

# Advanced Multiprocessor Programming

Jesper Larsson Träff

[traff@par.tuwien.ac.at](mailto:traff@par.tuwien.ac.at)

Research Group Parallel Computing

Faculty of Informatics, Institute of Computer Engineering

Vienna University of Technology (TU Wien)

## The relative power of synchronization primitives (Chap. 5)

- What registers cannot do: The basic result!
- Concurrent correctness
- More powerful synchronization primitives

Maurice Herlihy: Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124-149, 1991

Maurice Herlihy, Jeannette Wing: Linearizability: A correctness condition for concurrent objects. *ACM TOPLAS* 12(3): 463-492, 1990

### Consensus problem:

n threads need to agree on a common value among local values provided by the threads, and no thread should possibly block another thread

More formally:

A **consensus object** has a method `decide(v)` that can be called **at most once per thread** with some thread local **value v**.  
`decide(v)` returns a value that is

- Consistent: All threads return **the same value**
- Valid: **One of the values** provided in the calls is returned

The `decide()` method is **wait-free**: Any call to the method will finish in a finite number of operations, independently of other threads actions

Consensus protocol:

A class (algorithm) that implements consensus in a wait-free manner

A class **solves** the n-thread consensus problem, if there exists a consensus protocol for n threads using **any number of objects of the class** and **any number of atomic registers**. The class can be implemented in software (algorithm) or hardware (special registers and instructions)

Definition:

The **consensus number** of a class (algorithm/instruction) is the largest number of threads n for which the class solves the consensus problem. If no largest (finite) n exists, the consensus number is  $\infty$

Consensus interface:

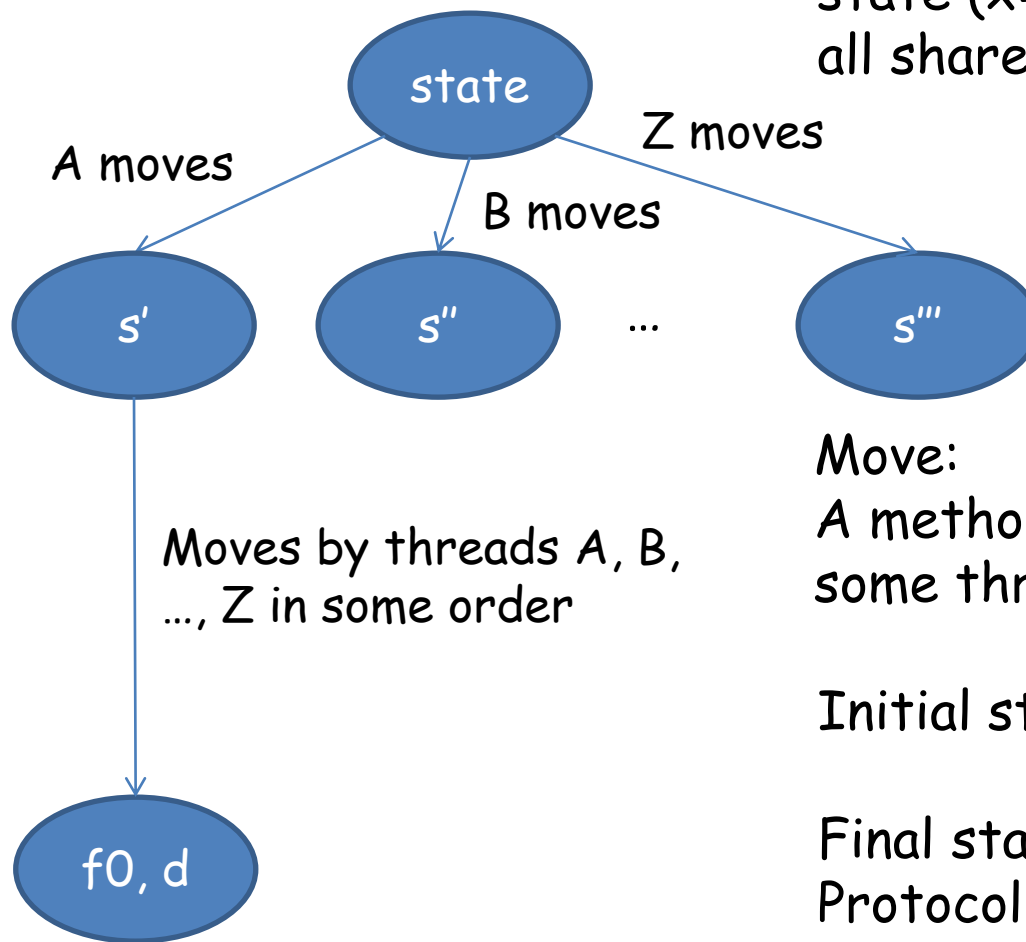
```
public interface Consense<T> {  
    T decide(T v); // decide on v provided by each thread  
}
```

Generic consensus protocol:

```
public abstract class Consensus<T> implements Consense {  
    protected T[] proposed = (T[])new Object[N];  
    void propose(T v) {  
        proposed[ThreadID.get()] = v;  
    }  
    abstract public T decide(T v);  
}
```

Not a solution (why?) to the consensus problem, but useful idea:  
 first thread that calls decide() determines value to be  
 returned by other threads

```
class LOCKconsensus extends Consensus {
    private int who = -1; // special non-thread ID
    private Lock lock;
    public Object decide(Object value) {
        lock.lock();
        try {
            propose(value); // store value
            if (who == -1) who = ThreadID.get(); // first thread
            return proposed[who];
        } finally {
            lock.unlock();
        }
    }
}
```



State:  
state ( $x=v$ ) of all threads, and  
all shared objects/variables

Move:  
A method call/state update by  
some thread

Initial state: Before any move

Final state:  
Protocol state after all threads  
have finished. **Decision value  $d$**   
is the value decided in that  
state.

Lemma: Protocol is wait-free, so tree is finite

Special case:

Binary consensus, each thread provides value true/false, 0/1

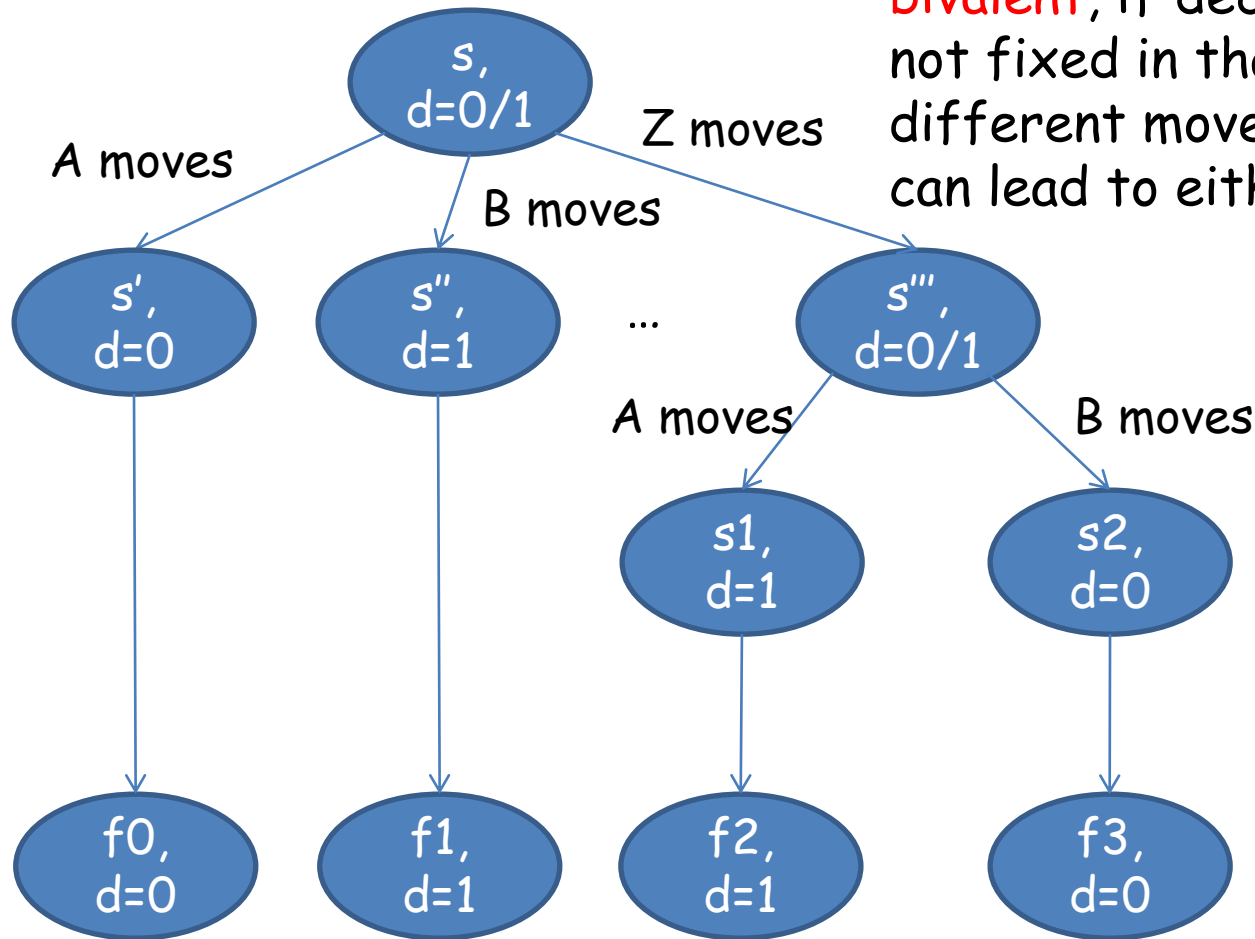
In the final state of a binary consensus protocol, the decision value  $d$  is therefore also either 0 or 1

**Note:**

The decision value  $d$  is not necessarily some explicit, actual register where the decision can be read off. A state has decided  $d=0$  or  $d=1$  if in that state the value that will eventually be returned to each thread is fixed.



Binary consensus state is **bivalent**, if decision value is not fixed in that state: different move sequences can lead to either  $d=0$  or  $d=1$



**Univalent:**  $d$  is now fixed  
 1-valent:  $d=1$   
 0-valent:  $d=0$

Lemma:

Every  $n$ -thread consensus protocol has an initial bivalent state

## Proof:

Consider an initial state where thread  $A$  contributes value 0, all other threads value 1.

If  $A$  finishes the protocol (it can: **wait-free**) before the remaining threads make a move,  $d=0$ , because  $A$  cannot know anything about the other threads value (0/1) before they make at least one move. Similarly, if some other thread  $B$  finishes before  $A$  or any other thread makes a move,  $d=1$ .

Thus, the initial state is bivalent

The “last” bivalent state is **critical**: A bivalent state is said to be **critical**, if **any move** by **any thread** will make the next state univalent

Lemma:

Any consensus protocol has a critical state

Proof:

Start in the initial **bivalent** state. As long as there is a thread that can move without making the state univalent, make the move. Since the protocol is wait-free, the state transition tree is finite. Therefore, eventually a state is reached where no such move is possible. This state is **critical**

### Theorem:

Atomic registers have consensus number 1

Proof:

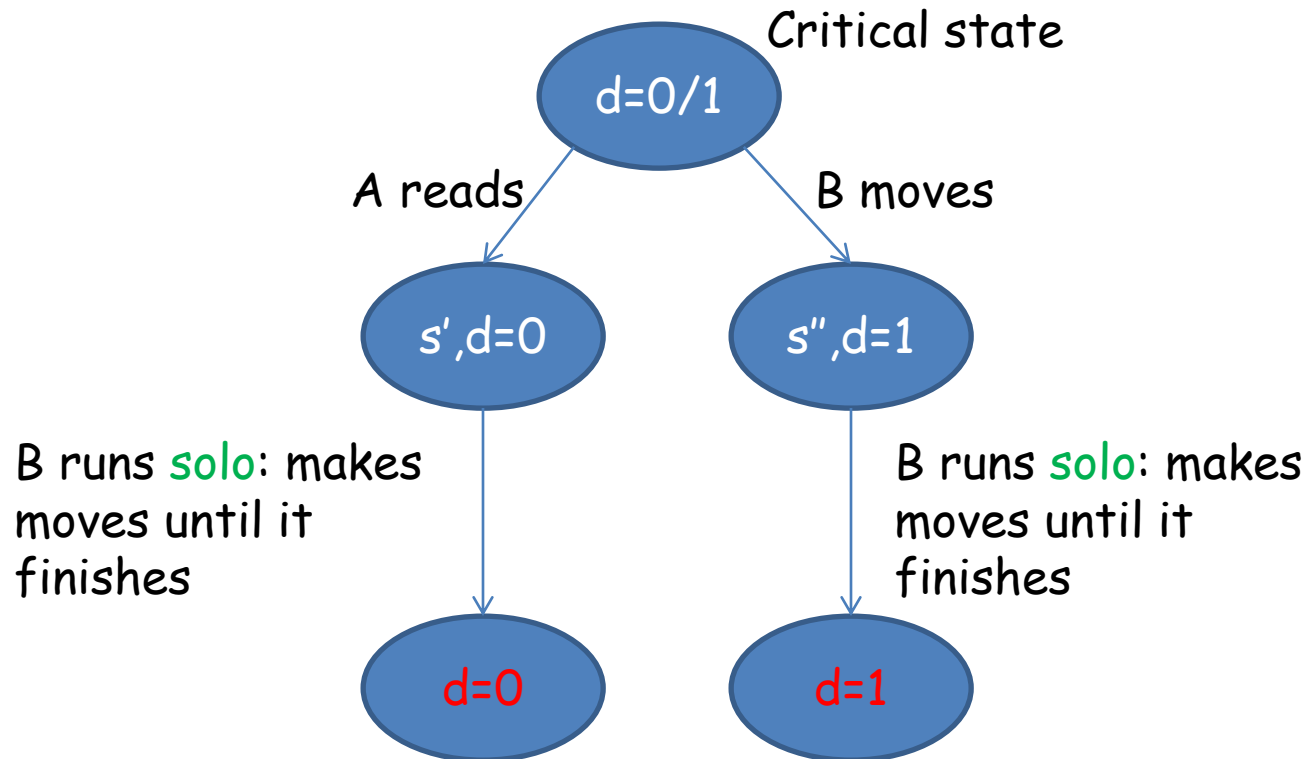
By **contradiction**. Suppose there exist a binary consensus 2-thread protocol using only atomic registers.

Run the protocol into a critical state. Assume thread A's next move leads to a 0-valent state, thread B's next move to a 1-valent state.

Three possibilities:

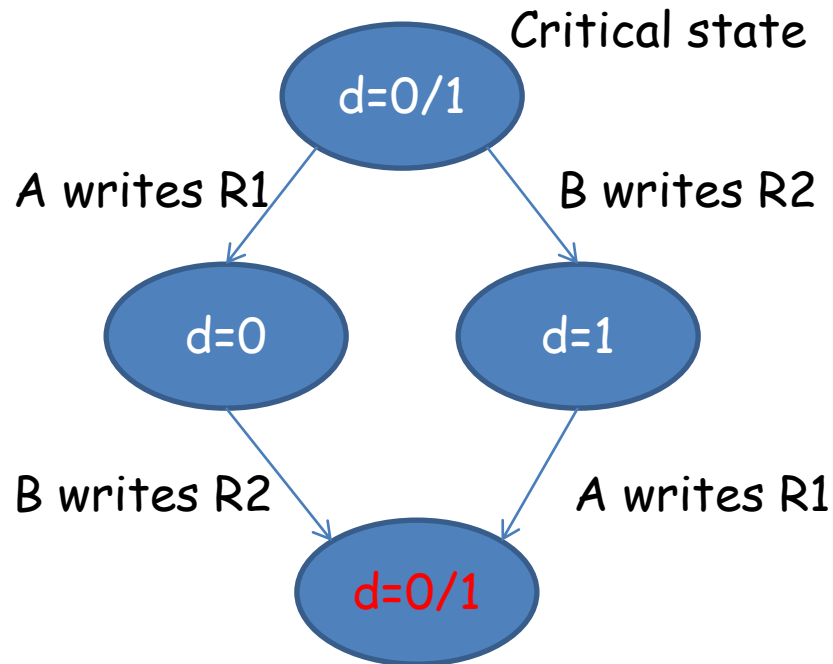
1. A's next move will read a register, B's will read or write
2. A's and B's next move will write to different registers
3. A's and B's next move will write to same register

## 1. Thread A's move will read



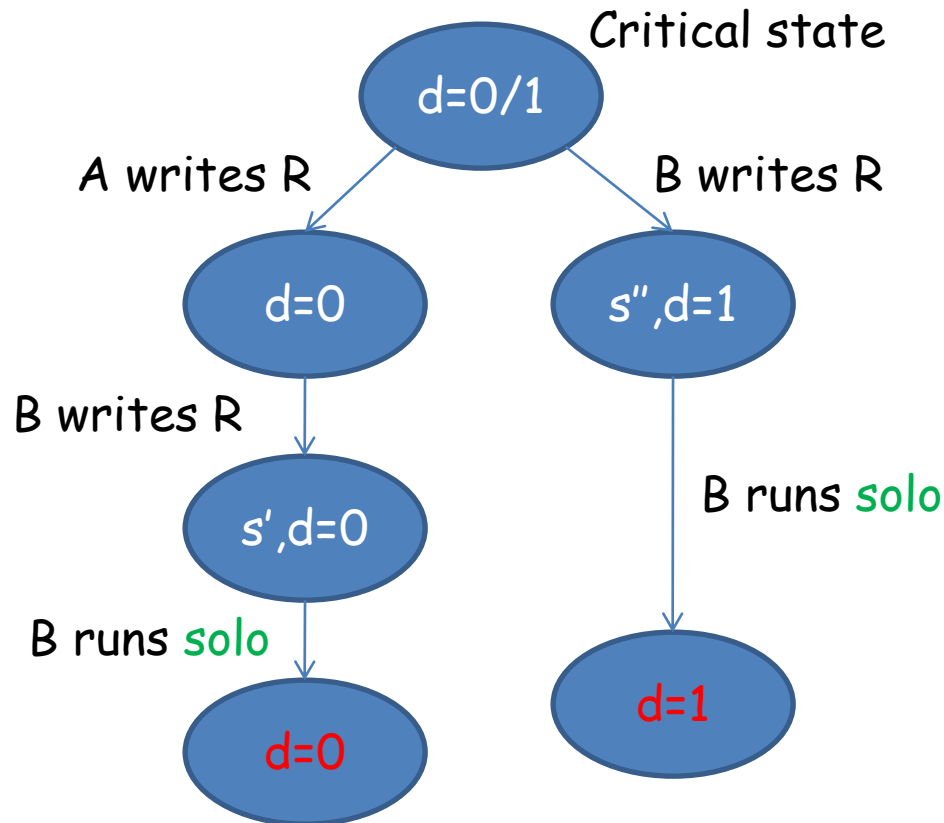
**Contradiction:** States  $s'$  and  $s''$  are indistinguishable to B, since A only read (local state change for thread A cannot have any effect for B), therefore B would have had to make the same decision in  $s'$  and  $s''$

## 2. Thread A and B will write to different registers



**Contradiction:** Since neither thread can determine which went first (the two writes “commute”), both sequences should lead into the same (univalent) state; but by critical state assumption, do not.

### 3. Thread A and B will write to same register



**Contradiction:**  $s'$  and  $s''$  are indistinguishable to B, since in  $s'$  B overwrote A's write to R. Therefore, B must decide the same value from  $s'$  and  $s''$

Main Corollary:

Atomic registers alone are insufficient to construct wait-free implementations of any concurrent object with consensus number greater than one

**Therefore:**

For constructing wait- (and lock-) free data structures/objects, (hardware support for) synchronization primitives that are more powerful than atomic reads and writes are needed



## Stronger $(m,n)$ -assignment registers

Let  $n \geq m > 1$ . An  $(m,n)$ -assignment register has  $n$  fields, and can atomically write  $m$  (index,value) pairs to  $m$  indexed fields.

The value for a given index can be read atomically.

**Note:**  $(m,n)$ -assignment is the “dual” of atomic snapshots

Lemma: The atomic snapshot object has consensus number 1  
(since it can be implemented with atomic registers)

Example: **lock-based** implementation of a (2,3)-assignment register

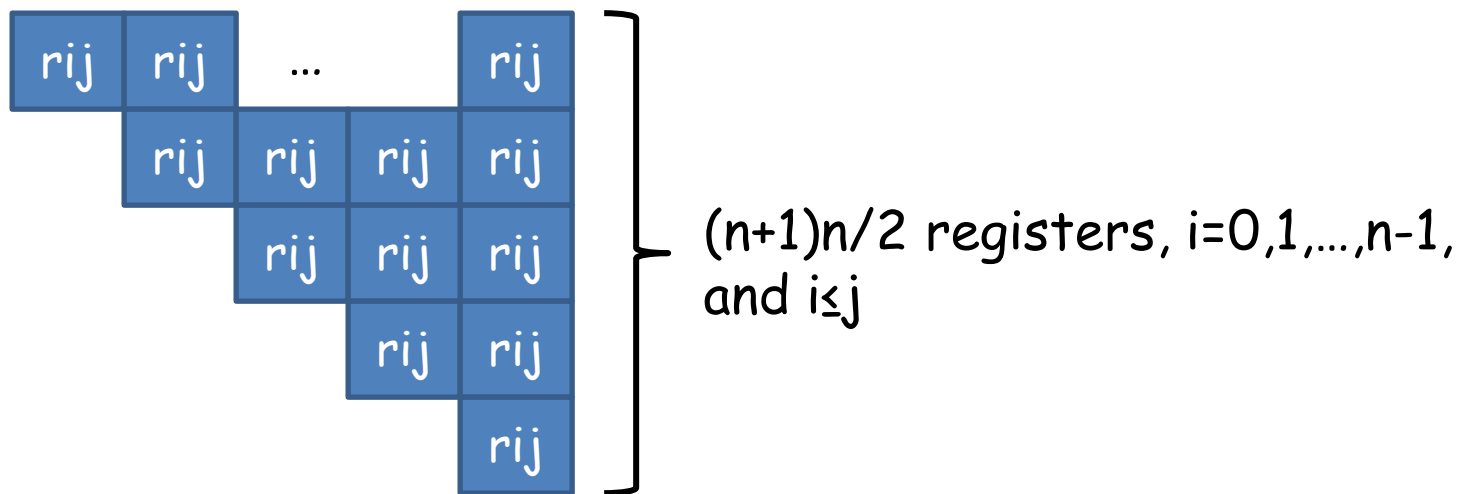
```
public class Assign23 {  
    int[] r23 = new int[3]; // use array for the 3 values  
    public Assign23(int init) {  
        r23[0] = init; r23[1] = init; r23[2] = init;  
    }  
    public synchronized void assign(T v0, T v1,  
                                     int i0, int i1) {  
        // atomically write 2 out of 3  
        r[i0] = v0; r[i1] = v1;  
    }  
    public synchronized int read(int i) {  
        return r[i];  
    }  
}
```

Java synchronized makes method call atomic (critical section)

### Theorem:

Atomic  $(n, n(n+1)/2)$ -register assignment for  $n > 1$  has consensus number at least  $n$

Proof: By **construction**, by giving a consensus algorithm for  $n$  threads



Initially, all registers are set to special  $\dagger$  value

decide(v):

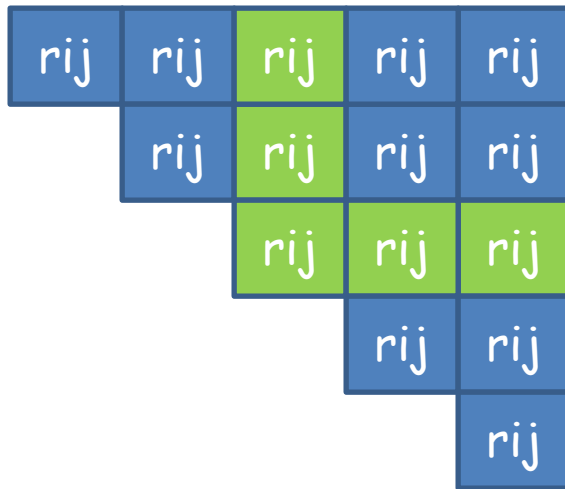
1. Thread  $i$  assigns "row"  $r_{ij} = v$ ,  $i \leq j$ , and "column"  $r_{ji} = v$ ,  $j < i$



Initially, all registers are set to special  $\dagger$  value

decide(v):

1. Thread  $i$  assigns "row"  $r_{ij} = v$ ,  $i \leq j$ , and "column"  $r_{ji} = v$ ,  $j < i$

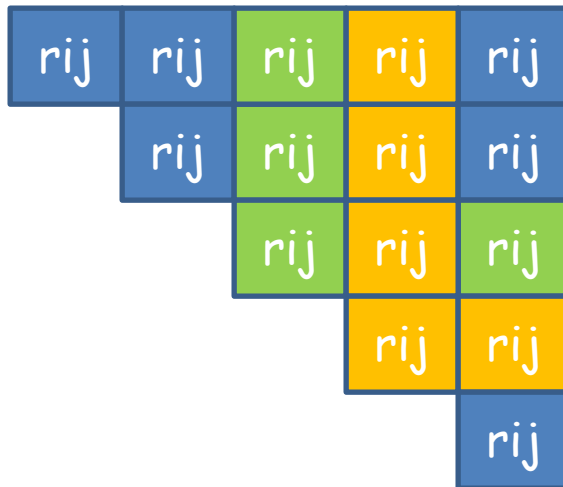


Thread 2 calls decide(v2)

Initially, all registers are set to special  $\dagger$  value

decide(v):

1. Thread  $i$  assigns "row"  $rij = v$ ,  $i \leq j$ , and "column"  $rji = v$ ,  $j < i$



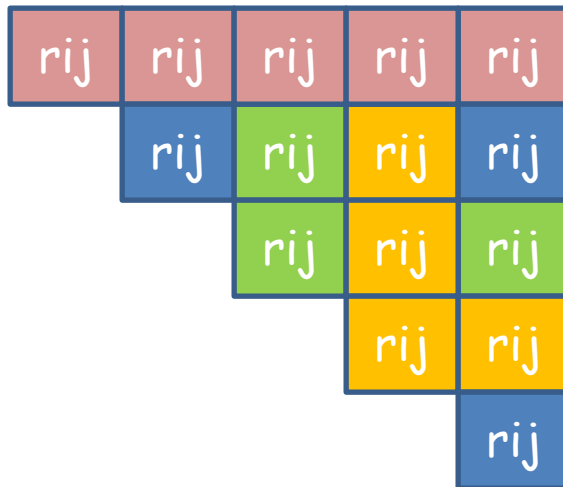
Thread 2 calls decide(v2)

Thread 3 calls decide(v3)

Initially, all registers are set to special  $\dagger$  value

decide(v):

1. Thread  $i$  assigns "row"  $rij = v$ ,  $i \leq j$ , and "column"  $rji = v$ ,  $j < i$



Thread 2 calls decide(v2)

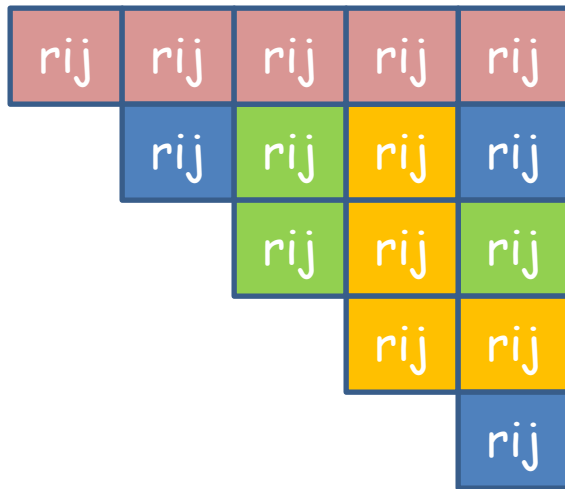
Thread 3 calls decide(v3)

Thread 0 calls decide(v0)

Initially, all registers are set to special  $\dagger$  value

decide(v):

1. Thread  $i$  assigns "row"  $r_{ij} = v$ ,  $i \leq j$ , and "column"  $r_{ji} = v$ ,  $j < i$



Thread 2 calls decide(v2)

Thread 3 calls decide(v3)

Thread 0 calls decide(v0)

**Note:**

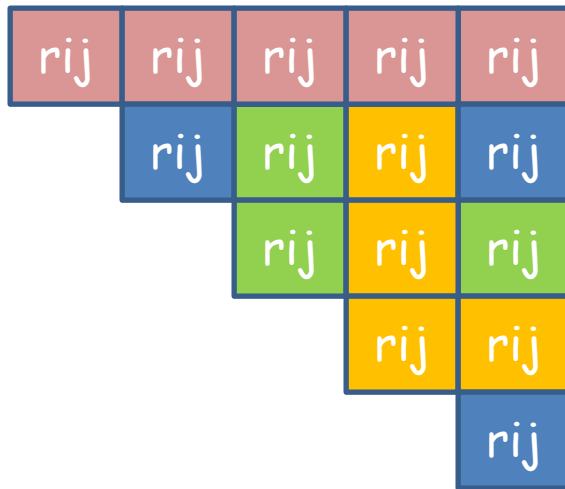
Only thread  $i$  writes to  $r_{ii}$ . If  $r_{jj} \neq \dagger$  then thread  $j$  has called decide()

Initially, all registers are set to special  $\dagger$  value



decide(v):

1. Thread  $i$  assigns "row"  $r_{ij} = v$ ,  $i \leq j$ , and "column"  $r_{ji} = v$ ,  $j < i$



Thread 2 calls decide(v2)

Thread 3 calls decide(v3)

Thread 0 calls decide(v0)

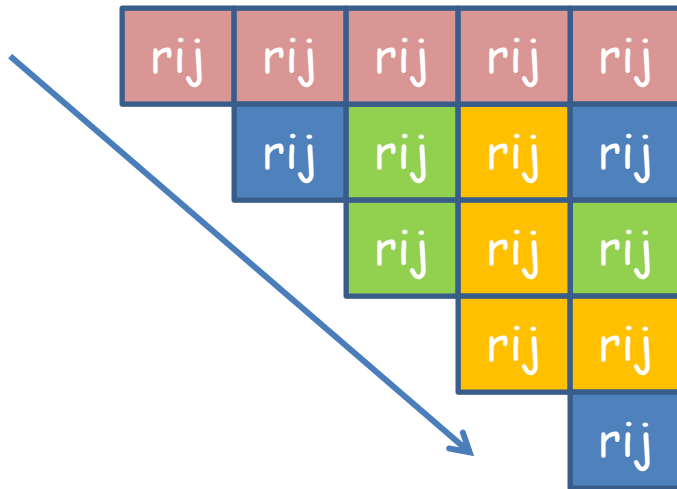
**Idea:**

Find the first thread that executed decide(), return this value as consensus value

Initially, all registers are set to special  $\dagger$  value

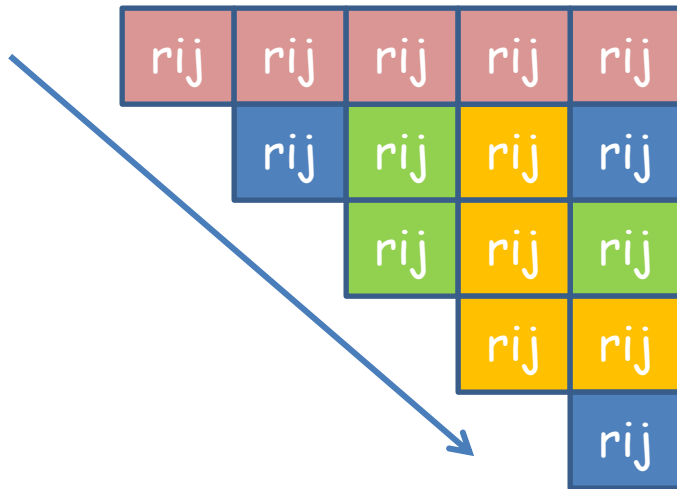
decide(v):

1. Thread  $i$  assigns "row"  $r_{ij} = v$ ,  $i \leq j$ , and "column"  $r_{ji} = v$ ,  $j < i$
2. Scan diagonal, read  $r_{ii}$ , for  $i=0, 1, \dots$ :



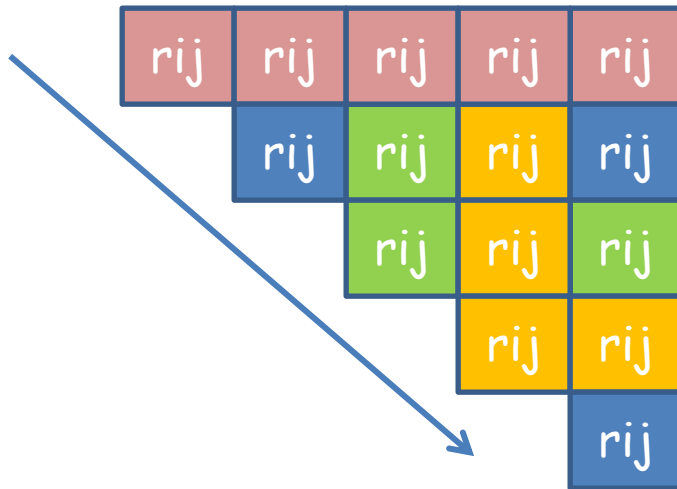
decide(v):

1. Thread  $i$  assigns "row"  $r_{ij} = v$ ,  $i \leq j$ , and "column"  $r_{ji} = v$ ,  $j < i$
2. Scan diagonal, read  $r_{ii}$ , for  $i=0, 1, \dots$ :
3. Let  $j$  be first index  $i$  where  $r_{ii} \neq t$ ;  $j=i$  is candidate for result



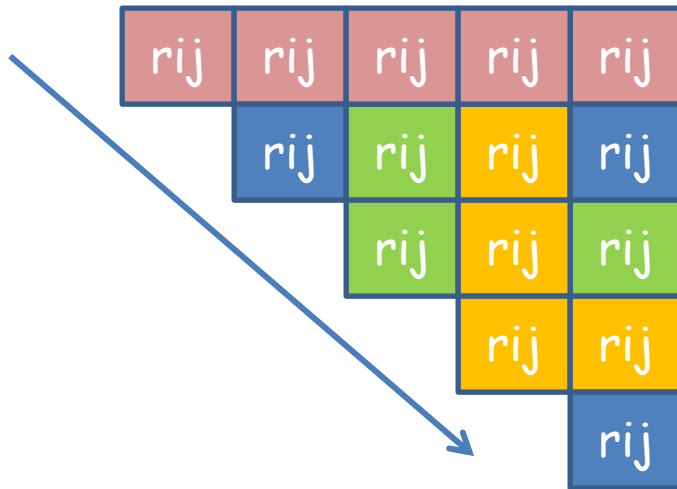
decide(v):

1. Thread  $i$  assigns "row"  $r_{ij} = v$ ,  $i \leq j$ , and "column"  $r_{ji} = v$ ,  $j < i$
2. Scan diagonal, read  $r_{ii}$ , for  $i=0, 1, \dots$ :
3. Let  $j$  be first index  $i$  where  $r_{ii} \neq t$ ;  $j=i$  is candidate for result
4. Continue scan, if  $r_{ii}=t$ ,  $j$  can still be result



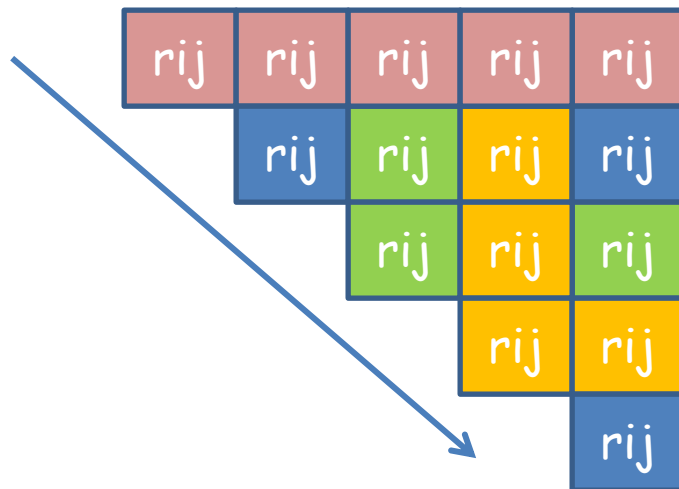
decide(v):

1. Thread  $i$  assigns "row"  $r_{ij} = v, i \leq j$ , and "column"  $r_{ji} = v, j < i$
2. Scan diagonal, read  $r_{ii}$ , for  $i=0, 1, \dots$ :
3. Let  $j$  be first index  $i$  where  $r_{ii} \neq t$ ;  $j=i$  is candidate for result
4. Continue scan, if  $r_{ii} = t$ ,  $j$  can still be result
5. If  $r_{ii} \neq r_{jj}$ , then if  $r_{ji} = r_{jj}$ , set  $j = i$ , new candidate for result



decide(v):

1. Thread  $i$  assigns "row"  $r_{ij} = v$ ,  $i \leq j$ , and "column"  $r_{ji} = v$ ,  $j < i$
2. Scan diagonal, read  $r_{ii}$ , for  $i=0, 1, \dots$ :
3. Let  $j$  be first index  $i$  where  $r_{ii} \neq \perp$ ;  $j=i$  is candidate for result
4. Continue scan, if  $r_{ii} = \perp$ ,  $j$  can still be result
5. If  $r_{ii} \neq r_{jj}$ , then if  $r_{ji} = r_{jj}$ , set  $j = i$ , new candidate for result



Thread 2 was first, its value  $r_{22}$  is returned as decided

6. Return  $r_{jj}$ : consensus value

## Simplified example: 2-consensus with a (2,3)-register

```

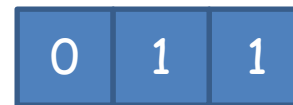
class MULTI2consensus extends Consensus {
    private final int NULL = -1;
    Assign23 value = new Assign23(NULL);
    public T decide(T v) {
        propose(v);
        int i = ThreadID.get();
        value.assign(i,i,i,i+1); // (2,3)-assignment
        int other = value.read((i+2)%3);
        if (other==NULL || other==value.read(1)) {
            return proposed[i]; // thread won
        } else return proposed[1-i]; // thread lost
    }
}

```

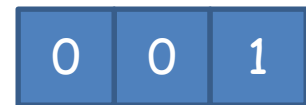
Thread 0, 3 cases:



0 alone, d=0



0 first, d=0



0 last, d=1

Corollary:

(2,3)-assignment registers have consensus number at least 2

Corollary:

It is **not** possible to construct an  $(m,n)$ -assignment register out of atomic registers

**Reminder:**

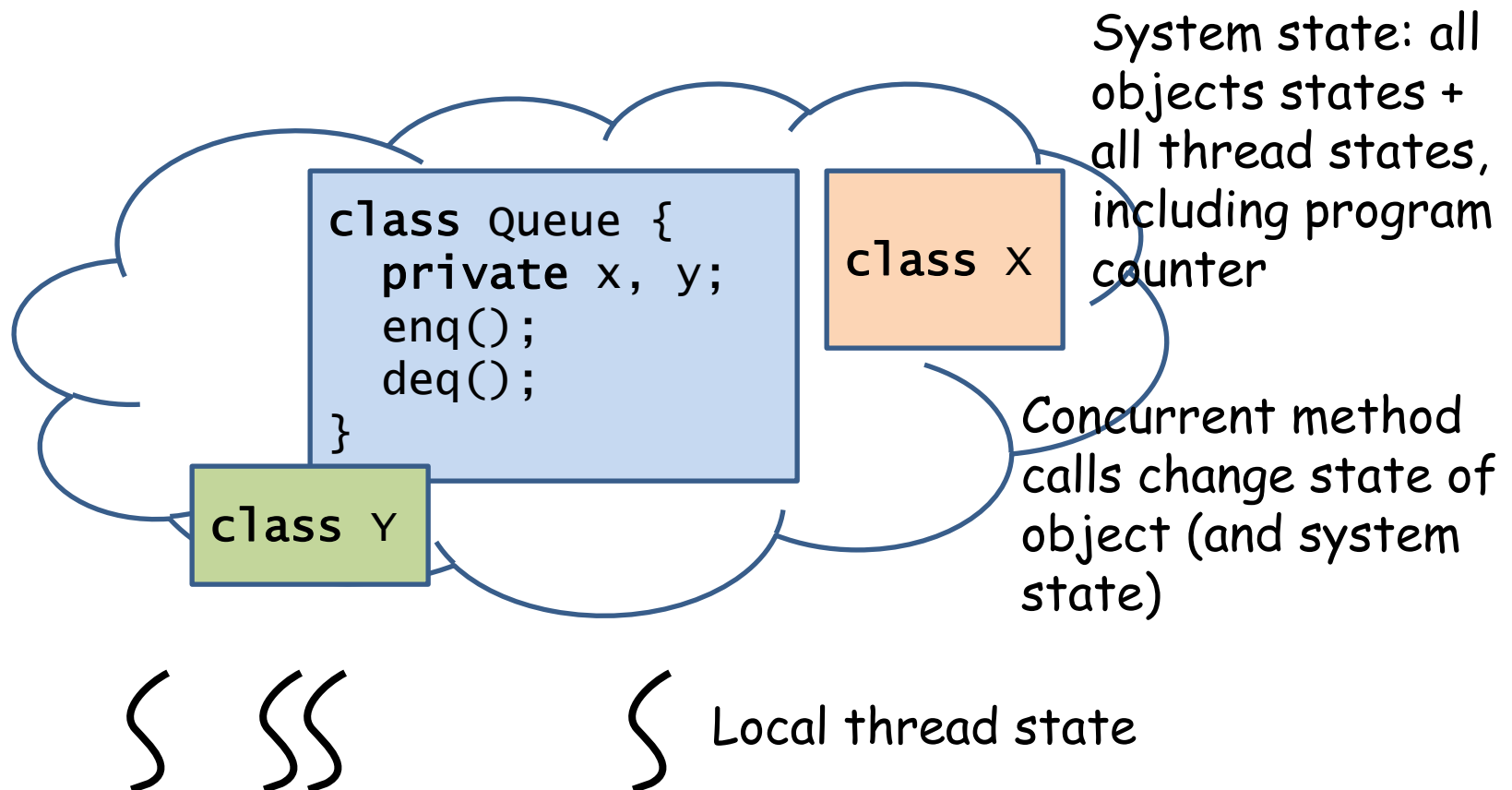
Atomic snapshot could be constructed from atomic registers; the **dual**, atomic multi-assignment **cannot**. Atomic multi-assignment is more demanding than snapshot

**Remark:**

Have  $(m,n)$ -assignment registers ever been implemented in hardware (**Exercise**: Check this)?



## Digression: Concurrent objects, correctness, consistency (Chap. 3)



## Mandatory properties of concurrent objects

**Safety** (correctness, whatever that means): "Nothing bad (violation of some requirements) can ever happen"

Counterpart of sequential correctness (e.g. of data structure like stack, queue, PQ, ...)

**Liveness**: Progress, conditions under which a thread or the execution as a whole can make progress; "something good will (eventually) happen"

What does it mean that a concurrent execution of a concurrent object is correct?

- Method calls take time (**intervals**, **not events**) and can overlap (**no total order on intervals**)
- Relate to sequential specification of object

**Example** (useful concurrent object): FIFO queue that supports (sequentially specified) **enqueue/dequeue** operations in a **wait-free manner**

- `q.enq(a)`: enqueues `a` in `q` (append to tail)
- `q.deq(b)`: if `q` nonempty, dequeues `b` from `q` (remove head), otherwise throw exception (or similar action, e.g. return `†`)
- Item dequeued is least recently item enqueued (whatever that means)

**State** of `q` described by the sequence of elements in `q`

Example:

**Lock-based** FIFO queue implementation, each queue operation is done in a **critical section**, protected by a lock

Such an implementation can be **deadlock free**, **starvation free**, may satisfy other **dependent progress criteria**, depending on the lock properties

Such an implementation is **blocking**: A delay by one thread can (**will**) cause other threads to wait.

It is **not wait-free**: It is not guaranteed that every method call will finish in a finite (bounded) number of steps

It is **not lock-free**: It is not guaranteed that some method call will always finish in a finite number of steps

```

class LockQueue<T> {
    int head, tail;
    T[] queue;
    Lock lock;
    public void enq(T x) throws Full {
        lock.lock();
        try {
            if (tail-head==queue.length) throw new Full();
            // insert at tail .....
        } finally {
            lock.unlock();
        }
    }
    public T deq() throws Empty {
        lock.lock();
        // ... if empty throw exception, else return head
        lock.unlock();
    }
}

```

Classical sequential correctness/specification:

**If** object in state  $s'$  satisfying **precondition**  $C'$ , **then** method call will result in state  $s''$  satisfying **postcondition**  $C''$

Concurrent correctness condition:

Relates **any concurrent execution** to the sequential specification (**preferably** in such a way **not** to enforce blocking/dependent progress). Such a condition is called **consistency** (Examples: sequential consistency, quiescent consistency, linearizable consistency, ...)

**Informally**: A concurrent execution is correct if it is consistent with some sequential execution

**Desirable**: If objects  $A$  and  $B$  are consistent (according to some condition), then any system using  $A$  and  $B$  together is also consistent. Such a consistency condition is called **compositional**.

Example: Sequential specification of queue class with methods `enq()` and `deq()`; represent queue state  $S = \langle x_1, x_2, x_3, \dots \rangle$  by a finite sequence of queued elements

Precondition:  $\langle Q \rangle$

`q.enq(a)`

Postcondition:  $\langle Q.a \rangle$

Precondition:  $\langle e.Q \rangle$

`v = q.deq()`

Postcondition:  $v == e, \langle Q \rangle$

Precondition:  $\langle \rangle$

`v = q.deq()`

Postcondition: EMPTY exception

**Note:** This specification is **total**, queue either returns value or throws exception (alternative: Returns special  $\dagger$  value)

## Sequential consistency:

- Method calls should appear to happen in a one-at-a-time ("atomic"), sequential order (but **not necessarily in real time**)
- **For each thread**, method calls should appear to take effect in **program order**

Example:

Program A:  
...  
q.enq(a);  
...  
x = q.deq();  
...

Program B:  
...  
q.enq(b);  
...

**Program order:** The order in which a **single thread** executes method calls.

**Note:** Method calls by different threads are unrelated by program order



## ACM Turing Award 2014

Lamport: ...

The result of a concurrent execution is as if all the operations were executed in some sequential order (interleaving), and the operations of each thread appear in the order specified by the program

Example:

Program A:

...

q.enq(a);

...

x = q.deq();

...

Program B:

...

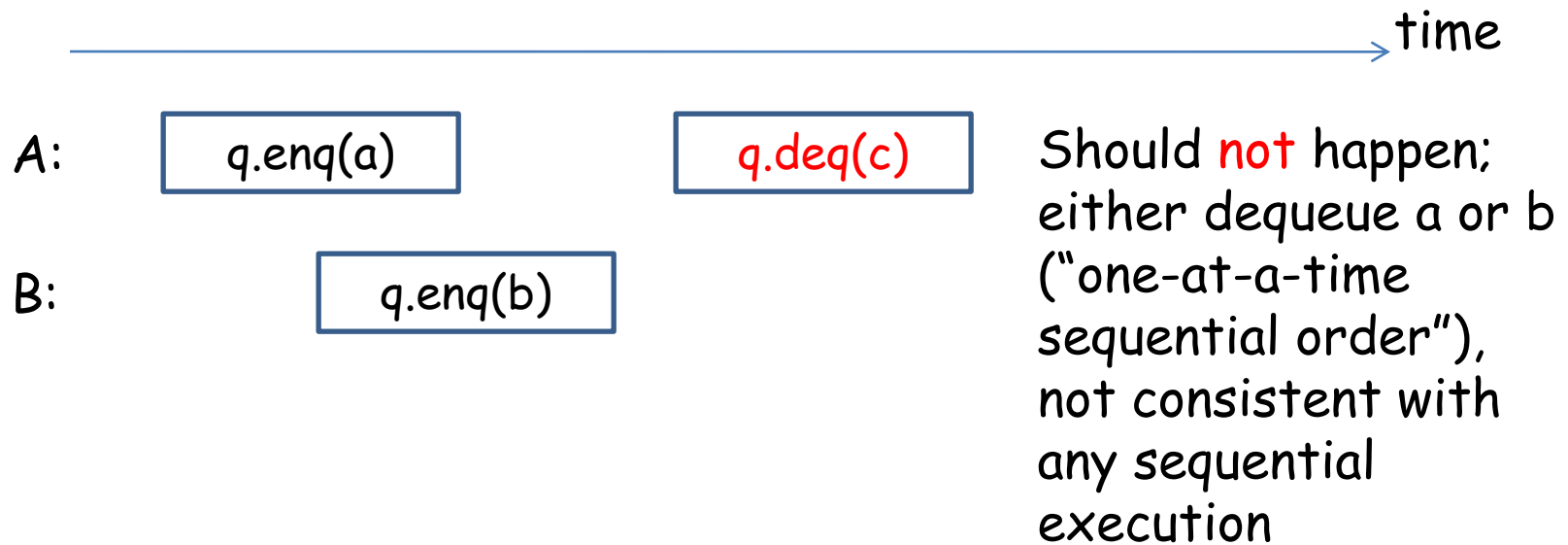
q.enq(b);

...

Leslie Lamport: How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. IEEE Trans. Computers 28(9): 690-691 (1979)

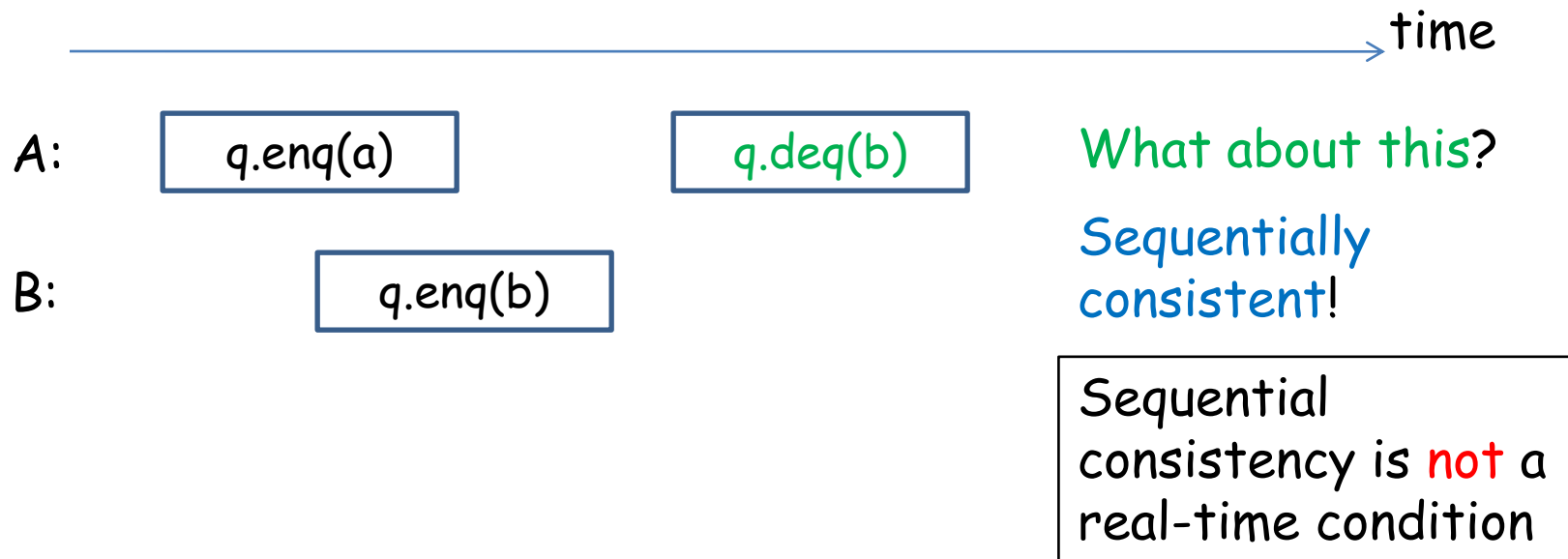
## Sequential consistency:

- Method calls should appear to happen in a one-at-a-time ("atomic"), sequential order
- For each thread, method calls should appear to take effect in **program order**



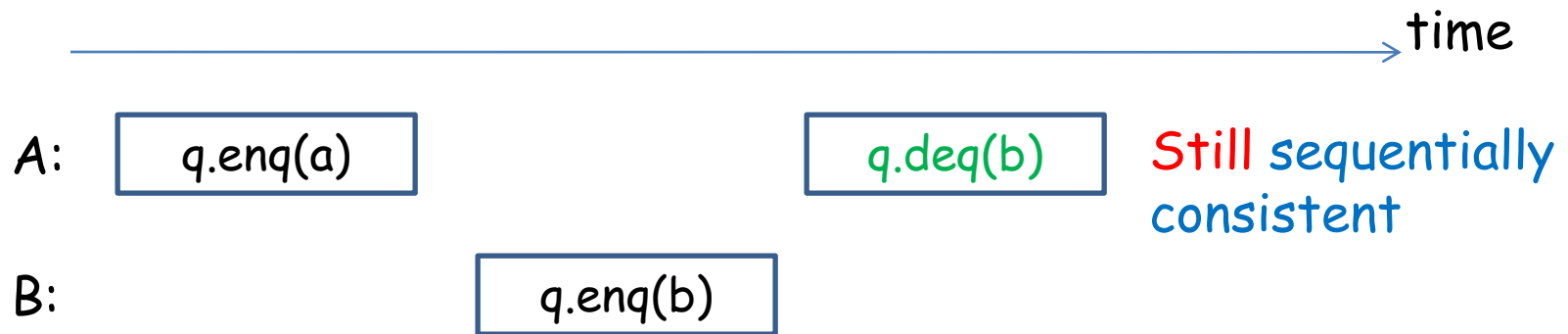
## Sequential consistency:

- Method calls should appear to happen in a one-at-a-time ("atomic"), sequential order
- For each thread, method calls should appear to take effect in **program order**



## Sequential consistency:

- Method calls should appear to happen in a one-at-a-time ("atomic"), sequential order
- For each thread, method calls should appear to take effect in **program order**

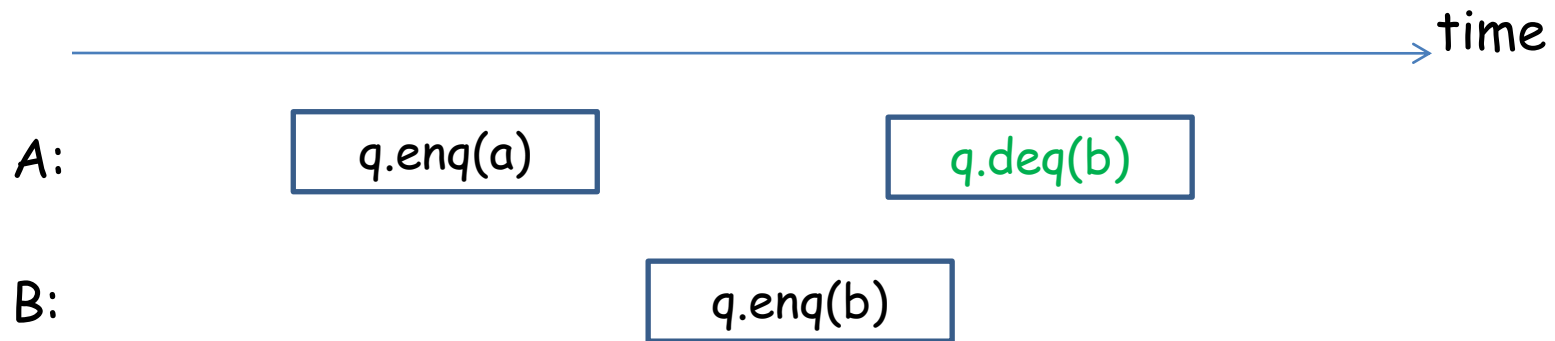


**Anticipation:**  
This history would not be linearizable

Sequential consistency is **not** a real-time condition

## Sequential consistency:

- Method calls should appear to happen in a one-at-a-time ("atomic"), sequential order
- For each thread, method calls should appear to take effect in **program order**

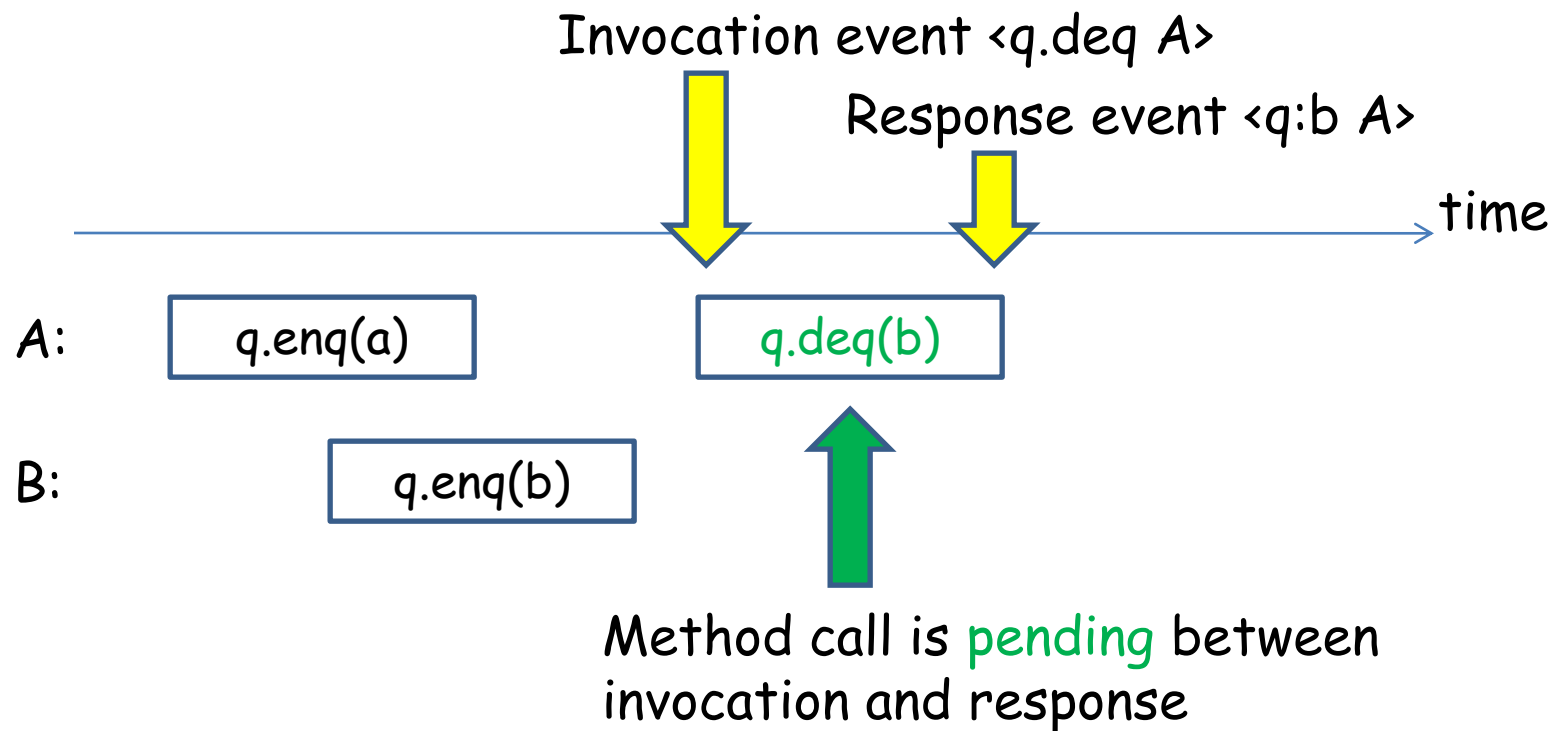


**Because:** A (real-time) sequentially correct execution can be constructed by moving the threads' method calls in time (enq(b) before enq(a))

Method calls take time...

...modeled by the interval between invocation and response...

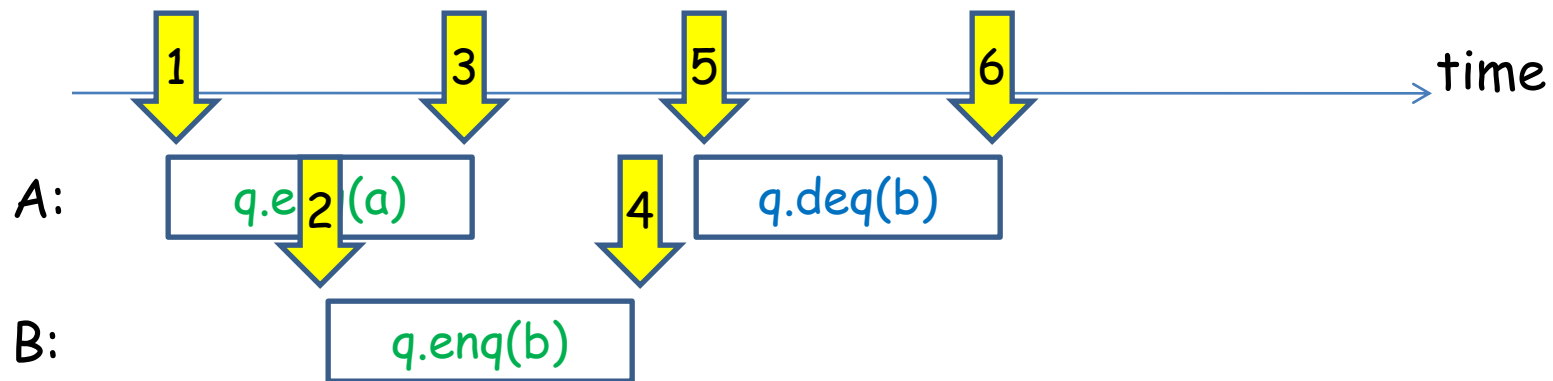
...invocations and responses are events (**totally ordered**)



## Some technicalities (part I): Histories, definitions

H: Finite sequence of (**totally ordered**) **invocation** and **response** events

H:  $\langle q.\text{enq}(a) \ A \rangle, \langle q.\text{enq}(b) \ B \rangle, \langle q:\text{ok} \ A \rangle, \langle q:\text{ok} \ B \rangle, \langle q.\text{deq} \ B \rangle, \langle q:b \ A \rangle$



- Invocation of method  $m$  on object  $x$  with arguments  $a^*$  by thread  $A$ :  $\langle x.m(a^*) \ A \rangle$
- Response by object  $x$  to thread  $B$  with values  $v^*$ :  $\langle x:v^* \ B \rangle$

**Note:**

The response to thread  $A$  to invocation of method  $m$  on object  $x$  by thread  $A$  is always to the most recent, preceding invocation of  $m$  (in other words, a thread that invokes a method waits for the response)

A response  $\langle x:v^* \ A \rangle$  **matches** an invocation  $\langle x.m(a^*) \ A \rangle$  in  $H$ : same object and same thread

A **method call** on method  $m$  of object  $x$  by thread  $A$  in  $H$  is a pair (interval) consisting of an invocation  $\langle x.m \ A \rangle$  and the next matching response event  $\langle x:v \ A \rangle$



Operations, notation:

A **subhistory**  $H'$  of some history  $H$  is a subsequence of  $H$

- $H|A$ : the (sub)history of  $H$  consisting of invocations and responses by thread  $A$  (**thread subhistory**)
- $H|x$ : the (sub)history of  $H$  consisting the invocations and responses on object  $x$  (**object subhistory**)

An invocation in  $H$  for which there is no matching response is **pending**; **complete( $H$ )** is the (sub)history where all pending invocations have been removed.

An **extension** of a(n incomplete) history  $H$  is a history  $H'$  constructed by possibly appending matching responses to pending invocations

Definition:

Histories  $H$  and  $G$  are **equivalent** if  $H|A = G|A$  for all threads  $A$   
(**Exercise**: Prove that this is an equivalence relation)

Definition:

A history  $H = e_1, e_2, e_3, \dots, e_n$  is **sequential** if  $e_1$  is an invocation, and each invocation  $e_i$  is immediately followed by a matching response  $e_{i+1}$

**Note**: A sequential history is a history with no overlapping method calls

A history is **well-formed** if each thread subhistory is sequential (no overlapping method calls by a single thread). Only well-formed histories considered here

**Note**: Object subhistories  $H|x$  of a well-formed history need not be sequential (there can be concurrent calls to  $x$ )

Definition:

A (sequential) specification of an object is the set of those sequential object histories that are correct; each such object history is **legal**

A sequential history  $S$  is **legal** if all object subhistories  $H|x$  are legal

End of technicalities (part I)

## Sequential consistency:

- Method calls should appear to happen in a one-at-a-time ("atomic"), sequential order
- For each thread, method calls should appear to take effect in **program order**

Means:

In any concurrent execution, there is a way to adjust (**slide**) the method calls in time (**no swapping**) such that

- they (for each thread) are consistent with **program order**
- meet the objects **sequential (correctness) specification**

More formally:

Any execution history  $H$  is equivalent to some correct sequential **execution history  $S$**  that satisfies program order for each thread

## Sequential consistency:

- Method calls should appear to happen in a one-at-a-time ("atomic"), sequential order
- For each thread, method calls should appear to take effect in **program order**

### Example:

#### Program A:

```
...  
q.enq(a);  
...  
x = q.deq();  
...
```

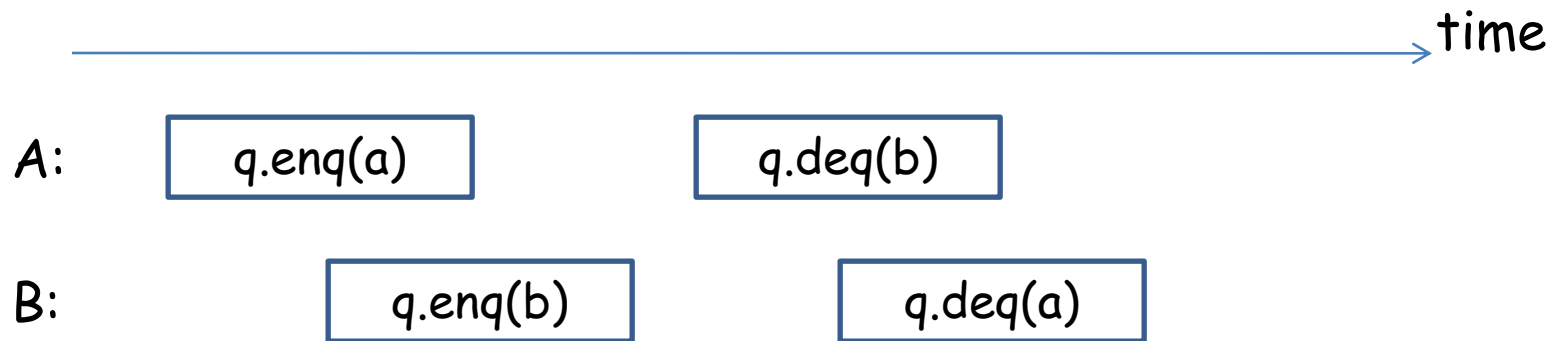
#### Program B:

```
...  
q.enq(b);  
...  
y = q.deq();  
...
```

Possible under sequential consistency to have  $x=b$  and  $y=a$ ?

## Sequential consistency:

- Method calls should appear to happen in a one-at-a-time ("atomic"), sequential order
- For each thread, method calls should appear to take effect in **program order**

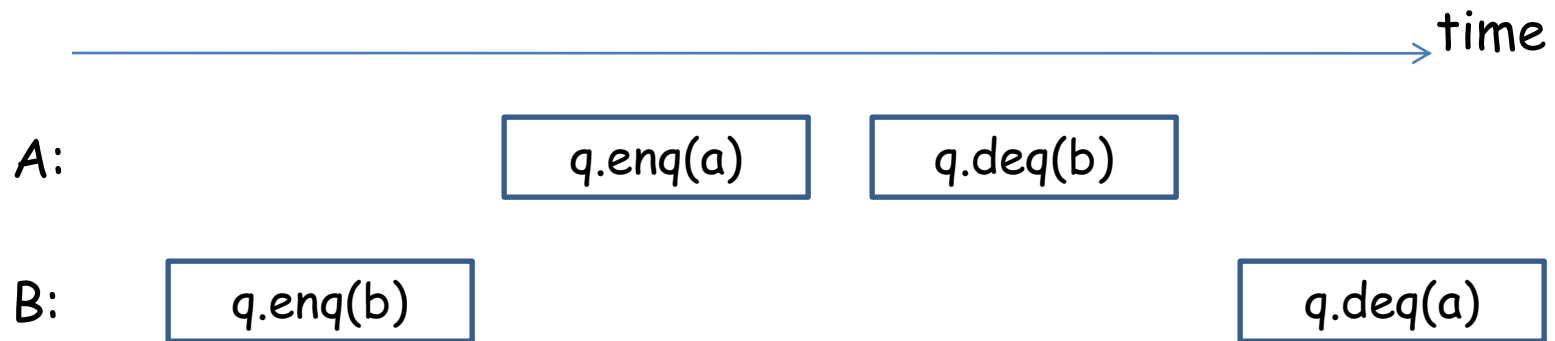


... is sequentially consistent:

Two possible executions that fulfill the conditions

## Sequential consistency:

- Method calls should appear to happen in a one-at-a-time ("atomic"), sequential order
- For each thread, method calls should appear to take effect in **program order**

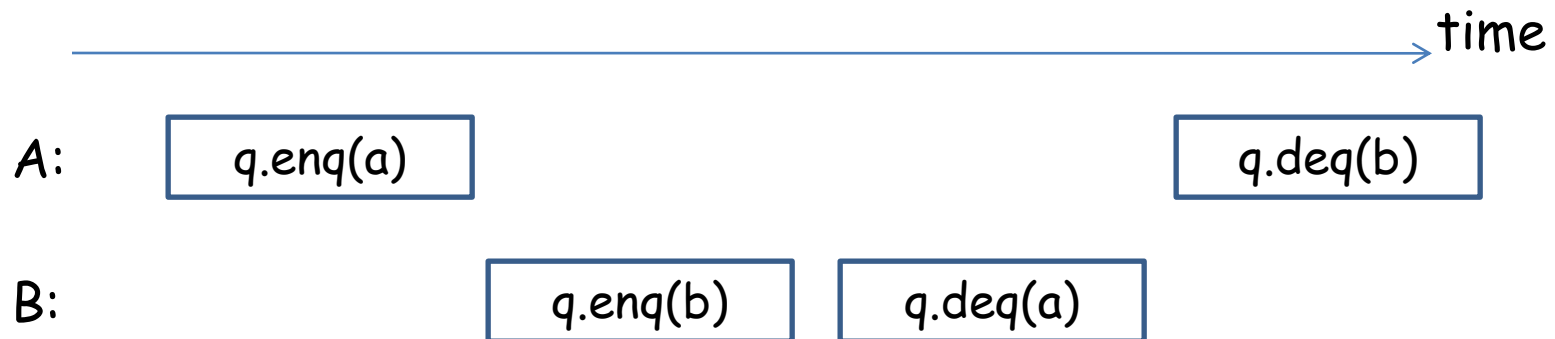


### Sequentially consistent execution 1:

$\langle q.enq(b) \ B \rangle \rightarrow \langle q.enq(a) \ A \rangle \rightarrow \langle q.deq(b) \ A \rangle \rightarrow \langle a.deq(a) \ B \rangle$

## Sequential consistency:

- Method calls should appear to happen in a one-at-a-time ("atomic"), sequential order
- For each thread, method calls should appear to take effect in **program order**



## Sequentially consistent execution 2:

$\langle q.enq(a) A \rangle \rightarrow \langle q.enq(b) B \rangle \rightarrow \langle q.deq(b) B \rangle \rightarrow \langle q.deq(b) A \rangle$



Key observation: Sequential consistency is **not compositional**

**Example**: Two queues p and q in two threads

Program A:

```
...  
p.enq(a);  
q.enq(a);  
...  
x = p.deq();  
...
```

Program B:

```
...  
q.enq(b);  
p.enq(b);  
...  
y = q.deq();  
...
```

Possible that  $x=b$  and  $y=a$ ? - as could happen with p or q in isolation?

Key observation: Sequential consistency is **not compositional**

Definition: A correctness property/consistency condition  $P$  is **compositional** if, whenever each object (and method) of the system satisfies  $P$ , then the system as a whole satisfies  $P$ .

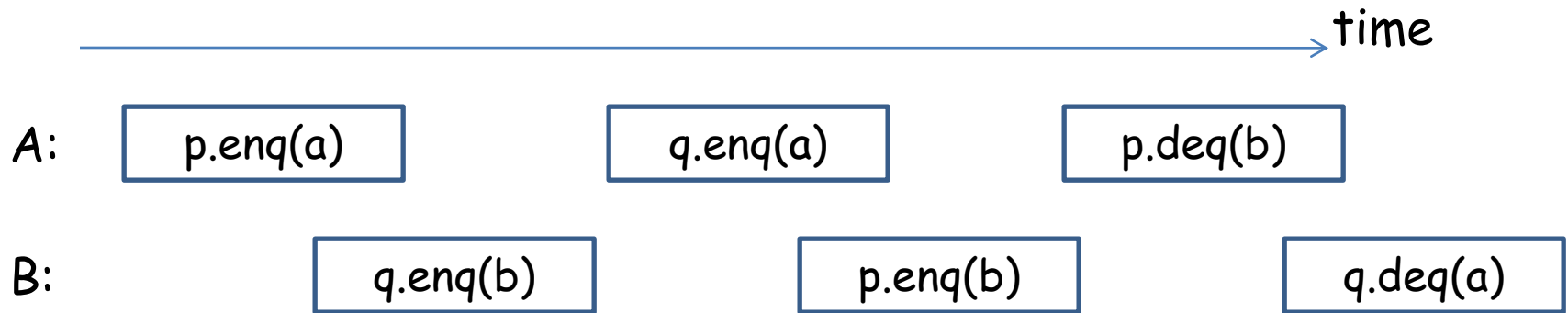
If all object subhistories of some execution history satisfy  $P$  (e.g. sequential consistency) then the execution history as a whole should satisfy  $P$

Compositionality is desirable:

Different data structures can be developed independently,  
different objects verified independently

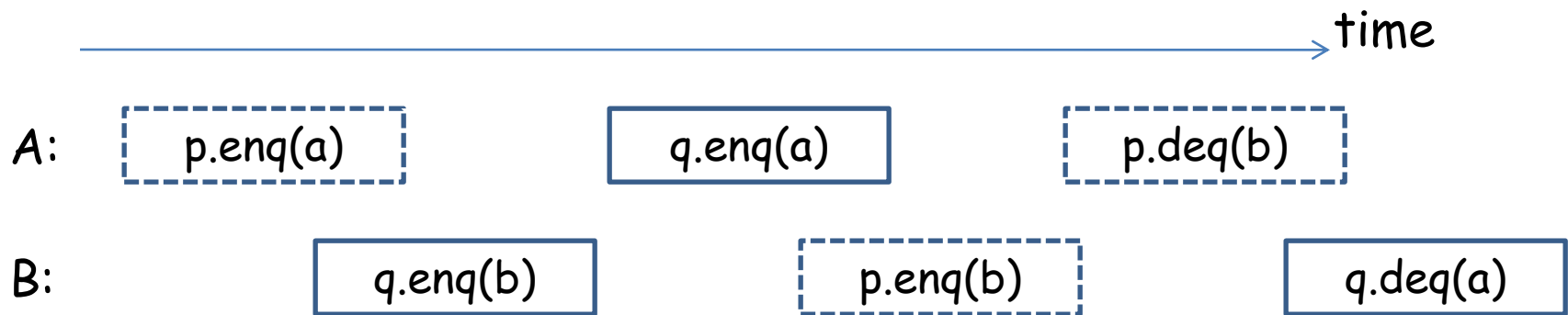
Unfortunately, sequential consistency is **not** compositional (see  
why on next slides)

Key observation: Sequential consistency is **not compositional**



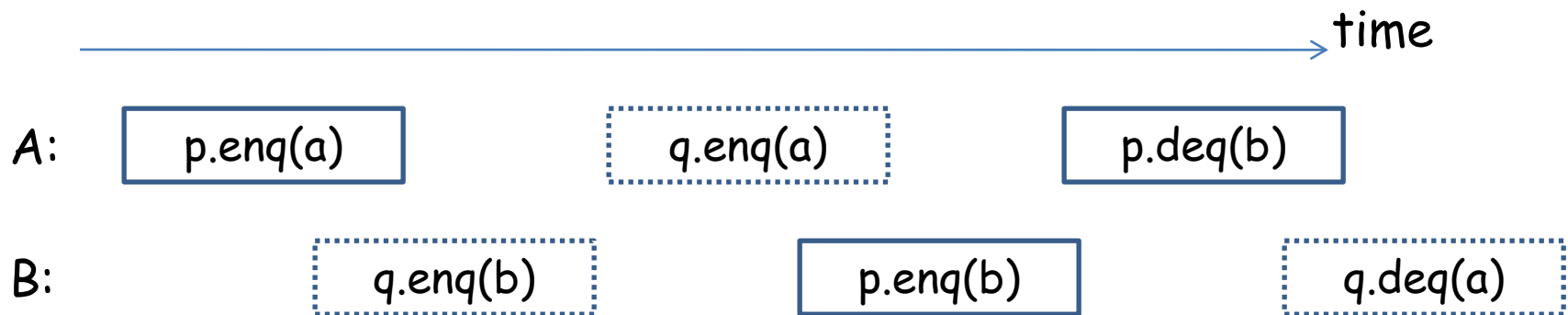
Two queues p and q; both sequences on p and q can be put in sequentially consistent order

Key observation: Sequential consistency is **not compositional**



Two queues p and q; both sequences on p and q can be put in sequentially consistent order

Key observation: Sequential consistency is **not compositional**

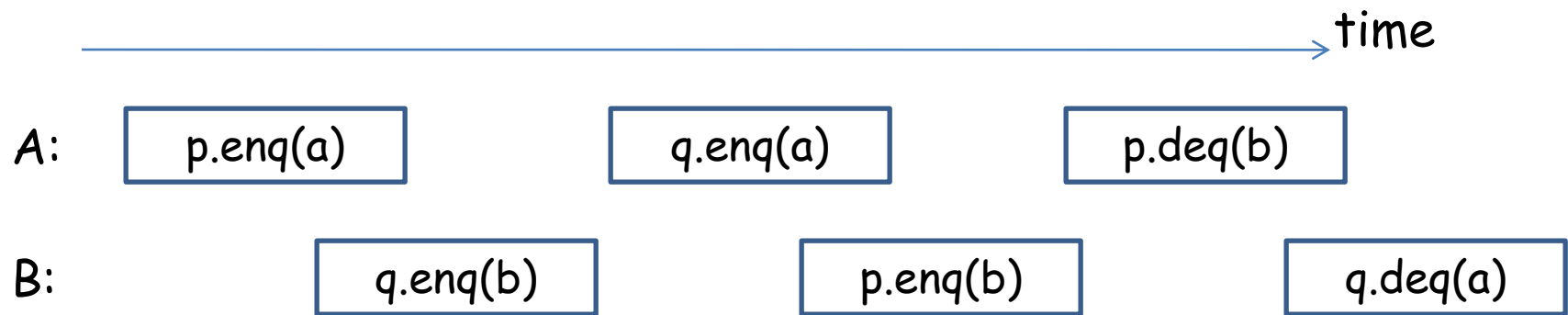


Two queues p and q; both sequences on p and q can be put in sequentially consistent order

More formally: Both p and q object subhistories sequentially consistent

**BUT...**

Key observation: Sequential consistency is **not compositional**

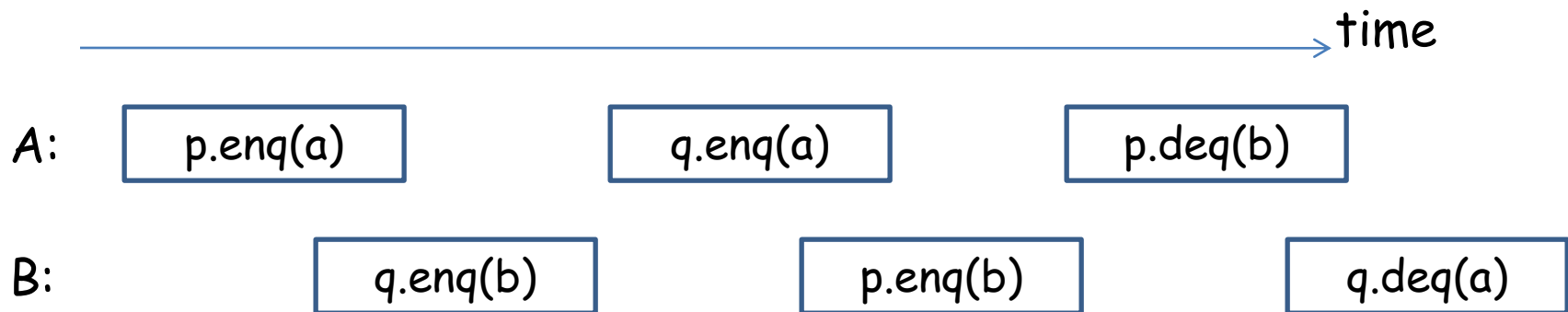


Queues are FIFO, so

$\langle p.enq(b) \ B \rangle \rightarrow \langle p.enq(a) \ A \rangle$  (because of  $\langle p.deq(b) \ A \rangle$  and  $\langle q.enq(a) \ A \rangle \rightarrow \langle q.enq(b) \ B \rangle$ )

**Note:** Here  $\rightarrow$  means "has completed before"

Key observation: Sequential consistency is **not compositional**



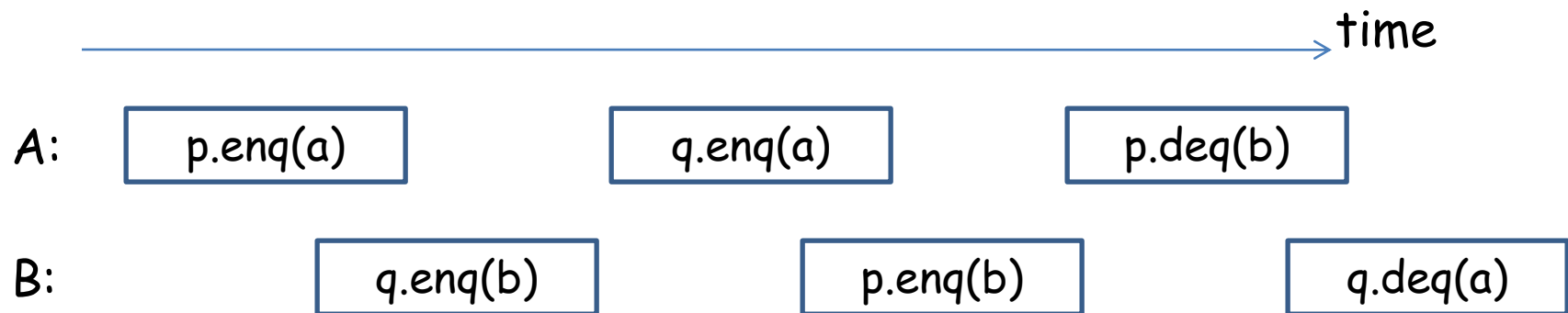
Now  $\langle p.enq(b) \ B \rangle \rightarrow \langle p.enq(a) \ A \rangle$  and  $\langle q.enq(a) \ A \rangle \rightarrow \langle q.enq(b) \ B \rangle$

Program order states

$\langle p.enq(a) \ A \rangle \rightarrow \langle q.enq(a) \ A \rangle$  and  $\langle q.enq(b) \ B \rangle \rightarrow \langle p.enq(b) \ B \rangle$



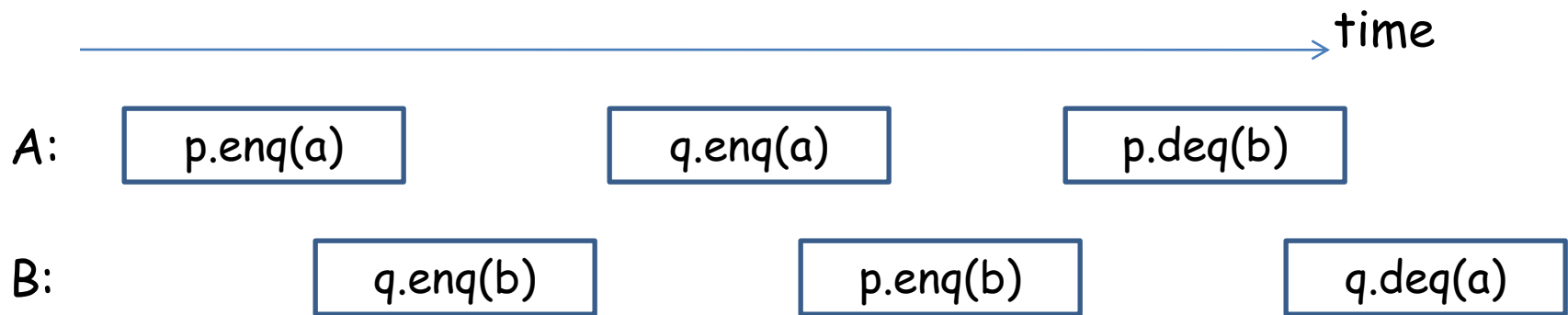
Key observation: Sequential consistency is **not compositional**



$\langle p.enq(b) \ B \rangle \rightarrow \langle p.enq(a) \ A \rangle$  and  $\langle q.enq(a) \ A \rangle \rightarrow \langle q.enq(b) \ B \rangle$   
 $\langle p.enq(a) \ A \rangle \rightarrow \langle q.enq(a) \ A \rangle$  and  $\langle q.enq(b) \ B \rangle \rightarrow \langle p.enq(b) \ B \rangle$

implies  $\langle p.enq(b) \ B \rangle \rightarrow \langle q.enq(b) \ B \rangle$  **contradicting** program order for B (likewise for A)

Key observation: Sequential consistency is **not compositional**



**Although** both p and q object subhistories of the example execution history were sequentially consistent, the history as a whole **violated program order**.

This means: **Sequential consistency is not compositional**

## Quiescent consistency:

- Method calls should appear to happen in a one-at-a-time ("atomic"), sequential order
- Method calls separated by a period of quiescence (no activity, no overlapping method calls,...) should appear to take effect in **real-time order**

Quiescent consistency relaxes the strict adherence to program order (for overlapping method calls) **AND** introduces a real-time component

Example:

A and B enqueue x and y, the queue object becomes quiescent, C enqueues z: The queue will hold x and y in some order, followed by z

**Difficulty:** How long should the period of quiescence be?

Theorem (without proof here): Quiescent consistency is compositional.

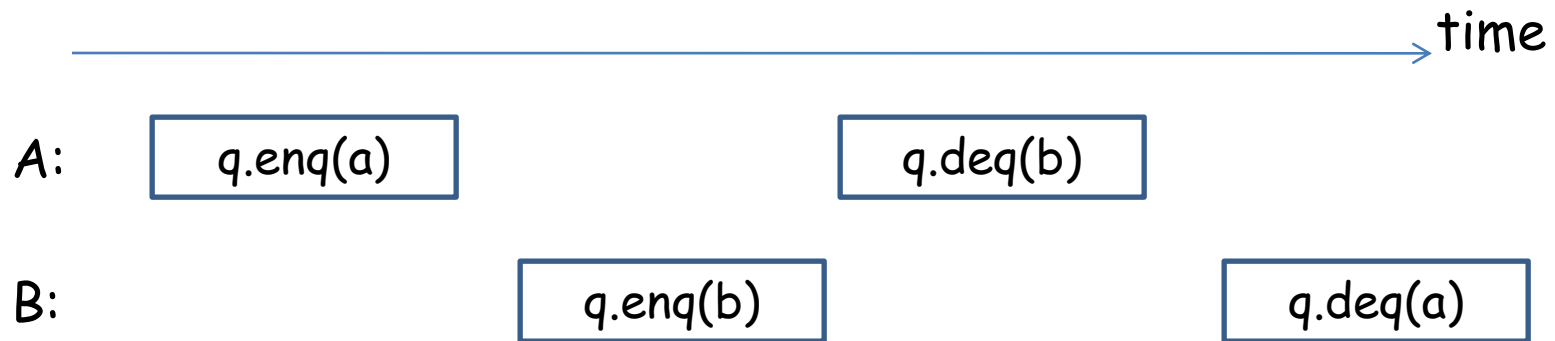
## Exercises

**Note:**

Quiescent and sequential consistency are incomparable: There are executions that are sequentially consistent but not quiescently consistent, and vice versa

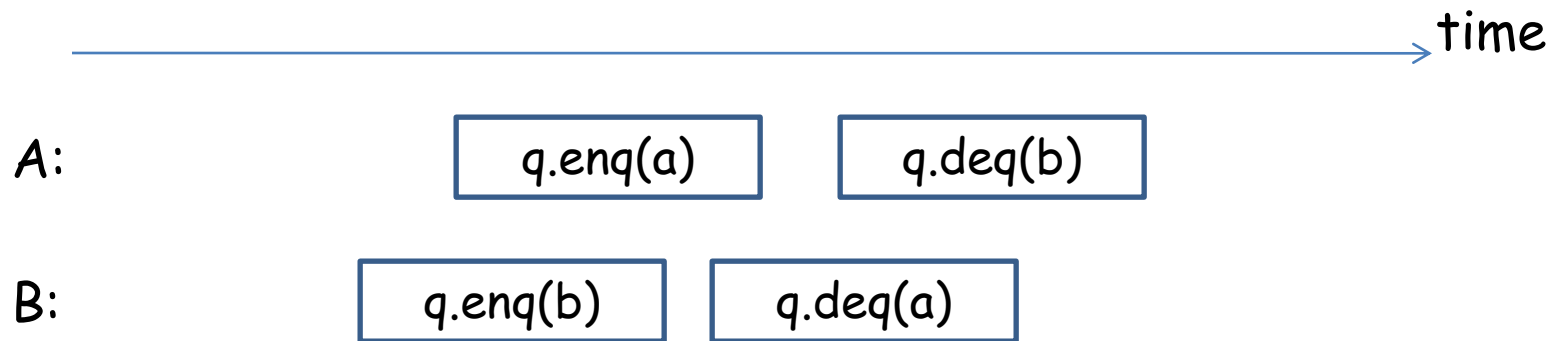
## Problem with sequential consistency:

- **Intuitive outcomes of programs**, attractive for reasoning about memory (too restrictive for real hardware): Updates by any thread become visible to other threads in program order, but delays are allowed
- **Does not respect real-time order**

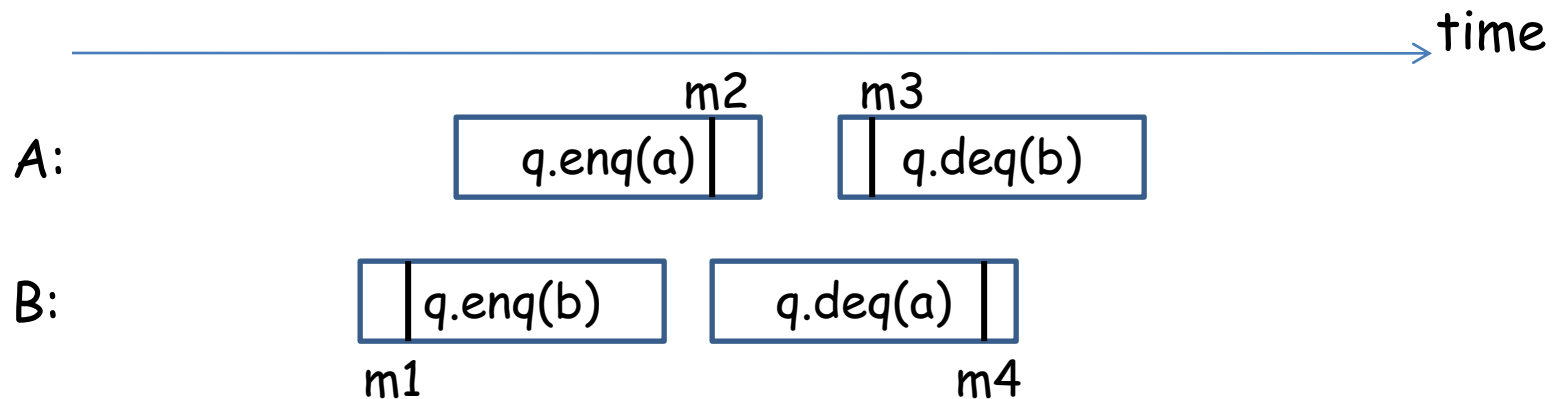


is **sequentially consistent**, but **not consistent** with the way the method calls happened in time

- Sequentially consistent, **AND**
- Possible to find points in time where method calls take effect such that the execution is sequentially correct and respects the real-time order in which the method calls happened



- Sequentially consistent, **AND**
- Possible to find points in time where method calls take effect such that the execution is sequentially correct and respects the real-time order in which the method calls happened



## Linearizability (linearizable consistency, atomicity):

- Method calls should appear to happen in a one-at-a-time ("atomic"), sequential order
- Each method call should appear to take effect in **real-time order**, **instantaneously at some point between invocation and response event**

## Terminology:

The points where method calls take effect are called **linearization points**



## Proving linearizability

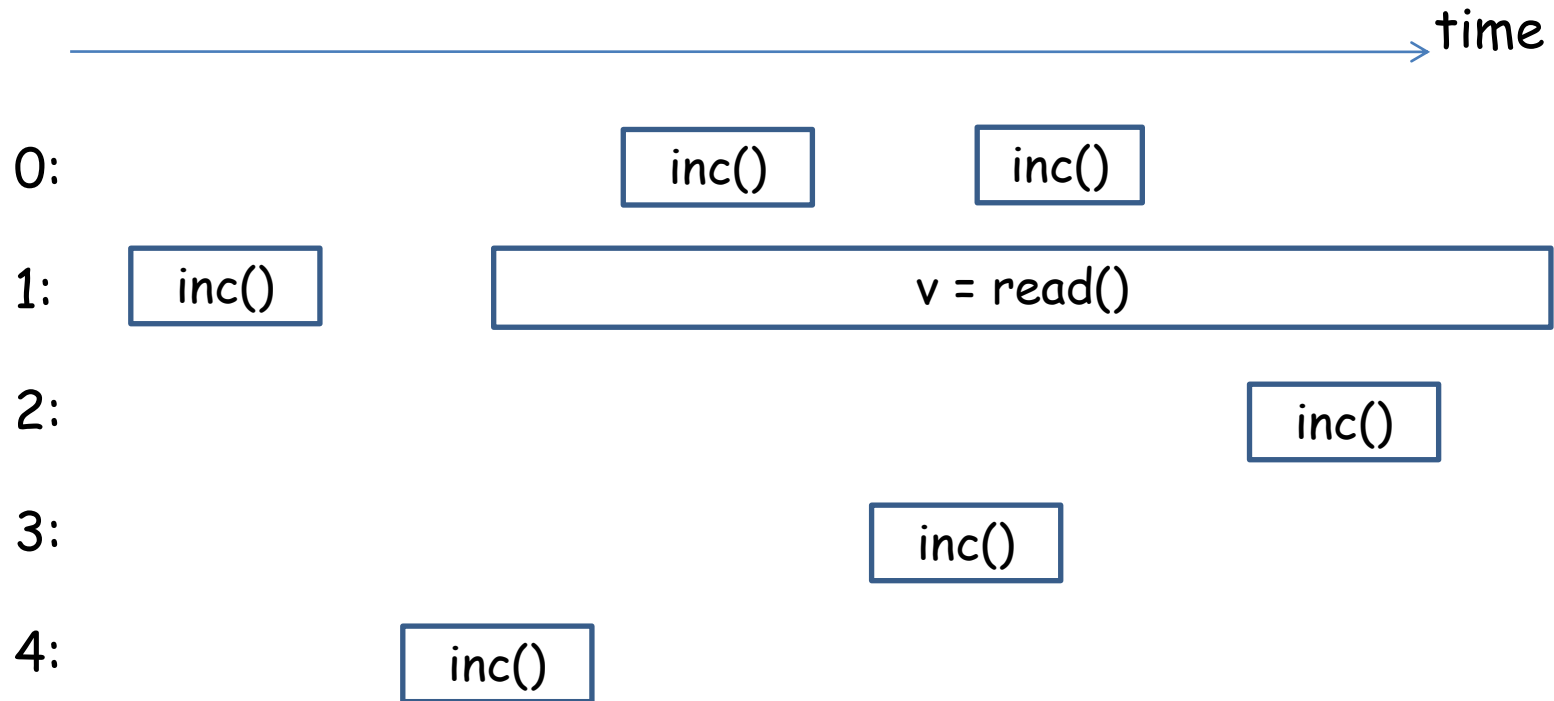
- Of some given history: Identify possible linearization points in the history, such that the linearization order is a correct sequential history. For any given history, there may be many different linearizations (see **exercises and later**)
- Of an object/algorithm: Show that any concurrent history that can be generated by the object is linearizable. Linearization points often correspond to the execution of some atomic operation (**see later**). Different executions may generate different linearizations. **But**: Linearization point may not always be the same instruction, and may depend on the actual history

Example: Wait-free, linearizable ("atomic") counter

```
class AtomicCounter {  
    int counter[N]; // N number of threads. Initially 0  
    public void inc() {  
        counter[ThreadID.get()]++; // non-atomic inc  
    }  
    public int read() {  
        int value = 0;  
        int i;  
        for (i=0; i<N; i++) value += counter[i];  
        return value;  
    }  
}
```

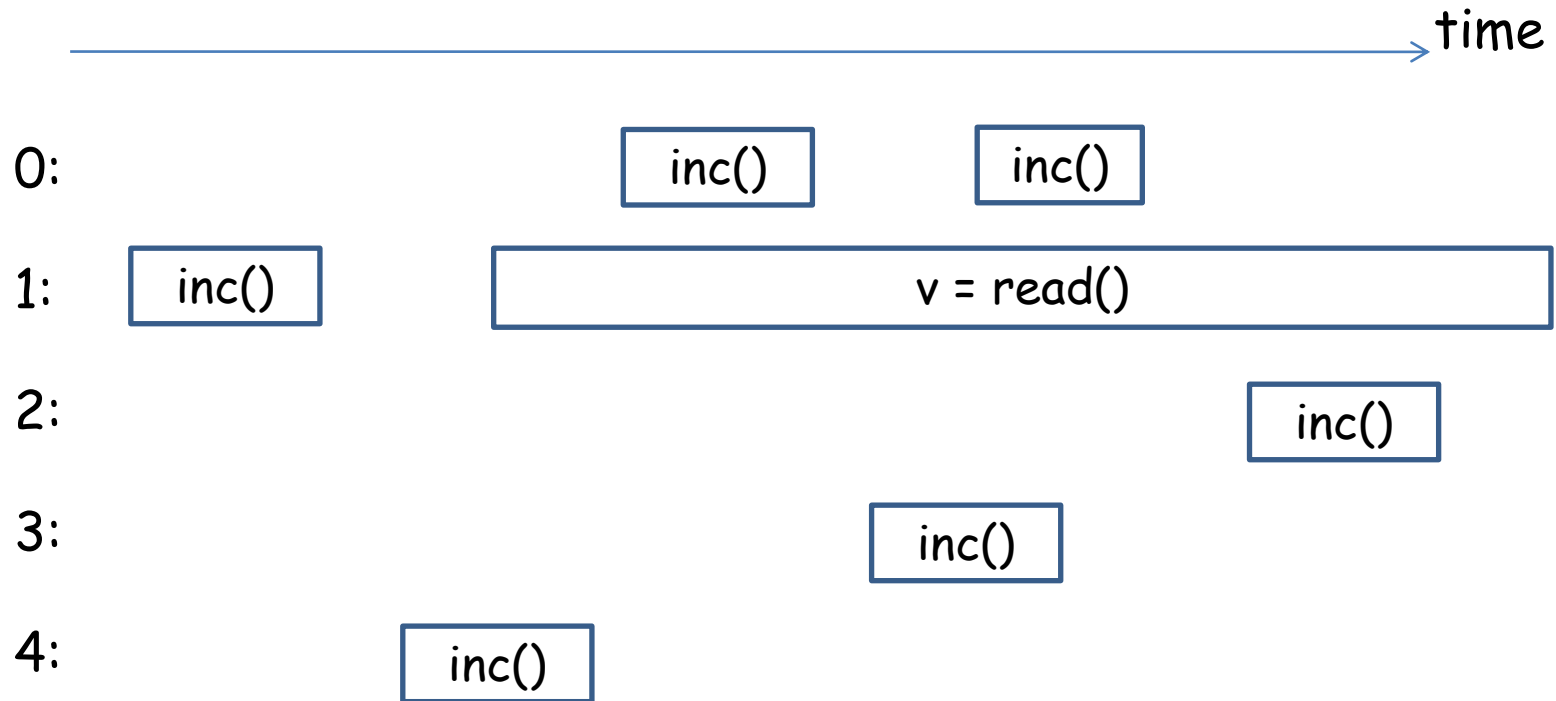
Uses only registers

**Idea:** Counter represented as an array of "per thread counts"

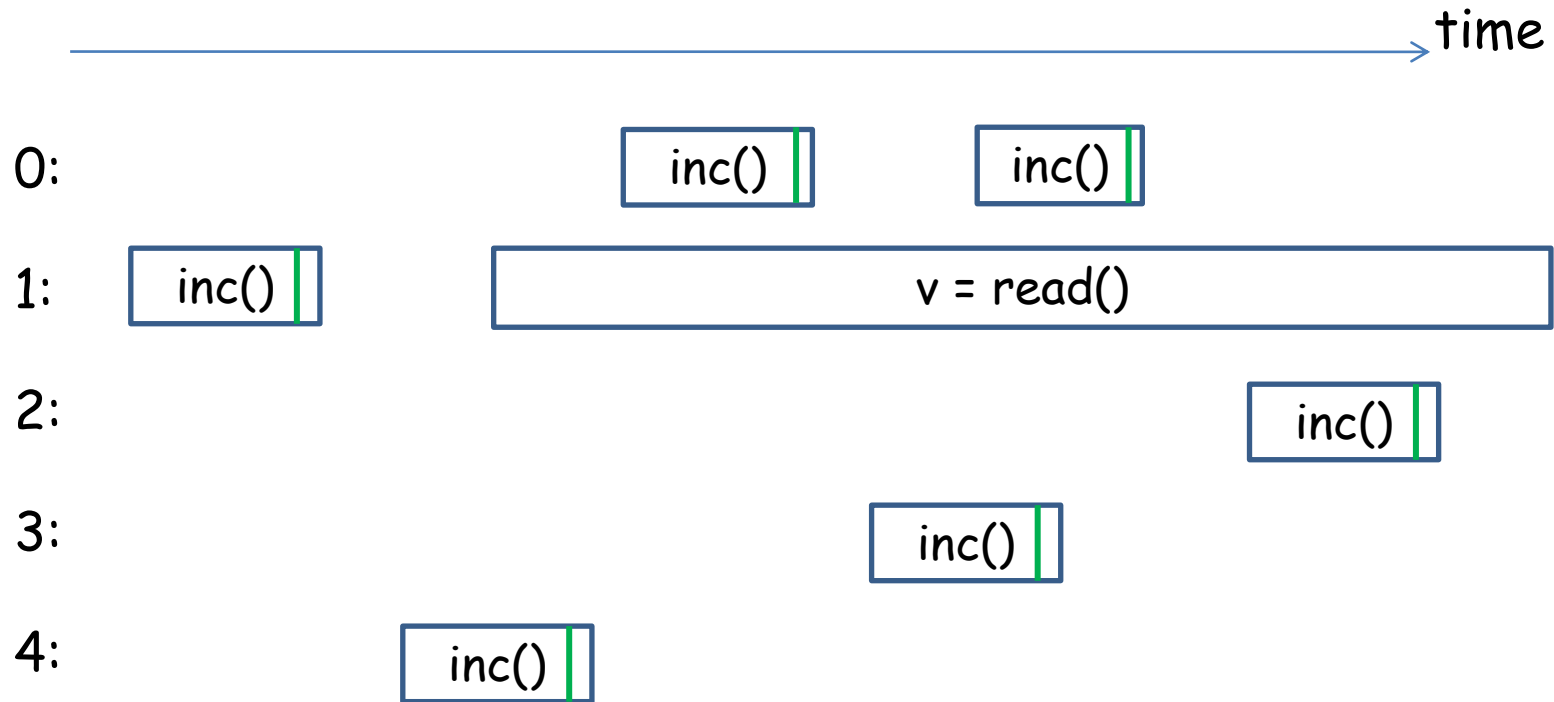


Claim: Counter history is linearizable (behaves atomically).

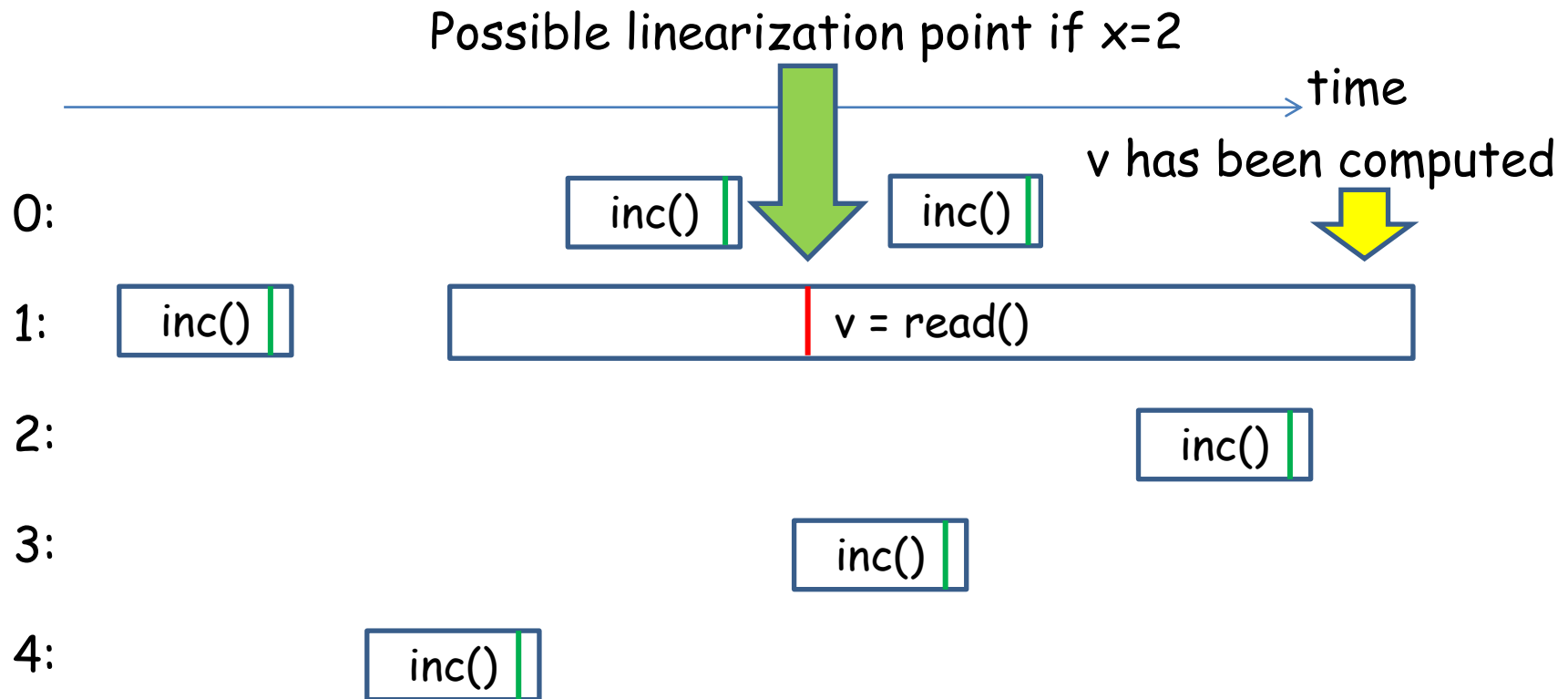
Question: When does `read()` linearize?



Lemma: Both `inc()` and `read()` operations are wait-free

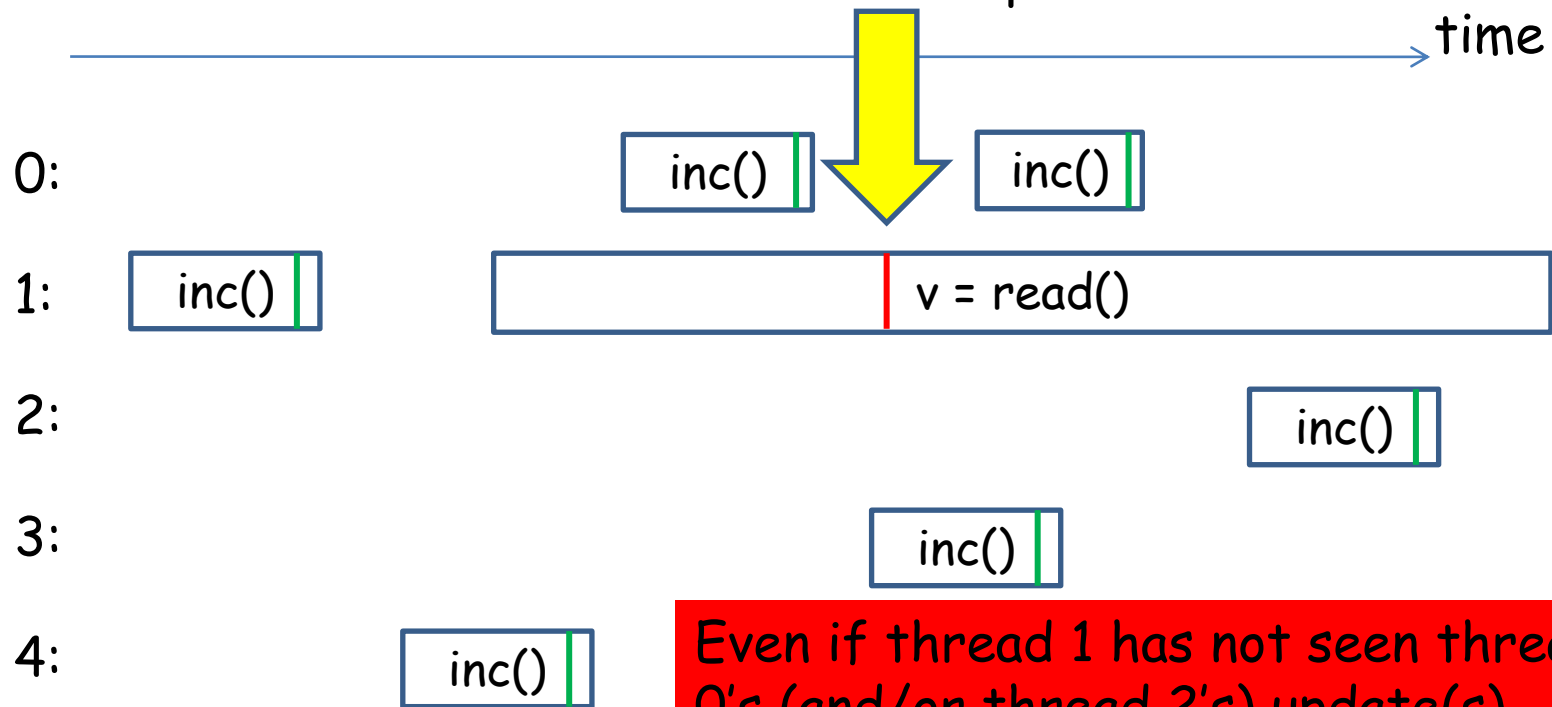


Lemma: The `inc()` operation for thread  $i$  linearizes at the point where `counter[i]` is updated



Lemma: Let  $k$  be the value of the counter before the read operation starts (consider all `inc()`  $\rightarrow$  `read()`); let  $v = k+x$  be the value returned by `read()`. Then `read()` linearizes between the  $x$ 'th and the  $(x+1)$ 'th overlapping `inc()` operation

Possible linearization point if  $x=2$

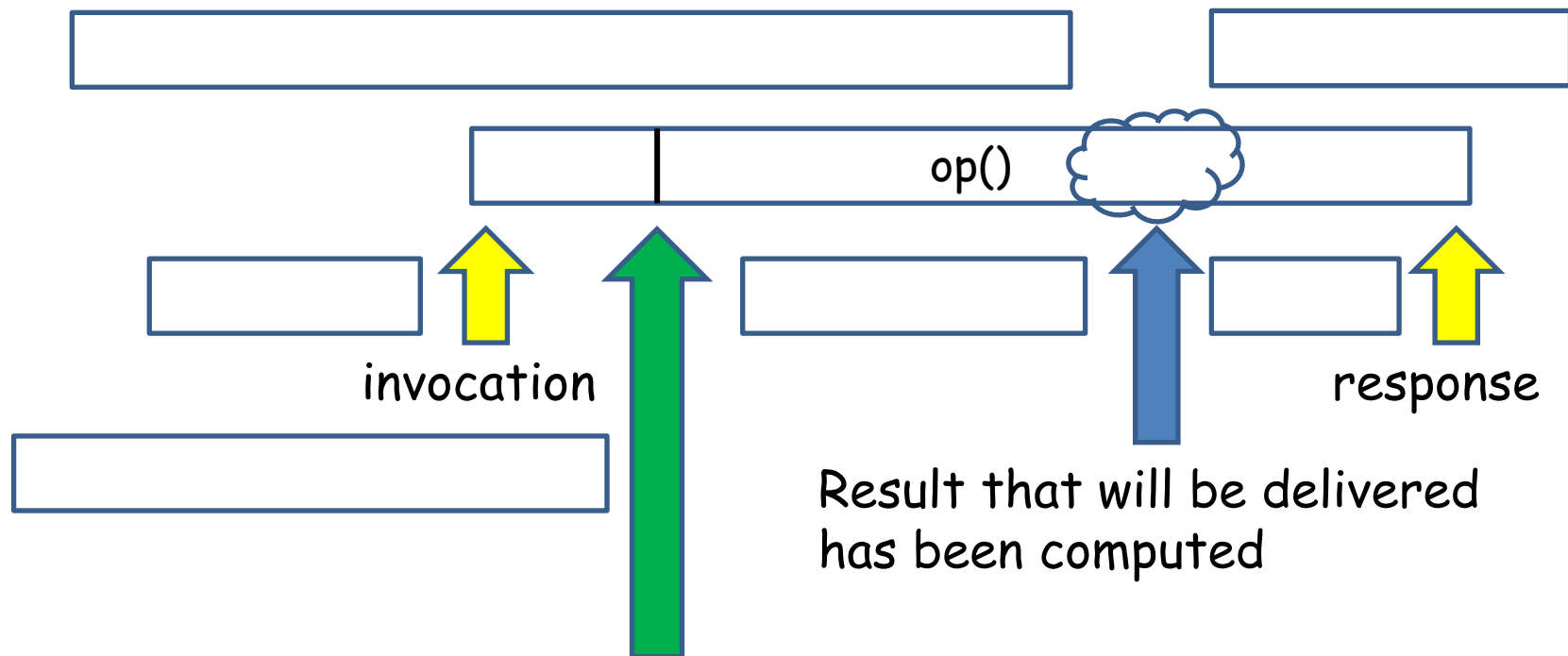


Even if thread 1 has not seen thread 0's (and/or thread 2's) update(s)

Lemma: Let  $k$  be the value of the counter before the read operation starts (consider all `inc()`  $\rightarrow$  `read()`); let  $v = k+x$  be the value returned by `read()`. Then `read()` linearizes between the  $x$ 'th and the  $(x+1)$ 'th overlapping `inc()` operation

Recall: Increment (`++`) is commutative

## Proving linearizability of given data structure/algorithm



Linearization point: Some (virtual) point during the method call where the delivered result would have been correct. A proof argues that this point can always be found; but it may depend on other, concurrent operations



### Linearizability:

- Method calls should appear to happen in a one-at-a-time ("atomic"), sequential order
- Each method call should appear to take effect in real-time order, instantaneously at some point between invocation and response event

### Theorem:

Linearizability is compositional

### Theorem:

Linearizability is a **non-blocking property**. A pending method call never has to wait for another pending method call to respond

### Corollary:

Every linearizable execution is sequentially consistent (not vice versa)

## Technicalities (part II): Formal definition and proofs

Method call in  $H$ : Pair of matching invocation and response events.

$m = \dots \langle x.m \ A \rangle, \dots, \langle x:v \ A \rangle, \dots$

Method call  $m_0$  is before method call  $m_1$ , written  $m_0 \rightarrow m_1$ , in  $H$  if  $H = \dots \langle x.m_0 \ A \rangle \dots \langle x:v \ A \rangle \dots \langle x'.m_1 \ A' \rangle \dots$  (response event of  $m_0$  before invocation event of  $m_1$ )

Since the interval of method call  $m_0$  may overlap the interval of the method call  $m_1$ ,  $\rightarrow$  is a **partial order**

**Important:** For a **sequential history**  $S$ , the relation  $\rightarrow$  is total. Since there are no overlapping method calls in  $S$ , either  $m_0 \rightarrow m_1$  or  $m_1 \rightarrow m_0$  for any method calls  $m_0, m_1$  in  $S$

Definition:

History  $H$  is linearizable if there is an **extension**  $H'$  and a **legal sequential history**  $S$  such that

1.  $\text{complete}(H')$  is equivalent to  $S$
  2. if  $m_0 \rightarrow m_1$  in  $H$  for method calls  $m_0, m_1$ , then  $m_0 \rightarrow m_1$  in  $S$ .
- $S$  is called a **linearization** of  $H$

- A method call that precedes another in  $H$  ("real time") must also do so in  $S$ .  $S$  totally orders (linearizes) the method calls in  $H$ , and  $S$  is a legal sequential history.
- It is allowed to add responses to some pending invocations in  $H$  ("these have already taken effect") in order to obtain the linearization  $S$

Theorem: History  $H$  is linearizable iff for each object  $x$ ,  $H|x$  is linearizable

Thus: Linearization is compositional

Proof (**only if**):

If  $H$  is linearizable to  $S$ , then  $S|x$  is a linearization of  $H|x$  for each object  $x$

Theorem: History  $H$  is linearizable iff for each object  $x$ ,  $H|x$  is linearizable

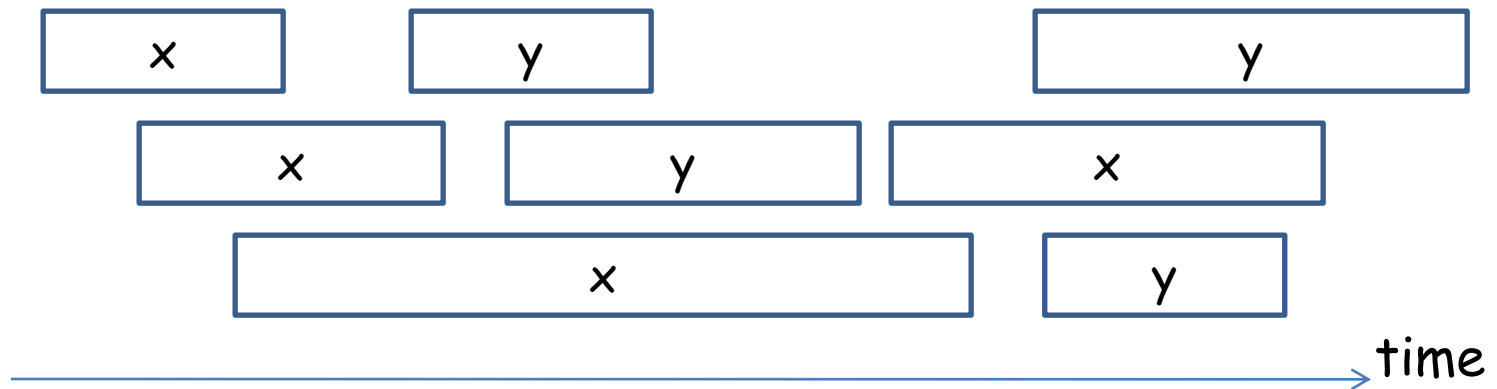
Proof (if):

**Induction** on the number of method calls in  $H$ . For each object  $x$  in  $H$ , pick a linearization  $S_x$  of  $H|x$ . Each  $S_x$  may include some responses not in  $H$ . Let  $R_x$  be all such responses, and  $H' = HR_x$ .

**Induction base**: If there is only one method call in  $H'$ ,  $H'$  is itself a linearization.

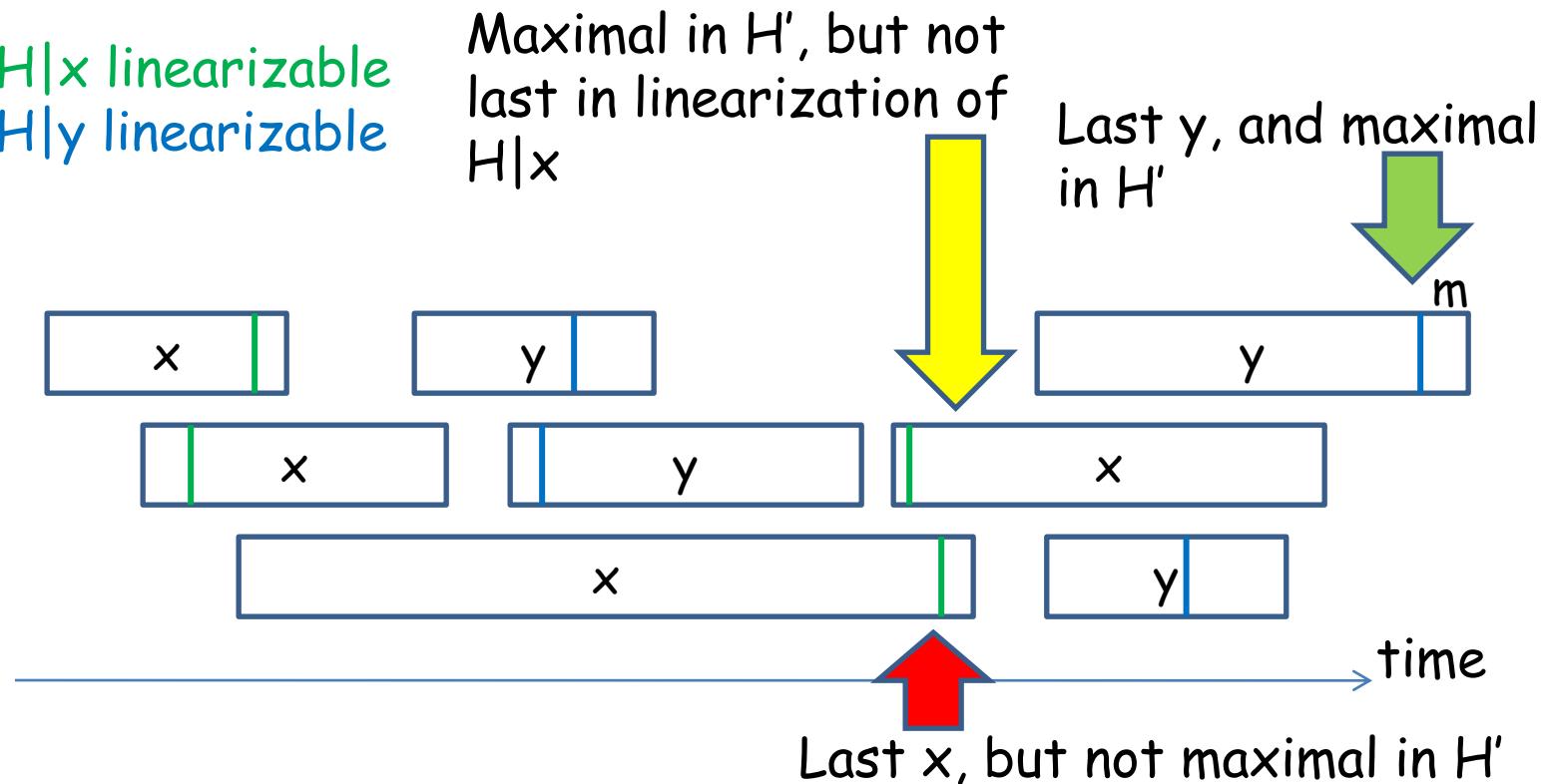
**Induction step**: Assume claim for fewer than  $k$  method calls. For each  $x$  consider last method call in  $S_x$ . One of these  $m$  must be maximal in  $H'$ , that is there is no  $m'$  with  $m \rightarrow m'$ . Let  $G'$  be  $H'$  with  $m$  removed. Now,  $H'$  is **equivalent** to  $G'm$  (because  $m$  was last and maximal). By induction hypothesis  $G'$  is linearizable to some  $S'$ , and both  $H$  and  $H'$  are linearizable to  $S'm$ .

Proof (illustration):



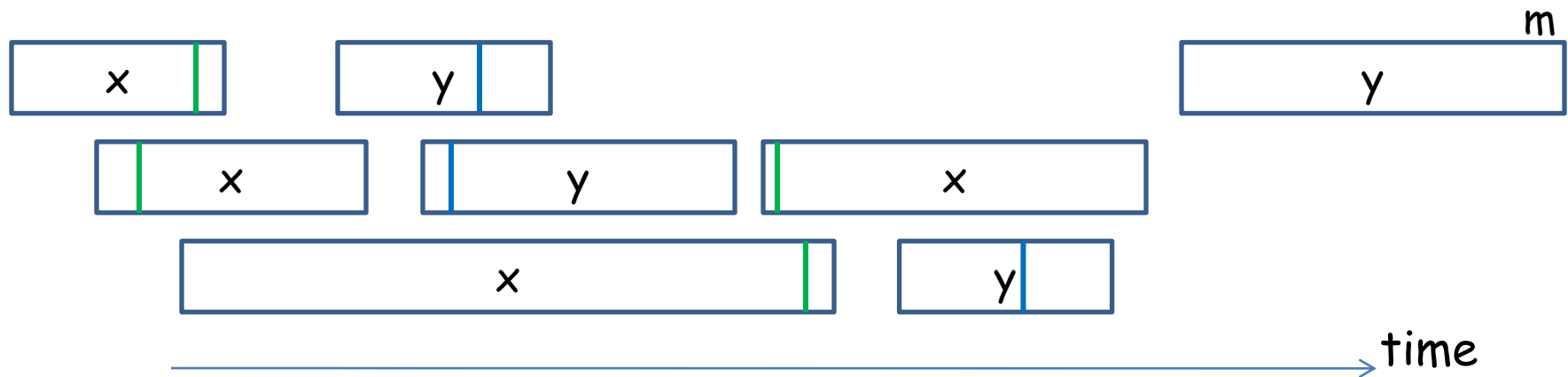
Proof (illustration): By assumption

- $H|x$  linearizable
- $H|y$  linearizable



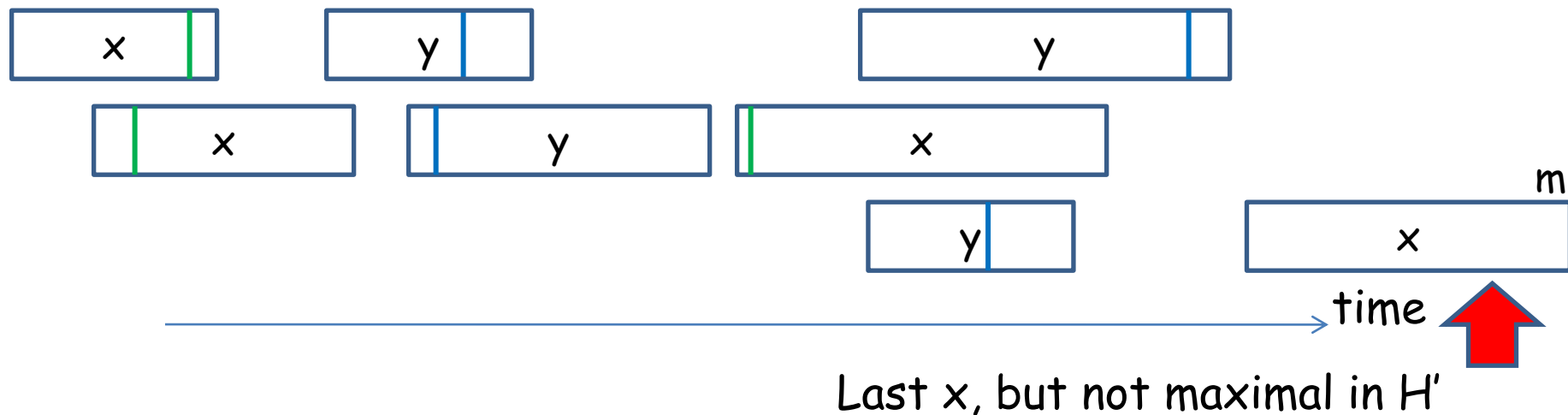
Remove  $m$  from  $H'$  to get  $G'$ ;  $H'$  is equivalent to  $G'm$  (because of maximality);  $G'$  has one less method call, by induction hypothesis  $G'$  is linearizable

$G$ 'm clearly equivalent to  $H$  ( $G_m|A = H|A$  for all threads  $A$ )





But for the  $G'$  obtained by removing a last, but not maximal  $m$ , it does not hold that  $G'm$  is equivalent to  $H$  (thread history for lower thread has changed)



Proof of maximality claim: "For each  $x$  consider last method call in  $Sx$ . One of these  $m$  must be maximal in  $H'$ , that is there is no  $m'$  with  $m \rightarrow m'$ ."

**Induction** on the number of objects  $x$ .

**Induction base:** If there is only one object  $x$  in  $H$ , then the last call  $m$  in the linearization  $Sx$  is maximal; if not, that is  $m \rightarrow m'$  for some  $m'$  in  $H$ , then also  $m \rightarrow m'$  in  $Sx$  and  $m$  could not have been last (**contradiction**).

**Induction step:** Pick an object  $x$  and remove all calls on  $x$  from  $H$ ; call this history  $H''$ . The last call  $m$  in the linearization of  $Hx$  is maximal in  $Hx$ . By the induction hypothesis, there is a last call  $m''$  in either of the object linearizations for  $H''$  that is maximal in  $H''$ . If  $m'' \rightarrow m$ ,  $m$  is maximal in  $H'$ , if  $m \rightarrow m''$ ,  $m''$  is maximal in  $H'$ , and if  $m$  and  $m''$  are overlapping either is maximal in  $H'$ .

Theorem: If  $\langle x.m \ A \rangle$  is a pending invocation of method  $m$  of  $x$  in a linearizable  $H$ , then there exist a matching response  $\langle x.v \ A \rangle$  such that  $H \langle x.v \ A \rangle$  is linearizable

Thus, linearizability does **not force a thread** (here  $A$ ) **to block**; linearizable  $H$  can be extended with matching responses

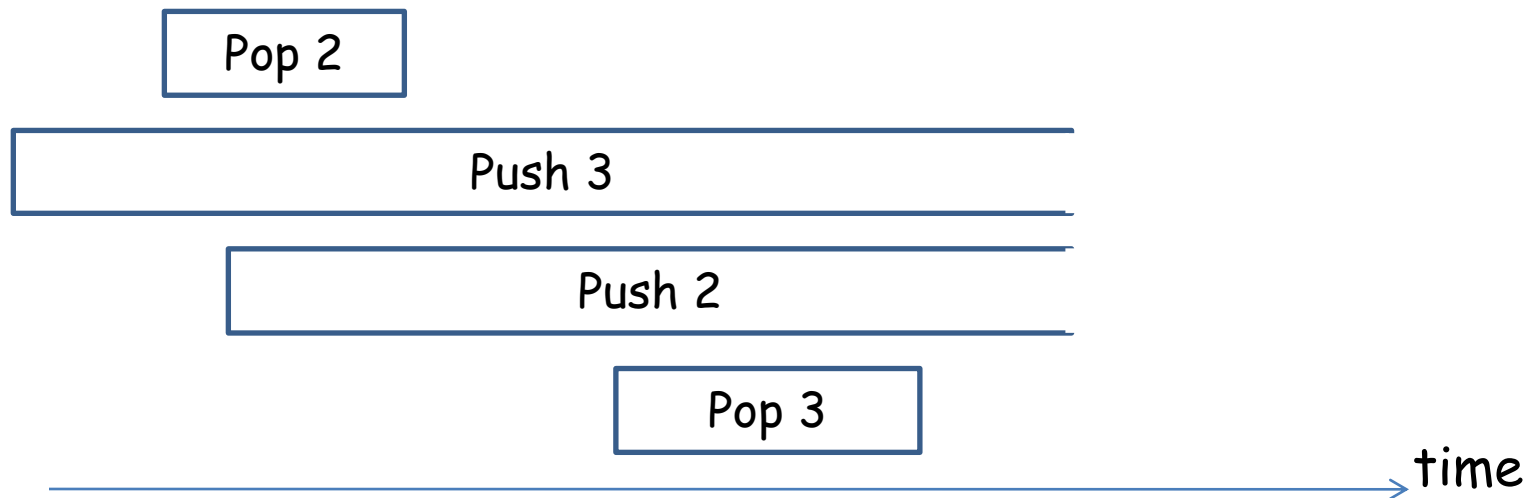
Proof:

Let  $S$  be a linearization of  $H$ . If  $S$  already includes  $\langle x.v \ A \rangle$ , then  $S$  is also linearization of  $H \langle x.v \ A \rangle$ . Otherwise,  $S$  does not include  $\langle x.m \ A \rangle$  (linearizations do not have pending invocations). There exists a response such that  $S \langle x.m \ A \rangle \langle x.v \ A \rangle$  is legal. This is a linearization of  $H \langle x.v \ A \rangle$

**Note**: Assumed that method calls are total, a response per sequential specification always possible

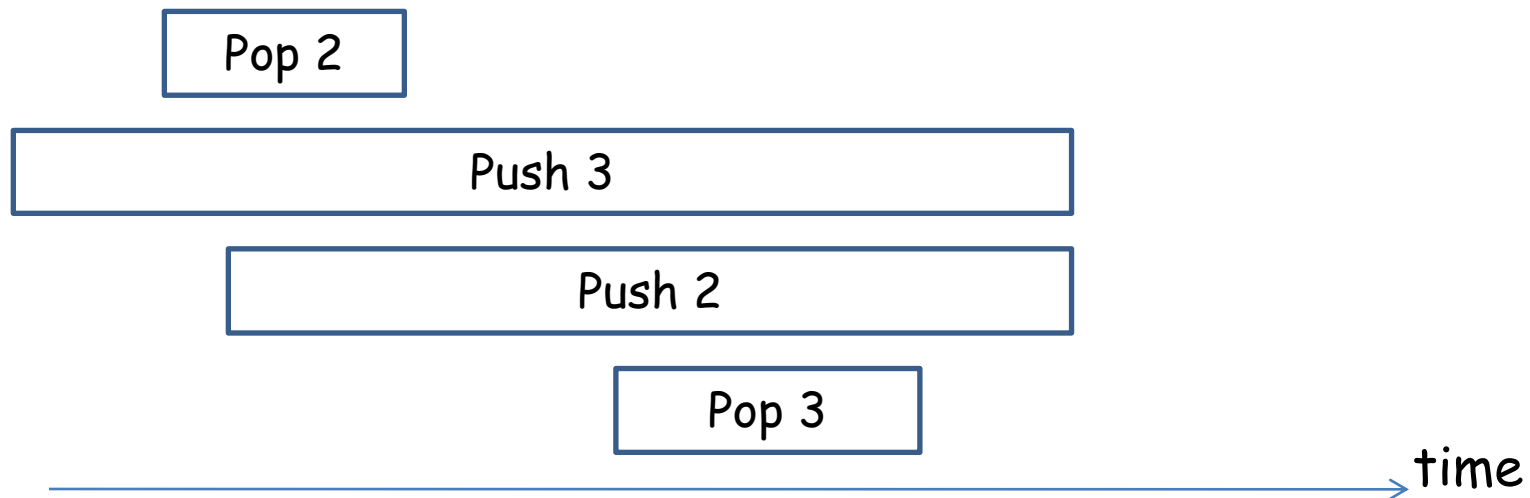
Example:

History H of LIFO stack operations, two pending Push operations, Pop 3 after Pop 2. Is this history linearizable?



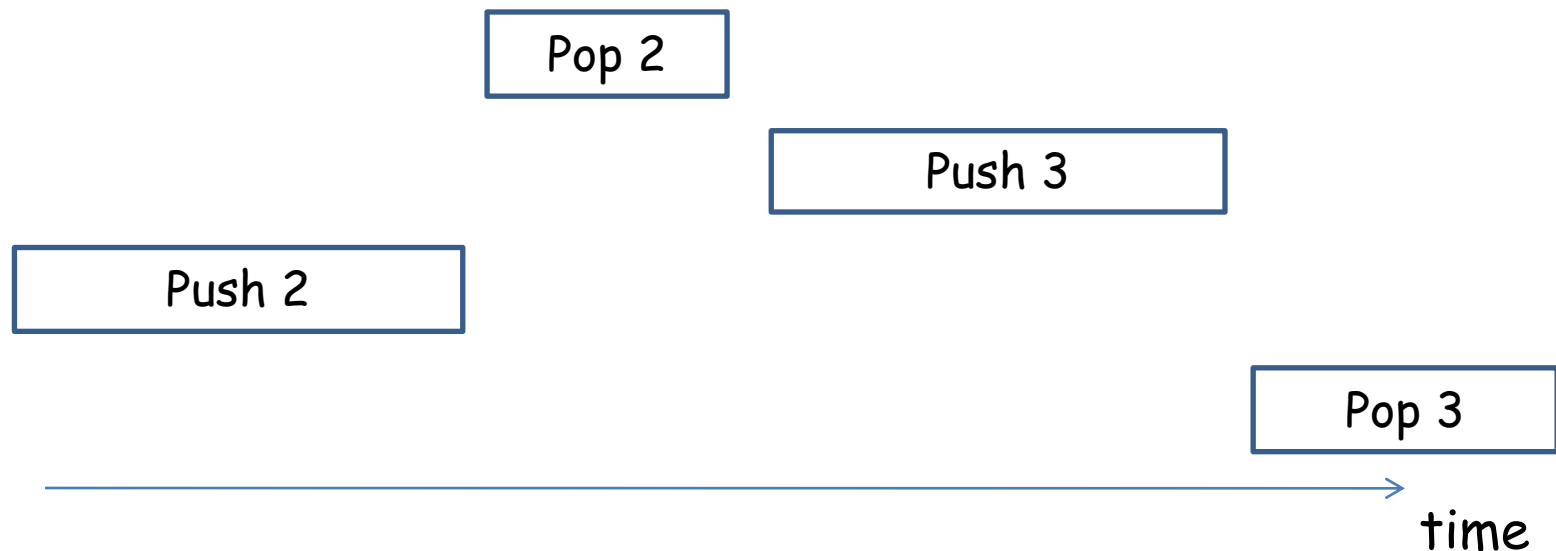
Example:

History H of LIFO stack operations, two pending Push operations, Pop 3 after Pop 2. For the linearization, H must first be extended with responses to the two push operations



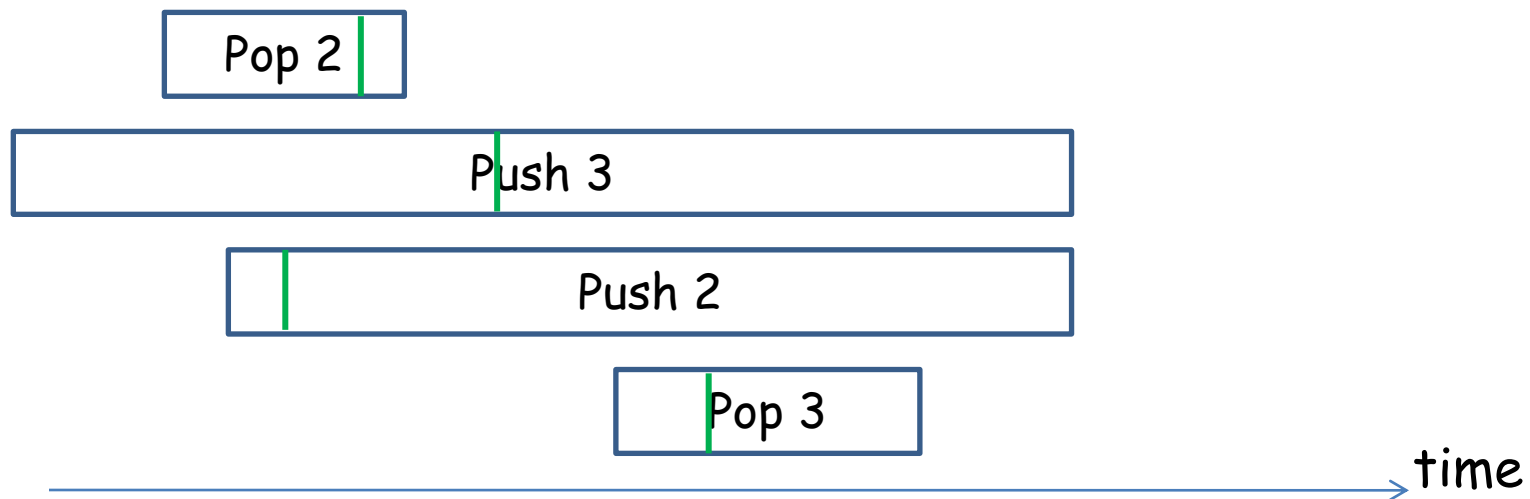
Example:

History H of LIFO stack operations, two pending Push operations, Pop 3 after Pop 2. A possible (only?) linearization is this sequential history:



Example:

History H of LIFO stack operations, two pending Push operations, Pop 3 after Pop 2. The linearization has linearization points in this order:



### Definition:

History  $H$  is sequentially consistent if there is an **extension  $H'$**  and a **legal sequential history  $S$**  such that

1.  $\text{complete}(H')$  is equivalent to  $S$
2. if  $m_0 \rightarrow m_1$  in  $H|A$  for method calls  $m_0, m_1$  for some thread  $A$ , then  $m_0 \rightarrow m_1$  in  $S$

Recall, linearization has:

2. if  $m_0 \rightarrow m_1$  in  $H$  for method calls  $m_0, m_1$ , then  $m_0 \rightarrow m_1$  in  $S$ .

### Corollary:

Linearizability implies sequential consistency (not vice versa)

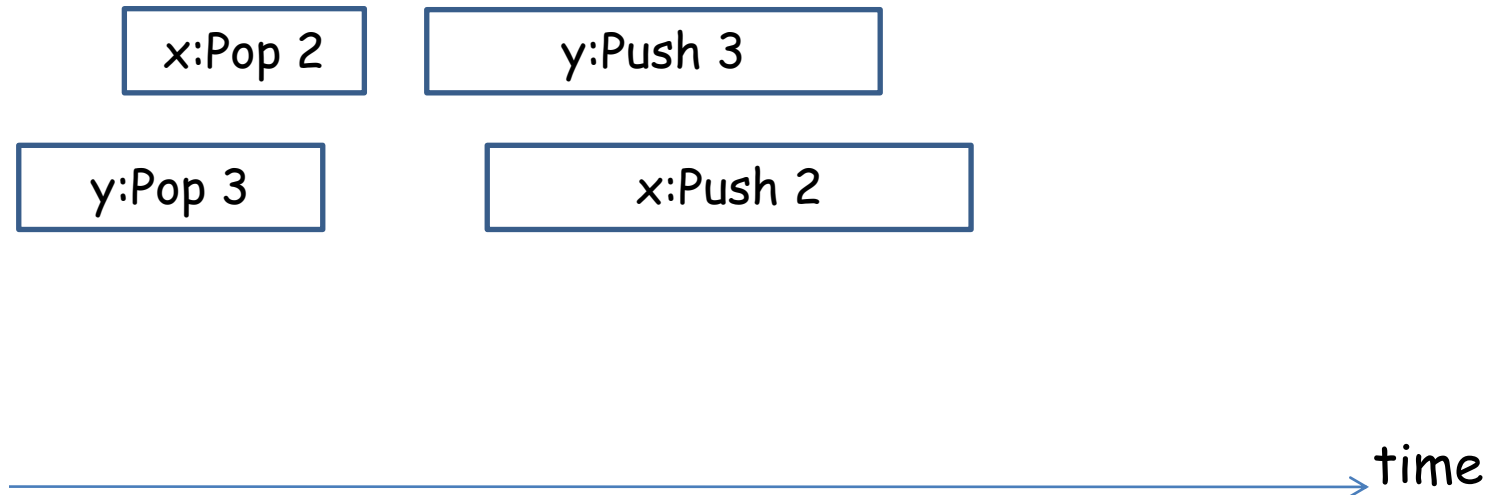


Theorem (without proof): Sequential consistency is **non-blocking**

But sequential consistency is **not** compositional (see previous **counterexamples**)

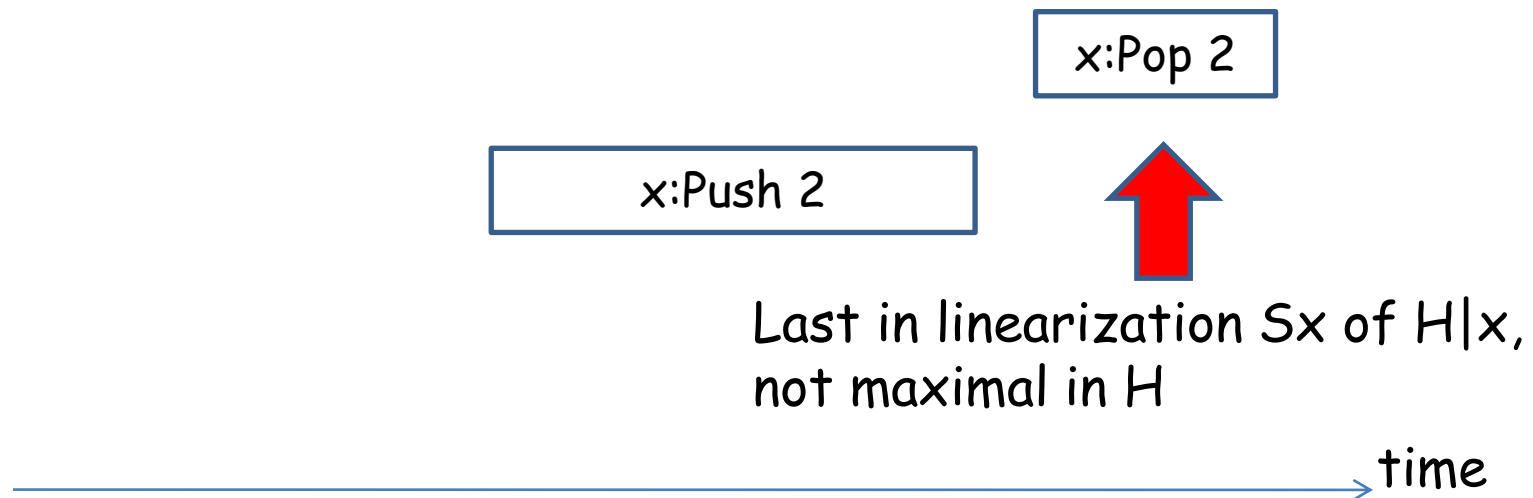
The proof of compositionality (if-part) for linearizability breaks down for sequential consistency:

The linearization points of each  $H|x$  may change (slide) the real-time history for object  $x$ . Thus no guarantee that any last  $m$  in an  $Sx$  will be maximal in  $H$



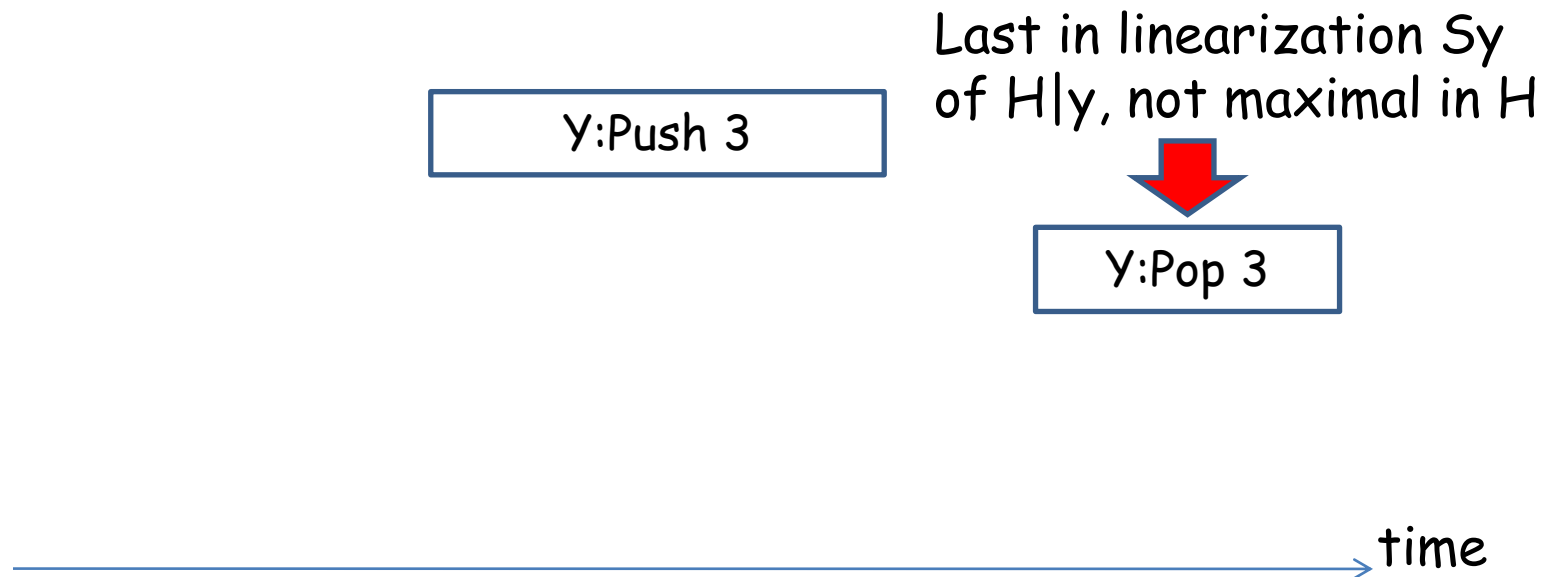
The proof of compositionality (if-part) for linearizability breaks down for sequential consistency:

The linearization points of each  $H|x$  may change (slide) the real-time history for object  $x$ . Thus no guarantee that any last  $m$  in an  $S_x$  will be maximal in  $H$



The proof of compositionality (if-part) for linearizability breaks down for sequential consistency:

The linearization points of each  $H|x$  may change (slide) the real-time history for object  $x$ . Thus no guarantee that any last  $m$  in an  $Sx$  will be maximal in  $H$



## Linearizability:

- Method calls should appear to happen in a one-at-a-time ("atomic"), sequential order
- Each method call should appear to take effect instantaneously at some point between invocation and response event

### Definition:

History  $H$  is linearizable if there is an **extension  $H'$**  and a **legal sequential history  $S$**  such that

1.  $\text{complete}(H')$  is equivalent to  $S$
  2. if  $m_0 \rightarrow m_1$  in  $H$  for method calls  $m_0, m_1$ , then  $m_0 \rightarrow m_1$  in  $S$
- $S$  is called a **linearization** of  $H$

End of technicalities (part II)

End of digression

## What "composable" does not mean

Composite operations consisting of component operations on different, linearizable objects are **not necessarily linearizable**.  
Most often not

Example: Accounts A and B with operations deposit(), withdraw(), balance()

### Transfer:

...

```
A.withdraw(100);  
B.deposit(100);
```

...

### Total:

...

```
total = A.balance();  
total += B.balance();
```

...

Transfer:

...

`A.withdraw(100);``B.deposit(100);`

...

Total:

...

`total = A.balance();``total += B.balance();`

...

Indicated history is not linearizable (correct)

Example: Start with 500 in both A and B, total will be 900, and **incorrect**

Would need to create new data structure with new operations `transfer(A,B,amount)`, and `total(A,B)`

Transactions can help (put the two operations into transactions)

Some desirable concurrent objects/data structures:

FIFO, LIFO queues, lists, ... with operations like their sequential counterparts (enqueue/dequeue, push/pop, insert/search/delete, ...)

SEE NEXT LECTURES FOR HOW TO IMPLEMENT THESE

Assume such data structures:

Can they solve the consensus problem? In other words: What are their power to solve concurrent coordination problems; and what is required to implement them?



Theorem:

Wait-free FIFO queues have consensus number at least 2

Proof:

By construction. Enqueue two values WIN and LOOSE, decide ("who was first"?) by dequeuing and checking value

```

class FIFOconsensus extends Consensus {
    private static final int WIN = 0; // elt for 1st thread
    private static final int LOSE = 1; // elt for 2nd thread
    Queue queue;
    public FIFOconsensus() {
        queue = new Queue();
        queue.enq(WIN); queue.enq(LOSE);
    }
    public T decide(T value) {
        propose(value);
        int i = ThreadID.get();
        int elt = queue.deq();
        if (elt==WIN) return proposed[i]; // thread won
        else return proposed[1-i]; // thread lost
    }
}

```

Corollary:

Atomic registers do not suffice to construct a wait-free FIFO queue!

By implication; similar constructions (exercise):  
Atomic registers do not suffice to construct wait-free lists, stacks, sets, bags, priority queues, ...

Proofs: See Exercises...

### Theorem:

FIFO queues have consensus number exactly 2

Proof:

By **contradiction**. Assume a binary consensus protocol for 3 threads A, B, C exists.

The protocol has a critical state. Let A's next move lead to a 0-valent state, B's next move to a 1-valent state (C indifferent).

**Observe:** A and B moves cannot commute (otherwise contradiction, indistinguishable state, cannot be both 0- and 1-valent), so A and B moves must be method calls of same shared resource. A and B cannot be about to read/write registers.

Thus A and B's moves must be method calls of a single shared FIFO queue object

3 possibilities:

1. A and B both call `deq()`
2. A calls `enq(a)`, B calls `deq()`
3. A and B both call `enq()`

See also:

M. Raynal: Concurrent programming: Algorithms, principles and foundations. Springer, 2013

with more careful arguments

1. A and B both calls `deq()`

Two sequences:

1. A `deq()`, B `deq()` 0-valent  $s'$ , at most 2 items gone
2. B `deq()`, A `deq()` 1-valent  $s''$ , at most 2 items gone

But states  $s'$  and  $s''$  are indistinguishable to C, so C must decide same value. **Contradiction!**

2. A calls  $\text{enq}(a)$  and B calls  $\text{deq}()$

If queue is nonempty, the two operations commute, so both sequences

1. A  $\text{enq}(a)$ , B  $\text{deq}(x)$ , 0-valent  $s'$
2. B  $\text{deq}(x)$ , A  $\text{enq}(a)$ , 1-valent  $s''$

lead to states  $s'$  and  $s''$  that are again indistinguishable to  $C$ , so  $C$  must decide same value from both. **Contradiction**

If queue is empty, the sequences

1. A  $\text{enq}(a)$ , 0-valent  $s'$
2. B  $\text{deq}()$ , A  $\text{enq}(a)$ , 1-valent  $s''$  **Contradiction** again

lead to states  $s'$  and  $s''$  that are indistinguishable to  $C$ .

### 3. A and B both call enq()

1. First A enq(a), then B enq(b). Let A run until deq(a), then run B until deq(b). State  $s'$  is 0-valent (A moved first)
2. First B enq(b), then A enq(a). Let A run until deq(b), then run B until deq(a). State  $s''$  is 1-valent (B moved first)

Ad:

1.  $\langle x_1, x_2, \dots, x_k, a \rangle$  is the FIFO state where global state became 0-valent
2.  $\langle x_1, x_2, \dots, x_k, b \rangle$  is the FIFO state where global state became 1-valent

The two states are indistinguishable to A and B until the elements  $x_1, x_2, \dots, x_k$  have been dequeued. In order for A or B to know what the decision value is,  $k+1$  values must be dequeued.



A, B, C make moves from initial state, assume k enq() operations

But **none** of these enq() operations make state univalent

FIFO queue with k elements

$d=0/1$

**Critical state**, next move by A will lead to 0-valent state, next move by B to 1-valent state (C could be either)

A enq(a)

$\langle x_1, x_2, \dots, x_k, a \rangle$

$d=0$

B enq(b), A runs until deq(a)

$\langle x_1, x_2, \dots, x_k, a, b \rangle$

$\langle b, y_1, y_2, \dots \rangle$

$d=0$

B deq(b)

$\langle y_1, y_2, \dots \rangle$

$s'$

A, B, C make moves from initial state, assume k enq() operations

But **none** of these enq() operations make state univalent

FIFO queue with k elements

A enq(a)

$\langle x_1, x_2, \dots, x_k, b \rangle$

B enq(b)

d=0

d=1

$\langle x_1, x_2, \dots, x_k, b, a \rangle$

$\langle a, y_1, y_2, \dots \rangle$

A enq(a), A runs until deq(b)

d=1

$\langle y_1, y_2, \dots \rangle$

B deq(a)

$S'''$

**Critical state**, next move by A will lead to 0-valent state, next move by B to 1-valent state (C could be either

### 3. A and B both call enq()

1. First A enq(a), then B enq(b). Let A run until deq(a), then run B until deq(b). State  $s'$  is 0-valent (A moved first)
2. First B enq(b), then A enq(a). Let A run until deq(b), then run B until deq(a). State  $s''$  is 1-valent (B moved first)

In order for A to return  $d=0$  (in case 1) it must observe the state of the queue by deq(x), and  $x=a$  since A moved first. Similarly for case 2.

But states  $s'$  and  $s''$  are indistinguishable to C, since the same number of elements have been dequeued (and enqueued, and updates to registers cannot help), so C must decide same value from both. **Contradiction**

## Hardware support(I)

R(ead)M(odify)W(rite) operations: Special instructions/methods that **atomically read and update** a **register r**. Examples

<code>r.getandset(v):</code>	returns value of r, replaces r with v
<code>r.getandinc():</code>	returns value of r, increments r
<code>r.getandadd(k):</code>	returns value of r, replaces r with r+k
<code>r.get():</code>	returns value of r

Sometimes called: test-and-set, fetch-and-add, ...

Formally: A method is an RMW method for a function class  $F$ , if there is some  $f$  in  $F$ , such that it atomically returns current register value  $r$  and replaces  $r$  with  $f(r)$

<code>r.getandset(v):</code>	returns value of <code>r</code> , replaces <code>r</code> with <code>v</code> F: constant functions $f.v(x) = v$
<code>r.getandinc():</code>	returns value of <code>r</code> , increments F: $f(x) = x+1$
<code>r.getandadd(k):</code>	returns value of <code>r</code> , replaces <code>r</code> with <code>r+k</code> F: $f.k(x) = x+k$
<code>r.get():</code>	returns value of <code>r</code> F: $f(x) = x$

trivial

Formally: A method is an RMW method for a function class  $F$ , if there is some  $f$  in  $F$ , such that it atomically returns current register value  $r$  and replaces  $r$  with  $f(r)$

An RMW method/register is **nontrivial** if at least one  $f$  in  $F$  is **not** the **identity function**

Theorem:

Any non-trivial RMW register has consensus number at least 2

Proof: by construction. Consensus protocol for 2 threads

```
class RMWconsensus extends Consensus {  
    // initialize to v such that  $f(v) \neq v$   
    private RMWRegister r = new RMWRegister(v);  
    public Object decide(Object value) {  
        propose(value);  
        int i = ThreadID.get();  
        if (r.RMW()==v) return proposed[i]; // thread won  
        else return proposed[1-i]; // thread lost  
    }  
}
```

## Specific RMW methods

A set of functions  $F$  is in *COMMON* iff for any  $f_i$  and  $f_j$  in  $F$ , either

- $f_i(f_j(x)) = f_j(f_i(x))$  ( $f_i$  and  $f_j$  **commute**)
- $f_i(f_j(x)) = f_i(x)$  or  $f_j(f_i(x)) = f_j(x)$  (either **overwrites** the other)

*COMMON* RMW methods:

$r.\text{getandset}(v)$ : returns value of  $r$ , replaces  $r$  with  $v$   
 $f.v(x) = v$  **overwrites**

$r.\text{getandinc}()$ : returns value of  $r$ , increments  
 $f(f(x)) = x+1+1$  **commutes**

$r.\text{getandadd}(k)$ : returns value of  $r$ , replaces  $r$  with  $r+k$   
 $f.k(f.j(x)) = x+k+j$  **commutes**

Theorem:

Any RMW register in *COMMON* has consensus number 2

Proof:

By **contradiction**. Assume a 3-consensus protocol for threads *A*, *B* and *C* exists.

A critical state exists. The next move by either thread cannot be register read/write, and the moves cannot commute. Thus either next move must be an RMW update to a single register *r* with some function *f* in *COMMON*.



Assume  $A$  is about to update  $r$  with  $f_A$ ,  $B$  about to update with  $f_B$

Two possible cases:

1.  $f_B$  overwrites  $f_A$ ,  $f_B(f_A(x)) = f_B(x)$
2.  $f_A$  and  $f_B$  commute,  $f_A(f_B(x)) = f_B(f_A(x))$

1.  $f_B$  overwrites  $f_A$ ,  $f_B(f_A(x)) = f_B(x)$

Consider two sequences

- $A$  moves,  $f_A(x)$ ,  $B$  moves,  $f_B(f_A(x)) = f_B(x)$ , 0-valent  $s'$
- $B$  moves,  $f_B(x)$ , 1-valent  $s''$

However, states  $s'$  and  $s''$  are indistinguishable to  $C$ , since the value of  $r$  is the same and only the internal states of  $A$  and  $B$  can differ. Thus  $C$  should decide same value in both  $s'$  and  $s''$ .

**Contradiction**

2.  $f_A$  and  $f_B$  commute,  $f_A(f_B(x)) = f_B(f_A(x))$

Consider two sequences

- A moves  $f_A(x)$ , then B moves  $f_B(f_A(x))$ , 0-valent  $s'$
- B moves  $f_B(x)$ , then A moves  $f_A(f_B(x))$ , 1-valent  $s''$

Again, states  $s'$  and  $s''$  that can differ only in the internal states of A and B, are indistinguishable to C, so C should decide same value in both. **Contradiction.**

Corollary:

FIFO queues (stack, ...) have the same consensus number as COMMON RMW, namely 2.

This means that it might well be possible to construct a wait-free queue (stack, ...) using only COMMON RMW registers and atomic registers.

## Hardware support(II)

An atomic *SWAP* operation swaps the contents of **two** registers

```
r1.SWAP(r2): atomically:  
temp = r1; r1 = r2; r2 = temp;
```

The atomic *SWAP* is not an RMW operation, it operates on two registers

Theorem: SWAP has consensus number  $n$

Proof: By **construction**. Consensus protocol for  $n$  threads

```
class SWAPconsensus extends Consensus {  
    private AtomicInteger R = new AtomicInteger(1);  
    private AtomicInteger A[n] =  
        new AtomicInteger[n](0);  
    public Object decide(Object value) {  
        propose(value);  
        int i = ThreadID.get();  
        A[i].SWAP(R); // first i gets 1  
        for (i=0; i<n; i++)  
            if (A[i]==1) return proposed[i];  
    }  
}
```

The `decide()` method is clearly wait-free (bounded loop)

First thread to call `decide()` and execute swap has  $A[i] == 1$  (from the initial value of  $R$ ), all other threads  $A[j] == 0$ , regardless of whether they have called `decide()` or not

Alternatively:

The algorithm maintains the invariant that  $R + \sum_{0 \leq i < n} A[i] = 1$

**Note:**

The number of threads  $n$  must be known in advance and remain fixed.

## Hardware support(III)

Compare-and-swap (CAS): Extended RMW operation on register  $r$  that **atomically** compares current content of  $r$  to an **expected value**  $e$  and if same replaces with an **update value**  $u$ , returns true if update was performed, false if not (alternatively: returns current content)

```
r.CAS(e,u): atomically:  
if (r.get()==e) { r.set(u); return true; }  
else return false;
```

Sometimes called test-and-set, compare-and-exchange, ..., and sometimes returns old value of register  $r$

CAS supported by Intel, AMD, Sun/Oracle Sparc, ...



Alternative compare-and-exchange/load (CEX) **atomically** compares current content of *r* to an **expected value** *e* and if same replaces with an **update value** *u*, returns true if update was performed, returns false if not and **updates** *u* (reference) to *e* (reference) atomically

```
r.CASL(e,u): atomically:  
if (r.get()==e) { r.set(u); return true; }  
else { u.set(e); return false; }
```

CEX semantics of the C11/C++ `atomic_compare_exchange` operation

LL/SC (load linked, store conditional):

LL reads (loads) address  $a$ ; the subsequent SC stores a new value to  $a$ , if  $a$  has **not** changed since the LL; if  $a$  has changed **SC fails**, and nothing happens to  $a$

**Note:**

In practical, HW implementations of LL/SC there may be other conditions leading SC to fail (distance/time between LL and SC too long)

LL/SC supported by IBM PowerPC, MIPS 11, ARM, ...

## RISC-V LR/SC (load reserved/store conditional)

```
# emulated RISC-V CAS
0:      lr.w a3,(a0)          # load reserved
      bne a3,a1,80          # not equal?
      sc.w a3,a2,(a0)        # store conditional
                                # (succeed if no change)
      bnez a3,0              # retry (wait-free?)
80:
```

ISA design argument for not supporting CAS: has three source registers, would require new operand instruction format

David Patterson, Andrew Waterman: The RISC-V Reader: An open architecture atlas. Strawberry Canyon LLC, 2017

## Other RISC-V atomics (amo: atomic memory operation)

`amo{add, and, or, swap, xor, max, min}.word`

all with acquire/release bits (see later)

Theorem: CAS has consensus number  $\infty$

Proof: By **construction**. Consensus protocol for any number of threads (number threads does **not** need to be known in advance)

```
class CASconsensus extends Consensus {
    private int first = -1;
    private AtomicInteger r = new AtomicInteger(first);
    public Object decide(Object value) {
        propose(value);
        int i = ThreadID.get();
        if (r.CAS(first,i))
            return proposed[i]; // thread won
        else return proposed[r.get()]; // thread lost
    }
}
```

Theorem: LL/SC has consensus number  $\infty$

Proof: By construction. **Exercise**

## Universality of consensus (Chap. 6)

Fundamental result: There exist classes of objects, such that given enough of them it is possible to construct wait-free, linearizable implementations of **any** sequentially specified concurrent object

An object with this property is called universal

Theorem: Objects with consensus number  $\infty$  are universal

Corollary: CAS is universal, LL/SC is universal

Proof by construction; constructions explicit **but very inefficient**

### Idea:

Use consensus to linearize concurrent operations on the sequentially specified object; each thread locally performs the sequence of operations on a local copy of the object to compute response.

The **deterministic**, sequential object is specified by a generic interface. The `invoke()` method is the invocation which returns a response

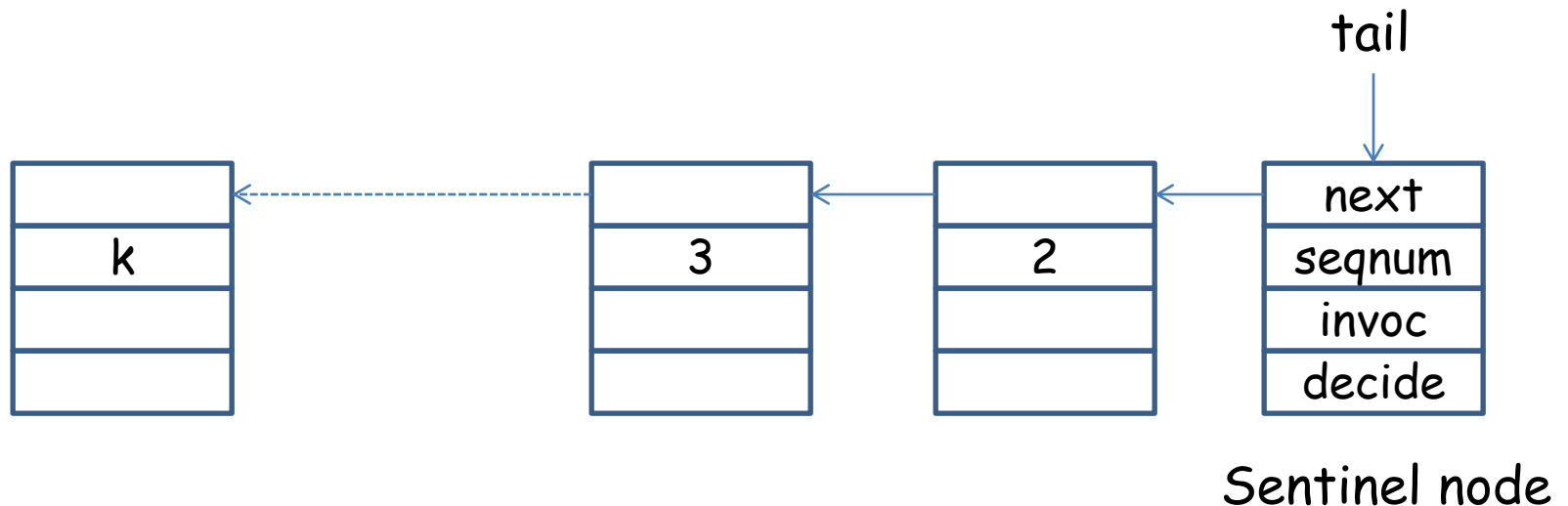
```
public interface SeqObject {  
    public abstract Response invoke();  
}
```



Linearized execution history represented by a list of Node's

```
public class Node {  
    public Invoc invoc; // parameters  
    public Consensus<Node> decideNext;  
    public Node next;  
    public int seqnum;  
    public Node(Invoc invoc) {  
        invoc = invoc;  
        decideNext = new Consensus<Node>;  
        seqnum = 0;  
    }  
    public static Node max(Node[] array) {  
        return maximum(array[i].seqnum for some i);  
    }  
}
```

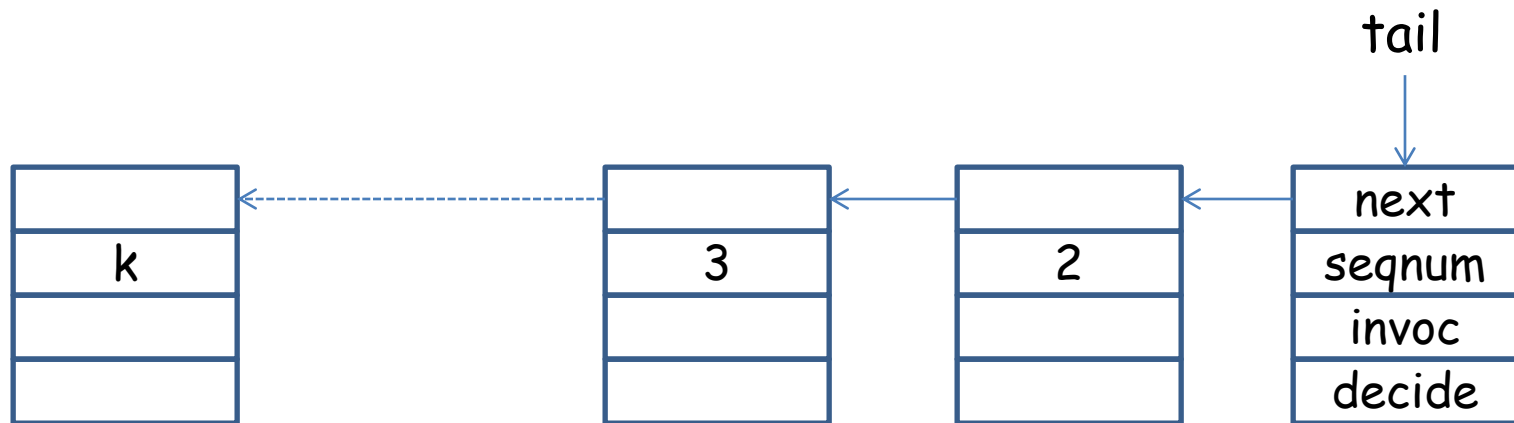
A history of  $k$  invocations



Thread  $i$ :

To compute response after  $k$  invocations, allocate local object, start with initial invoc-parameters, follow next pointers through list, update object state at each node

## A history of k invocations



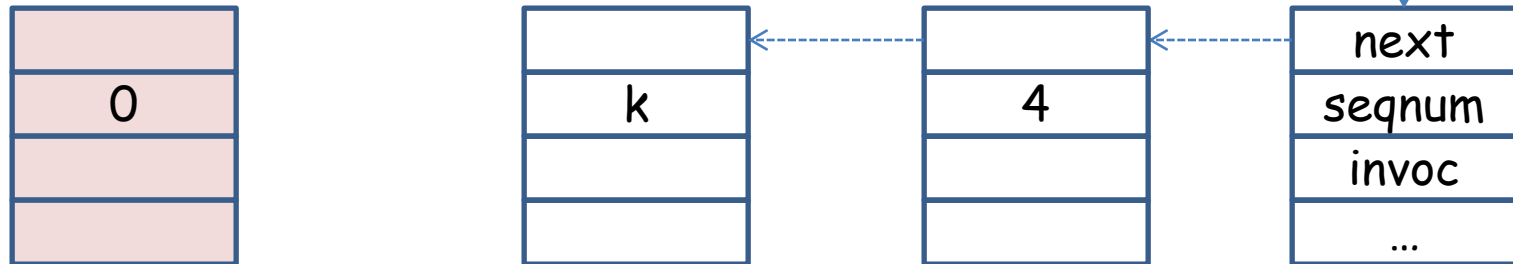
Sentinel node

```
SeqObject object = new SeqObject();
Node current = tail.next;
while (current != prefer) {
    object.invoke(current.invoc);
    current = current.next;
}
return object.invoke(current.invoc);
```

## Lock-free universal construction

Invocation by thread i:

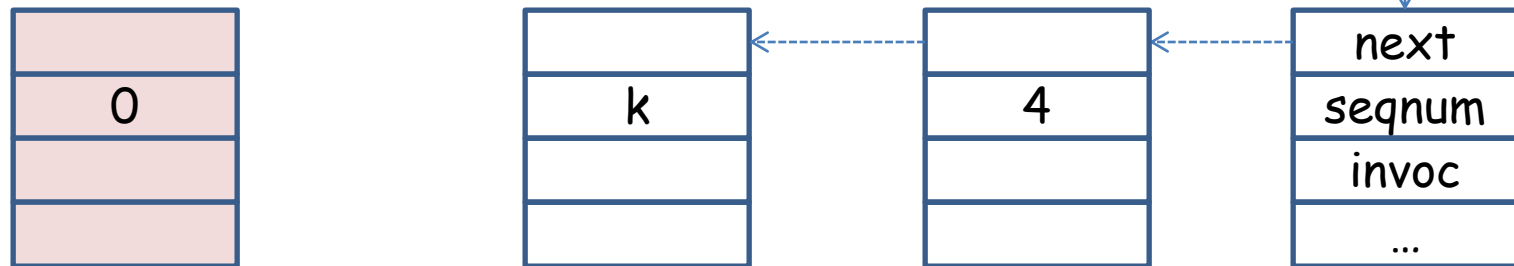
Allocate node, store arguments, use consensus to agree when it can be appended to list



## Lock-free universal construction

Invocation by thread i:

Allocate node, store arguments, use consensus to agree when it can be appended to list



Head-array shared by all threads for each thread references threads last head



Thread i:

Initializing head: Each thread points to fixed tail node

```
public class LockFreeUniversal {  
    private Node[] head;  
    private Node tail; // sentinel Node  
    public Universal() {  
        tail = new Node();  
        tail.seqnum = 1;  
        for (i=0; i<n; i++) head[i] = tail;  
    }  
    public Response invoke(Invoc invoc);  
}
```

Invariant: head[i] is last Node that thread i appended;  
seqnum's in list are successive (from tail.seqnum==1)

To append new invocation by thread  $i$  (= linearize operation):

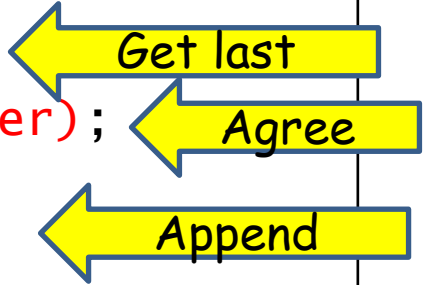
- Get overall last node (max seqnum in head[j] over all j)
- Apply consensus using this node's consensus object to find out which node shall be appended
- Append to list (**note**: more than one thread may do so, but they will do the same updates)

Repeat until own node is successfully appended (seqnum has been set)

```

public Response invoke(Invoc invoc) {
    int i = ThreadID.get();
    Node prefer = new Node(invoc); // new Node
    while (prefer.seqnum==0) {
        Node before = Node.max(head);
        Node after = before.decideNext(prefer);
        before.next = after;
        after.seqnum = before.seqnum+1;
        head[i] = after;
    }
    SeqObject object = new SeqObject();
    Node current = tail.next;
    while (current!=prefer) {
        object.invoke(current.invoc);
        current = current.next;
    }
    return object.invoke(current.invoc);
}

```



} prefer Node successfully appended; compute result



Theorem: The construction correctly implements the concurrent object and is lock-free

Proof:

On concurrent invocations, the use of the last Node's consensus object linearizes by choosing either of the invocations. This node is inserted correctly. Thread's "loosing" the consensus, reinsert a Node already in the list, but this is no harm.

Each thread calls each consensus object at most once: the outcome of the consensus is stored in `head[i]`, so a different object will be used in the next iteration.

Since a thread will iterate only when it loses consensus to some other thread, some thread is guaranteed to make progress and the construction is lock-free.

**Note:**

The values proposed to the consensus objects are Node references

**Swept under the rug:**

Memory management, must be wait-free (can be handled)

## Wait-free universal construction

In the lock-free construction, any thread can be overtaken an indefinite number of times. In a wait-free construction, each thread must be able to complete in a finite number of steps, regardless of what other threads do (outcome of consensus protocol)

To prevent this, use **helper scheme**: Each thread **announces** the node it wants to append, if unsuccessful in consensus, some other thread will insert the node

```
public class Universal {  
    private Node[] announce; // announced Nodes  
    private Node[] head;  
    private Node tail = new Node(); tail.seqnum = 1;  
    for (j=0; j<n; j++) {  
        head[j] = tail; announce[j] = tail;  
    }  
    public Response invoke(Invoc invoc);  
}
```

```

public Response invoke(Invoc invoc) {
    int i = ThreadID.get();
    announce[i] = new Node(invoc);
    head[i] = Node.max(head);
    while (announce[i].seqnum==0) {
        Node before = head[i];
        Node help = announce[(before.seqnum+1)%n];
        if (help.seqnum==0) prefer = help;
        else prefer = announce[i];
        Node after = before.decideNext(prefer);
        before.next = after;
        after.seqnum = before.seqnum+1;
    }
    SeqObject object = new SeqObject();
    Node current = tail.next;
    while (current!=announce[i]) {
        object.invoke(current.invoc);
        current = current.next;
    }
    head[i] = announce[i];
    return object.invoke(current.invoc);
}

```



Announce



Need help?



Try to help

prefer Node  
successfully  
appended;  
compute result

Theorem: The construction correctly implements the concurrent object and is wait-free

Proof: For details, see Herlihy-Shavit book

Some fundamental papers upon which this is all based:

Maurice Herlihy, Jeannette M. Wing: Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12(3): 463-492 (1990)  
Maurice Herlihy: Wait-Free Synchronization. *ACM Trans. Program. Lang. Syst.* 13(1): 124-149 (1991)

## Great for seminar talks

## More efficient, universal constructions

Panagiota Fatourou, Nikolaos D. Kallimanis: Highly-Efficient Wait-Free Synchronization. *Theory Comput. Syst.* 55(3): 475-520 (2014)

Panagiota Fatourou, Nikolaos D. Kallimanis: The RedBlue family of universal constructions. *Distributed Comput.* 33(6): 485-513 (2020)

Yehuda Afek, Dalia Dauber, Dan Touitou: Wait-free made fast (Extended Abstract). *STOC 1995*: 538-547

Alex Kogan, Erez Petrank: A methodology for creating fast wait-free data structures. *PPOPP 2012*: 141-150

James H. Anderson, Mark Moir: Universal Constructions for Large Objects. *IEEE Trans. Parallel Distrib. Syst.* 10(12): 1317-1332 (1999)

Andreia Correia, Pedro Ramalhete, Pascal Felber: A wait-free universal construction for large objects. *PPoPP 2020*: 102-116

## Lower consensus operations in terms of higher consensus

Fetch-and-add can easily be implemented using compare-and-swap

```
class FAA extends RMWregister {  
    private RMWRegister r = new RMWRegister();  
    public int fetchandadd(int k) {  
        int v = r.get();  
        while (!r.CAS(v,v+k)) v = r.get();  
        return ri;  
    }  
}
```

The implementation is correct and lock-free, but not wait-free. When a CAS fails for thread A, some other thread has successfully added its k. **Challenge:** Wait-free FAA by CAS



Challenge: Wait-free FAA by CAS.

See

Faith Ellen, Philipp Woelfel: An Optimal Implementation of Fetch-and-Increment. DISC 2013: 284-298

for a partial solution using LL/SC

## Extended compare-and-swap

**Atomically**, compare  $n$  values to  $n$  registers, and update all

```
class CASn {  
    private RMWRegister r1, r2, ..., rn;  
    public boolean CASn(r1,r2,...,rn,  
                        e1,e2,...,en,u1,u2,...,un) {  
        // atomically, synchronized  
        if (e1==r1.get() && e2==r2.get() && ... en==rn.get()) {  
            r1.set(u1);  
            r2.set(u2);  
            ...  
            rn.set(un);  
            return true;  
        } else return false;  
    }  
}
```

**Atomically**, compare  $k$  values to  $k$  registers, and update one of the registers

```
class kCAS {  
    private RMWRegister r[k];  
    public Boolean kCAS(r[],e[],j,u) {  
        for (i=0; i<k; i++) if (e[i]!=r[i].get())  
            return false;  
        r[j].set(u);  
        return true;  
    }  
}
```

These operations are sometimes useful (needed) for concurrent data structures, but usually not (?) supported in hardware.

Alternative: Emulate (**challenge**), or rely on **transactional memory**

## Transactional memory (example)

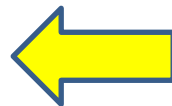
Transactional memory enables linearizability at a higher granularity.

Transaction: Sequence of instructions that are (semantically) executed atomically.

Transactions must be serializable, that is appear to execute sequentially in a one-at-a-time order, that is appear to take place instantaneously. Transactions are **non-blocking**.

Not Java, but what transactional models provide

```
atomic {  
  x = q0.deq();  
  q1.enq(x);  
}
```



Transaction

Even when transactions occur concurrently, no thread can observe that  $x$  has been removed from  $q0$  but is not present in  $q1$ , or that  $x$  is present in both  $q0$  and  $q1$ , or ...

## Implementation (hardware or software)

Transactions are executed speculatively, at the end of a transaction it is checked that atomicity is not violated, and if so, the transactions **commits**; if atomicity is violated, the transaction **aborts**

Hardware transactional support, e.g., in Intel, from Haswell architecture (THX).