CPSC 4240/5240: Parallel Programming Techniques

Lecture 10: Concurrent Data Structures Continued

Lecturer: Quanquan C. Liu

Spring 2026

Homework 2 is Out

- Due **next Thursday, late day Monday, March 2nd**
- See Files/Homework 2

Concurrent Linked List: Challenges

- Consider the following linked list and scenario
- Thread 1 wants to **remove C**
- Thread 2 wants to **add D before E**

What could possibly happen here?

Concurrent Linked List: Challenges

- Consider the following scenario
- Thread 1 wants to **remove C**
- Thread 2 wants to **add D before**

Example execution:

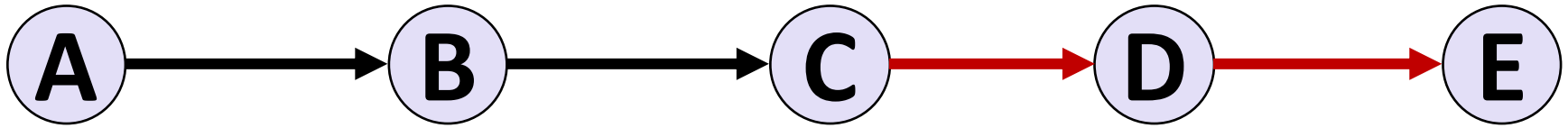
1) Thread 1 removes pointer to E **and pauses**

Concurrent Linked List: Challenges

- Consider the following scenario
- Thread 1 wants to **remove C**
- Thread 2 wants to **add D befor**

Example execution:

1) Thread 1 removes pointer to E **and pauses**
2) Thread 2 adds D

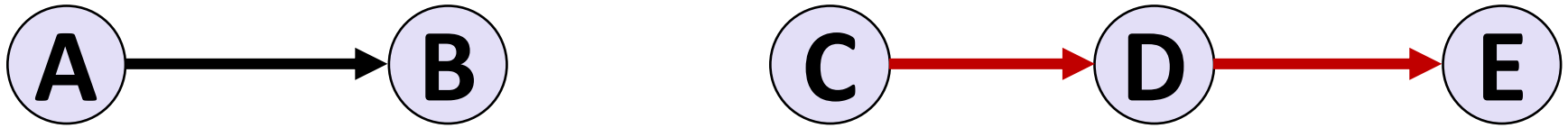


Concurrent Linked Lis

- Consider the following scenario
- Thread 1 wants to **remove C**
- Thread 2 wants to **add D before**

Example execution:

1) Thread 1 removes pointer to E **and pauses**
2) Thread 2 adds D
3) Thread 1 unpauses and removes pointer from B and adds pointer from B to E



Concurrent Linked Lis

- Consider the following scenario
- Thread 1 wants to **remove C**
- Thread 2 wants to **add D before**

Example execution:

1) Thread 1 removes pointer to E **and pauses**
2) Thread 2 adds D
3) Thread 1 unpauses and removes pointer from B and adds pointer from B to E

Lots of issues!

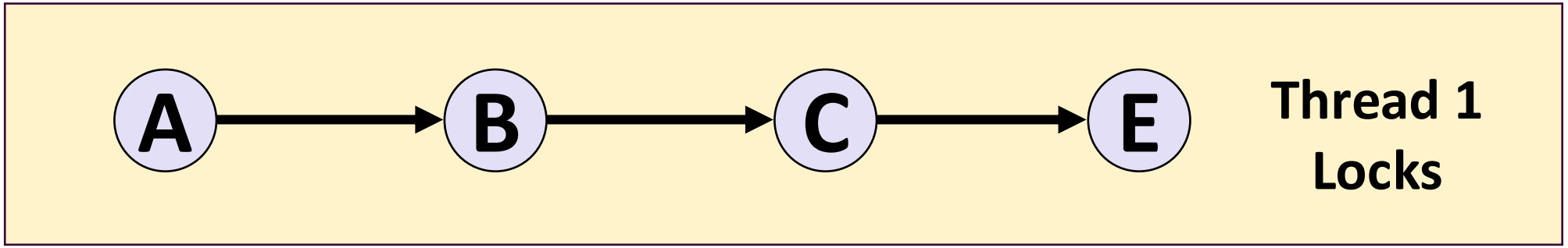
- Dangling pointer from D from empty memory address
  - Two pointers into E
  - No way to get to D

Example execution:

1) Thread 1 removes pointer to E **and pauses**
2) Thread 2 adds D
3) Thread 1 unpauses and removes pointer from B and adds pointer from B to E

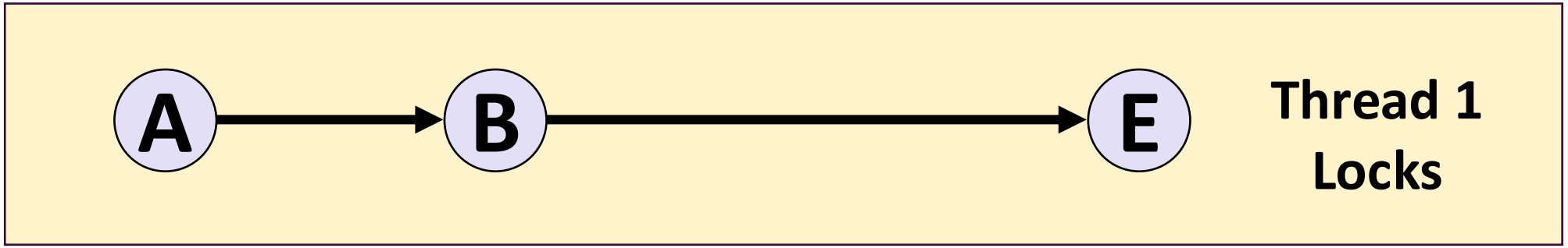
Concurrent Linked List using Locks

- Can just lock the entire linked list when you perform any operation
  - **Course-grained locking**



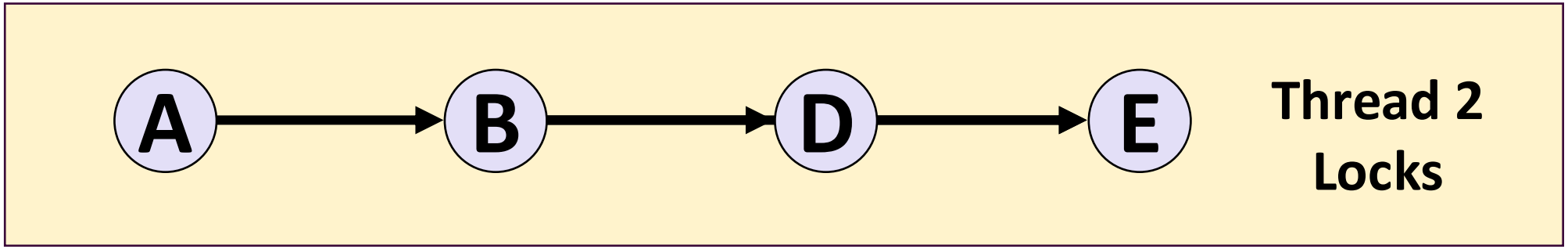
Concurrent Linked List using Locks

- Can just lock the entire linked list when you perform any operation
  - **Course-grained locking**



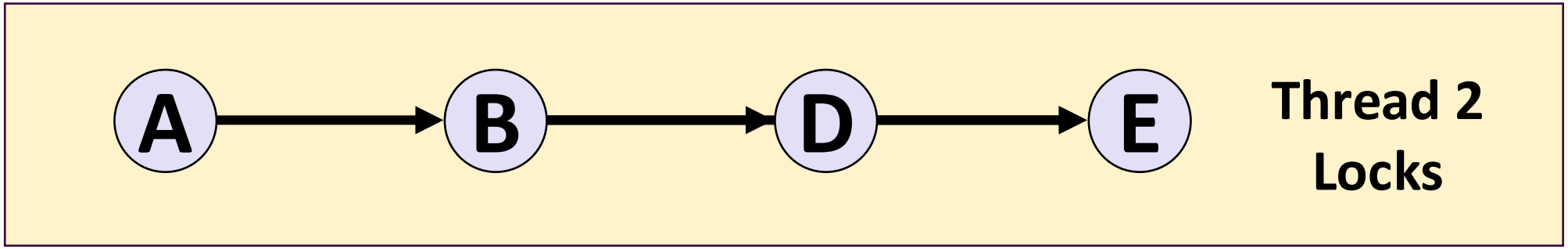
Concurrent Linked List using Locks

- Can just lock the entire linked list when you perform any operation
  - **Course-grained locking**



Concurrent Linked List using Locks

- Can just lock the entire linked list when you perform any operation
  - **Course-grained locking**
  - **Not good because too sequential, not using multicore processors at all**



Concurrent Linked List using Locks

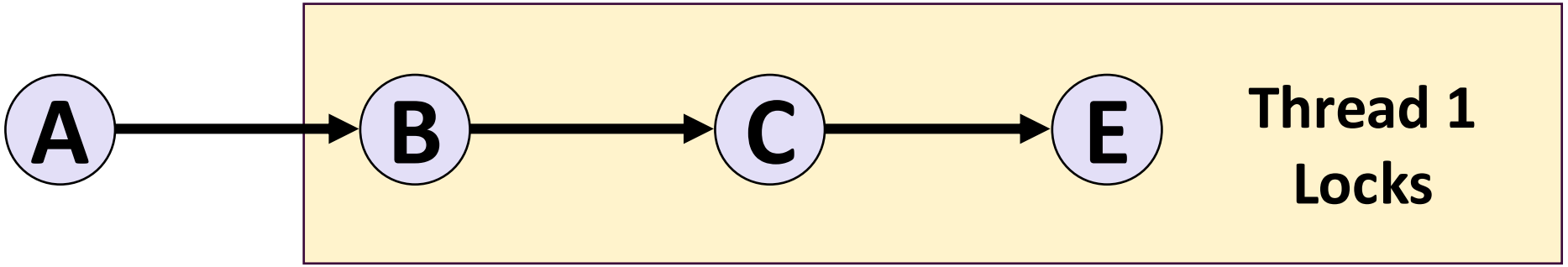
- **Fine-grained locking: what can you do?**

Concurrent Linked List using Locks

- **Fine-grained locking**
  - Lock only the nodes that are affected by each operation
  - E.g. only lock the neighbors!

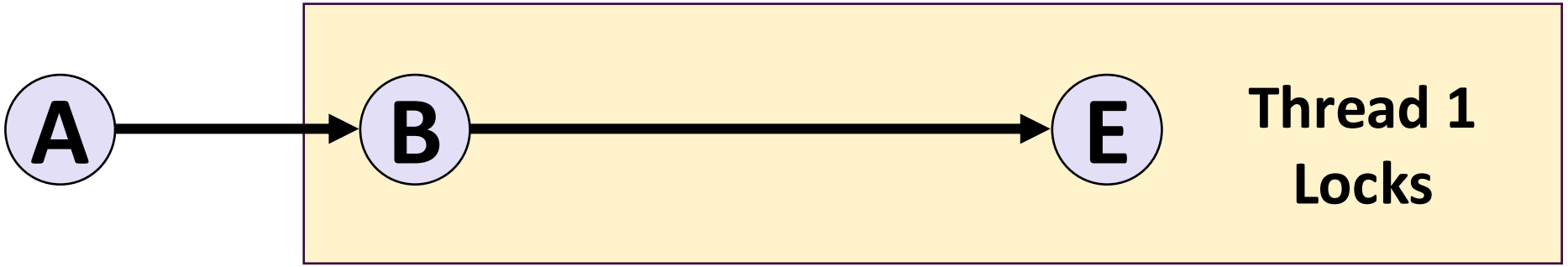
Concurrent Linked List using Locks

- **Fine-grained locking**
  - Lock only the nodes that are affected by each operation
  - E.g. only lock the neighbors!



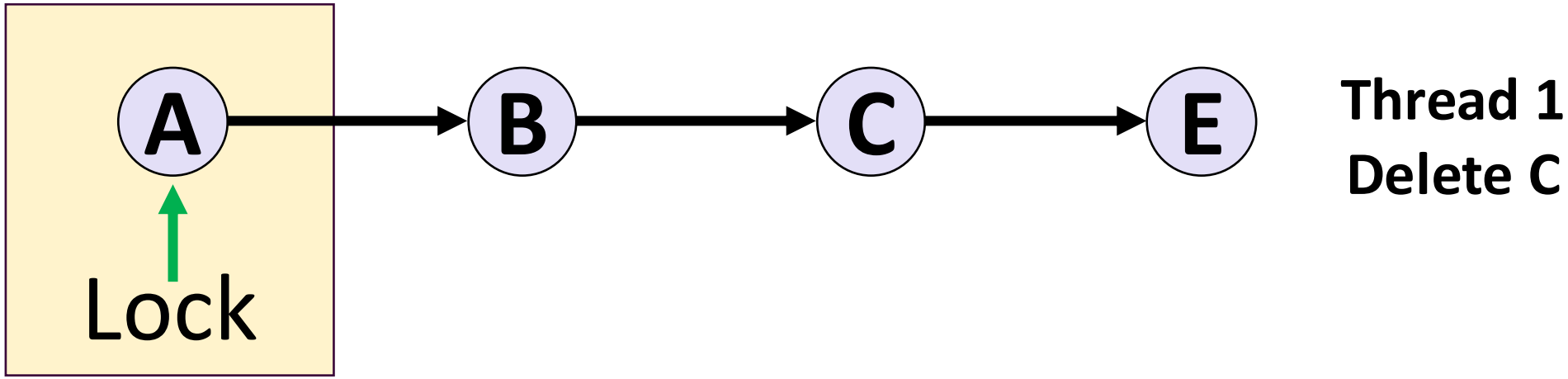
Concurrent Linked List using Locks

- **Fine-grained locking**
  - Lock only the nodes that are affected by each operation
  - E.g. only lock the neighbors!



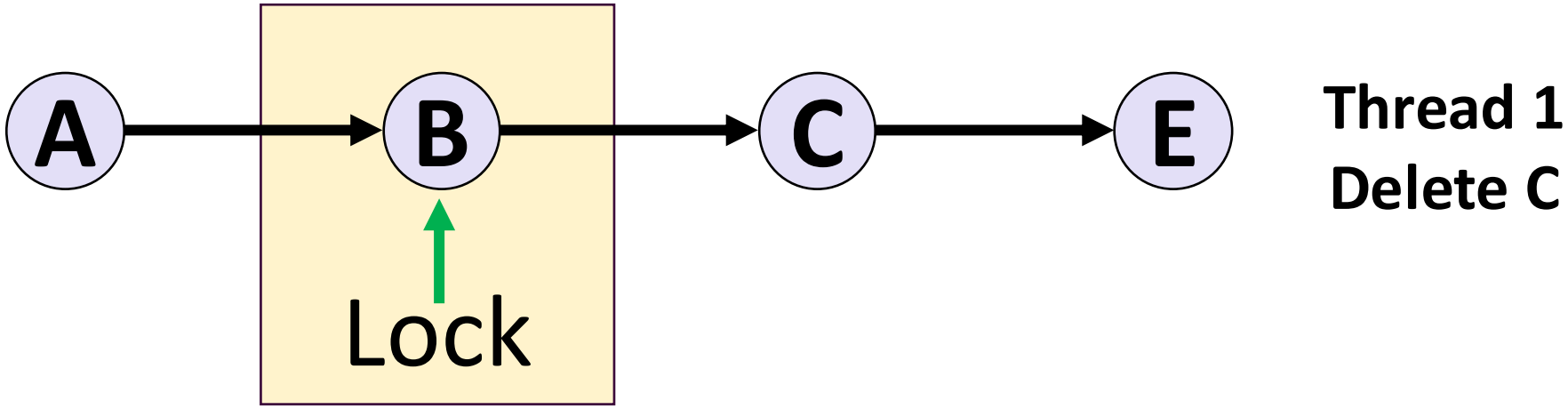
Concurrent Linked List using Locks

- **Deletion example step by step:**
  - First, start from the head node and find the element in the list
  - What do we need to do as we are trying to find the element in the list? What can go wrong?
  - We need to lock each node as we traverse because the node could have been changed while we are traversing



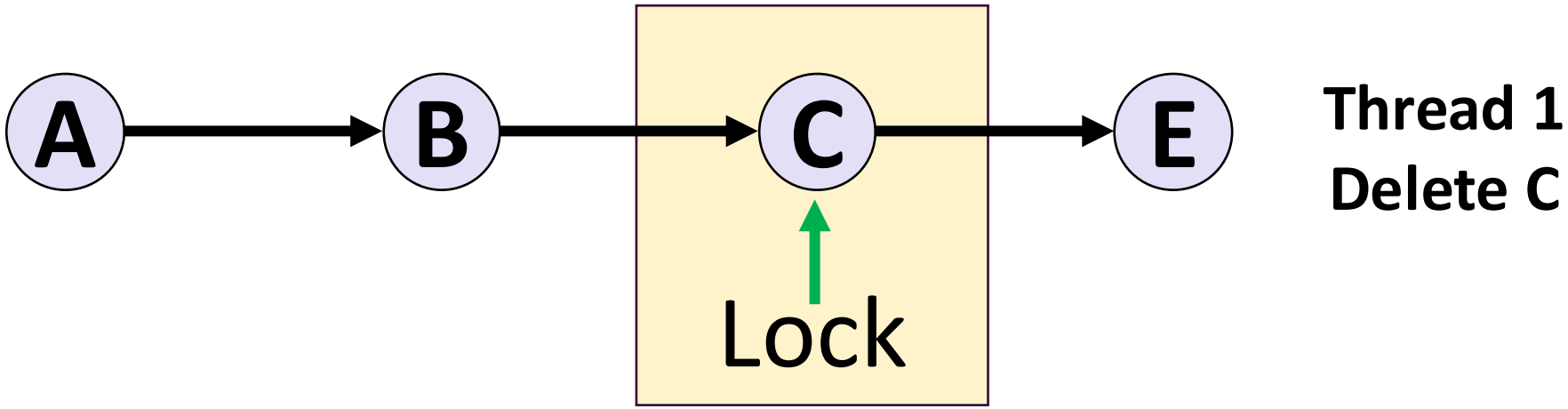
Concurrent Linked List using Locks

- **Deletion example step by step:**
  - First, start from the head node and find the element in the list
  - What do we need to do as we are trying to find the element in the list? What can go wrong?
  - We need to lock each node as we traverse because the node could have been changed while we are traversing



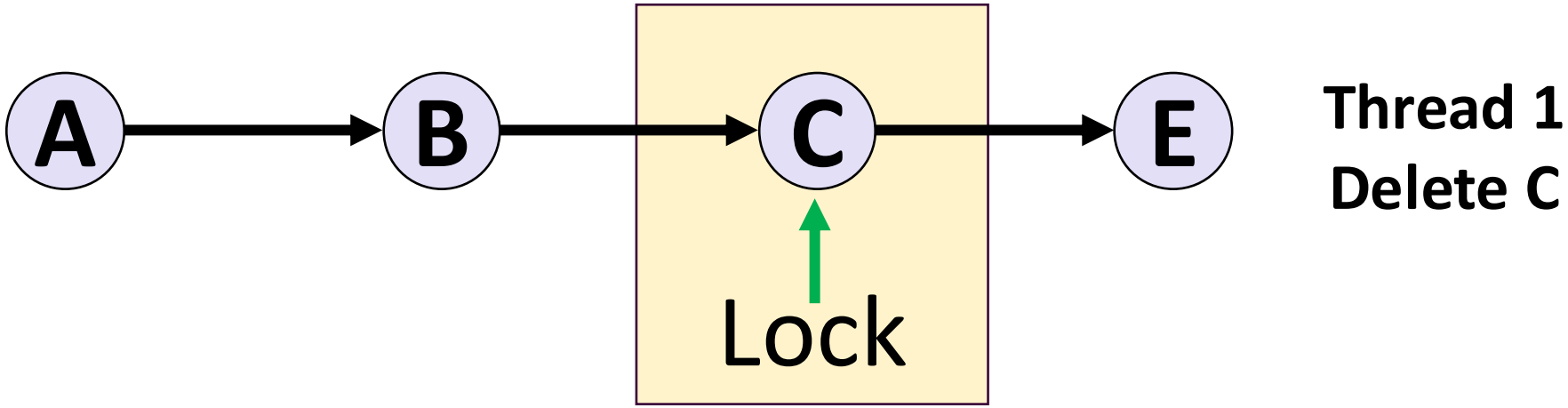
Concurrent Linked List using Locks

- **Deletion example step by step:**
  - First, start from the head node and find the element in the list
  - What do we need to do as we are trying to find the element in the list? What can go wrong?
  - We need to lock each node as we traverse because the node could have been changed while we are traversing



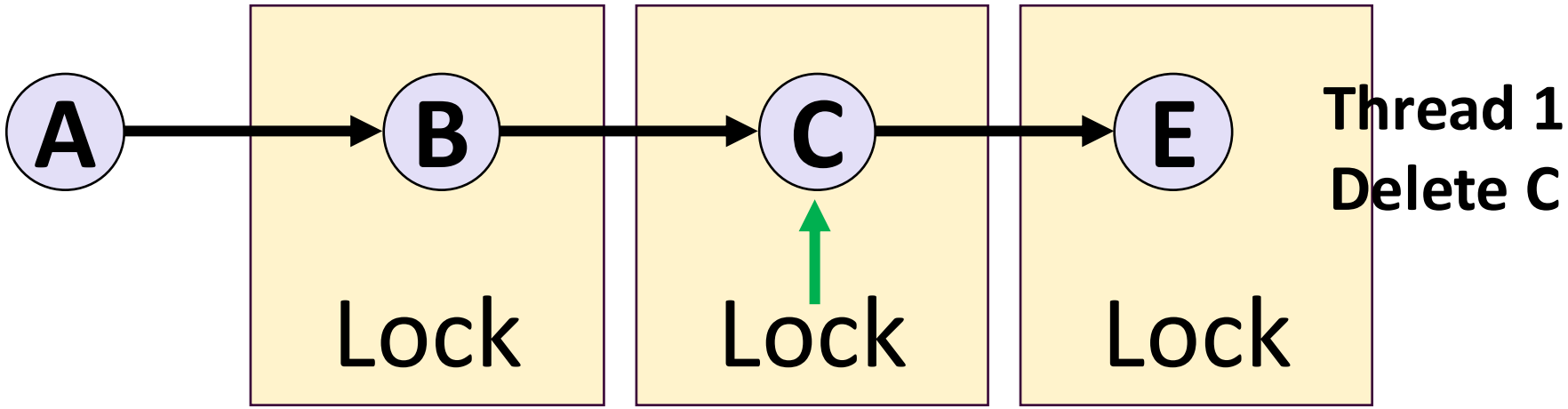
Concurrent Linked List using Locks

- **Deletion example step by step:**
  - First, start from the head node and find the element in the list
  - We need to lock each node as we traverse because the node could have been changed while we are traversing
  - Once we have found C, lock both predecessor and successor



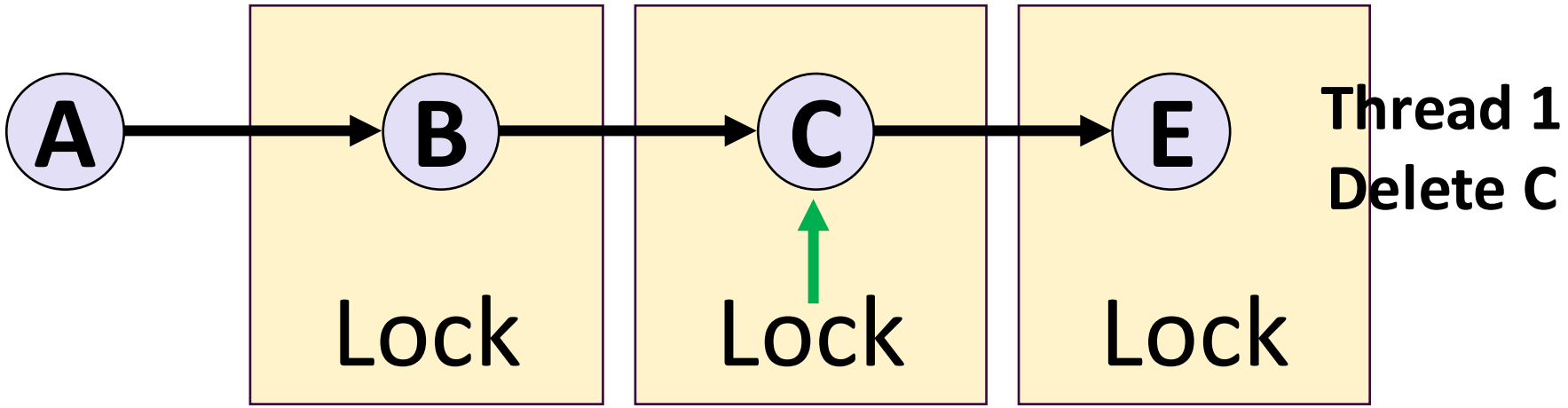
Concurrent Linked List using Locks

- **Deletion example step by step:**
  - First, start from the head node and find the element in the list
  - We need to lock each node as we traverse because the node could have been changed while we are traversing
  - Once we have found C, lock both predecessor and successor



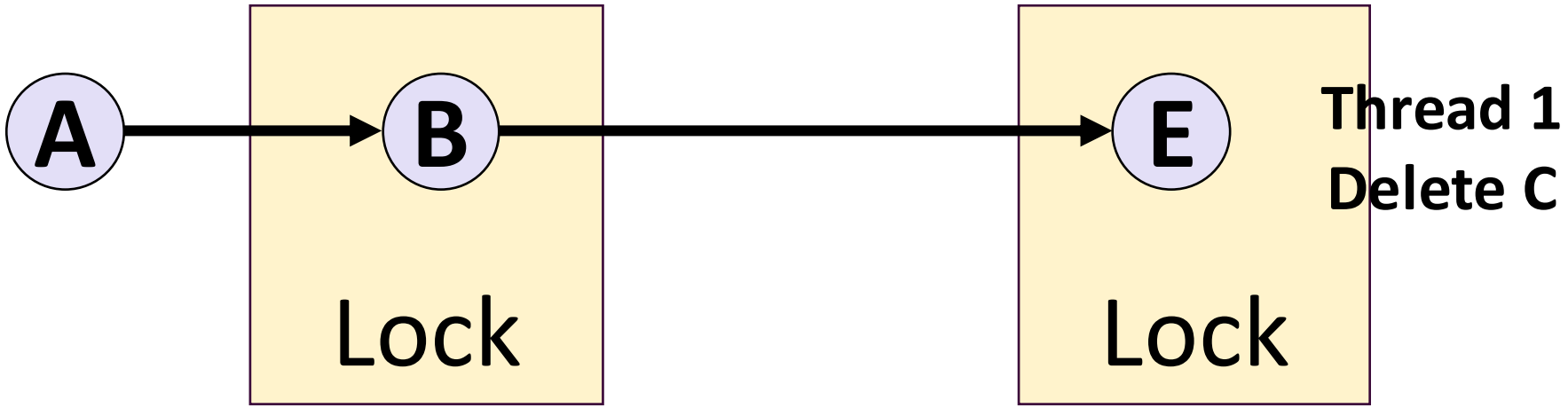
Concurrent Linked List using Locks

- **Deletion example step by step:**
  - First, start from the head node and find the element in the list
  - We need to lock each node as we traverse because the node could have been changed while we are traversing
  - Once we have found C, lock both predecessor and successor
  - After acquiring all locks, then perform operation



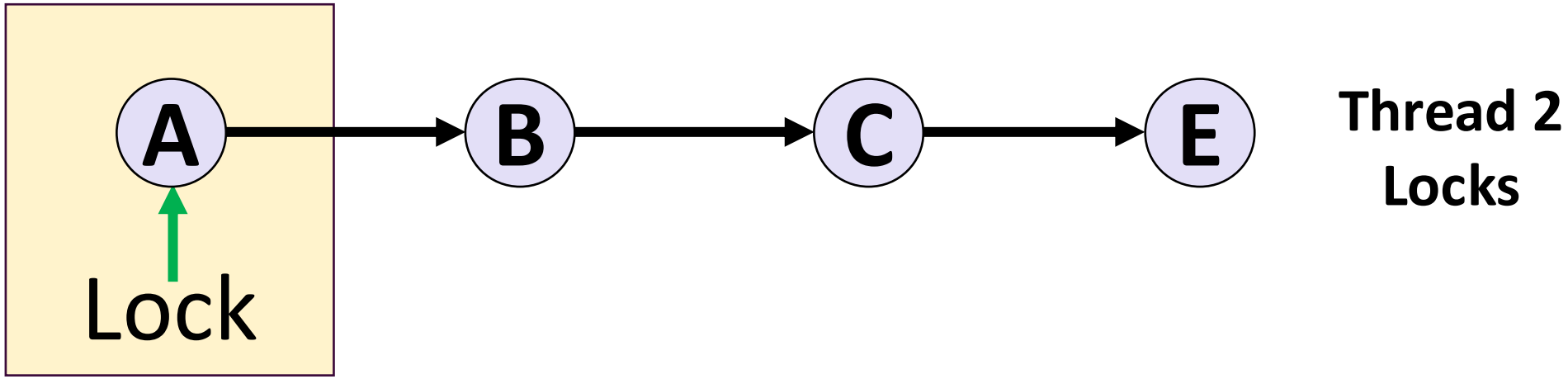
Concurrent Linked List using Locks

- **Deletion example step by step:**
  - First, start from the head node and find the element in the list
  - We need to lock each node as we traverse because the node could have been changed while we are traversing
  - Once we have found C, lock both predecessor and successor
  - After acquiring all locks, then perform operation



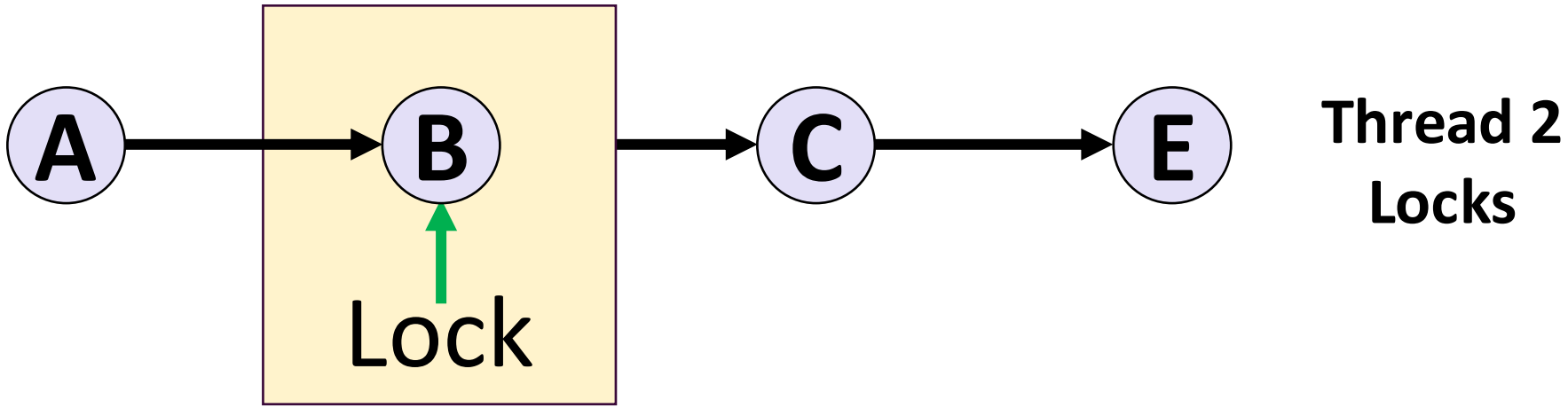
Concurrent Linked List using Locks

- **Insertion example step by step:**
  - First, start from the head node and find the element in the list
  - We need to lock each node as we traverse because the node could have been changed while we are traversing
  - Once we have found C, lock both predecessor and successor
  - After acquiring all locks, then perform operation



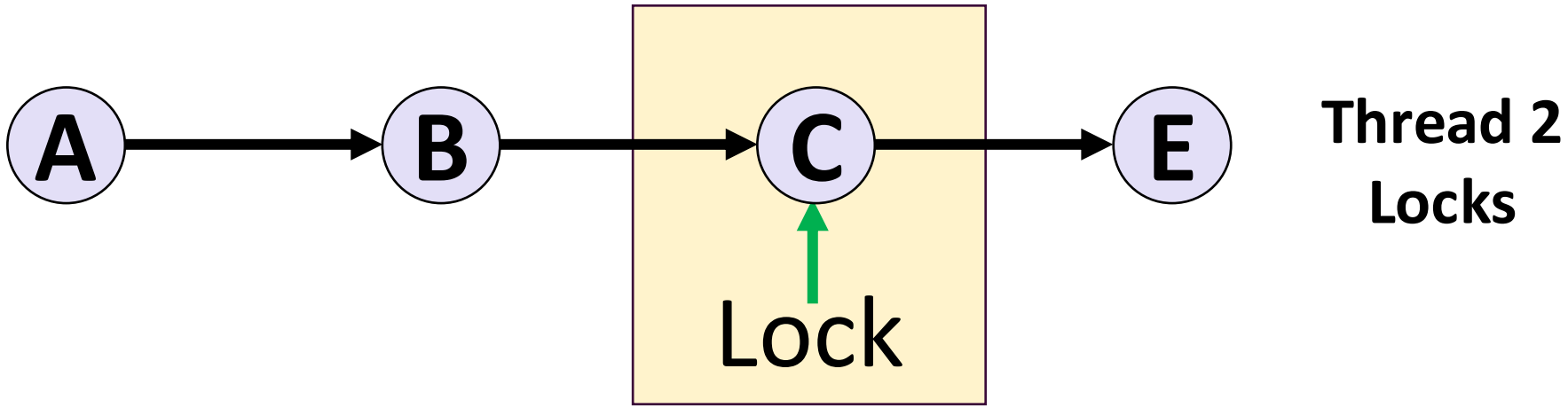
Concurrent Linked List using Locks

- **Insertion example step by step:**
  - First, start from the head node and find the element in the list
  - We need to lock each node as we traverse because the node could have been changed while we are traversing
  - Once we have found C, lock both predecessor and successor
  - After acquiring all locks, then perform operation



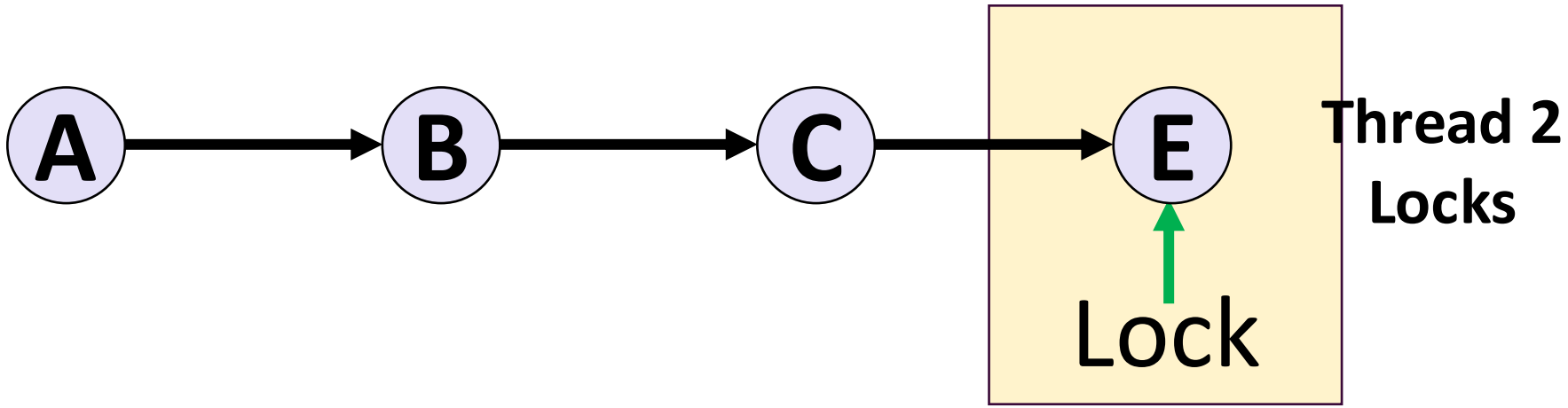
Concurrent Linked List using Locks

- **Insertion example step by step:**
  - First, start from the head node and find the element in the list
  - We need to lock each node as we traverse because the node could have been changed while we are traversing
  - Once we have found C, lock both predecessor and successor
  - After acquiring all locks, then perform operation



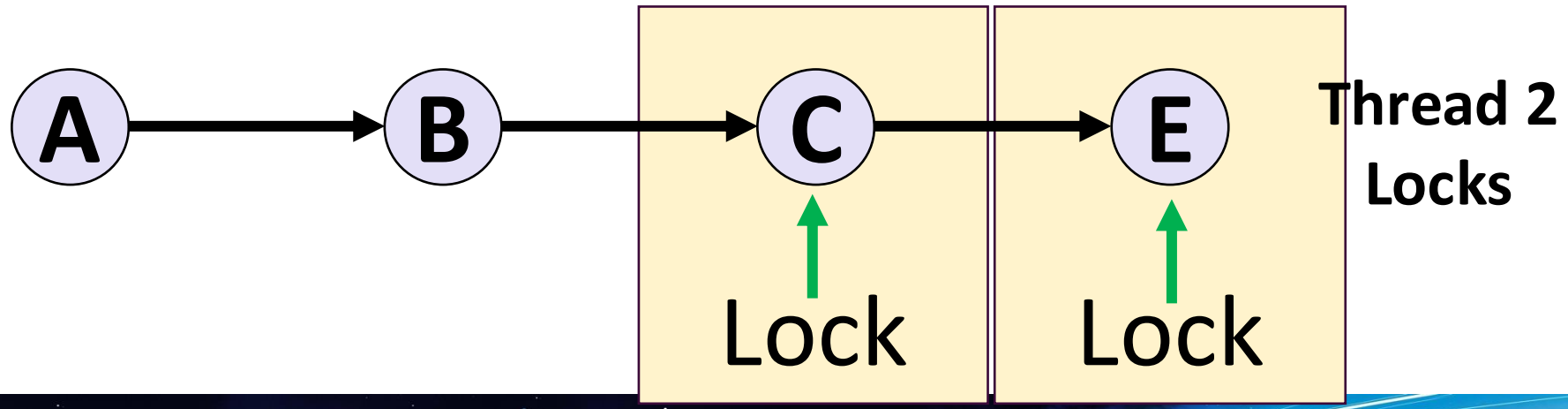
Concurrent Linked List using Locks

- **Insertion example step by step:**
  - First, start from the head node and find the element in the list
  - We need to lock each node as we traverse because the node could have been changed while we are traversing
  - Once we have found C, lock both predecessor and successor
  - After acquiring all locks, then perform operation



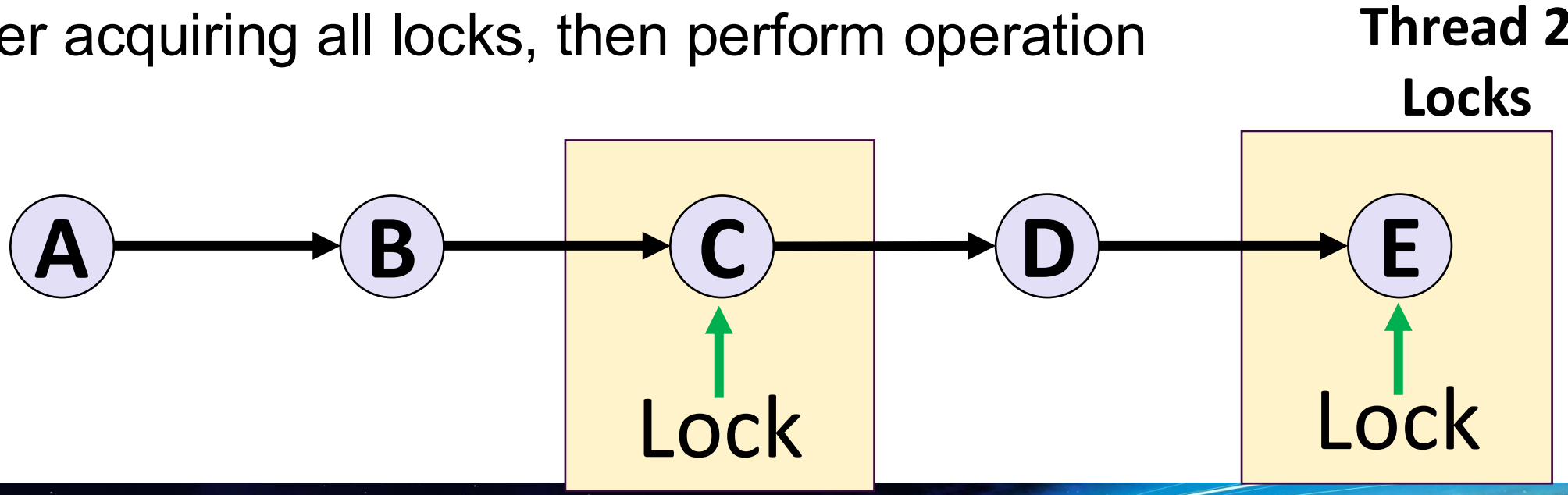
Concurrent Linked List using Locks

- **Insertion example step by step:**
  - First, start from the head node and find the element in the list
  - We need to lock each node as we traverse because the node could have been changed while we are traversing
  - Once we have found C, lock both predecessor and successor
  - After acquiring all locks, then perform operation



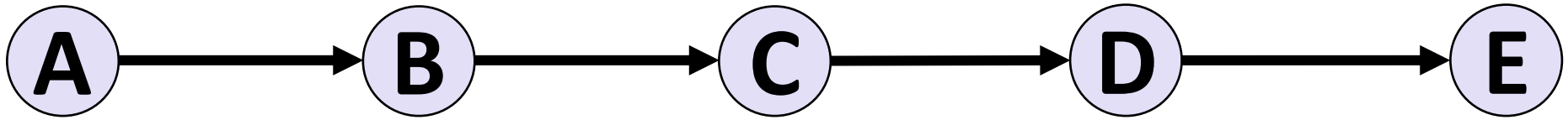
Concurrent Linked List using Locks

- **Insertion example step by step:**
  - First, start from the head node and find the element in the list
  - We need to lock each node as we traverse because the node could have been changed while we are traversing
  - Once we have found C, lock both predecessor and successor
  - After acquiring all locks, then perform operation



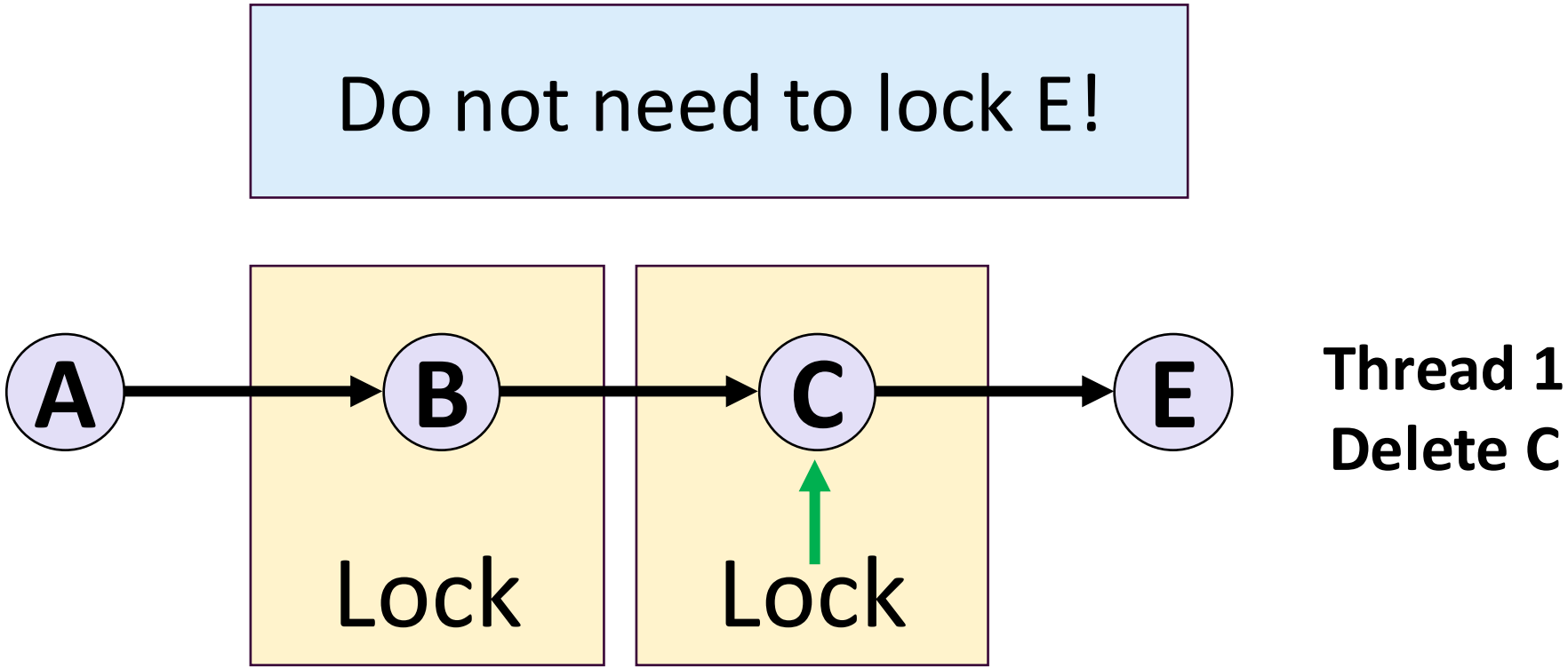
Concurrent Linked List using Locks

- **Insertion example step by step:**
  - First, start from the head node and find the element in the list
  - We need to lock each node as we traverse because the node could have been changed while we are traversing
  - Once we have found C, lock both predecessor and successor
  - After acquiring all locks, then perform operation



Lock-Based Concurrent Linked List

- **But this procedure has one more lock than necessary; what is this unnecessary lock?**

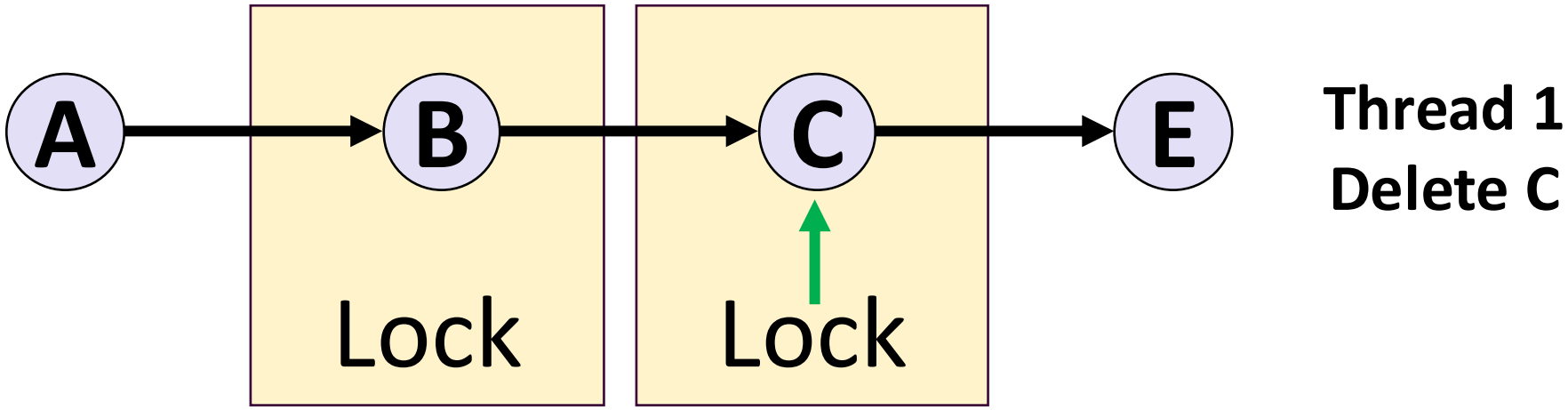


Lock-Based Concurrent Linked List

- **But this procedure has one more lock than necessary; what is this unnecessary lock?**

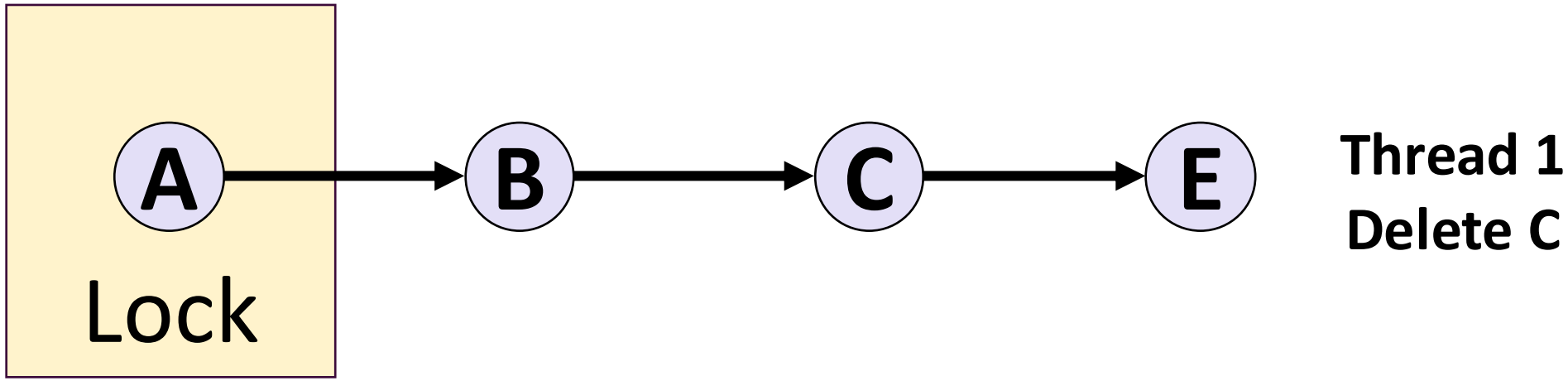
Do not need to lock E!

We are only changing pointers of B



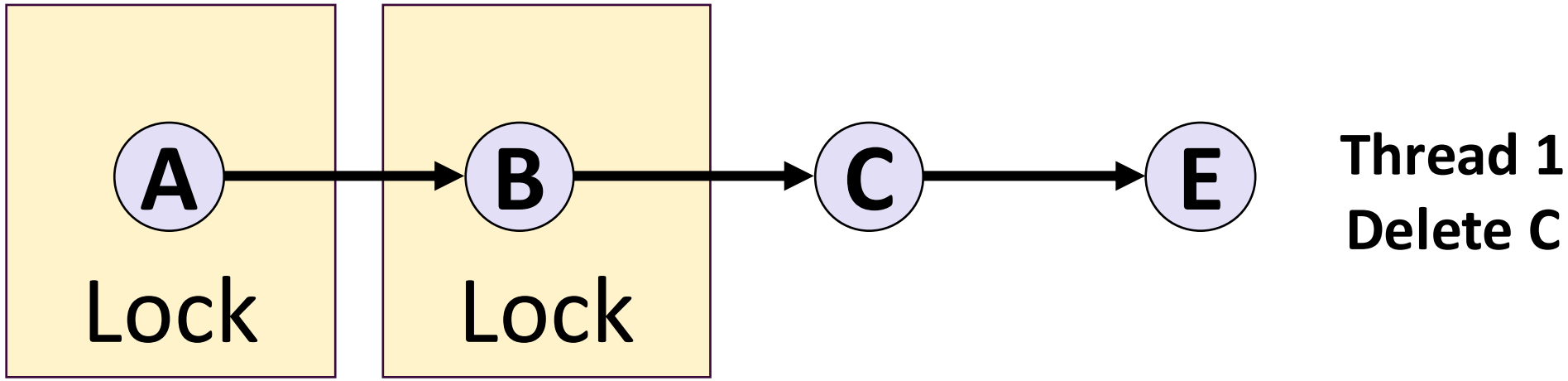
Lock-Based Concurrent Linked List

- **Revised procedure:**
  - Start at head and search for C



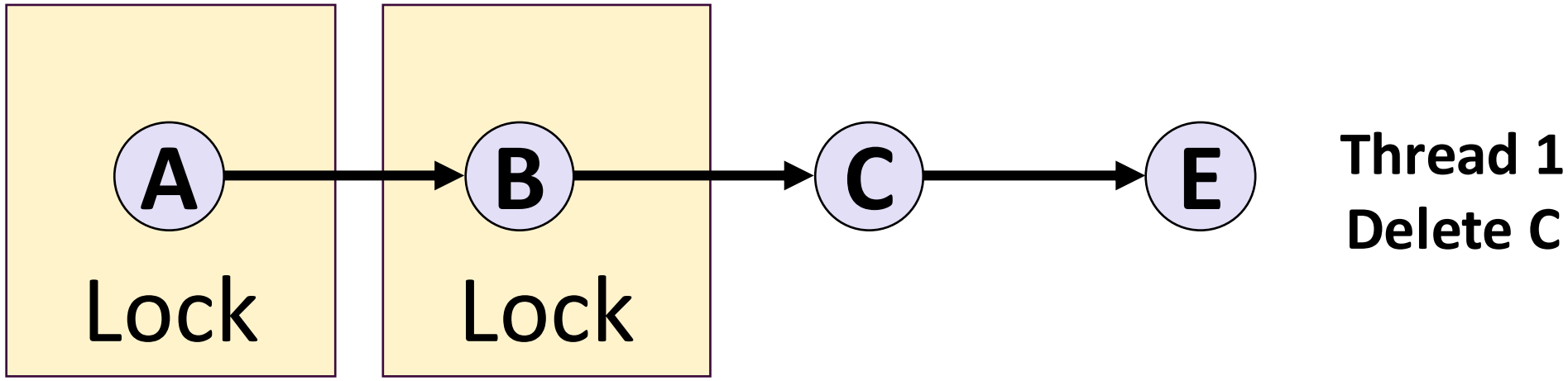
Lock-Based Concurrent Linked List

- **Revised procedure:**
  - Start at head and search for C
  - Lock next searched node



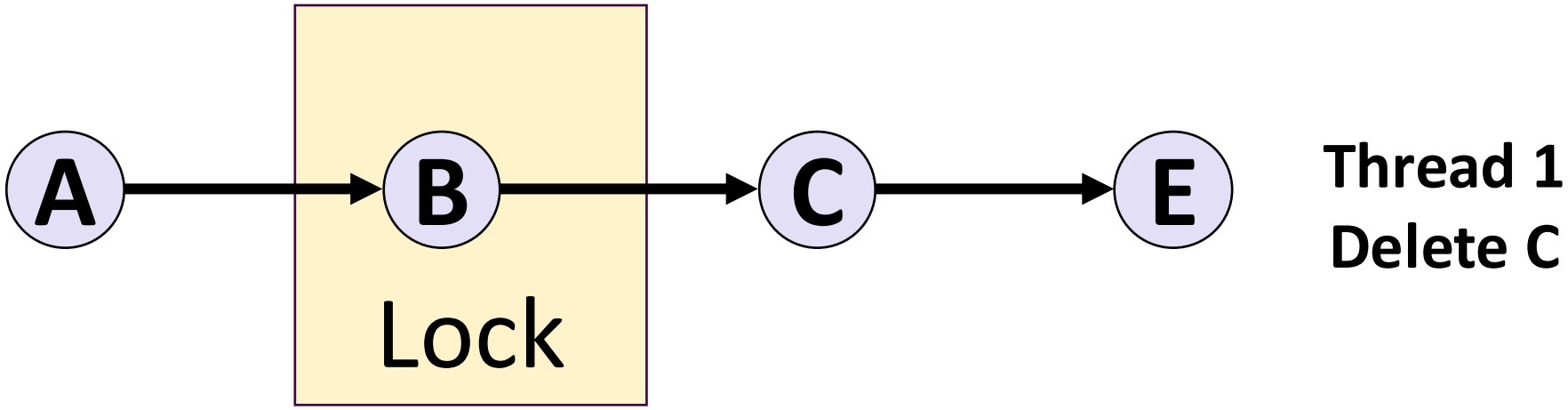
Lock-Based Concurrent Linked List

- **Revised procedure:**
  - Start at head and search for C
  - Lock next searched node
  - If next locked node is not C, remove lock on previous node



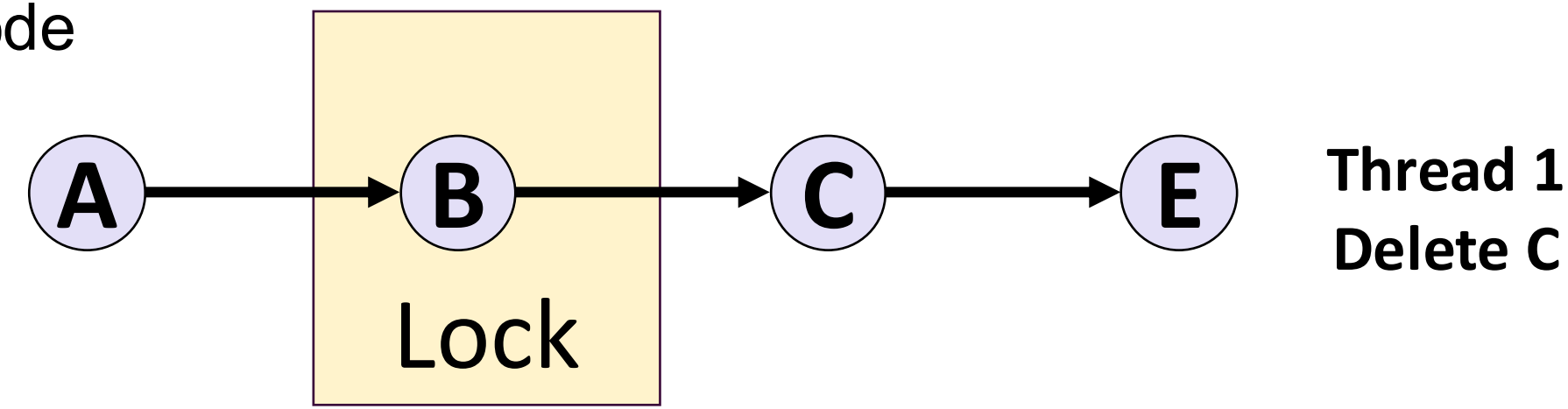
Lock-Based Concurrent Linked List

- **Revised procedure:**
  - Start at head and search for C
  - Lock next searched node
  - If next locked node is not C, remove lock on previous node



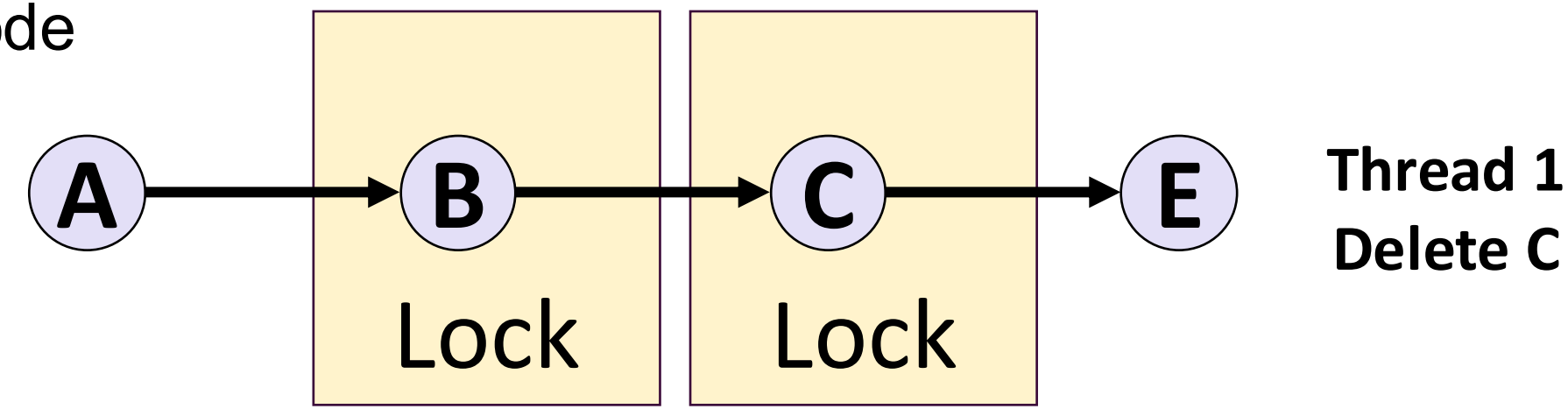
Lock-Based Concurrent Linked List

- **Revised procedure:**
  - Start at head and search for C
  - Lock next searched node
  - If next locked node is not C, remove lock on previous node
  - Continue until find C, if found C, do not remove lock on previous node



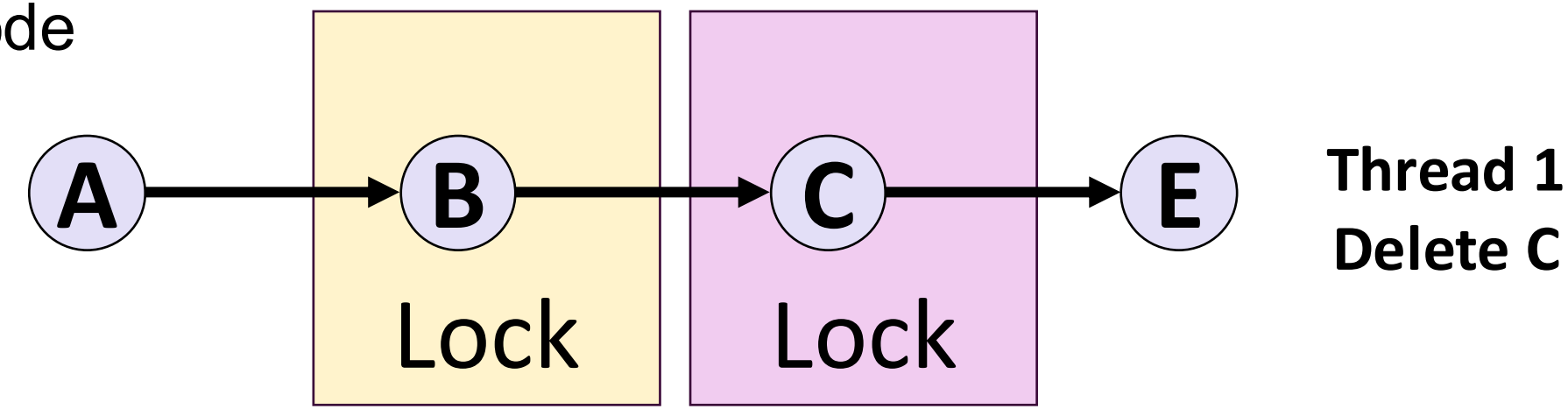
Lock-Based Concurrent Linked List

- **Revised procedure:**
  - Start at head and search for C
  - Lock next searched node
  - If next locked node is not C, remove lock on previous node
  - Continue until find C, if found C, do not remove lock on previous node



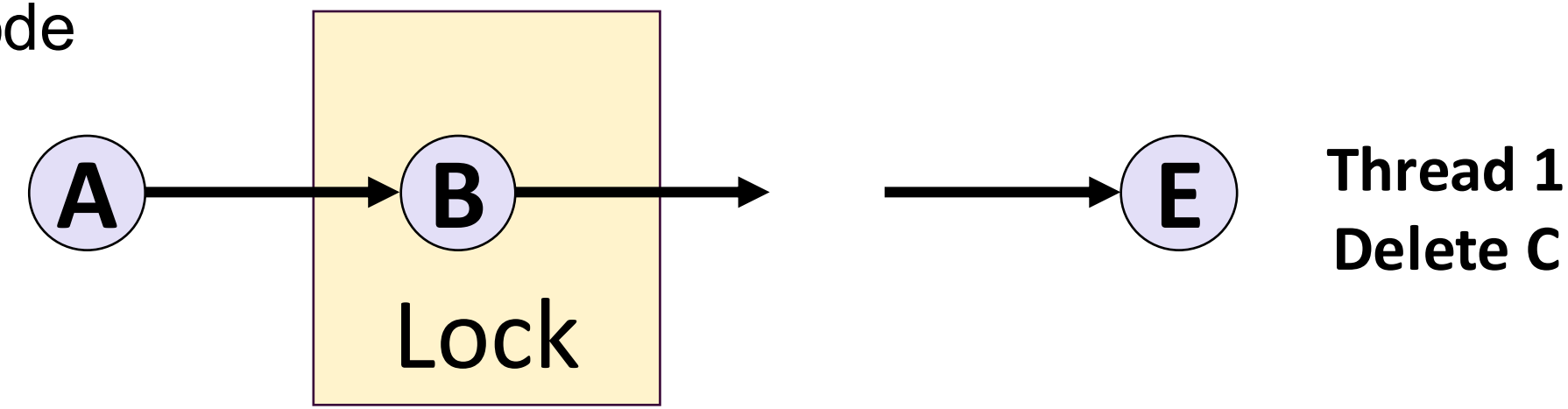
Lock-Based Concurrent Linked List

- **Revised procedure:**
  - Start at head and search for C
  - Lock next searched node
  - If next locked node is not C, remove lock on previous node
  - Continue until find C, if found C, do not remove lock on previous node



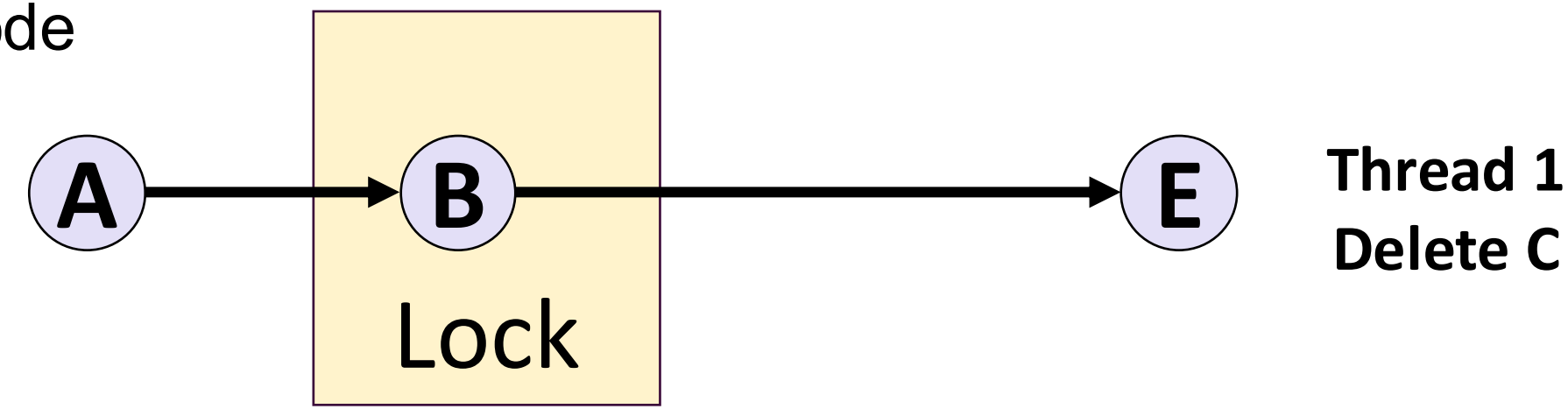
Lock-Based Concurrent Linked List

- **Revised procedure:**
  - Start at head and search for C
  - Lock next searched node
  - If next locked node is not C, remove lock on previous node
  - Continue until find C, if found C, do not remove lock on previous node



Lock-Based Concurrent Linked List

- **Revised procedure:**
  - Start at head and search for C
  - Lock next searched node
  - If next locked node is not C, remove lock on previous node
  - Continue until find C, if found C, do not remove lock on previous node



Lock-Based Concurrent Linked List

- **Revised procedure:**
  - Start at head and search for C
  - Lock next searched node
  - If next locked node is not C, remove lock on previous node
  - Continue until find C, if found C, do not remove lock on previous node

Thread 1
Delete C

Lock-Based Concurrent Linked List

- **Revised procedure:**
  - Start at head and search for C
  - Lock next searched node
  - If next locked node is not C, remove lock on previous node
  - Continue until find C, if found C, do not remove lock on previous node

This is called "Hand-over-Hand" Locking!

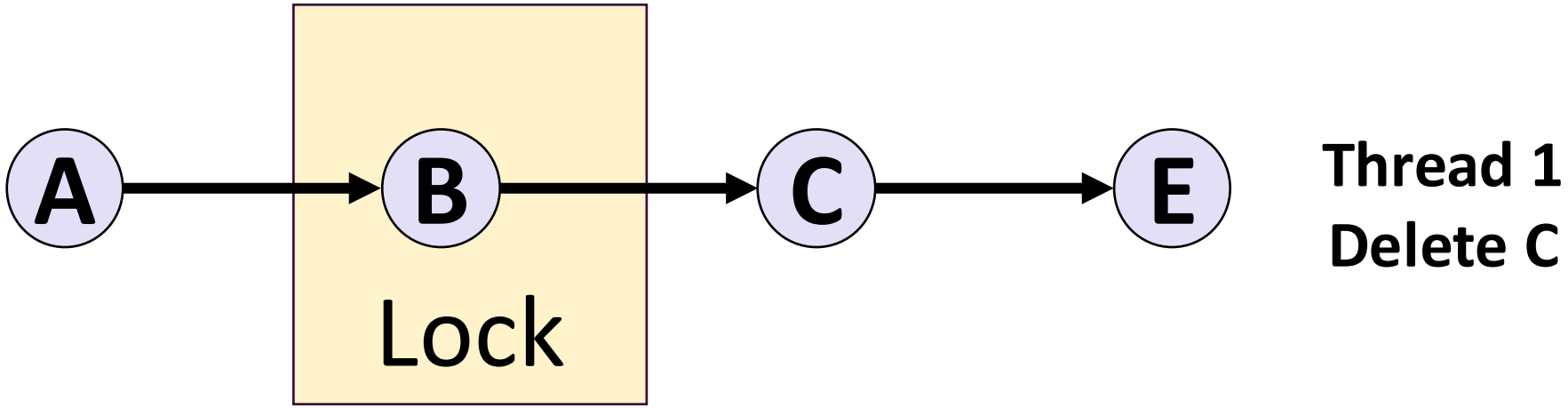
Thread 1
Delete C

Lock-Based Concurrent Linked List

- **Thought exercise:**
  - What if we get rid of the lock on C when deleting C?
  - **What could happen?**
  - Is the lock on C necessary?

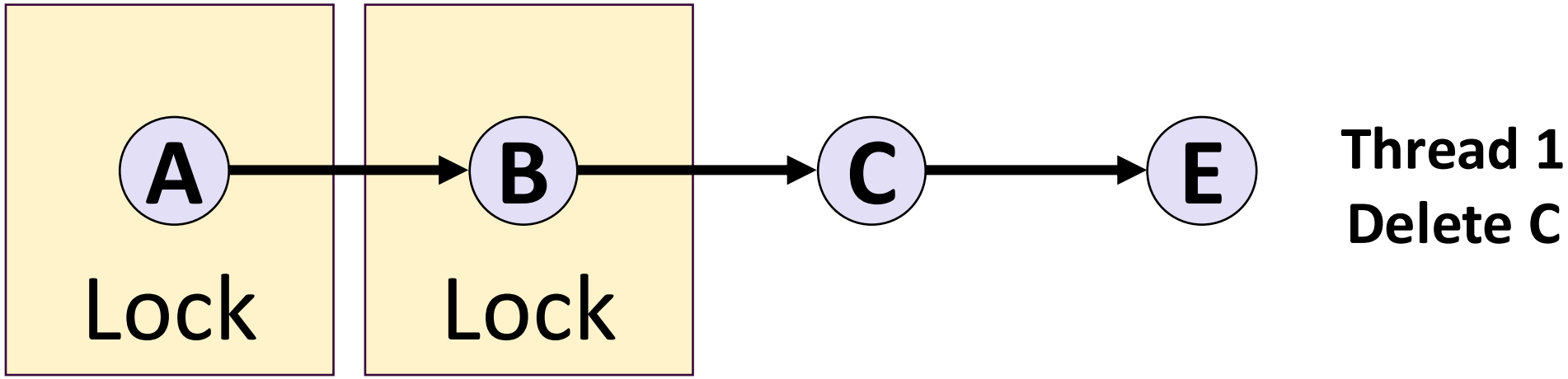
Lock-Based Concurrent Linked List

- **There is a problem with not locking C!**
- **What is the problem?**



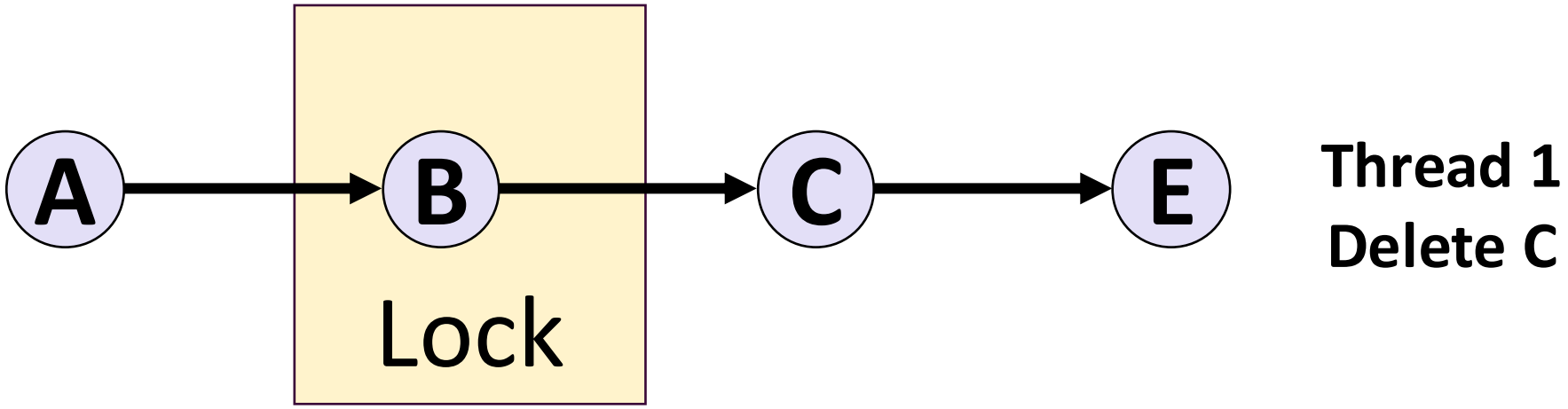
Lock-Based Concurrent Linked List

- **Thought Experiment Procedure:**
  - Start at head and search for C
  - If next node is not C, lock the node



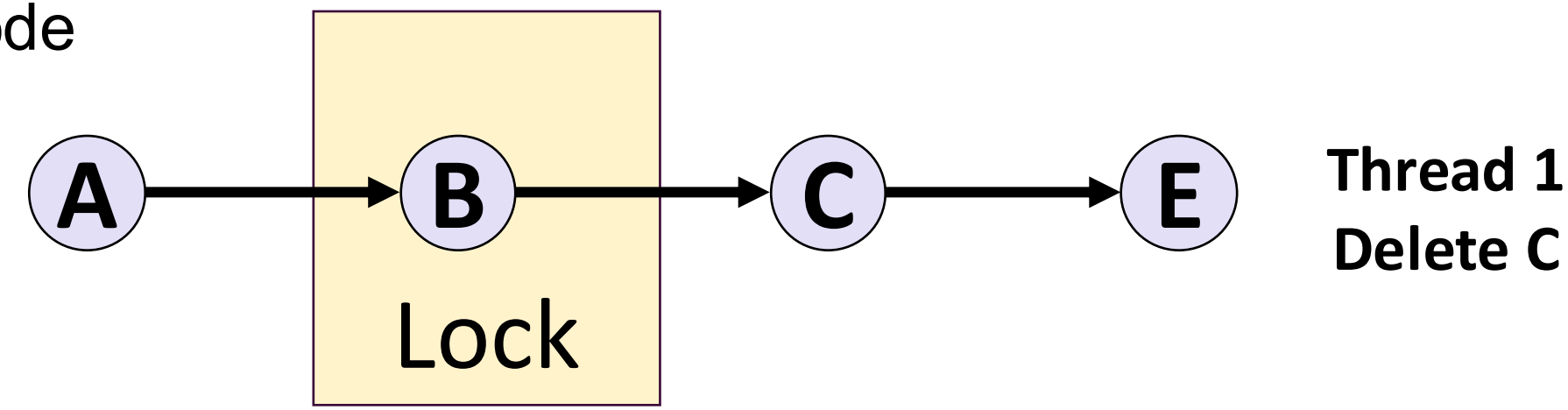
Lock-Based Concurrent Linked List

- **Thought Experiment Procedure:**
  - Start at head and search for C
  - If next node is not C, lock the node
  - If next locked node is not C, remove lock on previous node



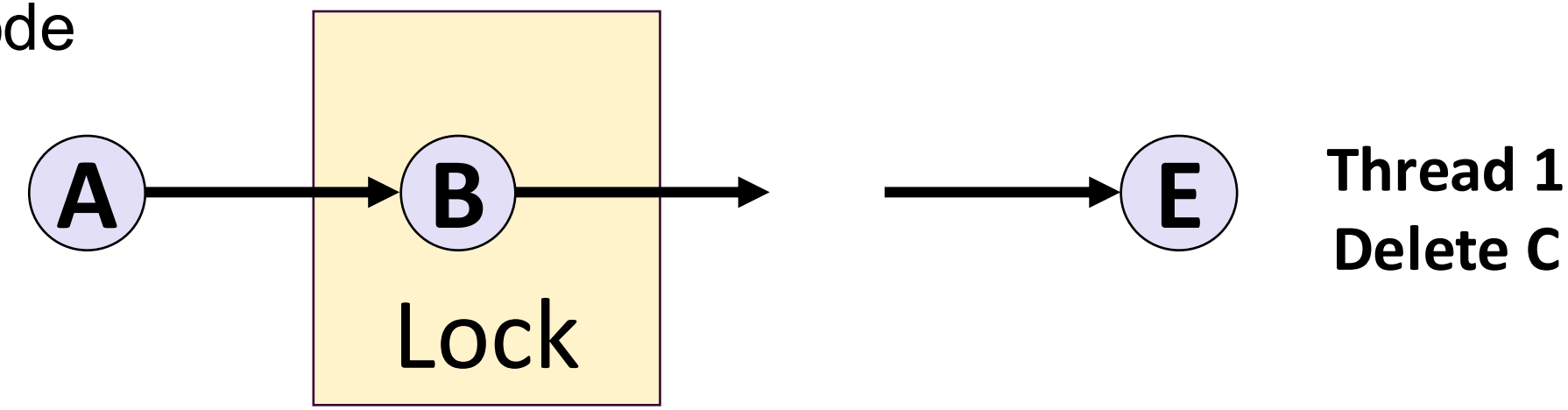
Lock-Based Concurrent Linked List

- **Thought Experiment Procedure:**
  - Start at head and search for C
  - If next node is not C, lock the node
  - If next locked node is not C, remove lock on previous node
  - Continue until find C, if found C, do not remove lock on previous node



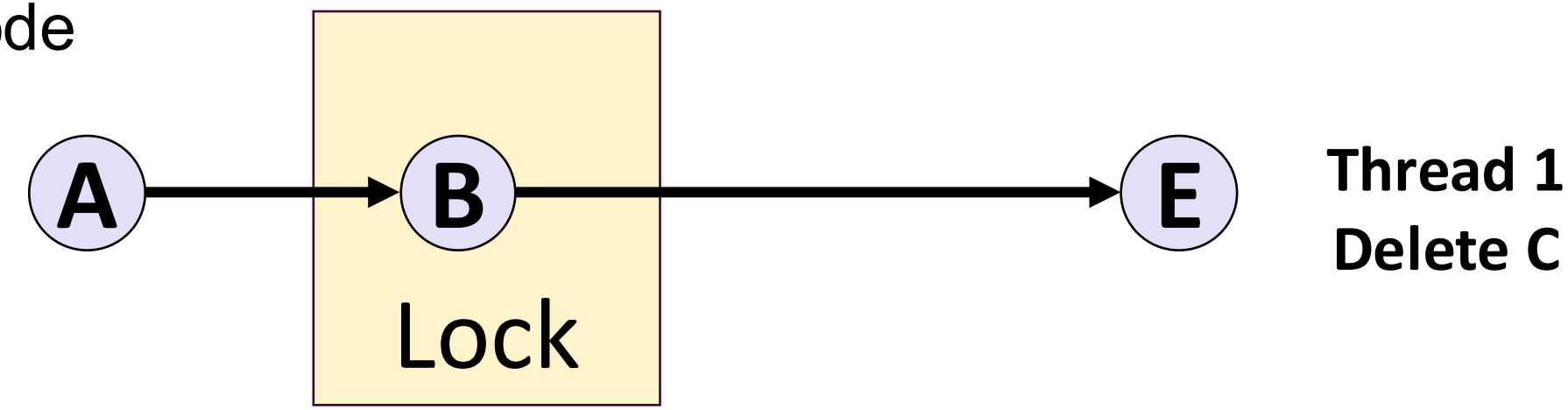
Lock-Based Concurrent Linked List

- **Thought Experiment Procedure:**
  - Start at head and search for C
  - If next node is not C, lock the node
  - If next locked node is not C, remove lock on previous node
  - Continue until find C, if found C, do not remove lock on previous node



Lock-Based Concurrent Linked List

- **Thought Experiment Procedure:**
  - Start at head and search for C
  - If next node is not C, lock the node
  - If next locked node is not C, remove lock on previous node
  - Continue until find C, if found C, do not remove lock on previous node



Lock-Based Concurrent Linked List

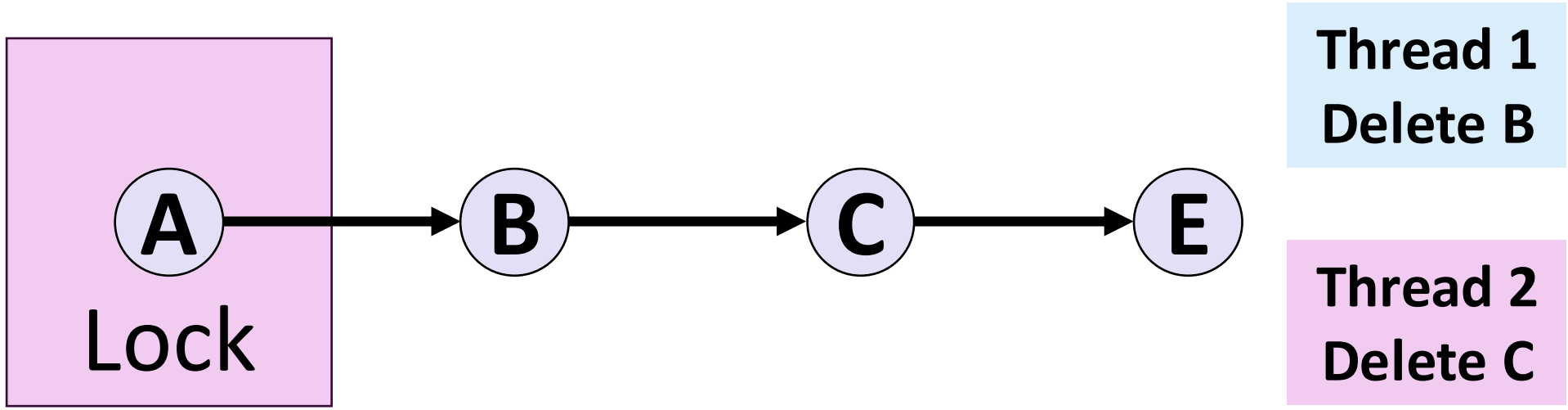
- **There is a problem with this thought experiment procedure!**
- **What is the problem?**
- Suppose two threads, **one deletes B** and **one deletes C**
- We had discussed issues with concurrent algorithms before: deadlock, livelock, correctness issues…etc.

Thread 1
Delete B

Thread 2
Delete C

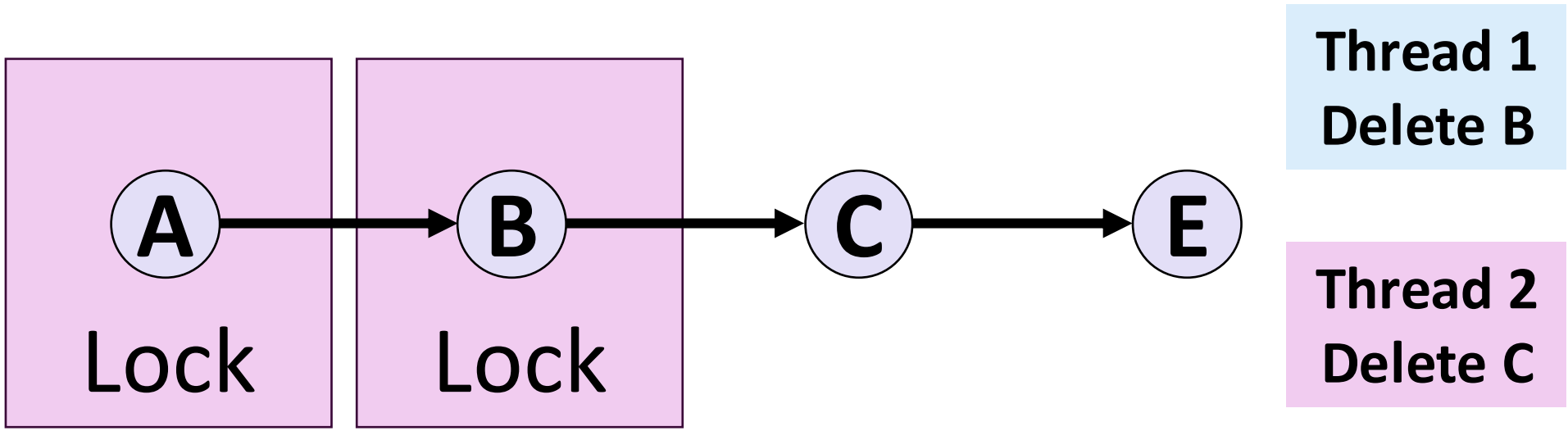
Lock-Based Concurrent Linked List Issue

- **There could be deadlock caused by an incorrect state!**



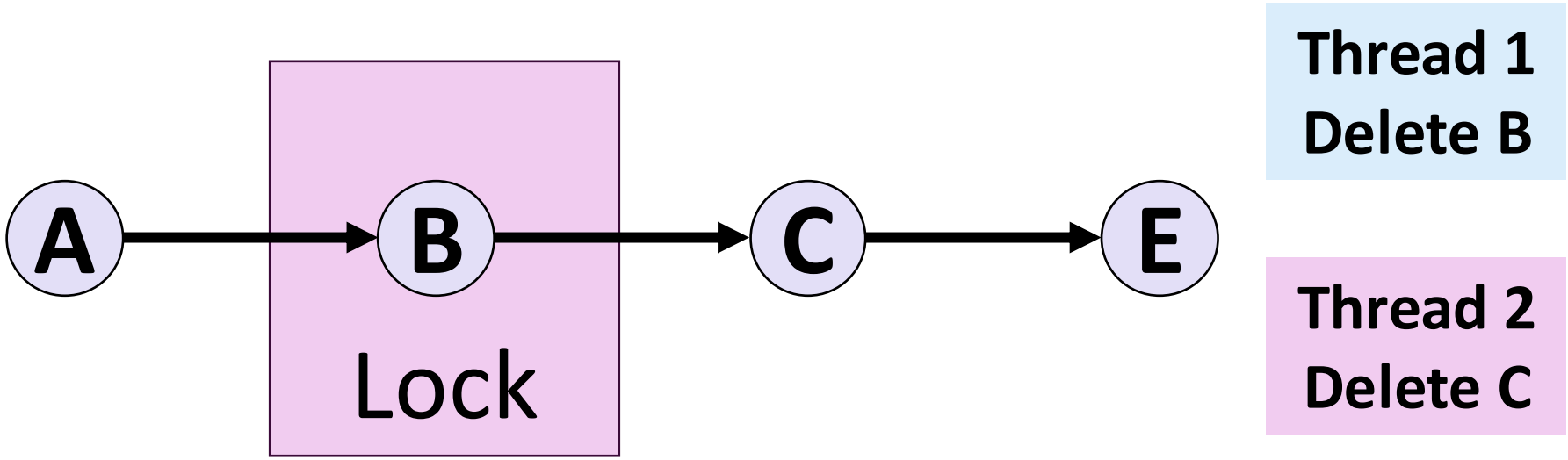
Lock-Based Concurrent Linked List Issue

- **There could be deadlock caused by an incorrect state!**



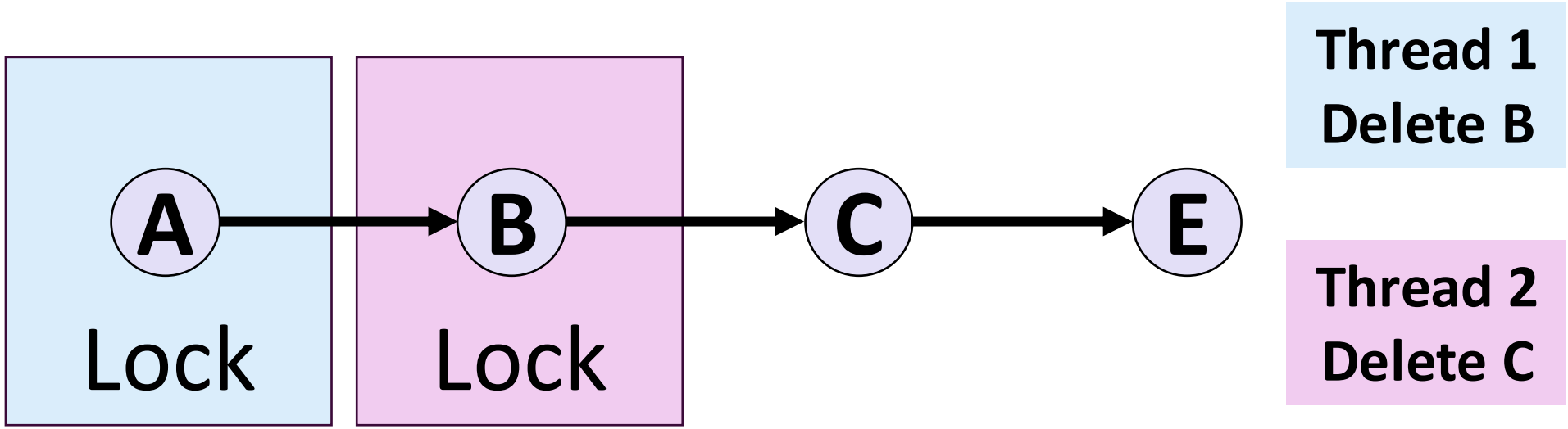
Lock-Based Concurrent Linked List Issue

- **There could be deadlock caused by an incorrect state!**



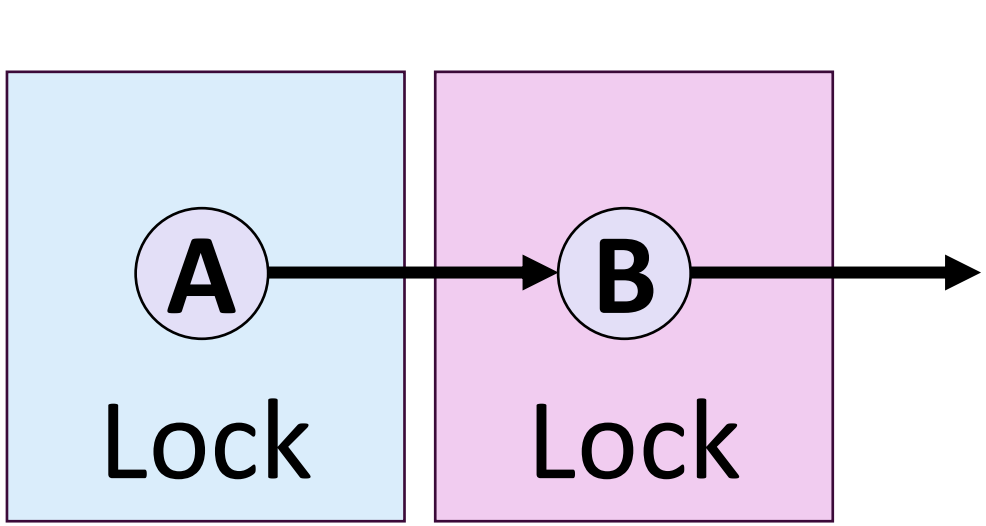
Lock-Based Concurrent Linked List Issue

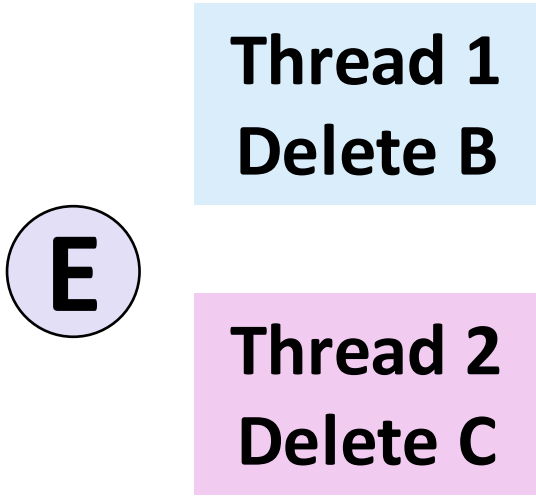
- **There could be deadlock caused by an incorrect state!**



Lock-Based Concurrent Linked List Issue

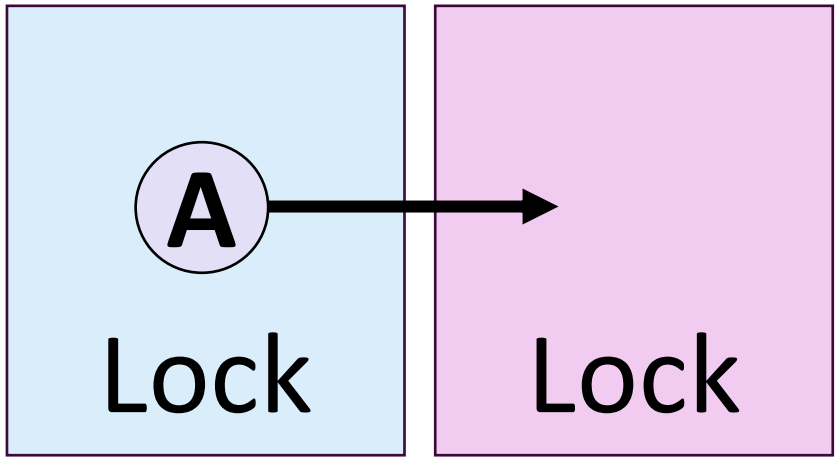
- **There could be deadlock caused by an incorrect state!**

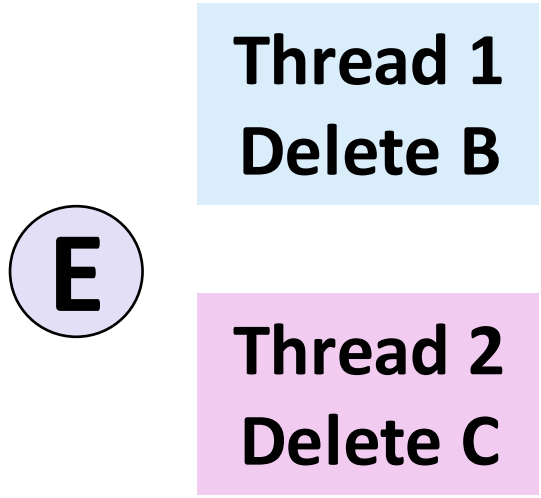




Lock-Based Concurrent Linked List Issue

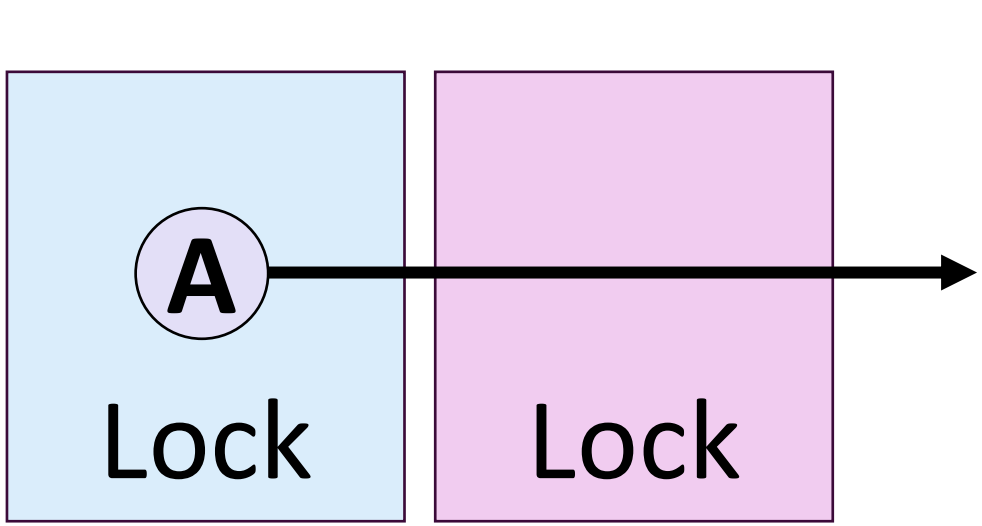
- **There could be deadlock caused by an incorrect state!**

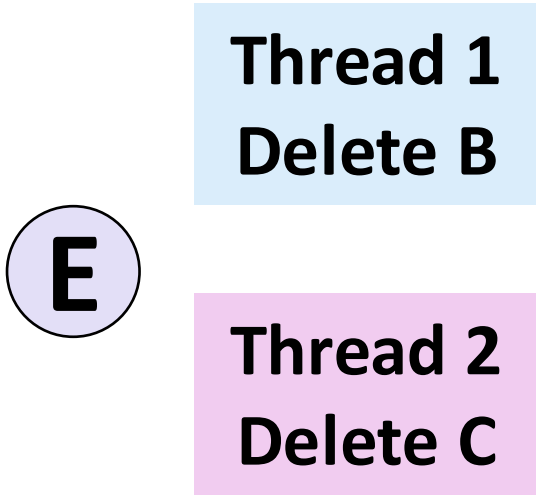




Lock-Based Concurrent Linked List Issue

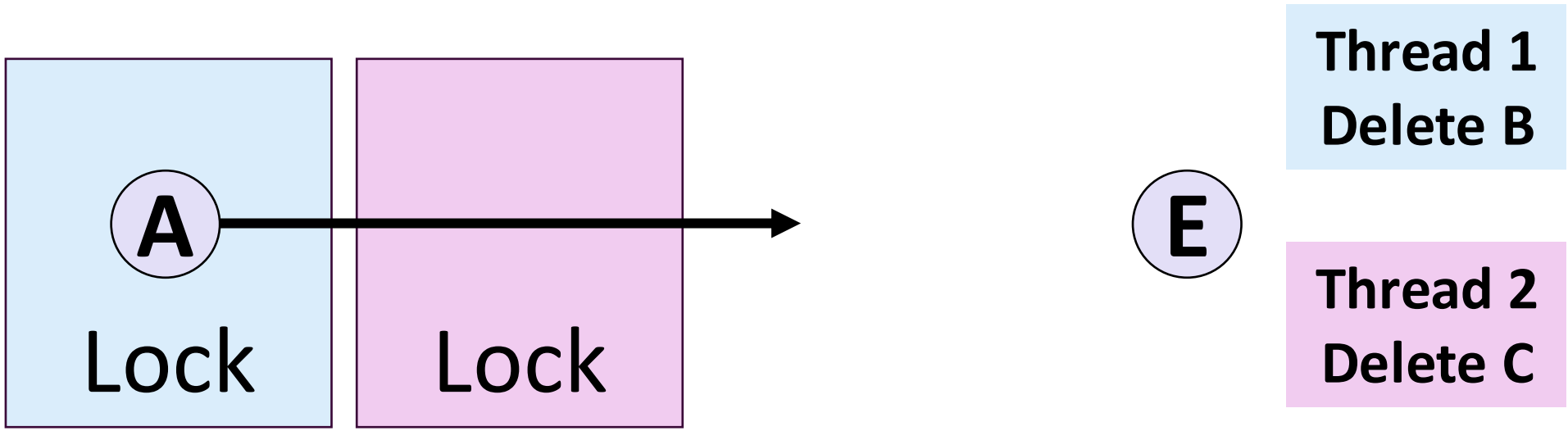
- **There could be deadlock caused by an incorrect state!**





Lock-Based Concurrent Linked List Issue

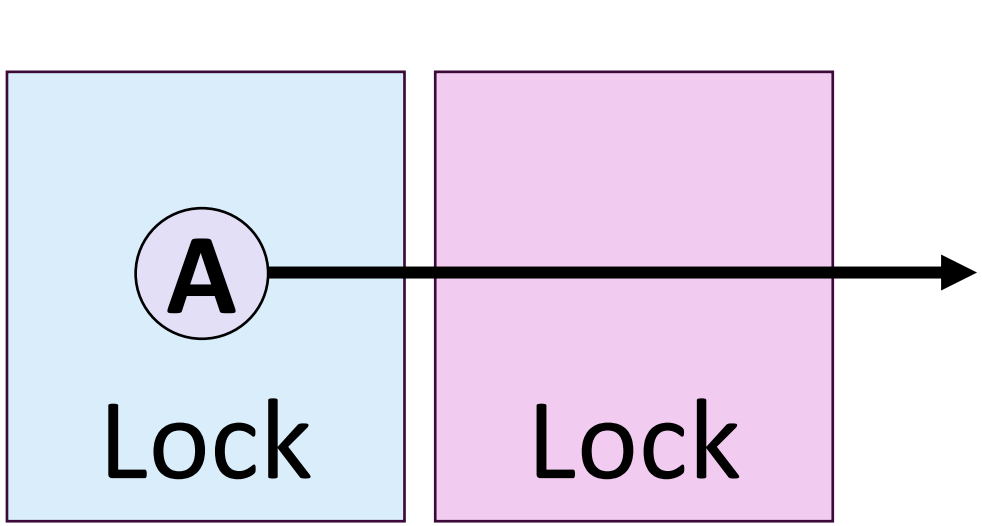
- **There could be deadlock caused by an incorrect state!**

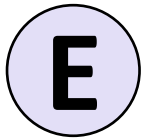


Thread 2 **cannot complete its procedure** because B no longer exists

Lock-Based Concurrent Linked List Issue

- **There could be deadlock caused by an incorrect state!**





Thread 1
Delete B

Thread 2
Delete C

Also A has a pointer to nowhere

Lock-Based Concurrent Linked List

- Despite being correct, this is not an ideal procedure still
- **What could cause suboptimal performance?**

Lock-Based Concurrent Linked List

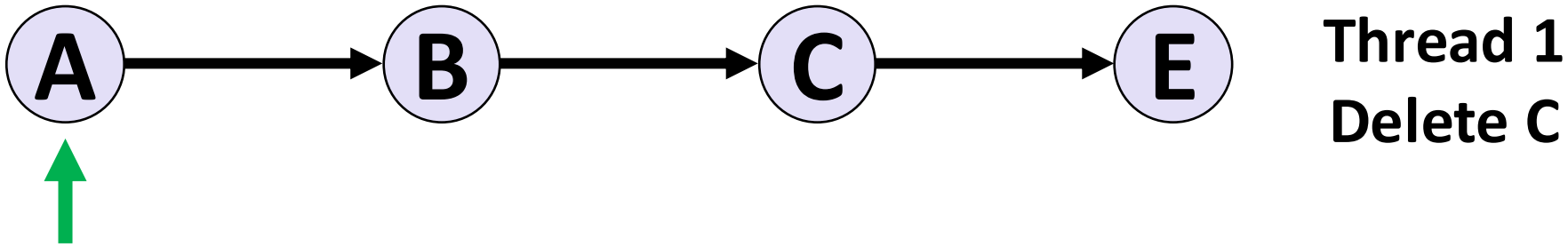
- Despite being correct, this is not an ideal procedure still
- Long chain of locking and unlocking
- Contention could cause this procedure to be very inefficient
- Serialized access to the procedure; deletions and insertions occur sequentially
- Increased latency from threads holding locks
  - A thread could take a long time to finish its operation; block other threads

Lock-Based Concurrent Linked List

- Currently, we do multiple locks:
  - Traversal requires a lock
  - Insertion requires three locks
  - Deletion requires two locks
- We can try to reduce the number of locks

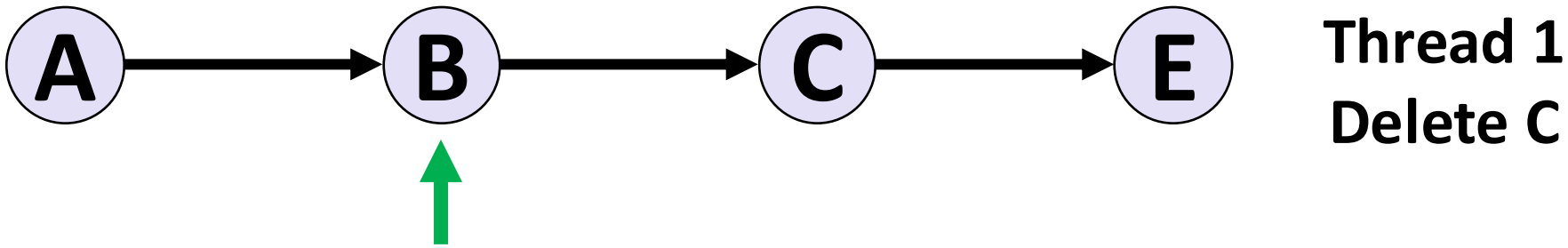
Optimistic Locking

- Be optimistic and **traverse without locking**
- **Only lock when you find the correct nodes**
- Once you have the locks, then perform the update
- Finally, release the locks



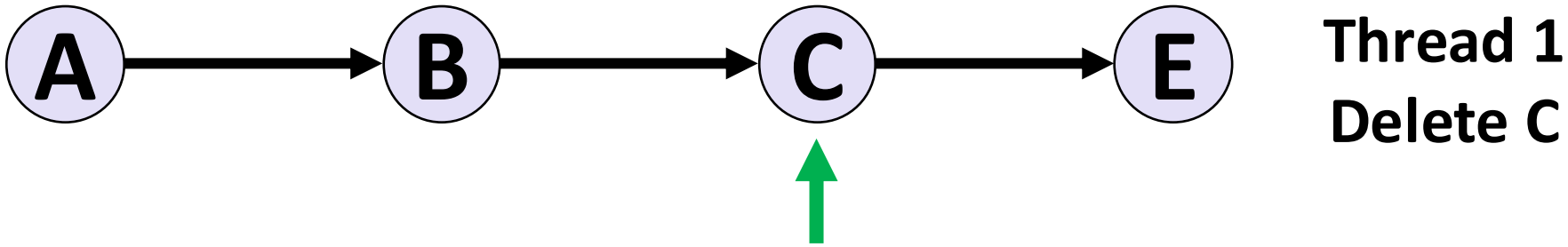
Optimistic Locking

- Be optimistic and **traverse without locking**
- **Only lock when you find the correct nodes**
- Once you have the locks, then perform the update
- Finally, release the locks



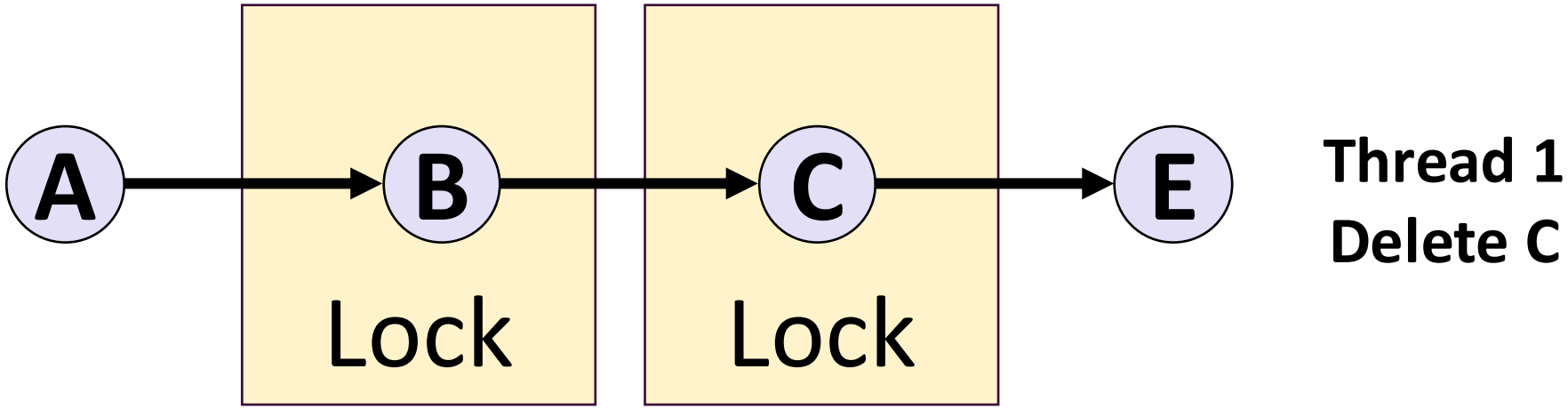
Optimistic Locking

- Be optimistic and **traverse without locking**
- **Only lock when you find the correct nodes**
- Once you have the locks, then perform the update
- Finally, release the locks



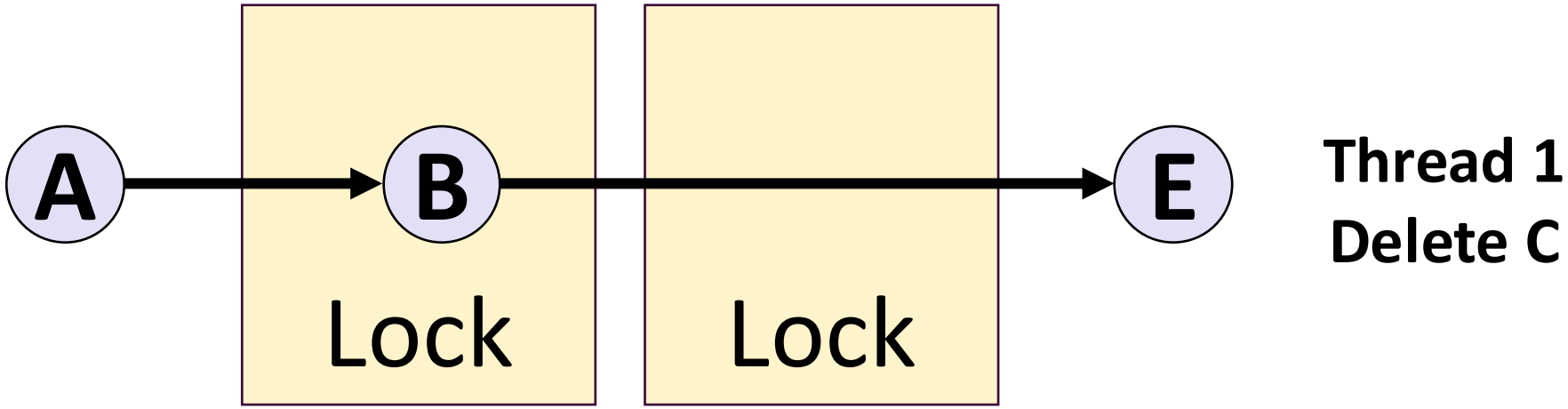
Optimistic Locking

- Be optimistic and **traverse without locking**
- **Only lock when you find the correct nodes**
- Once you have the locks, then perform the update
- Finally, release the locks



Optimistic Locking

- Be optimistic and **traverse without locking**
- **Only lock when you find the correct nodes**
- Once you have the locks, then perform the update
- Finally, release the locks

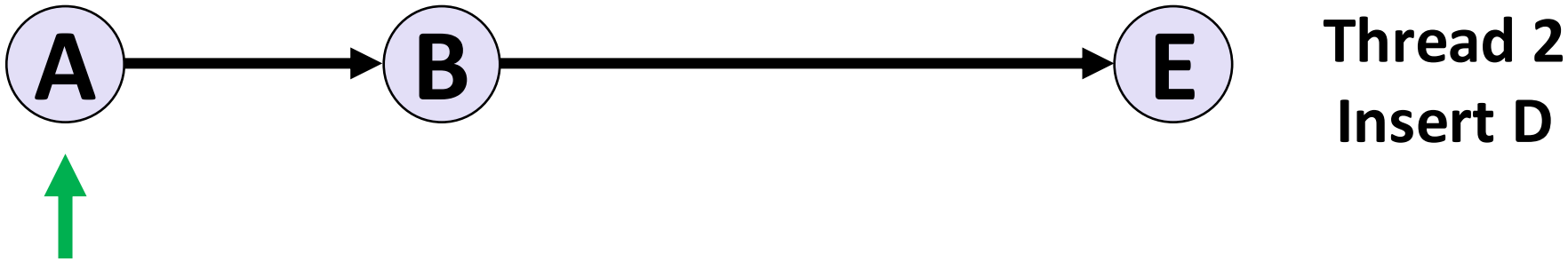


Optimistic Locking

- Be optimistic and **traverse without locking**
- **Only lock when you find the correct nodes**
- Once you have the locks, then perform the update
- Finally, release the locks

Optimistic Locking

- Be optimistic and **traverse without locking**
- **Only lock when you find the correct nodes**
- Once you have the locks, then perform the update
- Finally, release the locks



Optimistic Locking

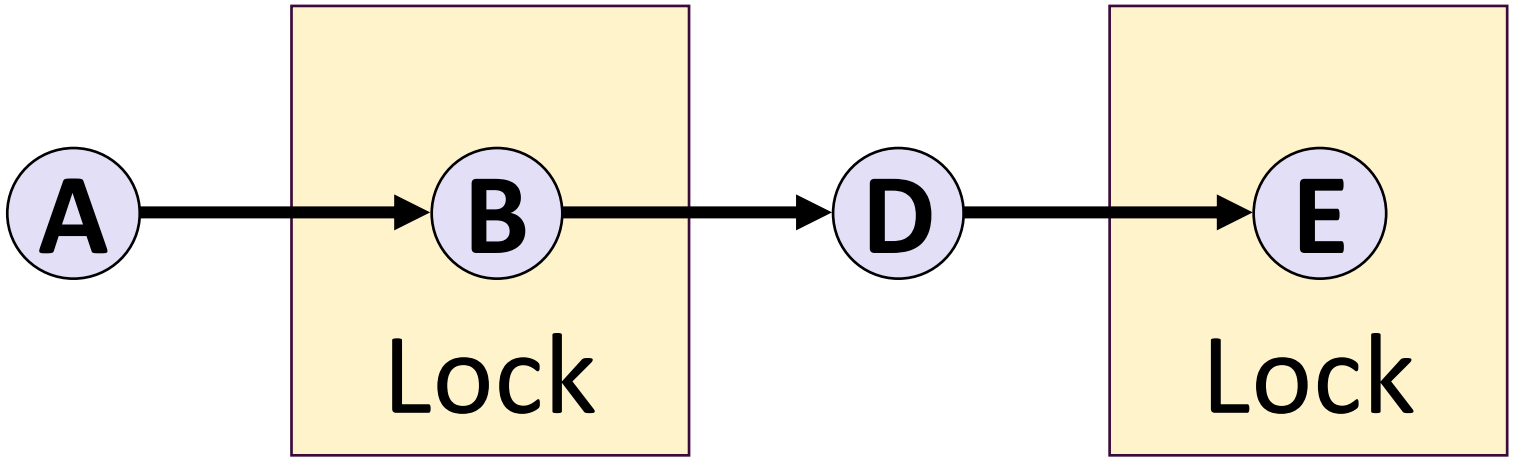
- Be optimistic and **traverse without locking**
- **Only lock when you find the correct nodes**
- Once you have the locks, then perform the update
- Finally, release the locks

Optimistic Locking

- Be optimistic and **traverse without locking**
- **Only lock when you find the correct nodes**
- Once you have the locks, then perform the update
- Finally, release the locks

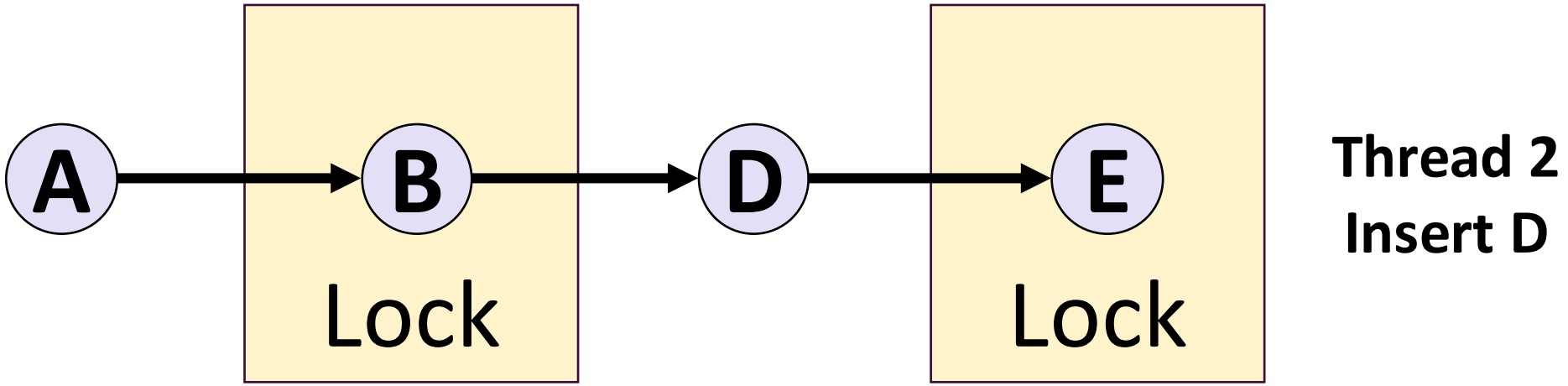
Optimistic Locking

- Be optimistic and **traverse without locking**
- **Only lock when you find the correct nodes**
- Once you have the locks, then perform the update
- Finally, release the locks



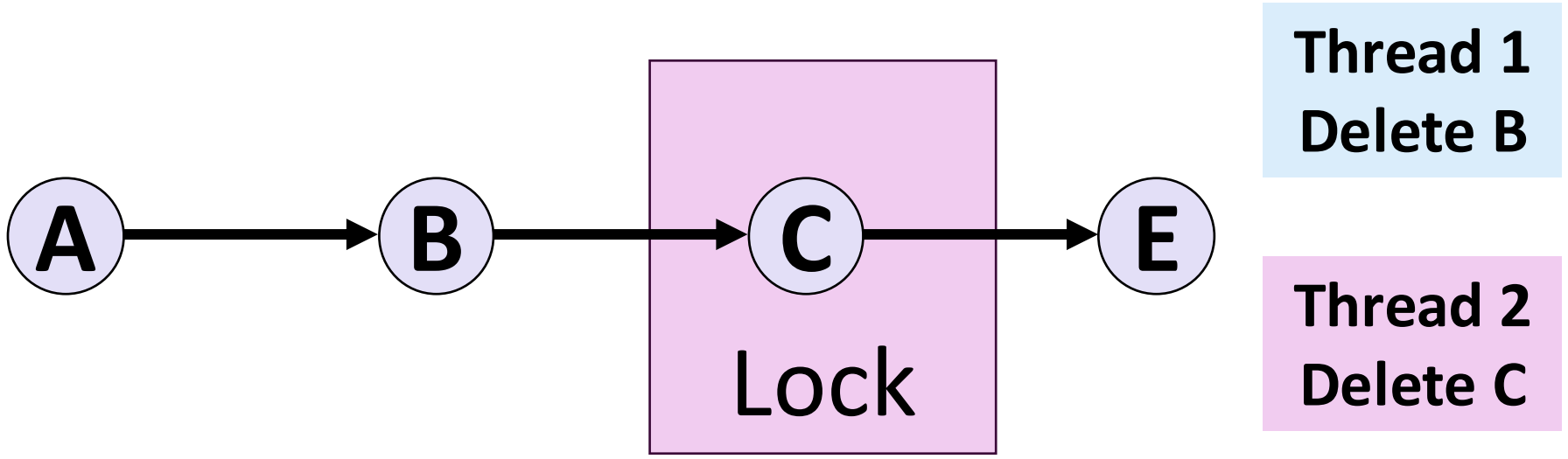
Optimistic Locking

- **Can anything go wrong with this new procedure?**



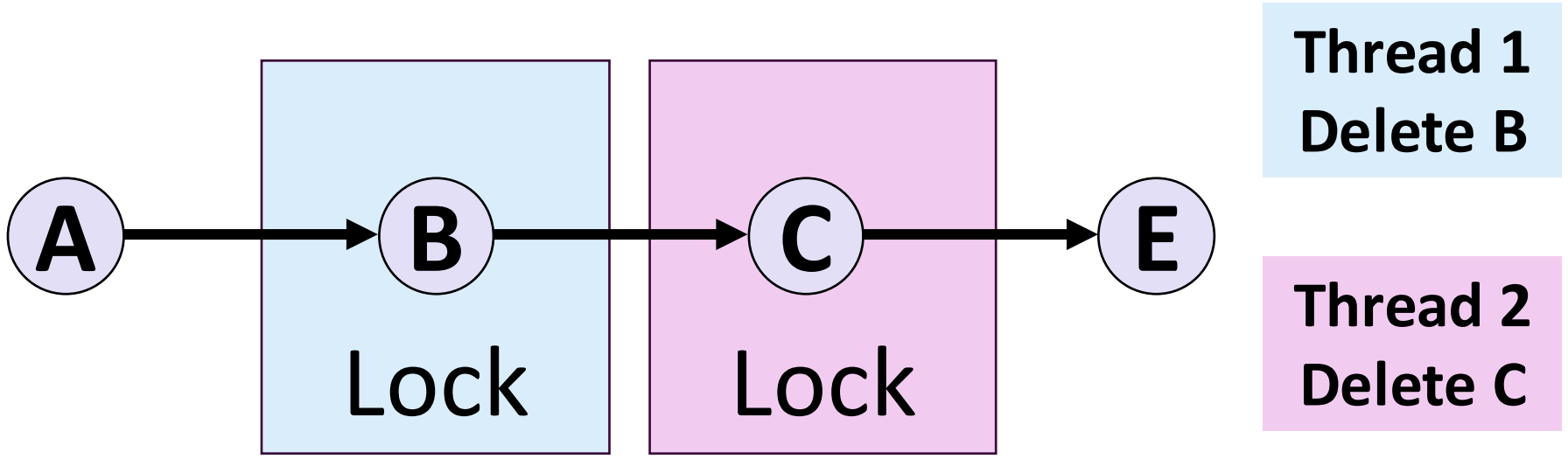
Optimistic Locking

- **Can anything go wrong with this new procedure?**
- First, **need to acquire locks on the predecessor then successor in order to prevent deadlocks**



Optimistic Locking

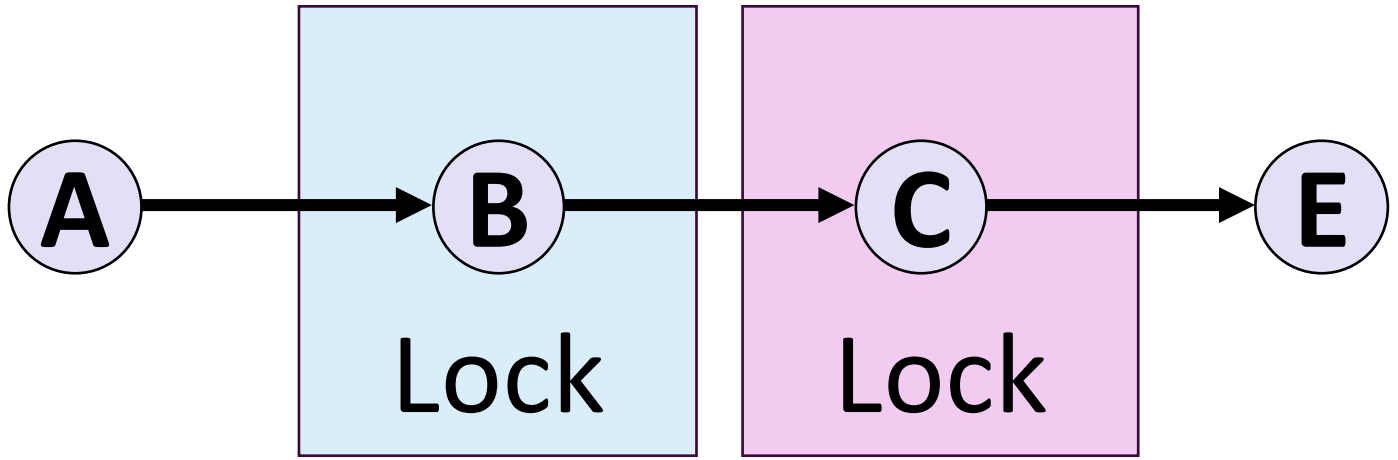
- **Can anything go wrong with this new procedure?**
- First, **need to acquire locks on the predecessor then successor in order to prevent deadlocks**



Optimistic Locking

- **Can anything go wrong with this new proc**
- First, **need to acquire locks on the predece successor in order to prevent deadlocks**

Deadlock! When **Thread 1 attempts to lock C** and **Thread 2 attempts to lock B**



Thread 1
Delete B

Thread 2
Delete C

Optimistic Locking

- **Can anything go wrong with this new procedure?**
- First, **need to acquire locks on the predecessor then successor in order to prevent deadlocks**
- **Anything else?**
  - What if you have a simultaneous insertion and deletion?

Optimistic Locking

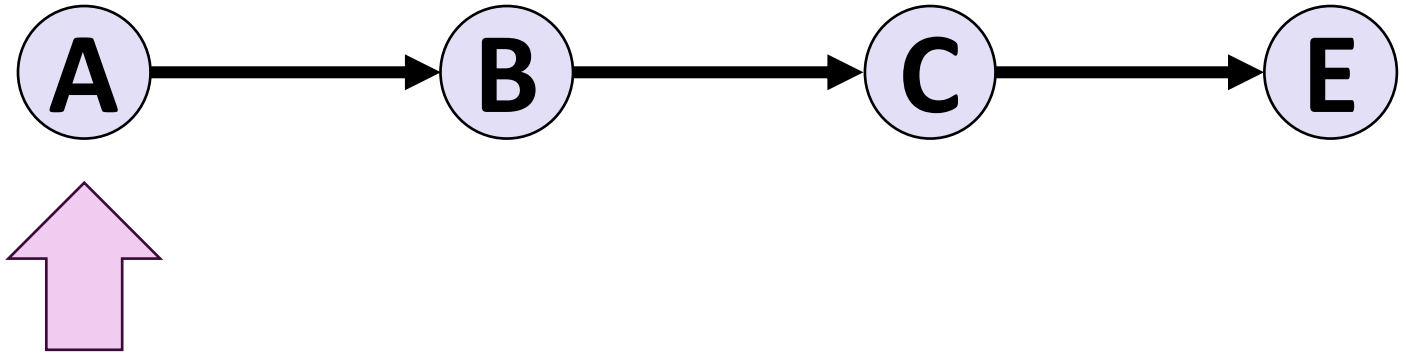
- **Anything else?**
  - What if you have a simultaneous insertion and deletion?

Thread 1
Delete C

Thread 2
Insert D

Optimistic Locking

- **Anything else?**
  - What if you have a simultaneous insertion and deletion?

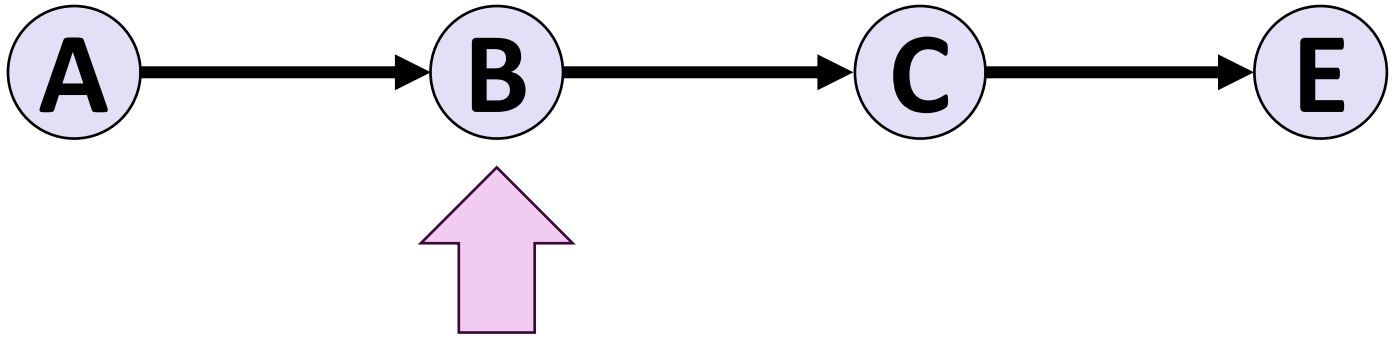


Thread 1
Delete C

Thread 2
Insert D

Optimistic Locking

- **Anything else?**
  - What if you have a simultaneous insertion and deletion?

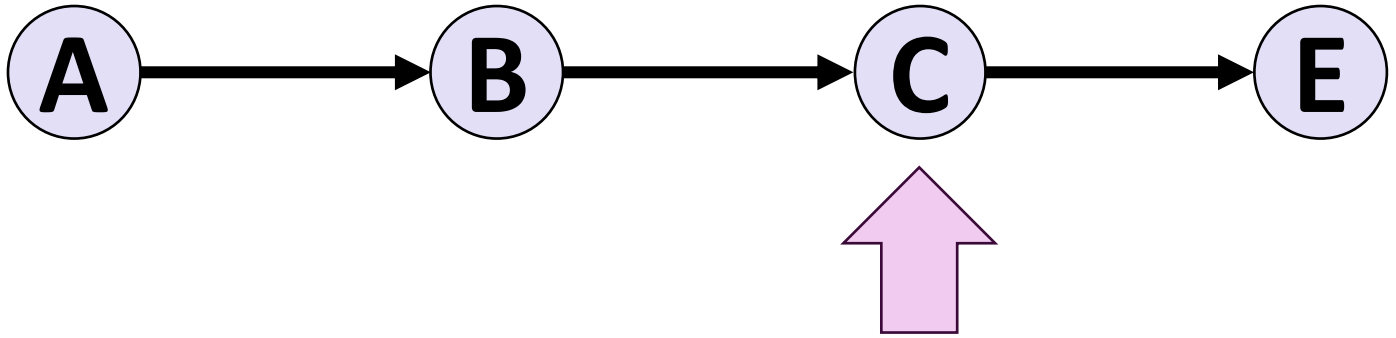


Thread 1
Delete C

Thread 2
Insert D

Optimistic Locking

- **Anything else?**
  - What if you have a simultaneous insertion and deletion?

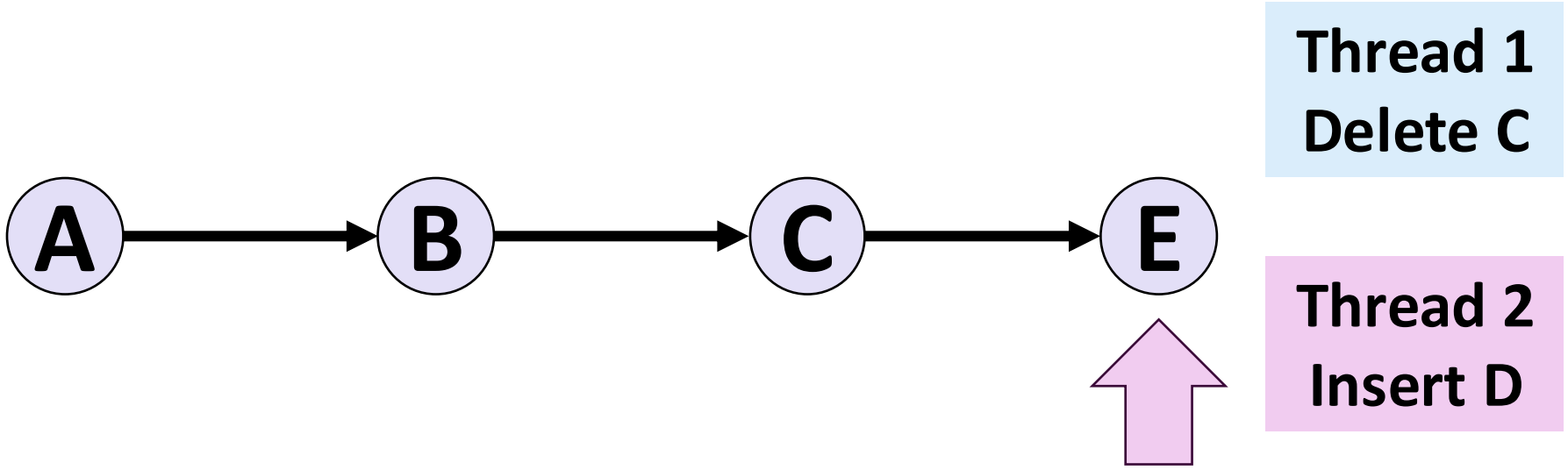


Thread 1
Delete C

Thread 2
Insert D

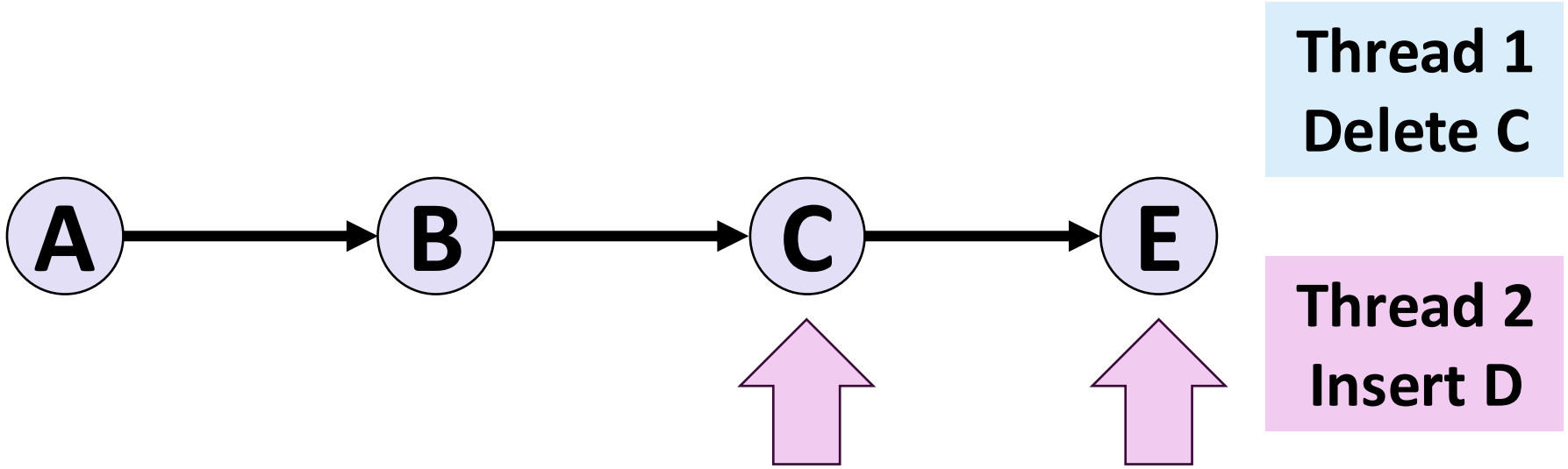
Optimistic Locking

- **Anything else?**
  - What if you have a simultaneous insertion and deletion?



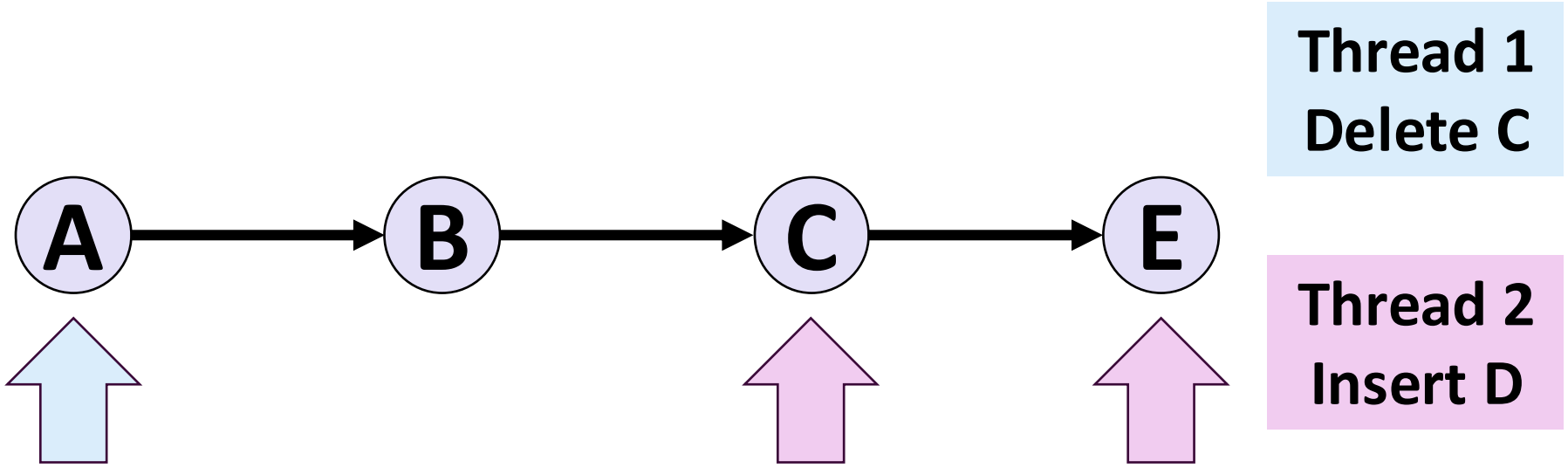
Optimistic Locking

- **Anything else?**
  - What if you have a simultaneous insertion and deletion?



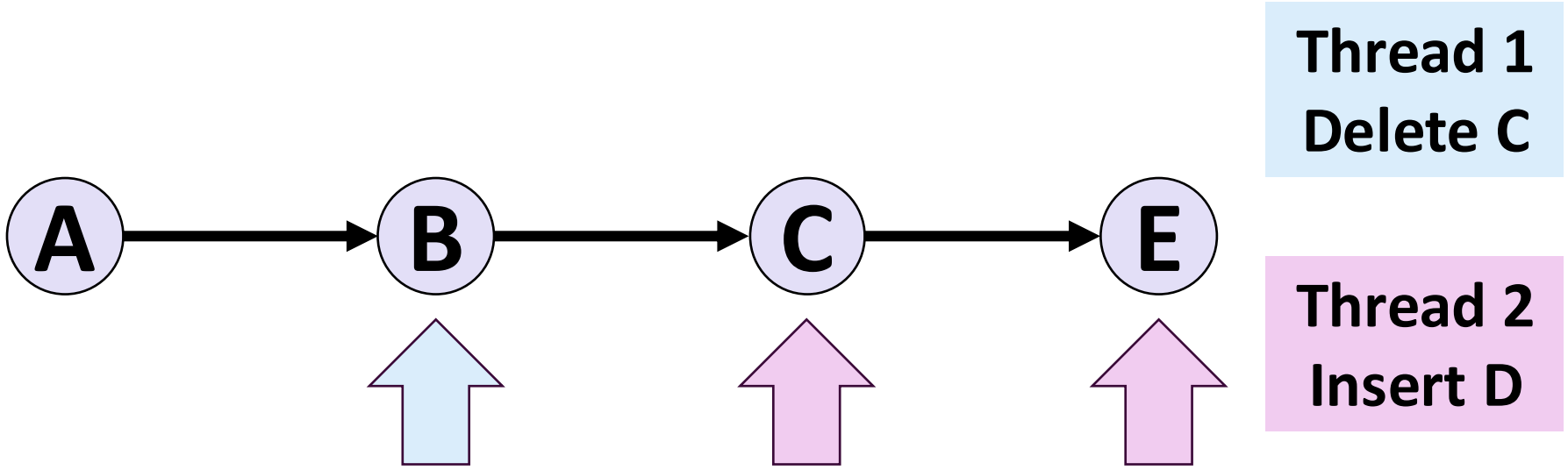
Optimistic Locking

- **Anything else?**
  - What if you have a simultaneous insertion and deletion?



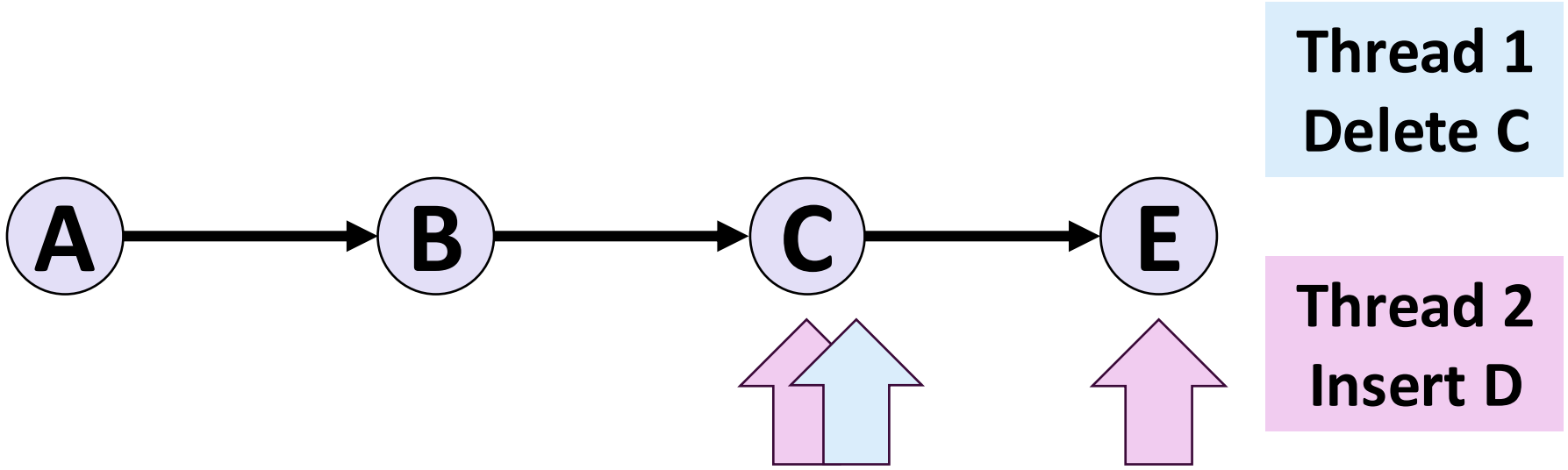
Optimistic Locking

- **Anything else?**
  - What if you have a simultaneous insertion and deletion?



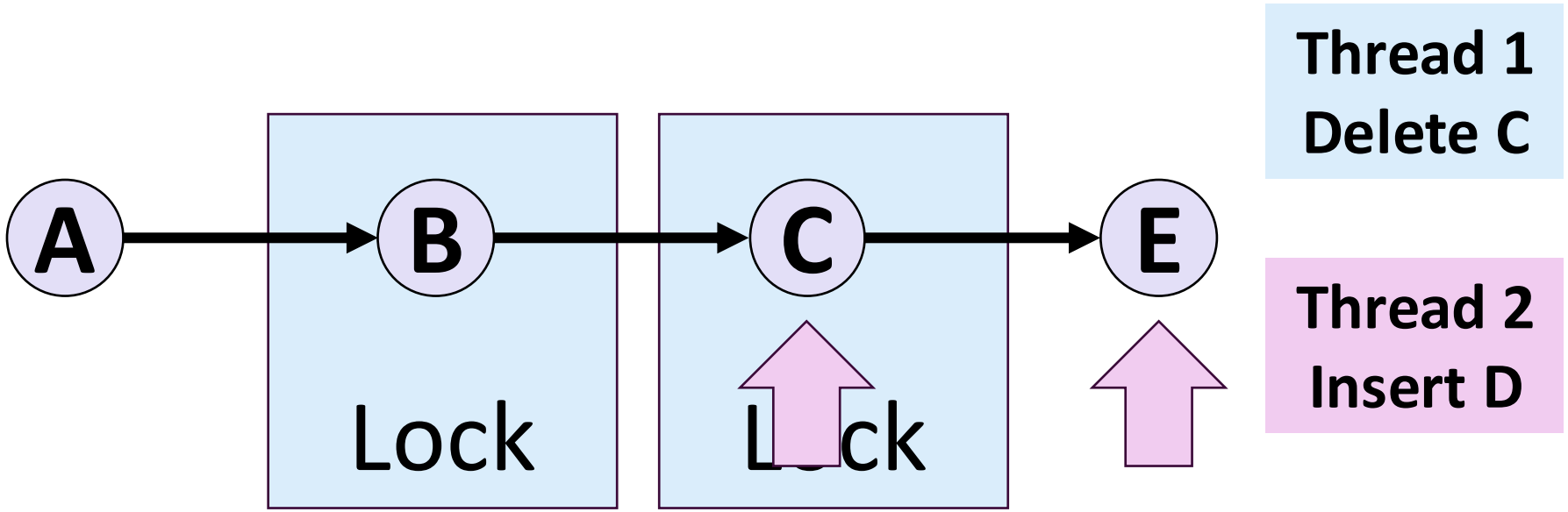
Optimistic Locking

- **Anything else?**
  - What if you have a simultaneous insertion and deletion?



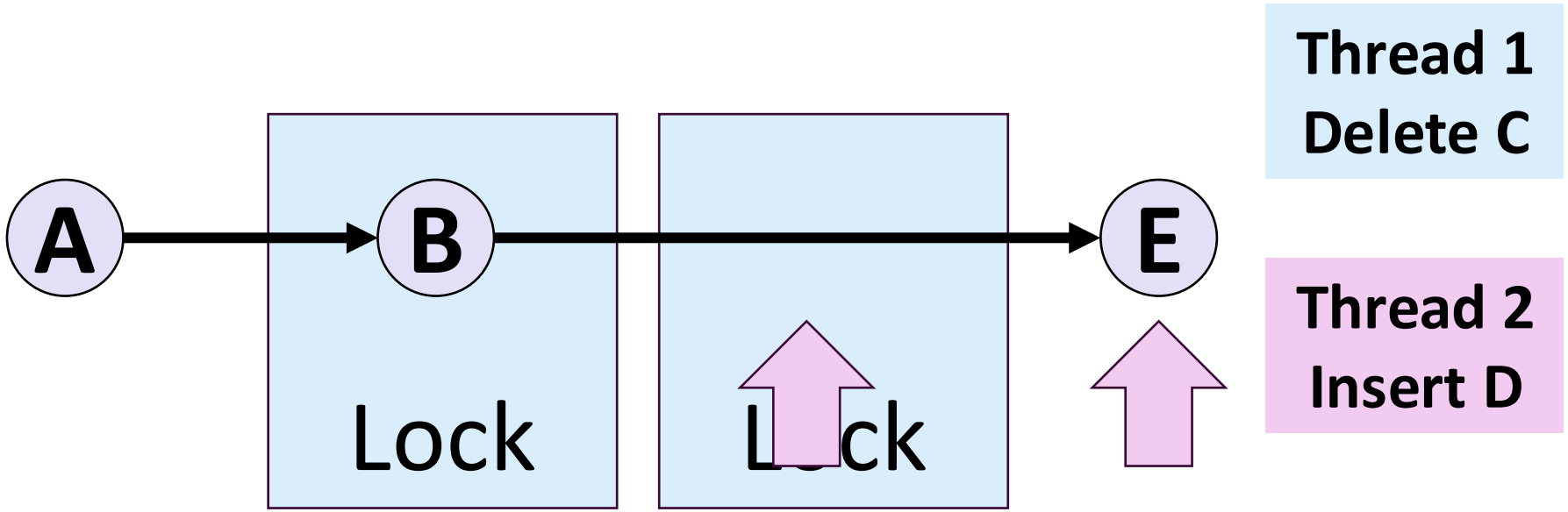
Optimistic Locking

- **Anything else?**
  - What if you have a simultaneous insertion and deletion?



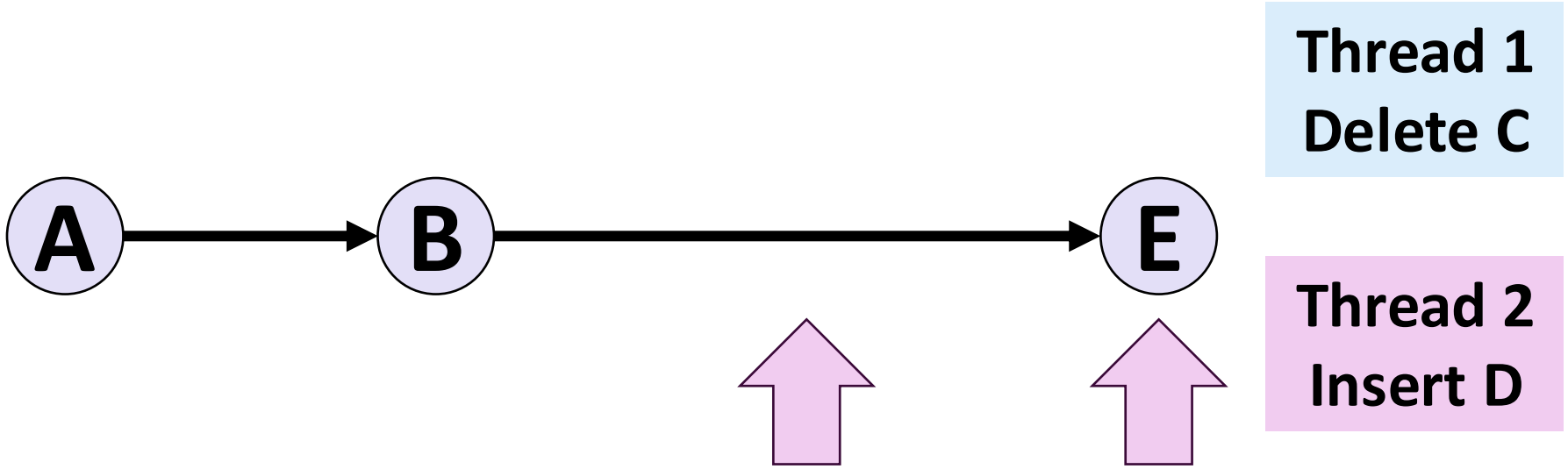
Optimistic Locking

- **Anything else?**
  - What if you have a simultaneous insertion and deletion?



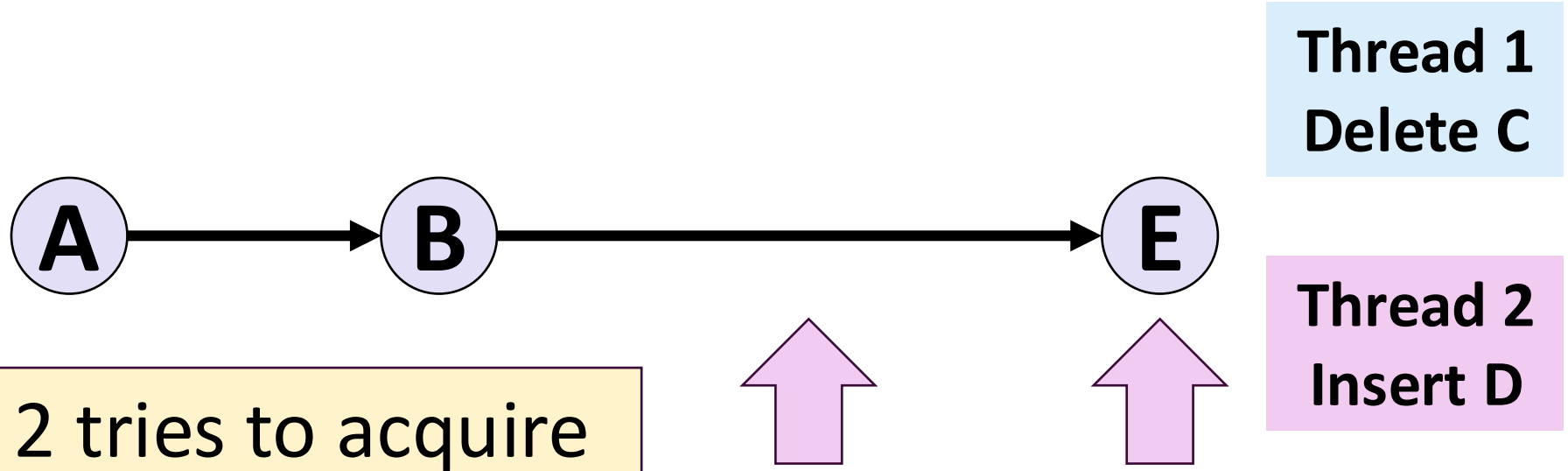
Optimistic Locking

- **Anything else?**
  - What if you have a simultaneous insertion and deletion?



Optimistic Locking

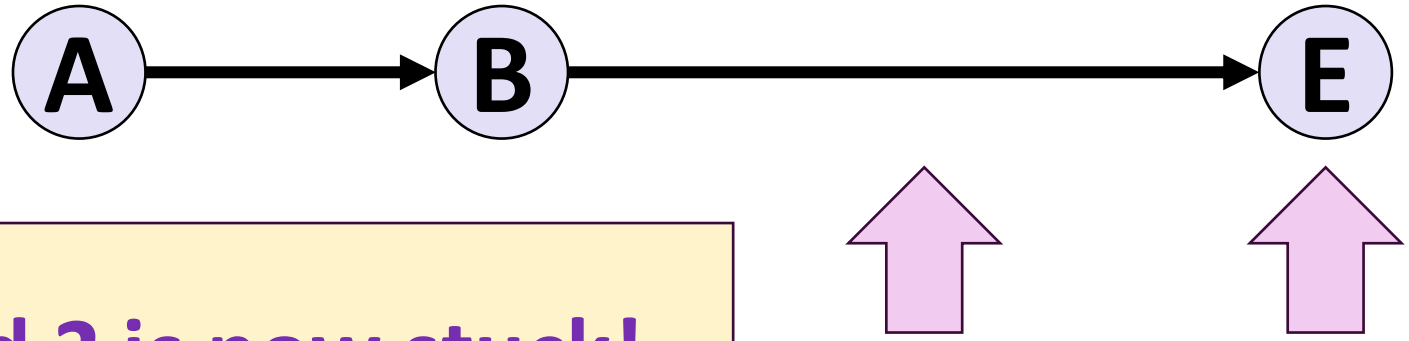
- **Anything else?**
  - What if you have a simultaneous insertion and deletion?



Thread 2 tries to acquire lock on C **but C is gone**!

Optimistic Locking

- **Anything else?**
  - What if you have a simultaneous insertion and deletion?



Thread 1
Delete C

Thread 2
Insert D

Thread 2 is now stuck!

Optimistic Locking **with Validation**

- **Validation is an important concept in parallel/concurrent algorithms**
  - Before you try to acquire locks, first validate that the state of the world is what you think it is

```
bool validate(Node* pred, Node* curr) {
        return (pred->next == curr);
}
```

Optimistic Locking **with Validation**

```
void insert(int val) {
    while (true) {
        Node* pred = head;
        Node* curr = pred->next;
        while (curr && curr->value < val) {
            pred = curr;
            curr = curr->next;
        }

        std::unique_lock<std::mutex> lockPred(pred->m);
        std::unique_lock<std::mutex> lockCurr;
        if (curr) {
            lockCurr = std::unique_lock<std::mutex>(curr->m);
        }
        if (validate(pred, curr)) {
            Node* newNode = new Node(val, curr);
            pred->next = newNode;
            // Locks are released automatically at scope exit
            return;
        }
    }
}
```

Optimistic Locking **with Validation**

```
void insert(int val) {
    while (true) {
        Node* pred = head;
        Node* curr = pred->next;
        while (curr && curr->value < val) {
            pred = curr;
            curr = curr->next;
        }

        std::unique_lock<std::mutex> lockPred(pred->m);
        std::unique_lock<std::mutex> lockCurr;
        if (curr) {
            lockCurr = std::unique_lock<std::mutex>(curr->m);
        }
```

```
        if (validate(pred, curr)) {
            Node* newNode = new Node(val, curr);
            pred->next = newNode;
            // Locks are released automatically at scope exit
            return;
        }
    }
}
```

Search for insertion place without locking

Optimistic Locking **with Validation**

```
void insert(int val) {
    while (true) {
        Node* pred = head;
        Node* curr = pred->next;
        while (curr && curr->value < val) {
            pred = curr;
            curr = curr->next;
        }

        std::unique_lock<std::mutex> lockPred(pred->m);
        std::unique_lock<std::mutex> lockCurr;
        if (curr) {
            lockCurr = std::unique_lock<std::mutex>(curr->m);
        }
        if (validate(pred, curr)) {
            Node* newNode = new Node(val, curr);
            pred->next = newNode;
            // Locks are released automatically at scope exit
            return;
        }
    }
}
```

Lock only when found insertion location

Optimistic Locking **with Validation**

```
void insert(int val) {
    while (true) {
        Node* pred = head;
        Node* curr = pred->next;
        while (curr && curr->value < val) {
            pred = curr;
            curr = curr->next;
        }

        std::unique_lock<std::mutex> lockPred(pred->m);
        std::unique_lock<std::mutex> lockCurr;
        if (curr) {
            lockCurr = std::unique_lock<std::mutex>(curr->m);
        }
```

```
        if (validate(pred, curr)) {
            Node* newNode = new Node(val, curr);
            pred->next = newNode;
            // Locks are released automatically at scope exit
            return;
        }
    }
}
```

Validate that the predecessor and successor still contain the link you want

Optimistic Locking **with Validation**

```
void insert(int val) {
    while (true) {
        Node* pred = head;
        Node* curr = pred->next;
        while (curr && curr->value < val) {
            pred = curr;
            curr = curr->next;
        }

        std::unique_lock<std::mutex> lockPred(pred->m);
        std::unique_lock<std::mutex> lockCurr;
        if (curr) {
            lockCurr = std::unique_lock<std::mutex>(curr->m);
        }
```

```
        if (validate(pred, curr)) {
            Node* newNode = new Node(val, curr);
            pred->next = newNode;
            // Locks are released automatically at scope exit
            return;
        }
    }
}
```

Add the new node

Optimistic Locking **with Validation**

```
void insert(int val) {
    while (true) {
        Node* pred = head;
        Node* curr = pred->next;
        while (curr && curr->value < val) {
            pred = curr;
            curr = curr->next;
        }

        std::unique_lock<std::mutex> lockPred(pred->m);
        std::unique_lock<std::mutex> lockCurr;
        if (curr) {
            lockCurr = std::unique_lock<std::mutex>(curr->m);
        }
        if (validate(pred, curr)) {
            Node* newNode = new Node(val, curr);
            pred->next = newNode;
            // Locks are released automatically at scope exit
            return;
        }
    }
}
```

But there is still a problem!

Optimistic Locking **with Validation**

```
void insert(int val) {
    while (true) {
        Node* pred = head;
        Node* curr = pred->next;
        while (curr && curr->value < val) {
            pred = curr;
            curr = curr->next;
        }

        std::unique_lock<std::mutex> lockPred(pred->m);
        std::unique_lock<std::mutex> lockCurr;
        if (curr) {
            lockCurr = std::unique_lock<std::mutex>(curr->m);
        }
        if (validate(pred, curr)) {
            Node* newNode = new Node(val, curr);
            pred->next = newNode;
            // Locks are released automatically at scope exit
            return;
        }
    }
}
```

What is the problem with this code?

Optimistic Locking **with Validation**

```
void insert(int val) {
    while (true) {
        Node* pred = head;
        Node* curr = pred->next;
        while (curr && curr->value < val) {
            pred = curr;
            curr = curr->next;
        }

        std::unique_lock<std::mutex> lockPred(pred->m);
        std::unique_lock<std::mutex> lockCurr;
        if (curr) {
            lockCurr = std::unique_lock<std::mutex>(curr->m);
        }
        if (validate(pred, curr)) {
            Node* newNode = new Node(val, curr);
            pred->next = newNode;
            // Locks are released automatically at scope exit
            return;
        }
    }
}
```

Node pred could be deleted before you attempt to lock it

Optimistic Locking **with Validation**

```
void insert(int val) {
    while (true) {
        Node* pred = head;
        Node* curr = pred->next;
        while (curr && curr->value < val) {
            pred = curr;
            curr = curr->next;
        }

        std::unique_lock<std::mutex> lockPred(pred->m);
        std::unique_lock<std::mutex> lockCurr;
        if (curr) {
            lockCurr = std::unique_lock<std::mutex>(curr->m);
        }
```

```
        if (validate(pred, curr)) {
            Node* newNode = new Node(val, curr);
            pred->next = newNode;
            // Locks are released automatically at scope exit
            return;
        }
    }
}
```

One fix: do not delete nodes from memory; only delete the pointers

Optimistic Locking **with Validation**

```
void insert(int val) {
    while (true) {
        Node* pred = head;
        Node* curr = pred->next;
        while (curr && curr->value < val) {
            pred = curr;
            curr = curr->next;
        }

        std::unique_lock<std::mutex> lockPred(pred->m);
        std::unique_lock<std::mutex> lockCurr;
        if (curr) {
            lockCurr = std::unique_lock<std::mutex>(curr->m);
        }
        if (validate(pred, curr)) {
            Node* newNode = new Node(val, curr);
            pred->next = newNode;
            // Locks are released automatically at scope exit
            return;
        }
    }
}
```

However, this could result in unbounded memory!

Optimistic Locking **with Validation**

```
void insert(int val) {
    while (true) {
        Node* pred = head;
        Node* curr = pred->next
        while (curr && curr->value < val) {
            pred = curr;
            curr = curr->next;
        }

        std::unique_lock<std::mutex> lockPred(pred->m);
        std::unique_lock<std::mutex> lockCurr;
        if (curr) {
            lockCurr = std::unique_lock<std::mutex>(curr->m);
        }
```

```
        if (validate(pred, curr)) {
            Node* newNode = new Node(val, curr);
            pred->next = newNode;
            // Locks are released automatically at scope exit
            return;
        }
    }
}
```

How do we fix this new problem? Think about it for next class.

Runtimes of different locking mechanisms

Linked List with 500 elements

Hand-over-Hand Locking: 39116 ms
Optimistic Locking: 6447 ms

6x improvement!

Lecture 10 Quiz