Concepts of Concurrent Computation
Spring 2015
Lecture 8: Correctness Conditions

Sebastian Nanz
Chris Poskitt

Chair of Software Engineering
ETH zürich

Today's agenda

1. concurrent objects and correctness

2. quiescent and sequential consistency

3. linearizability

Terminology: concurrent objects

- a concurrent object is a data object shared by concurrent processes

  => *has a type defining possible values, and primitive methods that provide the only means of creation/manipulation*
  => *e.g. a shared data structure, a shared message queue, ...*

- a concurrent system is a collection of sequential processes that communicate through concurrent objects

Specifying correctness of operations

- in a sequential system, it is easy to specify the behaviour of methods

  => *pre- and postconditions*

$$\{pre\}\ q.op\ \{post\}$$

  => *methods cannot be called on objects that are in an "intermediate state"*

- in a concurrent system, need to accommodate interleavings of method invocations

What does it mean for concurrent objects to be correct?

- typically boils down to some notion of equivalence with sequential behaviour

- consider a simple, lock-based concurrent FIFO queue

What does it mean for concurrent objects to be correct?

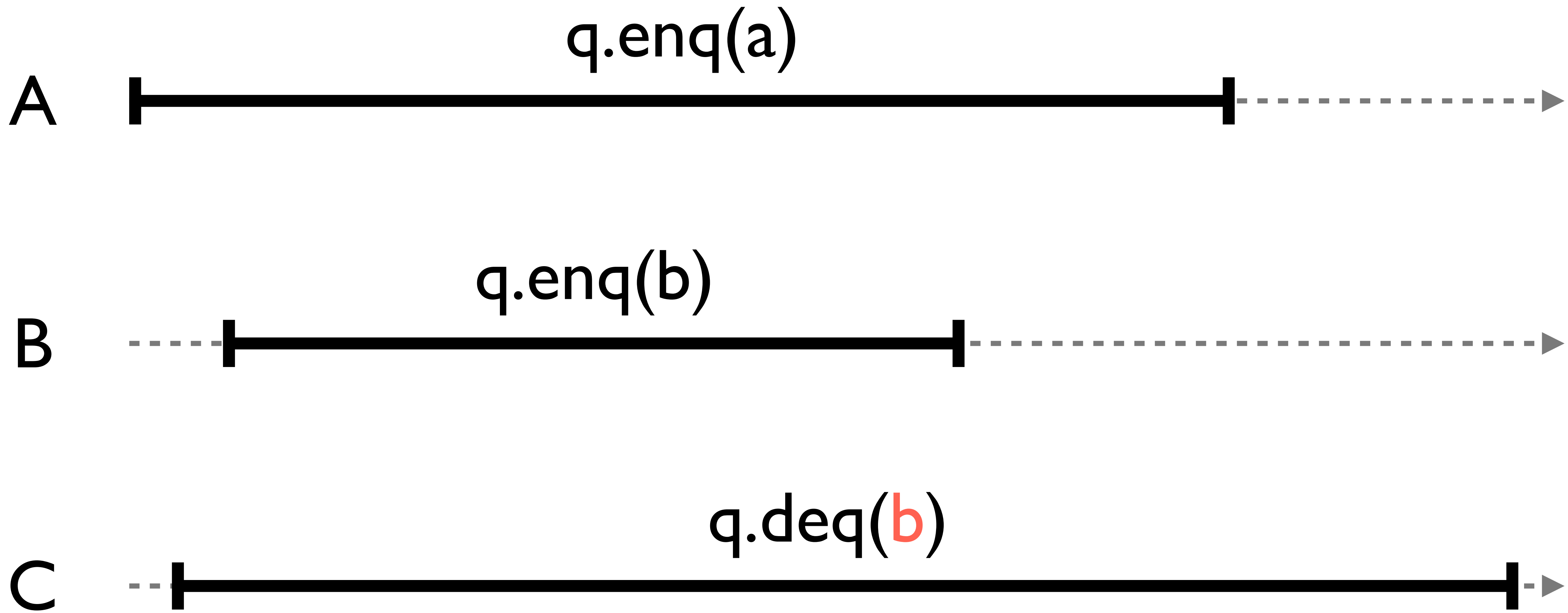
- typically boils down to some notion of equivalence with sequential behaviour

- consider a simple, lock-based concurrent FIFO queue

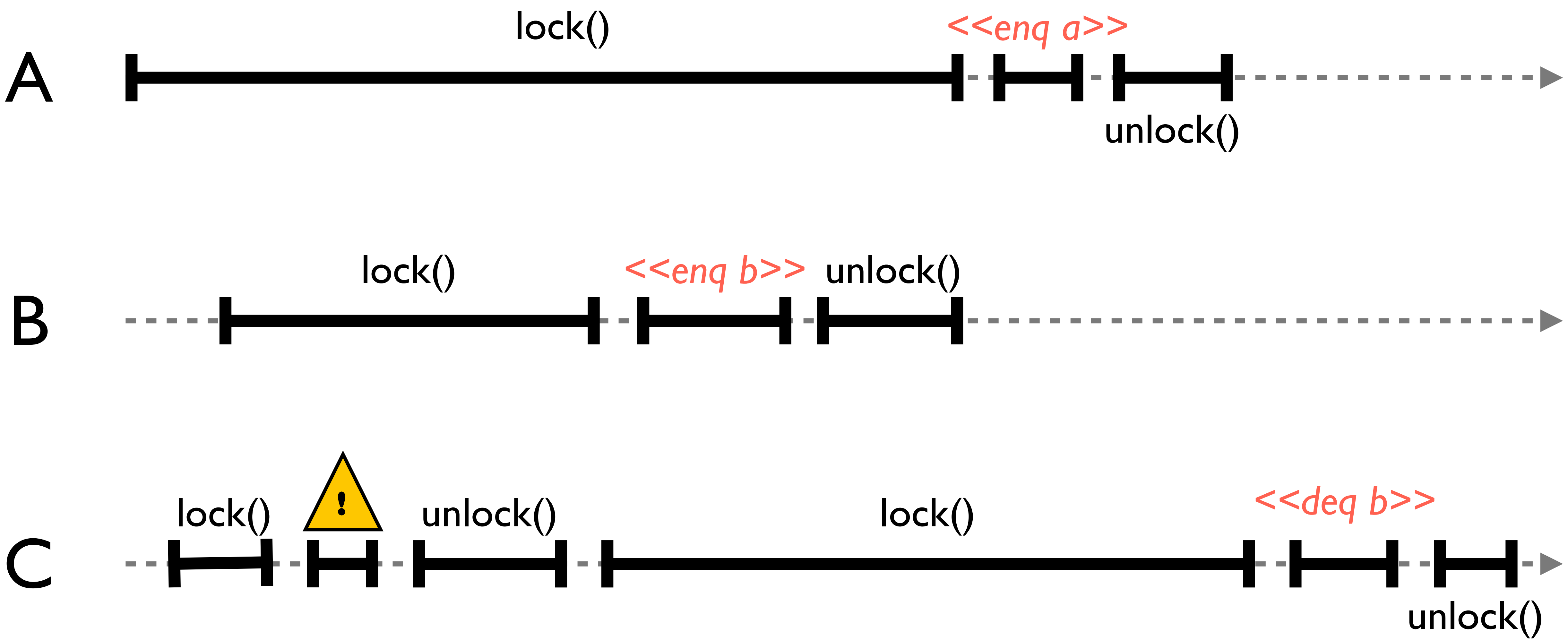
```
public void enq(int x) throws
FullException {
  lock.lock();
  try {
    if <<queue full>>
      { throw new FullException(); }
    <<enqueue x>>
    } finally {
      lock.unlock();
}}
```

```
public int deq() throws
EmptyException {
  lock.lock();
  try {
    if <<queue empty>>
      { throw new EmptyException(); }
    <<dequeue x>>
    } finally {
      lock.unlock();
}}
```

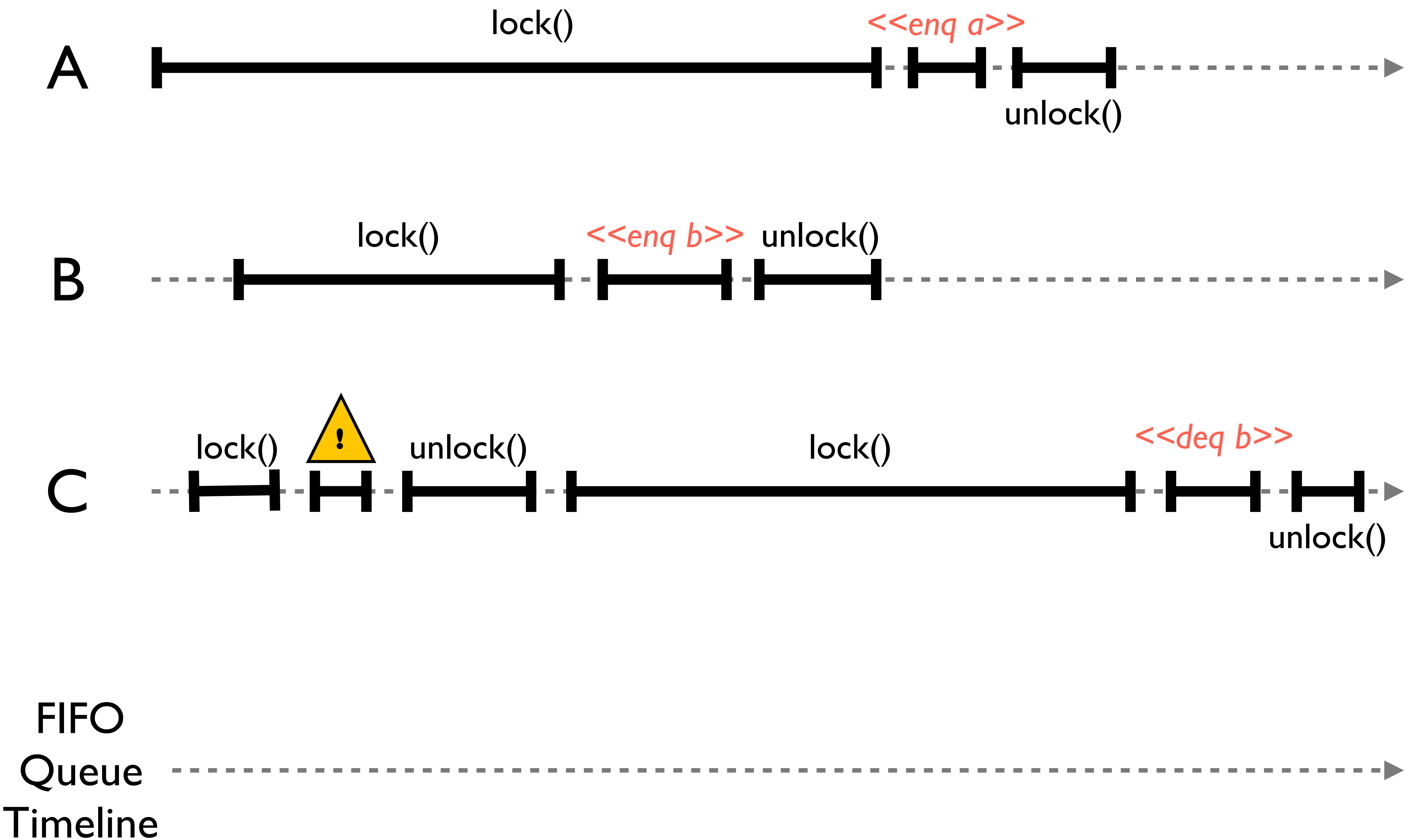
A lock-based concurrent FIFO queue



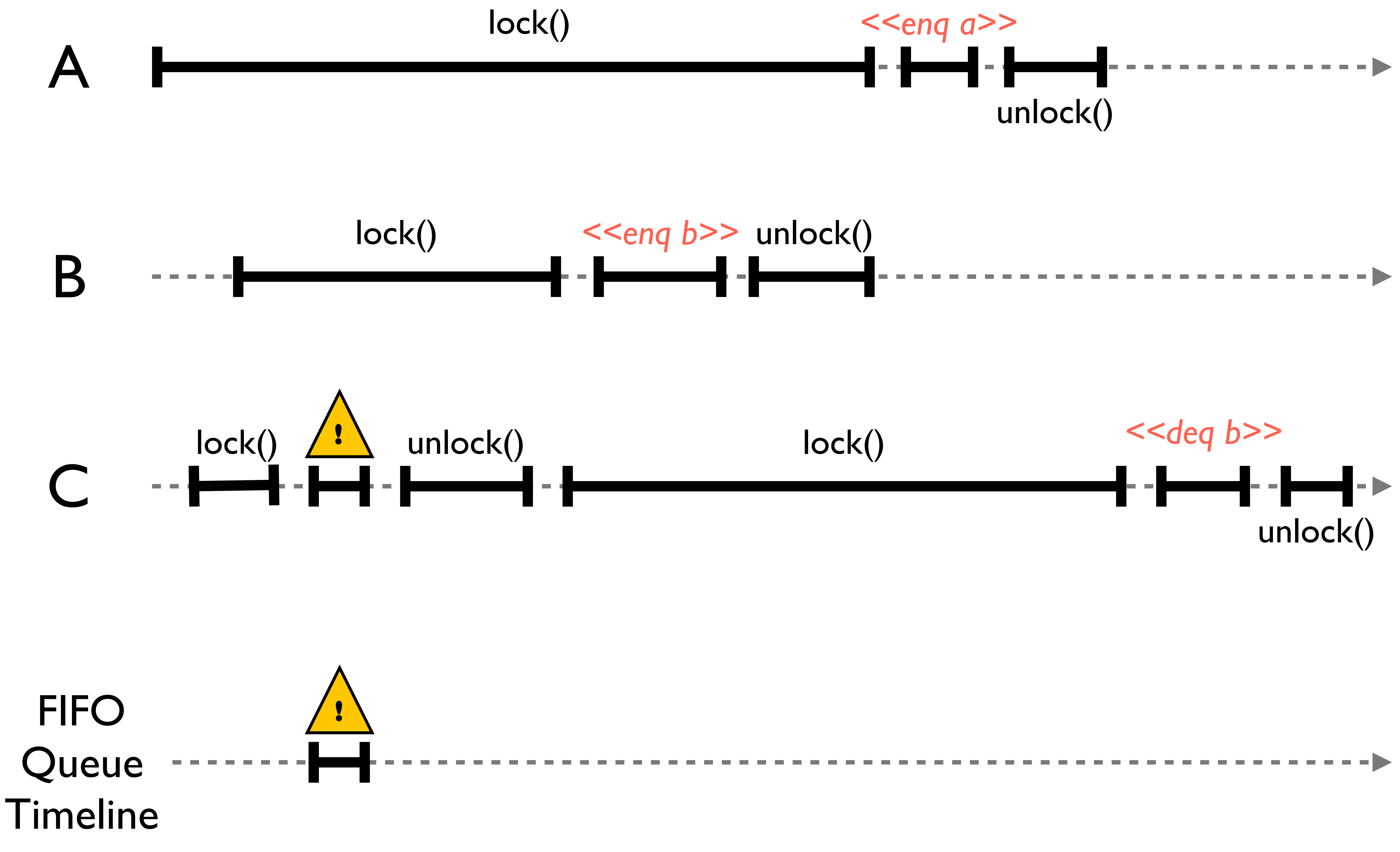
A lock-based concurrent FIFO queue



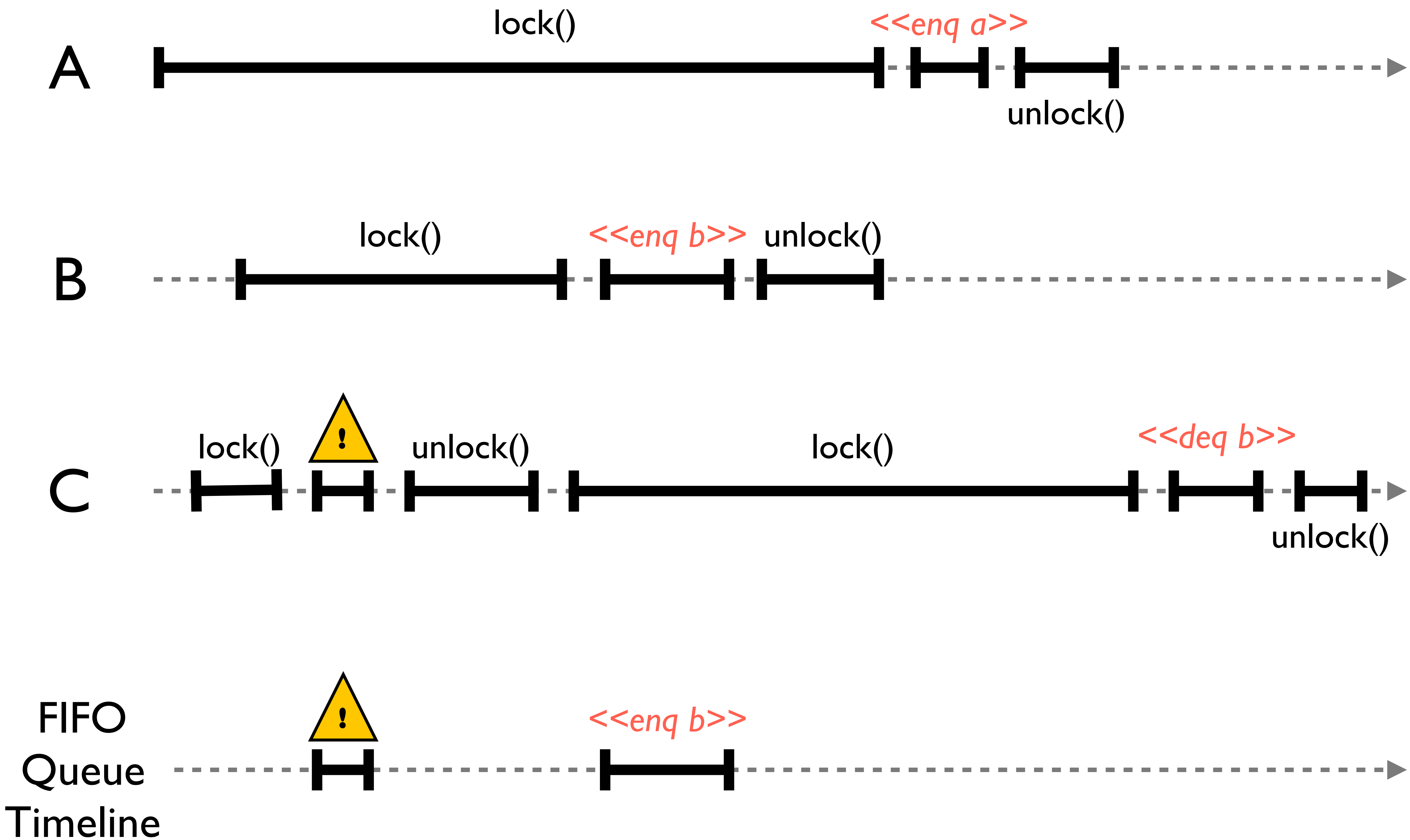
A lock-based concurrent FIFO queue



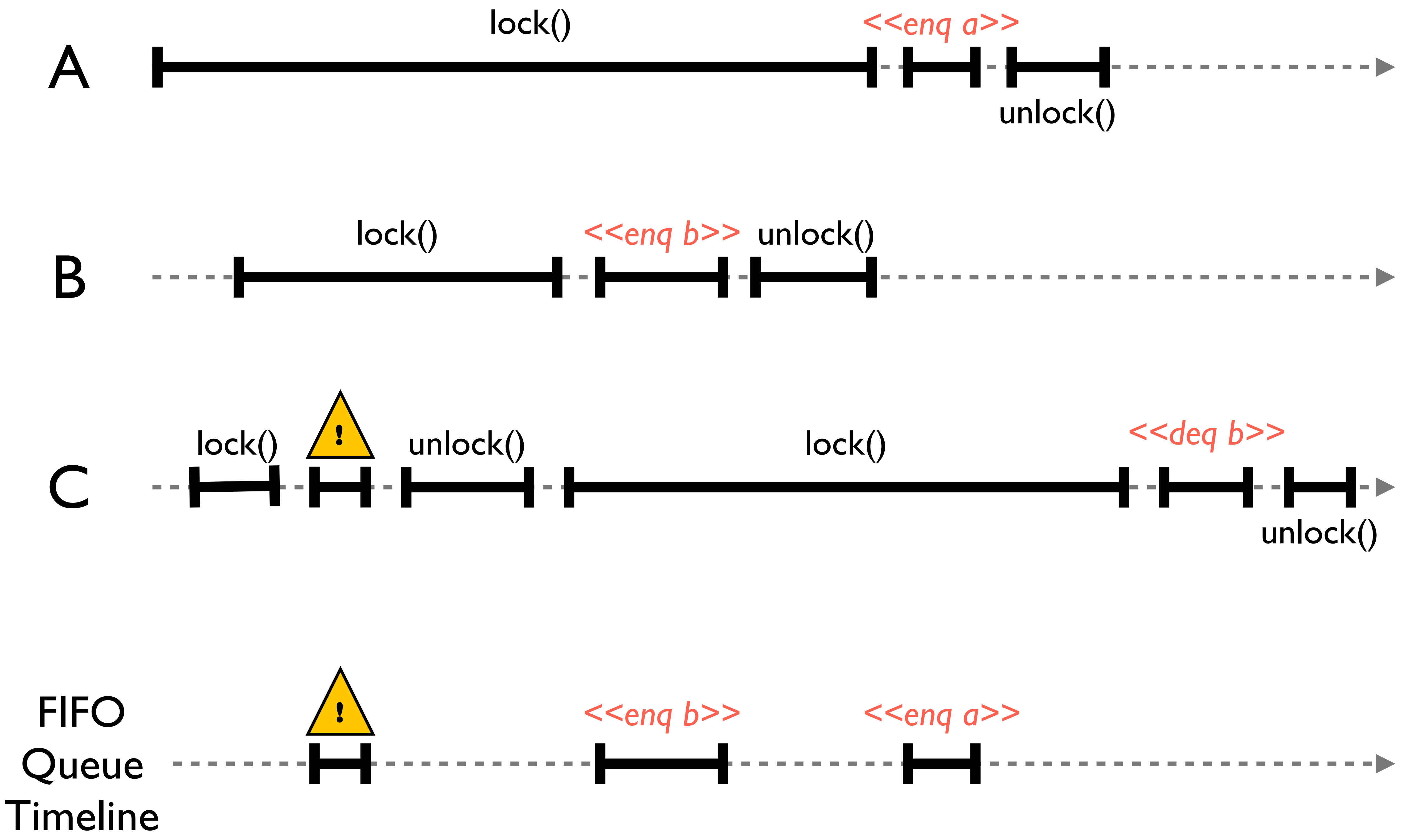
A lock-based concurrent FIFO queue



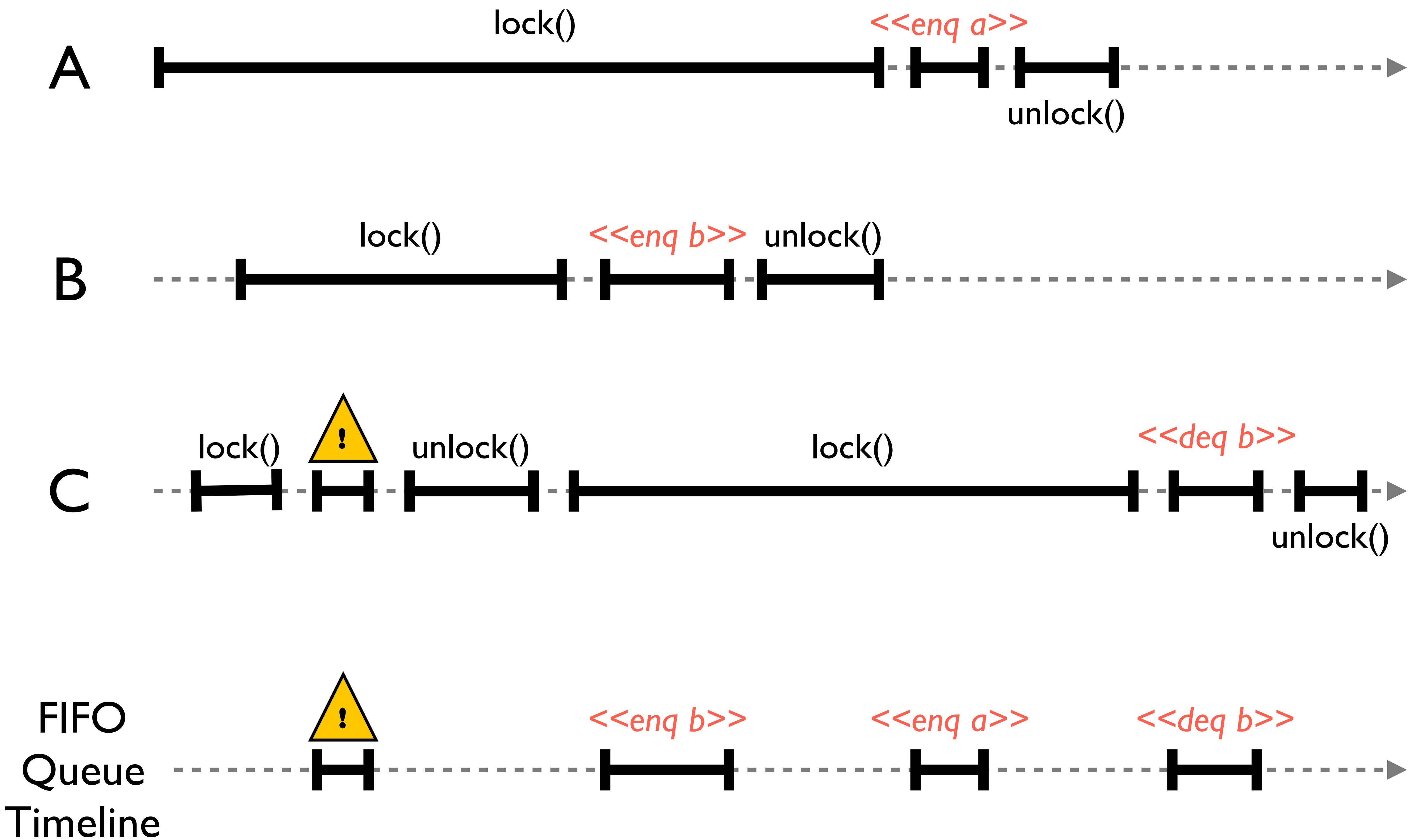
A lock-based concurrent FIFO queue



A lock-based concurrent FIFO queue



A lock-based concurrent FIFO queue



What happens if we drop the locks?

```
public void enq(int x) throws
FullException {
  lock.lock();
  try {
    if <<queue full>>
      { throw new FullException(); }
    <<enqueue x>>
    } finally {
      lock.unlock();
}}
```

```
public int deq() throws
EmptyException {
  lock.lock();
  try {
    if <<queue empty>>
      { throw new EmptyException(); }
    <<dequeue x>>
    } finally {
      lock.unlock();
}}
```

What happens if we drop the locks?

```
public void enq(int x) throws
FullException {
  lock.lock();
  try {
    if <<queue full>>
      { throw new FullException(); }
    <<enqueue x>>
    } finally {
      lock.unlock();
}}
```

```
public int deq() throws
EmptyException {
  lock.lock();
  try {
    if <<queue empty>>
      { throw new EmptyException(); }
    <<dequeue x>>
    } finally {
      lock.unlock();
}}
```

are there circumstances in which this queue can be correct?

what does "correct" mean?

Reasoning about concurrent objects: a principle

- concurrent objects may have methods with finer-grained locking or no locking at all

- need to be able to specify and reason about implementations without relying on method-level locking

- but the example illustrates an important principle:

> *it's easier to reason about concurrent objects if we can map their concurrent executions to sequential ones*

Which "equivalences" with sequential behaviour do we care about?

- do we care about program order, fairness, ...?

- in practice, different applications require different "strengths" of correctness conditions

  => *print job queue for a lightly loaded printer*
  => *banking server (e.g. transfer money from savings; withdraw £50)*
  => *stock-trading server*

We will consider three correctness conditions

strength of condition

quiescent consistency

=> *whenever an object becomes quiescent, then the execution so far is equivalent to some sequential execution of the completed calls*

sequential consistency

=> *method calls should appear to take effect in a sequential order consistent with the program order*

linearizability

=> *each method call should appear to take effect instantaneously at some moment between its invocation and response*

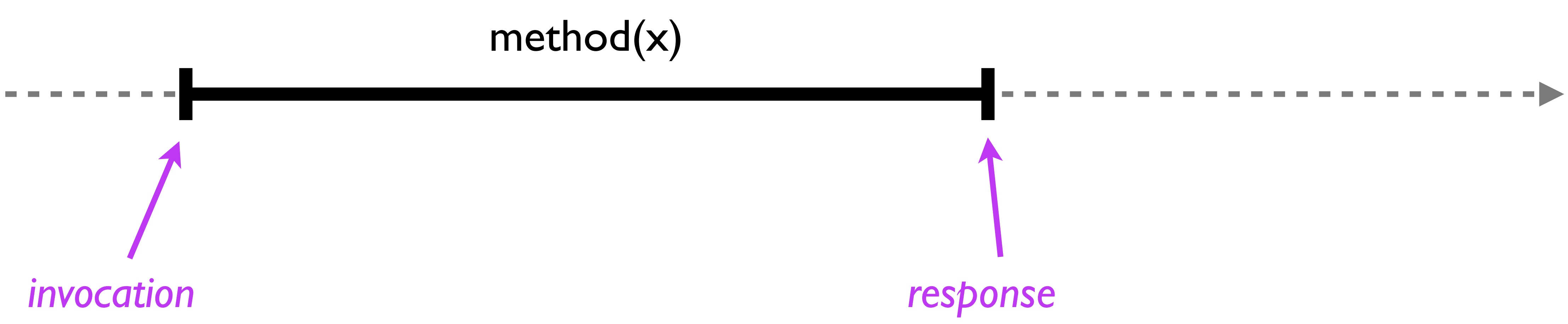
Next on the agenda

1. concurrent objects and correctness ✓
2. quiescent and sequential consistency
3. linearizability

Terminology: method calls

- individual threads sequentially execute method calls that have invocation and response events

- a method is pending if its call has occurred, but not its response

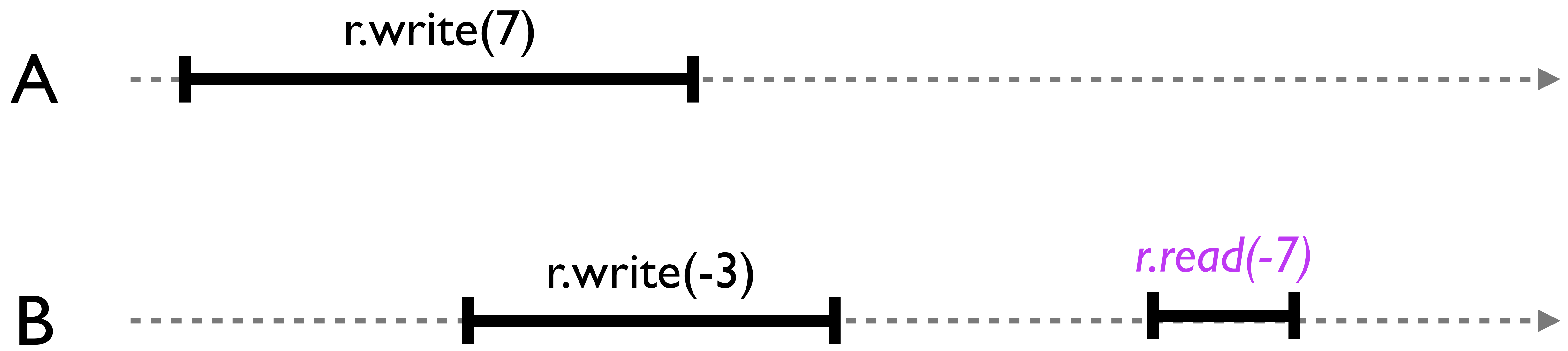


What conditions / restrictions do we need?

- let's derive some principles from examples of unacceptable behaviours

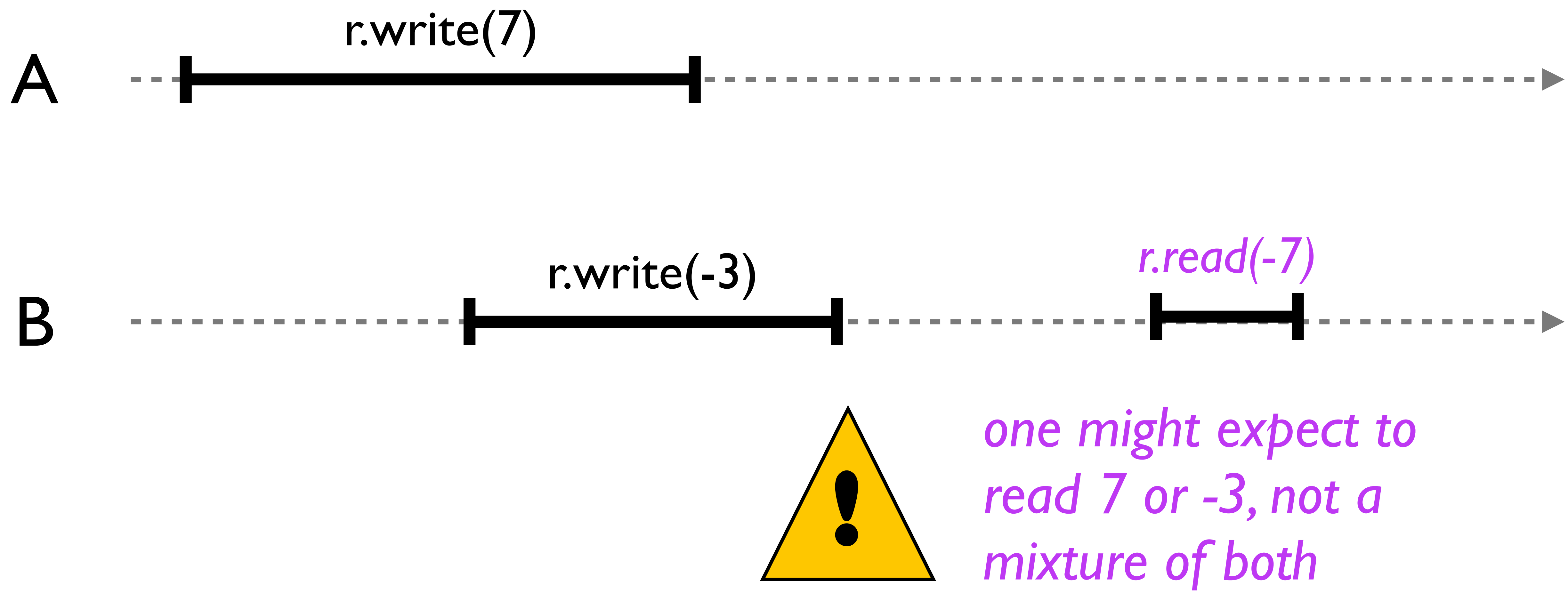
What conditions / restrictions do we need?

- let's derive some principles from examples of unacceptable behaviours



What conditions / restrictions do we need?

- let's derive some principles from examples of unacceptable behaviours

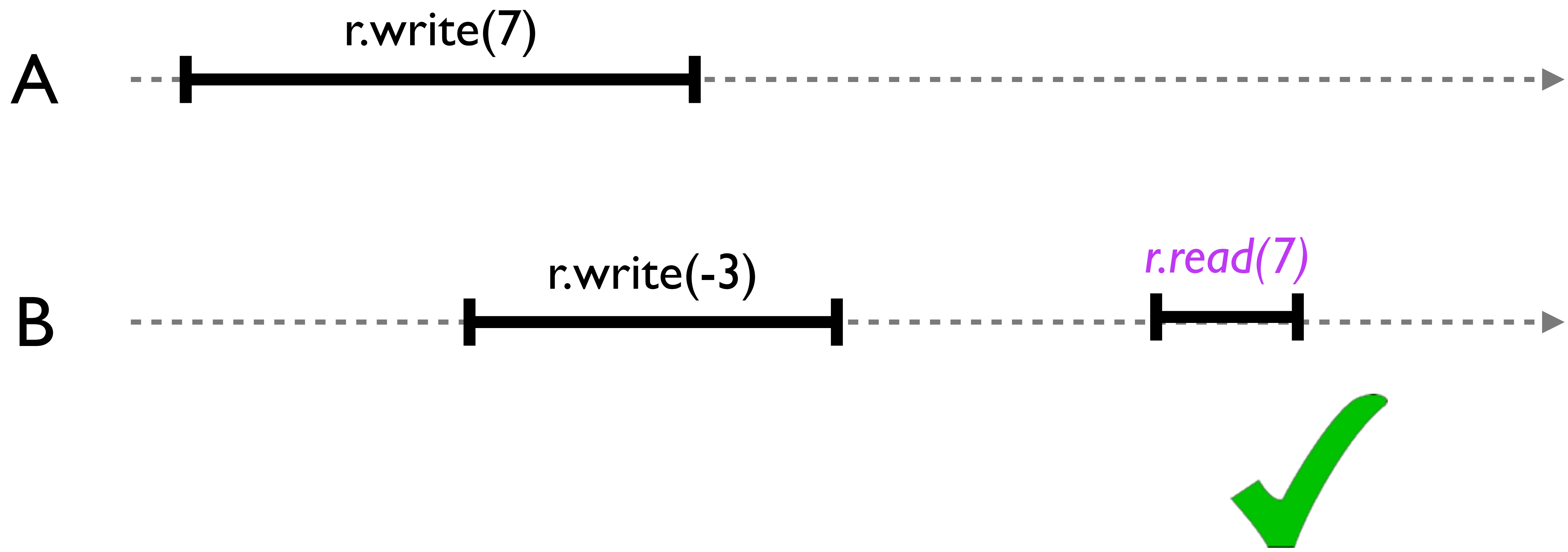


A principle

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

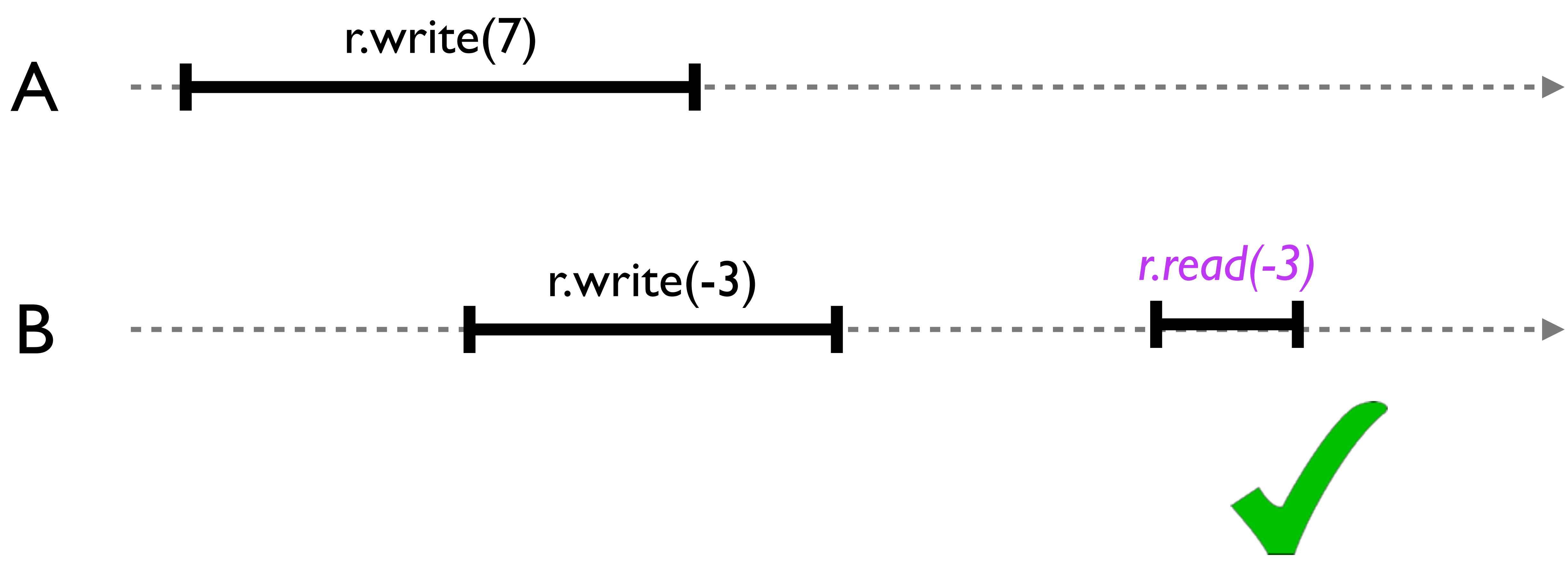
A principle

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



A principle

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



A principle

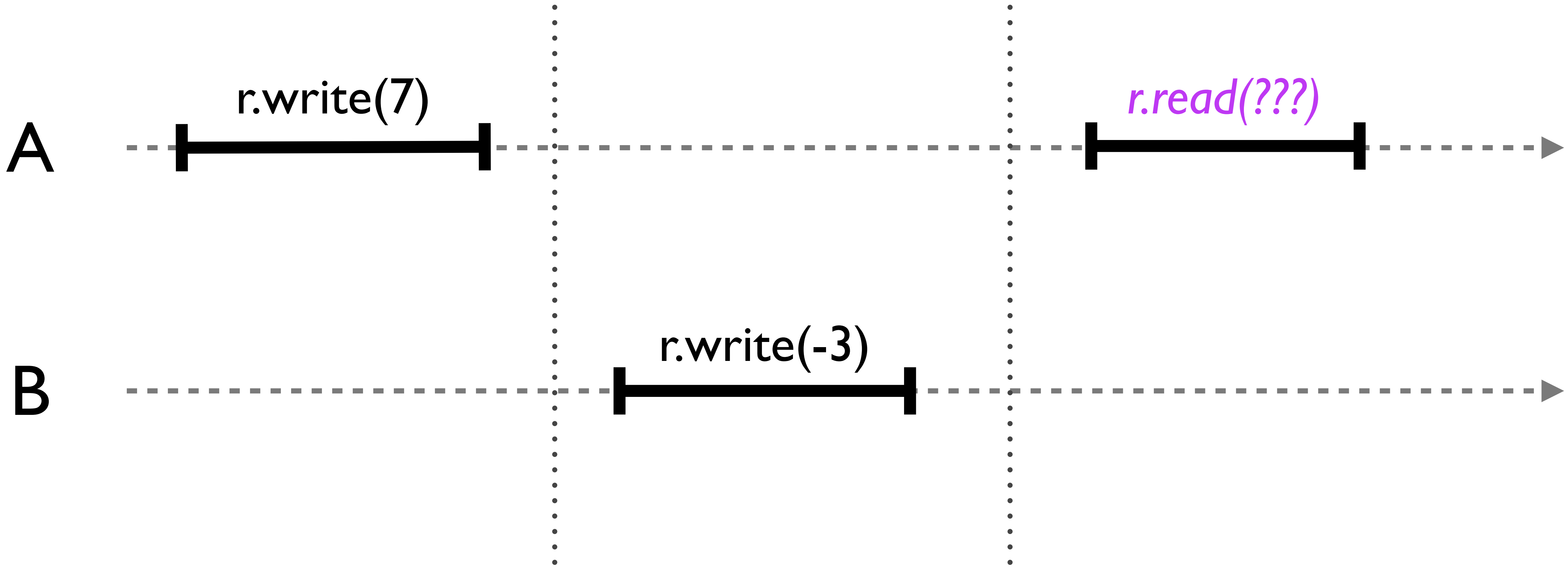
method calls should appear to happen in a one-at-a-time, sequential order

too weak alone!

permits, e.g. readers to always return the object's initial state

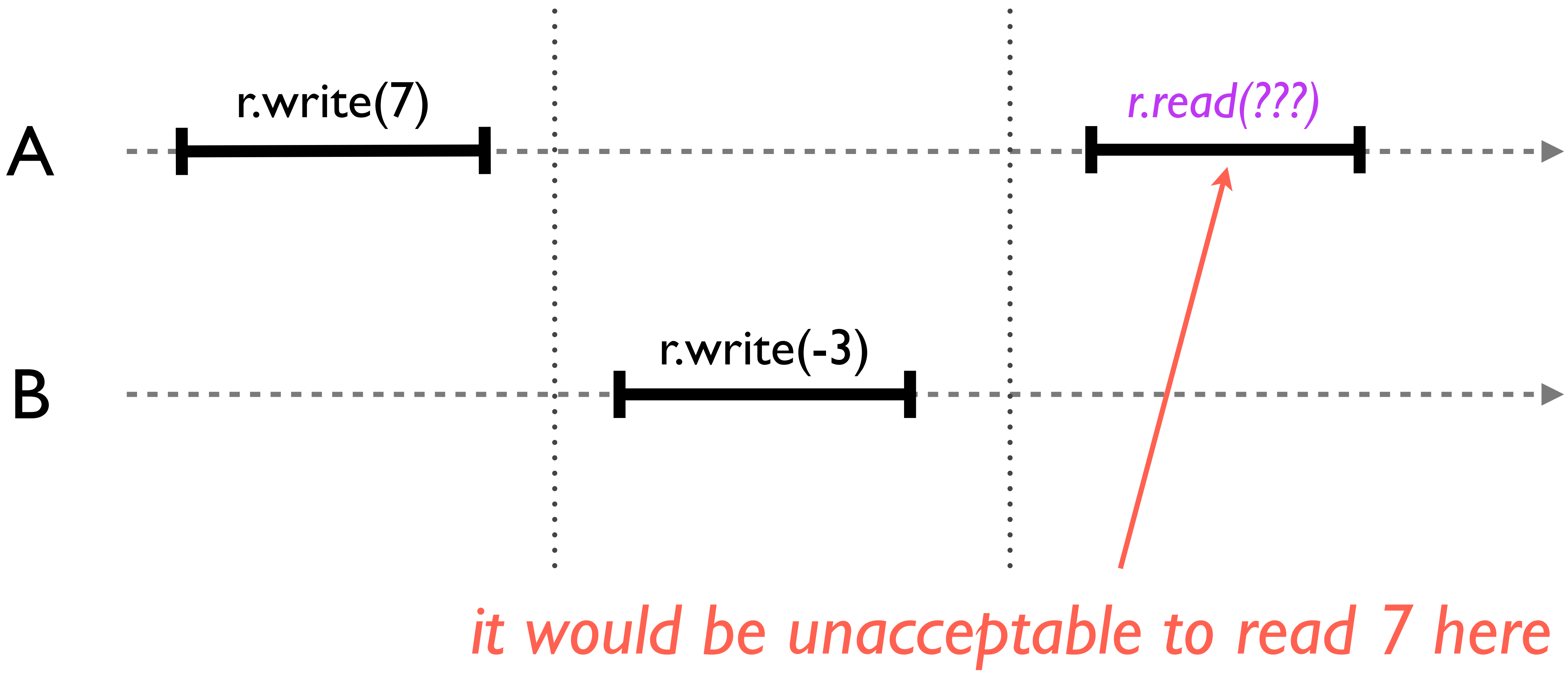
A principle

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



A principle

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



Quiescent consistency

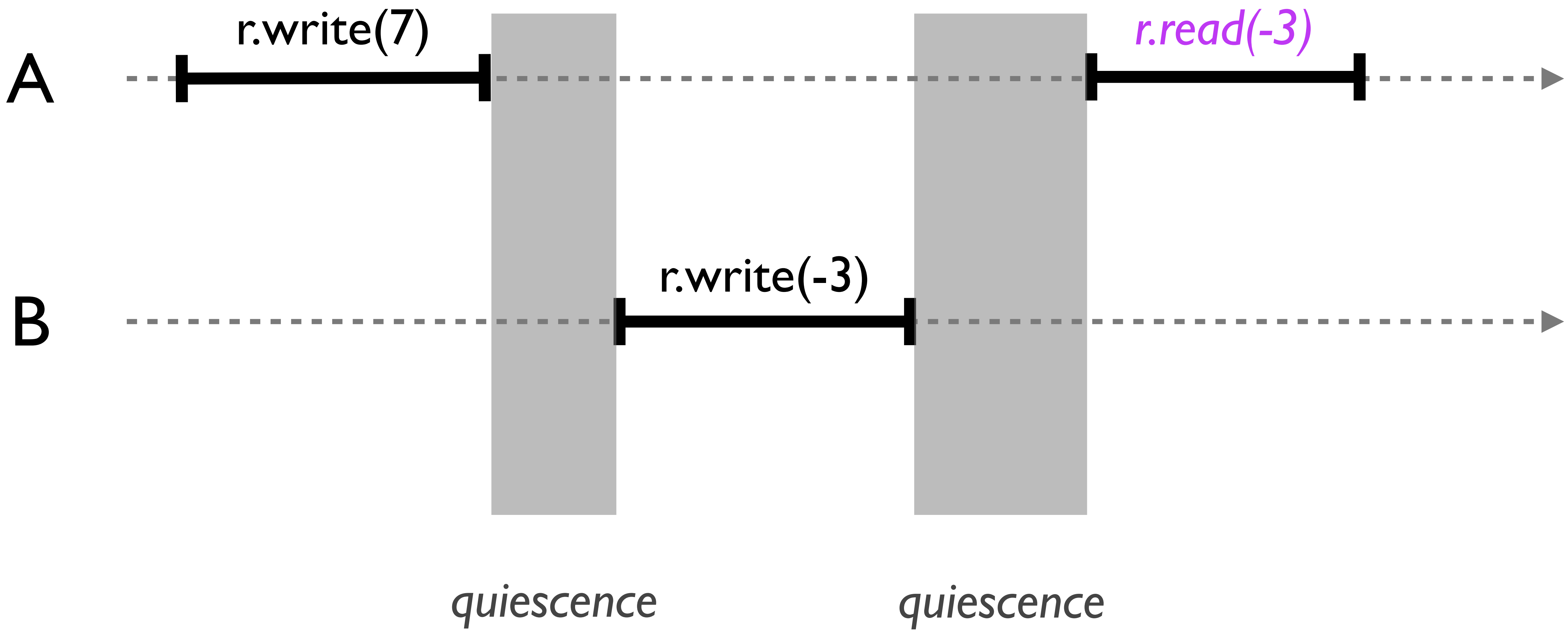
method calls should appear to happen in a one-at-a-time, sequential order

+

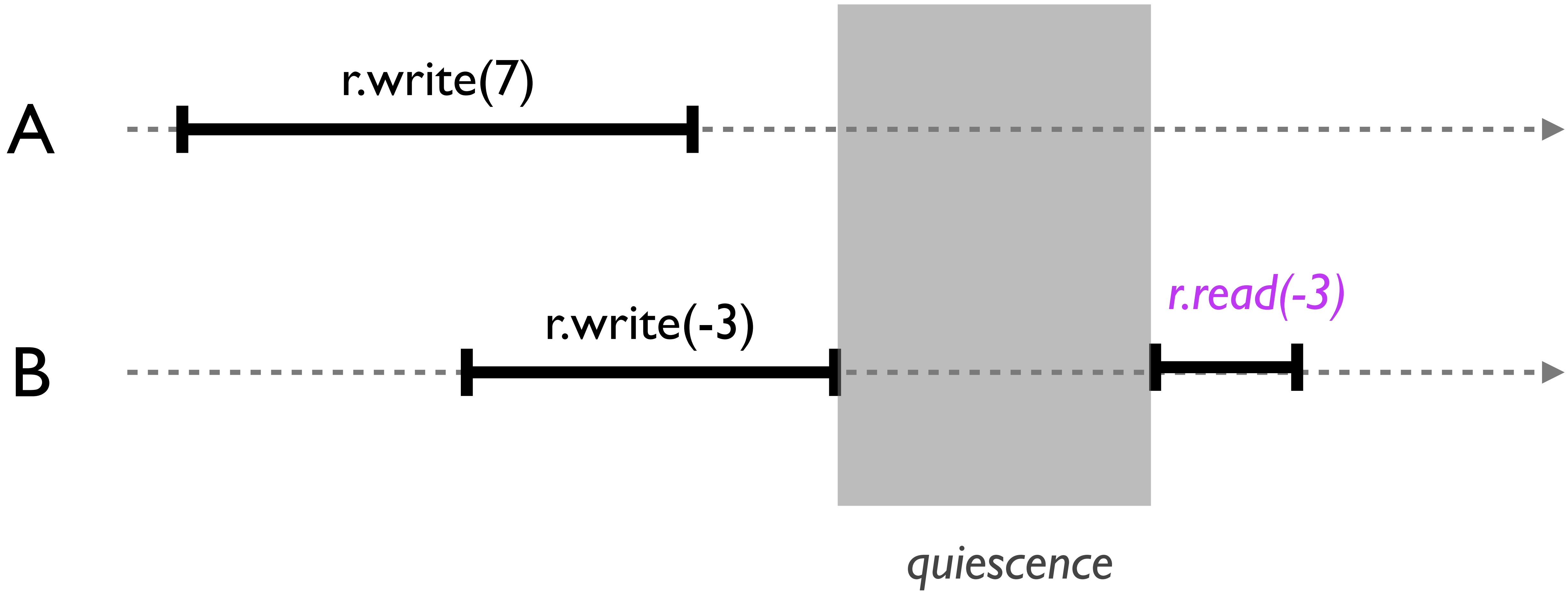
method calls separated by a period of quiescence should appear to take effect in their real-time order

NB: an object is quiescent if it has no pending method calls

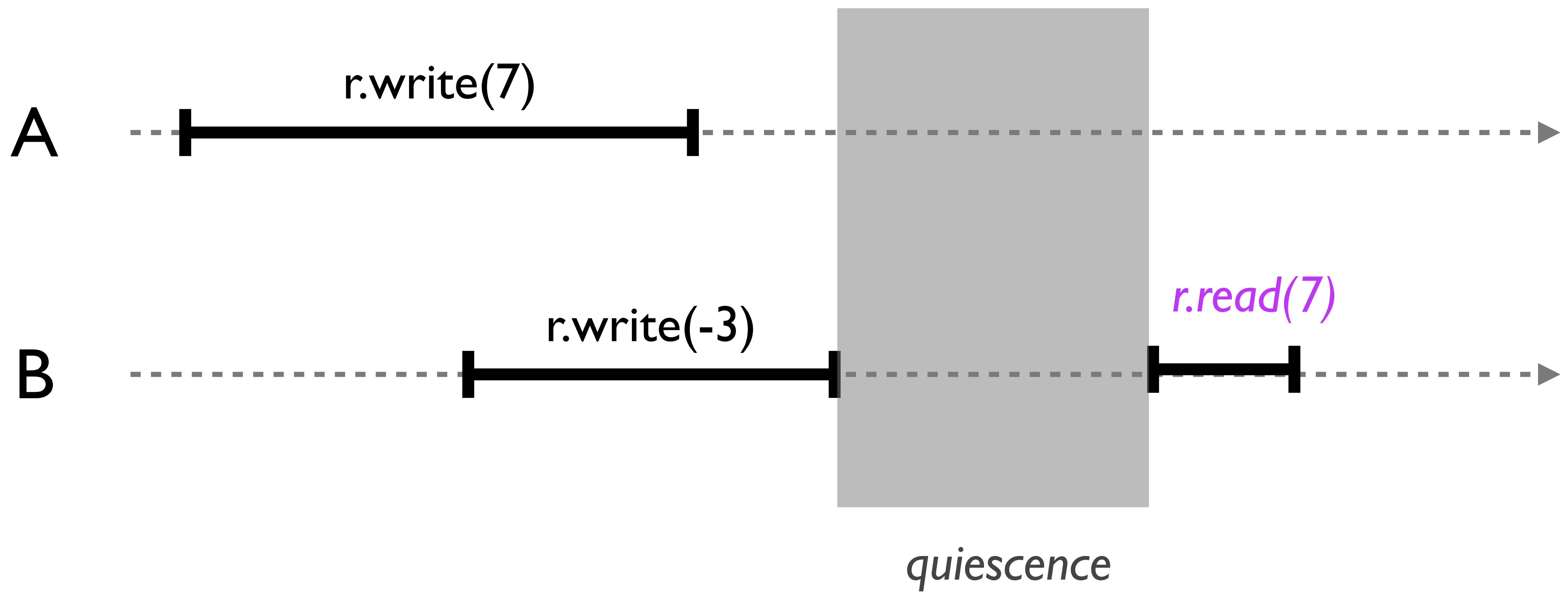
Quiescent consistency



Quiescent consistency



Quiescent consistency

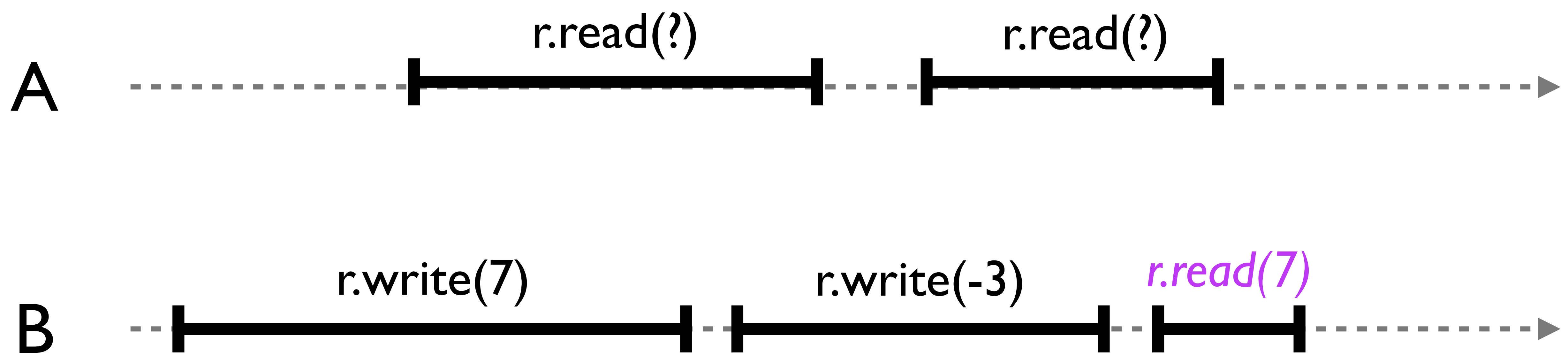


What if program order matters?

- should this behaviour be allowed? *(example by Huisman)*

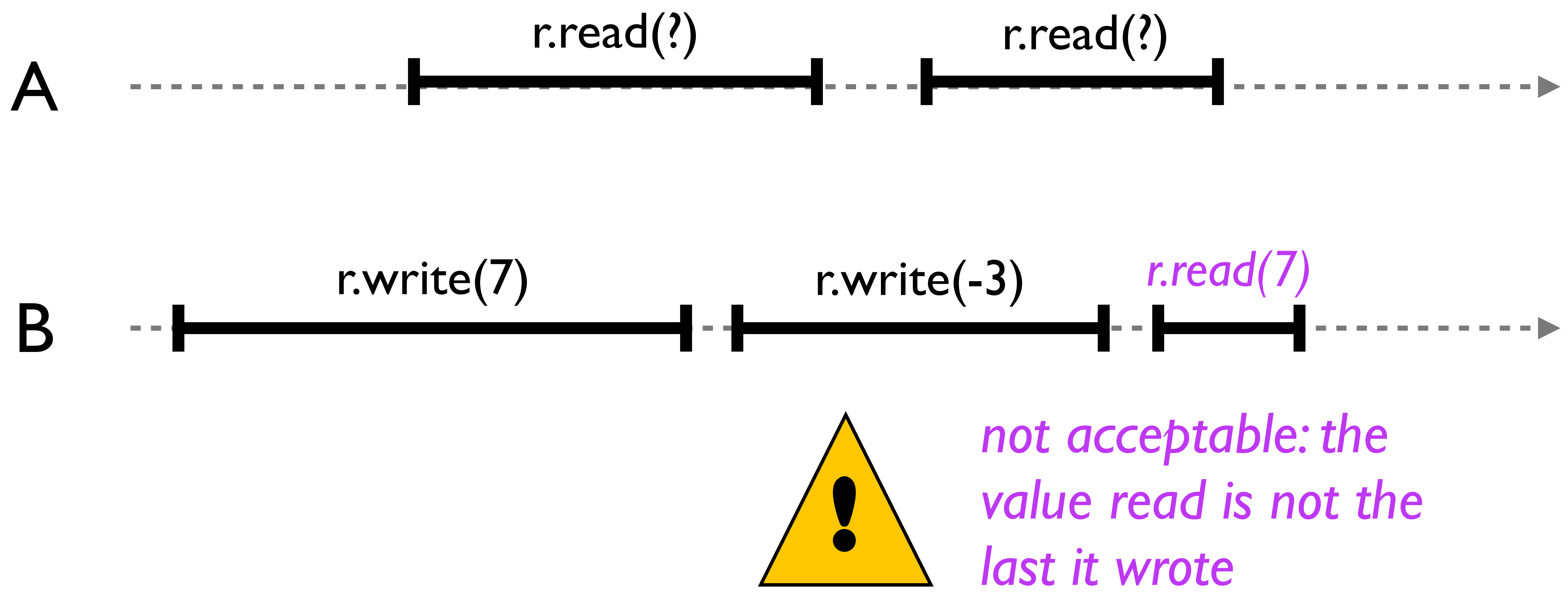
What if program order matters?

- should this behaviour be allowed? *(example by Huisman)*



What if program order matters?

- should this behaviour be allowed? *(example by Huisman)*



Sequential consistency

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

+

method calls should appear to take effect in <u>program order</u>

Sequential consistency

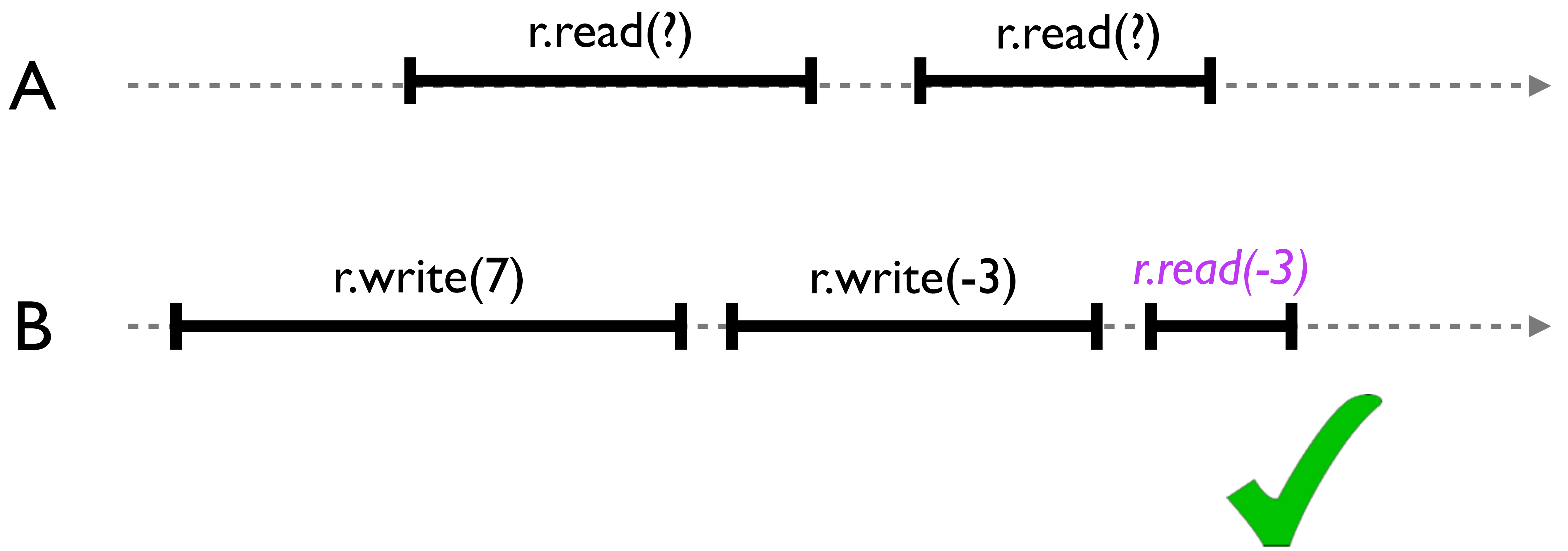
method calls should appear to happen in a one-at-a-time, sequential order

+

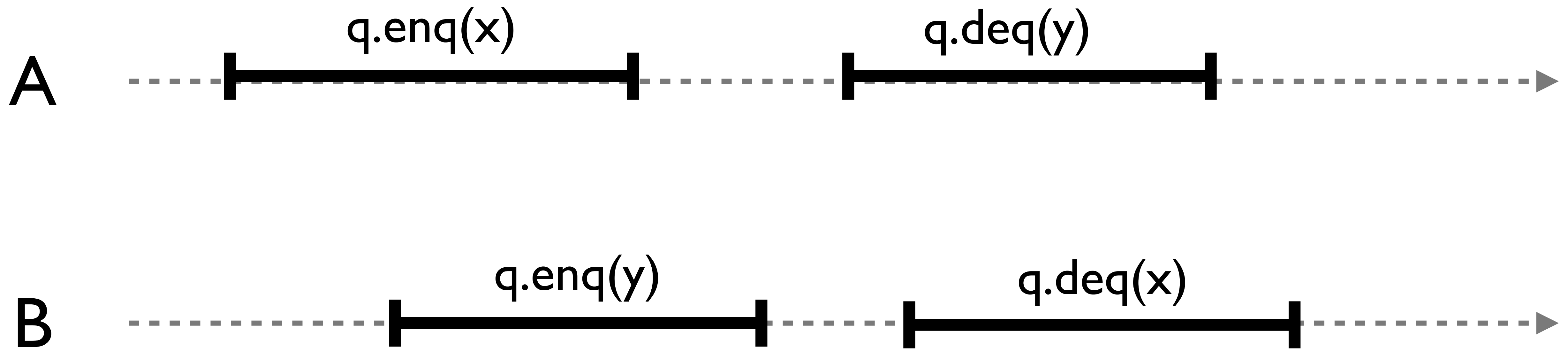
method calls should appear to take effect in <u>program order</u>

i.e. in any concurrent execution, there is a way to order the method calls sequentially so that they are (1) consistent with program order; and (2) meet the object's sequential specification

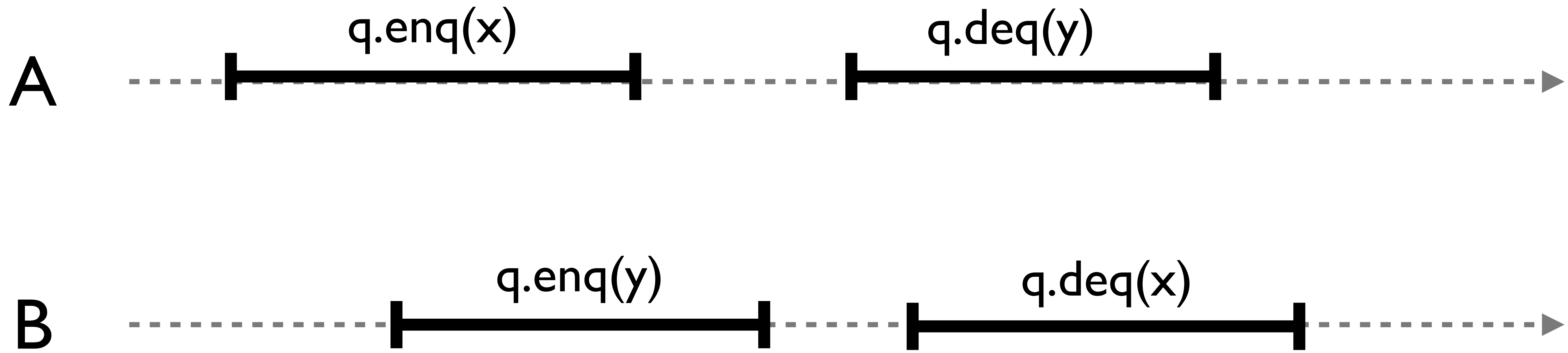
Sequential consistency



Is this execution sequentially consistent?

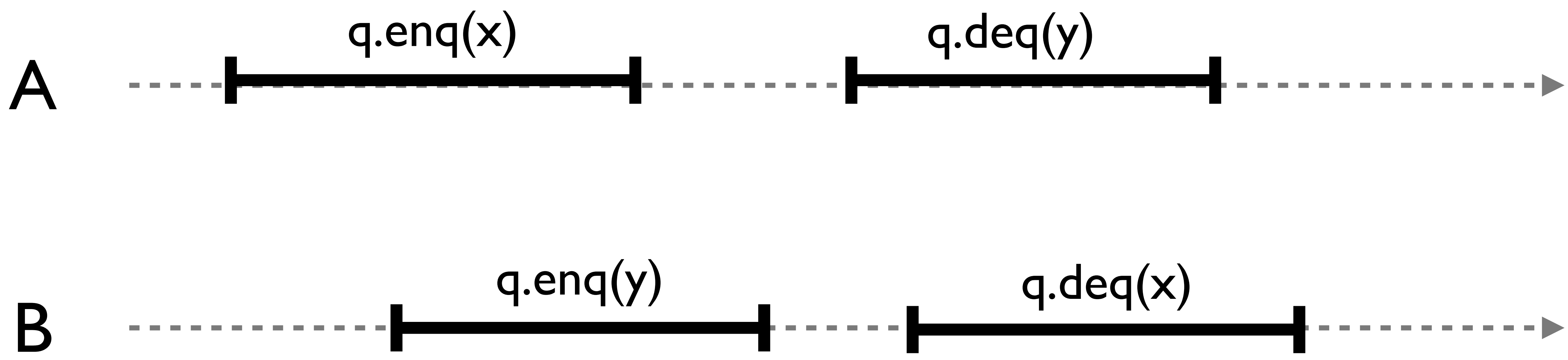


Is this execution sequentially consistent?



yes! two possible consistent orderings

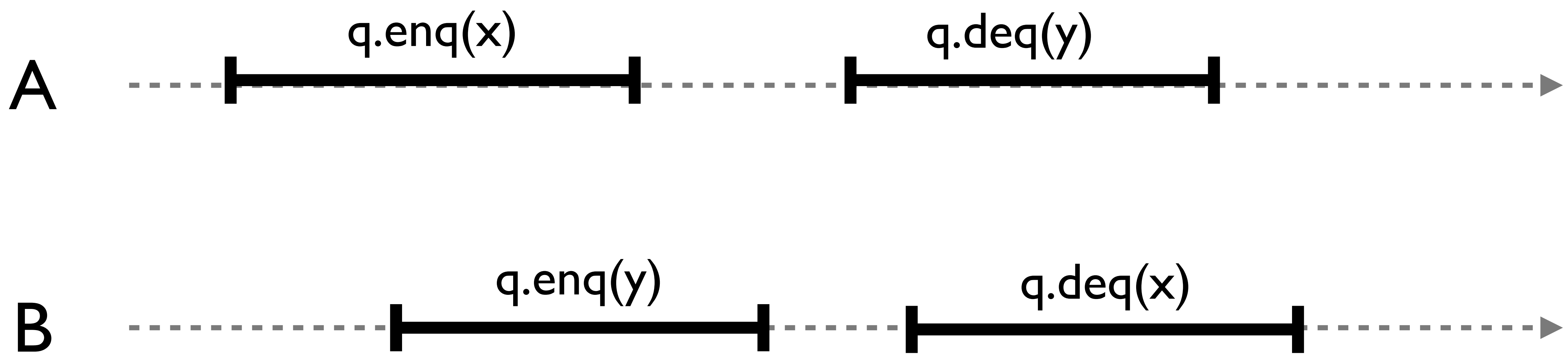
Is this execution sequentially consistent?



yes! two possible consistent orderings

A: enqueues x
B: enqueues y
B: dequeues x
A: dequeues y

Is this execution sequentially consistent?

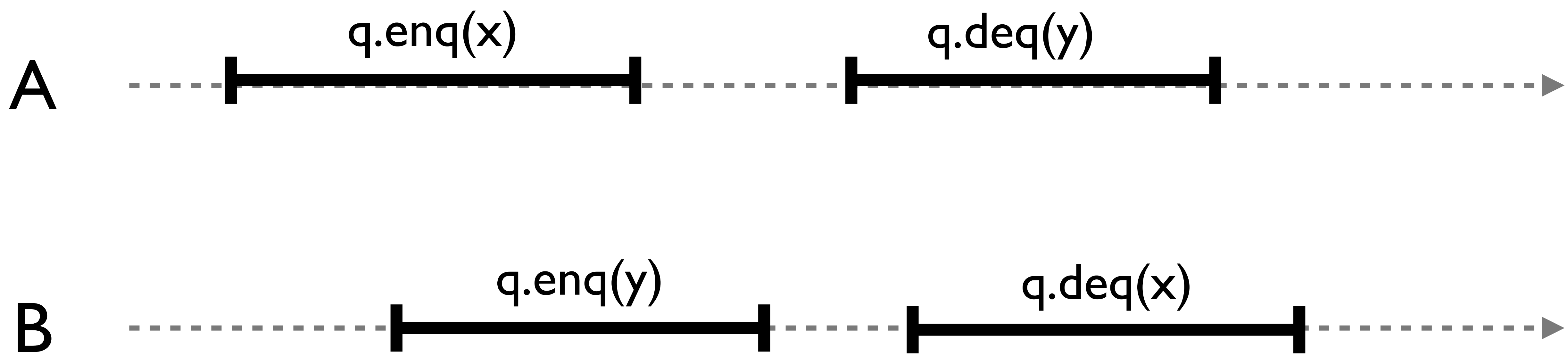


yes! two possible consistent orderings

A: enqueues x
B: enqueues y
B: dequeues x
A: dequeues y

B: enqueues y
A: enqueues x
A: dequeues y
B: dequeues x

Is this execution sequentially consistent?



yes! two possible consistent orderings

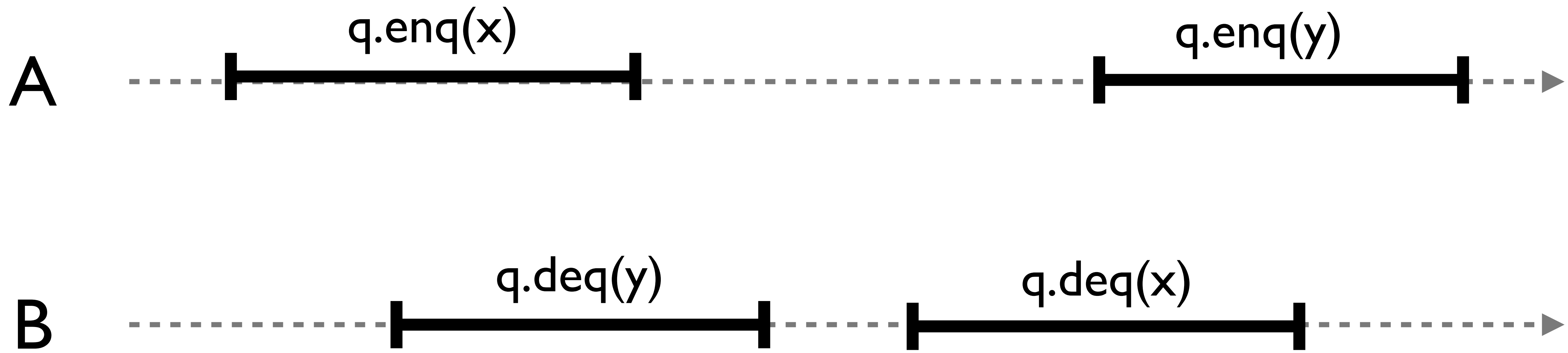
A: enqueues x
B: enqueues y
B: dequeues x
A: dequeues y

B: enqueues y
A: enqueues x
A: dequeues y
B: dequeues x

sequential program order preserved!

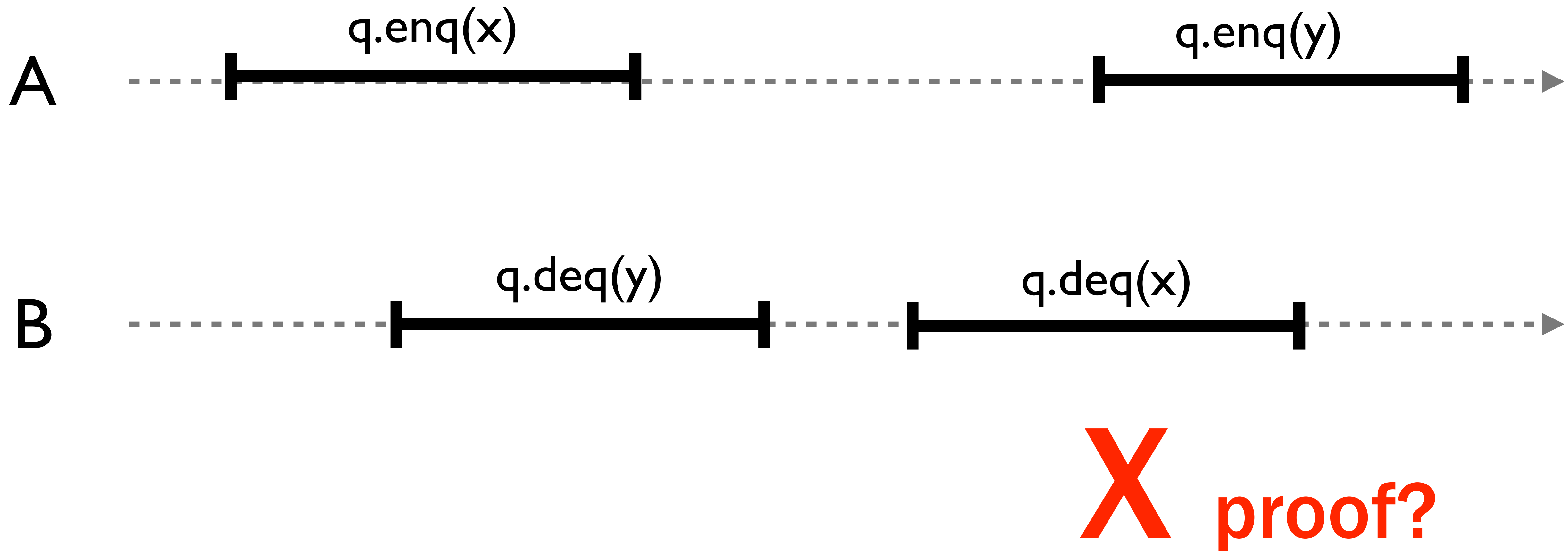
Is this execution sequentially consistent?

(from Huisman)



Is this execution sequentially consistent?

(from Huisman)



Quiescent vs. sequential consistency

- quiescent and sequential consistency are incomparable

  => *quiescent consistency does not necessarily preserve program order*
  => *sequential consistency is unaffected by quiescent periods*

- a correctness condition C is compositional if whenever every object satisfies C, the system as a whole satisfies C

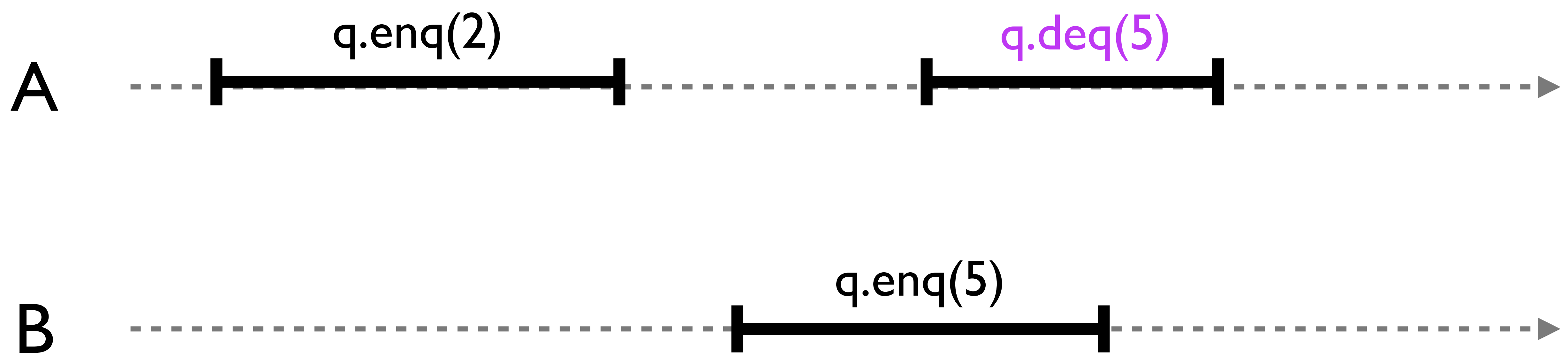
  => *quiescent consistency is compositional*
  => *sequential consistency, unfortunately, is not compositional*

Next on the agenda

1. concurrent objects and correctness
2. quiescent and sequential consistency
3. linearizability

Strengthening sequential consistency to gain compositionality

- should this (sequentially consistent) behaviour be allowed?



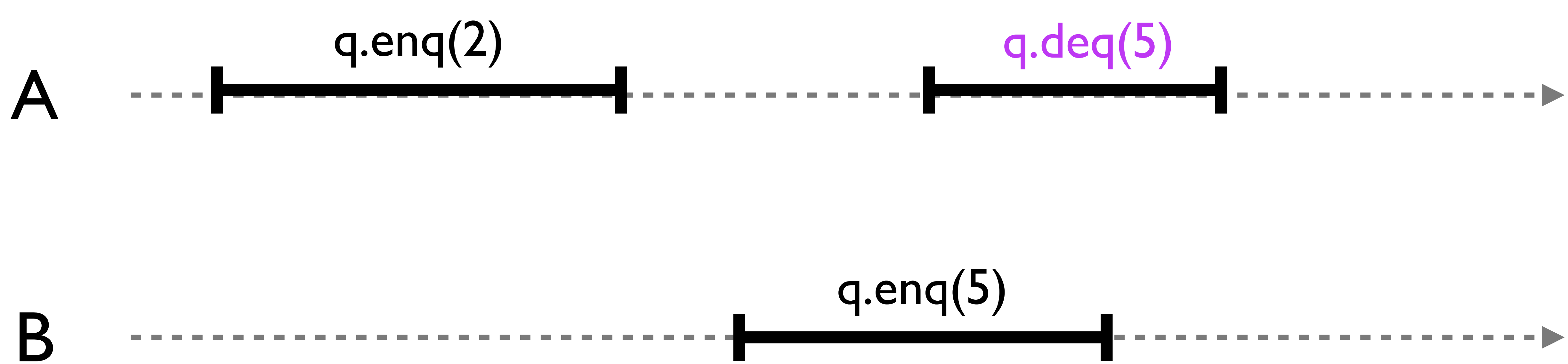
Linearizability

each method call should appear to take effect instantaneously at some moment <u>between its invocation and response</u>

an <u>object</u> is linearizable if all of its possible executions are linearizable

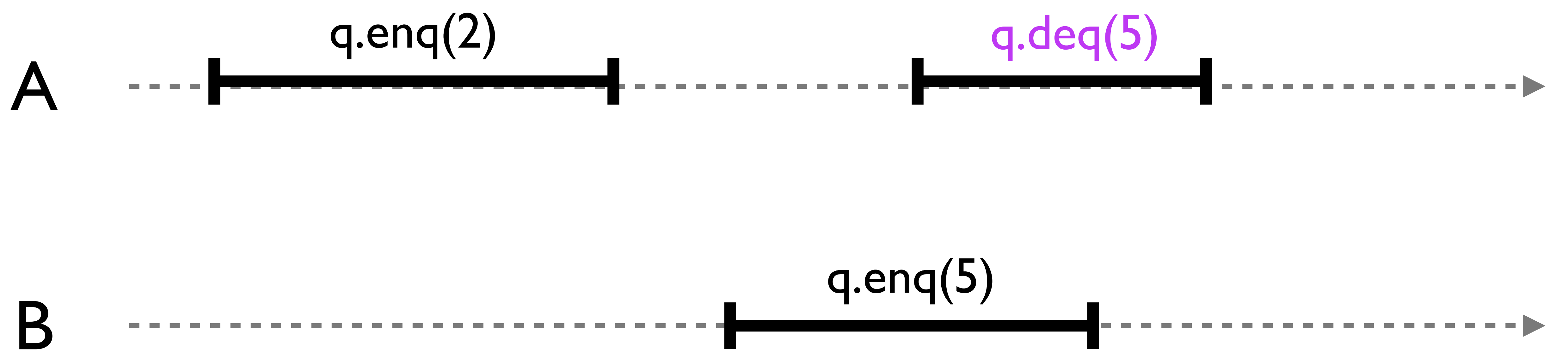
Linearizability

each method call should appear to take effect instantaneously at some moment <u>between its invocation and response</u>



Linearizability

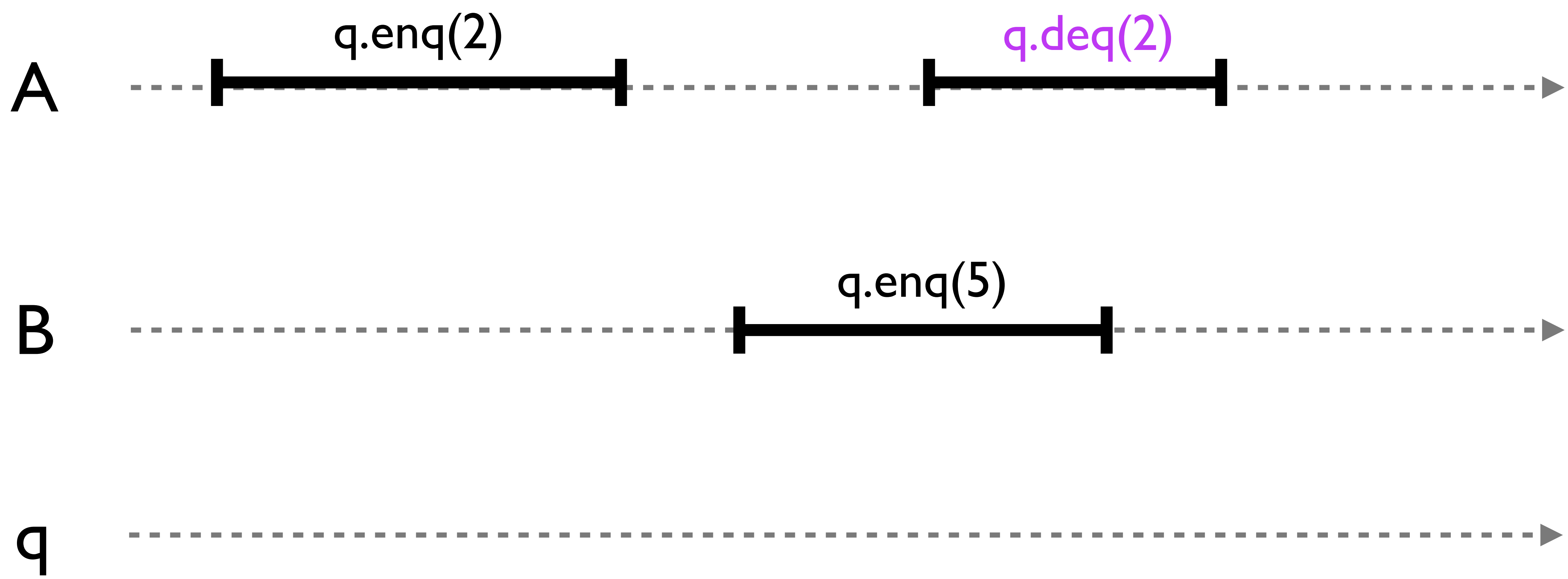
each method call should appear to take effect instantaneously at some moment <u>between its invocation and response</u>



X sequentially consistent, but not linearizable

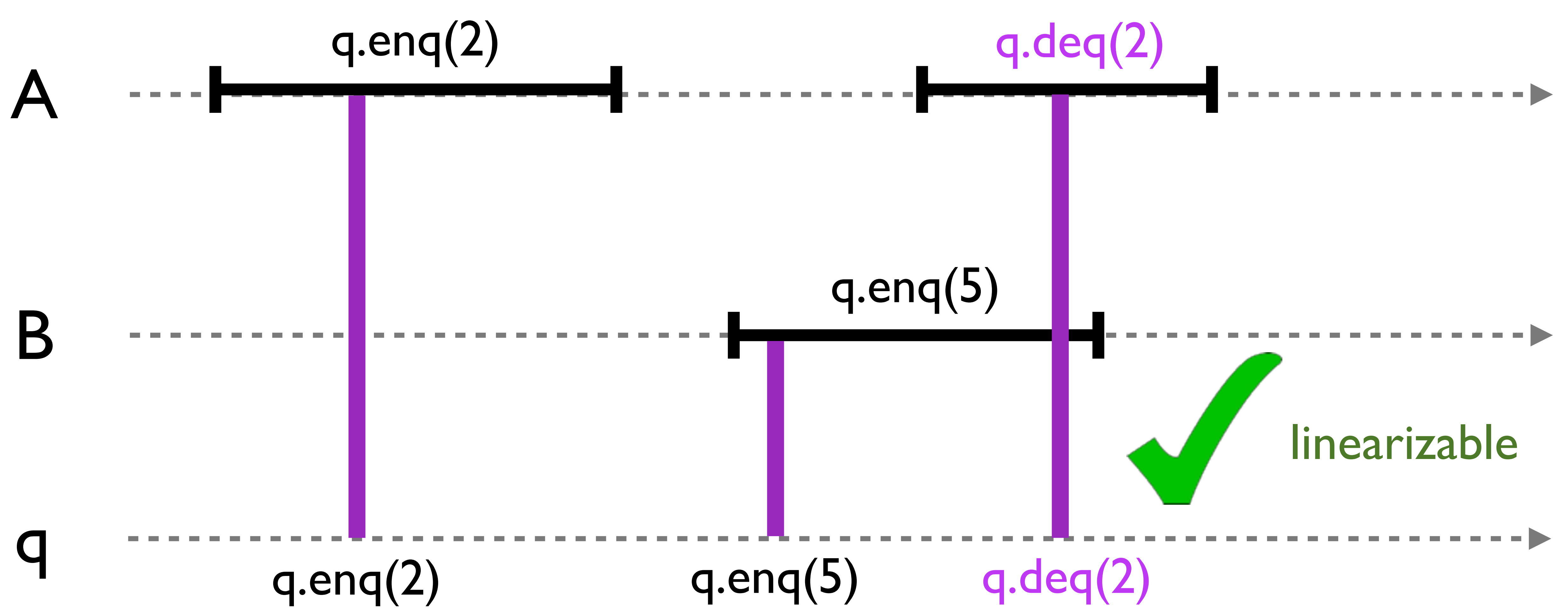
Linearizability

each method call should appear to take effect instantaneously at some moment between its invocation and response

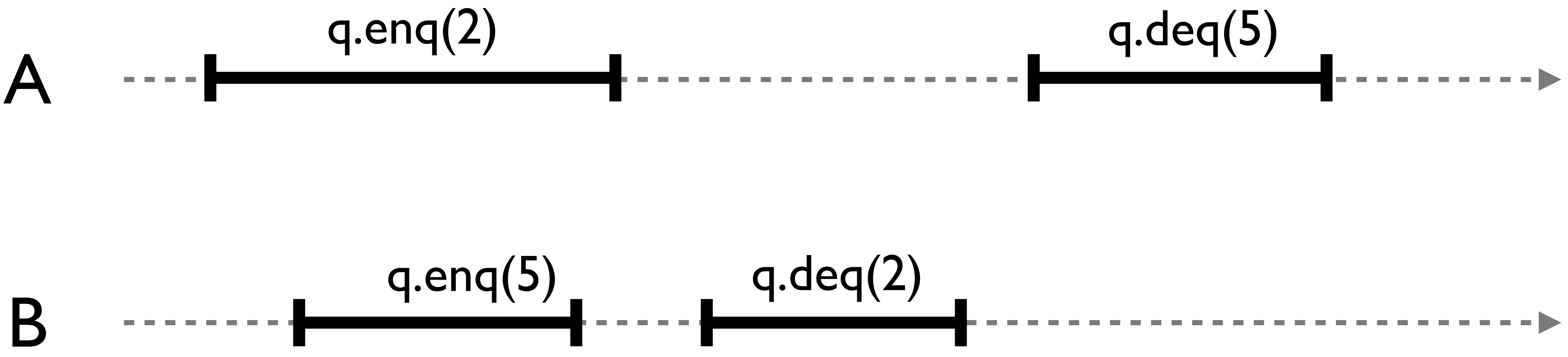


Linearizability

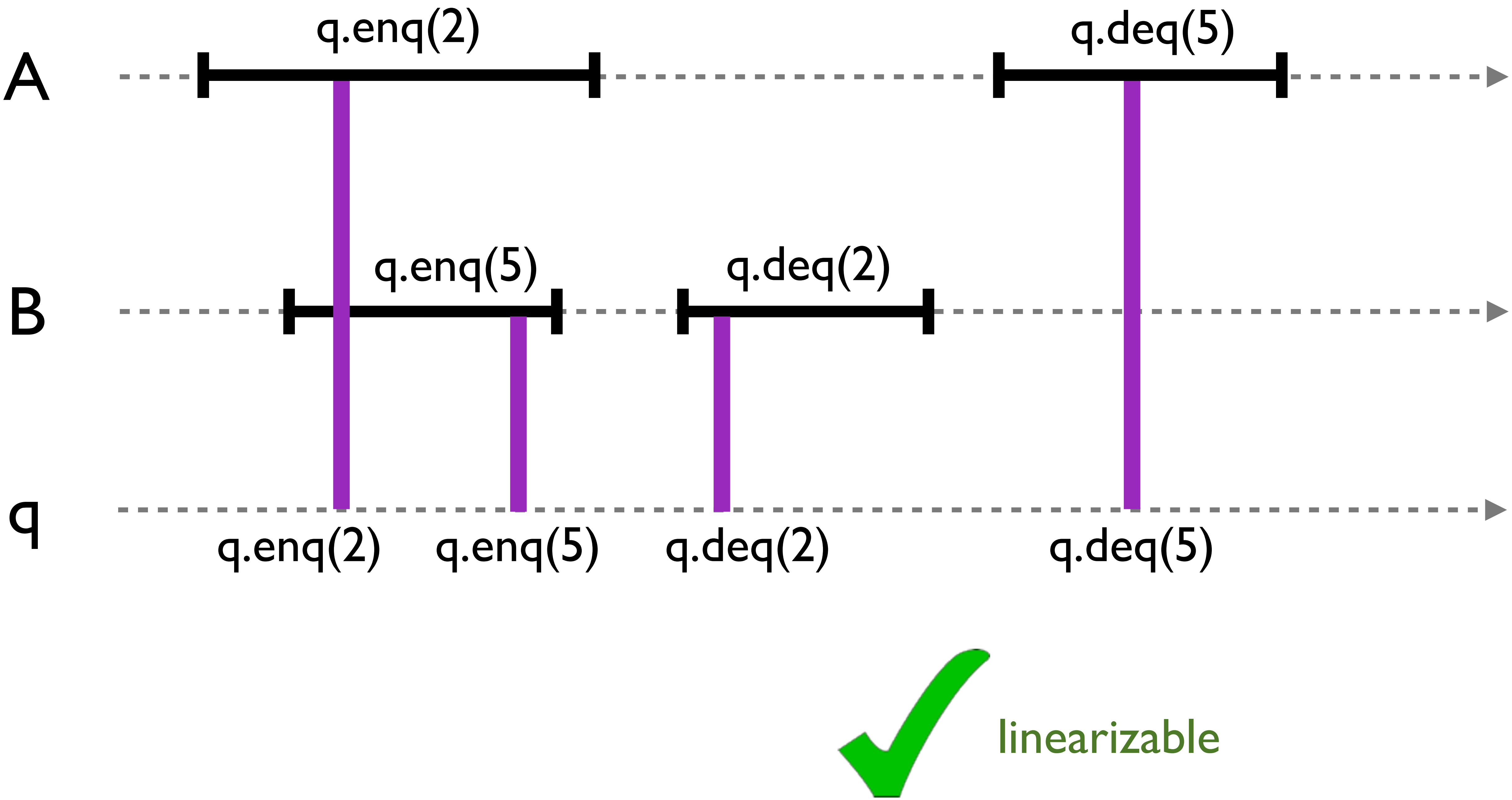
each method call should appear to take effect instantaneously at some moment <u>between its invocation and response</u>



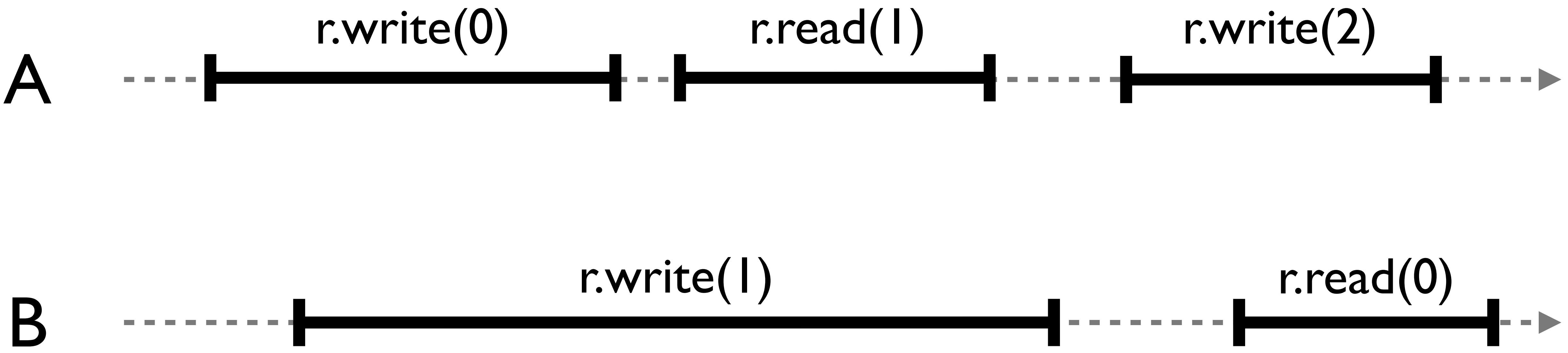
Linearizability: examples



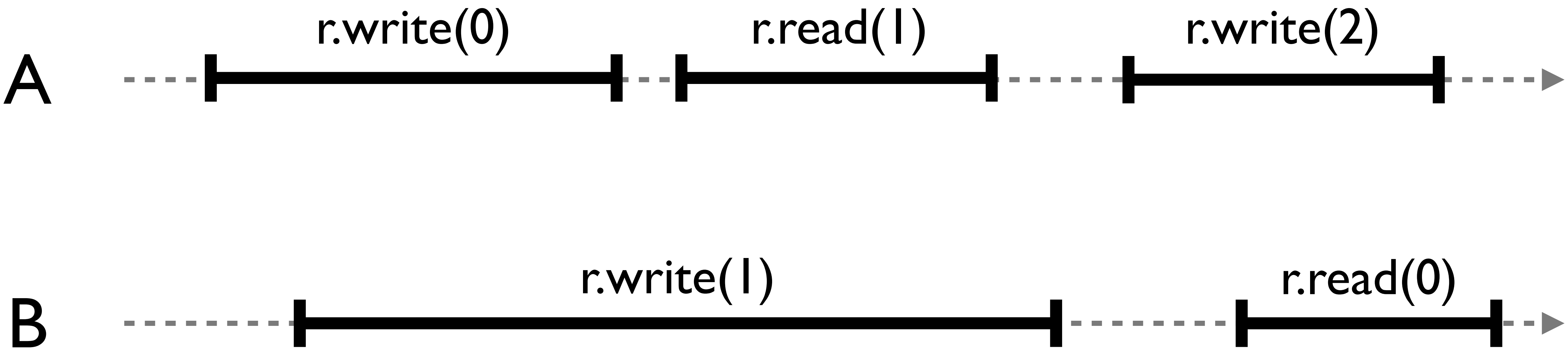
Linearizability: examples



Linearizability: examples

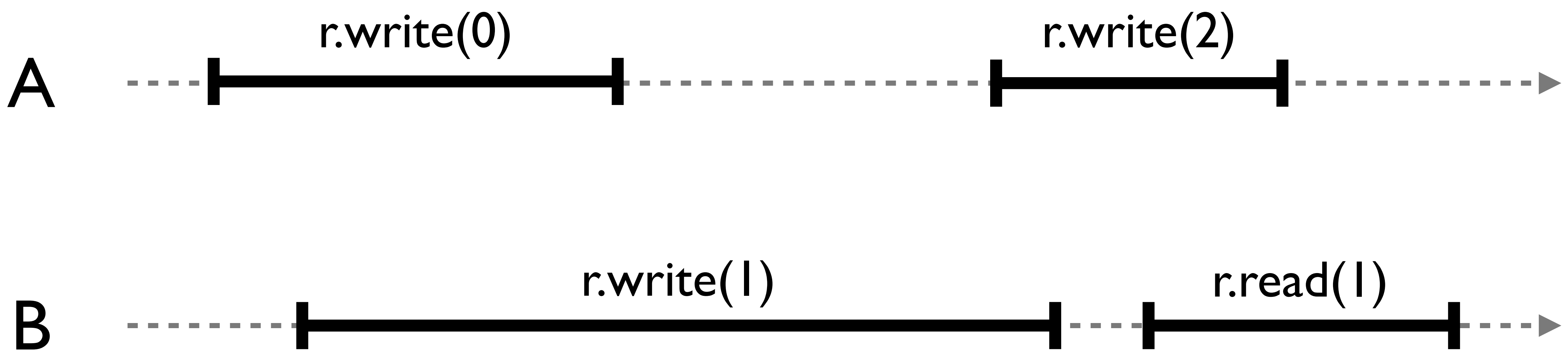


Linearizability: examples

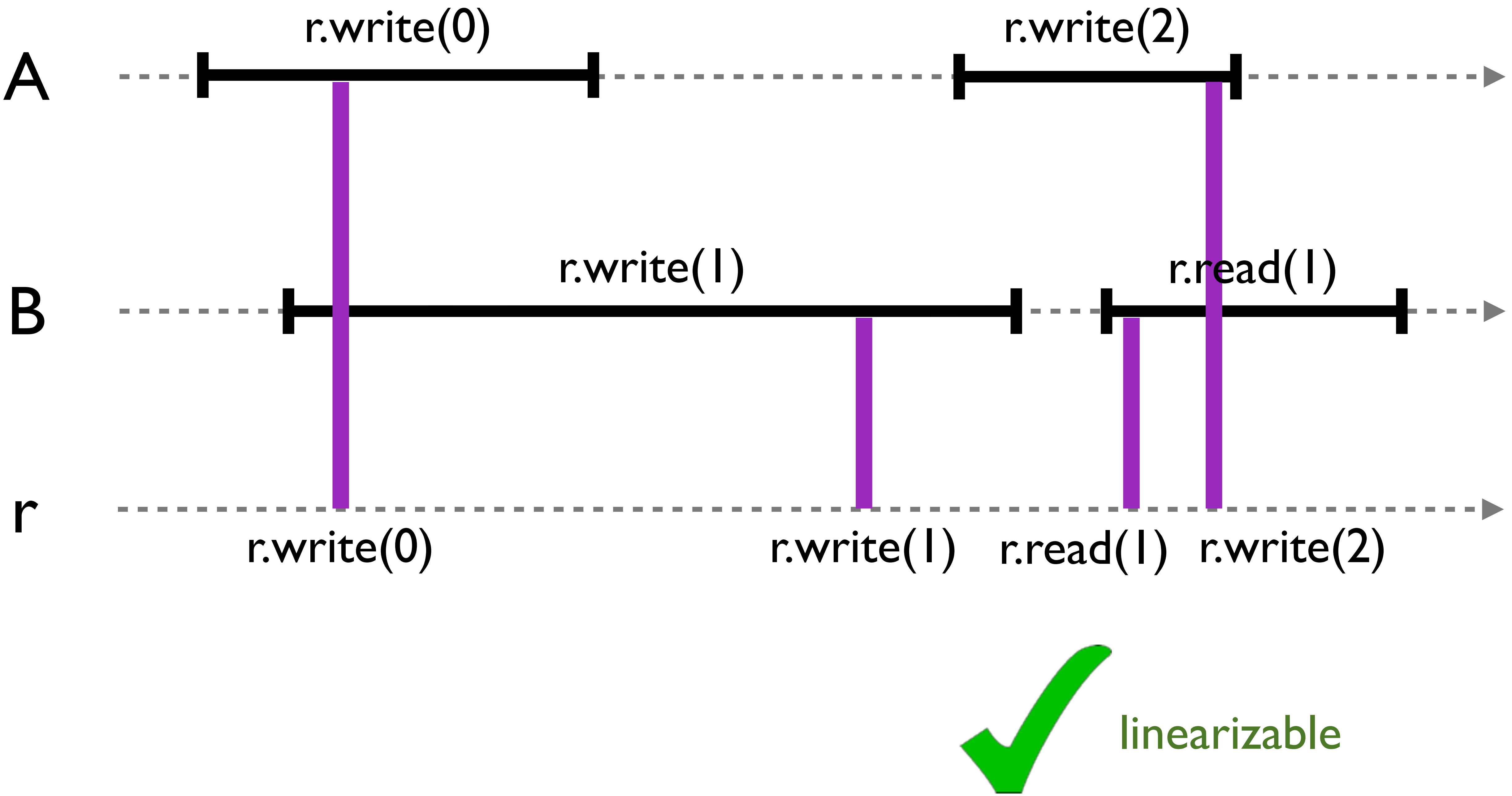


X not sequentially consistent, not linearizable

Linearizability: examples



Linearizability: examples



Linearization points

- to show that a concurrent object is linearizable, one must identify for each method a linearization point where the method takes effect
- for lock-based objects, these are the critical sections
- for lock-free approaches, the linearization point is a single step where they effects of the method call become visible to other method calls

Sequential consistency vs. linearizability

- linearizable executions are also sequentially consistent
- sequential consistency is less restrictive: allows method calls to take effect after their response
- linearizability is compositional: the result of composing linearizable objects is linearizable

Formal definitions

- a call of an operation is split into two events:

  invocation [A q.op(a_1, ..., a_n)]
  response [A q:Ok(r)]

- where A is a thread ID, q an object, op(a_1, ..., a_n) an invocation of call with arguments, and Ok(r) a successful response of call with result r

- a history is a sequence of invocation / response events

Formal definitions

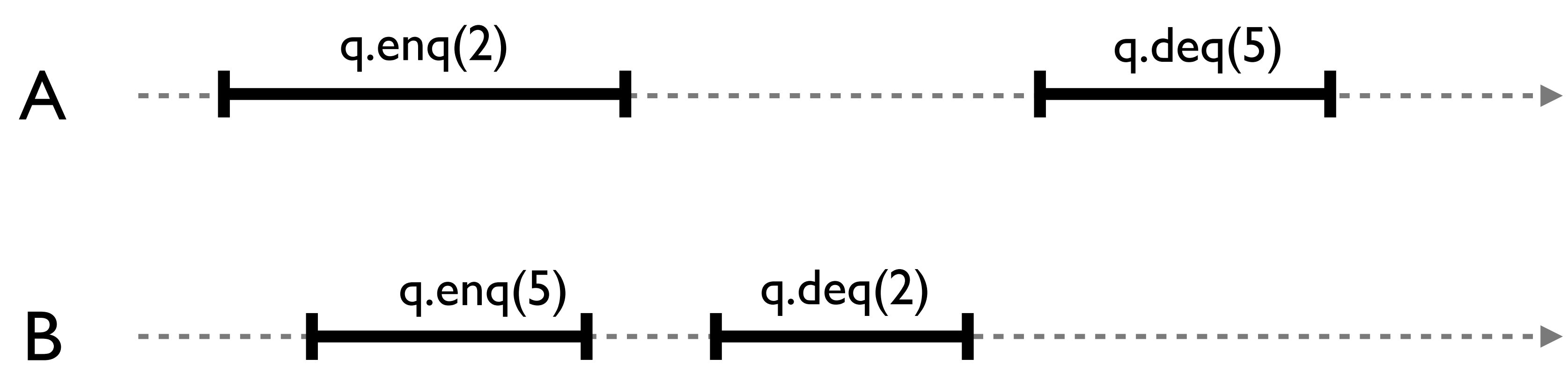
- a call of an operation is split into two events:

  invocation [A q.op(a_1, ..., a_n)]
  response [A q:Ok(r)]

- where A is a thread ID, q an object, op(a_1, ..., a_n) an invocation of call with arguments, and Ok(r) a successful response of call with result r

- a history is a sequence of invocation / response events

H = [A q.enq(2)], [B q.enq(5)], [B q.Ok], [A q.Ok], [B q.deq()], [B q.Ok(2)], [A q.deq()], [A q.Ok(5)]



Formal definitions

- we can define projections on objects and on threads

- assume we have a history:

H = [A q1.enq(2)], [B q2.enq(5)], [B q2.Ok], [A q1.Ok],
[B q1.deq()], [B q1.Ok(2)], [A q2.deq()], [A q2.Ok(5)]

- object projection:

H|q1 = [A q1.enq(2)], [A q1.Ok], [B q1.deq()], [B q1.Ok(2)]

- thread projection:

H|A = [A q1.enq(2)], [A q1.Ok], [A q2.deq()], [A q2.Ok(5)]

Formal definitions

- a response matches an invocation if their object and thread names agree

- a history is sequential if it starts with an invocation, and each invocation (except possibly the last) is immediately followed by a matching response

H = [A q.enq(2)], [A q.Ok], [B q.enq(5)], [B q.Ok]

- a sequential history is legal if it agrees with the sequential specification of each object

Formal definitions

- a call op_1 precedes another call op_2 ($op_1 \rightarrow op_2$) if op_1's response event occurs before op_2's invocation event

- we write $\rightarrow_H$ for the precedence relation induced by H

  => *e.g.* $q.enq(2) \rightarrow_H q.enq(5)$

- an invocation is pending if it has no matching response

- a history is complete if it does not have pending responses

- complete(H) is the subhistory of H with all pending invocations removed

Linearizability: the definition

- two histories H and G are equivalent if H|A = G|A for all threads A

- a history H is linearizable if it can be extended to a history G by adding zero or more response events, such that:

  => *complete(G) is equivalent to some legal sequential history S*

  => $\rightarrow_H \subseteq \rightarrow_S$ *(i.e. the precedences of H are maintained)*

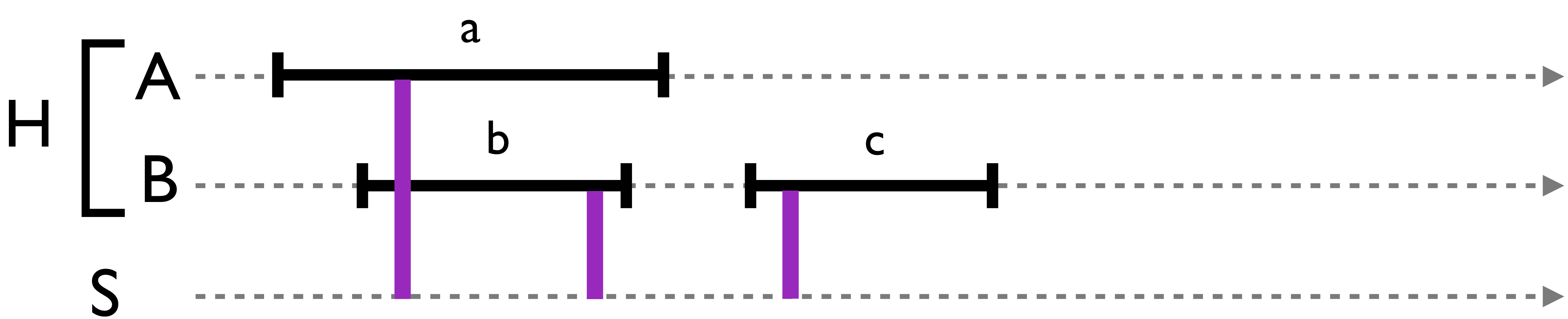
Linearizability: the definition

- two histories H and G are equivalent if H|A = G|A for all threads A

- a history H is linearizable if it can be extended to a history G by adding zero or more response events, such that:

  => complete(G) is equivalent to some legal sequential history S

  => $\rightarrow_H \subseteq \rightarrow_S$ *(i.e. the precedences of H are maintained)*



$\rightarrow_H = \{a \rightarrow c, b \rightarrow c\}$ $\rightarrow_S = \{a \rightarrow b, a \rightarrow c, b \rightarrow c\}$

Next on the agenda

1. concurrent objects and correctness ✓
2. quiescent and sequential consistency ✓
3. linearizability ✓

Final remarks

- correctness notions for concurrent objects boil down to "equivalences" with sequential computations

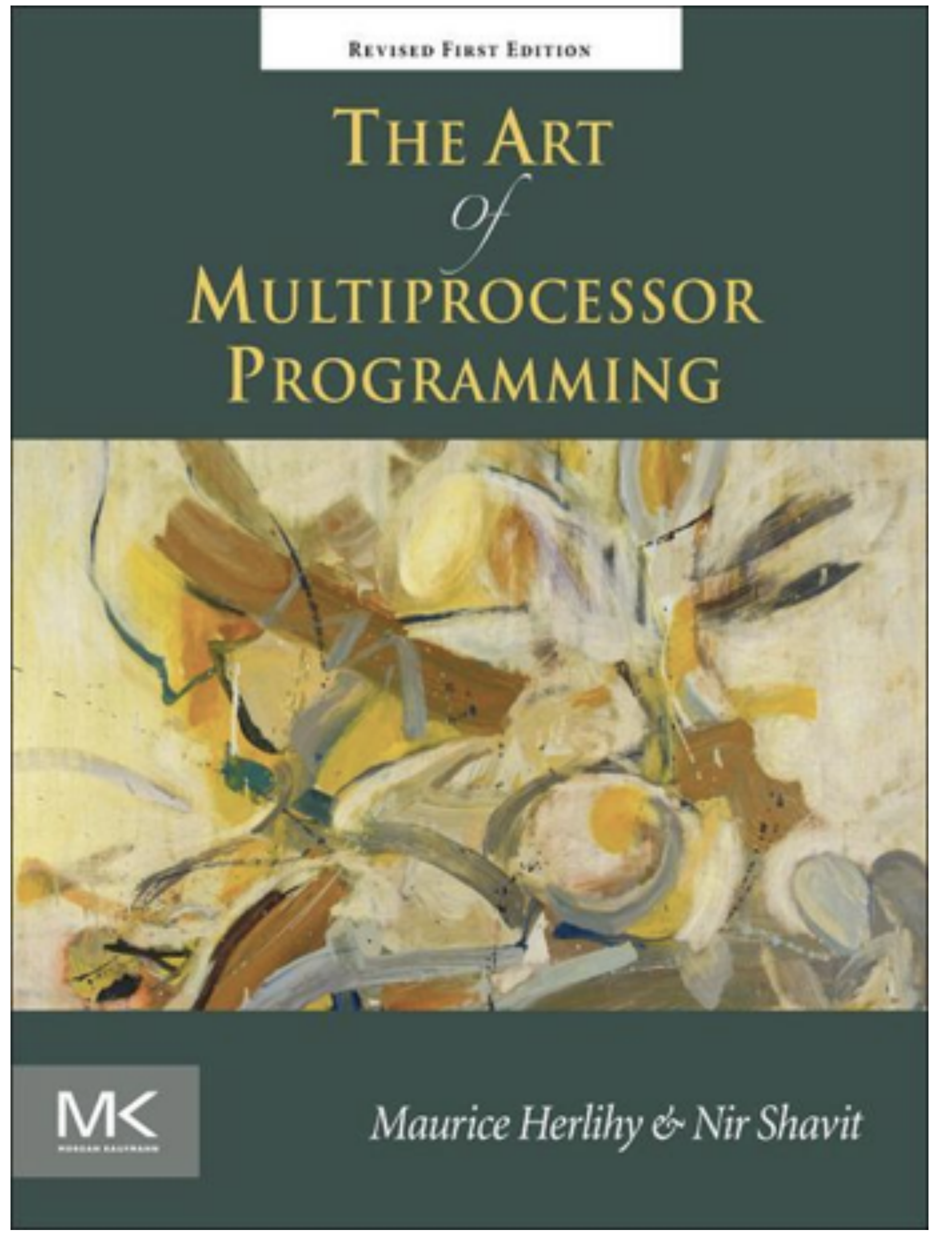
  => *quiescent consistency, sequential consistency, linearizability*
  => *objects we built in previous weeks were linearizable*

- correctness conditions depend on the application's needs

- in most modern multiprocessor architectures, memory reads/writes are not sequentially consistent

  => *too expensive!*
  => *must "ask" for it explicitly when needed*

Lecture based on Chapter 3 of:



recommended reading!