Design of Parallel and High-Performance Computing

Fall 2013

Lecture: Linearizability

Motivational video: https://www.youtube.com/watch?v=qx2dRIQXnbs

Instructor: Torsten Hoefler & Markus Püschel

TAs: Timo Schneider

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

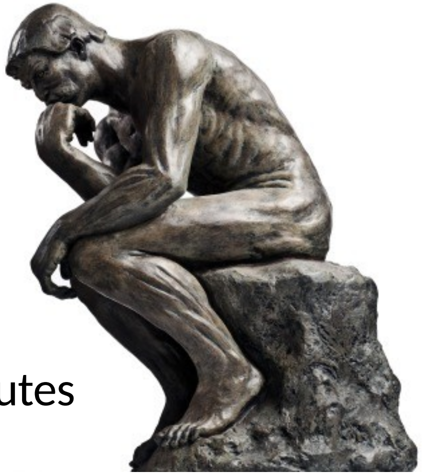
Review of last lecture

- **Cache-coherence is not enough!**
  - Many more subtle issues for parallel programs!

- **Memory Models**
  - Sequential consistency
  - Why threads cannot be implemented as a library ^^
  - Relaxed consistency models
  - x86 TLO+CC case study

- **Complexity of reasoning about parallel objects**
  - Serial specifications (e.g., pre-/postconditions)
  - Started to lock things …

Peer Quiz

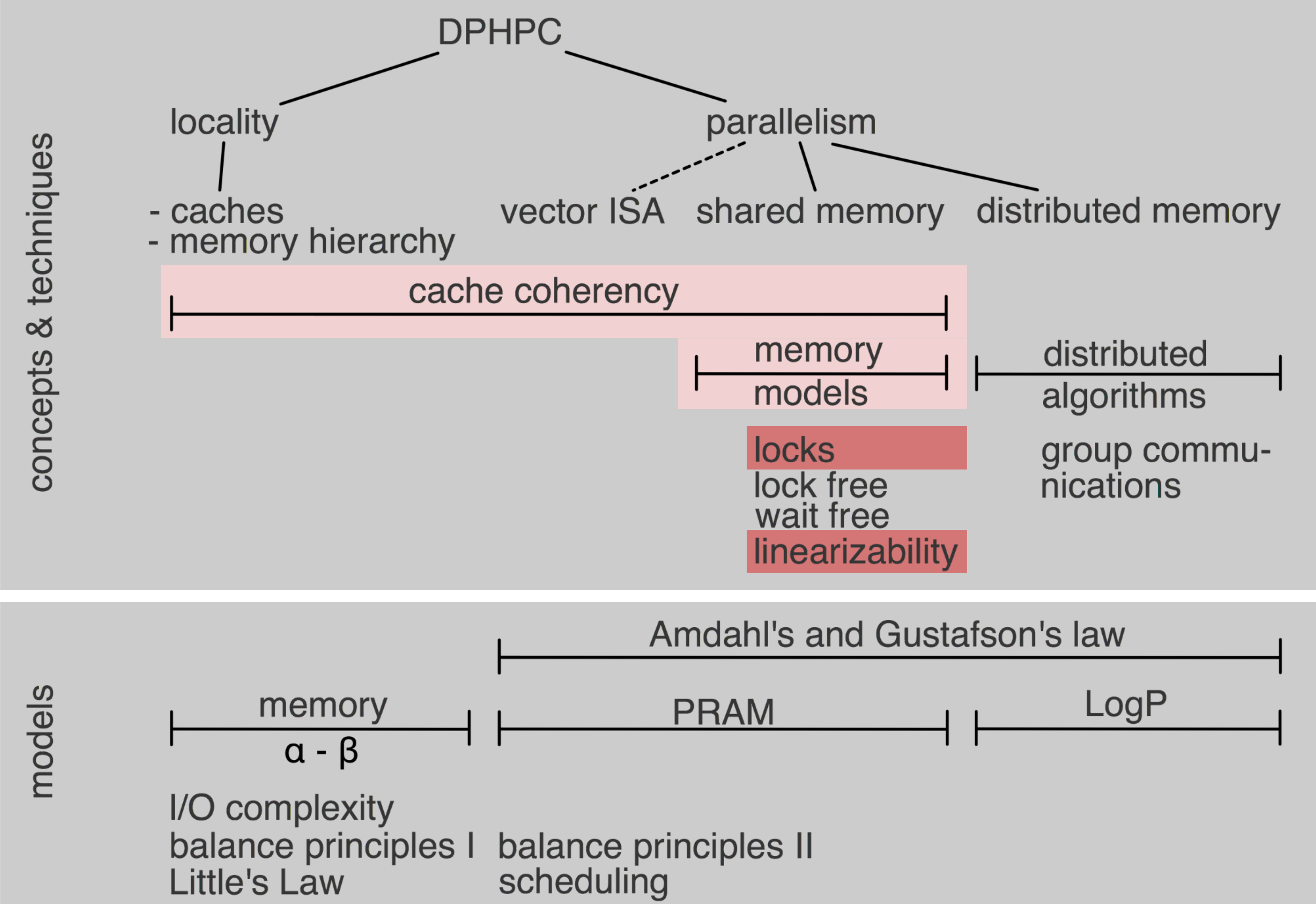


- **Instructions:**
  - Pick some partners (locally) and discuss each question for 2 minutes
  - We then select a random student (team) to answer the question

- **What are the problems with sequential consistency?**
  - Is it practical? Explain!
  - Is it sufficient for simple parallel programming? Explain!
  - How would you improve the situation?

- **How could memory models of practical CPUs be described?**
  - Is Intel's definition useful?
  - Why would one need a better definition?
  - Threads cannot be implemented as a library? Why does Pthreads work?

DPHPC Overview



concepts & techniques

DPHPC

locality

parallelism

- caches
- memory hierarchy

vector ISA

shared memory

distributed memory

cache coherency

memory models

distributed algorithms

locks

lock free

wait free

linearizability

group communications

models

Amdahl's and Gustafson's law

memory

α - β

PRAM

LogP

I/O complexity

balance principles I

Little's Law

balance principles II

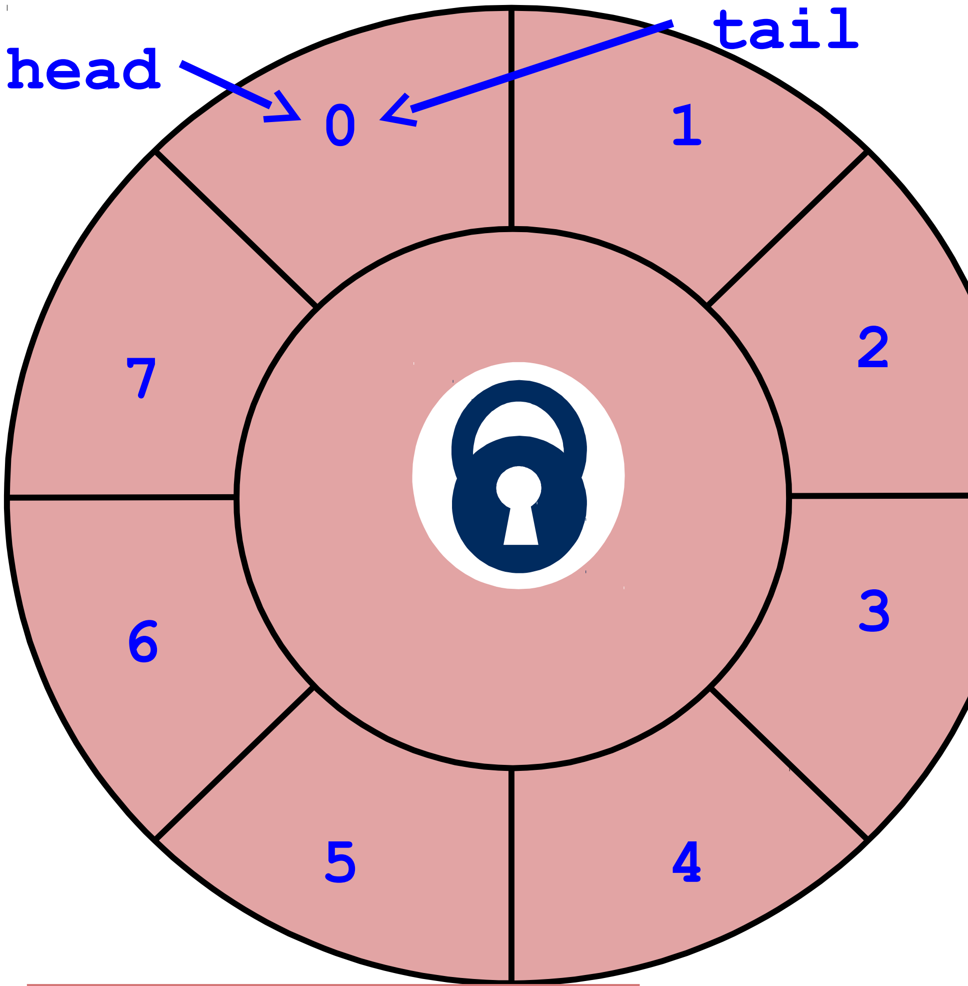
scheduling

Goals of this lecture

- **Queue:**
  - Problems with the locked queue
  - Wait-free two-thread queue
- **Linearizability**
  - Intuitive understanding (sequential order on objects!)
  - Linearization points
  - Linearizable executions
  - Formal definitions (Histories, Projections, Precedence)
- **Linearizability vs. Sequential Consistency**
  - Modularity
- **Maybe: lock implementations**

Lock-based queue

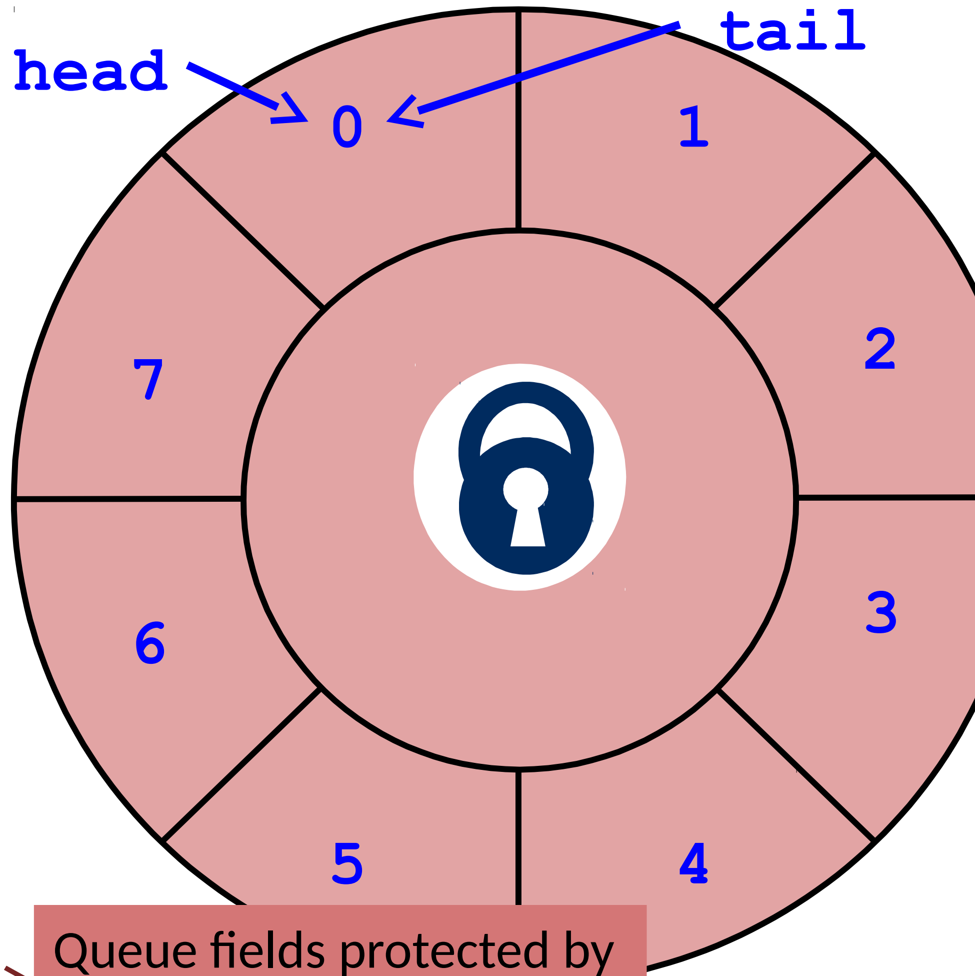
```
class Queue {

  private:
  int head, tail;
  std::vector<Item> items;
  std::mutex lock;

  public:
  Queue(int capacity) {
    head = tail = 0;
    items.resize(capacity);
  }
  ...
};
```



Queue fields protected by single shared lock!

Lock-based queue

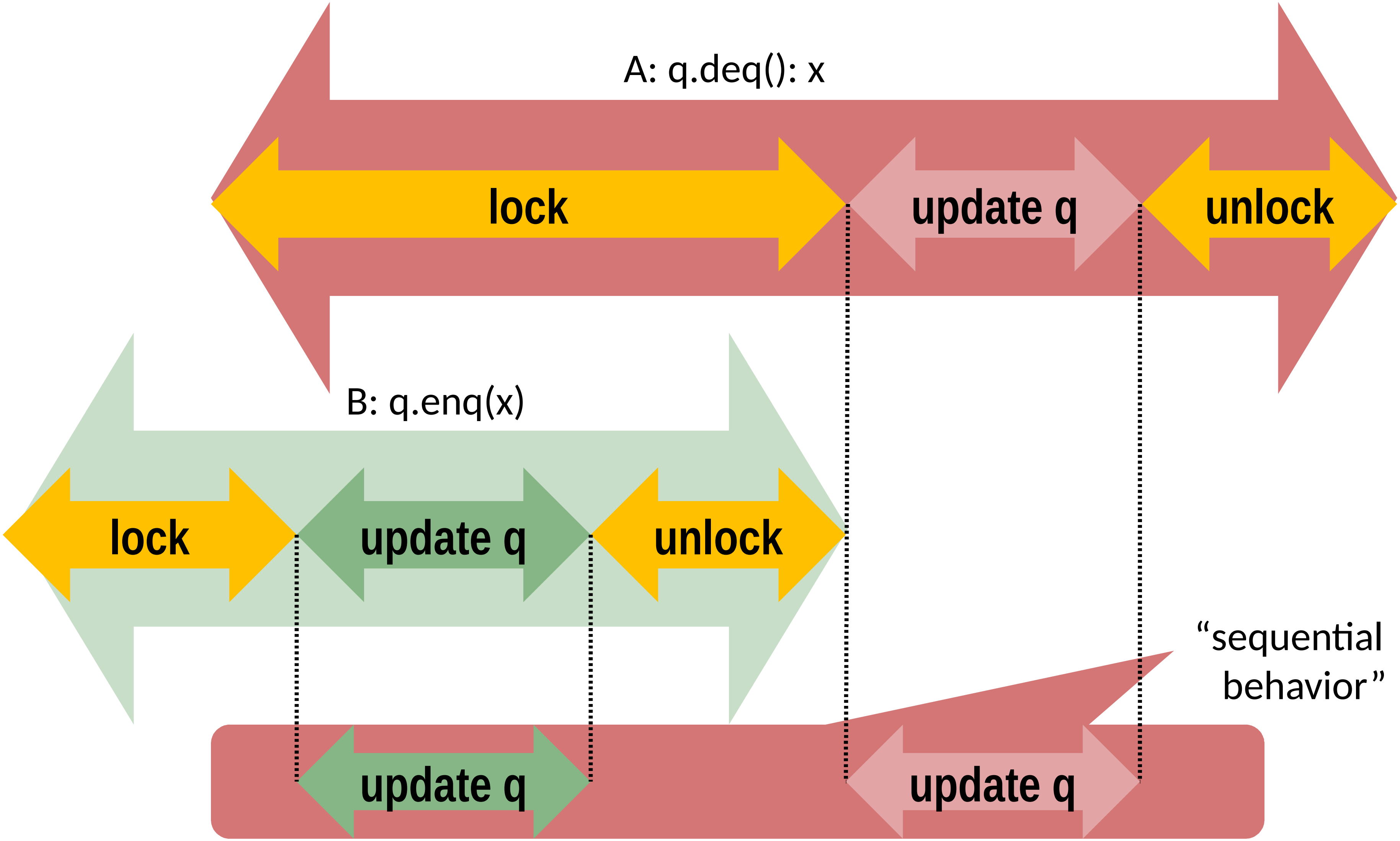
```
class Queue {
 …

 public:
 void enq(Item x) {
  std::lock_guard<std::mutex> l(lock)
  if((tail+1)%items.size()==head) {
   throw FullException;
  }
  items[tail] = x;
  tail = (tail+1)%items.size();
 }

 Item deq() {
  std::lock_guard<std::mutex> l(lock)
  if(tail == head) {
   throw FullException;
  }
  Item item = items[head];
  head = (head+1)%items.size();
  return item;
 }
};
```



Queue fields protected by single shared lock!

Class question: how is the lock ever unlocked?

Example execution



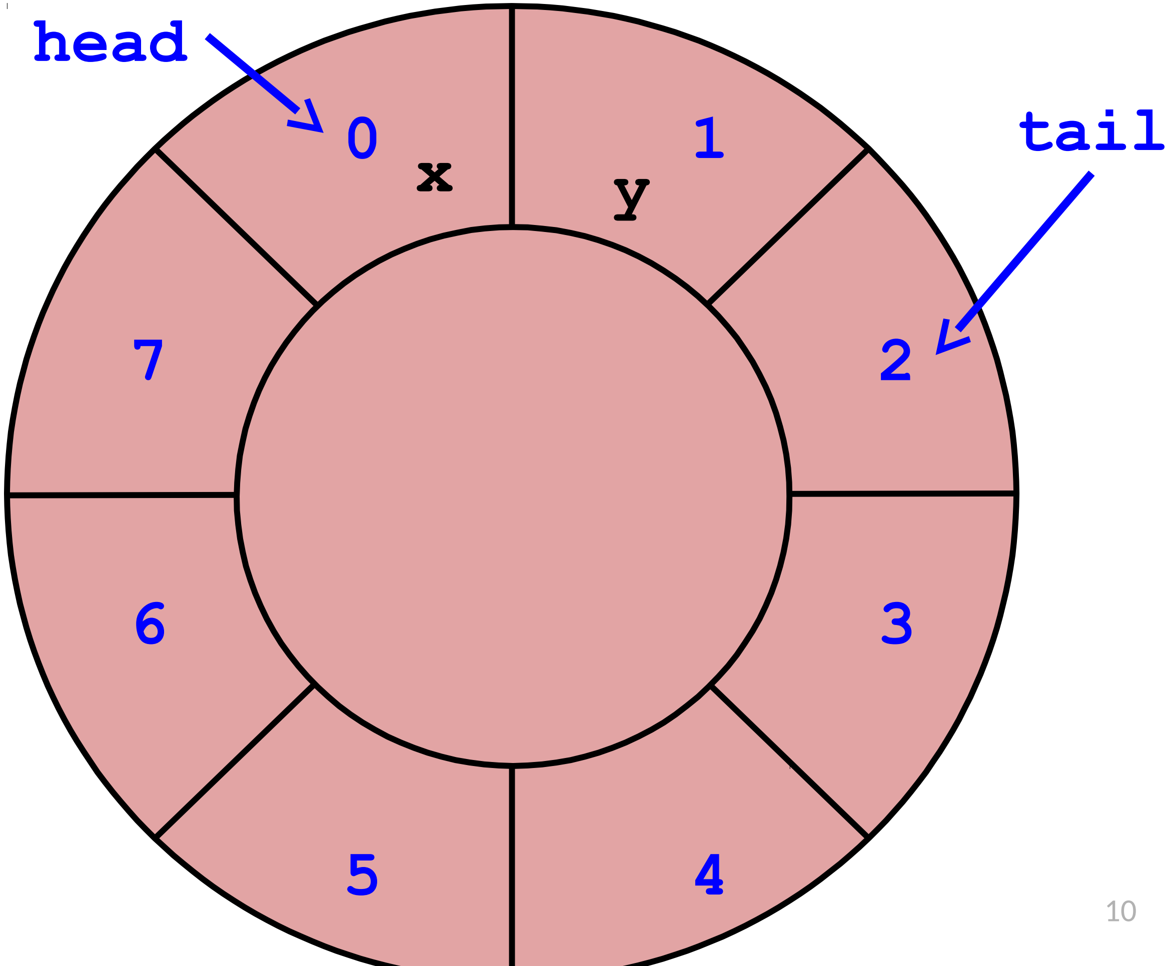
Correctness

- **Is the locked queue correct?**
  - Yes, only one thread has access if locked correctly
  - Allows us again to reason about pre- and postconditions
  - Smells a bit like sequential consistency, no?
- **Class question: What is the problem with this approach?**
  - Same as for SC ㅆ

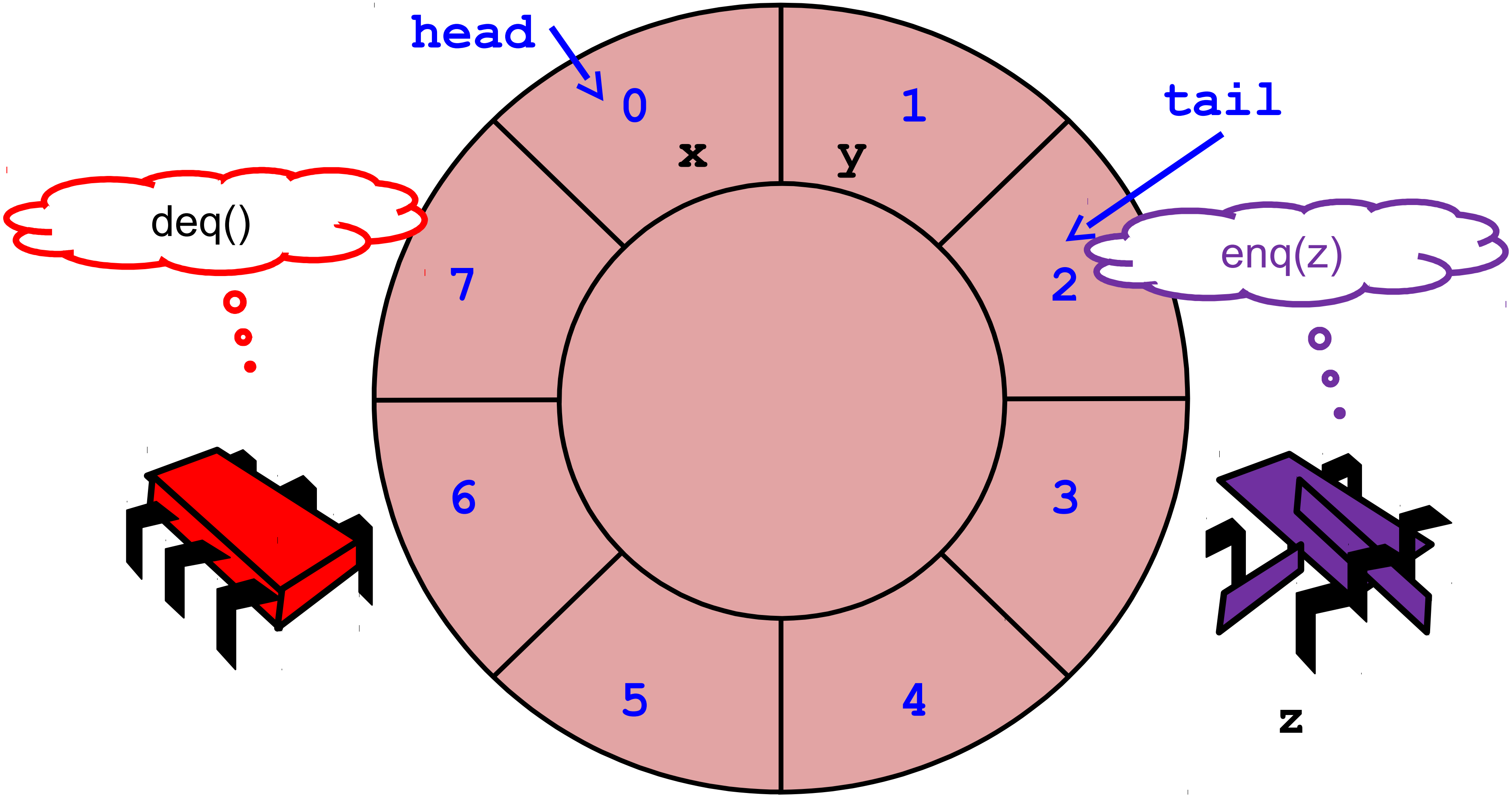
It does not scale!
What is the solution here?

Threads working at the same time?

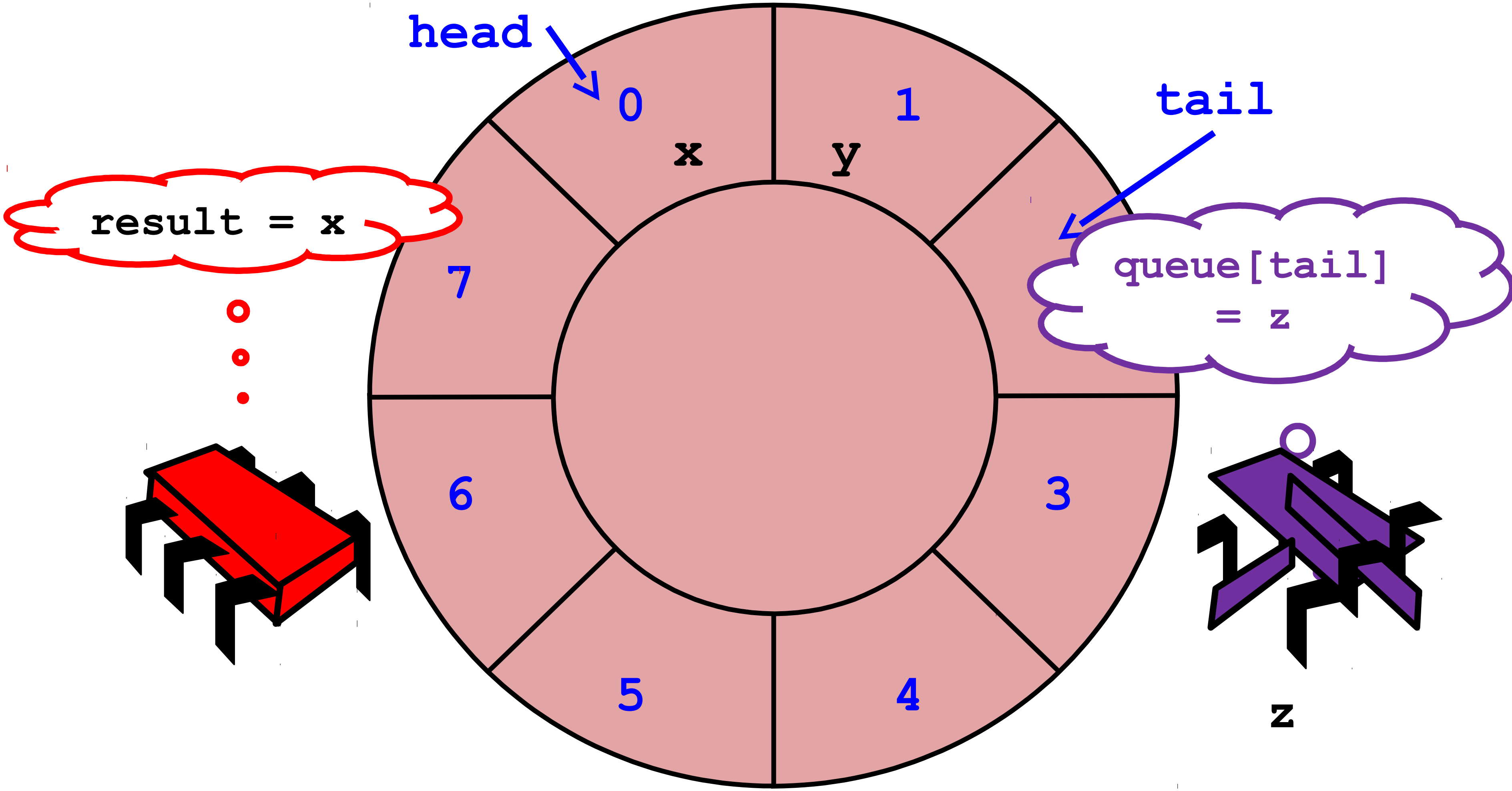
- **Same thing (concurrent queue)**
- **For simplicity, assume only two threads**
  - Thread A calls only enq()
  - Thread B calls only deq()



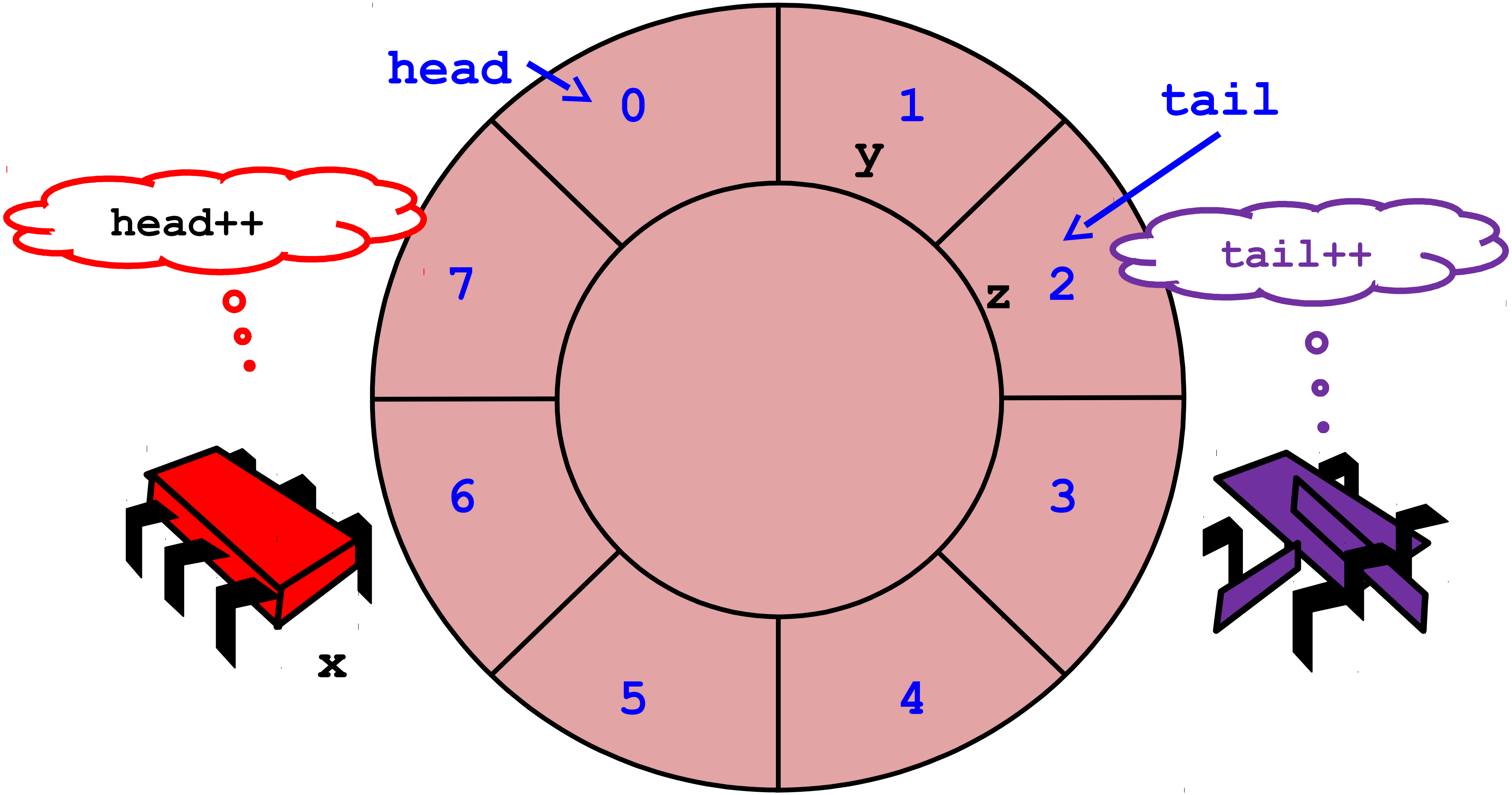
Wait-free 2-Thread Queue



Wait-free 2-Thread Queue



Wait-free 2-Thread Queue



Is this correct?

- **Hard to reason about correctness**
- **What could go wrong?**

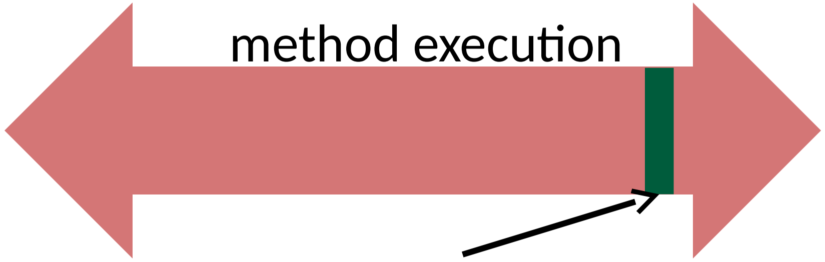
```
void enq(Item x) {
  if((tail+1)%items.size() == head) {
    throw FullException;
  }
  items[tail] = x;
  tail = (tail+1)%items.size();
}
```

```
Item deq() {
  if(tail == head) {
    throw EmptyException;
  }
  Item item = items[head];
  head = (head+1)%items.size();
  return item;
}
```

- Nothing (at least no crash)
- Yet, the **semantics** of the queue are funny (define “FIFO” now)!

Serial to Concurrent Specifications

- **Serial specifications are complex enough, so lets stick to them**
  - Define invocation and response events (start and end of method)
  - Extend the sequential concept to concurrency: linearizability
- **Each method should "take effect"**
  - Instantaneously
  - Between invocation and response events
- **A concurrent object is correct if its "sequential" behavior is correct**
  - Called "linearizable"



Linearization point = when method takes effect

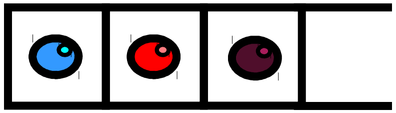
Linearizability

- **Sounds like a property of an execution …**
- **An object is called linearizable if all possible executions on the object are linearizable**
- **Says nothing about the order of executions!**

Example

```
void enq(Item x) {
  std::lock_guard<std::mutex> l(lock)
  if((tail+1)%items.size() == head) {
    throw FullException;
  }
  items[tail] = x;
  tail = (tail+1)%items.size();
}
```

```
Item deq() {
  std::lock_guard<std::mutex> l(lock)
  if(tail == head) {
    throw EmptyException;
  }
  Item item = items[head];
  head = (head+1)%items.size();
}
```



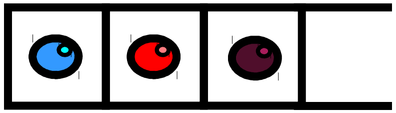
linearization points

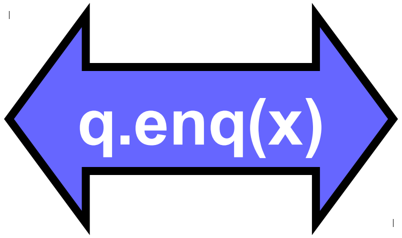
Example

```
void enq(Item x) {
  std::lock_guard<std::mutex> l(lock)
  if((tail+1)%items.size() == head) {
    throw FullException;
  }
  items[tail] = x;
  tail = (tail+1)%items.size();
}
```

```
Item deq() {
  std::lock_guard<std::mutex> l(lock)
  if(tail == head) {
    throw EmptyException;
  }
  Item item = items[head];
  head = (head+1)%items.size();
}
```

linearization points



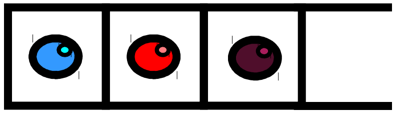


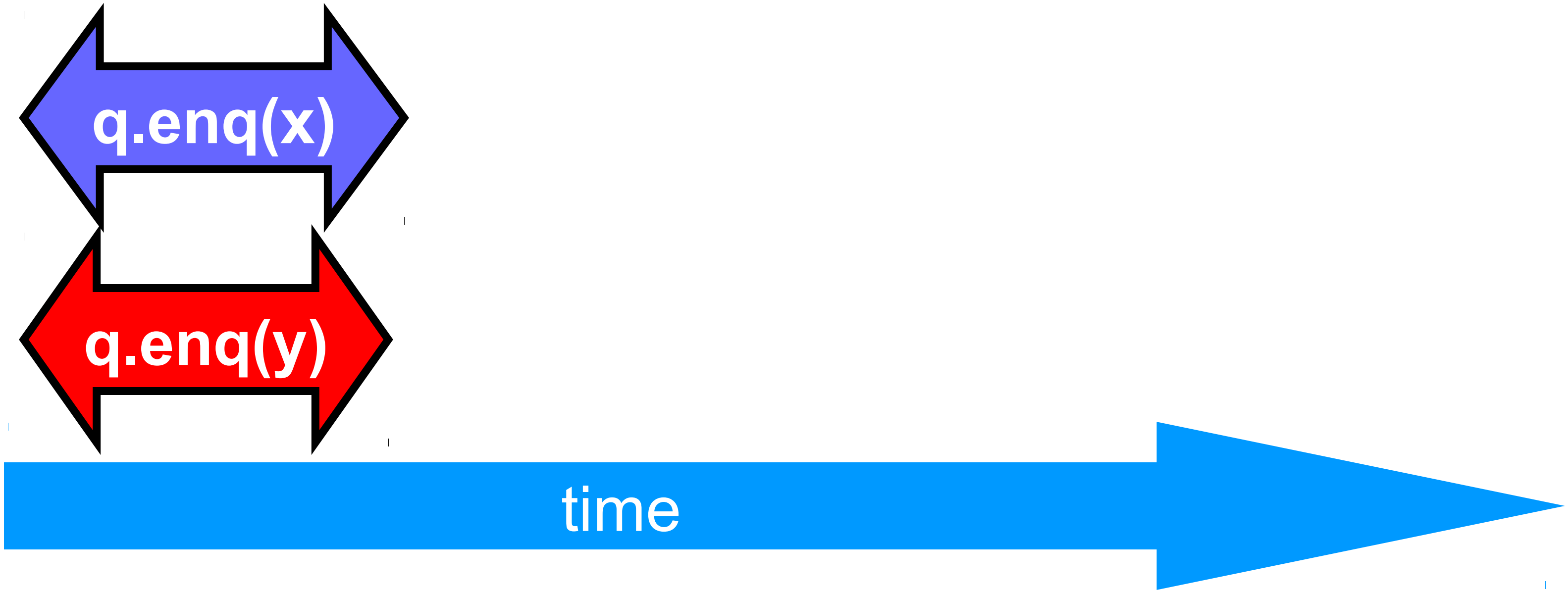
Example

```
void enq(Item x) {
  std::lock_guard<std::mutex> l(lock)
  if((tail+1)%items.size() == head) {
    throw FullException;
  }
  items[tail] = x;
  tail = (tail+1)%items.size();
}
```

```
Item deq() {
  std::lock_guard<std::mutex> l(lock)
  if(tail == head) {
    throw EmptyException;
  }
  Item item = items[head];
  head = (head+1)%items.size();
}
```

linearization points



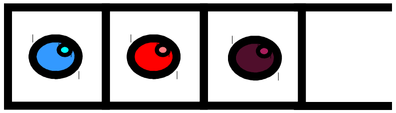


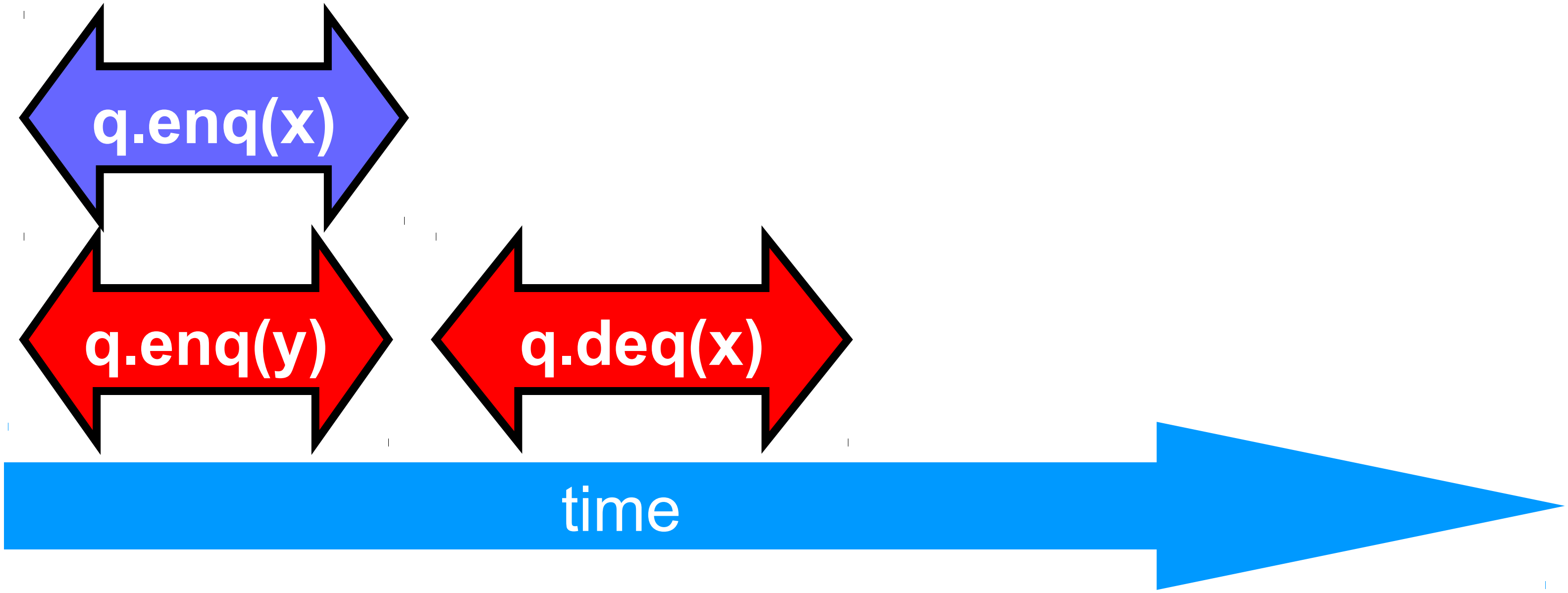
Example

```
void enq(Item x) {
  std::lock_guard<std::mutex> l(lock)
  if((tail+1)%items.size() == head) {
    throw FullException;
  }
  items[tail] = x;
  tail = (tail+1)%items.size();
}
```

```
Item deq() {
  std::lock_guard<std::mutex> l(lock)
  if(tail == head) {
    throw EmptyException;
  }
  Item item = items[head];
  head = (head+1)%items.size();
}
```

linearization points



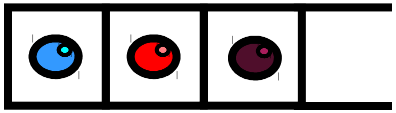


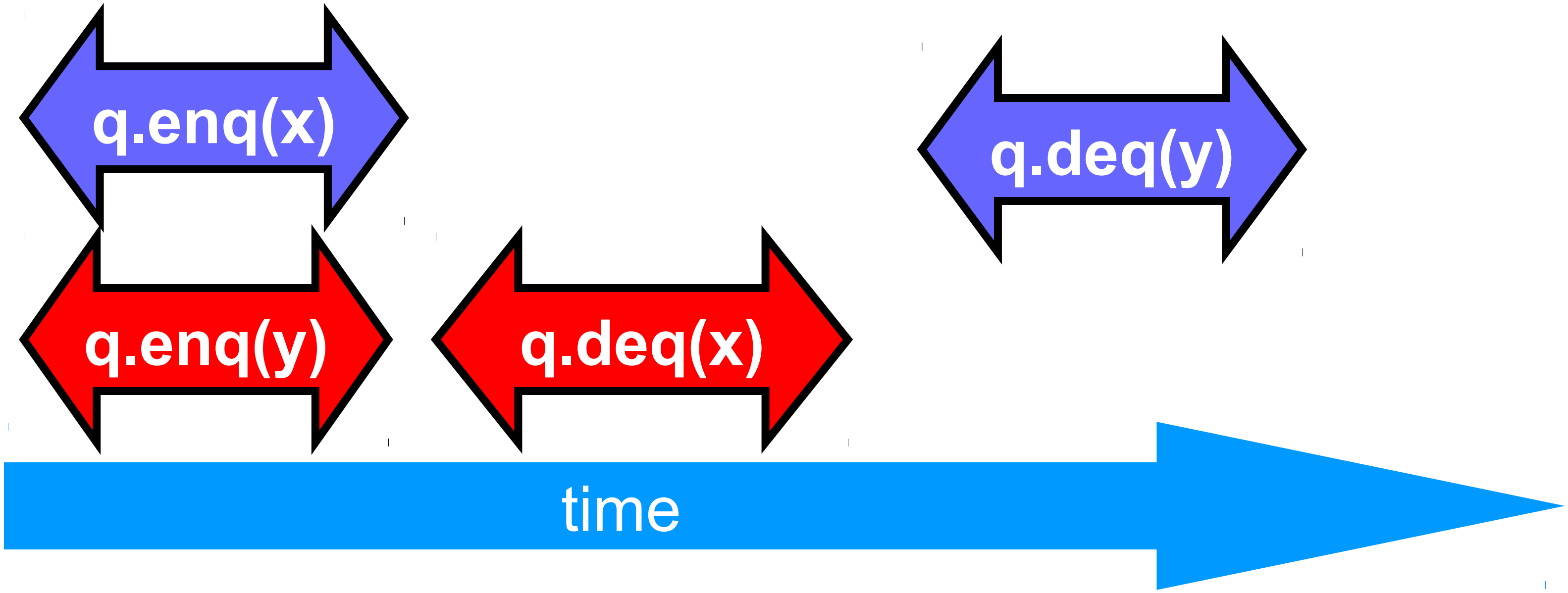
Example

```
void enq(Item x) {
  std::lock_guard<std::mutex> l(lock)
  if((tail+1)%items.size() == head) {
    throw FullException;
  }
  items[tail] = x;
  tail = (tail+1)%items.size();
}
```

```
Item deq() {
  std::lock_guard<std::mutex> l(lock)
  if(tail == head) {
    throw EmptyException;
  }
  Item item = items[head];
  head = (head+1)%items.size();
}
```

linearization points



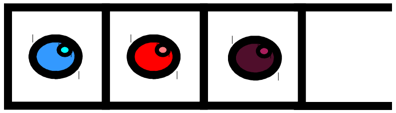


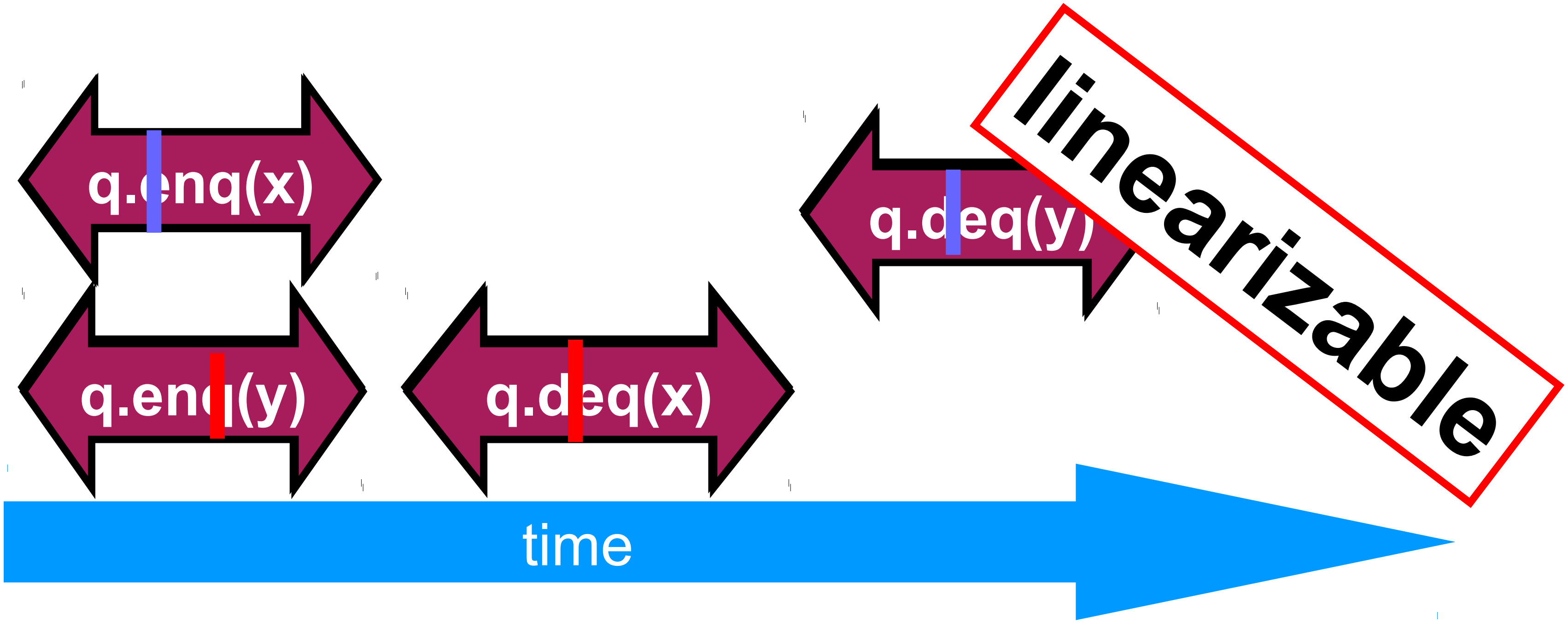
Example

```
void enq(Item x) {
  std::lock_guard<std::mutex> l(lock)
  if((tail+1)%items.size() == head) {
    throw FullException;
  }
  items[tail] = x;
  tail = (tail+1)%items.size();
}
```

```
Item deq() {
  std::lock_guard<std::mutex> l(lock)
  if(tail == head) {
    throw EmptyException;
  }
  Item item = items[head];
  head = (head+1)%items.size();
}
```

linearization points



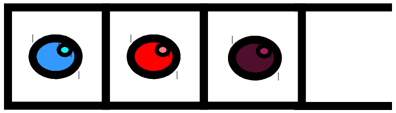


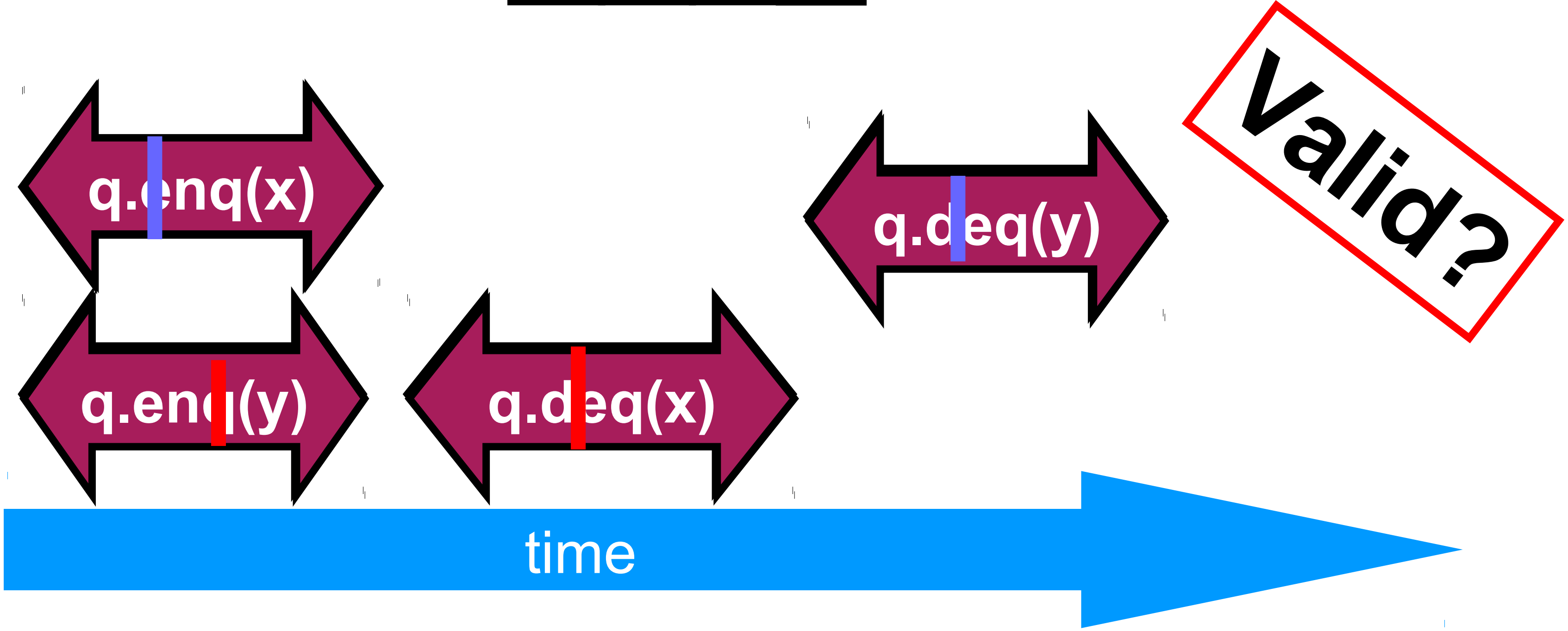
Example

```
void enq(Item x) {
  std::lock_guard<std::mutex> l(lock)
  if((tail+1)%items.size() == head) {
    throw FullException;
  }
  items[tail] = x;
  tail = (tail+1)%items.size();
}
```

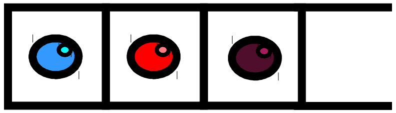
```
Item deq() {
  std::lock_guard<std::mutex> l(lock)
  if(tail == head) {
    throw EmptyException;
  }
  Item item = items[head];
  head = (head+1)%items.size();
}
```

linearization points

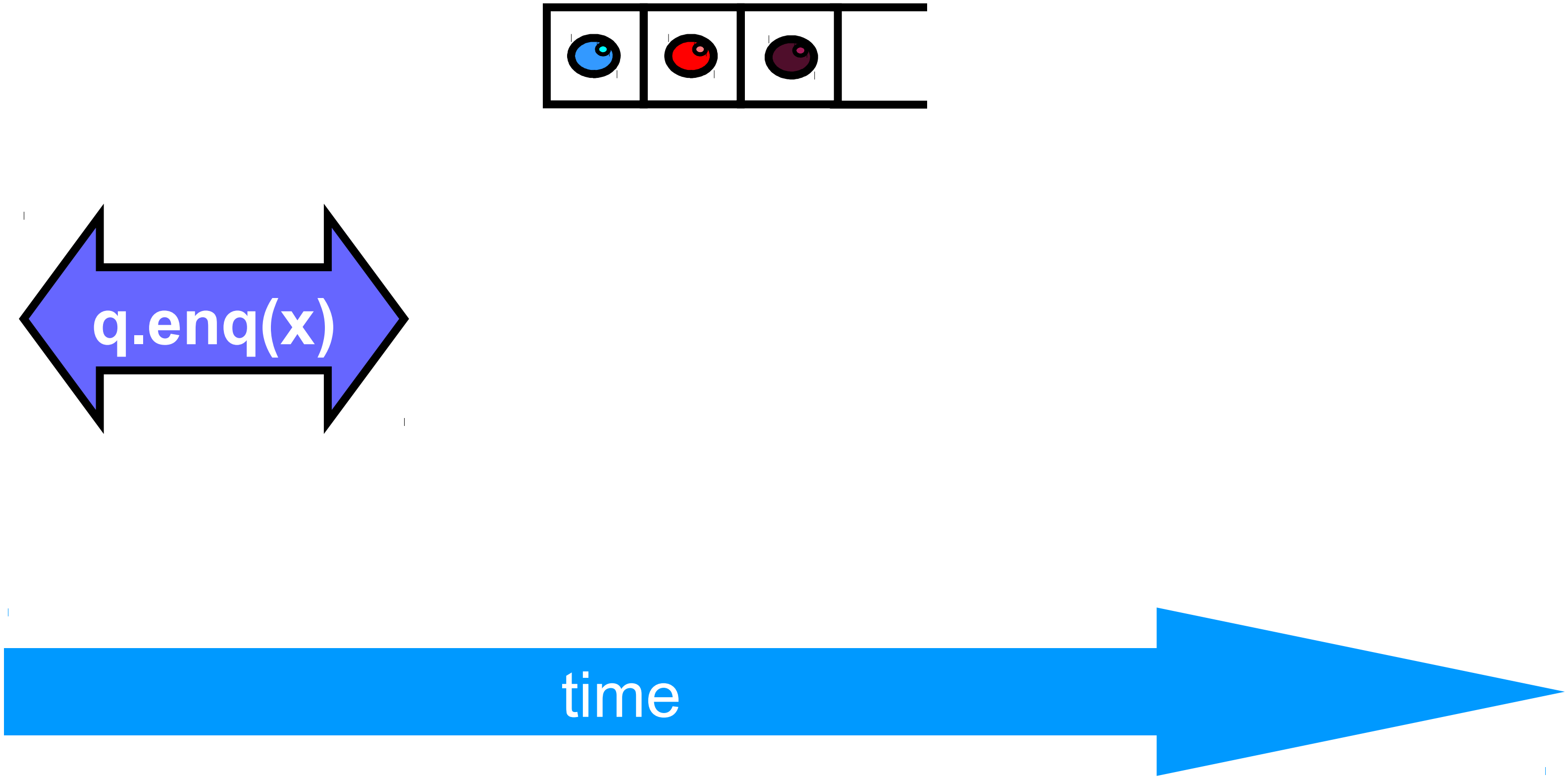




Example 2



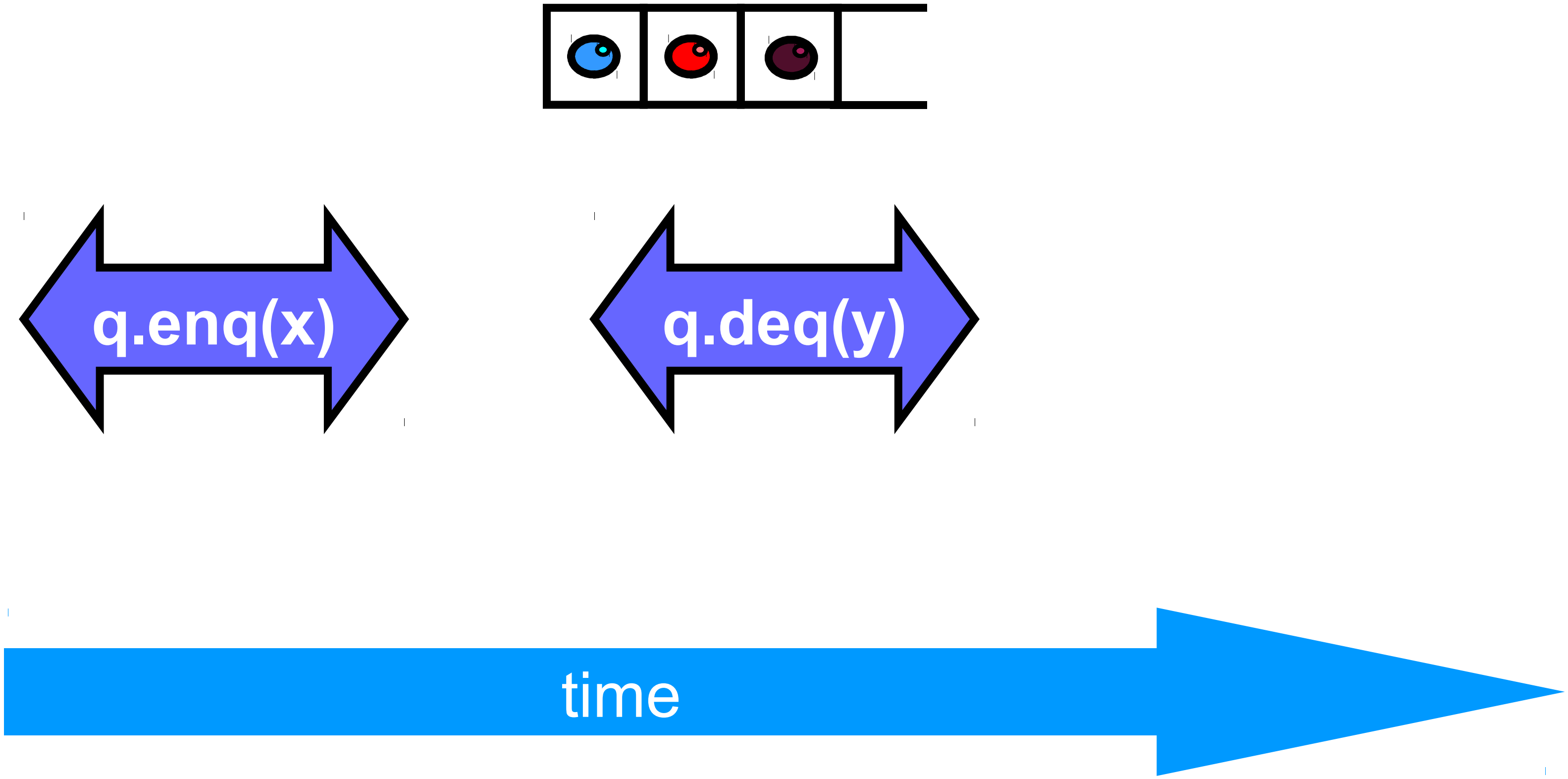
Example 2



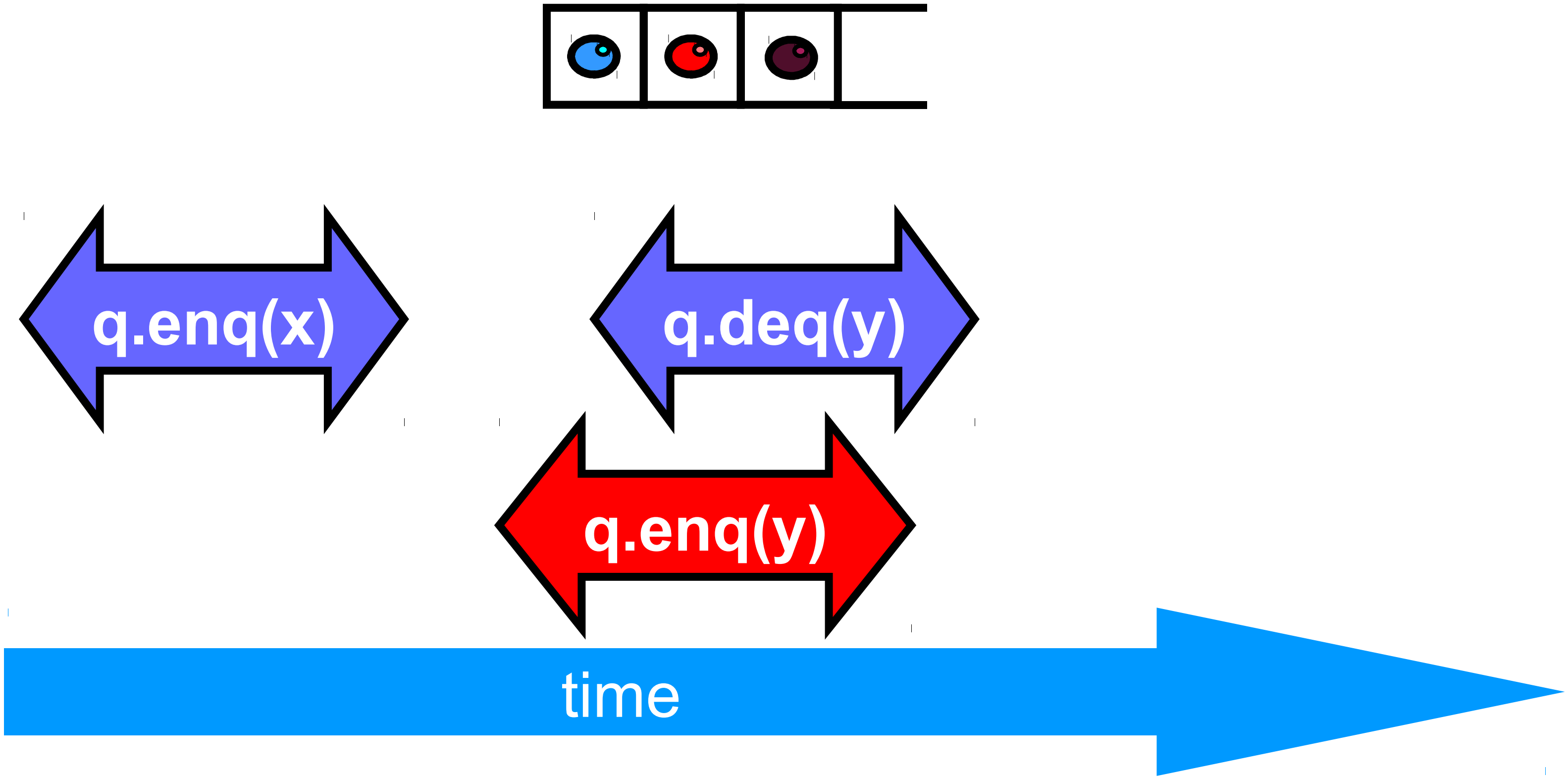
q.enq(x)

time

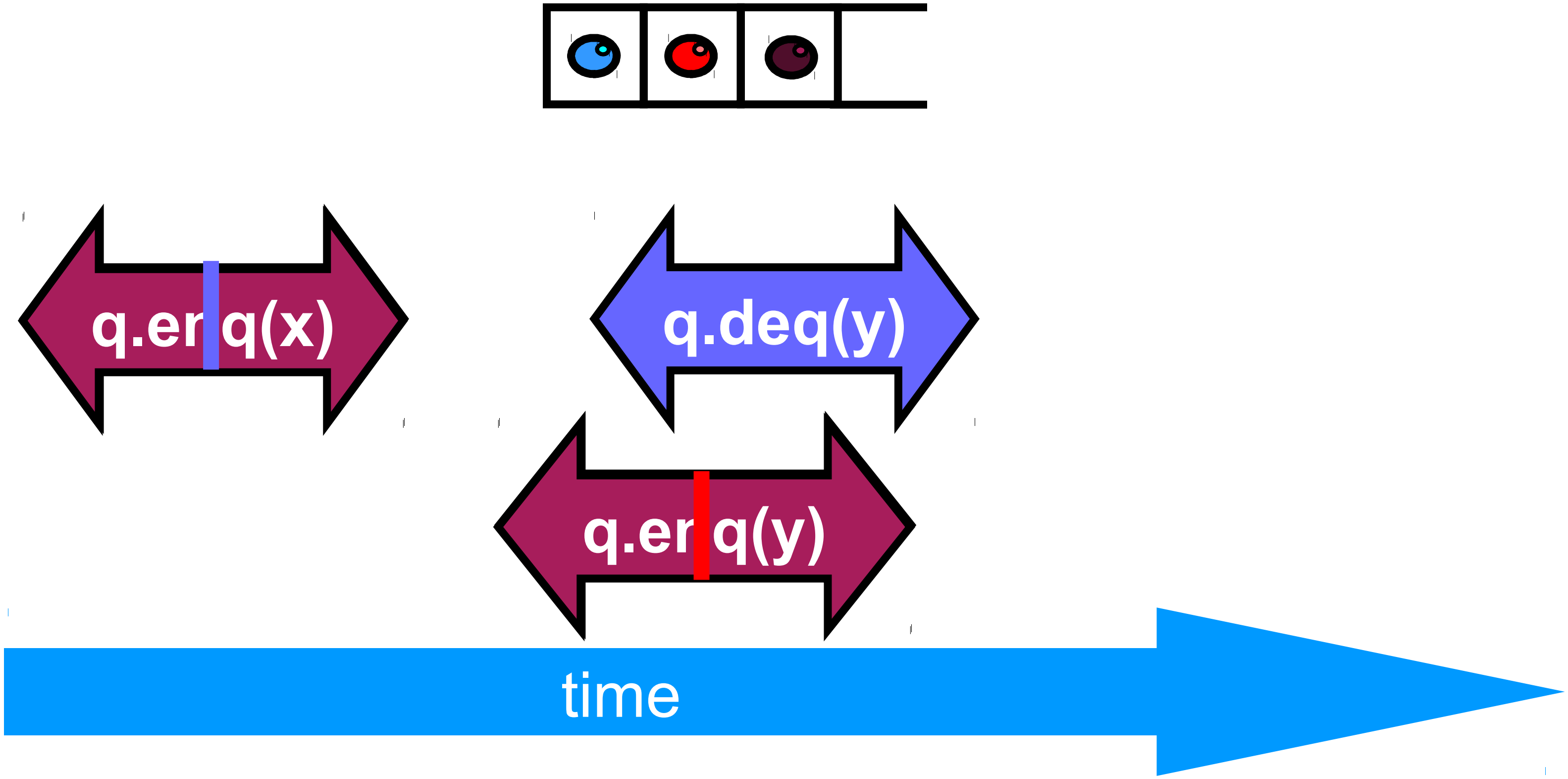
Example 2



Example 2



Example 2



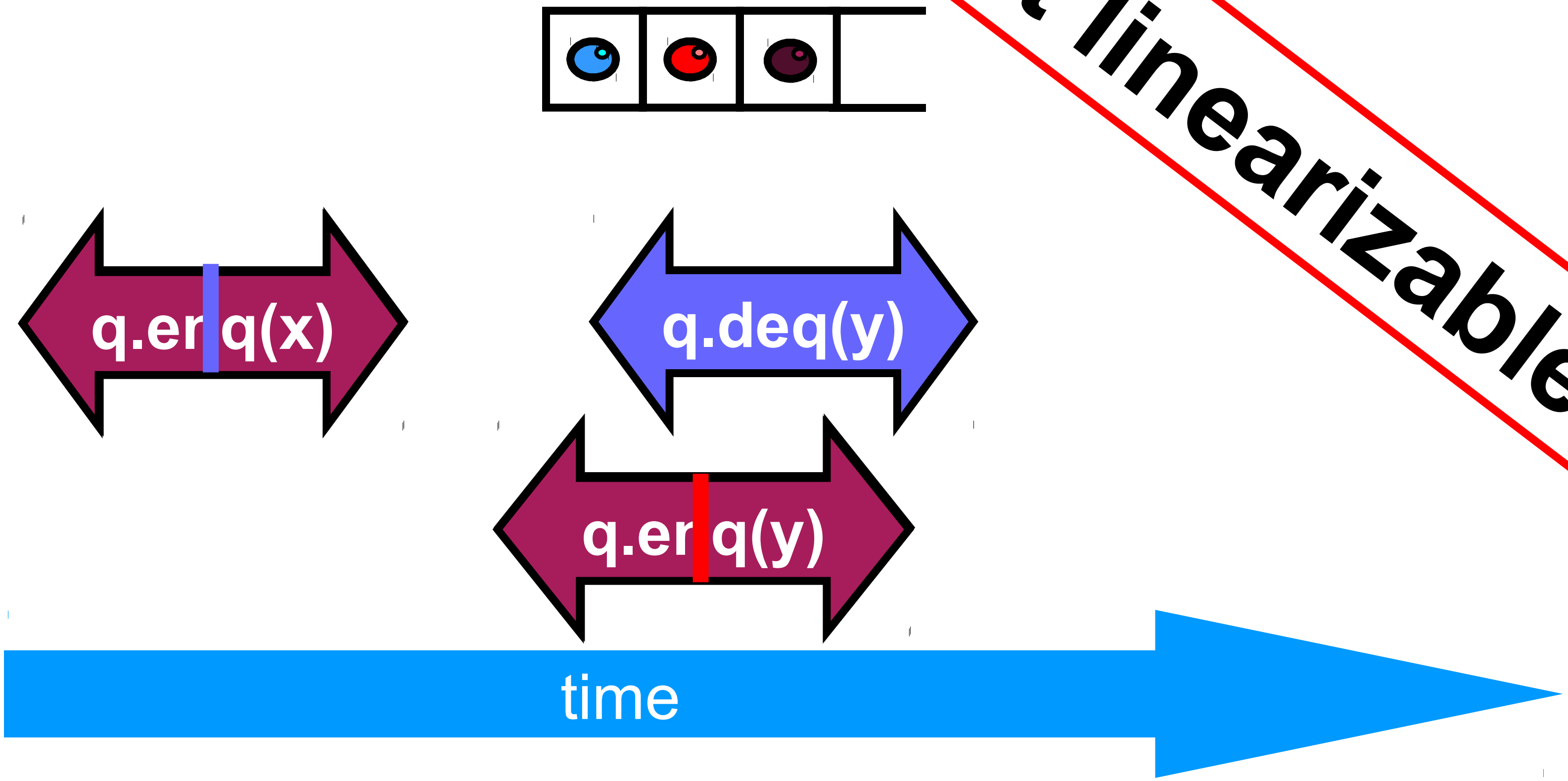
q.enq(x)

q.deq(y)

q.enq(y)

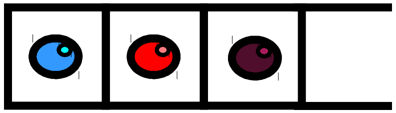
time

Example 2

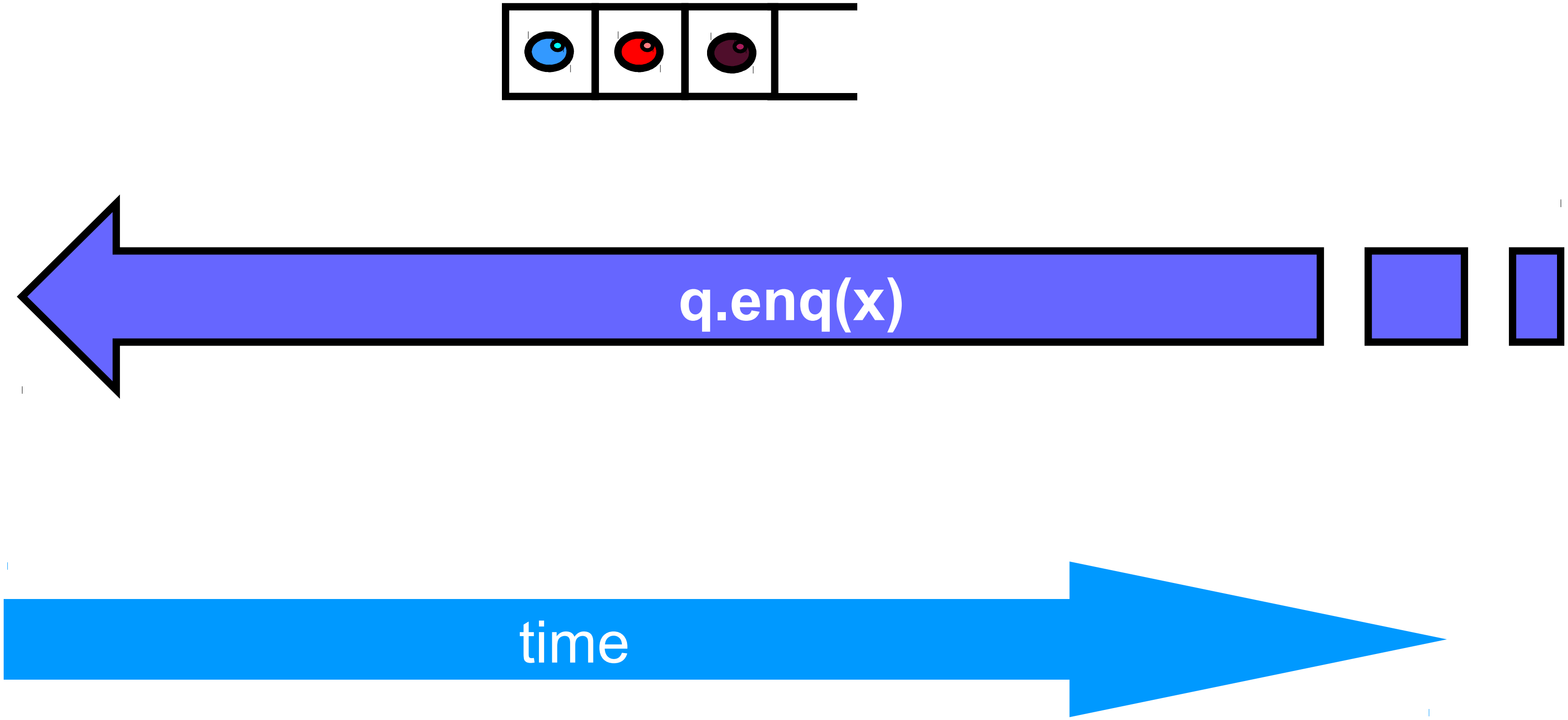


not linearizable

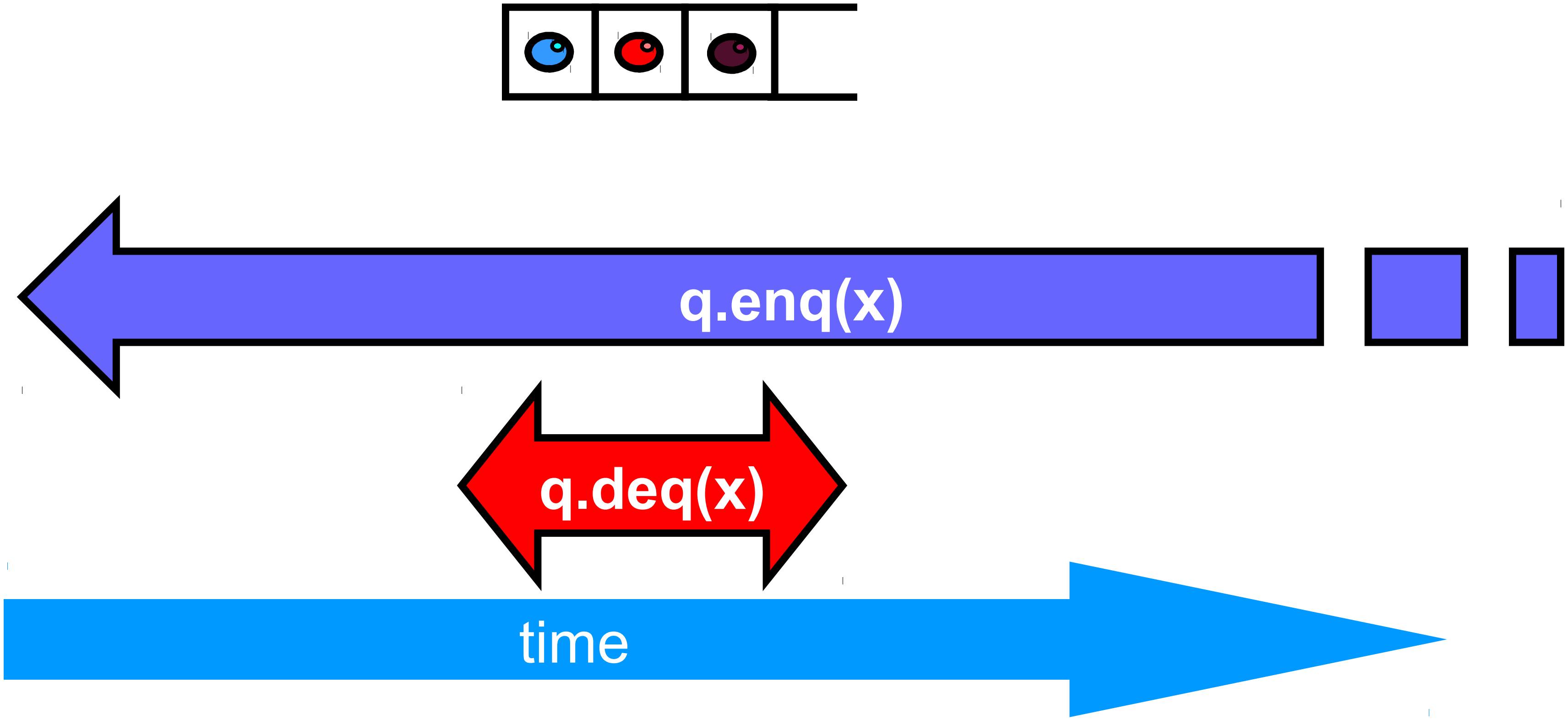
Example 3



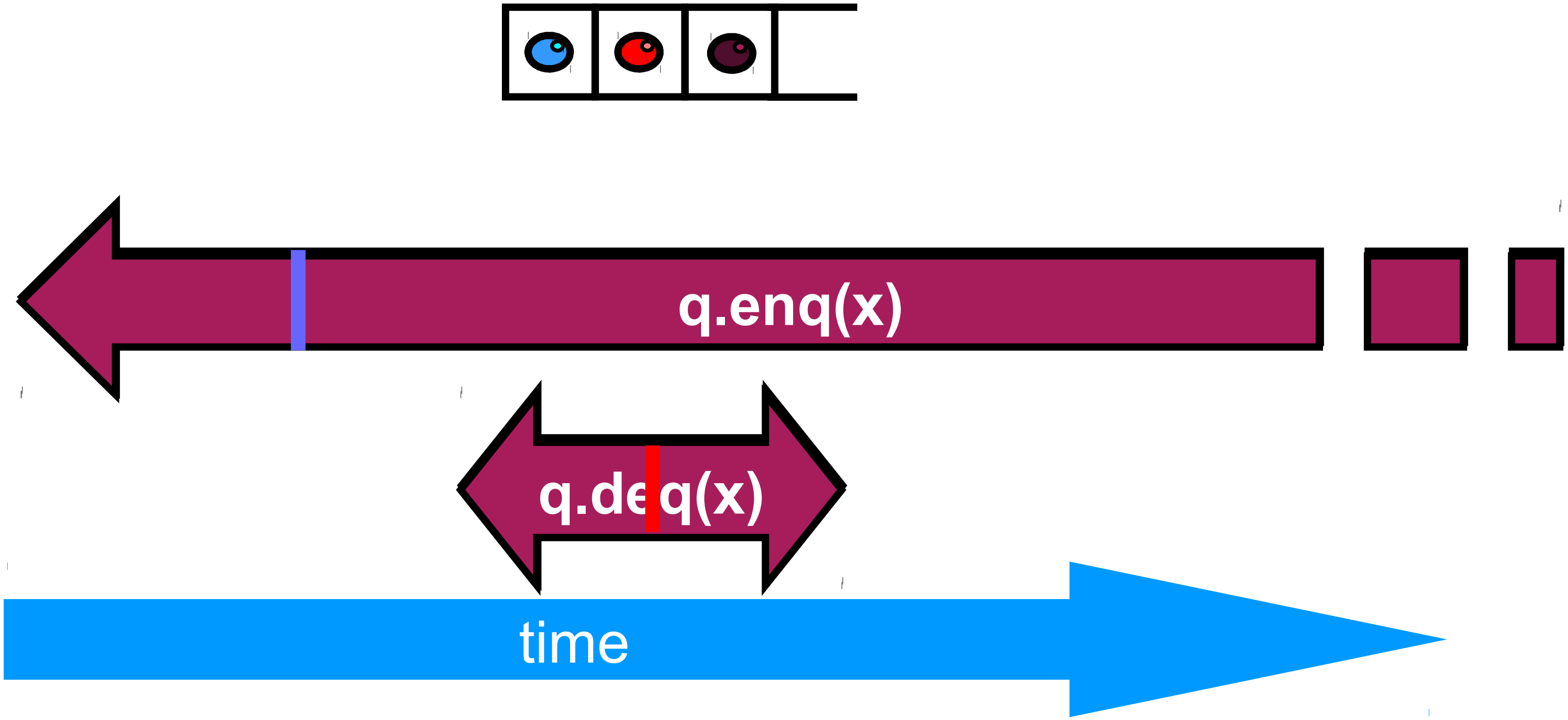
Example 3



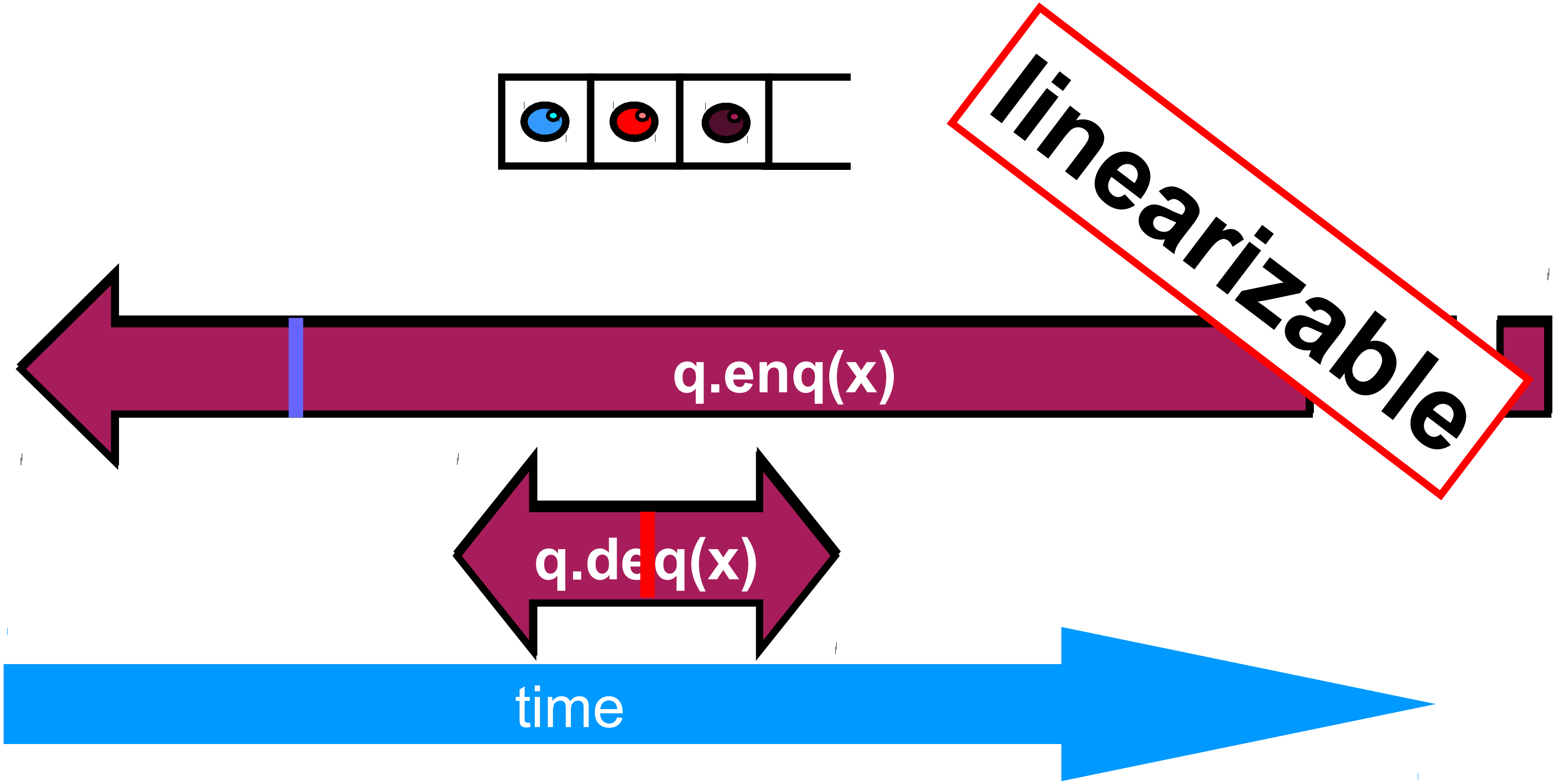
Example 3



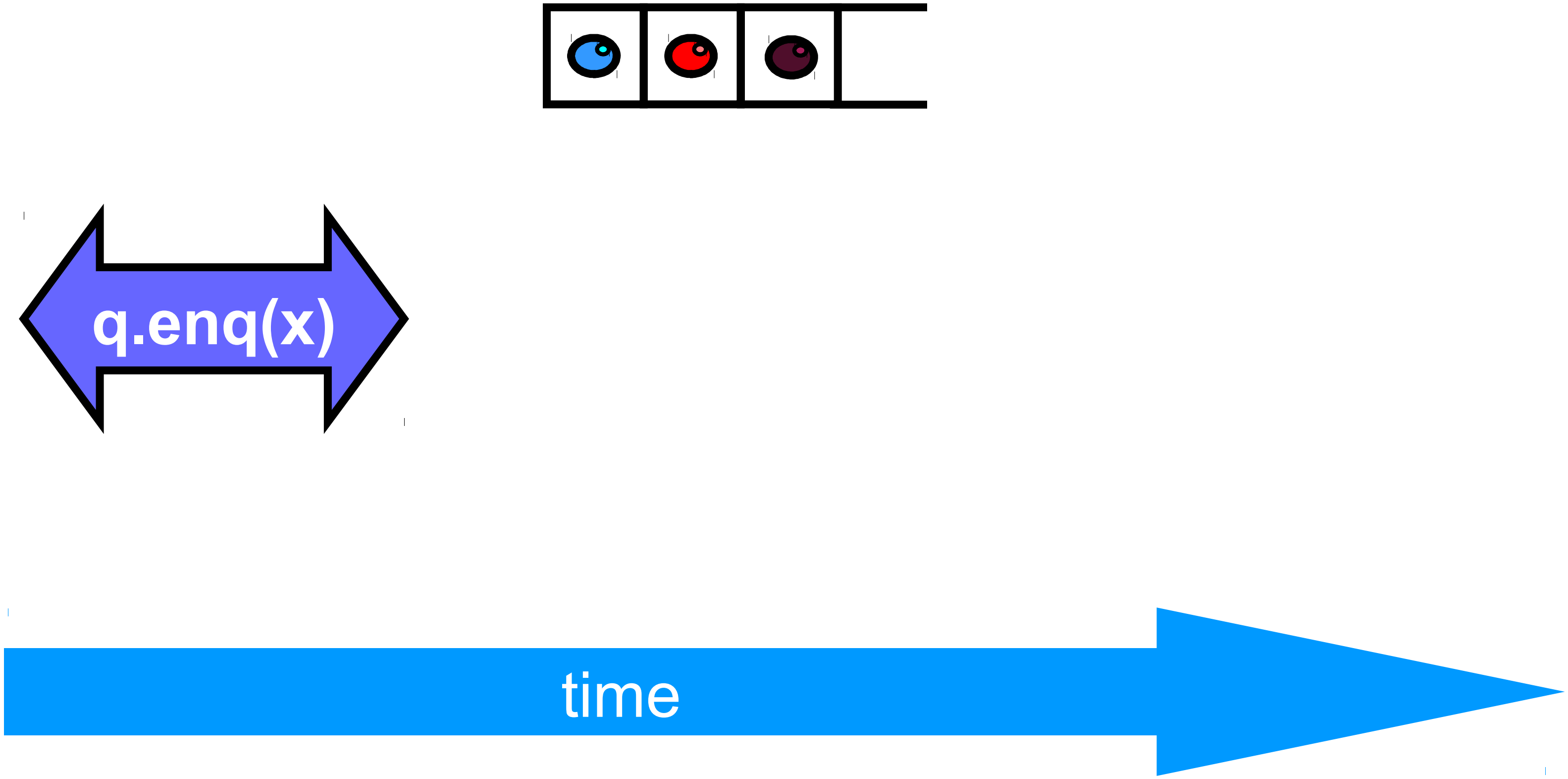
Example 3



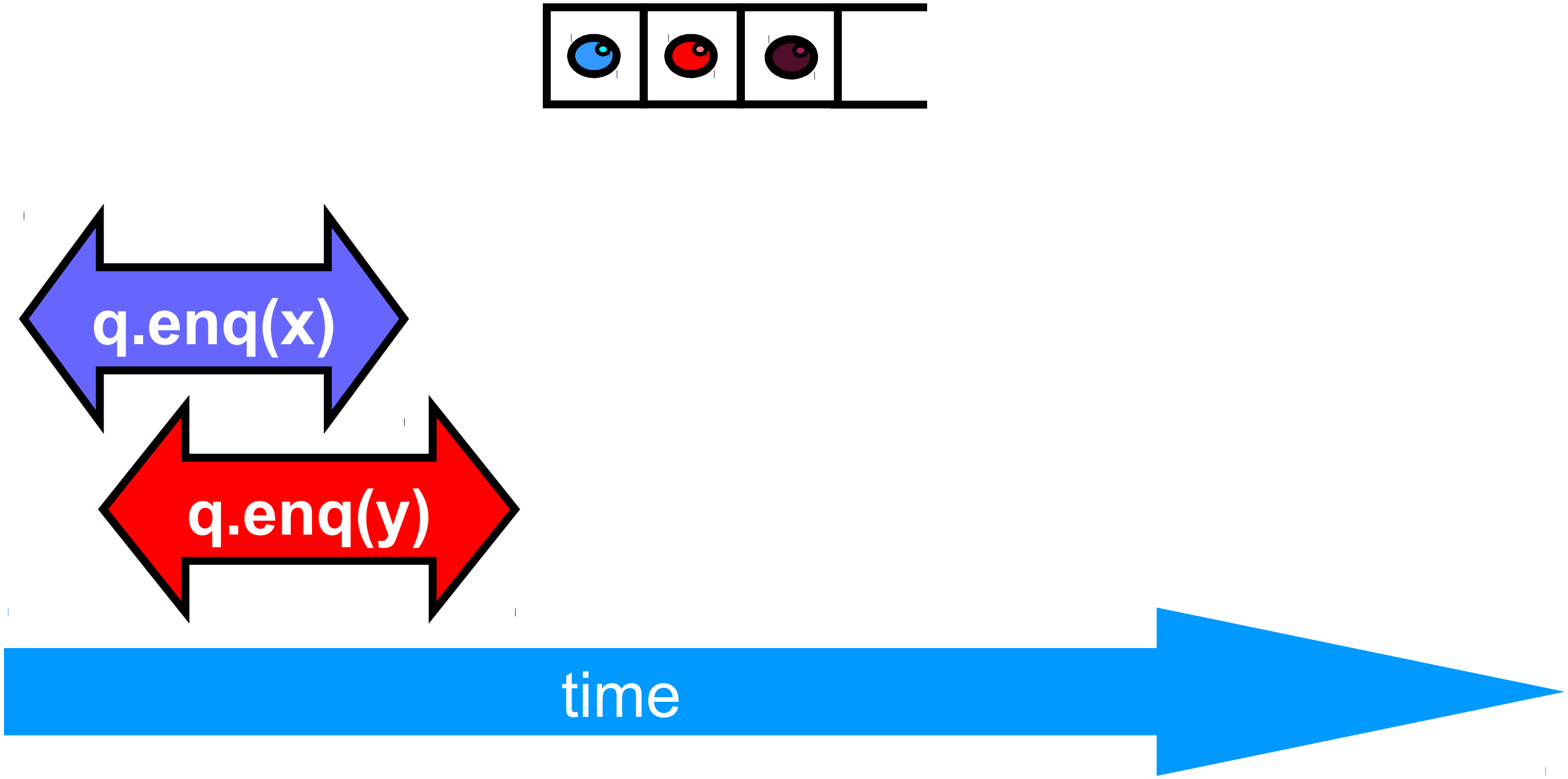
Example 3



Example 4



Example 4

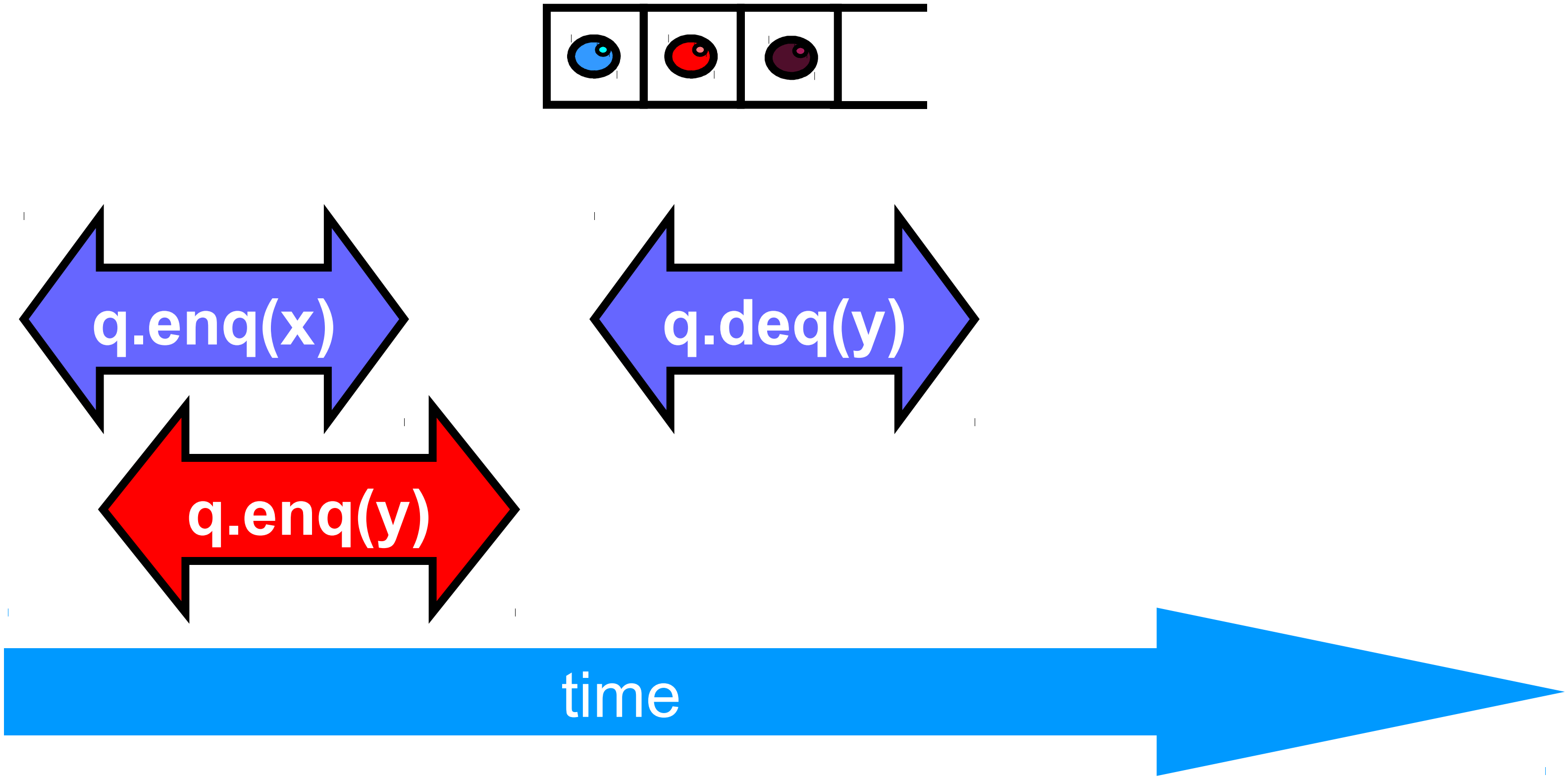


q.enq(x)

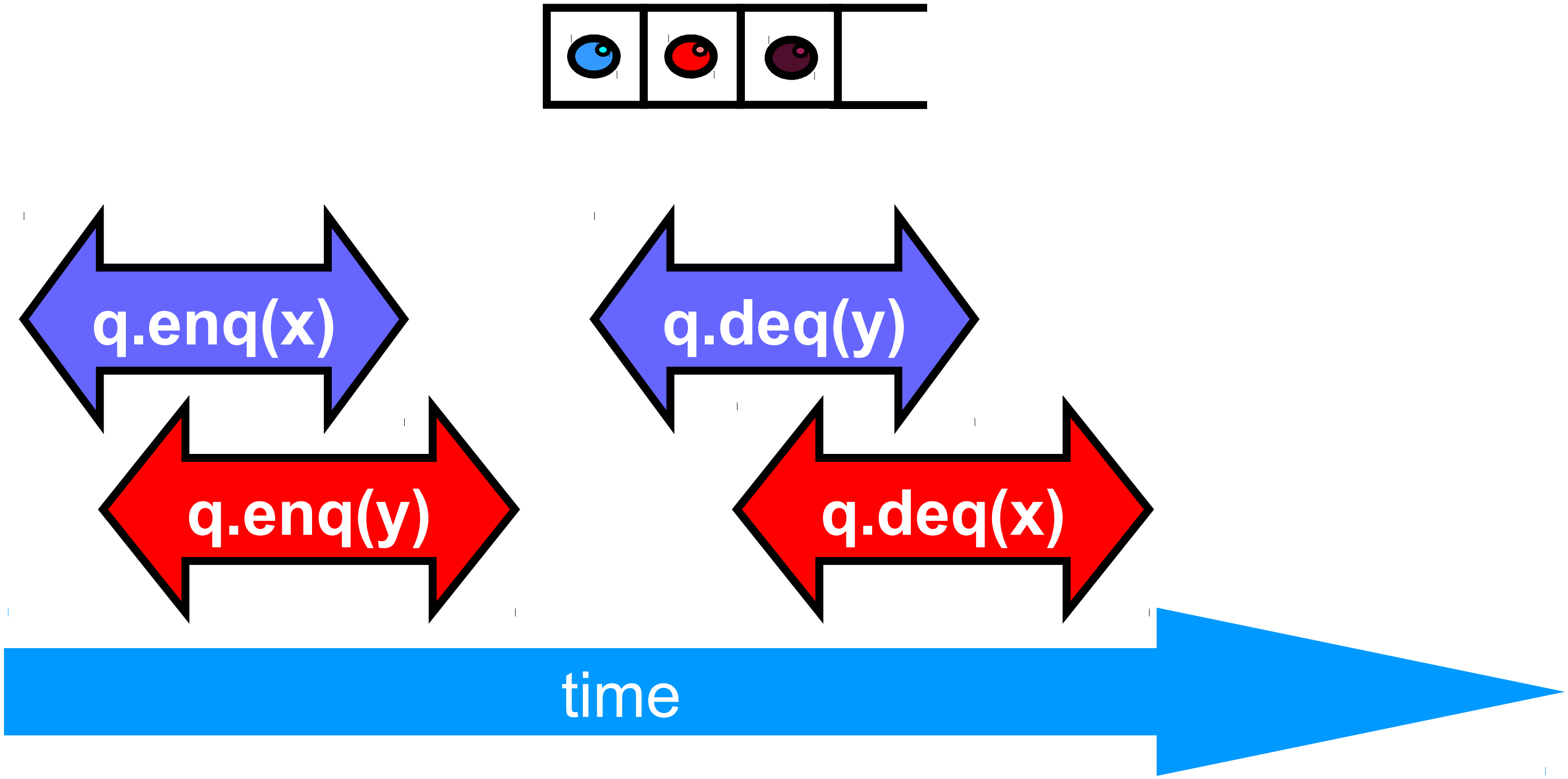
q.enq(y)

time

Example 4



Example 4



q.enq(x)

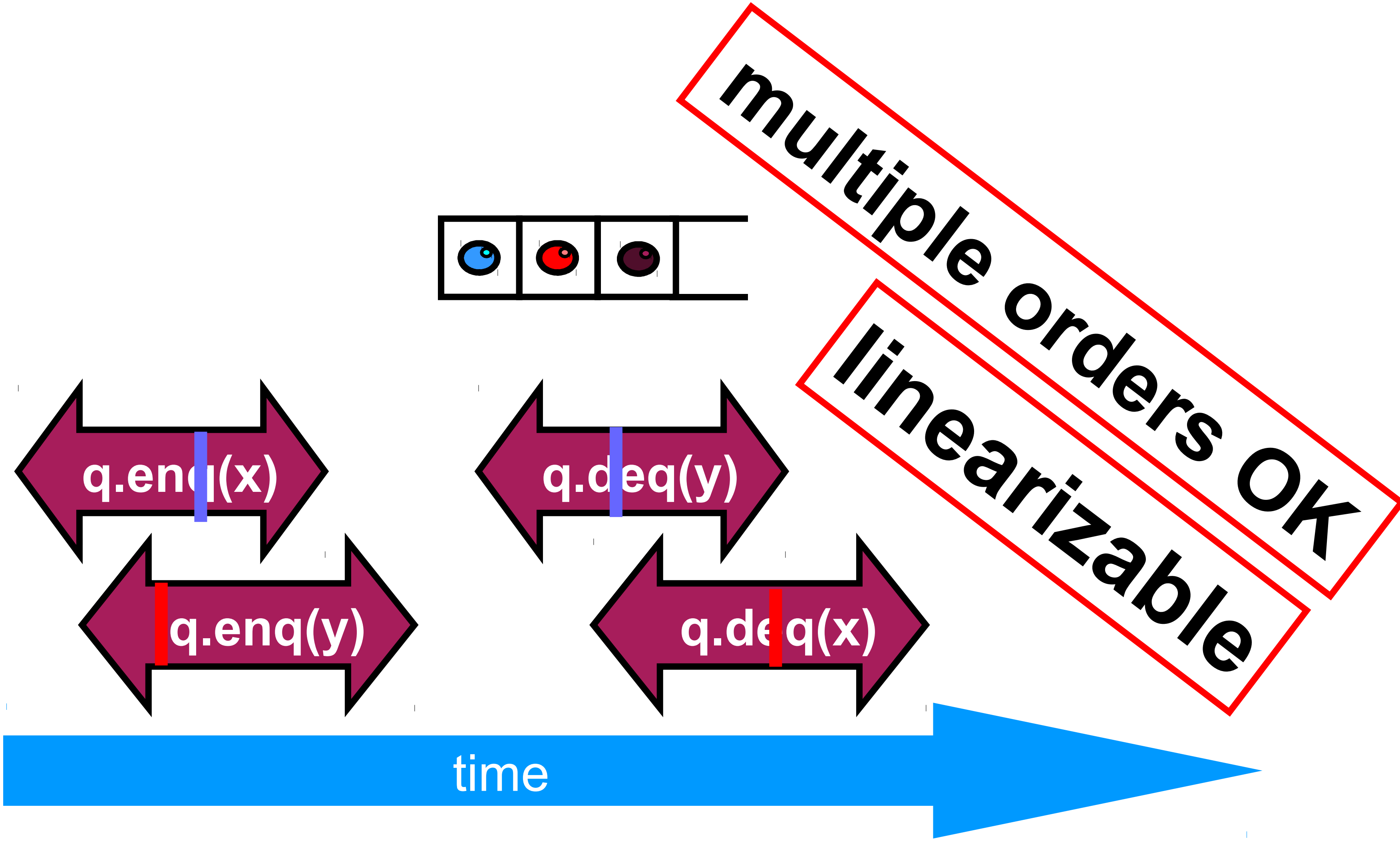
q.enq(y)

q.deq(y)

q.deq(x)

time

Example 4



multiple orders OK

linearizable

Is the lock-free queue linearizable?

- A) Only two threads, one calls only deq() and one calls only enq()?

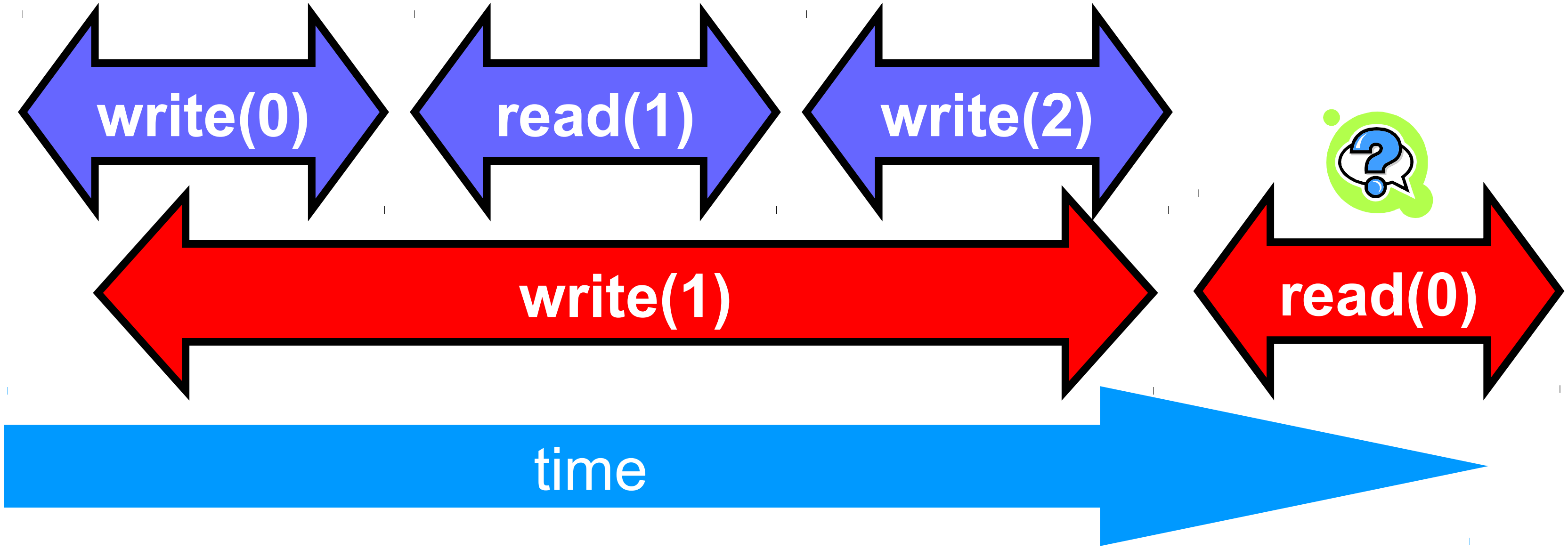
```
 void enq(Item x) {
   if((tail+1)%items.size() == head) {
     throw FullException;
   }
   items[tail] = x;
   tail = (tail+1)%items.size();
}
```

```
 Item deq() {
   if(tail == head) {
     throw EmptyException;
   }
   Item item = items[head];
   head = (head+1)%items.size();
   return item;
}
```

- B) Only two threads but both may call enq() or deq() independently
- C) An arbitrary number of threads, but only one calls enq()
- D) An arbitrary number of threads can call enq() or deq()
- E) If it’s linearizable, where are the linearization points?
  - Remark: typically executions are not constrained, so this is NOT linearizable

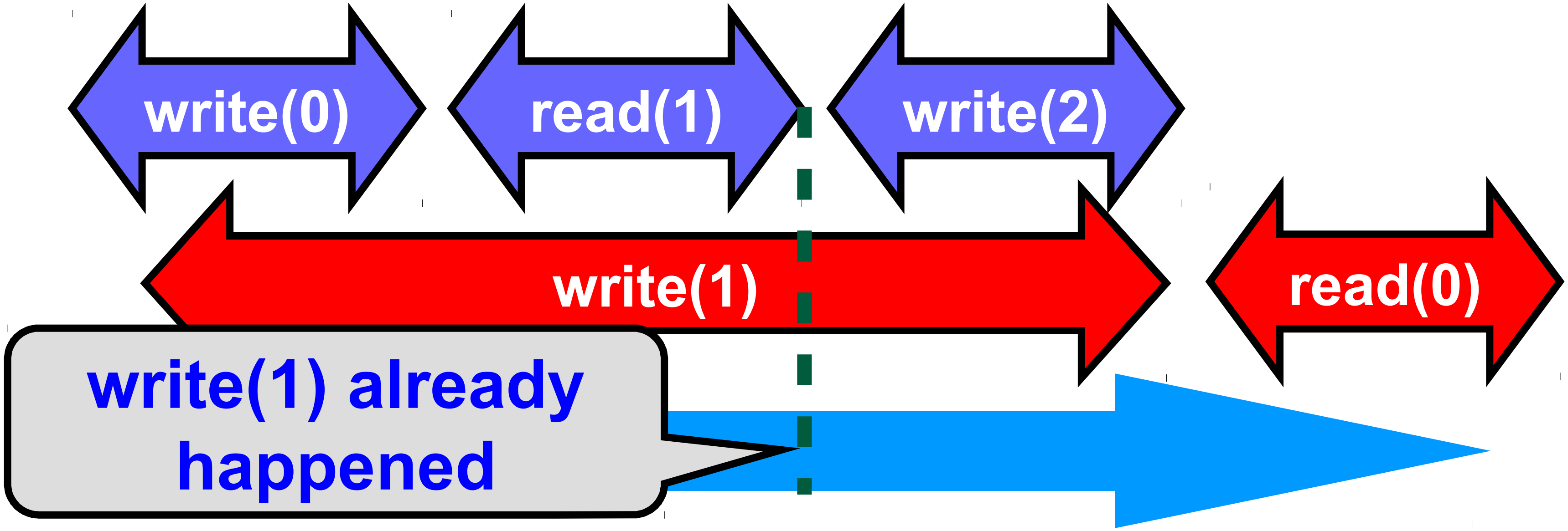
Read/Write Register Example

- Assume atomic update to a single read/write register!



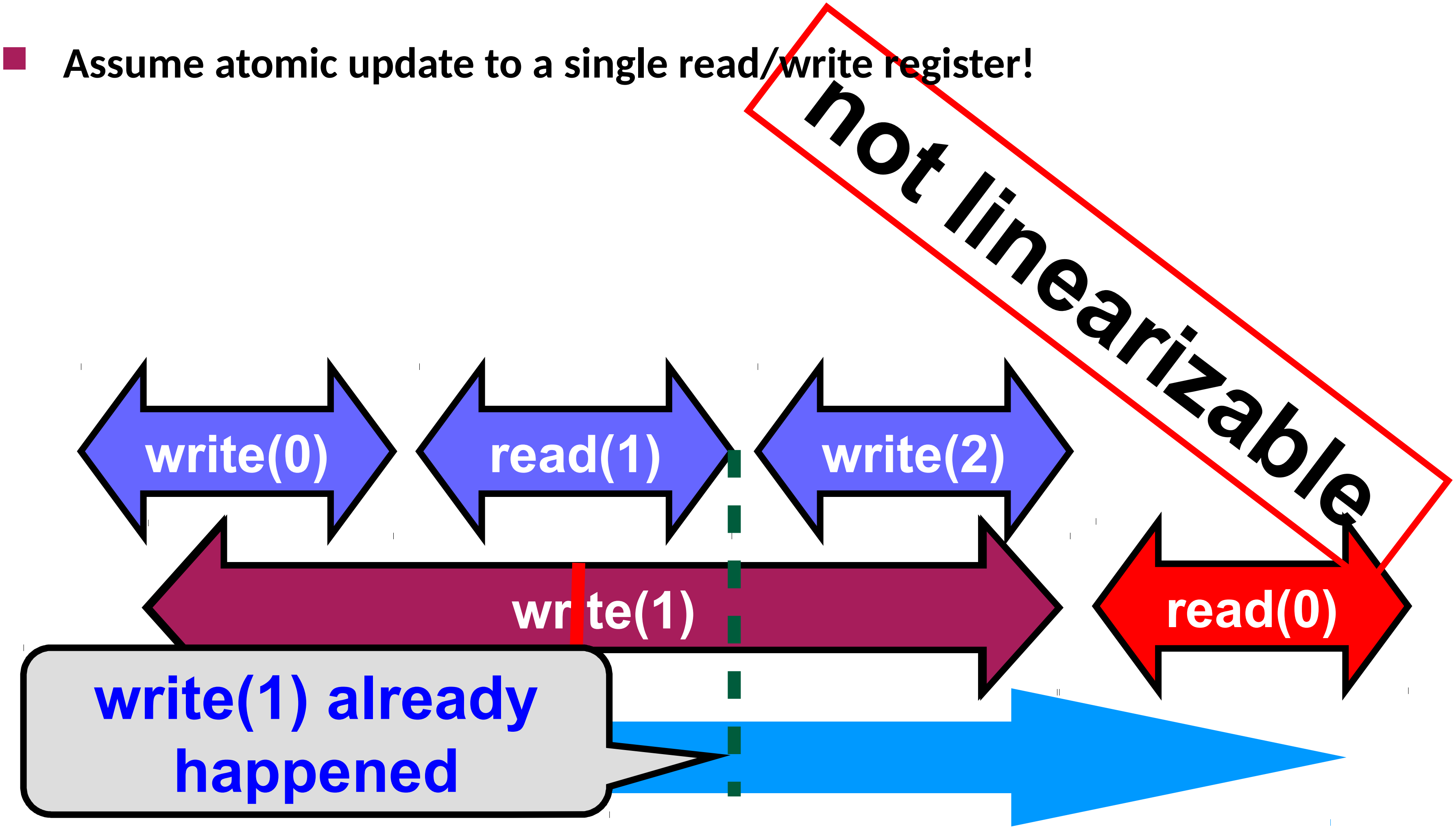
Read/Write Register Example

- Assume atomic update to a single read/write register!



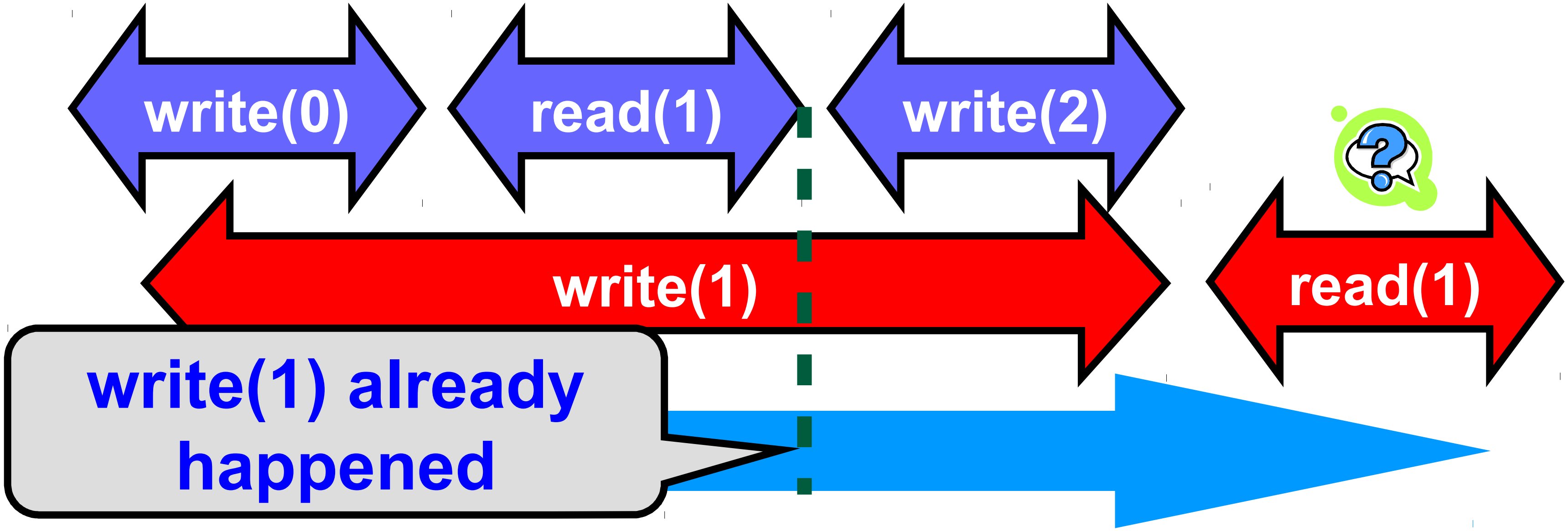
Read/Write Register Example

- Assume atomic update to a single read/write register!



Read/Write Register Example

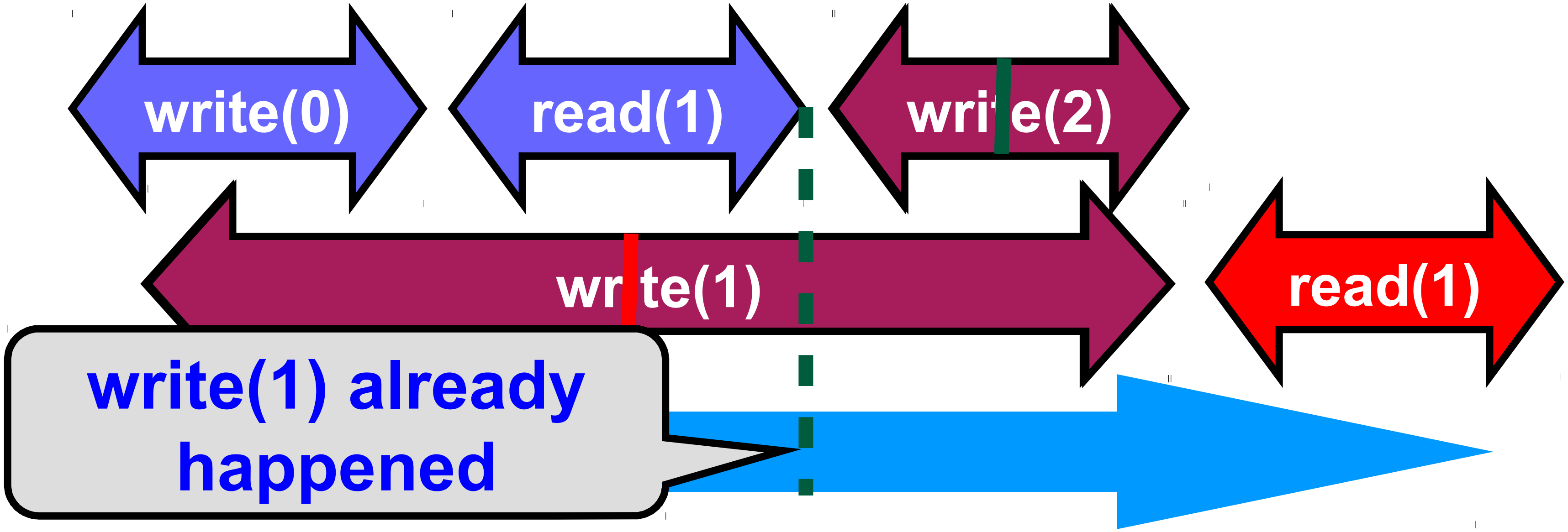
- Assume atomic update to a single read/write register!



write(1) already happened

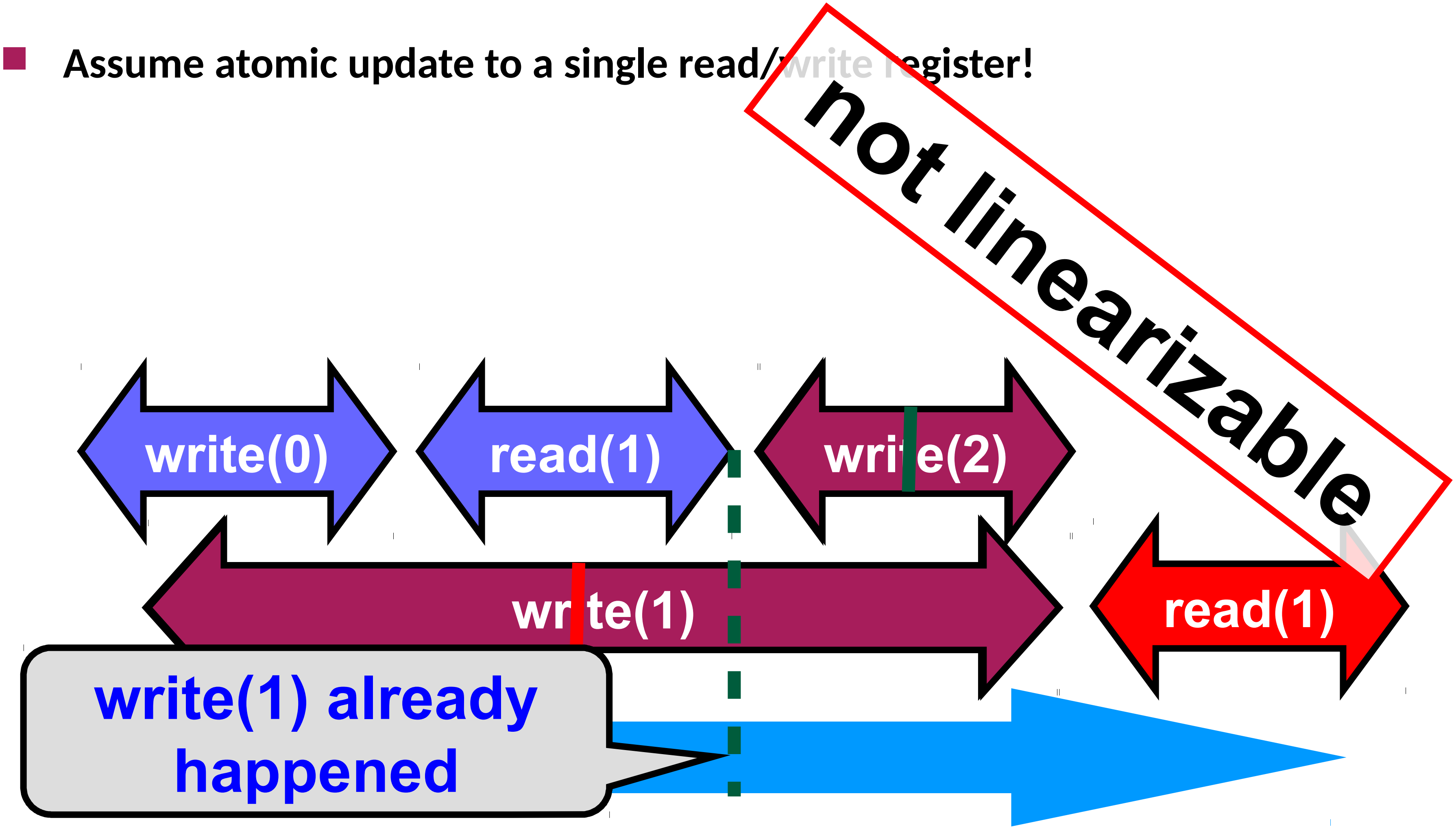
Read/Write Register Example

- Assume atomic update to a single read/write register!



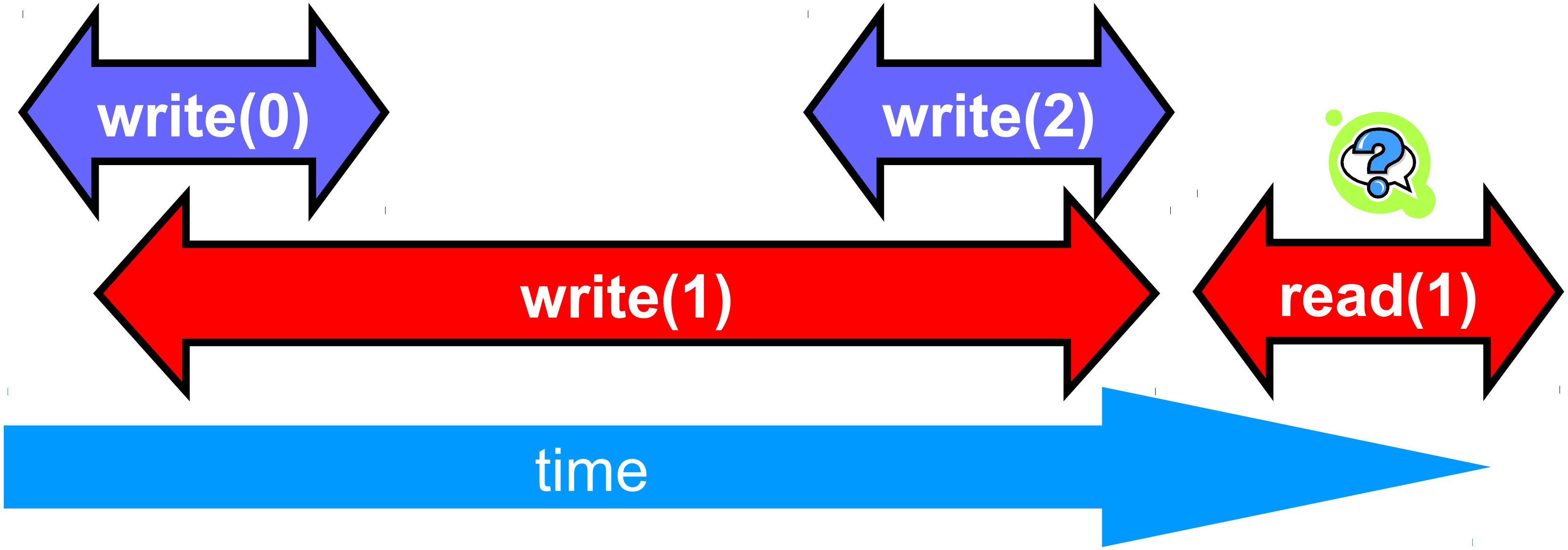
Read/Write Register Example

- Assume atomic update to a single read/write register!



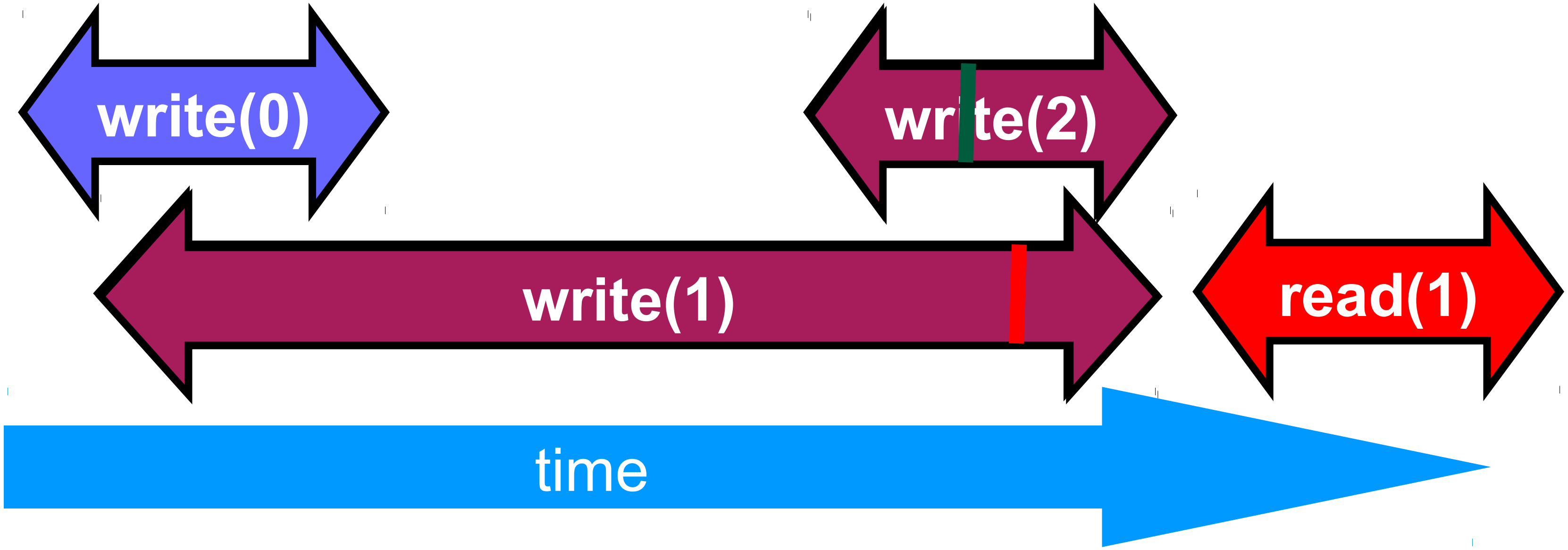
Read/Write Register Example

- Assume atomic update to a single read/write register!



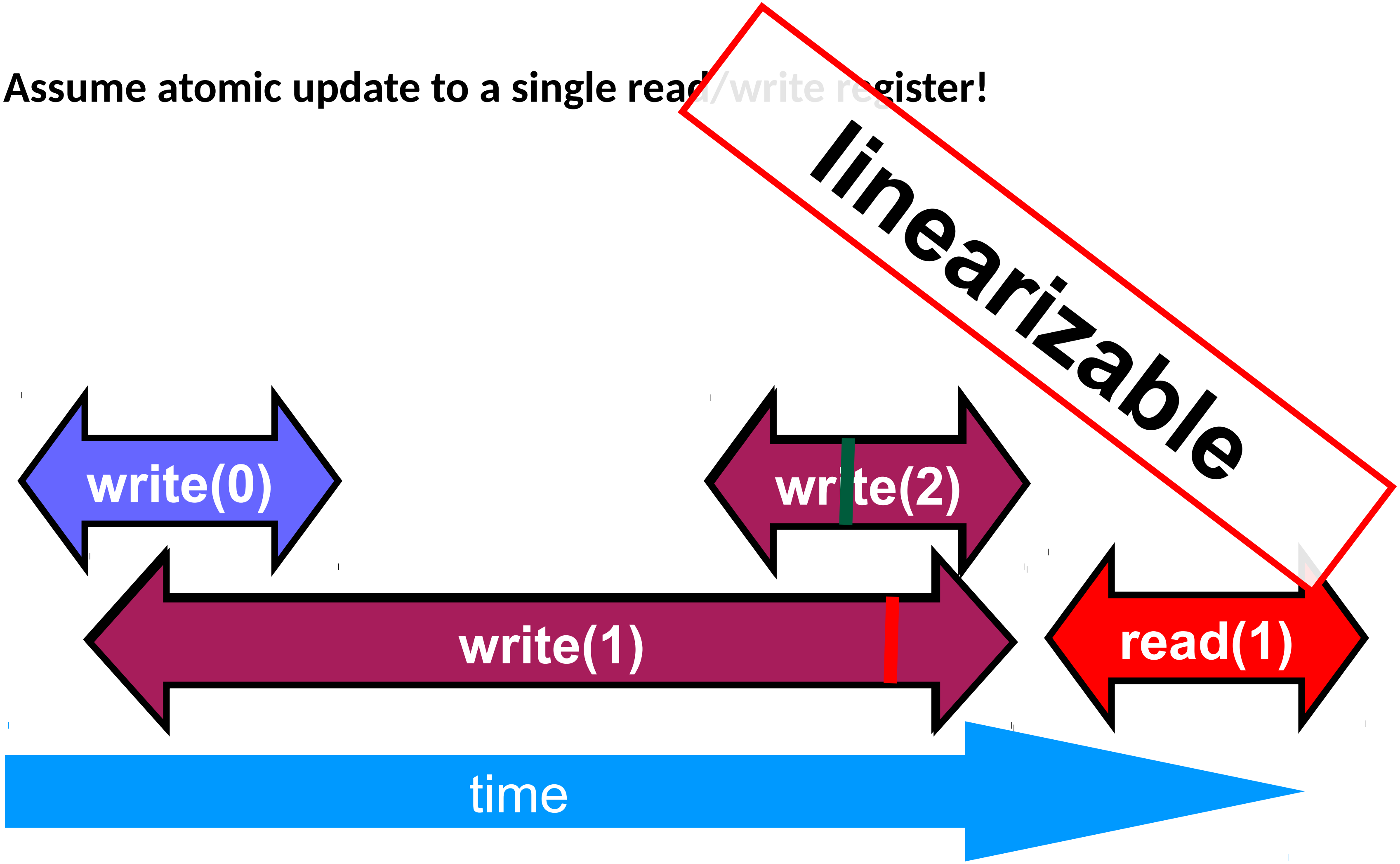
Read/Write Register Example

- Assume atomic update to a single read/write register!

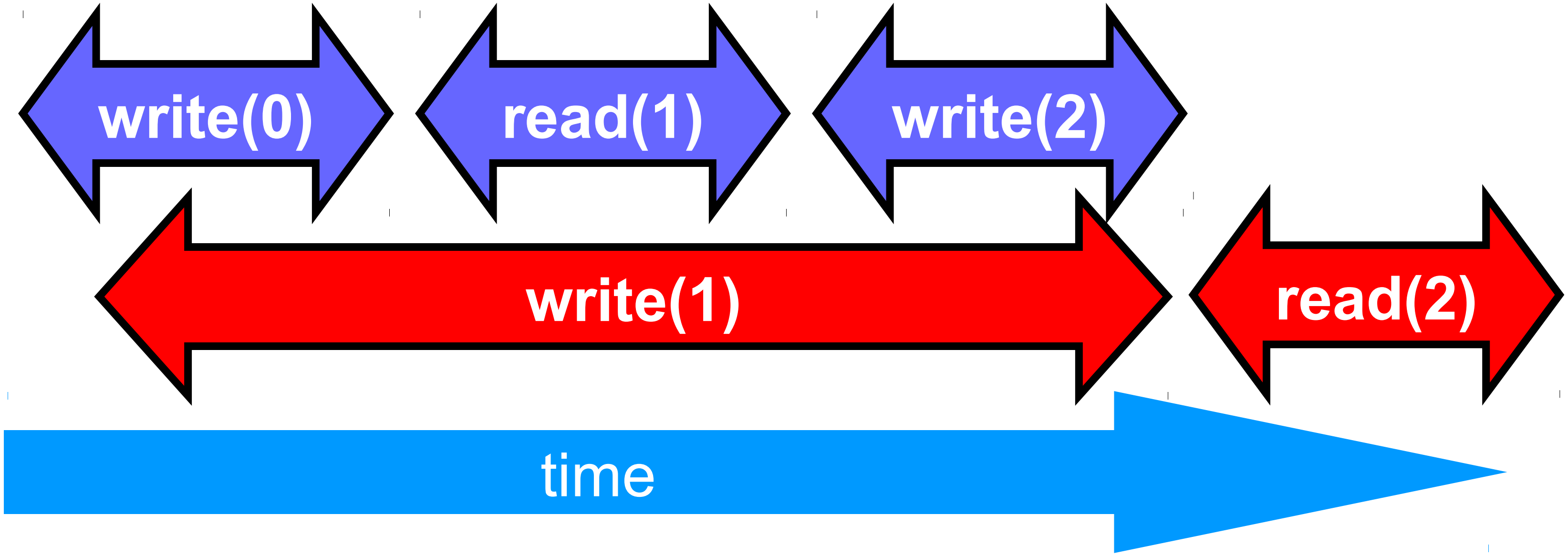


Read/Write Register Example

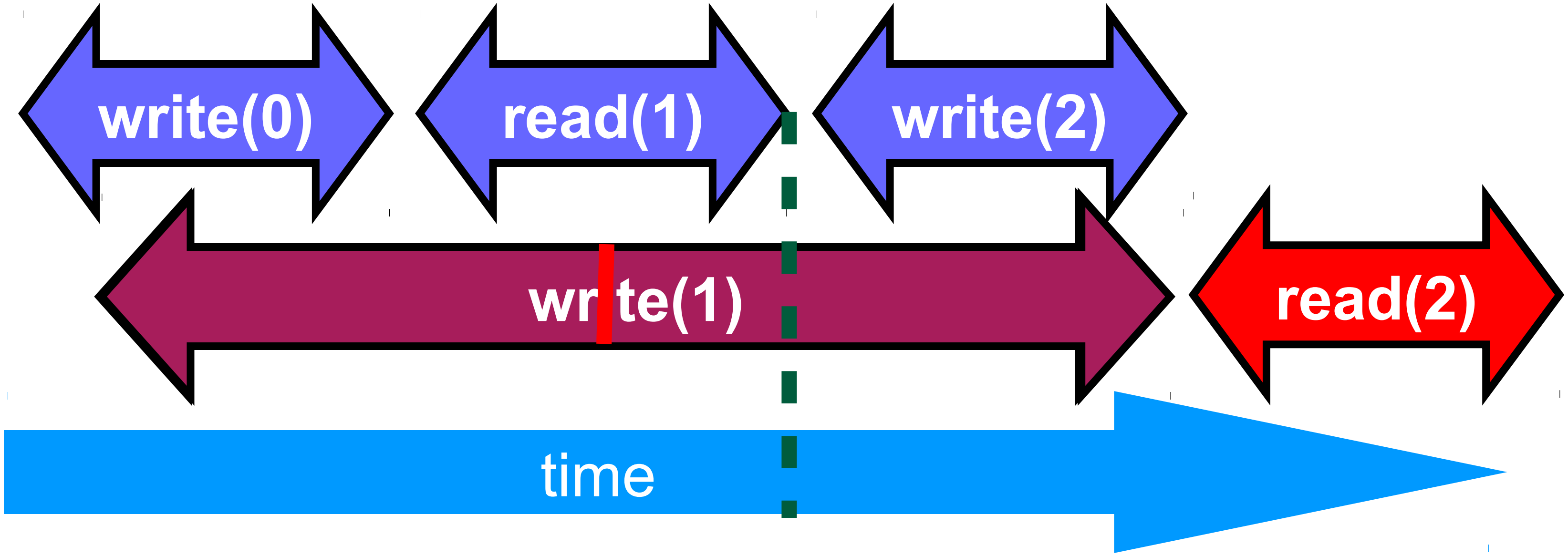
- Assume atomic update to a single read/write register!



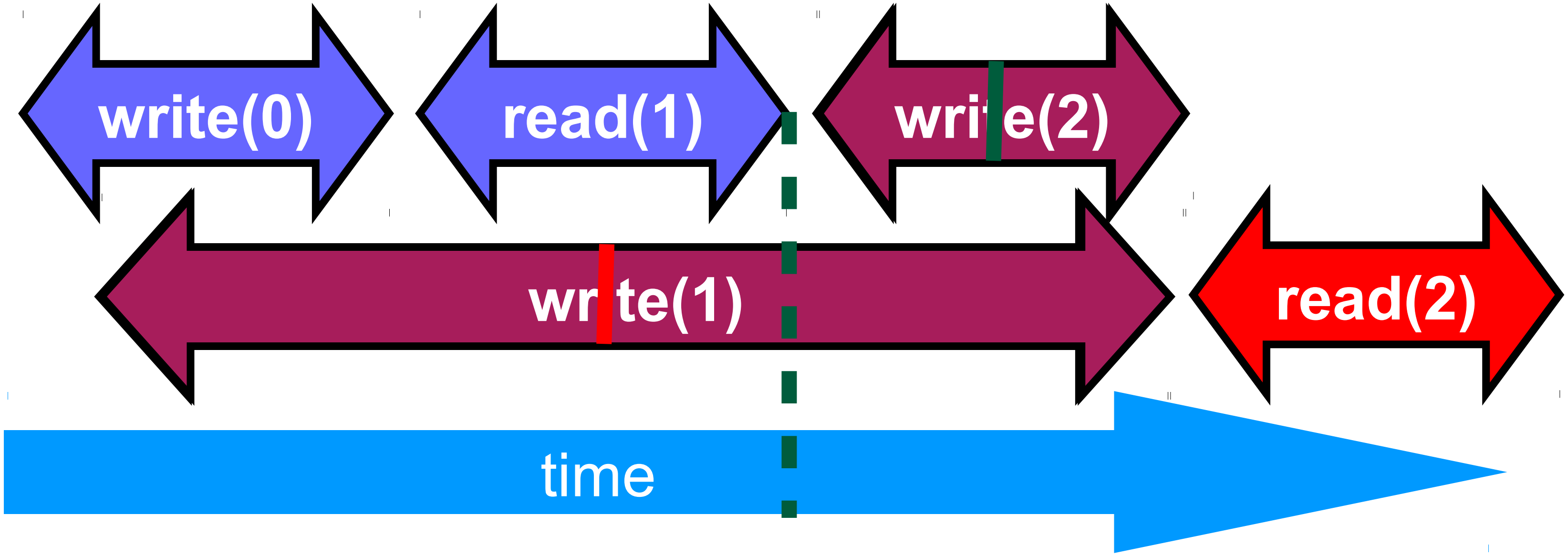
Read/Write Register Example



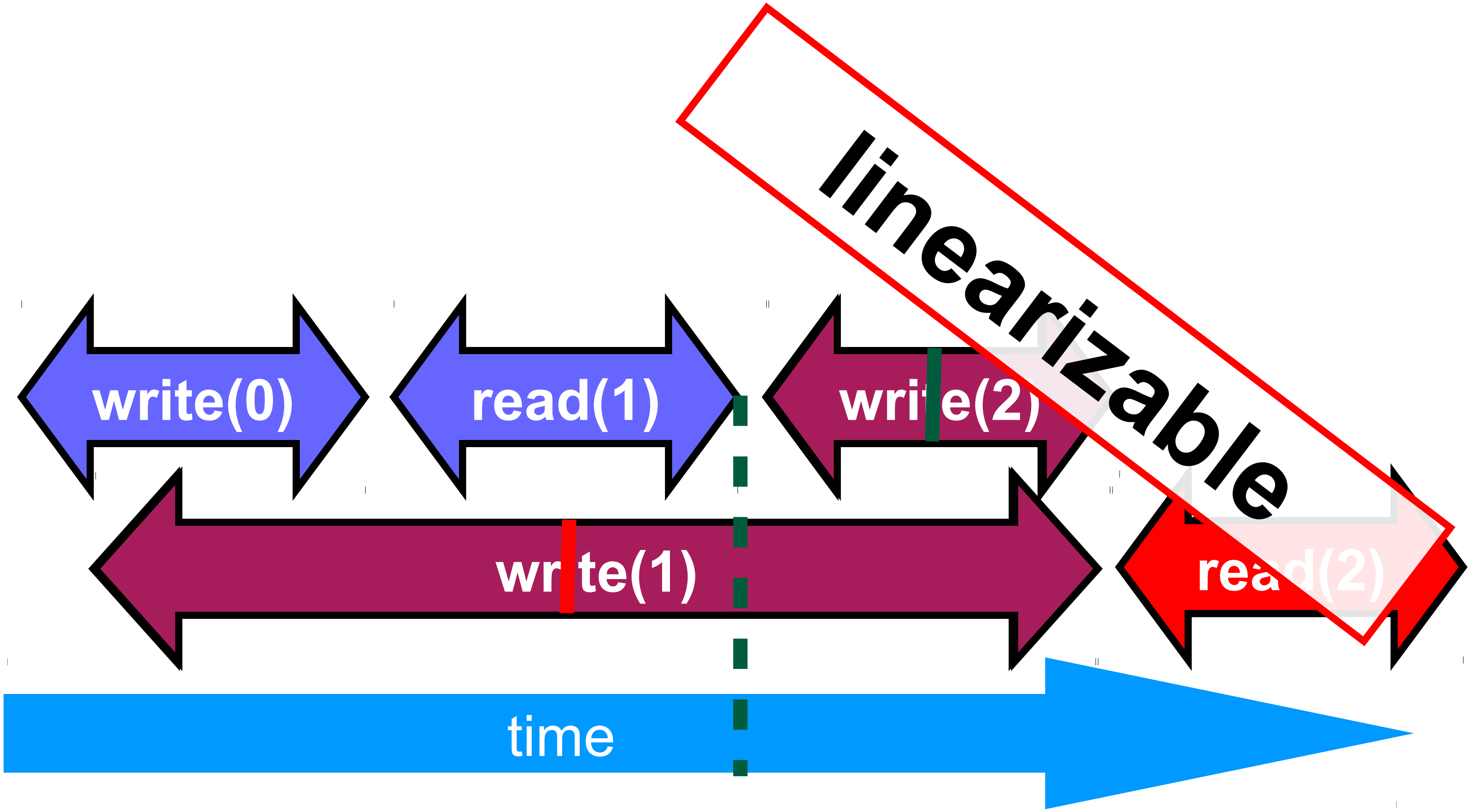
Read/Write Register Example



Read/Write Register Example



Read/Write Register Example



About Executions

- **Why?**
  - Can’t we specify the linearization point of each operation statically without describing an execution?

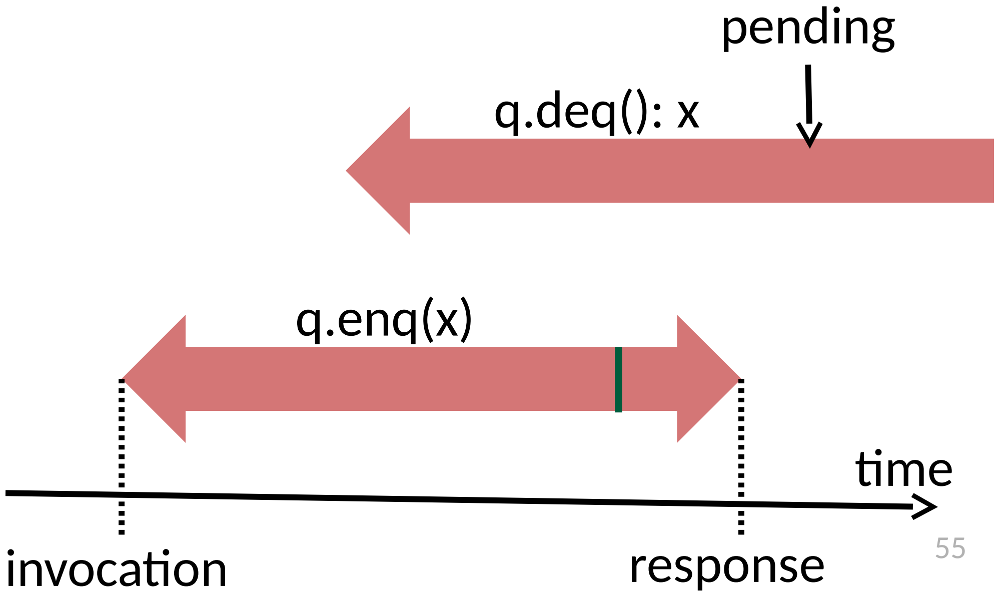
- **Not always**
  - In some cases, the linearization point depends on the execution

    Imagine a “check if one should lock” (not recommended!)

- **Define a formal model for executions!**

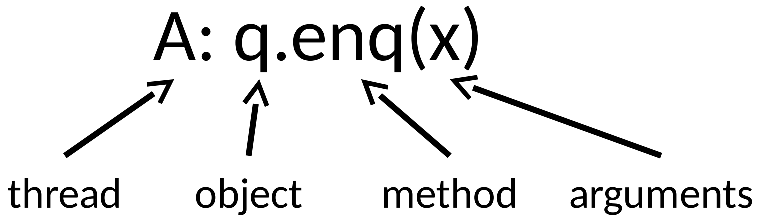
Properties of concurrent method executions

- **Method executions take time**
  - May overlap
- **Method execution = operation**
  - Defined by invocation and response events
- **Duration of method call**
  - Interval between the events



Formalization - Notation

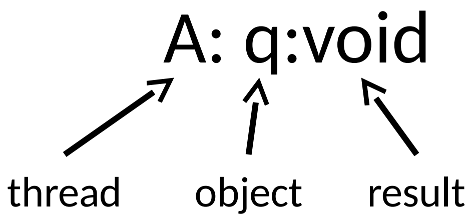
- **Invocation**



A: q.enq(x)

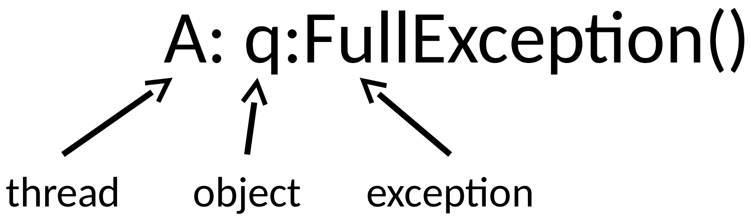
thread object method arguments

- **Response**



A: q:void

thread object result



A: q:FullException()

thread object exception

  - Question: why is the method name not needed in the response?

    Method is implicit (correctness criterion)!

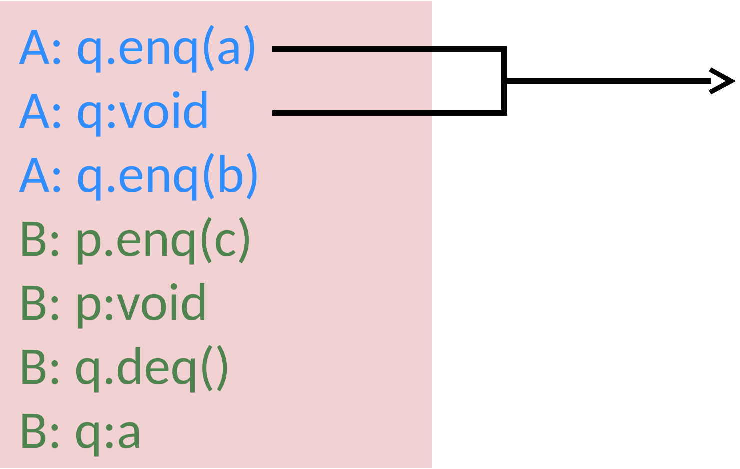
Concurrency

- **A concurrent system consists of a collection of sequential threads P_i**
- **Threads communicate via shared objects**

  For now!

History

- **Describes an execution**
  - Sequence of invocations and responses
  - H=



A: q.enq(a)
A: q:void
A: q.enq(b)
B: p.enq(c)
B: p:void
B: q.deq()
B: q:a

Invocation and response match if

- thread names are the same
- objects are the same

Remember: Method name is implicit!

Side Question: Is this history linearizable?

Projections on Threads

- **Threads subhistory H|P (“H at P”)**
  - Subsequences of all events in H whose thread name is P

H=	H|A=	H|B=
A: q.enq(a)	A: q.enq(a)	
A: q:void	A: q:void	
A: q.enq(b)	A: q.enq(b)	
B: p.enq(c)		B: p.enq(c)
B: p:void		B: p:void
B: q.deq()		B: q.deq()
B: q:a		B: q:a

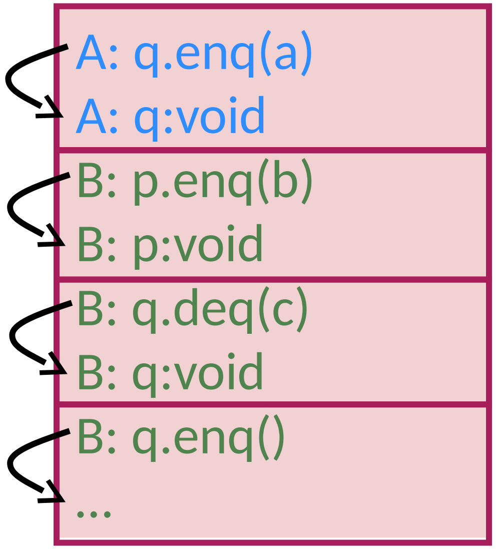
Projections on Objects

- **Objects subhistory H|o (“H at o”)**
  - Subsequence of all events in H whose object name is o

H=	H|p=	H|q=
A: q.enq(a)		A: q.enq(a)
A: q:void		A: q:void
A: q.enq(b)		A: q.enq(b)
B: p.enq(c)	B: p.enq(c)	
B: p:void	B: p:void	
B: q.deq()		B: q.deq()
B: q:a		B: q:a

Sequential Histories

- **A history H is sequential if**



A: q.enq(a)
A: q:void
B: p.enq(b)
B: p:void
B: q.deq(c)
B: q:void
B: q.enq()
...

- The first event of H is an invocation
- Each invocation (except possibly the last) is immediately followed by a matching response
- Each response is immediately followed by an invocation

Method calls of different threads do not interleave

- **A history H is concurrent if**
  - It is not sequential

Well-formed histories

- **Per-thread projections must be sequential**

a history is sequential if

- The first event of H is an invocation
- Each invocation (except possibly the last) is immediately followed by a matching response
- Each response is immediately followed by an invocation

H=

A: q.enq(x)
B: p.enq(y)
B: p:void
B: q.deq()
A: q:void
B: q:x

H|A=

A: q.enq(x)
A: q:void

H|B=

B: p.enq(y)
B: p:void
B: q.deq()
B: q:x

Equivalent histories

- **Per-thread projections must be the same**

H=

```
A: q.enq(x)
B: p.enq(y)
B: p:void
B: q.deq()
A: q:void
B: q:x
```

G=

```
A: q.enq(x)
B: p.enq(y)
A: q:void
B: p:void
B: q.deq()
B: q:x
```

H|A=G|A=

```
A: q.enq(x)
A: q:void
```

H|B=G|B=

```
B: p.enq(y)
B: p:void
B: q.deq()
B: q:x
```

Legal Histories

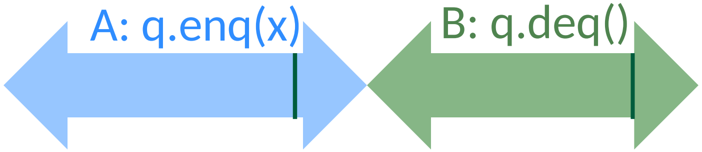
- **Sequential specification allows to describe what behavior we expect and tolerate**
  - When is a single-thread, single-object history legal?
- **Recall: Example**
  - Preconditions and Postconditions
  - Many others exist!
- **A sequential (multi-object) history H is legal if**
  - For every object x
  - H|x adheres to the sequential specification for x
- **Example: FIFO queue**
  - Correct internal state

    Order of removal equals order of addition
  - Full and Empty Exceptions

Precedence

A: q.enq(x)
B: q.enq(y)
B: q:void
A: q:void
B: q.deq()
B: q:x

A method execution precedes another if response event precedes invocation event

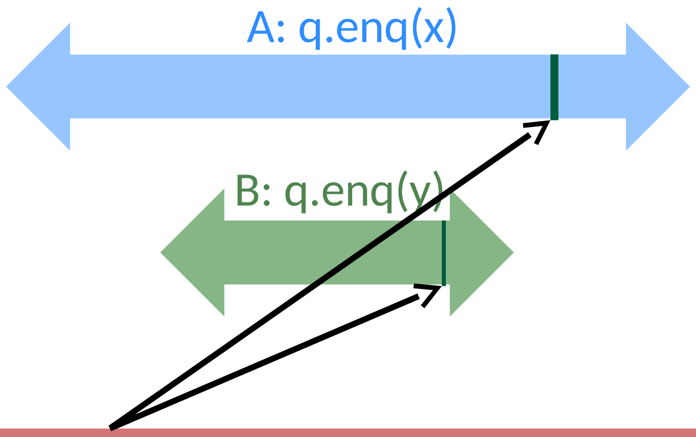


Precedence vs. Overlapping

- **Non-precedence = overlapping**

A: q.enq(x)
B: q.enq(y)
B: q:void
A: q:void
B: q.deq()
B: q:x

Some method executions overlap with others



Side Question: Is this a correct linearization order?

Complete Histories

- **A history H is complete**
  - If all invocations are matched with a response

H=

A: q.enq(x)
B: p.enq(y)
B: p:void
B: q.deq()
A: q:void
B: q:x

Complete

G=

A: q.enq(x)
B: p.enq(y)
B: p:void
B: q.deq()
A: q:void
A: q.enq(z)
B: q:x

Not complete

I=

A: q.enq(x)
B: p.enq(y)
B: p:void
B: q.deq()
A: q:void
B: q:x
B: q.deq()

Not complete

Which histories are complete and which are not?

Precedence Relations

- **Given history H**
- **Method executions m_0 and m_1 in H**
  - $m_0 \rightarrow_H m_1$ (m_0 precedes m_1 in H) if
  - Response event of m_0 precedes invocation event of m_1
- **Precedence relation $m_0 \rightarrow_H m_1$ is a**
  - Strict partial order on method executions

    Irreflexive, antisymmetric, transitive
- **Considerations**
  - Precedence forms a total order if H is sequential
  - Unrelated method calls □ may overlap □ concurrent

Definition Linearizability

- **A history H induces a strict partial order $<_H$ on operations**
  - $m_0 <_H m_1$ if $m_0 \rightarrow_H m_1$
- **A history H is linearizable if**
  - H can be extended to a complete history H'

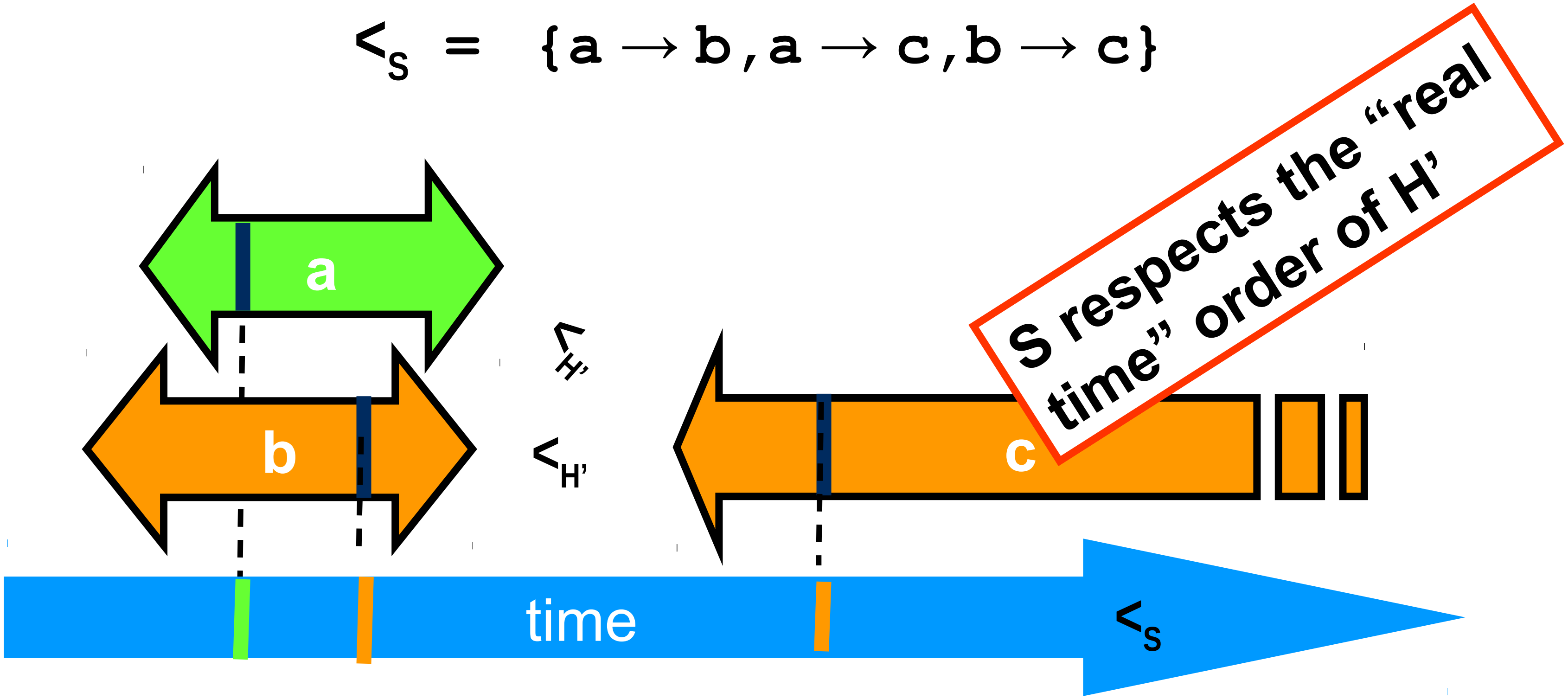
    by appending responses to pending operations or dropping pending operations
  - H' is equivalent to some legal sequential history S and
  - $<_{H'} \subseteq <_S$
- **S is a linearization of H**
- **Remarks:**
  - For each H, there may be many valid extensions to H'
  - For each extension H', there may be many S
  - Interleaving at the granularity of methods

Ensuring $<_{H'} \subseteq <_S$

- Find an S that contains H'

$<_{H'}$ = {a → c,b → c}

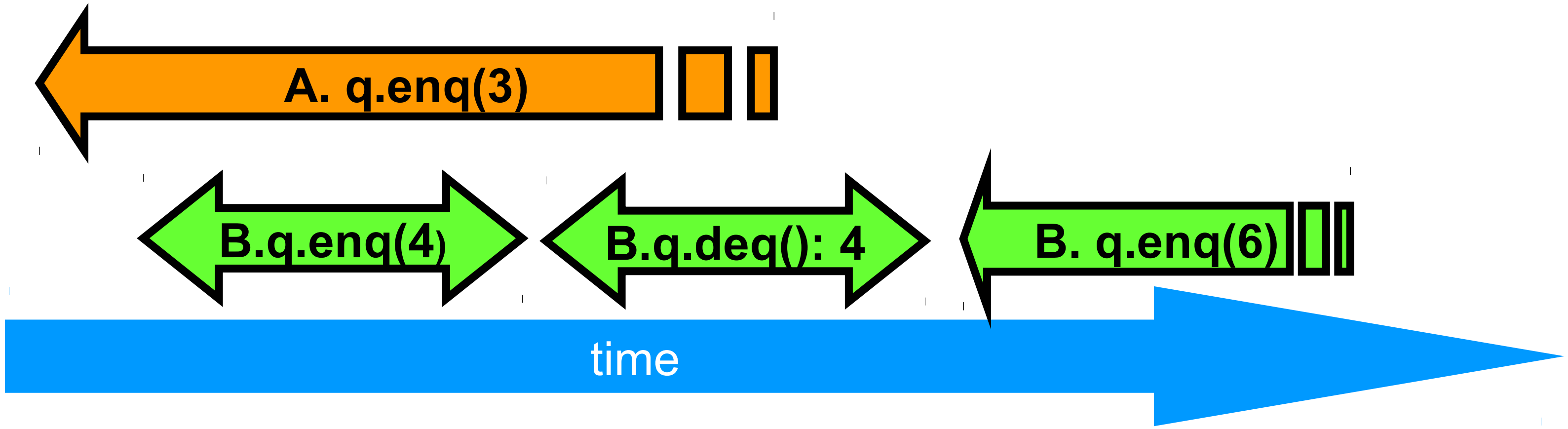
$<_S$ = {a → b,a → c,b → c}



S respects the "real time" order of H'

Example

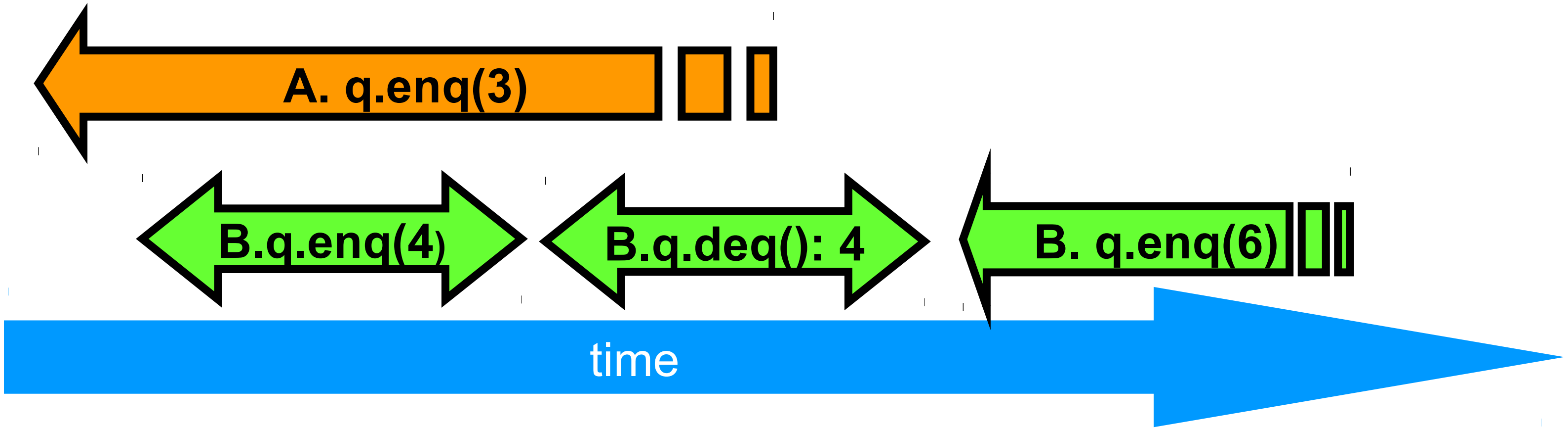
```
A q.enq(3)
B q.enq(4)
B q:void
B q.deq()
B q:4
B q:enq(6)
```



Example

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

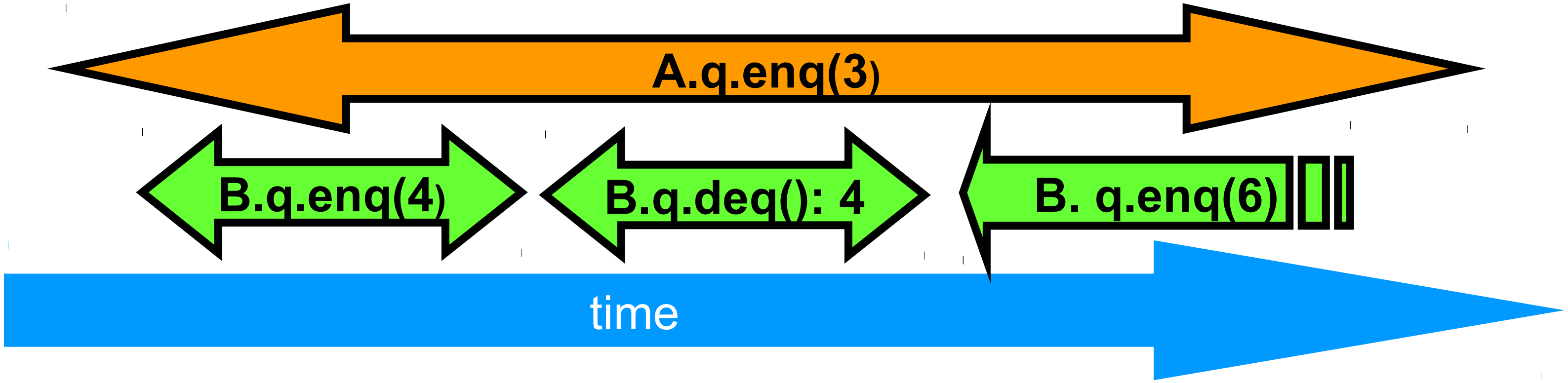
Complete this pending invocation



Example

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

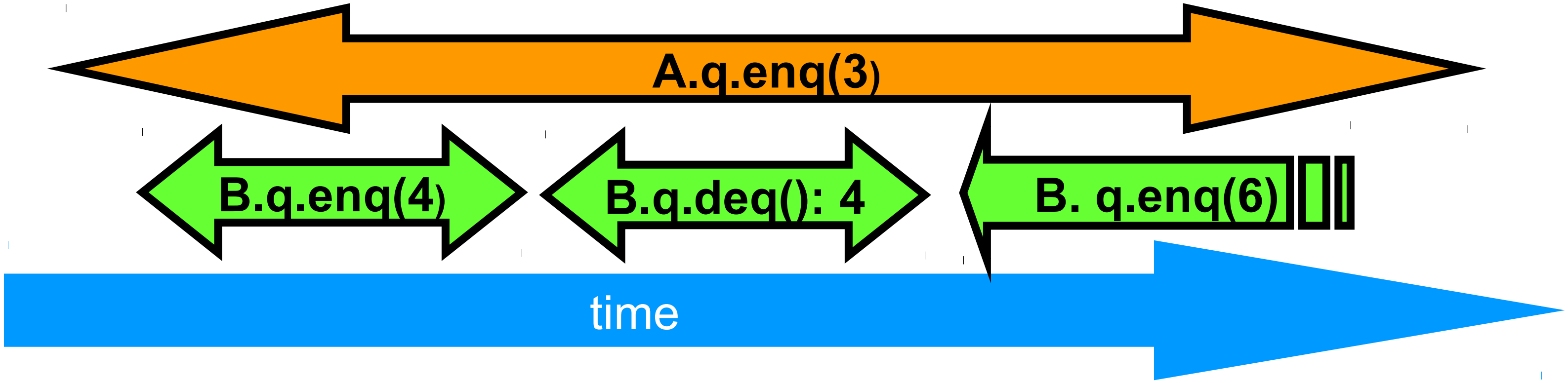
Complete this pending invocation



Example

discard this one

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

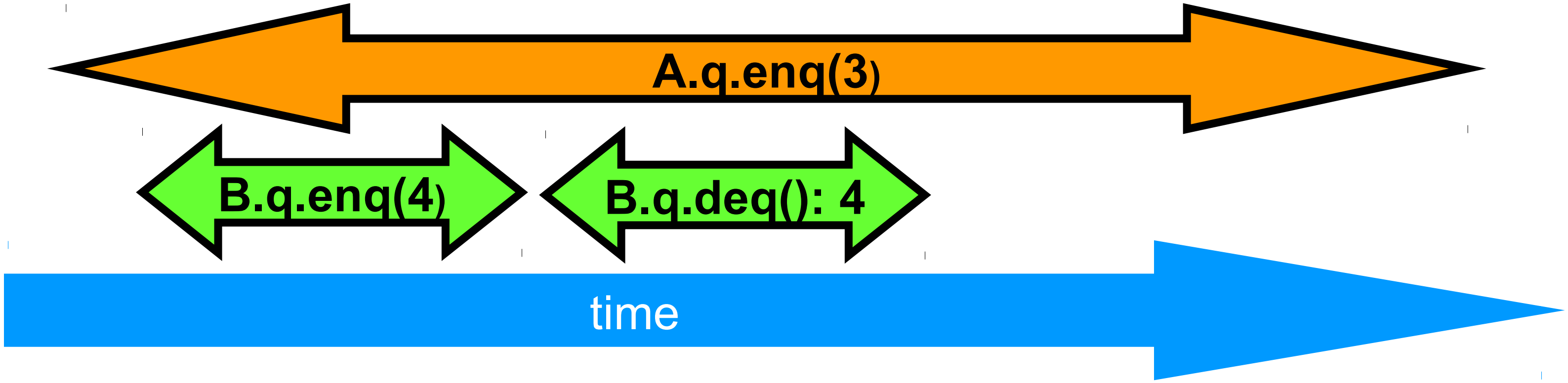


Example

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

discard this one

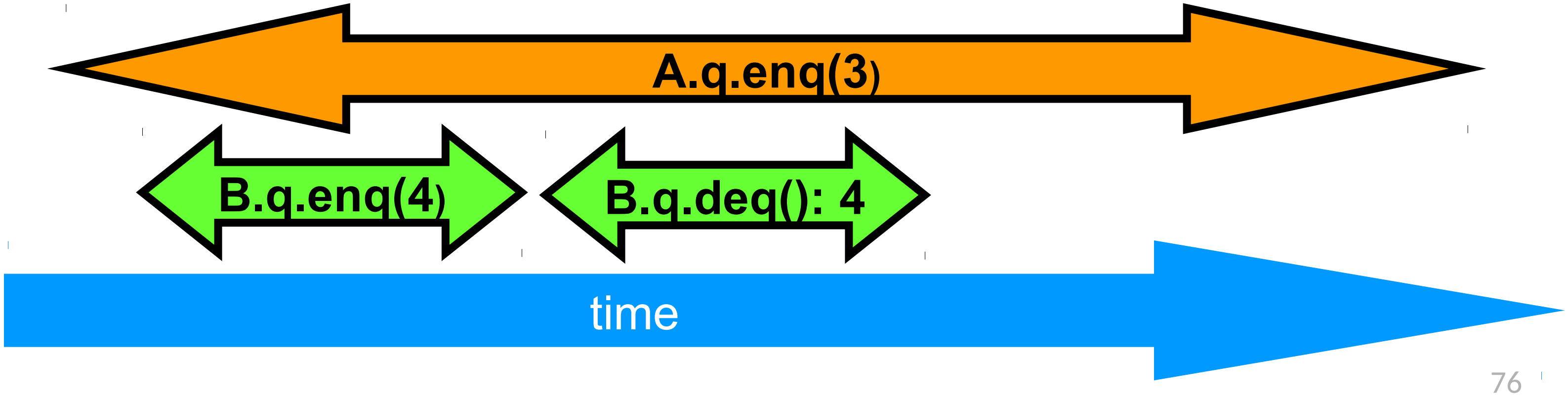
```
A q:void
```



Example

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

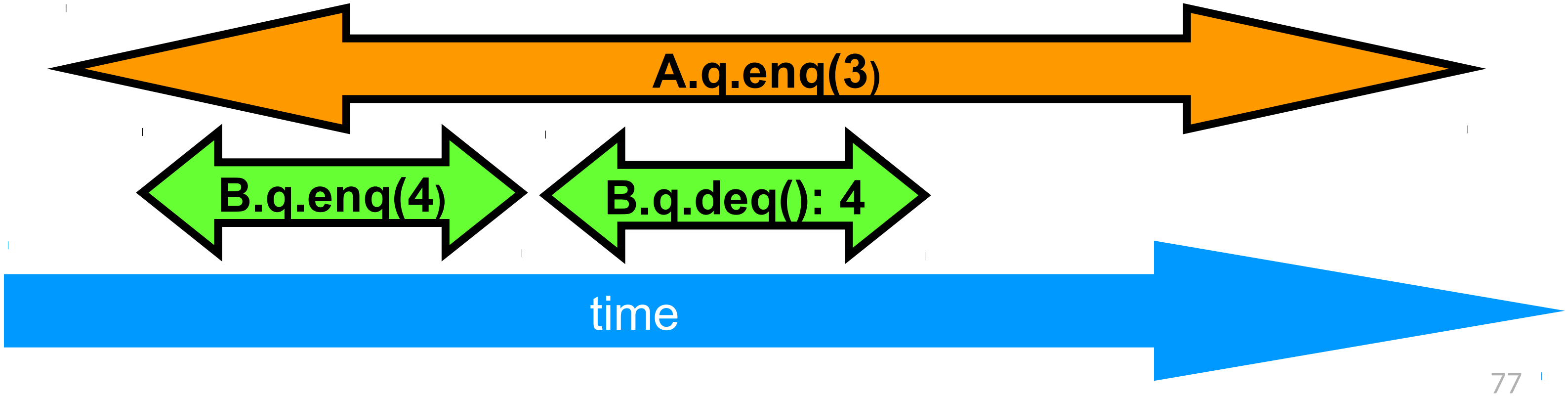
What would be an equivalent sequential history?



Example

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

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

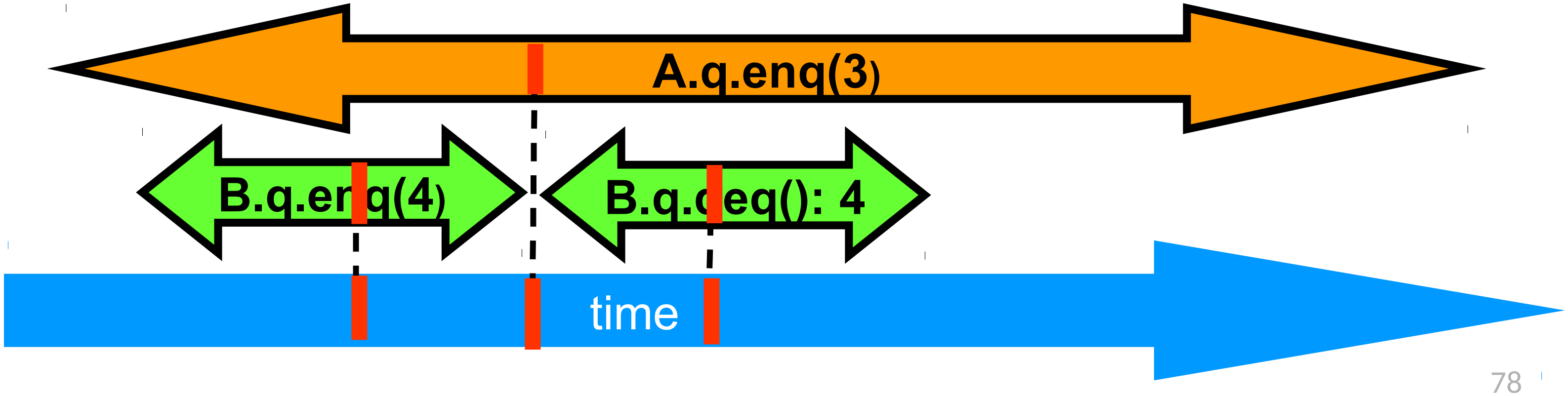


Example

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

Equivalent sequential history

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



Remember: Linearization Points

- **Identify one atomic step where a method “happens” (effects become visible to others)**
  - Critical section
  - Machine instruction (atomics, transactional memory …)
- **Does not always succeed**
  - One may need to define several different steps for a given method
  - If so, extreme care must be taken to ensur re-/postconditions
- **All possible executi e linearizable**

Now assuming wait-free two-thread queue?

```
void enq(Item x) {
  ~~std::lock_guard<std::mutex> l(lock)~~
  if((tail+1)%items.size() == head) {
    throw FullException;
  }
  items[tail] = x;
  tail = (tail+1)%items.size();
}
```

```
Item deq() {
  ~~std::lock_guard<std::mutex> l(lock)~~
  if(tail == head) {
    throw EmptyException;
  }
  Item item = items[head];
  head = (head+1)%items.size();
  return item;
}
```

Linearization points?

Composition

- **H is linearizable iff for every object x, H|x is linearizable!**
  - Corrollary: Composing linearizable objects results in a linearizable system

- **Reasoning**
  - Consider linearizability of objects in isolation

- **Modularity**
  - Allows concurrent systems to be constructed in a modular fashion
  - Compose independently-implemented objects

Linearizability vs. Sequential Consistency

- **Sequential consistency**
  - Correctness condition
  - For describing hardware memory interfaces
  - Remember: not *actual* ones!

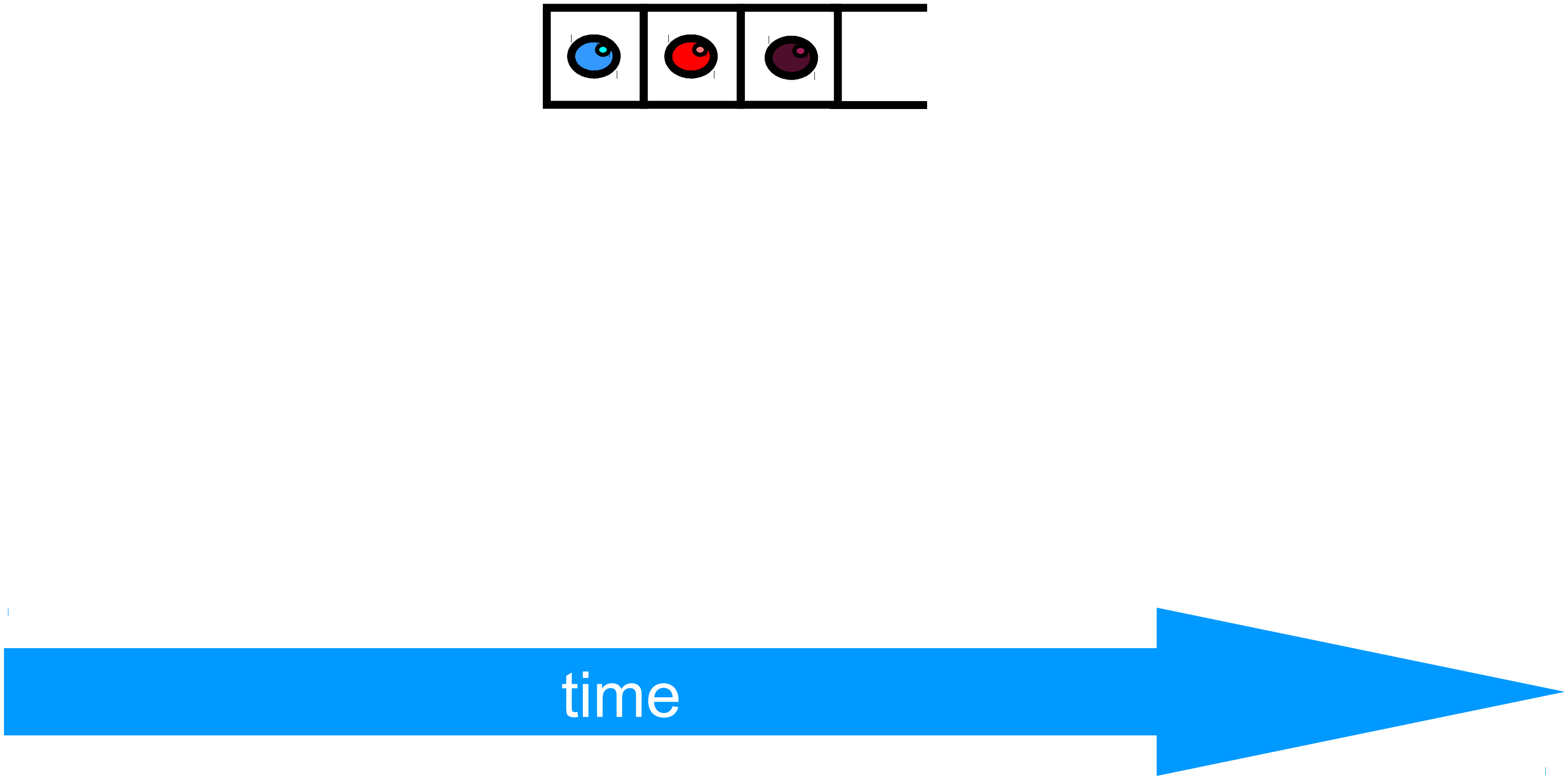
- **Linearizability**
  - Stronger correctness condition
  - For describing higher-level systems composed from linearizable components

    Requires understanding of object semantics

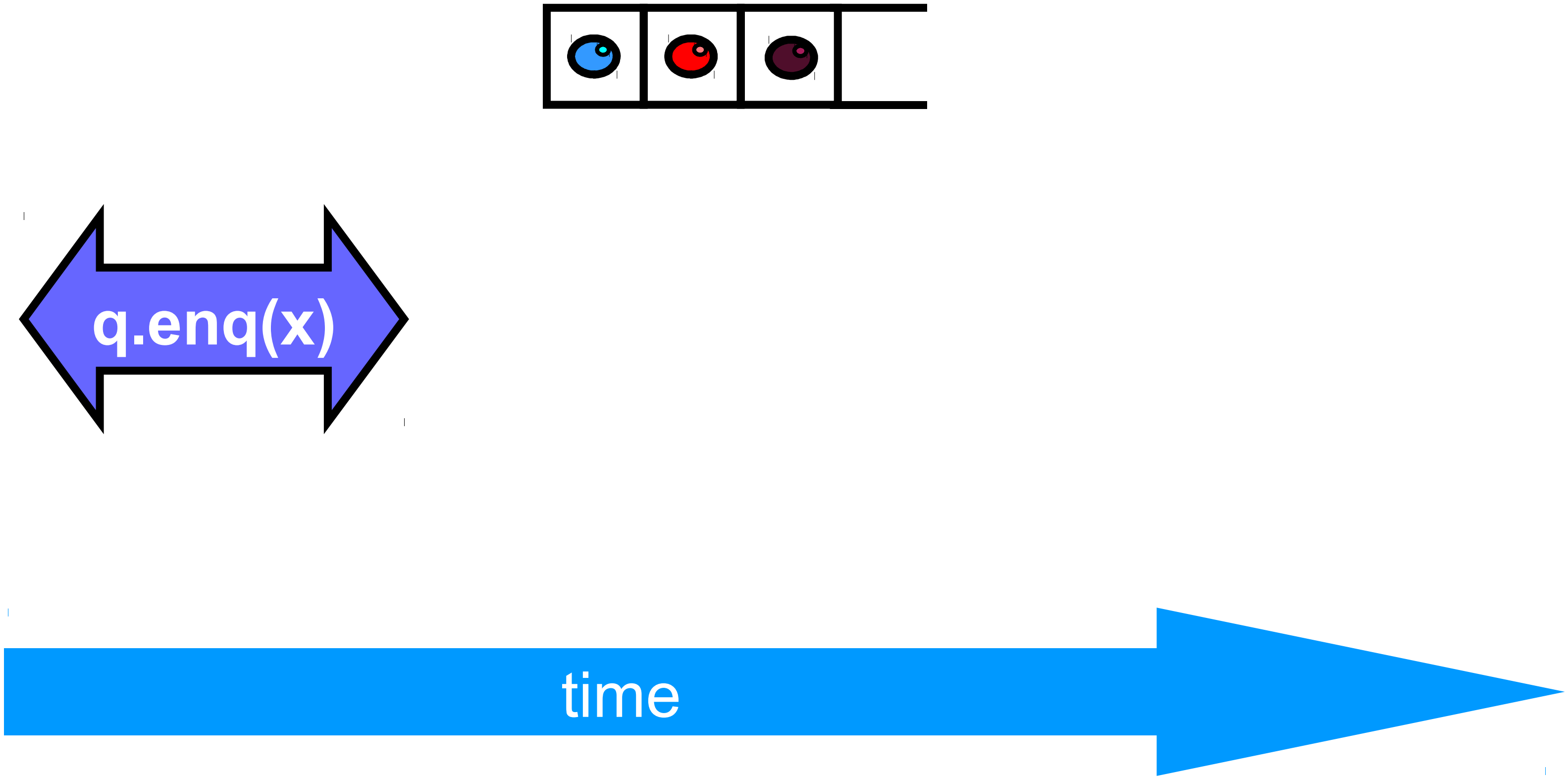
Map linearizability to sequential consistency

- **Variables with read and write operations**
  - Sequential consistency
- **Objects with a type and methods**
  - Linearizability
- **Map sequential consistency ↔ linearizability**
  - ← Reduce data types to variables with read and write operations
  - → Model variables as data types with read() and write() methods
- **Remember: Sequential consistency**
  - A history H is sequential if it can be extended to H’ and H’ is equivalent to some sequential history S
  - Note: Precedence order ($<_H \subseteq <_S$) does not need to be maintained

Example



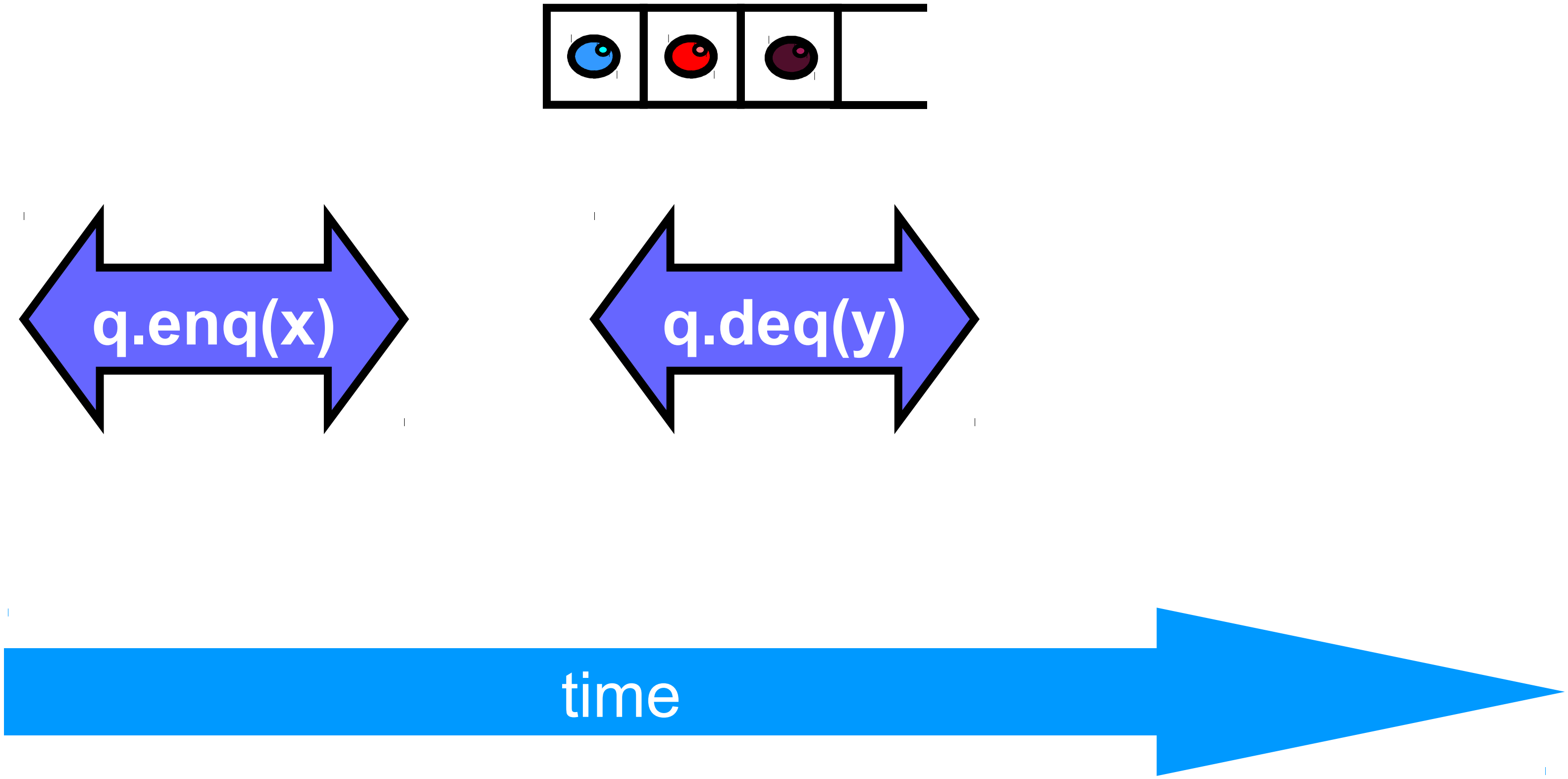
Example



q.enq(x)

time

Example



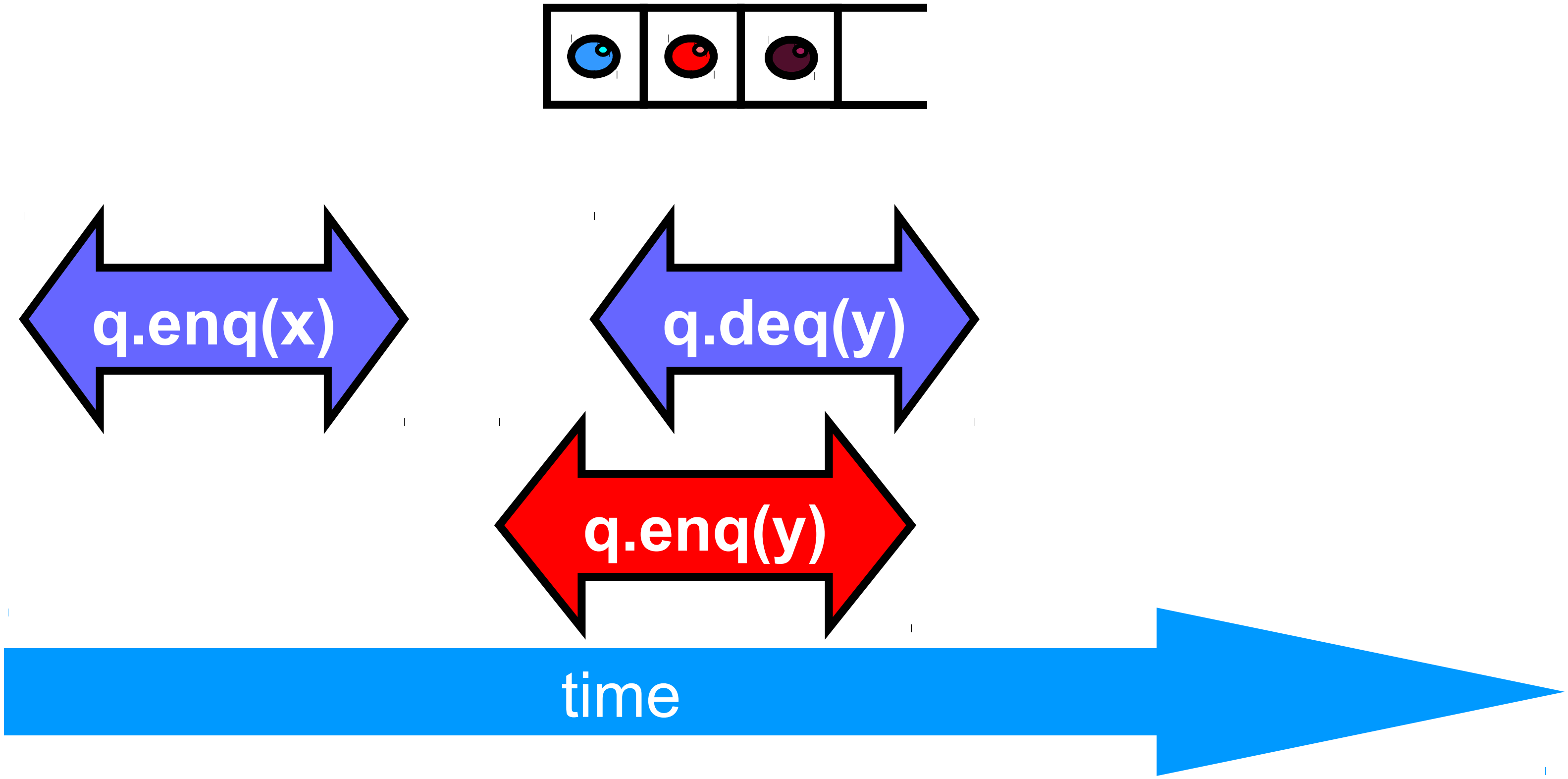
q.enq(x)

q.deq(y)

time

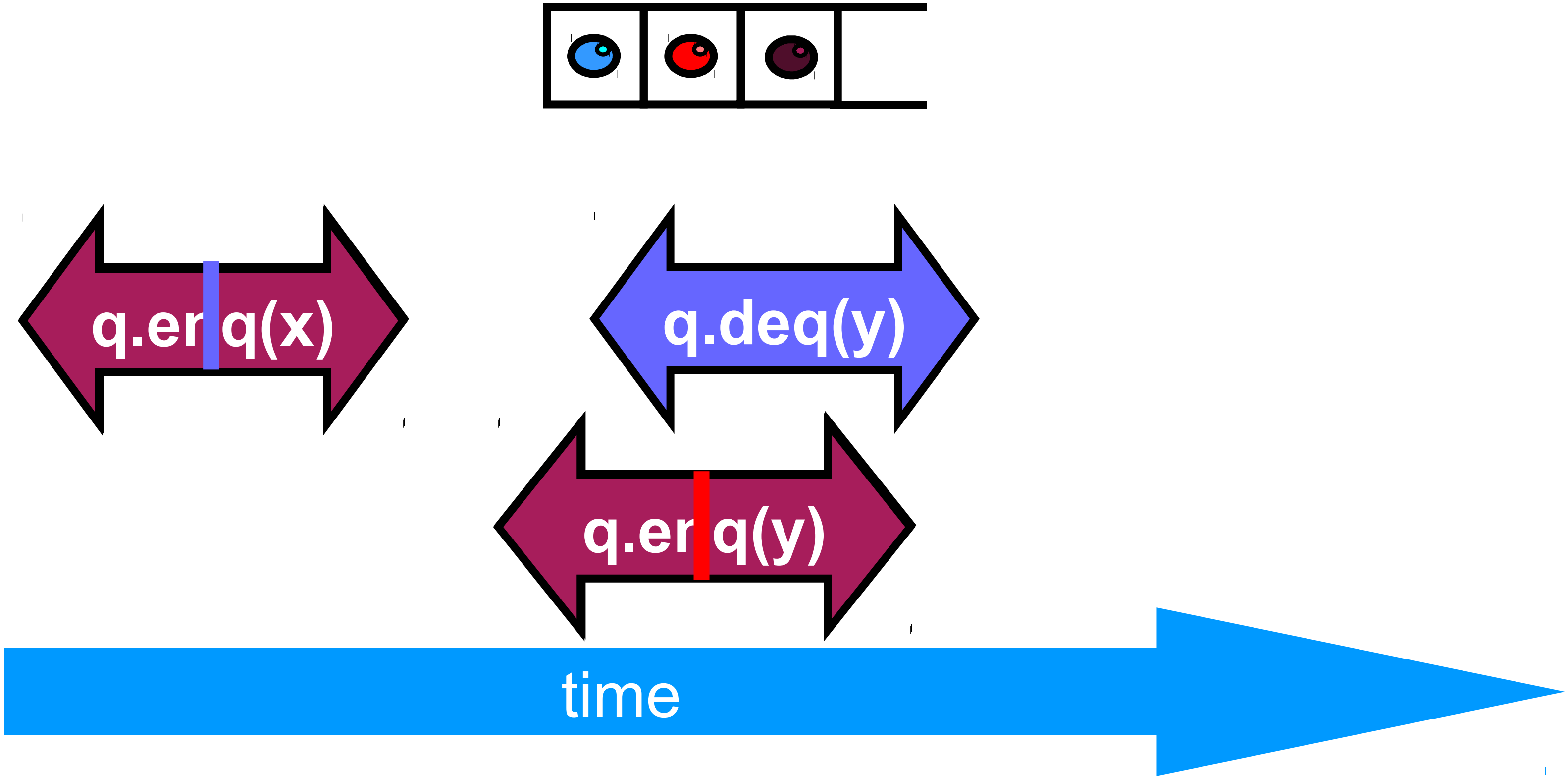
Example

Linearizable?



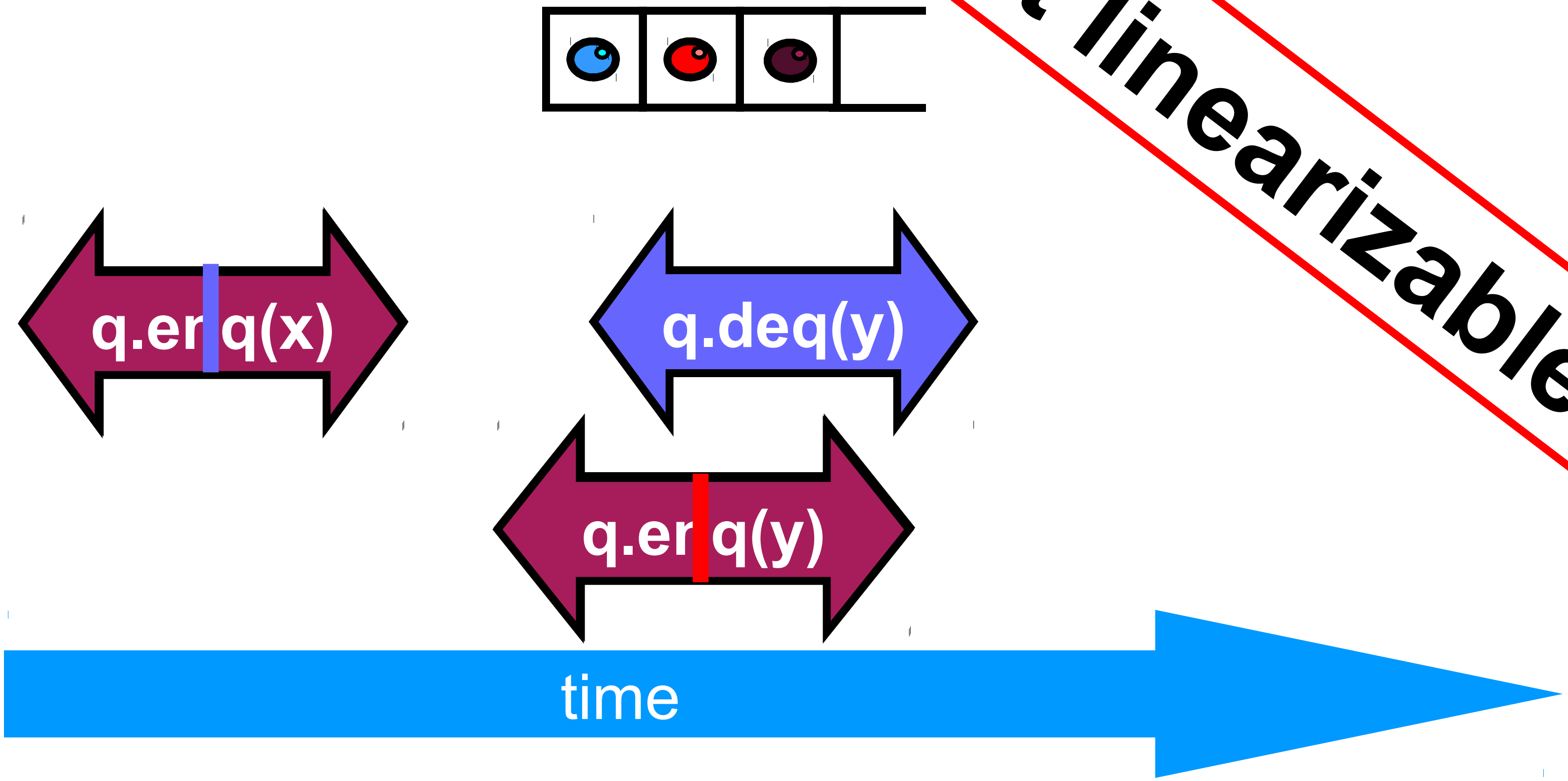
Example

Linearizable?



Example

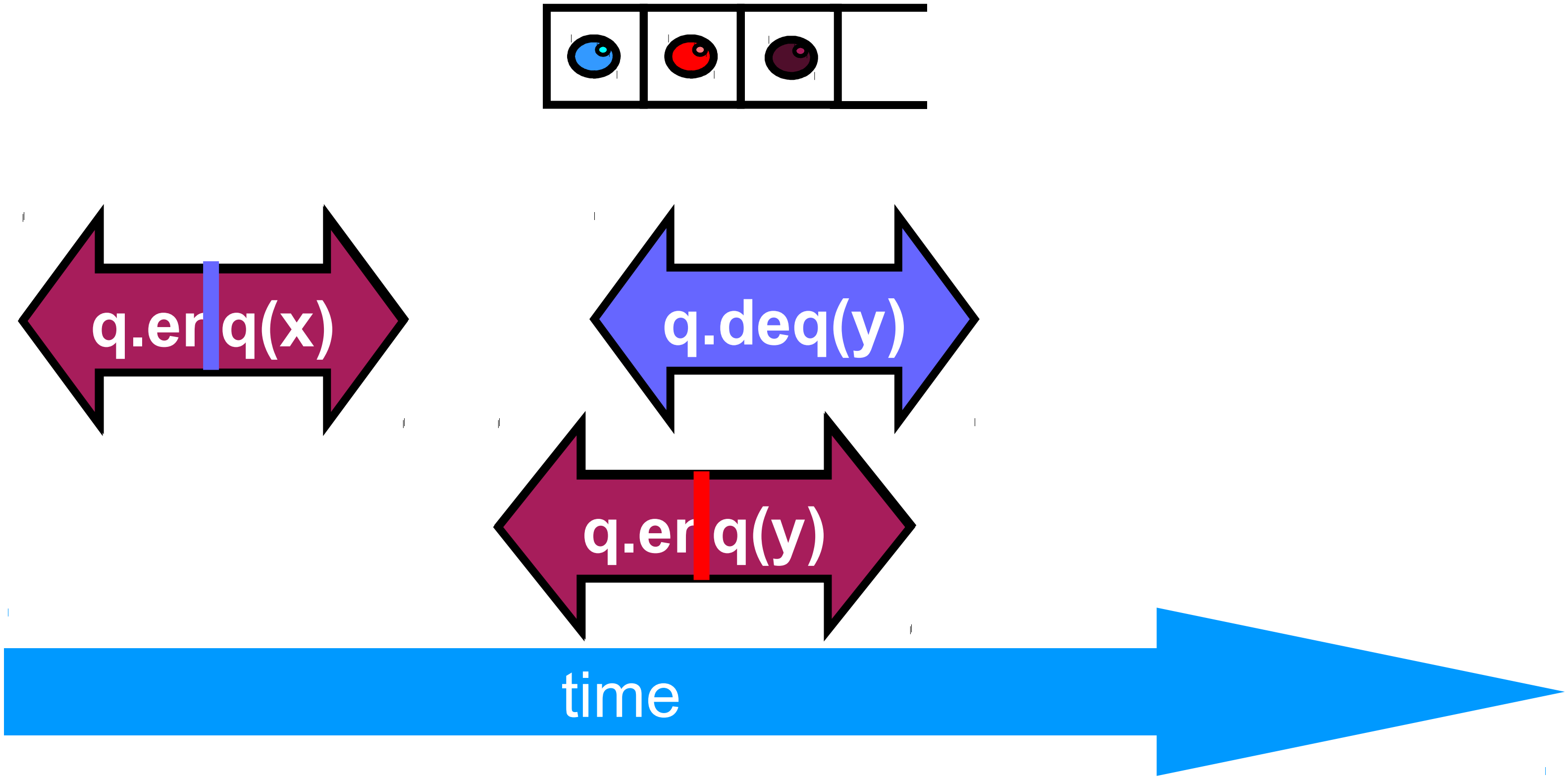
Linearizable?



not linearizable

Example

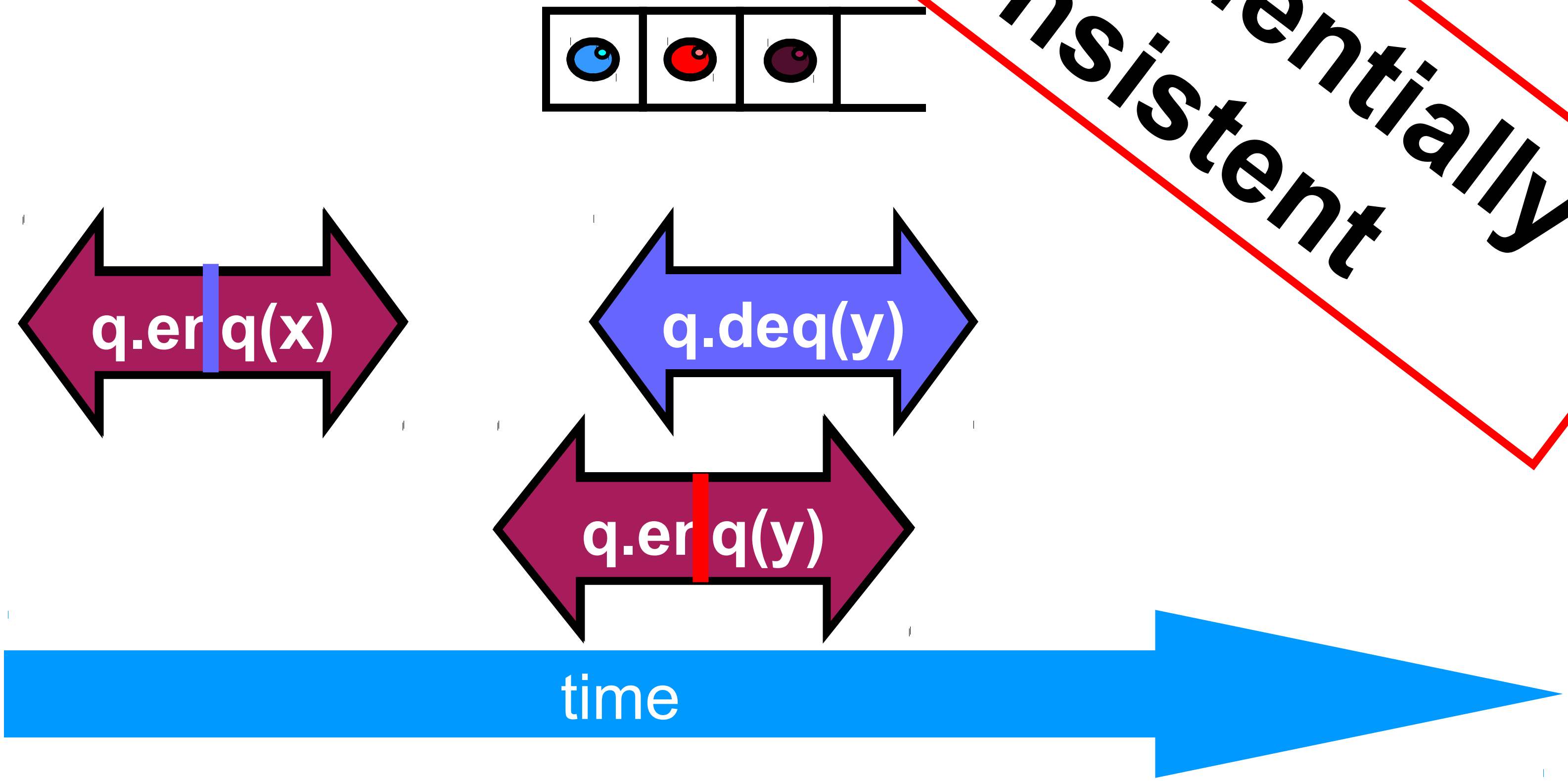
Sequentially consistent?



Example

Sequentially consistent?

yet sequentially consistent



Properties of sequential consistency

- **Theorem: Sequential consistency is not compositional**

H=

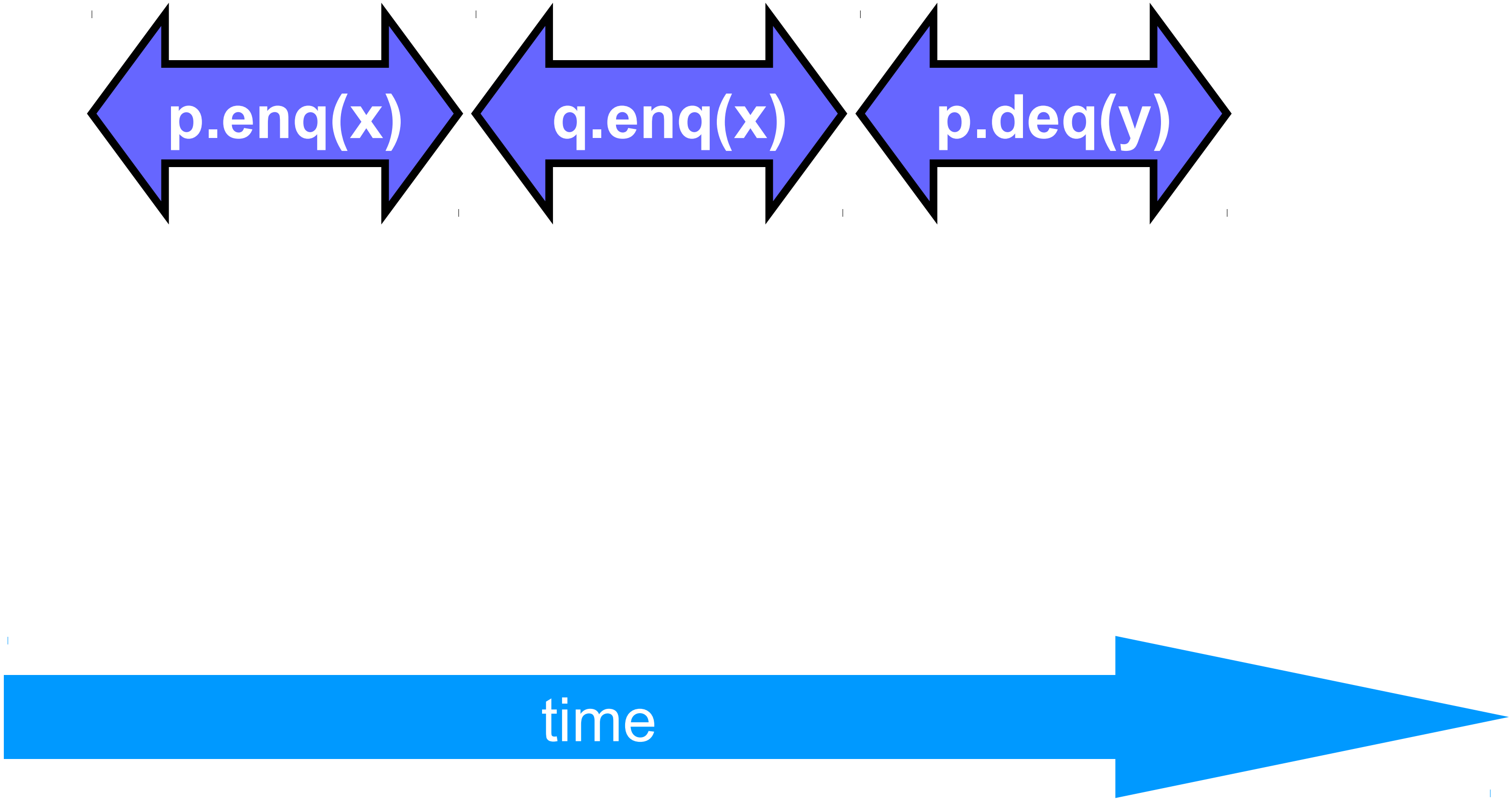
```
A: p.enq(x)
A: p:void
B: q.enq(y)
B: q:void
A: q.enq(x)
A: q:void
B: p.enq(y)
B: p:void
A: p.deq()
A: p:y
B: q.deq()
B: q:x
```

Compositional would mean:
“If H|p and H|q are sequentially consistent,
then H is sequentially consistent!”

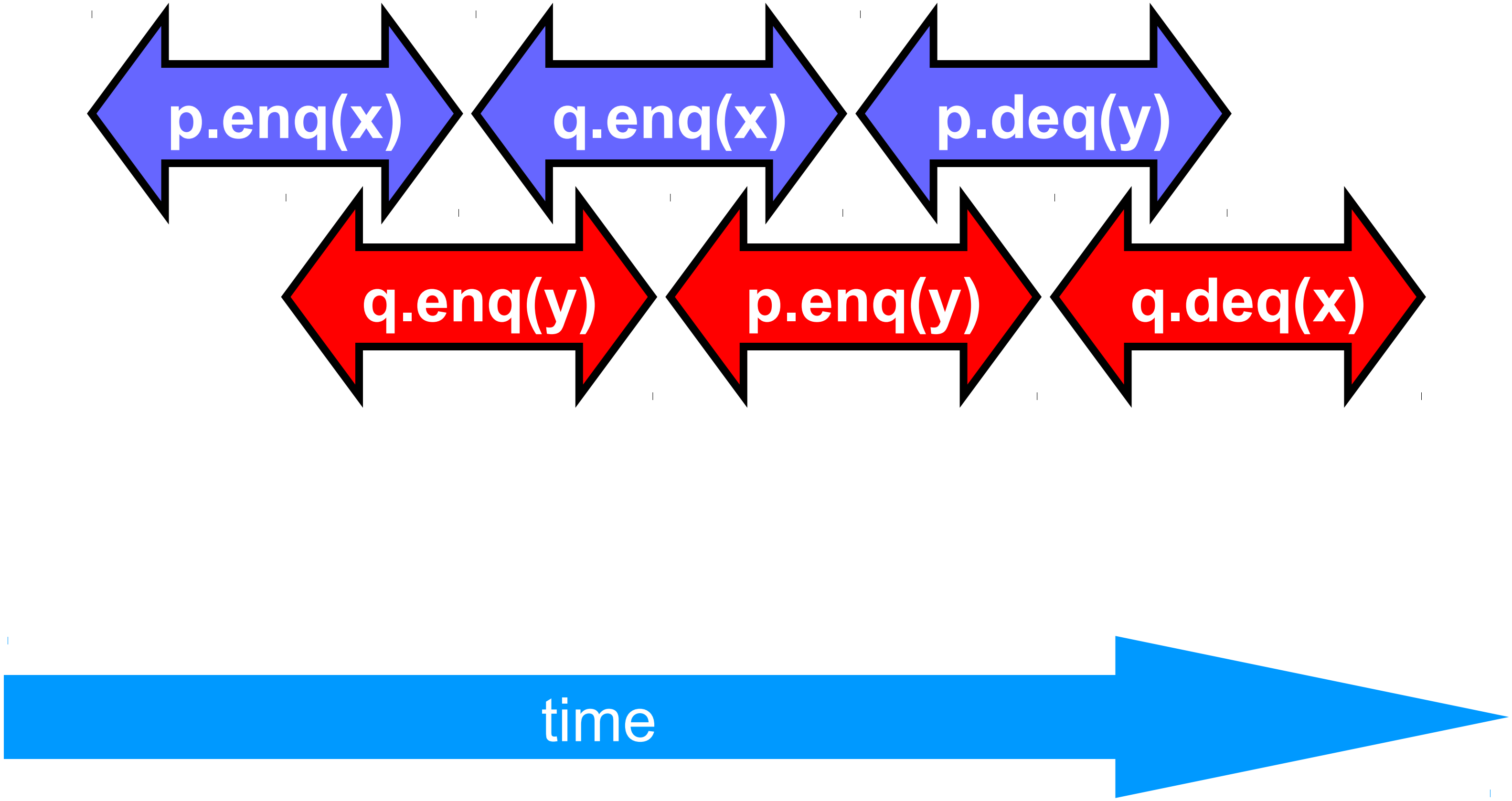
This is not guaranteed for SC schedules!

See following example!

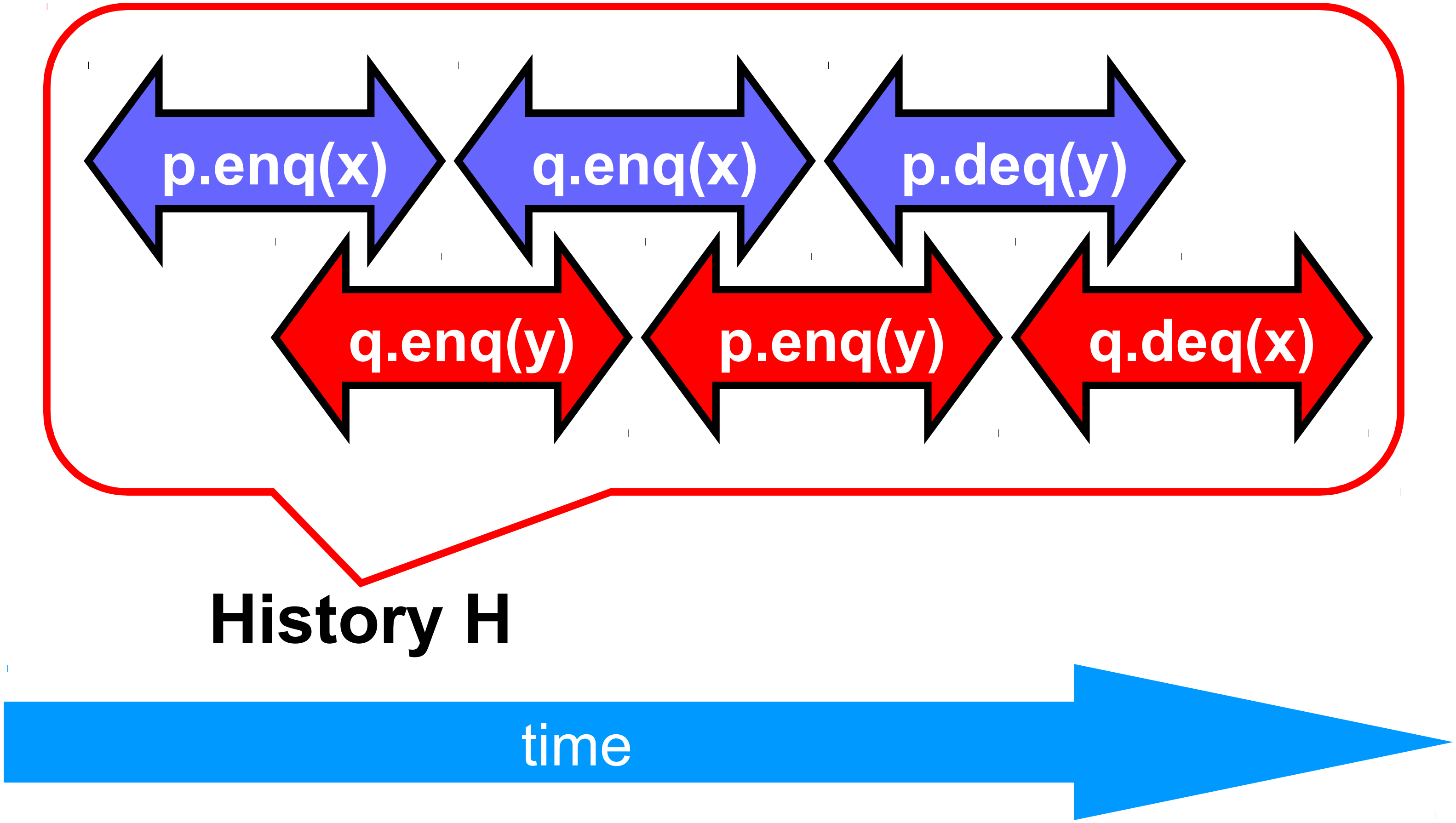
FIFO Queue Example



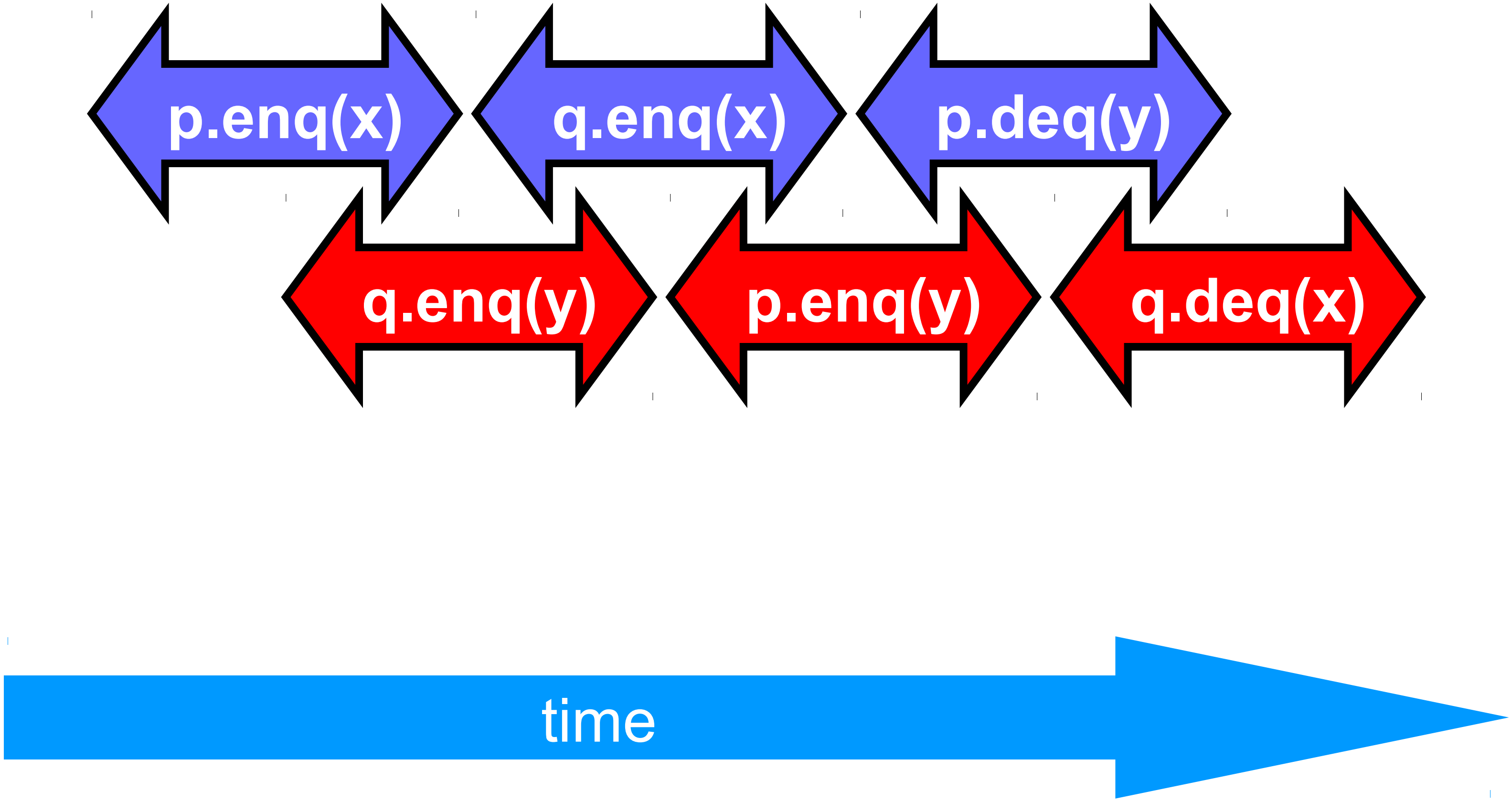
FIFO Queue Example



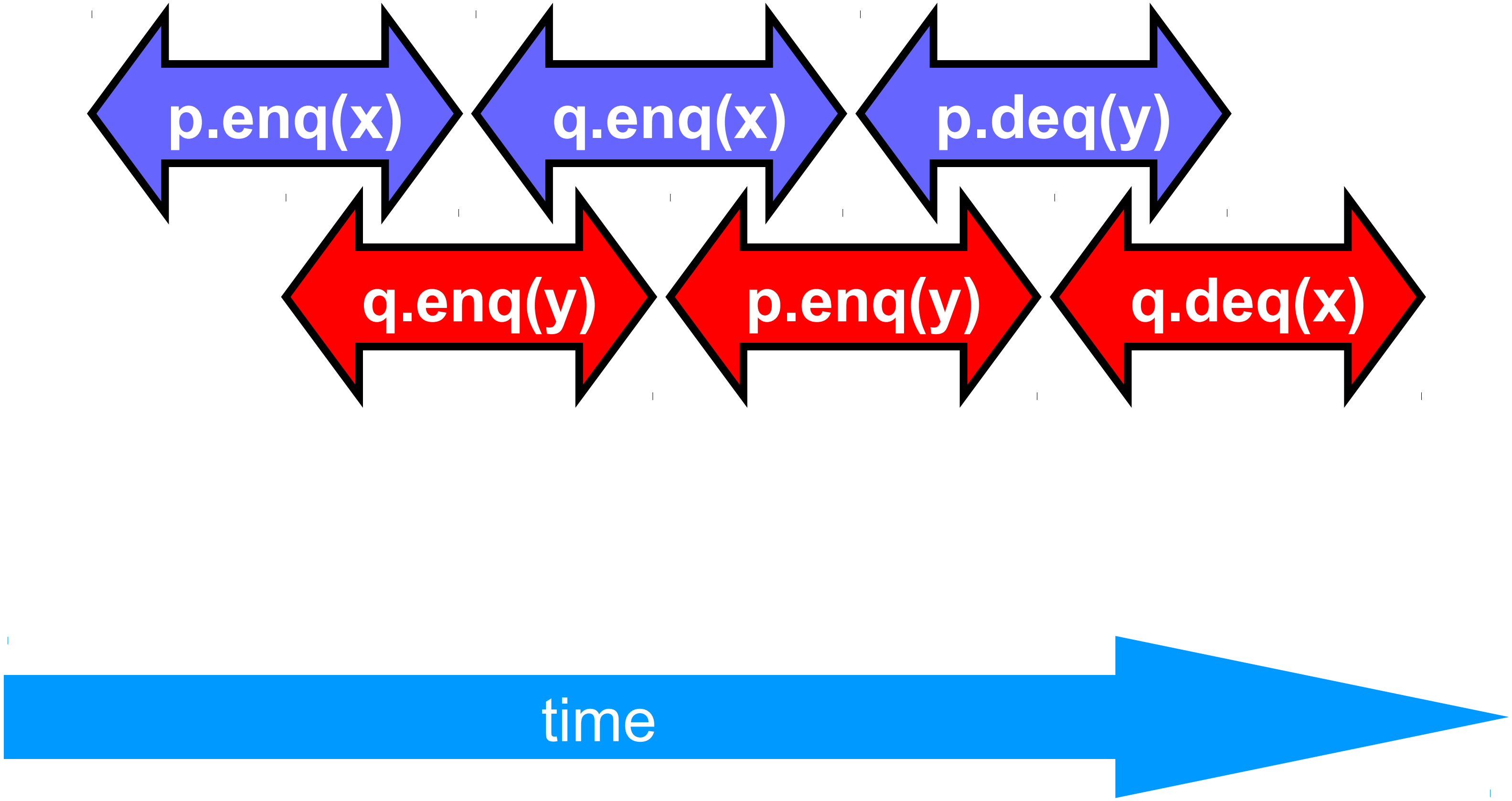
FIFO Queue Example



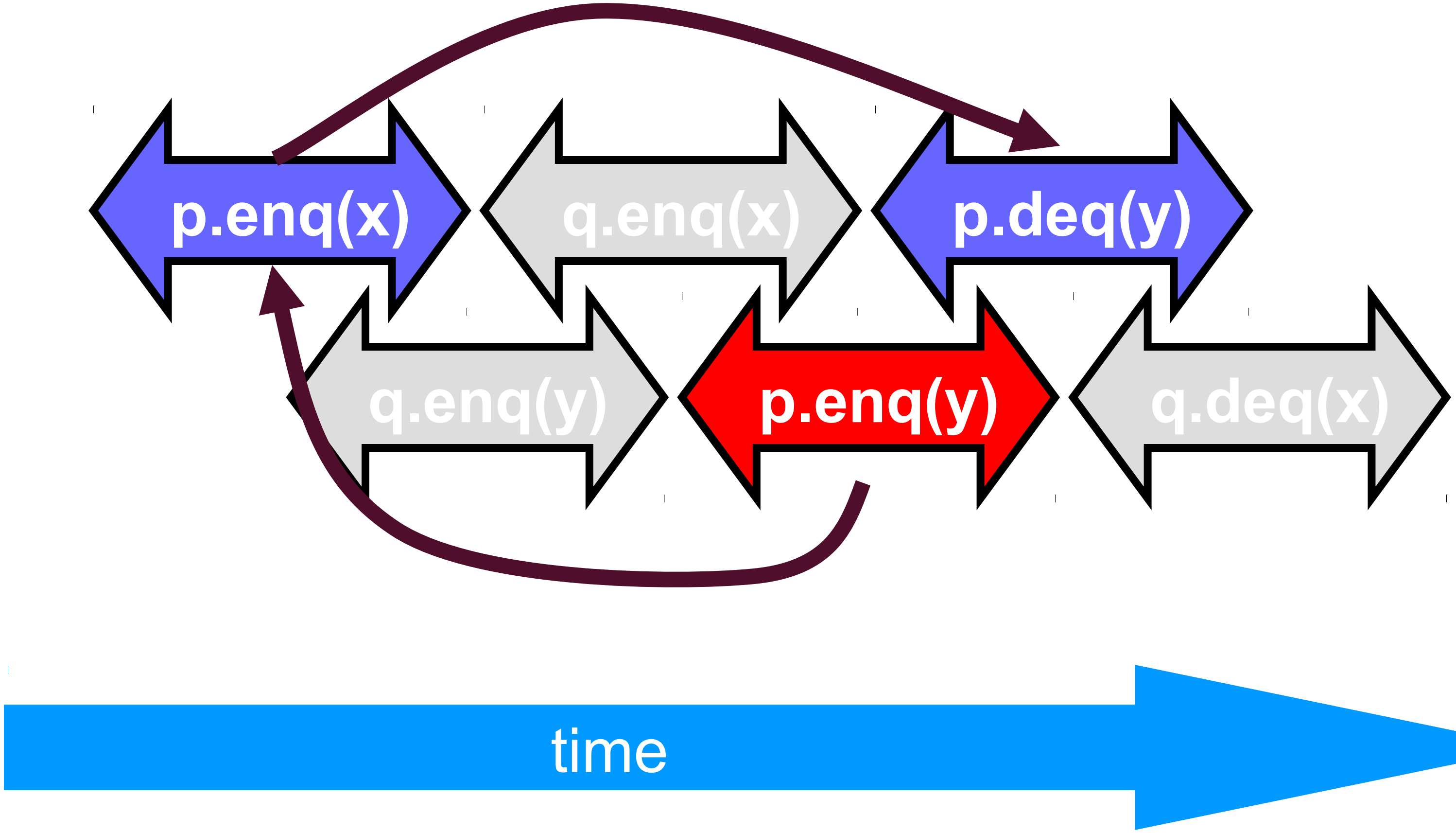
H|p Sequentially Consistent



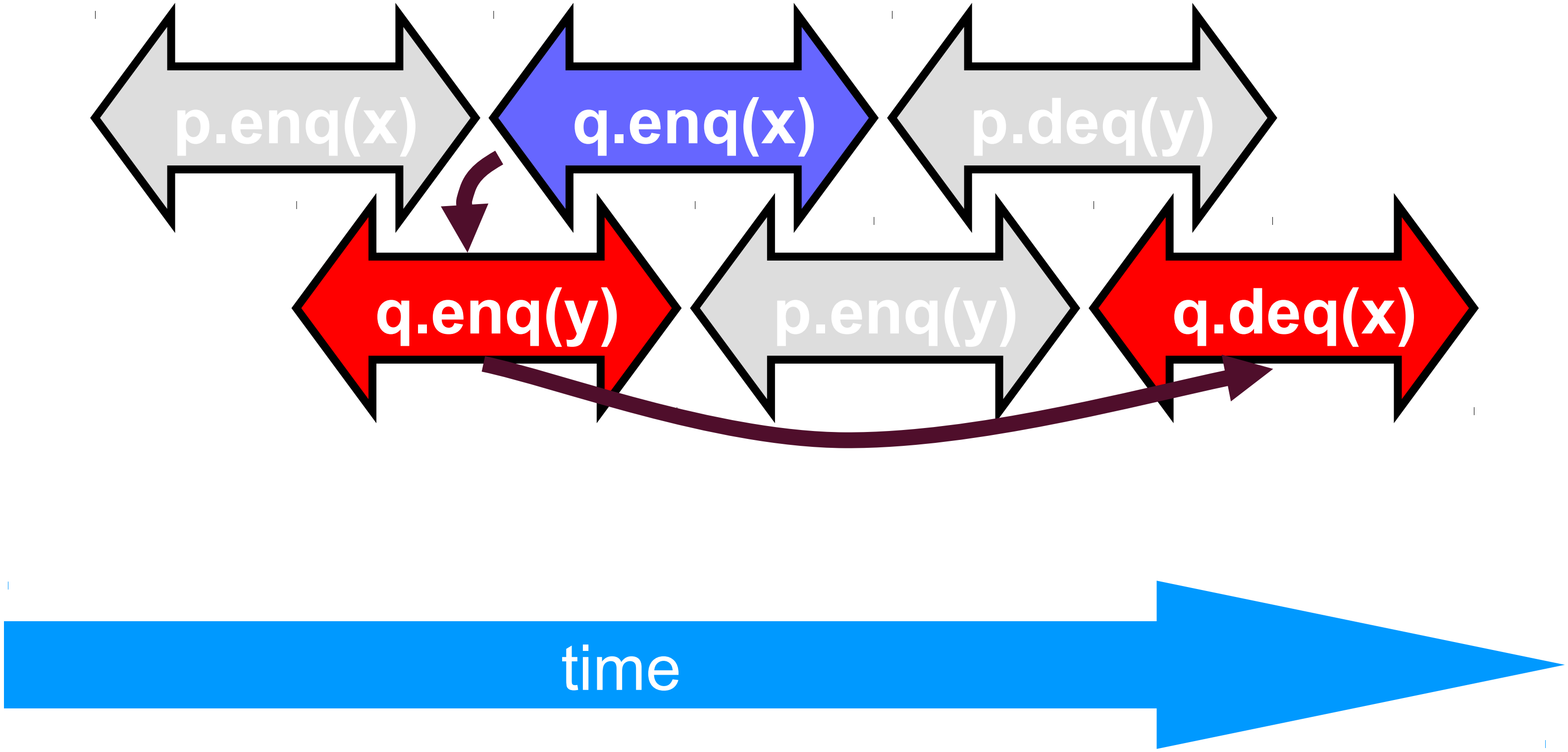
H|q Sequentially Consistent



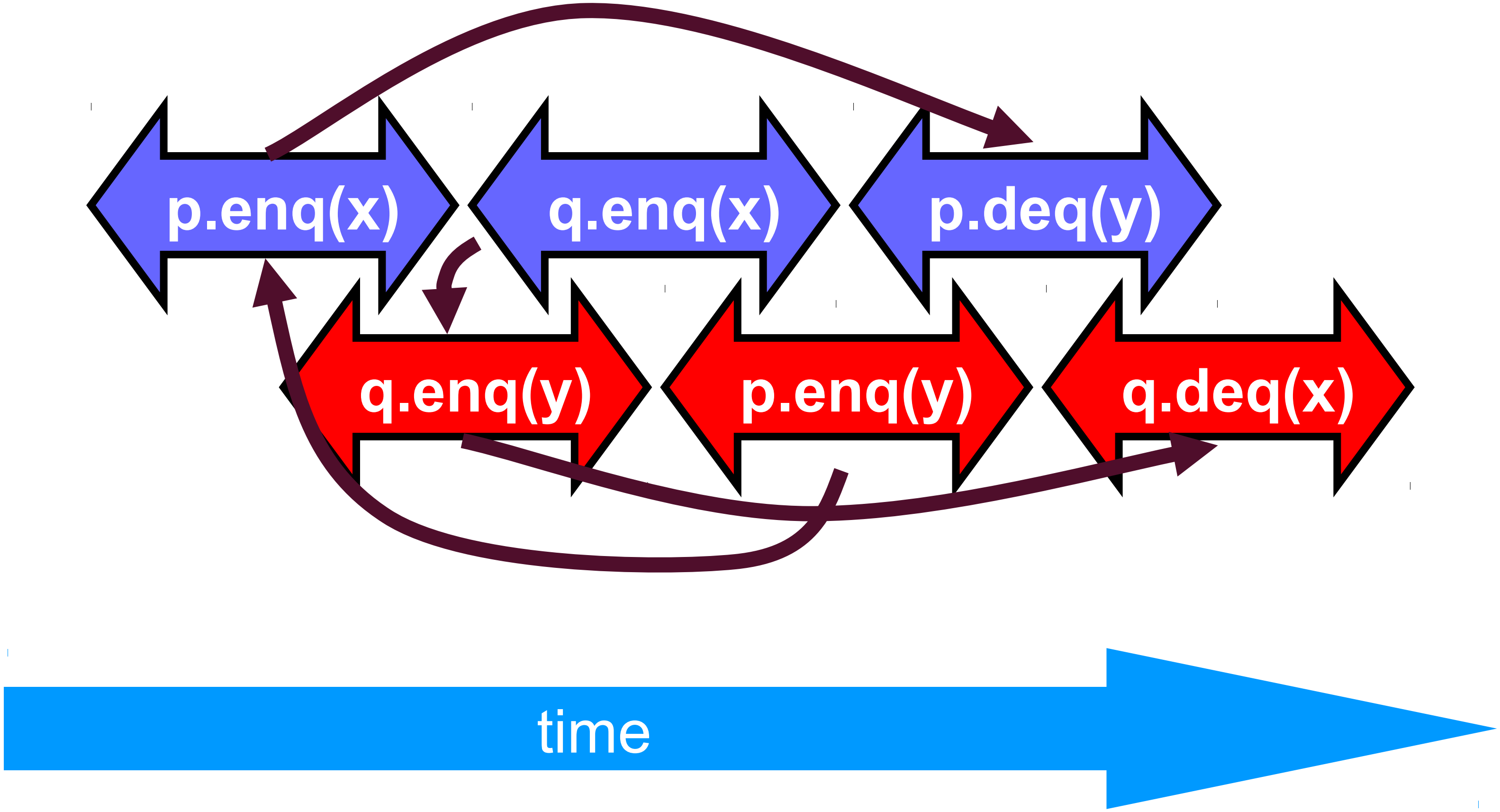
Ordering imposed by p



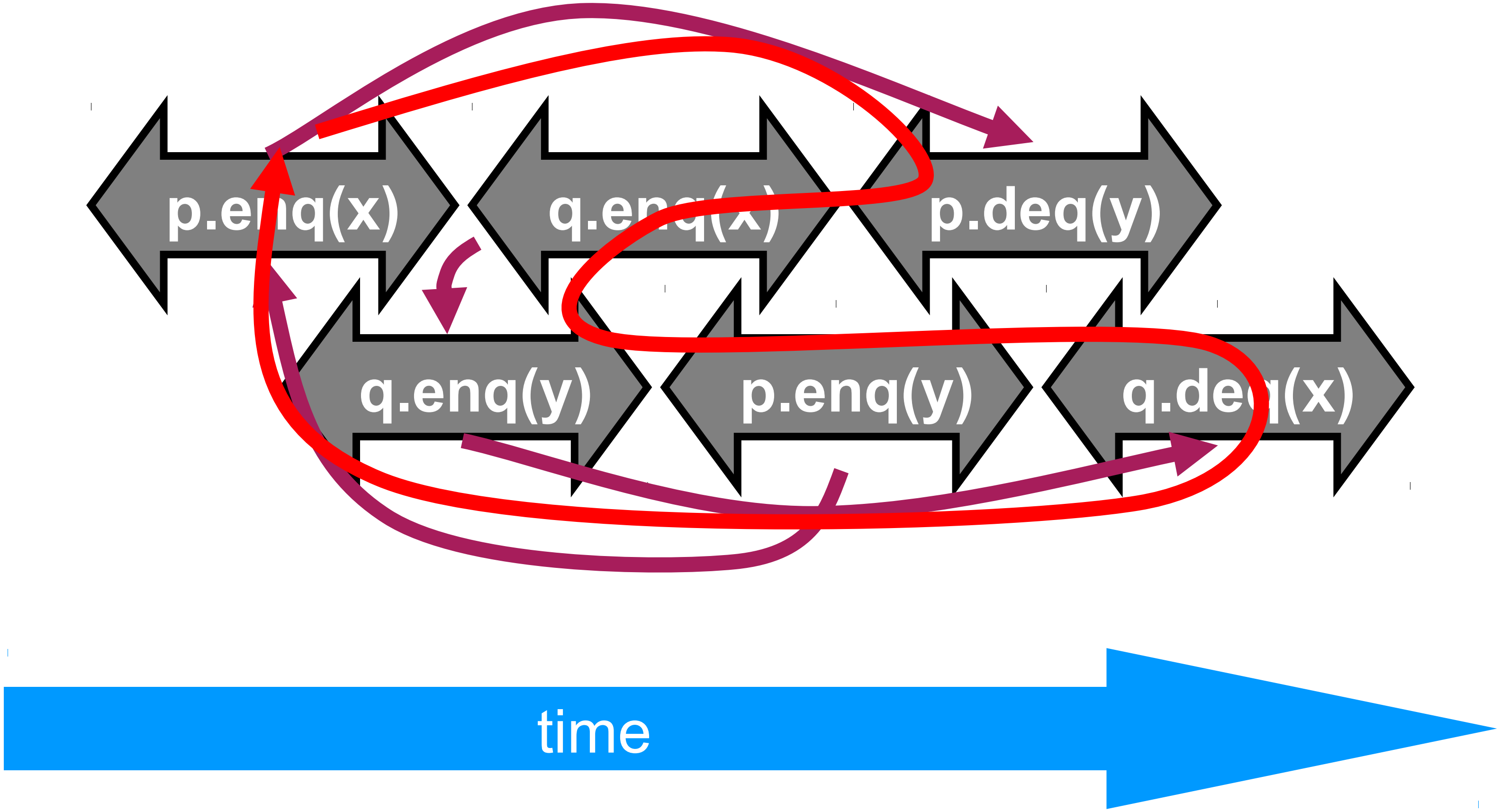
Ordering imposed by q



Ordering imposed by both



Combining orders



Example in our notation

- **Sequential consistency is not compositional – H|p**

H=

A: p.enq(x)
A: p:void
B: q.enq(y)
B: q:void
A: q.enq(x)
A: q:void
B: p.enq(y)
B: p:void
A: p.deq()
A: p:y
B: q.deq()
B: q:x

H|p=

A: p.enq(x)
A: p:void
B: p.enq(y)
B: p:void
A: p.deq()
A: p:y

(H|p)|A=

A: p.enq(x)
A: p:void
A: p.deq()
A: p:y

(H|p)|B=

B: p.enq(y)
B: p:void

H|p is sequentially consistent!

Example in our notation

- **Sequential consistency is not compositional – H|q**

H=

A: p.enq(x)
A: p:void
B: q.enq(y)
B: q:void
A: q.enq(x)
A: q:void
B: p.enq(y)
B: p:void
A: p.deq()
A: p:y
B: q.deq()
B: q:x

H|q=

B: q.enq(y)
B: q:void
A: q.enq(x)
A: q:void
B: q.deq()
B: q:x

(H|q)|A=

A: q.enq(x)
A: q:void

(H|q)|B=

B: q.enq(y)
B: q:void
B: q.deq()
B: q:x

H|q is sequentially consistent!

Example in our notation

- **Sequential consistency is not compositional**

H=

```
A: p.enq(x)
A: p:void
B: q.enq(y)
B: q:void
A: q.enq(x)
A: q:void
B: p.enq(y)
B: p:void
A: p.deq()
A: p:y
B: q.deq()
B: q:x
```

H|A=

```
A: p.enq(x)
A: p:void
A: q.enq(x)
A: q:void
A: p.deq()
A: p:y
```

H|B=

```
B: q.enq(y)
B: q:void
B: p.enq(y)
B: p:void
B: q.deq()
B: q:x
```

H is not sequentially consistent!

Correctness: Linearizability

- **Sequential Consistency**
  - Not composable
  - Harder to work with
  - Good (simple) way to think about hardware models

    Few assumptions (no semantics or time)

- **We will use *linearizability* in the remainder of this course unless stated otherwise**

  Consider routine entry and exit

Study Goals (Homework)

- **Define linearizability with your own words!**
- **Describe the properties of linearizability!**
- **Explain the differences between sequential consistency and linearizability!**

- **Given a history H**
  - Identify linearization points
  - Find equivalent sequential history S
  - Decide and explain whether H is linearizable
  - Decide and explain whether H is sequentially consistent
  - Give values for the response events such that the execution is linearizable

Language Memory Models

- **Which transformations/reorderings can be applied to a program**
- **Affects platform/system**
  - Compiler, (VM), hardware
- **Affects programmer**
  - What are possible semantics/output
  - Which communication between threads is legal?
- **Without memory model**
  - Impossible to even define “legal” or “semantics” when data is accessed concurrently
- **A memory model is a contract**
  - Between platform and programmer

History of Memory Models

- **Java's original memory model was broken [1]**
  - Difficult to understand => widely violated
  - Did not allow reorderings as implemented in standard VMs
  - Final fields could appear to change value without synchronization
  - Volatile writes could be reordered with normal reads and writes

    => *counter-intuitive for most developers*
- **Java memory model was revised [2]**
  - Java 1.5 (JSR-133)
  - Still some issues (operational semantics definition [3])
- **C/C++ didn't even have a memory model until recently**
  - Not able to make any statement about threaded semantics!
  - Introduced in C++11 and C11
  - Based on experience from Java, more conservative

[1] Pugh: "The Java Memory Model is Fatally Flawed", CCPE 2000

[2] Manson, Pugh, Adve: "The Java memory model", POPL'05

[3] Aspinall, Sevcik: "Java memory model examples: Good, bad and ugly", VAMP'07

Everybody wants to optimize

- **Language constructs for synchronization**
  - Java: volatile, synchronized, …
  - C++: atomic, (**NOT volatile**!), mutex, …

- **Without synchronization (defined language-specific)**
  - Compiler, (VM), architecture
  - Reorder and appear to reorder memory operations
  - Maintain sequential semantics per thread
  - Other threads may observe any order (have seen examples before)

Java and C++ High-level overview

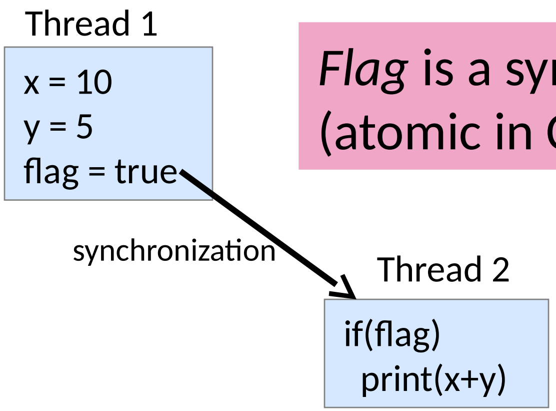
- **Relaxed memory model**
  - No global visibility ordering of operations
  - Allows for standard compiler optimizations
- **But**
  - Program order for each thread (sequential semantics)
  - Partial order on memory operations (with respect to synchronizations)
  - Visibility function defined
- **Correctly synchronized programs**
  - Guarantee sequential consistency
- **Incorrectly synchronized programs**
  - Java: maintain safety and security guarantees

    Type safety etc. (require behavior bounded by causality)
  - C++: undefined behavior

    No safety (anything can happen/change)

Communication between Threads: Intuition

- **Not guaranteed unless by:**
  - Synchronization
  - Volatile/atomic variables
  - Specialized functions/classes (e.g., java.util.concurrent, …)



```
x = 10
y = 5
flag = true
```

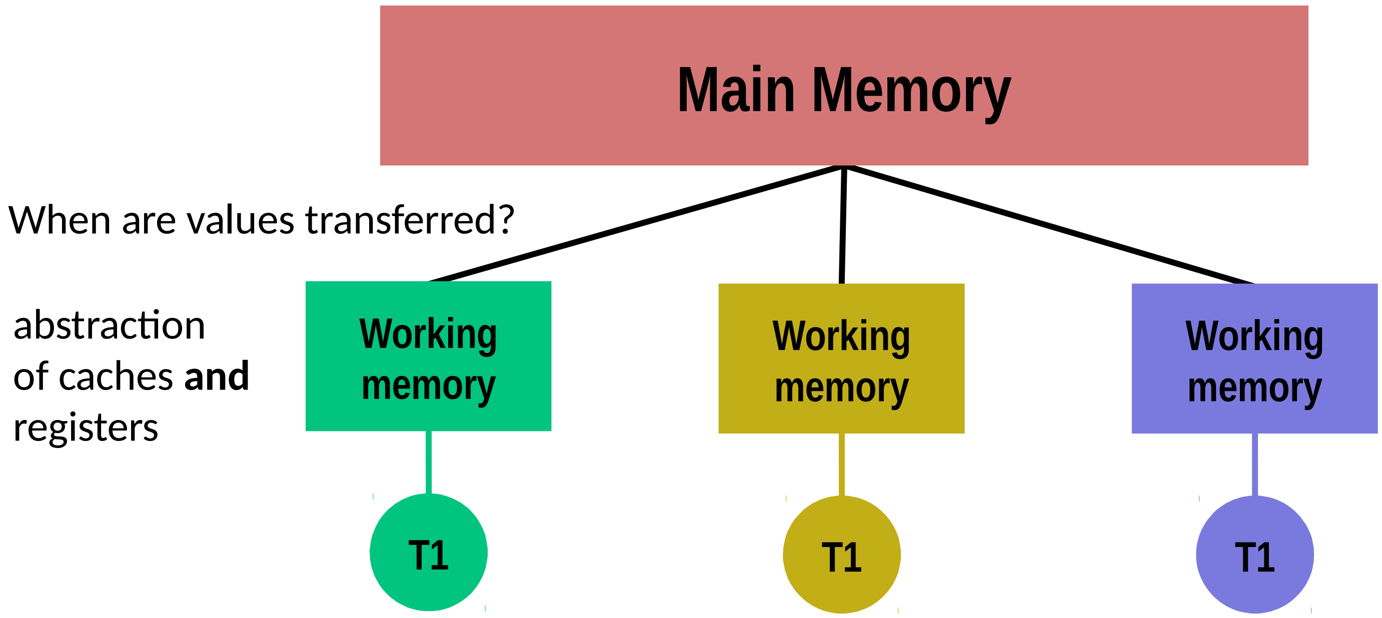
```
if(flag)
  print(x+y)
```

Flag is a synchronization variable (atomic in C++, volatile in Java),

i.e., all memory written by T1 must be visible to T2 after it reads the value true for *flag*!

Memory Model: Intuition

- **Abstract relation between threads and memory**
  - Local thread view!


Main Memory
When are values transferred?
abstraction of caches **and** registers
Working memory
Working memory
Working memory
T1
T1
T1

- **Does not talk about classes, objects, methods, …**
  - Linearizability is a higher-level concept!

Lock Synchronization

- **Java**

```
synchronized (lock) {
  // critical region
}
```

  - Synchronized methods as syntactic sugar

- **C++**

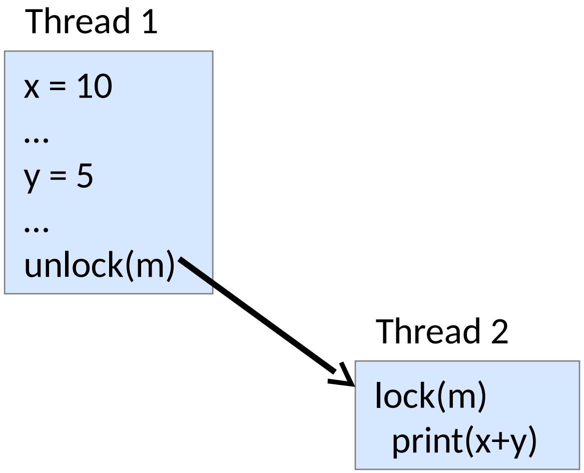
```
{
 unique_lock<mutex> l(lock);
 // critical region
}
```

  - Many flexible variants

- **Semantics:**
  - mutual exclusion
  - at most one thread may own a lock
  - a thread B trying to acquire a lock held by thread A blocks until thread A releases lock
  - note: threads may wait forever (no progress guarantee!)

Memory semantics

- **Similar to synchronization variables**



Thread 1

```
x = 10
…
y = 5
…
unlock(m)
```

Thread 2

```
lock(m)
  print(x+y)
```

  - All memory accesses before an unlock …
  - are ordered before and are visible to …
  - any memory access after a matching lock!

Synchronization Variables

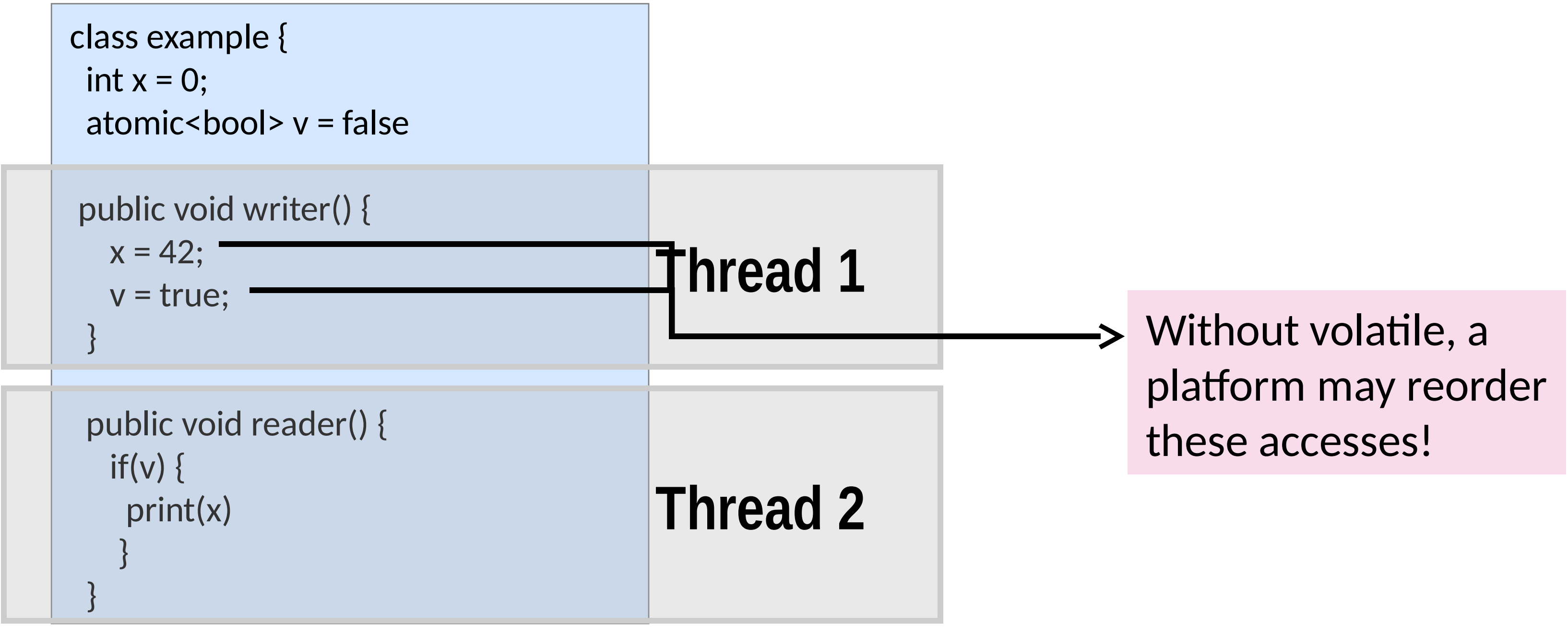
- **Variables can be declared volatile (Java) or atomic (C++)**

- **Reads and writes to synchronization variables**
  - Are totally ordered with respect to all threads
  - Must not be reordered with normal reads and writes

- **Compiler**
  - Must not allocate synchronization variables in registers
  - Must not swap variables with synchronization variables
  - May need to issue memory fences/barriers
  - …

Synchronization Variables

- **Write to a synchronization variable**
  - Similar memory semantics as unlock (no process synchronization!)
- **Read from a synchronization variable**
  - Similar memory semantics as lock (no process synchronization!)



```
class example {
  int x = 0;
  atomic<bool> v = false

 public void writer() {
    x = 42;
    v = true;
  }

  public void reader() {
    if(v) {
      print(x)
    }
  }
```

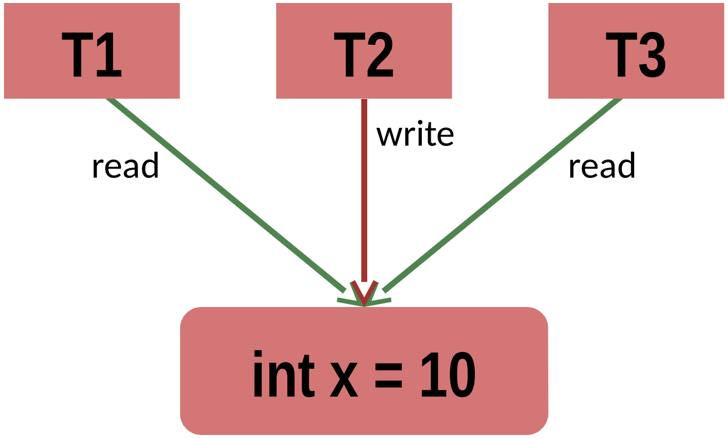
Thread 1

Thread 2

Without volatile, a platform may reorder these accesses!

Memory Model Rules

- **Java/C++: Correctly synchronized programs will execute sequentially consistent**
- **Correctly synchronized = data-race free**
  - iff all sequentially consistent executions are free of data races
- **Two accesses to a shared memory location form a data race in the execution of a program if**
  - The two accesses are from different threads
  - At least one access is a write and
  - The accesses are not synchronized


T1
T2
T3
read
write
read
int x = 10