CPSC 4240/5240: Parallel Programming Techniques

Lecture 9: Concurrent Data Structures

Lecturer: Quanquan C. Liu

Spring 2026

Compare-and-Swap (CAS)

- A type of test-and-set
- **Used very often** in parallel code
- **Compares** the contents of a memory location to an expected value
- If they match, then **swap** the contents of the memory location with new desired value
- Return true if successfully performed
- Technically **CAS is not a lock** but sometimes can act like one

Counters with CAS

Why is this more preferable to spinlocks?

```
#include <atomic>
#include <thread>
#include <iostream>
#include <vector>

static std::atomic<int> sharedCounter{0};

void worker(int threadId) {
    for (int i = 0; i < 1000; ++i) {
        int oldVal = sharedCounter.load(std::memory_order_relaxed);
        int newVal = oldVal + 1;

    while (!sharedCounter.compare_exchange_weak(
            oldVal,
            newVal,
            std::memory_order_release,
            std::memory_order_relaxed))
        {
            oldVal = sharedCounter.load(std::memory_order_relaxed);
            newVal = oldVal + 1;
        }
    }
}
```

```
int main() {
    std::vector<std::thread> threads;
    for (int i = 0; i < 4; ++i) {
        threads.emplace_back(worker, i);
    }

    for (auto &t : threads) {
        t.join();
    }

    std::cout << sharedCounter.load() << std::endl;
    return 0;
}
```

Counters with CAS

Does not take up CPU time while spinning!

```
#include <atomic>
#include <thread>
#include <iostream>
#include <vector>

static std::atomic<int> sharedCounter{0};

void worker(int threadId) {
    for (int i = 0; i < 1000; ++i) {
        int oldVal = sharedCounter.load(std::memory_order_relaxed);
        int newVal = oldVal + 1;

    while (!sharedCounter.compare_exchange_weak(
            oldVal,
            newVal,
            std::memory_order_release,
            std::memory_order_relaxed))
        {
            oldVal = sharedCounter.load(std::memory_order_relaxed);
            newVal = oldVal + 1;
        }
    }
}
```

```
int main() {
    std::vector<std::thread> threads;
    for (int i = 0; i < 4; ++i) {
        threads.emplace_back(worker, i);
    }

    for (auto &t : threads) {
        t.join();
    }

    std::cout << sharedCounter.load() << std::endl;
    return 0;
}
```

Advantages and Disadvantages of CAS

- **Lock-Free/Non-Blocking**
  - A thread can update shared data without acquiring a mutex or other blocking mechanism
  - Avoid deadlocks and offer better scalability
- **Fine-Grained Concurrency**
  - Multiple threads can concurrently update different data elements
  - More parallelism
- **Better Performance**
  - Low contention and short critical sections—better performance than mutex

Fetch-and-Add

- Return old value and increment it
- This is yet another atomic instruction for concurrency
- Can be used to atomically reserve a
  "ticket number"
- "`lock; xadd`" in x86 assembly

Fetch-and-Add

- fetch_add(1) on an std::atomic<int> is an **atomic read-modify-write** operation that proceeds as follows:
  - **Reads the current value** of the atomic integer (let's call this oldValue).
  - **Computes the new value** by adding 1 to oldValue (i.e., oldValue + 1).
  - **Writes this new value** back into the atomic integer.

Used often in parallel and concurrent programming!

Counters with Fetch-and-Add

```
#include <atomic>
#include <thread>
#include <iostream>
#include <vector>

static std::atomic<int> sharedCounter{0};

void worker(int threadId) {
    for (int i = 0; i < 1000; ++i) {
        // Atomically increment the shared counter
        sharedCounter.fetch_add(1, std::memory_order_relaxed);
    }
}
```

```
int main() {
    std::vector<std::thread> threads;
    threads.reserve(4);

    for (int i = 0; i < 4; ++i) {
        threads.emplace_back(worker, i);
    }

    for (auto &t : threads) {
        t.join();
    }

    std::cout << sharedCounter.load() << std::endl;
    return 0;
}
```

Timing Our Methods

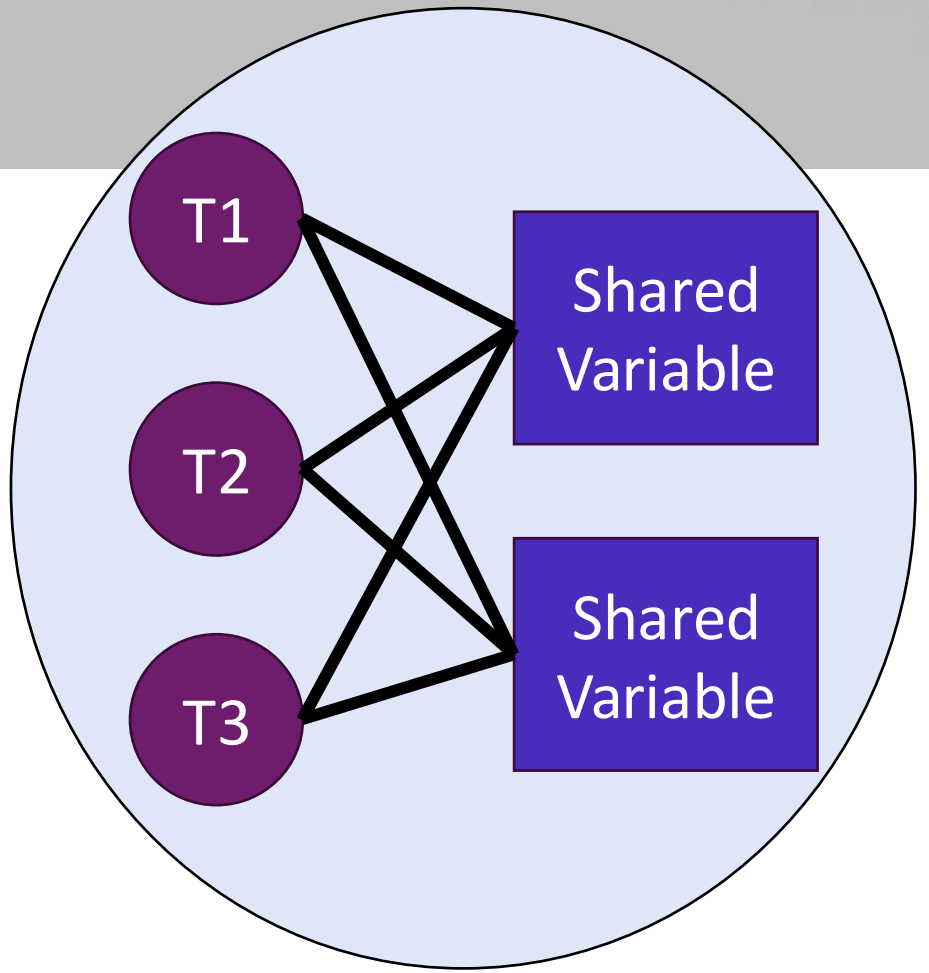
[Mutex] Median of 5 trials: **19.2411 ms**
[CAS] Median of 5 trials: **43.5042 ms**
[Fetch-and-Add] Median of 5 trials: **0.200666 ms**
[Spinlock] Median of 5 trials: **59.5665 ms**

How to evaluate a lock implementation?

- ***Correctness*** – must provide mutual exclusion
- ***Fairness*** – threads acquire lock in the order they request it
- ***Progress*** – if several threads request the lock, one must acquire it
  - (avoid *deadlock*)
- ***Bounded wait*** – no thread should wait forever (or starve).
- ***Performance*** – minimize latency/overhead introduced by the lock

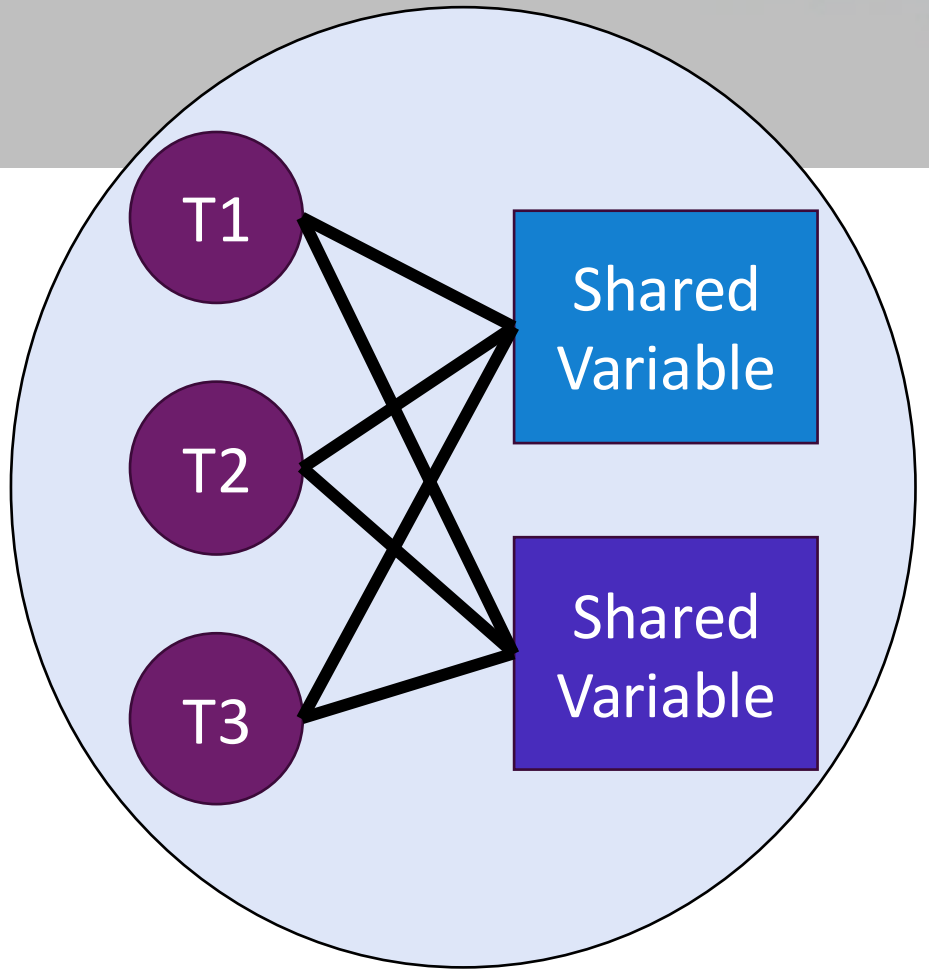
Asynchronous Shared Memory

- Threads communicate through shared variables
- **Adversarial scheduler** interleaves instructions from different threads
- Threads can be **arbitrarily slow** or **crash**



Asynchronous Shared Memory

- **Read-Write**: Read(x), Write(X, value)
- **Lock**: Lock(L), Unlock(L)
- **Compare-and-Swap (CAS)**: CAS(X, oldValue, newValue)



Analyzing Correctness: Linearizability

- Ensuring that a concurrent operation behaves **as if only one procedure** is executing the operation
- A process is **linearizable** if all operations:
  - Can be executed **instantaneously at some point** between its invocation and response
  - There exists a **sequential order** of these instantaneous points that is a valid sequential ordering
- Concurrent process is correct if this sequential linearization is correct

Analyzing Correctness: Linearizability

Procedure 1: Read(x)

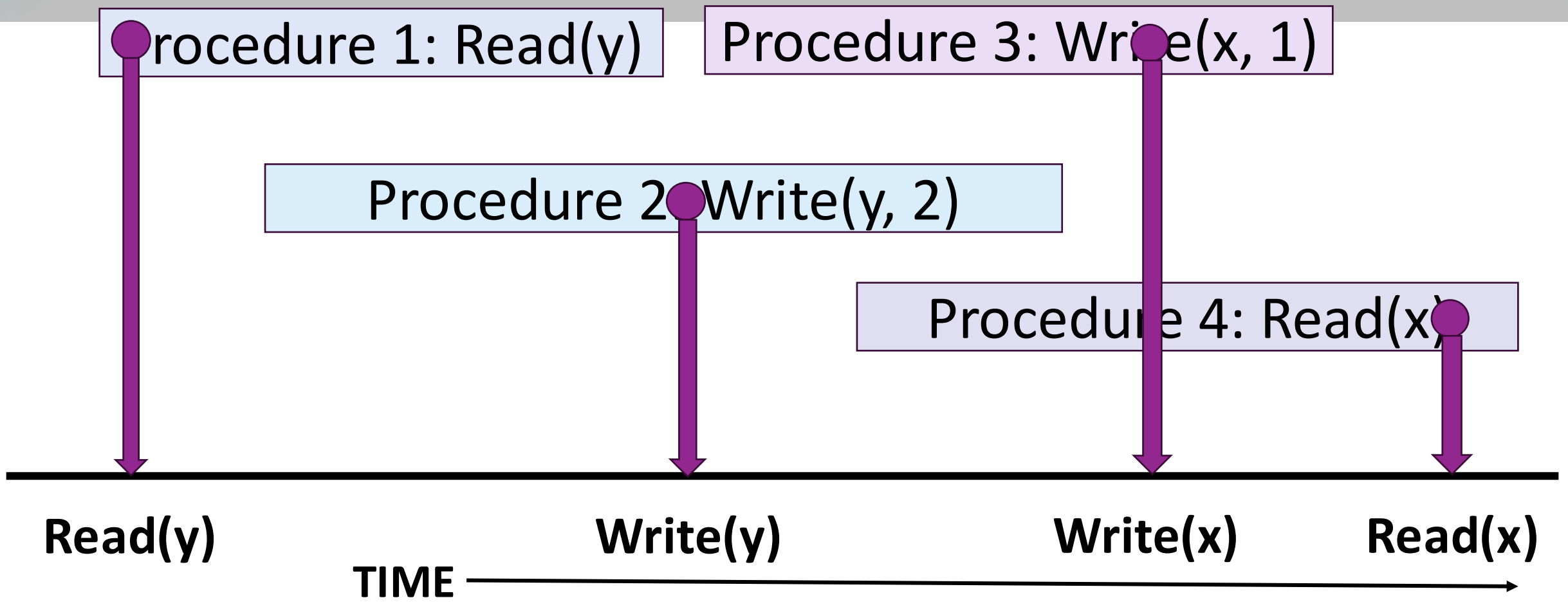
Procedure 3: Write(x, 1)

Procedure 2: Write(y, 2)

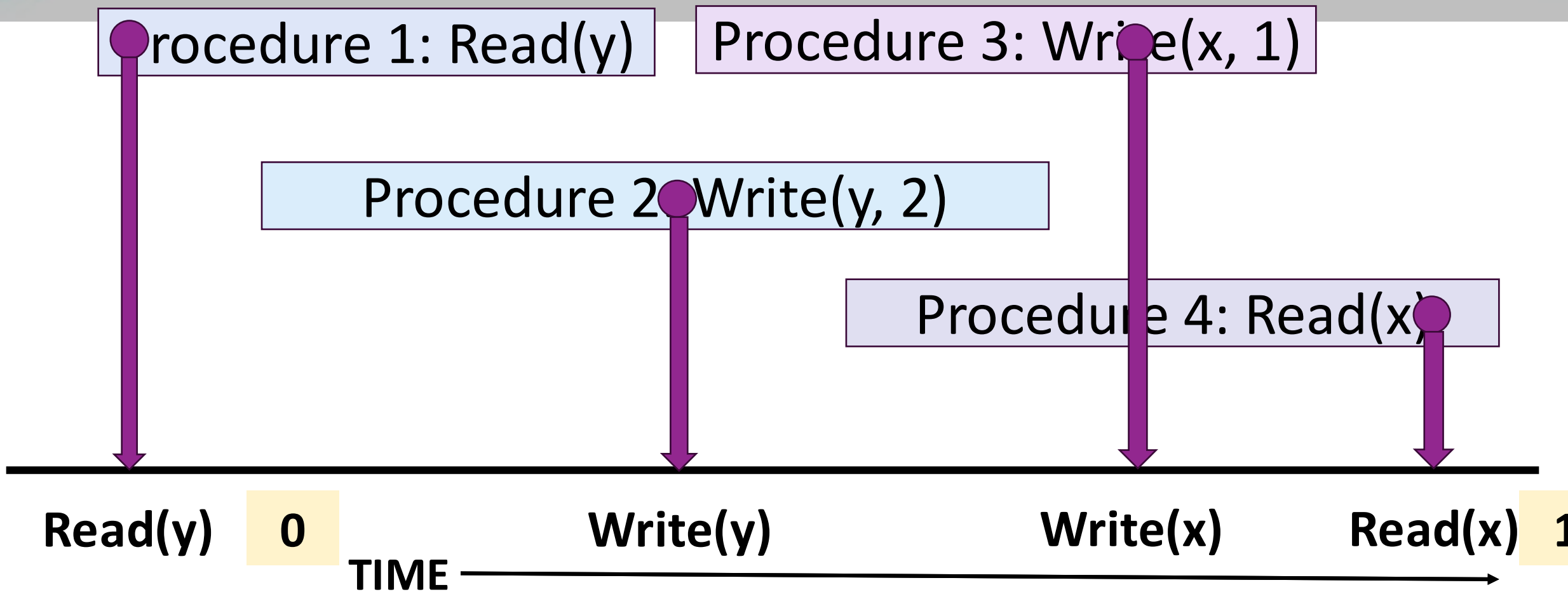
Procedure 4: Read(x)

Initially x and y both contain 0

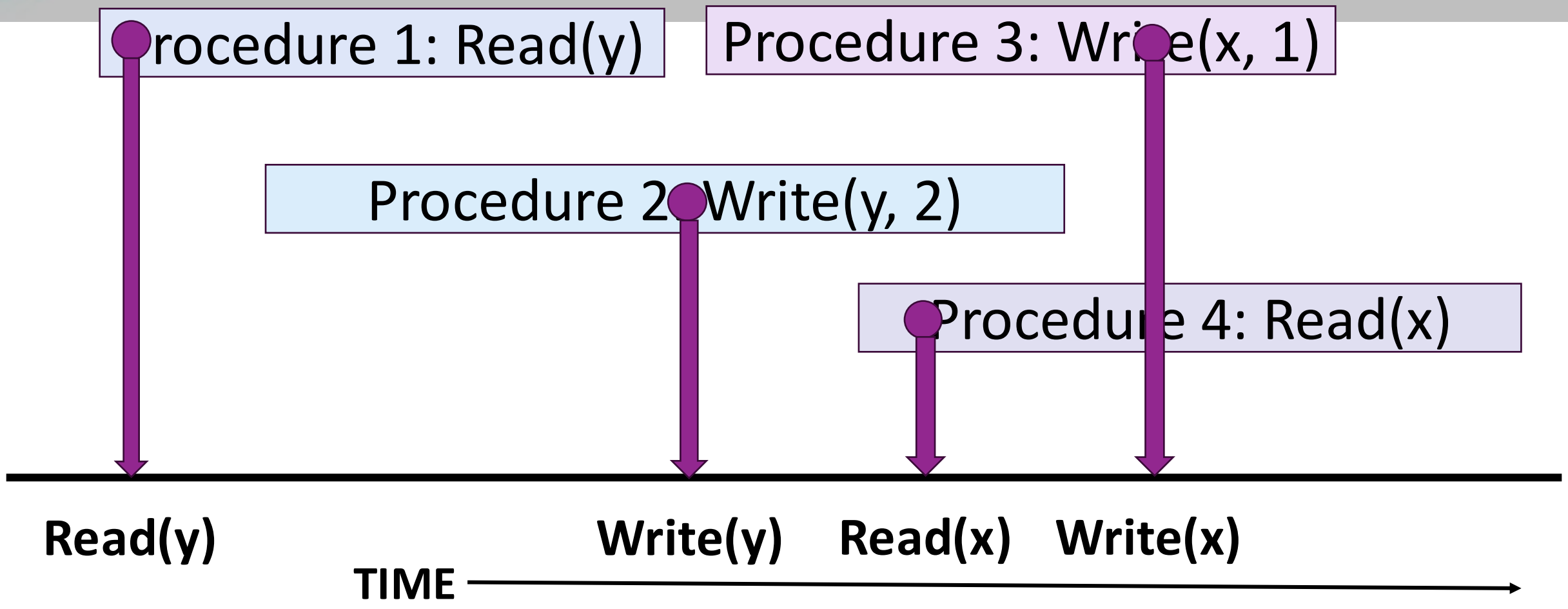
Analyzing Correctness: Linearizability



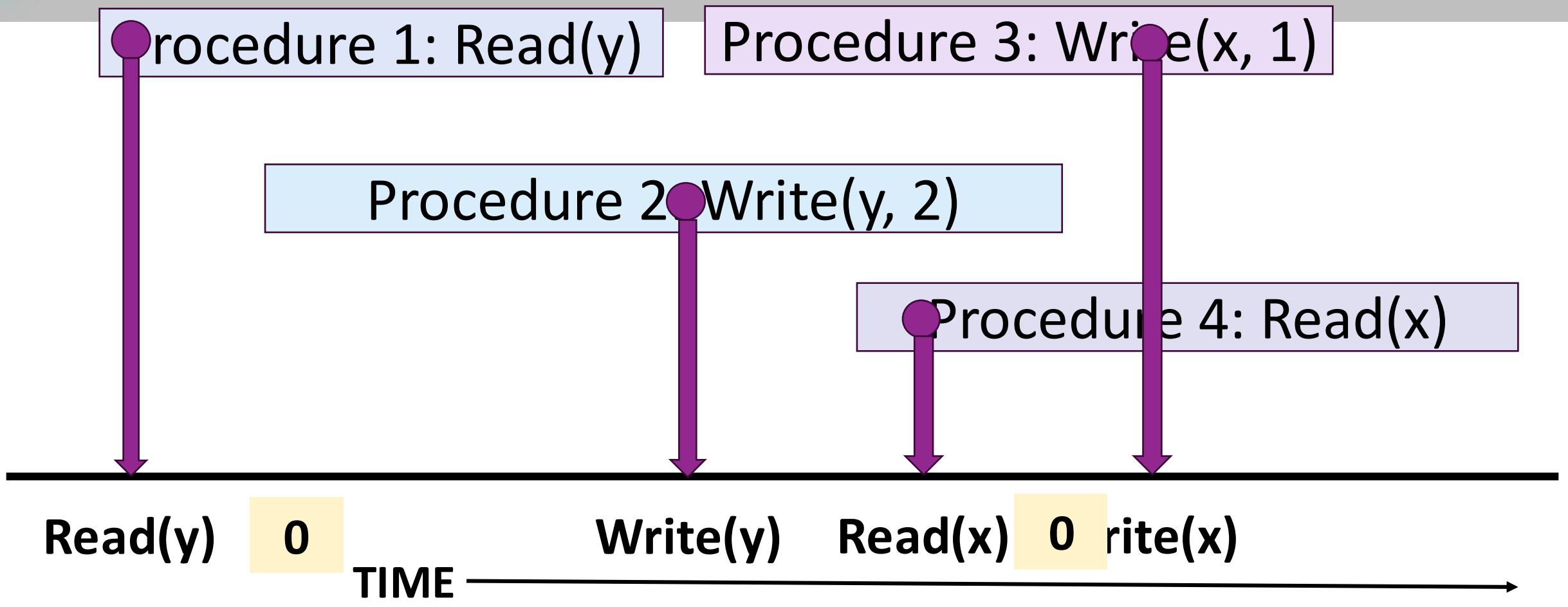
Analyzing Correctness: Linearizability



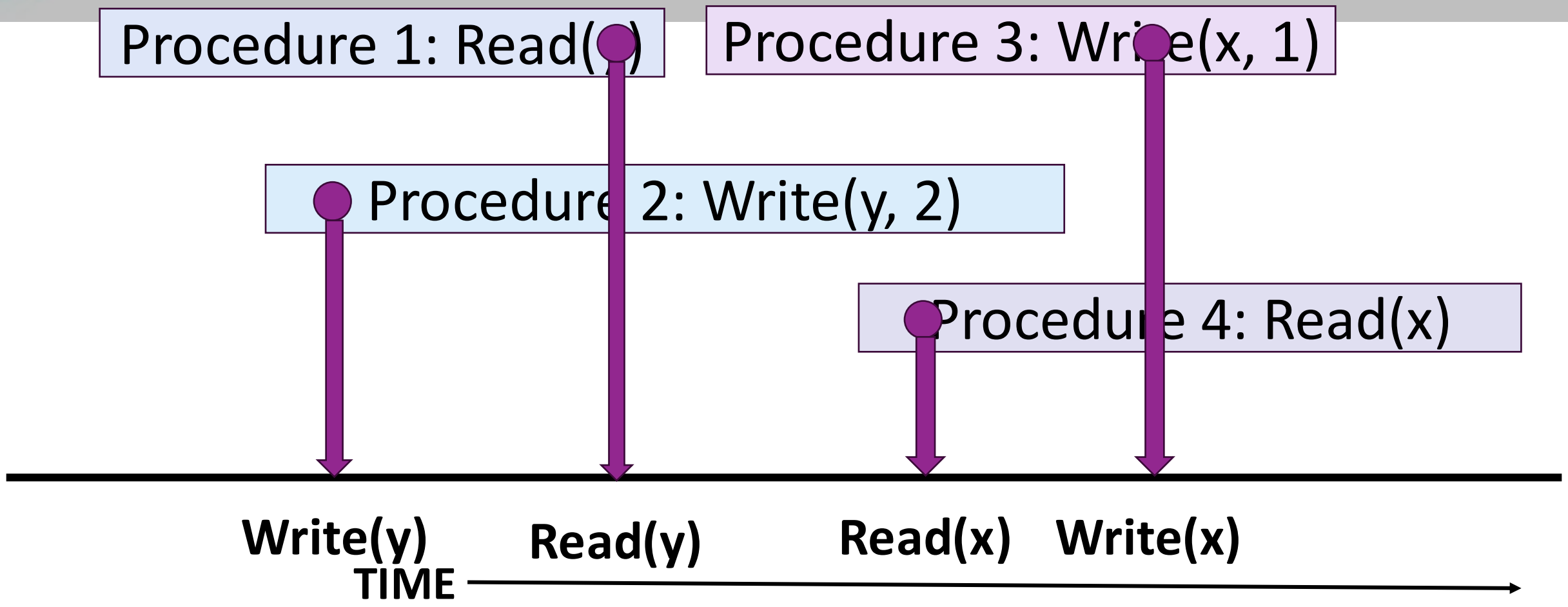
Analyzing Correctness: Linearizability



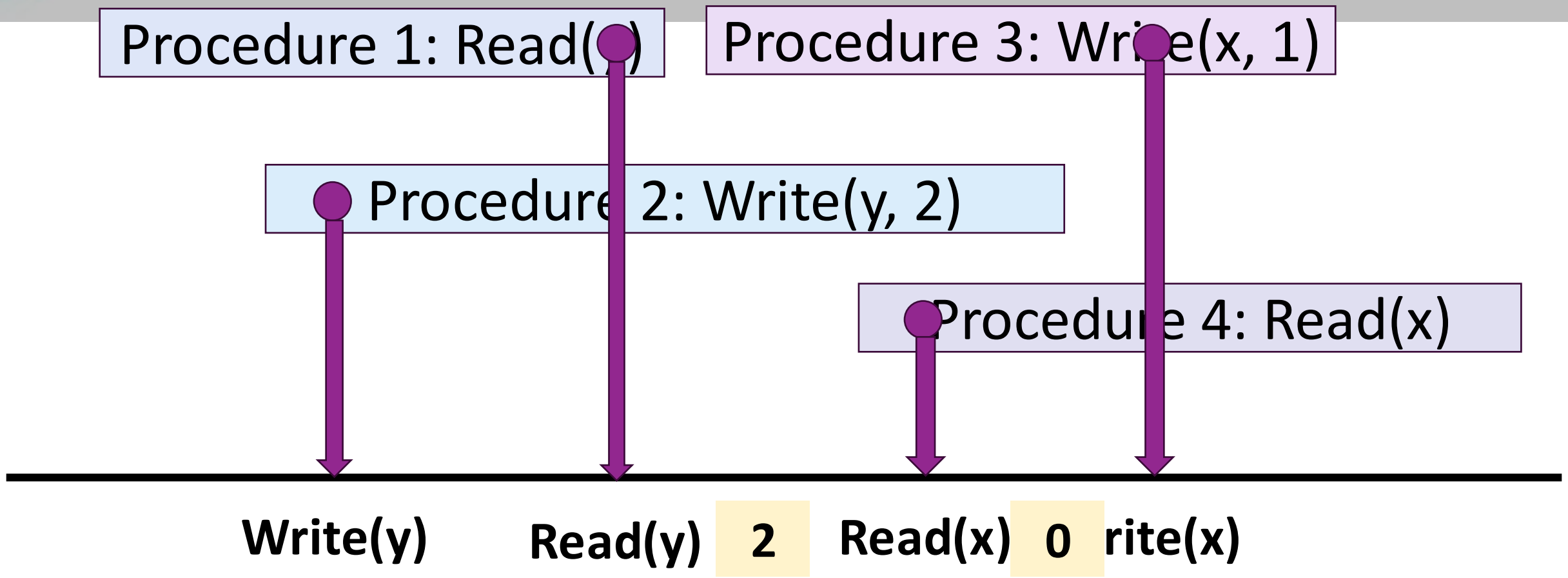
Analyzing Correctness: Linearizability



Analyzing Correctness: Linearizability

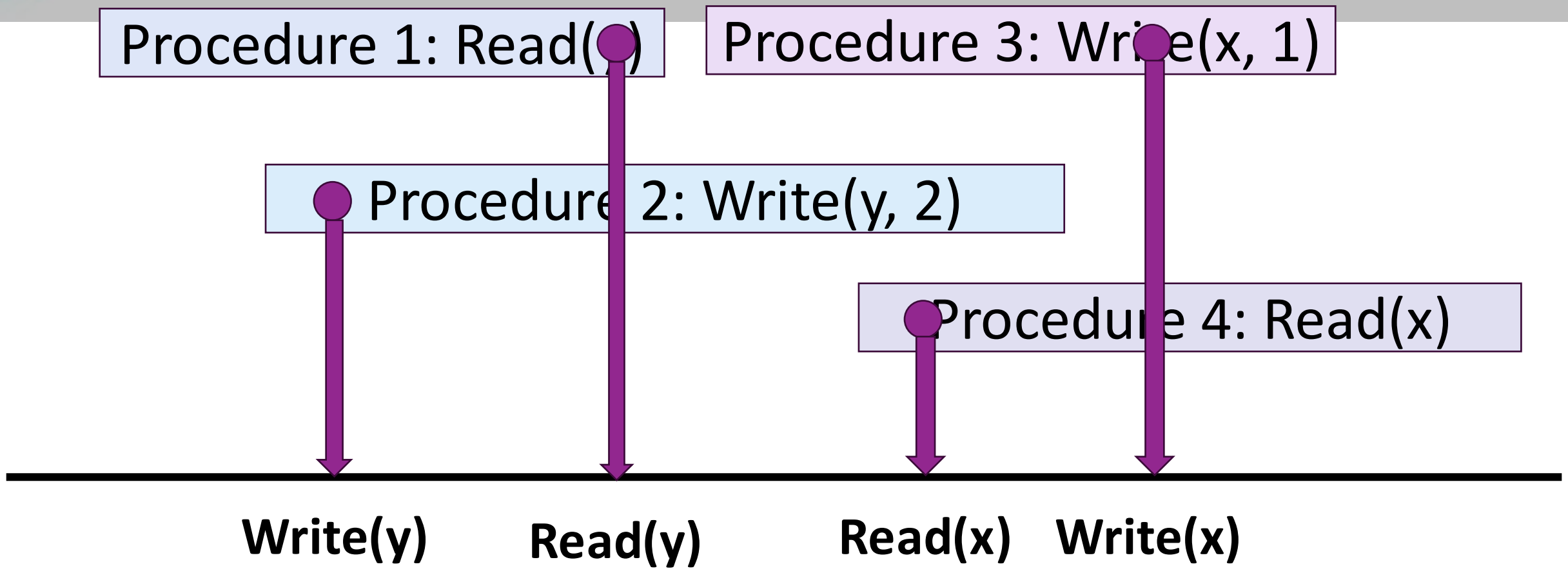


Analyzing Correctness: Linearizability



Analyzing Correctness: Linearizability

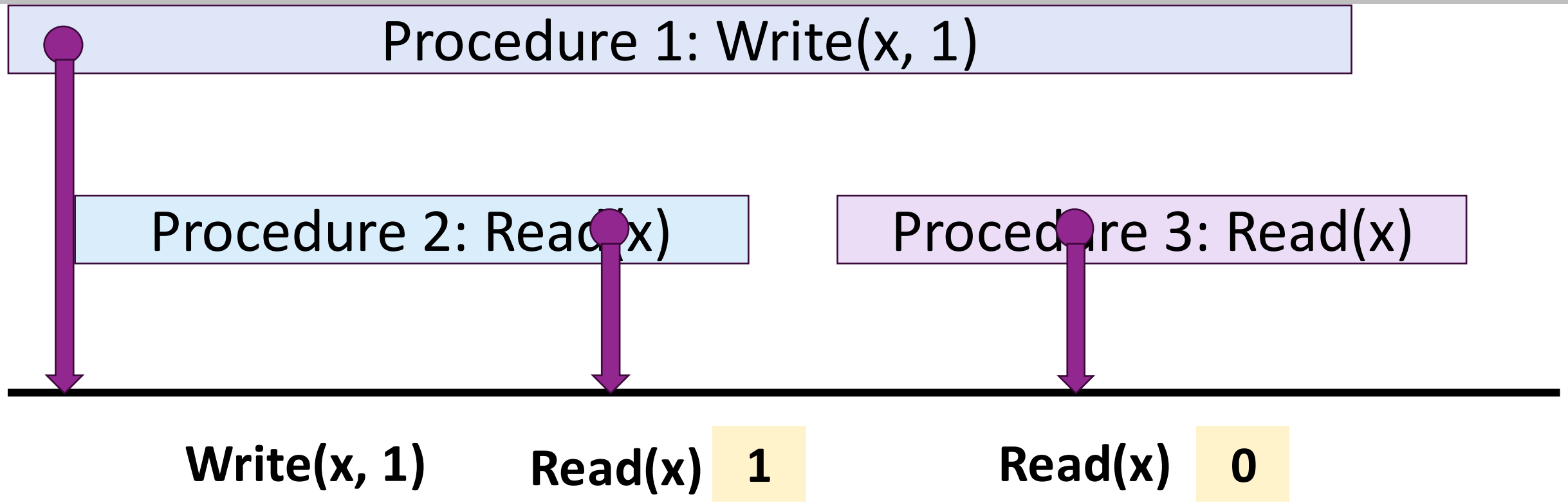
Point is called **linearization point**



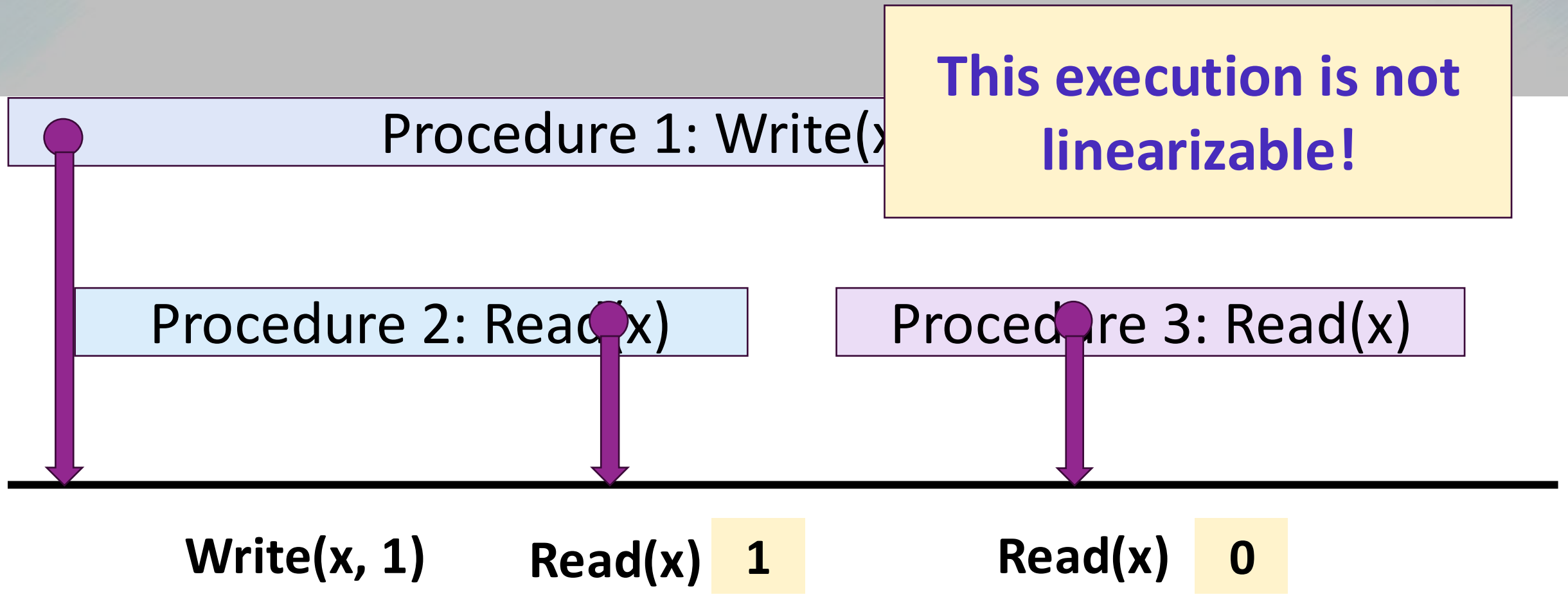
Analyzing Correctness: Linearizability

- Ensuring that a concurrent operation behaves **as if only one Procedure** is executing the operation
- A process is **linearizable** if all operations:
  - Can be executed **instantaneously at some point** between its invocation and response
  - There exists a **sequential order** of these instantaneous points that is a valid sequential ordering
- Concurrent process is correct if this sequential linearization is correct

Analyzing Correctness: Linearizability

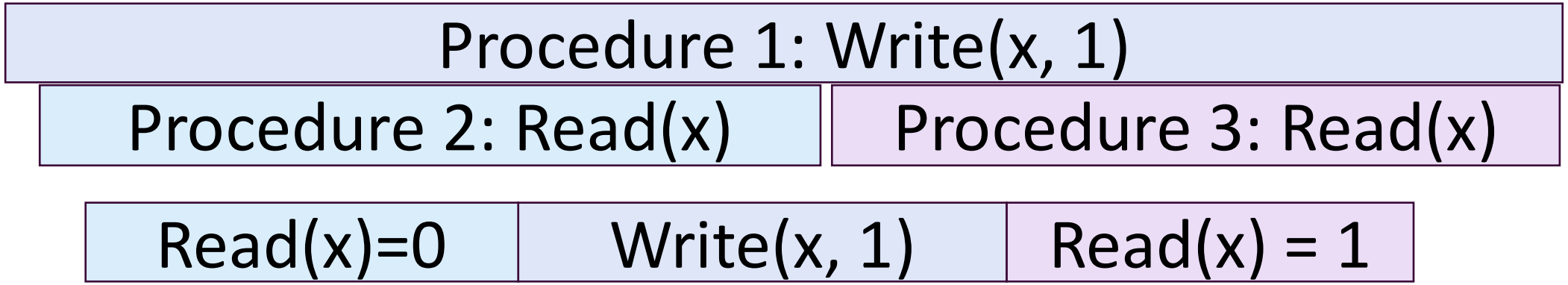


Analyzing Correctness: Linearizability



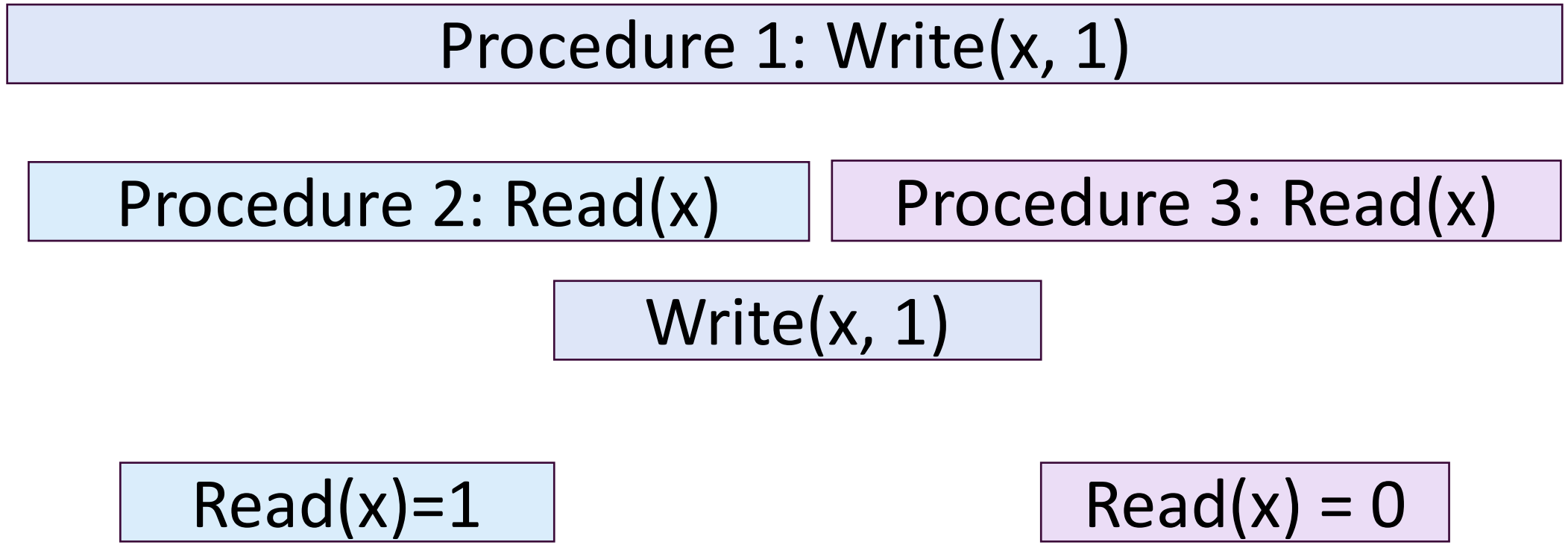
Linearizability Correctness

- **Sequential consistency** ensures a concurrent process can be ordered such that:
  - Operation appear to be applied sequentially, according to this order
  - Order is consistent with the program order of each Procedure



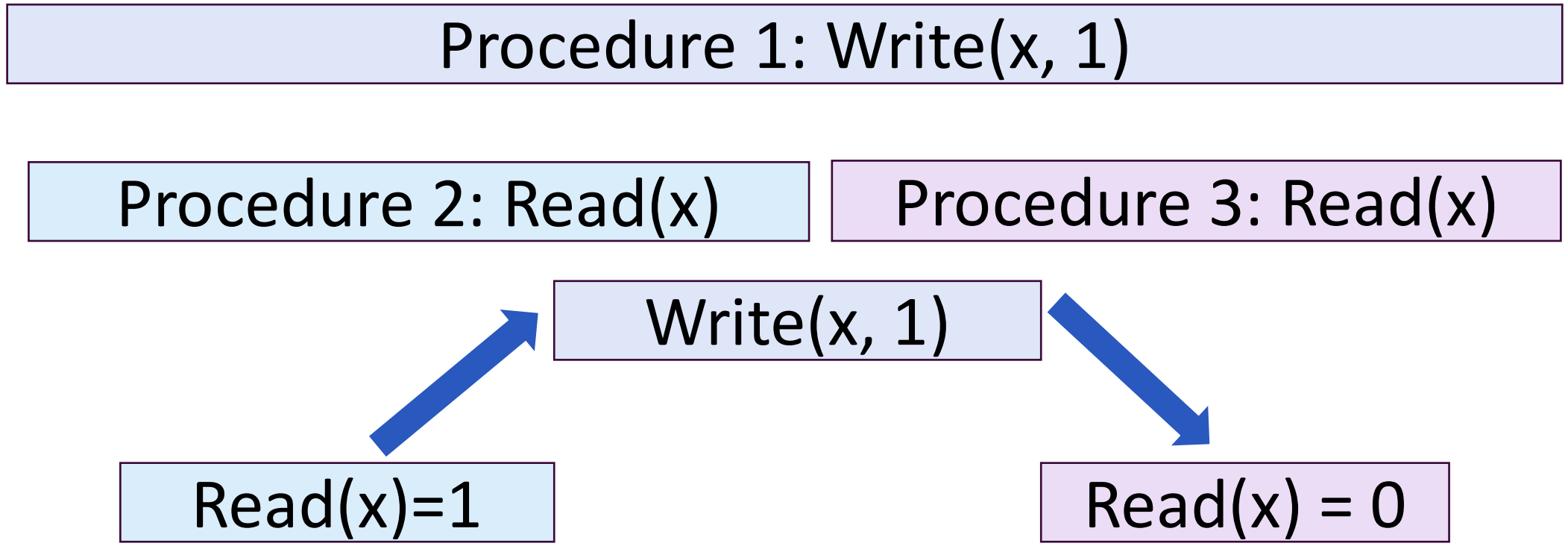
Linearizability Correctness

- Below is not valid!



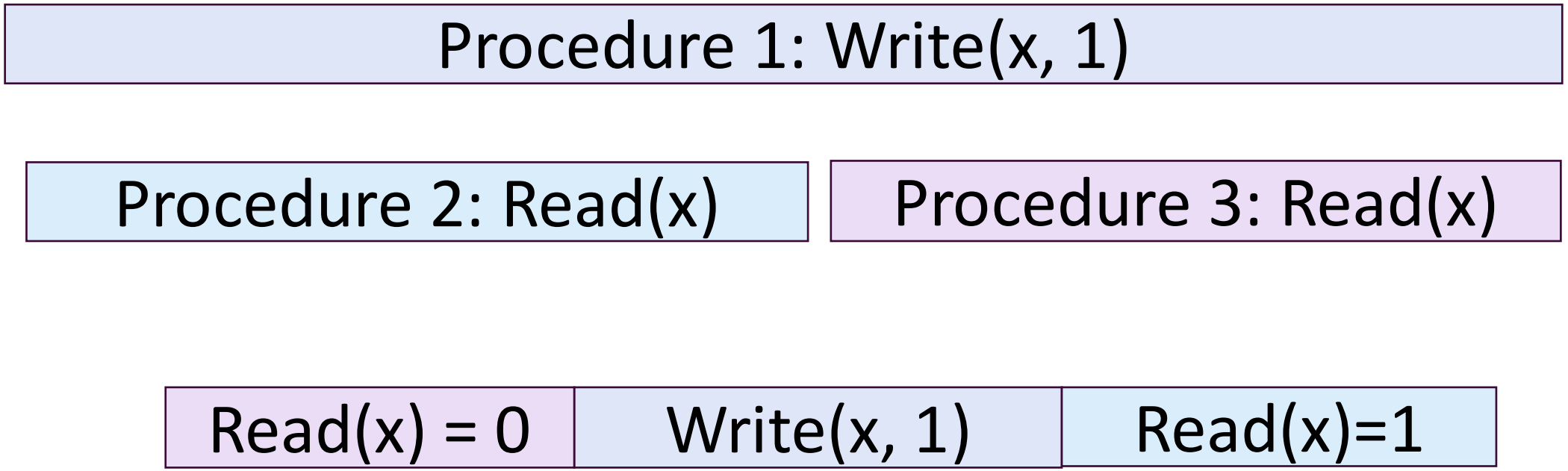
Linearizability Correctness

- Below is not valid!



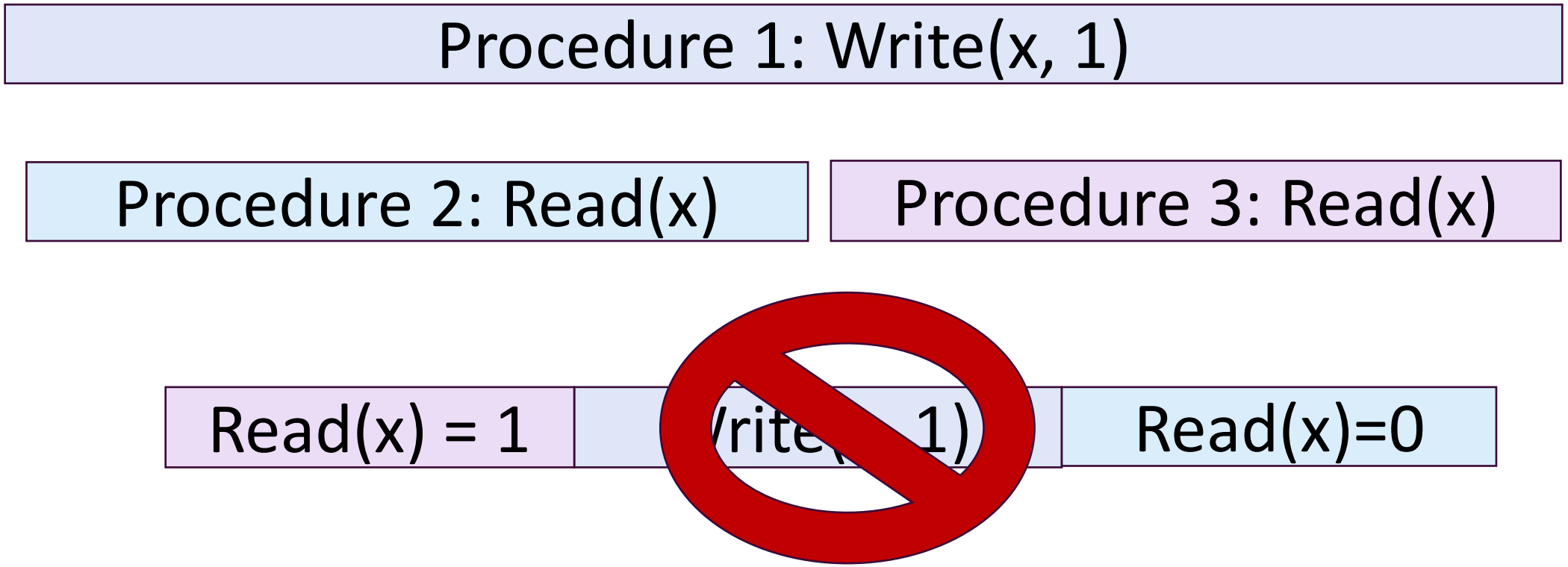
Linearizability Correctness

- Below is not valid!



Linearizability Correctness

- Below is not valid!



Which of these are linearizable executions or not?

Procedure 1: Write(x, 1)

Procedure 2: Write(y, 2)

Procedure 3: Read(y)

Procedure 4: Read(x)

- A) Read(y) returns 0, Read(x) returns 0
- B) Read(y) returns 0, Read(x) returns 1
- C) Read(y) returns 2, Read(x) returns 1
- D) Read(y) returns 2, Read(x) returns 0

Which of these are correct linearizations or not?

Procedure 1: Write(x, 1)

Procedure 2: Write(y, 2)

Procedure 3: Read(y)

Procedure 4: Read(x)

- A) Read(y) returns 0, Read(x) returns 0
- C) Read(y) returns 2, Read(x) returns 1
- B) Read(y) returns 0, Read(x) returns 1
- D) Read(y) returns 2, Read(x) returns 0

Why do we care about linearizability?

- Provides a **simple and intuitive mental model** for reasoning about concurrent operations
  - Consider atomic operations; better for thinking about how threads interleave operations
  - Predictable order for non-overlapping operations
  - Can be used to prove correctness
  - Can be used to **compose multiple linearizable objects**

Concurrent Data Structures

Implementations of data structures that are correct and allow for concurrent operations

Concurrent Set

```
template <typename T>
class Set {
public:
    bool add(const T& x);
    bool remove(const T& x);
    bool contains(const T& x);
};
```

- Concurrent set on T with three methods:
  - **Add element x into set**
  - **Remove element x from set**
  - **Check whether set contains x**

We will implement using a concurrent linked list!

Concurrent Linked List

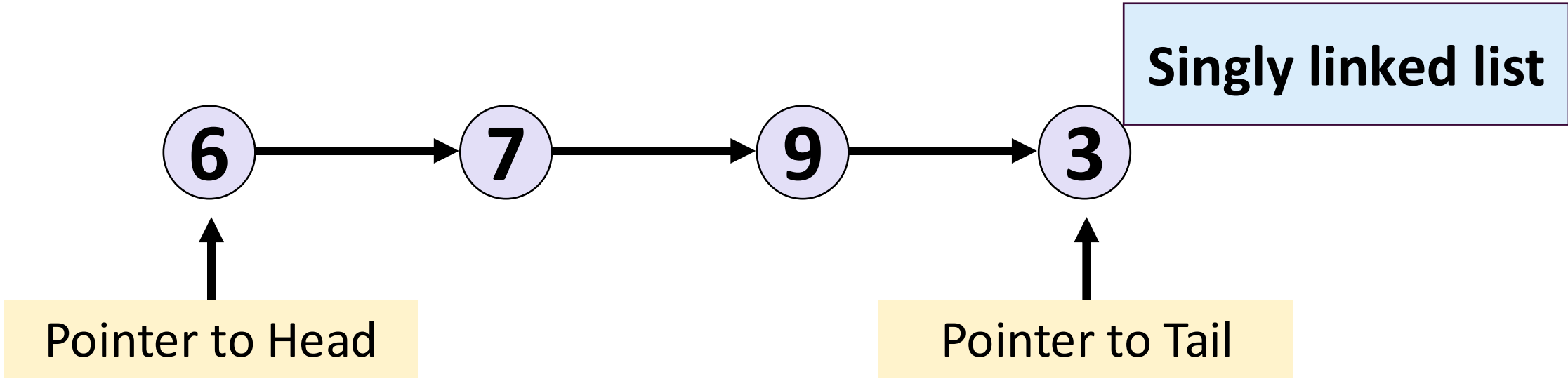
```
template <typename T>
class Set {
public:
    bool add(const T& x);
    bool remove(const T& x);
    bool contains(const T& x);
};
```

- Concurrent set on T with three methods:
  - **Add element x into set**
  - **Remove element x from set**
  - **Check whether set contains x**

Can generalize to other pointer-based structures including binary trees, b-trees, radix trees, etc.

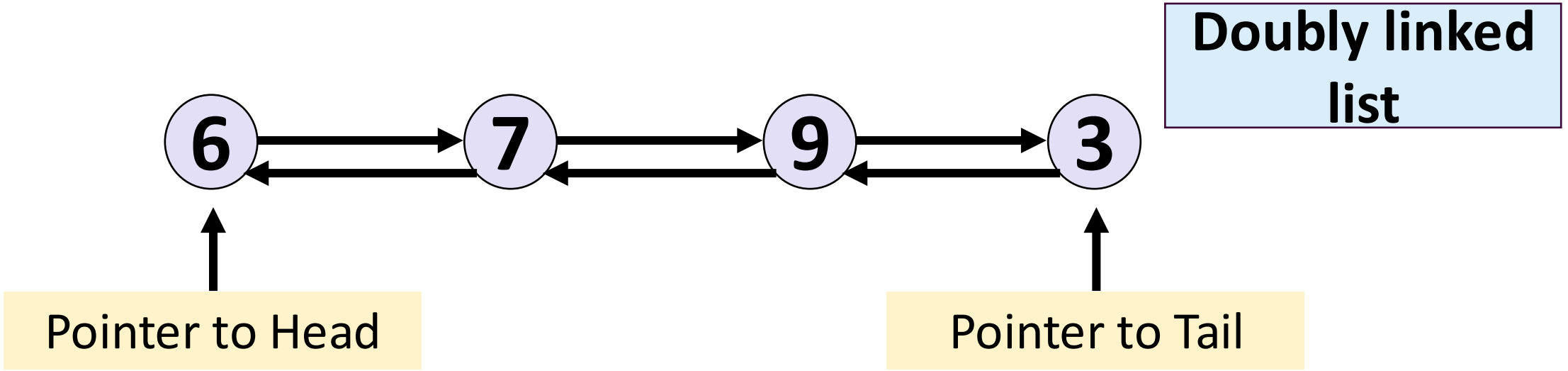
Linked Lists Interlude

- Data structure containing **nodes** and **links**
  - Each **node holds data**
  - Each link is a **pointer (or reference)** to the next node



Linked Lists Interlude

- Data structure containing **nodes** and **links**
  - Each **node holds data**
  - Each link is a **pointer (or reference)** to the next node



Linked Lists Interlude: Operations

- **Insert a node**:
  - Create a new node
  - Adjust pointers to correctly contain new node

Insert 2 between 6 and 7

Linked Lists Interlude: Operations

- **Insert a node**:
  - Create a new node
  - Adjust pointers to correctly contain new node

Insert 2 between 6 and 7

Linked Lists Interlude: Operations

- **Insert a node**:
  - Create a new node
  - Adjust pointers to correctly contain new node

Insert 2 between 6 and 7

Linked Lists Interlude: Operations

- **Delete a node**:
  - Locate desired deleted node
  - Update pointers to correctly maintain linked list

Delete node 7

Linked Lists Interlude: Operations

- **Delete a node**:
  - Locate desired deleted node
  - Update pointers to correctly maintain linked list

Delete node 7

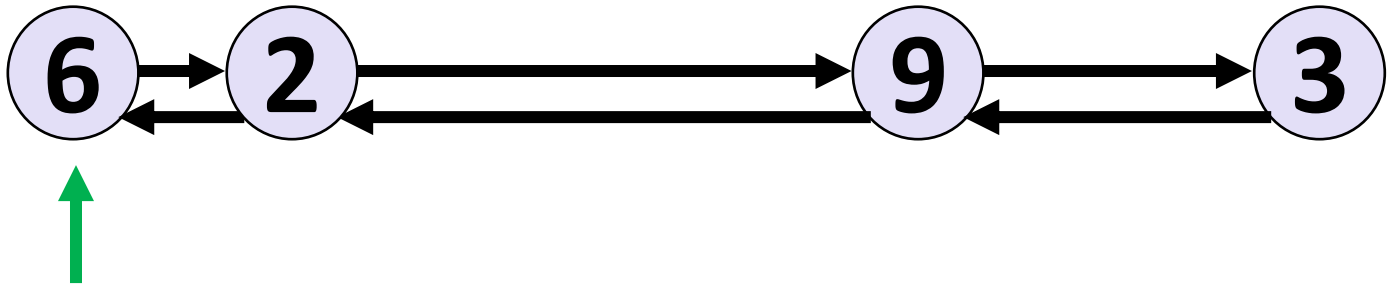
Linked Lists Interlude: Operations

- **Delete a node**:
  - Locate desired deleted node
  - Update pointers to correctly maintain linked list

Delete node 7

Linked Lists Interlude: Operations

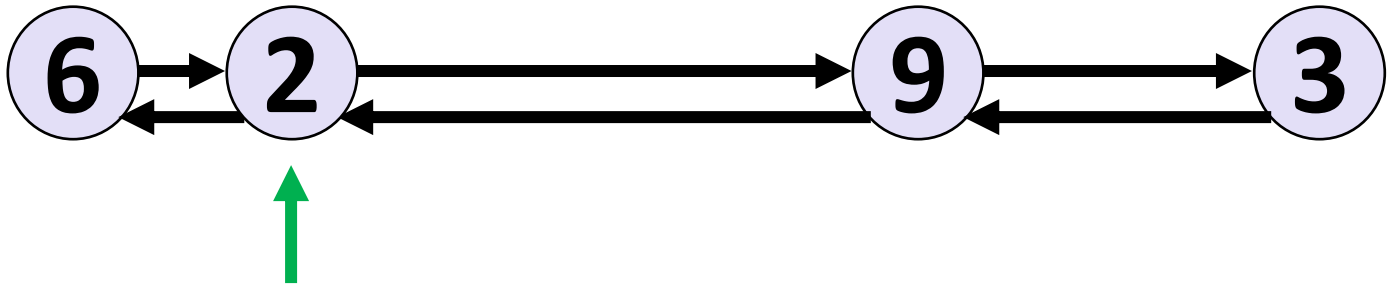
- **Find a node**:
  - Start at head and traverse the list using pointers till you find the element



Find 9

Linked Lists Interlude: Operations

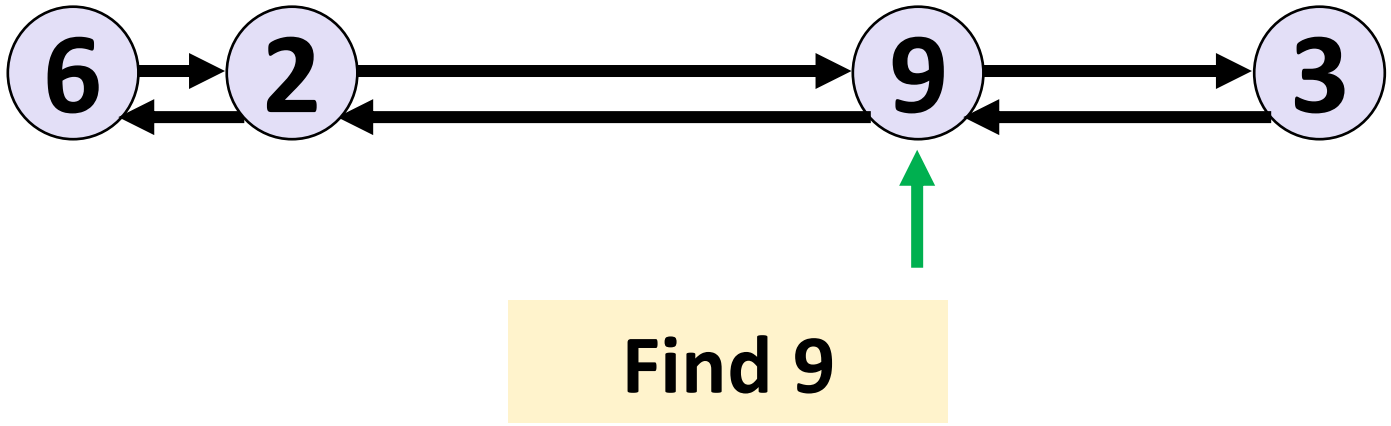
- **Find a node**:
  - Start at head and traverse the list using pointers till you find the element



Find 9

Linked Lists Interlude: Operations

- **Find a node**:
  - Start at head and traverse the list using pointers till you find the element



Linked Lists Interlude: Operations

- **Find a node**:
  - Start at head and traverse the list using pointers till you find the element