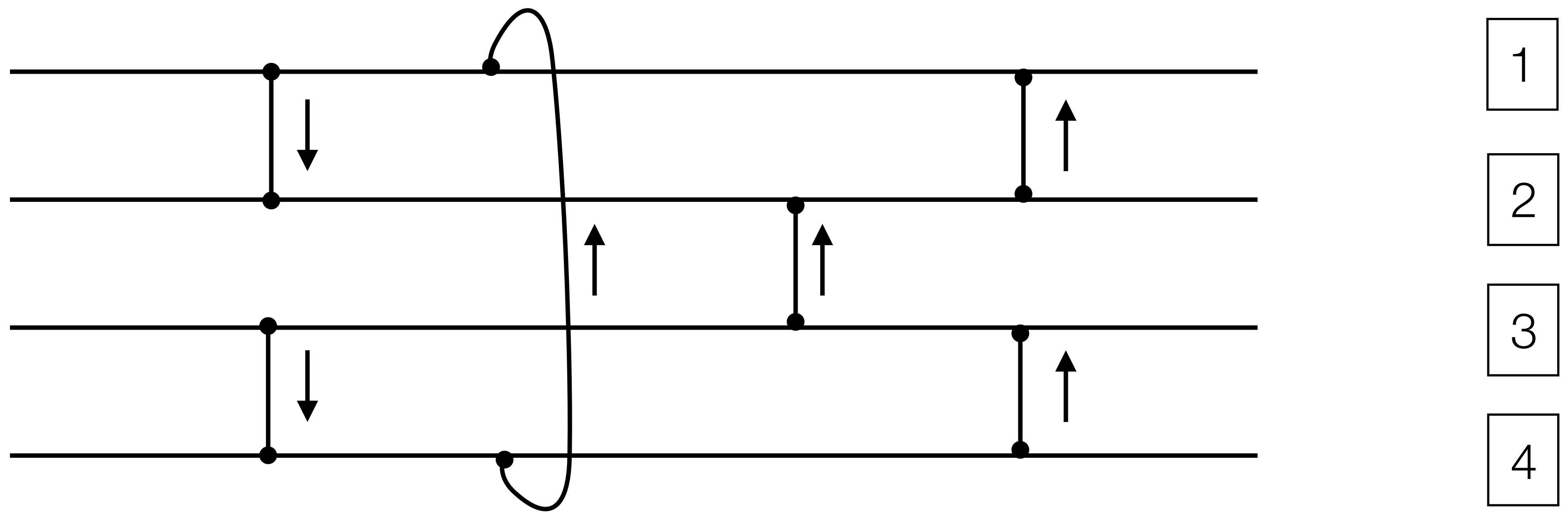
Quiescent Consistency: Counting Networks



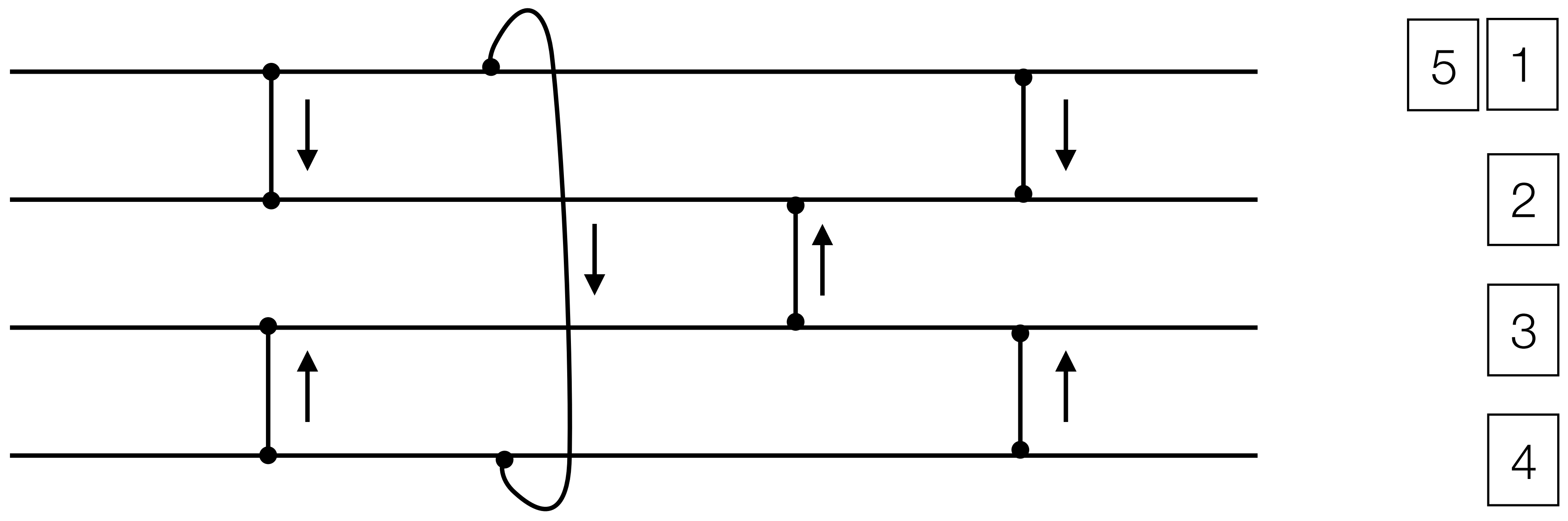
Quiescent Consistency: Counting Networks



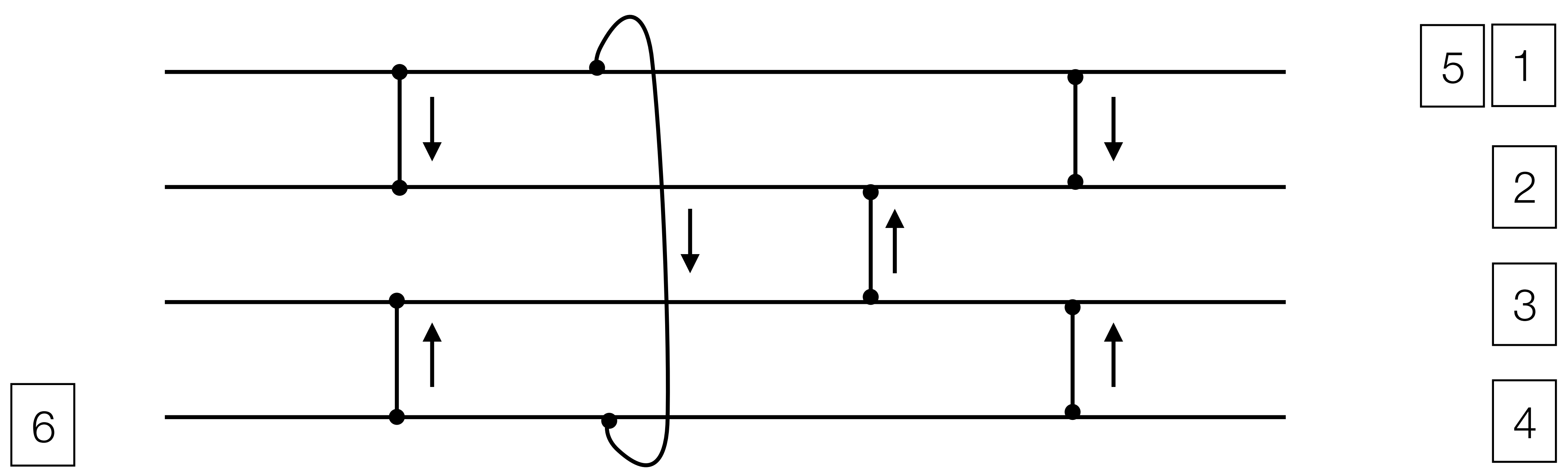
Quiescent Consistency: Counting Networks



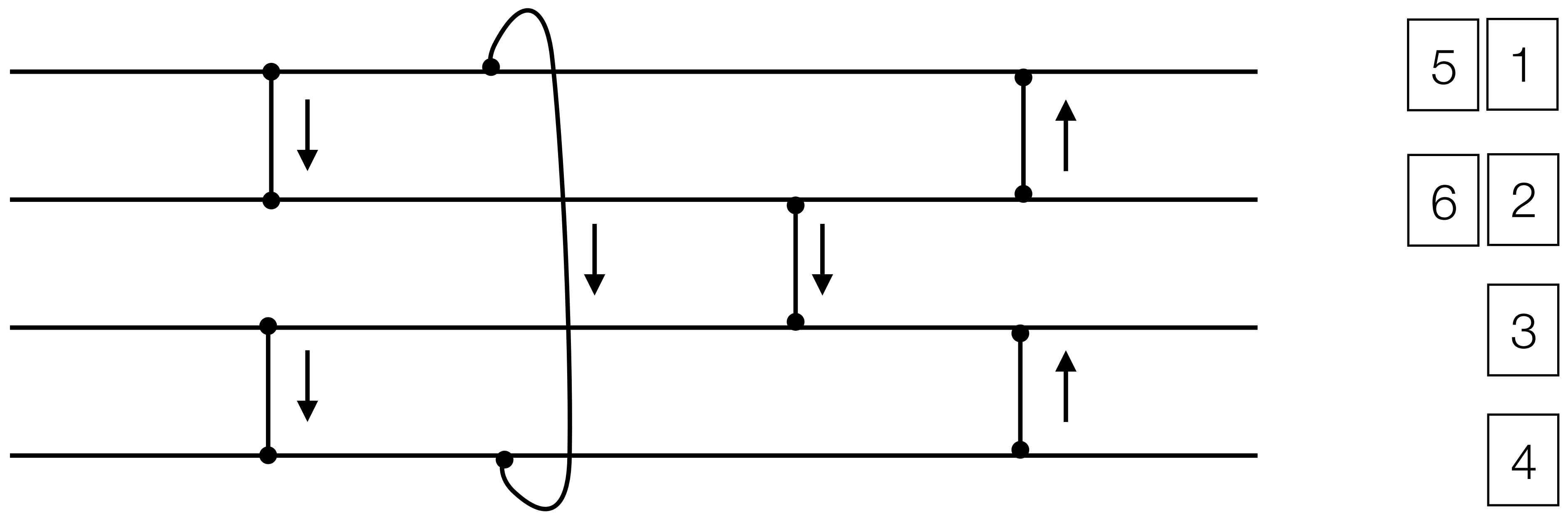
Quiescent Consistency: Counting Networks



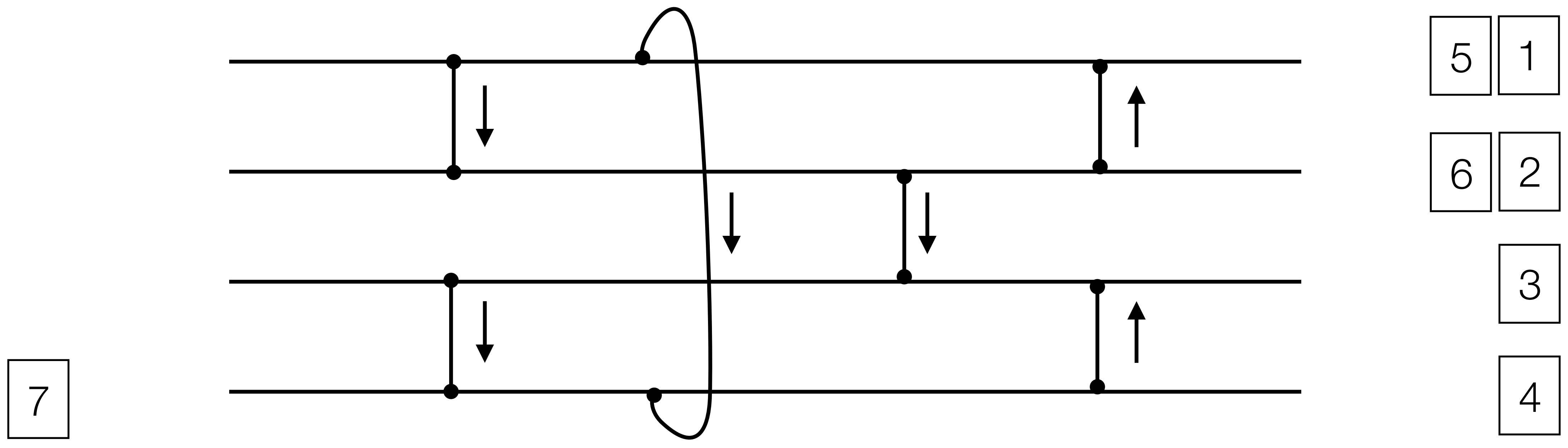
Quiescent Consistency: Counting Networks



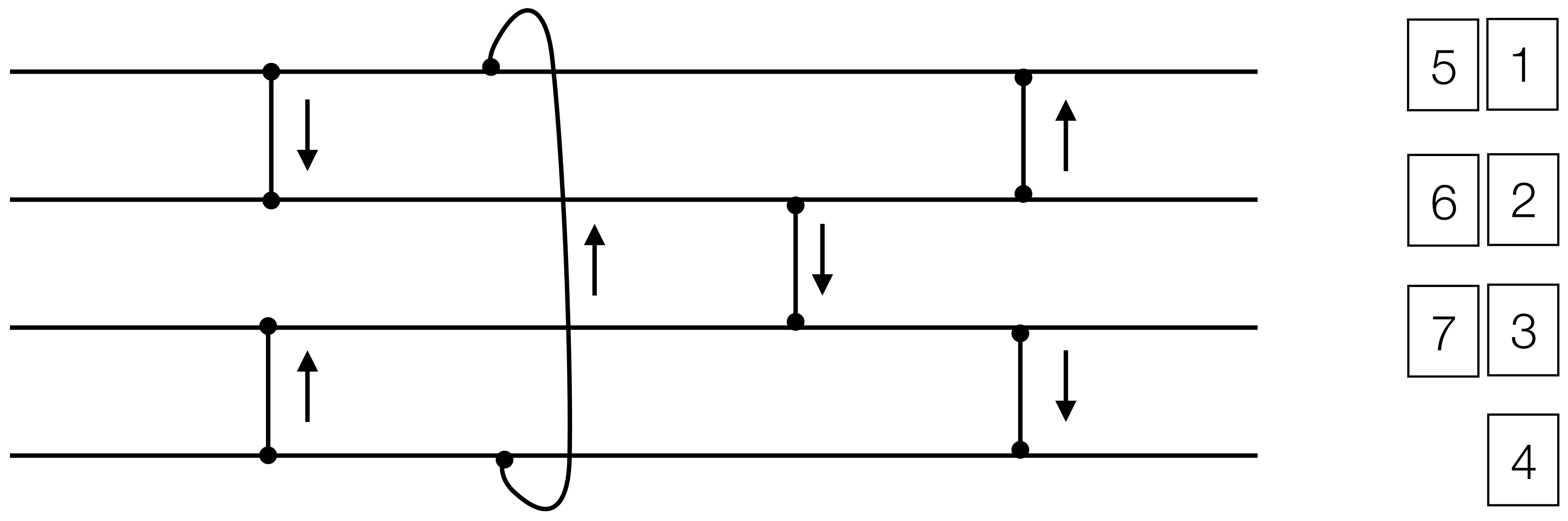
Quiescent Consistency: Counting Networks



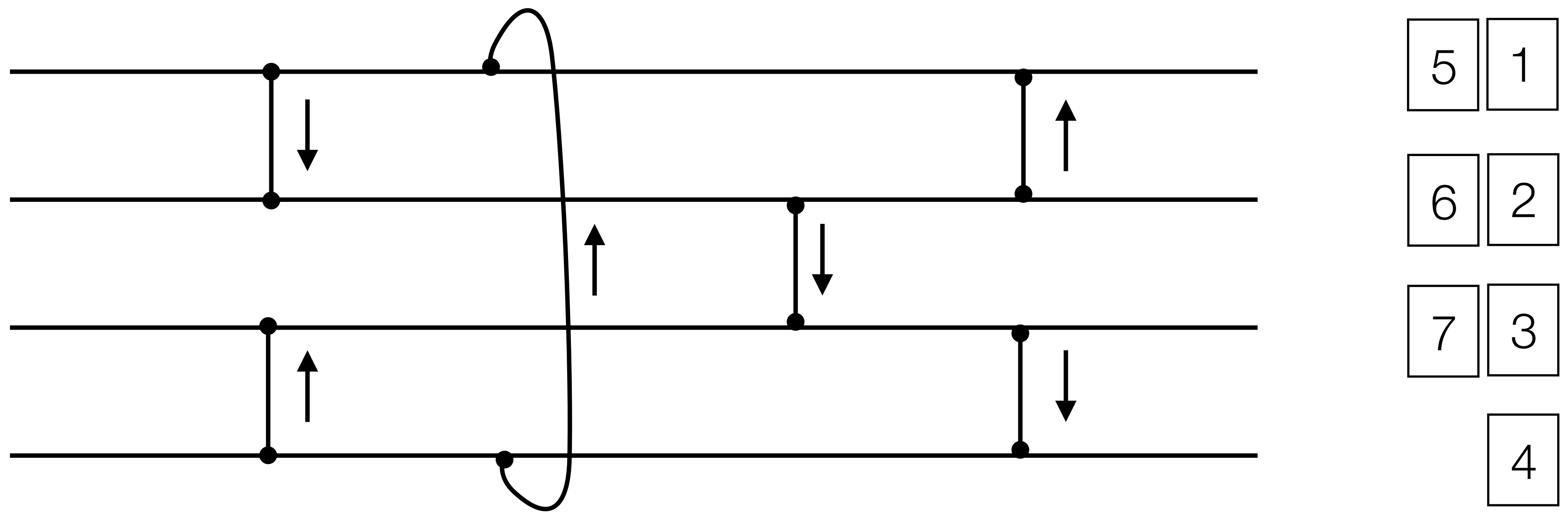
Quiescent Consistency: Counting Networks

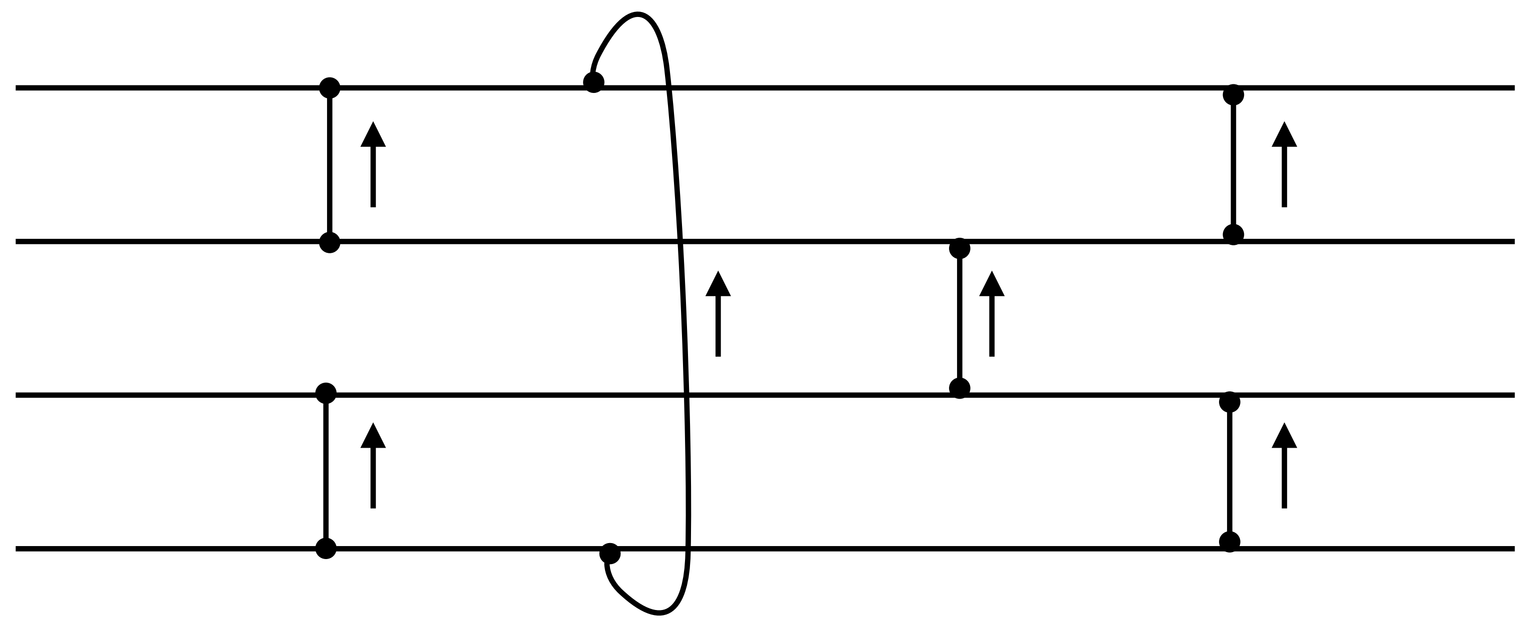


Quiescent Consistency: Counting Networks

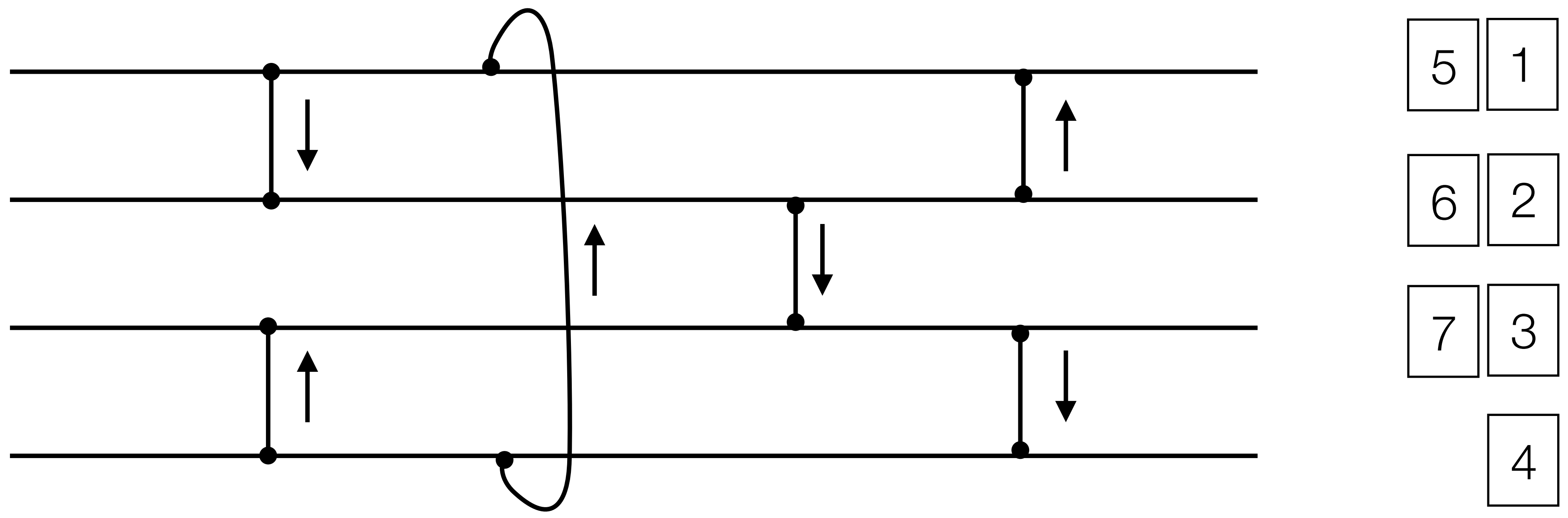


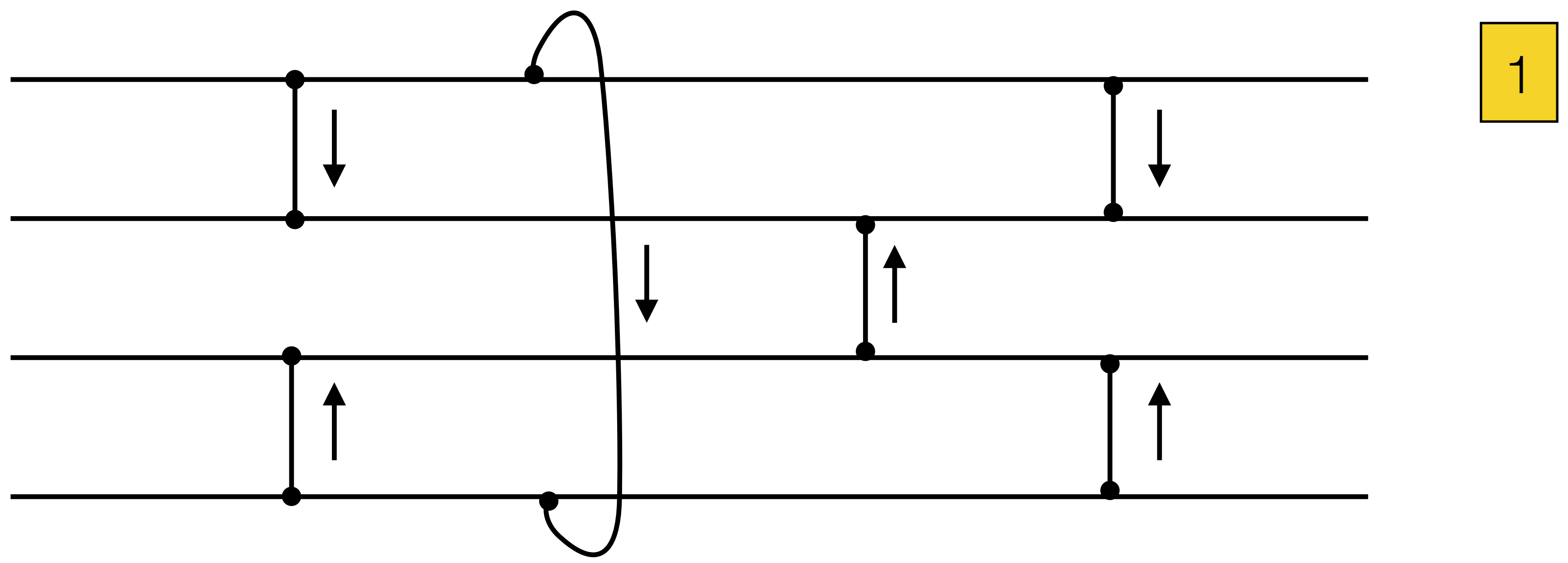
Quiescent Consistency: Counting Networks



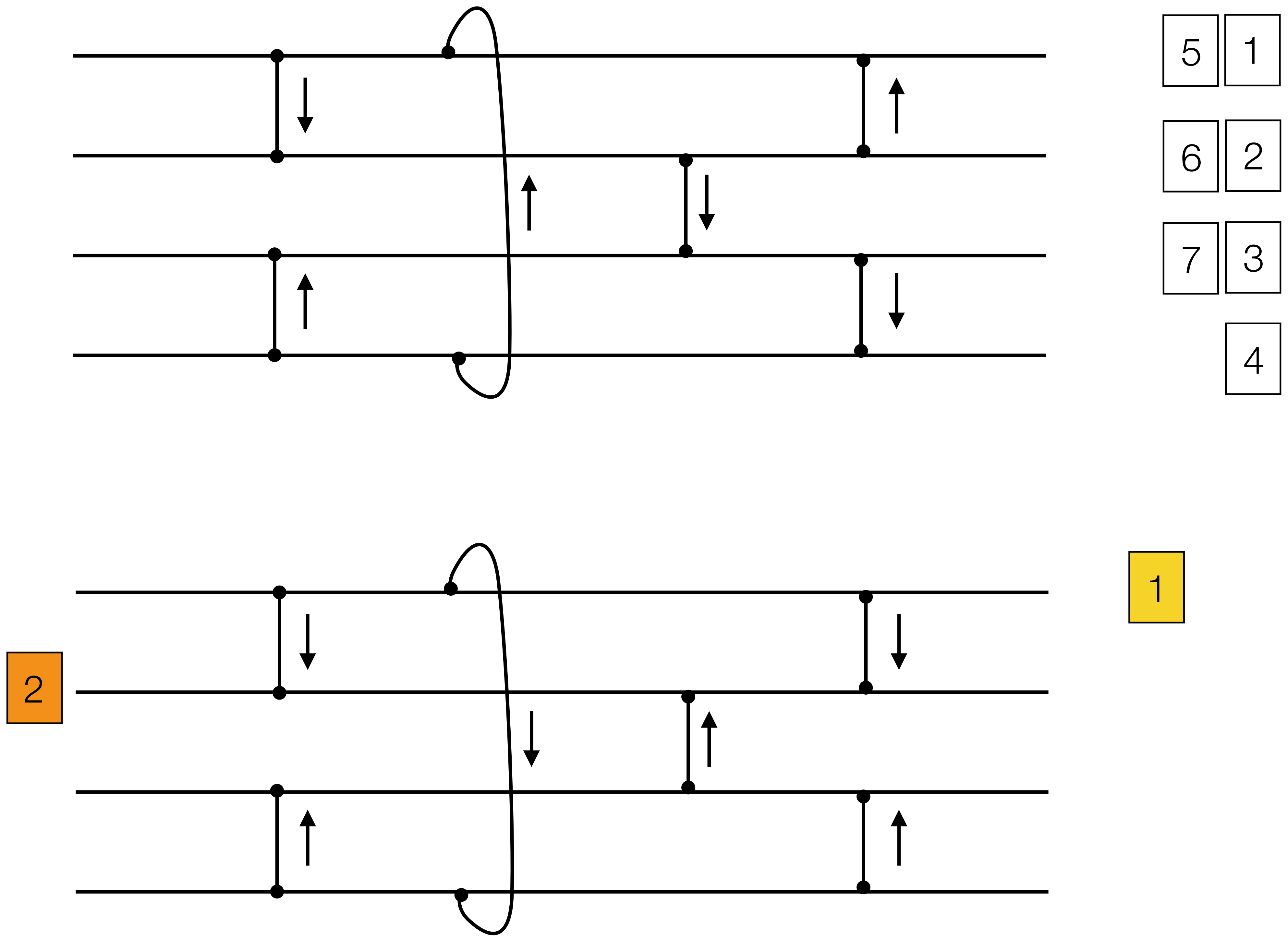


Quiescent Consistency: Counting Networks

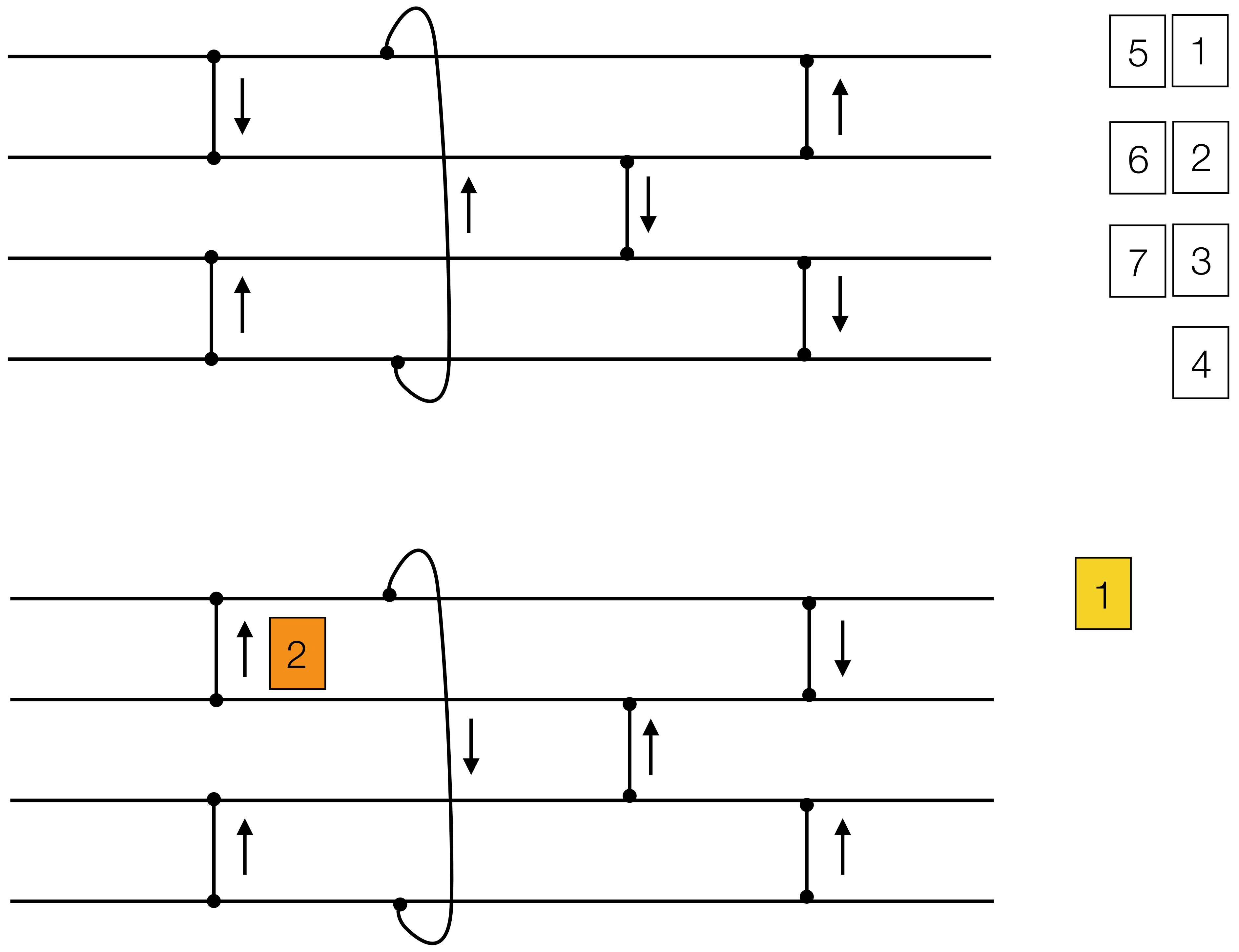




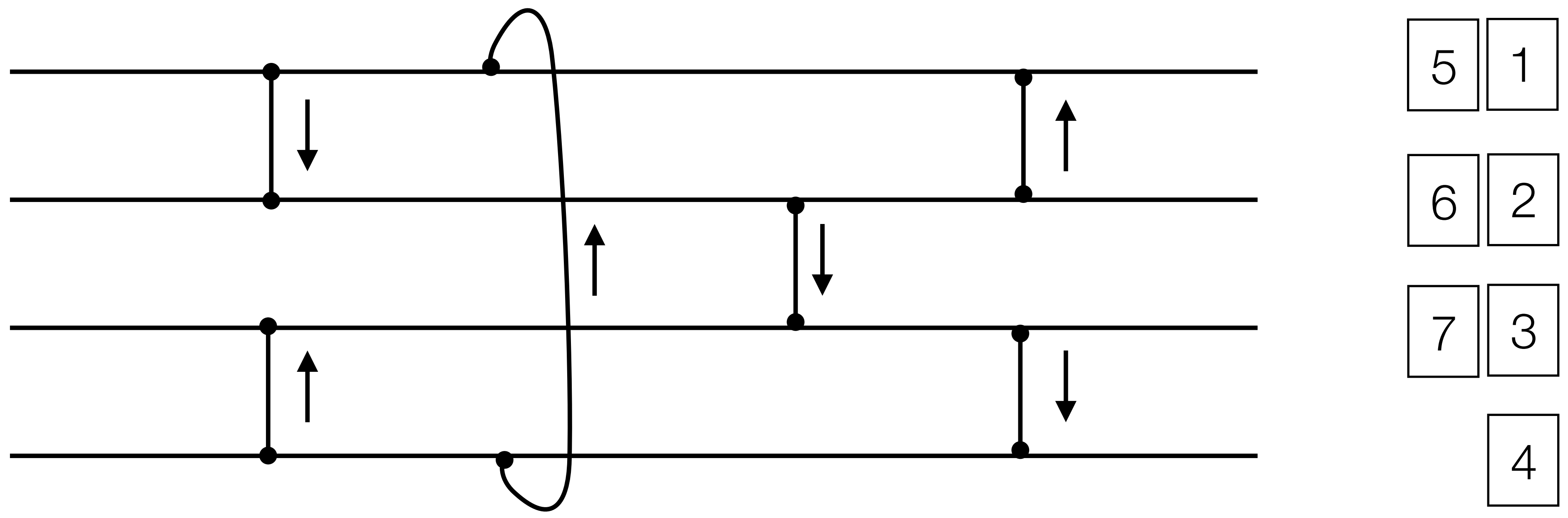
Quiescent Consistency: Counting Networks

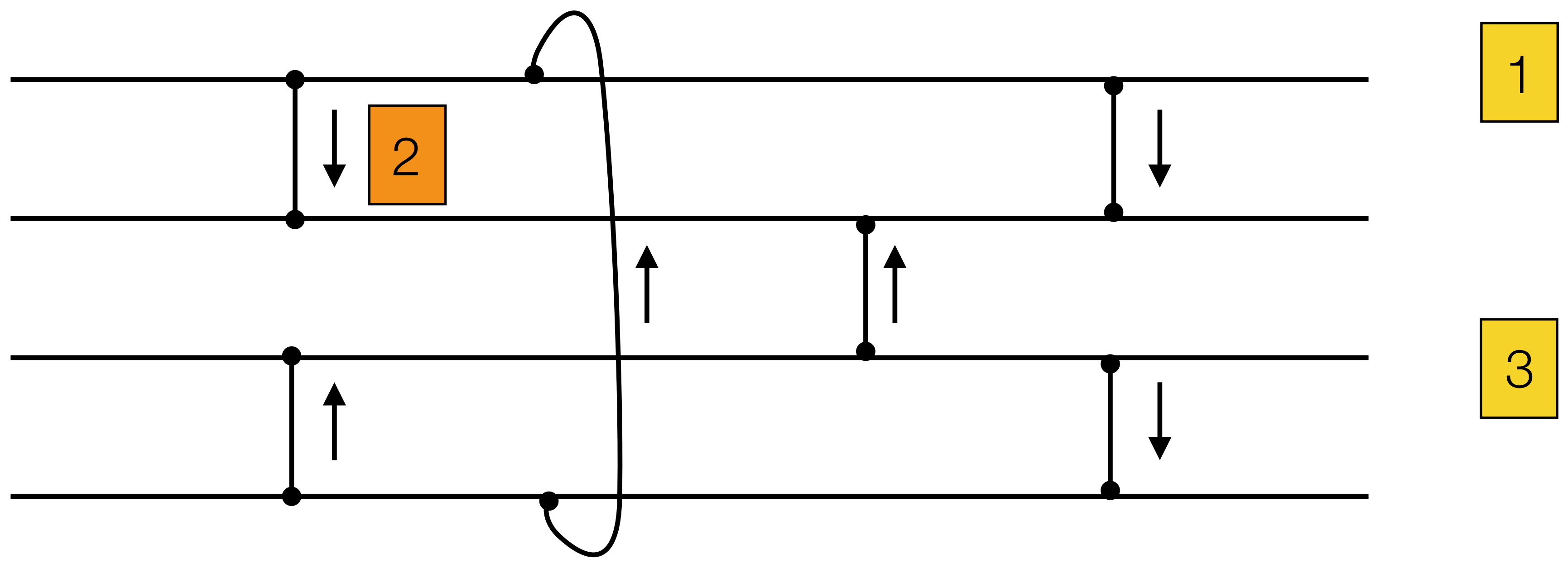


Quiescent Consistency: Counting Networks

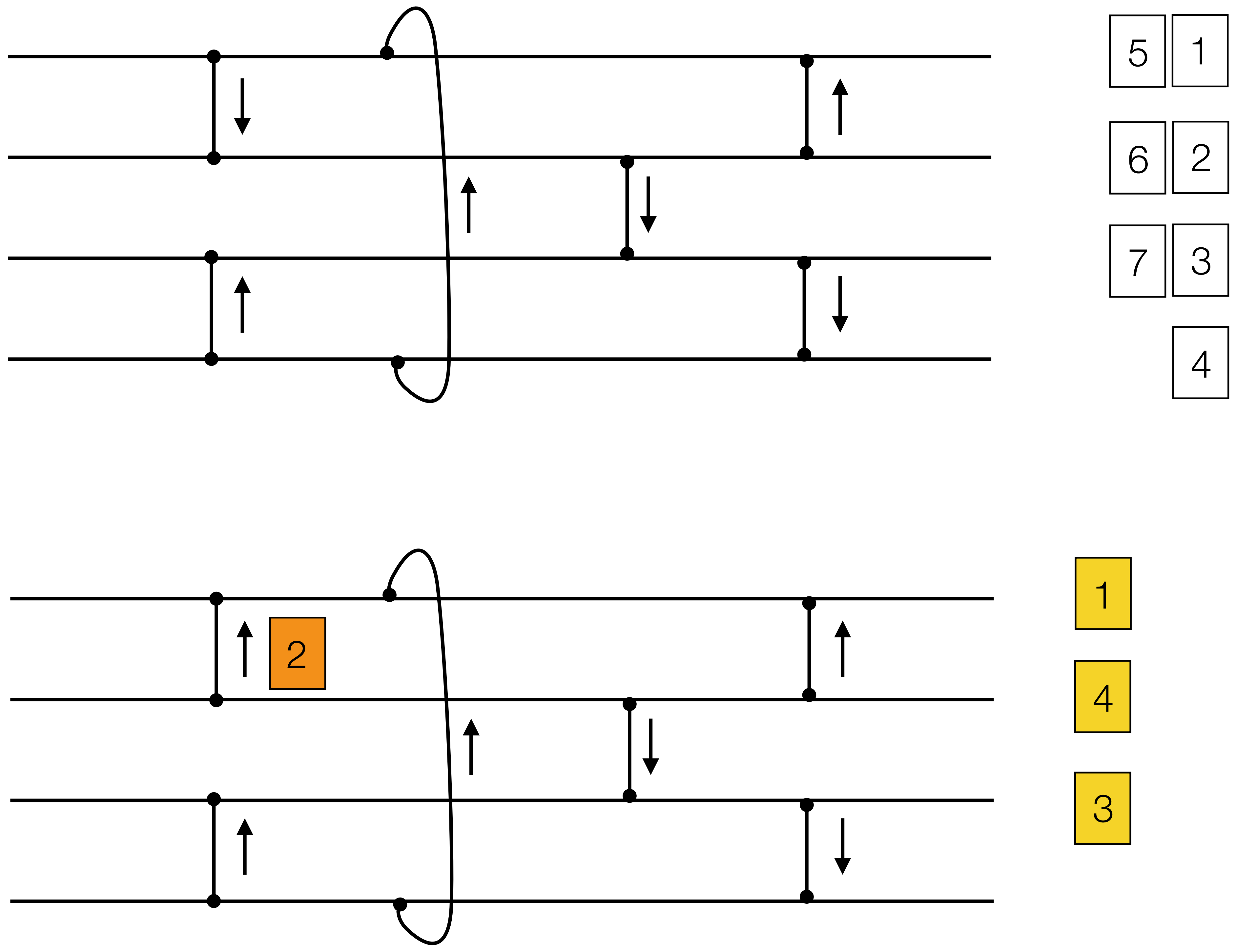


Quiescent Consistency: Counting Networks





Quiescent Consistency: Counting Networks



Quiescent Consistency

- Quiescence Consistency tells us nothing in the case where there are no Quiescent prefixes
- But guarantees sequential correctness if there is a single thread

To be continued …