Concepts of Concurrent Computation
Spring 2014
Lecture 8: Lock-Free Approaches

Bertrand Meyer
Sebastian Nanz
Chris Poskitt

Chair of Software Engineering
ETH zürich

What’s wrong with locks?

They are difficult to use correctly

- forget to take a lock?
- take too many locks?
- take the locks in the wrong order?
- take the wrong lock?

They are difficult to use correctly

- forget to take a lock? *danger of data race*
- take too many locks? *danger of deadlock*
- take the locks in the wrong order? *danger of deadlock*
- take the wrong lock? *???*

Blocking, faults, and performance...

- priority inversion

  => lower-priority thread preempted while holding a lock that a higher-priority thread needs

- convoying

  => multiple threads of the same priority contend repeatedly for the same lock

- fault tolerance

  => what if a faulty process halts whilst holding a lock?

- granularity of locking

  => **lock overhead** vs. **lock contention**

Blocking, faults, and performance...

- priority inversion

  => lower-priority thread preempted while holding a lock that a higher-priority thread needs

- convoying

  => multiple threads of the same priority contend repeatedly for the same lock

- fault tolerance

  => what if a faulty process halts whilst holding a lock?

- granularity of locking

  decreases with more locks

  => **lock overhead** vs. **lock contention**

  increases with more locks

Locks are not "composable" in general

- they don't support modular programming

  => i.e. 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 a "transfer" method?

Locks are not "composable" in general

- 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);
}
```

Locks are not "composable" in general

- 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);
}
```

have to add explicit locking code

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

How do we do concurrent programming without locks?

- message passing

  => no shared data at all
  => but: overheads of messaging, slower access to data, ...

- lock-free programming

  => instead of locks, use stronger atomic operations

- software transactional memory (STM)

  => based on the idea of database transactions

How do we do concurrent programming without locks?

- message passing

  => no shared data at all
  => but: overheads of messaging, slower access to data, ...

- lock-free programming

  => instead of locks, use stronger atomic operations

- software transactional memory (STM)

  => based on the idea of database transactions

Next on the agenda

1. lock-free programming

2. software transactional memory (STM)

3. linearisability and sequential consistency

Lock-free programming

- write shared-memory concurrent programs without using locks (but still ensuring thread safety)
- idea: use stronger atomic operations (typically provided by the hardware)
- designing general lock-free algorithms is difficult

=> focus instead on developing lock-free data structures
=> stack, list, queue, buffer, ...

Classes of lock-free algorithms

- typically distinguish two classes of lock-free algorithms

lock-free

wait-free

Classes of lock-free algorithms

- typically distinguish two classes of lock-free algorithms

lock-free

=> *guaranteed* *system-wide progress*

=> *i.e. infinitely often some process finishes*

wait-free

=> *guaranteed* *per-thread progress*

=> *i.e. all processes complete in a finite number of steps*

Classes of lock-free algorithms

- typically distinguish two classes of lock-free algorithms

lock-free

=> *<u>guaranteed</u> system-wide progress*

=> *i.e. infinitely often <u>some</u> process finishes*

wait-free

=> *<u>guaranteed</u> per-thread progress*

=> *i.e. all processes complete in a finite number of steps*

implies

Classes of lock-free algorithms

- typically distinguish two classes of lock-free algorithms

lock-free

=> *<u>guaranteed</u> system-wide progress*

=> *i.e. infinitely often <u>some</u> process finishes*

wait-free

=> *<u>guaranteed</u> per-thread progress*

=> *i.e. all processes complete in a finite number of steps*

implies

free from deadlock

free from deadlock <u>and</u> starvation

Compare-and-swap (CAS)

- compare-and-swap (CAS) combines a load and a store into a single atomic operation

- takes three arguments: a memory address x, an old value, and a new value

CAS (x, old, new)

- atomically reads the contents at x, and, if it contains old, updates it to new

Compare-and-swap (CAS)

- CAS must indicate whether or not it performed the substitution

  => by returning the value <u>read</u> from memory
  => or by a simple Boolean response

- latter variant sometimes called compare-and-set:

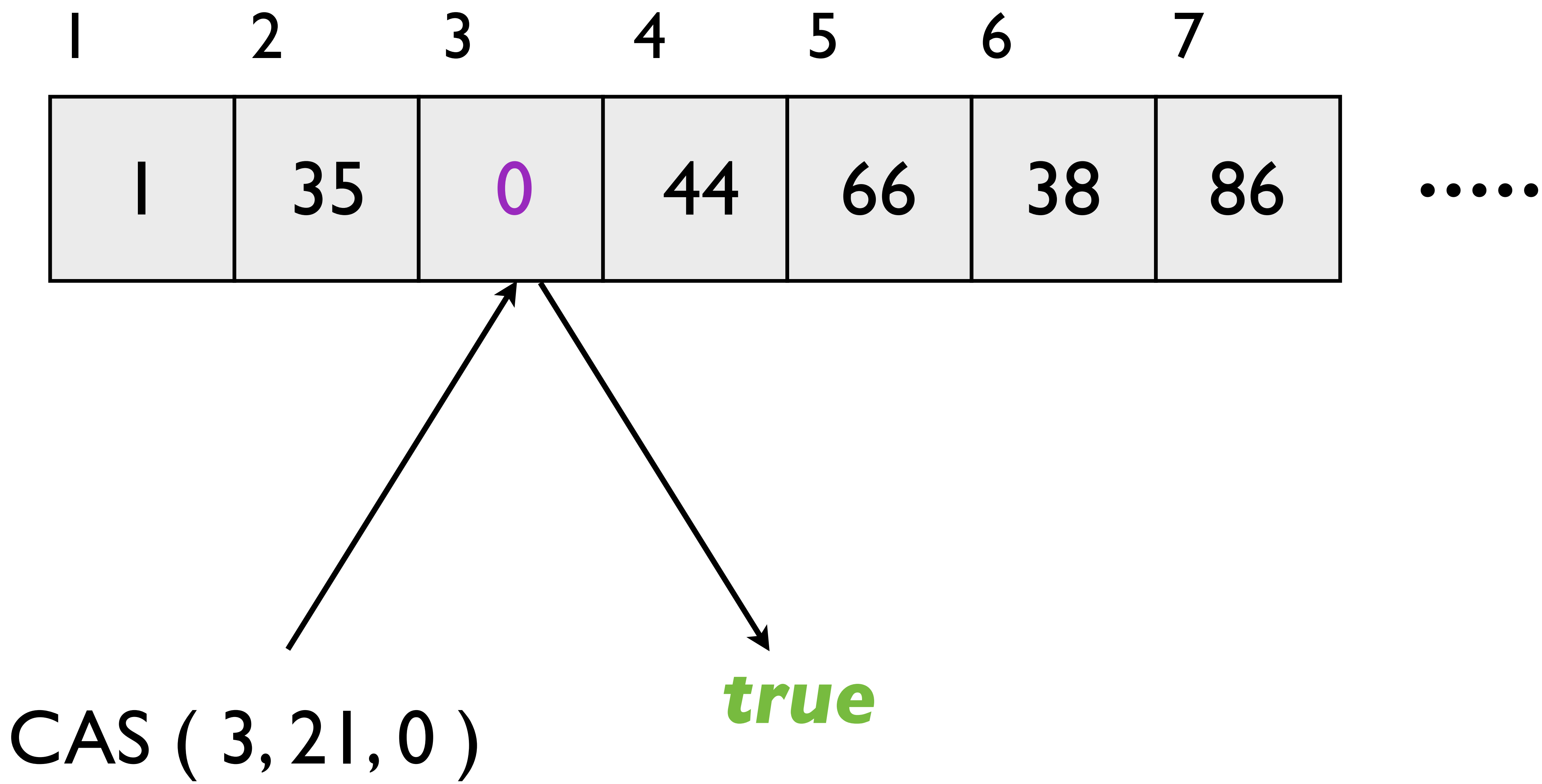
```
CAS (x, old, new)
   do-atomic
      if *x = old then
         *x := new;
         result := true
      else
         result := false
      end
   end
```

Compare-and-swap (CAS)

1	2	3	4	5	6	7	
1	35	21	44	66	38	86	.....

CAS (3, 21, 0)

Compare-and-swap (CAS)

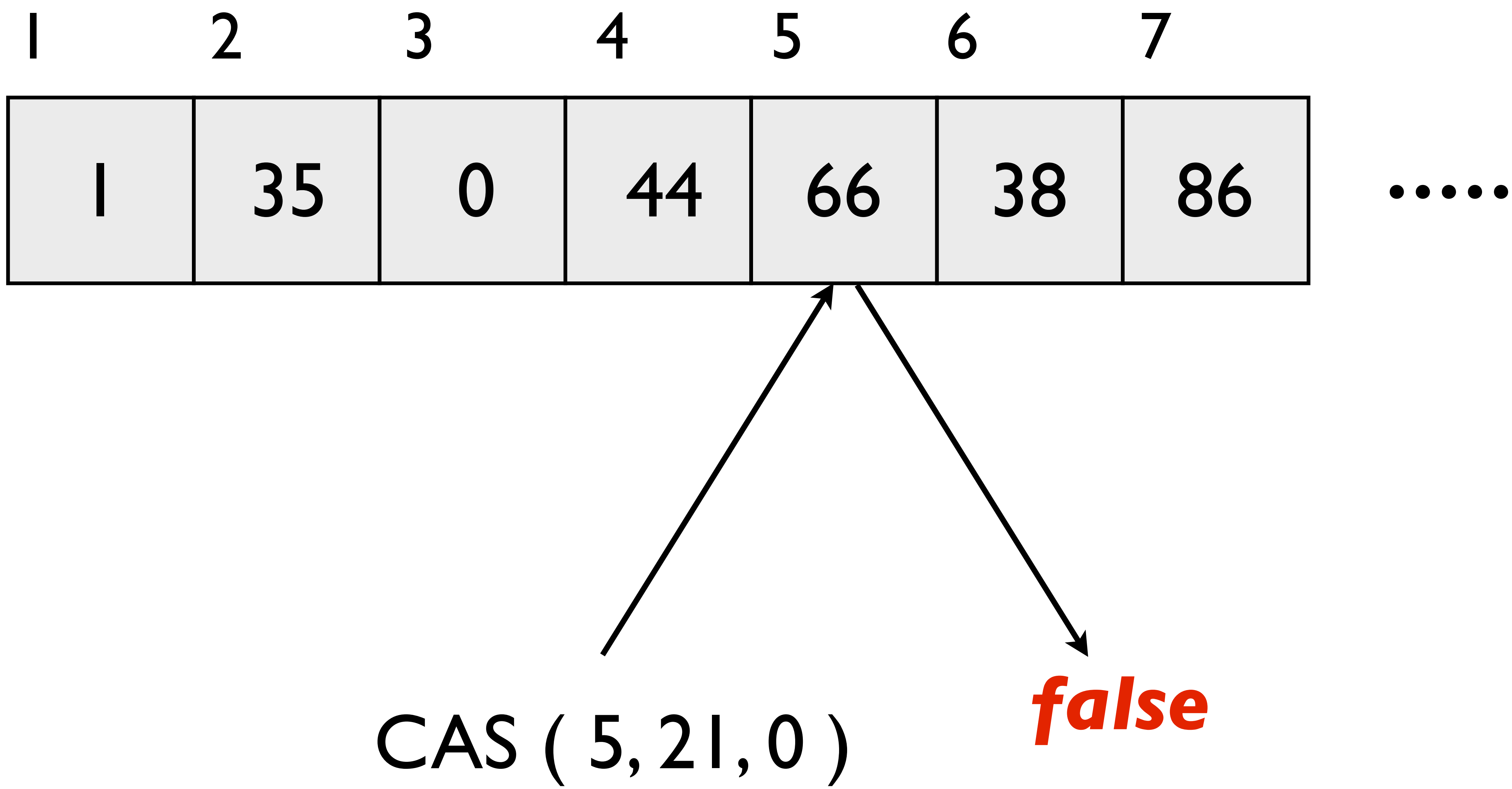


Compare-and-swap (CAS)

1	2	3	4	5	6	7	
1	35	0	44	66	38	86	…..

CAS (5, 21, 0)

Compare-and-swap (CAS)



Treiber stack

(a simple lock-free stack)

- CAS facilitates a lock-free stack implementation (due to Treiber, 1986)

- stack of integers represented as a linked list of nodes; the top of the stack denoted by the node head

```
class Node {
    Node* next;
    int item;
}

Node* head; // top of the stack
```

Treiber stack

(a simple lock-free stack)

- to implement *push* and *pop*, a common pattern is used:

  (1) read a value from the current state

  (2) compute an updated value based on the read one

  (3) atomically update the state by swapping the new for old

Treiber stack

(a simple lock-free stack)

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

Treiber stack

(a simple lock-free stack)

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

operation fails if another process has changed the head in the meantime (then loop repeats)

Treiber stack

(a simple lock-free stack)

```
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;
}
```

CAS can be fooled!

- consider the following pattern:

  T_1: a value is read from state A
  T_2: the state is changed to state B
  T_1: CAS operation does not distinguish between A and B, so assumes the state is still A

- called the ABA problem

- avoided in our stack since *push* always creates a new node (and old node's location is not freed)

Lock-free programming: discussion

- good performance in some situations, avoiding many of the problems of locks

    => deadlock, priority inversion, ...

- but difficult to correctly implement lock-free algorithms

    => e.g. the ABA problem
    => can lead to unnatural structuring of algorithms

- focused on lock-free data structures (well-established algorithms and implementations available)

Next on the agenda

1. lock-free programming ✓
2. software transactional memory (STM)
3. linearisability and sequential consistency

Motivating STM

- the conventional atomic primitives of lock-free approaches operate on one memory location at a time

  => algorithms can have an unnatural structure

- software transactional memory (STM) aims at simplifying atomic updates of multiple independent memory locations

- inspiration: transactions in database management systems

Database transactions

- a database transaction is a sequence of operations performed within a DBMS enjoying these properties:

  => *Atomicity:* transactions appear to execute completely or not at all
  => *Consistency:* transactions preserve consistency of the DB
  => *Isolation:* other operations cannot access data modified by an incomplete transaction
  => *Durability:* all committed transactions guaranteed to persist

- for STM, atomicity and isolation are most interesting

Software transactional memory (STM)

- development has focused on software implementations

  => starting with the work of Shavit & Touitou, 1995

  => based on earlier ideas of a multiprocessor *hardware architecture* to support lock-free programming (Herlihy & Moss, 1993)

- idea: allow code to be enclosed by an **atomic**-block

  => guarantee: executes *atomically* with respect to <u>other</u> atomic-blocks

(photo by Jukka Suomela)

Implementing STM

- an "optimistic" implementation scheme:

  => atomic-blocks run without locking; write to transaction log
  => onus placed on readers to check consistency
  => transaction can be committed, aborted, and/or re-executed

- many implementations of STM (quality varies!)

  => nice support in concurrent Haskell
  => facilitates *composability* and *modularity*
  => *http://research.microsoft.com/pubs/67418/2005-ppopp-composable.pdf*

STM: discussion

advantages:

=> *simple and effective programming model*
=> *transactions can be composed (Harris et al., 2005)*
=> *increased concurrency, no waiting for resources*

disadvantages:

=> *restrictions on operations within atomic-blocks, since rollback must be available (e.g. no externally observable effects)*
=> *performance loss with respect to fine-grained locking; the overhead of transaction logs and consistency checking*

Next on the agenda

1. lock-free programming ✓
2. software transactional memory (STM) ✓
3. linearisability and sequential consistency

Correctness conditions

- we can understand the execution of a system as operations of a collection of (sequential) processes on data objects

  => objects equipped with types and operations

- in a sequential system, it is easy to specify the behaviour of operations

  => pre- and postconditions

$$\{pre\}\ q.op\ \{post\}$$

  => operations cannot be called on objects that are in an "intermediate state"

Concurrent objects

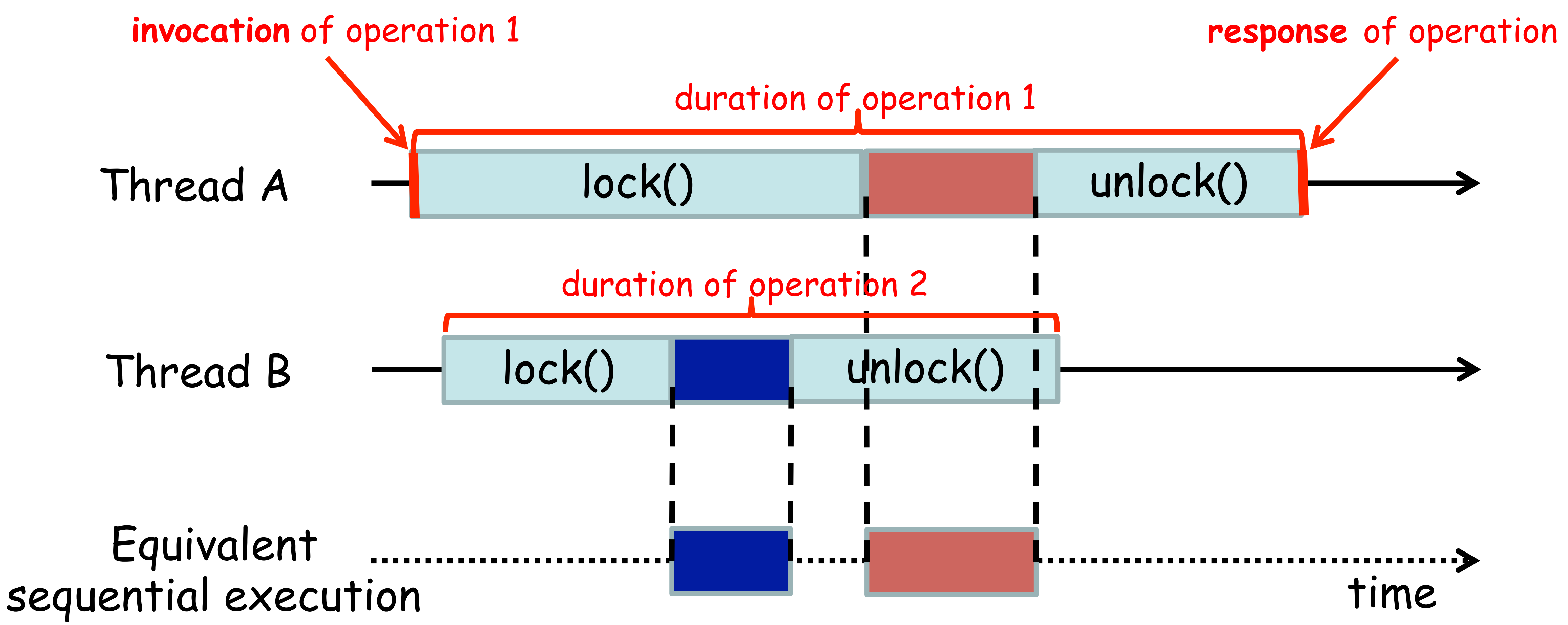
- in a concurrent system, operations can potentially be invoked on objects that are in intermediate states

- more difficult to define correctness for concurrent objects

- linearisability provides a correctness condition for concurrent objects

Linearisability: the intuition

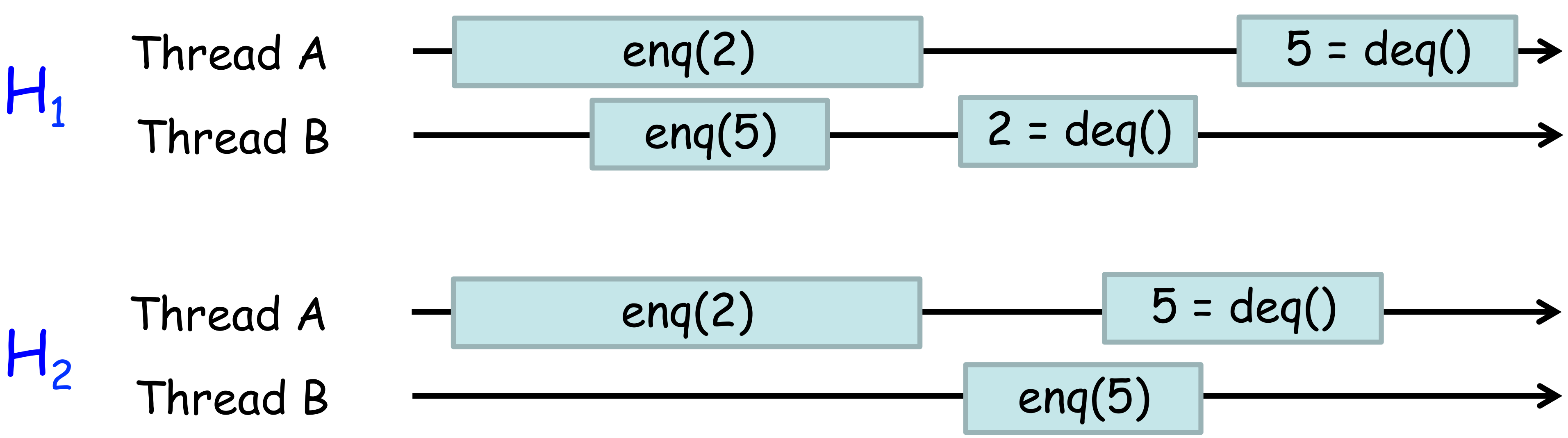
- idea: a concurrent object is linearisable if <u>every</u> concurrent execution of its operations can be shown to be "equivalent" (in some sense) to a sequential execution



Using the semantics of an object

- imagine an object implementing a FIFO queue with two operations, enq(x) and deq()

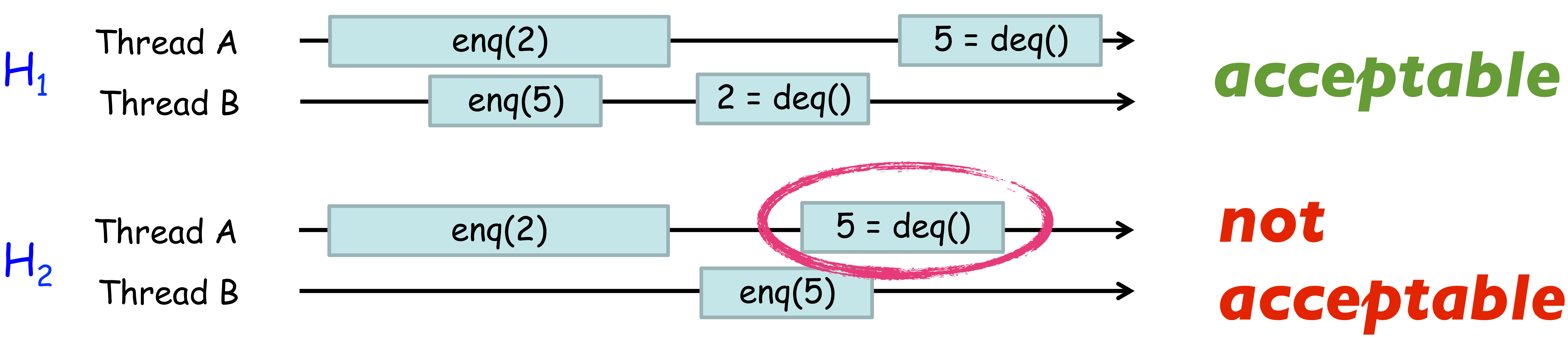
- decide whether a concurrent execution is correct using the object's intended semantics



Using the semantics of an object

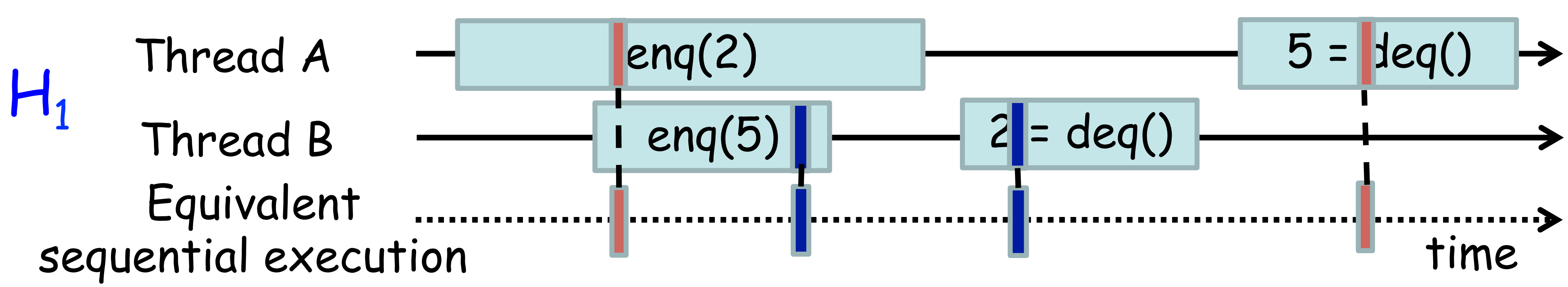
- imagine an object implementing a FIFO queue with two operations, enq(x) and deq()

- decide whether a concurrent execution is correct using the object's intended semantics

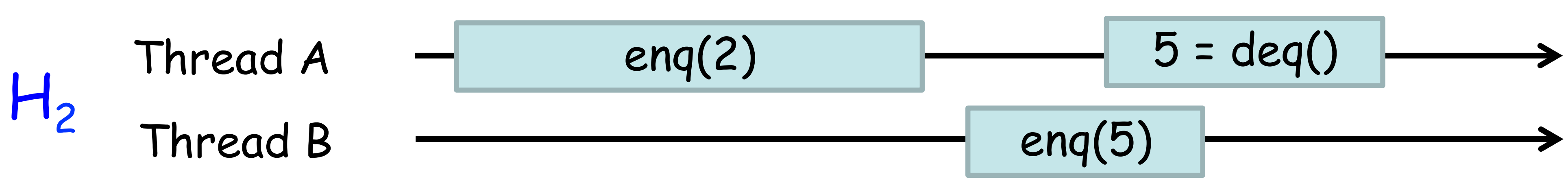


Observation

- **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



Histories

- a call of an operation is split into two events:

  invocation [A q.op(a_1, ..., a_n)]
  response [A q:Ok(r)]

- where A is a thread ID, q an object, op(a_1, ..., a_n) an invocation of call with arguments, and Ok(r) a successful response of call with result r

- a history is a sequence of invocation / response events

Histories

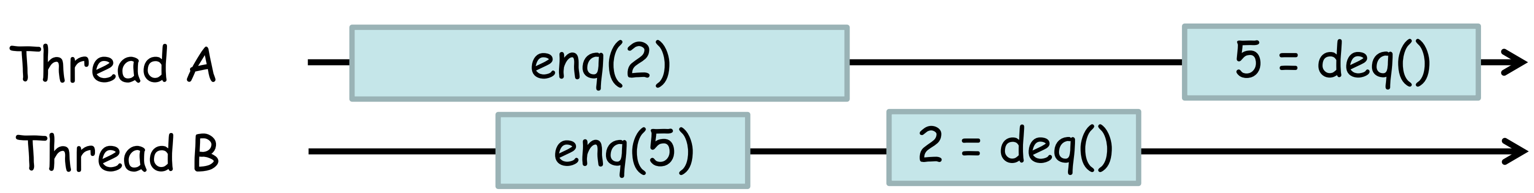
- a call of an operation is split into two events:

  invocation [A q.op(a_1, ..., a_n)]
  response [A q:Ok(r)]

- where A is a thread ID, q an object, op(a_1, ..., a_n) an invocation of call with arguments, and Ok(r) a successful response of call with result r

- a history is a sequence of invocation / response events

H_1 [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], ...

- 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 -> op_2) if op_1's response event occurs before op_2's invocation event

- we write $->_H$ for the precedence relation induced by H

  => e.g. q.enq(2) $->_H$ q.enq(5)

- 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

Linearisability: the definition

- two histories H and G are equivalent if H|A = G|A for all threads A

- a history H is linearisable if it can be extended to a history G by adding zero or more response events, such that:

  => complete(G) is equivalent to some legal sequential history S
  => $\to_H \subseteq \to_S$ (i.e. the precedences of H are maintained)

Linearisability: the definition

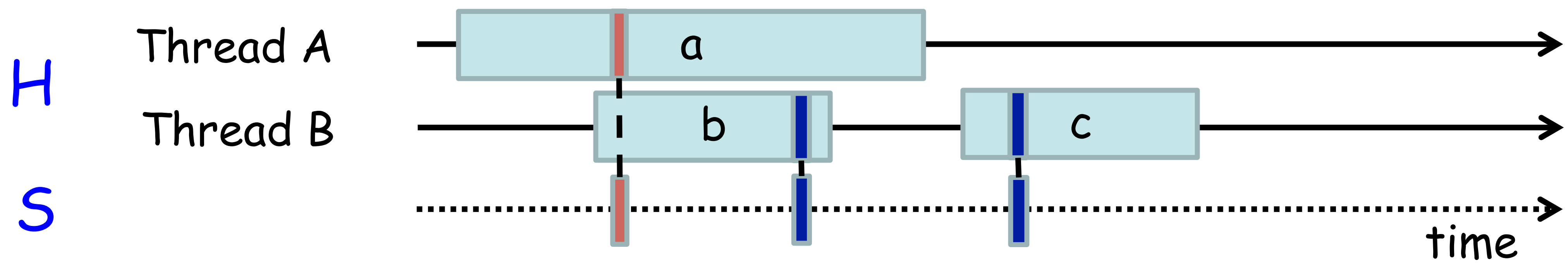
- two histories H and G are equivalent if H|A = G|A for all threads A

- a history H is linearisable if it can be extended to a history G by adding zero or more response events, such that:

  => complete(G) is equivalent to some legal sequential history S
  => $\rightarrow_H \subseteq \rightarrow_S$ (i.e. the precedences of H are maintained)

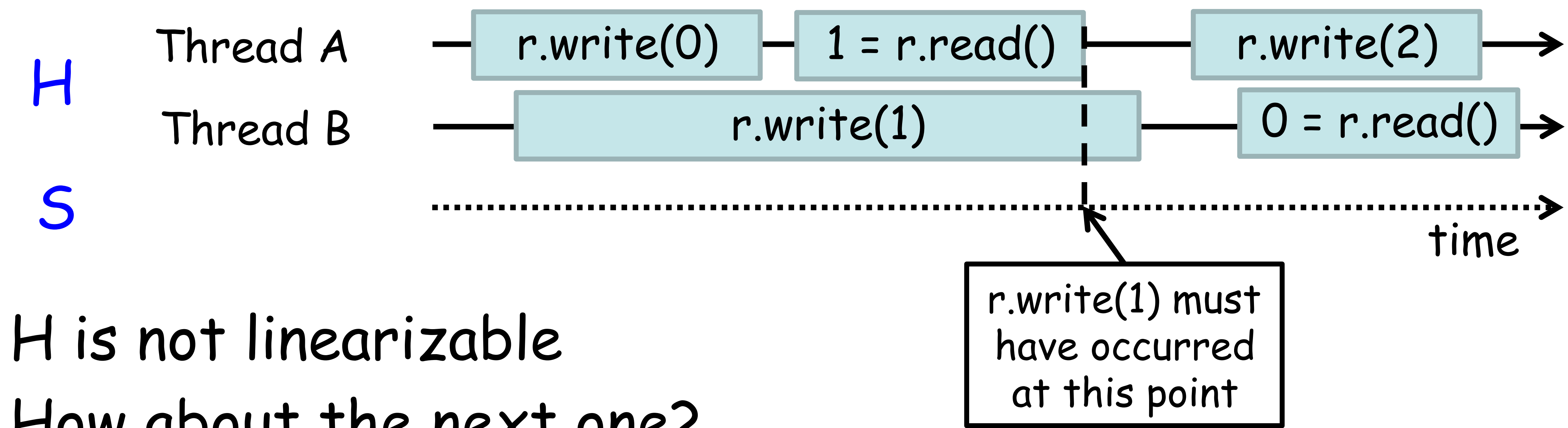
Example:

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



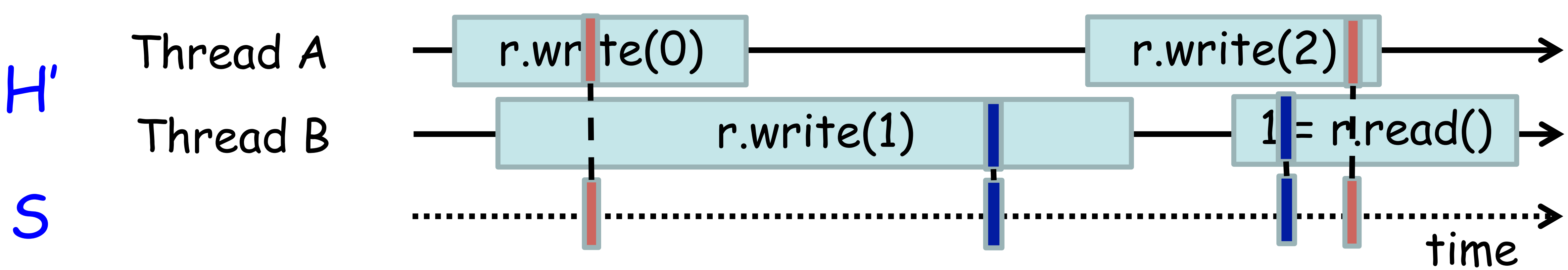
Example: linearisability

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 to a history G by adding zero or more response events, such that:

  => complete(G) is equivalent to some legal sequential history S

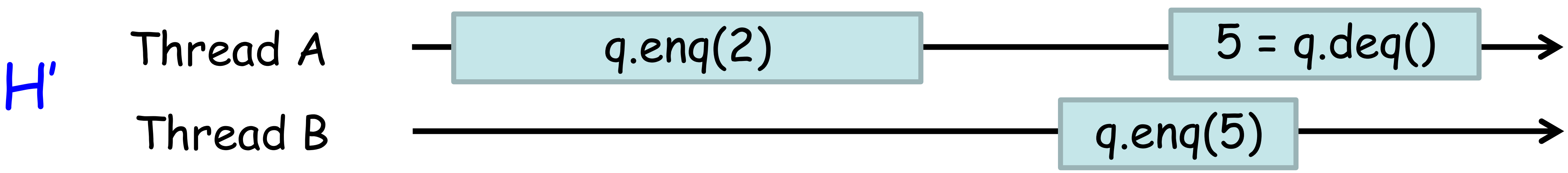
- note that $\to_H \subseteq \to_S$ is not a requirement

- idea: calls from a particular thread appear to take place in program order

Sequential consistency

H is not sequentially consistent:

H' is sequentially consistent but not linearizable:



Compositionality

- every linearisable history is also sequentially consistent
- linearisability is compositional: H is linearisable if and only if each object H|x is linearisable
- sequential consistency is not compositional

Thanks! Questions?

1. lock-free programming
2. software transactional memory (STM)
3. linearisability and sequential consistency