Replication and Consistency

03 Concurrent Objects

Annette Bieniusa

AG Softech
FB Informatik
TU Kaiserslautern

Winter Term 2019

Thank you!

These slides are based on companion material of the following books:

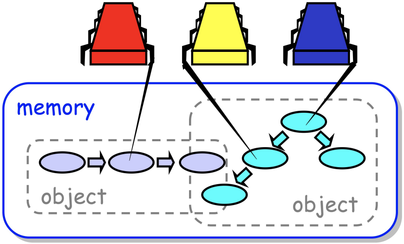
- **The Art of Multiprocessor Programming** by Maurice Herlihy and Nir Shavit
- **Synchronization Algorithms and Concurrent Programming** by Gadi Taubenfeld

TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

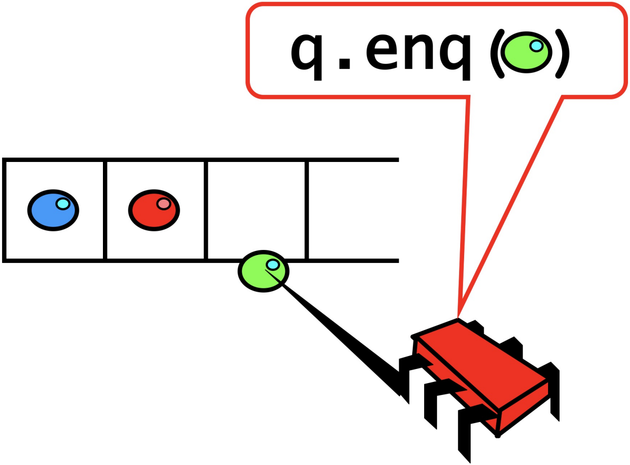
Goals of this lecture

What is a concurrent object?

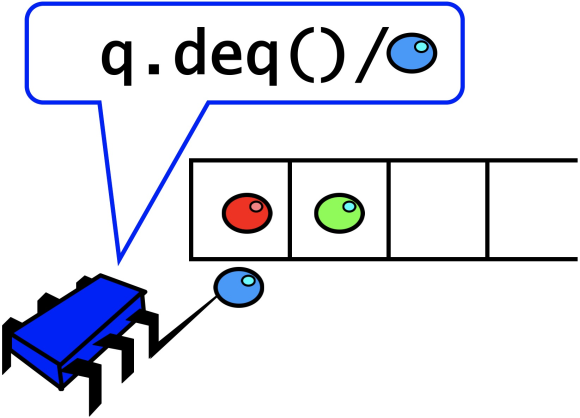
- How do we describe one?
- [How do we implement one?] $\Rightarrow$ Following lectures!
- How do we tell if it is correct?



Example: Concurrent FIFO-Queue



Example: Concurrent FIFO-Queue

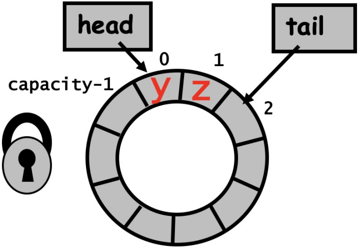


Implementation: Lock-based Queue

```
class LockBasedQueue<T> {
  int head, tail;
  T[] items;
  Lock lock;

  LockBasedQueue(int capacity) {
    head = 0; tail = 0;
    lock = new ReentrantLock();
    items = (T[]) new Object[capacity];
}
```

Sketch



- Initially, queue is empty: `head == tail`
- Queue is full once `head == tail - capacity`

Implementation: Dequeue

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

Implementation: Dequeue

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

Should be correct because modifications are mutually exclusive . . .

Let's get rid of the lock!

- Can we give an implementation without mutual exclusions
- For simplicity: Two-thread solution
  - One thread enqueues only
  - The other dequeues only

Wait-free Two-Thread Queue

```
class WaitFreeQueue<T> {

  int head = 0, tail = 0;
  items = new T[capacity];

  void enq(T x) {
    while (tail-head == capacity); // busy-wait
    items[tail % capacity] = x;
    tail++;
  }

  T deq() {
     while (tail == head);          // busy-wait
     T item = items[head % capacity];
     head++;
     return item;
  }
}
```

Wait-free Two-Thread Queue

```
class WaitFreeQueue<T> {

  int head = 0, tail = 0;
  items = new T[capacity];

  void enq(T x) {
    while (tail-head == capacity); // busy-wait
    items[tail % capacity] = x;
    tail++;
  }

  T deq() {
     while (tail == head);          // busy-wait
     T item = items[head % capacity];
     head++;
     return item;
  }
}
```

Is this correct? Probably for two threads...

How do we define “correctness” when modifications are not mutually exclusive?

Semantics for concurrent queue implementations

- Need a way to **specify** a concurrent queue object
- Need a way to prove that an algorithm **implements** the object's specification
- Let's talk about object specifications!

Correctness and Progress

- In a concurrent setting, we need to specify both the **safety** and the **liveness** properties of an object.
- Need a way to define
  - when an implementation is correct
  - the conditions under which it guarantees progress

Correctness and Progress

- In a concurrent setting, we need to specify both the **safety** and the **liveness** properties of an object.
- Need a way to define
  - when an implementation is correct
  - the conditions under which it guarantees progress

Let's begin with correctness!

Sequential Objects

- Each object has a **state**
  - Usually given by a set of fields
  - Queue example: sequence of items
- Each object has a set of **methods**
  - Only way to manipulate state
  - Queue example: `enq` and `deq` methods

Sequential Specifications

- If (precondition)
  - the object is in such-and-such a state before you call the method,
- Then (postcondition)
  - the method will return a particular value
  - or throw a particular exception.
- and (postcondition, con't)
  - the object will be in some other state when the method returns

Example: Pre- and Post-Conditions for Deque (Part 1)

- Precondition:
  - Queue is non-empty
- Postcondition:
  - Returns first item in queue
- Postcondition:
  - Removes first item in queue

Example: Pre- and Post-Conditions for Deque (Part 2)

- Precondition:
  - Queue is empty
- Postcondition:
  - Throws Empty exception
- Postcondition:
  - Queue state unchanged

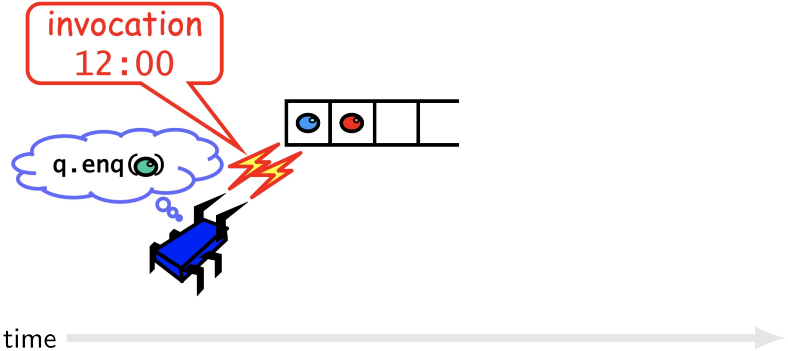
Why Sequential Specifications Totally Rock

- Interactions among methods captured by side-effects on object state
  - State meaningful between method calls
- Documentation size linear in number of methods
  - Each method described in isolation
- Can add new methods
  - Without changing descriptions of old methods

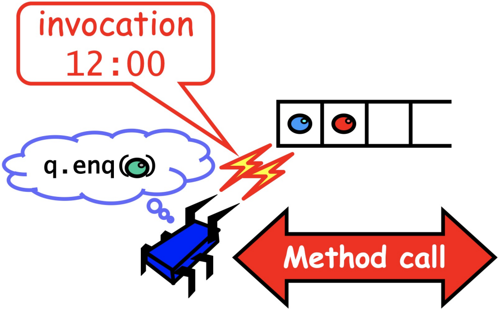
What About Concurrent Specifications?

- Methods?
- Documentation?
- Adding new methods (i.e. compositionality)?

Methods Take Time

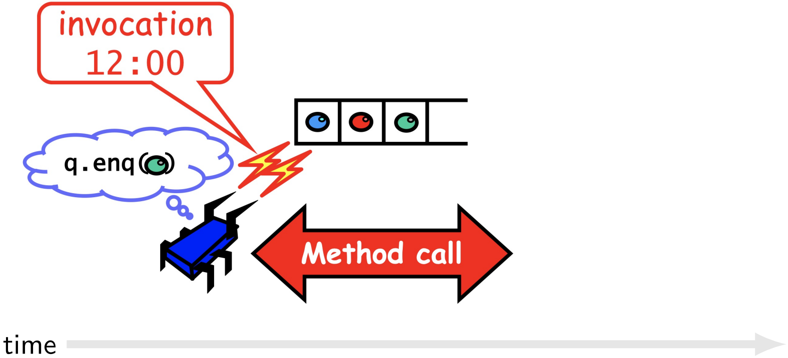


Methods Take Time



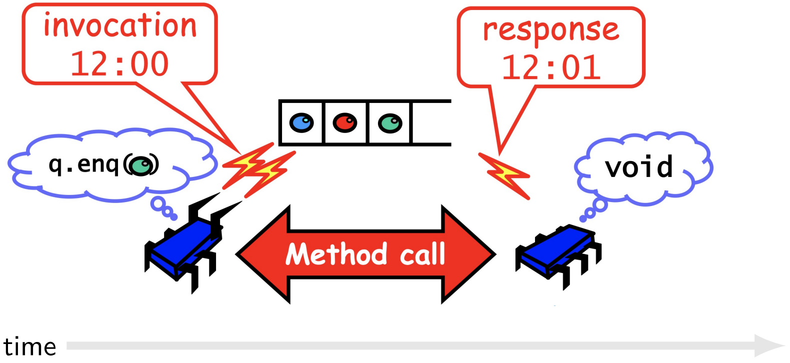
time

Methods Take Time



TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

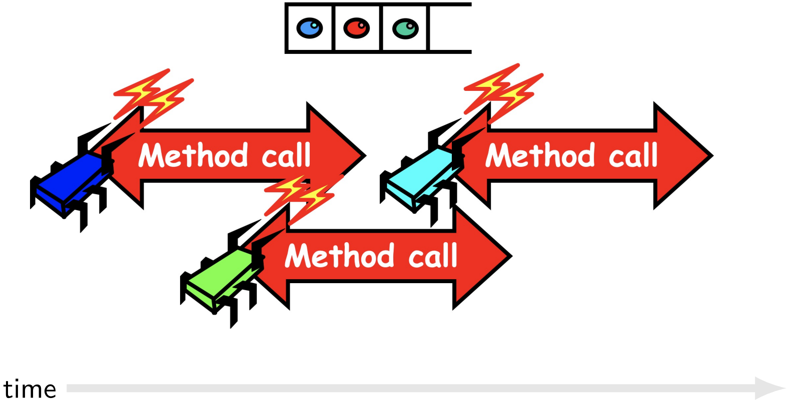
Methods Take Time



Sequential vs Concurrent

- Sequential
  - Methods take time? Who knew?
- Concurrent
  - Method call is **not an event**.
  - Method call is an **interval** that starts with an **invocation event** and ends with a **response event**.
  - A method call is **pending** if its call event has occurred, but not its response event.

Concurrent Methods take overlapping time



Sequential vs Concurrent

- Sequential
  - Object needs meaningful state only between method calls
- Concurrent
  - Because method calls overlap, object might never be "between method calls"

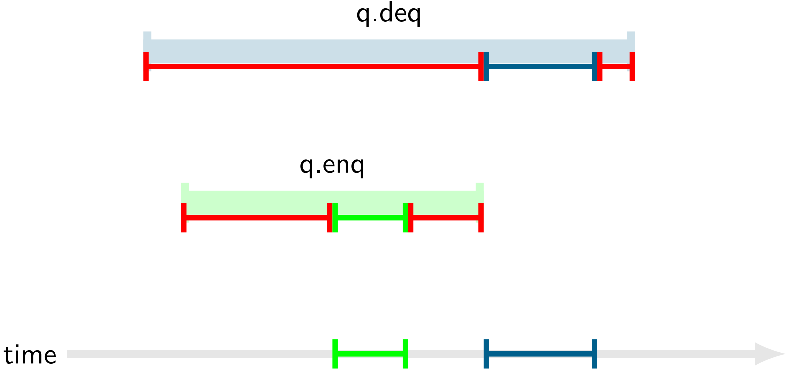
Sequential vs Concurrent

- Sequential
  - Each method described in isolation
  - Can add new methods without affecting older methods
- Concurrent
  - Everything can potentially interact with everything else
  - Must characterize all possible interactions with concurrent calls
  - What if two `enq`s overlap?
  - Two `deq`s? `enq` and `deq`? ...

The BIG Question

- What does it mean for a concurrent object to be correct?
- What is a concurrent FIFO queue?
  - **FIFO** means strict temporal order
  - **Concurrent** means ambiguous temporal order

Intuition: Concurrency with Mutual exclusion



- Behavior is actually “sequential”

Linearizability

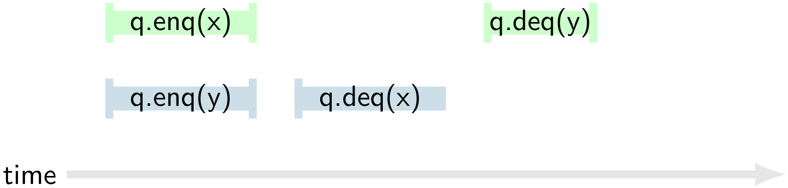
- Each method should "take effect" instantaneously between invocation and response events
- Object is correct if this "sequential" behavior is correct
- Any such concurrent object is **Linearizable™**

Is it really about the object?

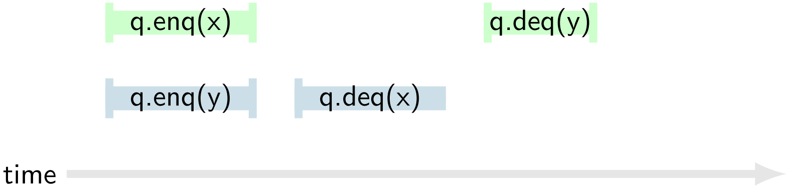
- Recall: Each method should "take effect" instantaneously between invocation and response events
- Sounds like a property of **an execution** ...
- A linearizable object is one all of whose possible executions are linearizable

Examples

Example 1

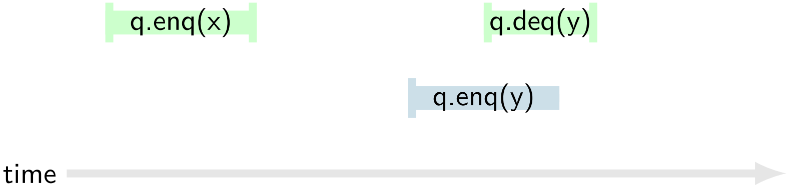


Example 1

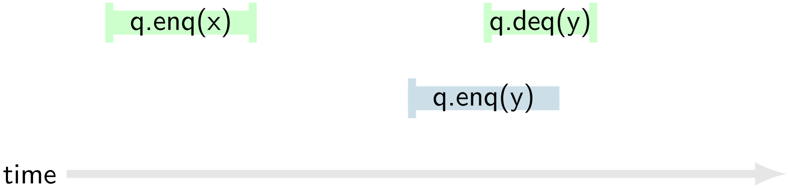


⇒ Linearizable (if linearization point of `q.enq(x)` is before linearization point of `q.enq(y)`)

Example 2

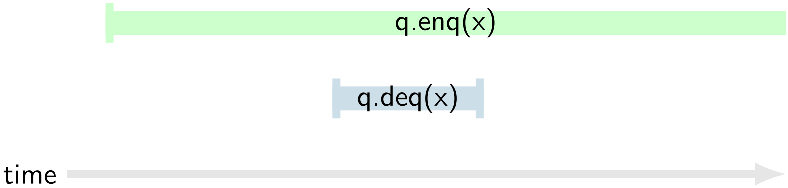


Example 2

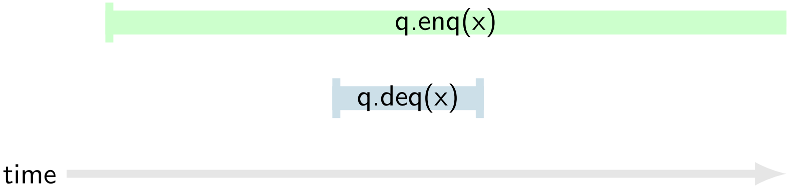


⇒ Not linearizable (`q.enq(y)` cannot be linearized before `q.enq(x)`)

Example 3

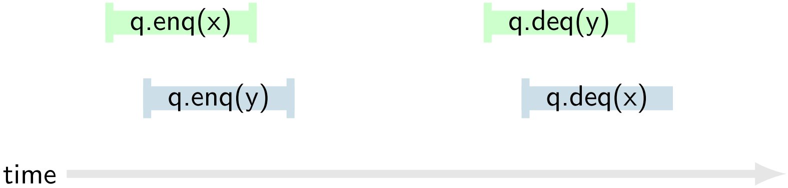


Example 3

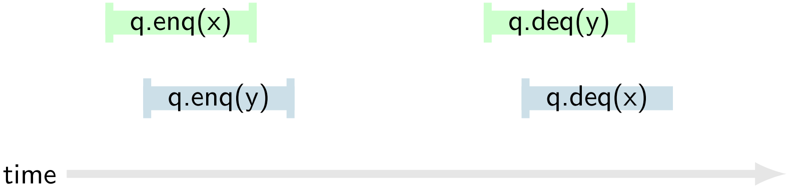


⇒ Linearizable

Example 4

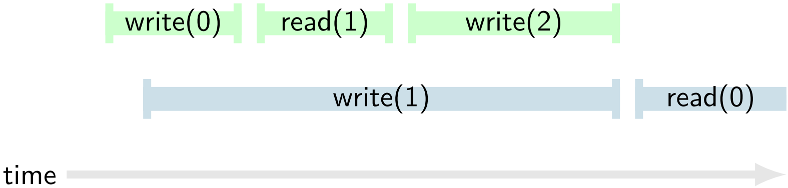


Example 4

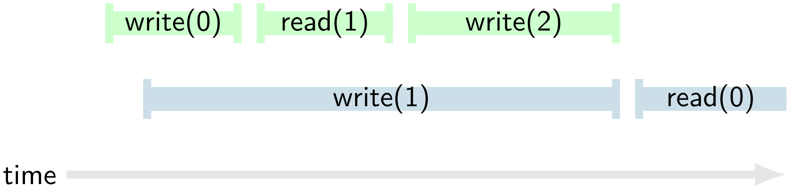


⇒ Linearizable (multiple orders possible)

Read/Write Register Example 1

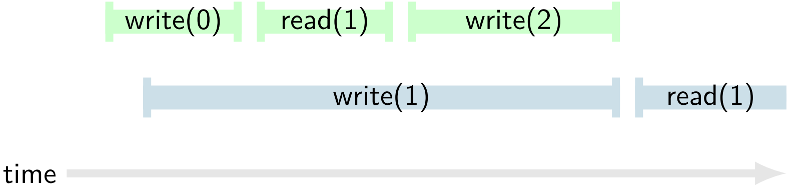


Read/Write Register Example 1

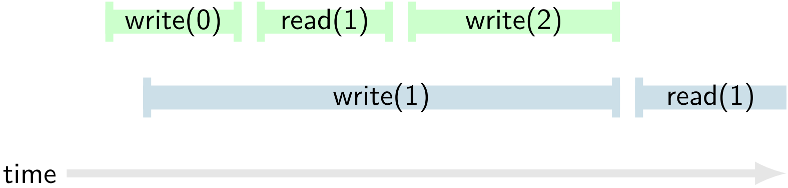


⇒ Not linearizable: `write(1)` happened before `read(0)`

Read/Write Register Example 2



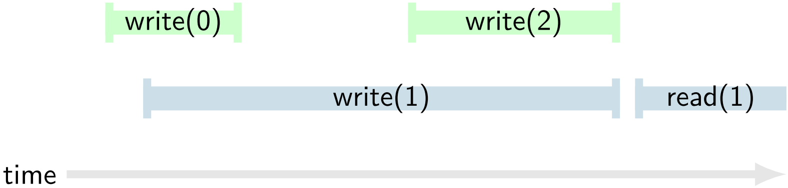
Read/Write Register Example 2



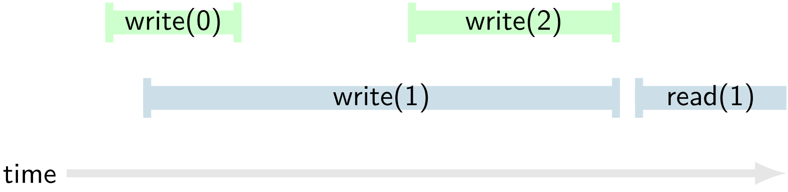
⇒ Not linearizable

- `write(1) happened before`write(2)‘,
- and `write(2)` happened before `read(1)`

Read/Write Register Example 3



Read/Write Register Example 3



⇒ Linearizable

Formal Definitions

Talking 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

Executions

Split method calls into two events:

- Invocation
  - method name & args
  - `q.enq(x)`
- Response
  - result or exception
  - `q.enq(x)` returns **`void`**
  - `q.deq()` returns `x`
  - `q.deq()` throws `empty`

Notation

Invocation

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

Response

```
A    q:  void
A    q:  empty()
```

Method is implicit

History

```
A     q.enq(3)
A     q: void
A     q.enq(5)
B     p.enq(4)
B     p: void
B     q.deq()
B     q: 3
```

Definitions

- A **history** H is a finite sequence of method invocation and response events.
- A **subhistory** of a history H is a subsequents of the events in H.
- A response **matches** an invocation if they have the same object and thread.
- A **method call** in a history H is a pair of an invocation and the next matching response in H.

Object Projections

$$H|q =$$

```
A    q.enq(3)
A    q:  void
A    q.enq(5)

B    q.deq()
B    q:  3
```

$$H|p =$$

```
B    p.enq(4)
B    p:  void
```

Thread Projections

$H|A =$

```
A    q.enq(3)
A    q:  void
A    q.enq(5)
```

$H|B =$

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

Complete Subhistory

$$complete(H) =$$

```
A    q.enq(3)
A    q:  void
//A  q.enq(5) -> Discard pending invocations
B    p.enq(4)
B    p:  void
B    q.deq()
B    q:  3
```

- The **complete subhistory** is the subsequence of H consisting of all matching invocations and responses.
- An invocation is **pending** if it has no matching response.

Sequential Histories

- Method calls of different threads do not interleave
- Final pending invocation ok

```
A q.enq(3)
A q:void    // matched
B p.enq(4)
B p:void    // matched
B q.deq()
B q:3       // matched
A q:enq(5)
```

A history H is **sequential** if the first event of H is an invocation and each invocation, except possibly the last, is immediately followed by a matching response.

Well-formed Histories

$H =$

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

A history H is well-formed if each thread subhistory is sequential.

Equivalent Histories

$H =$

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

$G =$

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

$$
\begin{aligned}
H|A &= G|A \\
H|B &= G|B
\end{aligned}
$$

Two histories H and G are equivalent if for every thread T, $H|T = G|T$.

Sequential Specifications

- A sequential specification is some way of telling whether a single-thread, single-object history is legal.
- For example:
  - Pre- and post-conditions
  - But plenty of other techniques exist
- *Here:* A sequential specification for an object is a set of sequential histories for that object.

A sequential (multi-object) history H is **legal** if for every object x, $H|x$ is in the sequential spec for x.

Recall: Precedence

- Given history H and method executions m_0 and m_1 in H, we say $m_0 \rightarrow_H m_1$, if m_0 precedes m_1.
- Relation $m_0 \rightarrow_H m_1$ is a
  - Partial order
  - Total order if H is sequential

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

A method call **precedes** another if response event precedes invocation event.
Example:

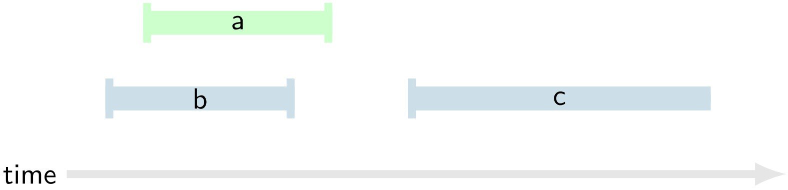
Linearizability(Herlihy and Wing 1990)

A history H is **linearizable** if it can be extended to some history G by

- appending zero or more responses to pending invocations
- and there is a legal sequential history S such that $complete(G)$ is equivalent to S and $\rightarrow_G \subset \rightarrow_S$.

What is $\rightarrow_G \subset \rightarrow_S$?

$$
\begin{aligned}
\rightarrow_G &= \{a \rightarrow c, b \rightarrow c\} \\
\rightarrow_S &= \{a \rightarrow c, b \rightarrow c, a \rightarrow b\}
\end{aligned}
$$



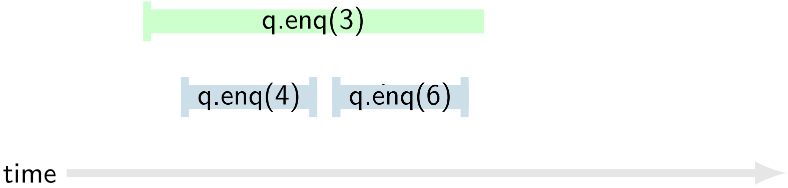
If method call m_0 precedes m_1 in G, then the same is true in S.

Remarks

- Some pending invocations took effect, so keep them
- Discard the rest
- Condition $\rightarrow_G \subset \rightarrow_S$ means that S respects "real-time order" of G
  $\Rightarrow$ Restriction on S
- When picking linearization points, they need to be within the intervals
- Only for unordered intervals, the order can be arbitrary

Example

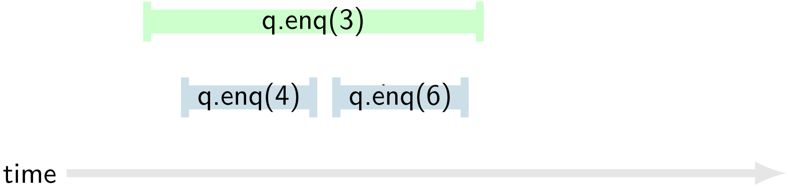
```
A q.enq(3)
B q.enq(4)
B q:void
B q.deq()
B q:4
B q:enq(6)
```



Example: Step 1

Complete pending invocation for A

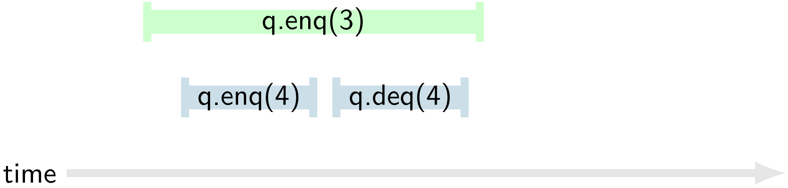
```
A q.enq(3)
B q.enq(4)
B q:void
B q.deq()
B q:4
B q:enq(6)
A q:void
```



Example: Step 2

Discard pending invocation for B

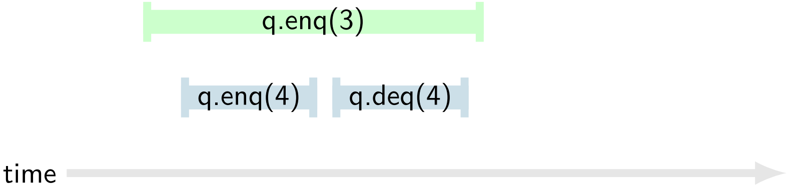
```
A q.enq(3)
B q.enq(4)
B q:void
B q.deq()
B q:4
A q:void
```



Example: Step 3

Construct equivalent sequential history and check if linearizable

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



Concurrency and Linearizability

- How much concurrency does linearizability allow?
- When must a method invocation block?

Concurrency and Linearizability

- How much concurrency does linearizability allow?
- When must a method invocation block?

- Focus on total methods
  - A method call is **total** if it is defined for every object state; otherwise, it is **partial**.
- Example:
  - `deq()` that throws `empty` exception
  - vs `deq()` that waits ...
- Why?
  - Otherwise, blocking might be unrelated to synchronization

Question

When does linearizability require a method invocation to block?

Question

When does linearizability require a method invocation to block?

Answer: Never – Linearizability is non-blocking!

Theorem: Non-blocking

Strong result:

- A pending invocation of a total method is never required to wait for another pending invocation to complete!

If method invocation `A q.inv(...)` is pending in history H, then there exists a response `A q:res` such that H extended by `A q:res` is linearizable.

Proof Sketch

- Pick any linearization S of H
- If S already contains invocation `A q.inv(...)` and response for every method call, then we are done.
- Otherwise, pick a response such that S gets extended by `A q.inv(...)` and further append `A q:res`
- Possible because object is total
- This extension S' is a linearisation of $H\cdot$ `A q.inv(...)` and hence also a linearization of H.

Theorem: Composability

History H is linearizable if and only if for every object x, $H|x$ is linearizable.

Why does it matter?

- Modularity
- Can prove linearizability of objects in isolation
- Can compose independently-implemented objects

TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

Proof Sketch

Direction $\Rightarrow$

- For each x pick any linearization of $H|x$.
- Let R_x be the appended missing responses to $H|x$ and let $\rightarrow_x$ be the linearization order.
- Let H' be H with R_x appended.

Induction on method calls in H'

- Base case: H' contains one method call $\Rightarrow$ Trivial, right? ;)
- Induction step:
  - For each object, let m be the last method call in $H'|x$ with respect to $\rightarrow_x$.
  - Let G' be H' with method m removed.
  - Because m was the last call, H' is equivalent to $G' \cdot m$.
  - By induction hypothesis, G' is linearizable to sequential history S', and H' and H are linearizable to $S' \cdot m$.

Direction $\Leftarrow$: Exercises

Reasoning about Linearizability: Locking

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

- Linearization points are when locks are released

Reasoning about Linearizability: Lock-free

```
class LockFreeQueue[T] {

  int head = 0, tail = 0;
  items = new T[capacity];

  void enq(T x) {
    while (tail-head == capacity) throw new FullExeption();
    items[tail % capacity] = x;
    tail++;
  }
  T deq() {
     if (tail == head) throw new EmptyExeption();
     T item = items[head % capacity];
     head++;
     return item;
}}
```

- Linearization order is order in which head and tail fields modified
- Remember that there is only one enqueuer and only one dequeuer

Strategy

- Identify one atomic step where method “happens”
  - Critical section
  - Machine instruction
- Doesn’t always work
  - Might need to define several different steps for a given method
- More on this in the exercises

Alternative: Sequential Consistency(Lamport 1979)

History H is **sequentially consistent** if it can be extended to G by

- appending zero or more responses to pending invocations
- discarding other pending invocations

so that G is equivalent to a legal sequential history S

Alternative: Sequential Consistency(Lamport 1979)

History H is **sequentially consistent** if it can be extended to G by

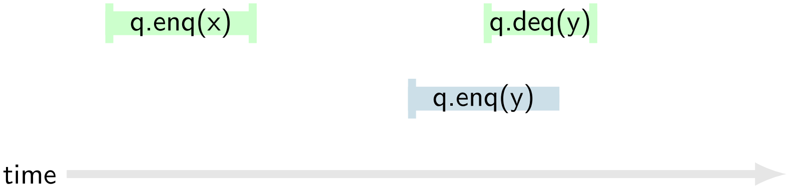
- appending zero or more responses to pending invocations
- discarding other pending invocations

so that G is equivalent to a legal sequential history S

How does this differ from linerizability?

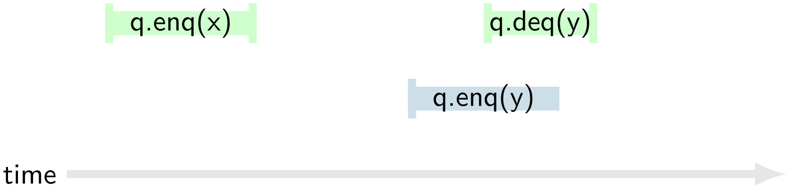
- Removed: "where $\rightarrow_G \subset \rightarrow_S$" !
- G must preserve program order in each thread, but does not need to preserve real-time order
- Can re-order non-overlapping operations done by different threads

Example



TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

Example

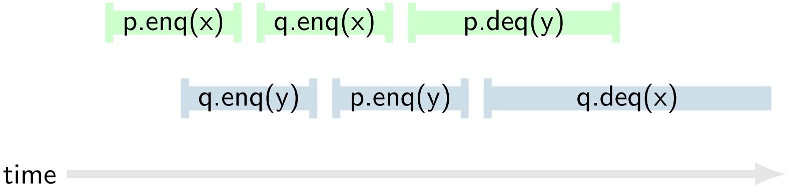


⇒ Not linearizable, but sequentially consistent

Theorem

Sequential consistency is not a local property (and thus not composable).

Example



- $H|q$ and $H|p$ are sequentially consistent
- Combining orders imposed by operations on p and q and program order yields cycle:

`p.enq(x)` $\rightarrow$ `q.enq(x)` $\rightarrow$ `q.enq(y)` $\rightarrow$ `p.enq(y)` $\rightarrow$ `p.enq(x)`

Summary

Sequential Consistency

- Not composable
- Harder to work with
- Good way to think about hardware models

Linearizability

- Operation takes effect instantaneously between invocation and response
- Uses sequential specification, locality implies composablity
- Good for high level objects We will use linearizability as in the remainder of this course unless stated otherwise.

TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

Summary

- Critical sections are an easy way to implement linearizability
  - Take sequential object
  - Make each method a critical section
- But:
  - Blocking
  - No concurrency
- We will look at linearizable blocking and non-blocking implementations of objects.

Copyright

Further reading

Herlihy, Maurice, and Jeannette M. Wing. 1990. "Linearizability: A Correctness Condition for Concurrent Objects." *ACM Trans. Program. Lang. Syst.* 12 (3): 463–92. https://doi.org/10.1145/78969.78972.

Lamport, Leslie. 1979. "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs." *IEEE Trans. Computers* 28 (9): 690–91. https://doi.org/10.1109/TC.1979.1675439.