Concurrent/Distributed Data Types

Correctness Criteria, Verification

Constantin Enea

Ecole Polytechnique

Parallel Data Processing

Software systems that support high-frequency parallel accesses to high-quantity data.

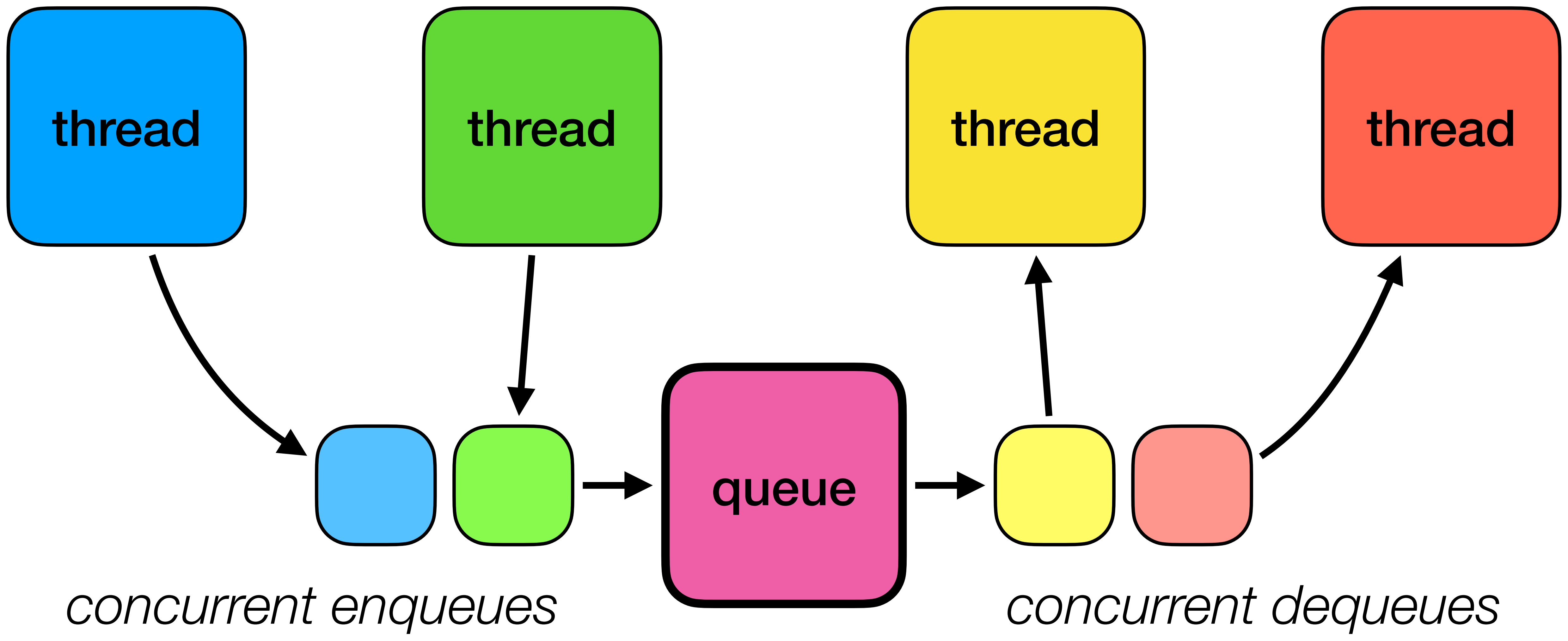
Rely on high-performance shared resources (data types).

- key-value maps, sets, queues, stacks, etc.

Deployed over a shared-memory or a network

Concurrent Objects

Multi-threaded programming



e.g. Java Development Kit SE

dozens of objects, including queues, maps, sets, lists, locks, atomic integers, …

Shared-State in Parallel Applications

- Parallelizing applications for efficiency

Sequential

```
// reading data for
future processing
q = new Queue();
while(…){
  X = readFile();
  q.enqueue(X);
}
```

Shared-State in Parallel Applications

- Parallelizing applications for efficiency

- multi-threading
- distributed over a network

Sequential

```
// reading data for
future processing
q = new Queue();
while(…){
  X = readFile();
  q.enqueue(X);
}
```

Parallel

```
q = new Queue();

while(…){                 while(…){
  X = readFile1();   ||     X = readFile2();
  q.enqueue(X);             q.enqueue(X);
}                         }
```

Shared-State in Parallel Applications

- Parallelizing applications for efficiency

- multi-threading
- distributed over a network

Sequential

```
// reading data for
future processing
q = new Queue();
while(…){
  X = readFile();
  q.enqueue(X);
}
```

Parallel

```
q = new Queue();
l = new Lock();
```

```
while(…){                  ||   while(…){
  X = readFile1();               X = readFile2();
  l.lock();                      l.lock();
  q.enqueue(X);                  q.enqueue(X);
  l.unlock();                    l.unlock();
}                               }
```

Shared-State in Parallel Applications

- Parallelizing applications for efficiency

- multi-threading
- distributed over a network

Sequential

```
// reading data for
future processing
q = new Queue();
while(…){
  X = readFile();
  q.enqueue(X);
}
```

Parallel

```
q = new Queue();
l = new Lock();
```

~~l = new Lock();~~

```
while(…){
  X = readFile1();
  l.lock();
  q.enqueue(X);
  l.unlock();
}
```

||

```
while(…){
  X = readFile2();
  l.lock();
  q.enqueue(X);
  l.unlock();
}
```

How to implement concurrent/distributed objects ?

How is correctness defined ? Verification ?

Verification Ingredients

- Specifying a Library: φ
- Implementing a Library: $\mathtt{L}$
- Verifying a Library implementation: $\mathtt{L} \models \varphi$

Specifying Sequential Objects

Object Specification

- How can we specify an object? (Library)
  - Objects API
  - Use cases
  - Pre and Post Conditions?
- What are the behaviors of a client using the library?

💡 for any client making library calls record the inputs and outputs of each call

java.util

Class Stack<E>

java.lang.Object
 java.util.AbstractCollection<E>
  java.util.AbstractList<E>
   java.util.Vector<E>
    java.util.Stack<E>

Method Summary

Methods

Modifier and Type	Method and Description
`boolean`	`empty()` Tests if this stack is empty.
`E`	`peek()` Looks at the object at the top of this stack without removing it from the stack.
`E`	`pop()` Removes the object at the top of this stack and returns that object as the value of this function.
`E`	`push(E item)` Pushes an item onto the top of this stack.
`int`	`search(Object o)` Returns the 1-based position where an object is on this stack.

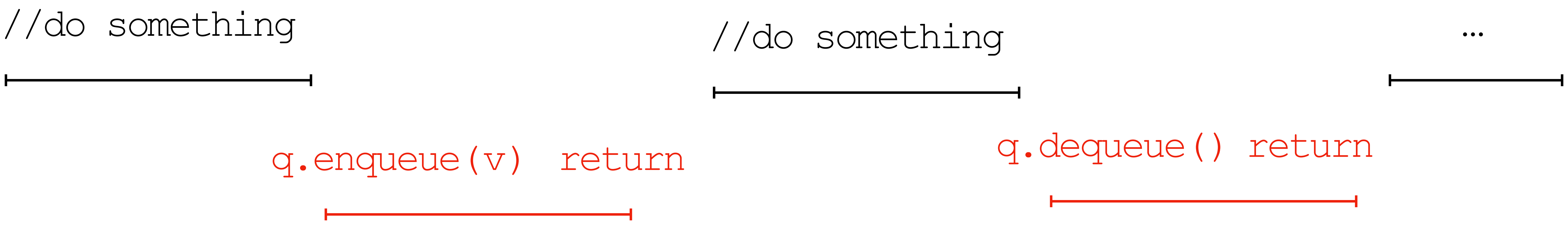
Methods inherited from class java.util.Vector

`add, add, addAll, addAll, addElement, capacity, clear, clone, contains, containsAll, copyInto, elementAt, elements, ensureCapacity, equals, firstElement, get, hashCode`

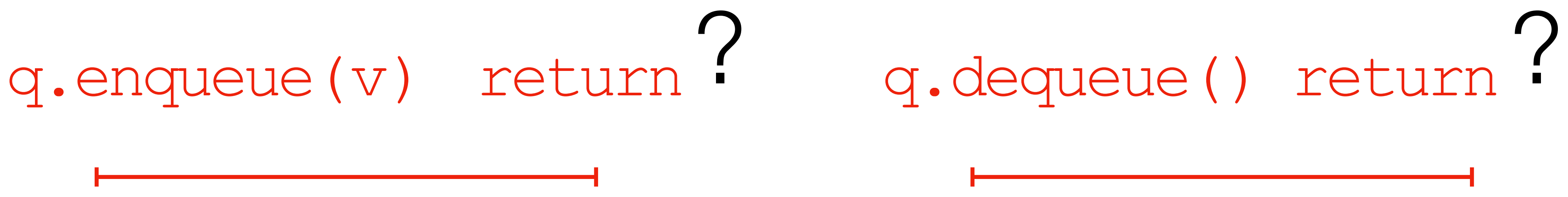
What is a client?

- What is a client of the Library?
    - Program that issues calls to a library instance

```
// do something
q.enque(v)
// do something
x = q.dequeue()
// …
```



- How do we specify a Data Structure (DS) generically?
    - Histories of calls and returns
    - Constraint possible return values



Well Encapsulated Objects

- *Global* object state:
  - Possibly *local* thread state
- A set of *operations* or *methods*
  - Input and output types
  - Methods are the only way to operate on the state

Sequential Object Specifications

- Library $L = <\Sigma, m_1, m_2, m_3>$

  State Space (Σ) Methods (m_1, m_2, m_3)

- Client C: Issues calls to the library methods
  - (Sequential) Most General Client [SMGC]

```
SMGC(L):
  while true do
    mi = choseMethodFrom(L);
    args = choseInputsFor(m);
    mi(args);
  od
```

- We will talk about histories of calls with values
  - ϵ denotes the empty sequence,
  - o denotes an operation (eg. <pop(),v>), and
  - δ denotes a sequence of operations

Specifying a Register

- Inductive histories of a Register:

1. ϵ is a Register History (RH)
2. `<read(),0>*` is a Register History
3. if δ is a RH, then so is $\delta \cdot$`<write(v),_>`
4. if $\delta \cdot$`<write(v),_>` is a RH, then so it is $\delta \cdot$`<read(),v>*`

Some examples on the board

Specifying a Stack

- Inductive histories of a Stack:
  1. ϵ is a Stack History (SH)
  2. if $\delta \cdot$<pop(),v> is a SH, then so is <push(w),_>$\cdot\delta$
  3. if δ is a SH, and $|\{$<pop(),v>$:\delta | v \neq \bot\}| = |$push(v),_>$:\delta\}|$, then so it is $\delta \cdot$<pop(),$\bot$>*
  4. same conditions as 3, and <pop(),$\bot$> does not occur in δ then, <push(w),$\bot$>$\cdot\delta\cdot$<pop(),w> is a SH
  5. if $\delta_0 \cdot$<pop(),$\bot$> is a SH, and δ_1 is a SH, then $\delta_0 \cdot \delta_1$ is SH

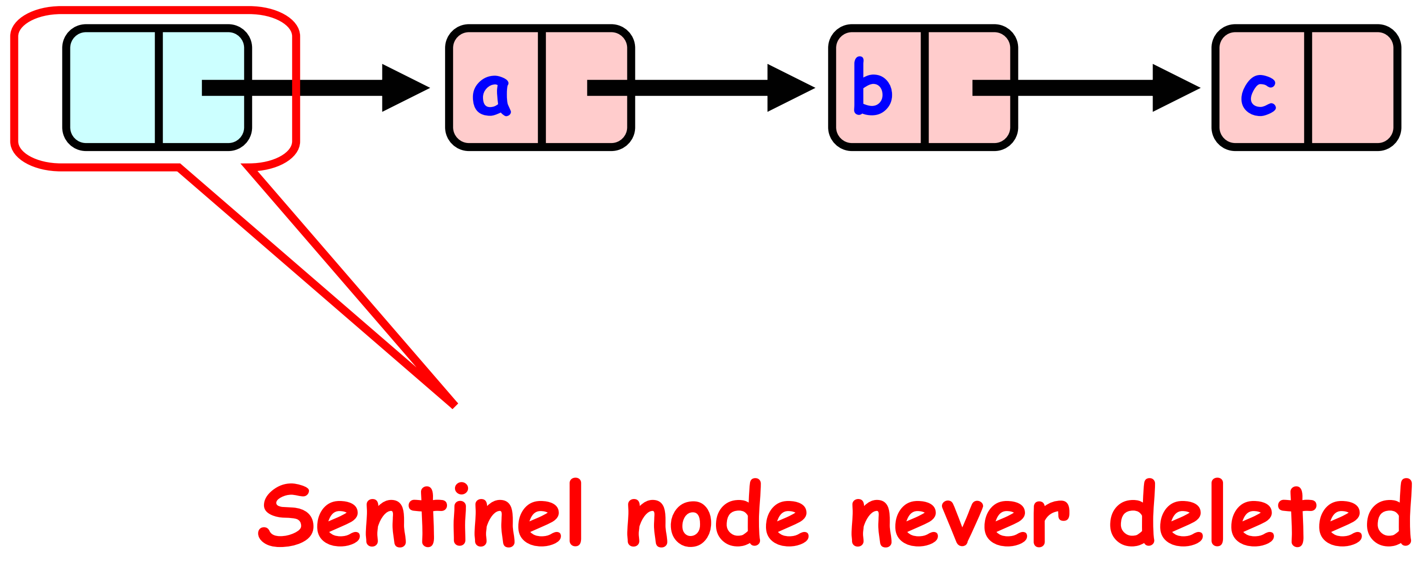
Specifying a Queue

Exercise

Implementations

- A set implementation based on sorted linked lists

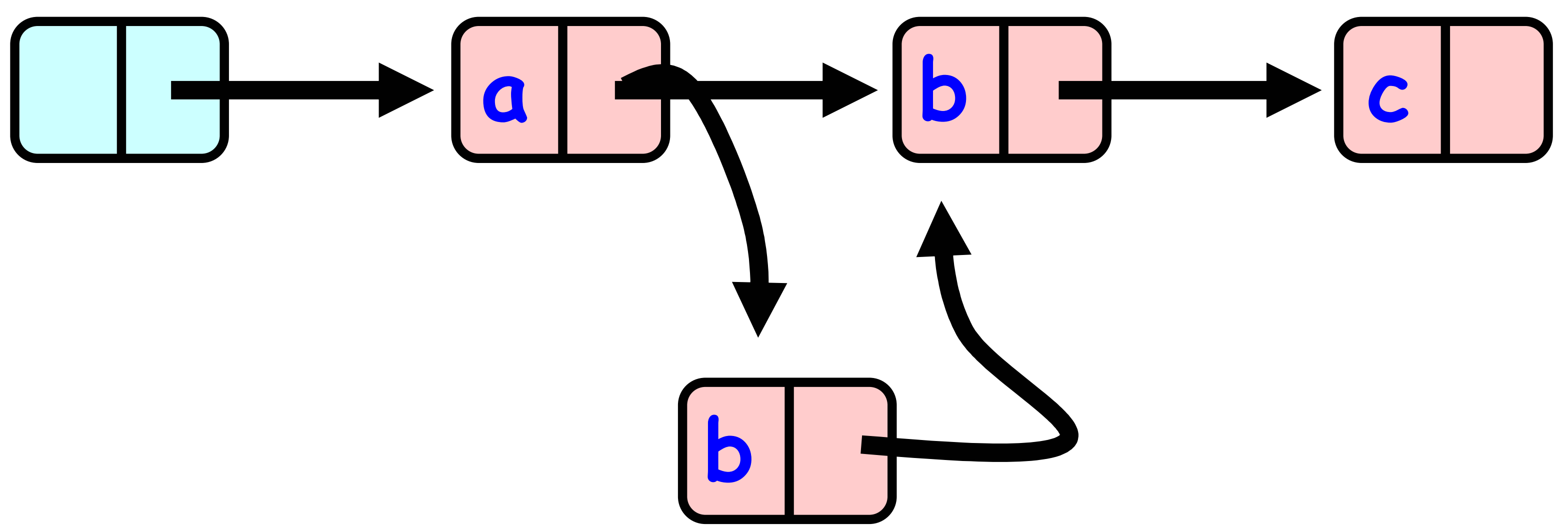
```
public class Entry {
 public Object value;
 public Entry next;
}
public class Set {
 Entry first;
 public boolean add(Object x) {…}
 public boolean remove(Object x) {…}
 public boolean contains(Object x) {…}
}
```



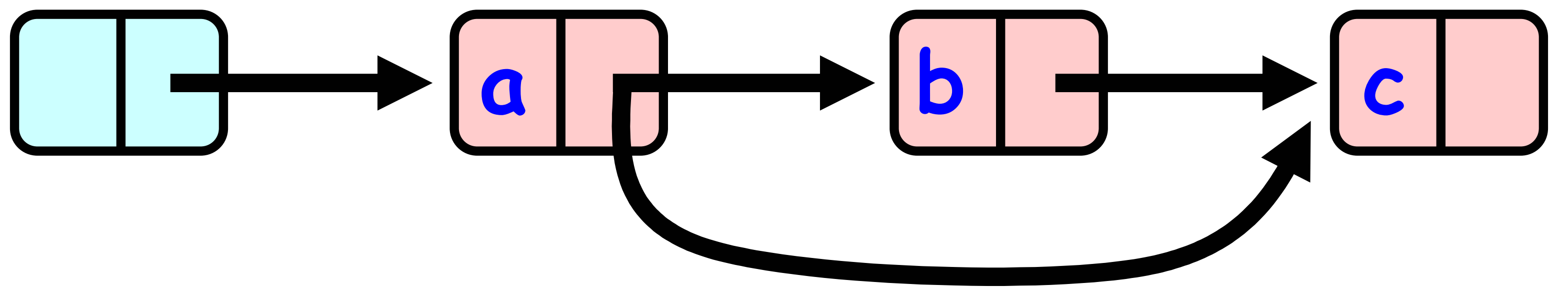
Implementations

- A set implementation based on sorted linked lists

adding an entry



removing an entry



Specifying Concurrent Objects

What about Concurrency?

```
while true do
  m_i = choseMethodFrom(L);
  args = choseInputsFor(m);
  m_i(args);
od
```

||

```
while true do
  m_i = choseMethodFrom(L);
  args = choseInputsFor(m);
  m_i(args);
od
```

s.push(v) return s.pop() return v ?

Concurrent Consistency Criteria

Should this be legal?

Concurrent Clients

- Most General Client (seq)

```
SMGC(L):
 while true do
  mi = choseMethodFrom(L);
  args = choseInputsFor(m);
  mi(args);
 od
```

- Most General Client (concurrent n threads)

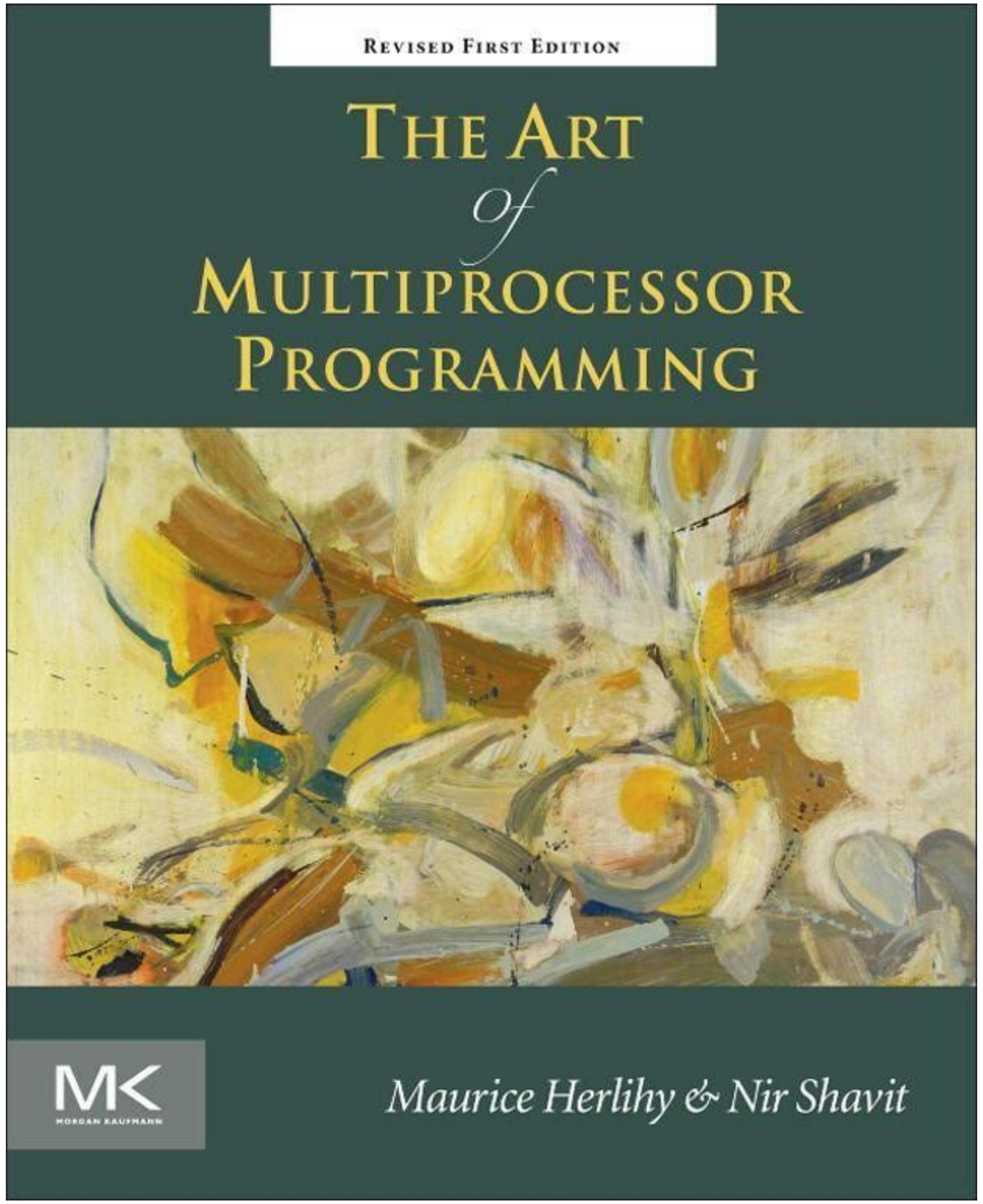
$$\mathtt{CMGC}_n\mathtt{(L)}: \quad \underbrace{\mathtt{SMGL(L)} \parallel \mathtt{SMGL(L)} \ \ldots \parallel \mathtt{SMGL(L)}}_{n}$$

- Concurrent Library Verification w.r.t. $\mathtt{CMGC}_n\mathtt{(L)}$ for any n

Concurrent Consistency Criteria

- Quiescence Consistency
- Sequential Consistency
- Linearizability

We will work with Registers to exemplify the definitions



Quiescent Consistency

- Method calls should appear to happen one-at-a-time, sequential order

`r.write(1); r.write(2);` ‖ `r.read(); r.read();`

r.write(1) ret   r.write(2) ret

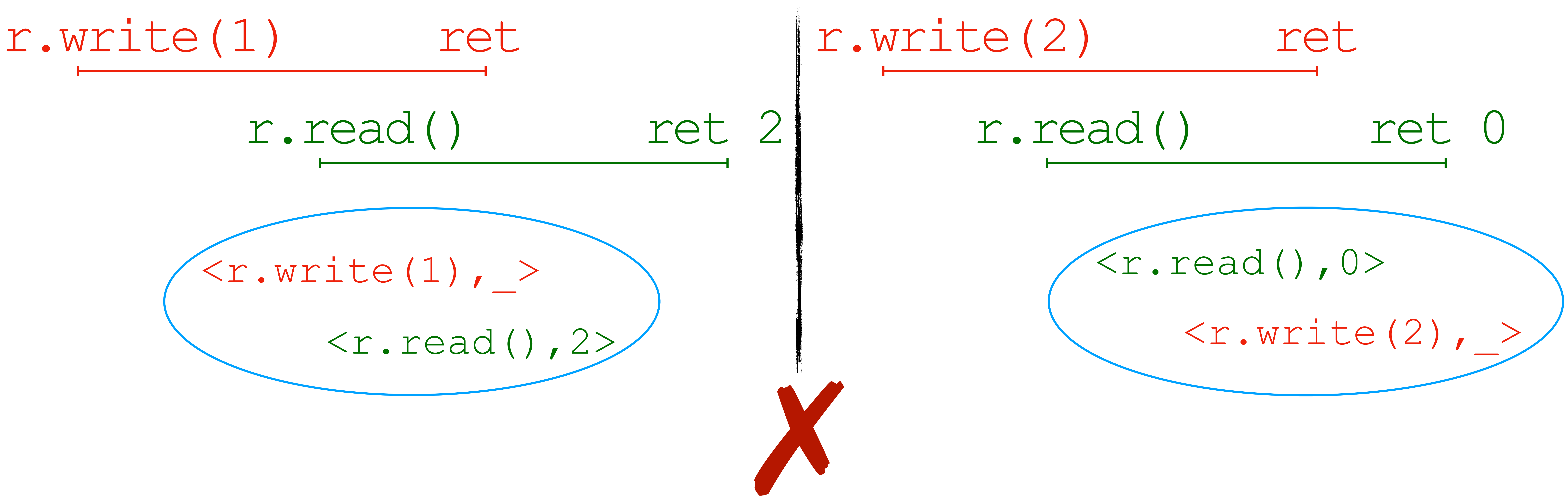
r.read() ret 2   r.read() ret 0

<r.write(1),_>
<r.read(),0>
<r.write(2),_> <r.read(),2>

<r.read(),0> <r.write(1),_> <r.write(2),_> <r.read(),2>

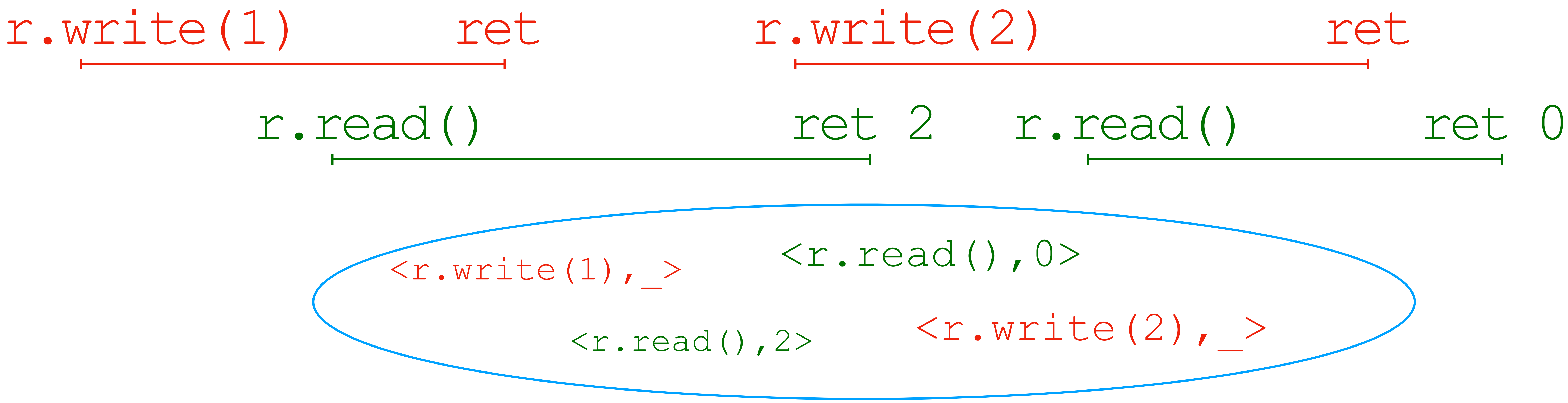
Quiescent Consistency

- Method calls should appear to happen one-at-a-time, sequential order
- Method calls separated by a period of quiescence should appear to take effect in their real time order



Quiescent Consistency

- Method calls should appear to happen one-at-a-time, sequential order
- Method calls separated by a period of quiescence should appear to take effect in their real time order



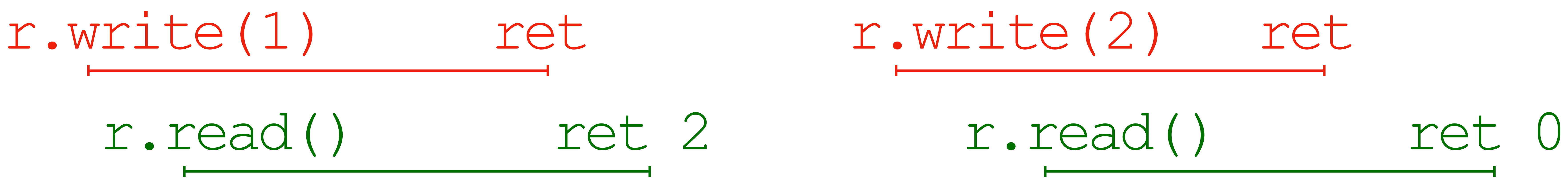
<r.read(),0> <r.write(1),_> <r.write(2),_> <r.read(),2>

Sequential Consistency

- *How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Computer Programs* [Lamport'79]
  - Each process issues operations in the order specified by its program.
  - Operations from all processors issued to a single object are serviced from a single FIFO queue. Issuing an operation consists in entering a request on this queue.

Sequential Consistency

```
r.write(1); ║ r.read();
r.write(2); ║ r.read();
```



<r.write(1),_> → <r.write(2),_>

<r.read(),2> → <r.read(),0>

✗

Sequential Consistency

```
r.write(1); ‖ r.read();
r.write(2); ‖ r.read();
```

r.write(1) ret   r.write(2) ret

r.read() ret 0   r.read() ret 1

<r.write(1),_> → <r.write(2),_>

<r.read(),0> → <r.read(),1>

<r.read(),0> <r.write(1),_> <r.read(),1> <r.write(2),_>

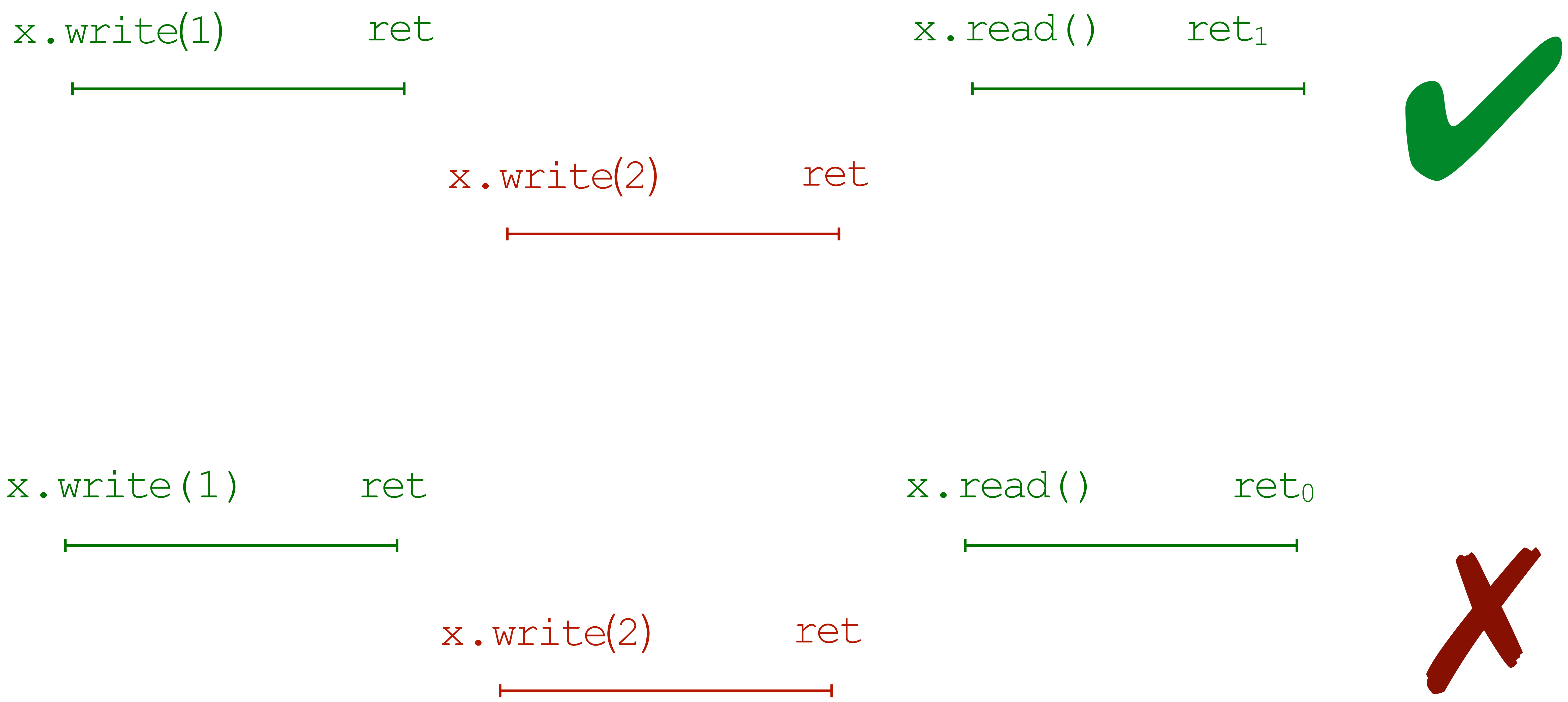
Sequential Consistency

- Quiescent Consistency +
- Method calls should appear to take effect in Program Order

program order ↓			program order ↓
	`r.write(1);`	`r.read();`	
	`r.write(2);`	`r.read();`	

- Each history δ induces a per-thread total order of operations
  - $o_1 <_\delta o_2$ iff o_1 and o_2 are on the same thread, and o_1 occurs before o_2 in δ
- A history δ is Sequentially Consistent if there exists an equivalent *Sequential* history δ' (i.e. same operations), and
  - $o1 <_\delta o2$ implies $o1 <_{\delta'} o2$

Sequential Consistency



Linearizability

- Same conditions as Sequential Consistency +
- Each method call should appear to take effect instantaneously at some moment between its invocation (call) and response (return)
- That is: we can pretend that the execution of each method is uninterrupted by other calls to the object
- De-facto standard for Concurrent Object Correctness (eg. `java.util.concurrent`)

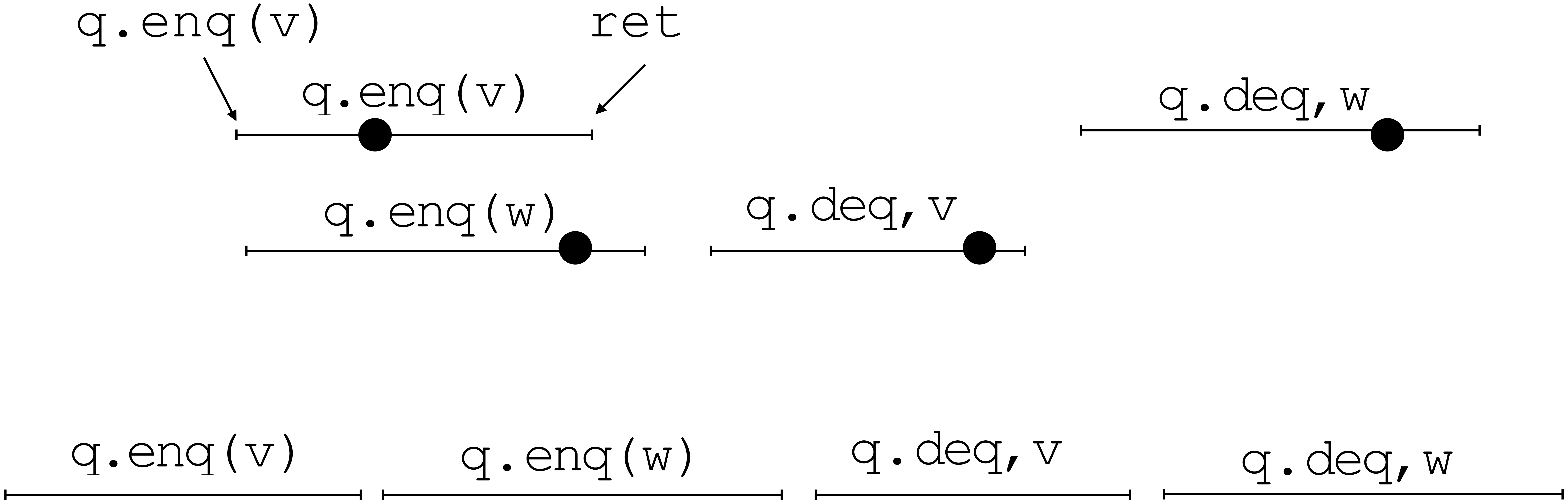
Linearizability: A Correctness Condition for Concurrent Objects
[Herlihy and Wing '90]

Linearizability

- Each history δ induces a partial order on operations such that
  - $o_1 \sqsubset_\delta o_2$ iff `ret` o_1 occurs before `call` o_2 in δ
- A history δ is Linearizable if there exists an equivalent *Sequential* history δ' (i.e. same operations), and
  - $o_1 \sqsubset_\delta o_2$ implies $o_1 \sqsubset_{\delta'} o_2$
- Ignoring uncompleted operations
- Strictly stronger than Sequential Consistency

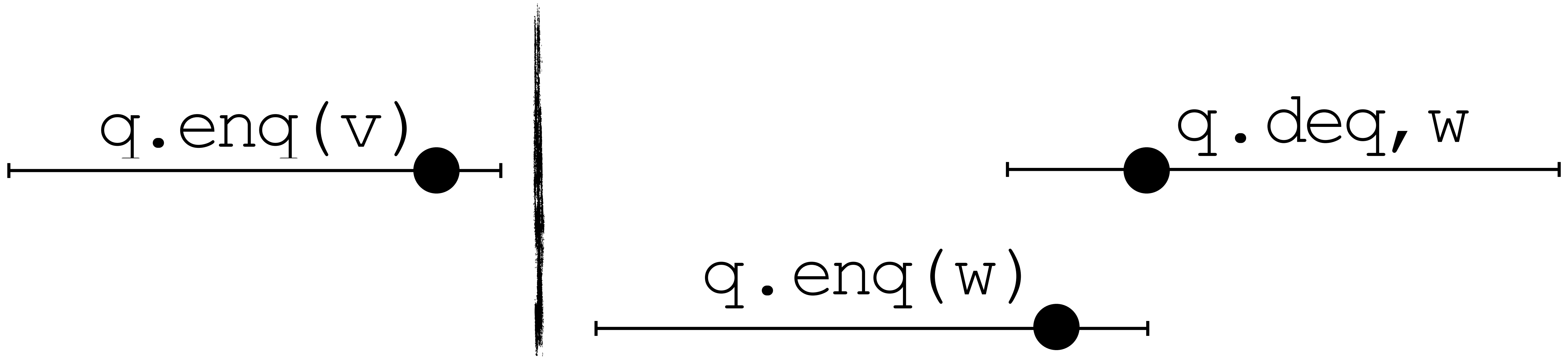
Linearizability

- Each operation takes place atomically within its call/return



Linearizability

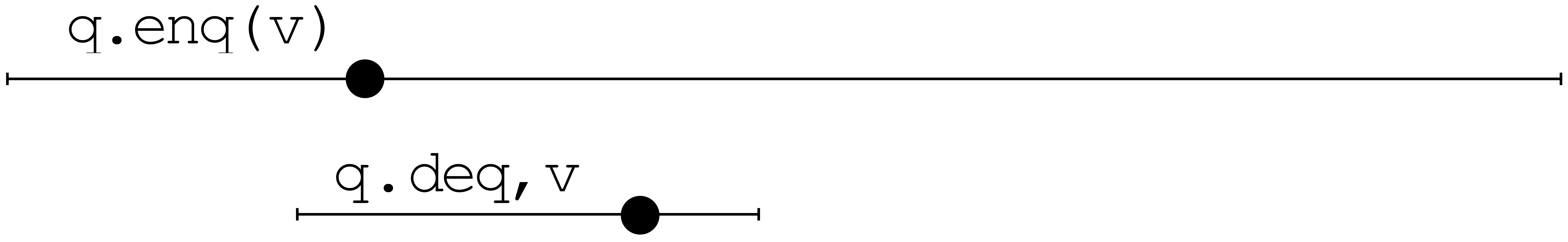
- Each operation takes place atomically within its call/return



Not Linearizable

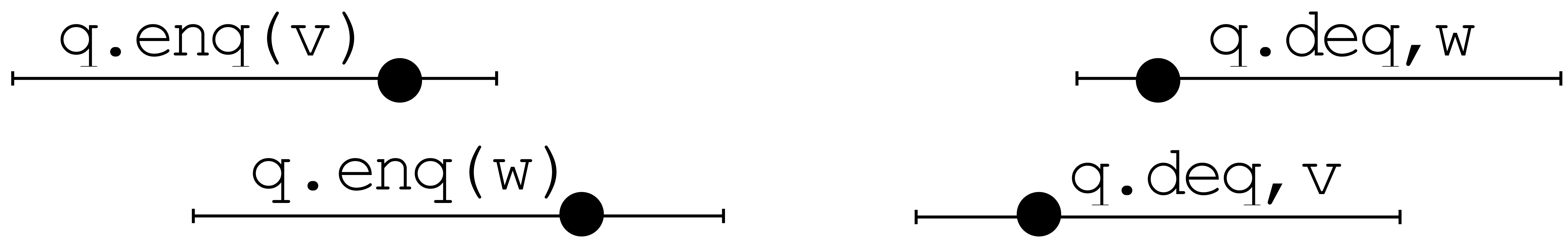
Linearizability

- Each operation takes place atomically within its call/return

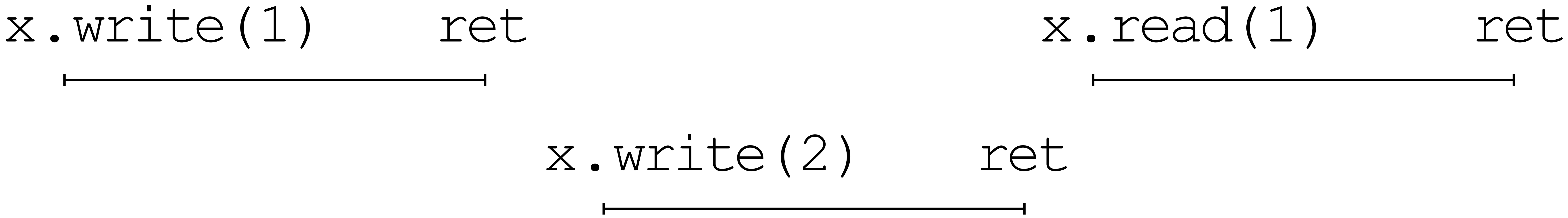


Linearizability

- Each operation takes place atomically within its call/return



Linearizability vs. Sequential Consistency



Not linearizable to begin with!

Observational Refinement

- Linearizability => observational refinement

Reference implementation

```
class AtomicStack {
  cell* top;
  Lock l;

  void push (int v) {
    l.lock();
    top->next = malloc(sizeof *x);
    top = top->next;
    top->data = v;
    l.unlock();
  }

  int pop () {
    ...
  }
}
```

minimize contention

Efficient implementation

```
class TreiberStack {
  cell* top;

  void push (int v) {
    cell* t;
    cell* x = malloc(sizeof *x);
    x->data = v;
    do {
      t = top;
      x->next = top;
    } while (!CAS(&top,t,x));
  }

  int pop () {
    ...
  }
}
```

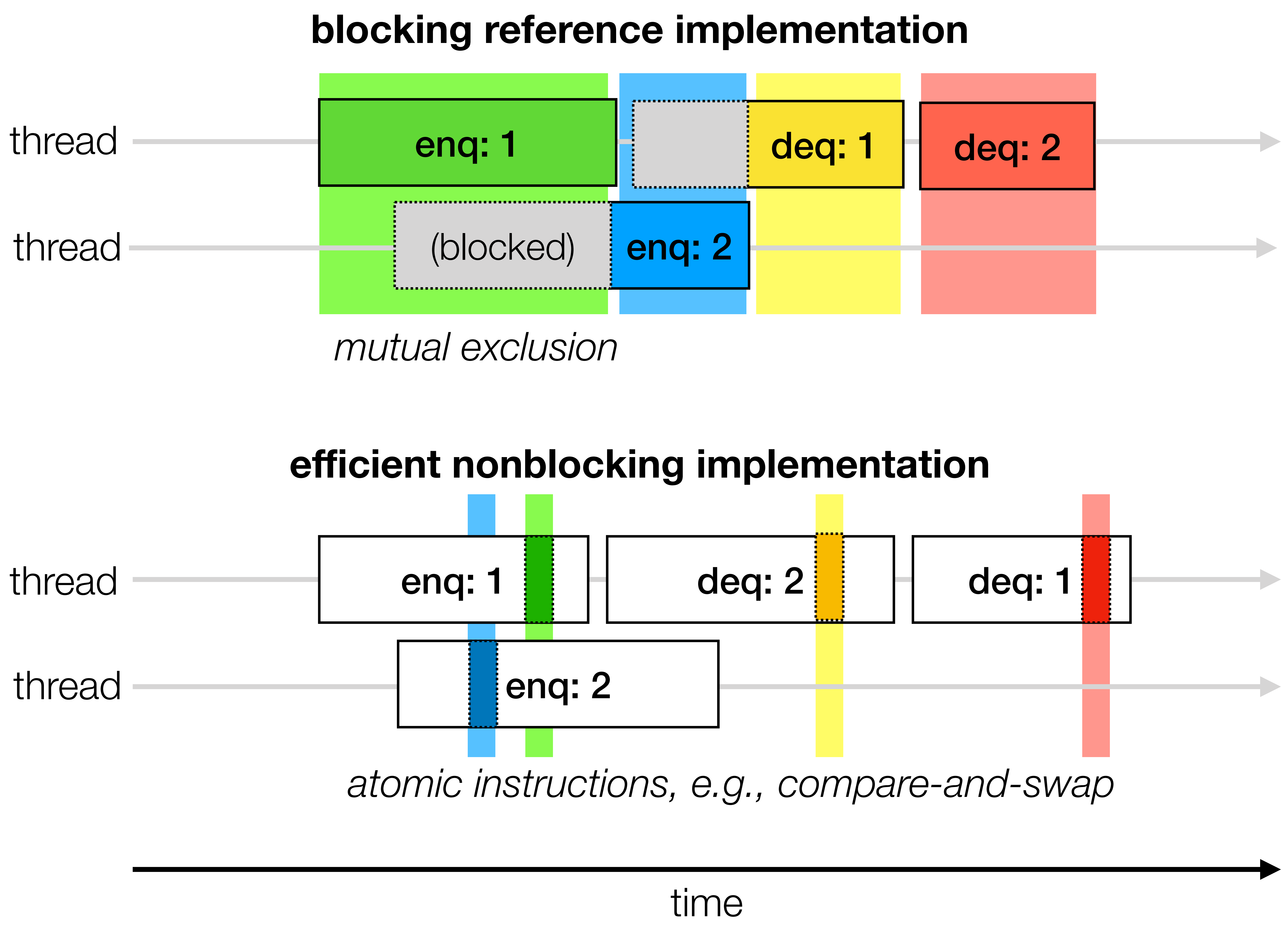
For every Client,
Client x Impl included in Client x Spec

Linearizability: Compositionallity

- Theorem: A history δ is linearizable if and only if for each object o in δ, δ_o is linearizable
  Proof: Simple induction on the number of operations appearing in δ
- Corollary: It is enough to show that each Library is linearizable to know that the system is

Some Object Implementations

Lock-Free Implementations



“Basic” Objects

Spin Lock

```
int Lock = 0;
TID owner = null;

void lock(){
  bool l;
  do {
    while(Lock == 1);
    l = cas(Lock, 0, 1);
  until (l);
  owner = getTID();
  return;
}
```

```
void unlock(){
  owner = null;
  Lock = 0;
  return;
}
```

Counter

```
class IntPtr {
  int val;
}
IntPtr COU;
```

```
void inc(int v){
  int n;
  while(true) {
   n = COU->val;
   if (cas(COU->val, n, n+v))
     break;
  }
  return;
}
```

```
void dec(int v) {
  int n;
  while(true) {
    n = COU->val;
    if (cas(COU-val, n, n-v))
      break;
  }
  return;
}
```

```
int read() {
  return COU->val;
}
```

Stack Implementations

Treiber Stack

```
class Node {
  Node tl;
  int val;
}
```

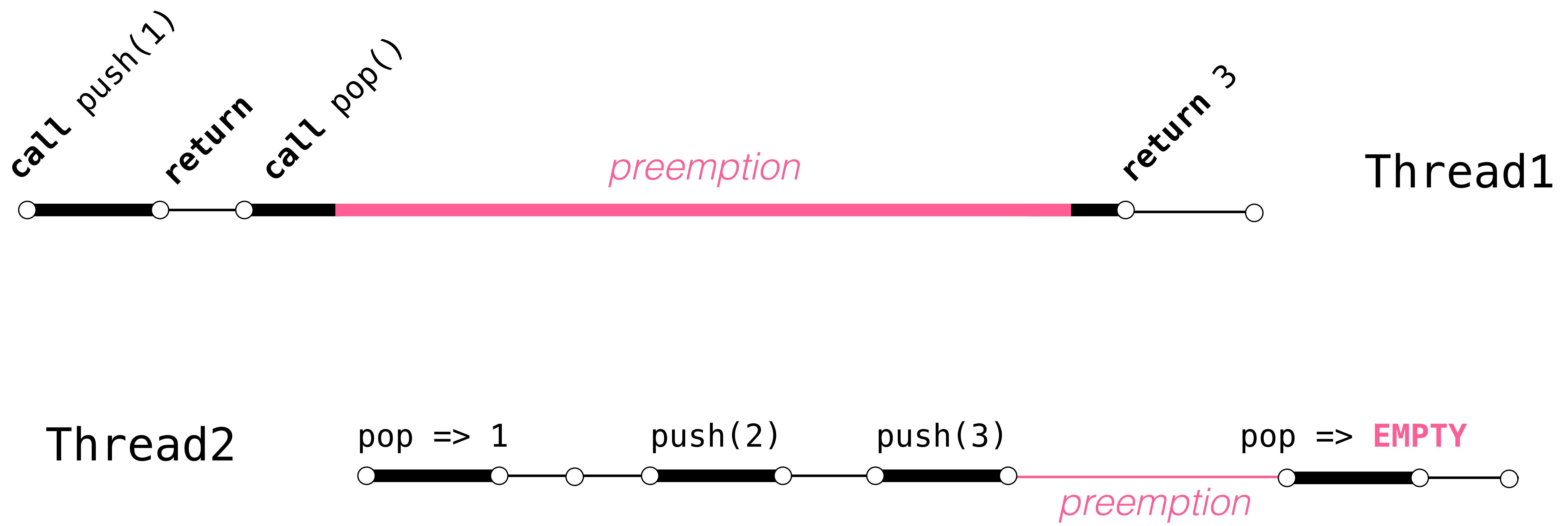
```
class NodePtr {
  Node val;
} TOP;
```

```
void push(int e) {
  Node y, n;
  y = new();
  y->val = e;
  while(true) {
    n = TOP->val;
    y->tl = n;
    if (cas(TOP->val, n, y))
      break;
  }
}
```

```
int pop() {
  Node y,z;
  while(true) {
    y = TOP->val;
    if (y==0) return EMPTY;
    z = y->tl;
    if (cas(TOP->val, y, z))
      break;
  }
  return y->val;
}
```


[Treiber'86]

Treiber Stack (ABA bug)



pushed: `1, 2, 3`
popped: `1, 3, EMPTY`

PROBLEM
not admitted by atomic stack

HSY Elimination Stack

Extremely simplified version: 1 collision

```
class Node {
  Node tl;
  int val;
}

class NodePtr {
  Node val;
} TOP;

class TidPtr {
  int val;
} clash;
```

```
void push(int e) {
  Node y, n;
  TID hisId;
  y = new();
  y->val = e;

  while (true) {
    n = TOP->val;
    y->tl = n;
    if (cas(TOP->val, n, y))
      return;
    //elimination scheme
   TidPtr t = new TidPrt();
   t->val = e;
    if (cas(clash,null,t)){
      wait(DELAY);
      //not eliminated
      if (cas(clash,t,null))
          continue;
      else break; //eliminated
    }
  }
}
```

```
int pop() {
  Node y,z;
  int t;
  TID hisId;
  while (true) {
    y = TOP->val;
    if (y == 0)
      return EMPTY;
    z = y->tl;
    t = y->val;
    if (cas(TOP->val, y, z)
      return t;
    //elimination scheme
    pusher = clash;
    while (pusher!=null){
     if (cas(clash,pusher,null)
      //eliminated push
      return pusher->val;
  }
}
```

[Hendler et al.'04]

Queue Implementations

Two Locks Queue

```
class Node {
  int val;
  Node tl;
}

class Queue {
  Node head;
  Node tail;
  thread_id hlock;
  thread_id tlock;
} Q;
```

```
void enqueue(int v) {
  Node n, t;
  n = new();
  n->val = v;
  n->tl = NULL;
  lock (&Q->tlock);
  temp = Q->tail;
  temp->tl = node;
  Q->tail = node;
  unlock (&Q->tlock);
}
```

```
int dequeue() {
  Node n, new_h;
  int v;
  lock (&Q->hlock);
  n = Q->head;
  new_h = n->tl;
  if (new_h == NULL) {
    unlock (&Q->hlock);
    return EMPTY;
  } else {
    value = new_h->val;
    Q->head = new_h;
    unlock (&Q->hlock);
    //dispose(n);
    return v;
  }
}
```

Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms

[Michael,Scott'96]

Michael and Scott Queue

```
class Node {
  int val;
  Node tl;
}

class Queue {
  Node head;
  Node tail;
} Q;
```

```
void enqueue(int v) {
  Node nd, nxt, tl;
  int b1;
  nd = new();
  nd->val = v;
  nd->tl = NULL;
  while(true) {
    tl = Q->tail;
    nxt = tl->tl
    if (Q->tail == tl) b1 = 1;
    else b1 = 0;
    if (b1!=0)
      if (nxt == 0)
        if (cas(tl->tl,nxt,nd))
          break;
      else cas(Q->tail,tl, nxt);
  }
  cas(Q->tail, tl, nd);
}
```

```
int dequeue() {
  Node nxt, hd, tl;
  int pval;
  while(true) {
    hd = Q->head;
    tl = Q->tail;
    nxt = hd->tl;
    if (Q->head != hd) continue;
    if (hd == tl) {
      if (nxt == NULL)
        return EMPTY;
      cas(Q->tail, tl, nxt);
    } else {
      pval = next->val;
      if (cas(Q->head, hd, nxt))
        return pval;
    }
  }
}
```

[Michael,Scott'96]

Herlihy Wing Queue

```
class Node {
  int val; // -1 NAN
  Node tl;
  thread_id alloc;
}

class Queue {
  Node head;
  Node tail;
} Q;
```

```
void enqueue(int value) {
  Node nd, tl;
  nd = new();
  nd->alloc = TID;
  nd->val = -1;
  nd->tl = NULL;
  atomic {
    tl = Q->tail;
    tl->tl = nd;
    Q->tail = nd;
  } // end of slot reservation;
  nd->val = value;//value written;
}
```

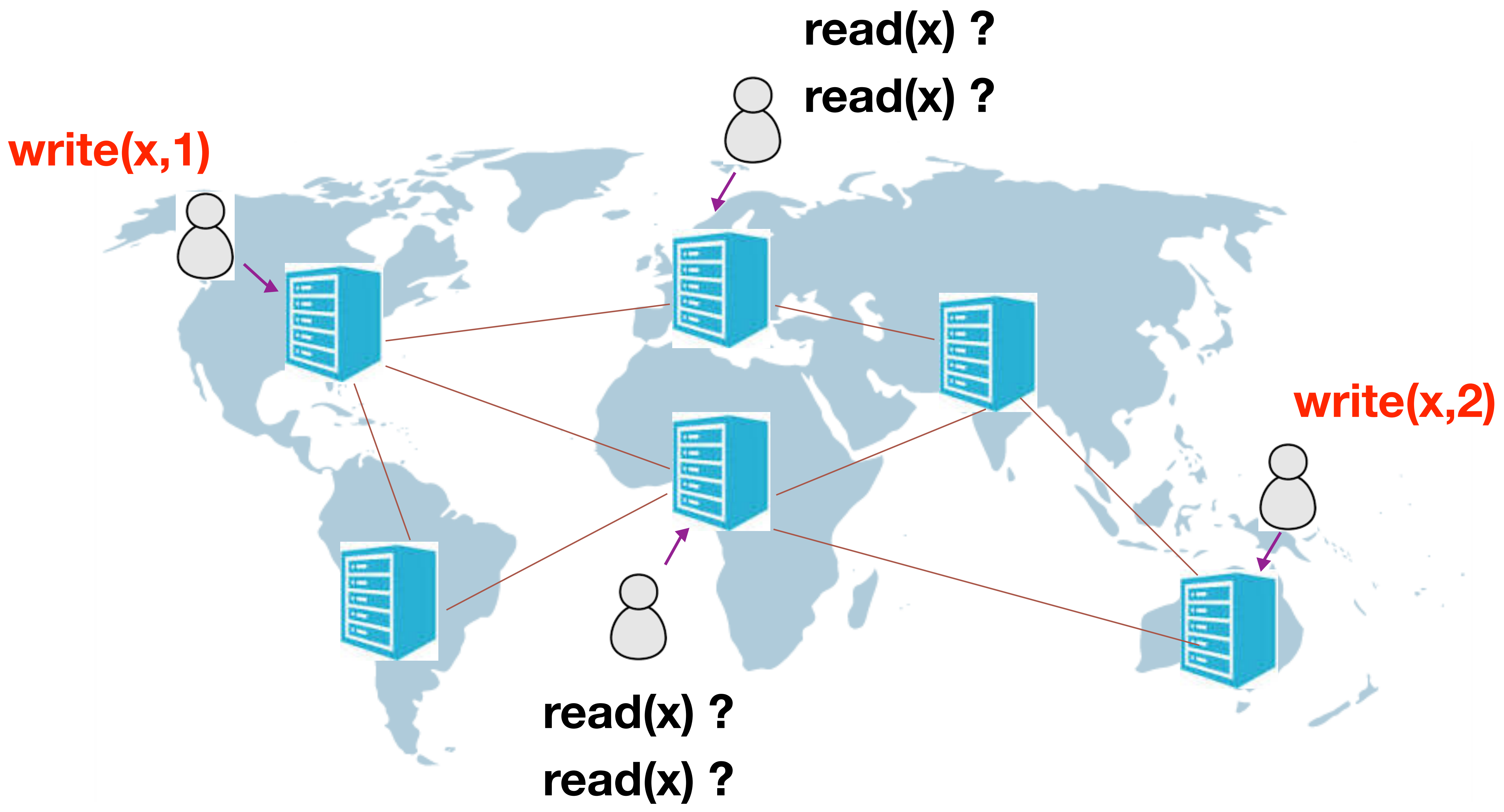
```
int dequeue() {
  Node curr, tail;
  int pval;
  while (true) {
    curr = Q->head;
    tail = Q->tail;
    while (curr != tail) {
      atomic { //atomic swap
        pval = curr->val;
        curr->val = -1;
        if (pval != -1)
          return pval;
        curr = curr->tl;
      }
    }
  }
}
```

Linearizability: A Correctness Condition for Concurrent Objects
[Herlihy and Wing ‘90]

Set Implementations (TAMP slides)

Replicated objects

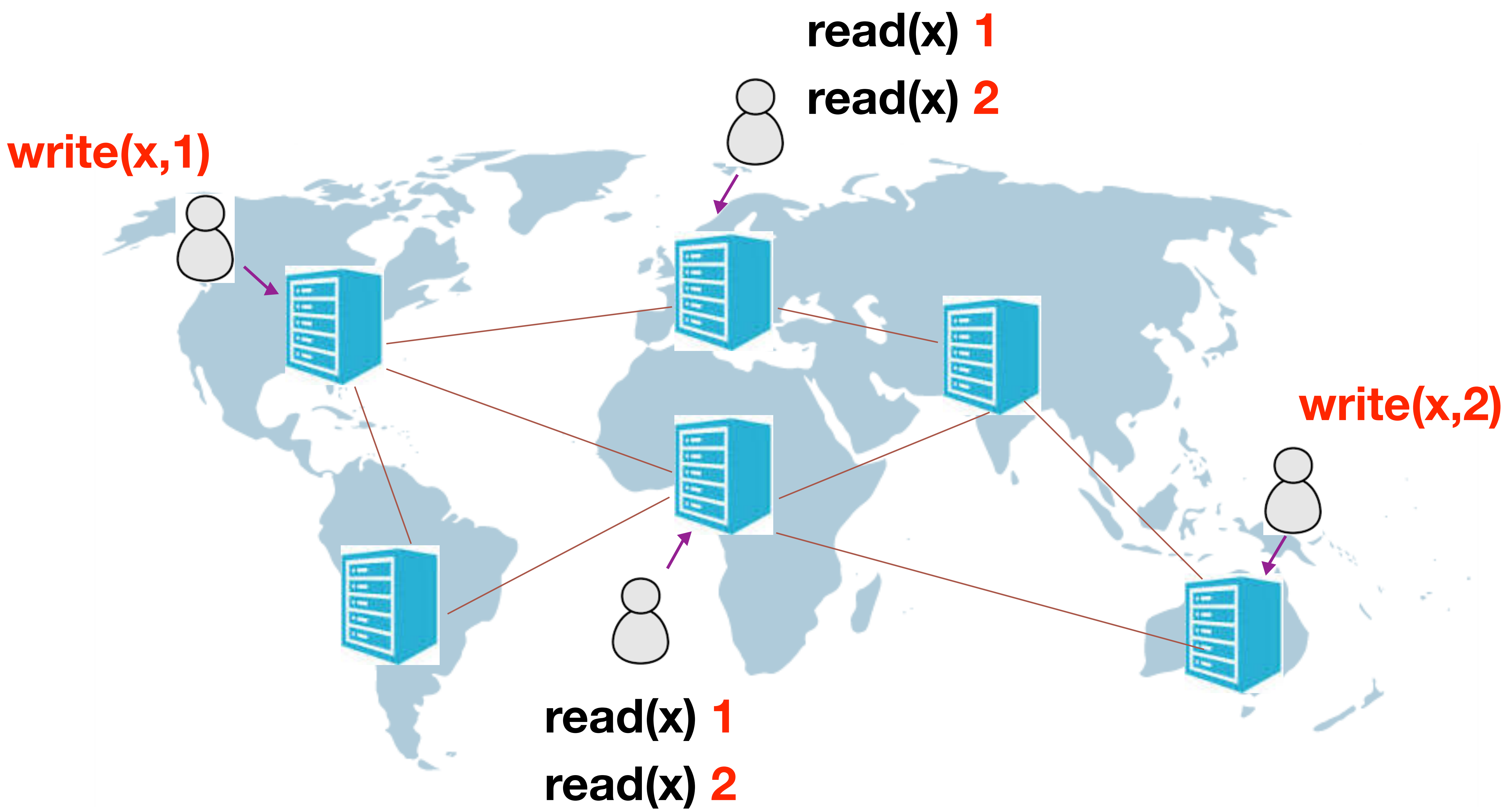
Distributed systems



Conflicting concurrent updates: how are they observed on different replicas ?

Adversarial environments: crashes, network partitions

Pessimistic Replication

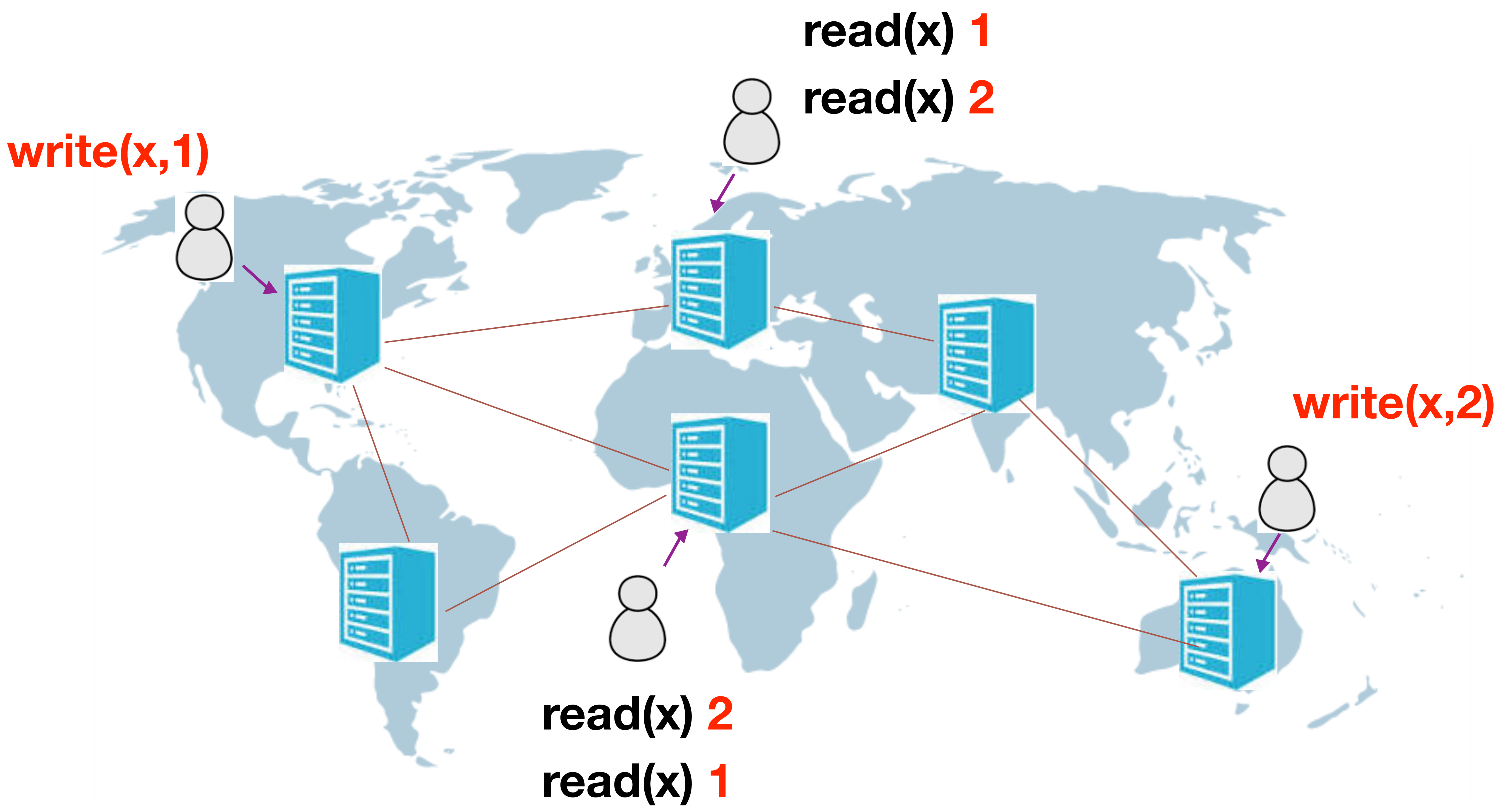


Using consensus algorithms to agree on an order between conflicting concurrent updates

✔ strong consistency ✘ availability

CAP theorem [Gilbert et al.'02]: strong cons. + availability + partition tolerance is impossible

Optimistic Replication



Each update is applied on the **local** replica and propagated **asynchronously** to other replicas

✗ strong consistency ✓ availability

Replicas may store different versions of data: **weak consistency**