



# Concepts of Concurrent Computation

Bertrand Meyer  
Sebastian Nanz

Lecture 10: Lock-free approaches



# Today's lecture

---

In this lecture you will learn about:

- Problems of the locking-based approach to shared-memory concurrent programming
- Lock-free programming, a synchronization technique based on atomic read-modify-write primitives
- Linearizability and sequential consistency, two correctness conditions for concurrent objects
- Software transactional memory (STM), a synchronization mechanism based on the idea of database transactions



# Motivation

# What's wrong with locks? (1)

---



- It's difficult to program with locks, because it's easy to ...
  - ... **forget a lock**: danger of data races.
  - ... take **too many locks**: danger of deadlock.
  - ... **take locks in the wrong order**: danger of deadlock.
  - ... **take the wrong lock**: the relation between the lock and the data it protects is not explicit in the program.
- Locks cause blocking:
  - Danger of **priority inversion**: if a lower-priority thread is preempted while holding a lock, higher-priority threads cannot proceed.
  - Danger of **convoying**: other threads queue up waiting while a thread holding a lock is blocked.

## What's wrong with locks? (2)

---



- Two concepts related to locks:
  - *lock overhead* (increases with more locks): time for acquiring and releasing locks, and other resources
  - *lock contention* (decreases with more locks): the situation that multiple processes wait for the same lock
- For performance, the developer has to **carefully choose the granularity of locking**: both lock overhead and contention need to be small.
- Locks are also problematic for designing fault-tolerant systems: If a faulty process halts while holding a lock, no other process can obtain the lock.

## What's wrong with locks? (3)



- Locks are **not composable** in general, i.e. they don't support modular programming (building larger programs from smaller blocks).

```
class Account {  
    int balance;  
    synchronized void deposit(int amount) {  
        balance = balance + amount;  
    }  
    synchronized void withdraw(int amount) {  
        balance = balance - amount;  
    }  
}
```

- How to implement the following method?  
`void transfer(Account acc1, Account acc2, int amount)`

## What's wrong with locks? (4)



- Although deposit and withdraw are correctly implemented by themselves, the following is incorrect:

```
void transfer(Account acc1, Account acc2, int amount) {  
    acc1.withdraw(amount);  
    acc2.deposit(amount);  
}
```

- Instead we would have to add explicit locking code:

```
void transfer(Account acc1, Account acc2, int amount) {  
    synchronized (acc1) {  
        synchronized (acc2) {  
            acc1.withdraw(amount);  
            acc2.deposit(amount);  
        }  
    }  
}
```

# Concurrent programming without locking

---



- Use a pure message-passing approach:
  - Since no data is shared, there is no need for locks
  - Of course message-passing approaches have their own drawbacks, for example
    - potentially larger **overhead of messaging**
    - the **need to copy data** which has to be shared
    - potentially **slower access to data**, e.g. to read-only data structures which need to be shared
- If a shared-memory approach is preferred, the only alternative to using locks is to make the implementation of a concurrent program ***lock-free***.



# Lock-free approaches

---



- Lock-free programming using **atomic read-modify-write primitives**, such as compare and swap (CAS)
- *Software transactional memory (STM)*, a programming model based on the idea of database transactions



# Lock-free programming

# Lock-free programming

---



- *Lock-free programming* is the idea to write shared-memory concurrent programs that don't use locks but can still ensure thread-safety
- Instead of locks, use stronger atomic operations such as **compare-and-swap** (atomic read-modify-write primitives)
  - These primitives typically have to be provided by hardware
- Coming up with general lock-free algorithms is hard
- Hence usually one focuses on developing **lock-free data structures**: stack, list, queue, buffer, ...

# Classes of lock-free algorithms

---



- For lock-free algorithms one typically distinguishes between the following two classes:
  - *lock-free*: some process completes in a finite number of steps (free from deadlock)
  - *wait-free*: all processes complete in a finite number of steps (free from starvation)
- Wait-free implies lock-free

## Compare-and-swap (recap)

---



- *Compare-and-swap (CAS)* takes three parameters: the address of a memory location, an old and a new value
- The new value is atomically written to the memory location if the content of the location agrees with the old value

```
CAS (x, old, new)
  do
    if *x = old then
      *x := new;
      result := true
    else
      result := false
    end
  end
end
```

# Simple lock-free stack (1)

---



- Using *CAS*, we obtain the following lock-free implementation of a stack, due to (Treiber, 1986)
- A stack of elements (here of integer type) is represented as a linked list of nodes
- The top of the stack is denoted by the node *head*

```
class Node {  
    Node* next;  
    int item;  
}  
  
Node* head; // top of the stack
```

## Simple lock-free stack (2)



- In the implementation of push and pop, a common pattern in lock-free algorithms is used:
  1. read a value from the current state
  2. compute an updated value based on the read value
  3. atomically update the state by swapping the new for the old value

```
void push (int value) {  
    Node* oldHead;  
    Node* newHead := new Node();  
    node.item := value;  
    do {  
        oldHead := head;  
        newHead.next := head;  
    } while (!CAS(&head, oldHead, newHead));  
}
```

## Simple lock-free stack (3)



- If the state changes between steps 1 and 3, the CAS-operation fails and the algorithm is repeated until success

```
int pop () {
    Node* oldHead;
    Node* newHead;
    do {
        oldHead := head;
        if(oldHead = null) return EMPTY;
        newHead := oldHead.next;
    } while(!CAS(&head, oldHead, newHead));
    return oldHead.item;
}
```



# The ABA problem (1)

---



- In the stack example, the following has to be avoided:
  - $T_1$ : starts `pop()` - reads value of current head as  $X$
  - $T_2$ : executes `pop()`, removing  $X$  from the stack
  - $T_2$ : modifies the stack arbitrarily
  - $T_2$ : executes `push( $X$ )`, putting  $X$  back on the stack
  - $T_1$ : finishes `pop()` - CAS succeeds, since  $X$  is on top



## The ABA problem (2)

---

- This problematic pattern is called the *ABA problem*:
  - a value is read from state A
  - the state is changed to state B
  - the CAS operation does not distinguish between A and B, so it assumes it is still A
- The problem is avoided in the simple stack example as push always puts a *new* node, and the old node's memory location is not freed up yet (if the memory address would be reused)

# More atomic operations

---



- Besides compare-and-swap there are many atomic operations suggested for lock-free programming
- Not all of the primitives suggested have been implemented in hardware
- The question is which primitives are worth implementing in hardware, i.e. whether these primitives have different strength
- This question has been answered: primitives can be ranked (Herlihy, 1993)

# Hierarchy of atomic primitives



- Atomic primitives can then be ranked according to their consensus number (Herlihy, 1993):

Consensus number	Primitive
1	read/write registers
2	test-and-set, fetch-and-add, queue, stack, ...
⋮	⋮
$2n - 2$	n-register assignment
⋮	⋮
$\infty$	memory-to-memory move and swap, compare-and-swap, load-link/store-conditional

# Consensus numbers

---



- Each primitive is given a **consensus number**, the maximum number of processes for which the primitive can implement a consensus protocol (a group of processes agrees on a common value)
- In a system of  $n$  or more processes, it is impossible to provide a wait-free implementation of a primitive with consensus number  $n$  from a primitive with lower consensus number

## Another primitive: load-link/store-conditional

---



- **load-link/store-conditional** is a pair of instructions which together implement an atomic read-modify-write operation
- **load-link** returns the current value of a memory location
- A subsequent **store-conditional** to the same location will store a new value only if no updates have occurred to that location since the load-link.
- The store-conditional is guaranteed to fail if updates have occurred, even if the value read by the load-link has since been restored.

# Lock-free programming: Discussion

---



- Lock-free programming can provide **good performance in some situations**, avoiding some of the problems mentioned for locks (e.g. priority inversion)
- It's **difficult to correctly implement lock-free algorithms** (see ABA-problem)
- Most work confined to data structures: for these well-established algorithms and implementations are available
- One main restriction is that most read-modify-write primitives operate only on a single word: this leads to **unnatural structuring of algorithms**



# Linearizability



# Sequential objects

---



- We can understand a the execution of a system as operations of a collection of (sequential) processes on data objects
- Each object has a type, describing its possible values and the operations for modifying them
- What does it mean for such objects to be correct?
- In a **sequential system**, where there is only one process, it is easy to specify the behavior of each operation:
  - Pre- and postconditions
  - Intermediate states never visible upon invocation of an operation

# Concurrent objects

---

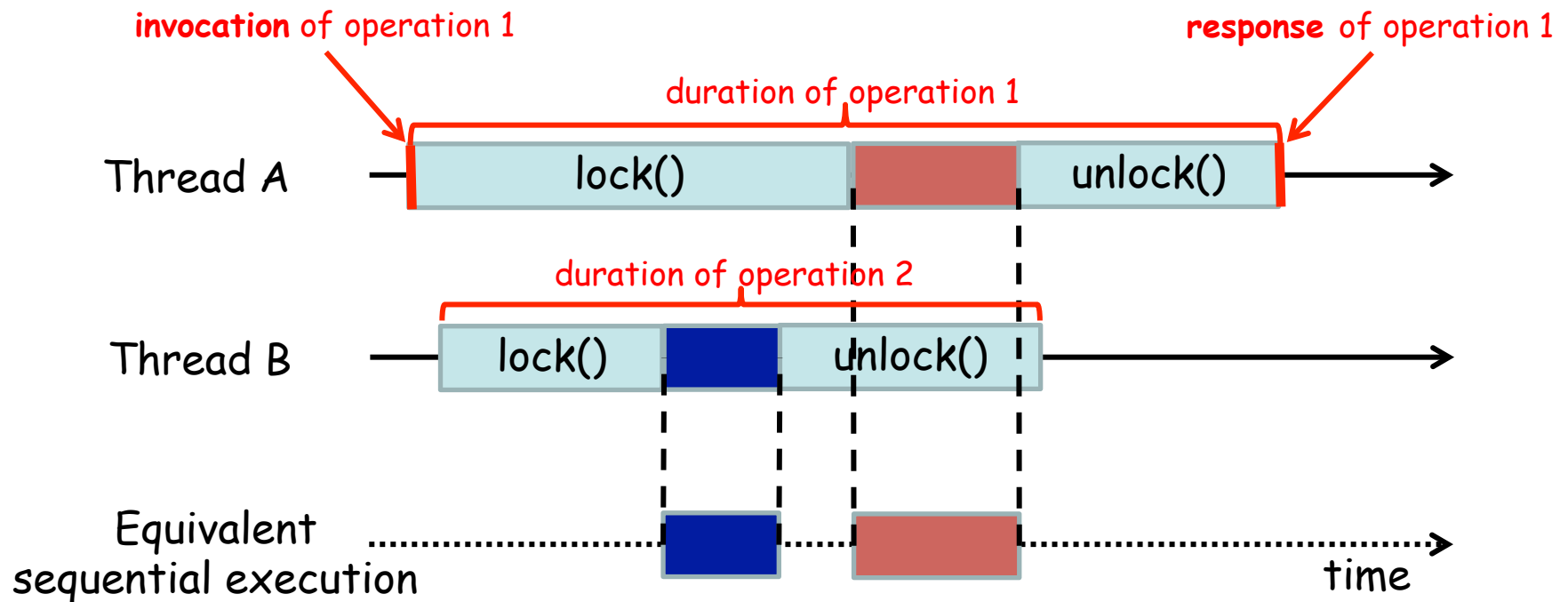


- In a **concurrent system**, operations can potentially be invoked on objects which are in intermediate states
- Hence it is more difficult to define correctness for concurrent objects
- *Linearizability* provides a **correctness condition** for concurrent objects

# Linearizability: Intuition



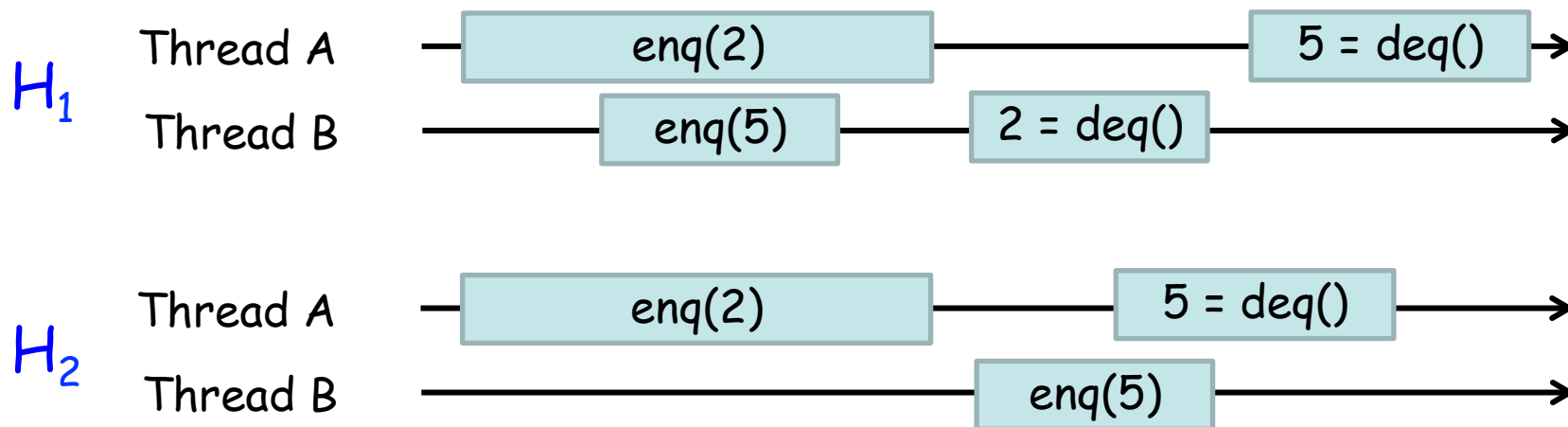
- **Idea:** A concurrent object is linearizable if **every** concurrent execution of its operations can be shown to be “equivalent” to a sequential execution



# Using the semantics of an object



- Imagine an object implementing a FIFO queue with two operations `enq(x)` and `deq()`.
- To decide whether a concurrent execution is correct, we have to use the object's intended semantics.

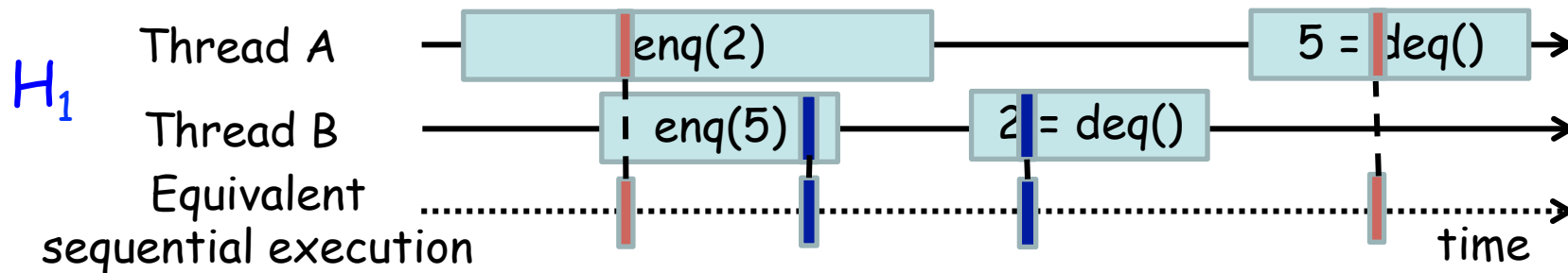


- History  $H_1$  is acceptable, it agrees with the semantics.
- History  $H_2$  is not acceptable: `enq(2)` was completed before `enq(5)` started, so 5 couldn't have been dequeued earlier.

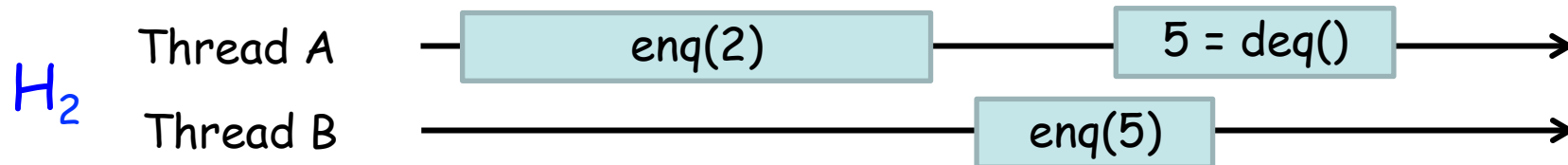
# Observation



- **Observation:** Each operation should **appear** to “take effect” **instantaneously** at some moment between its invocation and response



- For the second history, no equivalent sequential execution can be found:





- A call of an operation is split into two events:
  - Invocation:  $[A\ q.op(a_1, \dots, a_n)]$
  - Response:  $[A\ q:Ok(r)]$
- **Notation:**
  - $A$ : thread ID
  - $q$ : object
  - $op(a_1, \dots, a_n)$ : invocation of call with arguments
  - $Ok(r)$ : successful response of call with result  $r$
- A *history* is a sequence of invocation and response events
- **Example:** History  $H_1$  can be written as

$[A\ q.enq(2)], [B\ q.enq(5)], [B\ q:Ok], [A\ q:Ok],$   
 $[B\ q.deq()], [B\ q:Ok(2)], [A\ q.deq()], [A\ q:Ok(5)]$

# Projections



- We can define **projections** on objects and on threads
- Assume we have a history

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

- *Object projection:*

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

- *Thread projection:*

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

# Sequential histories



- A response *matches* an invocation if their object and thread names agree.
- A history is *sequential* if it starts with an invocation and each invocation, except possibly the last, is immediately followed by a matching response

$H = [A \ q.enq(2)], [A \ q:Ok], [B \ q.enq(5)], [B \ q:Ok], \dots$

- A sequential history is *legal* if it agrees with the sequential specification of each object.



## More definitions

---



- A call  $op_1$  *precedes* another call  $op_2$  ( $op_1 \rightarrow op_2$ ) if  $op_1$ 's response event occurs before  $op_2$ 's invocation event
- We write  $\rightarrow_H$  for the precedence relation induced by  $H$
- Example:  $q.enq(2) \rightarrow q.enq(5)$  in history  $H$
- An invocation is *pending* if it has no matching response
- A history is *complete* if it does not have pending responses
- $complete(H)$  is the subhistory of  $H$  with all pending invocations removed

# Linearizability

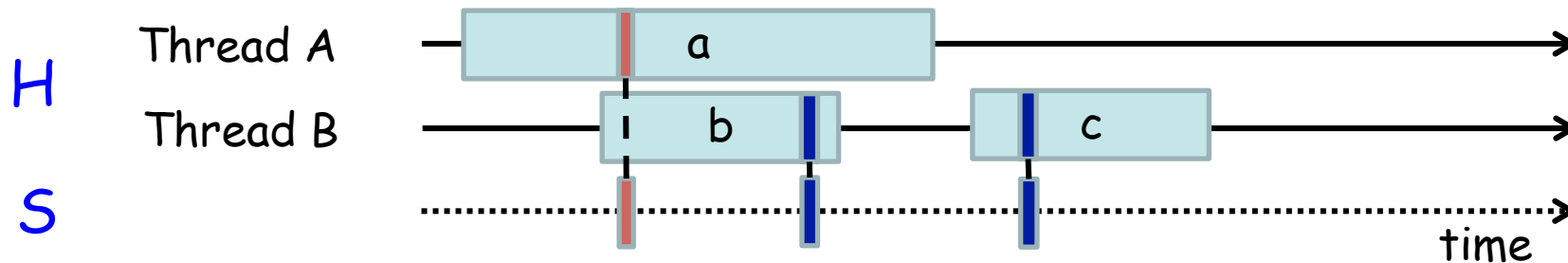


- Two histories  $H$  and  $G$  are *equivalent* if  $H|A = G|A$  for all threads  $A$
- A history  $H$  is *linearizable* if it can be extended by appending zero or more response events to a history  $G$  such that:
  - $\text{complete}(G)$  is equivalent to a legal sequential history  $S$
  - $\rightarrow_H \subseteq \rightarrow_S$

- **Example:**

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

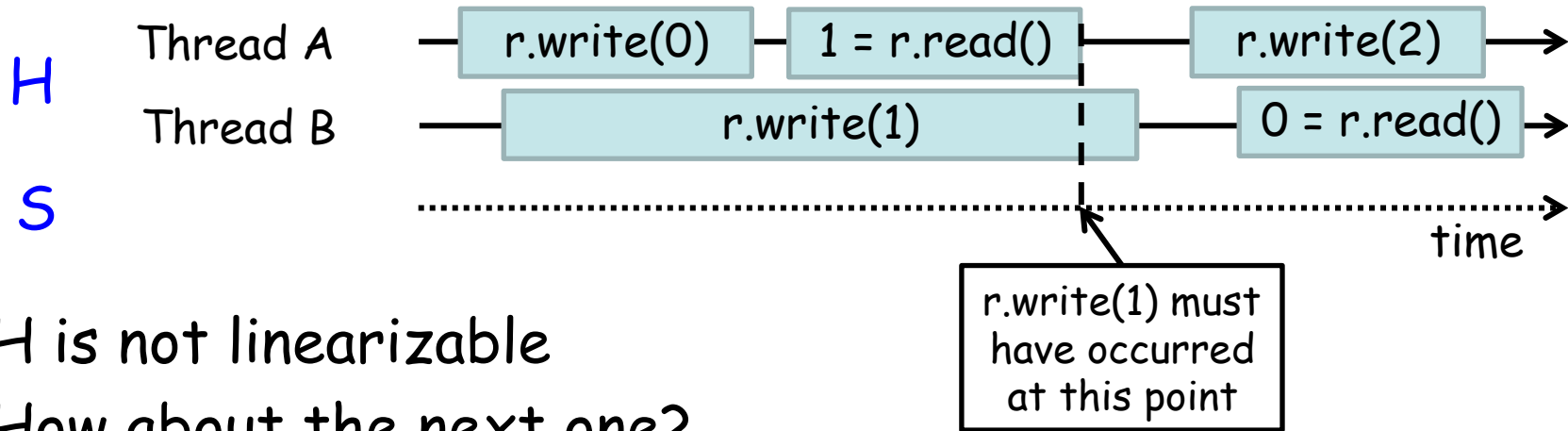
$$\rightarrow_S = \{a \rightarrow b, a \rightarrow c, b \rightarrow c\}$$



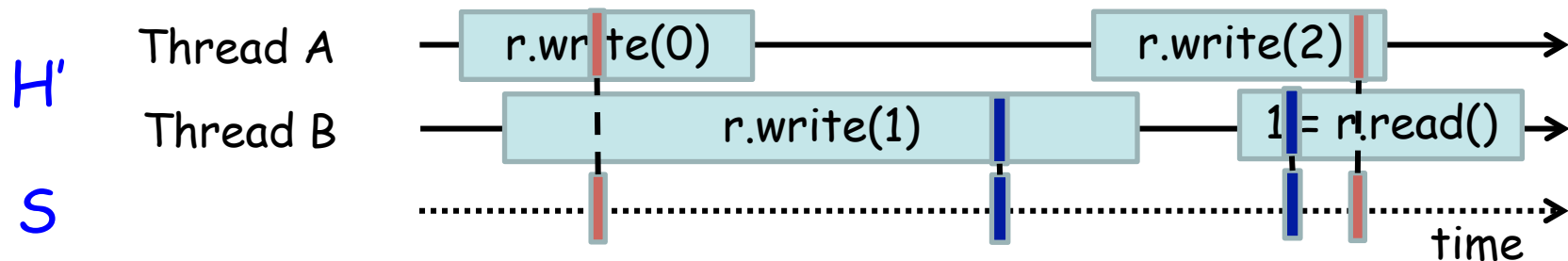
# Example: Linearizability



- Read/write registers:



- H is not linearizable
- How about the next one?



- H' is linearizable

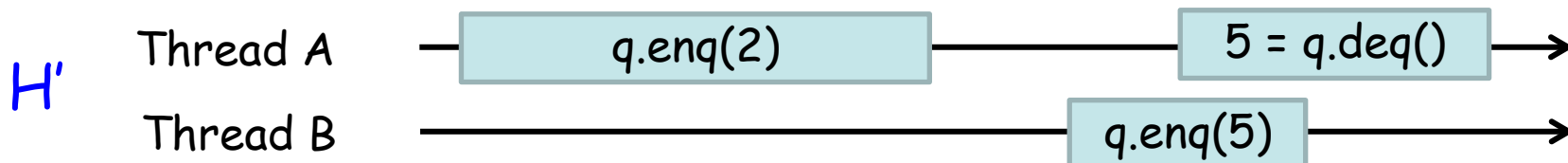
# Sequential consistency



- A history  $H$  is *sequentially consistent* if it can be extended by appending zero or more response events to a history  $G$  such that:
  - $\text{complete}(G)$  is equivalent to a legal sequential history  $S$
- **Idea:** Calls from a particular thread appear to take place in program order
- $H$  is **not** sequentially consistent:



- $H'$  is sequentially consistent but not linearizable:



# Compositionality

---



- Every linearizable history is also sequentially consistent.
- Linearizability is *compositional*:  $H$  is linearizable if and only if for each object  $H|x$  is linearizable.
- Sequential consistency on the other hand is **not** compositional.



# Software Transactional Memory (STM)

# Motivation

---



- As we have seen, lock-free programming has disadvantages in practice: algorithms can become very complex and have an unnatural structure
- This is because conventional atomic primitives can only operate on one word at a time
- Software transactional memory (STM) aims at simplifying atomic updates of multiple independent words
- STM uses the **idea of transaction from database management systems**

# Database transactions

---



- Database transaction: a sequence of operations performed within a database management systems, enjoying the ACID properties:
  - **Atomicity**: Transactions appear to execute completely, or not at all.
  - **Consistency**: Transactions preserve consistency of the database.
  - **Isolation**: Other operations cannot access data modified by a currently incomplete transaction.
  - **Durability**: All committed transactions are guaranteed to persist.
- In the context of STM, we are mostly interested in **Atomicity** and **Isolation**.



# (Hardware) transactional memory

---



- Software transactional memory is based on earlier ideas of a **multiprocessor hardware architecture** to support lock-free programming: **(hardware) transactional memory** (Herlihy and Moss, 1993)
- Not yet implemented, but implementation suggested:
  - adding some specialized cache
  - modifying **cache coherence protocols**, which maintain consistency between caches and memory and do much of the task already

# Software transactional memory

---



- Because of the lag of hardware implementation, development has focused on software implementations of the transaction idea, starting with the work of (Shavit and Touitou, 1995)
- **Idea:** Allow code to be enclosed by an **atomic**-block, with the guarantee that it executes atomically with respect to other atomic-blocks
- Currently mostly research prototypes
- The functional language Haskell offers some nice support

# STM implementations

---



- Many implementation variants are possible
- **Optimistic** implementation approach:
  - atomic-block runs without locking, but writes instead to a transaction log
  - upon completion of the atomic-block, the log is **validated** and if found consistent the changes are **committed**
  - if validation fails, the block is reexecuted



- Advantages:
  - **Simple** and effective programming model
  - Transactions may be **composed** (Harris et al., 2005)
  - Increased concurrency, no waiting for resources
- Disadvantages:
  - **Restrictions** on operations within atomic-blocks: since roll-back must be available, no externally observable effects such as IO are allowed
  - **Performance loss** with respect to fine-grained locking: with current implementations, the overhead of transaction logs and consistency checking amortizes only with larger numbers of processing units