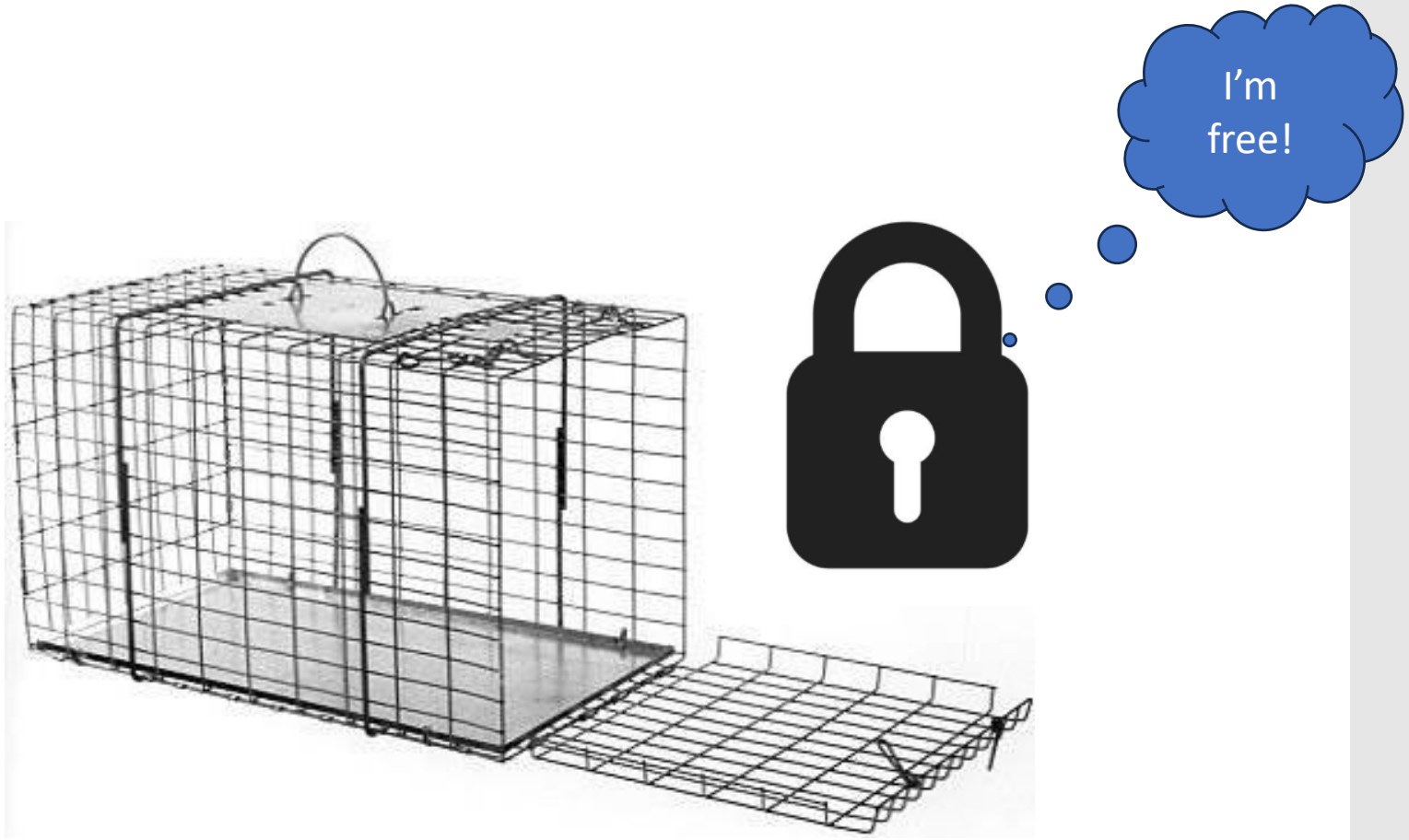
Shared Memory Synchronization Rides Again: Lock Freedom

cs378h

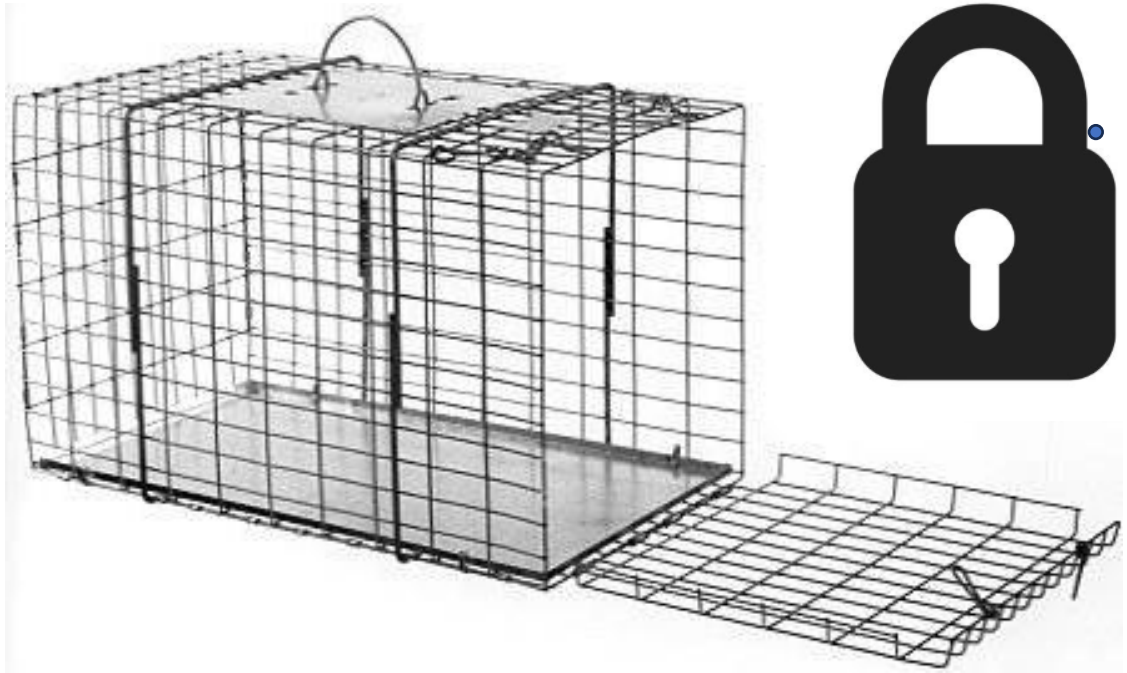
Shared Memory Synchronization Rides Again: Lock Freedom

cs378h



Shared Memory Synchronization Rides Again: Lock Freedom

cs378h



Today

Questions?

Administrivia

- Faux Quiz

Agenda:

- Lock Freedom

Faux Quiz Questions: 5 min, pick any 2

- What is obstruction freedom, wait freedom, lock freedom?
- How can one compose lock free data structures?
- What is the difference between linearizability and strong consistency? Between linearizability and serializability?
- What is the ABA problem? Give an example.
- How do lock-free data structures deal with the “inconsistent view” problem?

Non-Blocking Synchronization

Non-Blocking Synchronization

Locks: a litany of problems

Non-Blocking Synchronization

Locks: a litany of problems

- Deadlock

Non-Blocking Synchronization

Locks: a litany of problems

- Deadlock
- Priority inversion

Non-Blocking Synchronization

Locks: a litany of problems

- Deadlock
- Priority inversion
- Convoys

Non-Blocking Synchronization

Locks: a litany of problems

- Deadlock
- Priority inversion
- Convoys
- Fault Isolation

Non-Blocking Synchronization

Locks: a litany of problems

- Deadlock
- Priority inversion
- Convoys
- Fault Isolation
- Preemption Tolerance

Non-Blocking Synchronization

Locks: a litany of problems

- Deadlock
- Priority inversion
- Convoys
- Fault Isolation
- Preemption Tolerance
- Performance

Non-Blocking Synchronization

Locks: a litany of problems

- Deadlock
- Priority inversion
- Convoys
- Fault Isolation
- Preemption Tolerance
- Performance

Solution: don’t use locks

Non-Blocking Synchronization

Locks: a litany of problems

- Deadlock
- Priority inversion
- Convoys
- Fault Isolation
- Preemption Tolerance
- Performance

Lock-free programming

Lock-free programming

- Subset of a broader class: ***Non-blocking Synchronization***

Lock-free programming

- Subset of a broader class: ***Non-blocking Synchronization***
- Thread-safe access shared mutable state without mutual exclusion

Lock-free programming

- Subset of a broader class: ***Non-blocking Synchronization***
- Thread-safe access shared mutable state without mutual exclusion
- Possible without HW support
  - e.g. Lamport's Concurrent Buffer
  - …but not really practical wo HW

Lock-free programming

- Subset of a broader class: ***Non-blocking Synchronization***
- Thread-safe access shared mutable state without mutual exclusion
- Possible without HW support
  - e.g. Lamport's Concurrent Buffer
  - …but not really practical wo HW
- Built on atomic instructions like CAS + clever algorithmic tricks

Lock-free programming

- Subset of a broader class: ***Non-blocking Synchronization***
- Thread-safe access shared mutable state without mutual exclusion
- Possible without HW support
  - e.g. Lamport's Concurrent Buffer
  - …but not really practical wo HW
- Built on atomic instructions like CAS + clever algorithmic tricks
- Lock-free *algorithms* are hard, so

Lock-free programming

- Subset of a broader class: ***Non-blocking Synchronization***
- Thread-safe access shared mutable state without mutual exclusion
- Possible without HW support
  - e.g. Lamport's Concurrent Buffer
  - …but not really practical wo HW
- Built on atomic instructions like CAS + clever algorithmic tricks
- Lock-free *algorithms* are hard, so
- General approach: encapsulate lock-free algorithms in data structures
  - Queue, list, hash-table, skip list, etc.
  - New LF data structure → research result

Basic List Append

Basic List Append

```
struct Node
{
  int data;
  struct Node *next;
};
```

Basic List Append

```
struct Node
{
  int data;
  struct Node *next;
};
```

```
void append(Node** head_ref, int new_data) {
    Node* new_node = mknode(new_data, head_ref);
    if (*head_ref == NULL) {
        *head_ref = new_node;
        return;
    }
    while (last->next != NULL)
        last = last->next;
    last->next = new_node;
}
```

Basic List Append

```
struct Node
{
  int data;
  struct Node *next;
};
```

```
void append(Node** head_ref, int new_data) {
    Node* new_node = mknode(new_data, head_ref);
    if (*head_ref == NULL) {
        *head_ref = new_node;
        return;
    }
    while (last->next != NULL)
        last = last->next;
    last->next = new_node;
}
```

- Is this thread safe?

Basic List Append

```
struct Node
{
  int data;
  struct Node *next;
};
```

```
void append(Node** head_ref, int new_data) {
    Node* new_node = mknode(new_data, head_ref);
    if (*head_ref == NULL) {
        *head_ref = new_node;
        return;
    }
    while (last->next != NULL)
        last = last->next;
    last->next = new_node;
}
```

- Is this thread safe?
- What can go wrong?

Example: List Append

```
struct Node
{
   int data;
   struct Node *next;
};
```

```
void append(Node** head_ref, int new_data) {
    Node* new_node = mknode(new_data, head_ref);
    lock();
    if (*head_ref == NULL) {
       *head_ref = new_node;
    } else {
       while (last->next != NULL)
          last = last->next;
       last->next = new_node;
    }
    unlock();
}
```

Example: List Append

```
struct Node
{
   int data;
   struct Node *next;
};
```

```
void append(Node** head_ref, int new_data) {
    Node* new_node = mknode(new_data, head_ref);
    lock();
    if (*head_ref == NULL) {
        *head_ref = new_node;
    } else {
        while (last->next != NULL)
            last = last->next;
        last->next = new_node;
    }
    unlock();
}
```

Example: List Append

```
struct Node
{
   int data;
   struct Node *next;
};
```

```
void append(Node** head_ref, int new_data) {
    Node* new_node = mknode(new_data, head_ref);
    lock();
    if (*head_ref == NULL) {
        *head_ref = new_node;
    } else {
        while (last->next != NULL)
            last = last->next;
        last->next = new_node;
    }
    unlock();
}
```

Example: List Append

```
struct Node
{
   int data;
   struct Node *next;
};
```

```
void append(Node** head_ref, int new_data) {
    Node* new_node = mknode(new_data, head_ref);
    lock();
    if (*head_ref == NULL) {
        *head_ref = new_node;
    } else {
        while (last->next != NULL)
            last = last->next;
        last->next = new_node;
    }
    unlock();
}
```

- What property do the locks enforce?

Example: List Append

```
struct Node
{
   int data;
   struct Node *next;
};
```

```
void append(Node** head_ref, int new_data) {
    Node* new_node = mknode(new_data, head_ref);
    lock();
    if (*head_ref == NULL) {
       *head_ref = new_node;
    } else {
       while (last->next != NULL)
          last = last->next;
       last->next = new_node;
    }
    unlock();
}
```

- What property do the locks enforce?
- What does the mutual exclusion ensure?

Example: List Append

```
struct Node
{
   int data;
   struct Node *next;
};
```

```
void append(Node** head_ref, int new_data) {
    Node* new_node = mknode(new_data, head_ref);
    lock();
    if (*head_ref == NULL) {
       *head_ref = new_node;
    } else {
       while (last->next != NULL)
          last = last->next;
       last->next = new_node;
    }
    unlock();
}
```

- What property do the locks enforce?
- What does the mutual exclusion ensure?
- Can we ensure consistent view (invariants hold) sans mutual exclusion?

Example: List Append

```
struct Node
{
   int data;
   struct Node *next;
};
```

```
void append(Node** head_ref, int new_data) {
    Node* new_node = mknode(new_data, head_ref);
    lock();
    if (*head_ref == NULL) {
       *head_ref = new_node;
    } else {
       while (last->next != NULL)
          last = last->next;
       last->next = new_node;
    }
    unlock();
}
```

- What property do the locks enforce?
- What does the mutual exclusion ensure?
- Can we ensure consistent view (invariants hold) sans mutual exclusion?
- Key insight: allow inconsistent view and fix it up algorithmically

Example: List Append

```
void append(Node** head_ref, int new_data) {
    Node* new_node = mknode(new_data);
    new_node->next = NULL;
    while(TRUE) {
        Node * last = *head_ref;
        if(last == NULL) {
            if(cas(head_ref, new_node, NULL))
                break;
        }
        while(last->next != NULL)
            last = last->next;
        if(cas(&last->next, new_node, NULL))
            break;
    }
}
```

```
struct Node
    data;
uct Node *next;
```

e?

nsure?

- Can we ensure consistent view (invariants hold) sans mutual exclusion?
- Key insight: allow inconsistent view and fix it up algorithmically

Example: SP-SC Queue

```
next(x):
    if(x == Q_size-1) return 0;
    else return x+1;

Q_get(data):
    t = Q_tail;
    while(t == Q_head)
      ;
    data = Q_buf[t];
    Q_tail = next(t);
```

```
Q_put(data):
    h = Q_head;
    while(next(h) == Q_tail)
      ;
    Q_buf[h] = data;
    Q_head = next(h);
```

- Single-producer single-consumer
- Why/when does this work?

Example: SP-SC Queue

```
next(x):
    if(x == Q_size-1) return 0;
    else return x+1;

Q_get(data):
    t = Q_tail;
    while(t == Q_head)
      ;
    data = Q_buf[t];
    Q_tail = next(t);
```

```
Q_put(data):
    h = Q_head;
    while(next(h) == Q_tail)
      ;
    Q_buf[h] = data;
    Q_head = next(h);
```

- Single-producer single-consumer
- Why/when does this work?

1. Q_head is last write in Q_put, so Q_get never gets “ahead”.
2. *single* p,c only (as advertised)
3. Requires fence before setting Q head
4. Devil in the details of “wait”
5. No lock → “optimistic”

Lock-Free Stack

```
struct Node
{
  int data;
  struct Node *next;
};
```

```
void push(int t) {
    Node* node = new Node(t);
    do {
        node->next = head;
    } while (!cas(&head, node, node->next));
}

bool pop(int& t) {
   Node* current = head;
   while(current) {
       if(cas(&head, current->next, current)) {
           t = current->data;
           return true;
       }
       current = head;
       }
   return false;
}
```

Lock-Free Stack

```
struct Node
{
  int data;
  struct Node *next;
};
```

```
void push(int t) {
    Node* node = new Node(t);
    do {
        node->next = head;
    } while (!cas(&head, node, node->next));
}

bool pop(int& t) {
   Node* current = head;
   while(current) {
       if(cas(&head, current->next, current)) {
           t = current->data;
           return true;
       }
       current = head;
       }
   return false;
}
```

- Why does is it work?

Lock-Free Stack

```
struct Node
{
  int data;
  struct Node *next;
};
```

```
void push(int t) {
    Node* node = new Node(t);
    do {
        node->next = head;
    } while (!cas(&head, node, node->next));
}

bool pop(int& t) {
   Node* current = head;
   while(current) {
       if(cas(&head, current->next, current)) {
           t = current->data; // problem?
           return true;
       }
       current = head;
       }
   return false;
}
```

- Why does is it work?

Lock-Free Stack

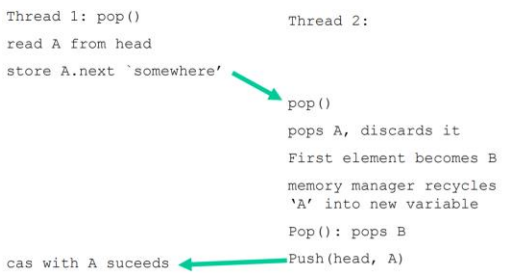
```
struct Node
{
  int data;
  struct Node *next;
};
```

```
void push(int t) {
    Node* node = new Node(t);
    do {
        node->next = head;
    } while (!cas(&head, node, node->next));
}

bool pop(int& t) {
    Node* current = head;
    while(current) {
        if(cas(&head, current->next, current)) {
            t = current->data; // problem?
            return true;
        }
        current = head;
        }
    return false;
}
```

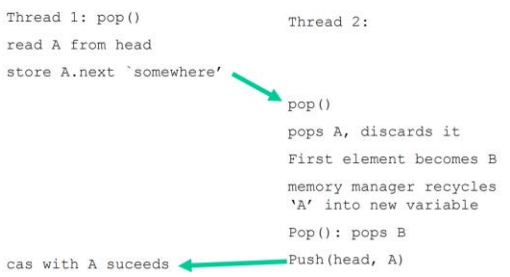
- Why does is it work?
- Does it enforce all invariants?

Lock-Free Stack: ABA Problem



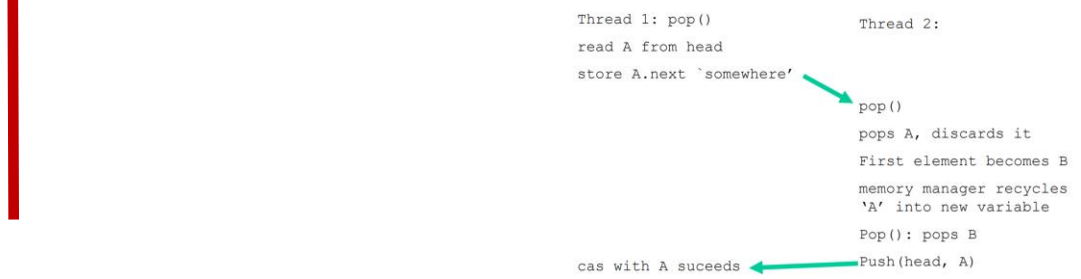
Lock-Free Stack: ABA Problem

```
Node* pop() {
    Node* current = head;
    while(current) {
        if(cas(&head, current->next, current))
            return current;
        current = head;
    }
    return false;
}
```



Lock-Free Stack: ABA Problem

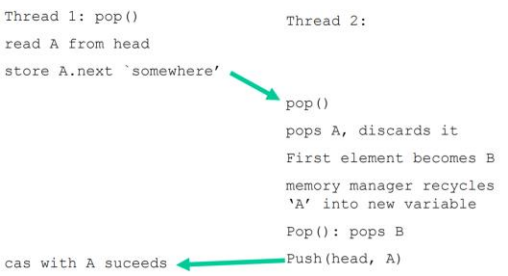
```
Node* pop() {
    Node* current = head;
    while(current) {
        if(cas(&head, current->next, current))
            return current;
        current = head;
    }
    return false;
}
```



Lock-Free Stack: ABA Problem

```
Node* pop() {
    Node* current = head;
    while(current) {
        if(cas(&head, current->next, current))
            return current;
        current = head;
    }
    return false;
}
```


```
Node* pop() {
    Node* current = head;
    while(current) {
```

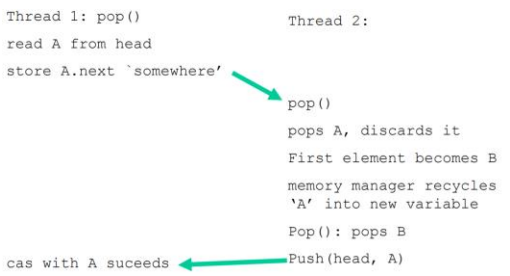



Lock-Free Stack: ABA Problem

```
Node* pop() {
    Node* current = head;
    while(current) {
        if(cas(&head, current->next, current))
            return current;
        current = head;
    }
    return false;
}
```

```
Node* pop() {
    Node* current = head;
    while(current) {
```

```
Node * node = pop();
delete node;
node = new Node(blah_blah);
push(node);
```

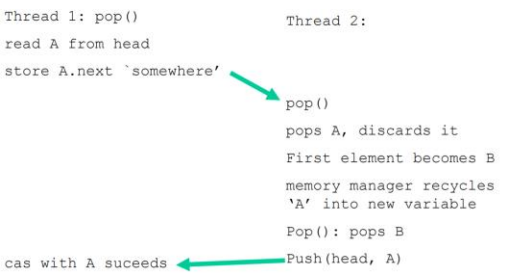


Lock-Free Stack: ABA Problem

```
Node* pop() {
    Node* current = head;
    while(current) {
        if(cas(&head, current->next, current))
            return current;
        current = head;
    }
    return false;
}
```


```
Node* pop() {
    Node* current = head;
    while(current) {
        if(cas(&head, current->next, current))
            return current;
        current = head;
    }
    return false;
}
```


```
Node * node = pop();
delete node;
node = new Node(blah_blah);
push(node);
```

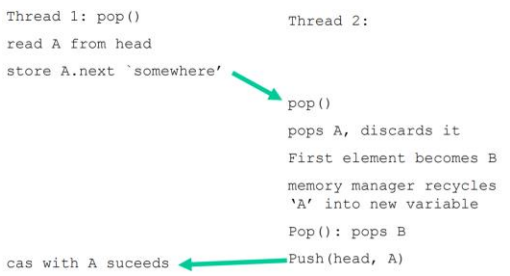


Lock-Free Stack: ABA Problem

```
Node* pop() {
    Node* current = head;
    while(current) {
        if(cas(&head, current->next, current))
            return current;
        current = head;
    }
    return false;
}
```

```
Node* pop() {
    Node* current = head;
    while(current) {
        if(cas(&head, current->next, current))
            return current;
        current = head;
    }
    return false;
}
```

```
Node * node = pop();
delete node;
node = new Node(blah_blah);
push(node);
```

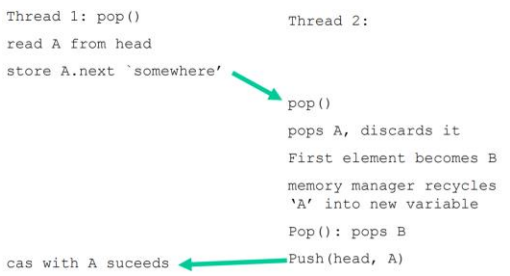


Lock-Free Stack: ABA Problem

```
Node* pop() {
    Node* current = head;
    while(current) {
        if(cas(&head, current->next, current))
            return current;
        current = head;
    }
    return false;
}
```

```
Node* pop() {
    Node* current = head;
    while(current) {
        if(cas(&head, current->next, current))
            return current;
        current = head;
    }
    return false;
}
```

```
Node * node = pop();
delete node;
node = new Node(blah_blah);
push(node);
```



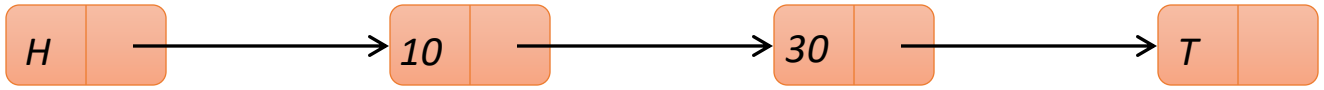
ABA Problem

- Thread 1 observes shared variable → ‘A’
- Thread 1 calculates using that value
- Thread 2 changes variable to B
  - if Thread 1 wakes up now and tries to CAS, CAS fails and Thread 1 retries
- Instead, Thread 2 changes variable back to A!
  - CAS succeeds despite mutated state
  - Very bad if the variables are pointers

- Keep update count → DCAS
- Avoid re-using memory
- Multi-CAS support → HTM

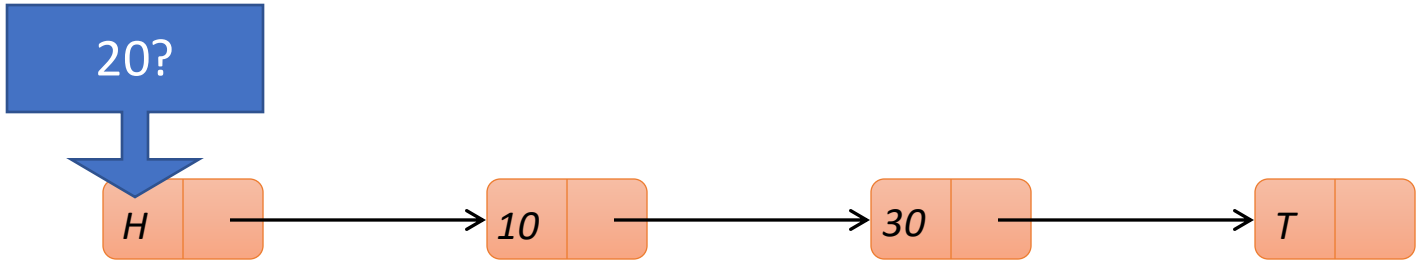
Correctness: Searching a sorted list

- find(20):



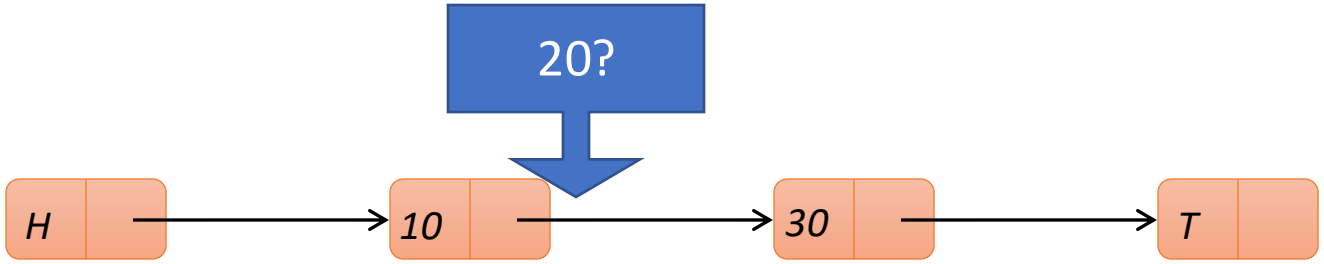
Correctness: Searching a sorted list

- find(20):



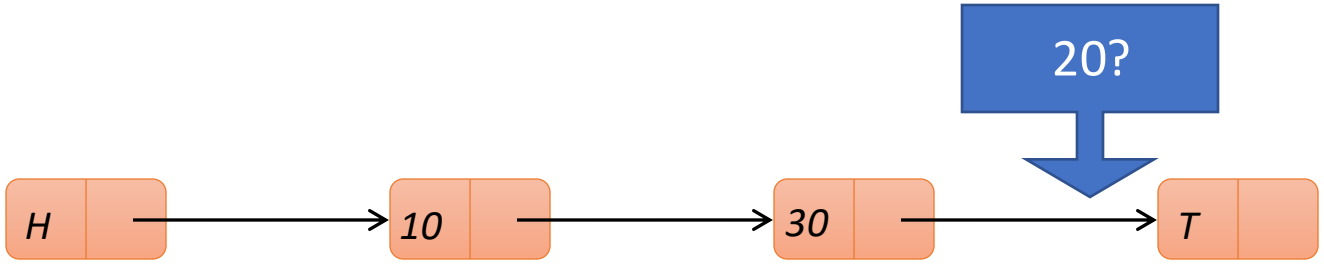
Correctness: Searching a sorted list

- find(20):



Correctness: Searching a sorted list

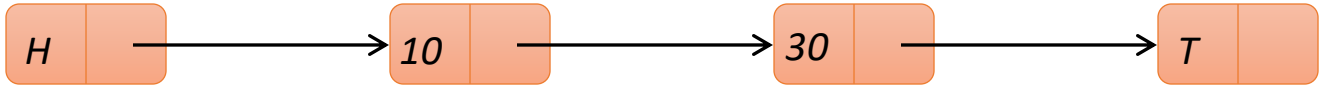
- find(20):



find(20) -> false

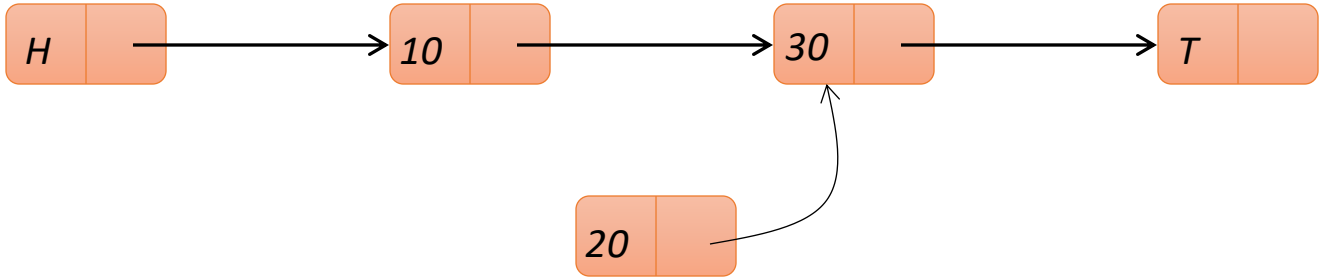
Inserting an item with CAS

- insert(20):



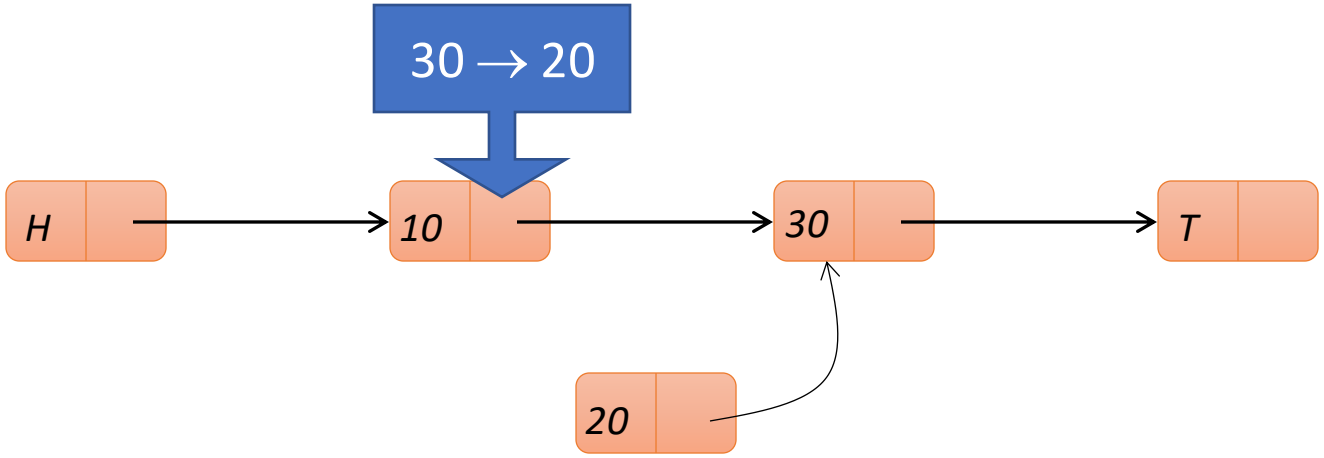
Inserting an item with CAS

- insert(20):



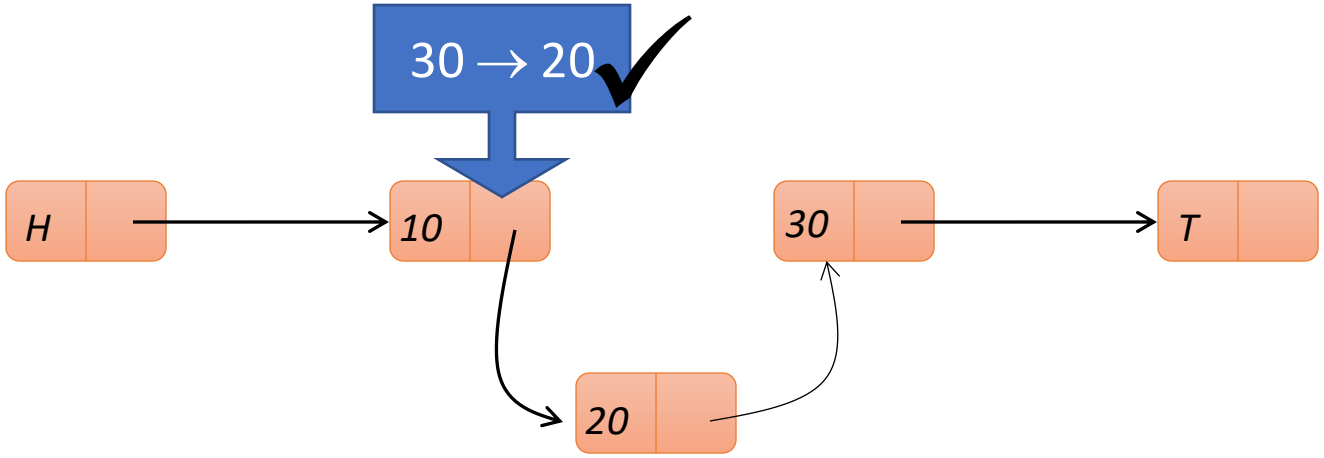
Inserting an item with CAS

- insert(20):



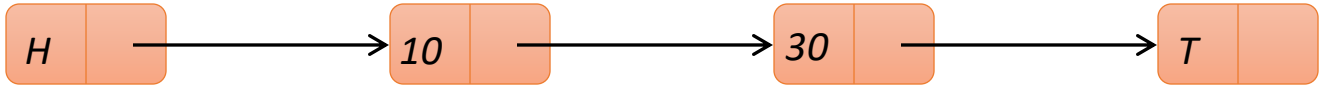
Inserting an item with CAS

- insert(20):



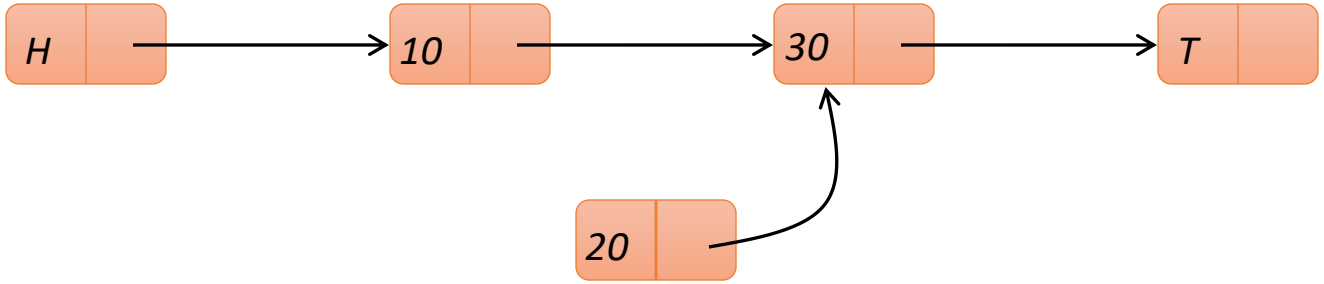
insert(20) -> true

Inserting an item with CAS



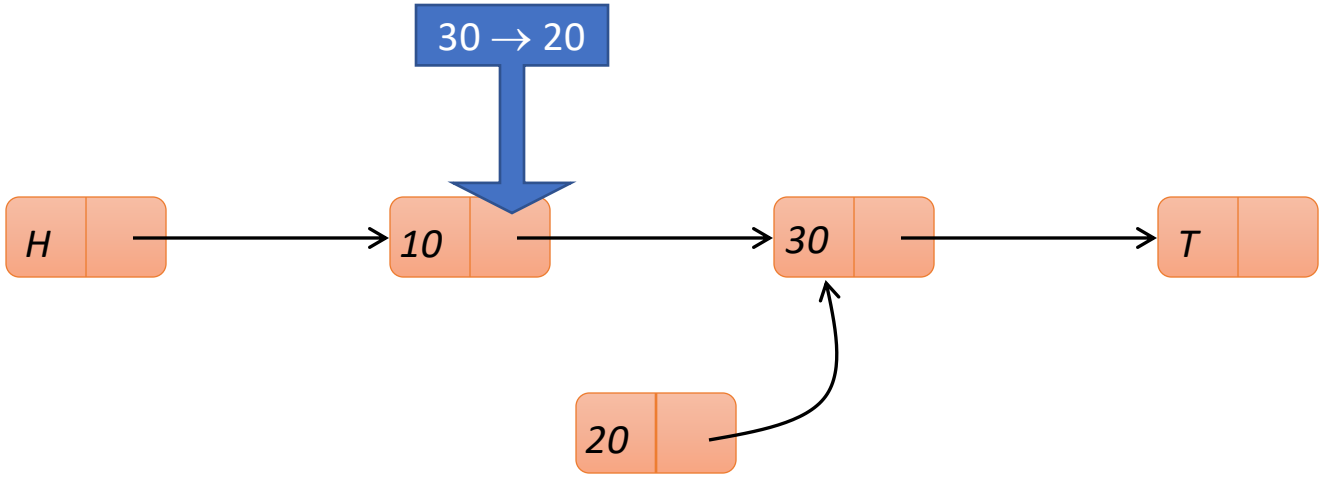
Inserting an item with CAS

- insert(20):



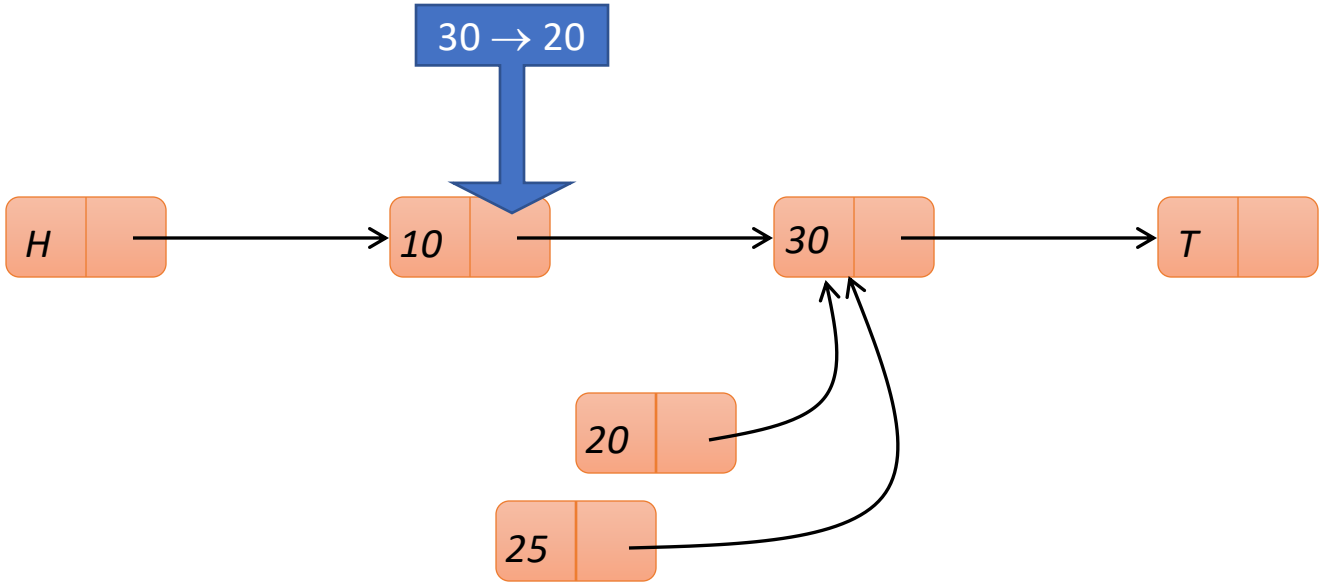
Inserting an item with CAS

- insert(20):



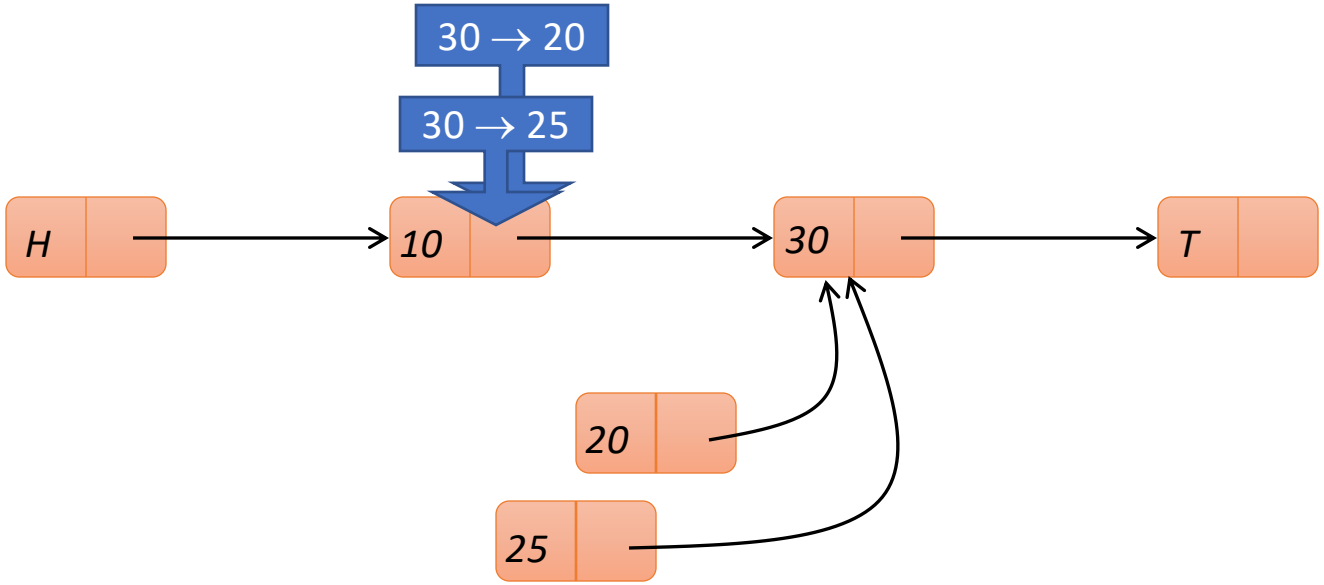
Inserting an item with CAS

- insert(20):
- insert(25):



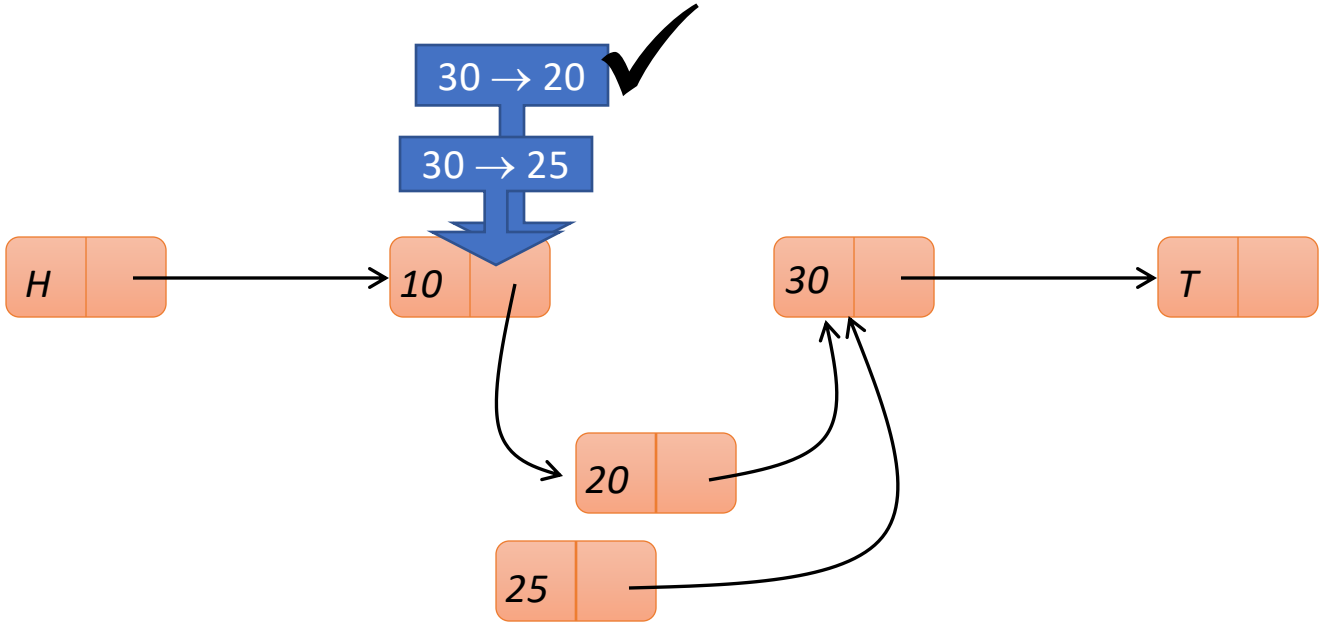
Inserting an item with CAS

- insert(20):
- insert(25):



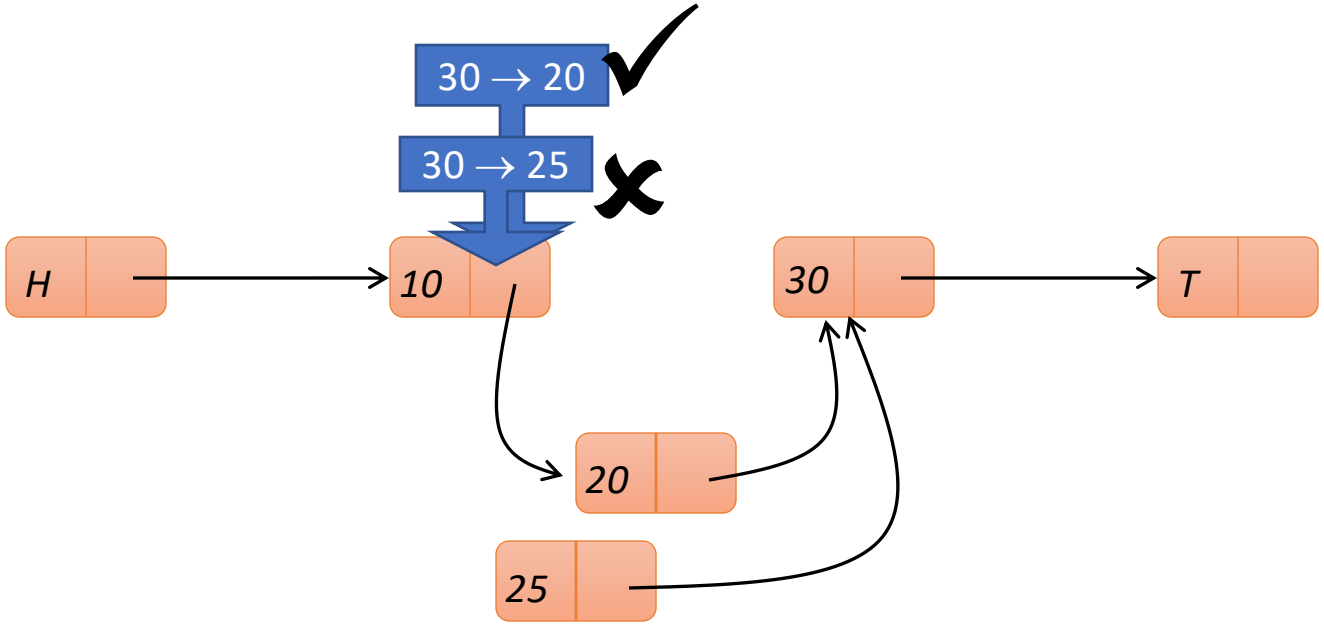
Inserting an item with CAS

- insert(20):
- insert(25):



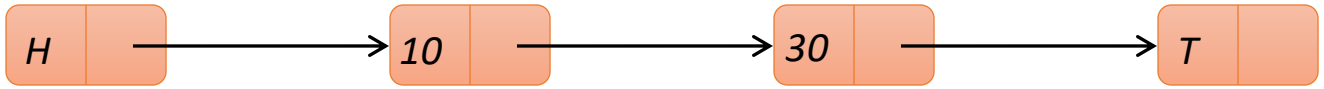
Inserting an item with CAS

- insert(20):
- insert(25):



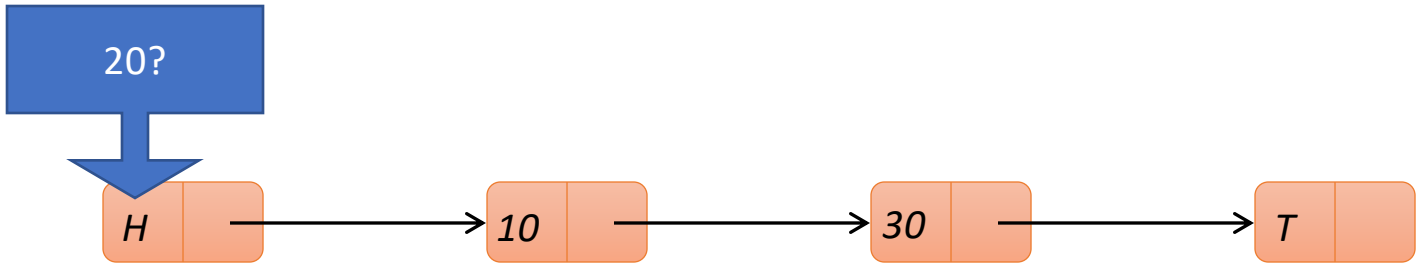
Searching and inserting together

- find(20)



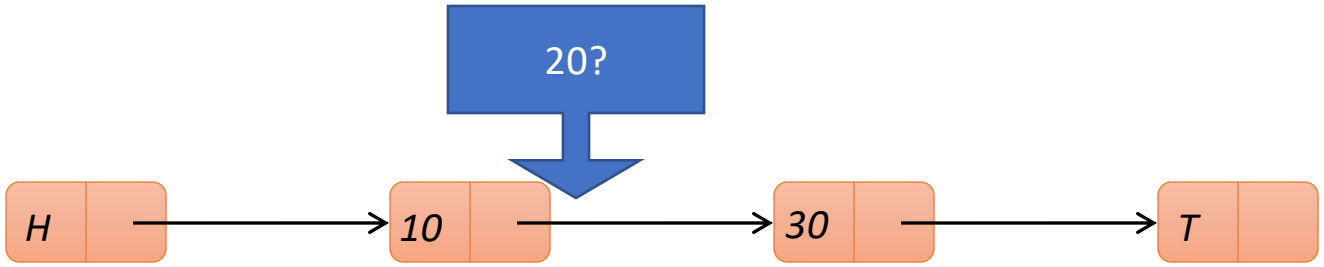
Searching and inserting together

- find(20)



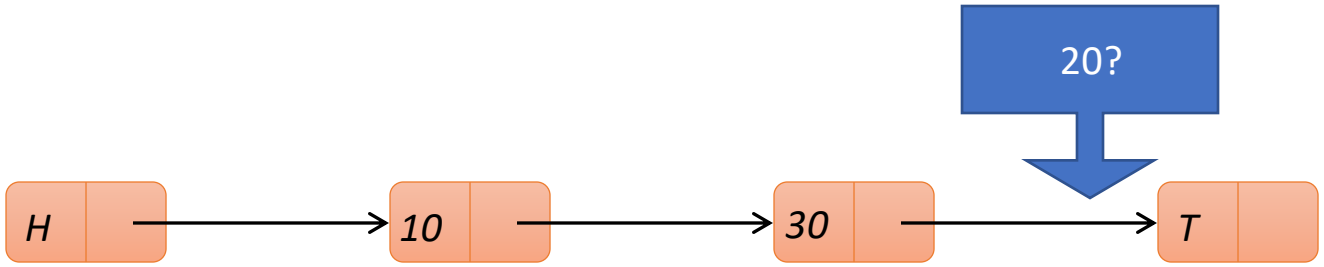
Searching and inserting together

- find(20)



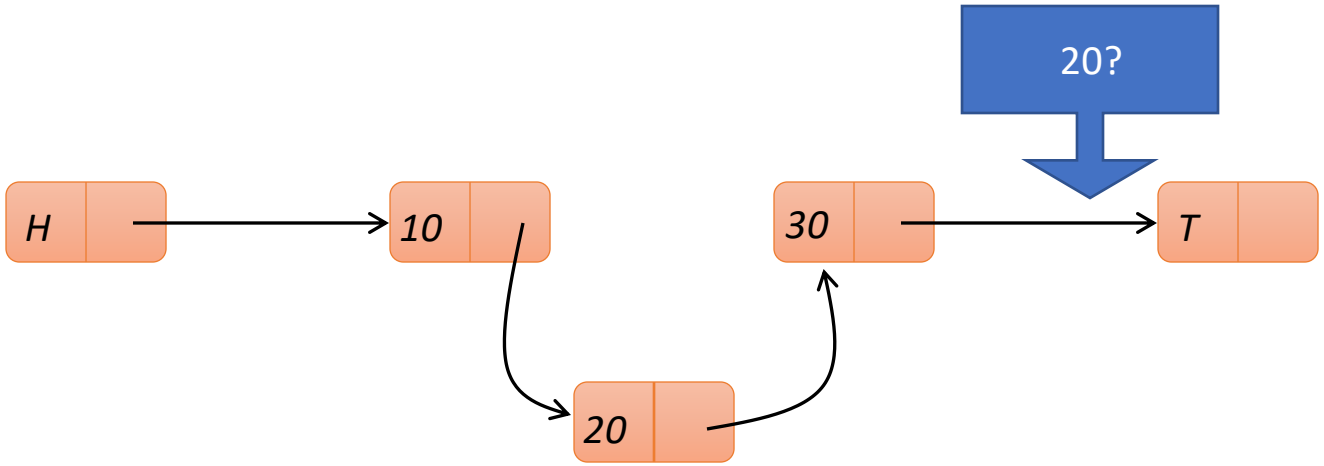
Searching and inserting together

- find(20)



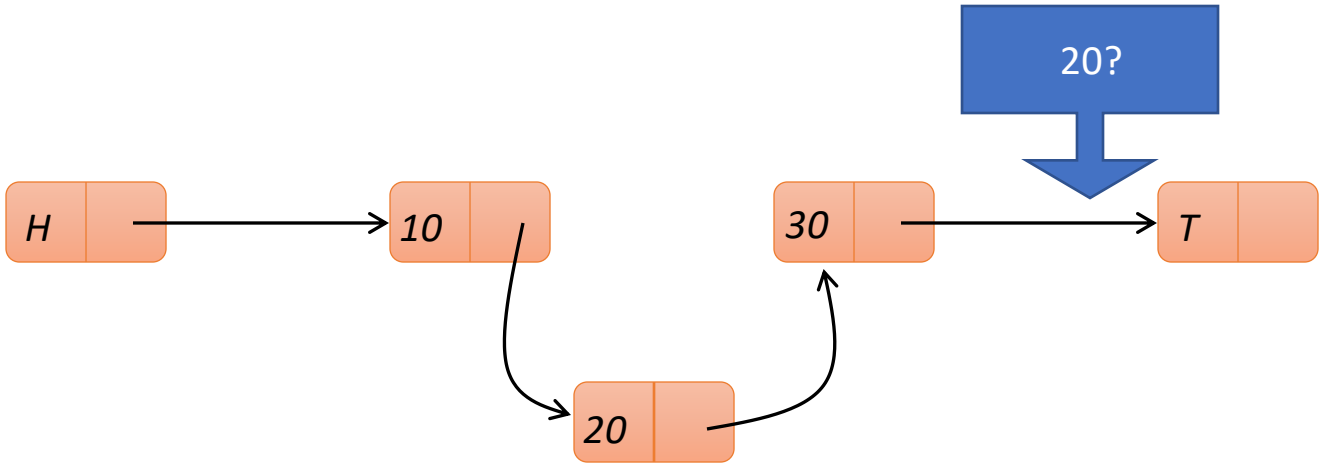
Searching and inserting together

- find(20)
- insert(20) -> true



Searching and inserting together

- find(20) -> false
- insert(20) -> true



Searching and inserting together

- find(20) -> false

This thread saw 20 was not in the set...

- insert(20) -> true

...but this thread succeeded in putting it in!

- Is this a correct implementation?
- Should the programmer be surprised if this happens?
- What about more complicated mixes of operations?

Correctness criteria

Informally:

Look at the behavior of the data structure

- what operations are called on it
- what their results are

If behavior is indistinguishable from atomic calls to a sequential implementation then the concurrent implementation is correct.

Sequential history

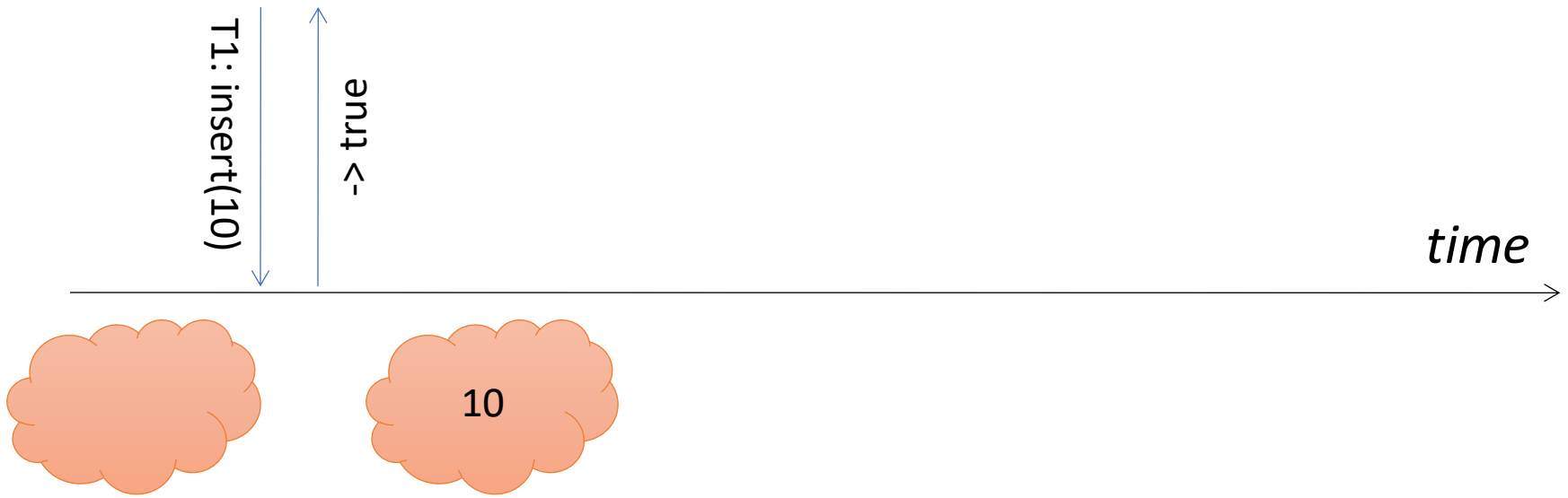
- No overlapping invocations

Sequential history

- No overlapping invocations

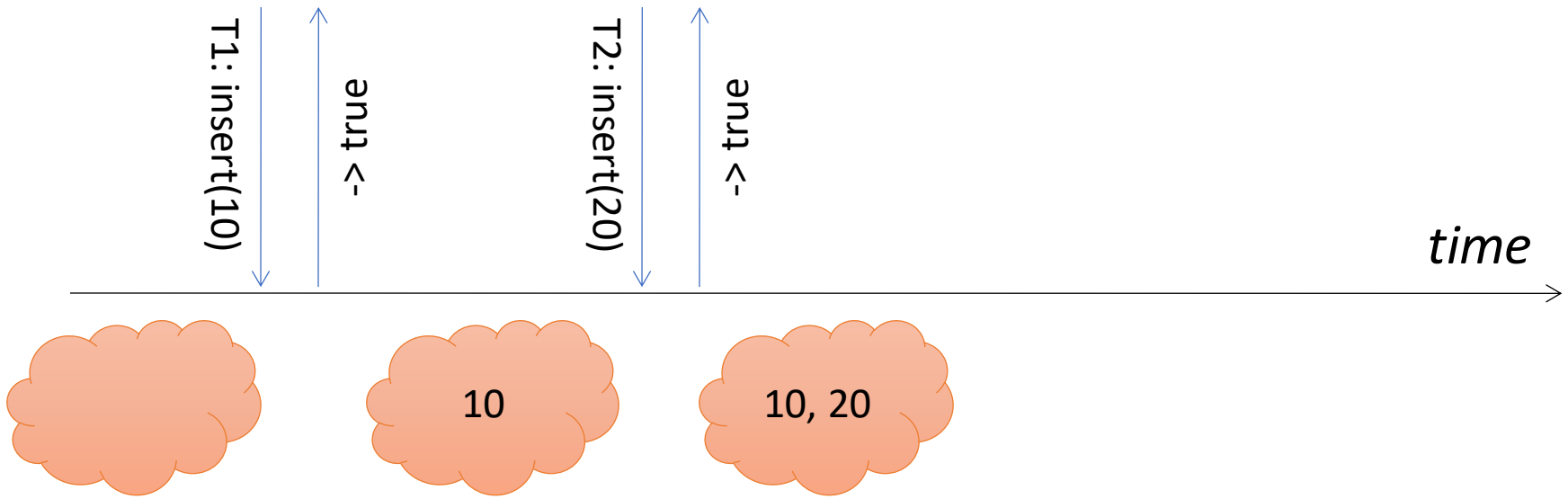
Sequential history

- No overlapping invocations



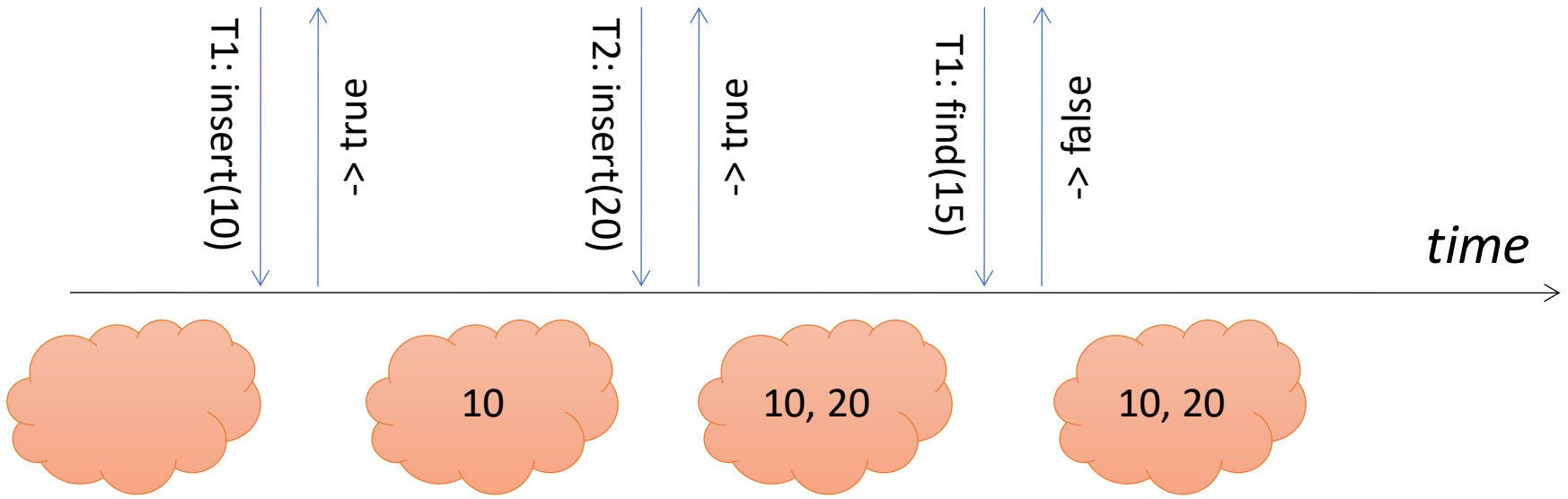
Sequential history

- No overlapping invocations



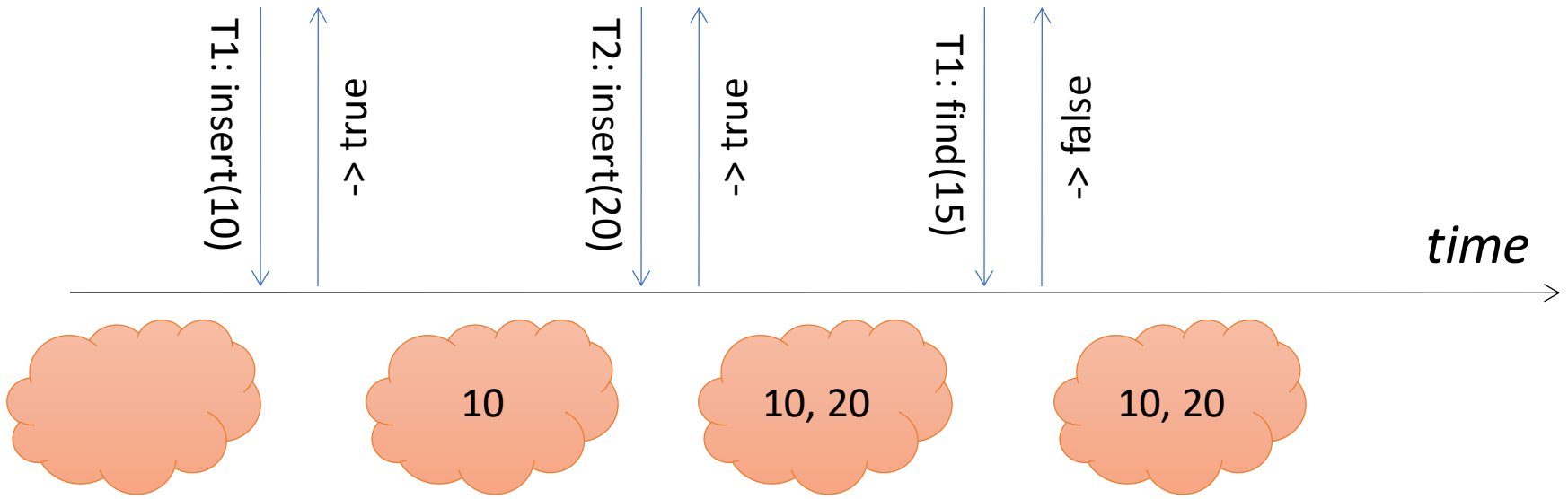
Sequential history

- No overlapping invocations



Sequential history

- No overlapping invocations

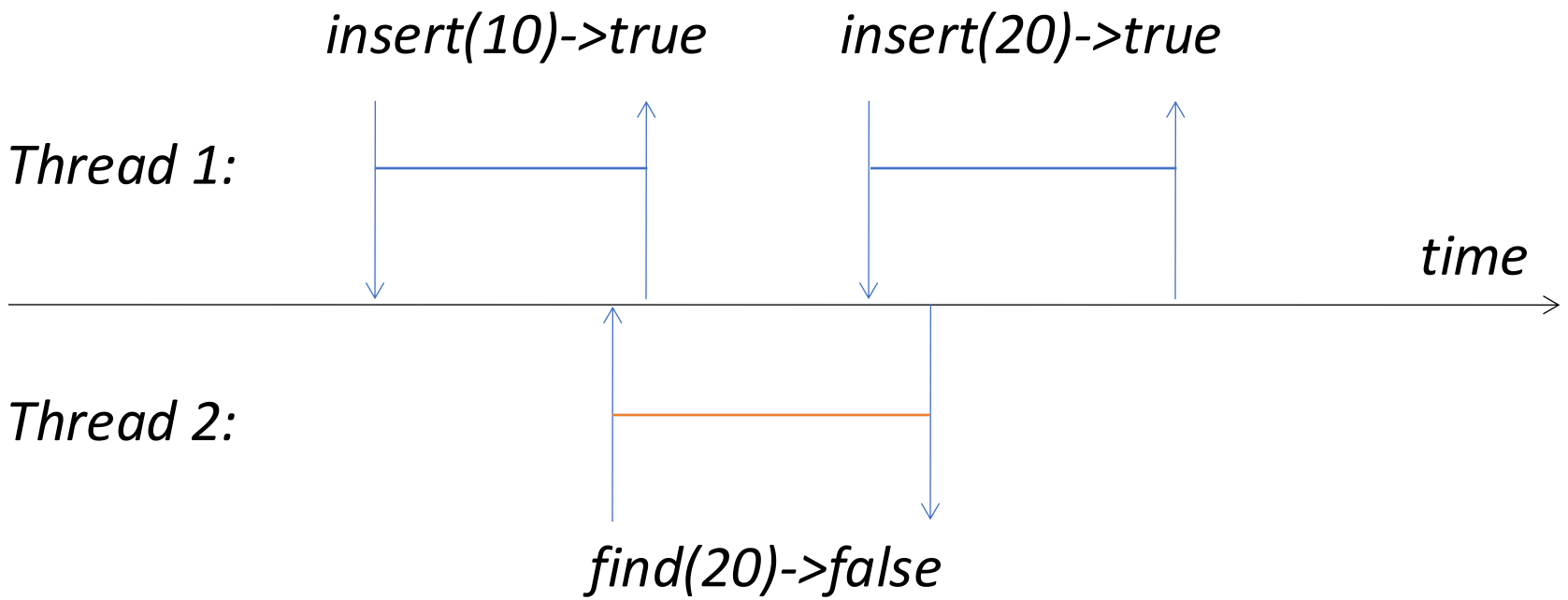


Linearizability: concurrent behaviour should be similar

- even when threads can see intermediate state
- Recall: mutual exclusion precludes overlap

Concurrent history

Allow *overlapping* invocations

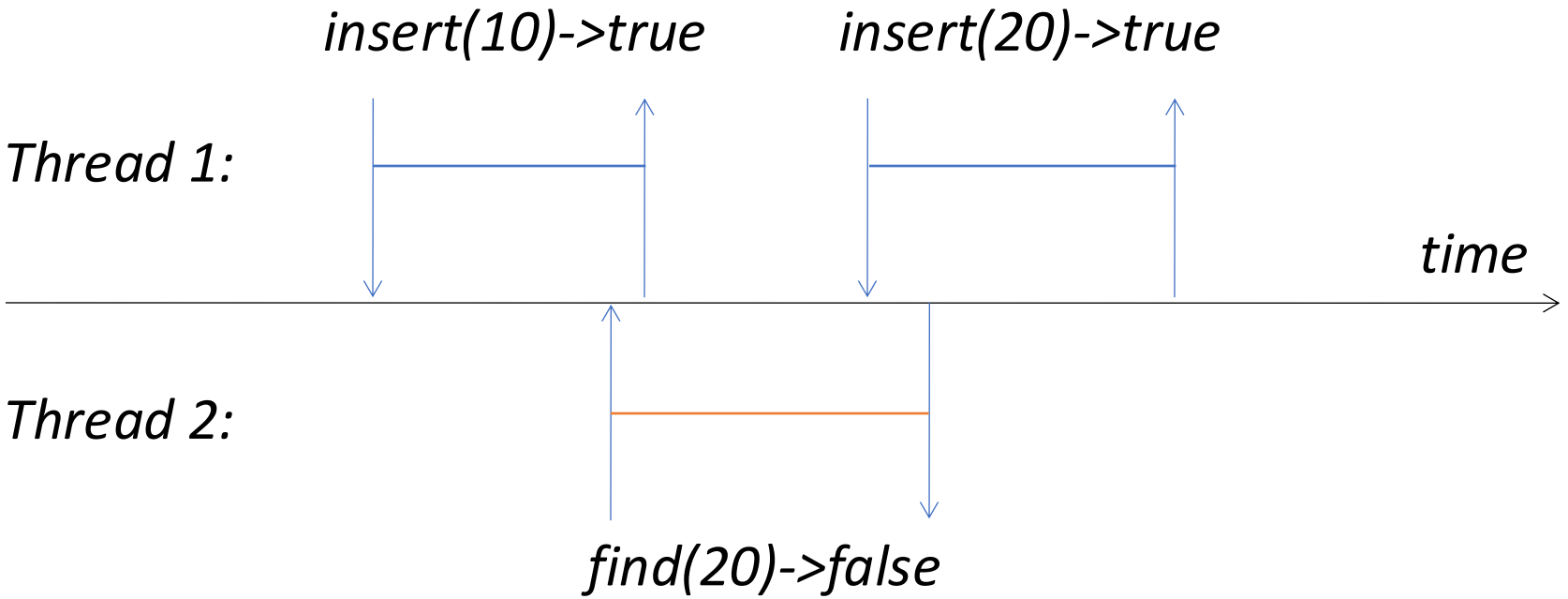


Concurrent history

Allow *overlapping* invocations

Linearizability:

- Is there a correct sequential history:
  - Same results as the concurrent one
  - Consistent with the timing of the invocations/responses?
  - Start/end impose ordering constraints

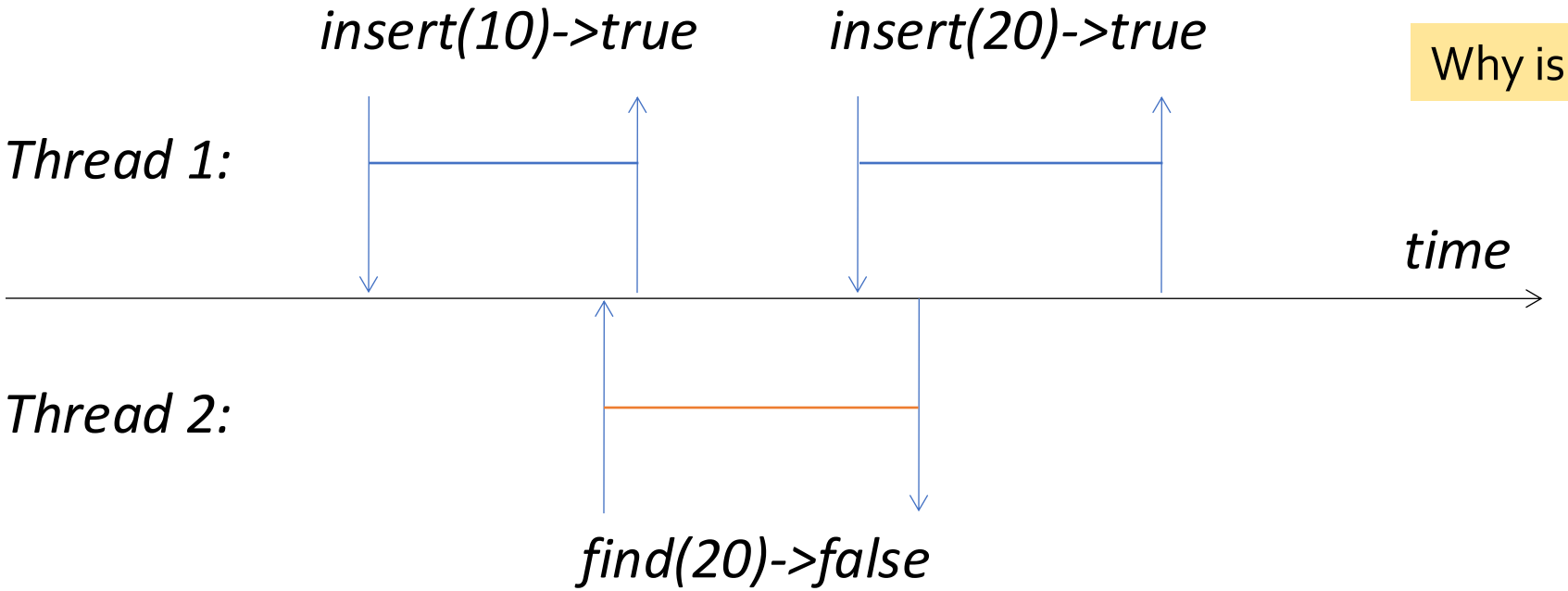


Concurrent history

Allow *overlapping* invocations

Linearizability:

- Is there a correct sequential history:
  - Same results as the concurrent one
  - Consistent with the timing of the invocations/responses?
  - Start/end impose ordering constraints



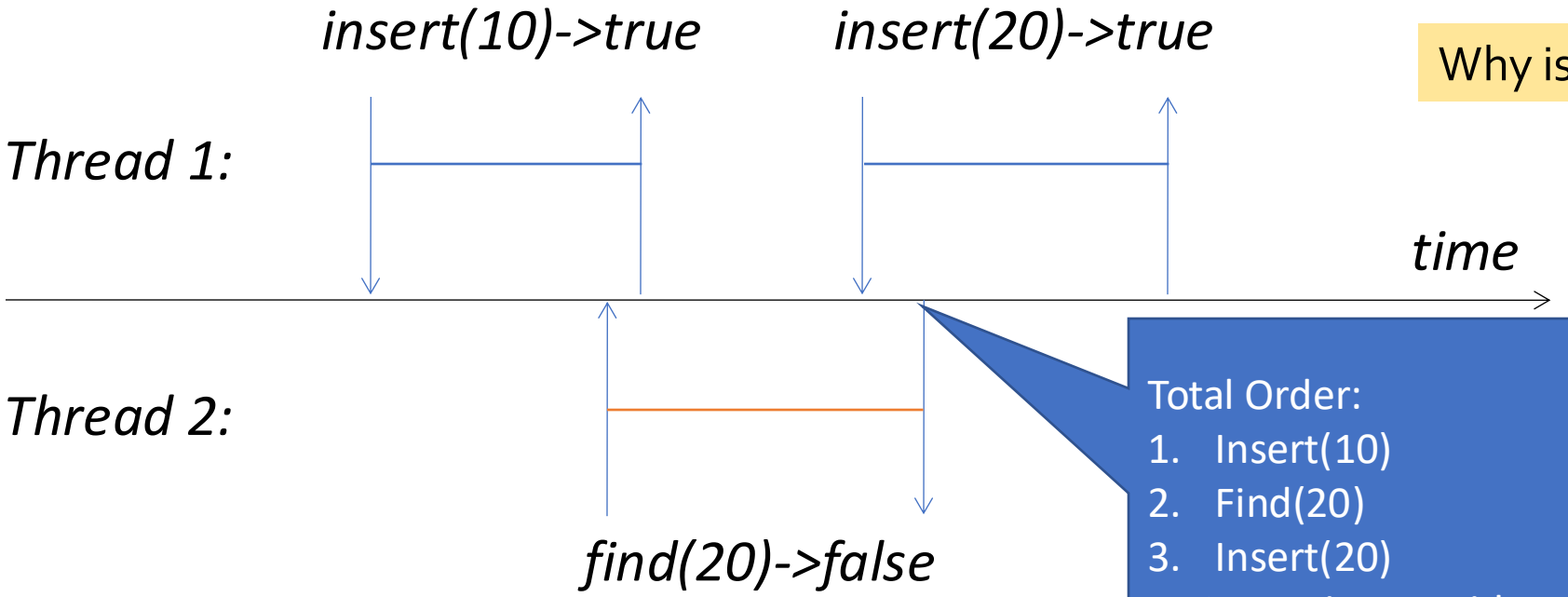
Why is this one OK?

Concurrent history

Allow *overlapping* invocations

Linearizability:

- Is there a correct sequential history:
  - Same results as the concurrent one
  - Consistent with the timing of the invocations/responses?
  - Start/end impose ordering constraints



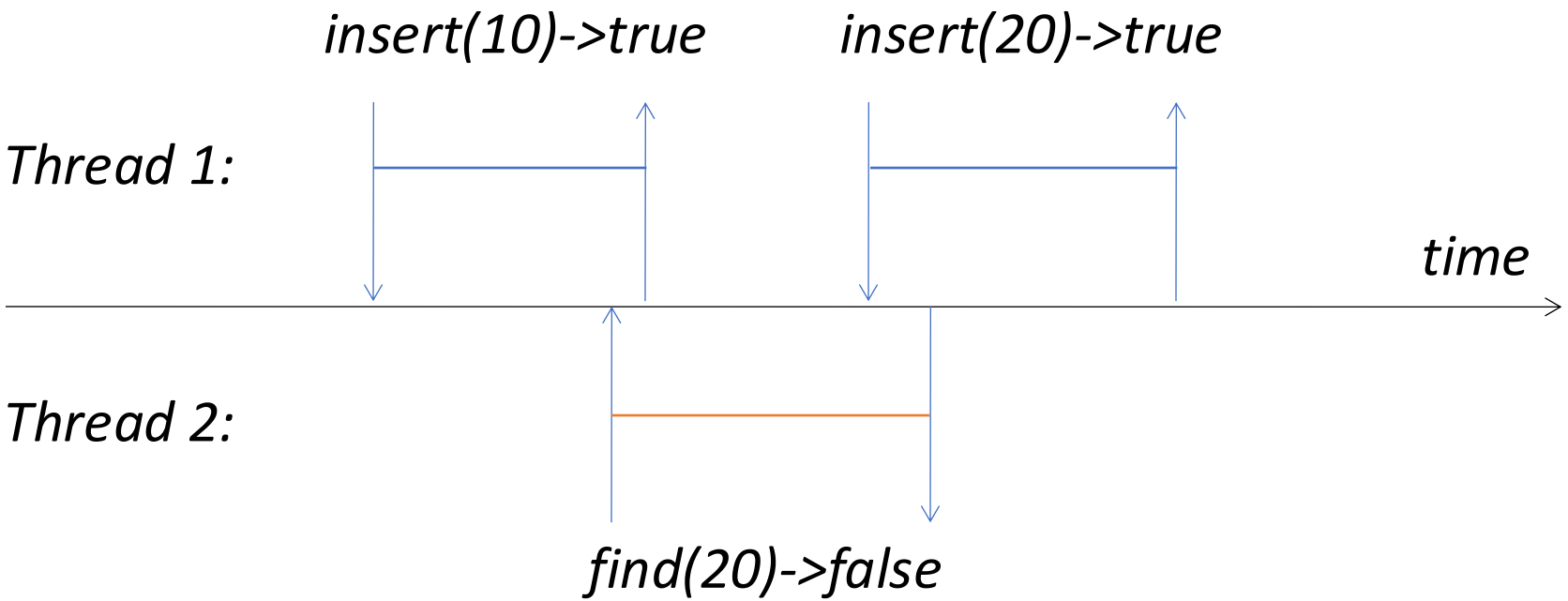
Why is this one OK?

Total Order:

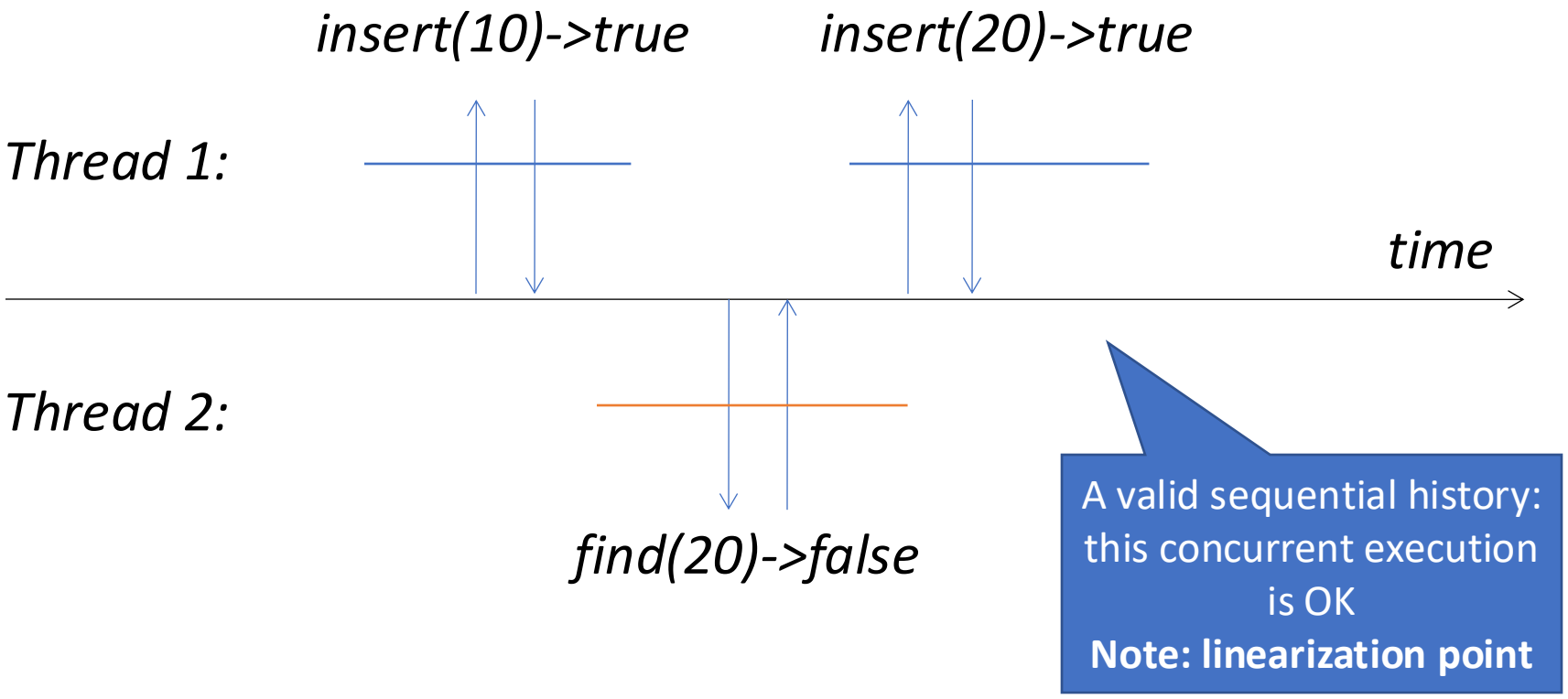
1. Insert(10)
2. Find(20)
3. Insert(20)

- Is consistent with real-time order
- 2, 3 overlap, but return order OK

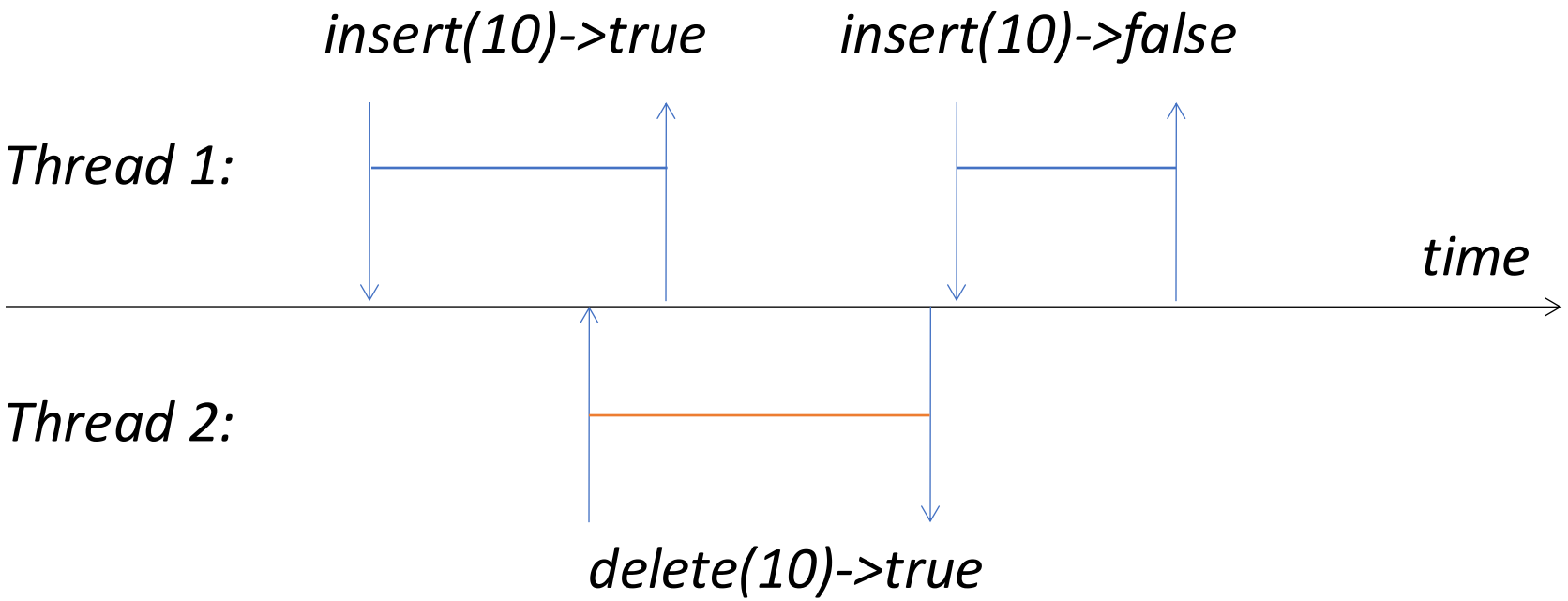
Example: linearizable



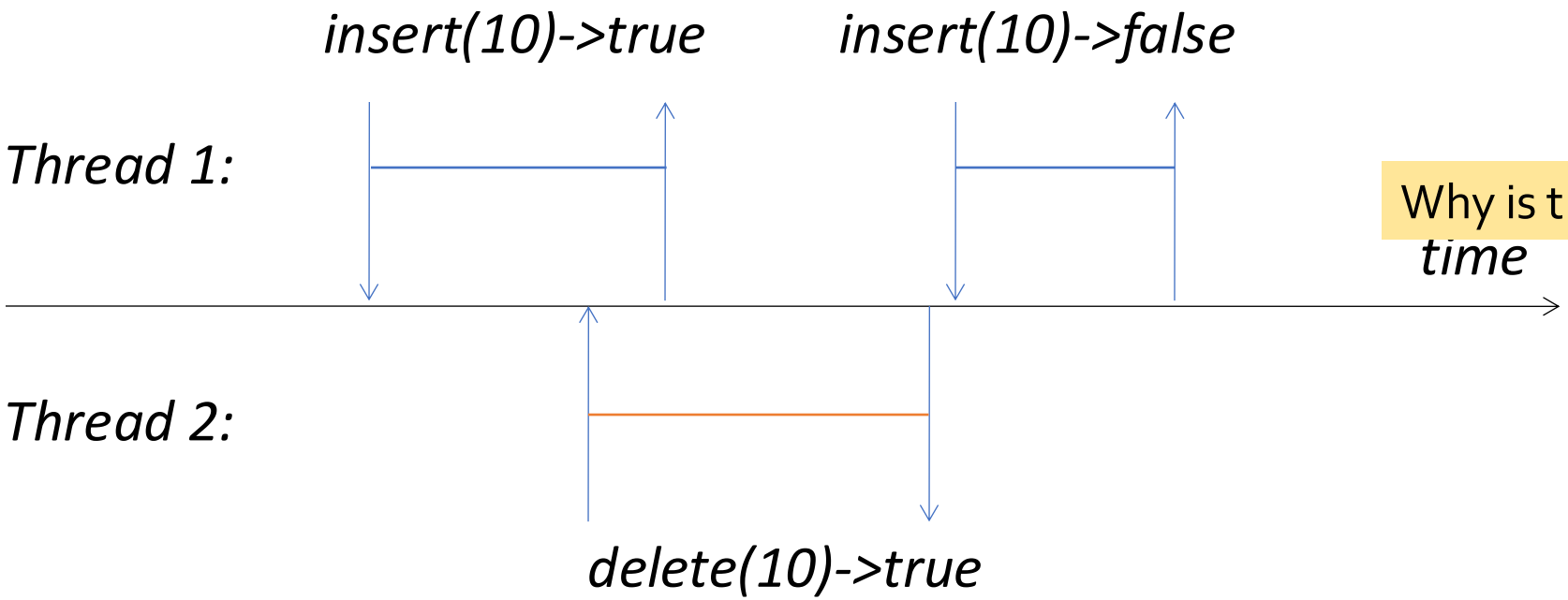
Example: linearizable



Example: not linearizable

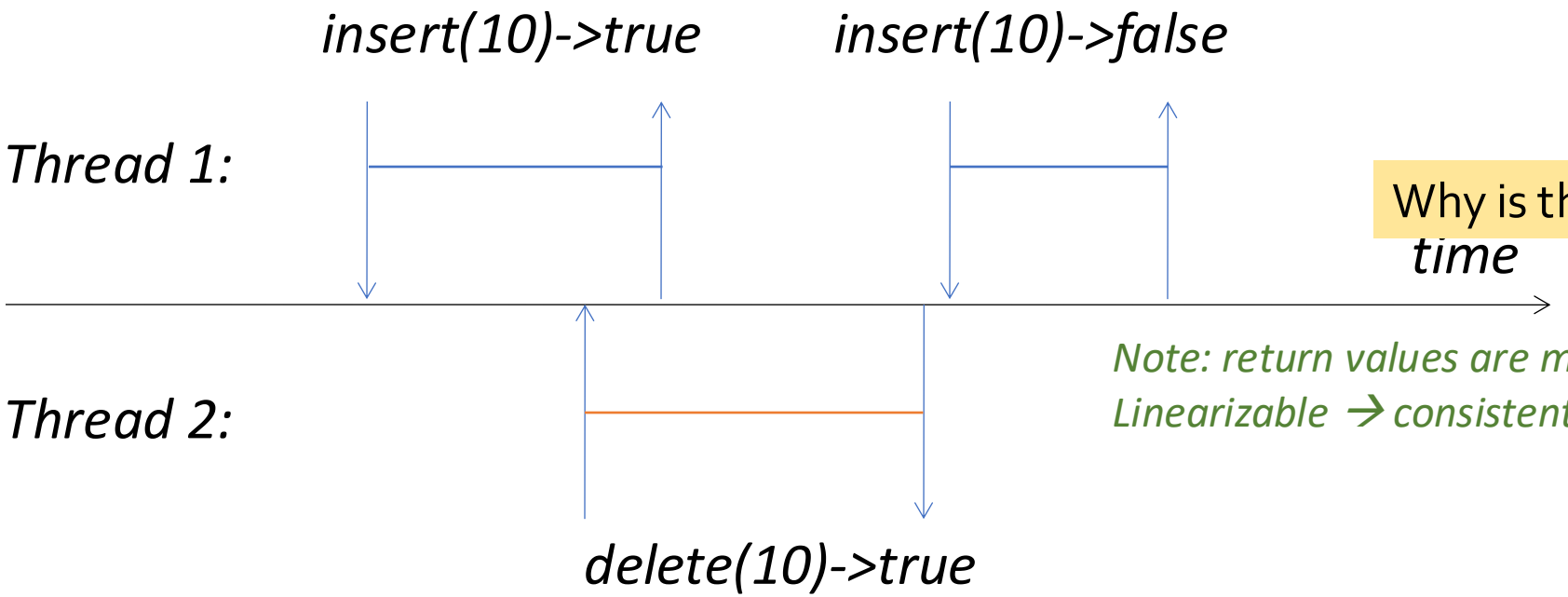


Example: not linearizable



Why is this one NOT OK?

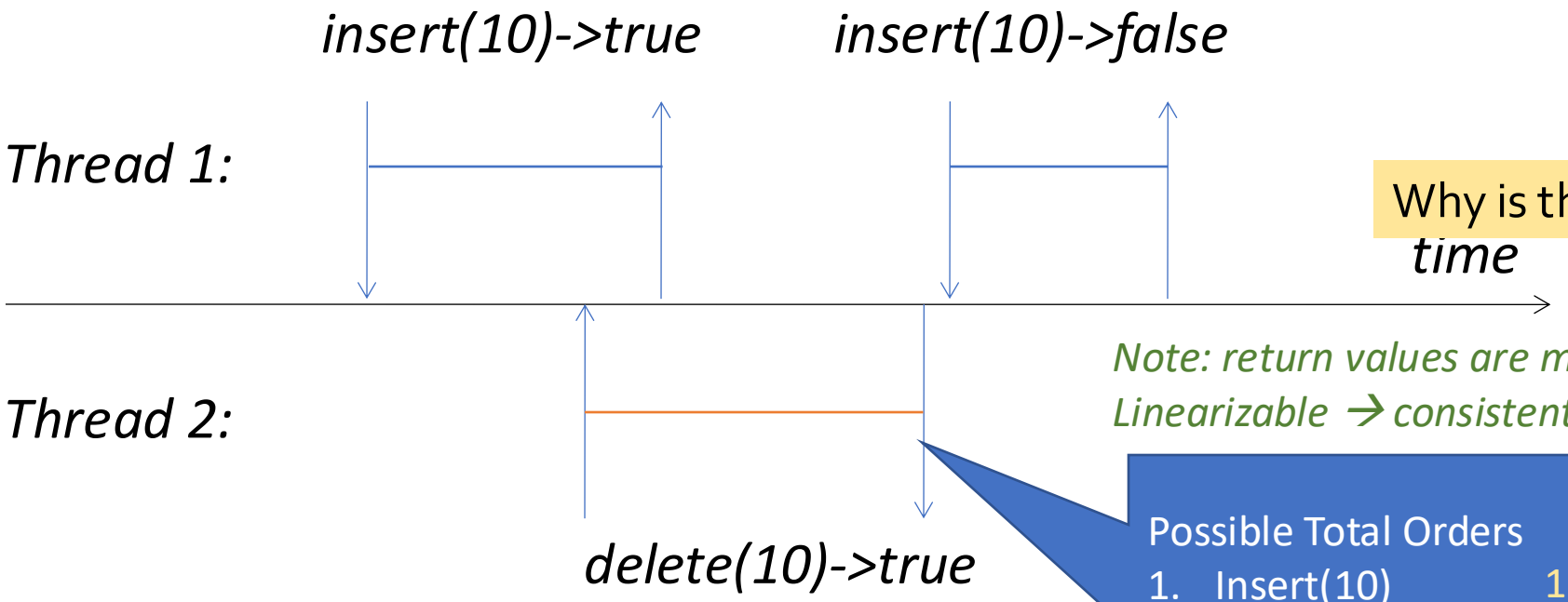
Example: not linearizable



Why is this one NOT OK?

Note: return values are meaningful!
Linearizable → consistent with return values

Example: not linearizable



Why is this one NOT OK?

Note: return values are meaningful!

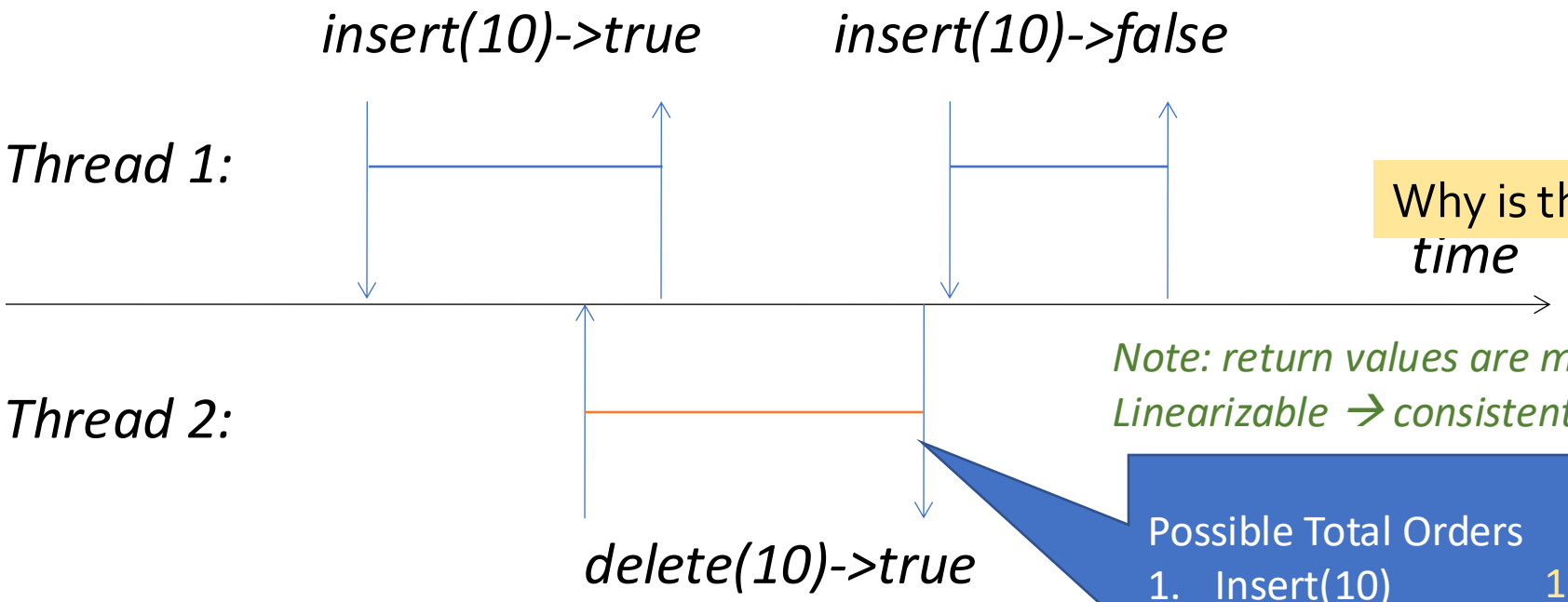
Linearizable → consistent with return values

Possible Total Orders

1. Insert(10)	1. Delete(10)
2. Delete(10)	2. Insert(10)
3. Insert(10)	3. Insert(10)

- Both consistent with real-time order
- 1, 2 overlap, but 3 doesn't

Example: not linearizable



Why is this one NOT OK?

Note: return values are meaningful!
Linearizable → consistent with return values

Possible Total Orders

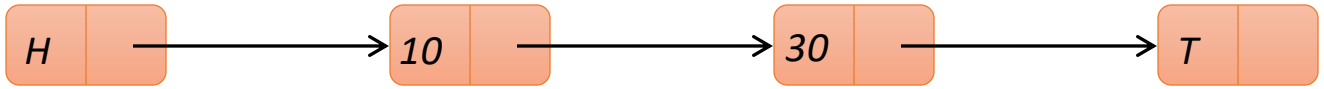
1. Insert(10)	1. Delete(10)
2. Delete(10)	2. Insert(10)
3. Insert(10)	3. Insert(10)

- Both consistent with real-time order
- 1, 2 overlap, but 3 doesn't

How can things like this happen?

Example Revisited

- find(20)

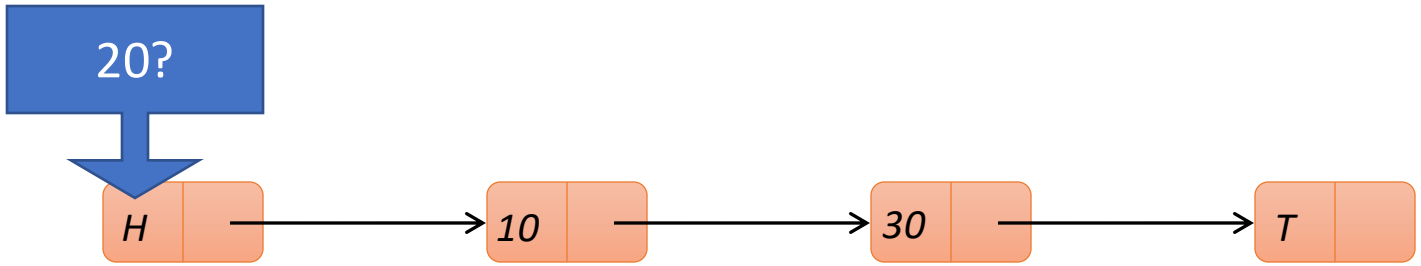


Thread 1:

Thread 2:

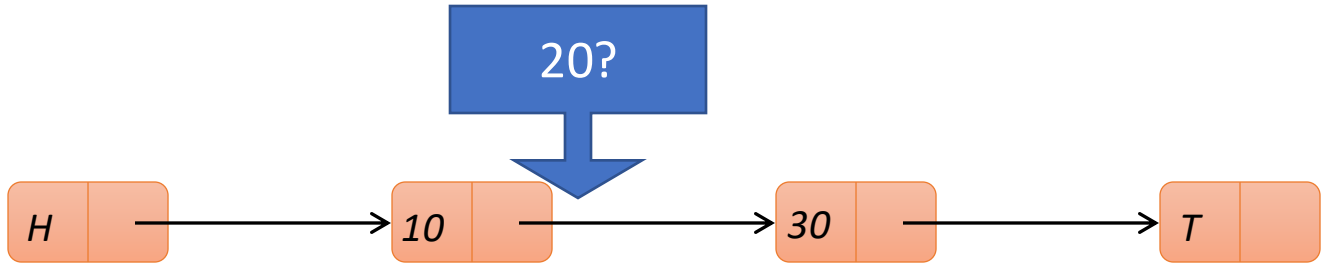
Example Revisited

- find(20)



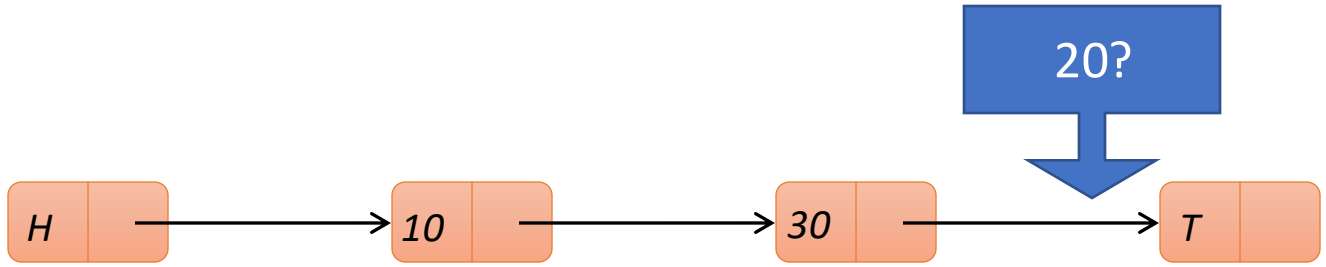
Example Revisited

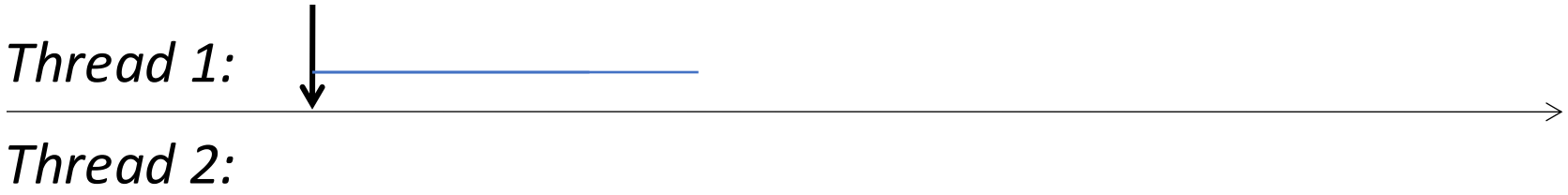
- find(20)



Example Revisited

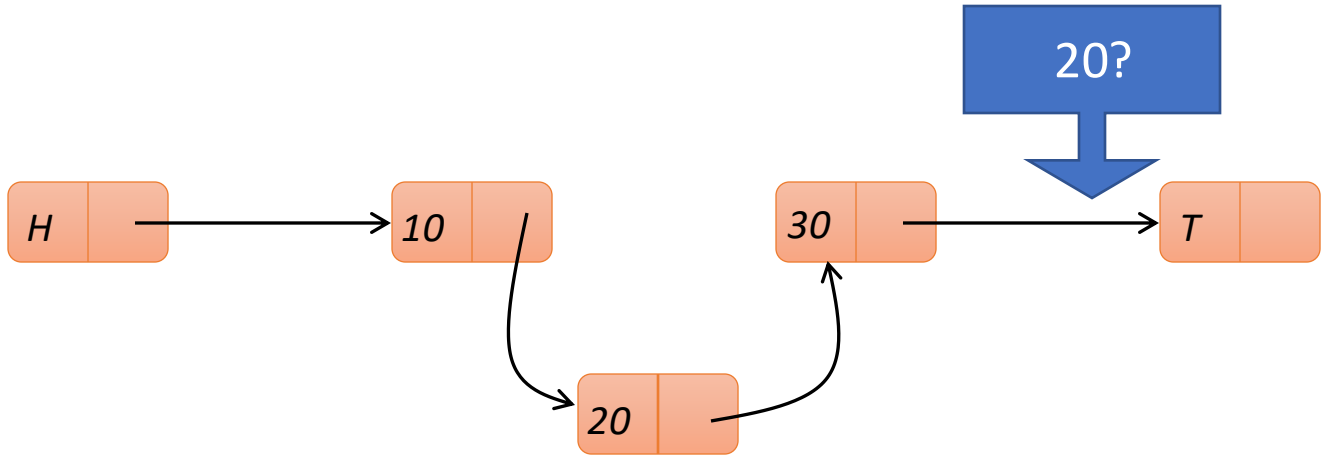
- find(20)





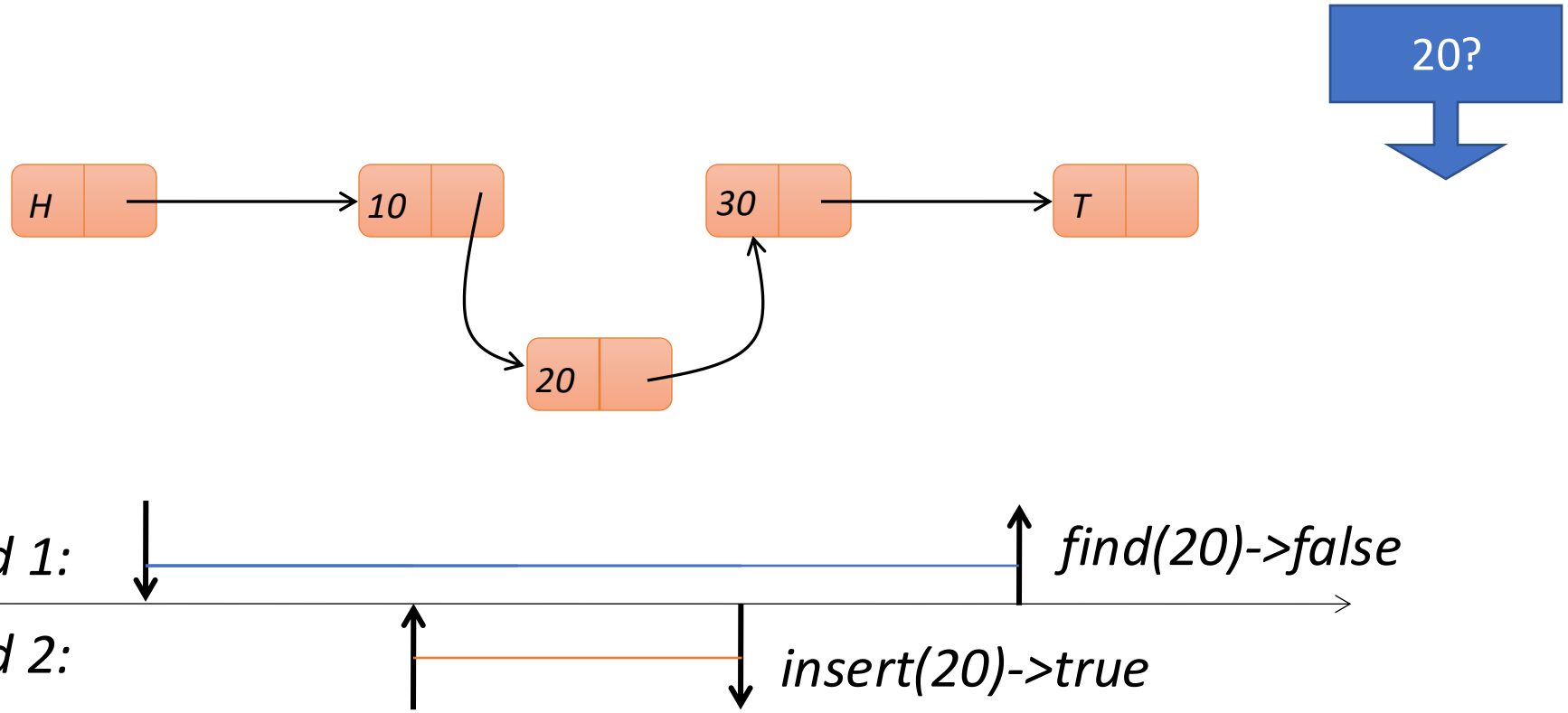
Example Revisited

- find(20)
- insert(20) -> true



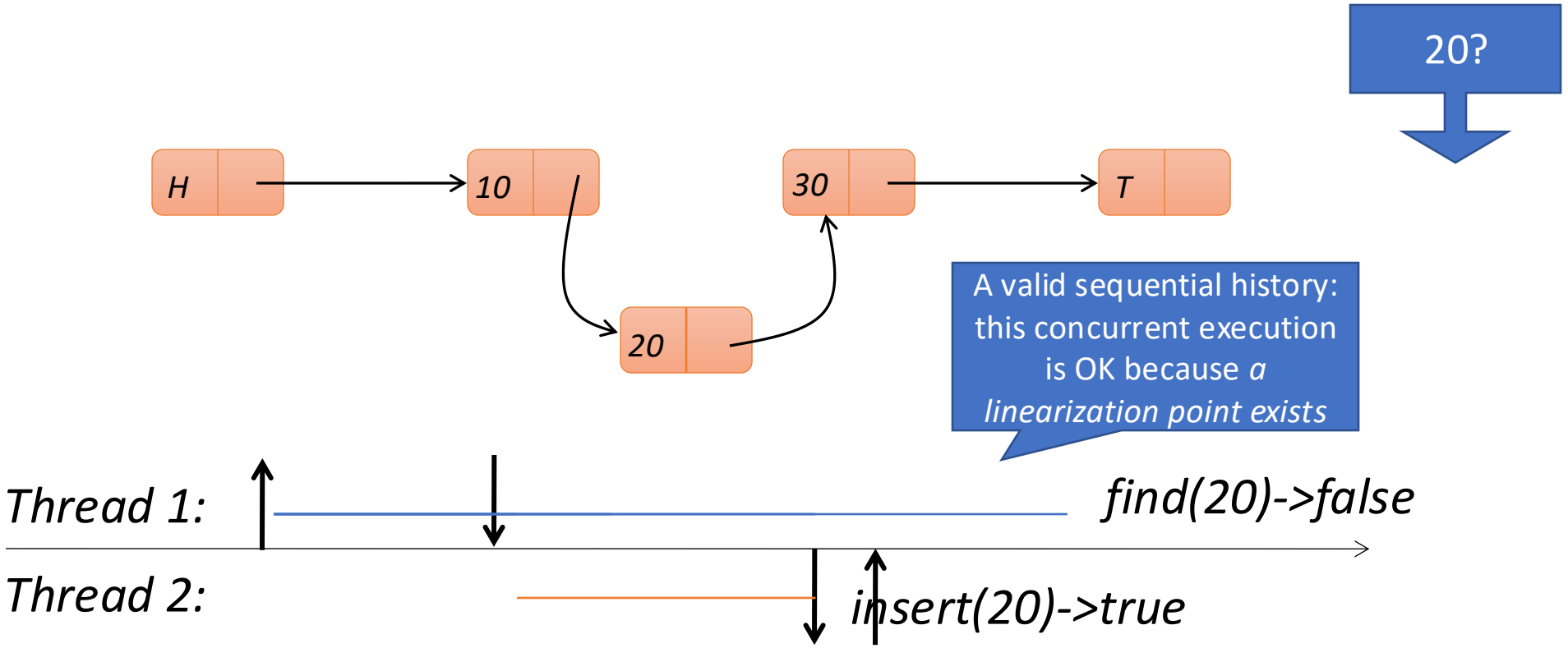
Example Revisited

- find(20) -> false
- insert(20) -> true



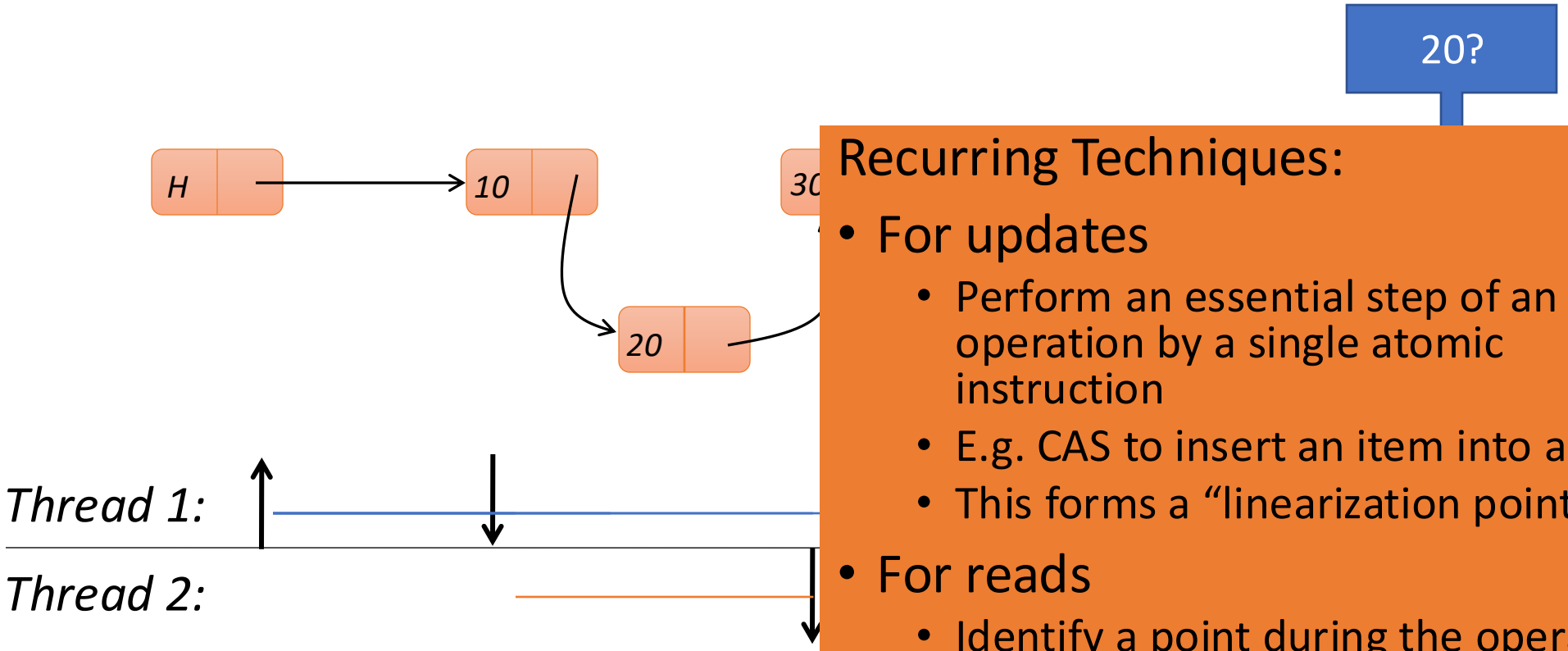
Example Revisited

- find(20) -> false
- insert(20) -> true



Example Revisited

- find(20) -> false
- insert(20) -> true



Recurring Techniques:

- For updates
  - Perform an essential step of an operation by a single atomic instruction
  - E.g. CAS to insert an item into a list
  - This forms a “linearization point”
- For reads
  - Identify a point during the operation’s execution when the result is valid
  - Not always a specific instruction

Formal Properties

Formal Properties

- Wait-free

Formal Properties

- Wait-free
  - A thread finishes its own operation if it continues executing steps

Formal Properties

- Wait-free
  - A thread finishes its own operation if it continues executing steps
  - Strong: everyone eventually finishes

Formal Properties

- Wait-free
  - A thread finishes its own operation if it continues executing steps
  - Strong: everyone eventually finishes
- Lock-free

Formal Properties

- Wait-free
  - A thread finishes its own operation if it continues executing steps
  - Strong: everyone eventually finishes
- Lock-free
  - Some thread finishes its operation if threads continue taking steps

Formal Properties

- Wait-free
  - A thread finishes its own operation if it continues executing steps
  - Strong: everyone eventually finishes
- Lock-free
  - Some thread finishes its operation if threads continue taking steps
  - Weaker: some forward progress guaranateed, but admits unfairness, live-lock, etc.

Formal Properties

- Wait-free
  - A thread finishes its own operation if it continues executing steps
  - Strong: everyone eventually finishes
- Lock-free
  - Some thread finishes its operation if threads continue taking steps
  - Weaker: some forward progress guaranateed, but admits unfairness, live-lock, etc.
- Obstruction-free

Formal Properties

- Wait-free
  - A thread finishes its own operation if it continues executing steps
  - Strong: everyone eventually finishes
- Lock-free
  - Some thread finishes its operation if threads continue taking steps
  - Weaker: some forward progress guaranateed, but admits unfairness, live-lock, etc.
- Obstruction-free
  - A thread finishes its own operation if it runs in isolation

Formal Properties

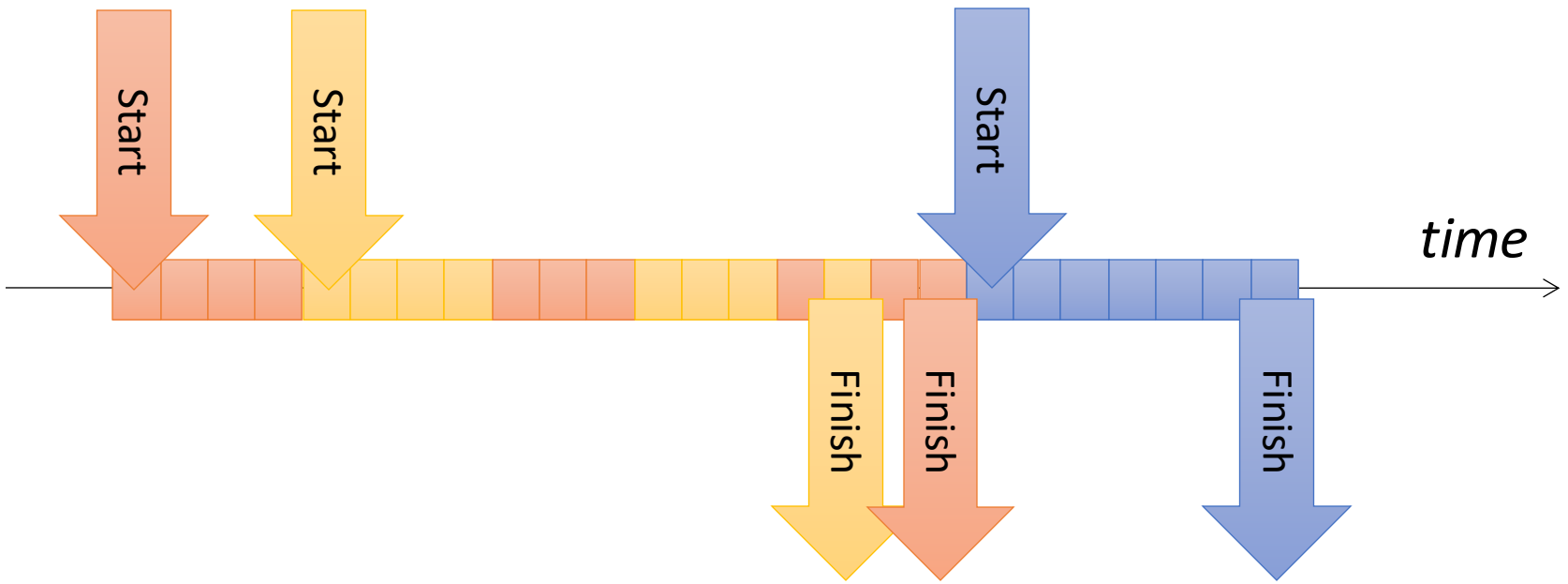
- Wait-free
  - A thread finishes its own operation if it continues executing steps
  - Strong: everyone eventually finishes
- Lock-free
  - Some thread finishes its operation if threads continue taking steps
  - Weaker: some forward progress guaranateed, but admits unfairness, live-lock, etc.
- Obstruction-free
  - A thread finishes its own operation if it runs in isolation
  - Very weak. Means if you remove contention, someone finishes

Wait-free

- A thread finishes its own operation if it continues executing steps

Wait-free

- A thread finishes its own operation if it continues executing steps

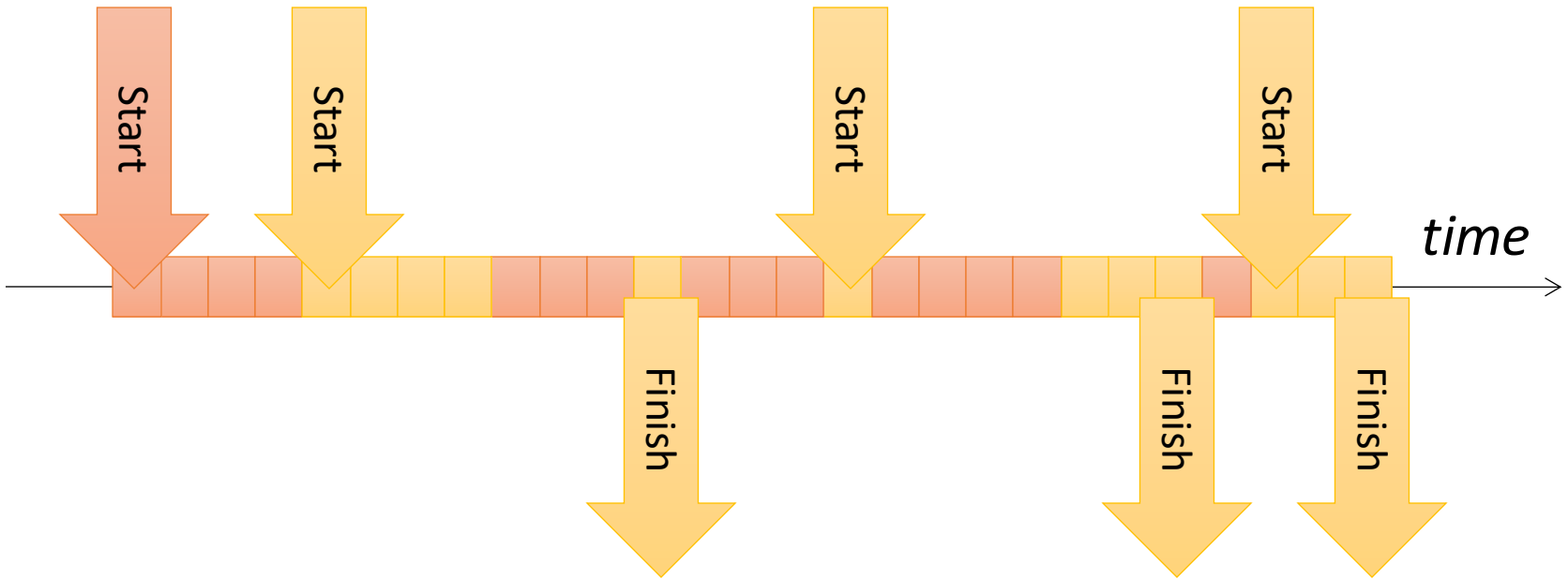


Lock-free

- Some thread finishes its operation if threads continue taking steps

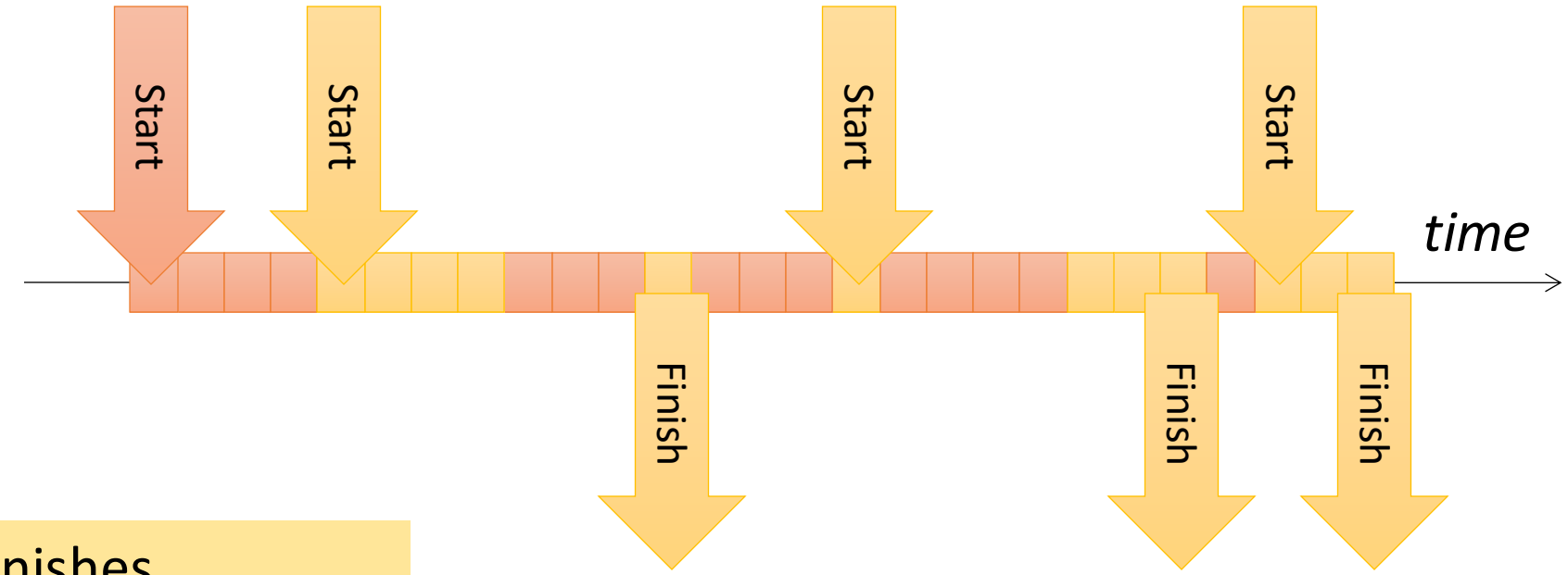
Lock-free

- Some thread finishes its operation if threads continue taking steps



Lock-free

- Some thread finishes its operation if threads continue taking steps



- Red never finishes
- Orange does
- Still lock-free

Obstruction-free

Obstruction-free

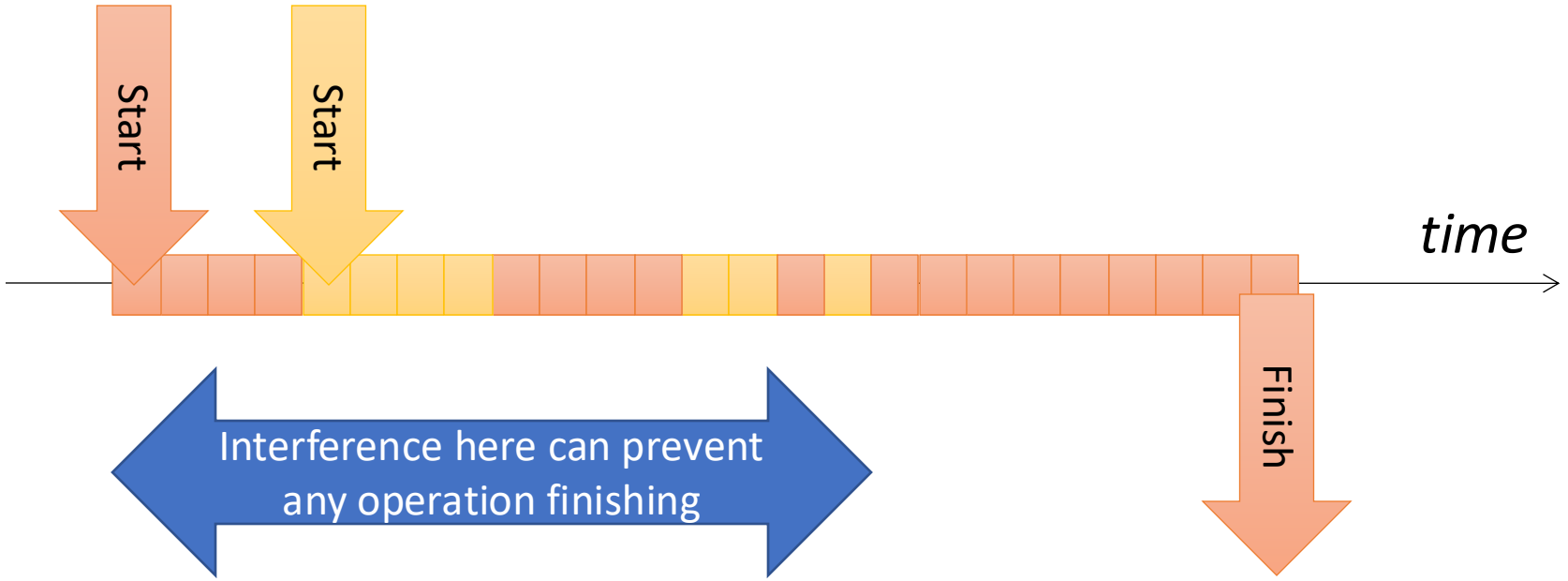
- A thread finishes its own operation if it runs in isolation

Obstruction-free

- A thread finishes its own operation if it runs in isolation
- *Meaning, if you de-schedule contenders*

Obstruction-free

- A thread finishes its own operation if it runs in isolation
- *Meaning, if you de-schedule contenders*

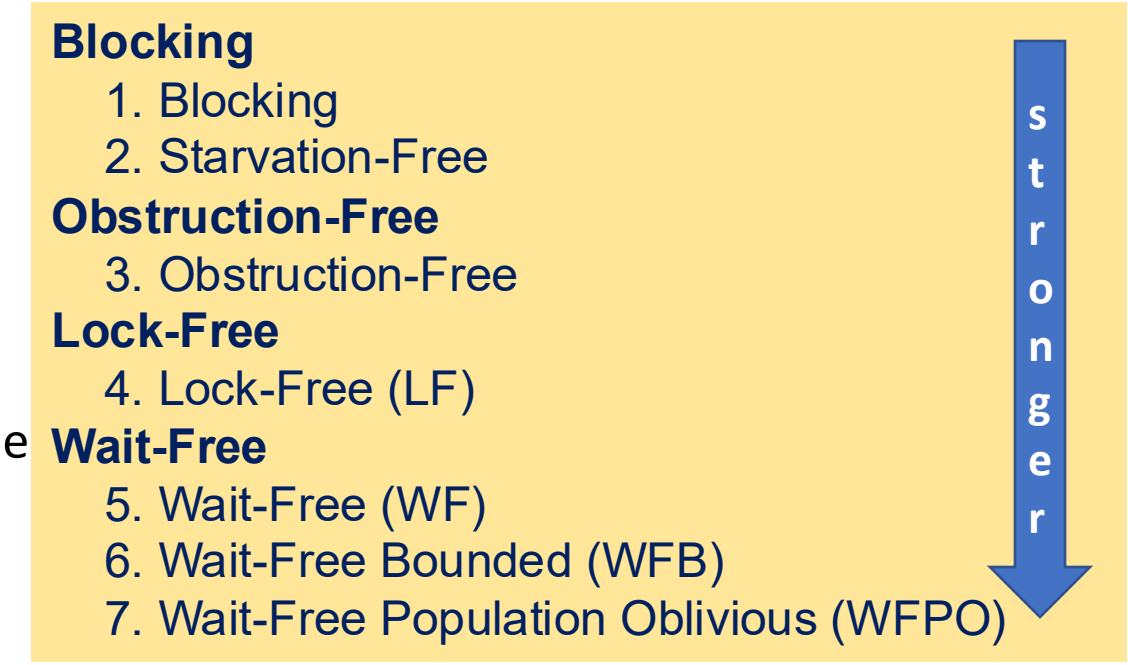


Formal Properties

- Wait-free
  - A thread finishes its own operation if it continues executing steps
  - Strong: everyone eventually finishes
- Lock-free
  - Some thread finishes its operation if threads continue taking steps
  - Weaker: some forward progress guaranateed, but admits unfairness, live-lock, etc.
- Obstruction-free
  - A thread finishes its own operation if it runs in isolation
  - Very weak. Means if you remove contention, someone finishes

Formal Properties

- Wait-free
  - A thread finishes its own operation if it continue
  - Strong: everyone eventually finishes
- Lock-free
  - Some thread finishes its operation if threads continue taking steps
  - Weaker: some forward progress guaranateed, but admits unfairness, live-lock, etc.
- Obstruction-free
  - A thread finishes its own operation if it runs in isolation
  - Very weak. Means if you remove contention, someone finishes



Formal Properties

- Wait-free
  - A thread finishes its own operation if it continue
  - Strong: everyone eventually finishes
- Lock-free
  - Some thread finishes its operation if threads continue taking steps
  - Weaker: some forward progress guaranateed, but admits unfairness, li
- Obstruction-free
  - A thread finishes its own operation if it runs in isolation
  - Very weak. Means if you remove contention, someone finishes

Blocking
1. Blocking
2. Starvation-Free

Obstruction-Free
3. Obstruction-Free

Lock-Free
4. Lock-Free (LF)

Wait-Free
5. Wait-Free (WF)
6. Wait-Free Bounded (WFB)
7. Wait-Free Population Oblivious (WFPO)

stronger

Linearizability Properties

Linearizability Properties

- non-blocking
  - one method is never forced to wait to sync with another.

Linearizability Properties

- non-blocking
  - one method is never forced to wait to sync with another.
- local property:
  - a system is linearizable iff each individual object is linearizable.
  - gives us **composability**.

Linearizability Properties

- non-blocking
  - one method is never forced to wait to sync with another.
- local property:
  - a system is linearizable iff each individual object is linearizable.
  - gives us **composability**.
- Why is it important?
  - Serializability is not composable.

Linearizability Properties

- non-blocking
  - one method is never forced to wait to sync with another.
- local property:
  - a system is linearizable iff each individual object is linearizable.
  - gives us **composability**.
- Why is it important?
  - Serializability is not composable.

Huh? Composable?

Composability

Composability

```
T * list::remove(Obj key){
  LOCK(this);
  tmp = __do_remove(key);
  UNLOCK(this);
  return tmp;
}
```

Composability

```
T * list::remove(Obj key){
  LOCK(this);
  tmp = __do_remove(key);
  UNLOCK(this);
  return tmp;
}

void list::insert(Obj key, T * val){
  LOCK(this);
  __do_insert(key, val);
  UNLOCK(this);
}
```

Composability

```
T * list::remove(Obj key){
  LOCK(this);
  tmp = __do_remove(key);
  UNLOCK(this);
  return tmp;
}
```

```
void list::insert(Obj key, T * val){
  LOCK(this);
  __do_insert(key, val);
  UNLOCK(this);
}
```

```
void move(list s, list d, Obj key){
  tmp = s.remove(key);
  d.insert(key, tmp);
}
```

Composability

Thread-safe?

```
T * list::remove(Obj key){
  LOCK(this);
  tmp = __do_remove(key);
  UNLOCK(this);
  return tmp;
}

void list::insert(Obj key, T * val){
  LOCK(this);
  __do_insert(key, val);
  UNLOCK(this);
}
```

```
void move(list s, list d, Obj key){
  tmp = s.remove(key);
  d.insert(key, tmp);
}
```

Composability

```
T * list::remove(Obj key){
  LOCK(this);
  tmp = __do_remove(key);
  UNLOCK(this);
  return tmp;
}
```

```
void list::insert(Obj key, T * val){
  LOCK(this);
  __do_insert(key, val);
  UNLOCK(this);
}
```

```
void move(list s, list d, Obj key){
  LOCK(s);
  LOCK(d);
  tmp = s.remove(key);
  d.insert(key, tmp);
  UNLOCK(d);
  UNLOCK(s);
}
```

Composability

```
T * list::remove(Obj key){
  LOCK(this);
  tmp = __do_remove(key);
  UNLOCK(this);
  return tmp;
}
```

```
void list::insert(Obj key, T * val){
  LOCK(this);
  __do_insert(key, val);
  UNLOCK(this);
}
```

```
void move(list s, list d, Obj key){
  LOCK(s);
  LOCK(d);
  tmp = s.remove(key);
  d.insert(key, tmp);
  UNLOCK(d);
  UNLOCK(s);
}
```

- Lock-based code doesn’t compose

Composability

```
T * list::remove(Obj key){
  LOCK(this);
  tmp = __do_remove(key);
  UNLOCK(this);
  return tmp;
}
```

```
void list::insert(Obj key, T * val){
  LOCK(this);
  __do_insert(key, val);
  UNLOCK(this);
}
```

```
void move(list s, list d, Obj key){
  LOCK(s);
  LOCK(d);
  tmp = s.remove(key);
  d.insert(key, tmp);
  UNLOCK(d);
  UNLOCK(s);
}
```

- Lock-based code doesn’t compose
- If list were a linearizable concurrent data structure, composition OK?

Composability

```
T * list::remove(Obj key){
  LOCK(this);
  tmp = __do_remove(key);
  UNLOCK(this);
  return tmp;
}
```

```
void list::insert(Obj key, T * val){
  LOCK(this);
  __do_insert(key, val);
  UNLOCK(this);
}
```

```
void move(list s, list d, Obj key){
  LOCK(s);
  LOCK(d);
  tmp = s.remove(key);
  d.insert(key, tmp);
  UNLOCK(d);
  UNLOCK(s);
}
```

Painting with a very broad brush
Composition with linearizability is really about composed schedules

- Lock-based code doesn't compose
- If list were a linearizable concurrent data structure, composition OK?

Linearizability Properties

- non-blocking
  - one method is never forced to wait to sync with another.
- applies to histories *and* objects *and* systems:
  - a history can be linearizable
  - an object is linearizable if all valid histories of its use are linearizable
- local property:
  - a system is linearizable iff each individual object is linearizable.
  - gives us **composability**.
- Why is it important?
  - Serializability is not composable.
  - Core hypotheses:
    - structuring all as concurrent objects buys composability
    - structuring all as concurrent objects is tractable/possible

More on Composability and Compositionality

More on Composability and Compositionality

- High level /informal meaning:
  - Can you compose codes that provide property P
  - …and expect the composition to preserve P?

More on Composability and Compositionality

- High level /informal meaning:
  - Can you compose codes that provide property P
  - …and expect the composition to preserve P?
- More nuanced meanings:
  - Can you compose codes
  - Can you compose schedules

More on Composability and Compositionality

- High level /informal meaning:
  - Can you compose codes that provide property P
  - …and expect the composition to preserve P?
- More nuanced meanings:
  - Can you compose codes
  - Can you compose schedules
- These are related but differ in subtle ways

More on Composability and Compositionality

- High level /informal meaning:
  - Can you compose codes that provide property P
  - …and expect the composition to preserve P?
- More nuanced meanings:
  - Can you compose codes
  - Can you compose schedules
- These are related but differ in subtle ways
- Non-composability of serializability is really about composing schedules

Consider A Concurrent Register

Consider A Concurrent Register

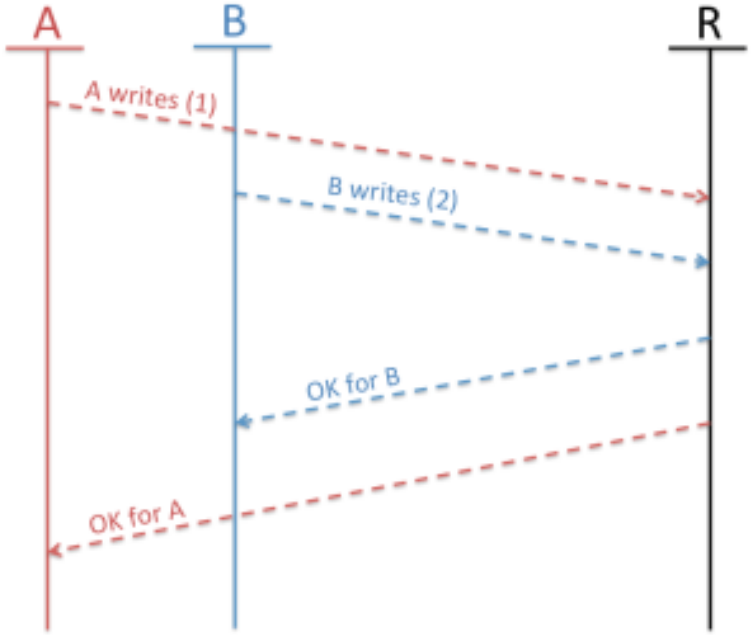
- Threads A, B write integers to a register R

Consider A Concurrent Register

- Threads A, B write integers to a register R
- Because it is concurrent, method invocations overlap

Consider A Concurrent Register

- Threads A, B write integers to a register R
- Because it is concurrent, method invocations overlap



Two Concurrent Registers

Two Concurrent Registers

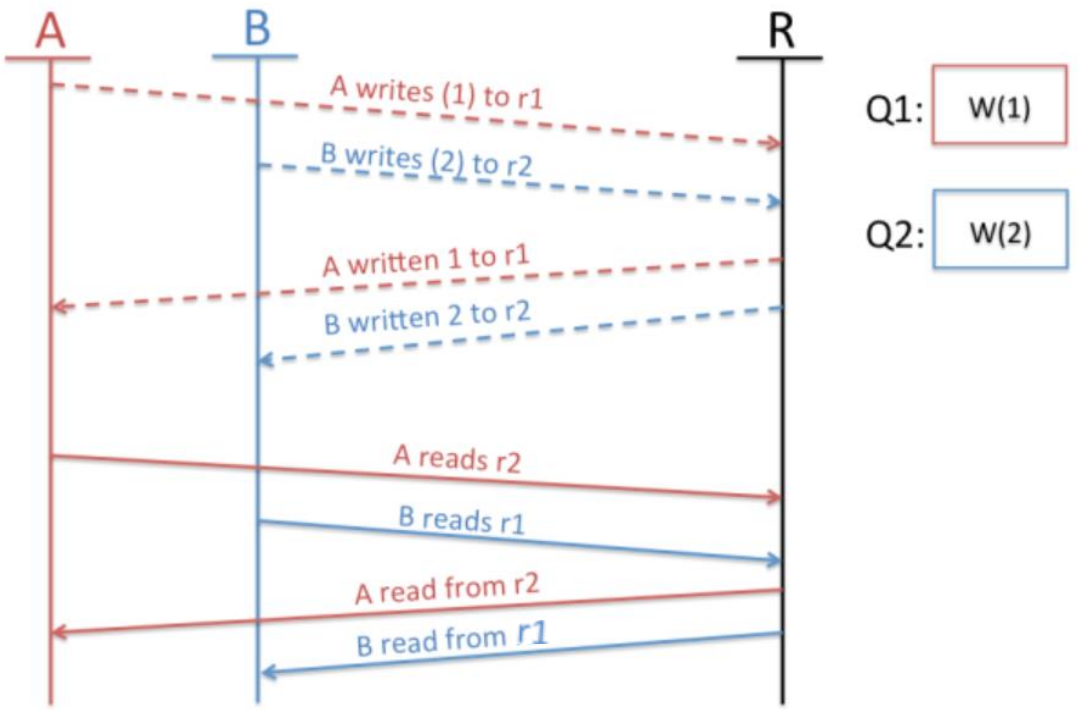
- Register value is initially zero

Two Concurrent Registers

- Register value is initially zero
- The following operations occur:
  - Thread A:
    - write r1 = 1
    - read r2 → ?
  - Thread B:
    - B: write r2 -> 2
    - B: read r1 → ?

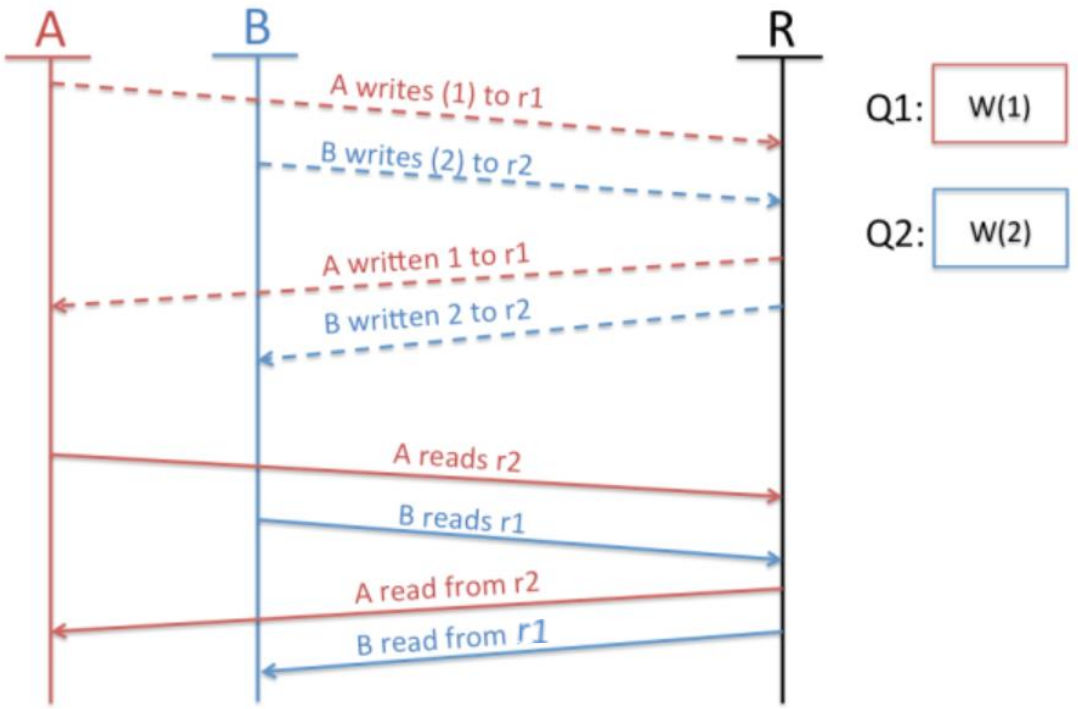
Two Concurrent Registers

- Register value is initially zero
- The following operations occur:
  - Thread A:
    - write r1 = 1
    - read r2 → ?
  - Thread B:
    - B: write r2 -> 2
    - B: read r1 → ?



Two Concurrent Registers

- Register value is initially zero
- The following operations occur:
  - Thread A:
    - write r1 = 1
    - read r2 → ?
  - Thread B:
    - B: write r2 -> 2
    - B: read r1 → ?
- Serializability:
  - Execution equivalent to *some serial order*
  - All see same order

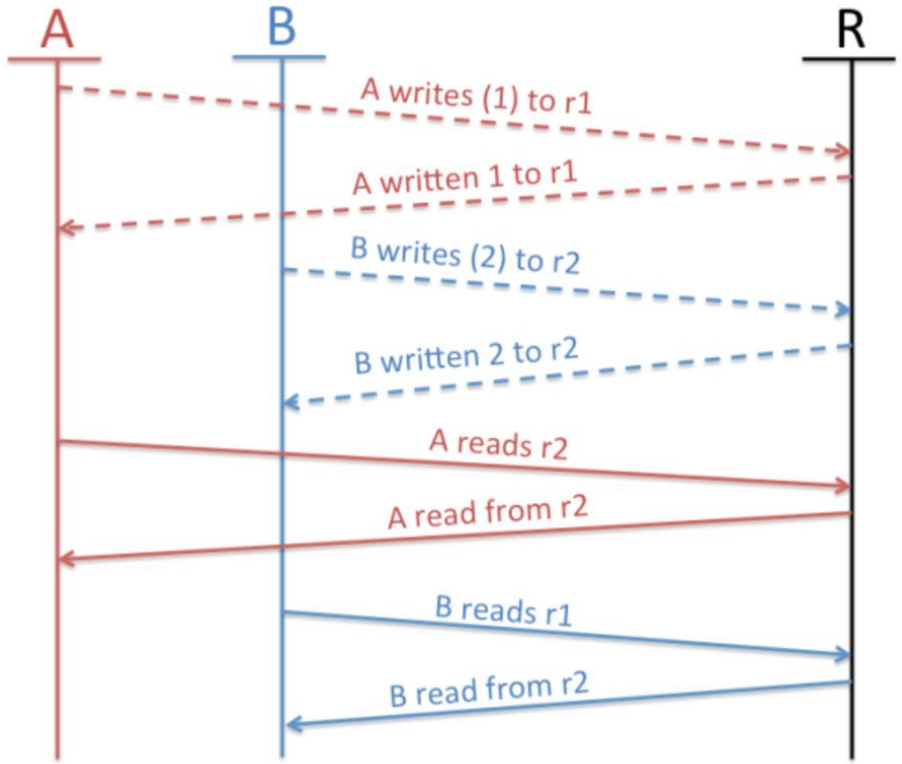


Histories for multiple concurrent registers

- Consider all possible permutations of atomic invocations
  - (That respect program order)

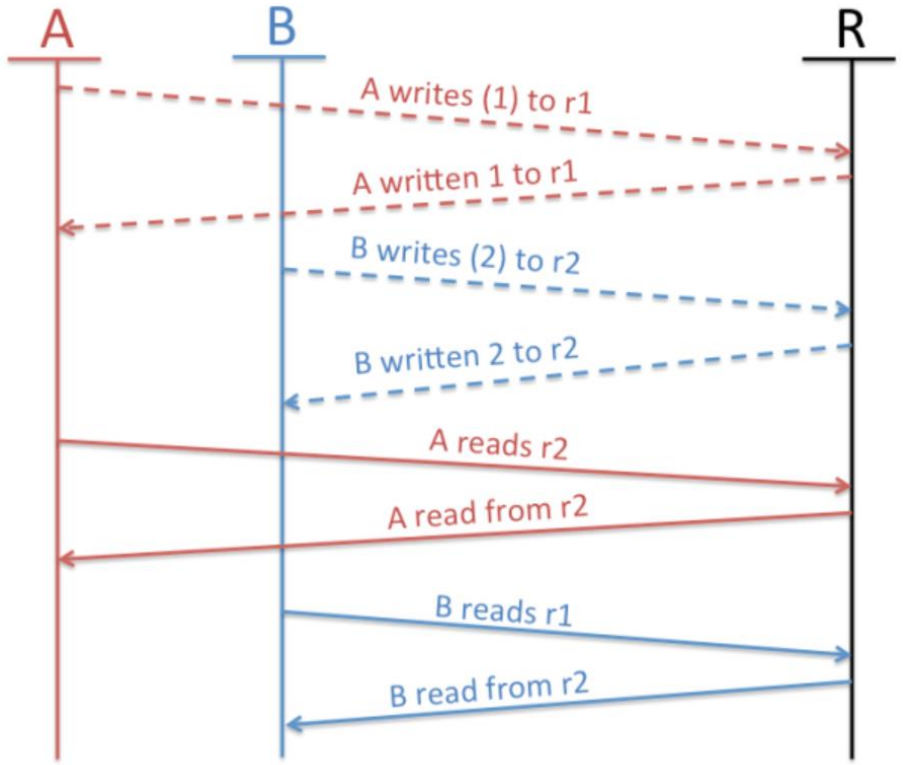
Histories for multiple concurrent registers

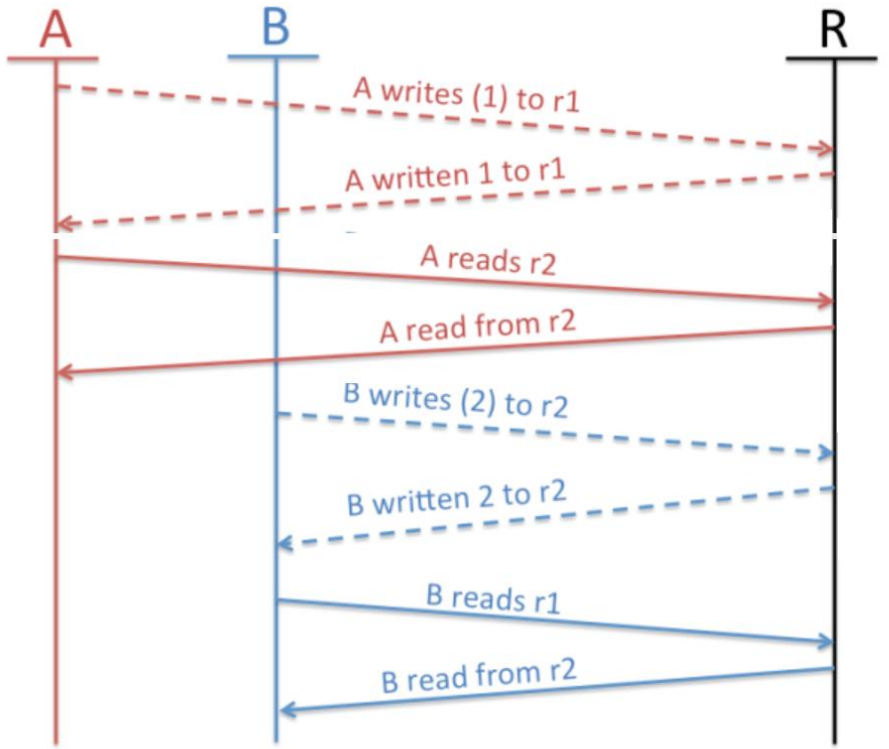
- Consider all possible permutations of atomic invocations
  - (That respect program order)



Histories for multiple concurrent registers

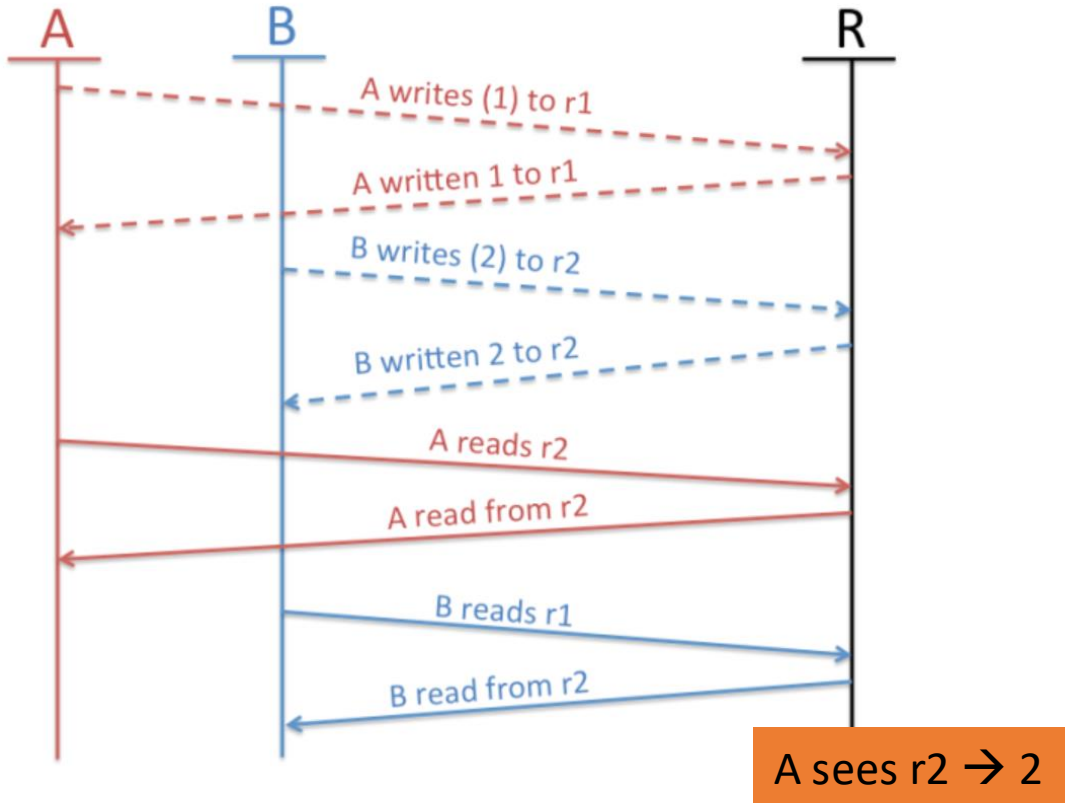
- Consider all possible permutations of atomic invocations
  - (That respect program order)

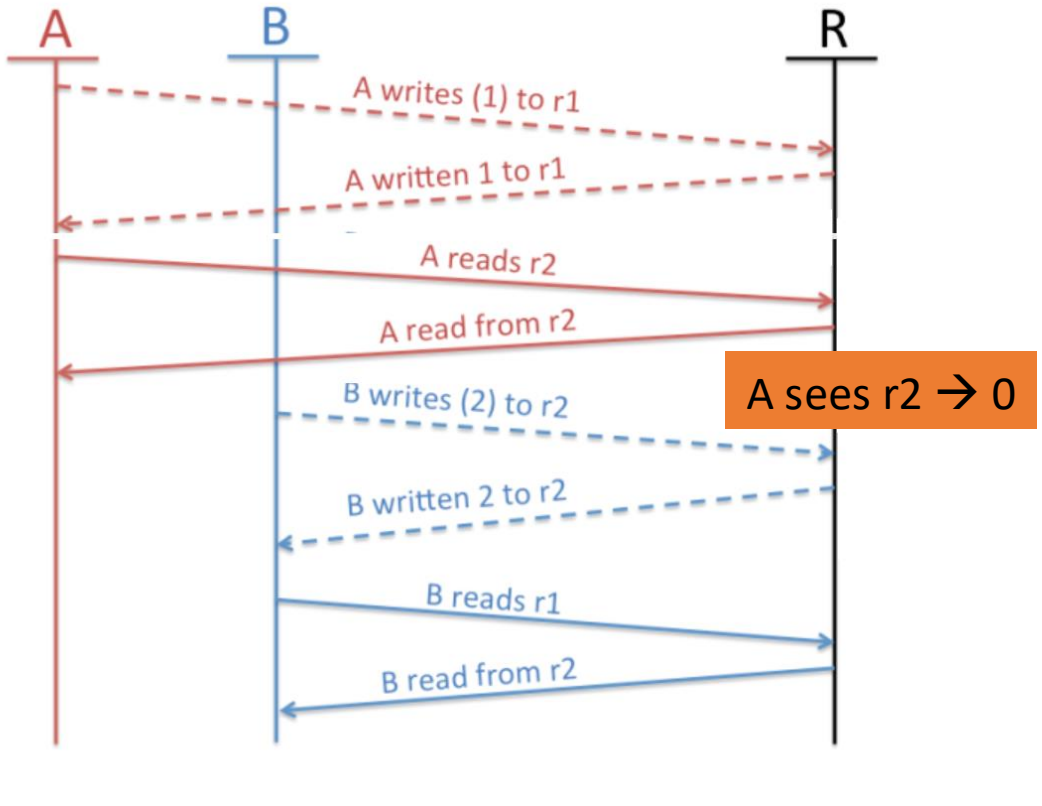




Histories for multiple concurrent registers

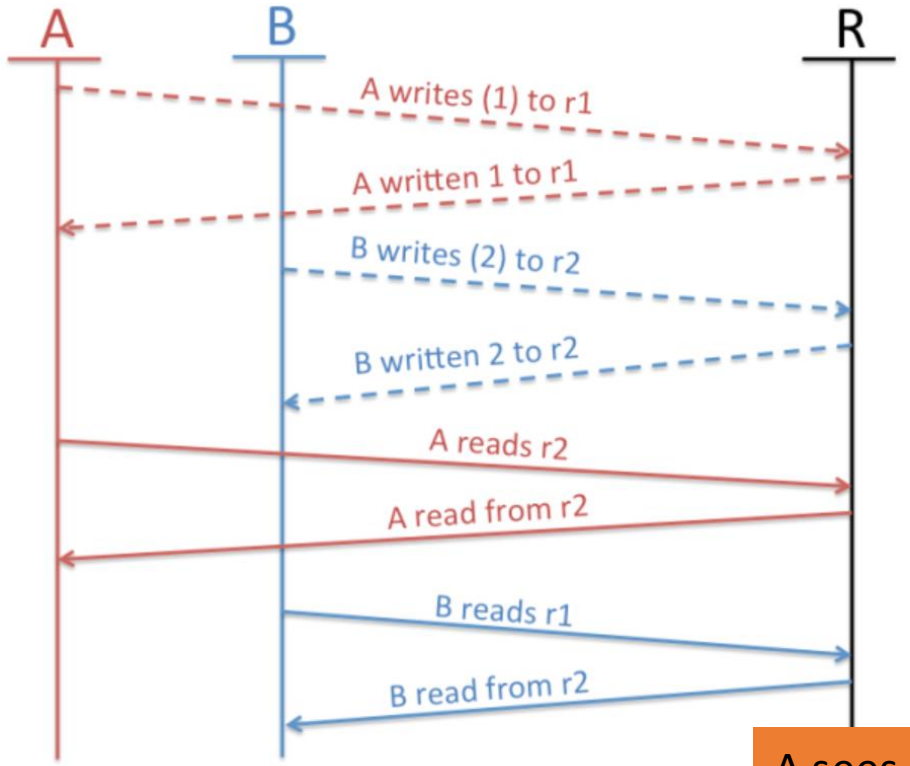
- Consider all possible permutations of atomic invocations
  - (That respect program order)



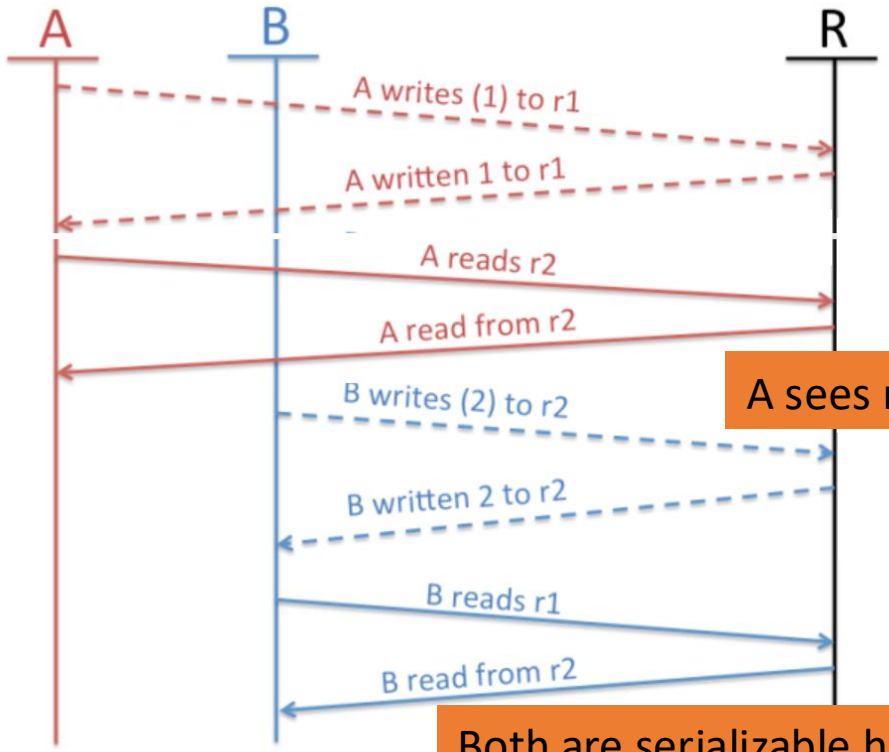


Histories for multiple concurrent registers

- Consider all possible permutations of atomic invocations
  - (That respect program order)



A sees r2 → 2



A sees r2 → 0

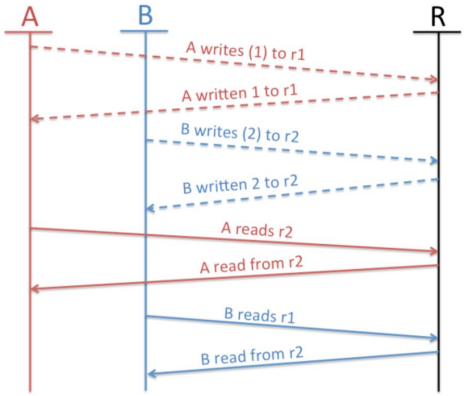
Both are serializable histories from the perspective of A

Histories for multiple concurrent registers

- Consider all possible permutations of atomic invocations
  - (That respect program order)
  - Call them “sub-histories”: from A, B “perspective”

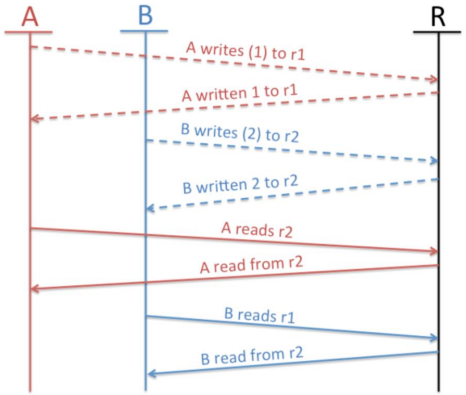
Histories for multiple concurrent registers

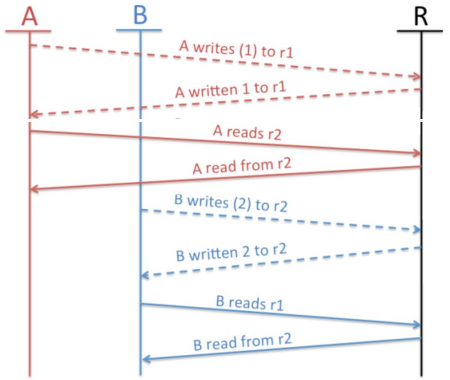
- Consider all possible permutations of atomic invocations
  - (That respect program order)
  - Call them “sub-histories”: from A, B “perspective”



Histories for multiple concurrent registers

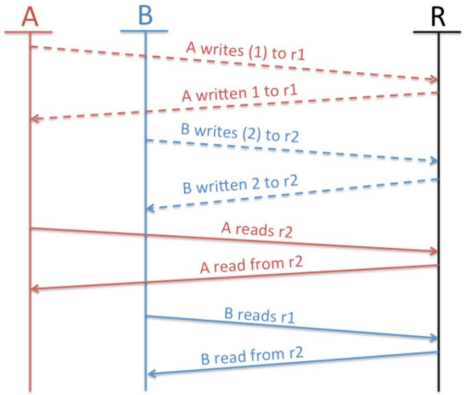
- Consider all possible permutations of atomic invocations
  - (That respect program order)
  - Call them “sub-histories”: from A, B “perspective”

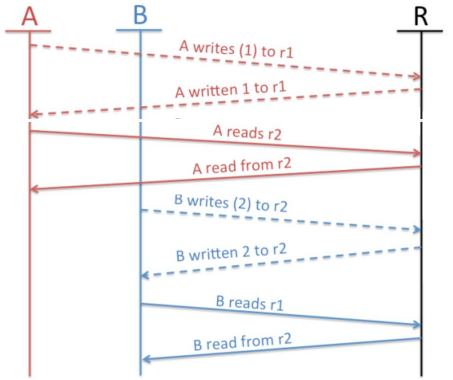




Histories for multiple concurrent registers

- Consider all possible permutations of atomic invocations
  - (That respect program order)
  - Call them “sub-histories”: from A, B “perspective”



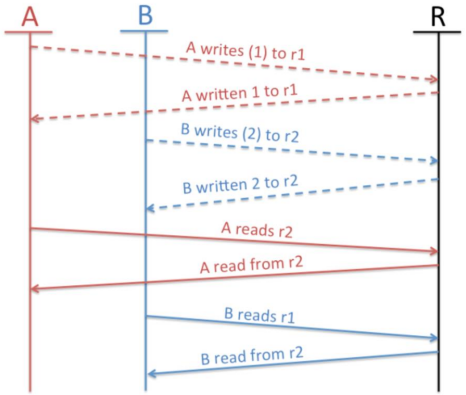


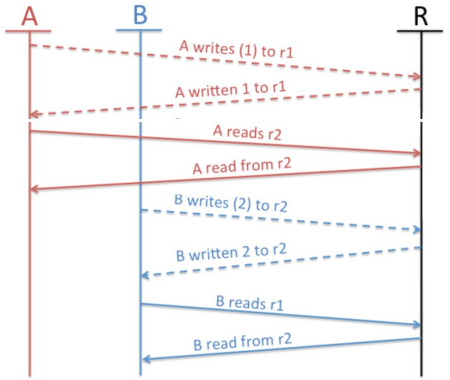
...

Histories for multiple concurrent registers

- Consider all possible permutations of atomic invocations
  - (That respect program order)
  - Call them “sub-histories”: from A, B “perspective”

Sub-History	Outcome
H1a	A writes r1=1, reads r2 → 0
H2a	A writes r1=1, reads r2 → 2
H1b	B writes r2=2, reads r1 → 0
H2b	B writes r2=2, reads r1 → 1



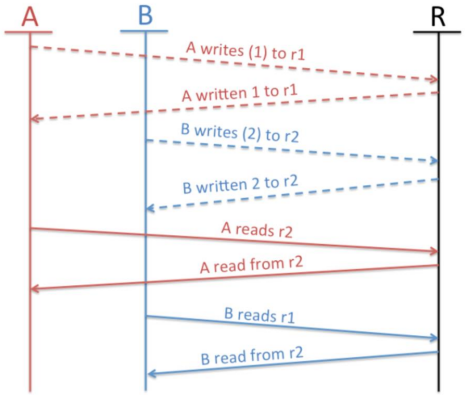


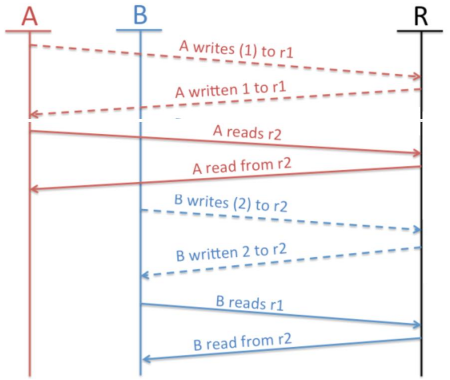
…

Histories for multiple concurrent registers

- Consider all possible permutations of atomic invocations
  - (That respect program order)
  - Call them “sub-histories”: from A, B “perspective”

Sub-History	Outcome
H1a	A writes r1=1, reads r2 → 0
H2a	A writes r1=1, reads r2 → 2
H1b	B writes r2=2, reads r1 → 0
H2b	B writes r2=2, reads r1 → 1





…

From the perspective of threads A, B, all sub-histories are serializable

- They respect program order for each of A, B
- And are equivalent to *some* serial execution
- If we “compose” these histories, some composed histories not serializable

Histories for multiple concurrent registers

Histories for multiple concurrent registers

- Compose sub-histories to form all possible histories

Histories for multiple concurrent registers

- Compose sub-histories to form all possible histories

Sub-History	Outcome
H1a	A writes r1=1, reads r2 → 0
H2a	A writes r1=1, reads r2 → 2
H1b	B writes r2=2, reads r1 → 0
H2b	B writes r2=2, reads r1 → 1

Histories for multiple concurrent registers

- Compose sub-histories to form all possible histories

Sub-History	Outcome
H1a	A writes r1=1, reads r2 → 0
H2a	A writes r1=1, reads r2 → 2
H1b	B writes r2=2, reads r1 → 0
H2b	B writes r2=2, reads r1 → 1

History	Effect
H1ab	A writes r1=1, B writes r2=2 A reads r2 → 0, B reads r1 → 0
H2ab	A writes r1=1, B writes r2=2 A reads r2 → 0, B reads r1 → 1
H3ab	A writes r1=1, B writes r2=2 A reads r2 → 2, B reads r1 → 0
H4ab	A writes r1=1, B writes r2=2 A reads r2 → 2, B reads r1 → 1

Histories for multiple concurrent registers

- Compose sub-histories to form all possible histories
- Composition of serializable histories → non-serializable histories

Sub-History	Outcome
H1a	A writes r1=1, reads r2 → 0
H2a	A writes r1=1, reads r2 → 2
H1b	B writes r2=2, reads r1 → 0
H2b	B writes r2=2, reads r1 → 1

History	Effect
H1ab	A writes r1=1, B writes r2=2 A reads r2 → 0, B reads r1 → 0
H2ab	A writes r1=1, B writes r2=2 A reads r2 → 0, B reads r1 → 1
H3ab	A writes r1=1, B writes r2=2 A reads r2 → 2, B reads r1 → 0
H4ab	A writes r1=1, B writes r2=2 A reads r2 → 2, B reads r1 → 1

Histories for multiple concurrent registers

- Compose sub-histories to form all possible histories
- Composition of serializable histories → non-serializable histories
- Ex. H1ab is not serializable

Sub-History	Outcome
H1a	A writes r1=1, reads r2 → 0
H2a	A writes r1=1, reads r2 → 2
H1b	B writes r2=2, reads r1 → 0
H2b	B writes r2=2, reads r1 → 1

History	Effect
H1ab	A writes r1=1, B writes r2=2 A reads r2 → 0, B reads r1 → 0
H2ab	A writes r1=1, B writes r2=2 A reads r2 → 0, B reads r1 → 1
H3ab	A writes r1=1, B writes r2=2 A reads r2 → 2, B reads r1 → 0
H4ab	A writes r1=1, B writes r2=2 A reads r2 → 2, B reads r1 → 1

Histories for multiple concurrent registers

- Compose sub-histories to form all possible histories
- Composition of serializable histories → non-serializable histories
- Ex. H1ab is not serializable

Sub-History	Outcome
H1a	A writes r1=1, reads r2 → 0
H2a	A writes r1=1, reads r2 → 2
H1b	B writes r2=2, reads r1 → 0
H2b	B writes r2=2, reads r1 → 1

History	Effect
H1ab	A writes r1=1, B writes r2=2 A reads r2 → 0, B reads r1 → 0
H2ab	A writes r1=1, B writes r2=2 A reads r2 → 0, B reads r1 → 1
H3ab	A writes r1=1, B writes r2=2 A reads r2 → 2, B reads r1 → 0
H4ab	A writes r1=1, B writes r2=2 A reads r2 → 2, B reads r1 → 1

4 serializable sub-histories composed
To form 4 complete histories,
Only H4ab is actually serializable

…Now back to Linearizability

- A concurrent register implementation that allows H1ab, H2ab, H3ab is not linearizable
- To be linearizable its implementation must disallow those histories
- Simplest technique to achieve this: make writes atomic (CAS)
  - such that overlapping reads cannot be reordered (e.g. A reads r2->0 after B writes r2=2
  - ***Equivalent to adding constraint that operations respect real time order of operations***

…Now back to Linearizability

History	Effect
H1ab	A writes r1=1, B writes r2=2 reads r2 → 0, B reads r1 → 0
H2ab	A writes r1=1, B writes r2=2 reads r2 → 0, B reads r1 → 1
H3ab	A writes r1=1, B writes r2=2 reads r2 → 2, B reads r1 → 0
H4ab	A writes r1=1, B writes r2=2 reads r2 → 2, B reads r1 → 1

- A concurrent register implementation that allows H1ab, H2ab, H3ab is not linearizable
- To be linearizable its implementation must disallow those histories
- Simplest technique to achieve this: make writes atomic (CAS)
  - such that overlapping reads cannot be reordered (e.g. A reads r2->0 after B writes r2=2
  - ***Equivalent to adding constraint that operations respect real time order of operations***

…Now back to Linearizability

History	Effect
H1ab	A writes r1=1, B writes r2=2 reads r2 → 0, B reads r1 → 0
H2ab	A writes r1=1, B writes r2=2 reads r2 → 0, B reads r1 → 1
H3ab	A writes r1=1, B writes r2=2 reads r2 → 2, B reads r1 → 0
H4ab	A writes r1=1, B writes r2=2 reads r2 → 2, B reads r1 → 1

This history IS linearizable

- A concurrent register implementation that allows H1ab, H2ab, H3ab is not linearizable
- To be linearizable its implementation must disallow those histories
- Simplest technique to achieve this: make writes atomic (CAS)
  - such that overlapping reads cannot be reordered (e.g. A reads r2->0 after B writes r2=2
  - ***Equivalent to adding constraint that operations respect real time order of operations***

…Now back to Linearizability

These histories are NOT linearizable

This history IS linearizable

History	Effect
H1ab	A writes r1=1, B writes r2=2 reads r2 → 0, B reads r1 → 0
H2ab	A writes r1=1, B writes r2=2 reads r2 → 0, B reads r1 → 1
H3ab	A writes r1=1, B writes r2=2 reads r2 → 2, B reads r1 → 0
H4ab	A writes r1=1, B writes r2=2 reads r2 → 2, B reads r1 → 1

- A concurrent register implementation that allows H1ab, H2ab, H3ab is not linearizable
- To be linearizable its implementation must disallow those histories
- Simplest technique to achieve this: make writes atomic (CAS)
  - such that overlapping reads cannot be reordered (e.g. A reads r2->0 after B writes r2=2
  - ***Equivalent to adding constraint that operations respect real time order of operations***

Another Perspective

Serializability
invocations and responses can be reordered to yield a sequential history
the sequential history is correct according to the sequential definition of the object

Linearizability
invocations and responses can be reordered to yield a sequential history
the sequential history is correct according to the sequential definition of the/each object
if a response preceded an invocation in the original history, it must still precede it in the sequential reordering

Linearizability Redux

- non-blocking
  - one method is never forced to wait to sync with another.
- applies to histories *and* objects *and* systems:
  - a history can be linearizable
  - an object is linearizable if all valid histories of its use are linearizable
- local property:
  - a system is linearizable iff each individual object is linearizable.
  - gives us **composability**.

Practical difficulties:

- Key-value mapping
- Population count
- Iteration
- Resizing the bucket array

Practical difficulties:

- Key-value ma
- Population co
- Iteration
- Resizing the h

Options to consider when implementing a “difficult” operation:

Practical difficulties:

- Key-value ma
- Population co
- Iteration
- Resizing the k

Options to consider when implementing a “difficult” operation:

Relax the semantics
(e.g., non-exact count, or non-linearizable count)

Practical difficulties:

- Key-value ma
- Population co
- Iteration
- Resizing the h

Options to consider when implementing a “difficult” operation:

Relax the semantics
(e.g., non-exact count, or non-linearizable count)

Fall back to a simple implementation if permitted
(e.g., lock the whole table for resize)

Practical difficulties:

- Key-value ma
- Population co
- Iteration
- Resizing the h

Options to consider when implementing a "difficult" operation:

- Relax the semantics
  (e.g., non-exact count, or non-linearizable count)
- Fall back to a simple implementation if permitted
  (e.g., lock the whole table for resize)
- Design a clever implementation
  (e.g., split-ordered lists)

Practical difficulties:

- Key-value ma
- Population co
- Iteration
- Resizing the h

Options to consider when implementing a "difficult" operation:

- Relax the semantics
  (e.g., non-exact count, or non-linearizable count)
- Fall back to a simple implementation if permitted
  (e.g., lock the whole table for resize)
- Design a clever implementation
  (e.g., split-ordered lists)
- Use a different data structure
  (e.g., skip lists)