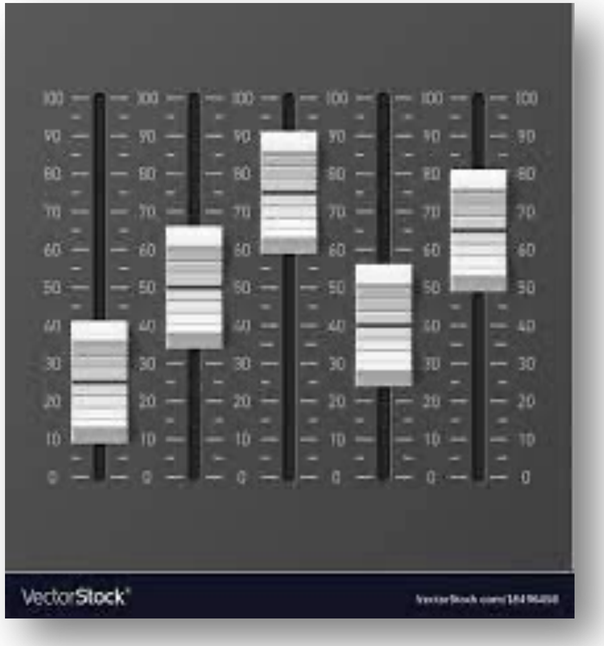
CSE113: Parallel Programming

April 29, 2021

- **Topic**: Concurrent Objects 2
  - More SC examples!
  - Linearizability
  - A concurrent set



Announcements

- Midterm will be released today by midnight (probably earlier)
  - No discussions, only private clarifying questions to teach staff.
  - We will keep a running discussion on Canvas for clarifying questions
  - Give yourself time to do both homework 2 and midterm

- We are working on grades for HW1, hopefully by next week.

Announcements

Homework

- We can start sharing results next week (throughput, variance)
- Is variance a good metric for part 1? Maybe not the best. Have a look at @76
  - coefficient of variation
  - changing results to percentages

- What does fairness mean in #2?
  - You can do it with sleeps, yields
  - You can also do it logically.
  - Try both! (next year I will require both ☺)

- Part 3:
  - You do not need to ”upgrade” the lock from reader to writer atomically! You do need to perform the swap atomically though.

Announcements

- Guest lecture on May 20!
  - Hugues Evrard (Google) will talk about message passing concurrency
  - Alastair Donaldson (Imperial College London) will talk about testing GPU compilers

Quiz

- Thank you! Quiz numbers almost exactly matched attendance last time

Quiz

- Discuss answers
- Question using non-thread safe objects: Java has finally blocks, C++ has destructors

```
void foo() {
   m.lock();
   x = vector.at(120);
   m.unlock();
}
```

```
void foo() {
   lock_guard<mutex> lock(m);
   x = vector.at(120);
}
```

Lecture schedule

- Revisiting sequential consistency
- Linearizablity
- Progress Properties
- Implementing a set

Lecture schedule

- **Revisiting sequential consistency**
- Linearizablity
- Progress Properties
- Implementing a set

More SC examples!!

To make up for my mistake last lecture

Global variable:

```
CQueue<int> q;
```

Thread 0:

```
q.enq(6);
q.enq(7);
```

Thread 1:

```
int t0 = q.dec();
int t1 = q.dec();
```

Is it possible for t0 to contain 7 and t1 to contain 6?

Global variable:

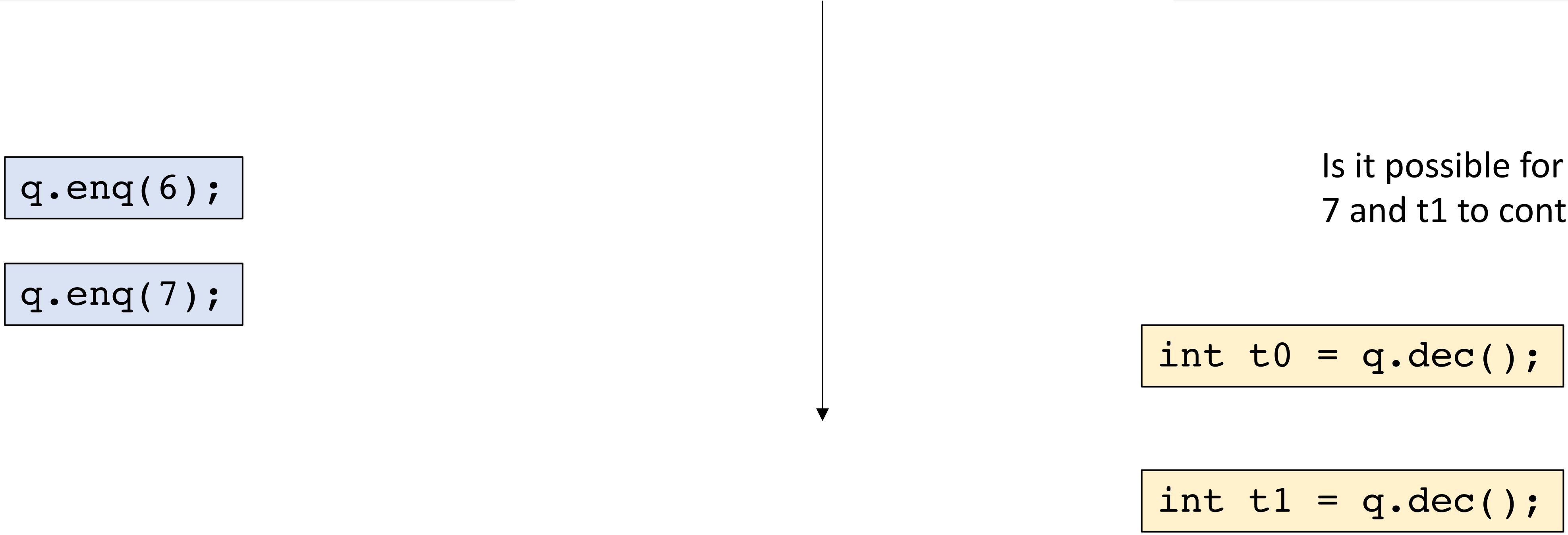
```
CQueue<int> q;
```

Thread 0:

```
q.enq(6);
q.enq(7);
```

Thread 1:

```
int t0 = q.dec();
int t1 = q.dec();
```



`q.enq(6);`

`q.enq(7);`

Is it possible for t0 to contain 7 and t1 to contain 6?

`int t0 = q.dec();`

`int t1 = q.dec();`

Global variable:

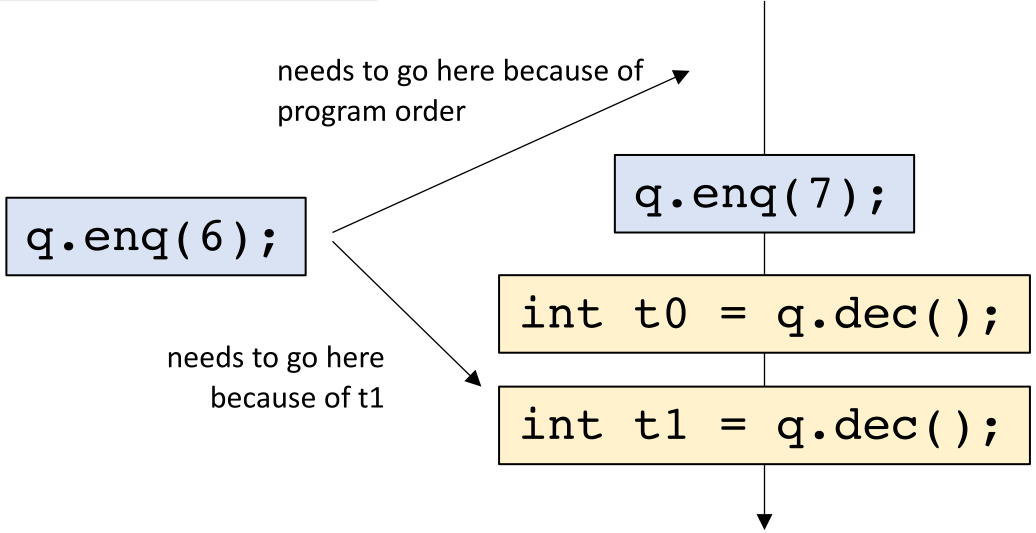
```
CQueue<int> q;
```

Thread 0:

```
q.enq(6);
q.enq(7);
```

Thread 1:

```
int t0 = q.dec();
int t1 = q.dec();
```



needs to go here because of program order

```
q.enq(7);
```

```
q.enq(6);
```

```
int t0 = q.dec();
```

needs to go here because of t1

```
int t1 = q.dec();
```

Is it possible for t0 to contain 7 and t1 to contain 6?

Global variable:

```
CStack<int> s;
```

FILO object

Thread 0:

```
s.push(6);
s.push(7);
```

Thread 1:

```
int t0 = s.pop();
int t1 = s.pop();
```

Is it possible for t0 to contain 7 and t1 to contain 6?

Global variable:

```
CStack<int> s;
```

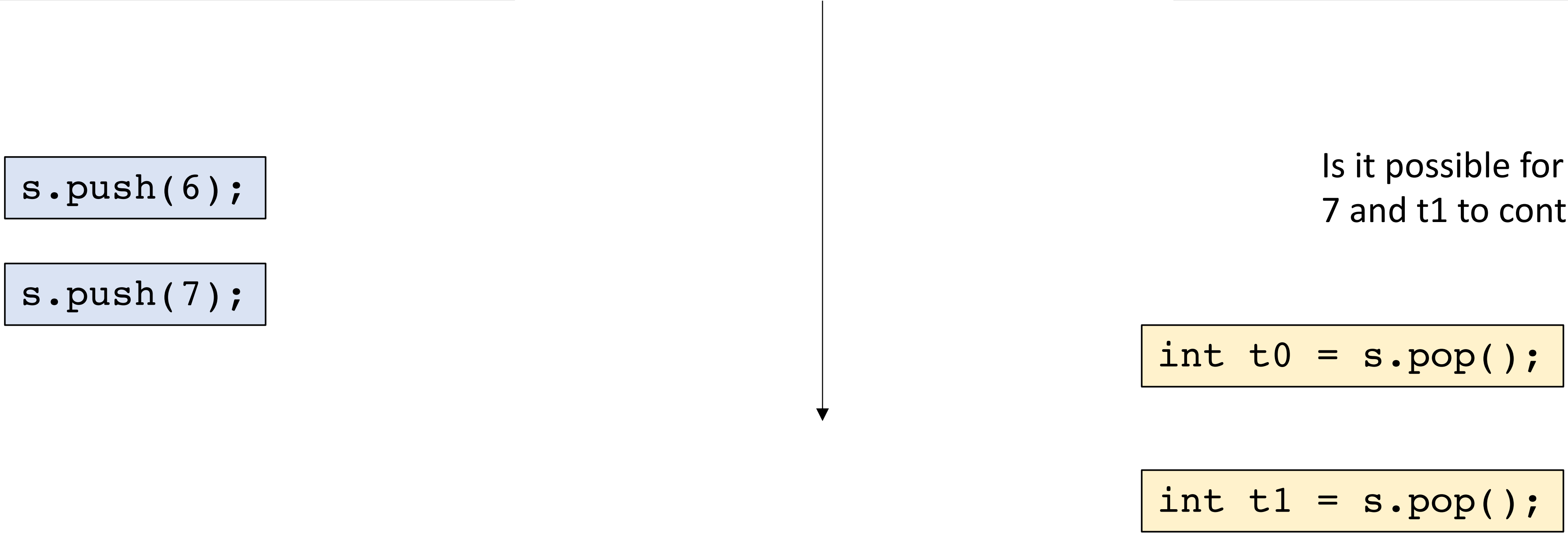
FILO object

Thread 0:

```
s.push(6);
s.push(7);
```

Thread 1:

```
int t0 = s.pop();
int t1 = s.pop();
```



```
s.push(6);
```

```
s.push(7);
```

```
int t0 = s.pop();
```

```
int t1 = s.pop();
```

Is it possible for t0 to contain 7 and t1 to contain 6?

Global variable:

```
CStack<int> s;
```

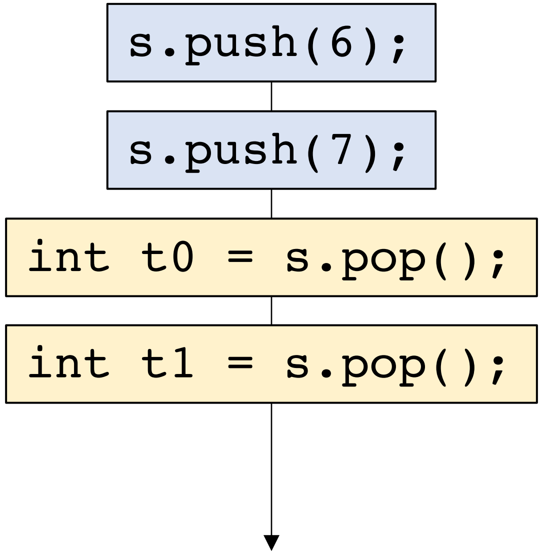
FILO object

Thread 0:

```
s.push(6);
s.push(7);
```

Thread 1:

```
int t0 = s.pop();
int t1 = s.pop();
```



```
s.push(6);
s.push(7);
int t0 = s.pop();
int t1 = s.pop();
```

Is it possible for t0 to contain 7 and t1 to contain 6?

Global variable:

```
CStack<int> s;
```

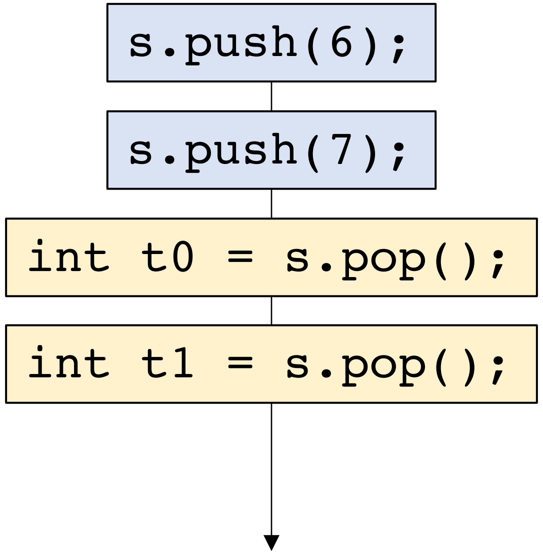
FILO object

Thread 0:

```
s.push(6);
s.push(7);
```

Thread 1:

```
int t0 = s.pop();
int t1 = s.pop();
```



```
s.push(6);
s.push(7);
int t0 = s.pop();
int t1 = s.pop();
```

Is it possible for t0 to contain 7 and t1 to contain 0?

Global variable:

```
CStack<int> s;
```

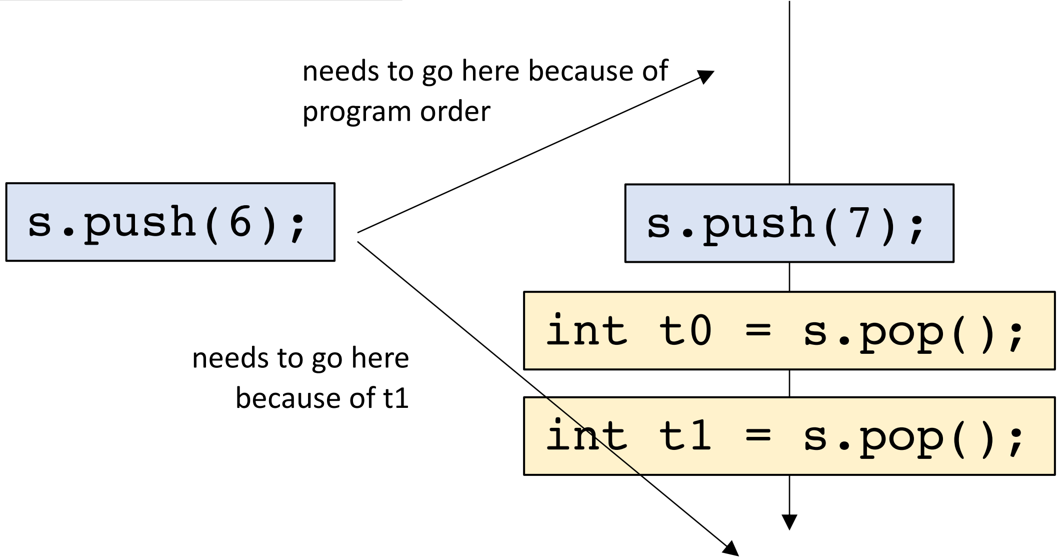
FILO object

Thread 0:

```
s.push(6);
s.push(7);
```

Thread 1:

```
int t0 = s.pop();
int t1 = s.pop();
```



Is it possible for t0 to contain 7 and t1 to contain 0?

Global variable:

```
CQueue<int> q,p;
```

Multiple objects

Thread 0:

```
p.enq(1);
int t0 = q.dec();
```

Thread 1:

```
q.enq(1);
int t1 = p.dec();
```

Is it possible for t0 and t1 to contain 0 at the end of this program?

Global variable:

```
CQueue<int> q,p;
```

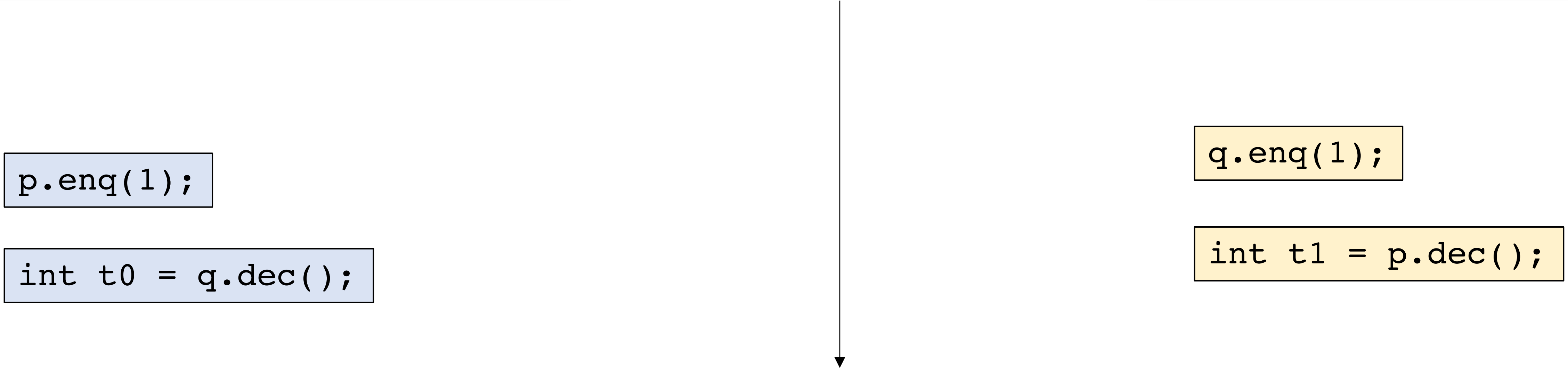
Multiple objects

Thread 0:

```
p.enq(1);
int t0 = q.dec();
```

Thread 1:

```
q.enq(1);
int t1 = p.dec();
```



```
p.enq(1);
```

```
int t0 = q.dec();
```

```
q.enq(1);
```

```
int t1 = p.dec();
```

Is it possible for t0 and t1 to contain 0 at the end of this program?

Global variable:

```
CQueue<int> q,p;
```

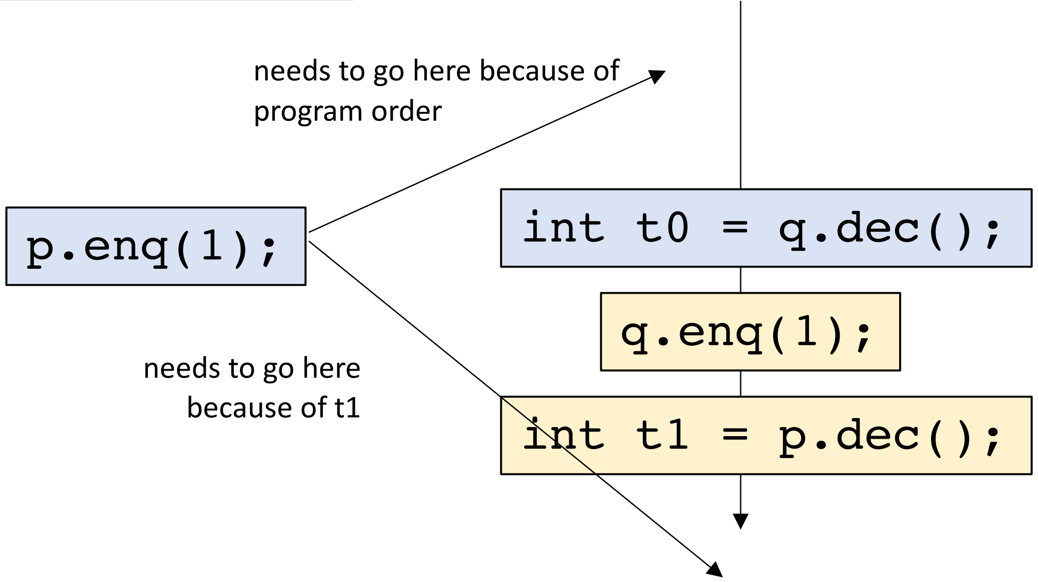
Multiple objects

Thread 0:

```
p.enq(1);
int t0 = q.dec();
```

Thread 1:

```
q.enq(1);
int t1 = p.dec();
```



Is it possible for t0 and t1 to contain 0 at the end of this program?

Global variable:

```
CQueue<int> q,p;
```

Multiple objects

Thread 0:

```
int t0 = q.dec();
p.enq(1);
```

Thread 1:

```
int t1 = p.dec();
q.enq(1);
```

Is it possible for t0 and t1 to both contain 1 at the end of this program?

Global variable:

```
CQueue<int> q,p;
```

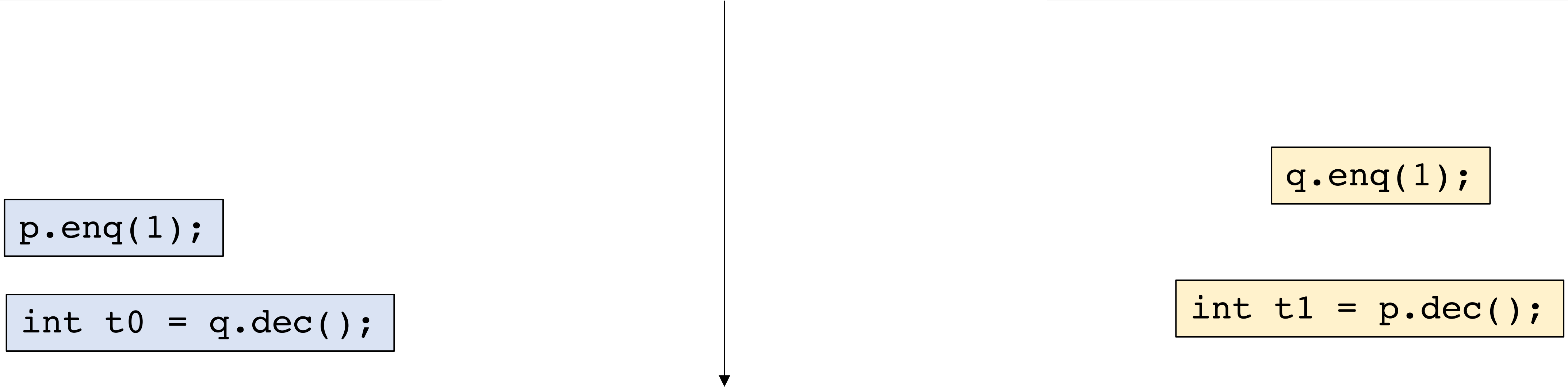
Multiple objects

Thread 0:

```
int t0 = q.dec();
p.enq(1);
```

Thread 1:

```
int t1 = p.dec();
q.enq(1);
```



```
q.enq(1);
```

```
p.enq(1);
```

```
int t1 = p.dec();
```

```
int t0 = q.dec();
```

Is it possible for t0 and t1 to both contain 1 at the end of this program?

Global variable:

```
CQueue<int> q,p;
```

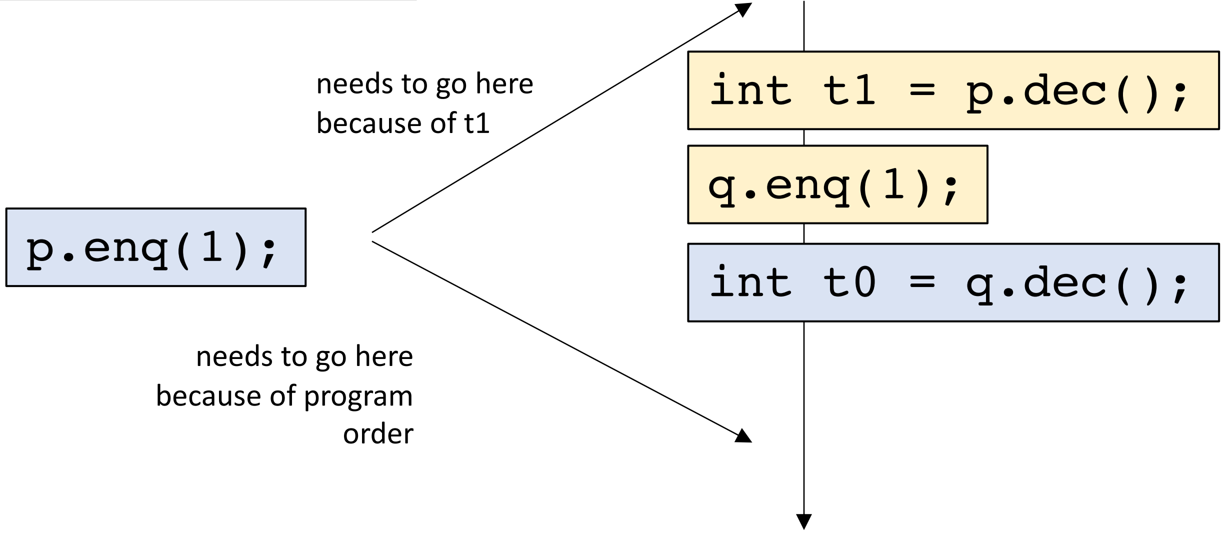
Multiple objects

Thread 0:

```
int t0 = q.dec();
p.enq(1);
```

Thread 1:

```
int t1 = p.dec();
q.enq(1);
```



```
p.enq(1);
```

needs to go here because of t1

needs to go here because of program order

```
int t1 = p.dec();
q.enq(1);
int t0 = q.dec();
```

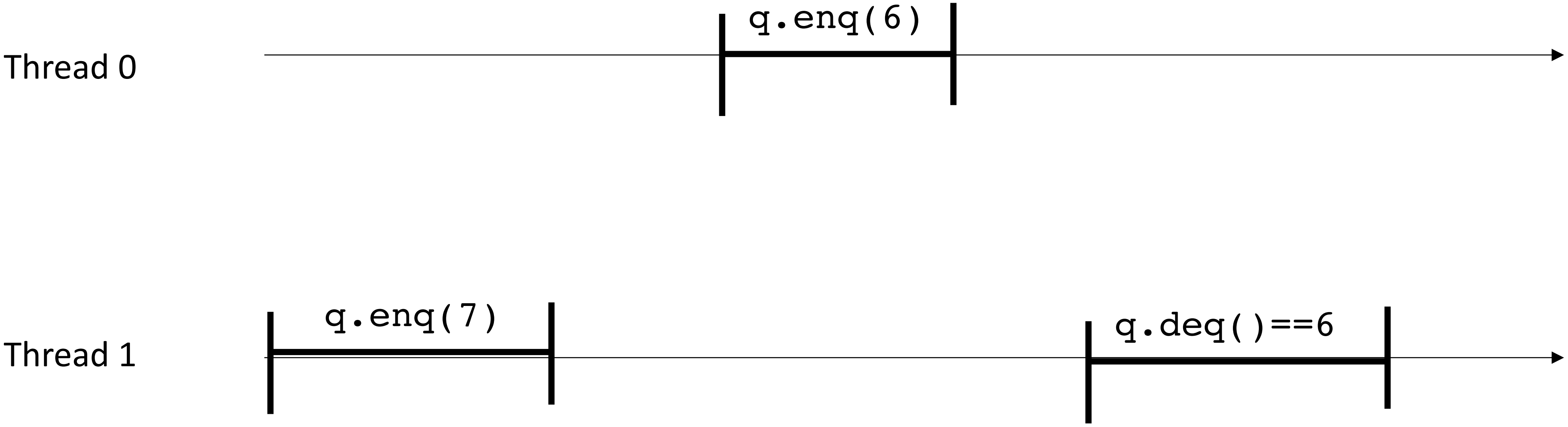
Is it possible for t0 and t1 to both contain 1 at the end of this program?

Remember the issue with sequential const.

Sequential consistency and real time

- Add in real time:

This timeline seems strange...



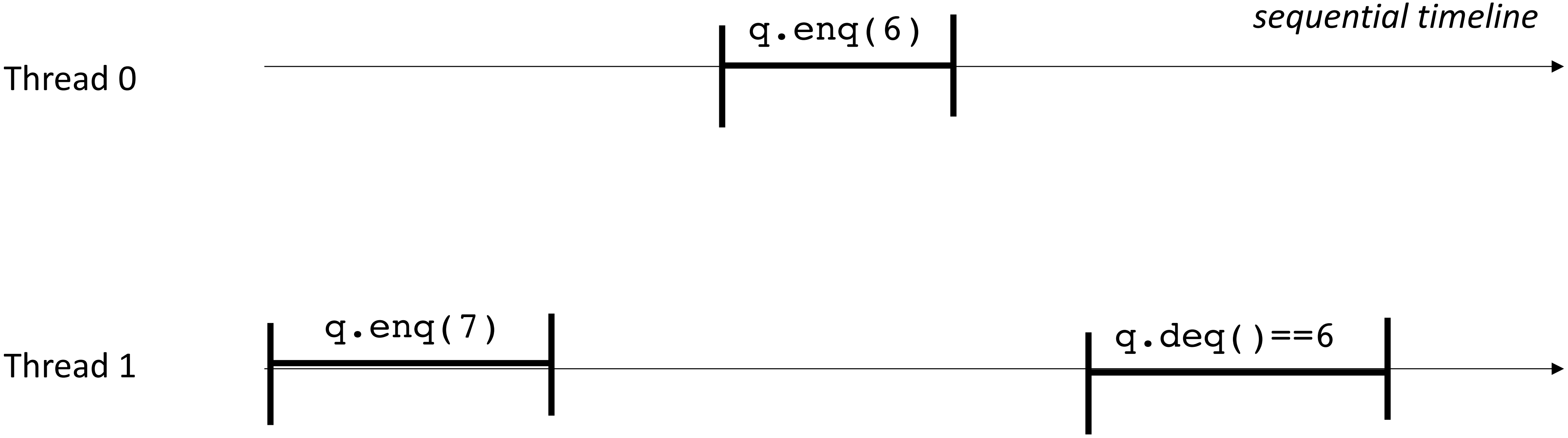
real time line

Sequential consistency and real time

- Add in real time:

This execution is allowed in sequential consistency!

SC doesn't care about real time, only if it can construct its virtual sequential timeline



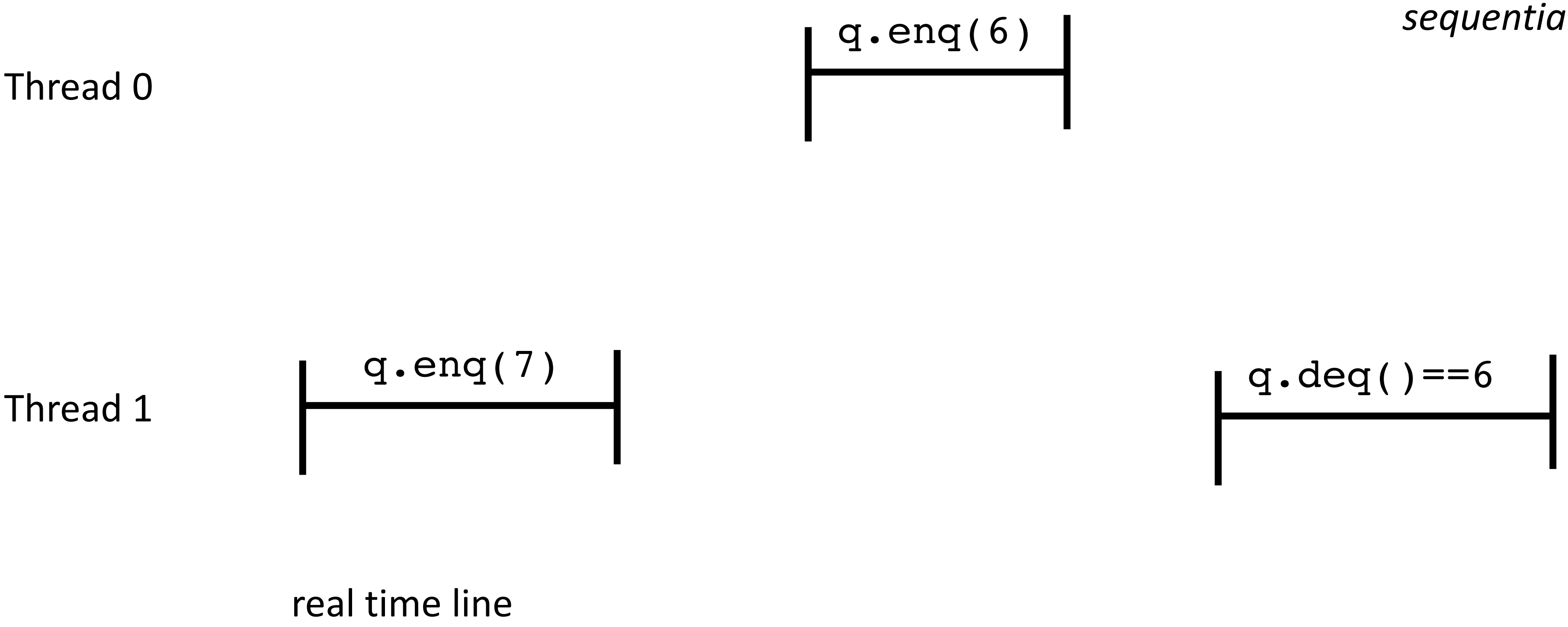
real time line

Sequential consistency and real time

- Add in real time:

This execution is allowed in sequential consistency!

SC doesn't care about real time, only if it can construct its virtual sequential timeline



Sequential consistency and real time

- Add in real time:

This execution is allowed in sequential consistency!

SC doesn't care about real time, only if it can construct its virtual sequential timeline

`q.enq(6)`

Thread 0

`q.enq(7);  q.deq()==6`

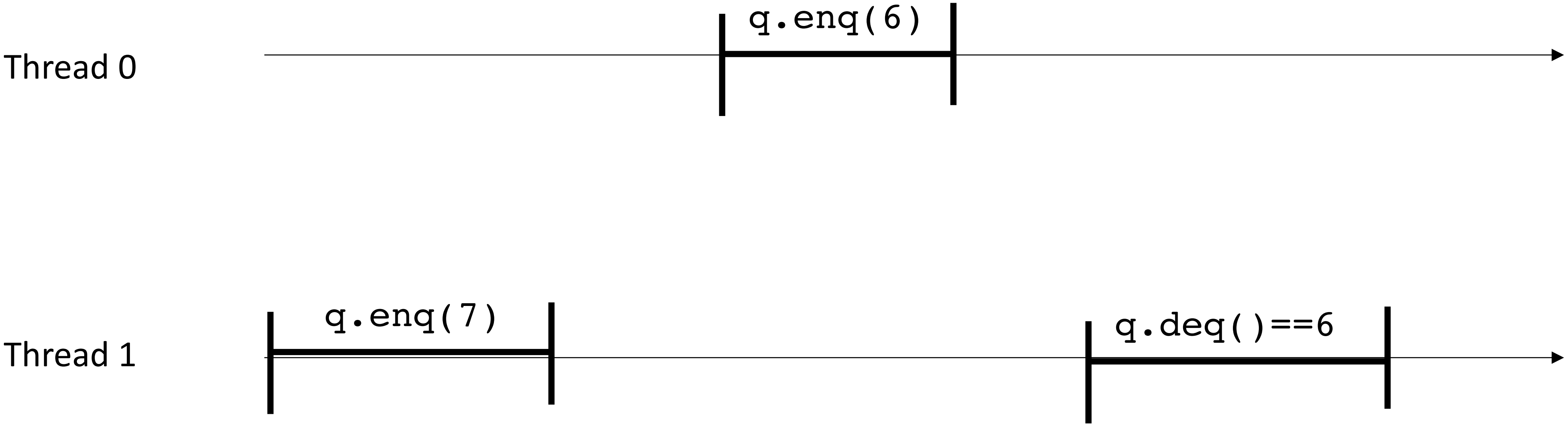
Thread 1

real time line

Sequential consistency and real time

Why might this actually happen?

- Add in real time:

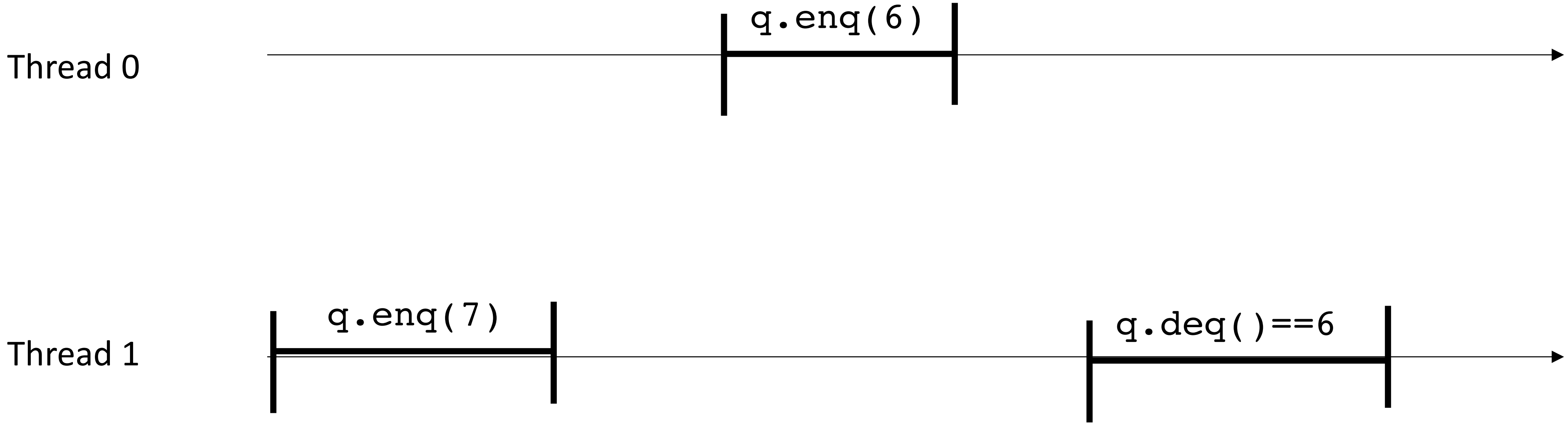


Sequential consistency and real time

- Add in real time:

Why might this actually happen?

asynchronous calls (like printf), e.g. it buffers the value before publishing it?
Lazy publishing (e.g. cache values in registers)?

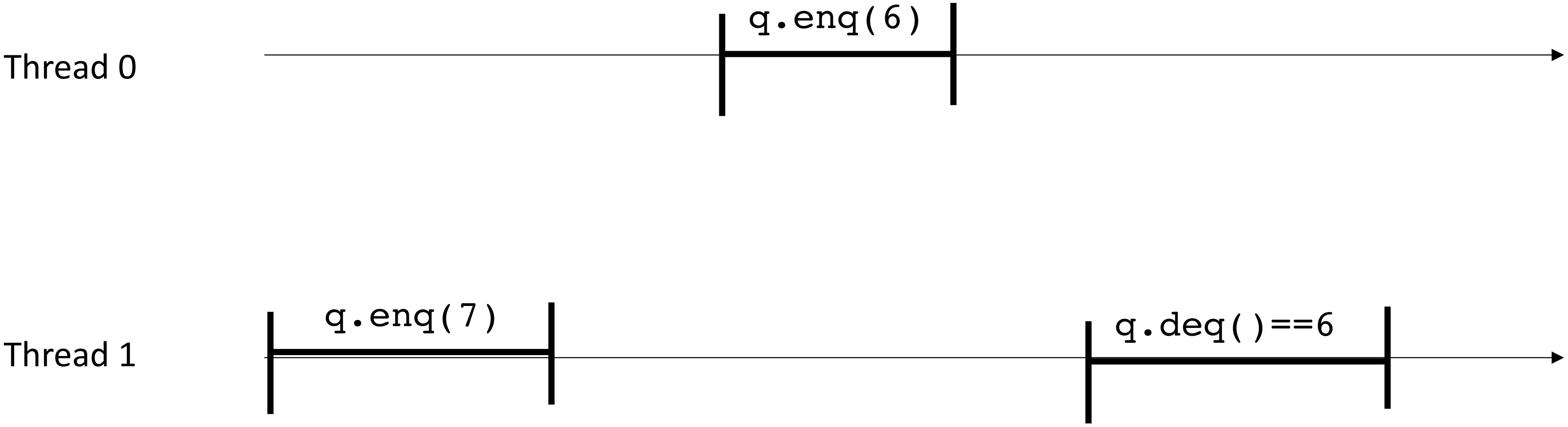


Sequential consistency and real time

- Add in real time:

Why might this actually happen?

asynchronous calls (like printf), e.g. it buffers the value before publishing it?
Lazy publishing (e.g. cache values in registers)?



Sequential consistency and real time

- Add in real time:

`q.enq(6)`

Thread 0

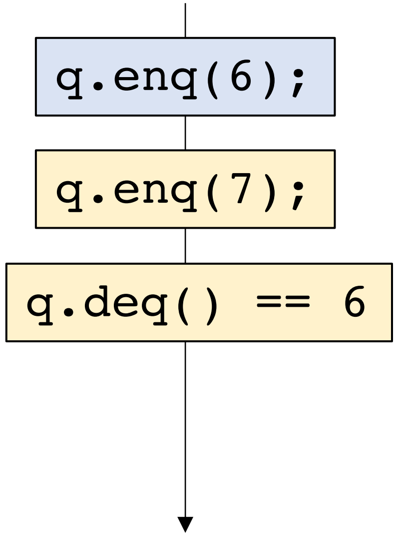
`q.enq(7);` `q.deq()==6`

Thread 1

real time line

This execution is allowed in sequential consistency!

SC doesn’t care about real time, only if it can construct its virtual sequential timeline



`q.enq(6);`

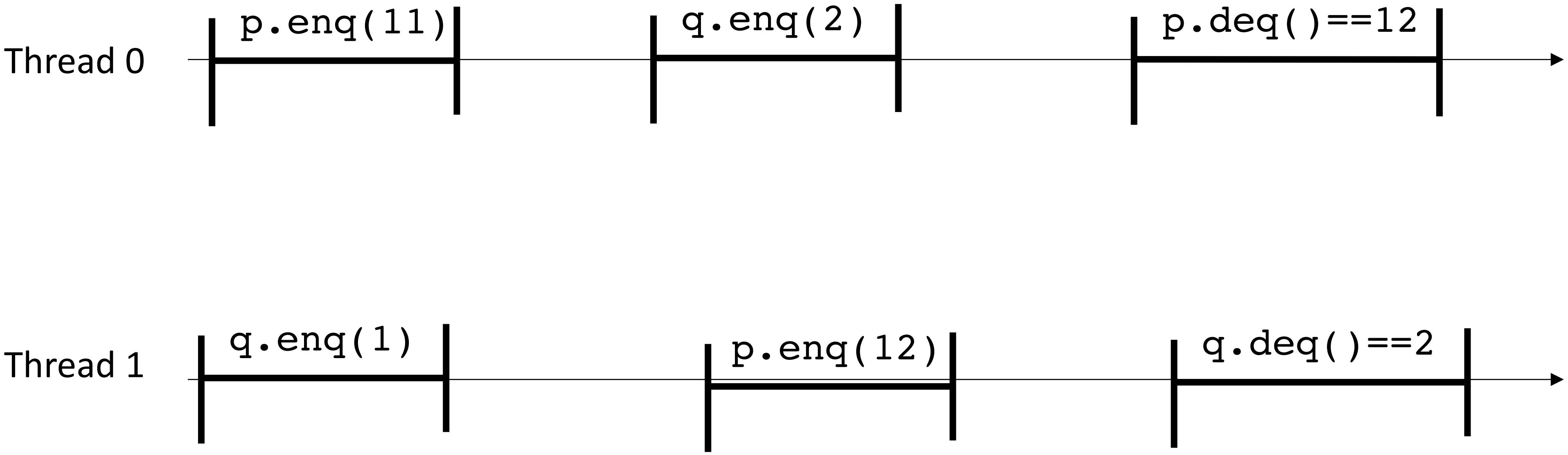
`q.enq(7);`

`q.deq() == 6`

Sequential consistency and real time

- Add in real time:

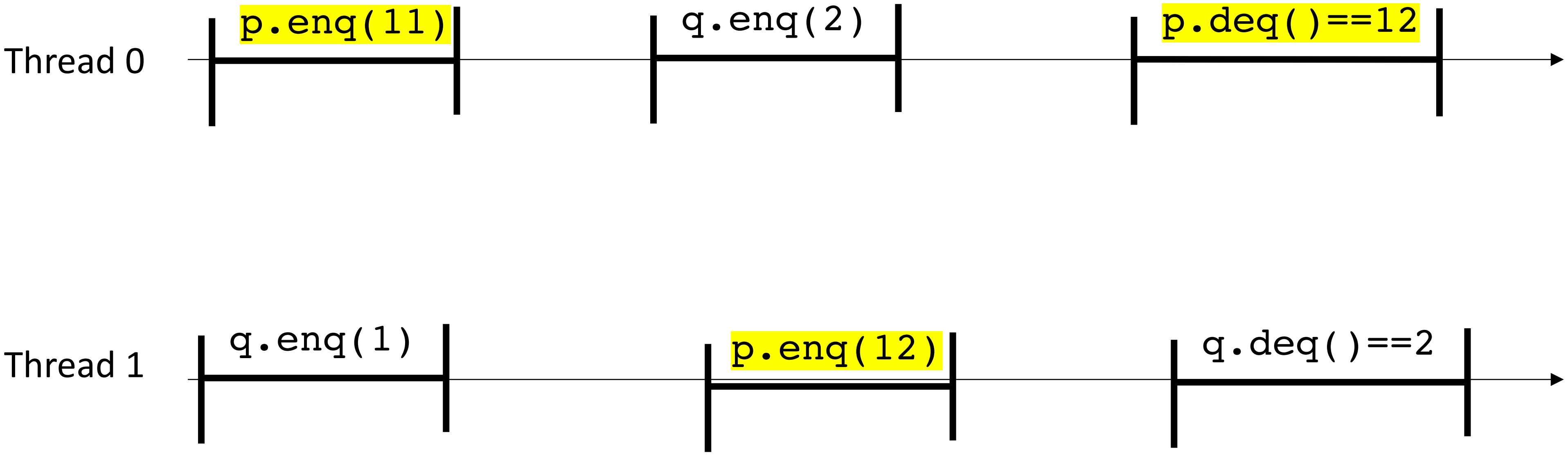
2 objects now: p and q



Sequential consistency and real time

- Add in real time:

2 objects now: p and q
Consider each object in isolation



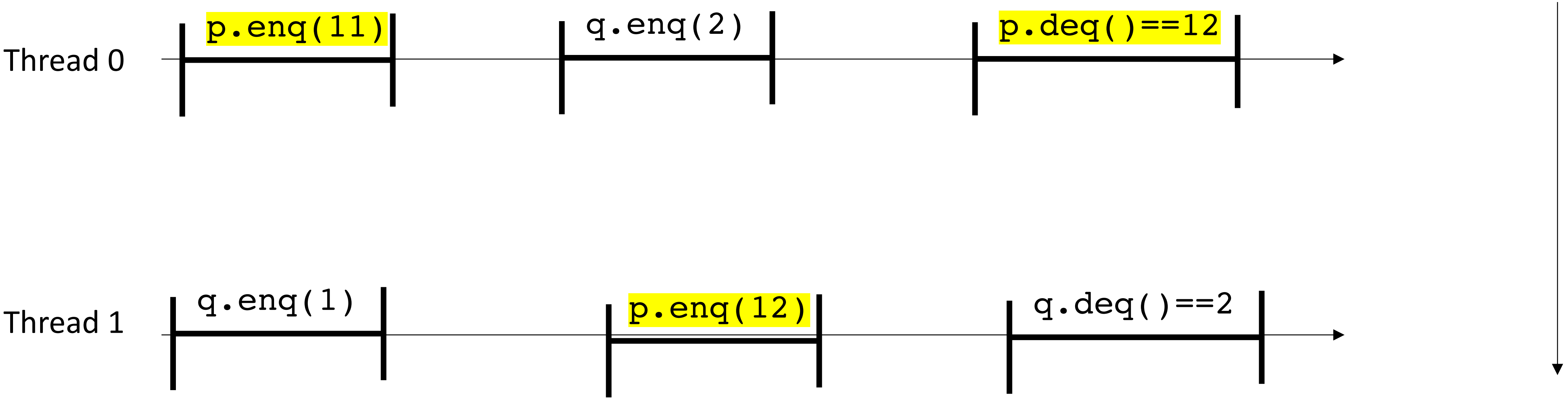
Thread 0
`p.enq(11)`
`q.enq(2)`
`p.deq()==12`

Thread 1
`q.enq(1)`
`p.enq(12)`
`q.deq()==2`

Sequential consistency and real time

- Add in real time:

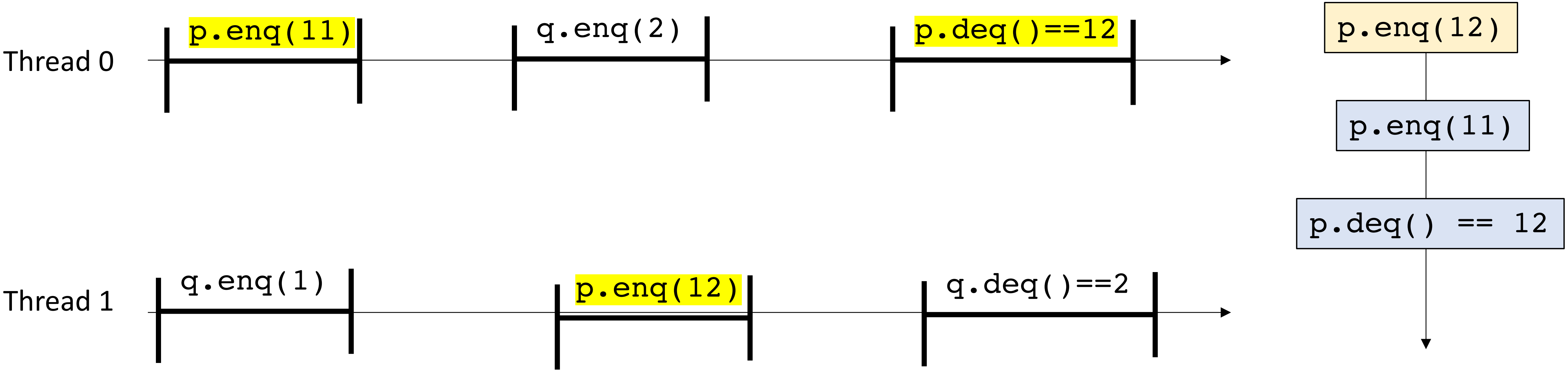
2 objects now: p and q
Consider each object in isolation



Sequential consistency and real time

- Add in real time:

2 objects now: p and q
Consider each object in isolation



Thread 0
`p.enq(11)`
`q.enq(2)`
`p.deq()==12`

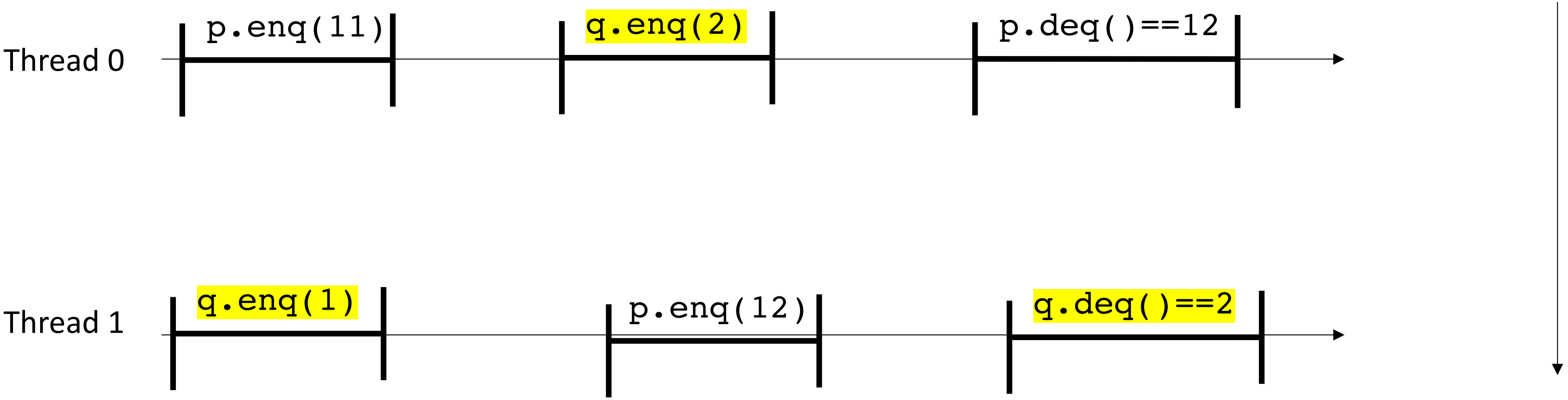
Thread 1
`q.enq(1)`
`p.enq(12)`
`q.deq()==2`

`p.enq(12)`
`p.enq(11)`
`p.deq() == 12`

Sequential consistency and real time

- Add in real time:

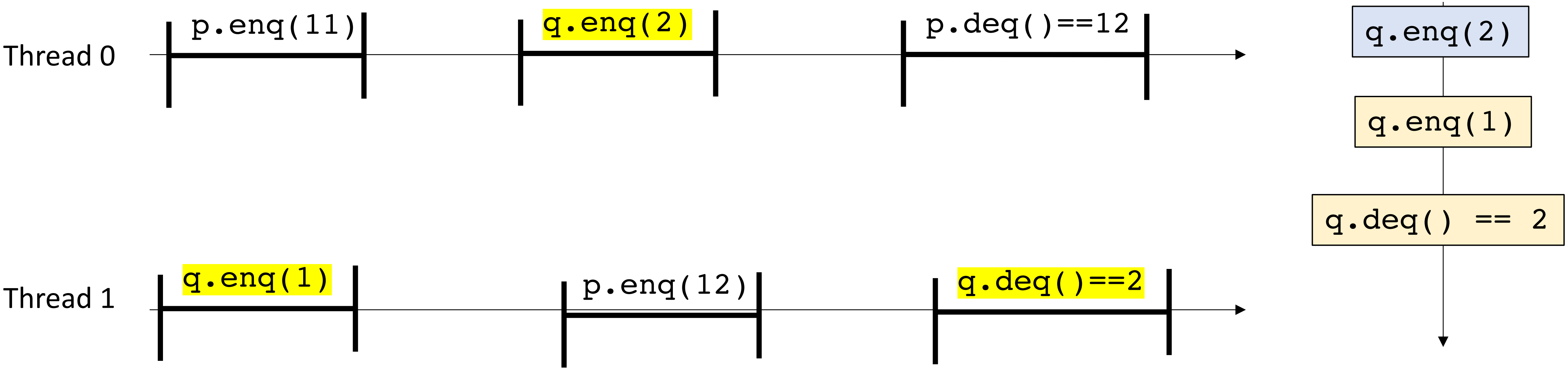
2 objects now: p and q
Consider each object in isolation



Sequential consistency and real time

- Add in real time:

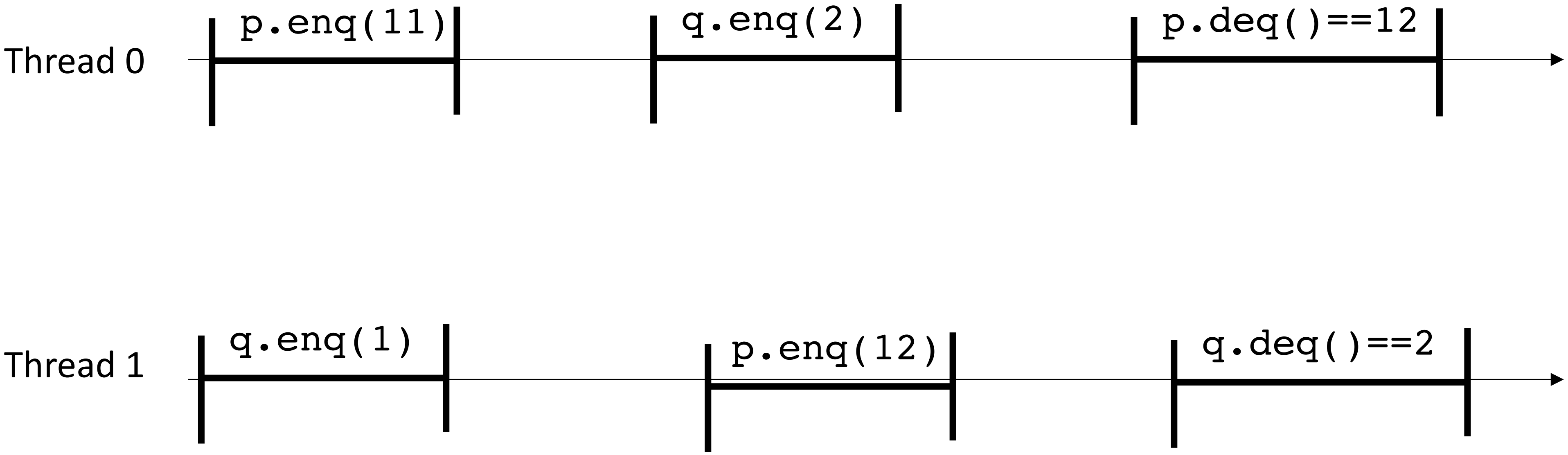
2 objects now: p and q
Consider each object in isolation



Sequential consistency and real time

- Add in real time:

Now consider them all together



Global variable:

```
CQueue<int> p,q;
```

Thread 0:

```
p.enq(11)
q.enq(2)
p.deq()==12
```

Thread 1:

```
q.enq(1)
p.enq(12)
q.deq()==2
```

Global variable:

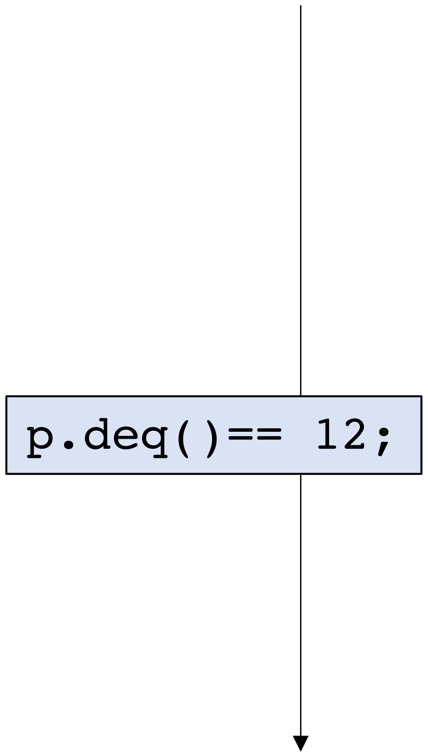
```
CQueue<int> p,q;
```

Thread 0:

```
p.enq(11)
q.enq(2)
p.deq()==12
```

Thread 1:

```
q.enq(1)
p.enq(12)
q.deq()==2
```



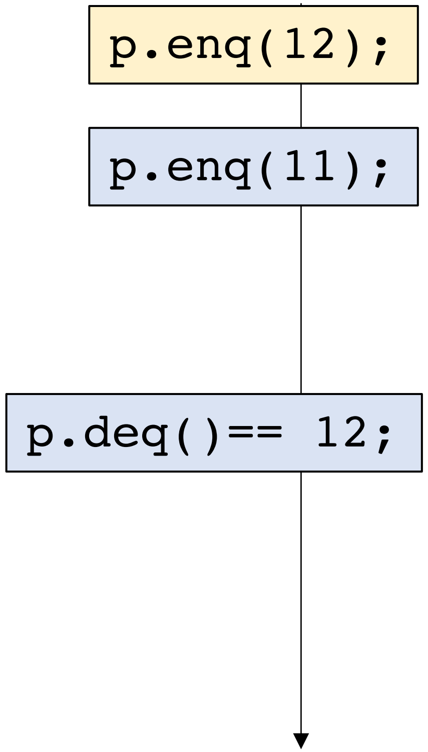
`p.deq()== 12;`

Global variable:

```
CQueue<int> p,q;
```

Thread 0:

```
p.enq(11)
q.enq(2)
p.deq()==12
```



```
p.enq(12);
p.enq(11);
p.deq()== 12;
```

Thread 1:

```
q.enq(1)
p.enq(12)
q.deq()==2
```

Global variable:

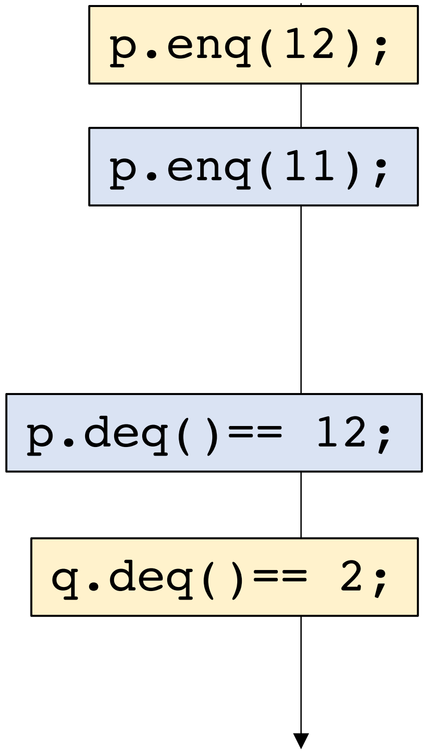
```
CQueue<int> p,q;
```

Thread 0:

```
p.enq(11)
q.enq(2)
p.deq()==12
```

Thread 1:

```
q.enq(1)
p.enq(12)
q.deq()==2
```



```
p.enq(12);
p.enq(11);
p.deq()== 12;
q.deq()== 2;
```

Global variable:

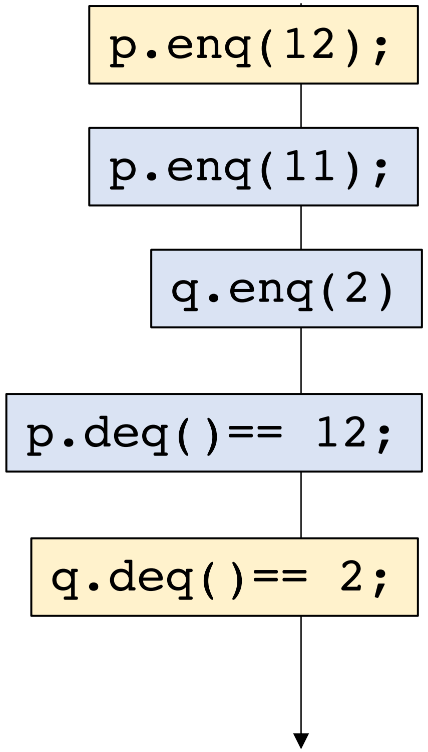
```
CQueue<int> p,q;
```

Thread 0:

```
p.enq(11)
q.enq(2)
p.deq()==12
```

Thread 1:

```
q.enq(1)
p.enq(12)
q.deq()==2
```



```
p.enq(12);
p.enq(11);
q.enq(2)
p.deq()== 12;
q.deq()== 2;
```

Global variable:

```
CQueue<int> p,q;
```

Thread 0:

```
p.enq(11)
q.enq(2)
p.deq()==12
```

```
p.enq(12);
p.enq(11);
q.enq(2)
p.deq()== 12;
q.deq()== 2;
```

```
q.enq(1);
```

where to put this?

Thread 1:

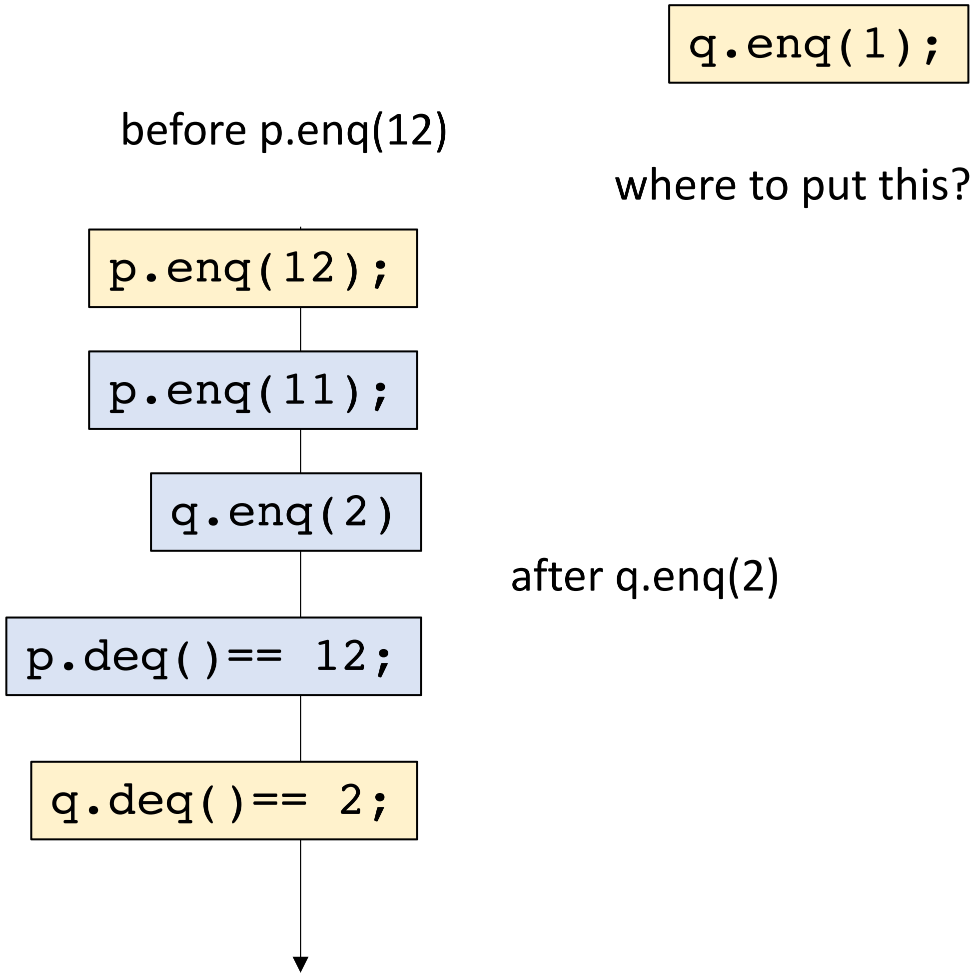
```
q.enq(1)
p.enq(12)
q.deq()==2
```

Global variable:

```
CQueue<int> p,q;
```

Thread 0:

```
p.enq(11)
q.enq(2)
p.deq()==12
```



```
q.enq(1);
```

before p.enq(12)

where to put this?

```
p.enq(12);
p.enq(11);
q.enq(2)
```

after q.enq(2)

```
p.deq()== 12;
q.deq()== 2;
```

Thread 1:

```
q.enq(1)
p.enq(12)
q.deq()==2
```

What does this mean?

- Even if objects in isolation are sequentially consistent
- Programs composed of multiple objects might not be!
- We would like to be able to use more than 1 object in our programs!

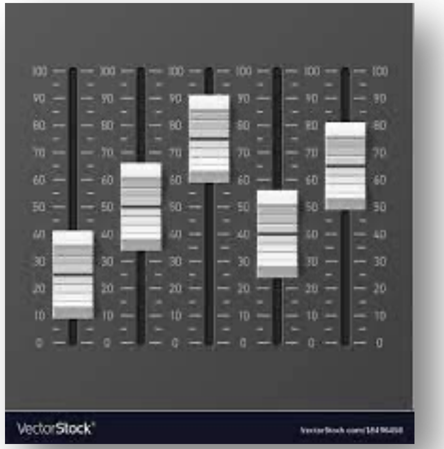
Lecture schedule

- Revisiting sequential consistency
- **Linearizablity**
- Progress Properties
- Implementing a set

Linearizability

- Linearizability
  - Defined in term of real-time histories
  - We want to ask if an execution is allowed under linearizability

- Slightly different game:
  - sequential consistency is a game about stacking lego bricks
  - linearizability is about sliders



Linearizability

each operation has a linearizability point

- does not overlap with other with other linearizability points

- indivisible computation (critical section, atomic RMW, atomic load, atomic store)

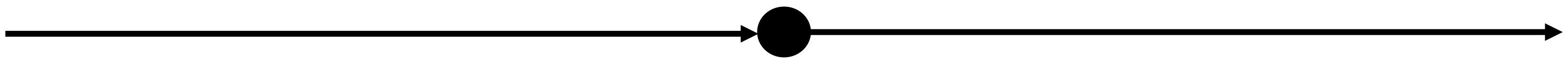
- object update (or read) occurs exactly at this point

●

Linearizability

each operation has a linearizability point

- does not overlap with other with other linearizability points
- indivisible computation (critical section, atomic RMW, atomic load, atomic store)
- object update (or read) occurs exactly at this point



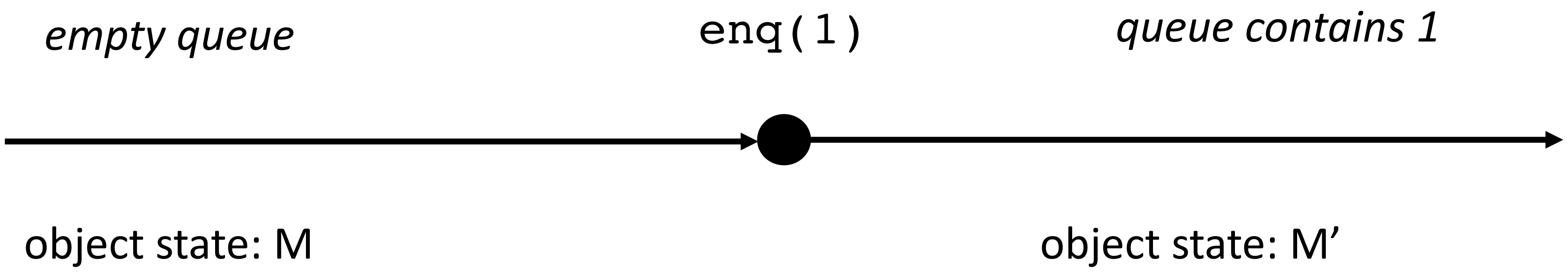
object state: M

object state: M’

Linearizability

each operation has a linearizability point

- does not overlap with other with other linearizability points
- indivisible computation (critical section, atomic RMW, atomic load, atomic store)
- object update (or read) occurs exactly at this point



empty queue

`enq(1)`

queue contains 1

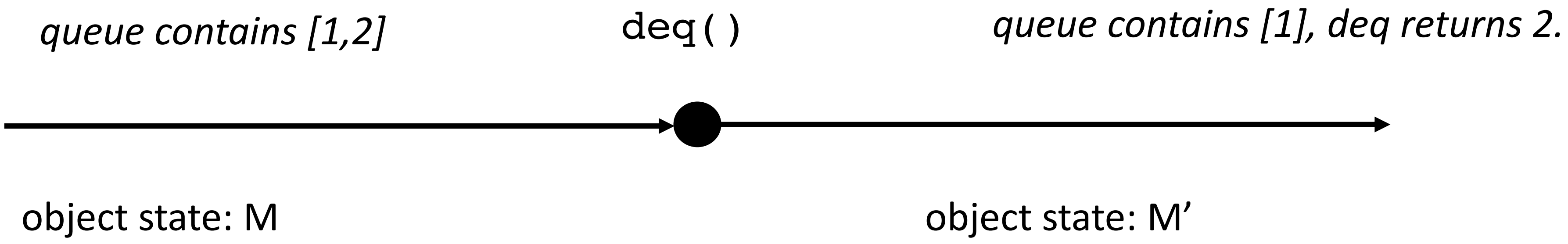
object state: M

object state: M’

Linearizability

each operation has a linearizability point

- does not overlap with other with other linearizability points
- indivisible computation (critical section, atomic RMW, atomic load, atomic store)
- object update (or read) occurs exactly at this point



queue contains [1,2]

`deq()`

queue contains [1], deq returns 2.

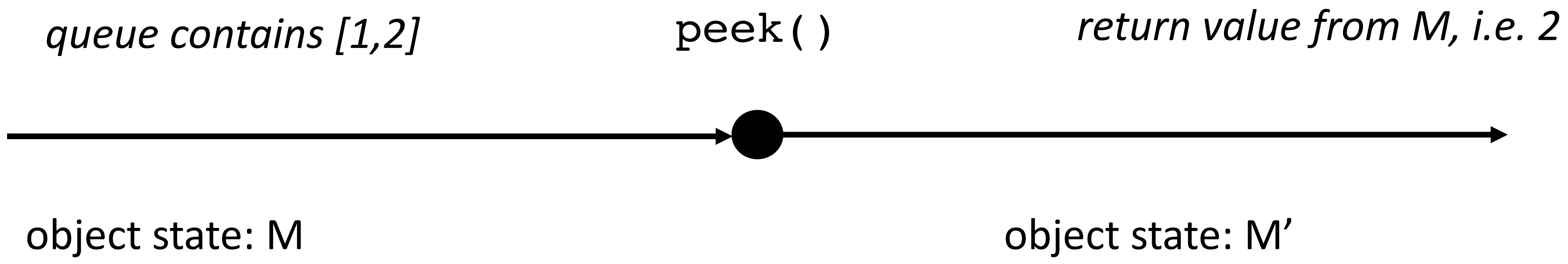
object state: M

object state: M’

Linearizability

each operation has a linearizability point

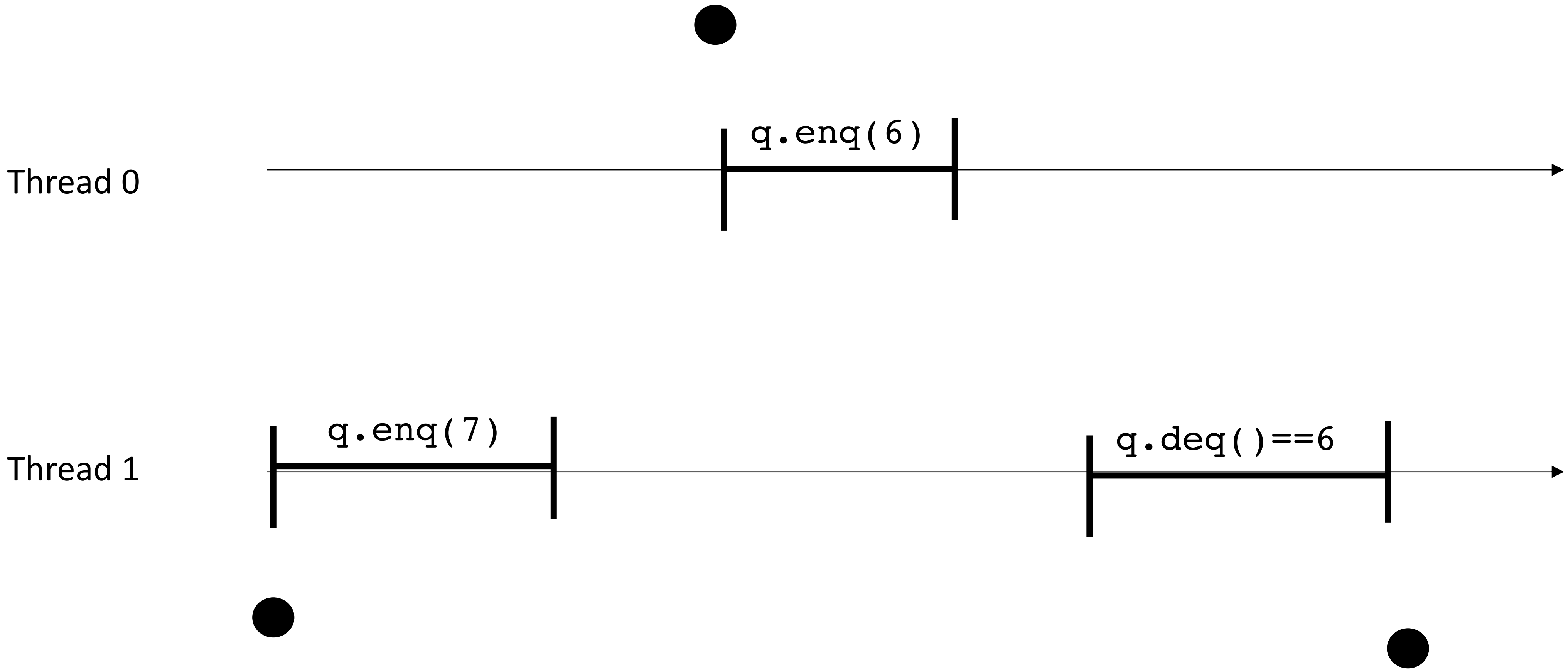
- does not overlap with other with other linearizability points
- indivisible computation (critical section, atomic RMW, atomic load, atomic store)
- object update (or read) occurs exactly at this point



Linearizability

each command gets a linearization point.

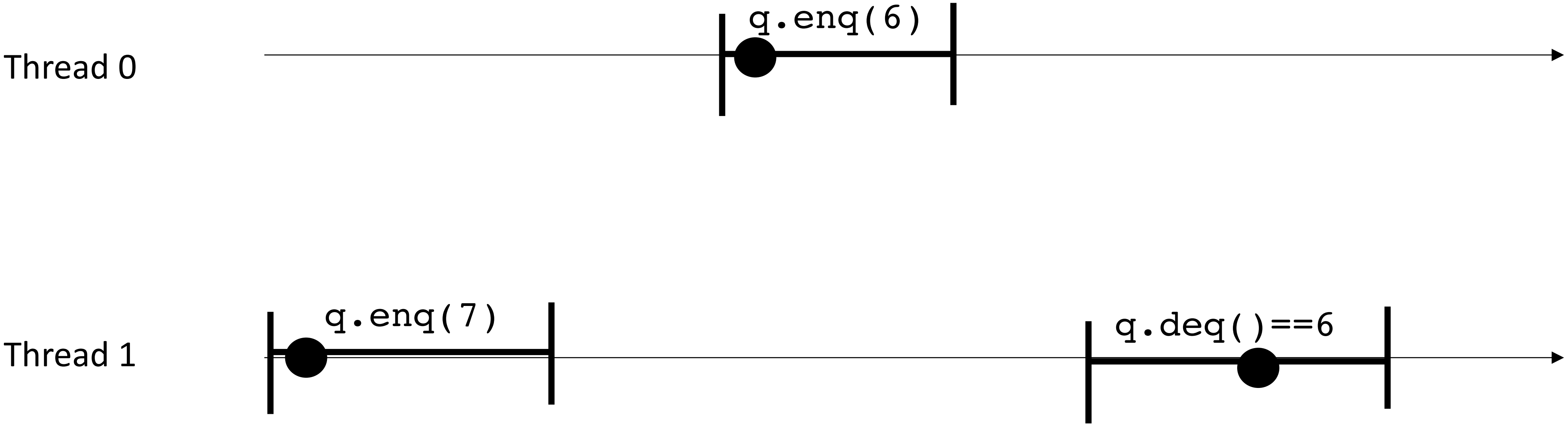
You can place the point any where between its innovation and response!



Linearizability

each command gets a linearization point.

You can place the point any where between its innovation and response!

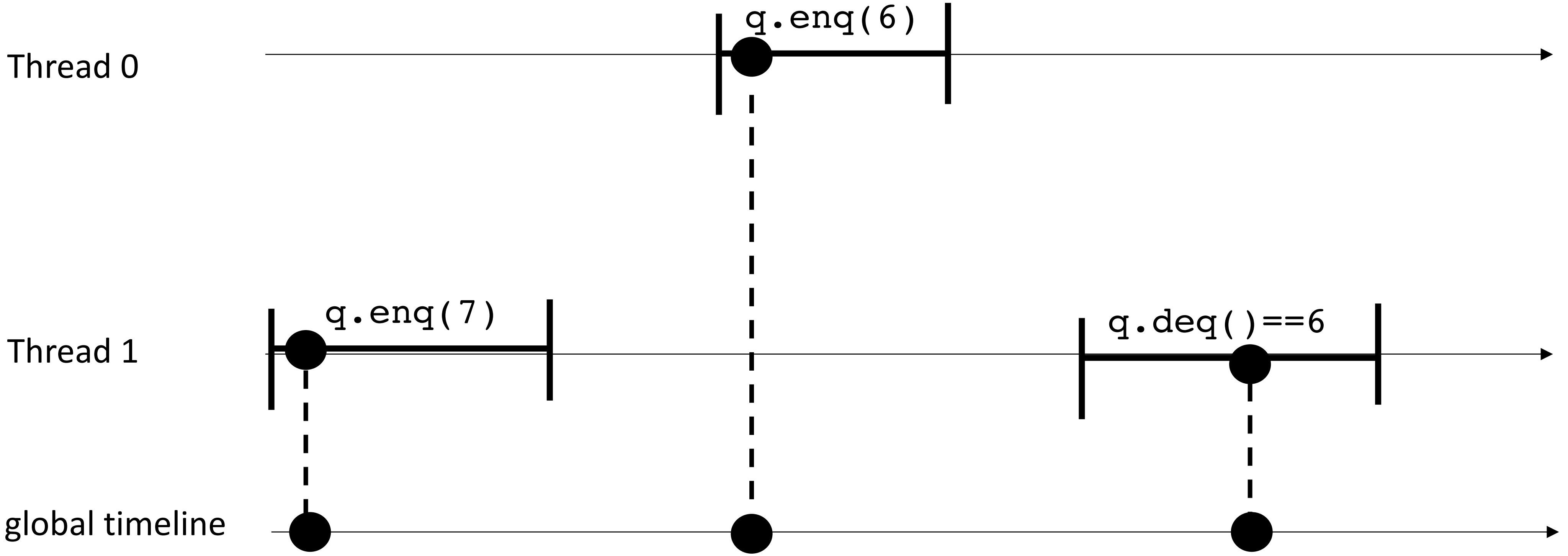


Linearizability

each command gets a linearization point.

You can place the point any where between its innovation and response!

Project the linearization points to a global timeline



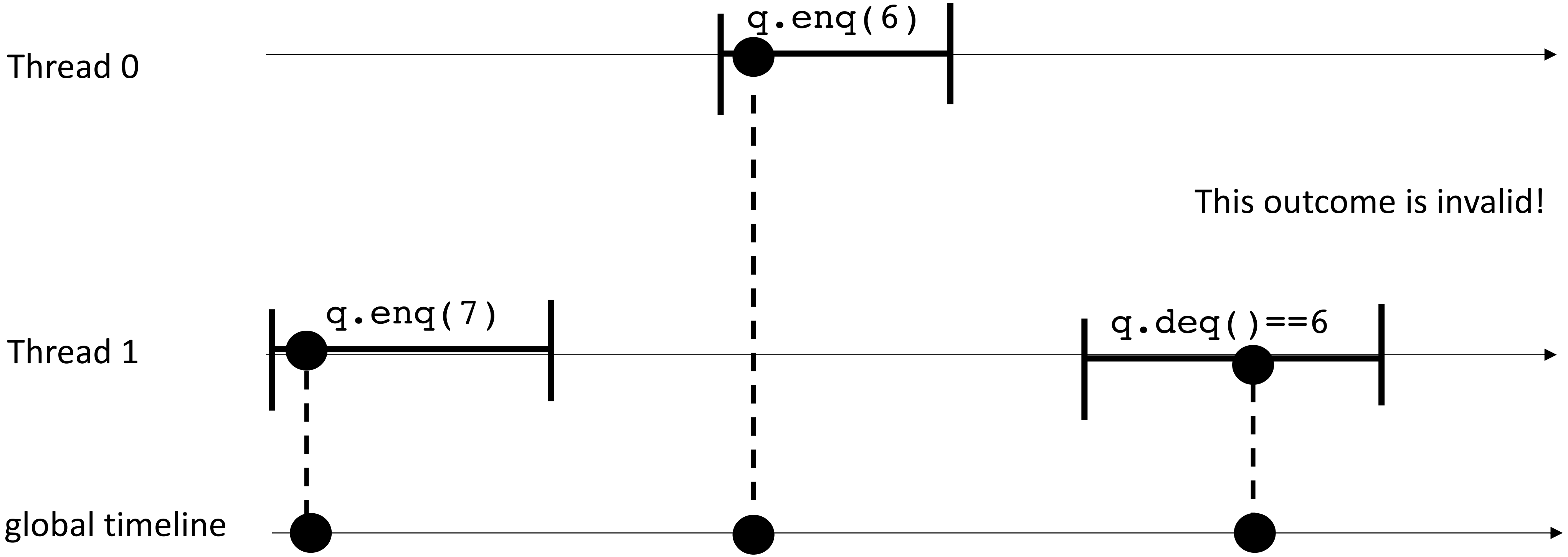
reason sequentially!

Linearizability

each command gets a linearization point.

You can place the point any where between its innovation and response!

Project the linearization points to a global timeline



This outcome is invalid!

reason sequentially!

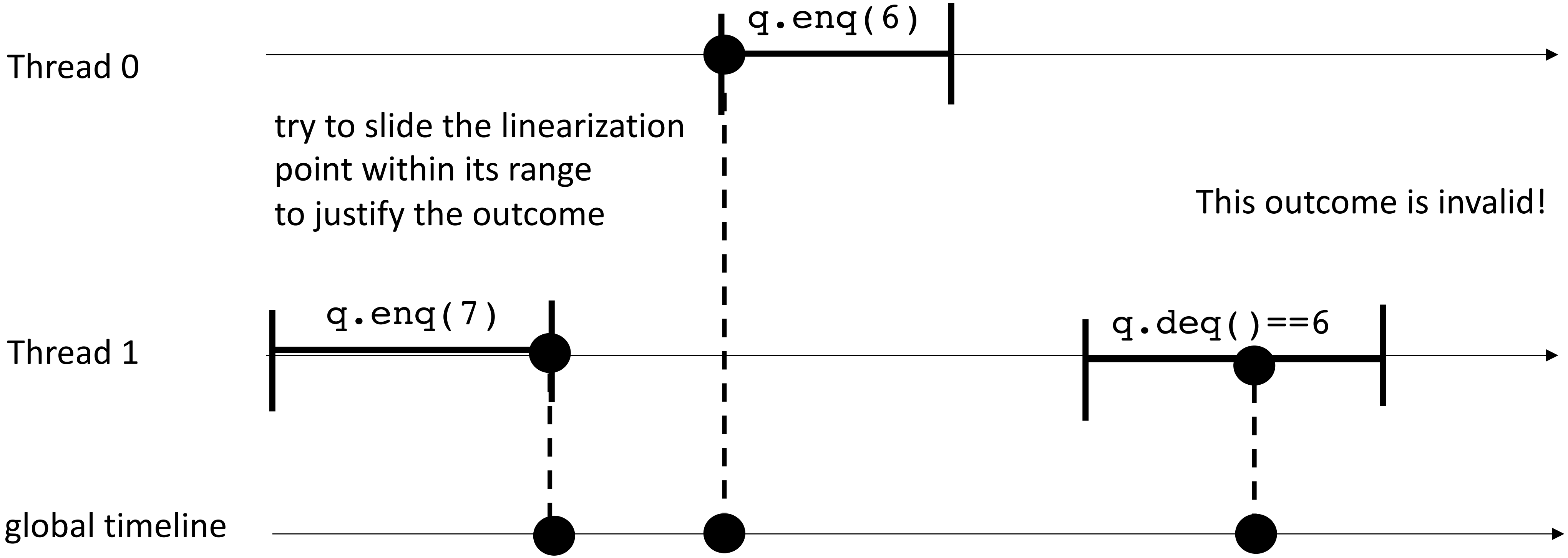
Linearizability

each command gets a linearization point.

You can place the point any where between its innovation and response!

Project the linearization points to a global timeline

slider game!

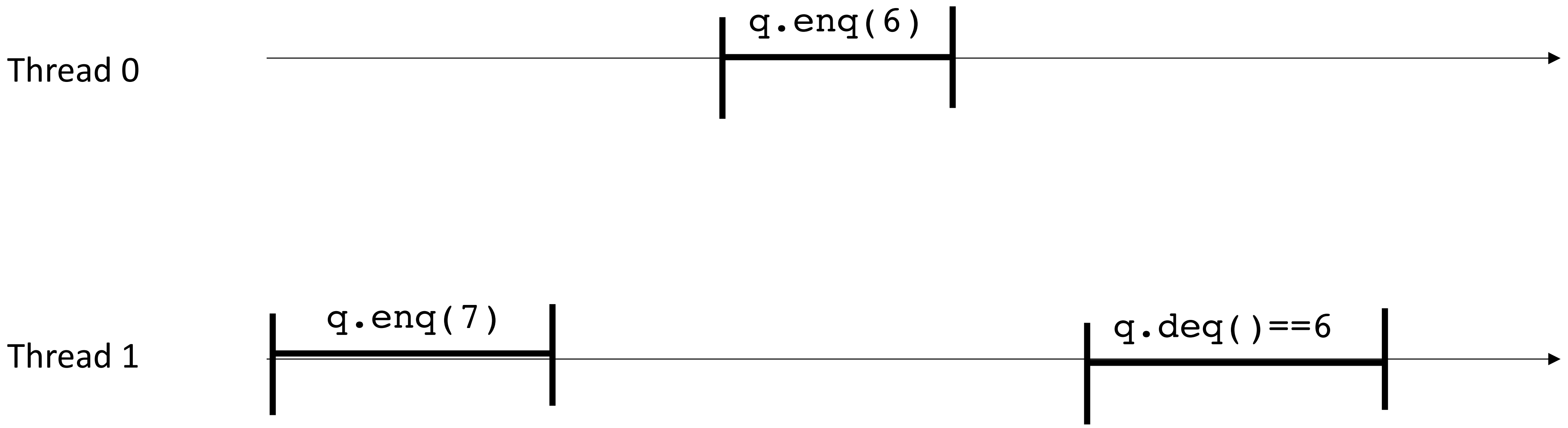


try to slide the linearization point within its range to justify the outcome

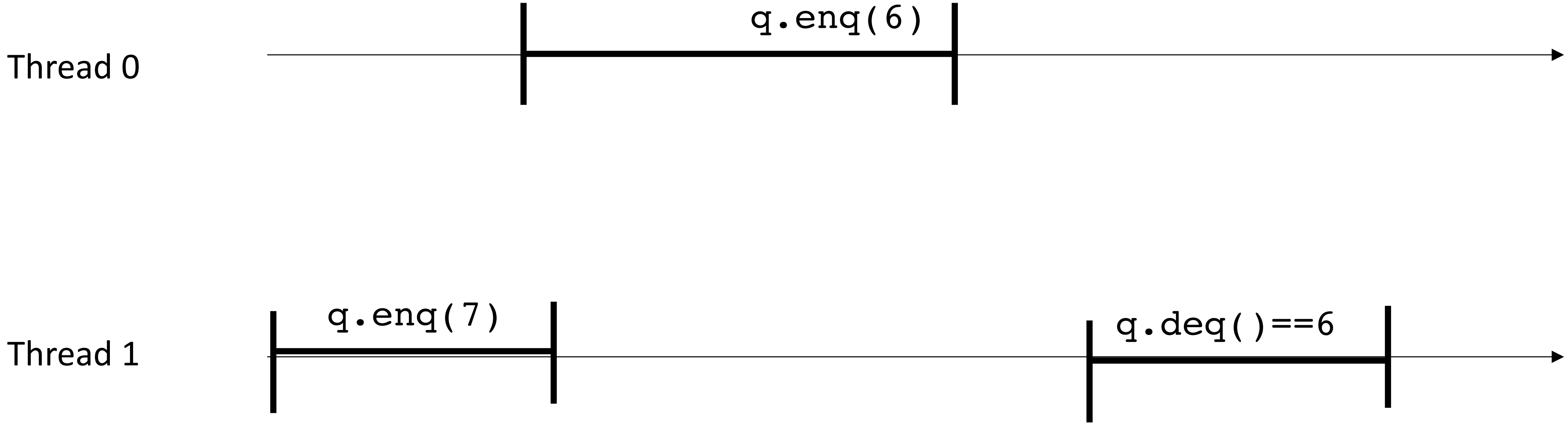
This outcome is invalid!

reason sequentially!

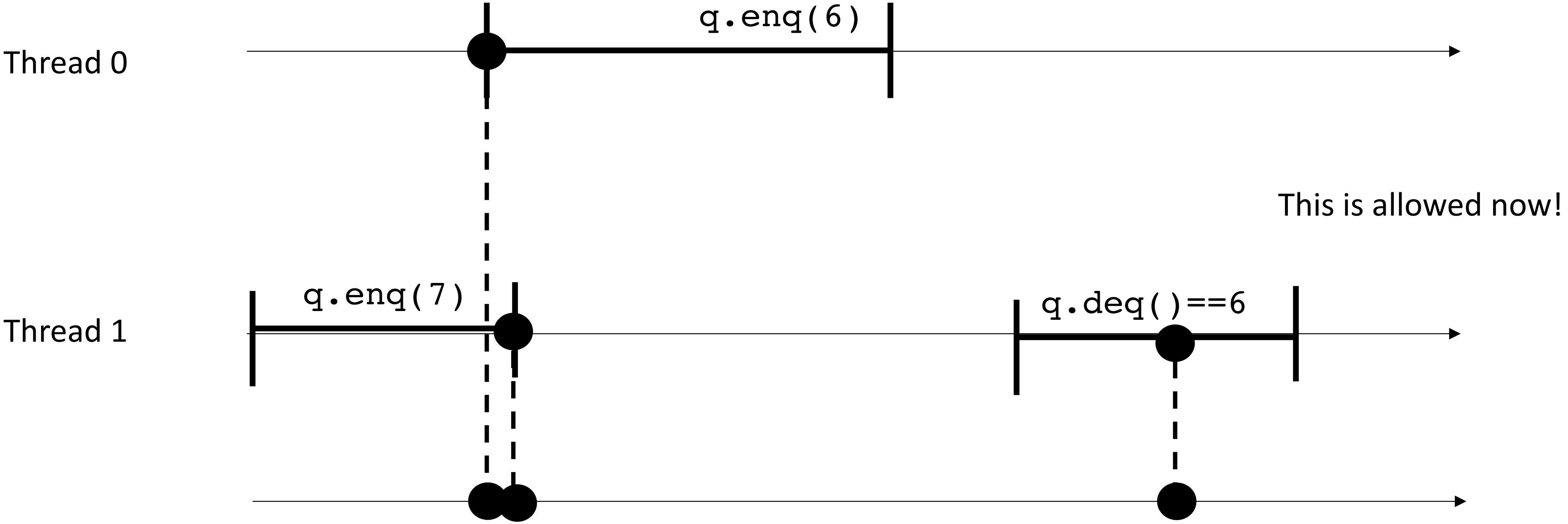
Linearizability



Linearizability

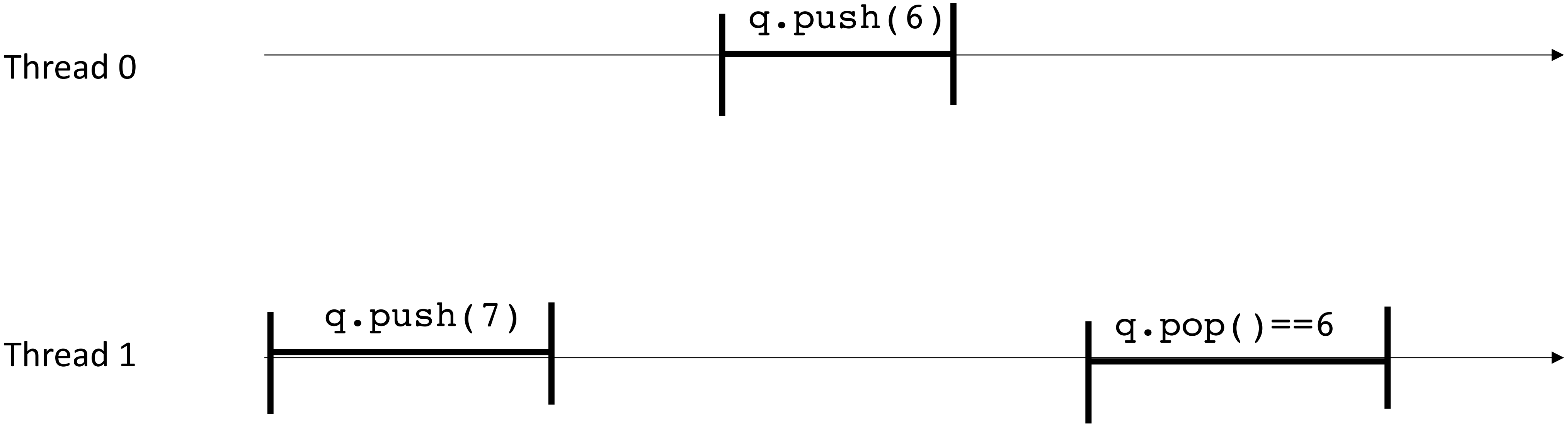


Linearizability

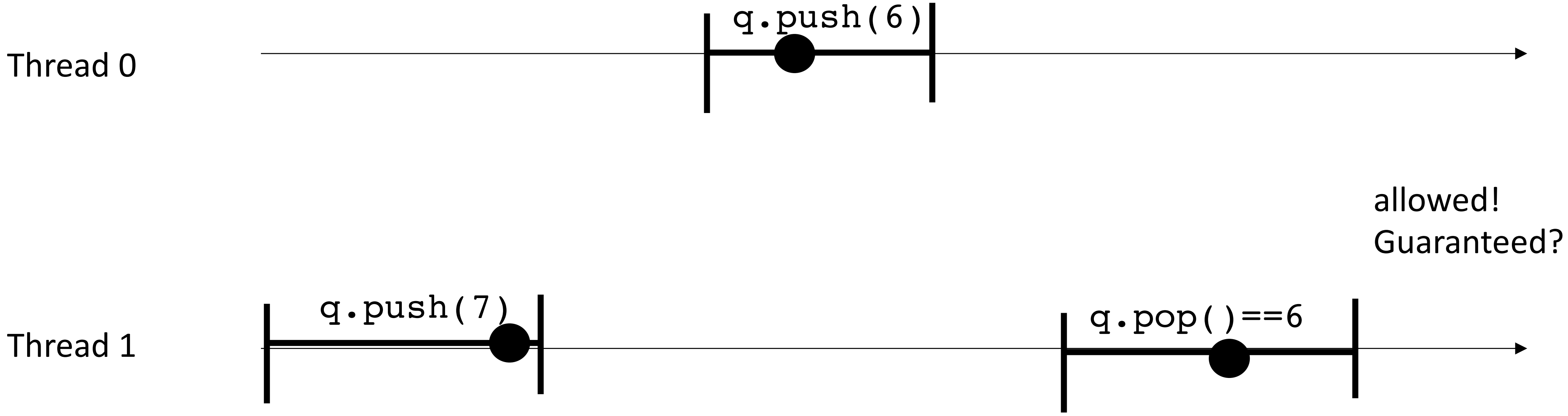


This is allowed now!

Linearizability

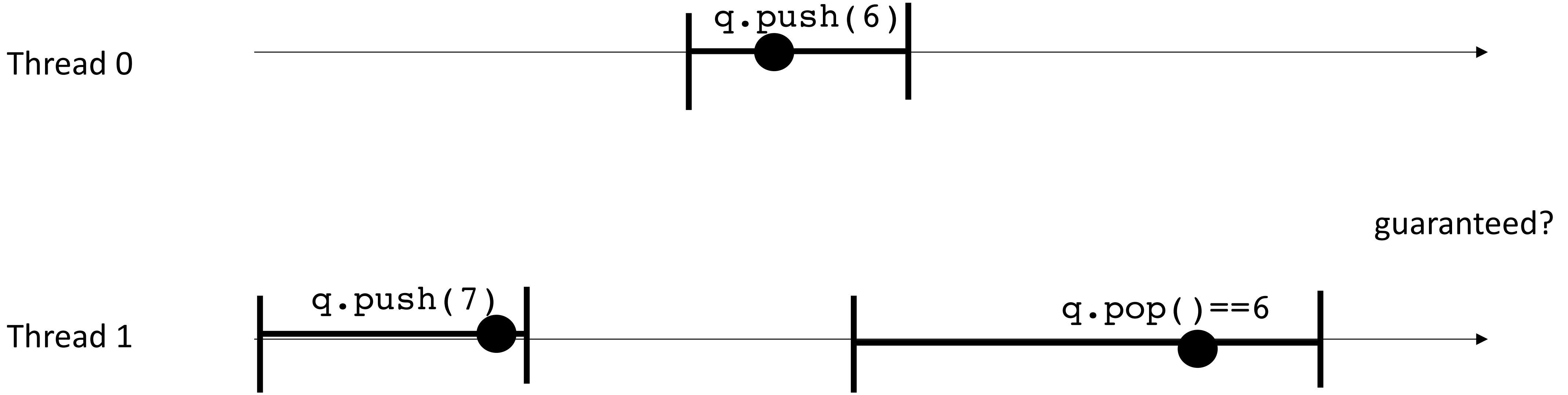


Linearizability

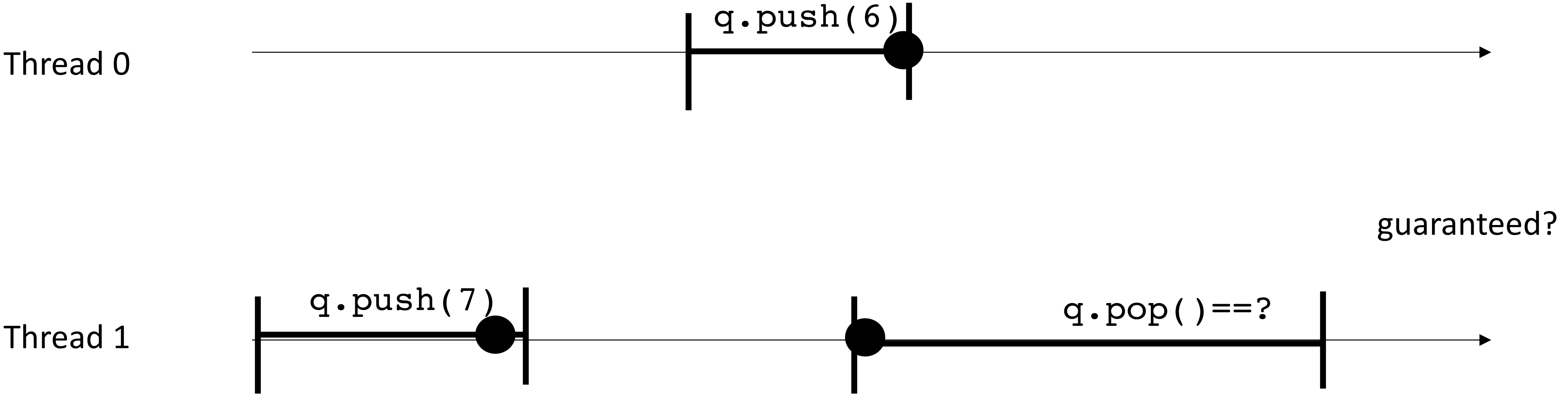


allowed!
Guaranteed?

Linearizability



Linearizability



Linearizability

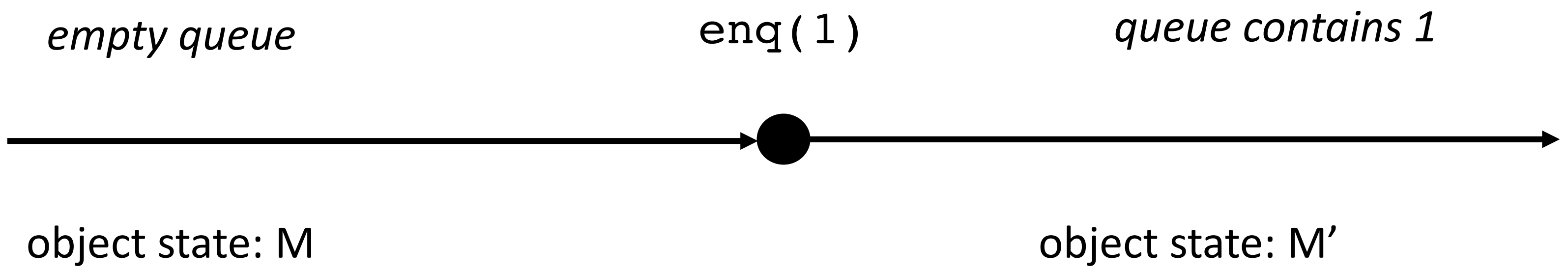
- We spent a bunch of time on SC... did we waste our time?
  - No!
  - Linearizability is strictly stronger than SC. Every linearizable execution is SC, but not the other way around.

  - If a behavior is disallowed under SC, it is also disallowed under linearizability.

- Overall strategy:
  - Write our objects to be linearizable: need to identify linearizable points
  - Reason about our programs using SC: no need for timelines, just need code

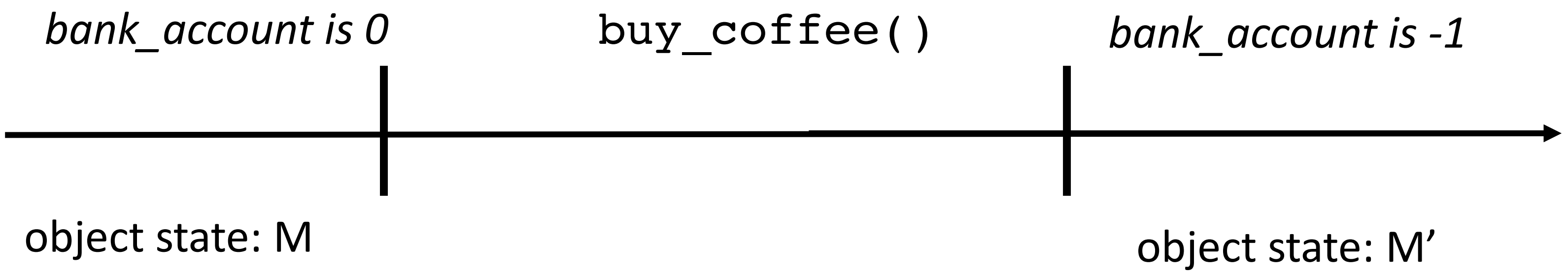
Linearizability

- How do we write our programs to be linearizable?
  - Identify the linearizability point
  - One indivisible region (e.g. an atomic store, atomic load, atomic RMW, or critical section) where the method call takes effect. Modeled as a point.



Linearizability

- Locked data structures are linearizable.



bank_account is 0

`buy_coffee()`

bank_account is -1

object state: M

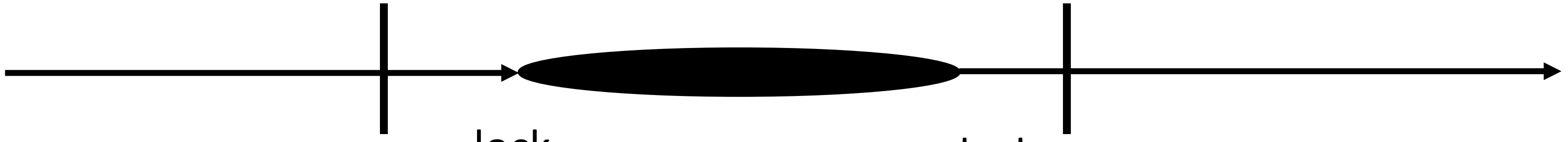
object state: M’

```
class bank_account {
  public:
    bank_account() {
      balance = 0;
    }

    void buy_coffee() {
      m.lock();
      balance -= 1;
      m.unlock();
    }

    void get_paid() {
      m.lock();
      balance += 1;
      m.unlock();
    }

  private:
    int balance;
    mutex m;
};
```

Linearizability

- Locked data structures are linearizable.

typically modeled as the point the lock is acquired or released



bank_account is 0

`buy_coffee()`

bank_account is -1

object state: M

lock

unlock

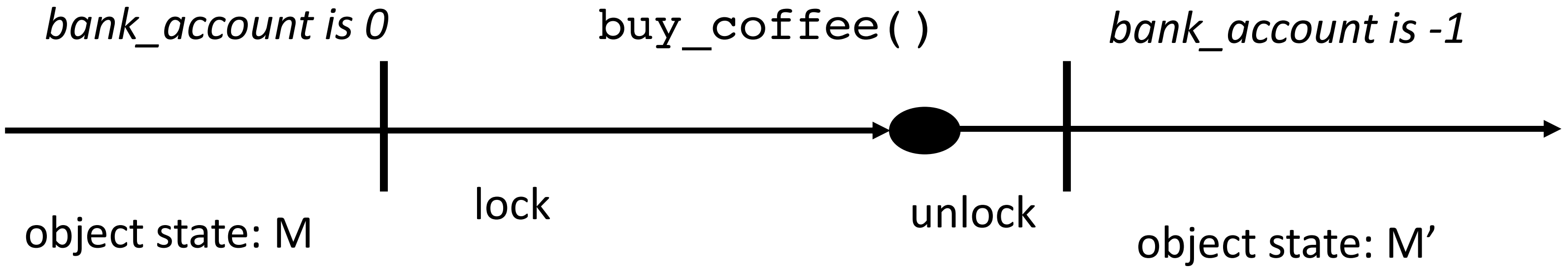
object state: M’

```
class bank_account {
  public:
    bank_account() {
      balance = 0;
    }

    void buy_coffee() {
      m.lock();
      balance -= 1;
      m.unlock();
    }

    void get_paid() {
      m.lock();
      balance += 1;
      m.unlock();
    }

  private:
    int balance;
    mutex m;
};
```

Linearizability

- Locked data structures are linearizable.

typically modeled as the point the lock is acquired or released
lets say released.

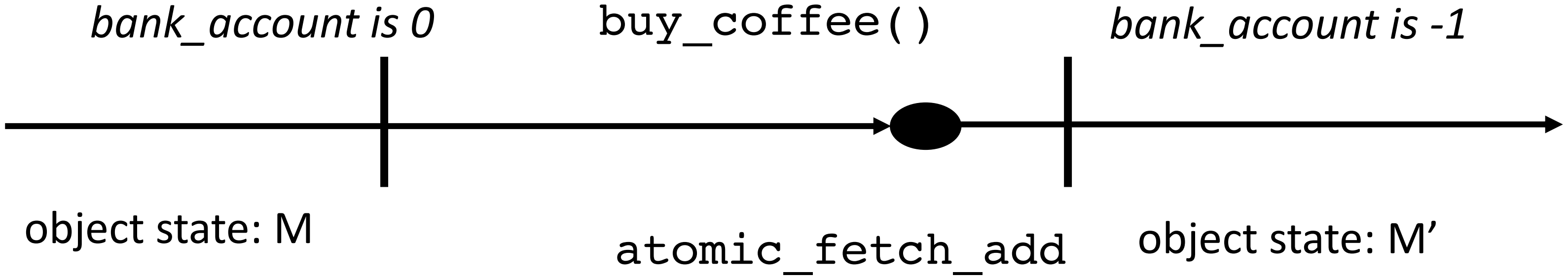


```
class bank_account {
  public:
    bank_account() {
      balance = 0;
    }

    void buy_coffee() {
      m.lock();
      balance -= 1;
      m.unlock();
    }

    void get_paid() {
      m.lock();
      balance += 1;
      m.unlock();
    }

  private:
    int balance;
    mutex m;
};
```

Linearizability

- Our lock-free bank account is linearizable:
  - The atomic operation is the linearizable point

```
class bank_account {
  public:
    bank_account() {
      balance = 0;
    }

    void buy_coffee() {
      atomic_fetch_add(&balance, -1);
    }

    void get_paid() {
      atomic_fetch_add(&balance, 1);
    }

  private:
    atomic_int balance;
};
```



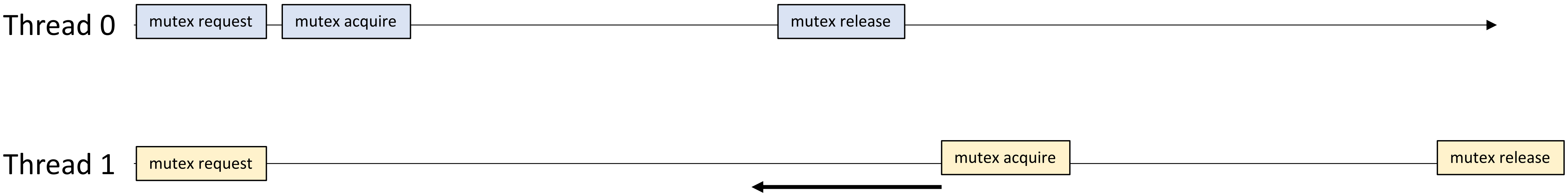
Lecture schedule

- Revisiting sequential consistency
- Linearizablity
- **Progress Properties**
- Implementing a set

Progress properties

- Going back to specifications:

Recall the mutex



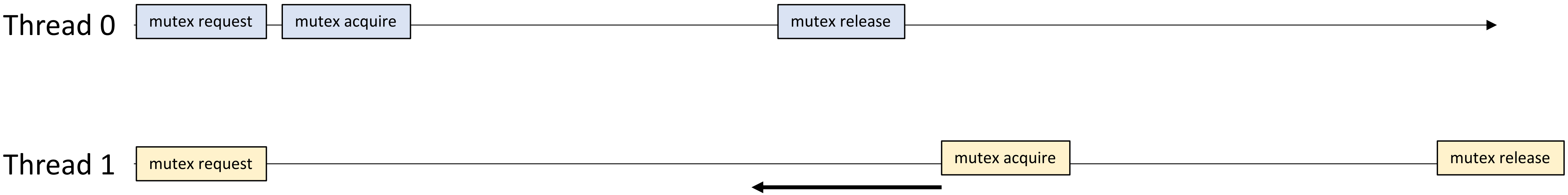
what is stopping this?

Progress properties

- Going back to specifications:

Thread 0 is stopping Thread 1 from making progress.
If delays in one thread can cause delays in other threads, we say that it is blocking

Recall the mutex



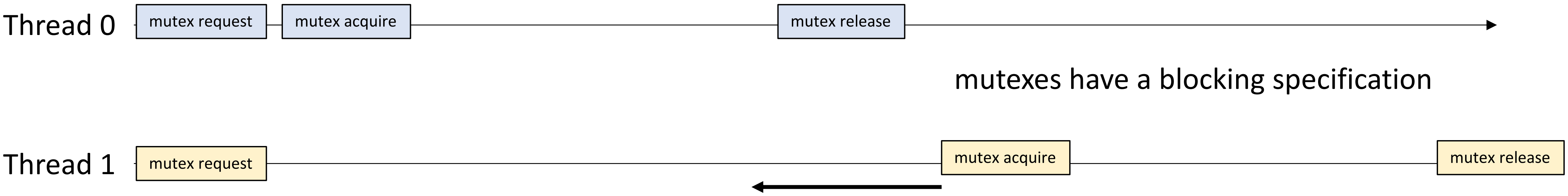
what is stopping this?

Progress properties

- Going back to specifications:

Recall the mutex

Thread 0 is stopping Thread 1 from making progress.
If delays in one thread can cause delays in other threads, we say that it is blocking



Thread 0 mutex request mutex acquire mutex release

mutexes have a blocking specification

Thread 1 mutex request mutex acquire mutex release

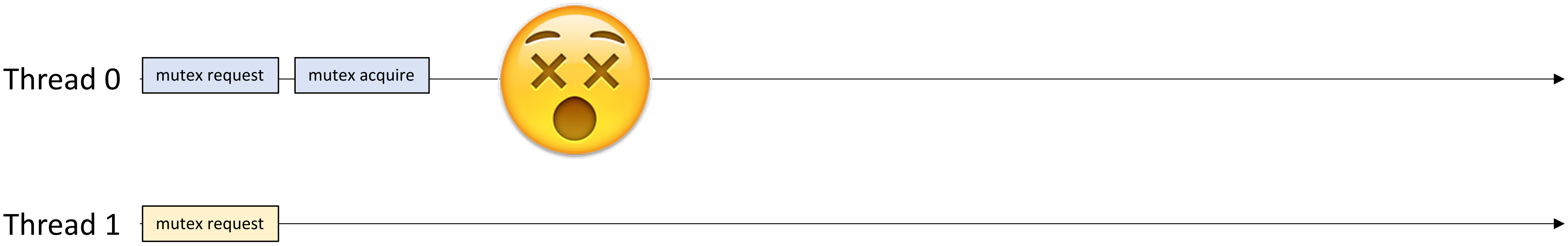
what is stopping this?

Progress properties

- Going back to specifications:

Thread 0 is stopping Thread 1 from making progress.
If delays in one thread can cause delays in other threads, we say that it is blocking

Recall the mutex

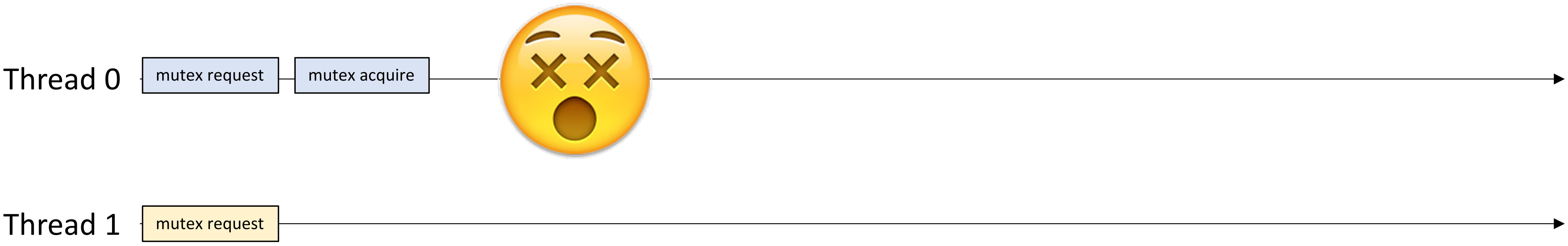


What now?!

Progress properties

- Going back to specifications:

Recall the mutex



Thread 0

mutex request

mutex acquire

Thread 1

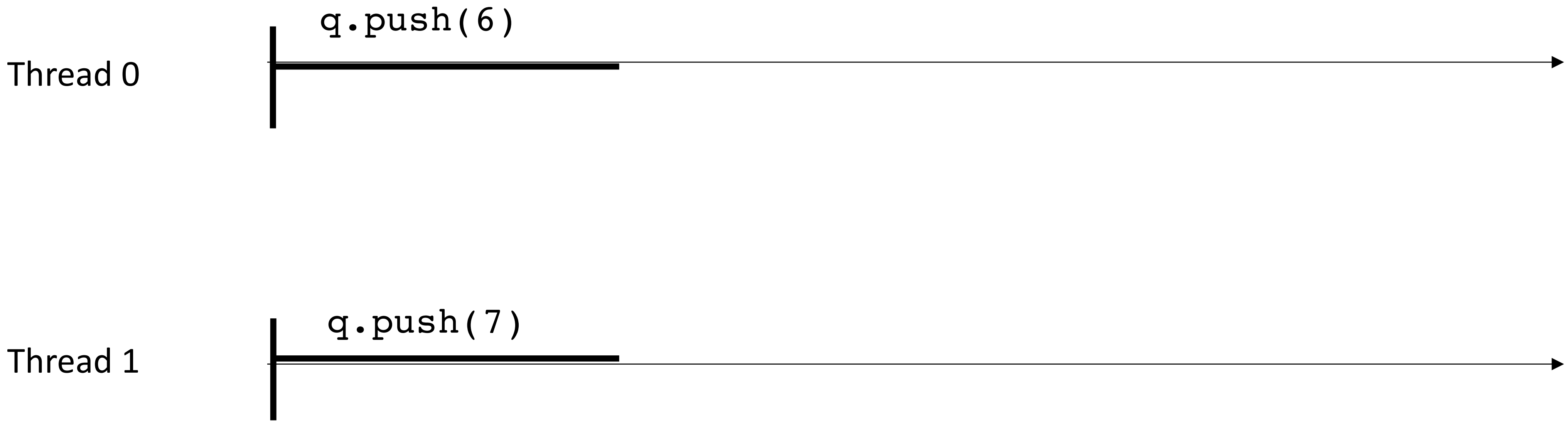
mutex request

Thread 0 is stopping Thread 1 from making progress.
If delays in one thread can cause delays in other threads, we say that it is blocking

What now?!

Linearizability

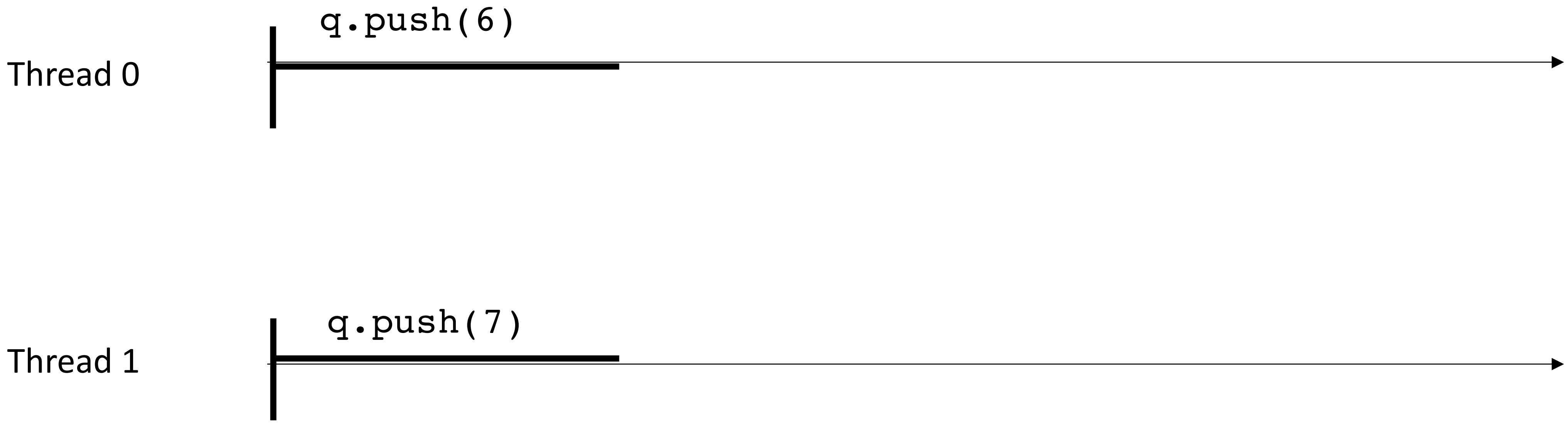
Two unfinished commands.



Linearizability

Two unfinished commands.

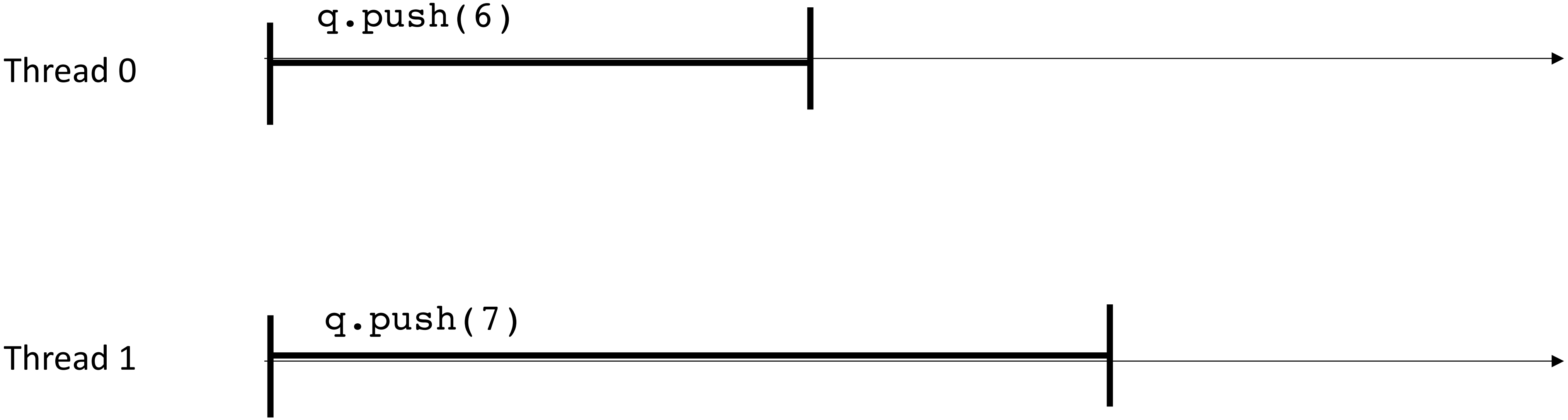
Linearizability does not dictate that one needs to wait for another



Linearizability

Two unfinished commands.

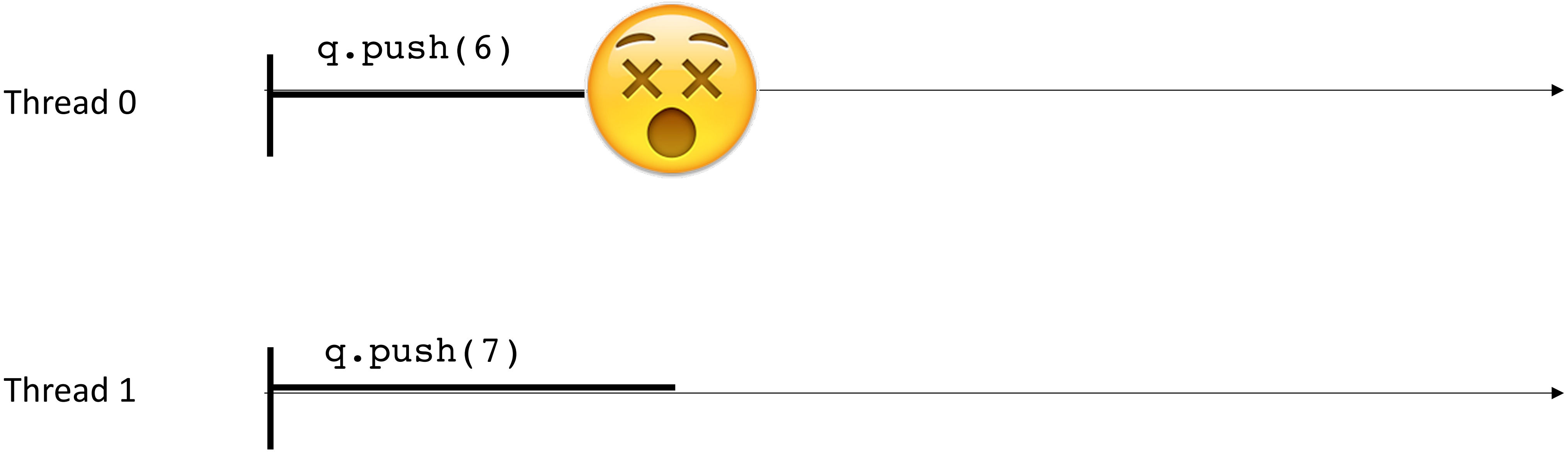
Linearizability does not dictate that one needs to wait for another



Linearizability

Two unfinished commands.

Linearizability does not dictate that one needs to wait for another



Thread 0

`q.push(6)`

Thread 1

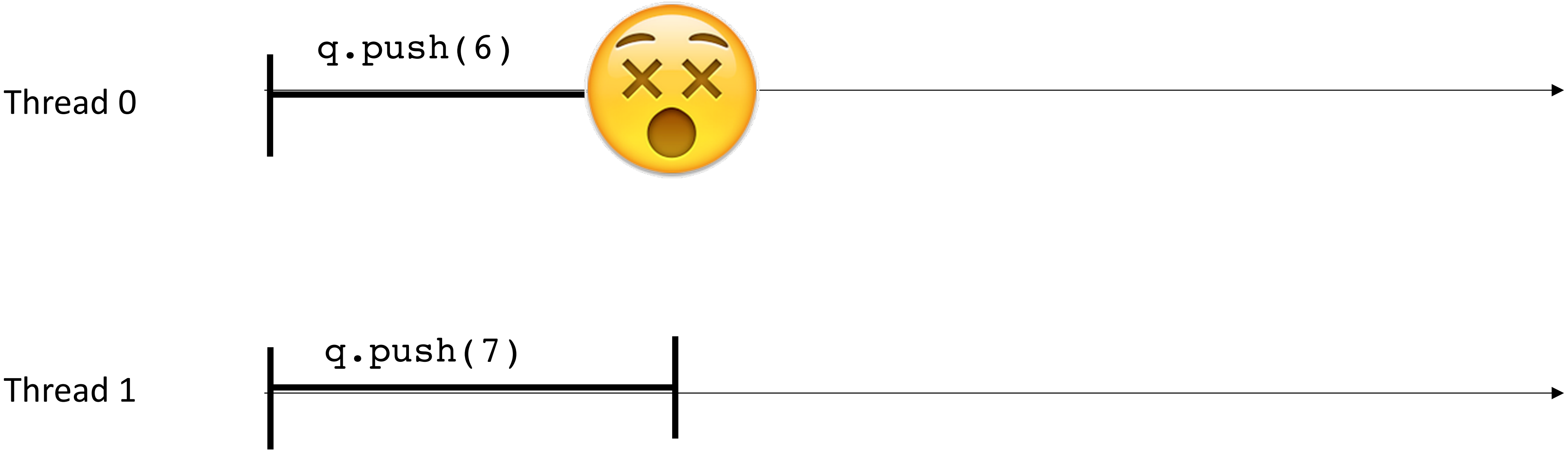
`q.push(7)`

for mutexes, the specification required that the system hang.

Linearizability

Two unfinished commands.

Linearizability does not dictate that one needs to wait for another



Thread 0

`q.push(6)`

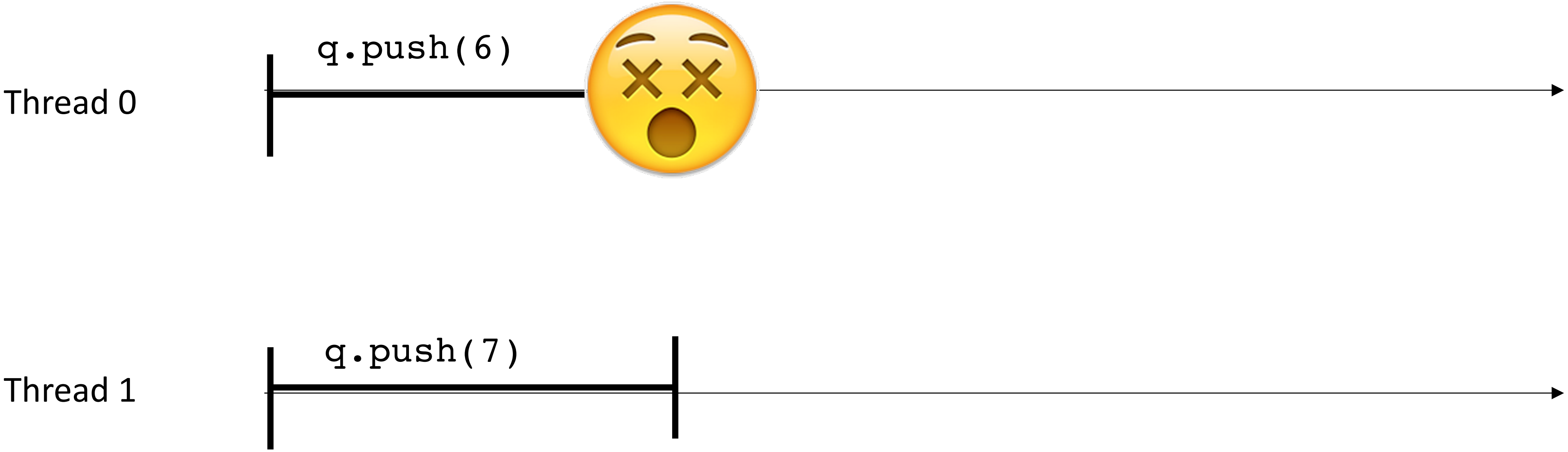
Thread 1

`q.push(7)`

for mutexes, the specification required that the system hang.
no such specification here.

Linearizability

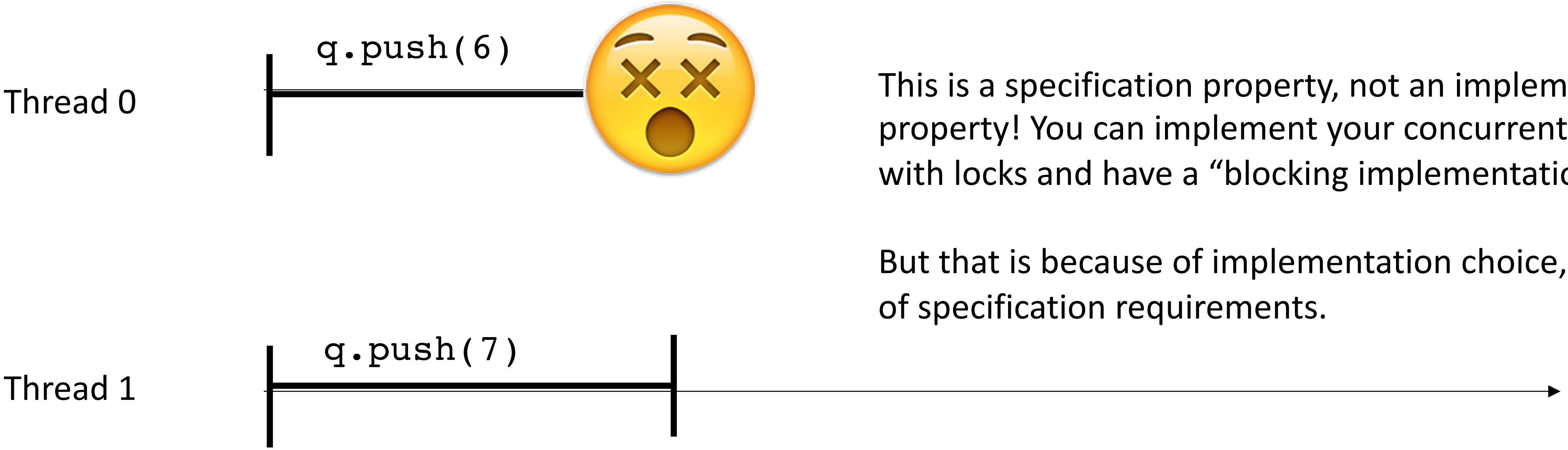
Non-blocking specification:
Every thread is allowed to continue executing
REGARDLESS of the behavior of other threads



for mutexes, the specification required that the system hang.
no such specification here.

Linearizability

Non-blocking specification:
Every thread is allowed to continue executing
REGARDLESS of the behavior of other threads



This is a specification property, not an implementation property! You can implement your concurrent objects with locks and have a “blocking implementation”.

But that is because of implementation choice, not because of specification requirements.

Terminology overview

- Thread-safe object:
- Lock-free object:
- Blocking specification:
- Non-blocking specification:
- (non-)blocking implementation:

Terminology overview

- Sequential consistency:

- Linearizability:

- Linearizability point:

Lecture schedule

- Revisiting sequential consistency

- Linearizablity

- Progress Properties

- **Implementing a set**

An example

- A sorted list:

Slides change style: I borrowed slides (with permission) from Roberto Palmieri (Lehigh University). They are based off slides by the book author

Set Interface

- Unordered collection of items
- No duplicates

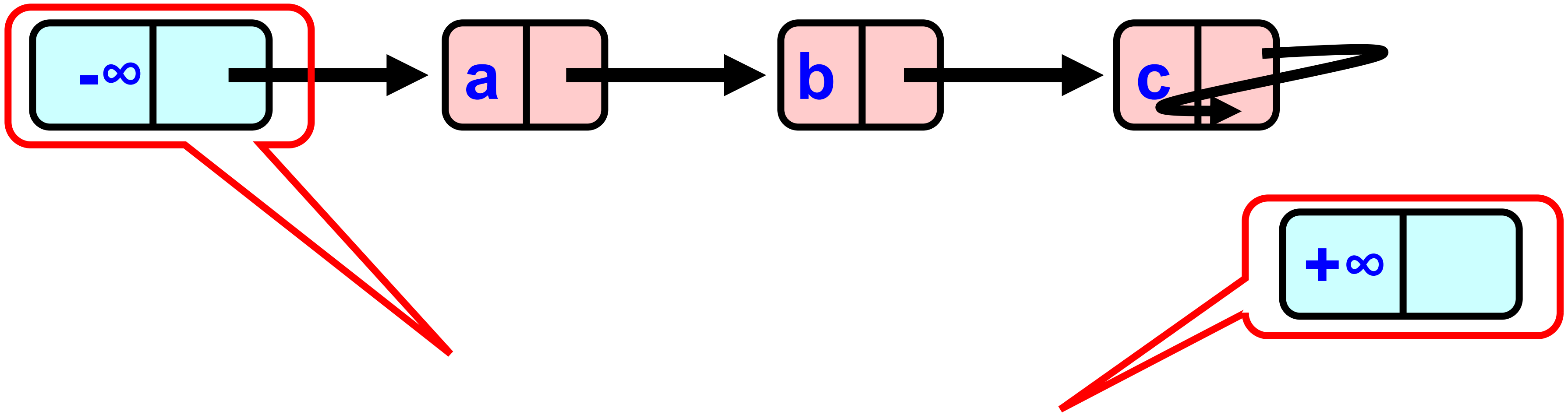
Set Interface

- Unordered collection of items
- No duplicates
- Methods
  - **`add(x)`** put **`x`** in set
  - **`remove(x)`** take **`x`** out of set
  - **`contains(x)`** tests if **`x`** in set

List Node

```
class Node {
 public:
   Value v;
   int key;
   Node *next;
}
```

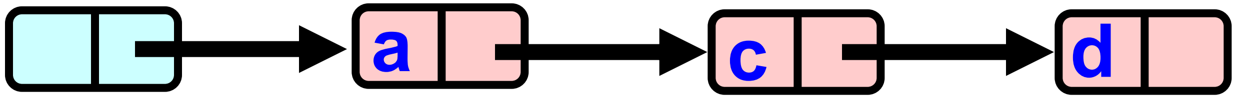
The List-Based Set



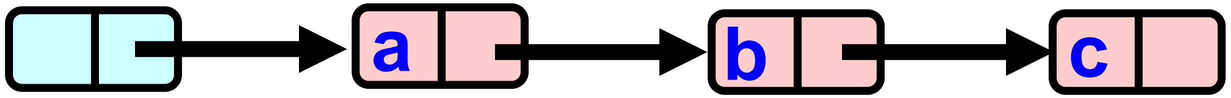
Sorted with Sentinel nodes
(min & max possible keys)

Sequential List Based Set

add(b)

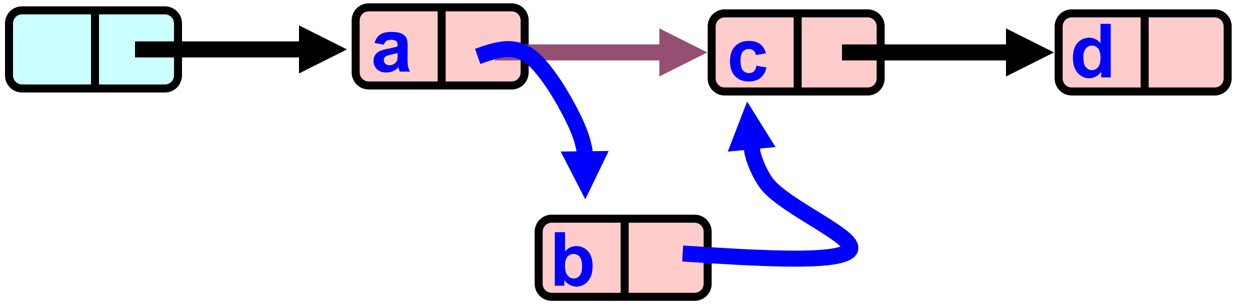


remove(b)

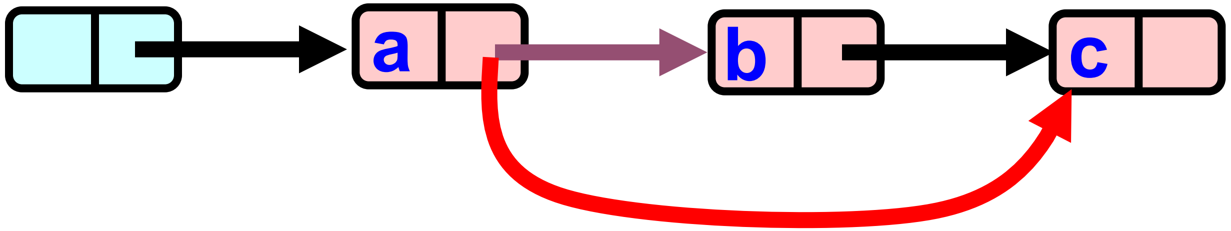


Sequential List Based Set

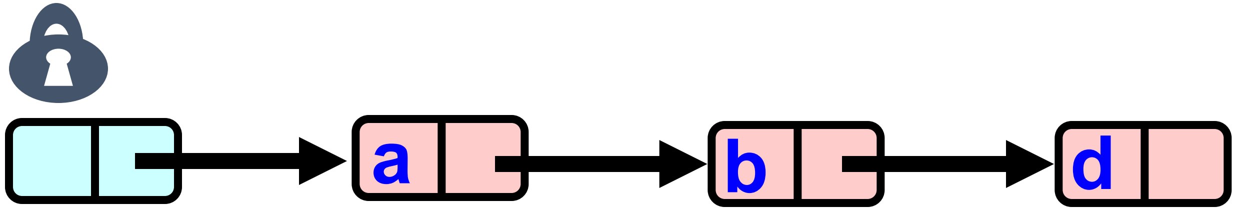
add(b)



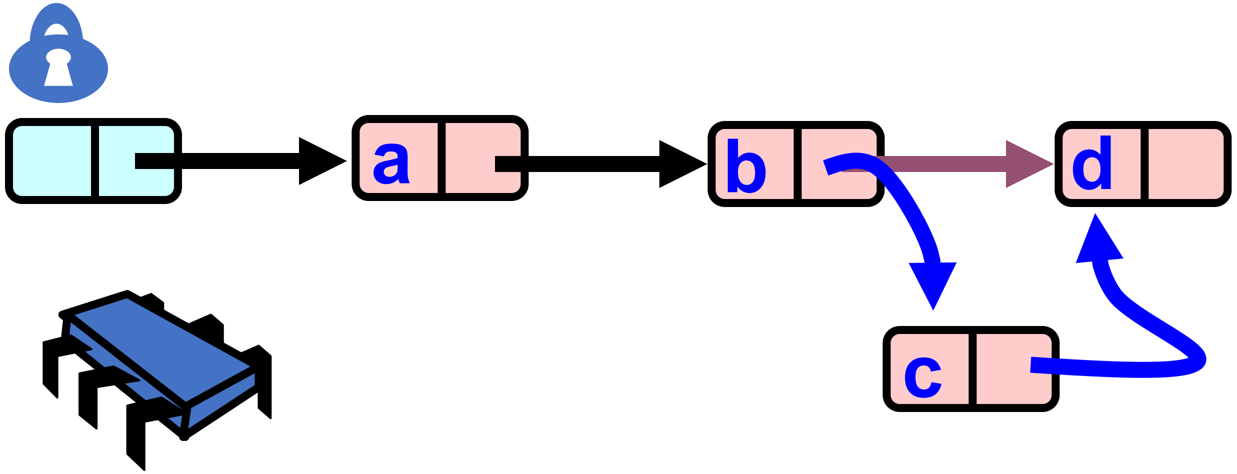
remove(b)



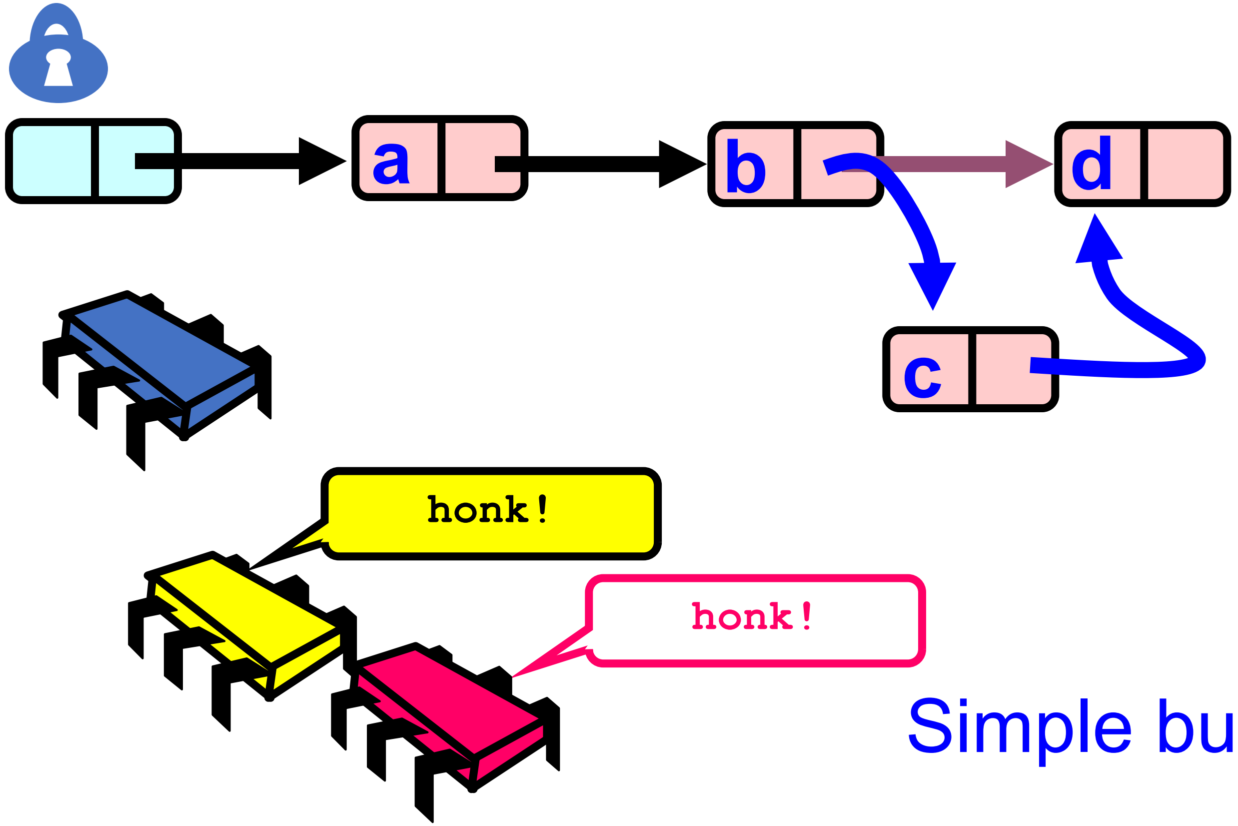
Coarse-Grained Locking



Coarse-Grained Locking



Coarse-Grained Locking

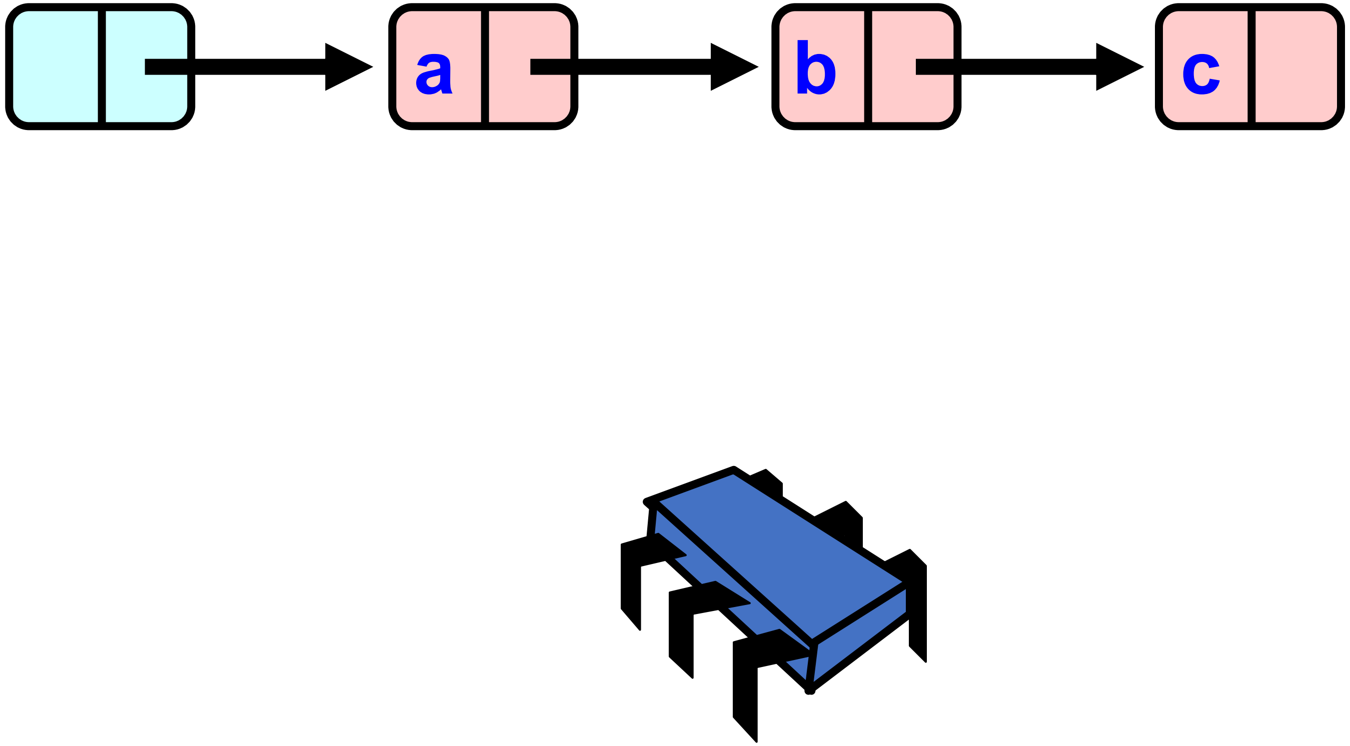


Simple but inefficient!

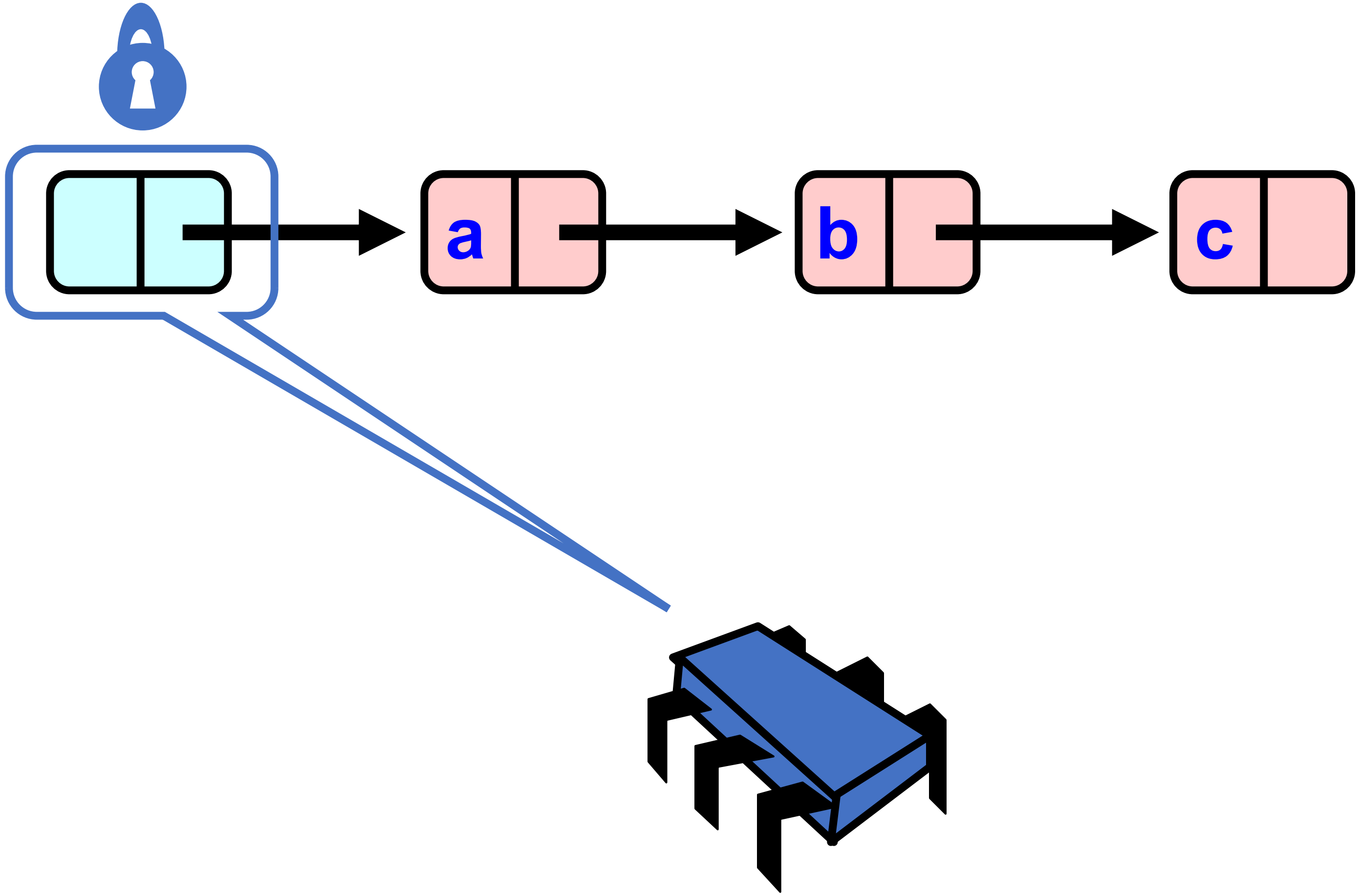
Fine-grained Locking

- Requires **careful** thought
- Split object into pieces
    - Each piece has own lock
    - Methods that work on disjoint pieces need not exclude each other

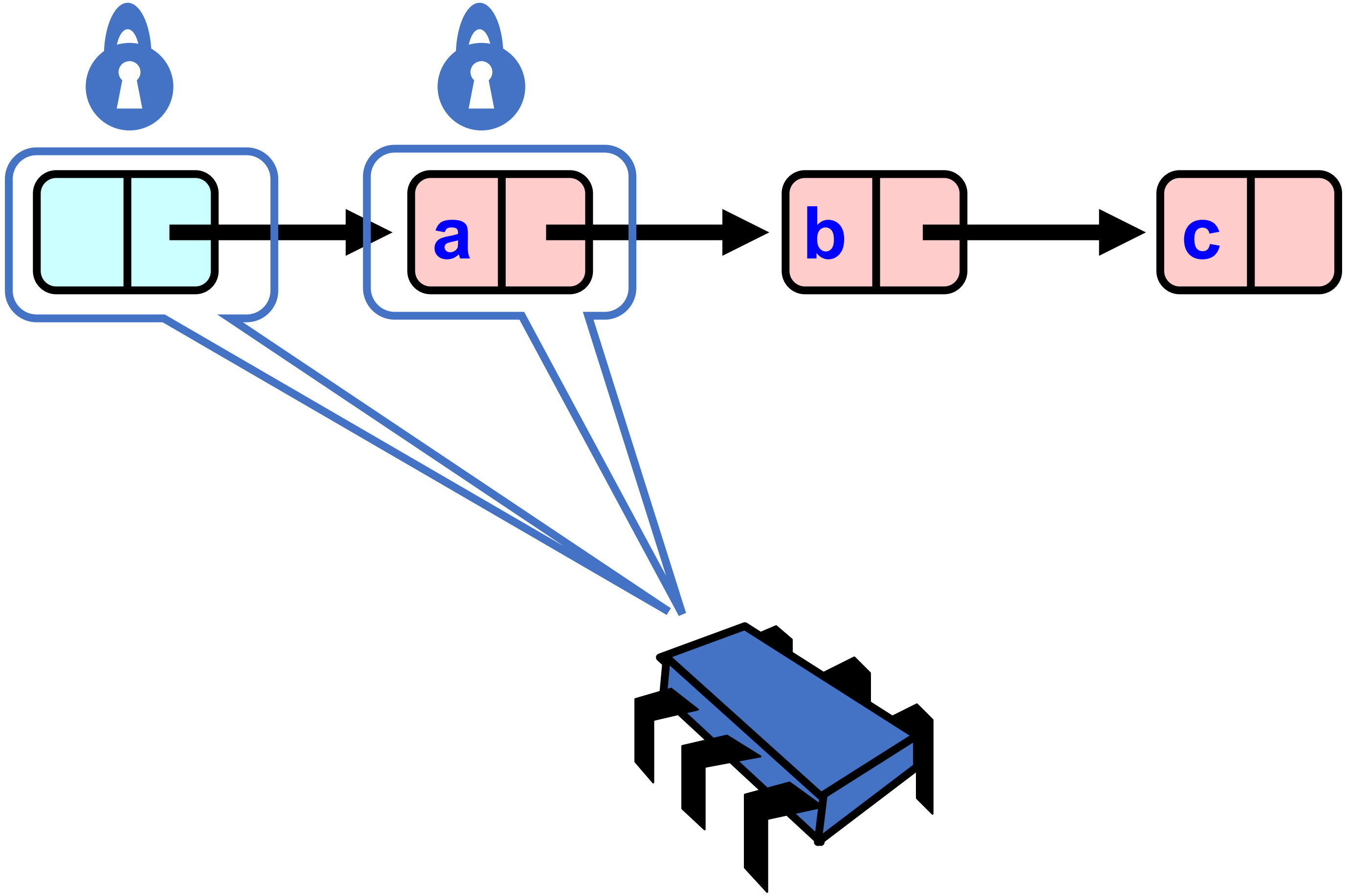
Hand-over-Hand locking



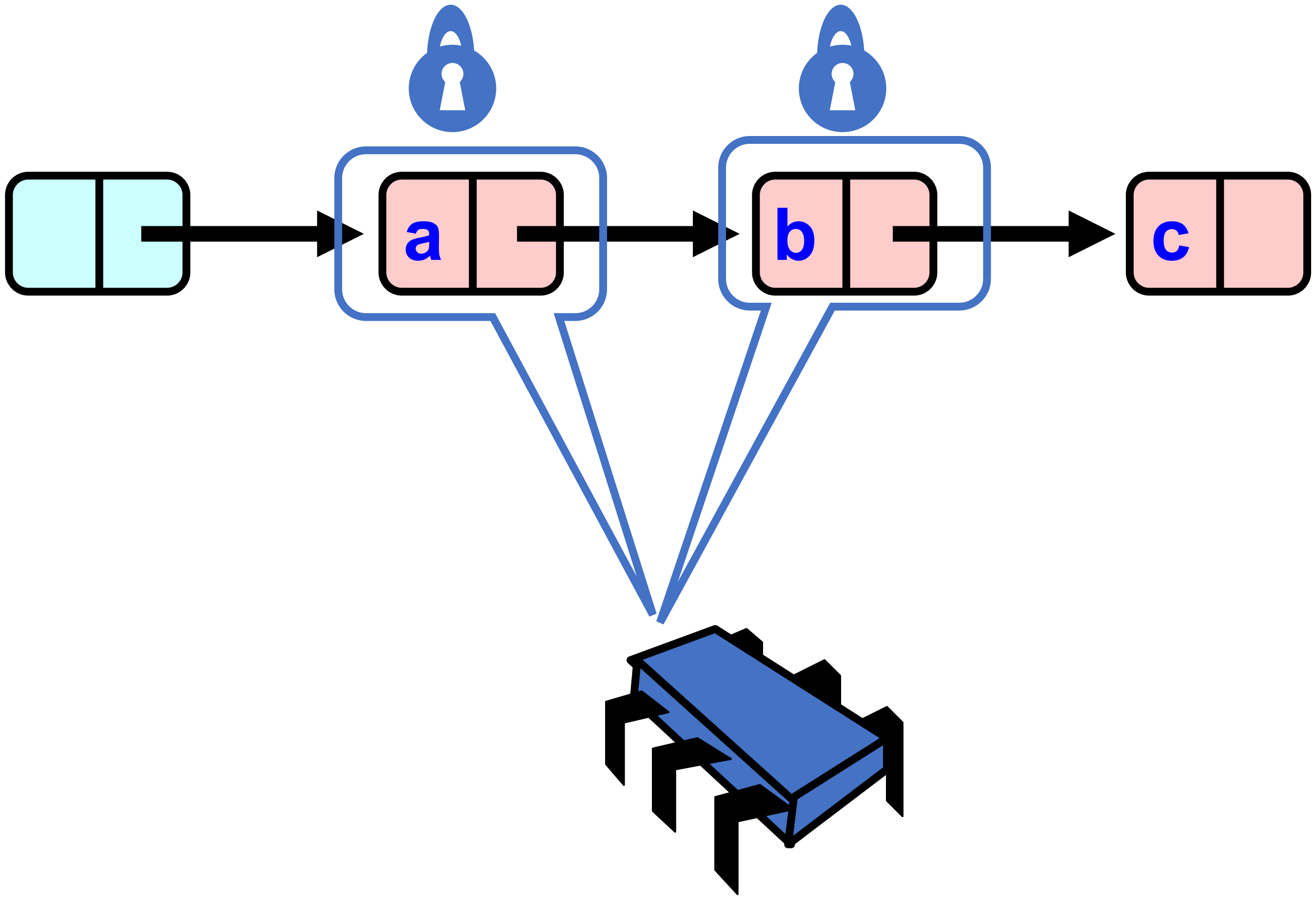
Hand-over-Hand locking



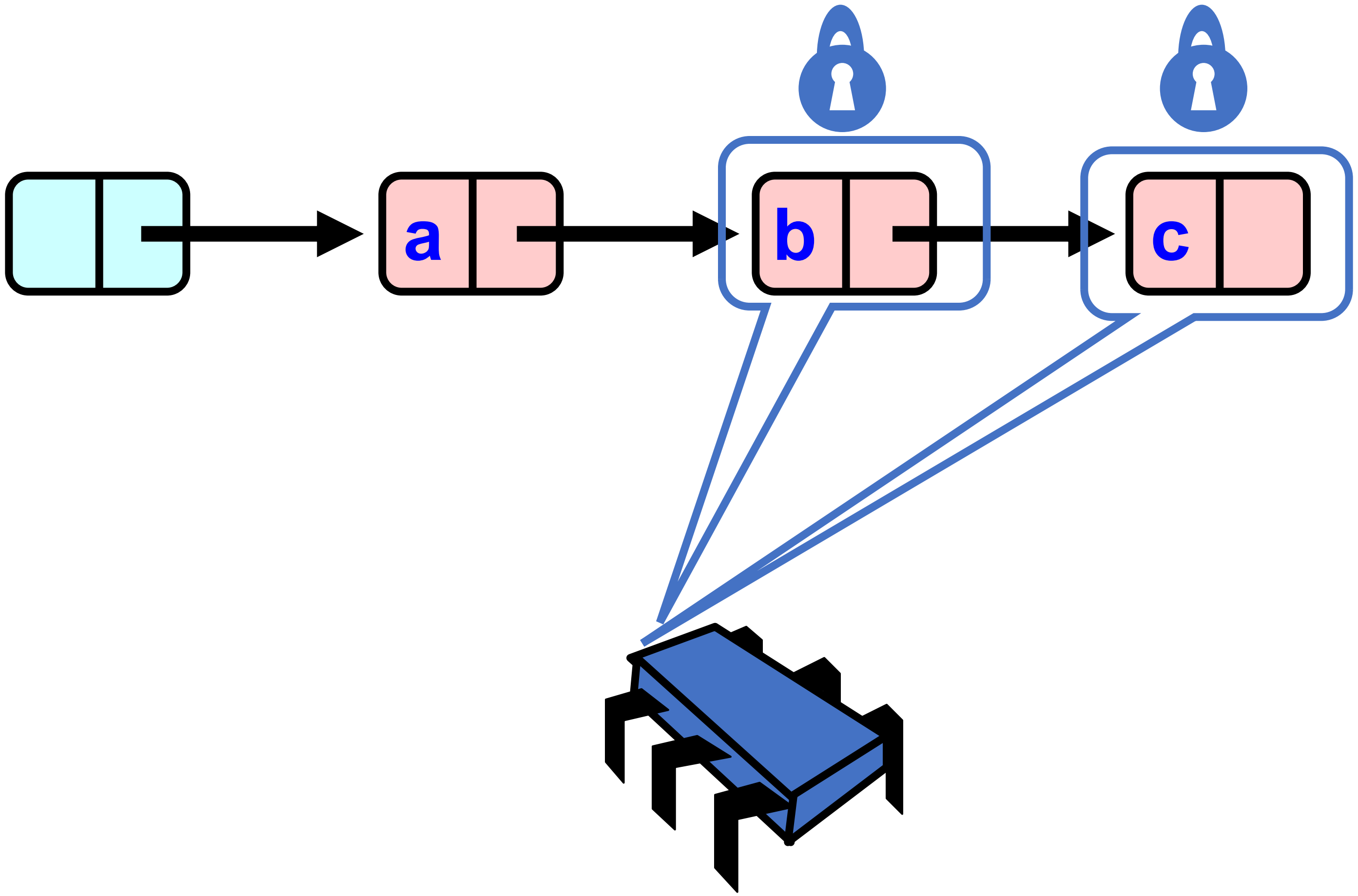
Hand-over-Hand locking



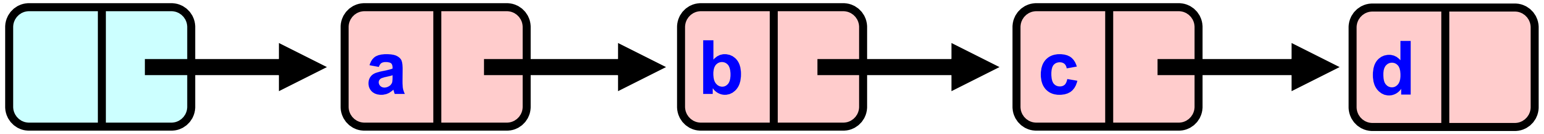
Hand-over-Hand locking

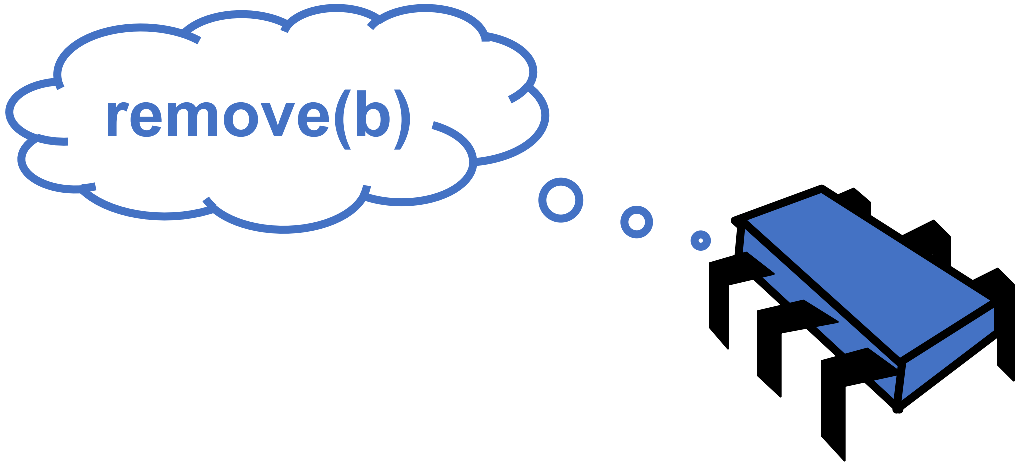


Hand-over-Hand locking

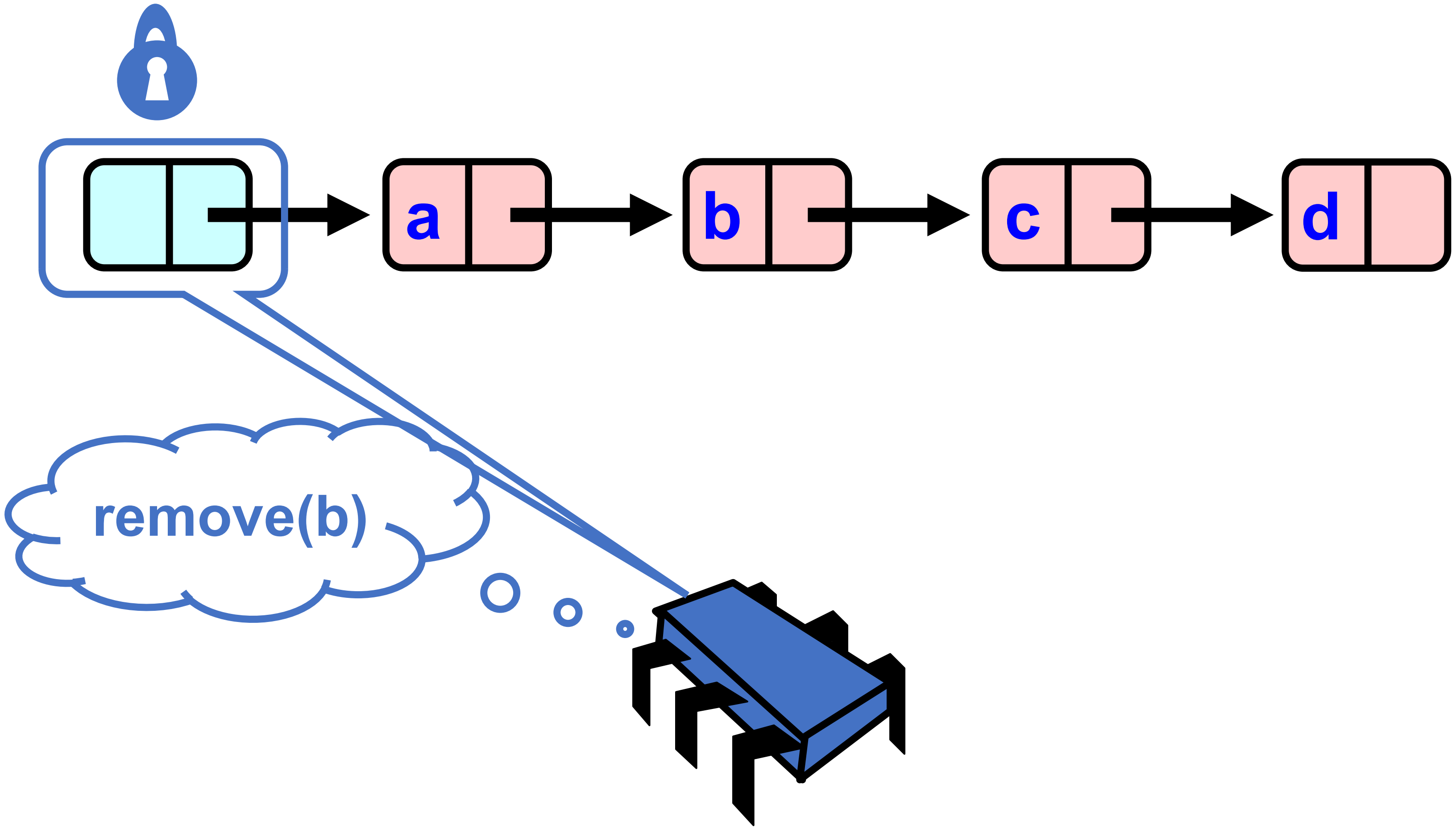


Removing a Node

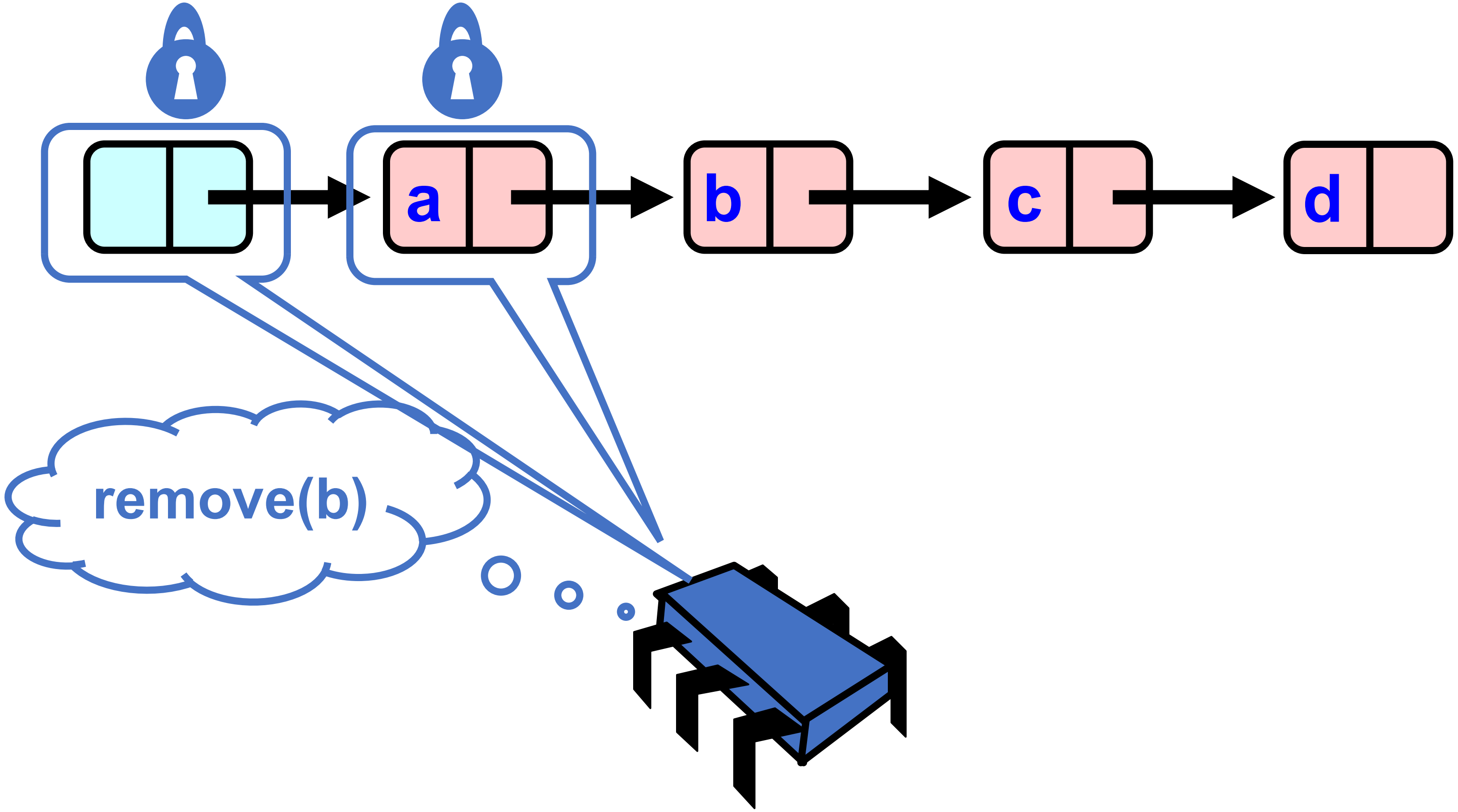




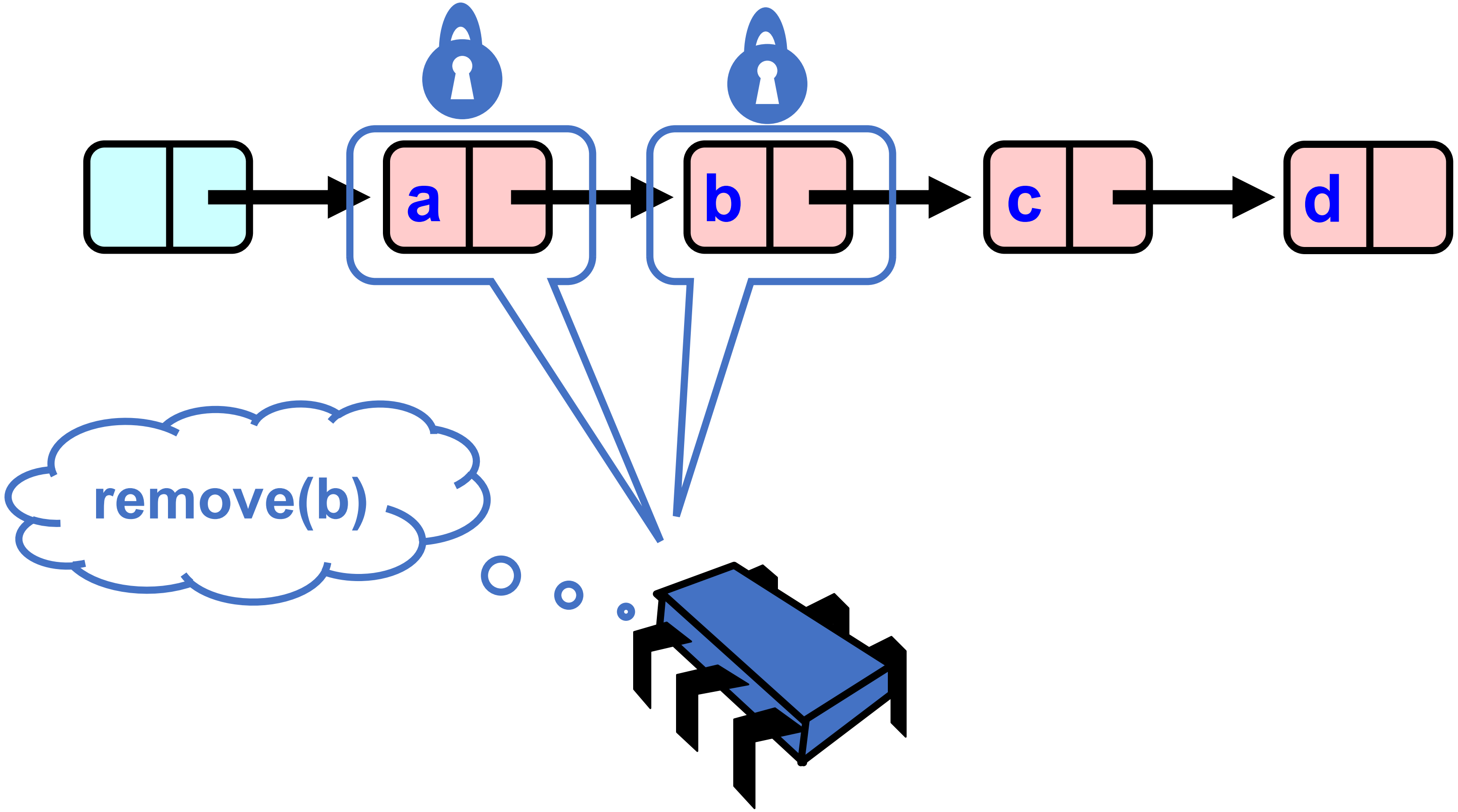
Removing a Node



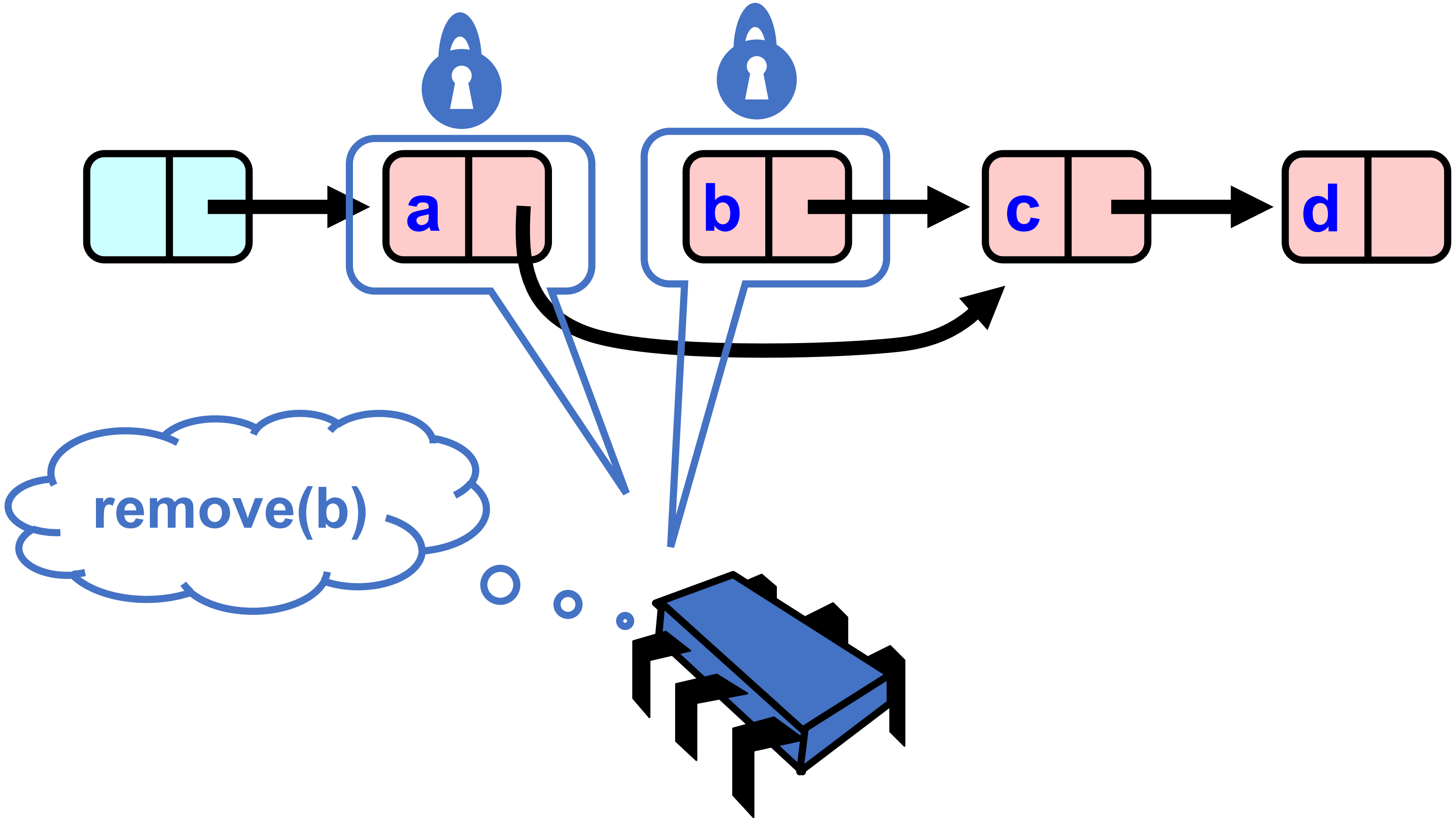
Removing a Node



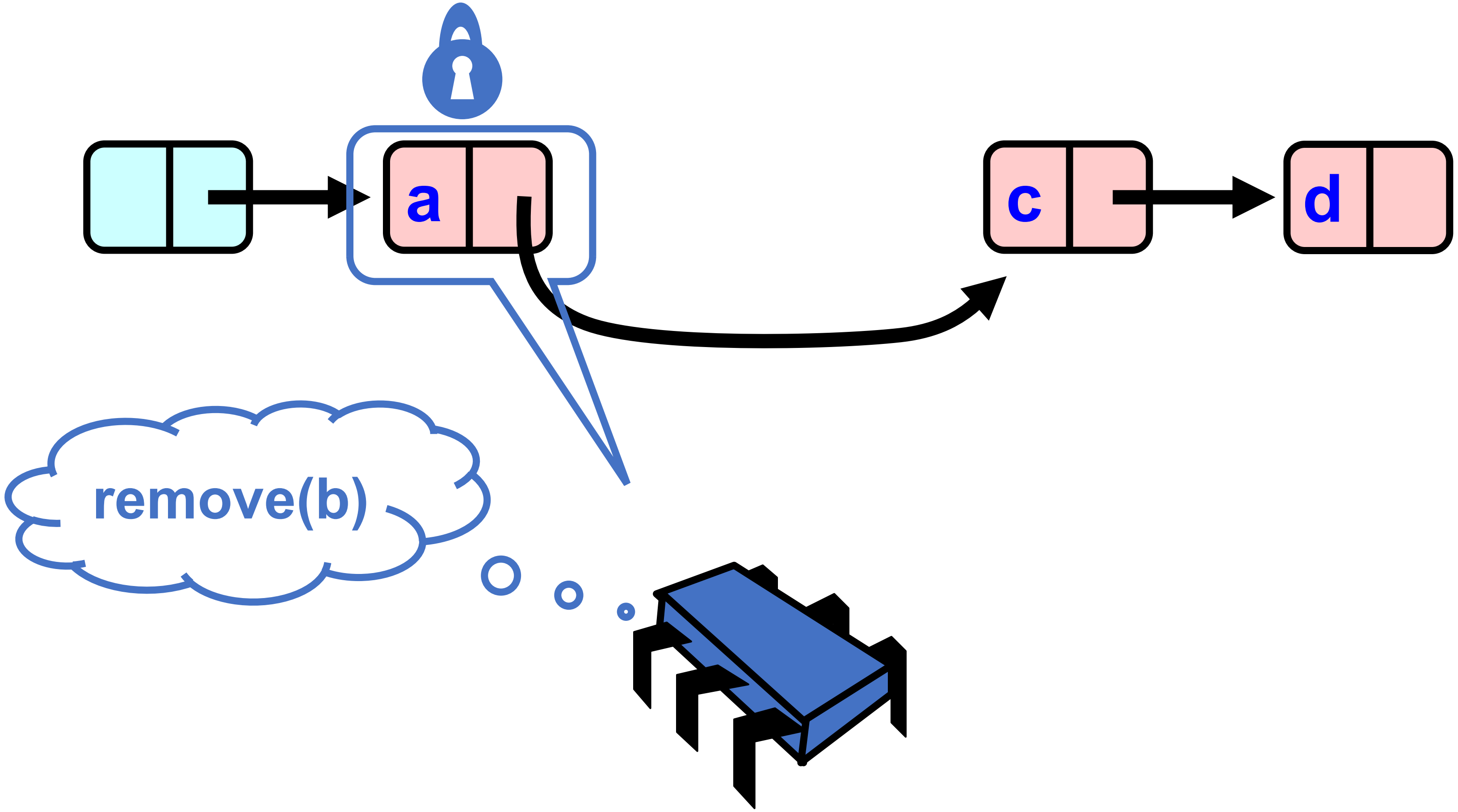
Removing a Node



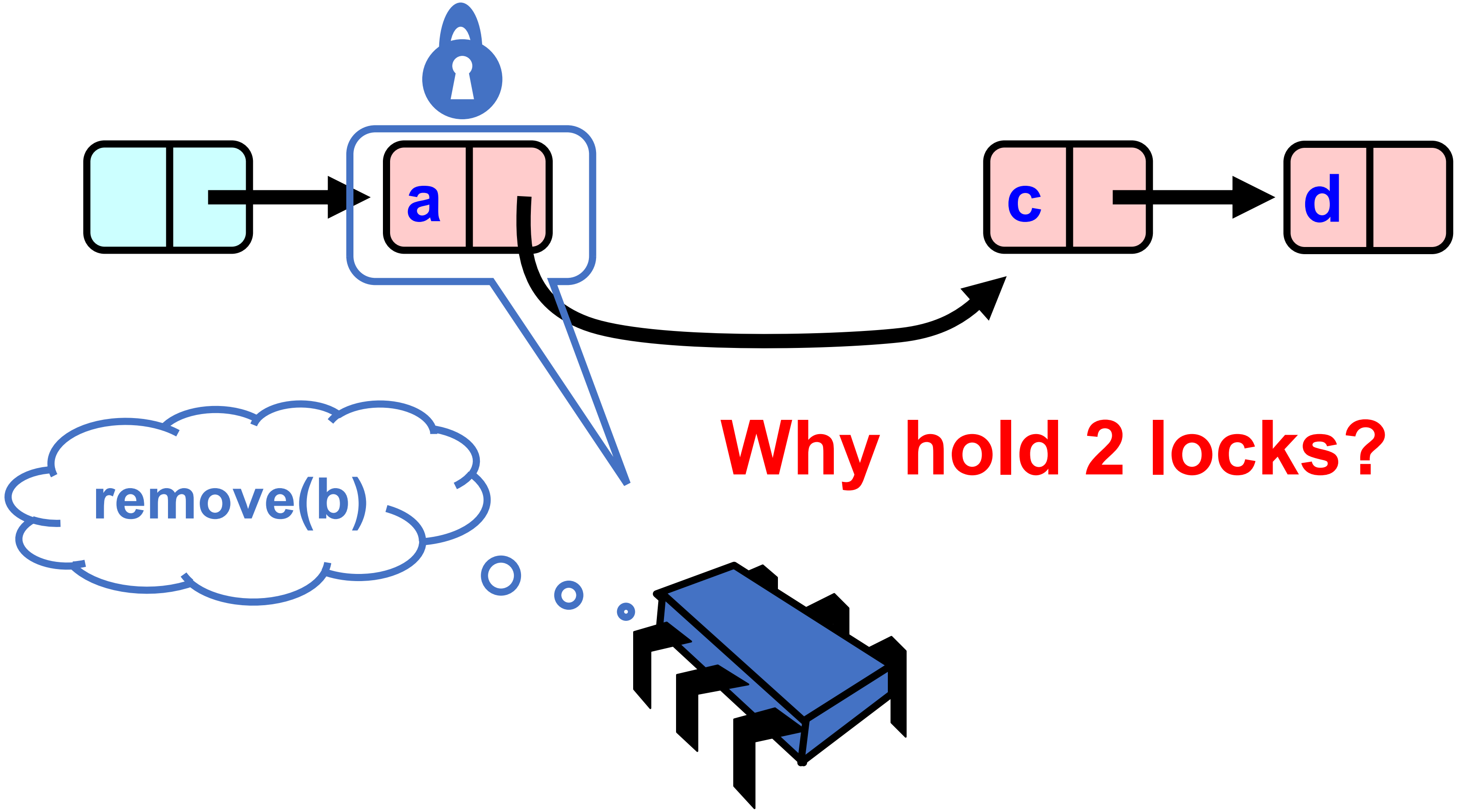
Removing a Node



Removing a Node

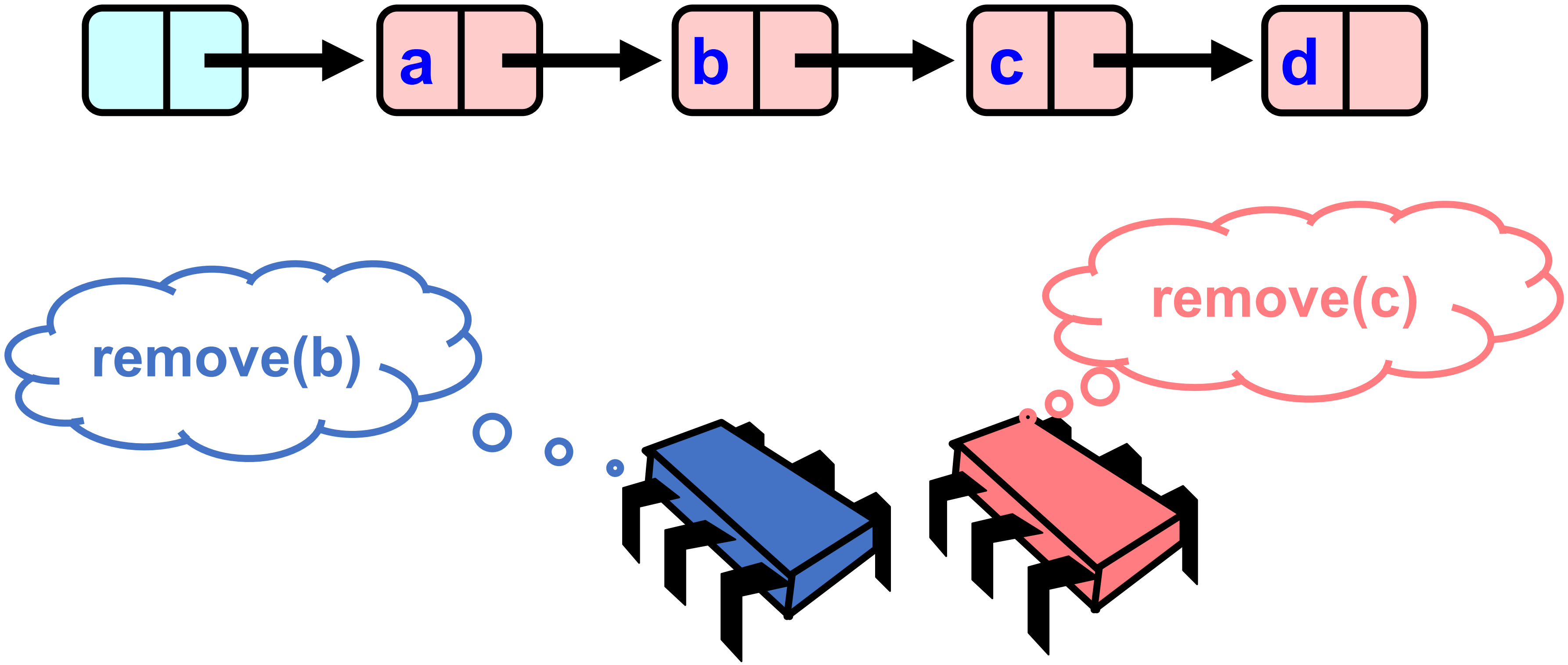


Removing a Node

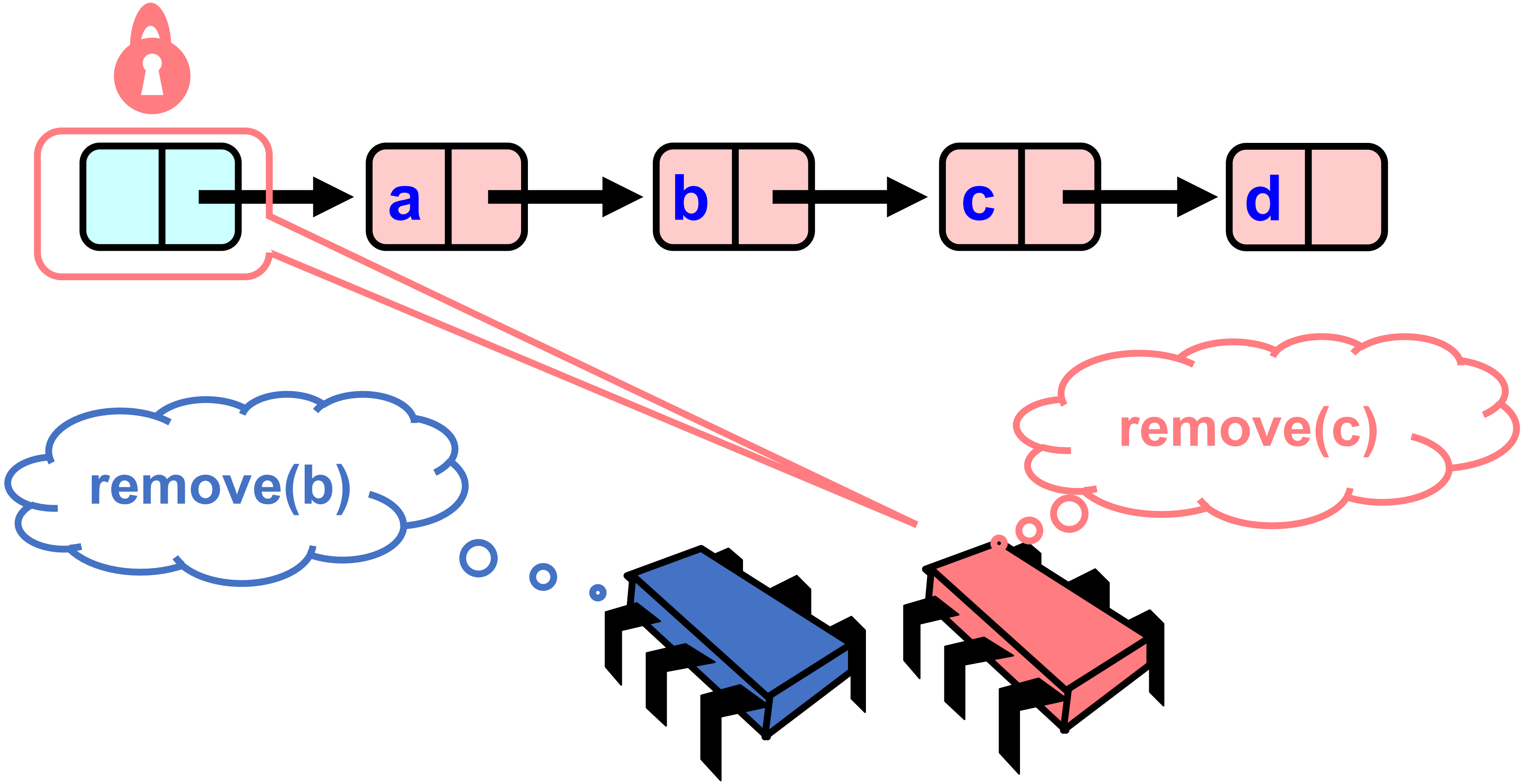


Why hold 2 locks?

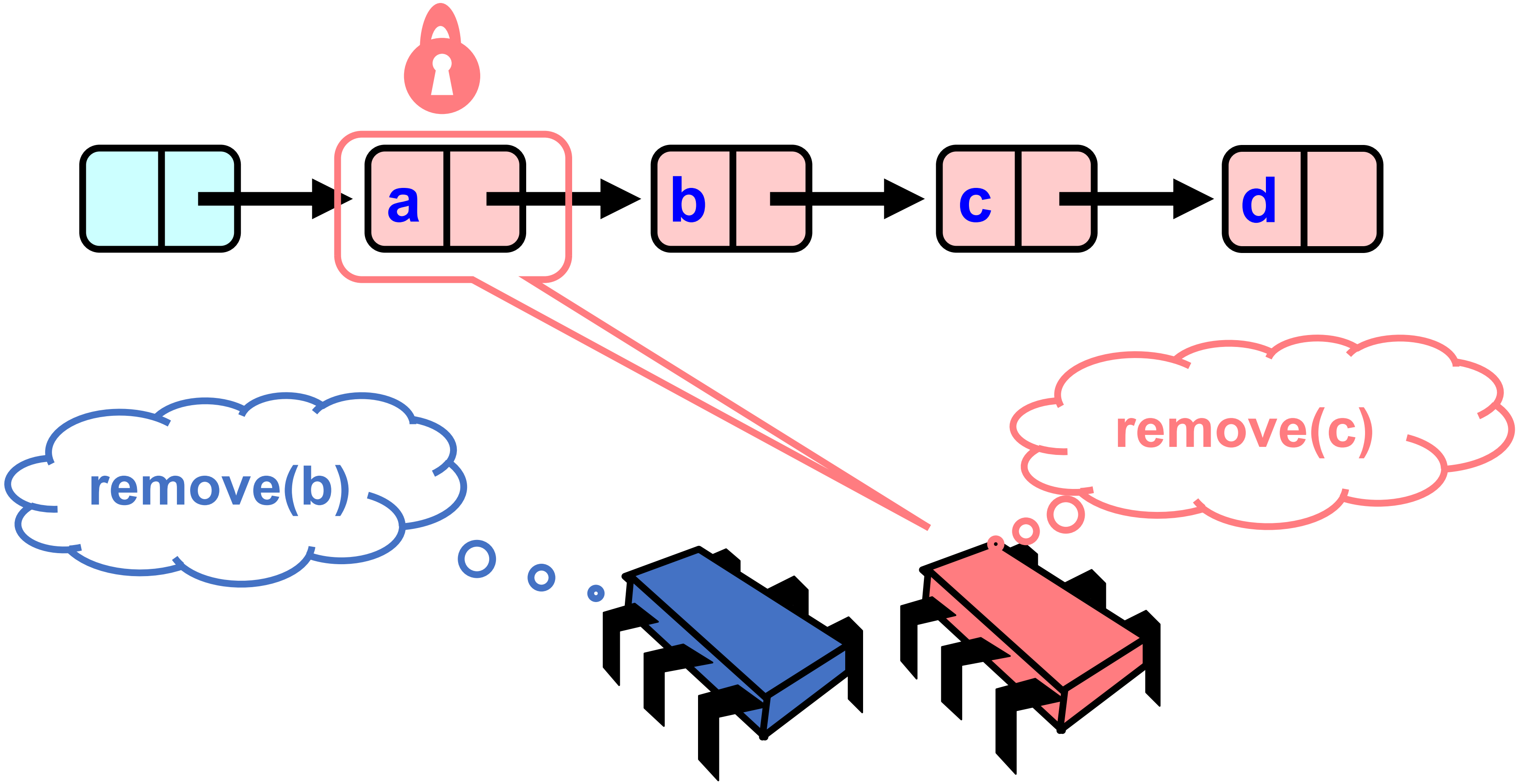
Concurrent Removes



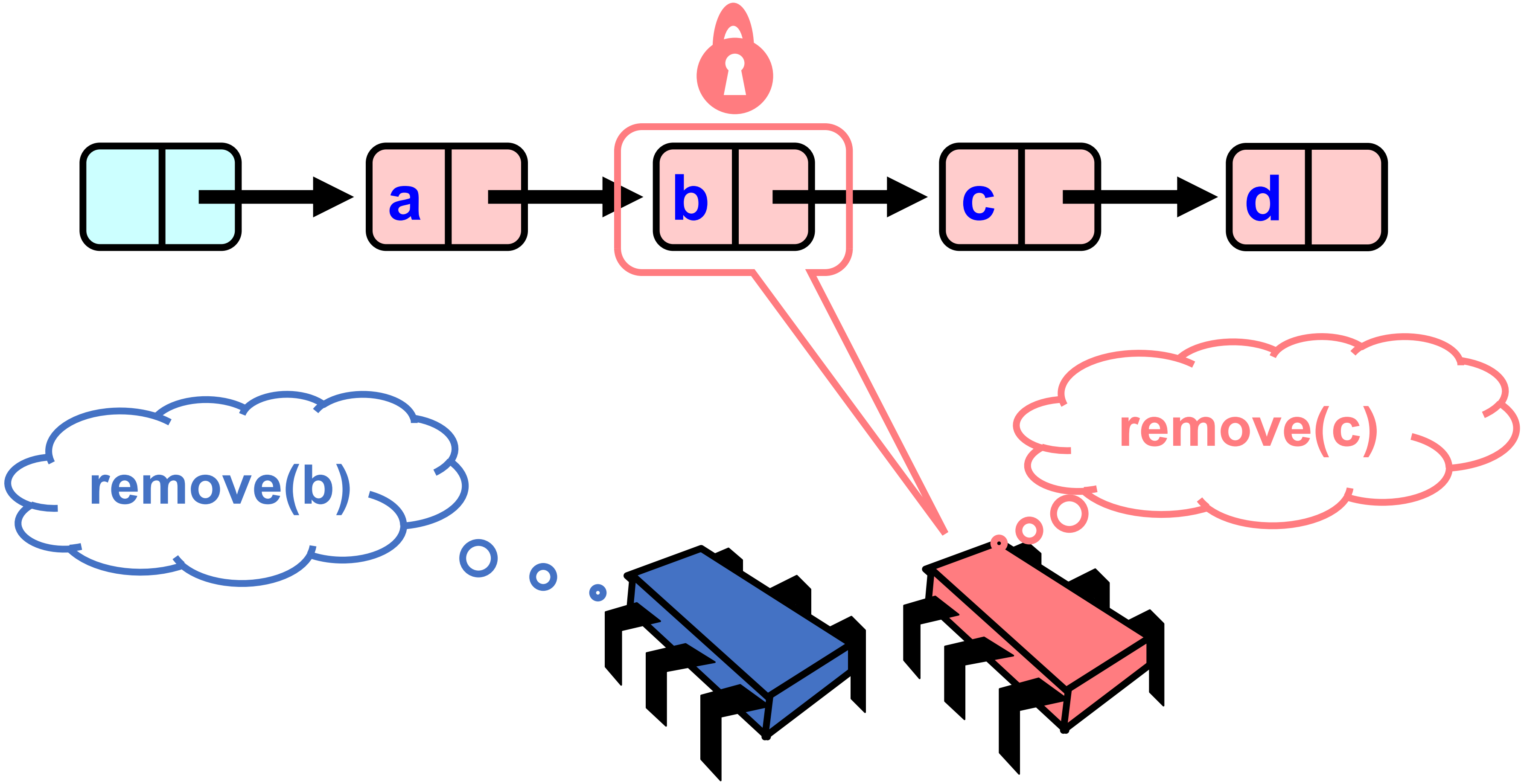
Concurrent Removes



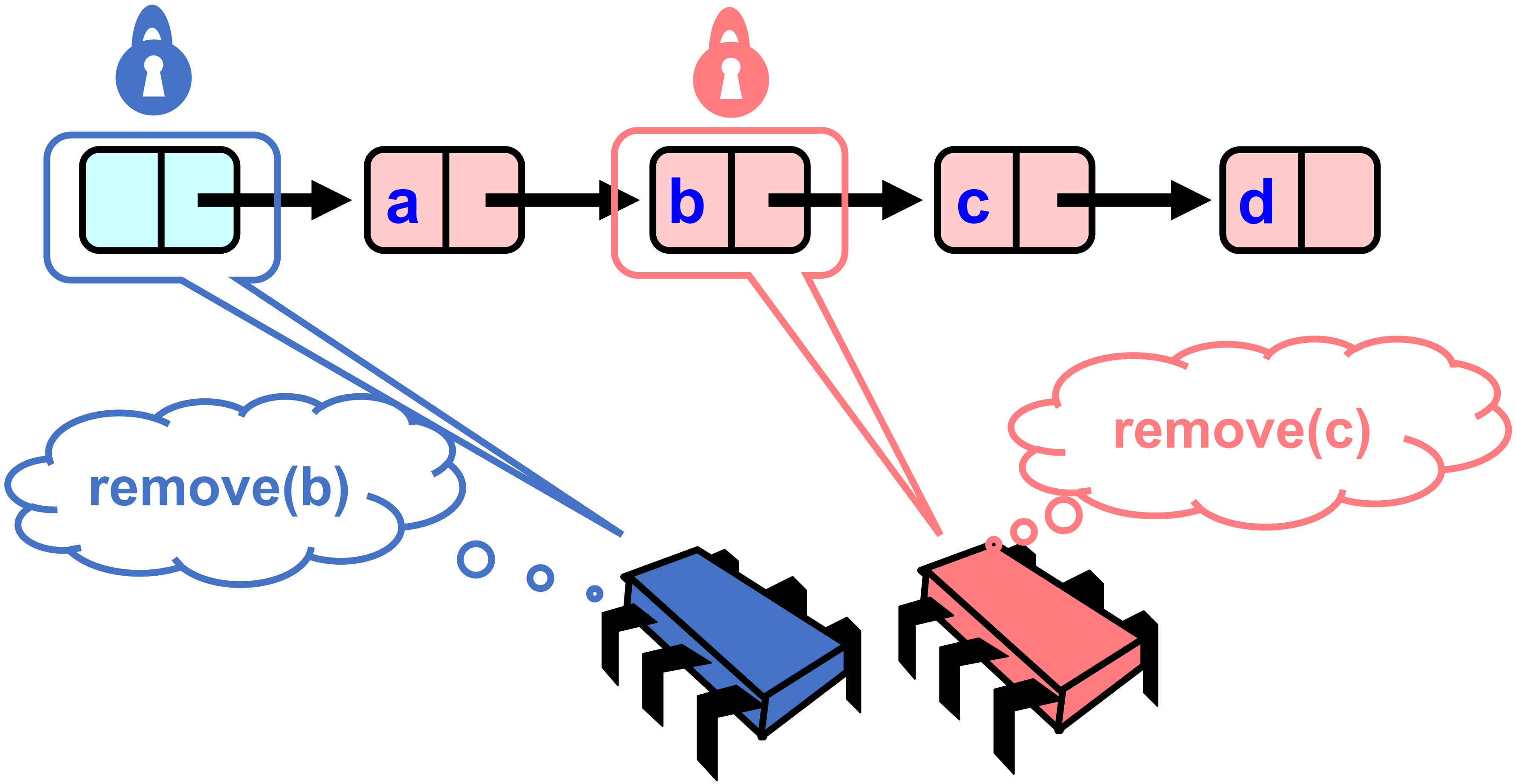
Concurrent Removes



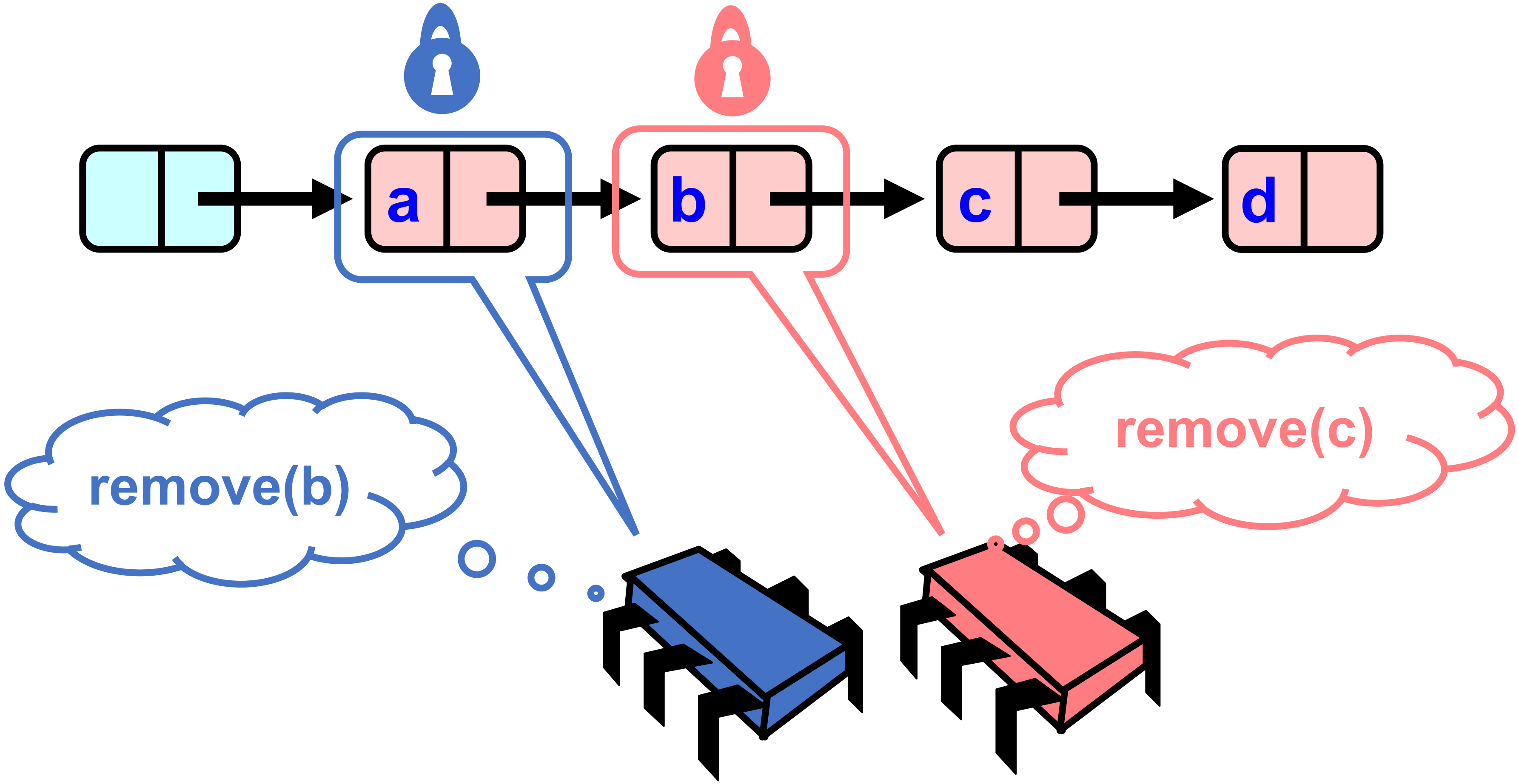
Concurrent Removes



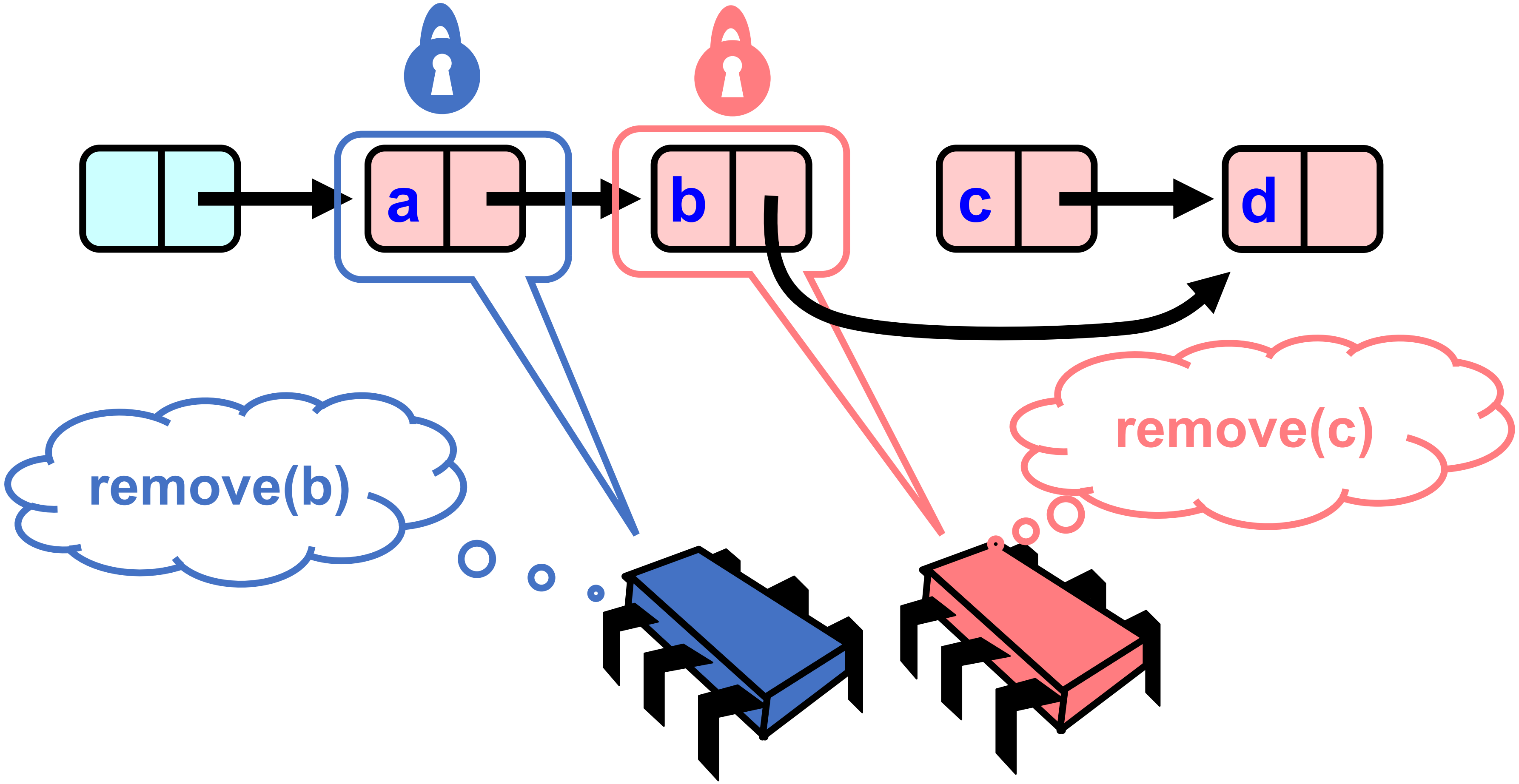
Concurrent Removes



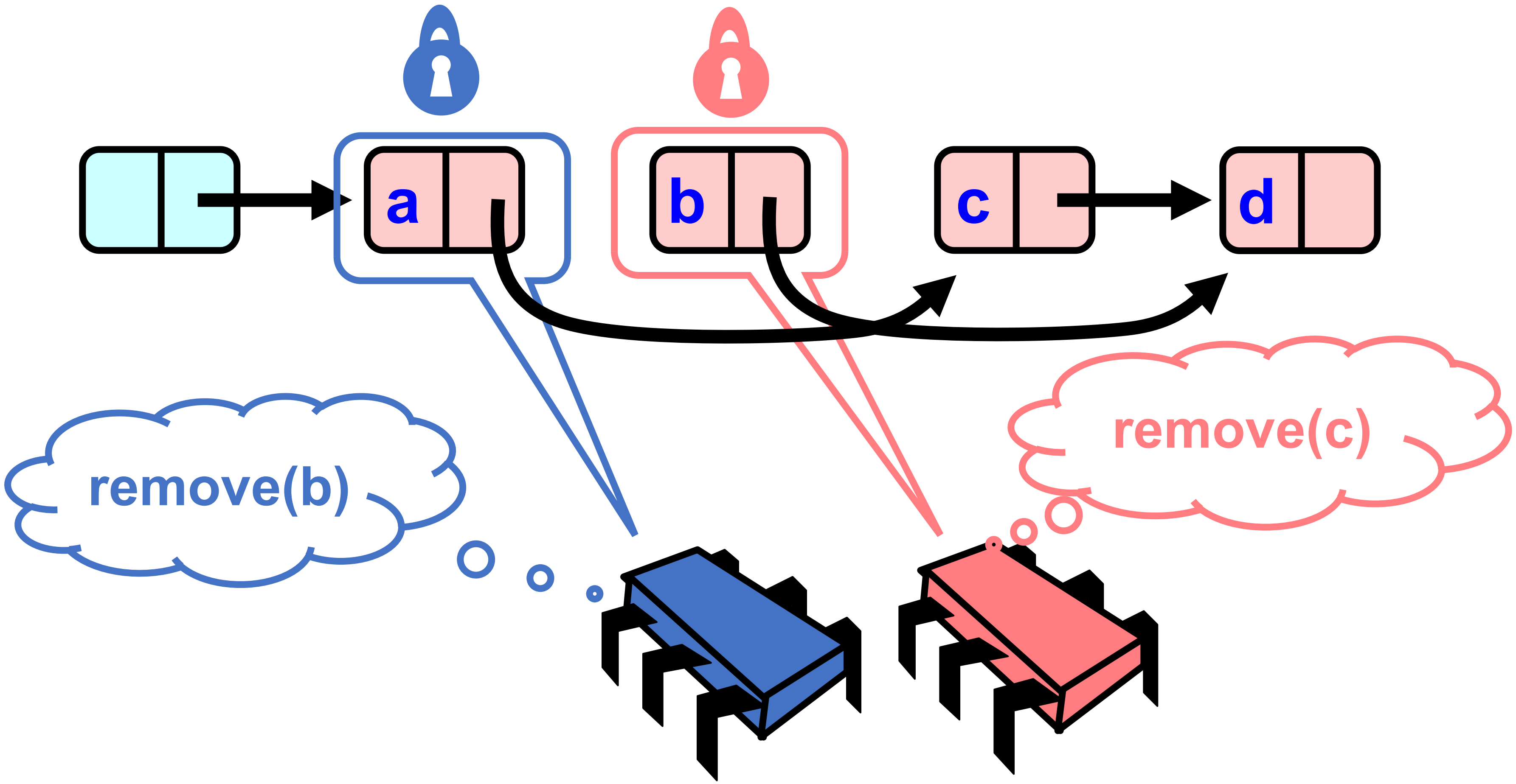
Concurrent Removes



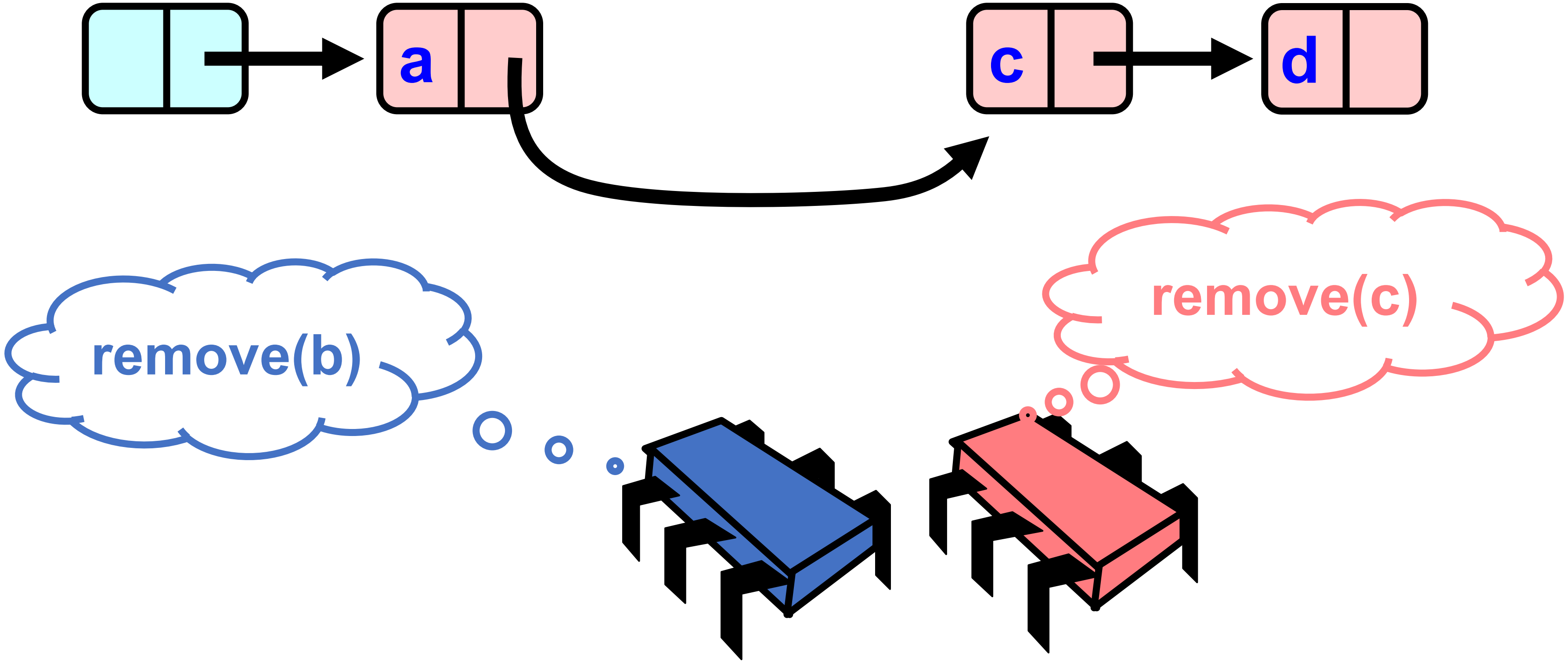
Concurrent Removes



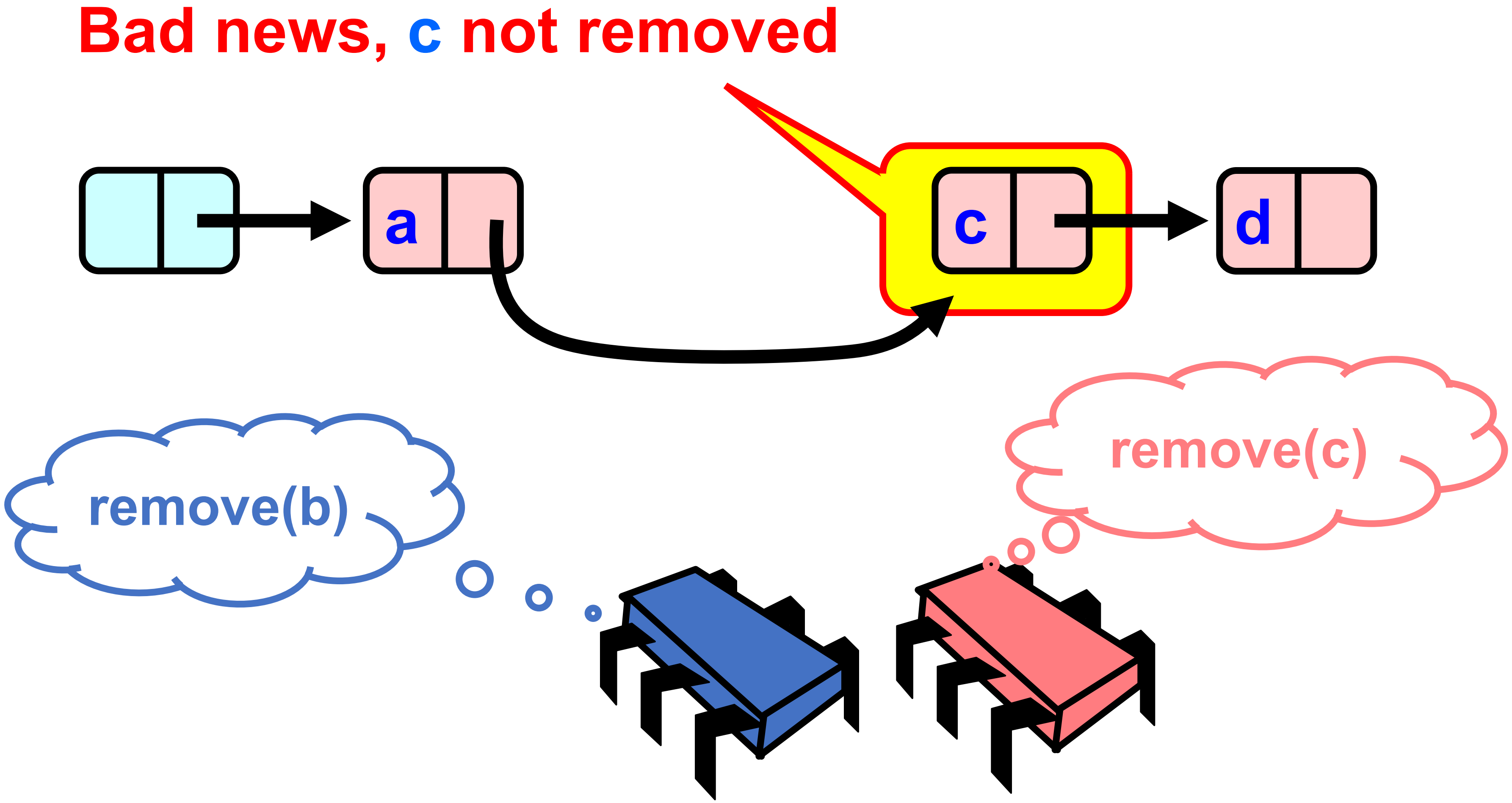
Concurrent Removes



Uh, Oh

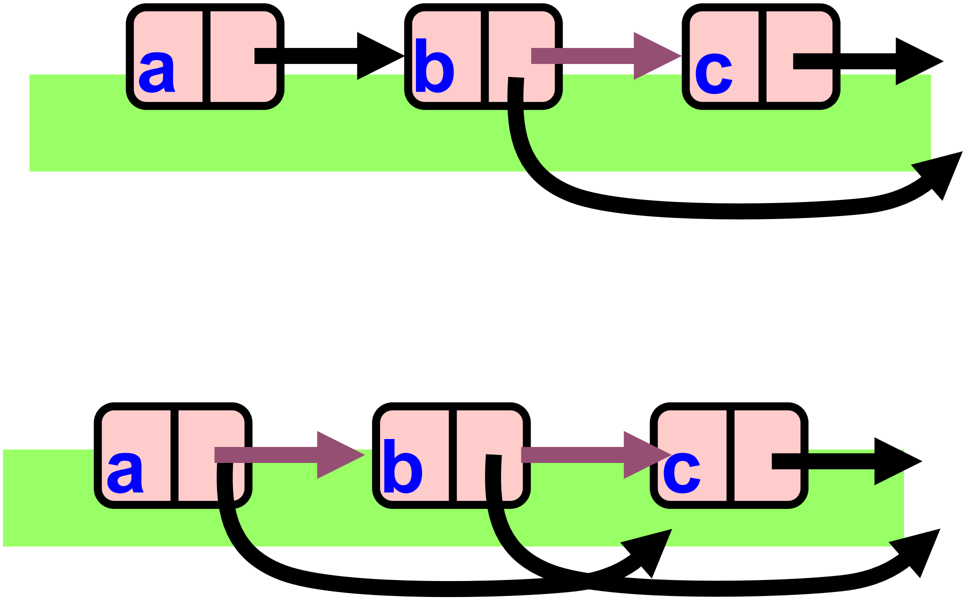


Uh, Oh



Problem

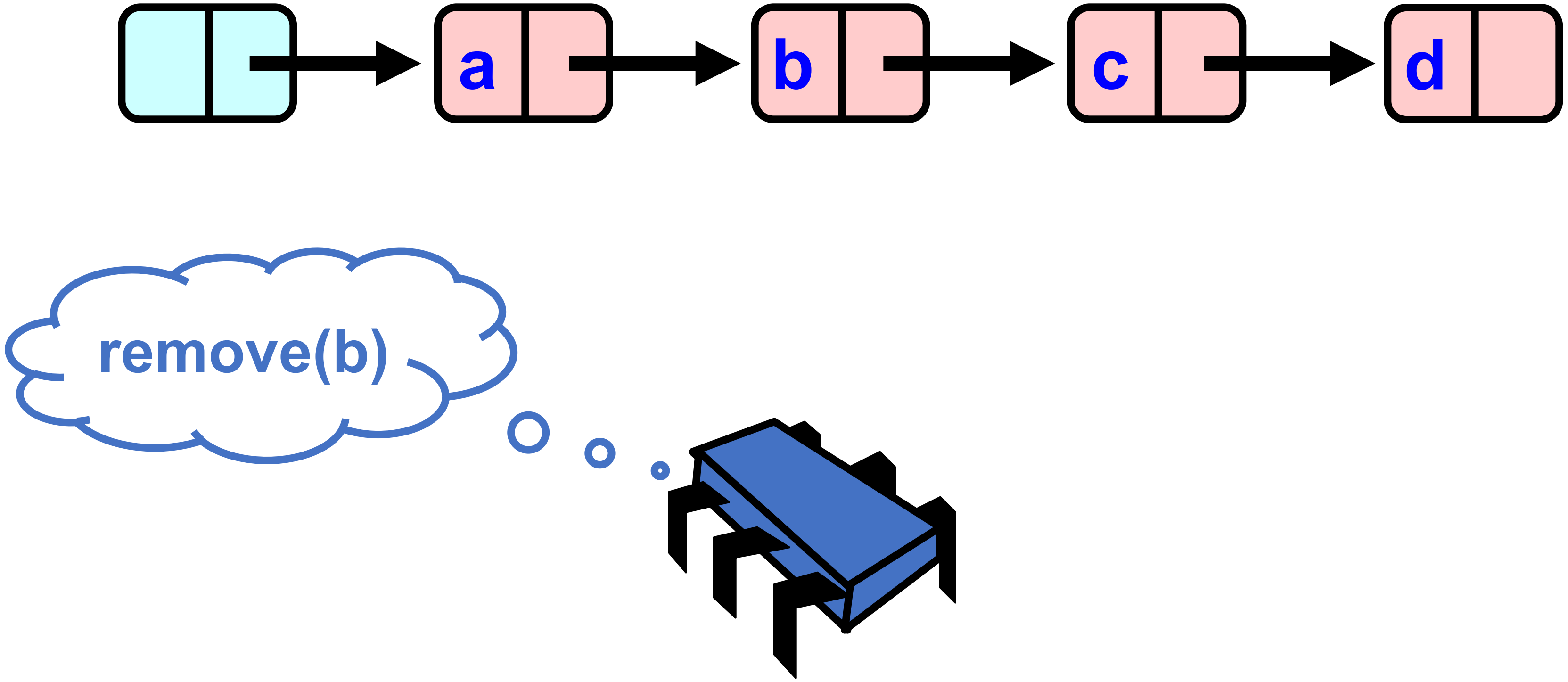
- To delete node c
  - Swing node b’s next field to d
- Problem is,
  - ***Data conflict:***
  - Someone deleting b concurrently could direct a pointer to C



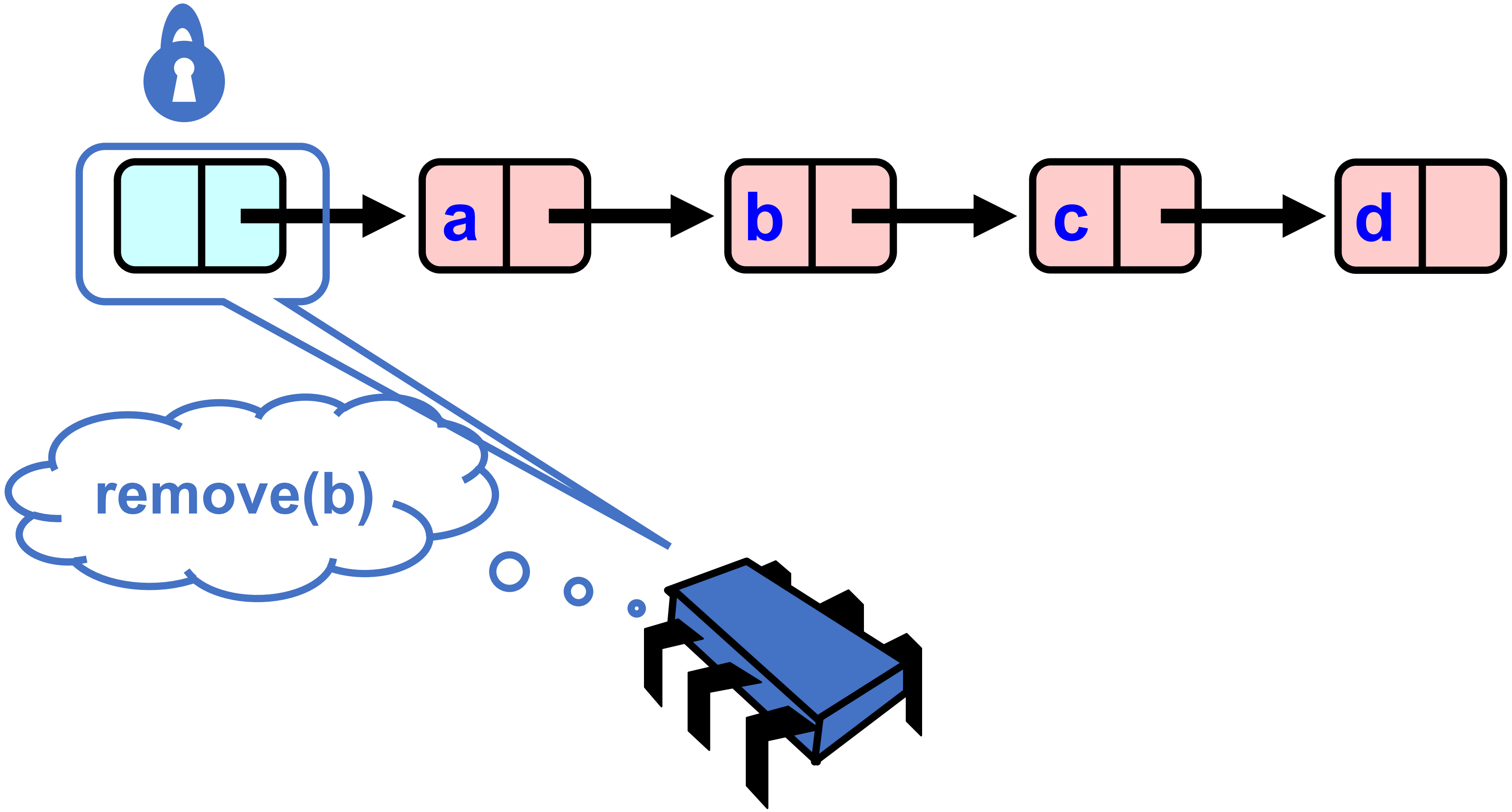
Insight

- If a node is locked
  - No one can delete node’s *successor*
- If a thread locks
  - Node to be deleted
  - And its predecessor
  - Then it works

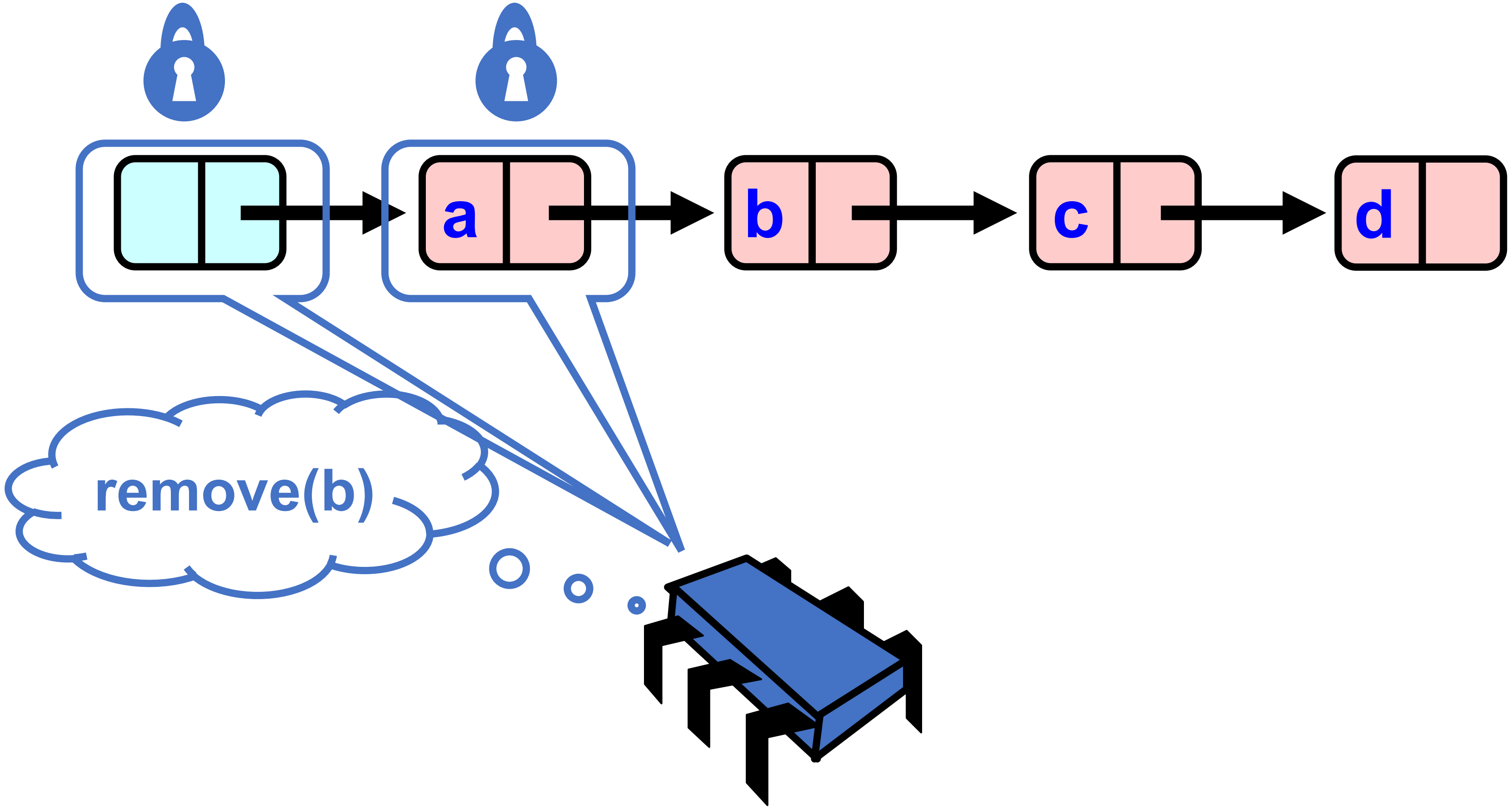
Hand-Over-Hand Again



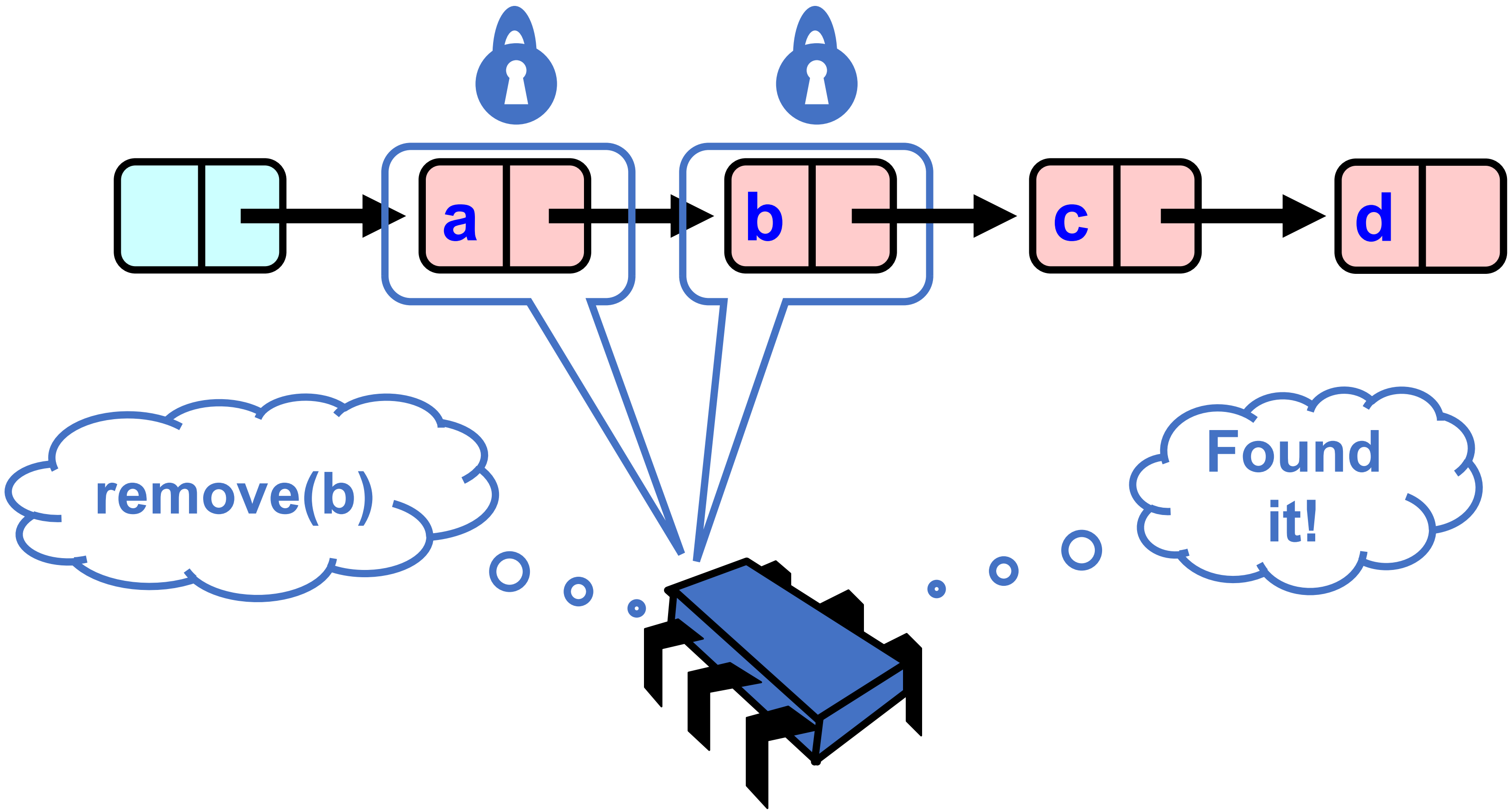
Hand-Over-Hand Again



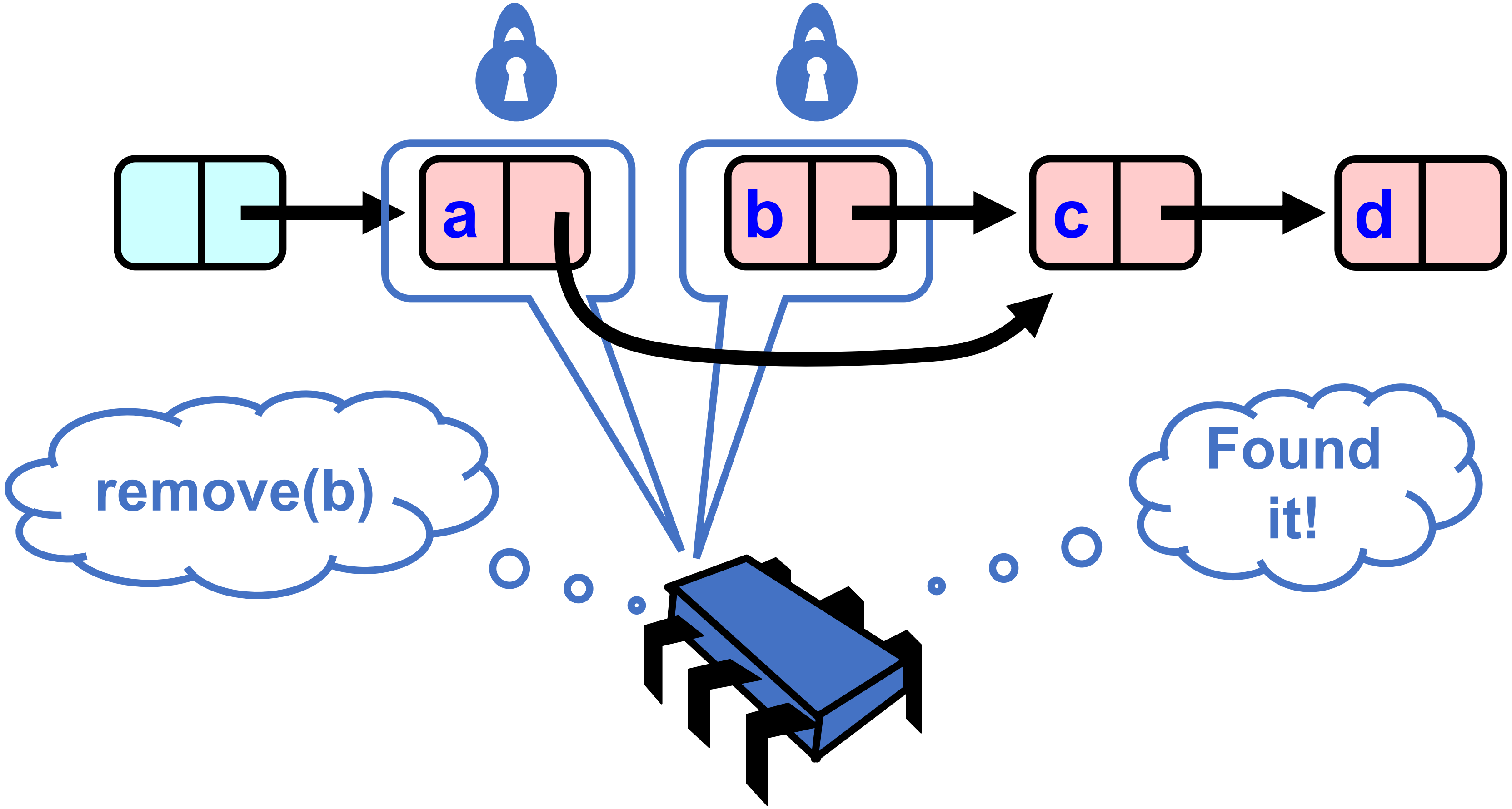
Hand-Over-Hand Again



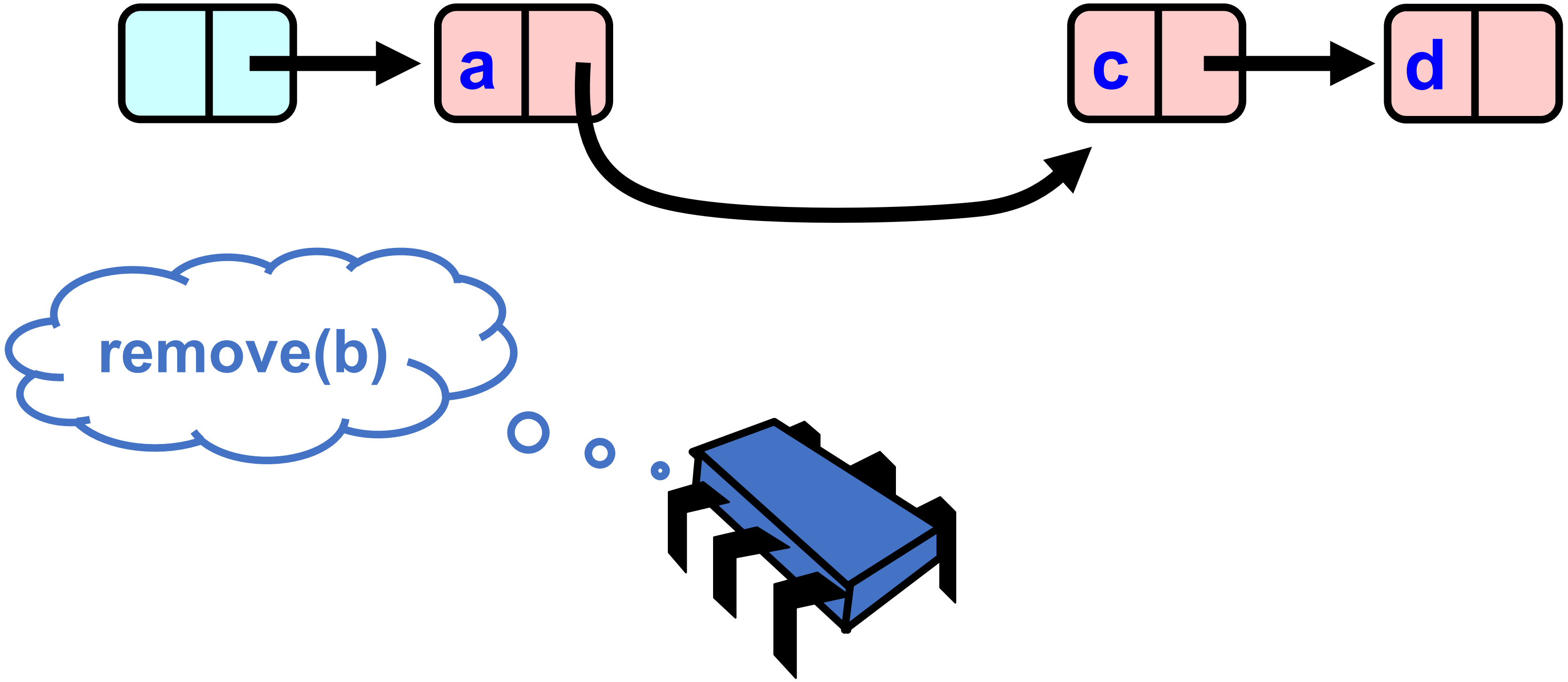
Hand-Over-Hand Again



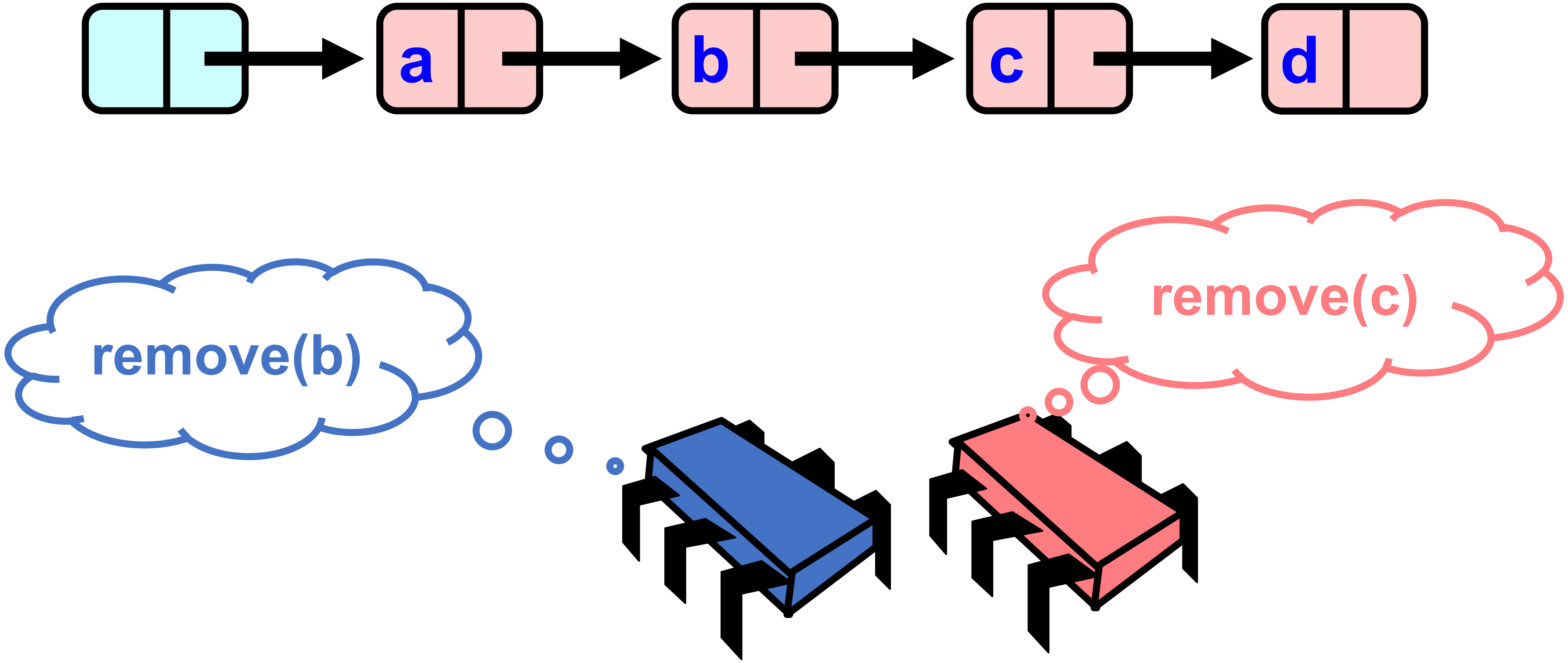
Hand-Over-Hand Again



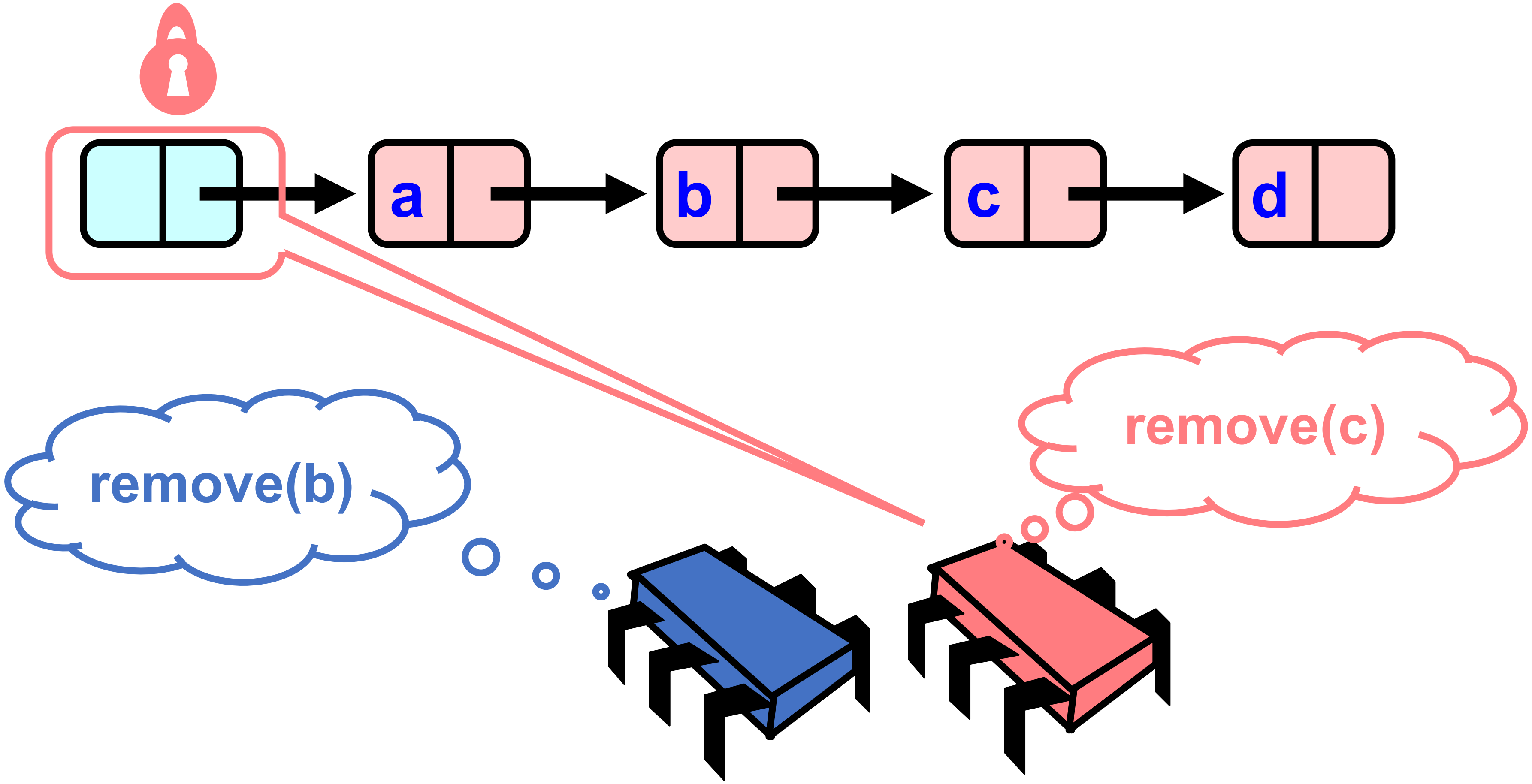
Hand-Over-Hand Again



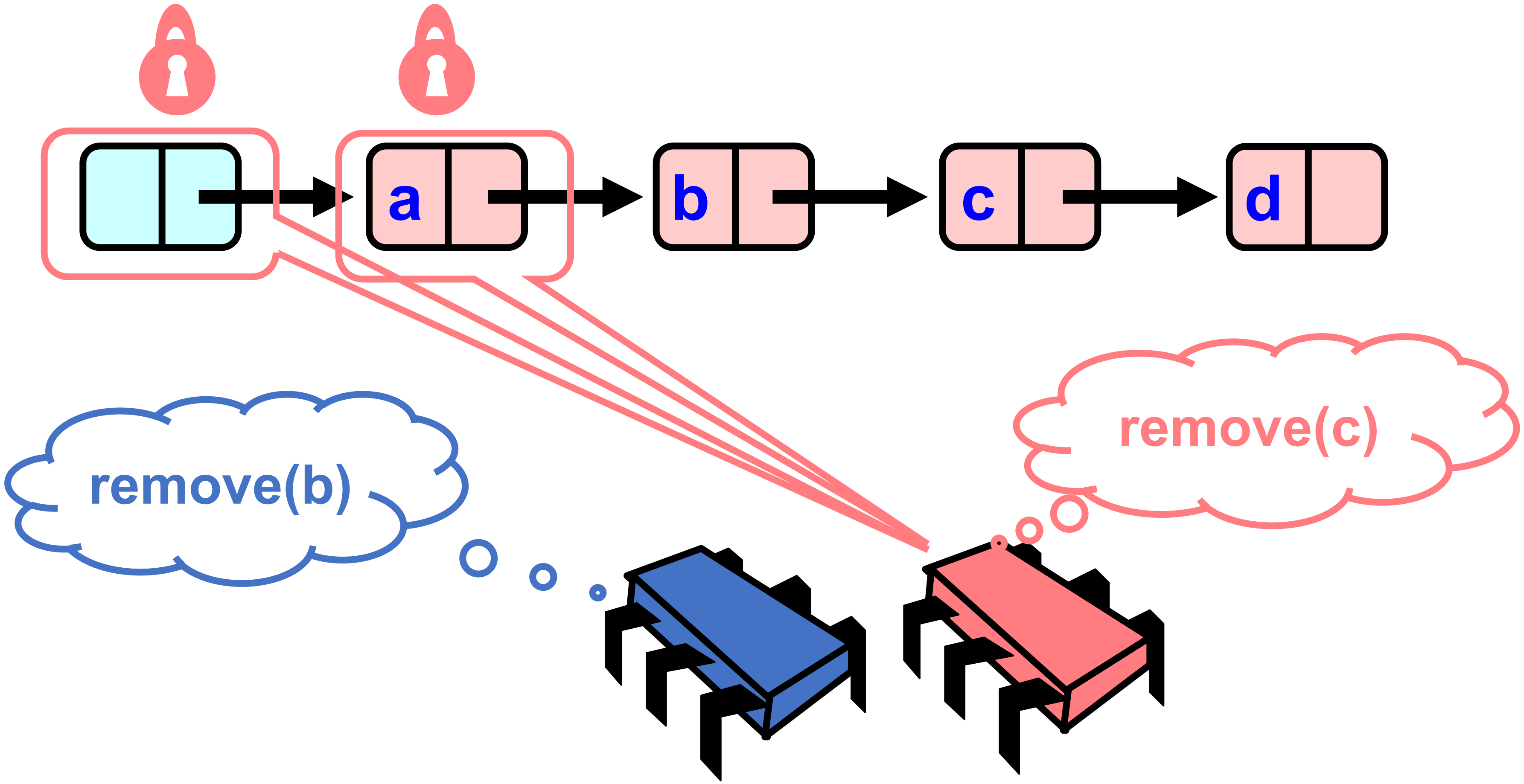
Removing a Node



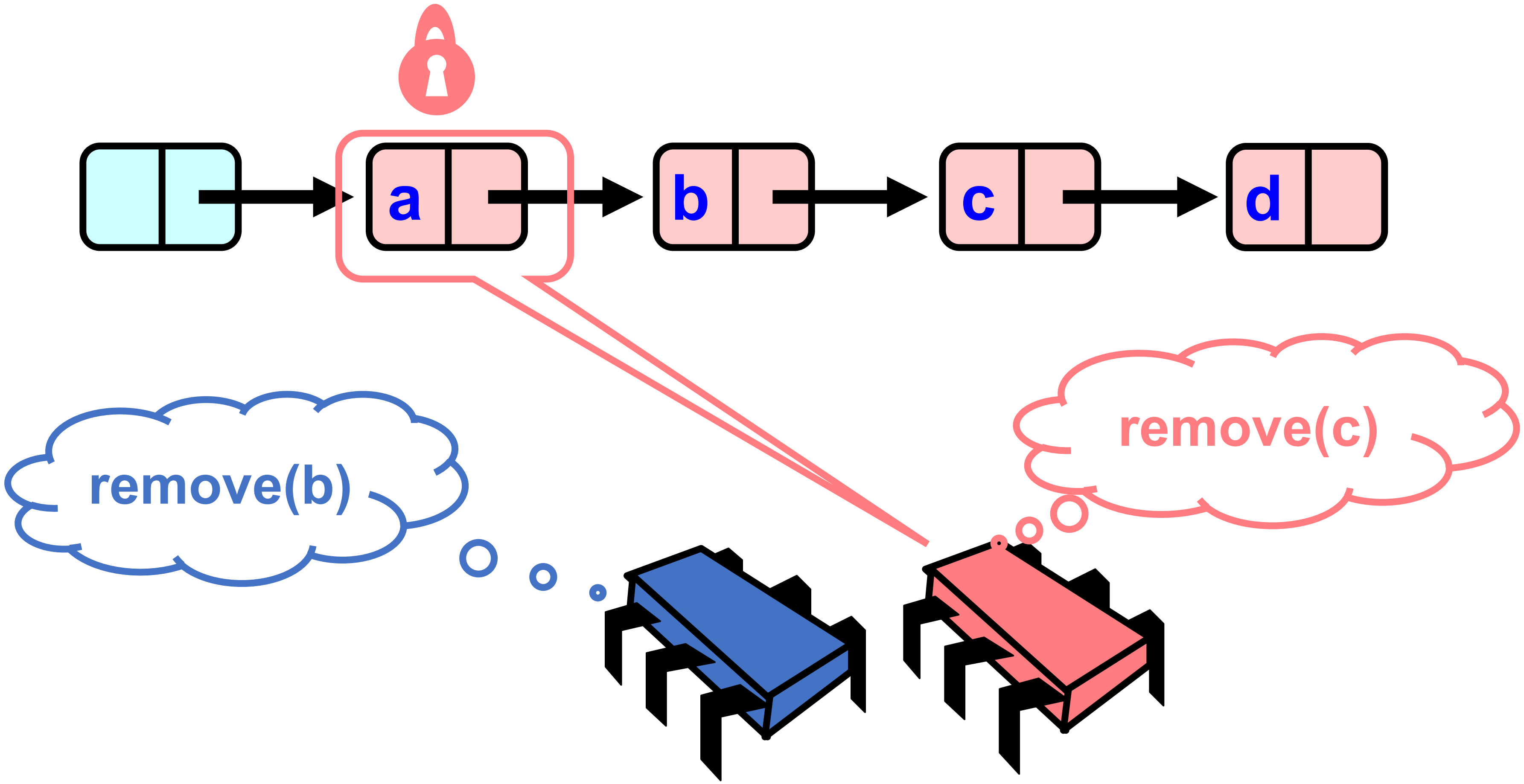
Removing a Node



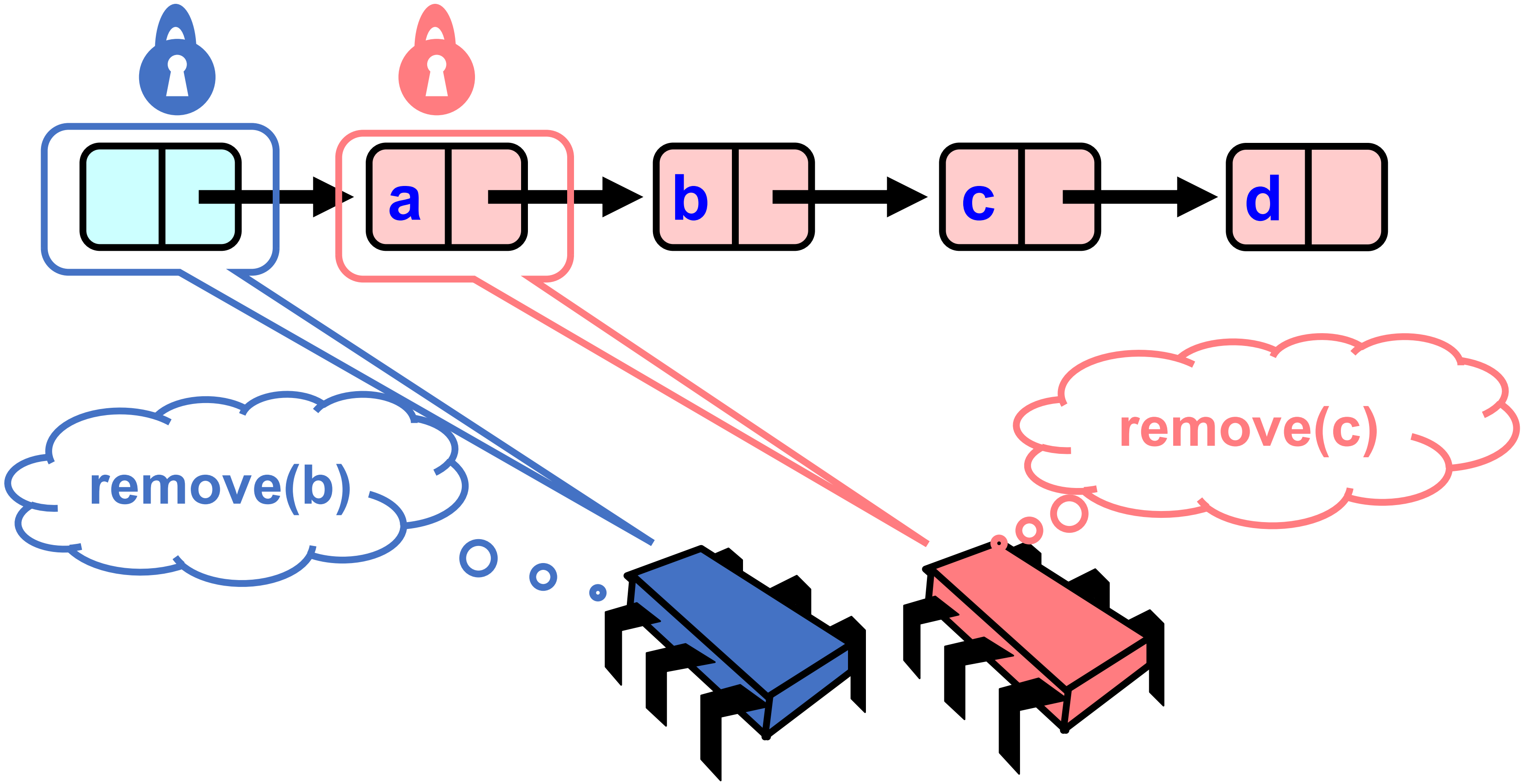
Removing a Node



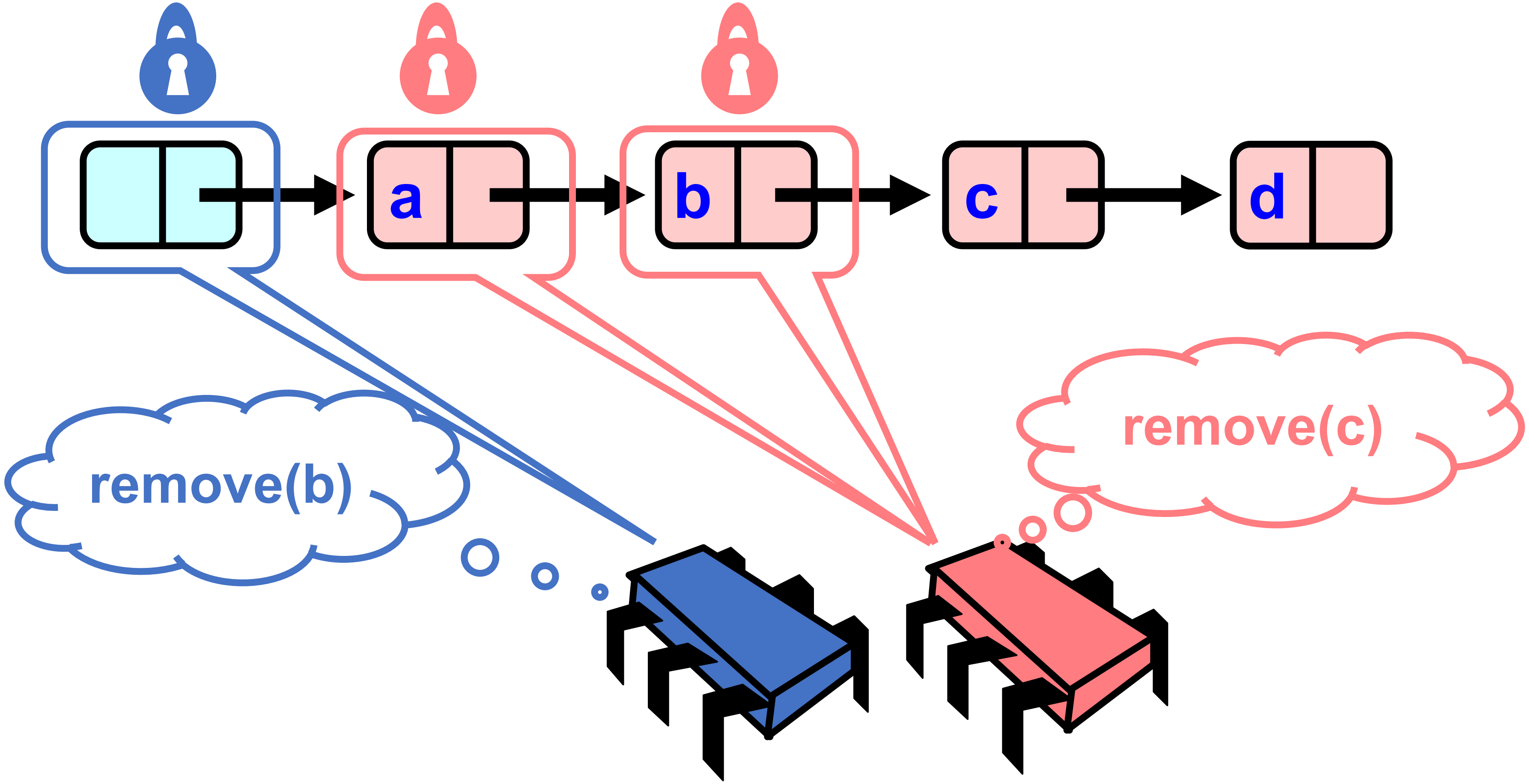
Removing a Node



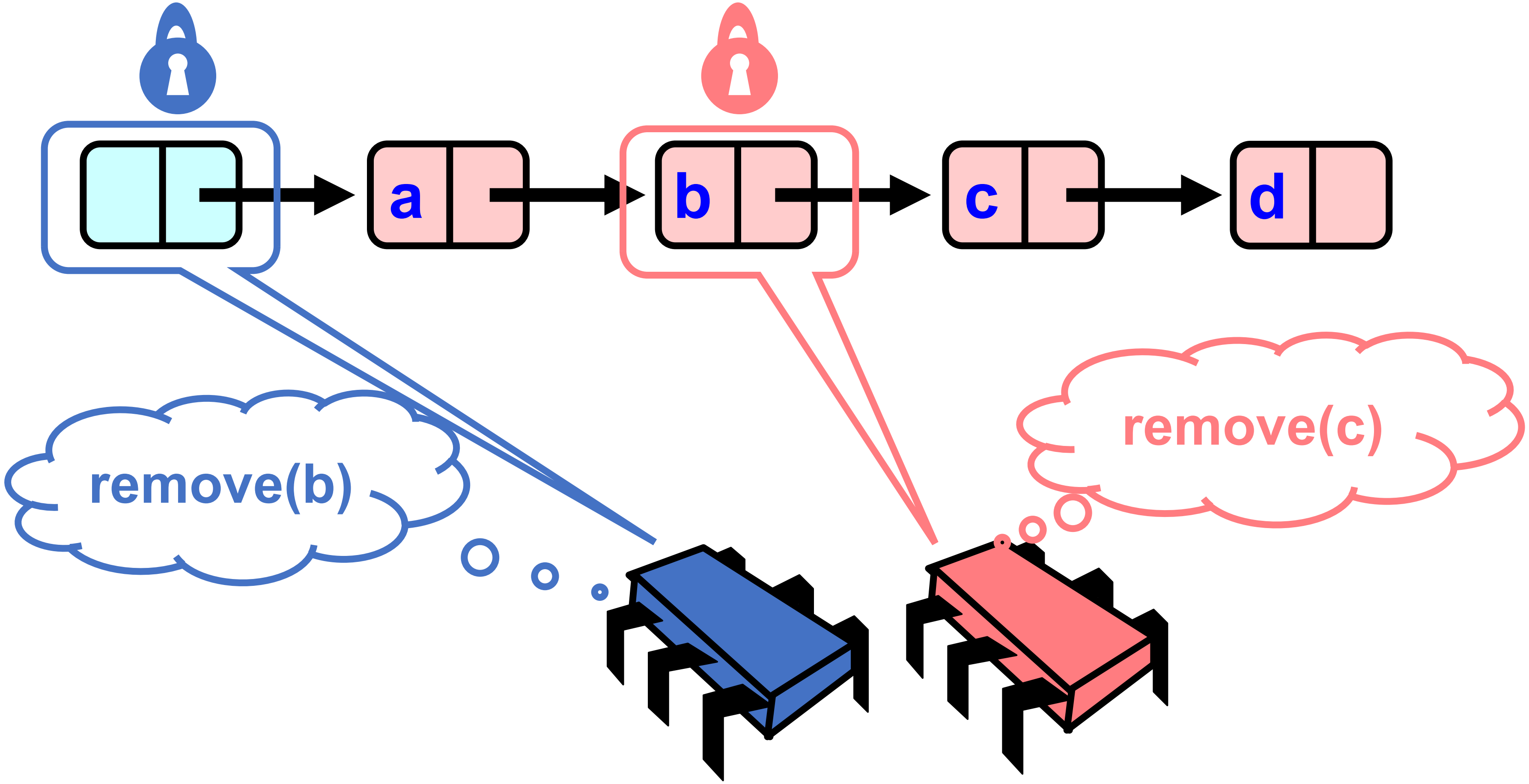
Removing a Node



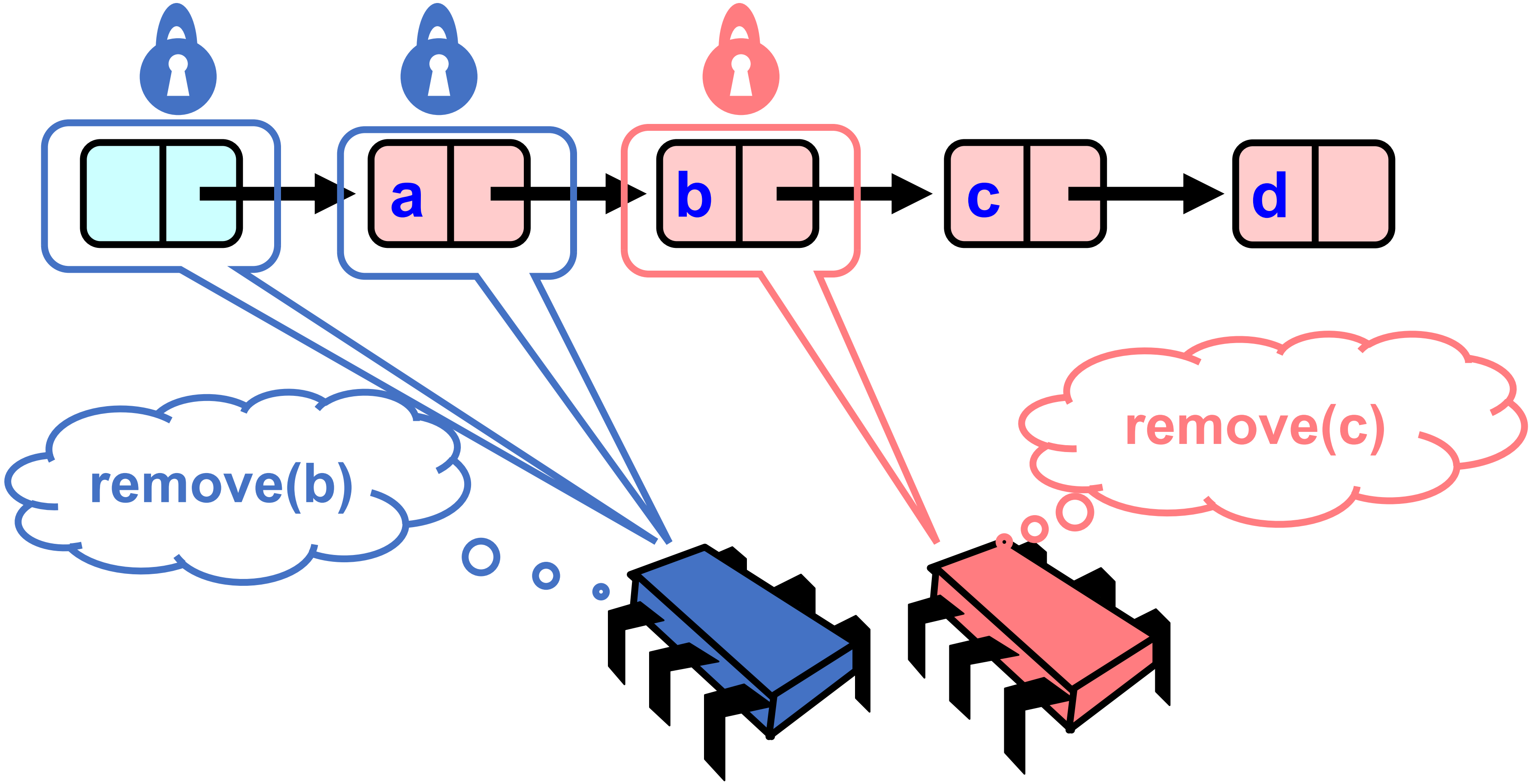
Removing a Node



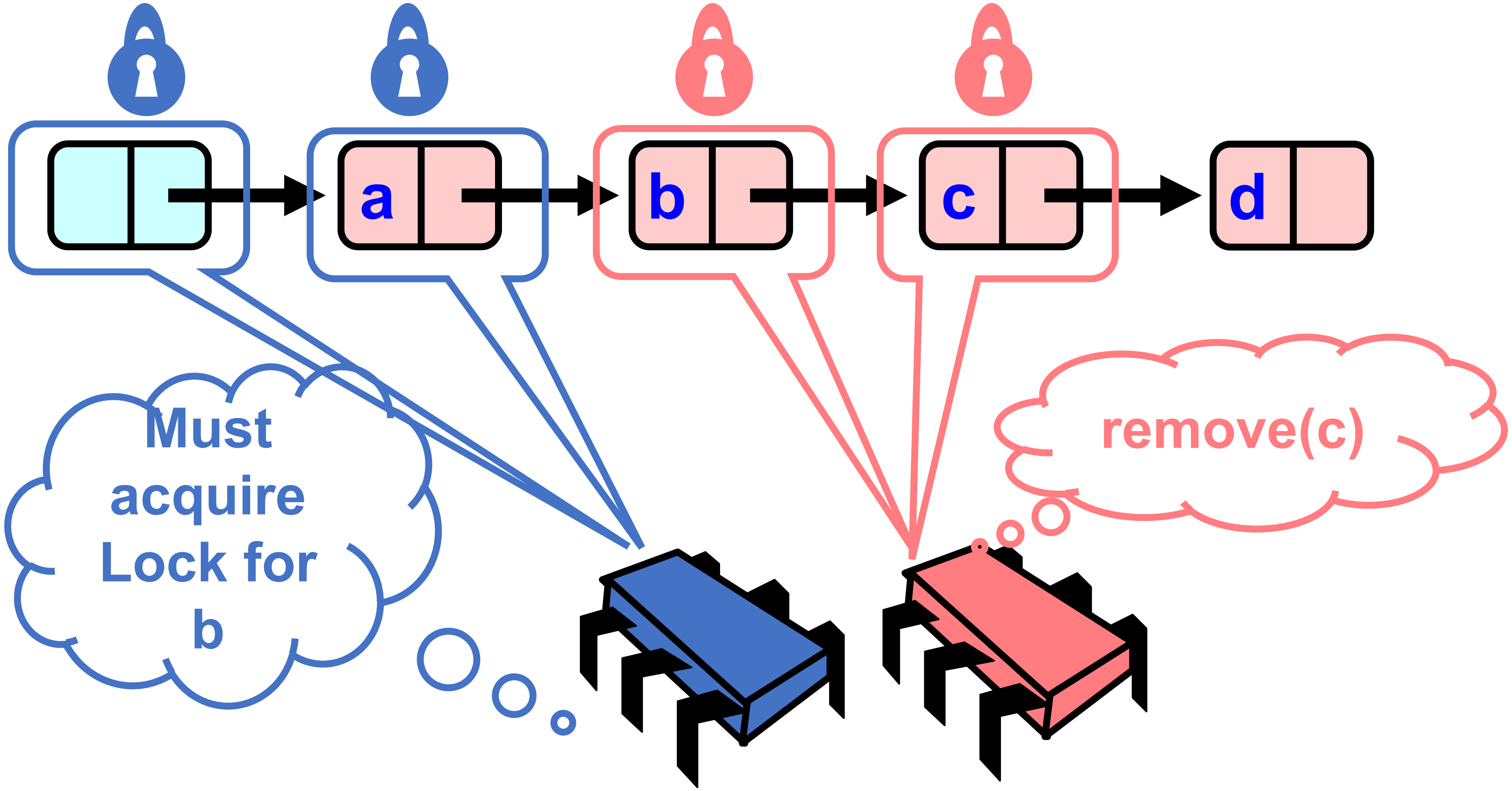
Removing a Node



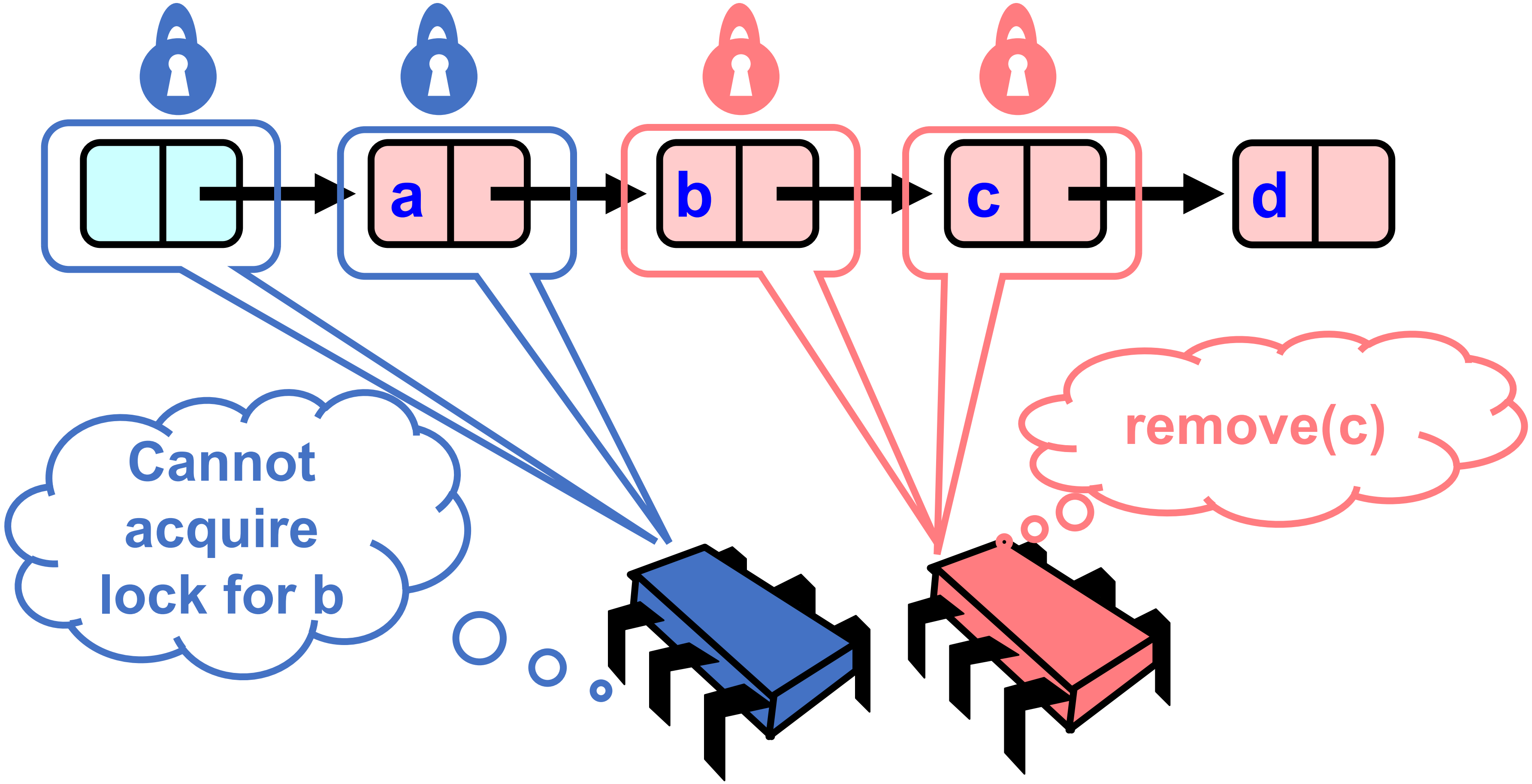
Removing a Node



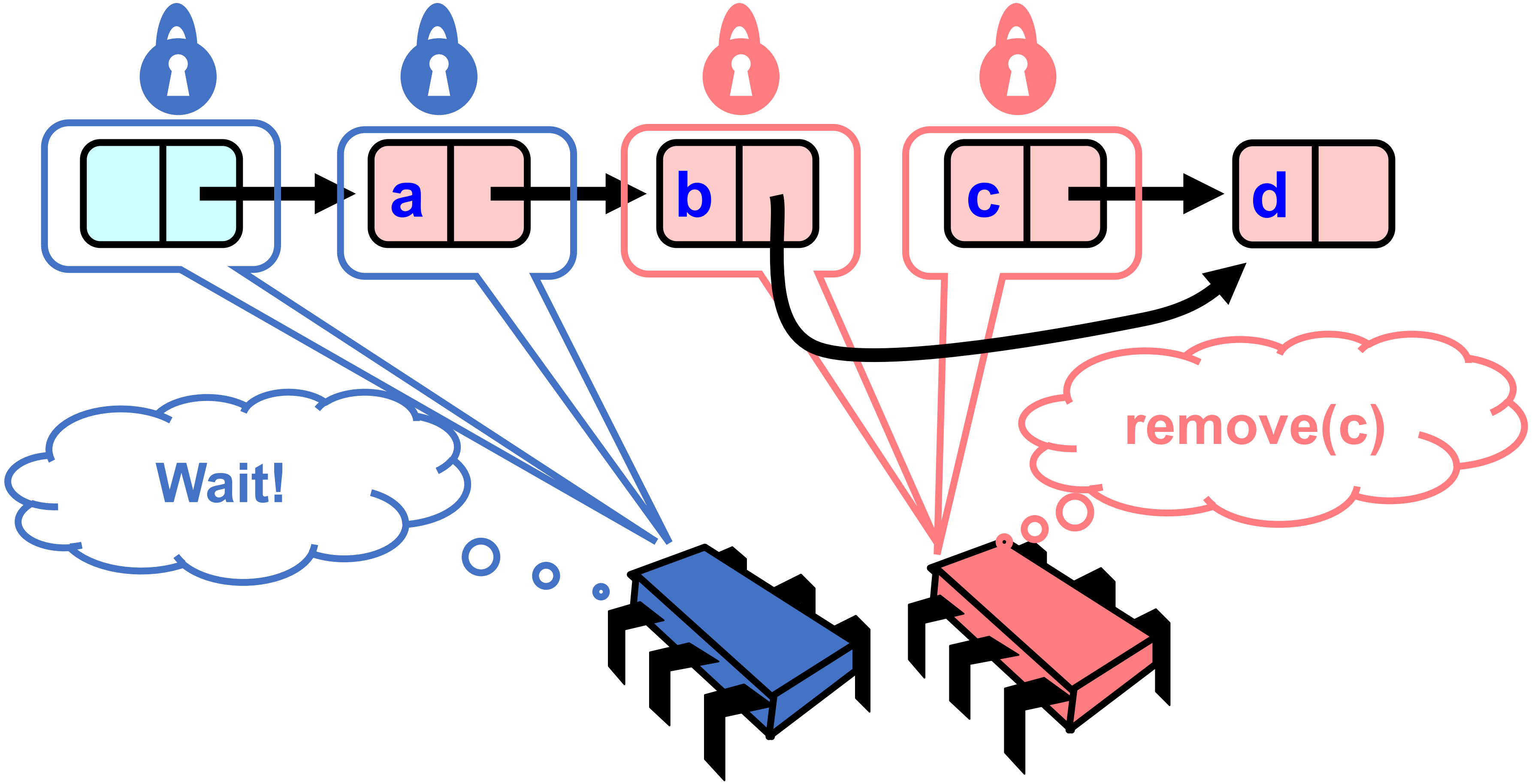
Removing a Node



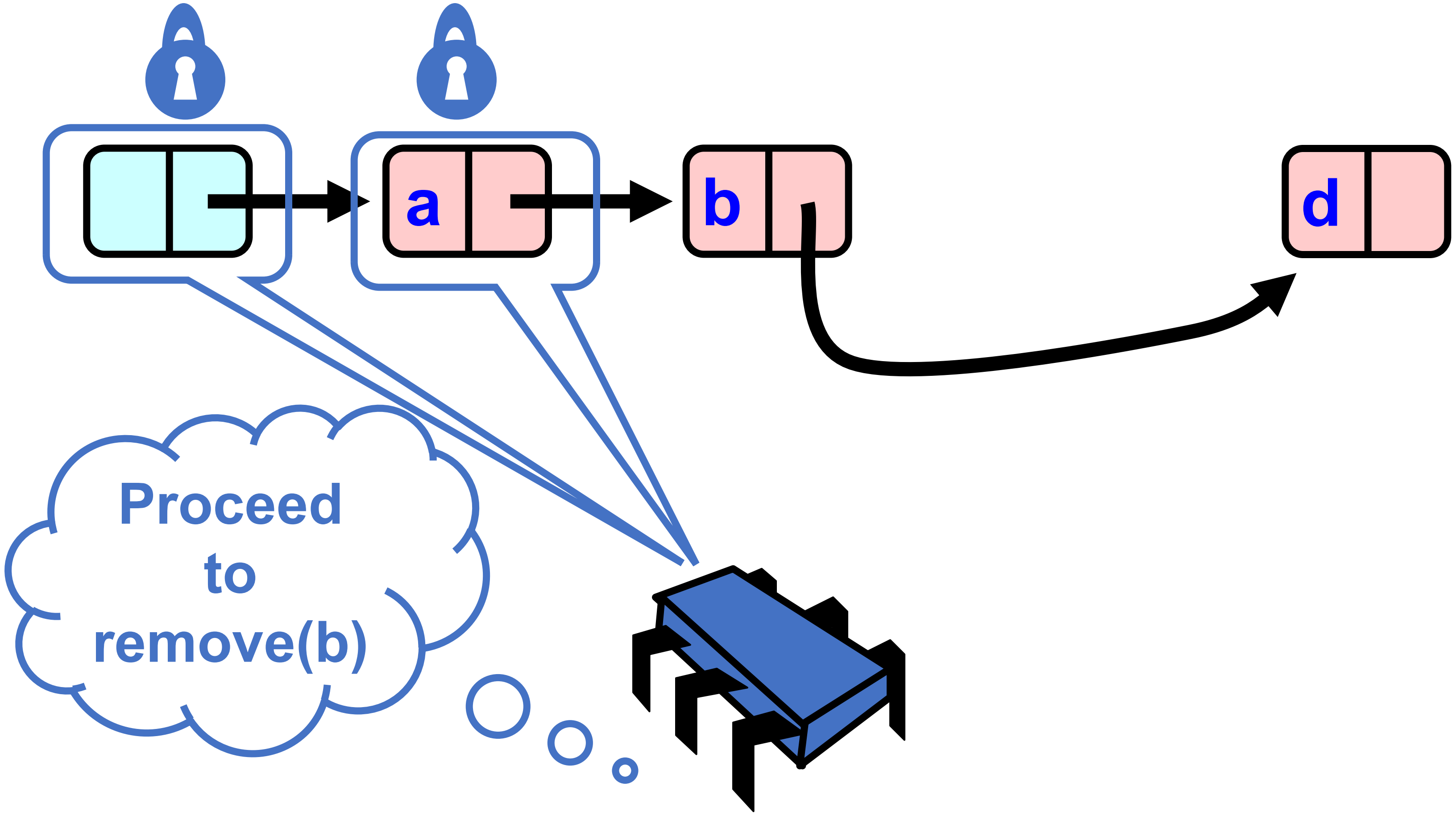
Removing a Node



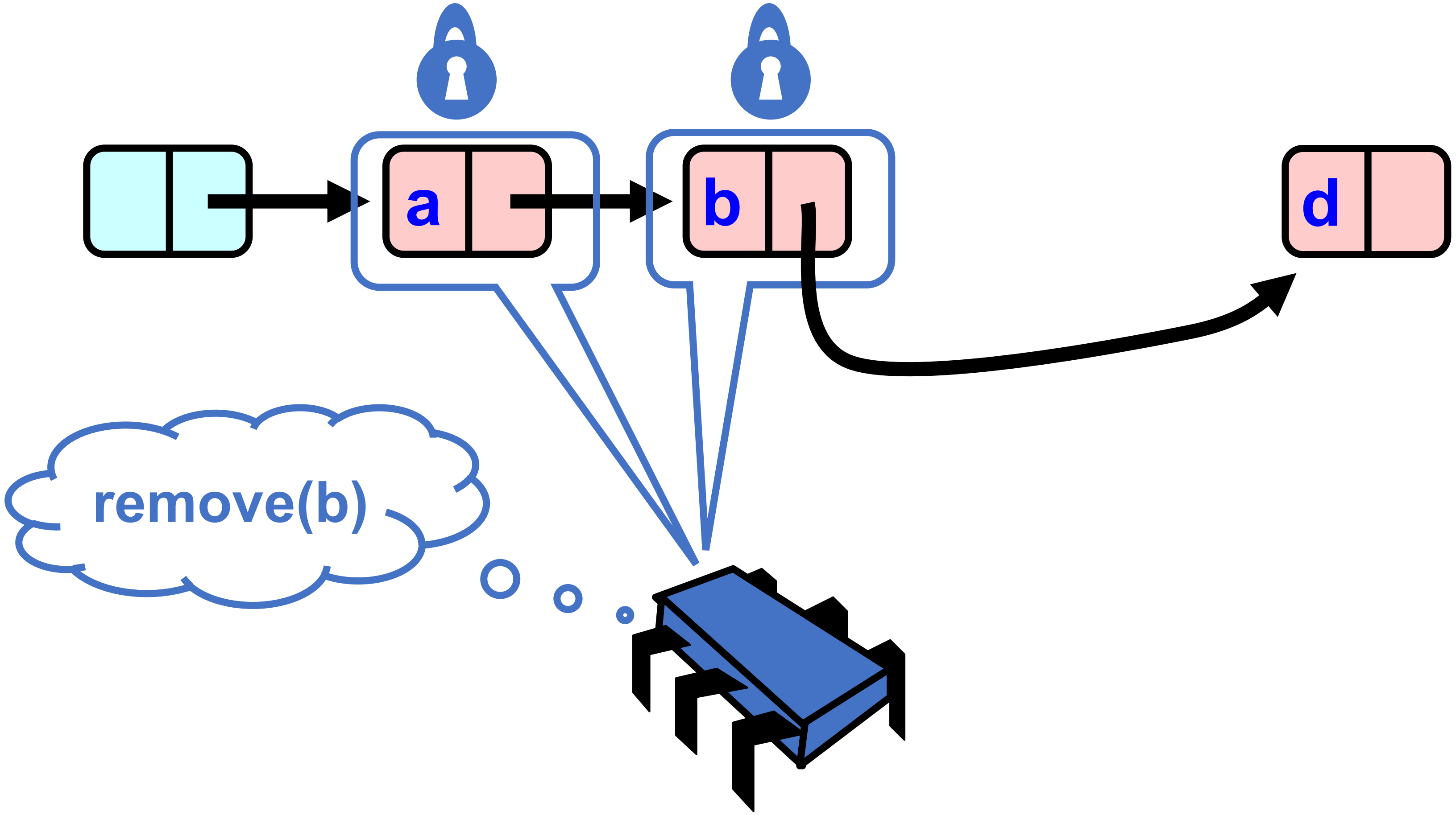
Removing a Node



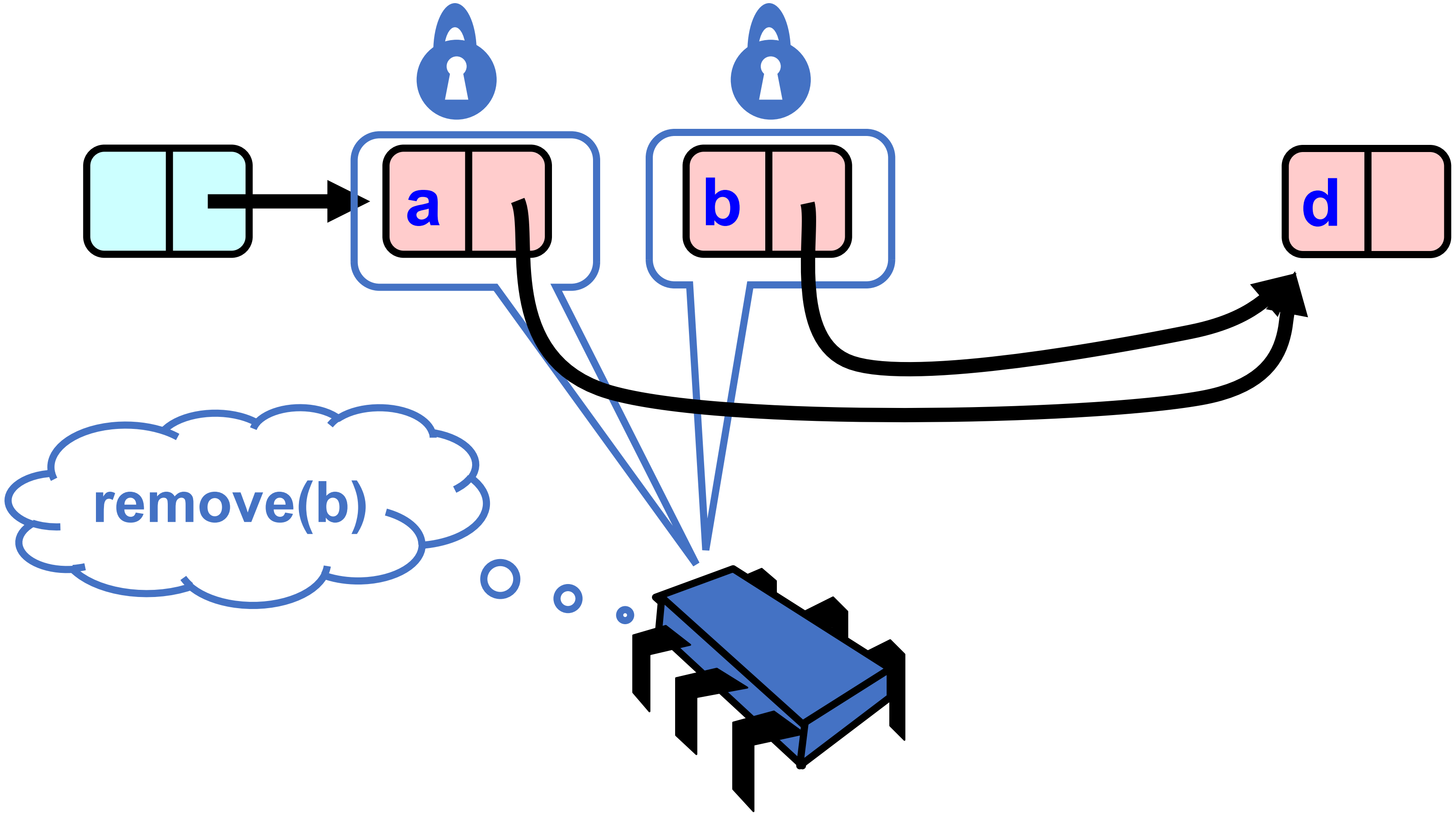
Removing a Node



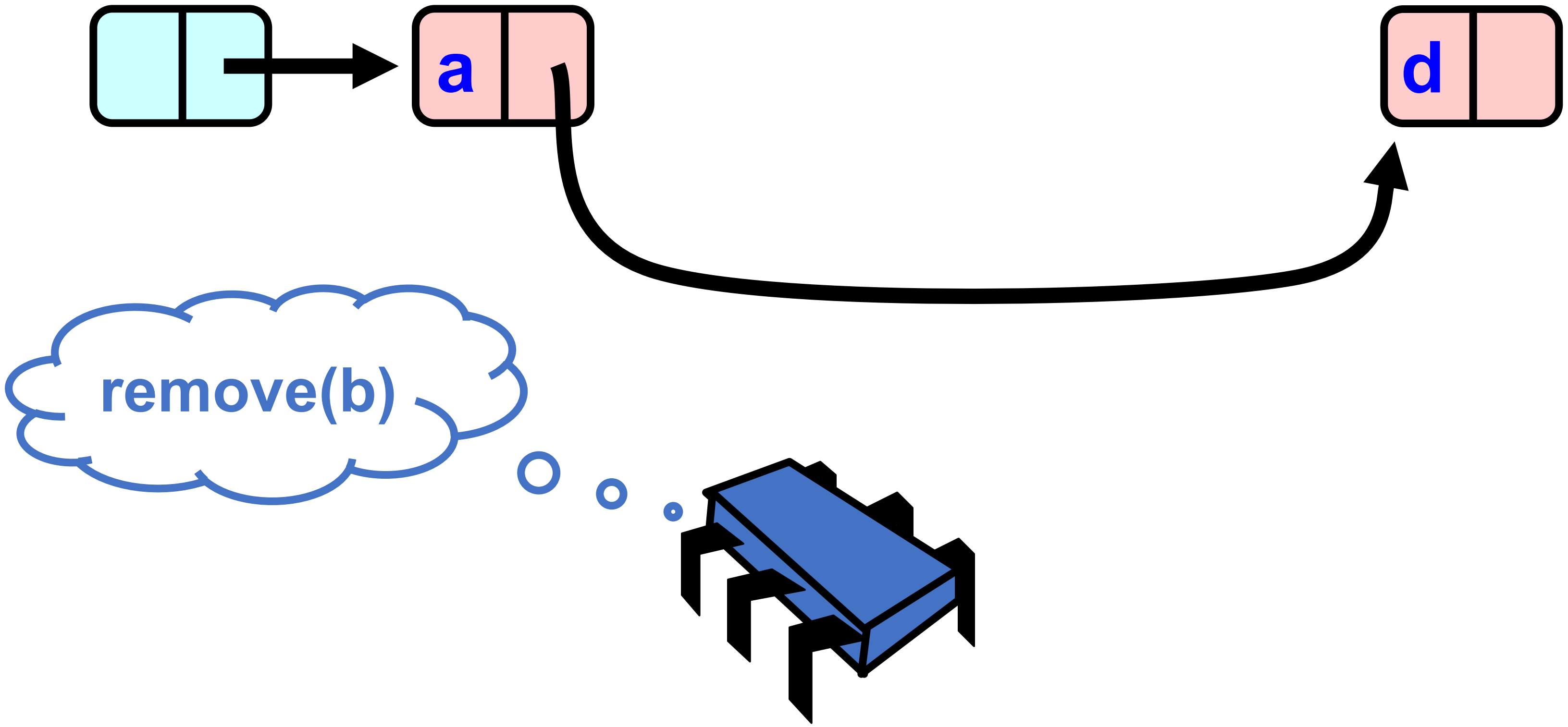
Removing a Node



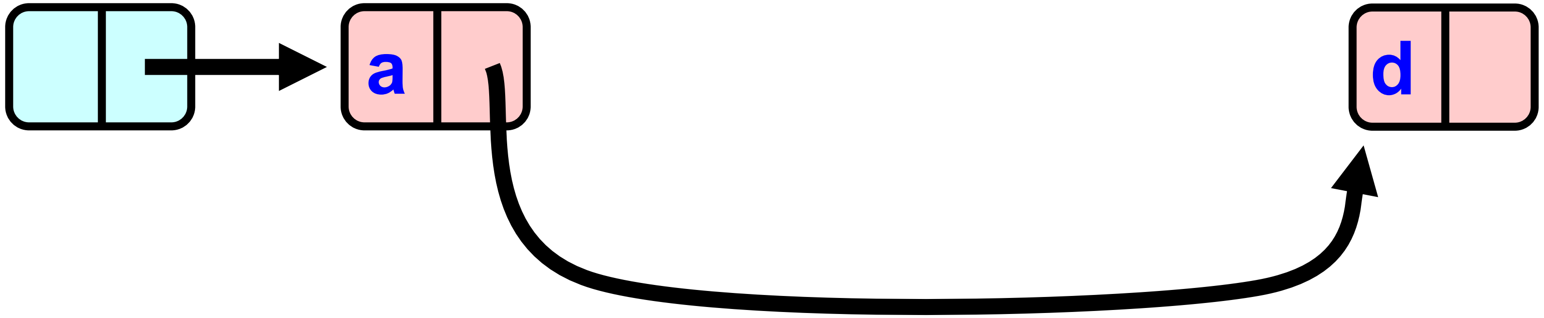
Removing a Node



Removing a Node



Removing a Node



Adding Nodes

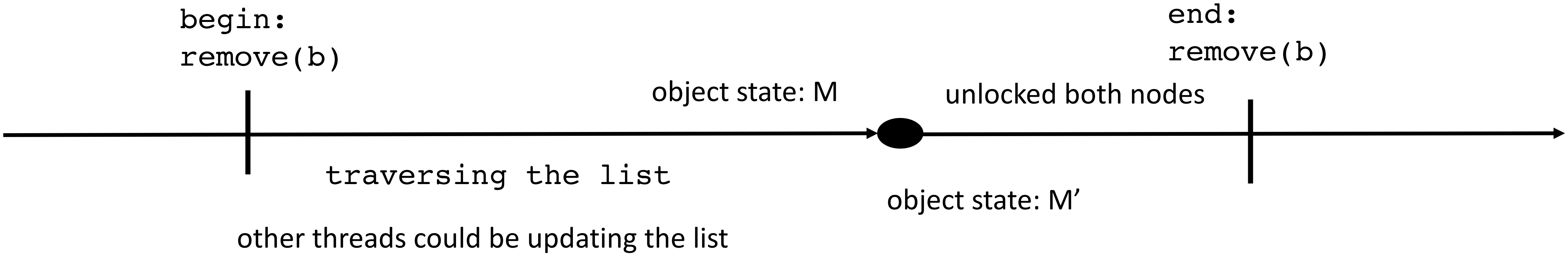
- To add node e
    - Must lock predecessor
    - Must lock successor
- Neither can be deleted
    - Is successor lock actually required?

Drawbacks

- Better than coarse-grained lock
  - Threads can traverse in parallel
- Still not ideal
  - Long chain of acquire/release
  - Inefficient

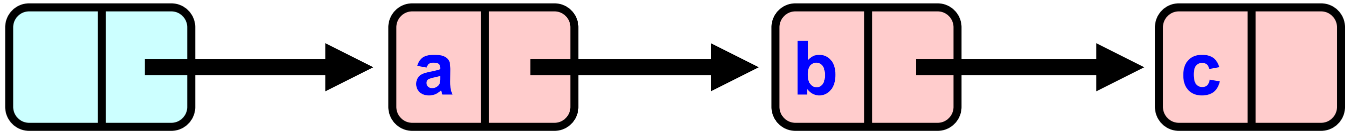
Linearizability point

- The double node critical section:
  - In parallel, other threads can update other parts of the list (ahead or behind)
  - But when we release the double locks, our update is complete

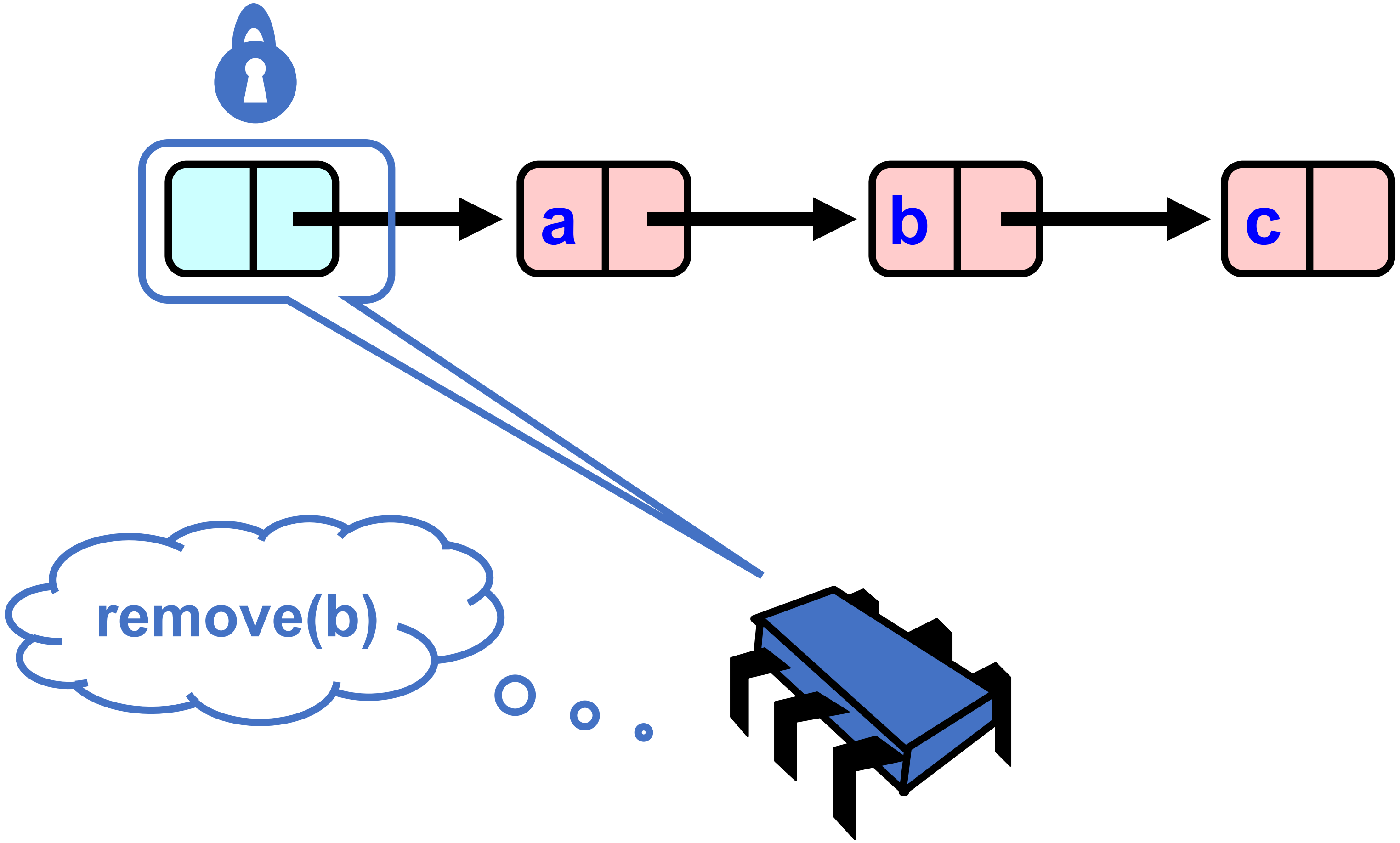


```
void remove(Value v) {
  Node* pred = NULL, *curr = NULL;
  head.lock();
  pred = head;
  curr = pred.next();
  curr.lock();
  while (curr.value != v) {
    pred.ulock();
    pred = curr;
    curr = curr.next();
    curr.lock();
  }
  pred.next = curr.next;
  curr.unlock();
  pred.unlock();
}
```

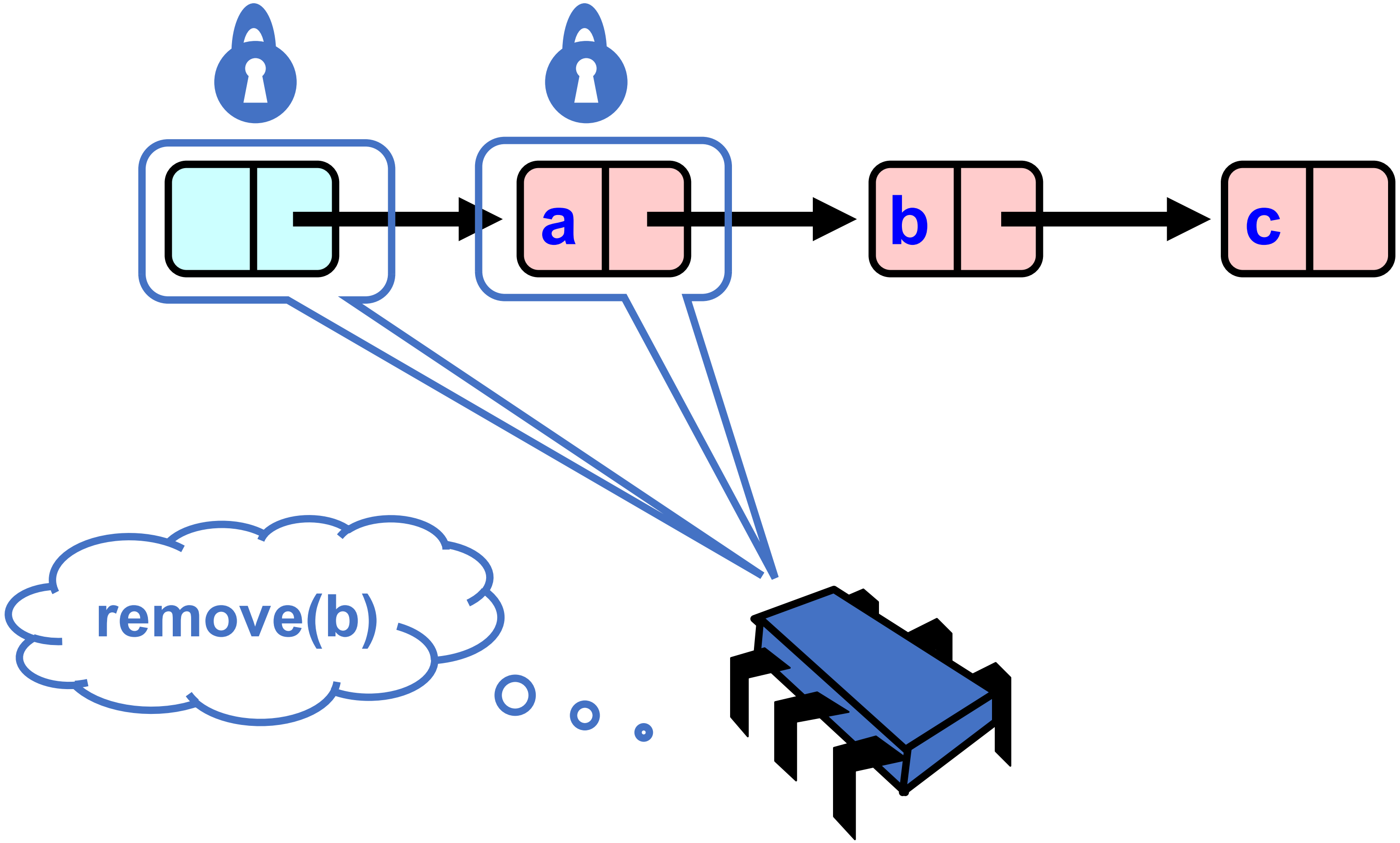
```
void remove(Value v) {
  Node* pred = NULL, *curr = NULL;
  head.lock();
  pred = head;
  curr = pred.next();
  curr.lock();
  while (curr.value != v) {
    pred.ulock();
    pred = curr;
    curr = curr.next();
    curr.lock();
  }
  pred.next = curr.next;
  curr.unlock();
  pred.unlock();
}
```



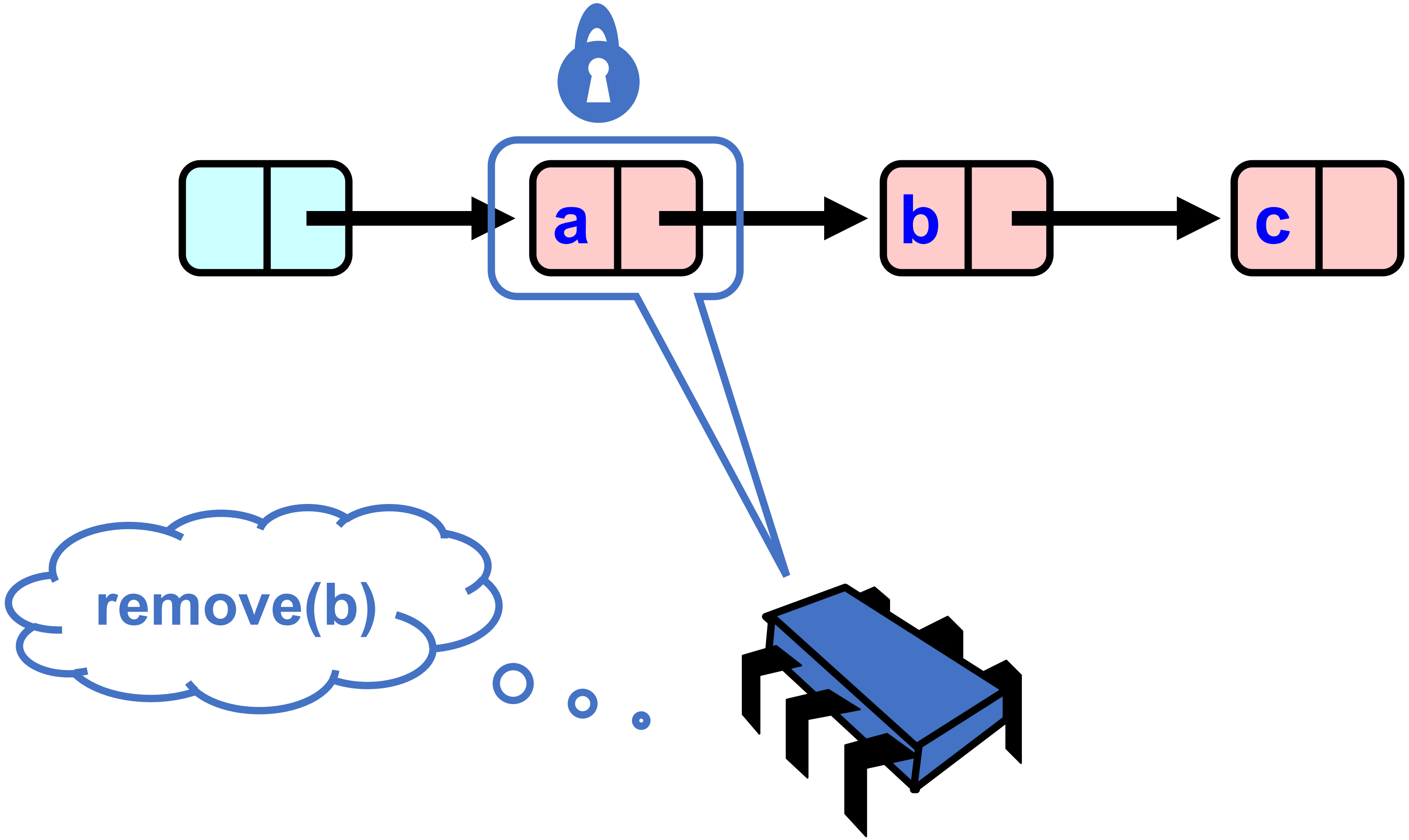
```
void remove(Value v) {
  Node* pred = NULL, *curr = NULL;
  head.lock();
  pred = head;
  curr = pred.next();
  curr.lock();
  while (curr.value != v) {
    pred.ulock();
    pred = curr;
    curr = curr.next();
    curr.lock();
  }
  pred.next = curr.next;
  curr.unlock();
  pred.unlock();
}
```



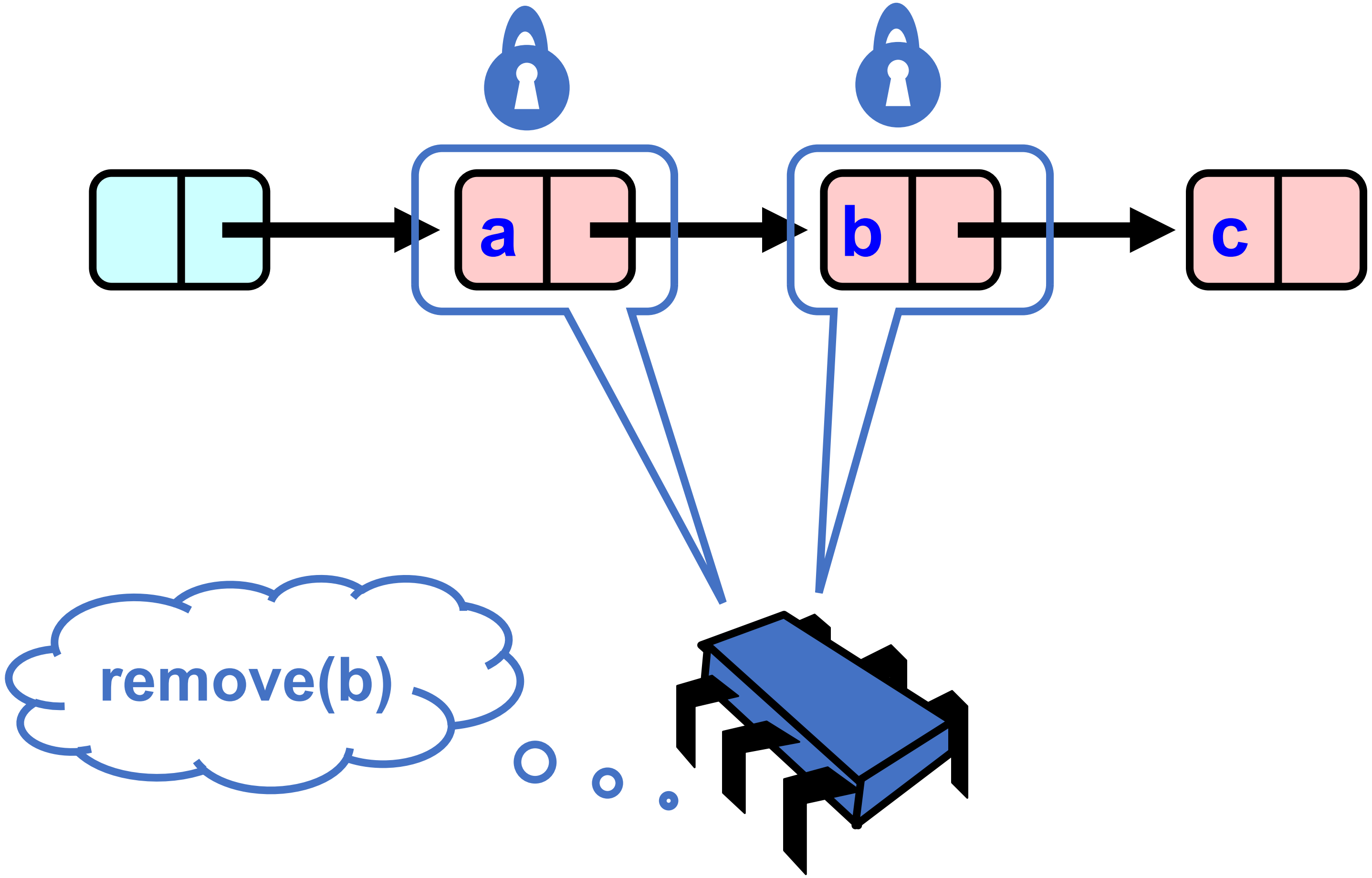
```
void remove(Value v) {
  Node* pred = NULL, *curr = NULL;
  head.lock();
  pred = head;
  curr = pred.next();
  curr.lock();
  while (curr.value != v) {
    pred.ulock();
    pred = curr;
    curr = curr.next();
    curr.lock();
  }
  pred.next = curr.next;
  curr.unlock();
  pred.unlock();
}
```



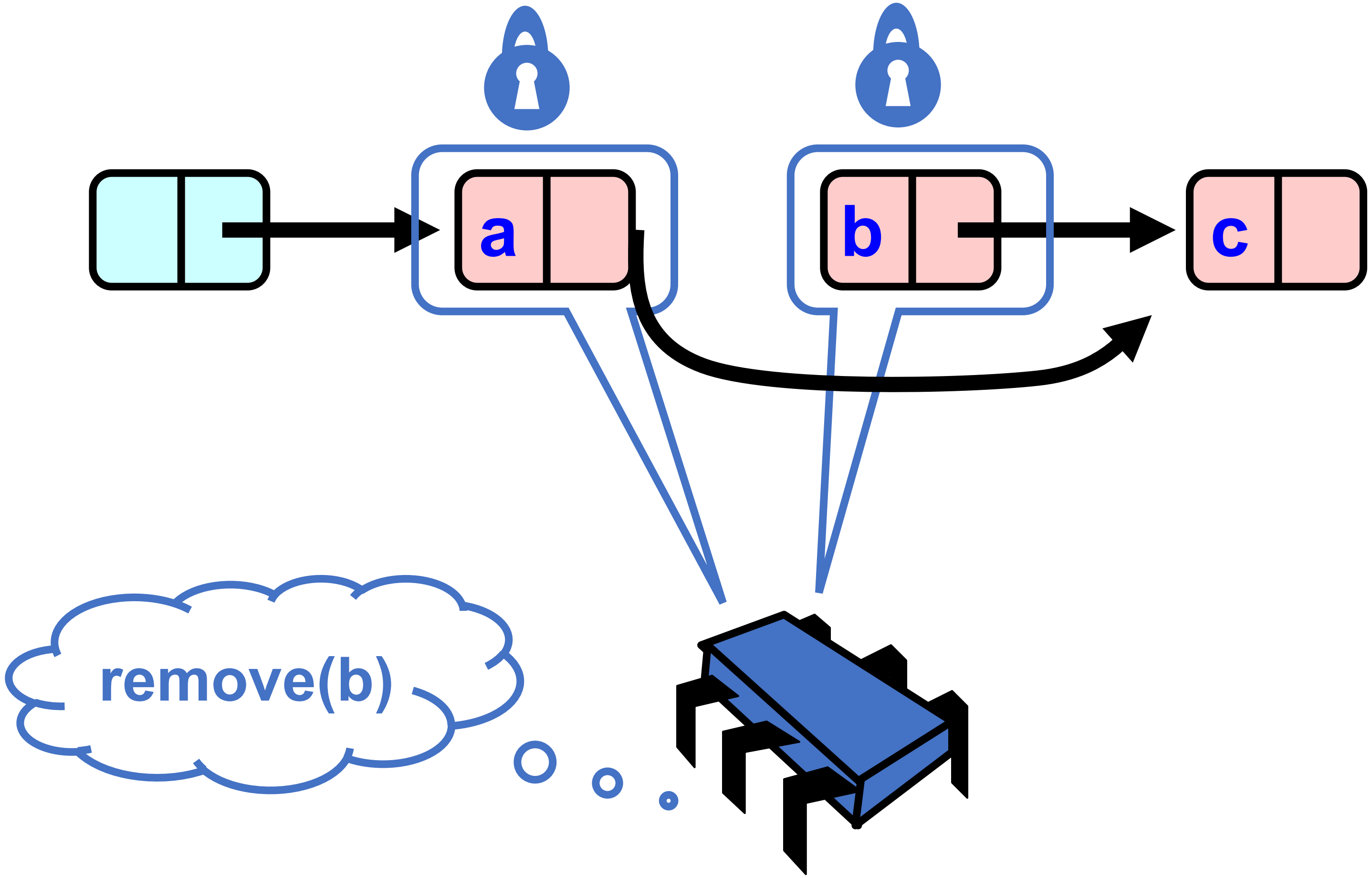
```
void remove(Value v) {
  Node* pred = NULL, *curr = NULL;
  head.lock();
  pred = head;
  curr = pred.next();
  curr.lock();
  while (curr.value != v) {
    pred.ulock();
    pred = curr;
    curr = curr.next();
    curr.lock();
  }
  pred.next = curr.next;
  curr.unlock();
  pred.unlock();
}
```



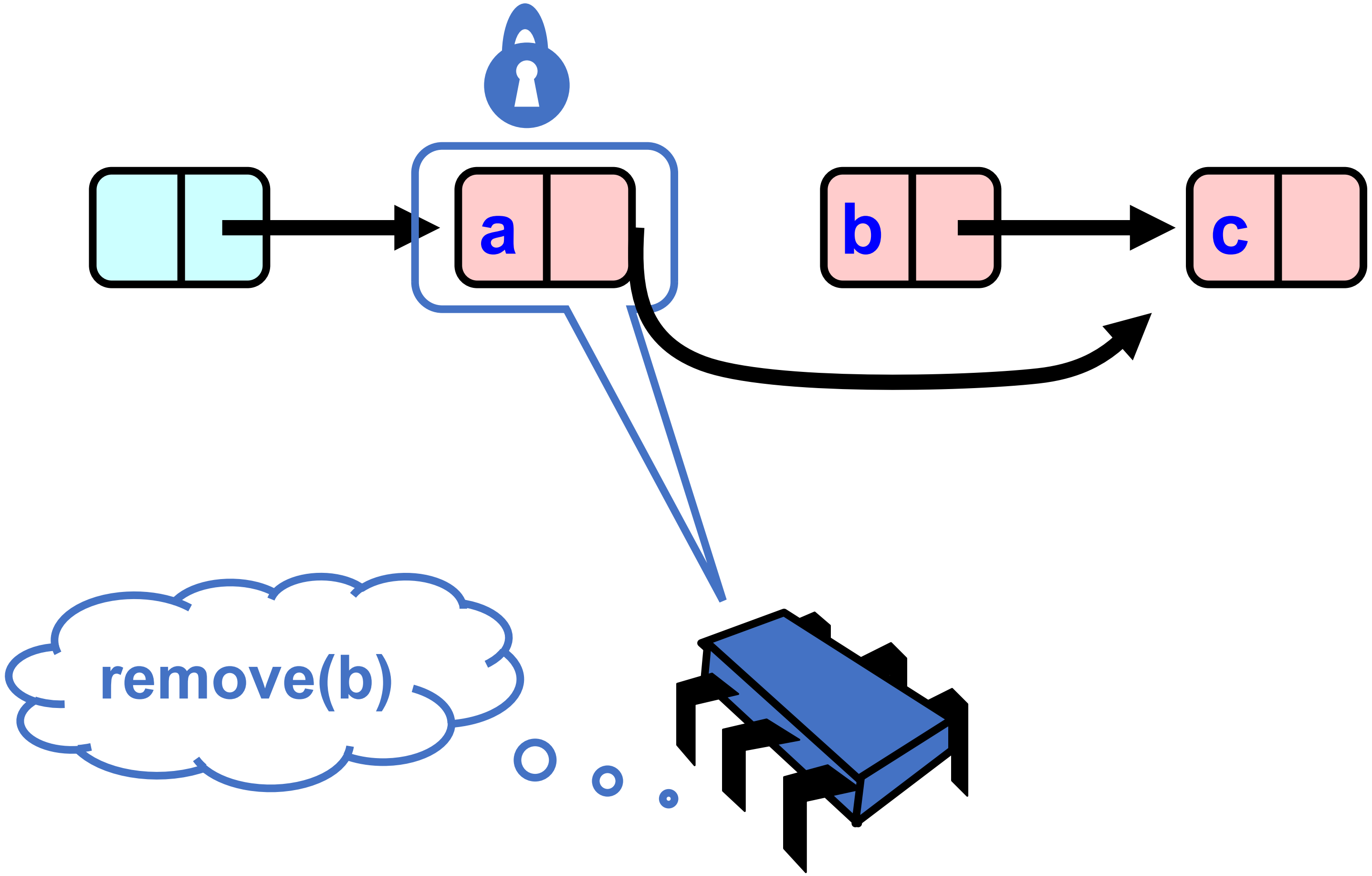
```
void remove(Value v) {
  Node* pred = NULL, *curr = NULL;
  head.lock();
  pred = head;
  curr = pred.next();
  curr.lock();
  while (curr.value != v) {
    pred.ulock();
    pred = curr;
    curr = curr.next();
    curr.lock();
  }
  pred.next = curr.next;
  curr.unlock();
  pred.unlock();
}
```



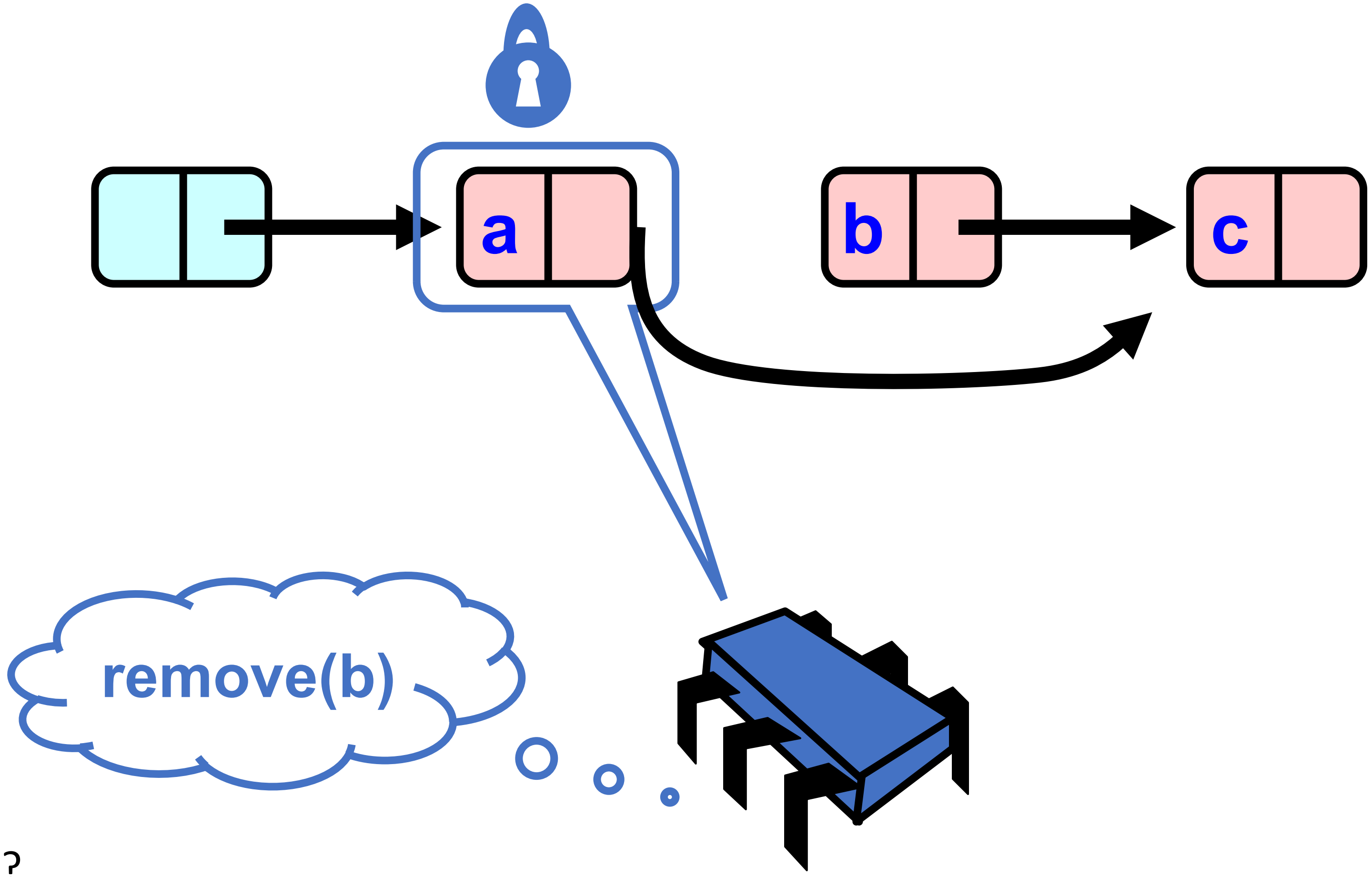
```
void remove(Value v) {
  Node* pred = NULL, *curr = NULL;
  head.lock();
  pred = head;
  curr = pred.next();
  curr.lock();
  while (curr.value != v) {
    pred.ulock();
    pred = curr;
    curr = curr.next();
    curr.lock();
  }
  pred.next = curr.next;
  curr.unlock();
  pred.unlock();
}
```



```
void remove(Value v) {
  Node* pred = NULL, *curr = NULL;
  head.lock();
  pred = head;
  curr = pred.next();
  curr.lock();
  while (curr.value != v) {
    pred.ulock();
    pred = curr;
    curr = curr.next();
    curr.lock();
  }
  pred.next = curr.next;
  curr.unlock();
  pred.unlock();
}
```

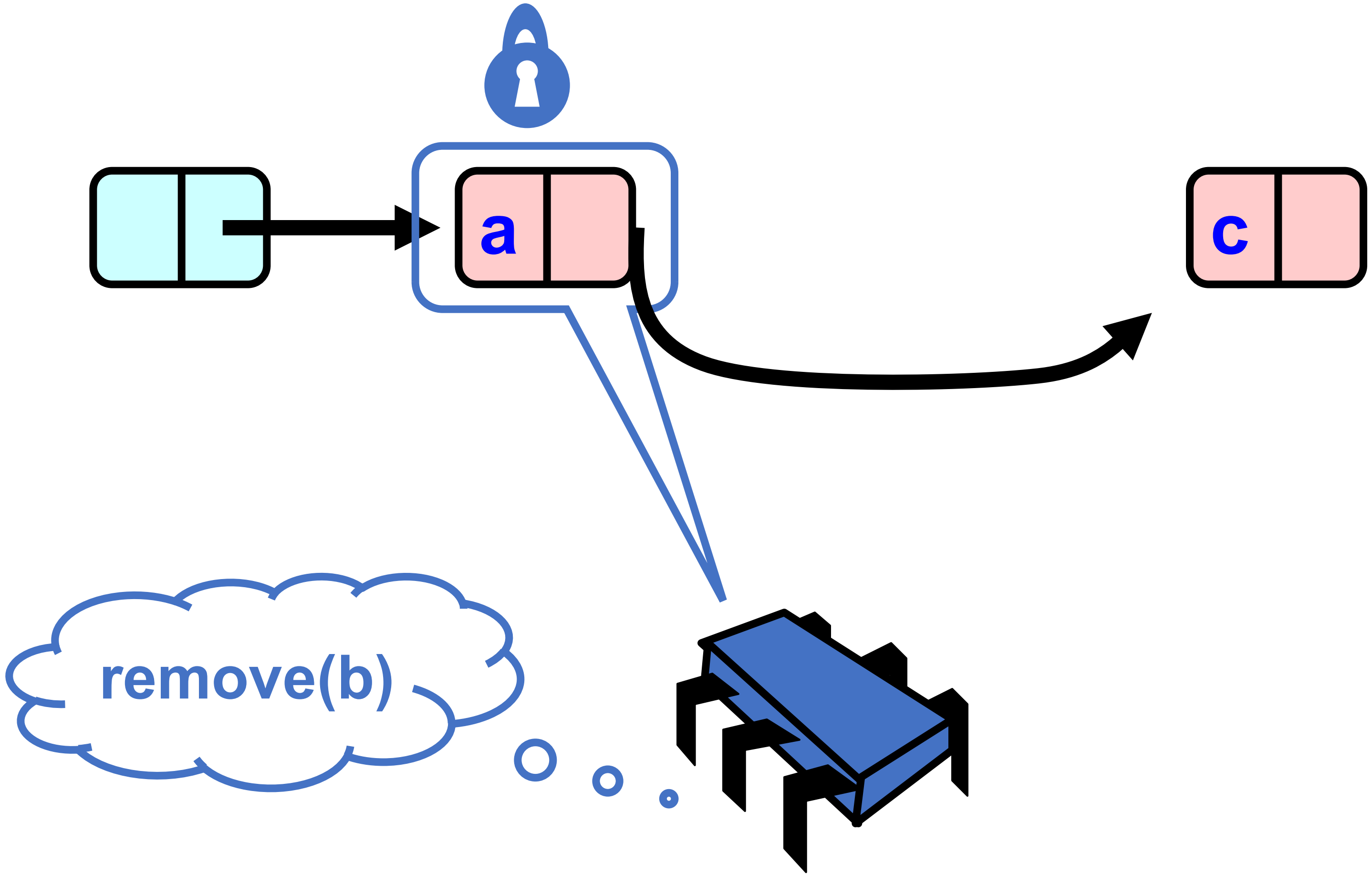


```
void remove(Value v) {
  Node* pred = NULL, *curr = NULL;
  head.lock();
  pred = head;
  curr = pred.next();
  curr.lock();
  while (curr.value != v) {
    pred.ulock();
    pred = curr;
    curr = curr.next();
    curr.lock();
  }
  pred.next = curr.next;
  curr.unlock();
  pred.unlock();
}
```

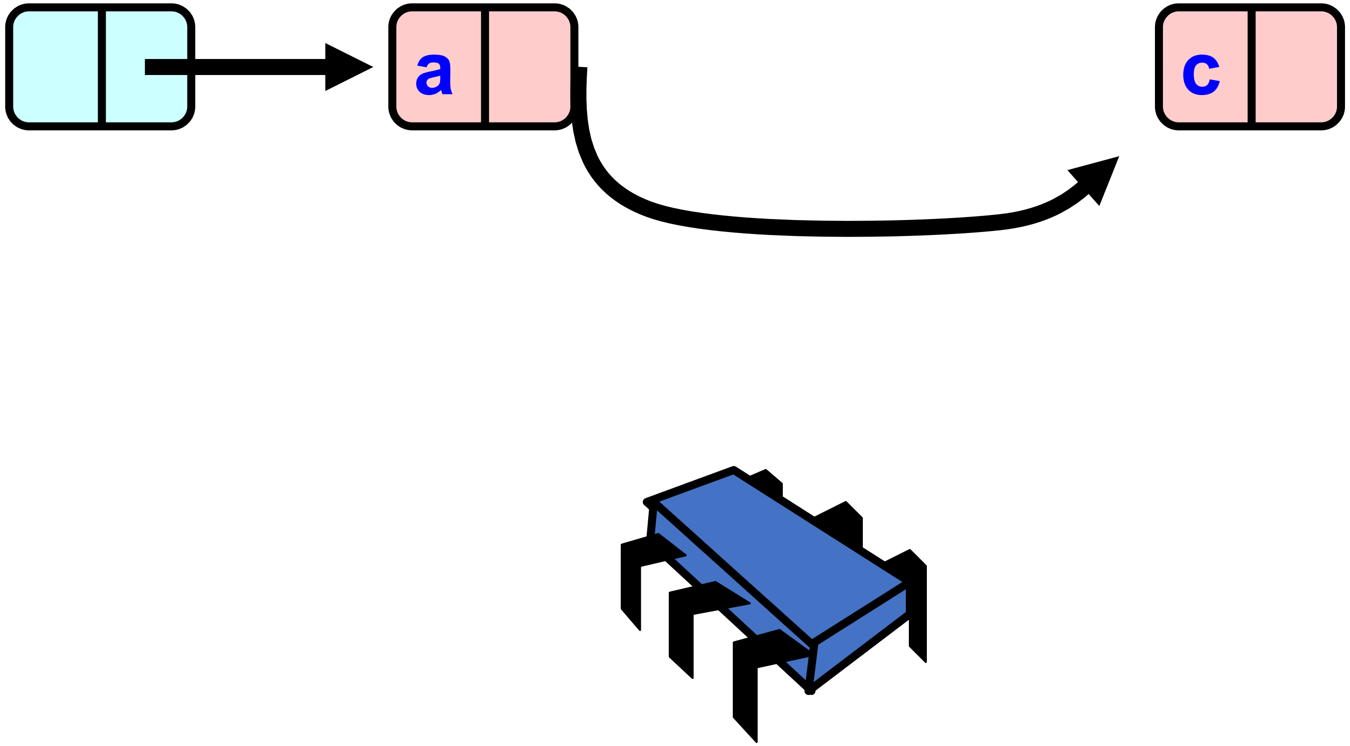


What are we missing?

```
void remove(Value v) {
  Node* pred = NULL, *curr = NULL;
  head.lock();
  pred = head;
  curr = pred.next();
  curr.lock();
  while (curr.value != v) {
    pred.ulock();
    pred = curr;
    curr = curr.next();
    curr.lock();
  }
  pred.next = curr.next;
  curr.unlock();
  pred.unlock();
}
```



```
void remove(Value v) {
  Node* pred = NULL, *curr = NULL;
  head.lock();
  pred = head;
  curr = pred.next();
  curr.lock();
  while (curr.value != v) {
    pred.ulock();
    pred = curr;
    curr = curr.next();
    curr.lock();
  }
  pred.next = curr.next;
  curr.unlock();
  pred.unlock();
}
```



Next week

- Reduce the locking even more!
- We will make the list completely lock free!
- Concurrent Queues
  - ABA problem
  - Specialized Queues