

Design of Parallel and High-Performance Computing

Fall 2013

Lecture: Linearizability

Instructor: Torsten Hoefler & Markus Püschel

TA: Timo Schneider

ETH

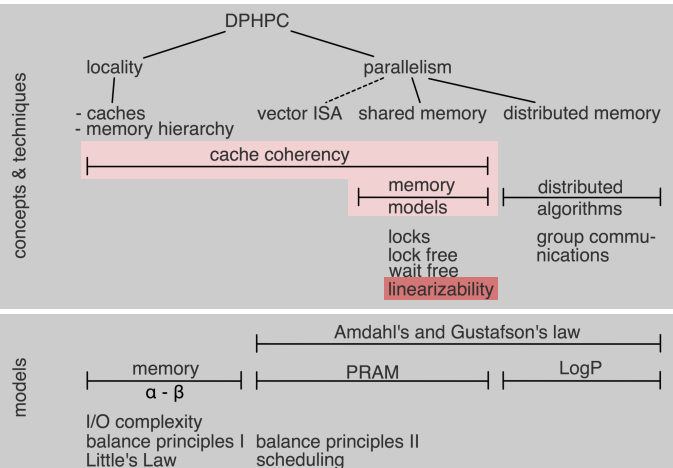
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zürich

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

2

DPHPC Overview



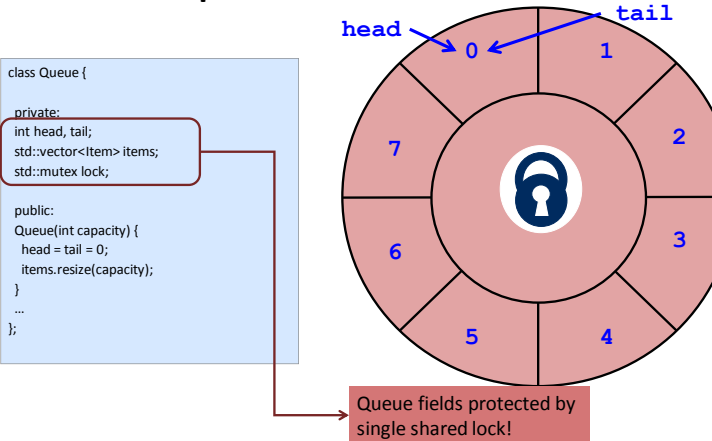
3

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

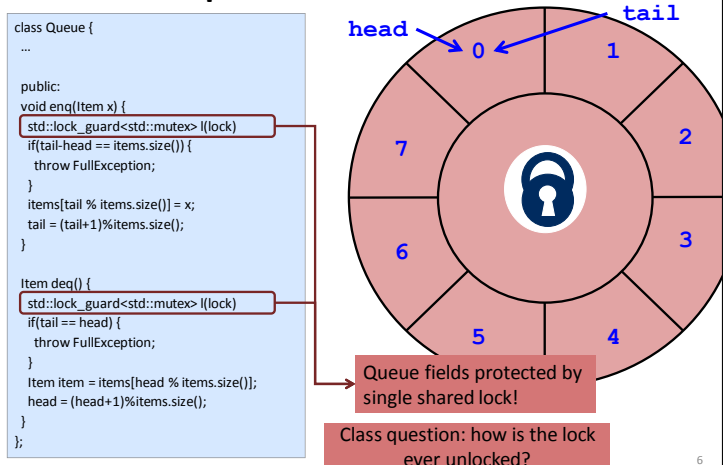
4

Lock-based queue



5

Lock-based queue



6

C++ Resource Acquisition is Initialization

- RAII – suboptimal name
- Can be used for locks (or any other resource acquisitions)

- Constructor grabs resource
- Destructor frees resource

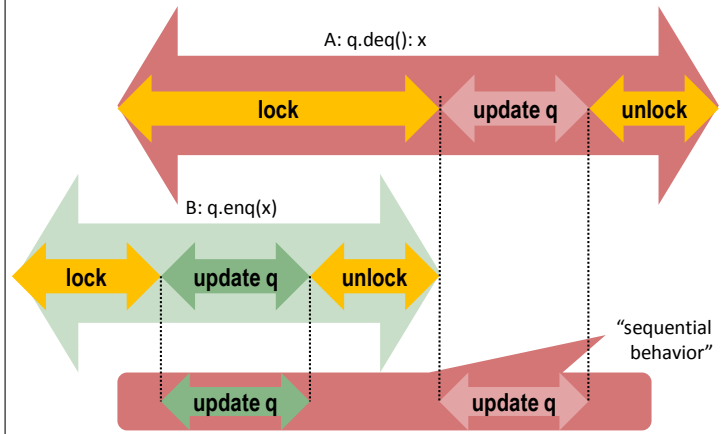
```
class lock_guard<typename mutex_impl> {
    mutex_impl &_mtx; // ref to the mutex

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

- 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

7

Example execution



8

Correctness

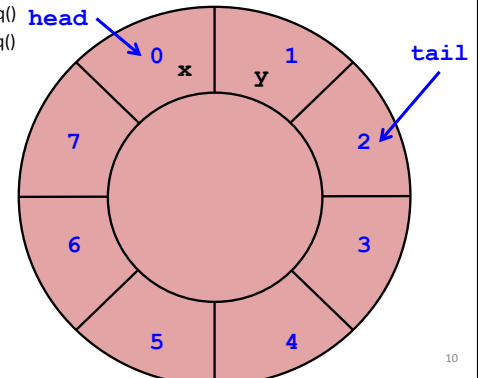
- Is the locked queue correct?
 - Yes, only one thread has access if locked correctly
 - Allows us again to reason about pre- and postconditions
 - Smells a bit like sequential consistency, no?
- Class question: What is the problem with this approach?
 - Same as for SC ☹

It does not scale!
What is the solution here?

9

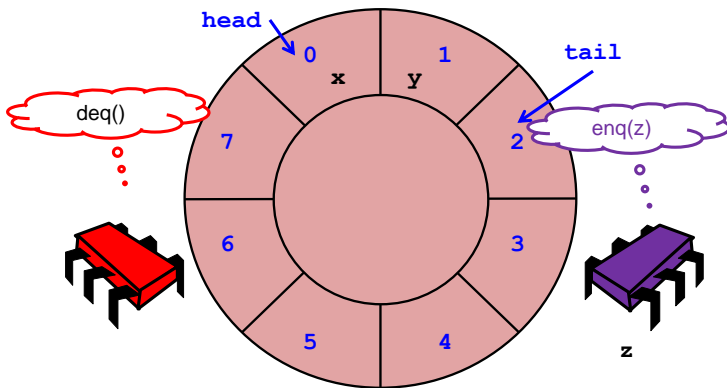
Threads working at the same time?

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

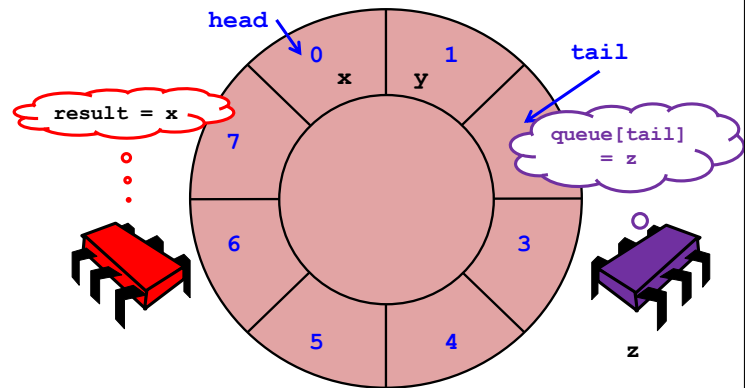


10

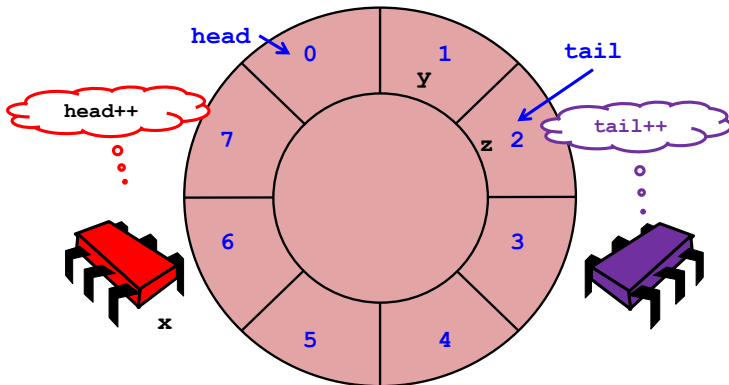
Wait-free 2-Thread Queue



Wait-free 2-Thread Queue



Wait-free 2-Thread Queue



Is this correct?

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

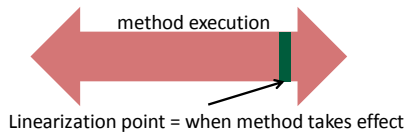
```
void enq(Item x) {
    if (tail - head == items.size()) {
        throw FullException();
    }
    items[tail % items.size()] = x;
    tail = (tail + 1) % items.size();
}
```

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

- 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 concept to concurrency: **linearizability**
- Each method should "take effect"
 - Instantaneously
 - Between invocation and response events
- Concurrent object is correct if this "sequential" behavior is correct
 - Called "linearizable"



Linearizability

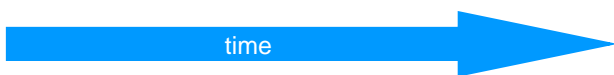
- Sounds like a property of an execution ...
- An object is called linearizable if all possible executions on the object are linearizable
- Says nothing about the order of executions!

Example

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

linearization points

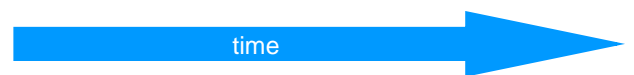
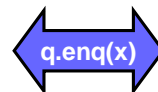


Example

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

linearization points



Example

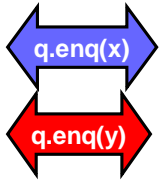
```

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



linearization points

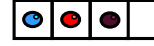


Example

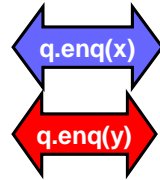
```

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



linearization points



Example



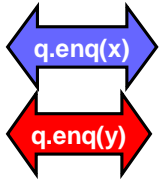
```

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



linearization points



Example

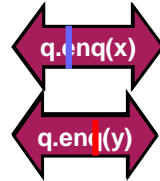
```

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



linearization points



linearizable

Example



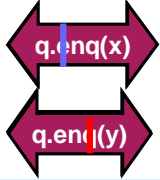
```

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



linearization points

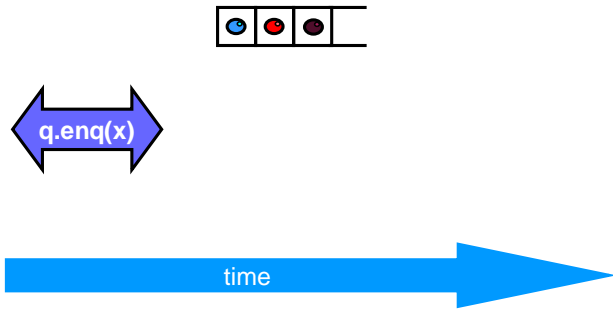


Valid?

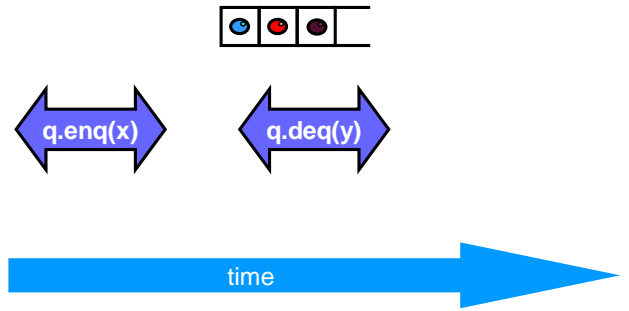
Example 2



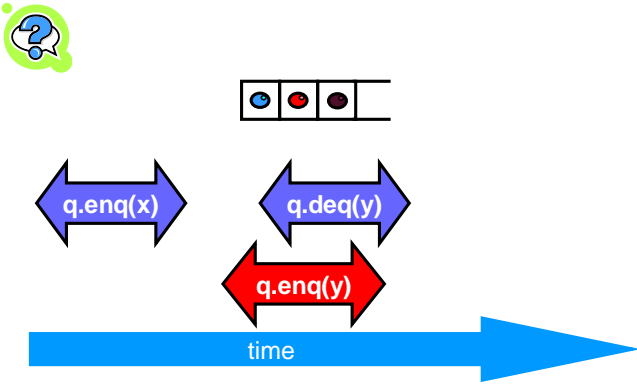
Example 2



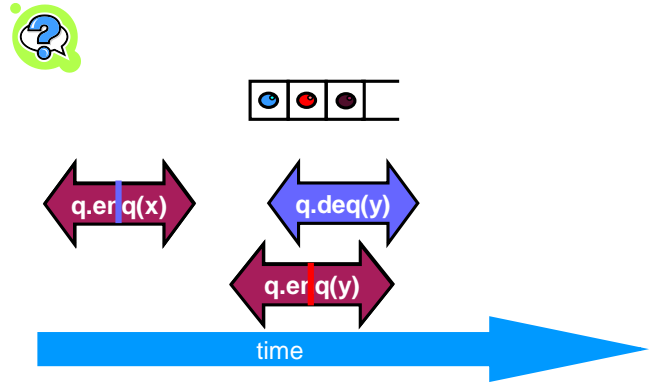
Example 2



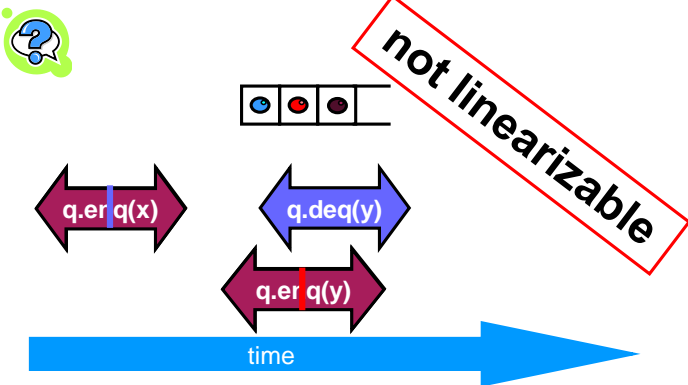
Example 2



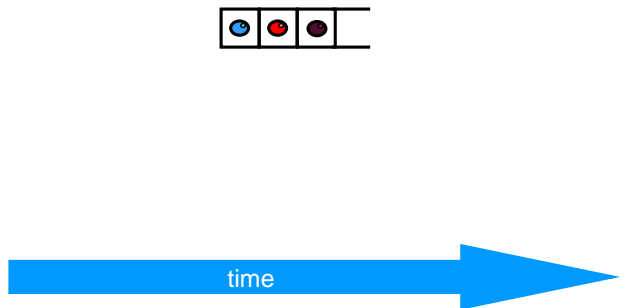
Example 2



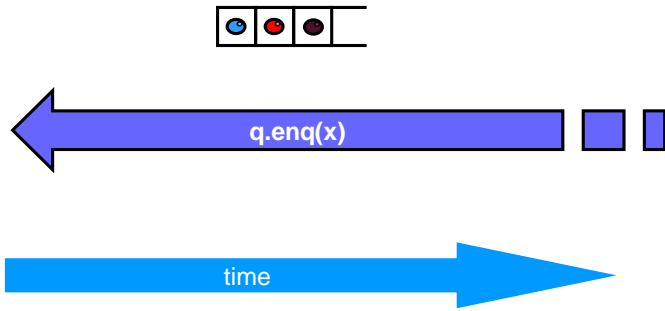
Example 2



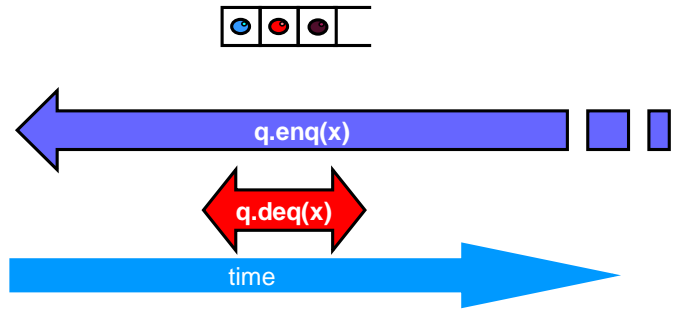
Example 3



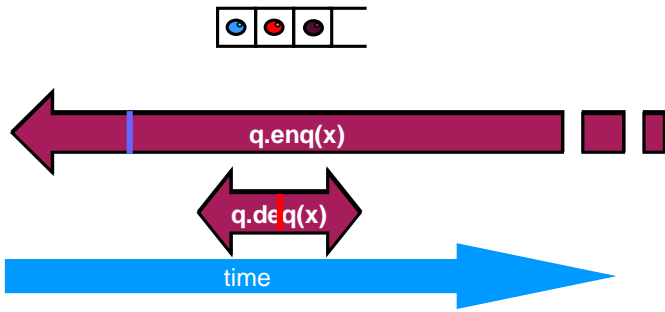
Example 3



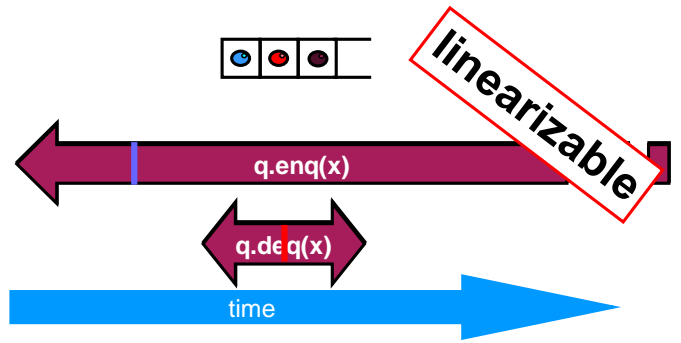
Example 3



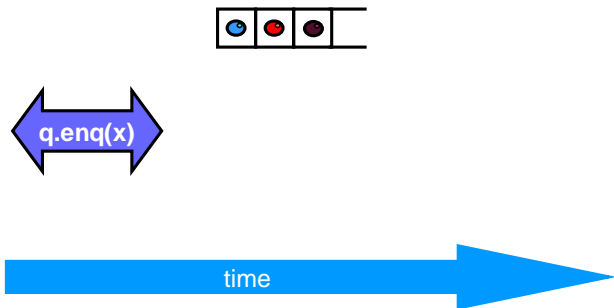
Example 3



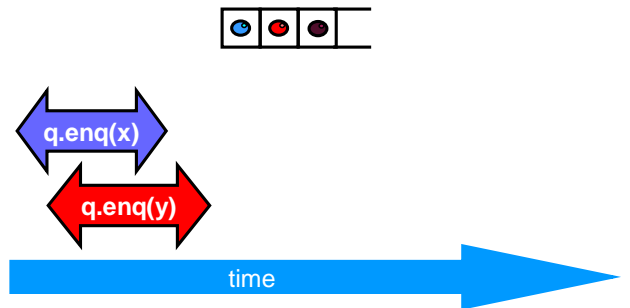
Example 3



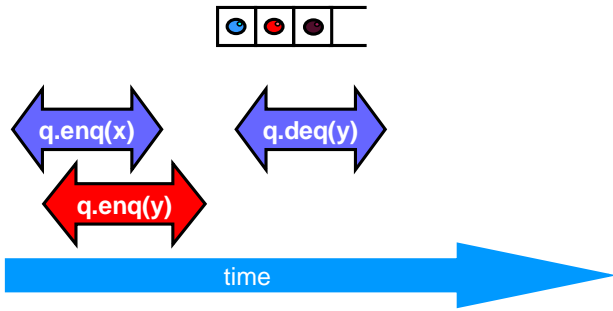
Example 4



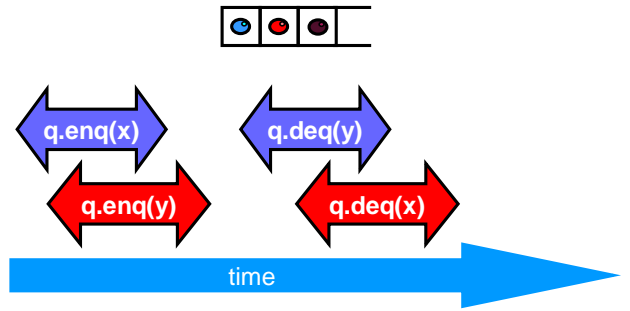
Example 4



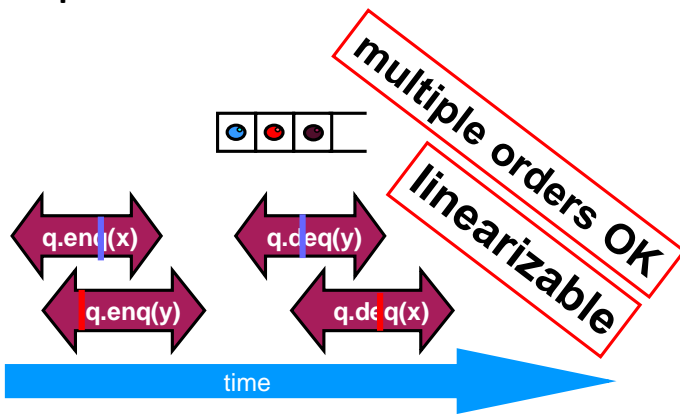
Example 4



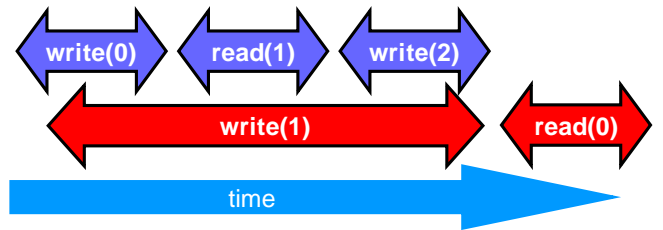
Example 4



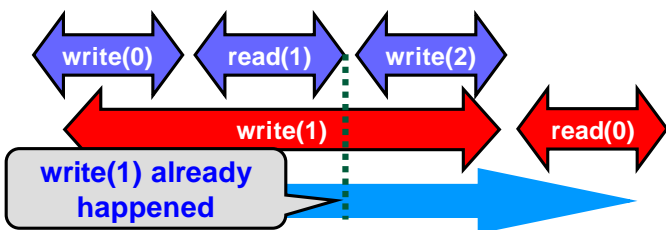
Example 4



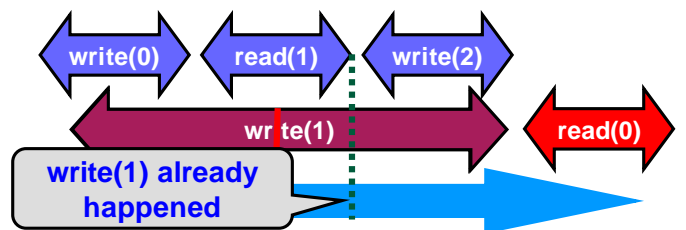
Read/Write Register Example



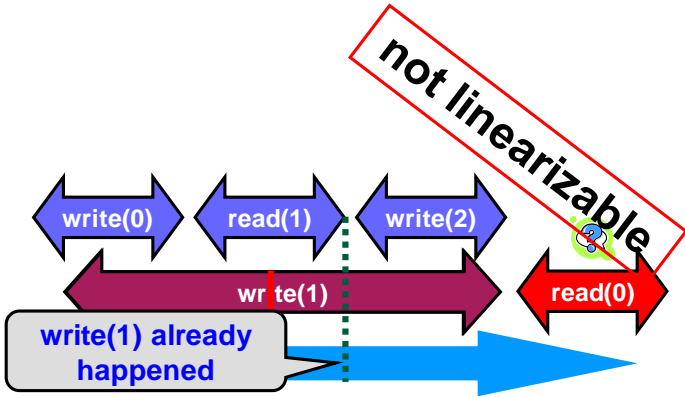
Read/Write Register Example



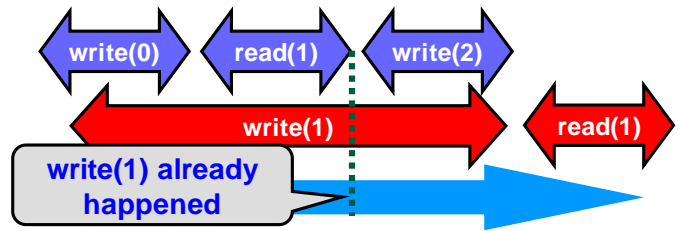
Read/Write Register Example



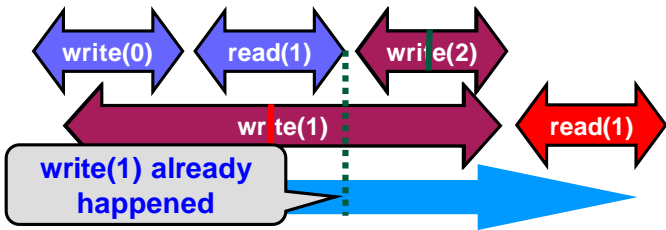
Read/Write Register Example



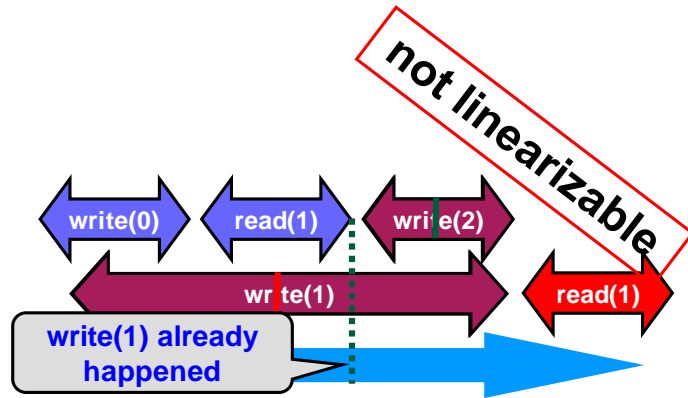
Read/Write Register Example



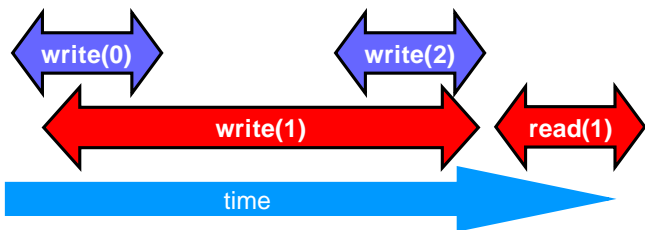
Read/Write Register Example



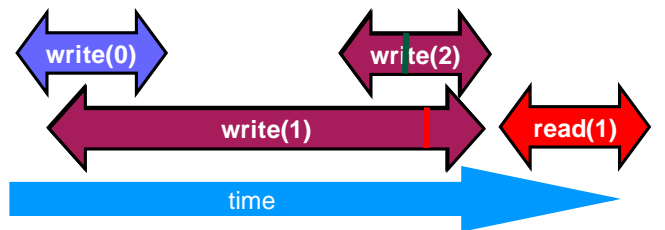
Read/Write Register Example



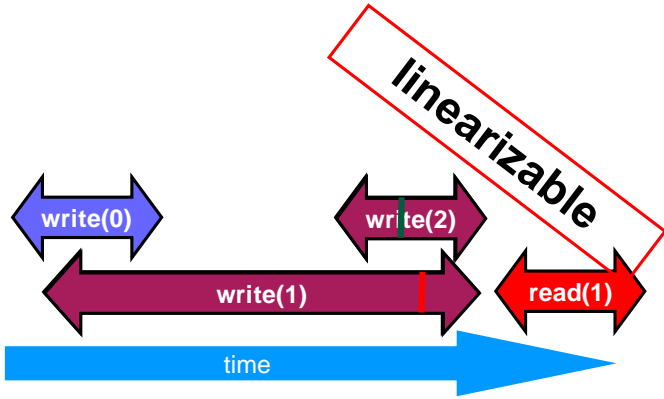
Read/Write Register Example



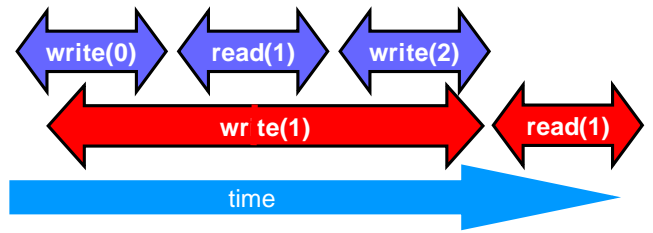
Read/Write Register Example



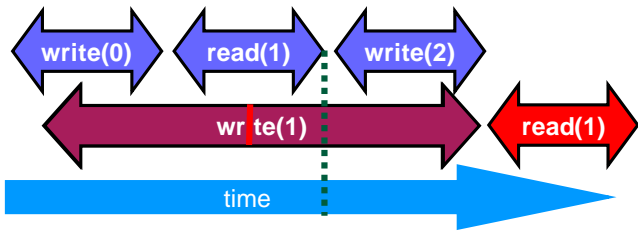
Read/Write Register Example



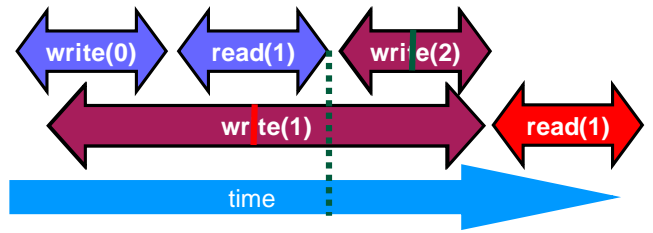
Read/Write Register Example



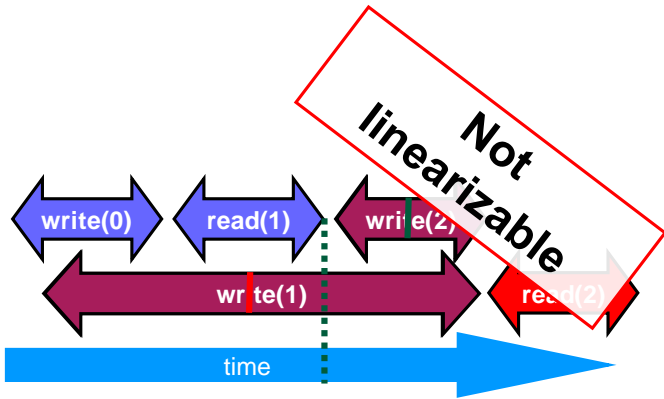
Read/Write Register Example



Read/Write Register Example



Read/Write Register Example

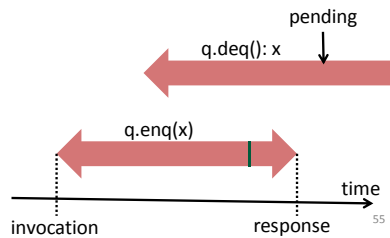


About Executions

- **Why?**
 - Can't we specify the linearization point of each operation without describing an execution?
- **Not always**
 - In some cases, linearization point depends on the execution
- **Define a formal model for executions!**

Properties of concurrent method executions

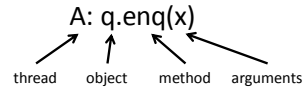
- Method executions take time
 - May overlap
- Method execution = operation
 - Defined by invocation and response events
- Duration of method call
 - Interval between the events



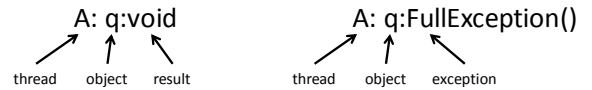
55

Formalization - Notation

Invocation



Response



- Method is implicit (correctness criterion)!

56

Concurrency

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

57

History

Describes an execution

- Sequence of invocations and responses
- $H =$

```
A: q.enq(a)
A: q:void
A: q.enq(b)
B: p.eng(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!

58

Projections on Threads

Threads subhistory $H|P$ ("H at P")

- Subsequences of all events in H whose thread name is P

$H =$

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

$H|A =$

```
A: q.enq(a)
A: q:void
A: q.enq(b)
```

$H|B =$

```
B: p.eng(c)
B: p:void
B: q.deq()
B: q:a
```

59

Projections on Objects

Objects subhistory $H|o$ ("H at o")

- Subsequence of all events in H whose object name is o

$H =$

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

$H|p =$

```
B: p.eng(c)
B: p:void
```

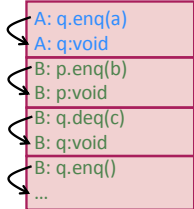
$H|q =$

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

60

Sequential Histories

- A history H 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

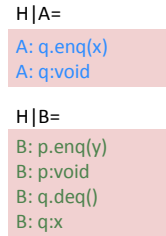
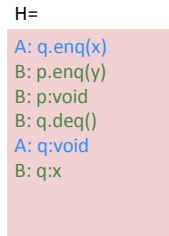
Method calls of different threads **do not interleave**

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

61

Well-formed histories

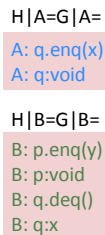
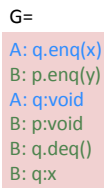
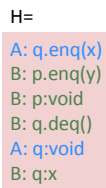
- Per-thread projections must be sequential



62

Equivalent histories

- Per-thread projections must be the same



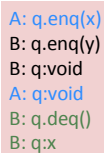
63

Legal Histories

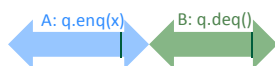
- Sequential specification allows to describe what behavior we expect and tolerate
 - When is a single-thread, single-object history **legal**?
- Recall: Example
 - Preconditions and Postconditions
 - Many others exist!
- A sequential (multi-object) history H is legal if
 - For every object x
 - H|x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

64

Precedence



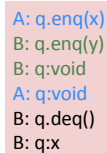
A method execution **precedes** another if response event precedes invocation event



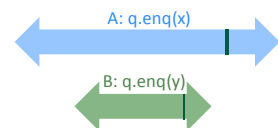
65

Precedence vs. Overlapping

- Non-precedence = overlapping



Some method executions **overlap** with others



66

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 \rightarrow overlap \rightarrow concurrent

67

Definition Linearizability

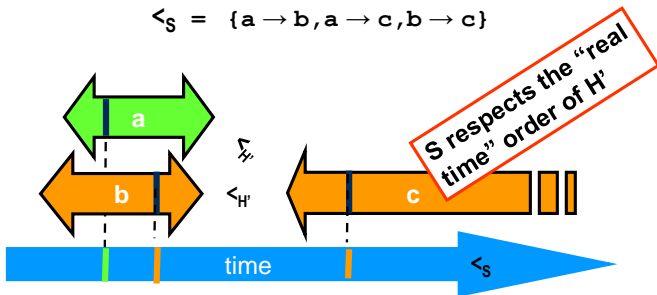
- A history H induces a strict partial order $<_H$ on operations
 - $m_0 <_H m_1$ if $m_0 \rightarrow_H m_1$
- A history H is **linearizable** if
 - H can be extended to a history H'
 - by appending responses to pending operations or dropping pending operations
 - H' is equivalent to some legal sequential 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

68

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

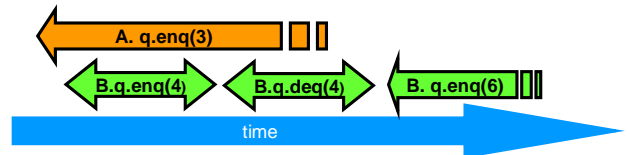
- Find an S that contains H'

$$\begin{aligned} <_{H'} &= \{a \rightarrow c, b \rightarrow c\} \\ <_S &= \{a \rightarrow b, a \rightarrow c, b \rightarrow c\} \end{aligned}$$



Example

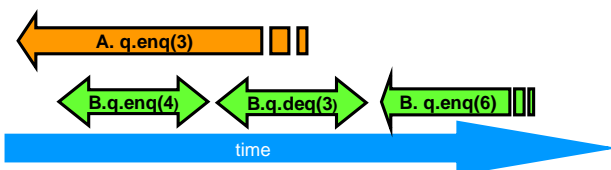
```
A q.enq(3)
B q.enq(4)
B q:void
B q.deq()
B q:4
B q:enq(6)
```



Example

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

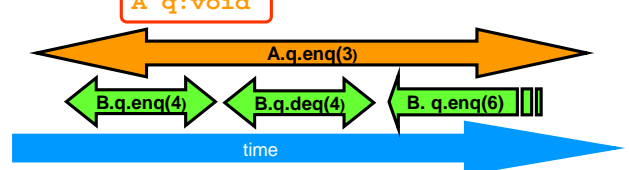
Complete this pending invocation



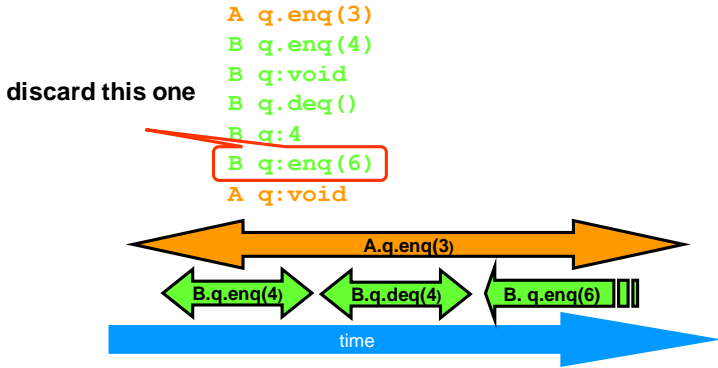
Example

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

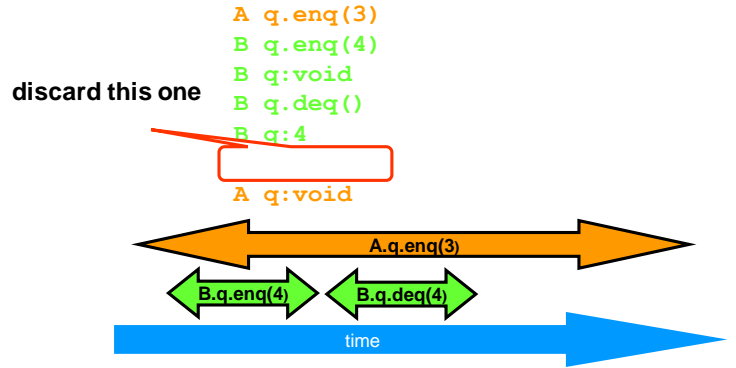
Complete this pending invocation



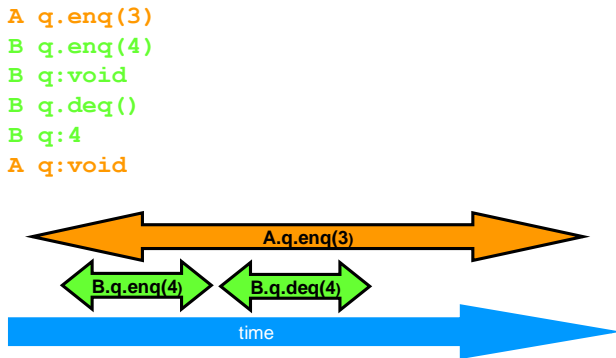
Example



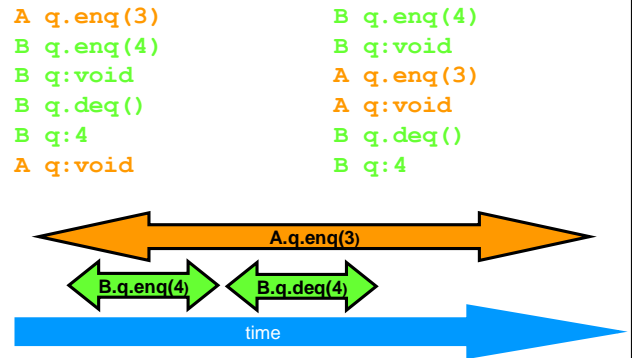
Example



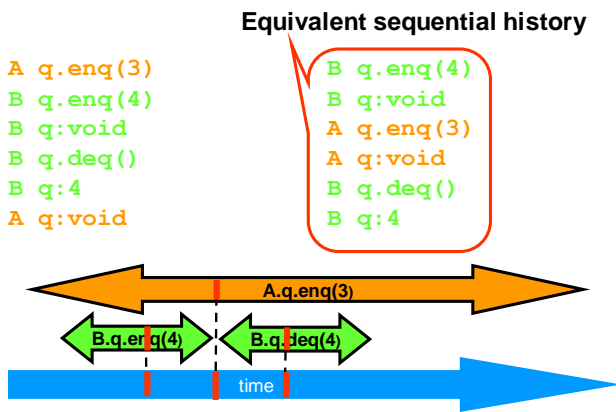
Example



Example



Example



Linearization Points

- Identify one atomic step where a method “happens” (effects become visible to others)
 - Critical section
 - Machine instruction (atoms, 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 on the object must be linearizable

```

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

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

79

Linearizability vs. Sequential Consistency

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

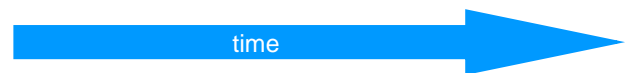
80

Map linearizability to sequential consistency

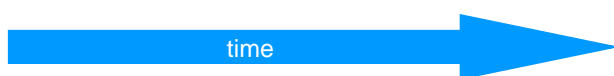
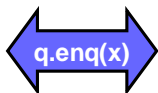
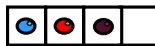
- **Variables with read and write operations**
 - Sequential consistency
- **Objects with a type and methods**
 - Linearizability
- **Map sequential consistency \leftrightarrow linearizability**
 - Reduce data types to variables with read and write operations
 - Model variables as data types with read() and write() methods
- **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 ($\prec_H \subseteq \prec_S$) does not need to be maintained

81

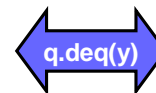
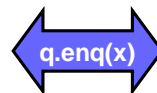
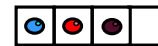
Example



Example

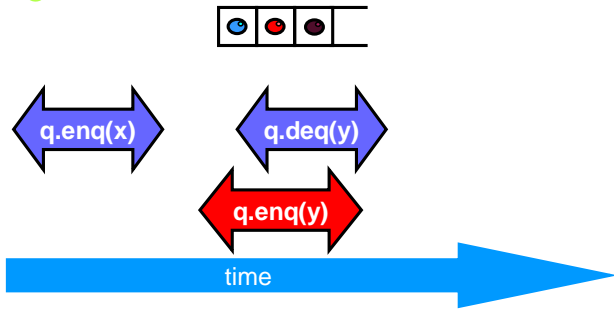


Example



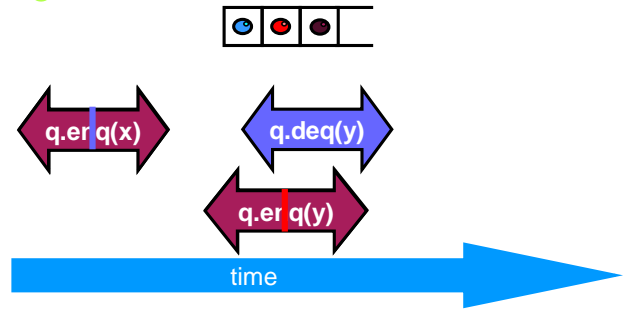
Example

Linearizable?



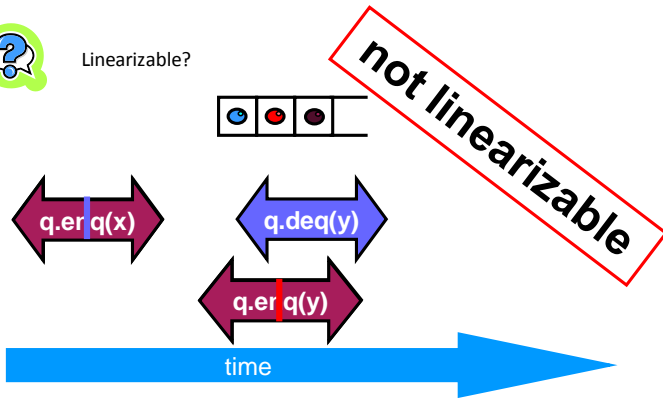
Example

Linearizable?



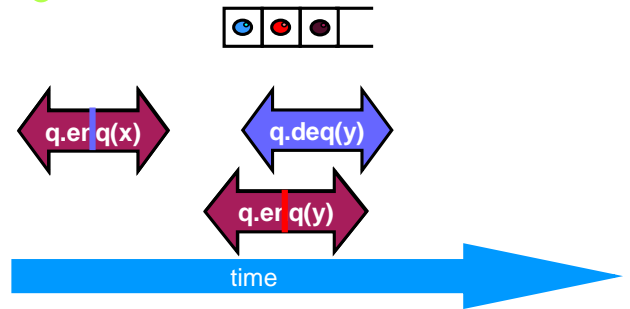
Example

Linearizable?



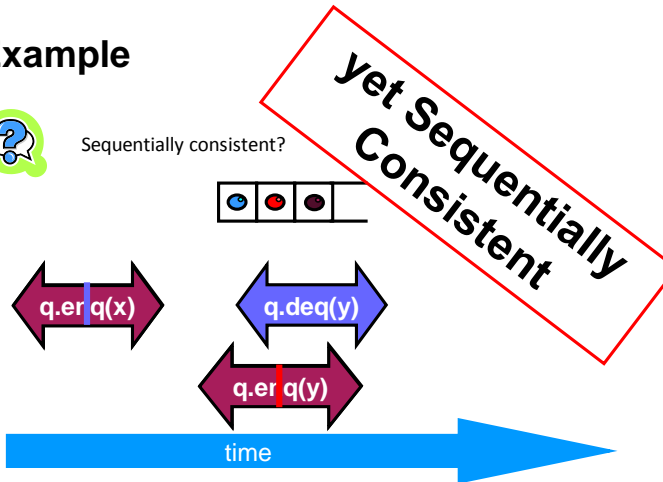
Example

Sequentially consistent?



Example

Sequentially consistent?



Properties of sequential consistency

■ Theorem: Sequential consistency is not compositional

H=

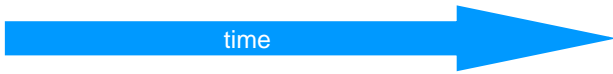
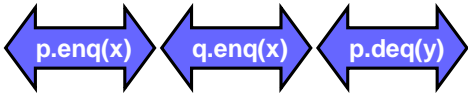
- A: p.enq(x)
- A: p:void
- B: q.enq(y)
- B: q:void
- A: q.enq(x)
- A: q:void
- B: p.enq(y)
- B: p:void
- A: p.deq()
- A: p:y
- B: q.deq()
- B: q:x

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

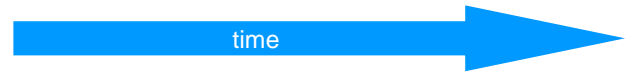
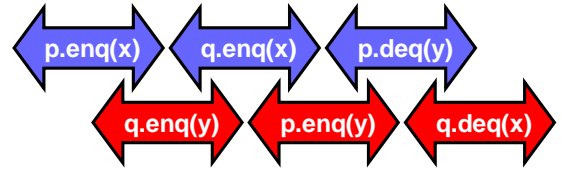
This is not guaranteed for SC schedules!

See following example!

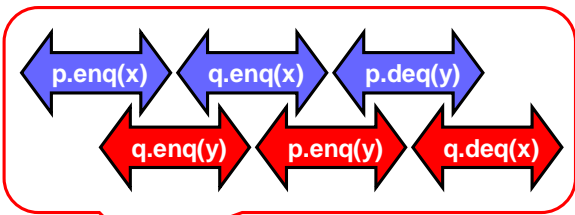
FIFO Queue Example



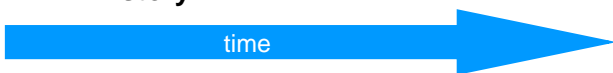
FIFO Queue Example



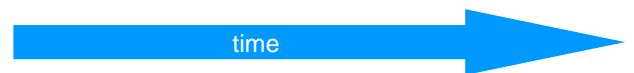
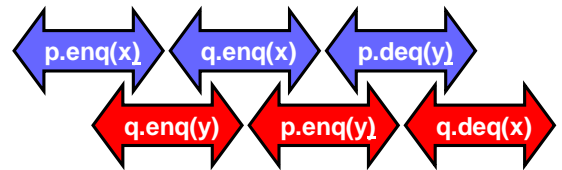
FIFO Queue Example



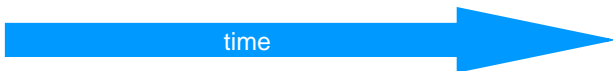
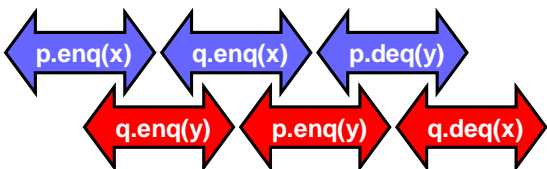
History H



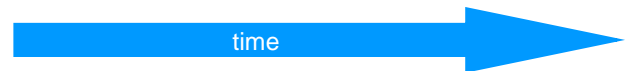
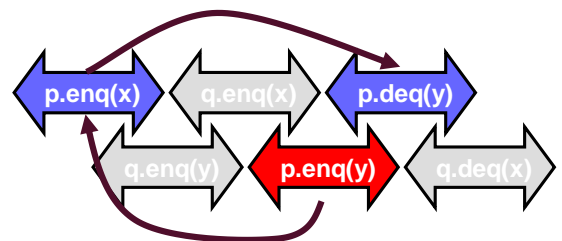
Hip Sequentially Consistent



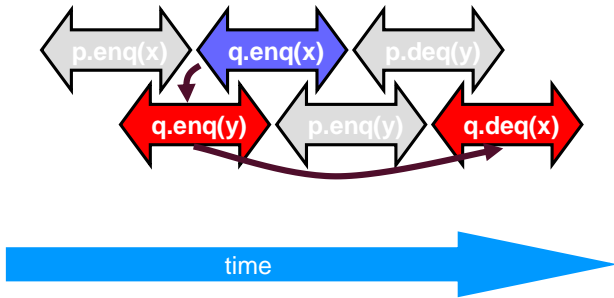
Hiq Sequentially Consistent



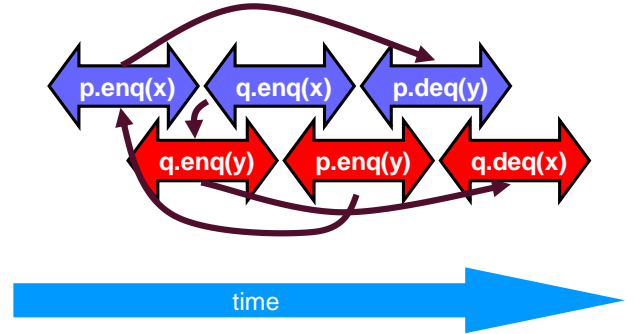
Ordering imposed by p



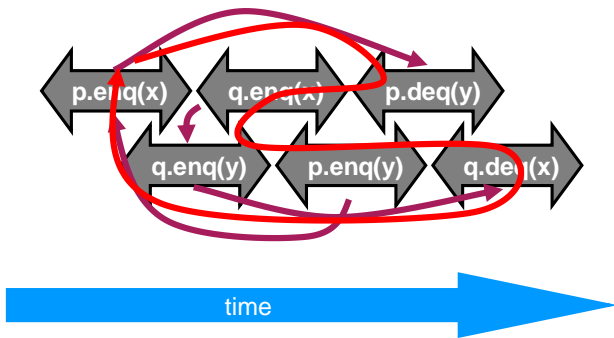
Ordering imposed by q



Ordering imposed by both



Combining orders



Example in our notation

■ Sequential consistency is not compositional

H=	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!

100

Example in our notation

■ Sequential consistency is not compositional

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!

101

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!

102

Correctness: Linearizability

- **Sequential Consistency**
 - Not composable
 - Harder to work with
 - Good way to think about hardware models
- **We will use *linearizability* as in the remainder of this course unless stated otherwise**

Study Goals

- **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

104

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

105

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

106

Everybody wants to optimize

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

107

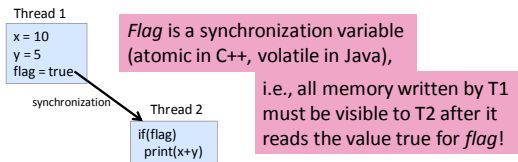
Java and C++ High-level overview

- **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)

108

Communication between Threads Intuition

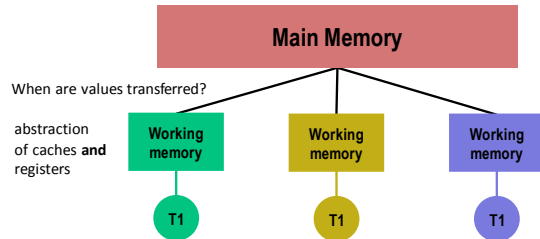
- Not guaranteed unless by:
 - Synchronization
 - Volatile/atomic variables
 - Specialized functions/classes (e.g., java.util.concurrent,)



109

Memory Model Intuition

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



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

110

Lock Synchronization

Java

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

- Synchronized methods as syntactic sugar

C++

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

- Many flexible variants

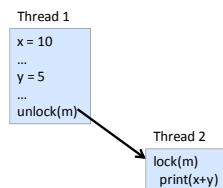
Semantics:

- 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!)

111

Memory semantics

- Similar to synchronization variables



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

112

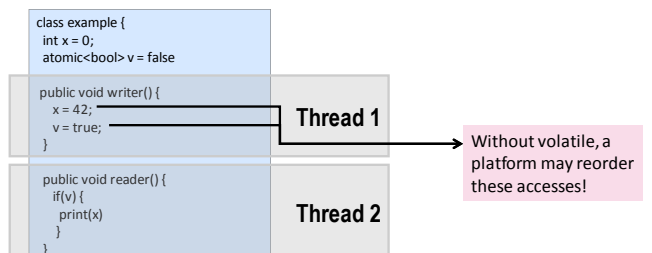
Synchronization Variables

- 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
 - ...

113

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!)



114

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

