Design of Parallel and High-Performance Computing

Fall 2013

Lecture: Linearizability

Instructor: Torsten Hoefler & Markus Püschel

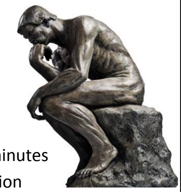
TAs: Timo Schneider, Arnamoy Bhattacharyya

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Review of last lecture

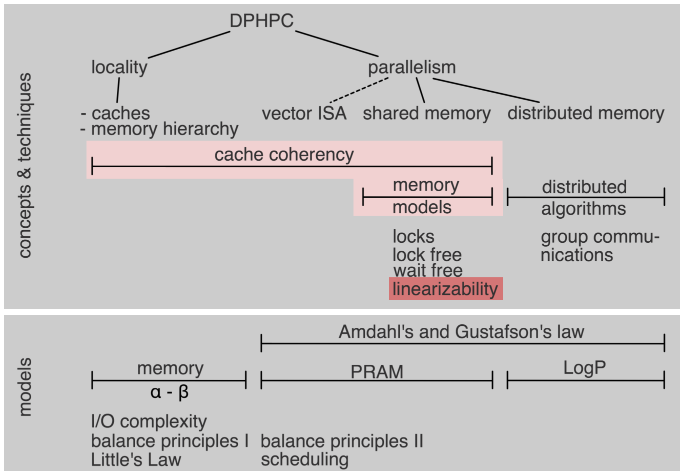
- **Cache-coherence is not enough!**
  - Many more subtle issues for parallel programs!
- **Memory Models**
  - Sequential consistency
  - Why threads cannot be implemented as a library ☺
  - Relaxed consistency models
  - x86 TLO+CC case study
- **Complexity of reasoning about parallel objects**
  - Serial specifications (e.g., pre-/postconditions)
  - Started to lock things …

Peer Quiz



- **Instructions:**
  - Pick some partners (locally) and discuss each question for 4 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? Explain!
  - How would you improve the situation?
- **How could memory models of practical CPUs be described?**
  - Is the Intel definition useful?
  - Why would one need a better definition?
  - Threads cannot be implemented as a library? Why does Pthreads work?

DPHPC Overview



Goals of this lecture

- **Queue:**
  - Locked

    C++ locking (small detour)
  - 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

Recap: x86 Memory model: TLO + CC

- **Total lock order (TLO)**
  - Instructions with "lock" prefix enforce total order across all processors
  - Implicit locking: xchg (locked compare and exchange)
- **Causal consistency (CC)**
  - Write visibility is transitive
- **Eight principles**
  - After some revisions ☺

The Eight x86 Principles

1. **"Reads are not reordered with other reads." (R→R)**
2. **"Writes are not reordered with other writes." (W→W)**
3. **"Writes are not reordered with older reads." (R→W)**
4. **"Reads may be reordered with older writes to different locations but not with older writes to the same location." (NO W→R!)**
5. **"In a multiprocessor system, memory ordering obeys causality (memory ordering respects transitive visibility). (some more orders)**
6. **"In a multiprocessor system, writes to the same location have a total order." (implied by cache coherence)**
7. **"In a multiprocessor system, locked instructions have a total order." (enables synchronized programming!)**
8. **"Reads and writes are not reordered with locked instructions. "(enables synchronized programming!)**

Principle 1 and 2

Reads are not reordered with other reads. (R→R)

Writes are not reordered with other writes. (W→W)

All values zero initially

P1	P2
a = 1 b = 2	r1 = b r2 = a

- If r1 == 2, then r2 must be 1!
- Not allowed: r1 == 1, r2 == 0
- Reads and writes observed in program order
- Cannot be reordered!

Principle 3

Writes are not reordered with older reads. (R→W)

All values zero initially

P1	P2
r1 = a b = 1	r2 = b a = 1

- If r1 == 1, then P2:W(a) → P1:R(a), thus r2 must be 0!
- If r2 == 1, then P1:W(b) → P1:R(b), thus r1 must be 0!
- Not allowed: r1 == 1 and r2 == 1

Principle 4

Reads may be reordered with older writes to different locations but not with older writes to the same location. (NO W→R!)

All values zero initially

P1	P2
a = 1 r1 = b	b = 1 r2 = a

- Allowed: r1=0, r2=0
- Sequential consistency can be enforced with mfence
- Attention: may allow reads to move into critical sections!

Principle 5

In a multiprocessor system, memory ordering obeys causality (memory ordering respects transitive visibility). (some more orders)

All values zero initially

P1	P2	P3
a = 1	r1 = a b = 1	r2 = b r3 = a

- If r1 == 1 and r2==1, then r3 must read 1
- Not allowed: r1 == 1, r2 == 1, and r3 == 0
- Provides some form of atomicity

Principle 6

In a multiprocessor system, writes to the same location have a total order. (implied by cache coherence)

All values zero initially

P1	P2	P3	P4
a=1	a=2	r1 = a r2 = a	r3 = a r4 = a

- Not allowed: r1 == 1, r2 == 2, r3 == 2, r4 == 1
- If P3 observes P1's write before P2's write, then P4 will also see P1's write before P2's write
- Provides some form of atomicity

Principle 7

In a multiprocessor system, locked instructions have a total order.
(enables synchronized programming!)

All values zero initially, registers r1==r2==1

P1	P2	P3	P4
xchg(a,r1)	xchg(b,r2)	r3 = a r4 = b	r5 = b r6 = a

- Not allowed: r3 == 1, r4 == 0, r5 == 1, r6 ==0
- If P3 observes ordering P1:xchg → P2:xchg, P4 observes the same ordering
- (xchg has implicit lock)

Principle 8

Reads and writes are not reordered with locked instructions.
(enables synchronized programming!)

All values zero initially but r1 = r3 = 1

P1	P2
xchg(a,r1) r2 = b	xchg(b,r3) r4 = a

- Not allowed: r2 == 0, r4 == 0
- Locked instructions have total order, so P1 and P2 agree on the same order
- If volatile variables use locked instructions → practical sequential consistency

An Alternative View: x86-TSO

- **Sewell el al.: "x86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors", CACM May 2010**

"[...] **real multiprocessors typically do not provide the sequentially consistent memory** that is assumed by most work on semantics and verification. Instead, they have relaxed memory models, varying in subtle ways between processor families, in which different hardware threads may have only loosely consistent views of a shared memory. Second, **the public vendor architectures, supposedly specifying what programmers can rely on, are often in ambiguous informal prose (a particularly poor medium for loose specifications), leading to widespread confusion**. [...] We present a new x86-TSO programmer's model that, to the best of our knowledge, suffers from none of these problems. **It is mathematically precise** (rigorously defined in HOL4) but can be presented as an **intuitive abstract machine which should be widely accessible to working programmers**. [...]"

Notions of Correctness

- **We discussed so far:**
  - Read/write of the same location
    Cache coherence (write propagation and serialization/atomicity)
  - Read/write of multiple locations
    Memory models (visibility order of updates by cores)

- **Now: objects (variables/fields with invariants defined on them)**
  - Invariants "tie" variables together
  - Sequential objects
  - Concurrent objects

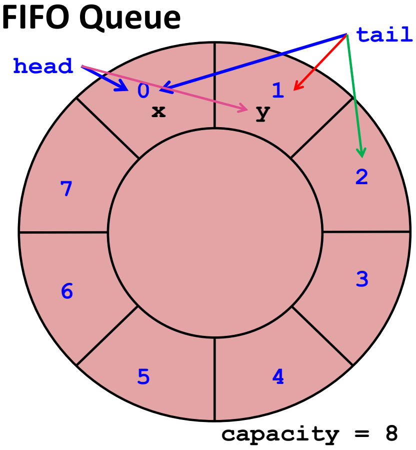
Sequential Objects

- **Each object has a type**
- **A type is defined by a class**
  - Set of fields forms the state of an object
  - Set of methods (or free functions) to manipulate the state

- **Remark**
  - An Interface is an abstract type that defines behavior
    A class implementing an interface defines several types

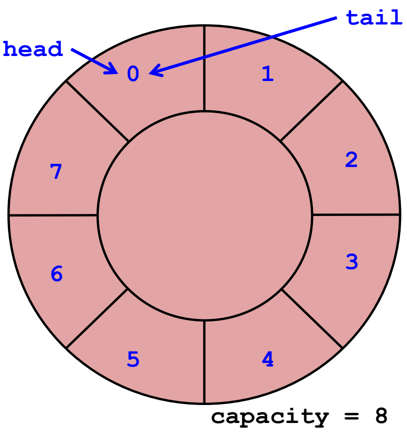
Running Example: FIFO Queue

- **Insert elements at tail**
- **Remove elements from head**
  - Initial: head = tail = 0
  - enq(x)
  - enq(y)
  - deq() [x]
  - …



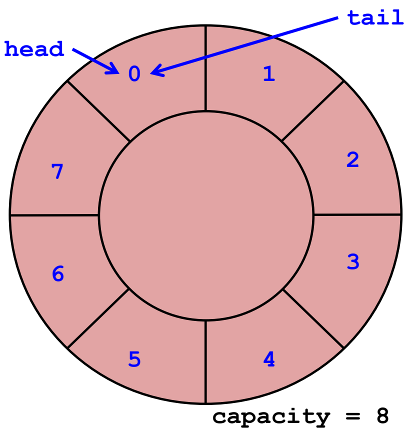
Sequential Queue

```
class Queue {

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

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



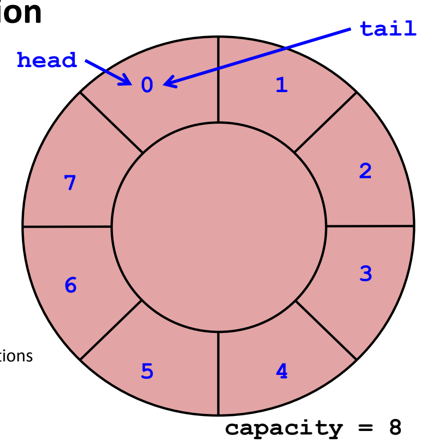
Sequential Queue

```
class Queue {
 ...

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

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



Sequential Execution

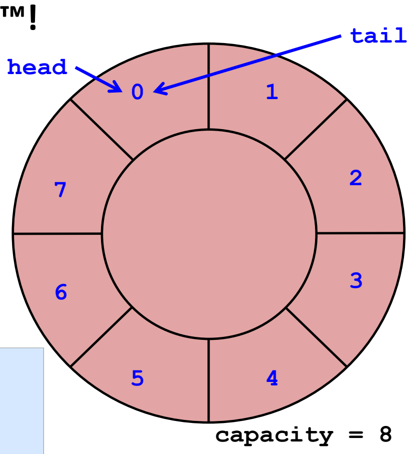
- **(The) one process executes operations one at a time**
  - Sequential ☺
- **Semantics of operation defined by specification of the class**
  - Preconditions and postconditions



Design by Contract™!

- **Preconditions:**
  - Specify conditions that must hold before method executes
  - Involve state and arguments passed
  - Specify obligations a client must meet before calling a method
- **Example: enq()**
  - Queue must not be full!

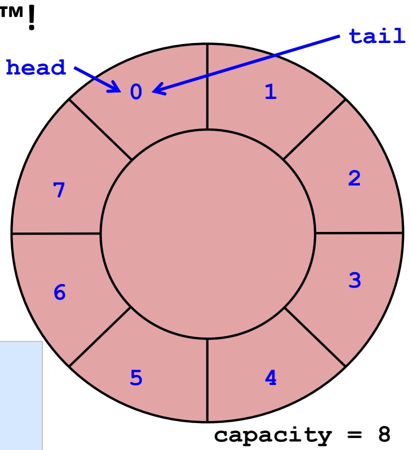
```
class Queue {
 ...
 void enq(Item x) {
   assert(tail-head < items.size()-1);
   ...
 }
};
```



Design by Contract™!

- **Postconditions:**
  - Specify conditions that must hold after method executed
  - Involve old state and arguments passed
- **Example: enq()**
  - Queue must contain element!

```
class Queue {
 ...
 void enq(Item x) {
  ...                    creative assertion ☺
  assert( (tail == old tail + 1) &&
        (items[old tail] == x) );
 }
};
```



Sequential specification

- **if(precondition)**
  - Object is in a **specified state**
- **then(postcondition)**
  - The method returns a particular value or
  - Throws a particular exception **and**
  - Leaves the object in a specified state
- **Invariants**
  - Specified conditions (e.g., object state) must hold **anytime** a client could invoke an objects method!

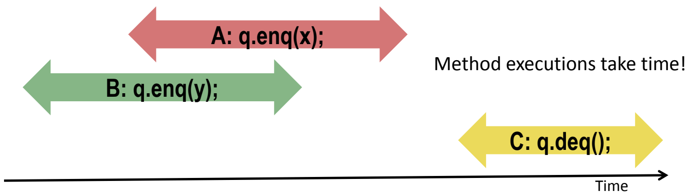
Advantages of sequential specification

- **State between method calls is defined**
  - Enables reasoning about objects
  - Interactions between methods captured by side effects on object state
- **Enables reasoning about each method in isolation**
  - Contracts for each method
  - Local state changes global state
- **Adding new methods**
  - Only reason about state changes that the new method causes
  - If invariants are kept: **no need to check old methods**
  - **Modularity!**

Concurrent execution - State

- **Concurrent threads invoke methods on possibly shared objects**
  - At overlapping time intervals!

Property	Sequential	Concurrent
State	Meaningful and clearly defined between method executions	Overlapping method executions → object may never be "between method executions"



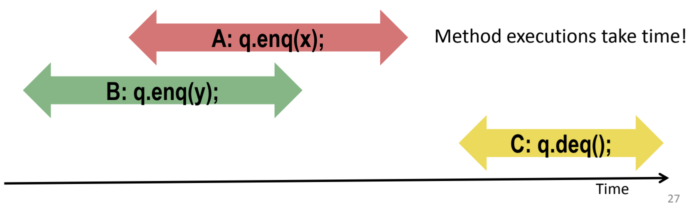
Method executions take time!

Concurrent execution - Reasoning

- **Reasoning must now include all possible interleavings**
  - Of changes caused by methods themselves

Property	Sequential	Concurrent
Reasoning	Consider each method in isolation; invariants on state before/after execution.	Need to consider all possible interactions; all intermediate states during execution

- Consider: enq() || enq() and deq() || deq() and deq() || enq()



Method executions take time!

Concurrent execution - Method addition

- **Reasoning must now include all possible interleavings**
  - Of changes caused by and methods themselves

Property	Sequential	Concurrent
Add Method	Without affecting other methods; invariants on state before/after execution.	Everything can potentially interact with everything else

- **Consider adding a method that returns the last item enqueued**

```
Item peek() {
  if(last-head == 0)  throw Exception;
  return items[(tail-1) % items.size()];
}
```

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

```
Item deq() {
  Item item = items[head];
  head = (head+1)%items.size();
}
```

  - peek() || enq(): what if tail has not yet been incremented?
  - peek() || deq(): what if last item is being dequeued?

Concurrent objects

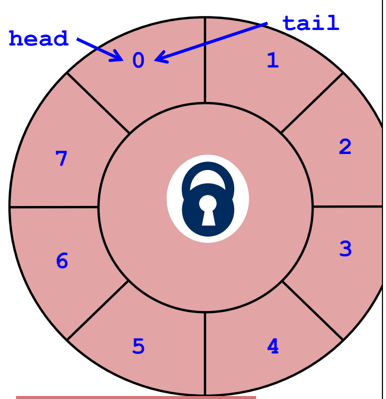
- **How do we describe one?**
  - No pre-/postconditions ☹
- **How do we implement one?**
  - Plan for exponential number of interactions
- **How do we tell if an object is correct?**
  - Analyze all exponential interactions
  - Wait, what? Exponential? Why?

    Dependencies could form circles with diameter > 2

Is it time to panic for software engineers?
Who has a solution?

Lock-based queue

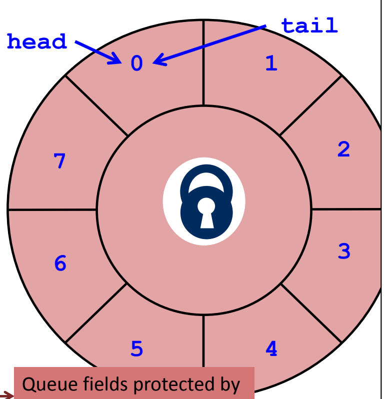
```
class Queue {

 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)%size==head) {
      throw FullException;
    }
    items[tail % items.size()] = x;
    tail = (tail+1)%items.size();
  }

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



Queue fields protected by single shared lock!

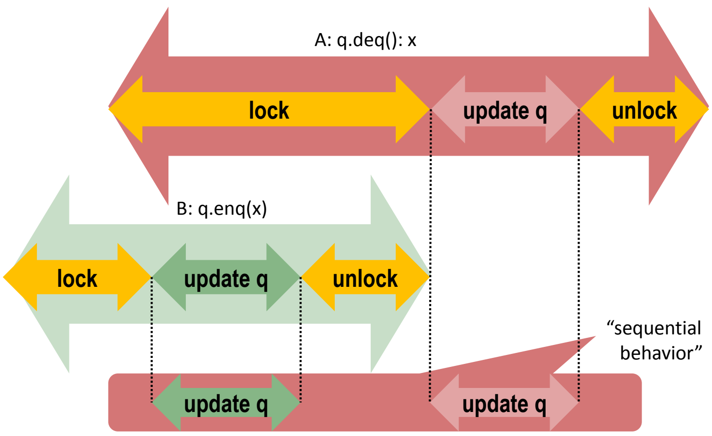
Class question: how is the lock ever unlocked?

C++ Resource Acquisition is Initialization

- **Detour – RAII – suboptimal name**
- **Can be used for locks (or any other resource acquisition)**
  - Constructor grabs resource
  - Destructor frees resource
- **Behaves as if**
  - Implicit unlock at end of block!
- **Main advantages**
  - Always free lock at exit
  - No "lost" locks due to exceptions or strange control flow (goto ☺)
  - Very easy to use

```
class lock_guard<typename mutex_impl> {
  mutex_impl &_mtx;  // ref to the mutex

  public:
    scoped_lock(mutex_impl & mtx ) : _mtx(mtx) {
      _mtx.lock();  // lock  mutex in constructor
    }
    ~scoped_lock() {
      _mtx.unlock();  // unlock mutex in destructor
    }
};
```


Example execution



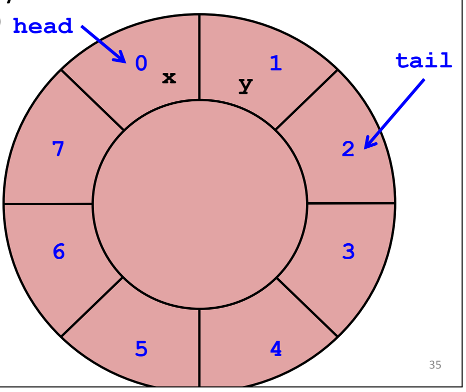
Correctness

- **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 ☺

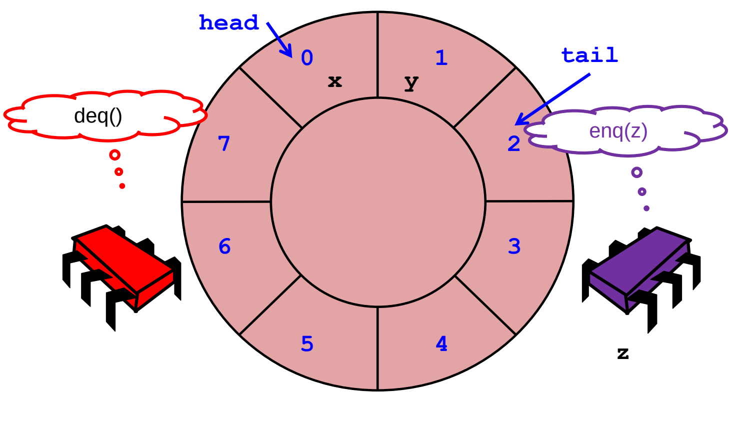
It does not scale!
What is the solution here?

Threads working at the same time?

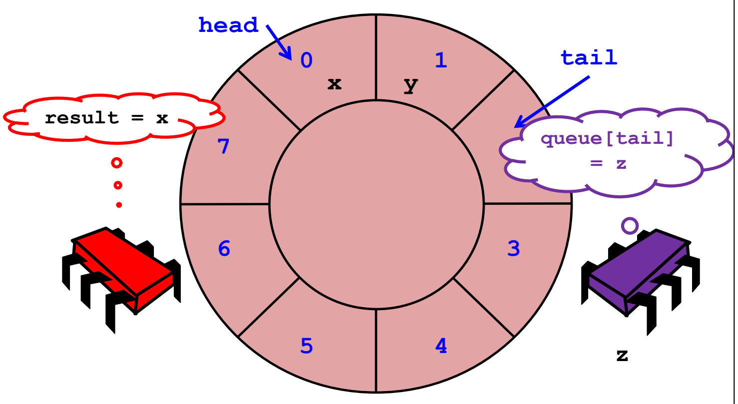
- **Same thing (concurrent queue)**
- **For simplicity, assume only two threads**
  - Thread A calls only enq()
  - Thread B calls only deq()



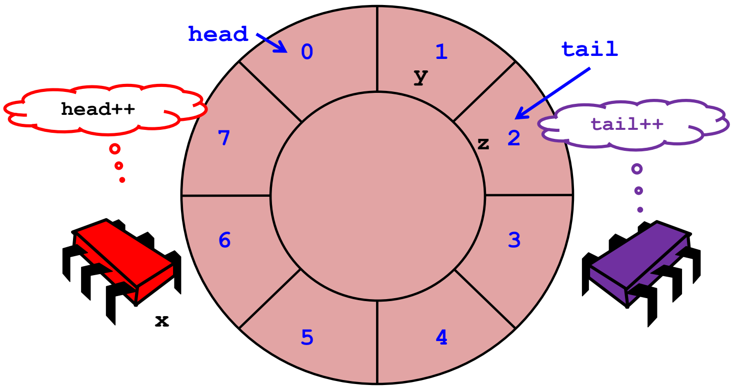
Wait-free 2-Thread Queue



Wait-free 2-Thread Queue



Wait-free 2-Thread Queue



Is this correct?

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

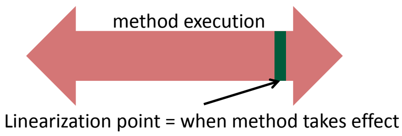
```
void enq(Item x) {
  if((tail+1)%size==head) {
    throw FullException;
  }
  items[tail] = x;
  tail = (tail+1)%items.size();
}
```

```
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 it's "sequential" behavior is correct**
  - Called "linearizable"



Linearizability

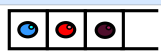
- **Sounds like a property of an execution …**
- **An object is called linearizable if all possible executions on the object are linearizable**
- **Says nothing about the order of executions!**

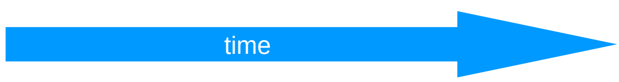
Example

```
void enq(Item x) {
  std::lock_guard<std::mutex> l(lock)
  if((tail+1)%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

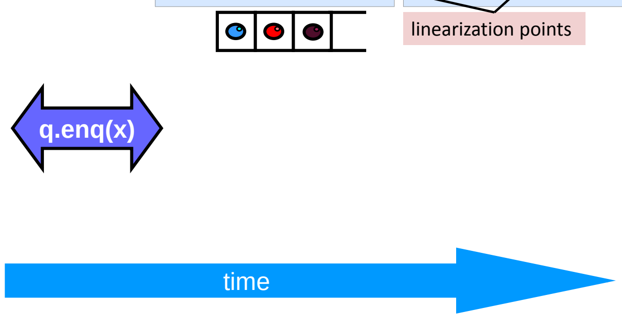




Example

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

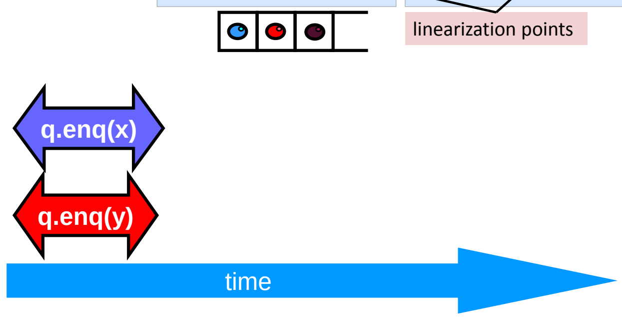
```
Item deq() {
  std::lock_guard<std::mutex> l(lock)
  if(tail == head) {
    throw EmptyException;
  }
  Item item = items[head];
  head = (head+1)%items.size();
}
```



Example

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

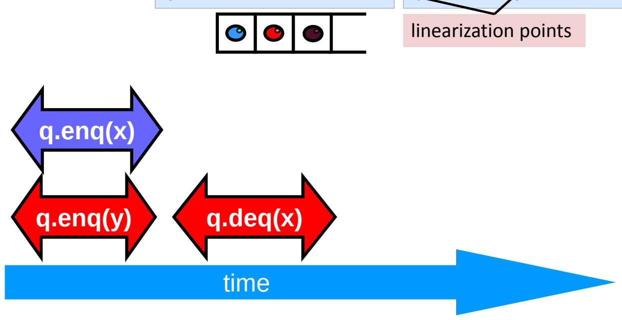
```
Item deq() {
  std::lock_guard<std::mutex> l(lock)
  if(tail == head) {
    throw EmptyException;
  }
  Item item = items[head];
  head = (head+1)%items.size();
}
```



Example

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

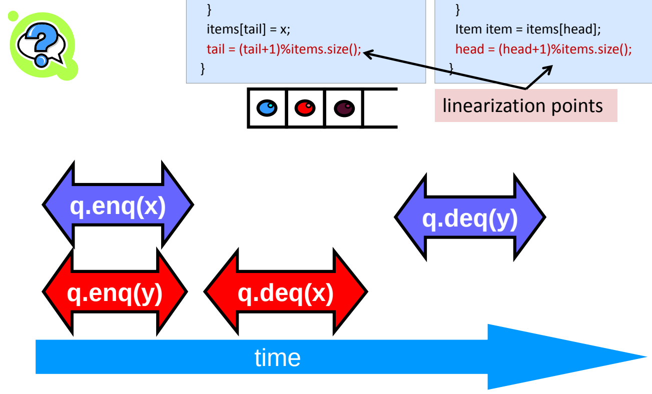
```
Item deq() {
  std::lock_guard<std::mutex> l(lock)
  if(tail == head) {
    throw EmptyException;
  }
  Item item = items[head];
  head = (head+1)%items.size();
}
```



Example

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

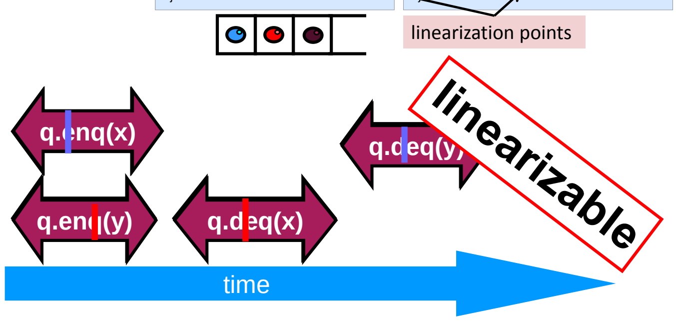
```
Item deq() {
  std::lock_guard<std::mutex> l(lock)
  if(tail == head) {
    throw EmptyException;
  }
  Item item = items[head];
  head = (head+1)%items.size();
}
```



Example

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

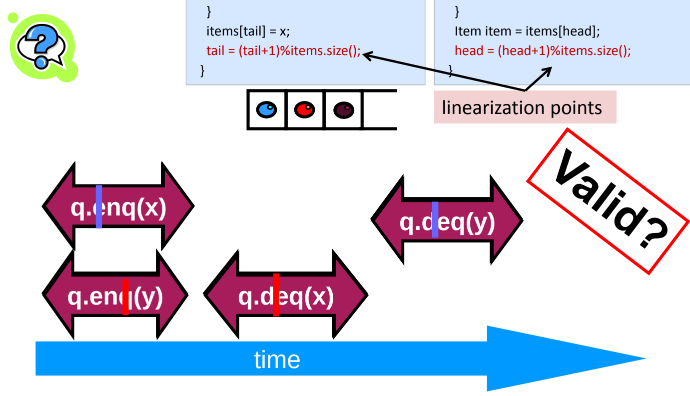
```
Item deq() {
  std::lock_guard<std::mutex> l(lock)
  if(tail == head) {
    throw EmptyException;
  }
  Item item = items[head];
  head = (head+1)%items.size();
}
```



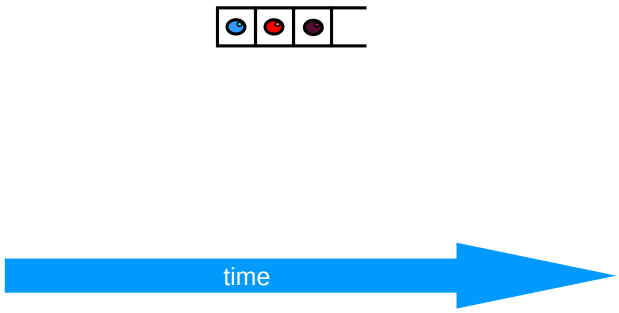
Example

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

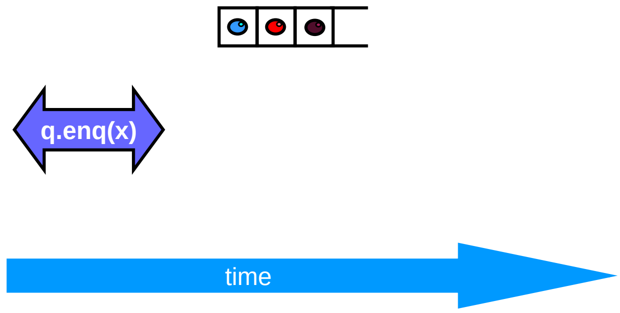
```
Item deq() {
  std::lock_guard<std::mutex> l(lock)
  if(tail == head) {
    throw EmptyException;
  }
  Item item = items[head];
  head = (head+1)%items.size();
}
```



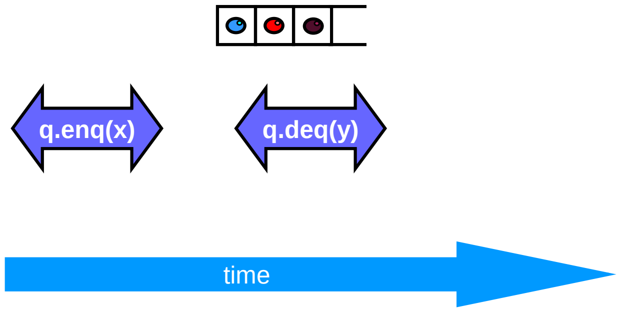
Example 2



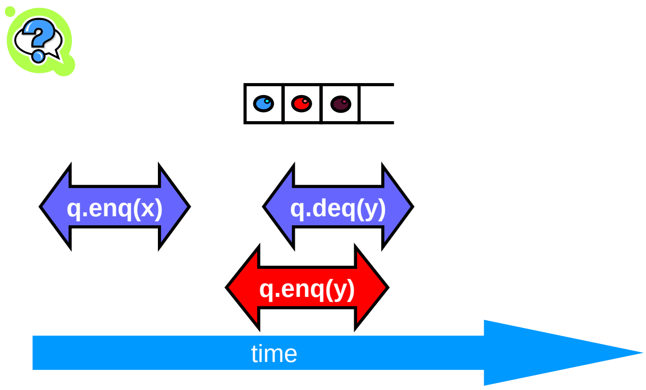
Example 2



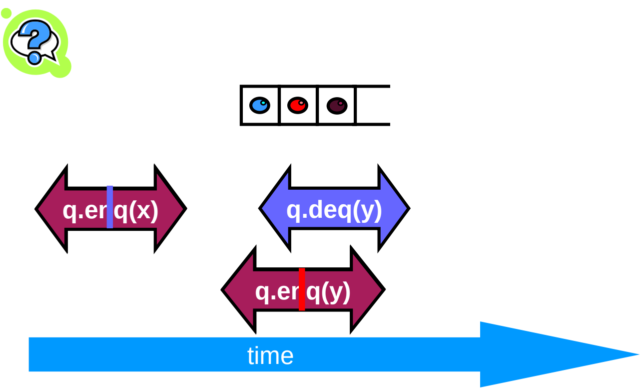
Example 2



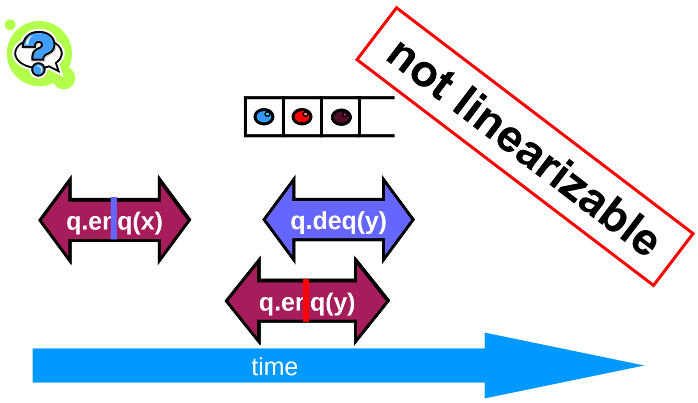
Example 2



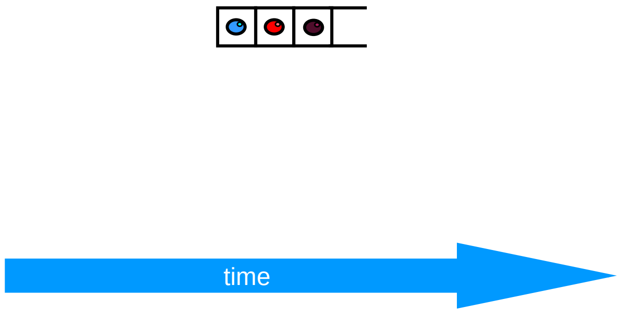
Example 2



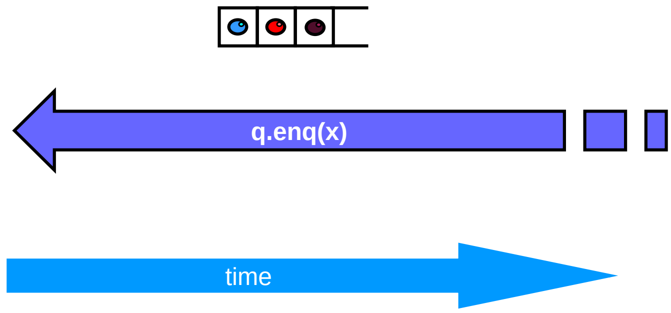
Example 2



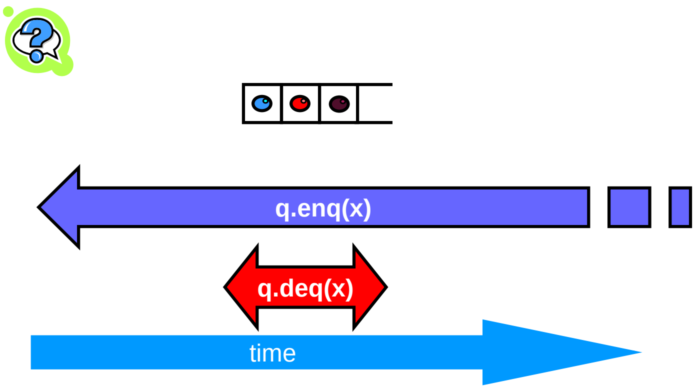
Example 3



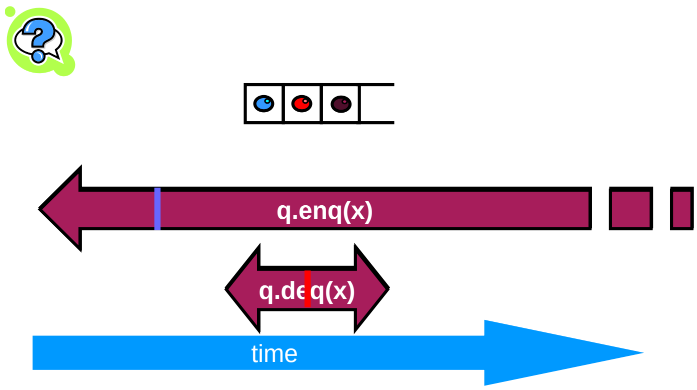
Example 3



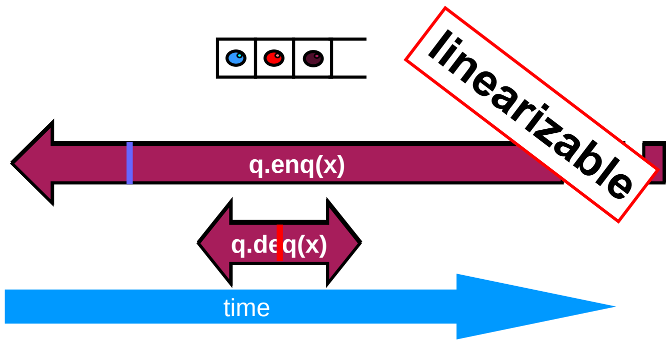
Example 3



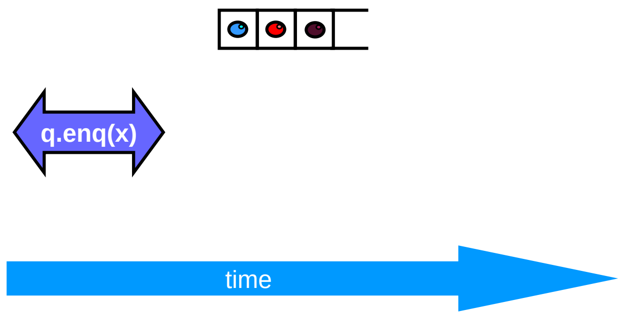
Example 3



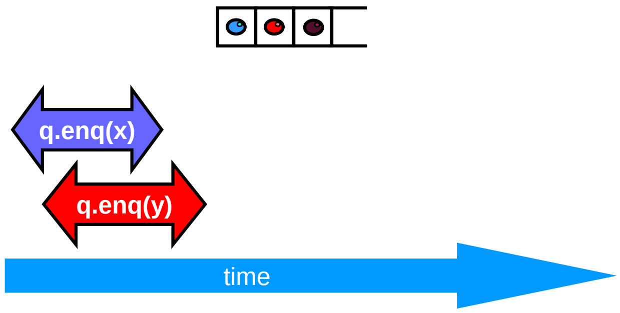
Example 3



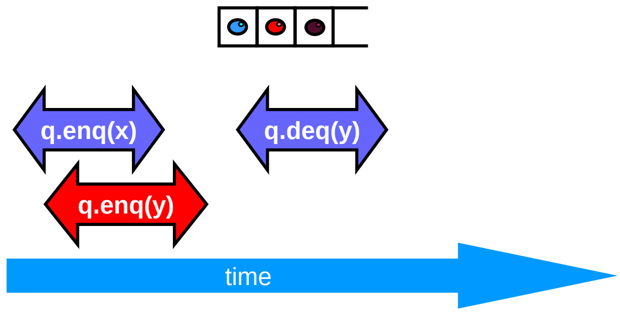
Example 4



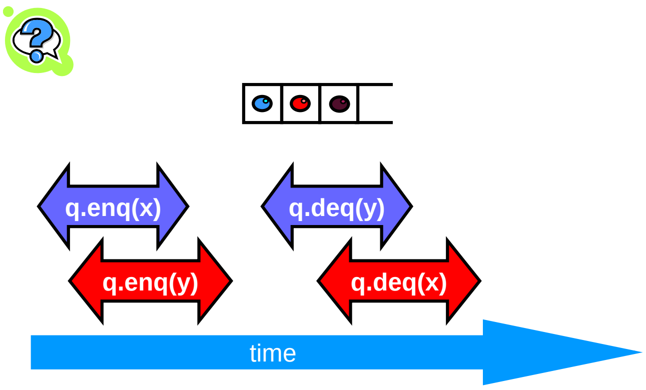
Example 4



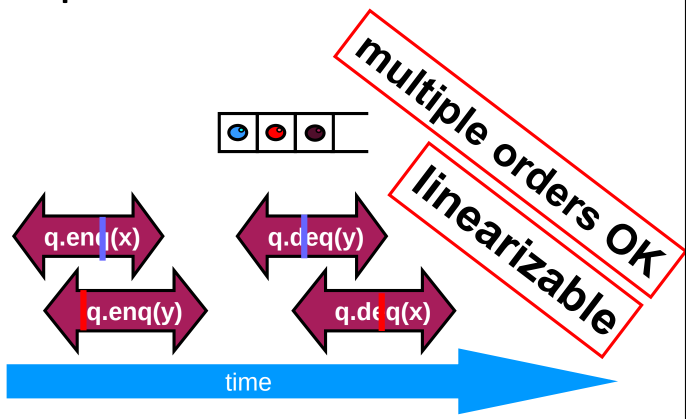
Example 4



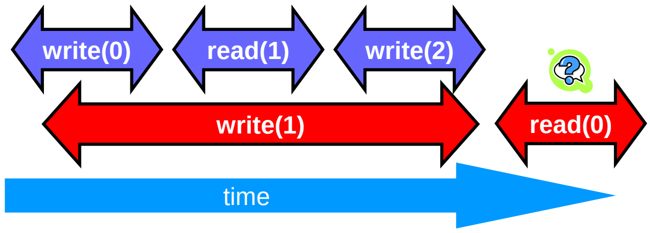
Example 4



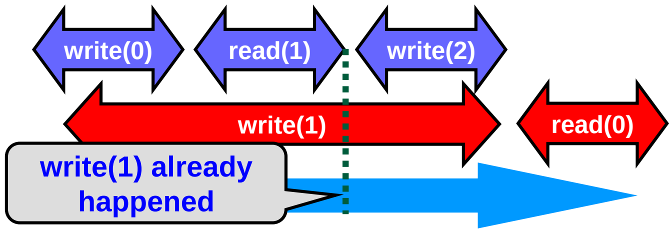
Example 4



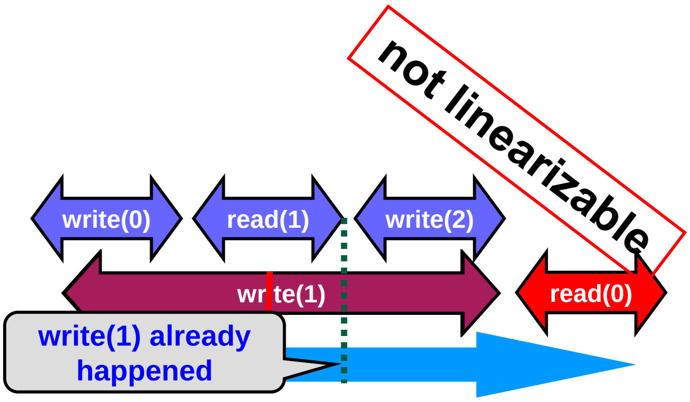
Read/Write Register Example



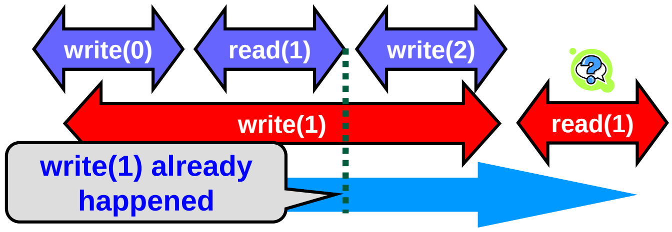
Read/Write Register Example



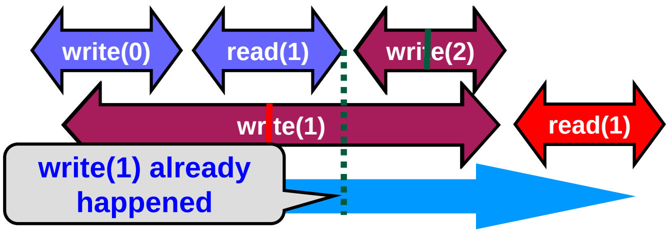
Read/Write Register Example



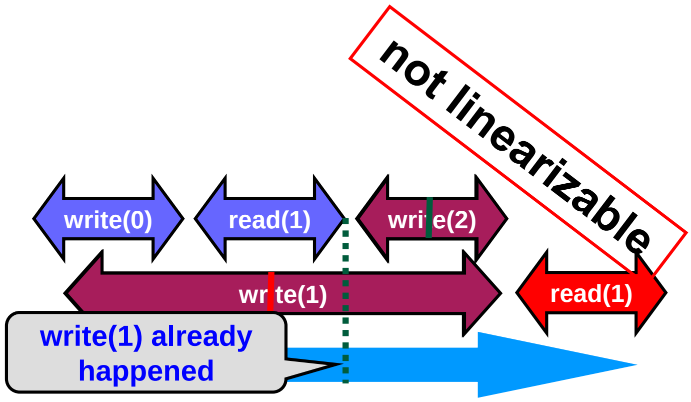
Read/Write Register Example



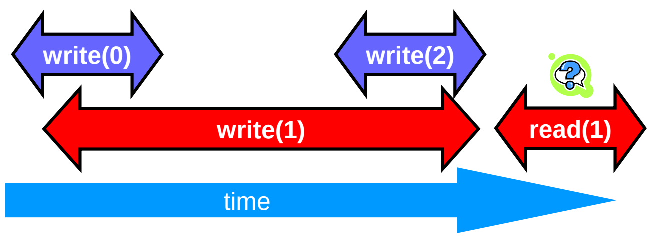
Read/Write Register Example



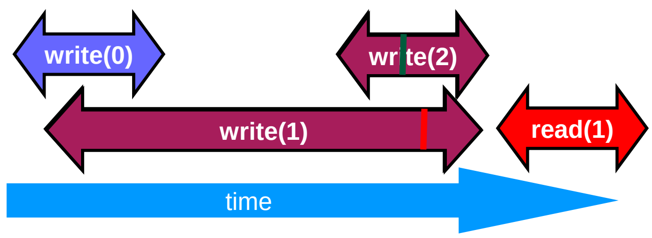
Read/Write Register Example



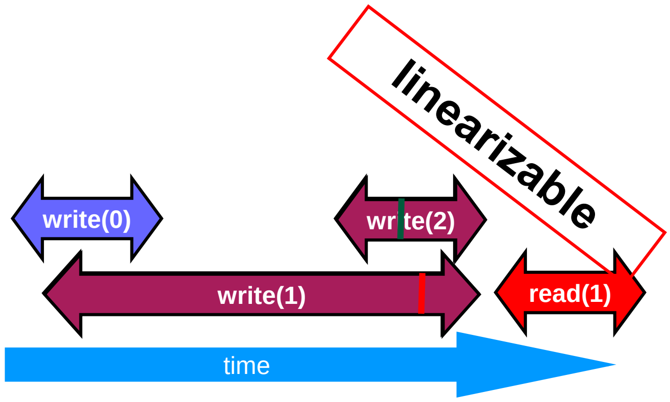
Read/Write Register Example



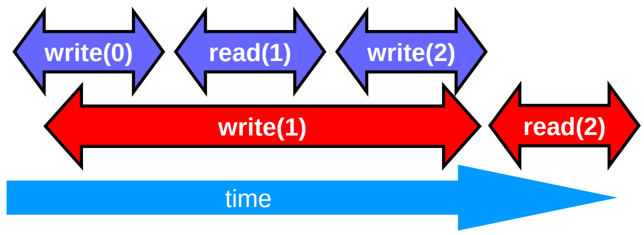
Read/Write Register Example



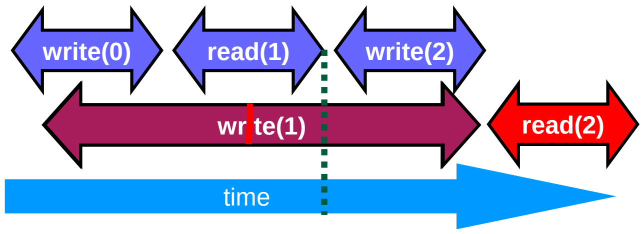
Read/Write Register Example



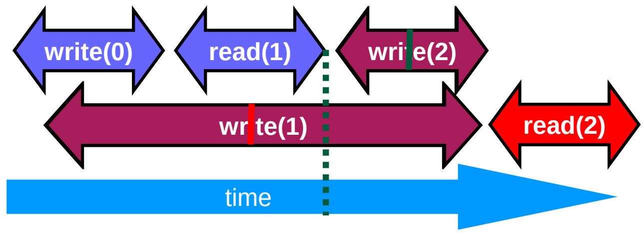
Read/Write Register Example



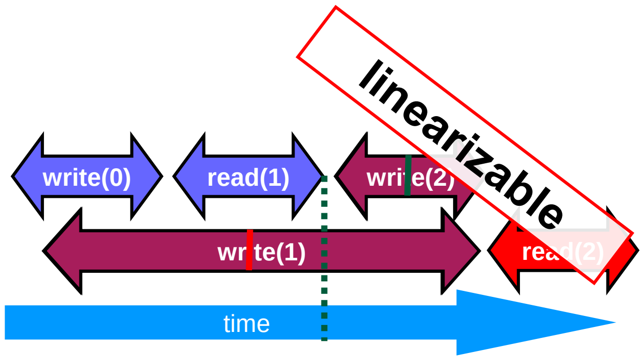
Read/Write Register Example



Read/Write Register Example



Read/Write Register Example



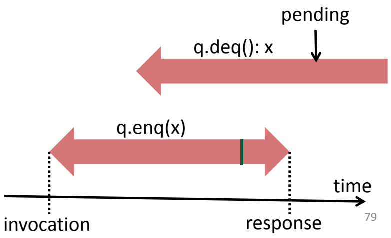
About Executions

- **Why?**
  - Can't we specify the linearization point of each operation 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

  - 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

Note: 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()
...

- 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

a history is sequential if

- 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

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

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

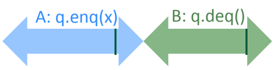
- **Sequential specification allows to describe what behavior we expect and tolerate**
  - When is a single-thread, single-object history legal?
- **Recall: Example**
  - Preconditions and Postconditions
  - Many others exist!
- **A sequential (multi-object) history H is legal if**
  - For every object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

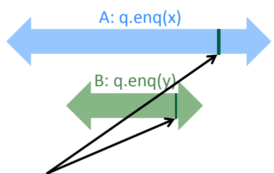


Precedence vs. Overlapping

- **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=	G=	I=
A: q.enq(x)	A: q.enq(x)	A: q.enq(x)
B: p.enq(y)	B: p.enq(y)	B: p.enq(y)
B: p:void	B: p:void	B: p:void
B: q.deq()	B: q.deq()	B: q.deq()
A: q:void	A: q:void	A: q:void
B: q:x	A: q.enq(z)	B: q:x
	B: q:x	B: q.deq()
Complete	Not complete	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

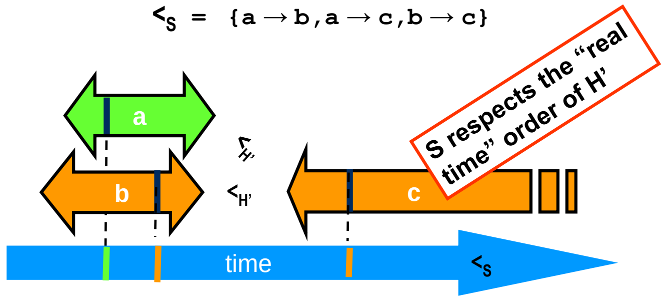
- **A history H induces a strict partial order $<_H$ on operations**
  - $m_0 <_H m_1$ if $m_0 \rightarrow_H m_1$
- **A history H is linearizable if**
  - H can be extended to a complete history H'

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

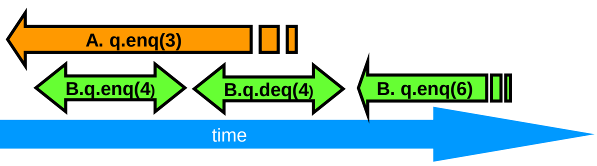
- **Find an S that contains H'**

$$<_{H'} = \{a \rightarrow c, b \rightarrow c\}$$
$$<_S = \{a \rightarrow b, a \rightarrow c, b \rightarrow c\}$$



Example

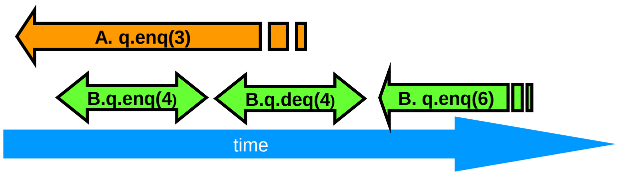
```
A q.enq(3)
B q.enq(4)
B q:void
B q.deq()
B q:4
B q:enq(6)
```



Example

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

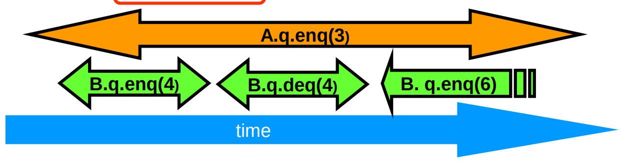
Complete this pending invocation



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

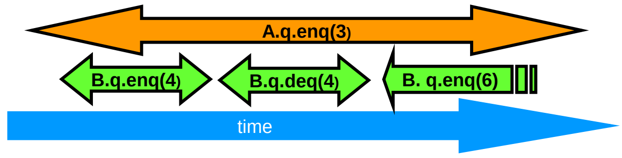
Complete this pending invocation



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

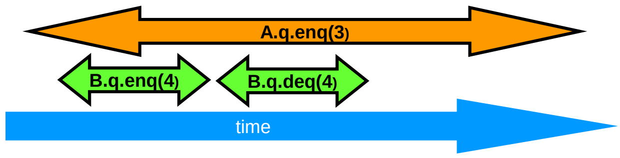


Example

discard this one

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

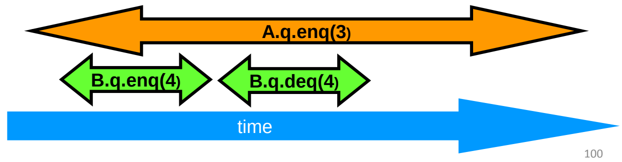
A q:void



Example

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

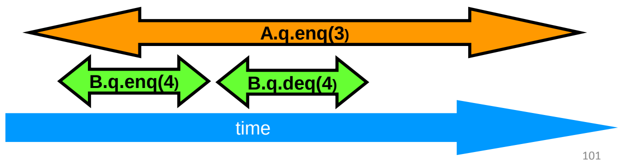
What would be an equivalent sequential history?



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

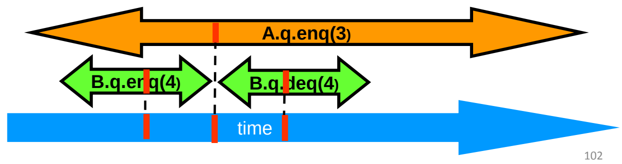


Example

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



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 ensure pre-/postconditions
- **All possible executions must be linearizable**

Now assuming wait-free two-thread queue?

```
void enq(Item x) {
  std::lock_guard<std::mutex> l(lock);
  if(tail-head == items.size()) {
    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?

Composition

- **H is linearizable iff for every object x, H|x is linearizable!**
  - 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

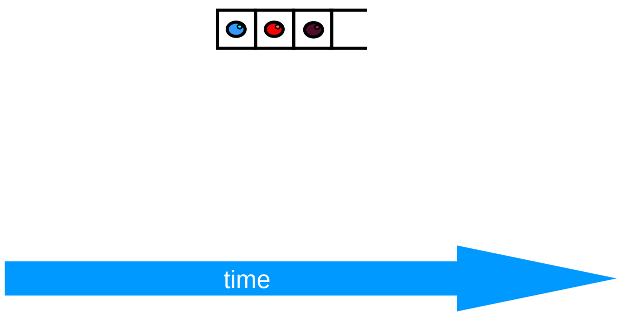
- **Sequential consistency**
  - Correctness condition
  - For describing hardware memory interfaces
  - Remember: not *actual* ones!
- **Linearizability**
  - Stronger correctness condition
  - For describing higher-level systems composed from linearizable components

    Requires understanding of object semantics

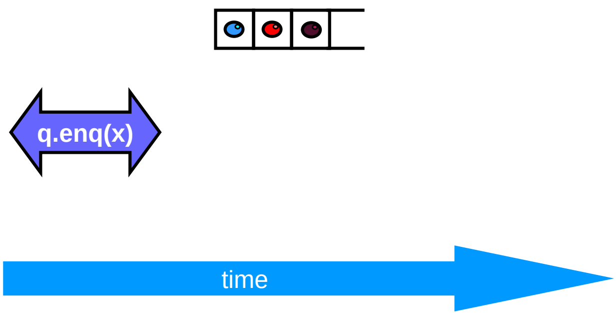
Map linearizability to sequential consistency

- **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

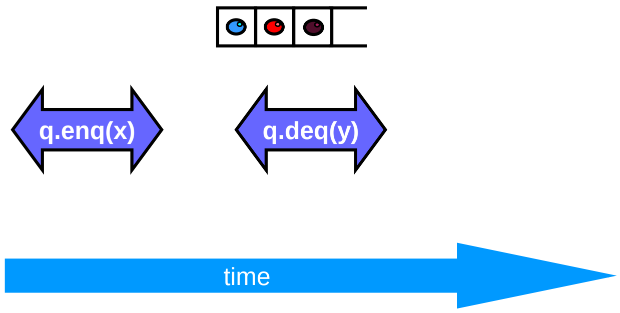
Example



Example

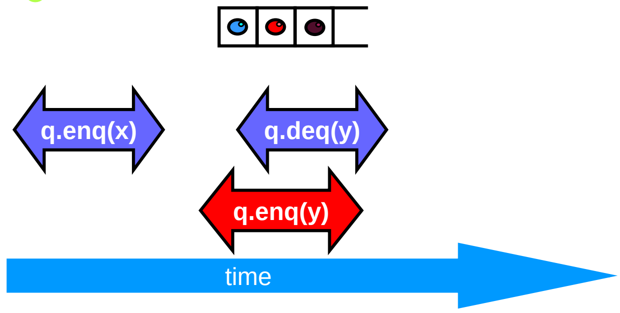


Example



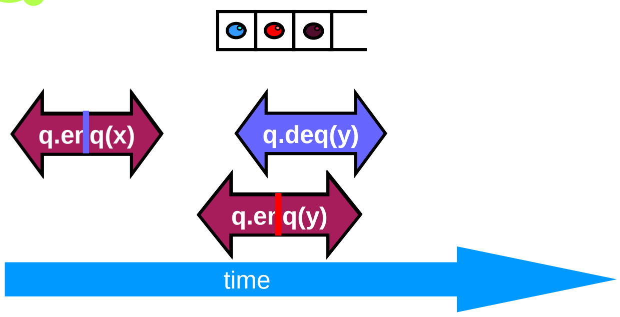
Example

Linearizable?



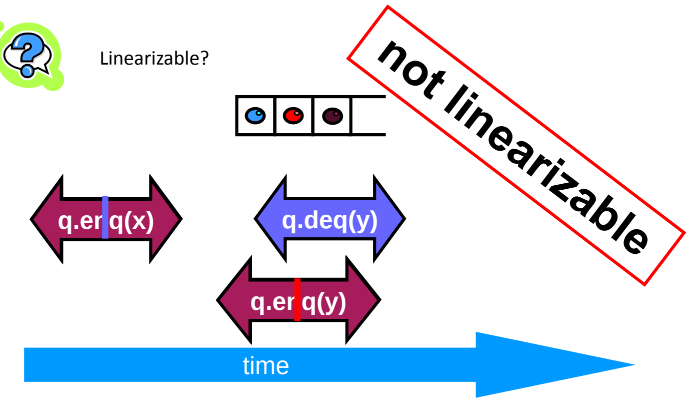
Example

Linearizable?



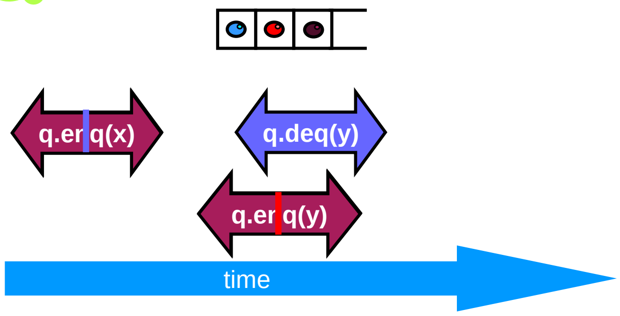
Example

Linearizable?



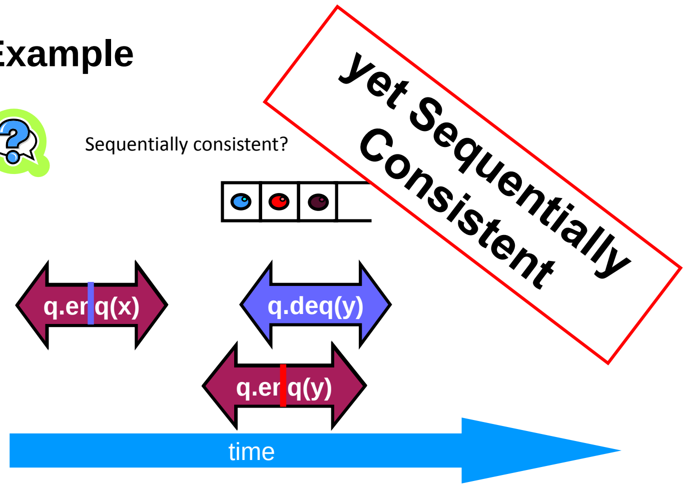
Example

Sequentially consistent?



Example

Sequentially consistent?



Properties of sequential consistency

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

H=

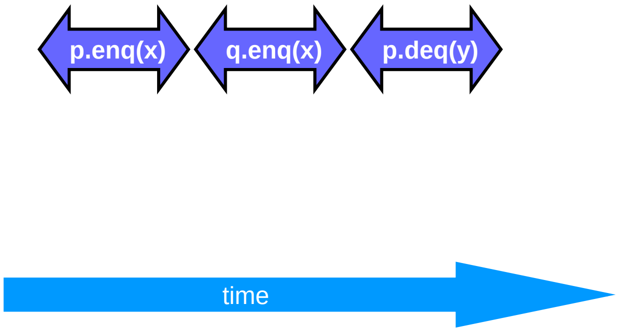
A: p.enq(x)
A: p:void
B: q.enq(y)
B: q:void
A: q.enq(x)
A: q:void
B: p.enq(y)
B: p:void
A: p.deq()
A: p:y
B: q.deq()
B: q:x

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

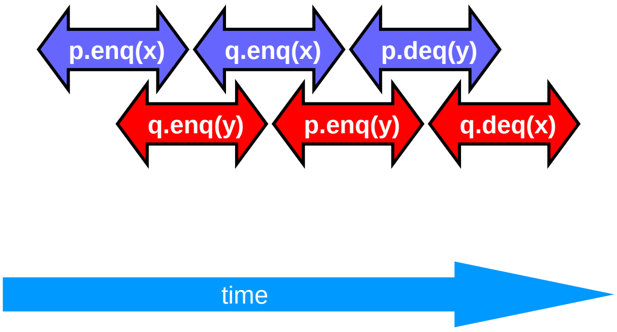
This is not guaranteed for SC schedules!

See following example!

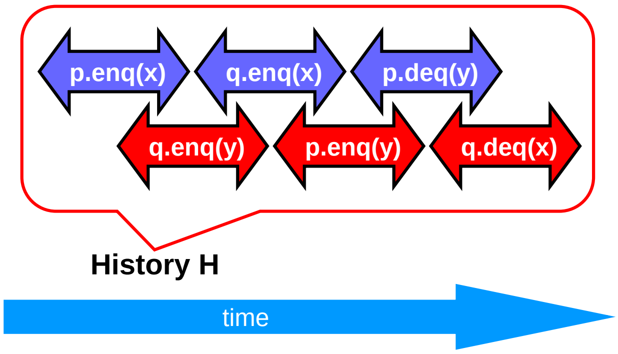
FIFO Queue Example



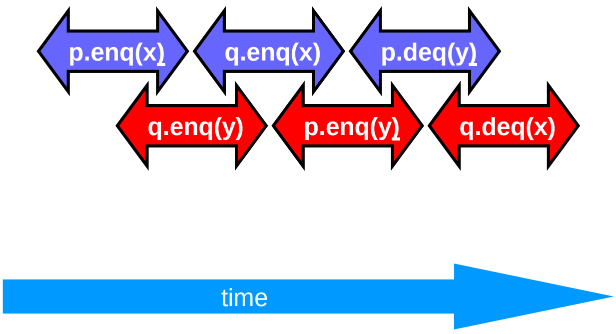
FIFO Queue Example



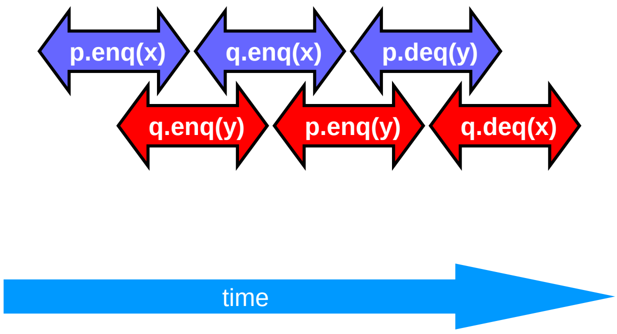
FIFO Queue Example



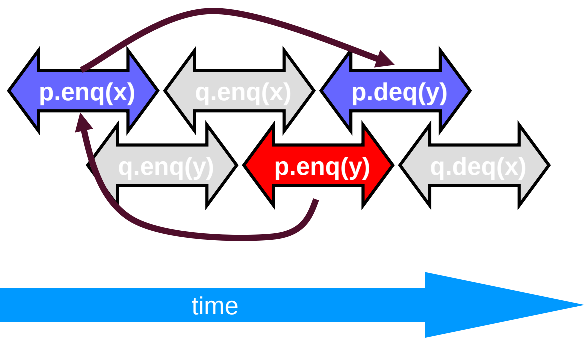
H|p Sequentially Consistent



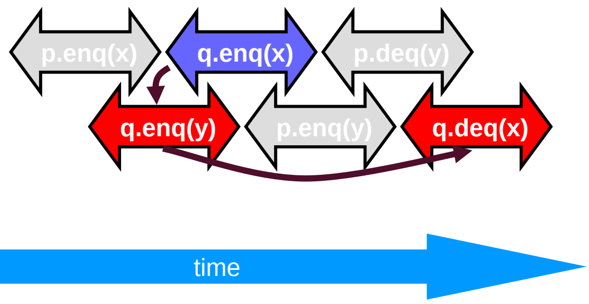
H|q Sequentially Consistent



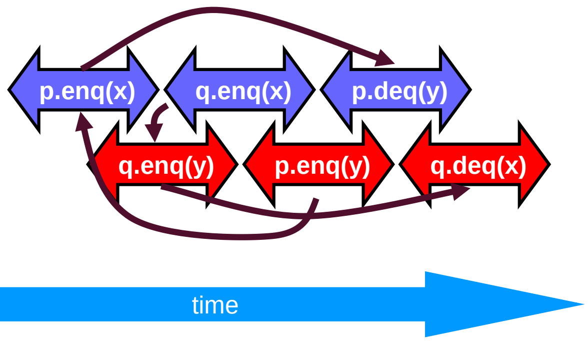
Ordering imposed by p



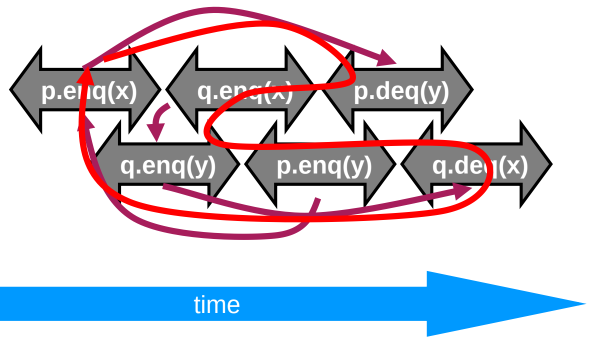
Ordering imposed by q



Ordering imposed by both



Combining orders



Example in our notation

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

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

H|p is sequentially consistent!

Example in our notation

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

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

H|q is sequentially consistent!

Example in our notation

- **Sequential consistency is not compositional**

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

H is not sequentially consistent!

Correctness: Linearizability

- **Sequential Consistency**
  - Not composable
  - Harder to work with
  - Good way to think about hardware models
- **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**
  - 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**
  - Java 1.5 (JSR-133)
  - Still some issues (operational semantics definition)
- **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

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)