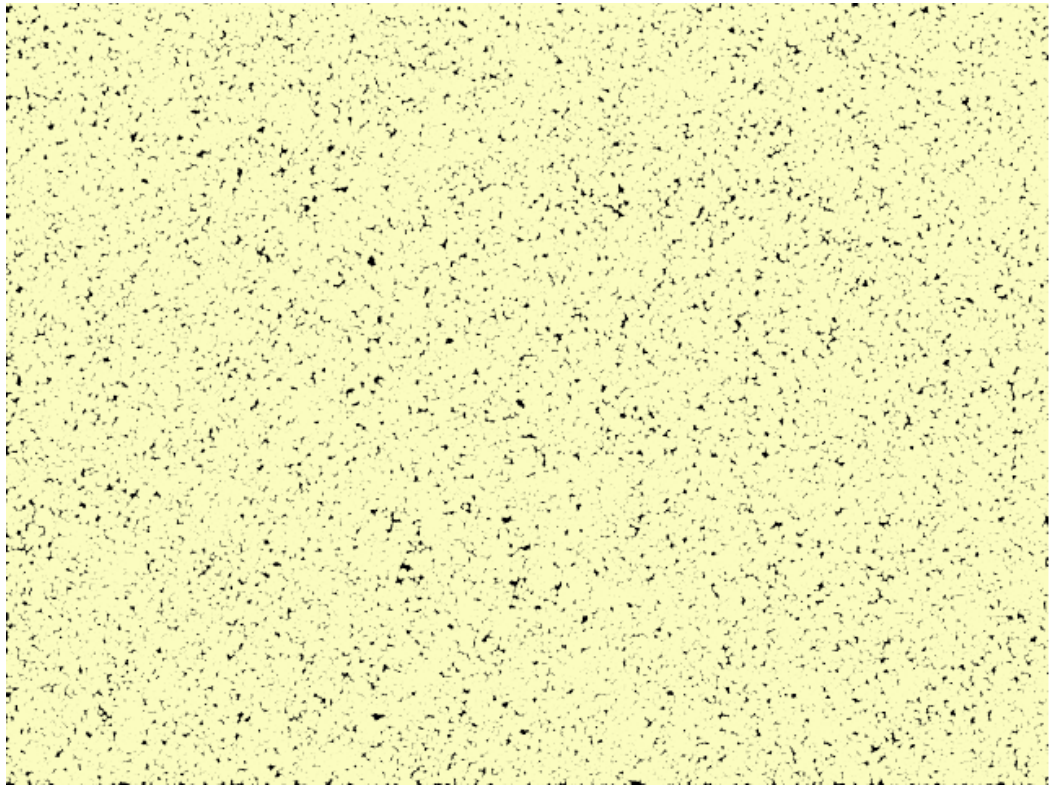
CSE113: Parallel Programming

March 1, 2023

- **Topics**:
  - Intro to module 4
  - Barriers



Announcements

- Midterm grades are out
  - Let us know within 1 week if there are issues
- We are working on HW 2 grades and will strive to get them released by Monday
- Homework 3 is due today (last late day)
- We are hoping to get HW 4 assigned today
  - There are some issues with the server we are trying to work out

Previous quiz

When a CAS operation fails in a lock-free linked list, the implementation should:

- ◯ spin on the same CAS (like a mutex), it will eventually succeed
- ◯ throw an exception
- ◯ retry the operation from the start, even if it means traversing the list again
- ◯ The CAS operation will never fail because it is an atomic operation

Previous quiz

An object wrapped in the C++ atomic template allows you to:

Check all that apply

- [ ] atomically execute method calls
- [ ] load and store the object atomically
- [ ] perform atomic CAS on the object
- [ ] locks each method call automatically

Previous quiz

Which are possible ways that hardware implements RMW operations? check all that apply:

- [ ] special loads and stores that track if there has been a conflict
- [ ] sleeping all other threads while one thread accesses the location
- [ ] having the hardware lock the cache line
- [ ] flushing the pipeline to avoid ILP interleavings

Previous quiz

It is the end of module 3: concurrent data structures:

Feel free to write a few sentences about how you found the module, e.g. what you found most interesting, what you found unclear, etc.

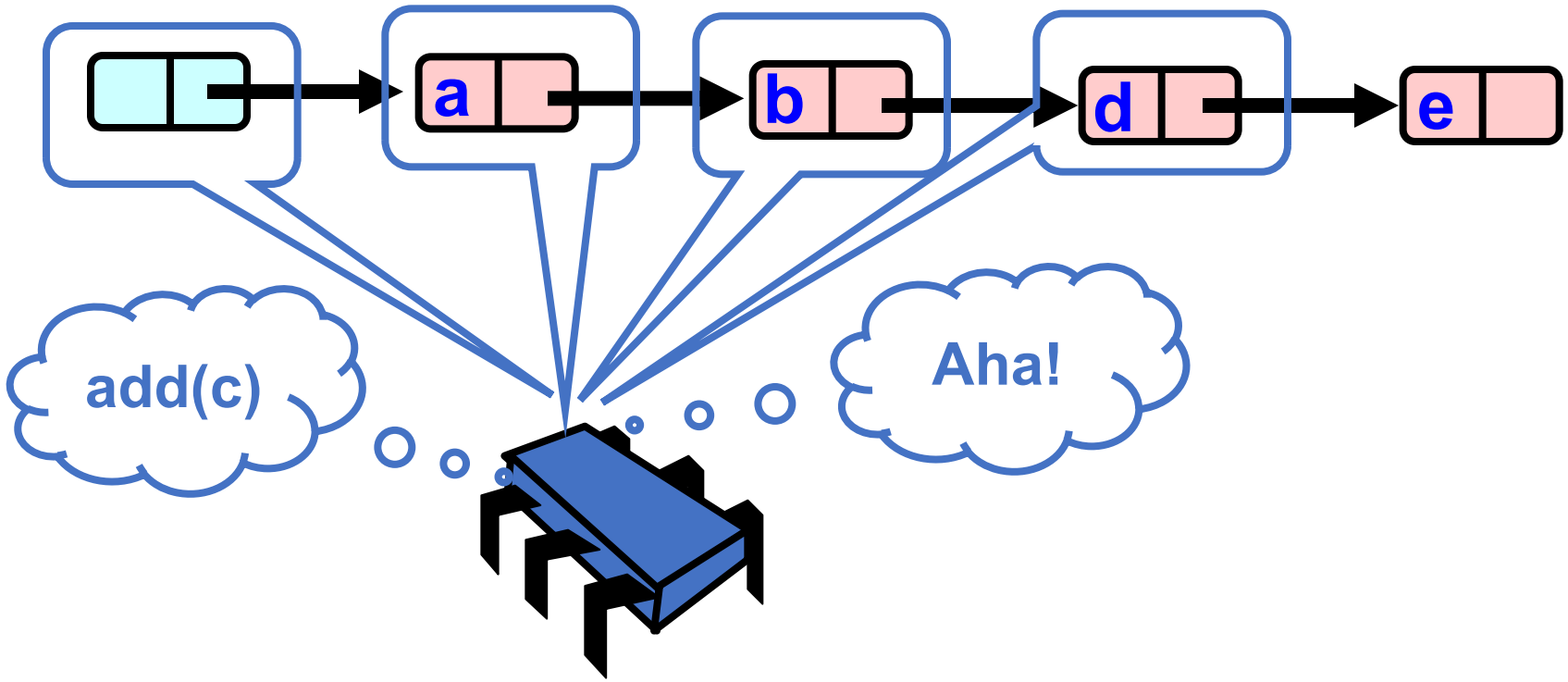
See you on Wednesday to start module 4!

Review

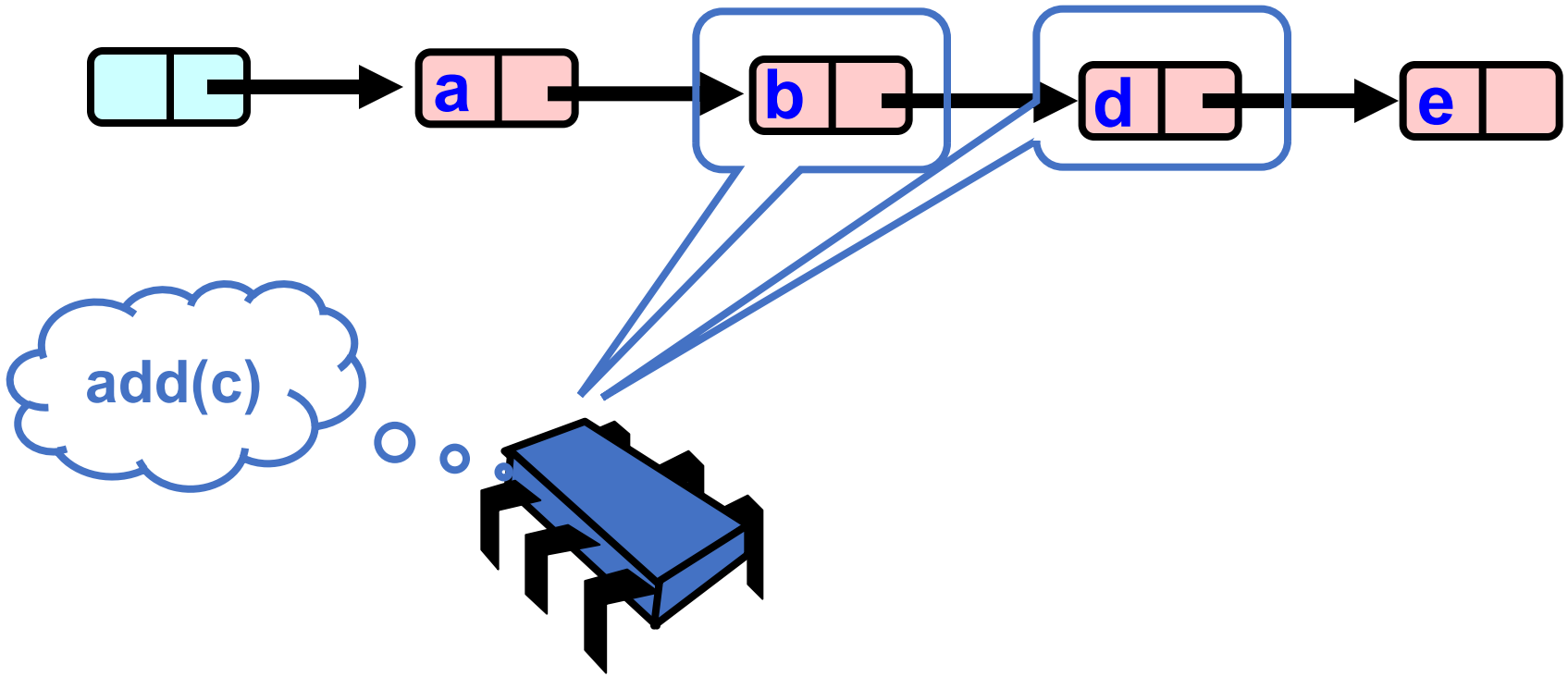
- Optimizing the concurrent set

Single traversal operations

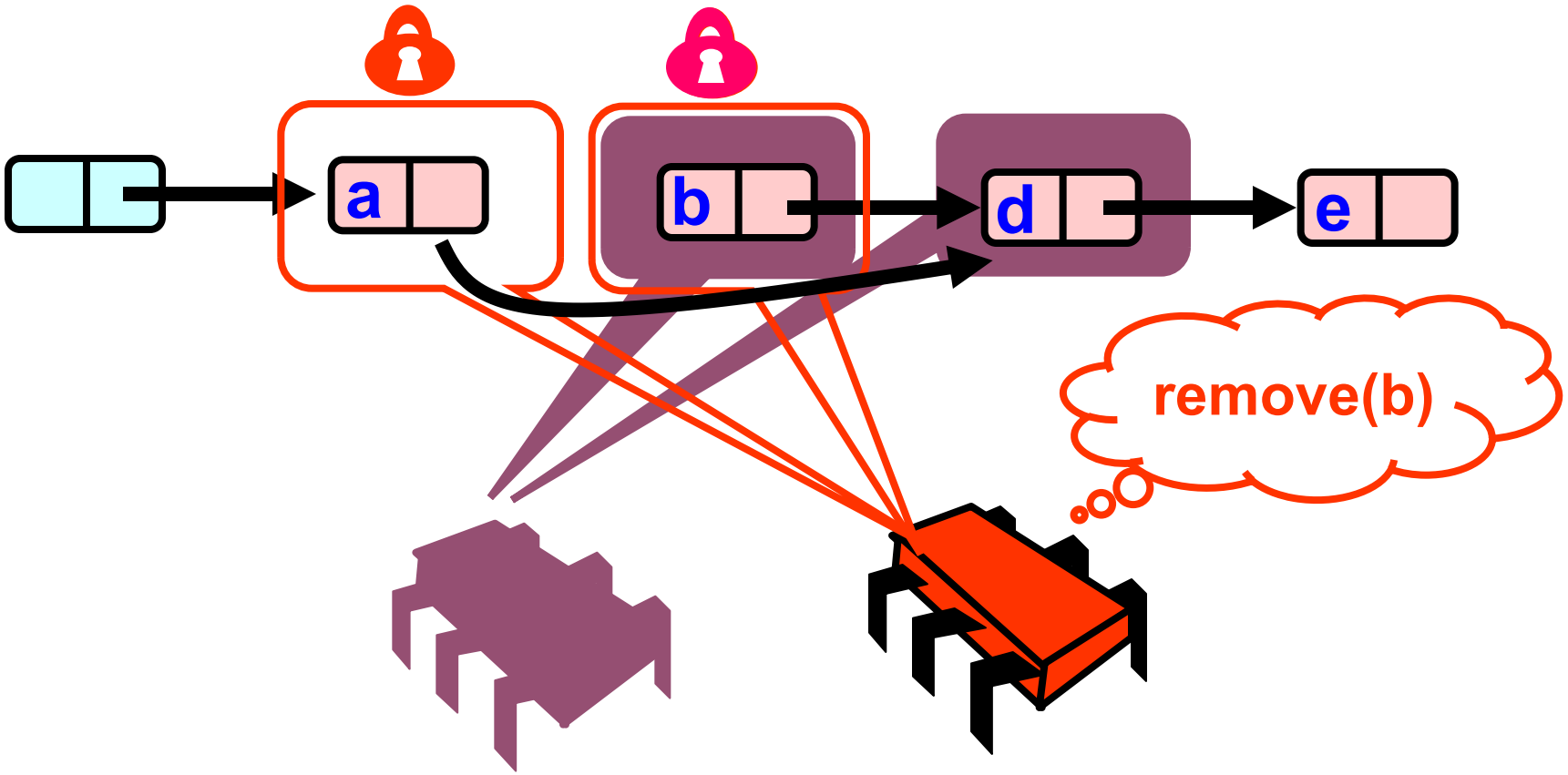
What could go wrong?



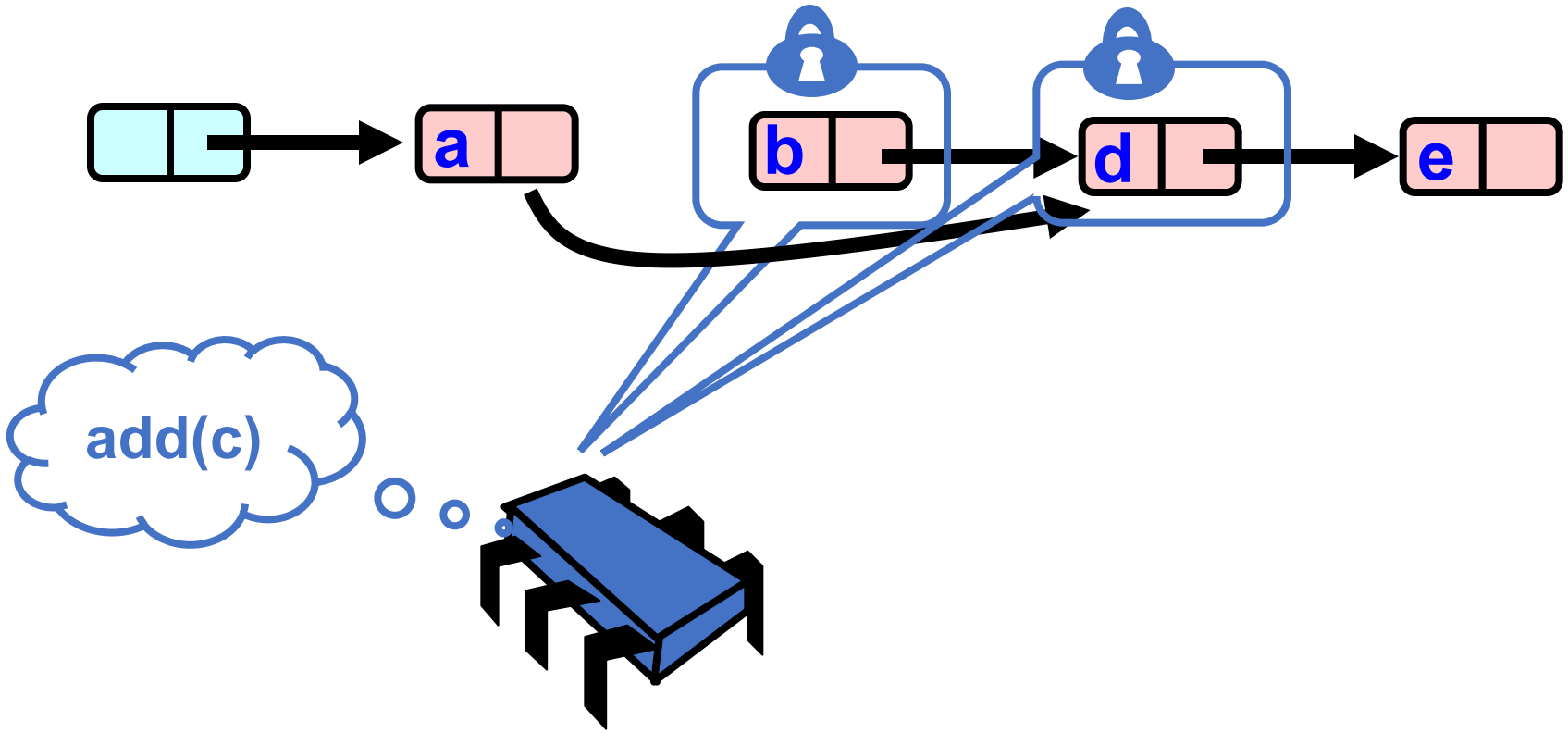
What could go wrong?



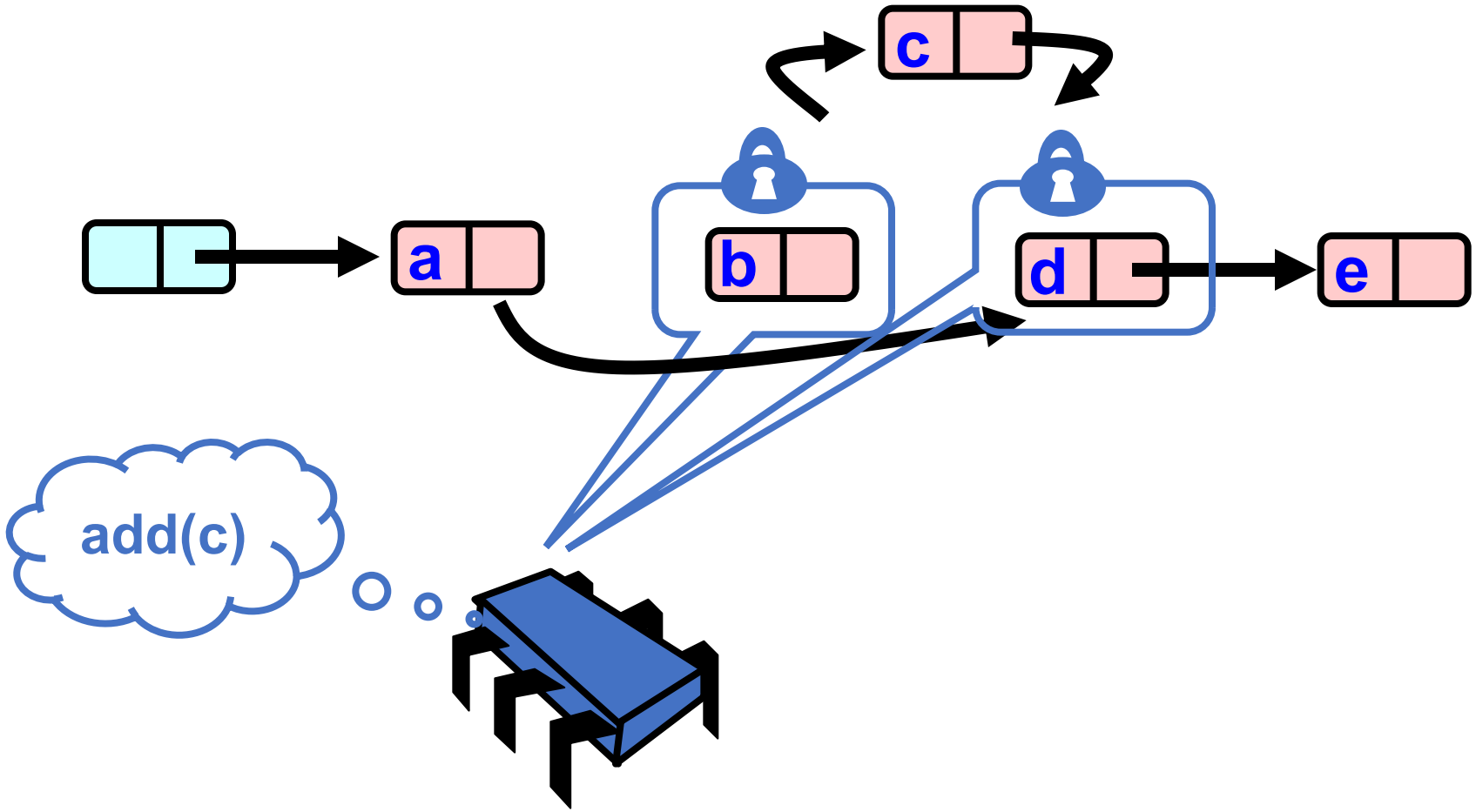
What could go wrong?



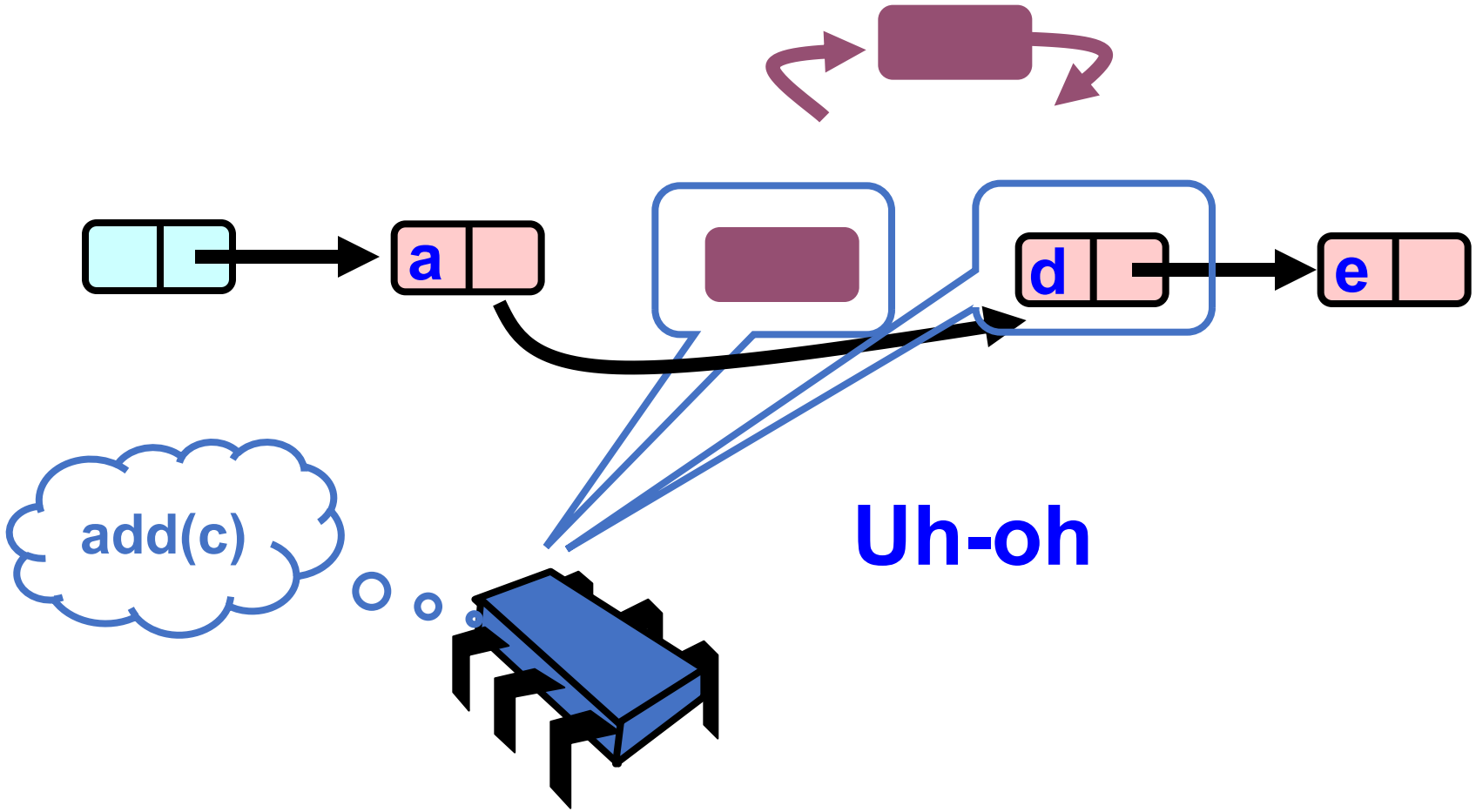
What could go wrong?



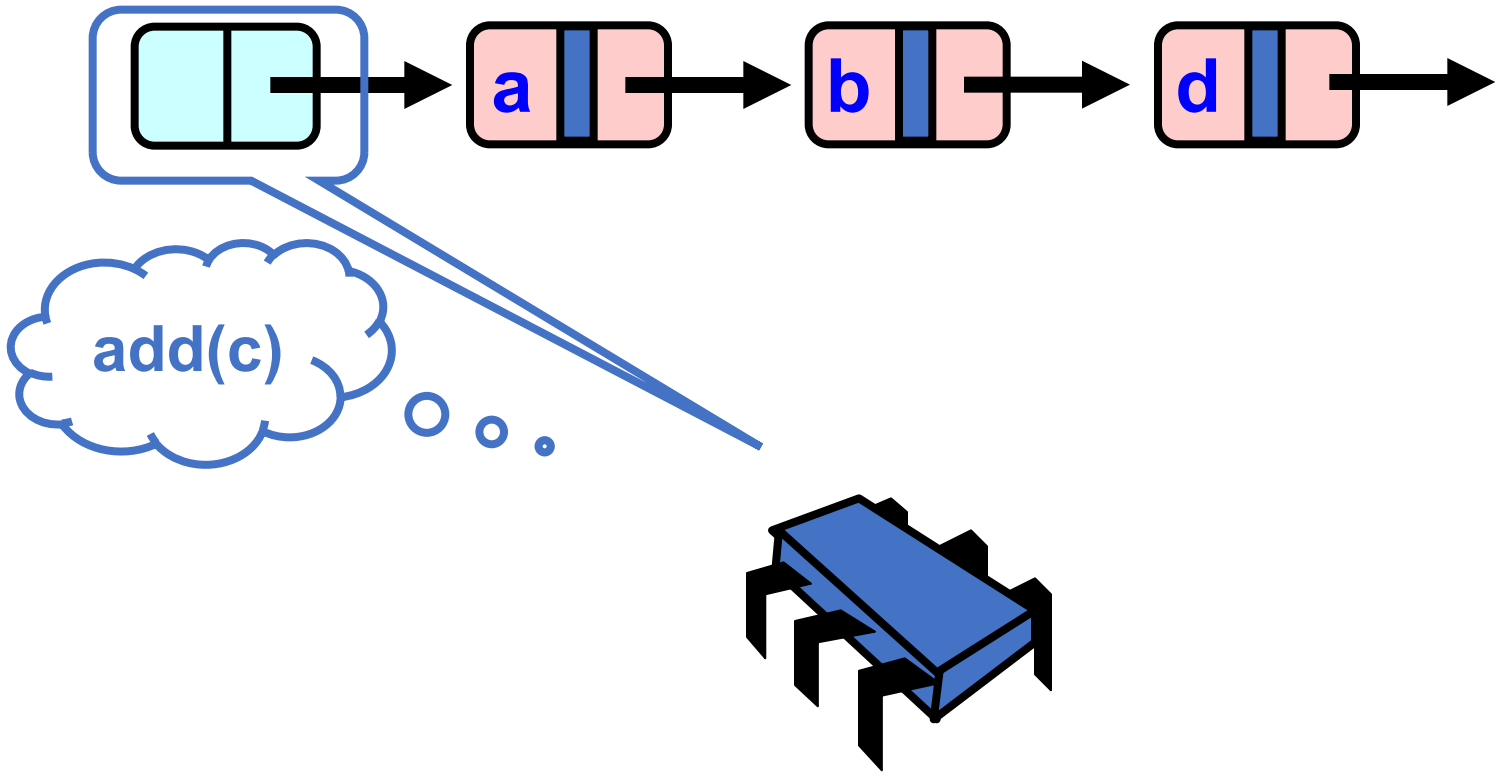
What could go wrong?



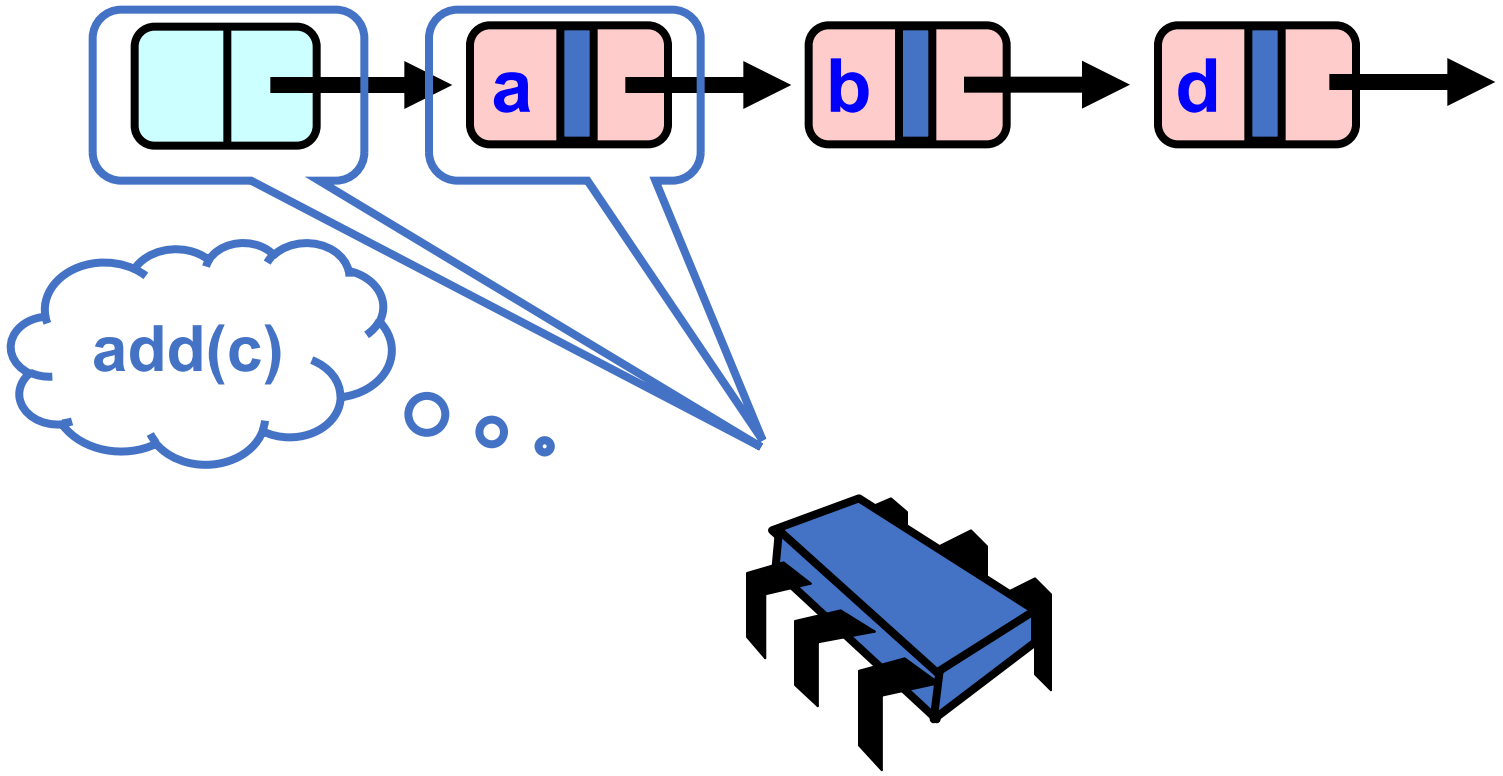
What could go wrong?



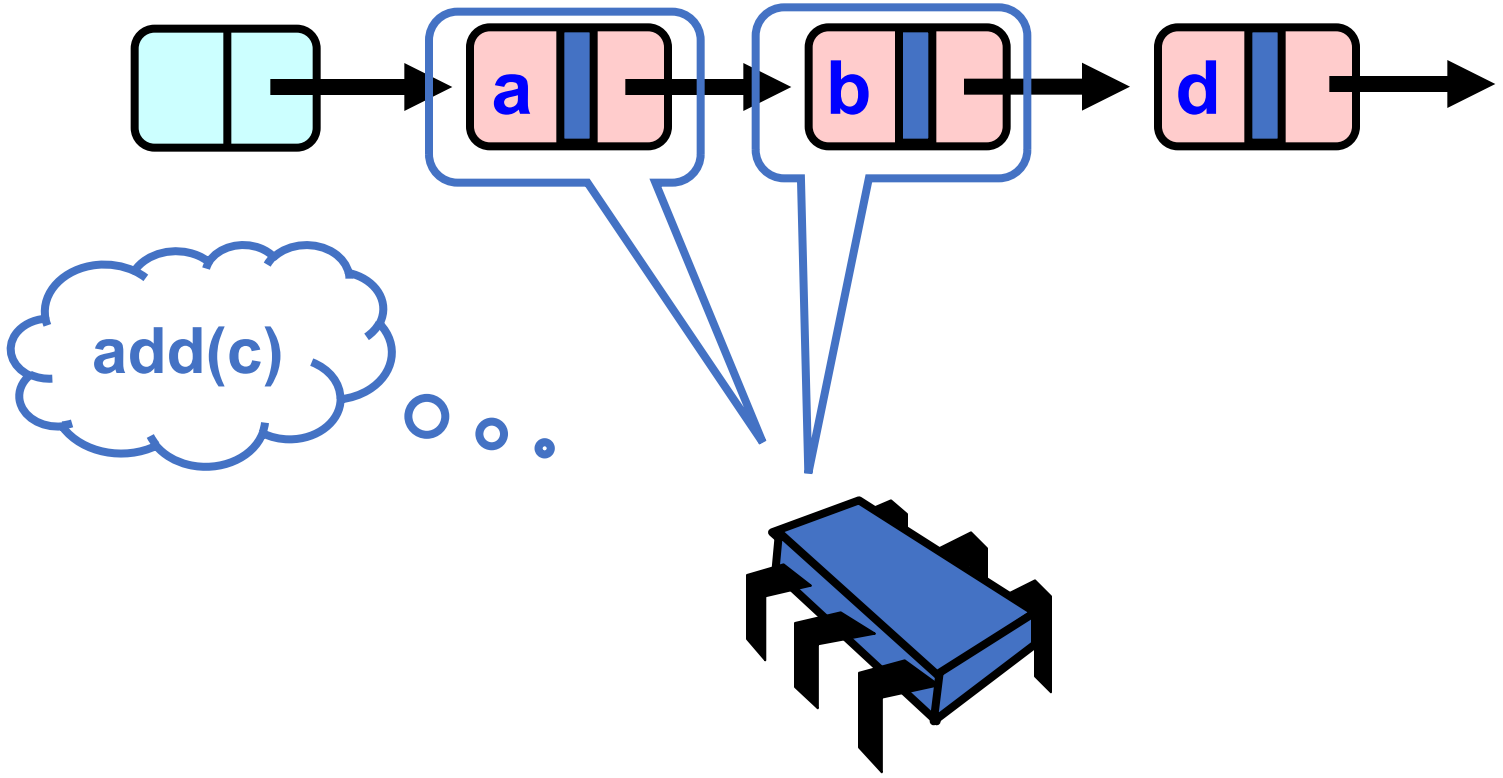
Fixed with logical flag



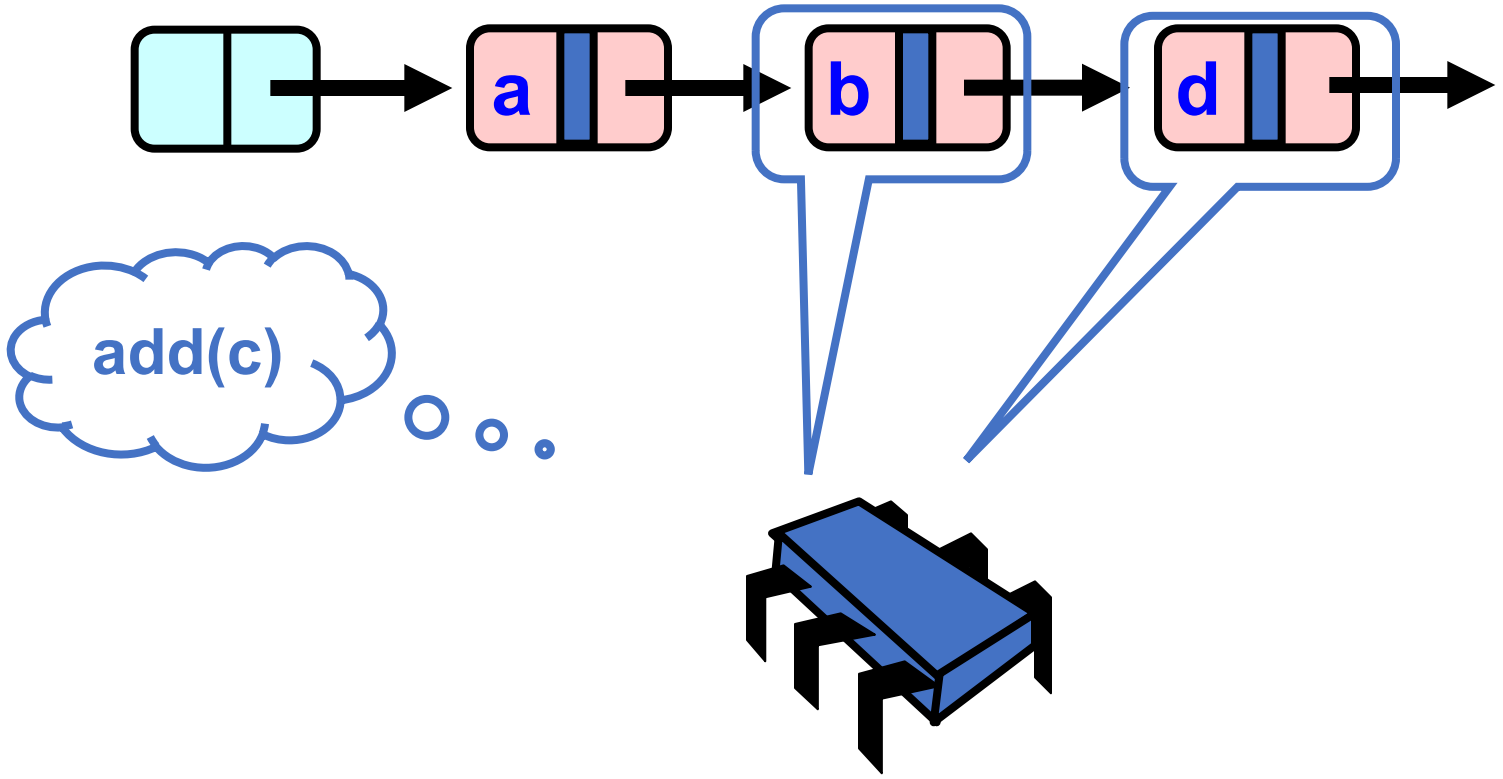
Fixed with logical flag



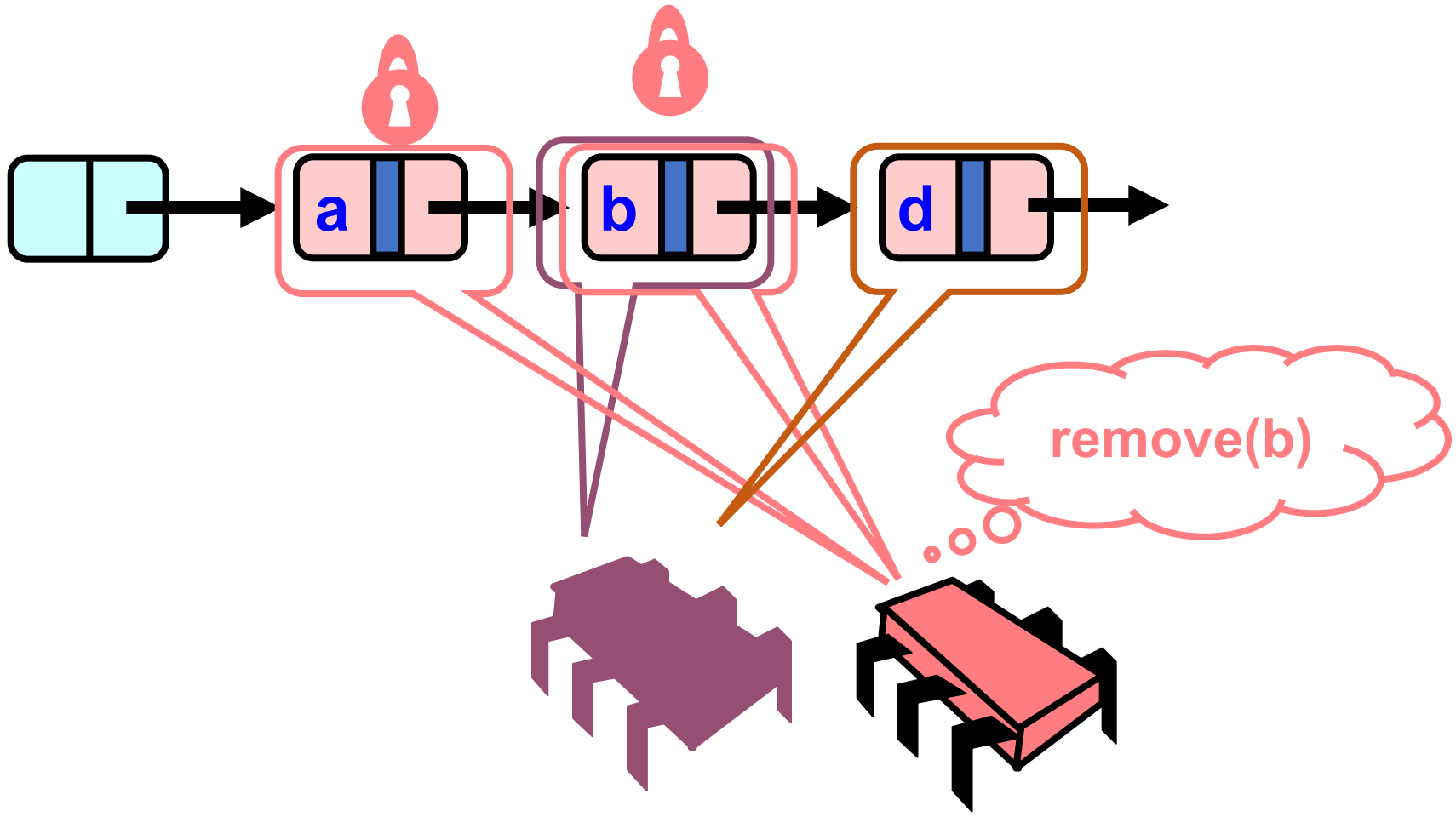
Fixed with logical flag



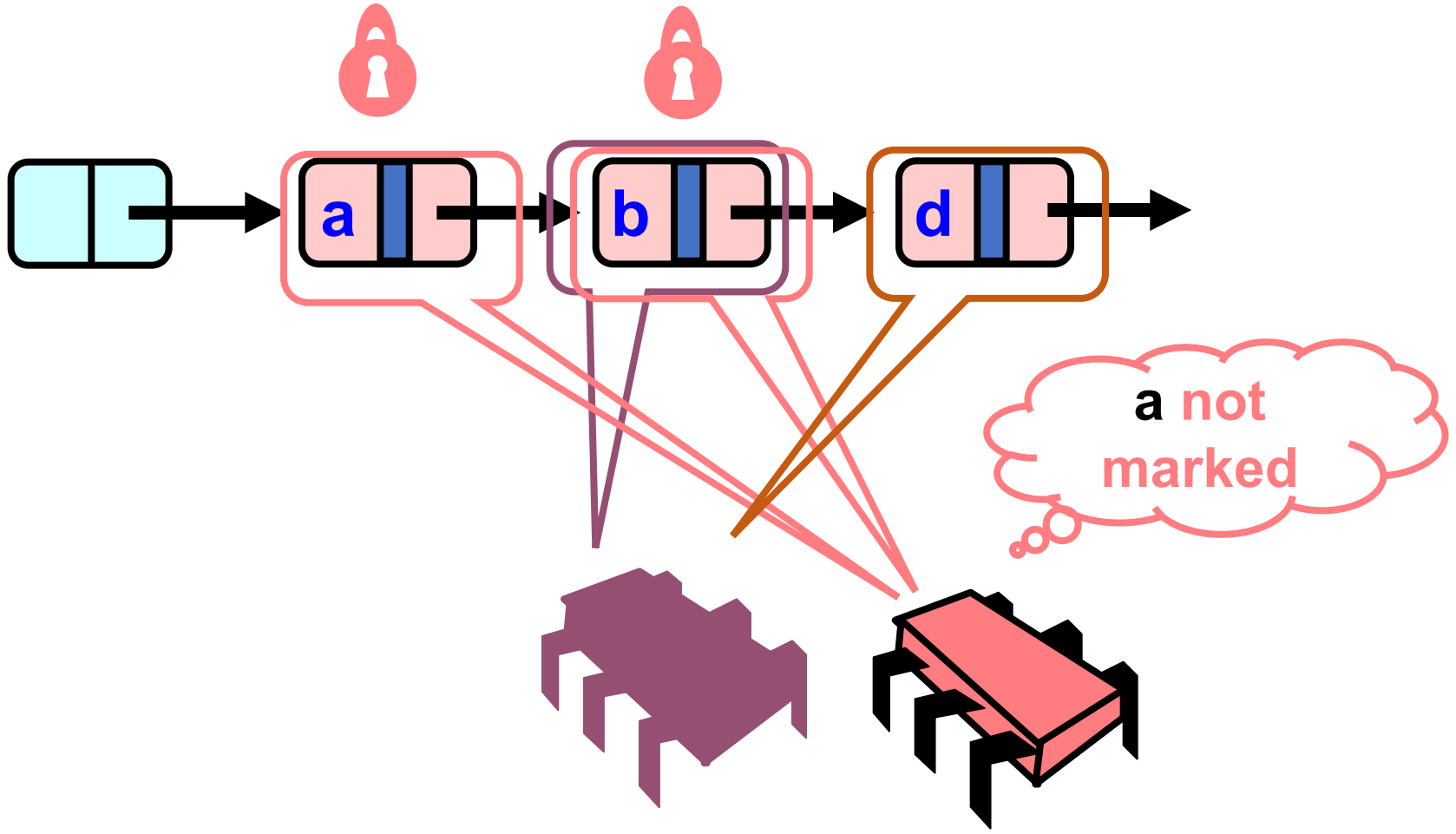
Fixed with logical flag



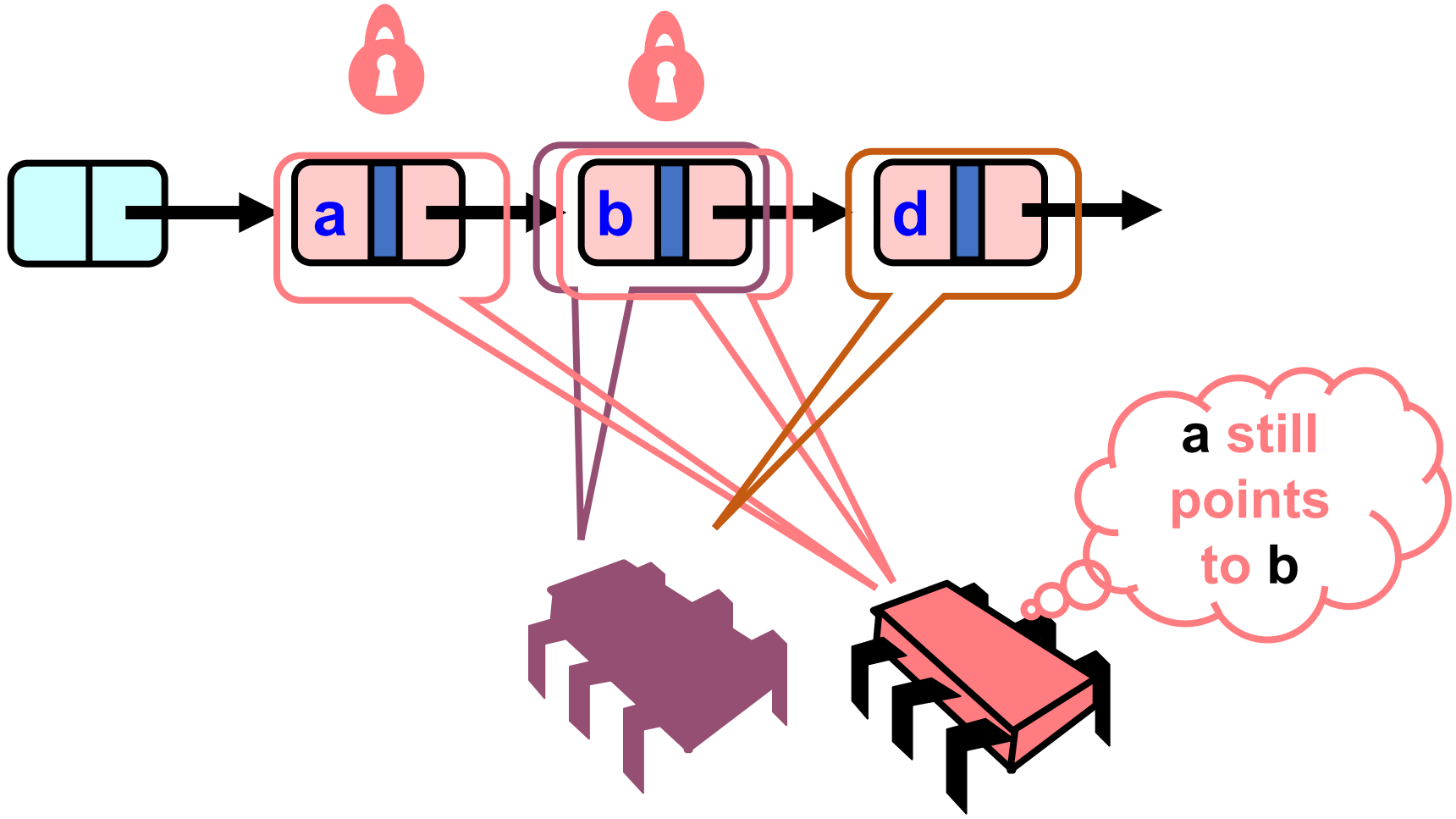
Fixed with logical flag



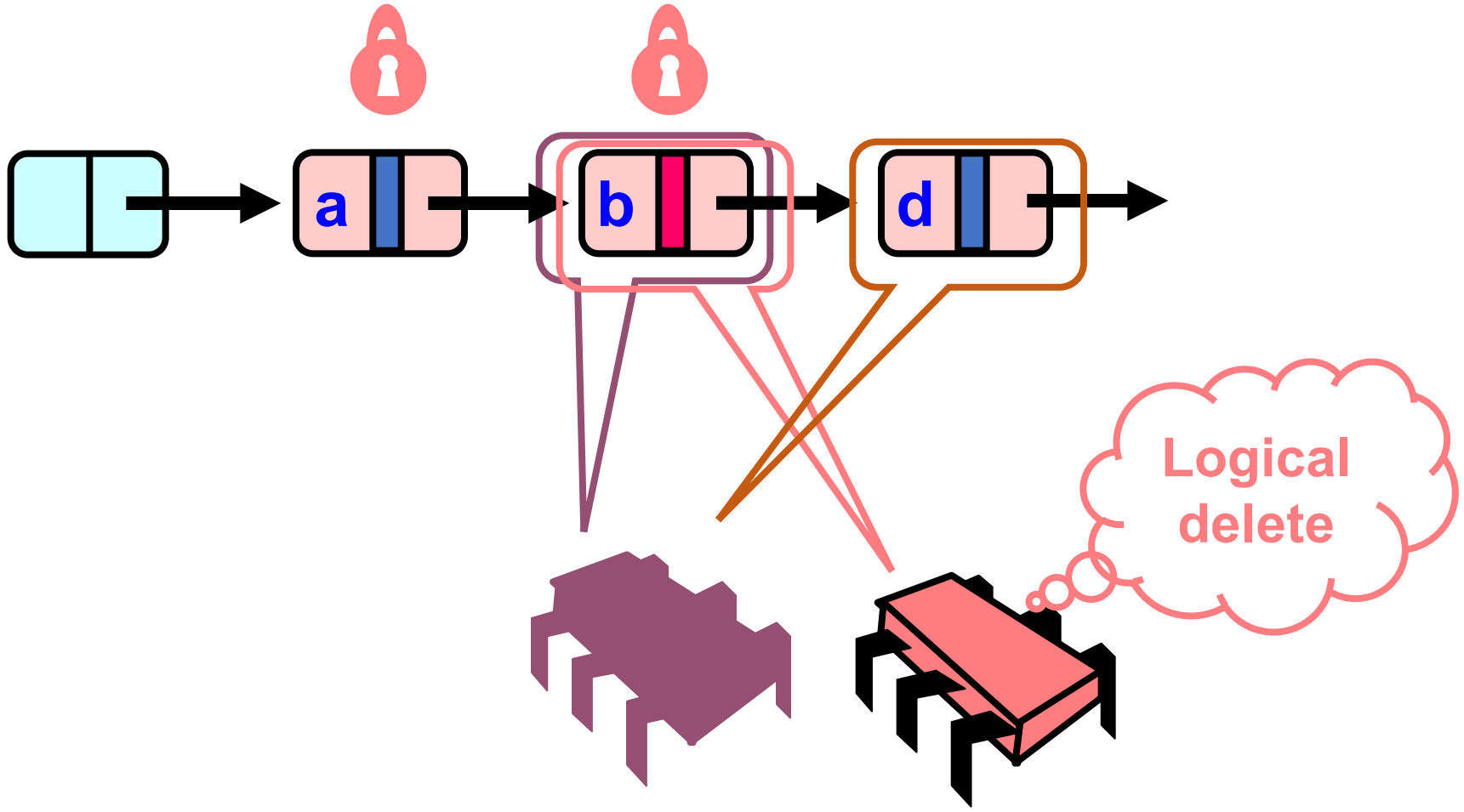
Fixed with logical flag



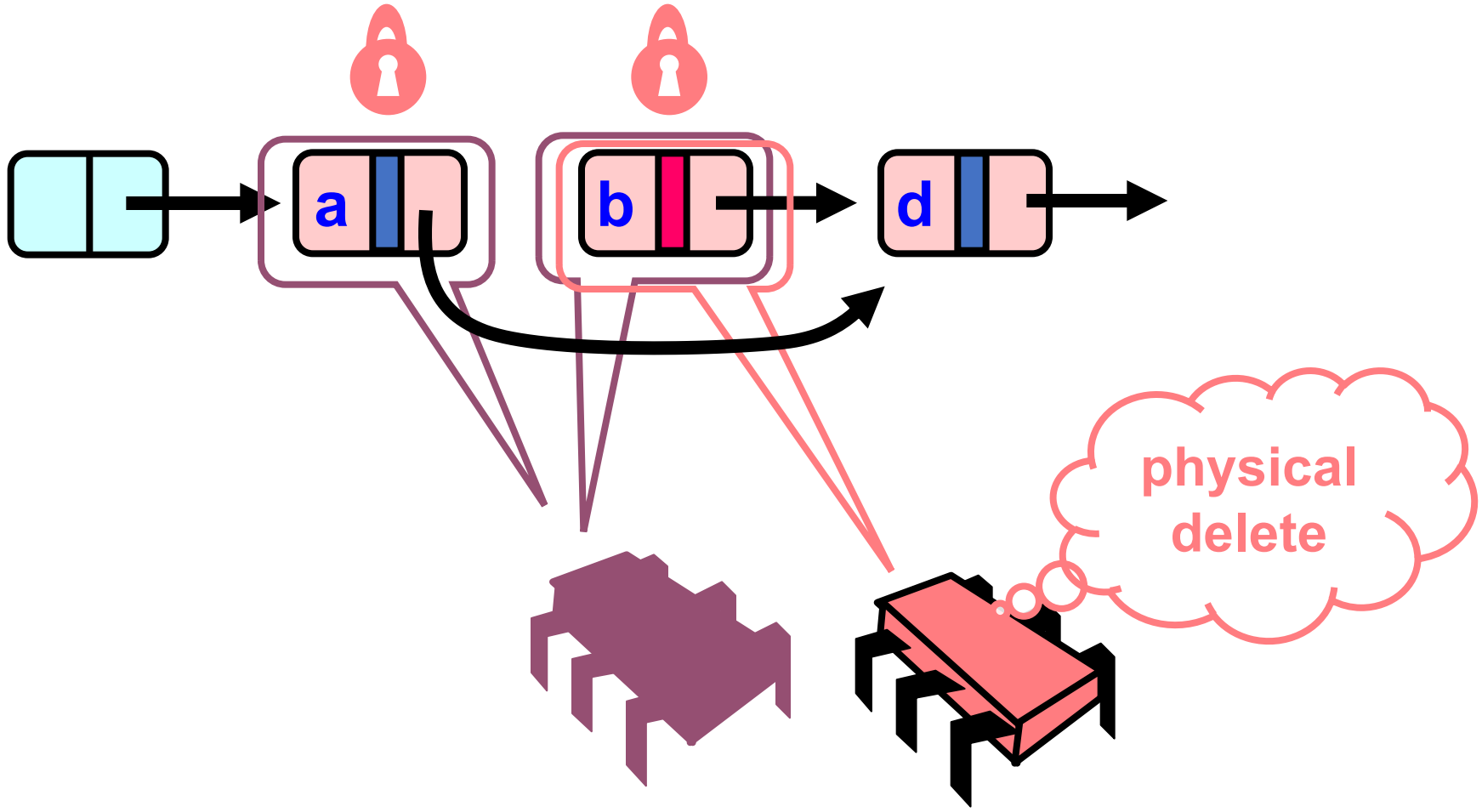
Fixed with logical flag



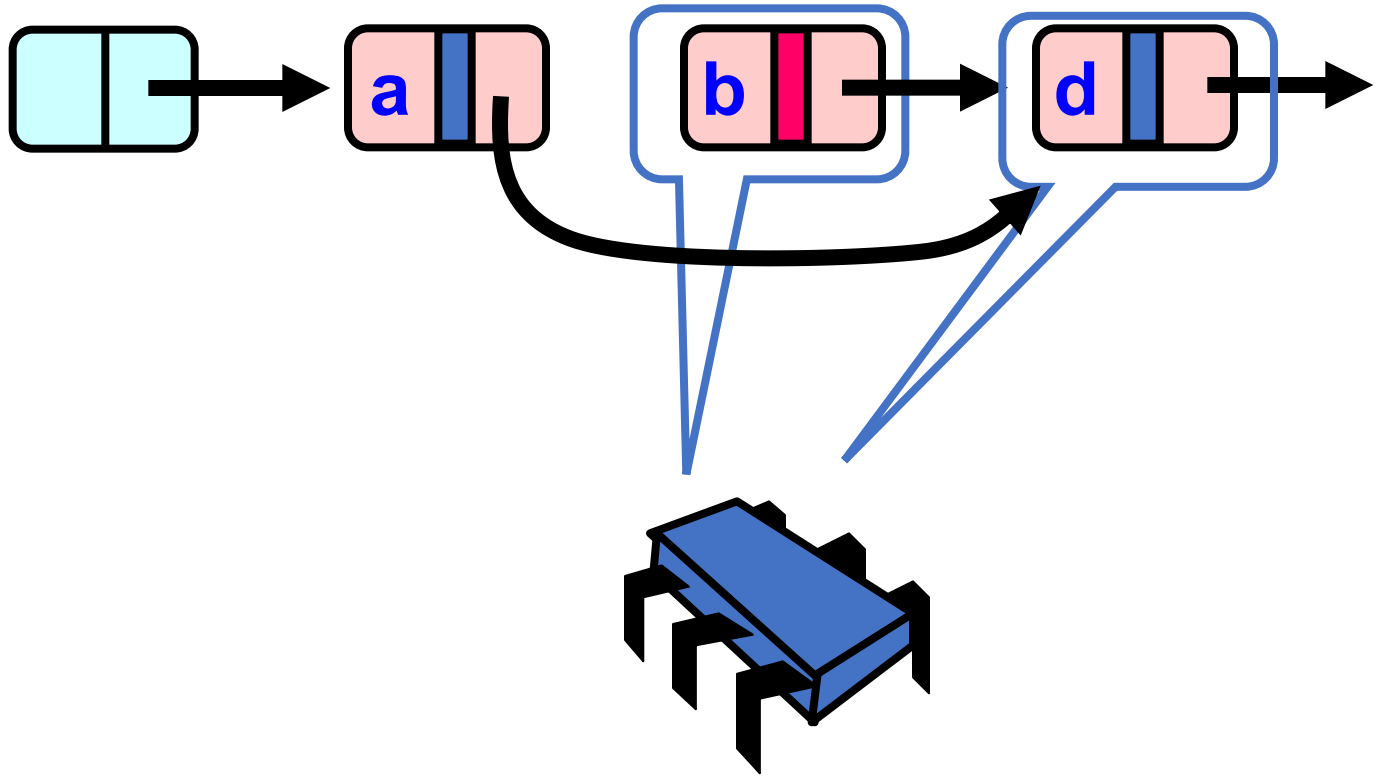
Fixed with logical flag



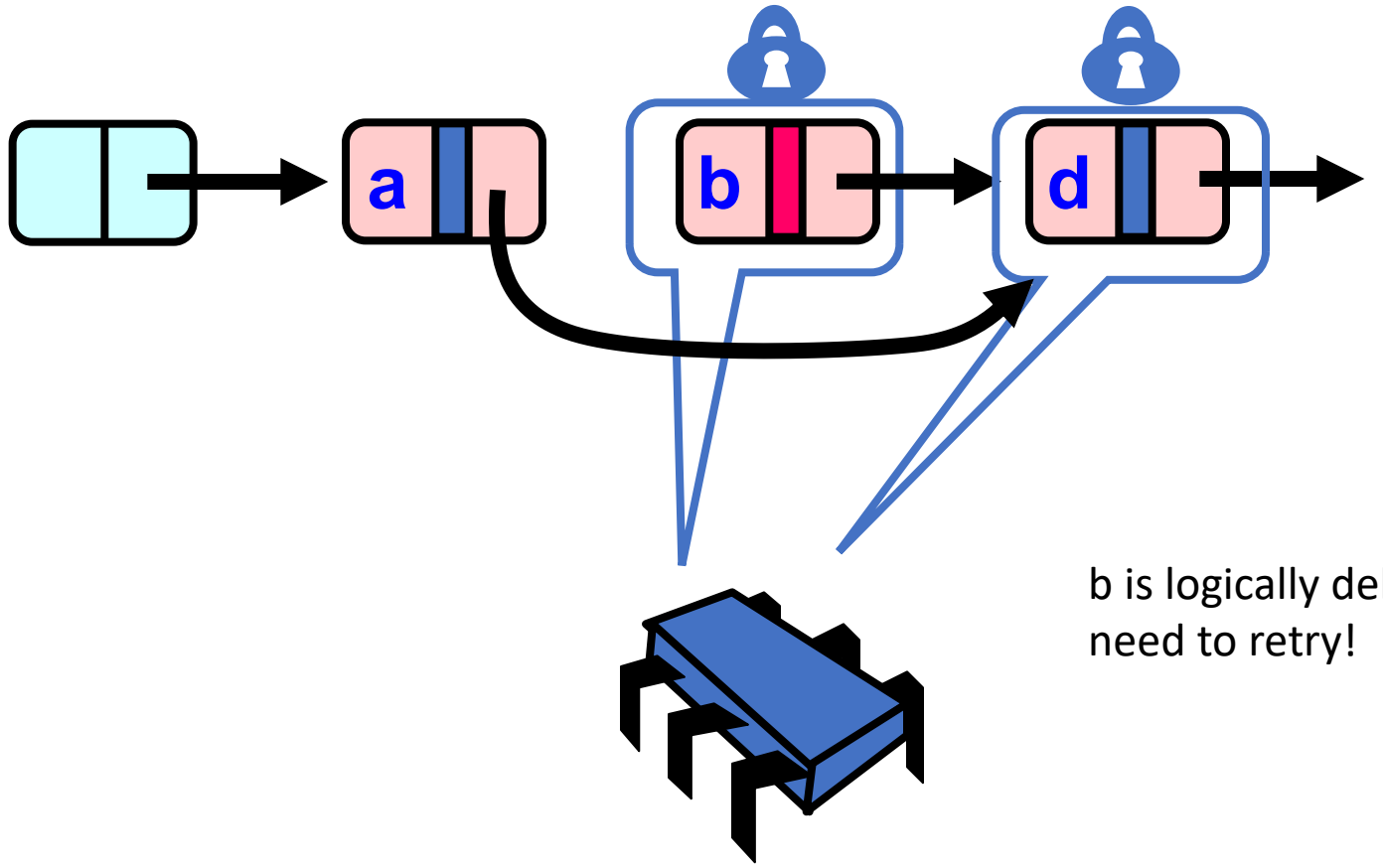
Fixed with logical flag



Fixed with logical flag



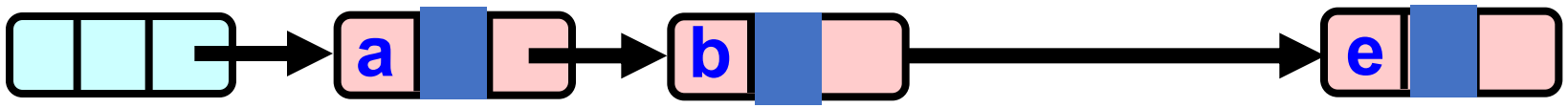
Fixed with logical flag



b is logically deleted so we need to retry!

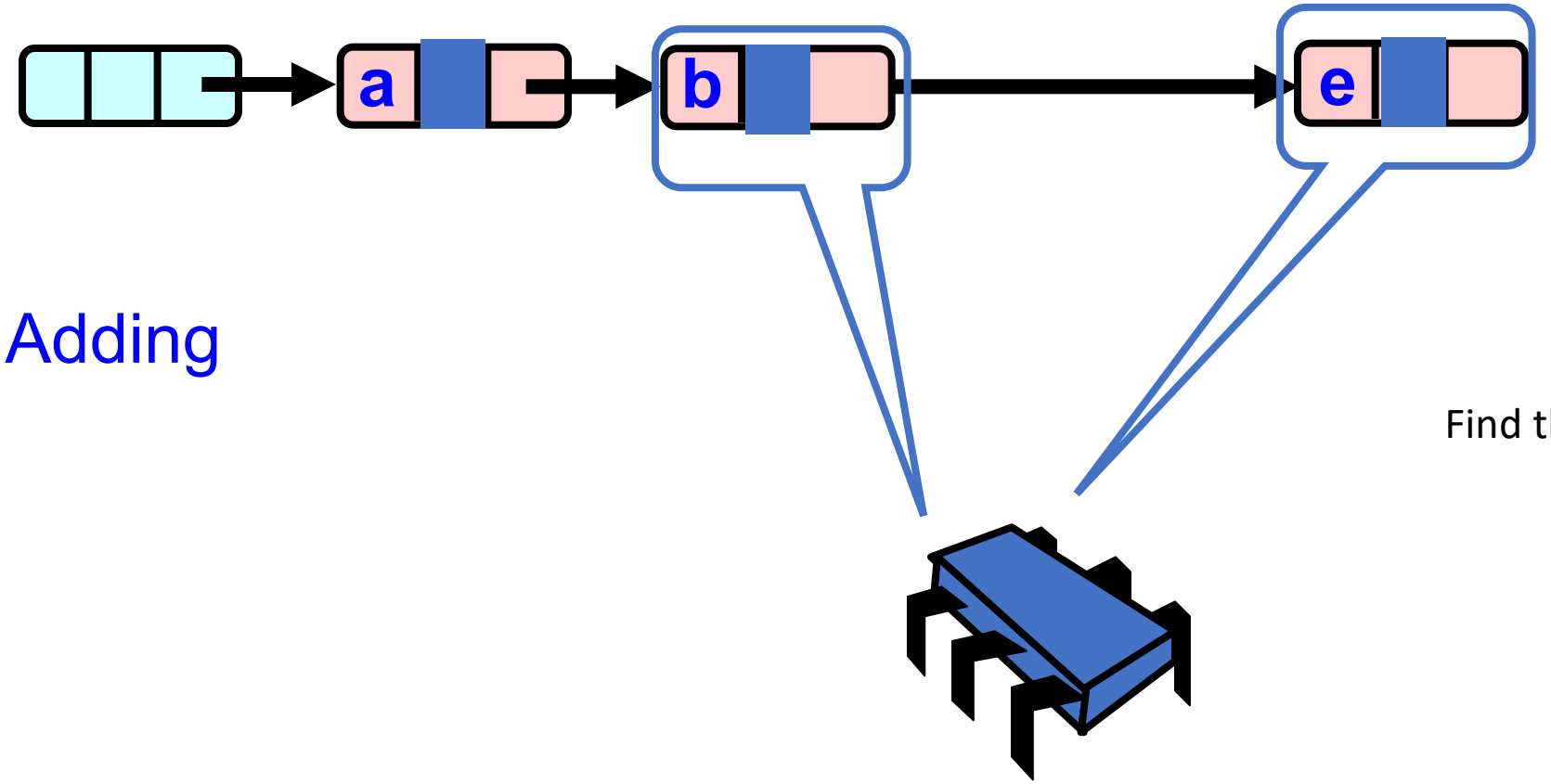
CAS based insertion

Lock-free Lists

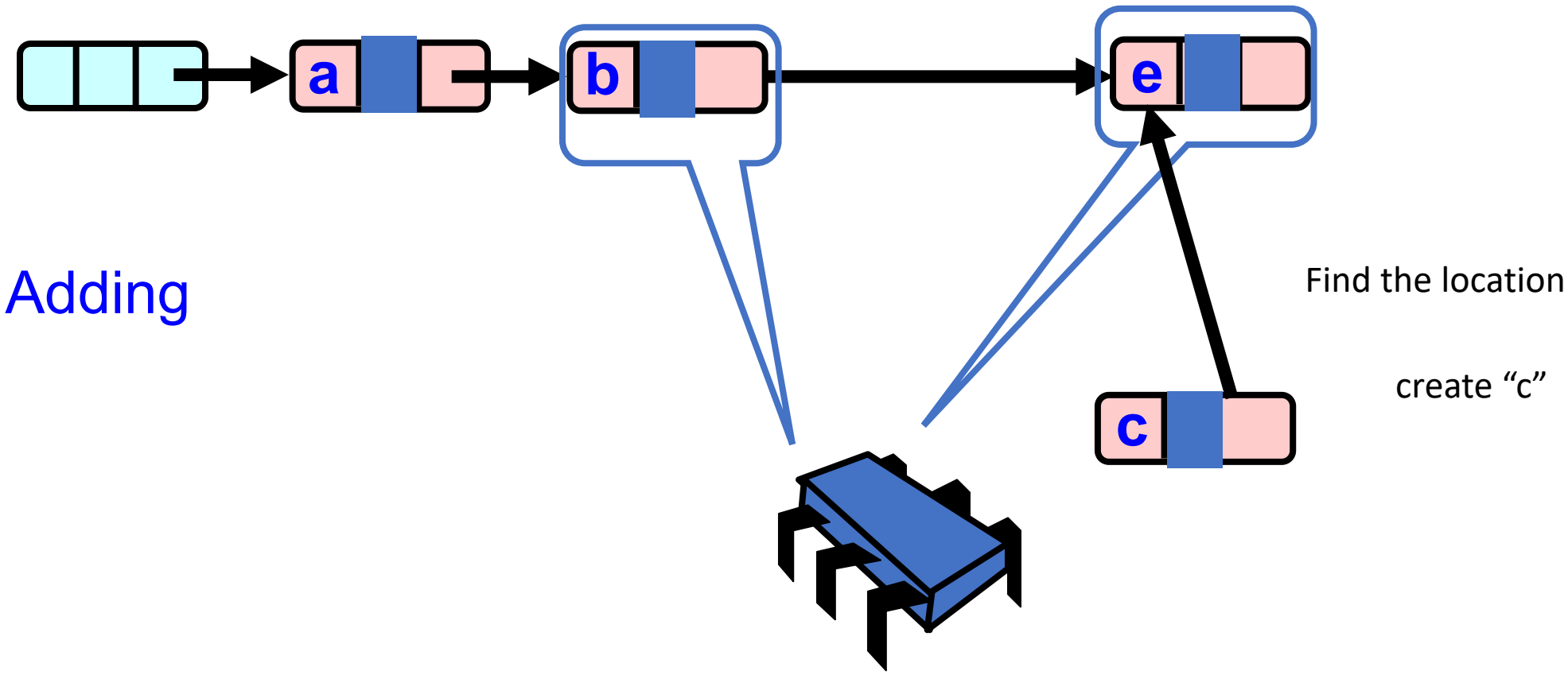


Adding

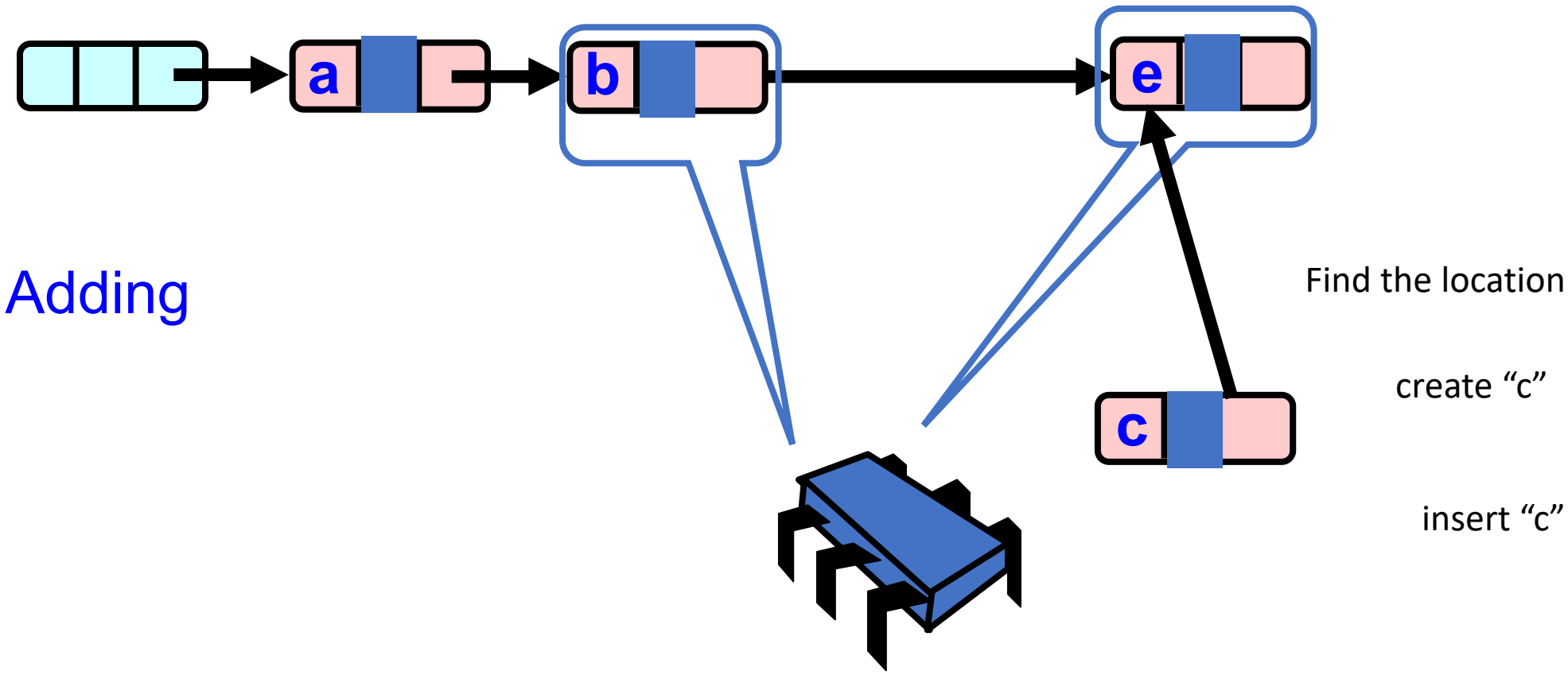
Lock-free Lists



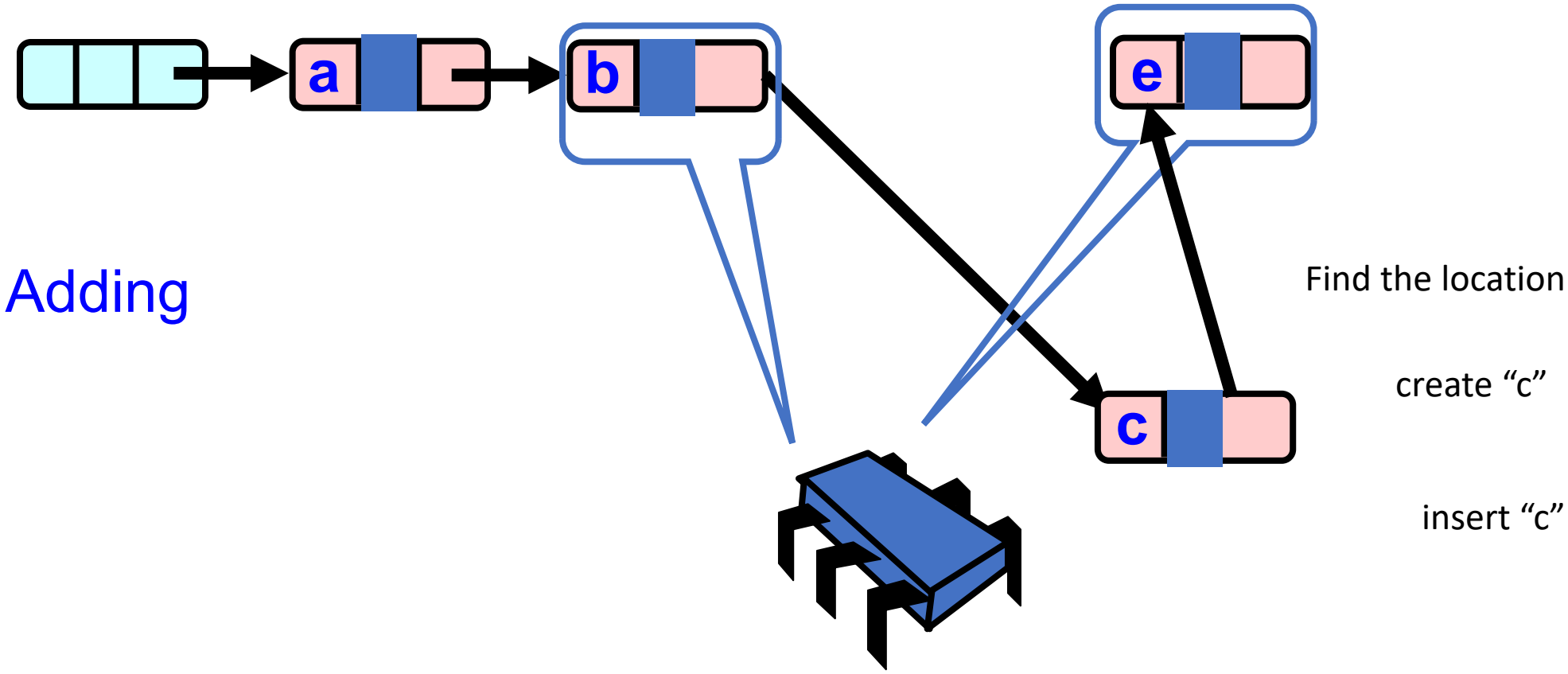
Lock-free Lists



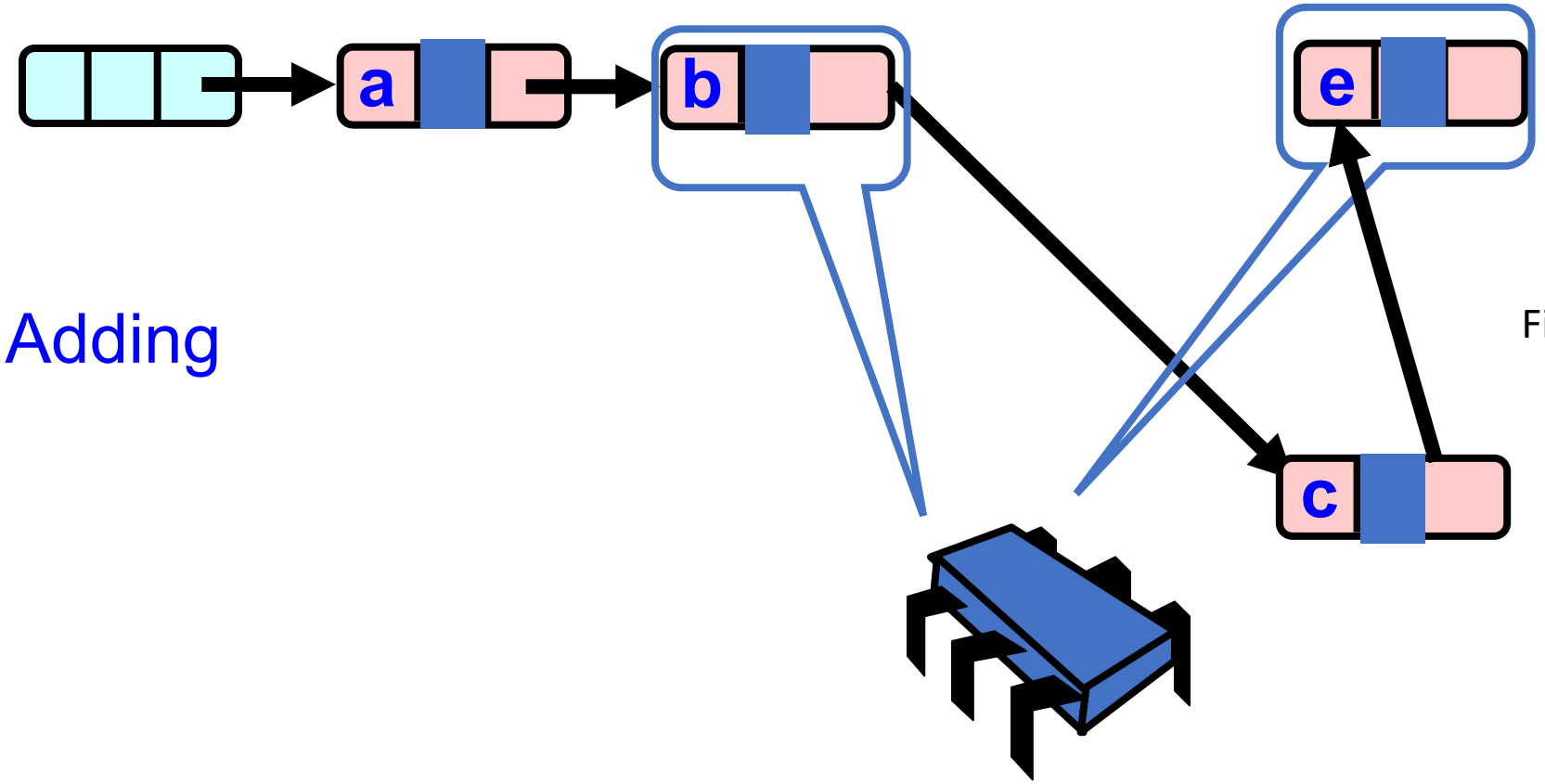
Lock-free Lists



Lock-free Lists



Lock-free Lists



Adding

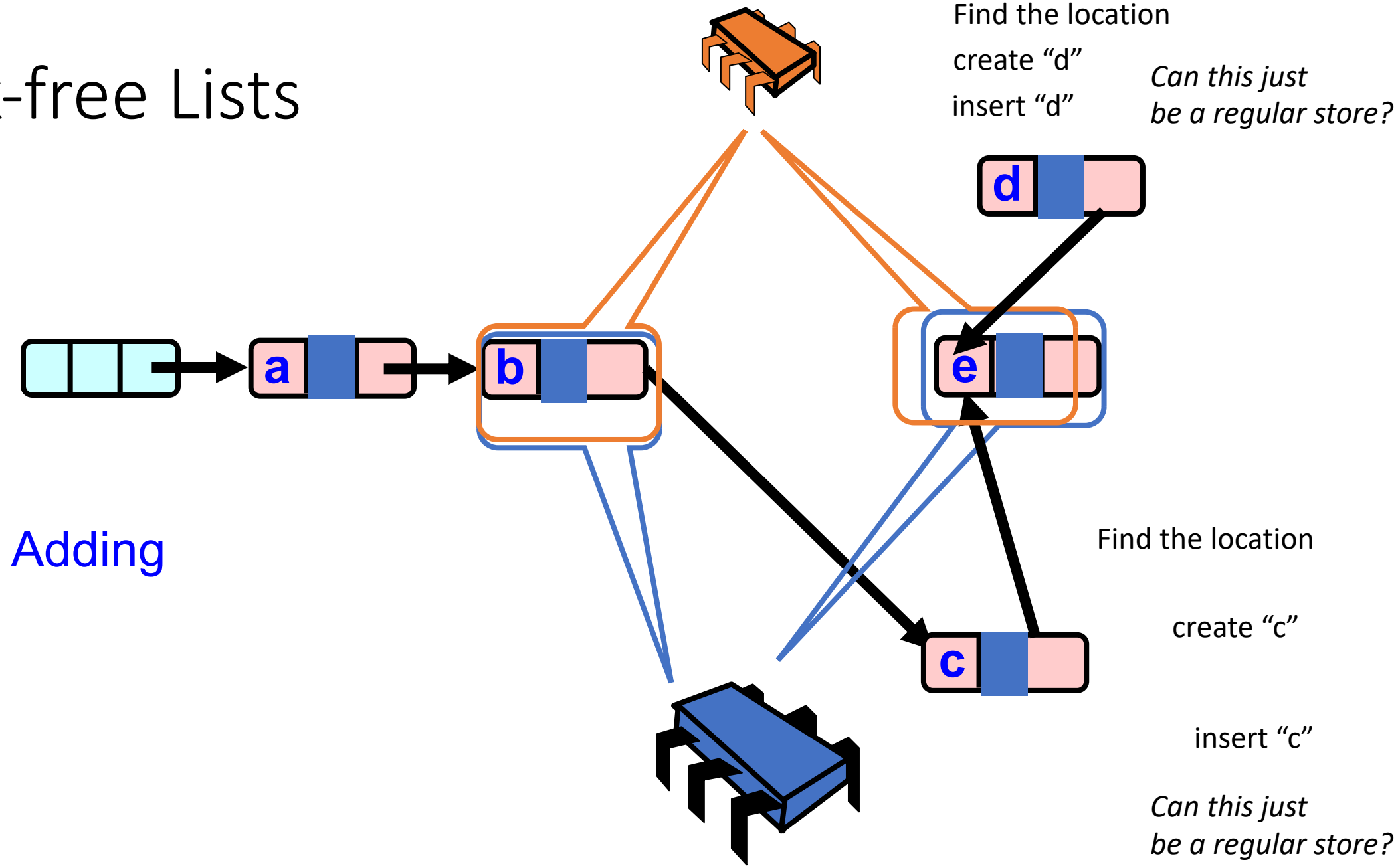
Find the location

create “c”

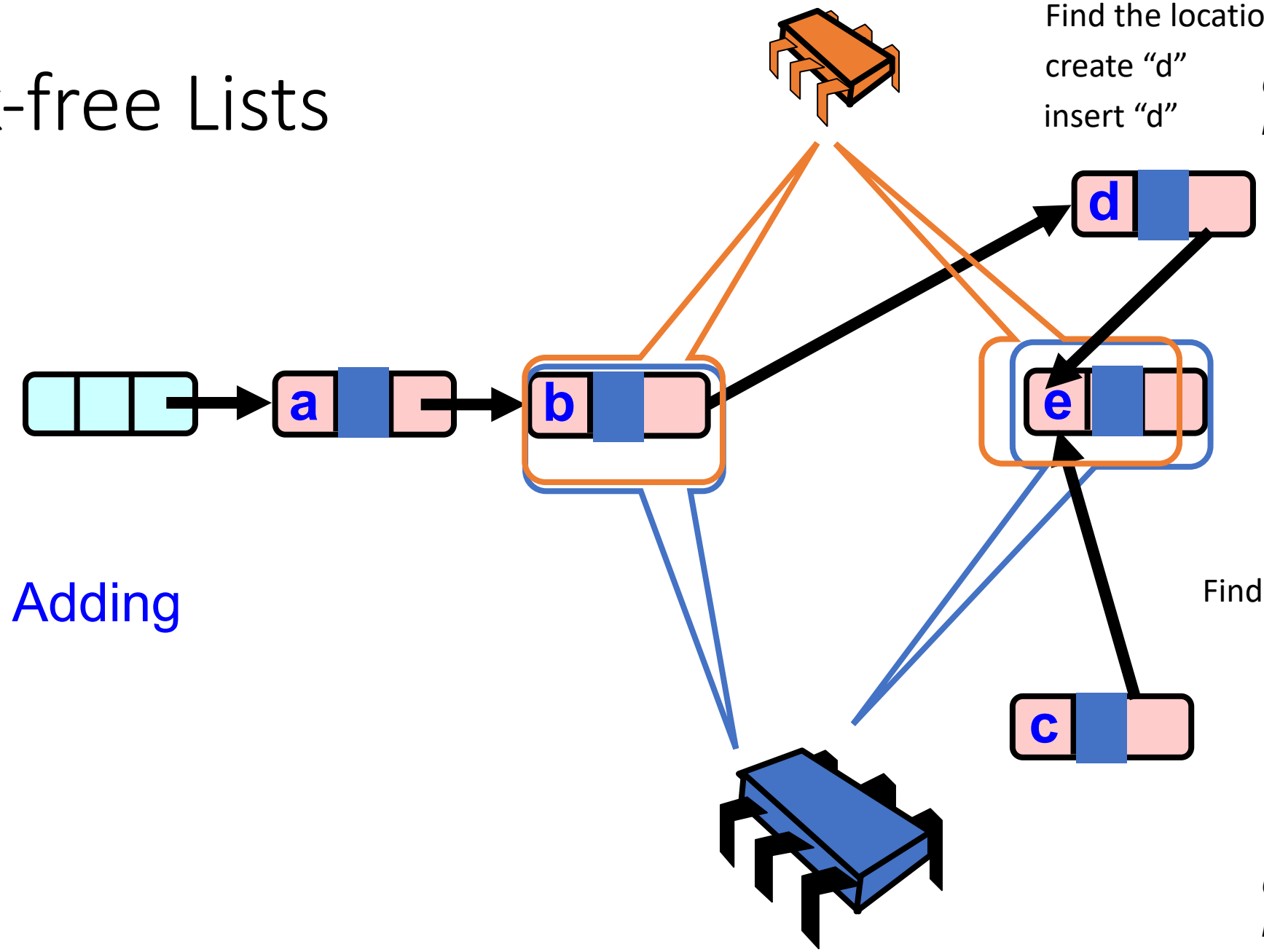
insert “c”

Can this just be a regular store?

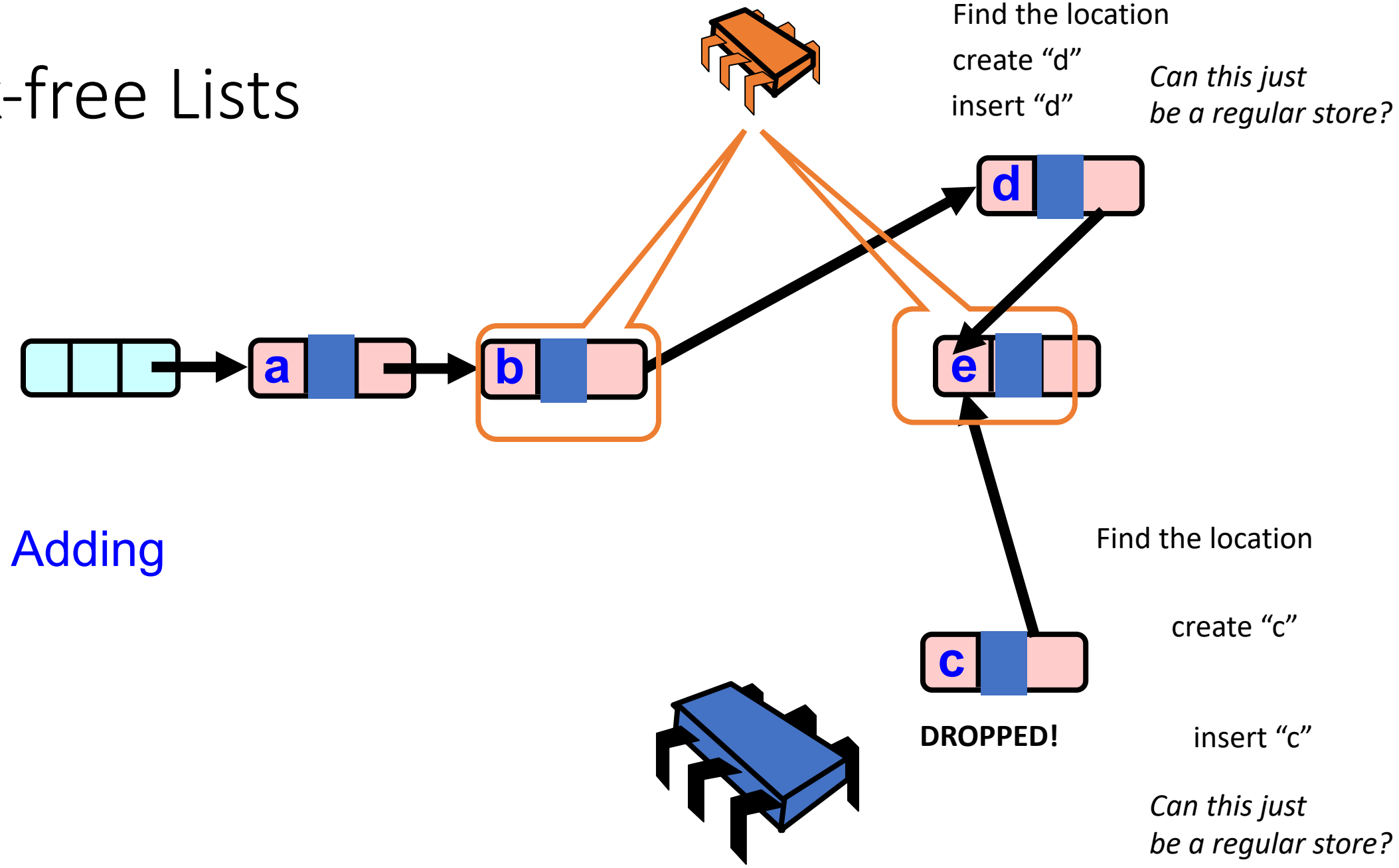
Lock-free Lists



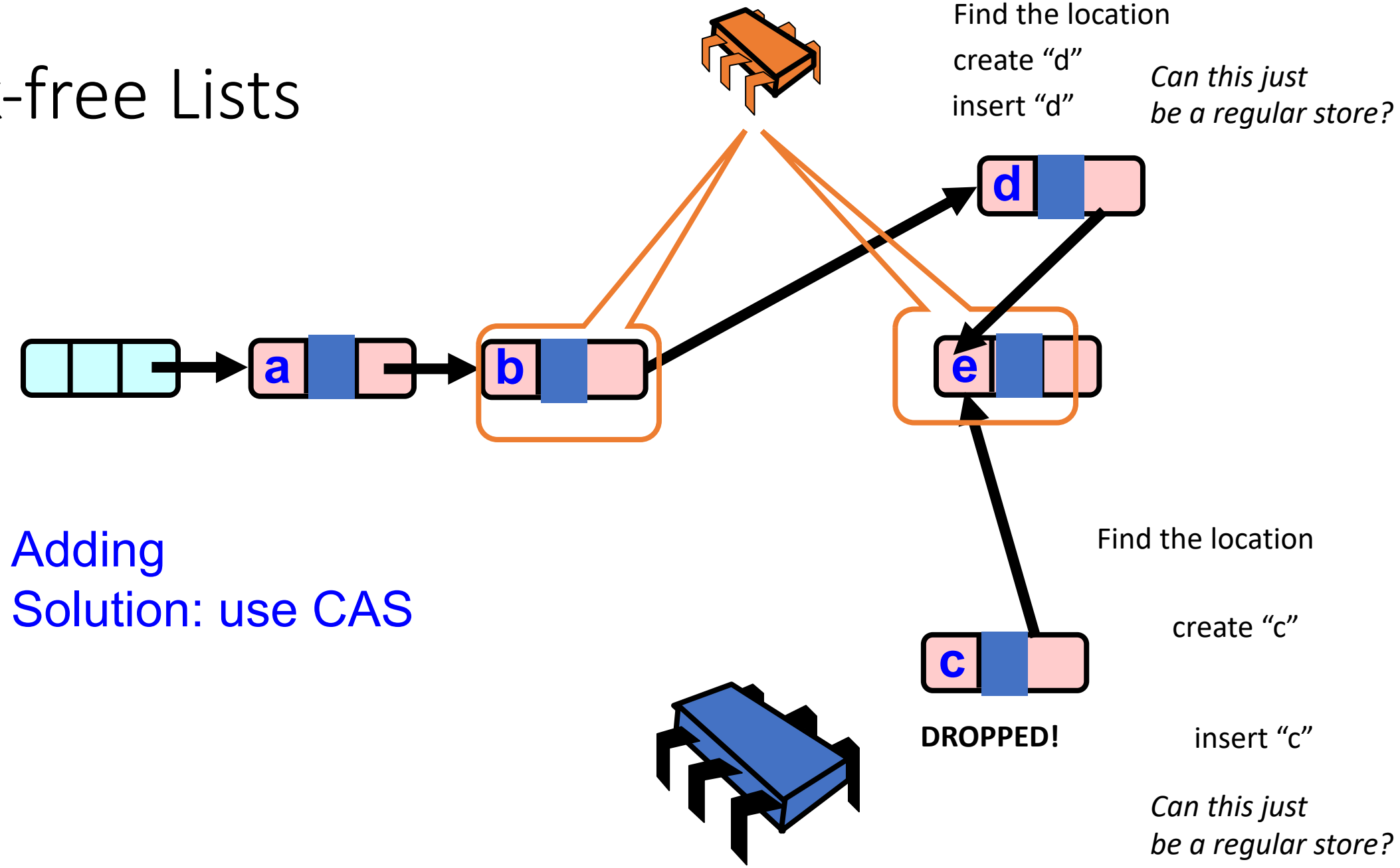
Lock-free Lists



Lock-free Lists



Lock-free Lists

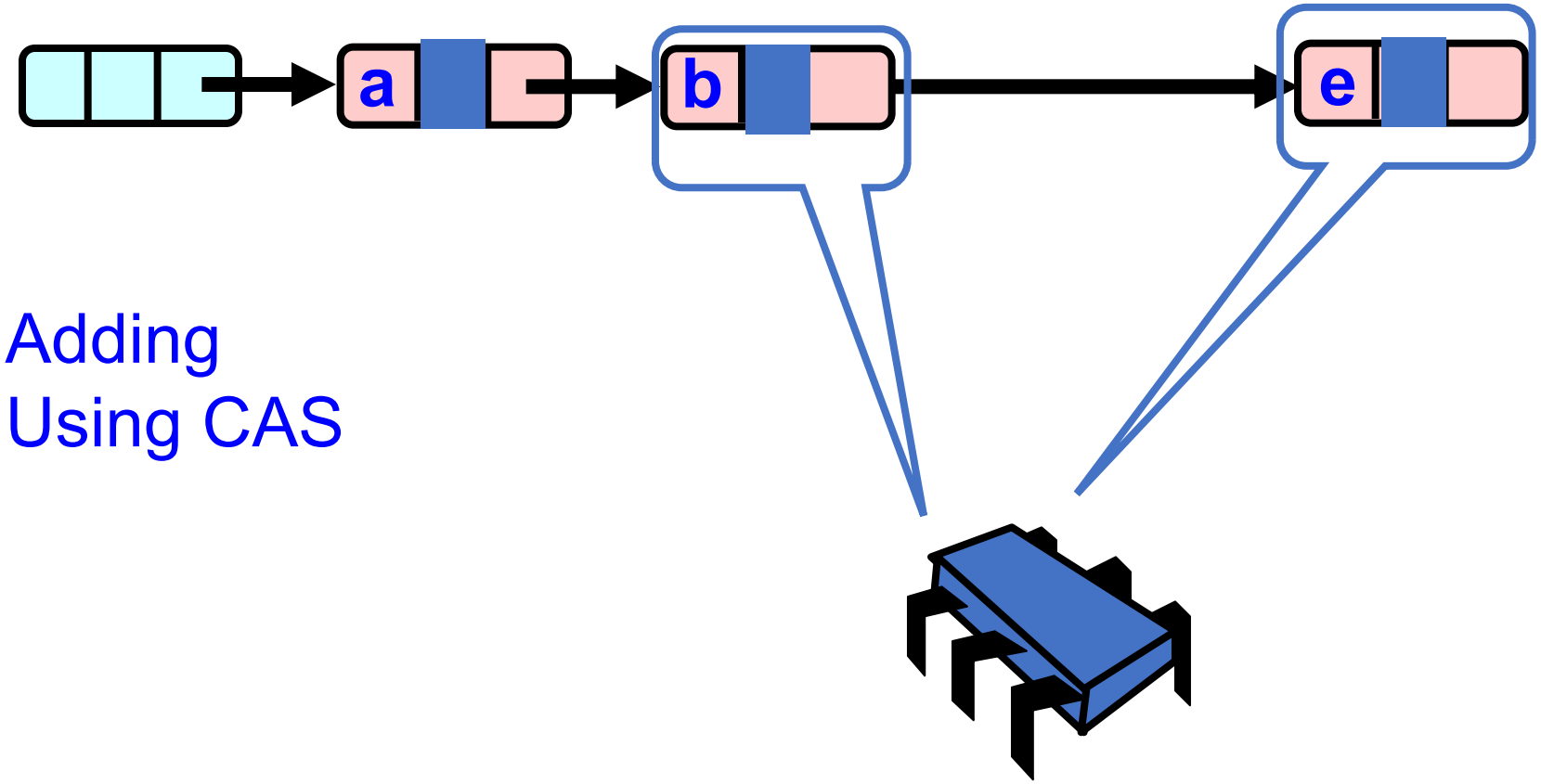


Lock-free Lists

Find the location

Cache your insertion point!

b.next == e



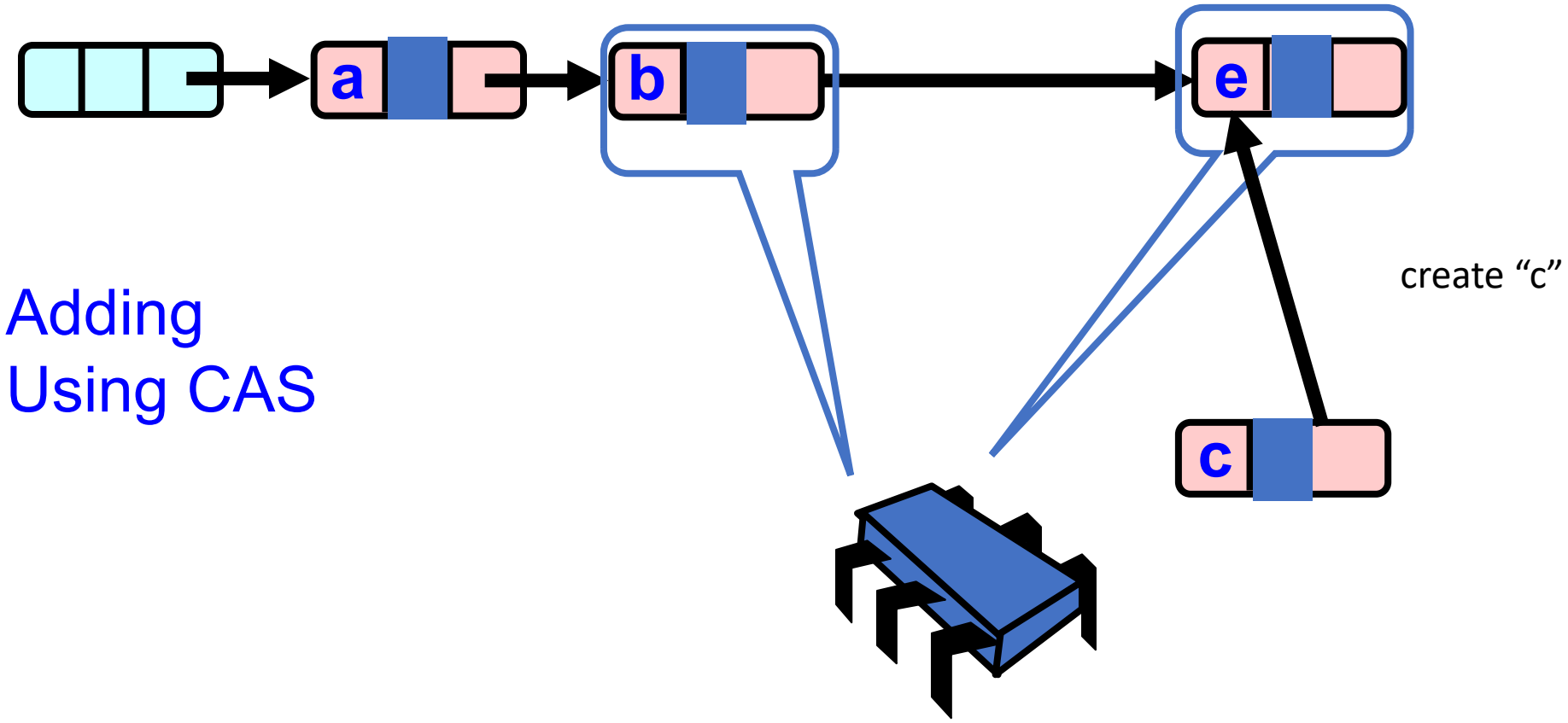
Adding
Using CAS

Lock-free Lists

Find the location

Cache your insertion point!

b.next == e



Adding
Using CAS

create “c”

Lock-free Lists

Only insert if your insertion point is valid!

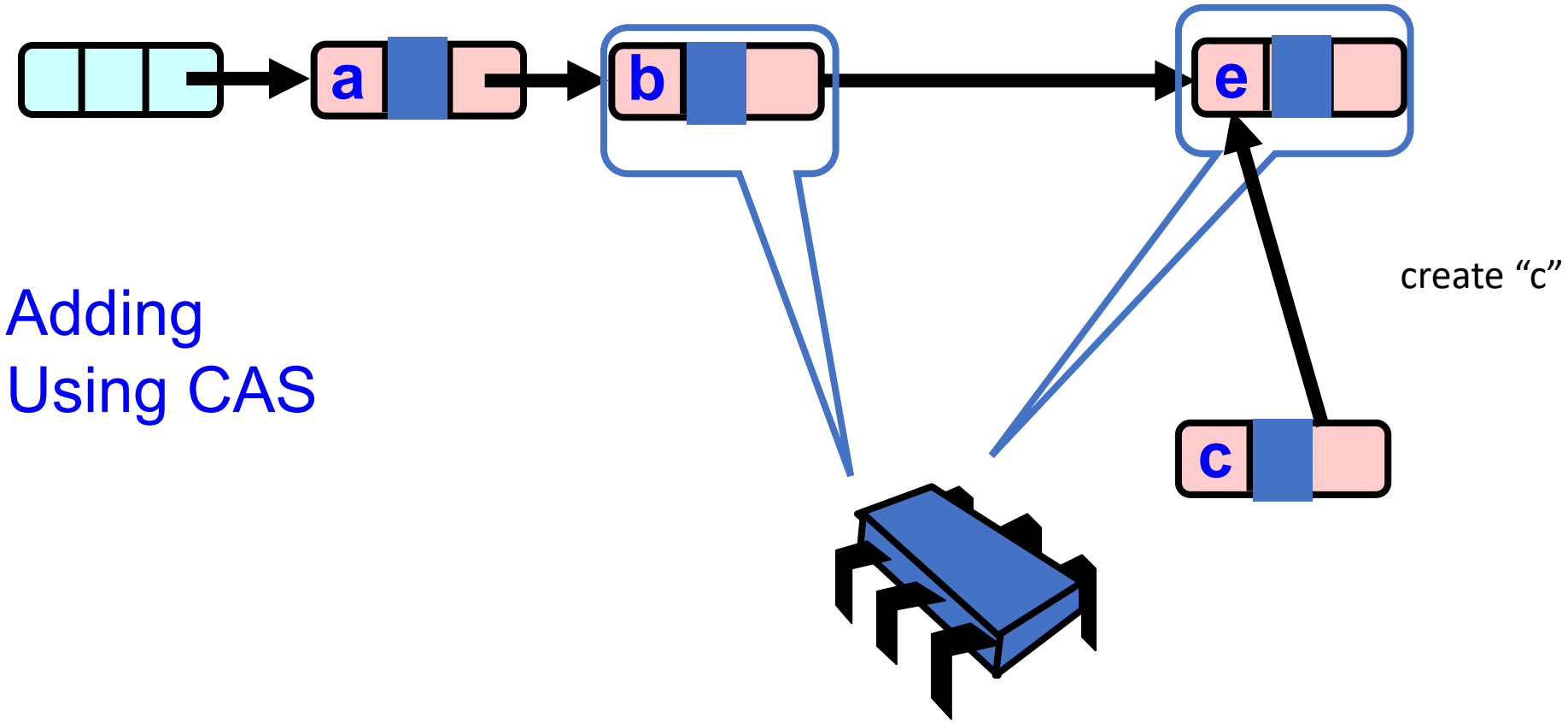
CAS(b.next, e, c);

Find the location

Cache your insertion point!

b.next == e

*notion is being abused here: e and c will be node **



Adding
Using CAS

create “c”

Lock-free Lists

Only insert if your insertion point is valid!

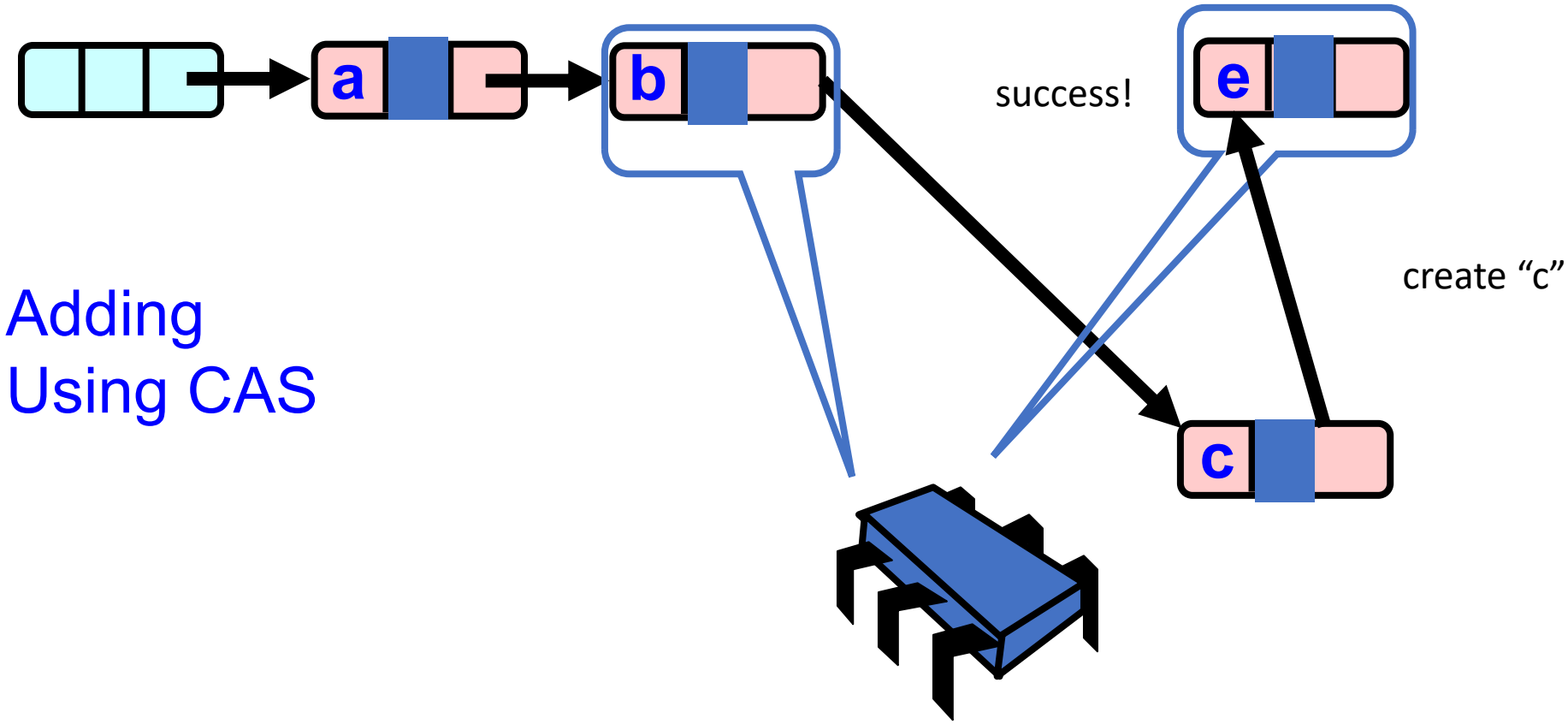
CAS(b.next, e, c);

Find the location

Cache your insertion point!

b.next == e

*notion is being abused here: e and c will be node **



Adding
Using CAS

Lock-free Lists

Only insert if your insertion point is valid!

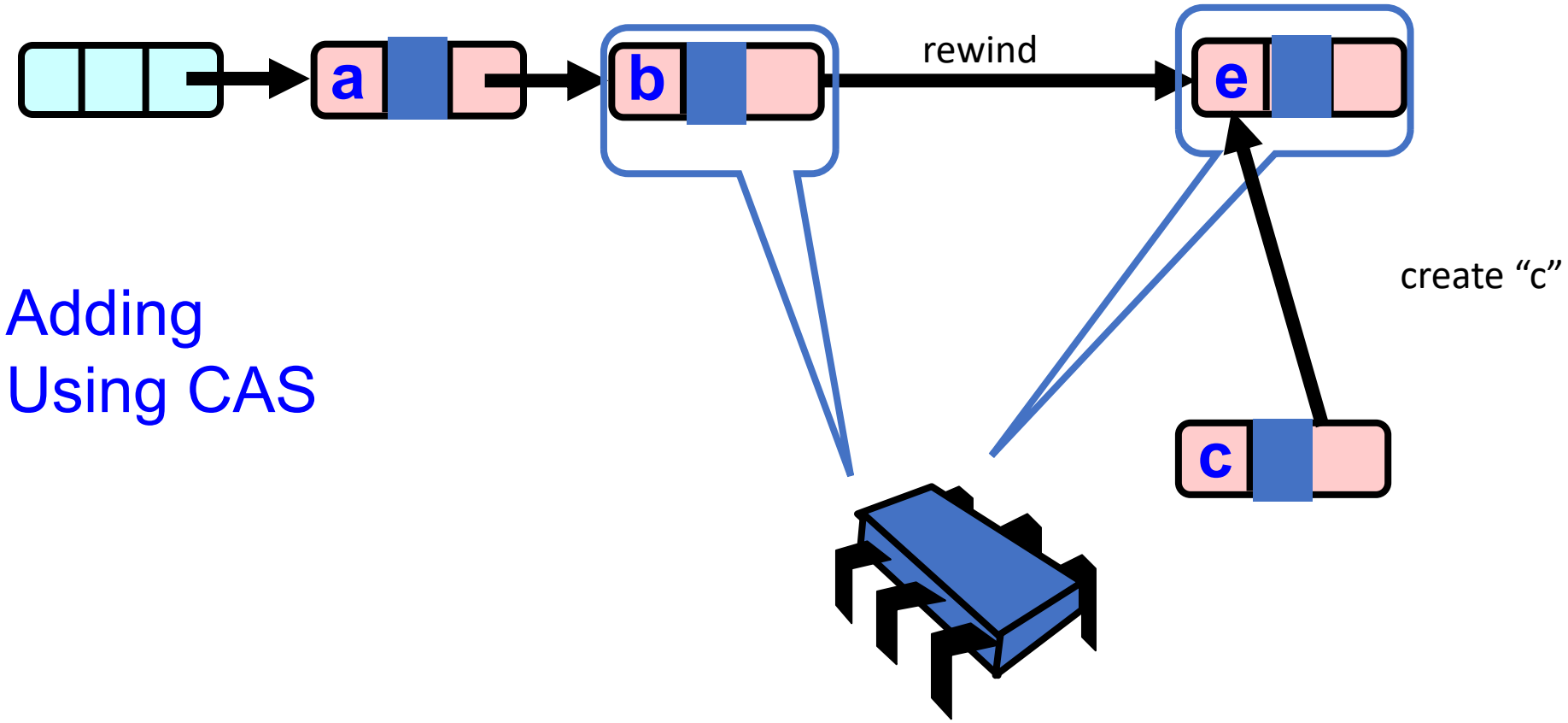
CAS(b.next, e, c);

Find the location

Cache your insertion point!

b.next == e

*notion is being abused here: e and c will be node **



Adding
Using CAS

Lock-free Lists

Only insert if your insertion point is valid!

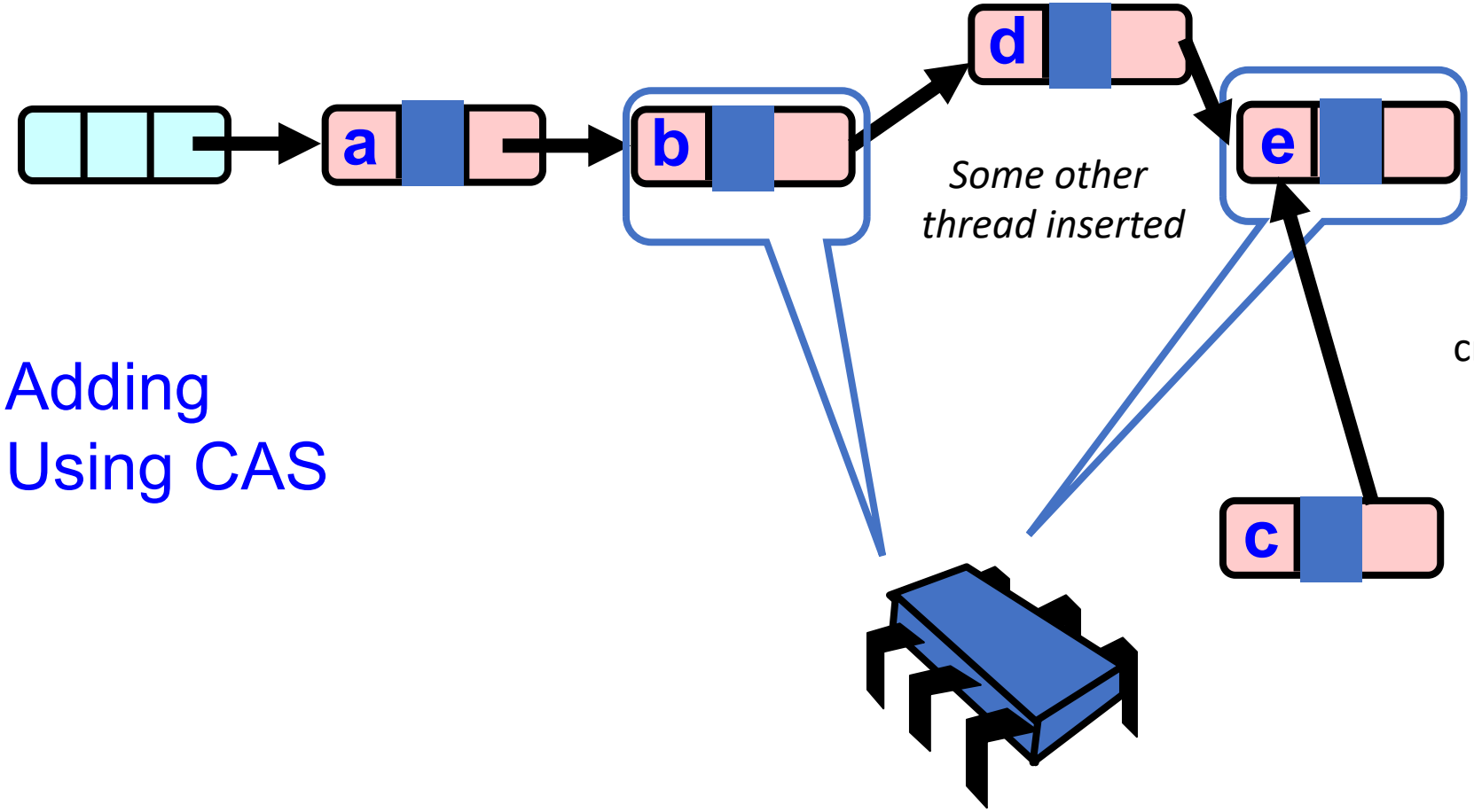
CAS(b.next, e, c);

Find the location

Cache your insertion point!

b.next == e

*notion is being abused here: e and c will be node **



Adding Using CAS

Lock-free Lists

Only insert if your insertion point is valid!

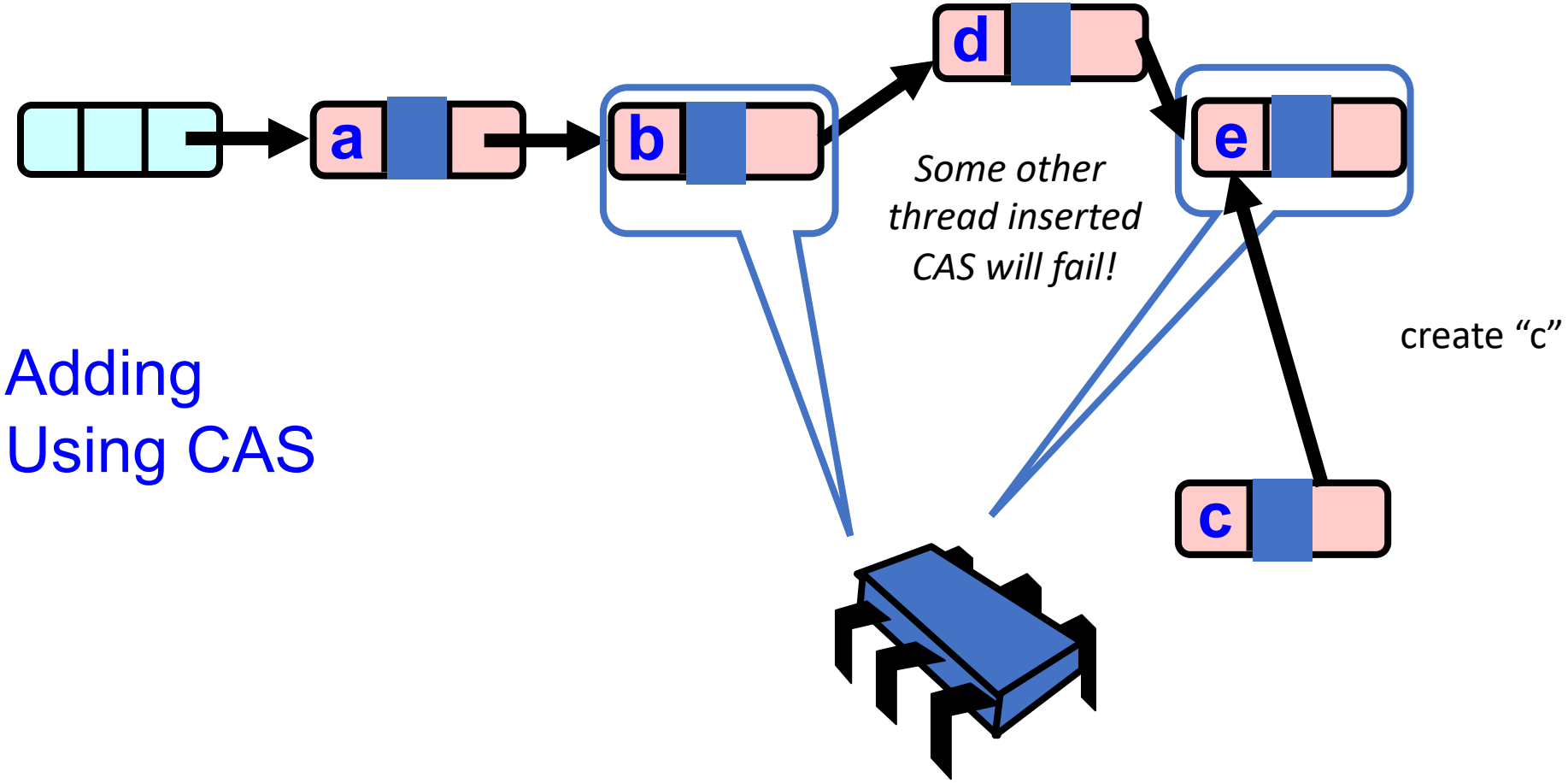
CAS(b.next, e, c);

Find the location

Cache your insertion point!

b.next == e

*notion is being abused here: e and c will be node **



Adding Using CAS

Lock-free Lists

Only insert if your insertion point is valid!

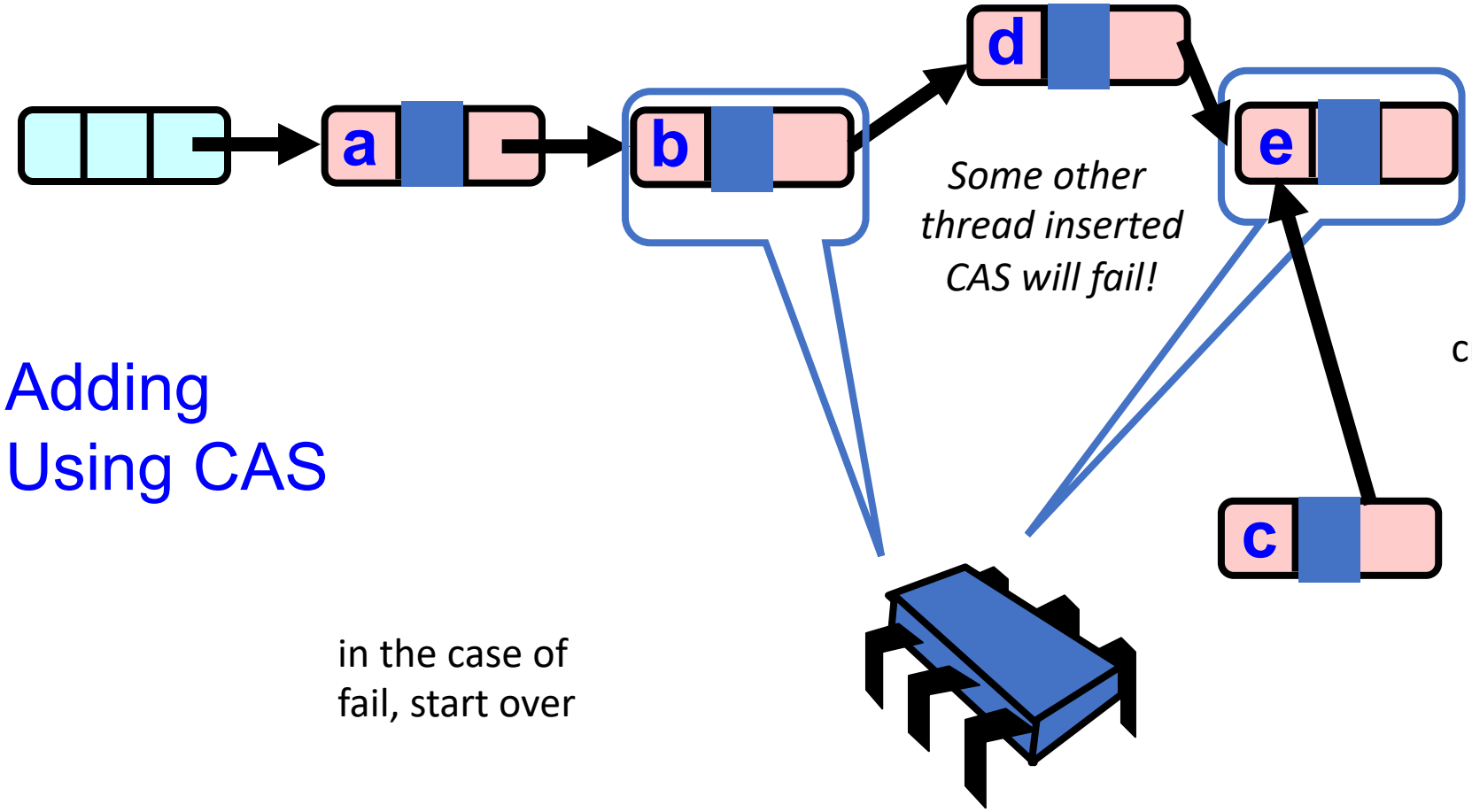
CAS(b.next, e, c);

Find the location

Cache your insertion point!

b.next == e

*notion is being abused here: e and c will be node **



Adding Using CAS

in the case of fail, start over

Further considerations

- need to include “valid bit” in compare and swap to make sure the node is still valid
- Can ”pack the bit” into the address (there will always be room because addresses are byte addressable, and addresses 8 bytes)
- More details in the book!

Schedule

- Module 4 introduction

- Barriers
  - Specification
  - Implementation

Schedule

- **Module 4 introduction**

- Barriers
  - Specification
  - Implementation

Reasoning about concurrency

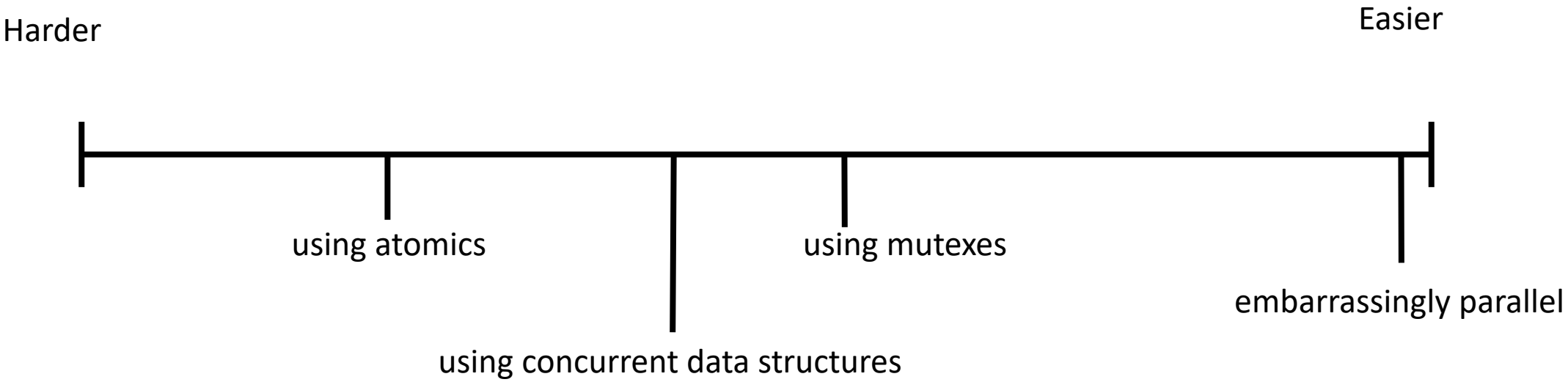
Mental model of concurrency

- Functional
  - **Interleavings** - events from different threads can interleave
  - **Atomicity** - what events are indivisible?
  - **Specifications** - how can we create useful abstractions (mutexes, concurrent data structures)

- Performance
  - **Increase parallelism** - judicious use of mutexes, load balancing
  - **Cache behaviors** - threads should try to utilize their own cache lines
  - **Operating system** - yielding/sleeping threads
  - **Architectural details** - instruction-level parallelism

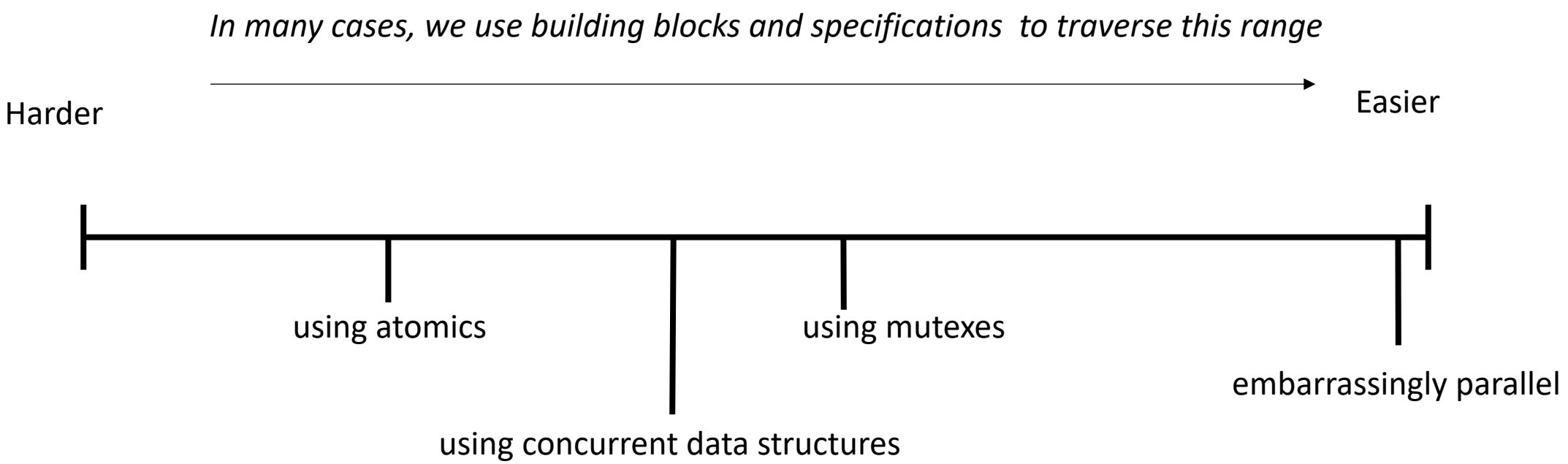
Reasoning about concurrency

- Depending on your needs, programs become more/less complex to reason about.



Reasoning about concurrency

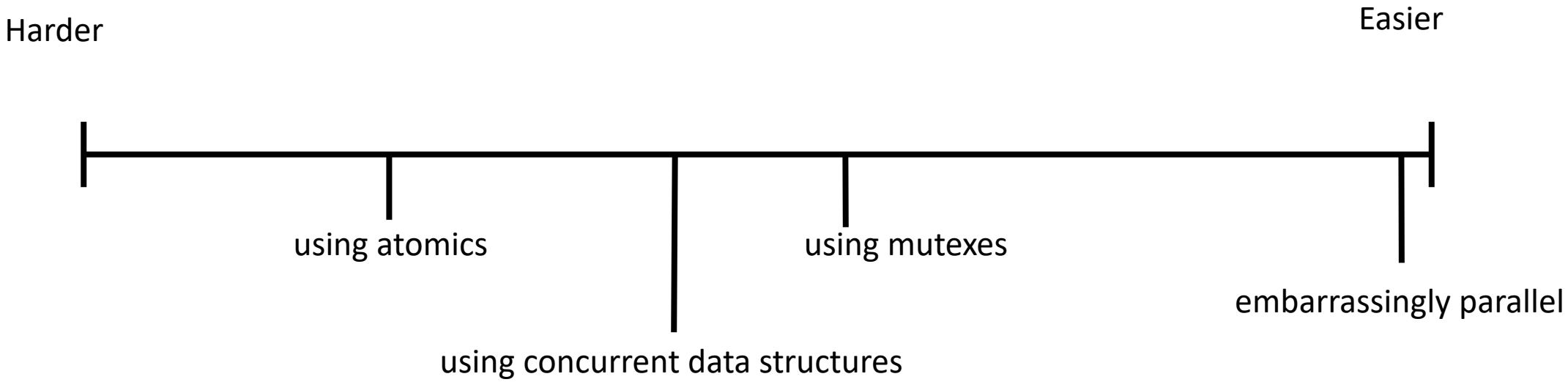
- Depending on your needs, programs become more/less complex to reason about.



Reasoning about concurrency

- Depending on your needs, programs become more/less complex to reason about.

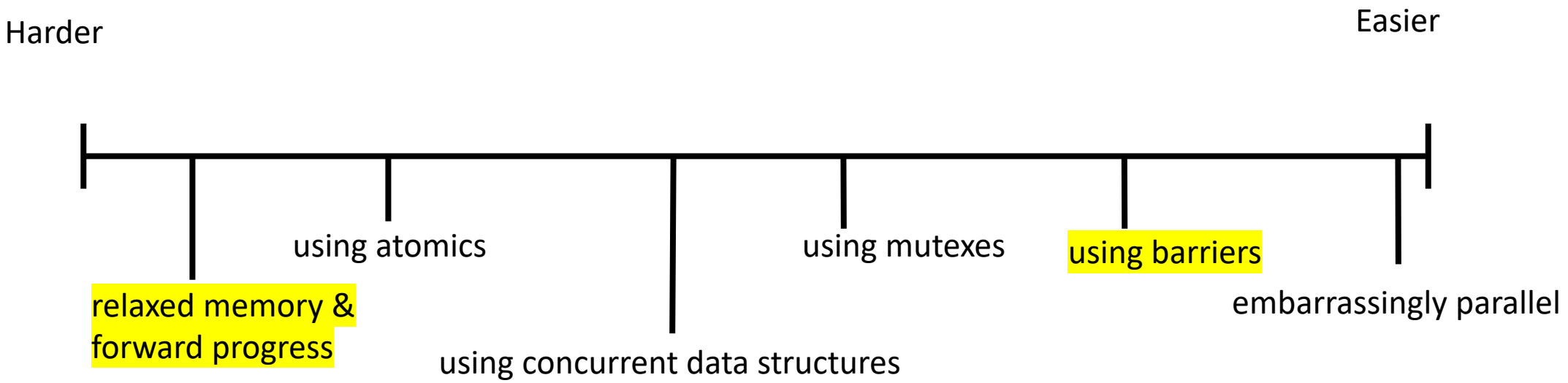
To get a more complete picture, we will fill in some of the gaps here



Reasoning about concurrency

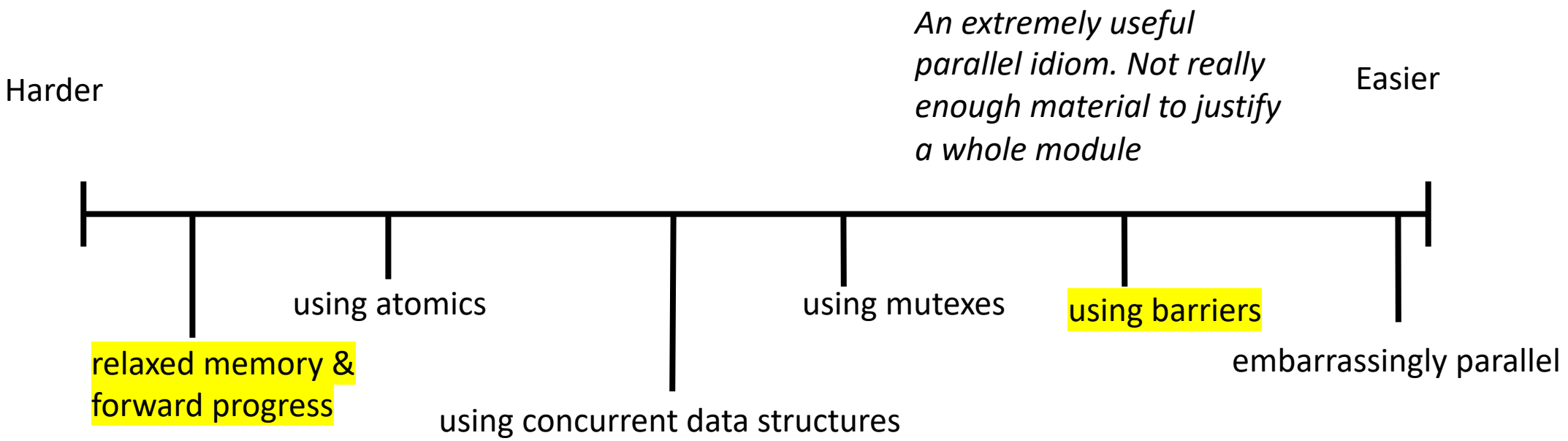
- Depending on your needs, programs become more/less complex to reason about.

To get a more complete picture, we will fill in some of the gaps here



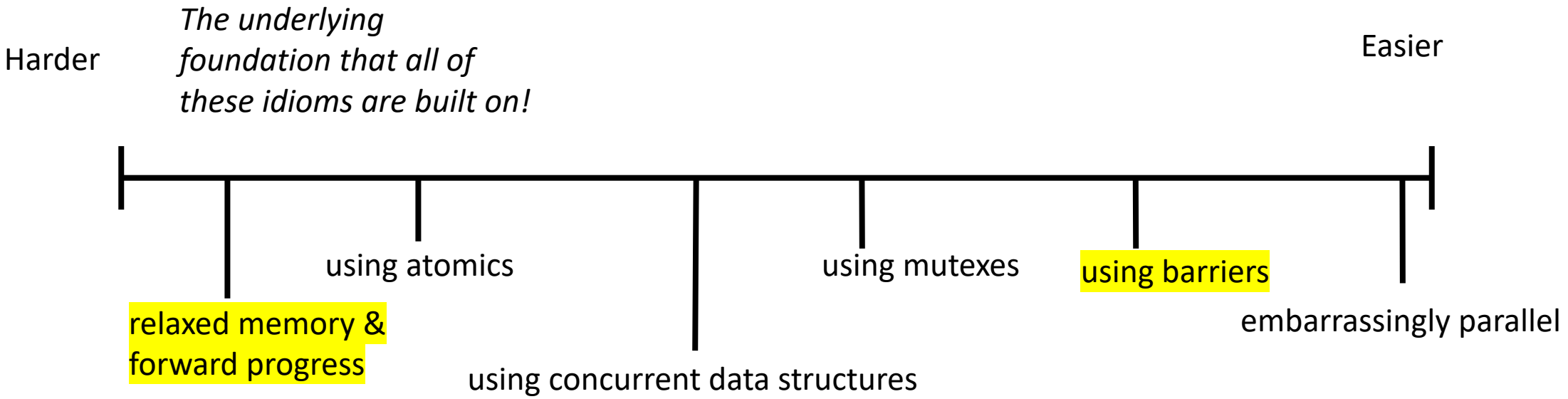
Reasoning about concurrency

- Depending on your needs, programs become more/less complex to reason about.



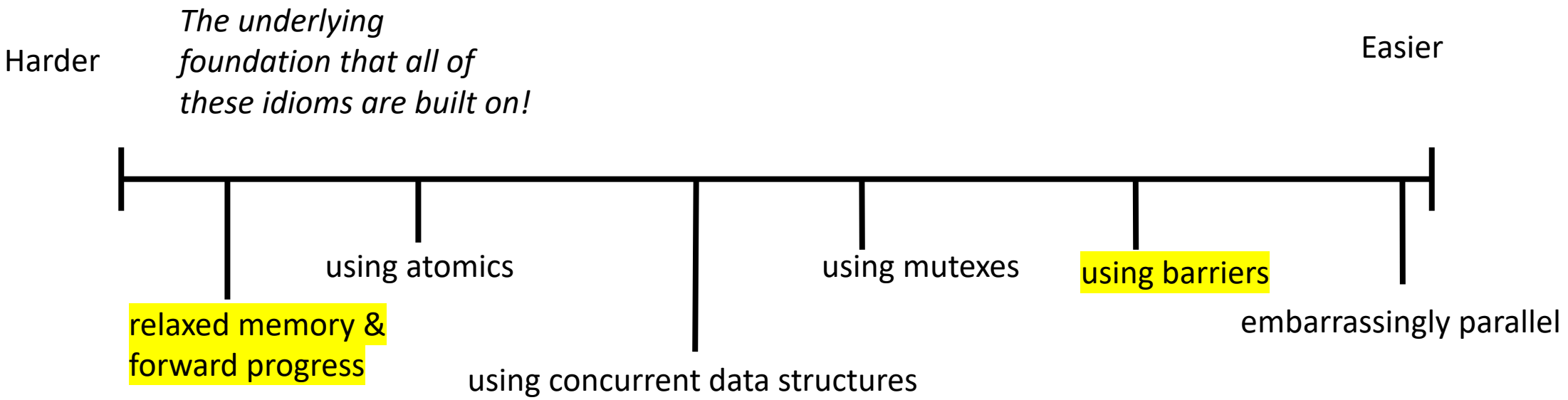
Reasoning about concurrency

- Depending on your needs, programs become more/less complex to reason about.



Reasoning about concurrency

- **On newer architectures, you may only get specifications about the relaxed memory and progress. It is up to the user to implement their own atomics, mutexes, concurrent data structures, etc!**

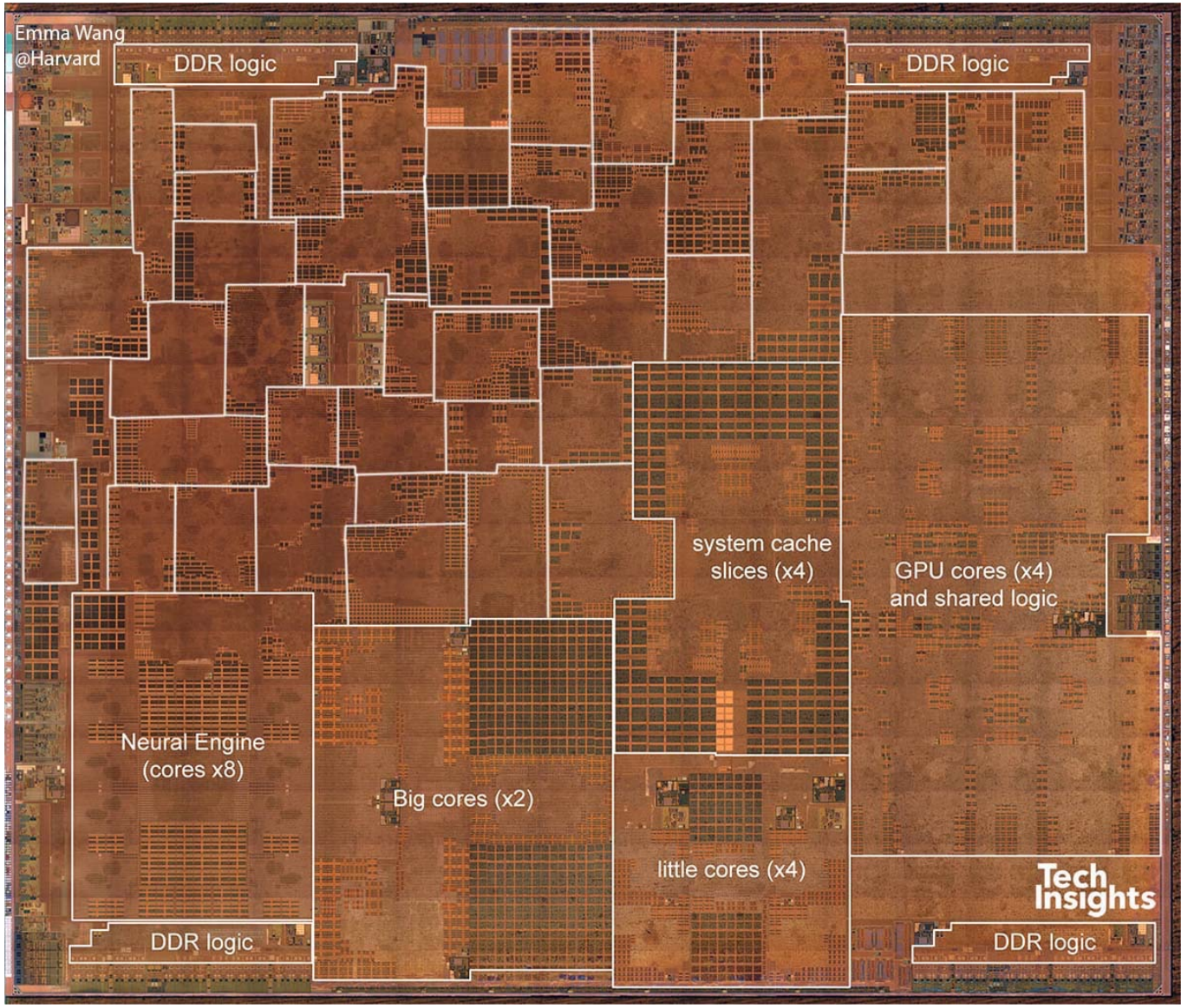


Modern chips

- From David Brooks lab at Harvard:

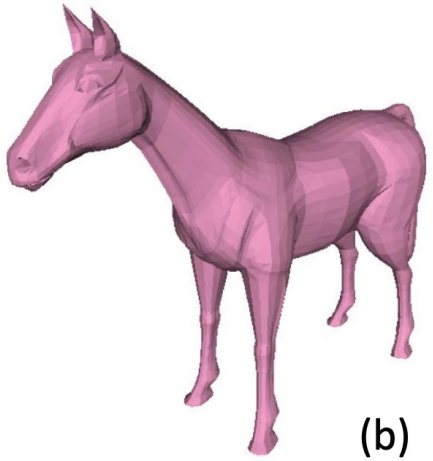
  http://vlsiarch.eecs.harvard.edu/research/accelerators/die-photo-analysis/

- GPUs/accelerators will have different guarantees w.r.t. atomics and memory orderings



Historical issues on Nvidia GPUs

(a)

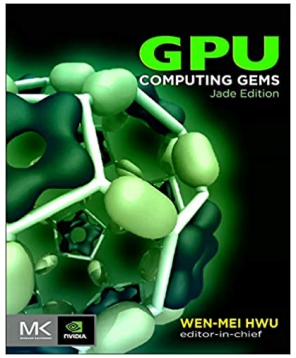


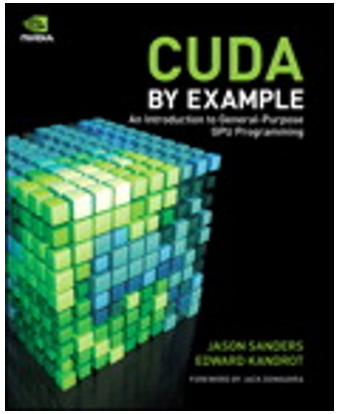
(b)

(c)

(d)

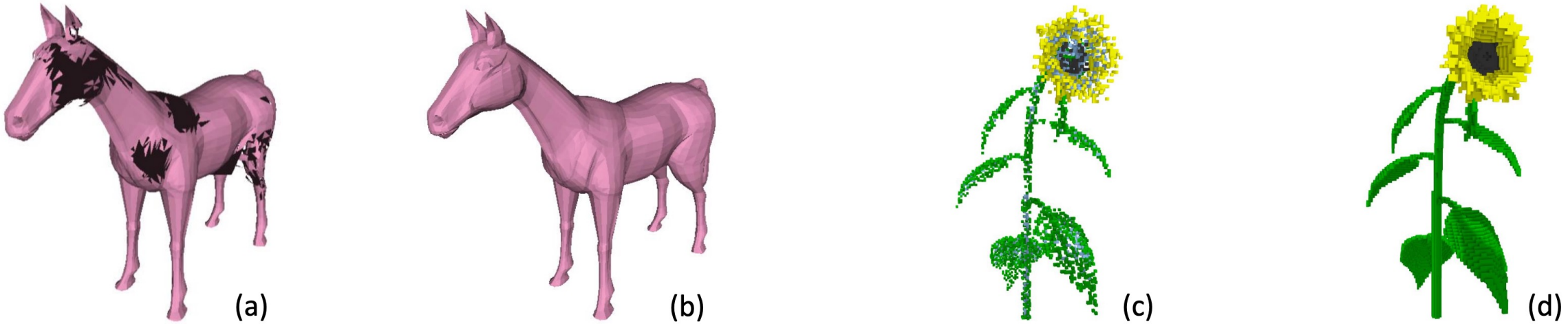
Issue implementing a concurrent data structure related to memory orderings





Issue implementing a mutex relating to memory orderings

Historical issues on Nvidia GPUs



In both cases, they had reasoned about their implementations exactly how we have! The issues came from lower in the stack: memory orderings

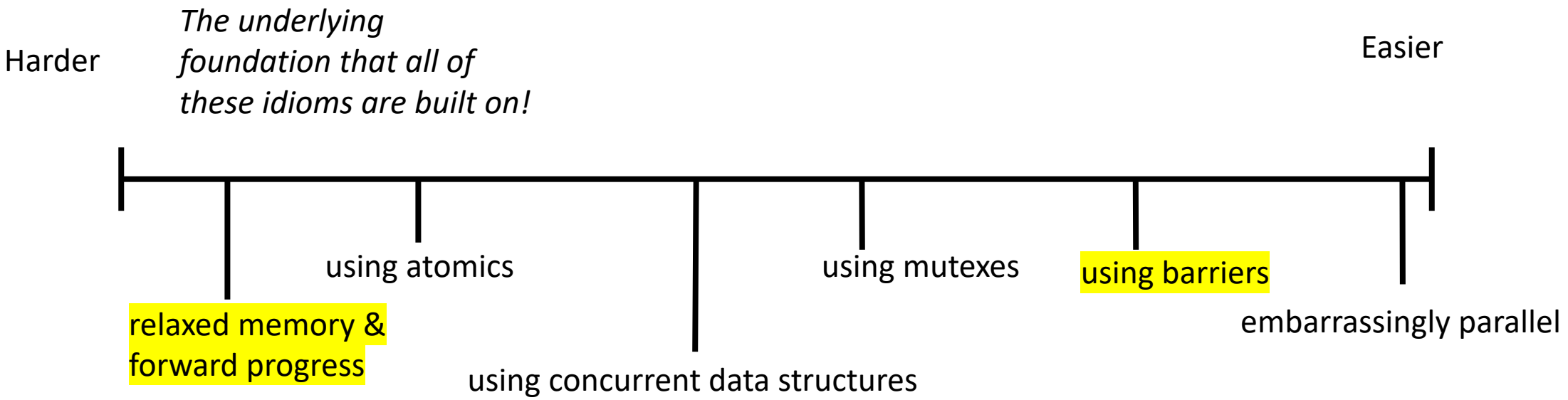
running a mutex on an iPad GPU?

This one has to do with functional properties of the scheduler

video demo

Reasoning about concurrency

- **On newer architectures, you may only get specifications about the relaxed memory and progress. It is up to the user to implement their own atomics, mutexes, concurrent data structures, etc!**



Schedule

- Module 4 introduction

- **Barriers**
  - **Specification**
  - Implementation

Barriers

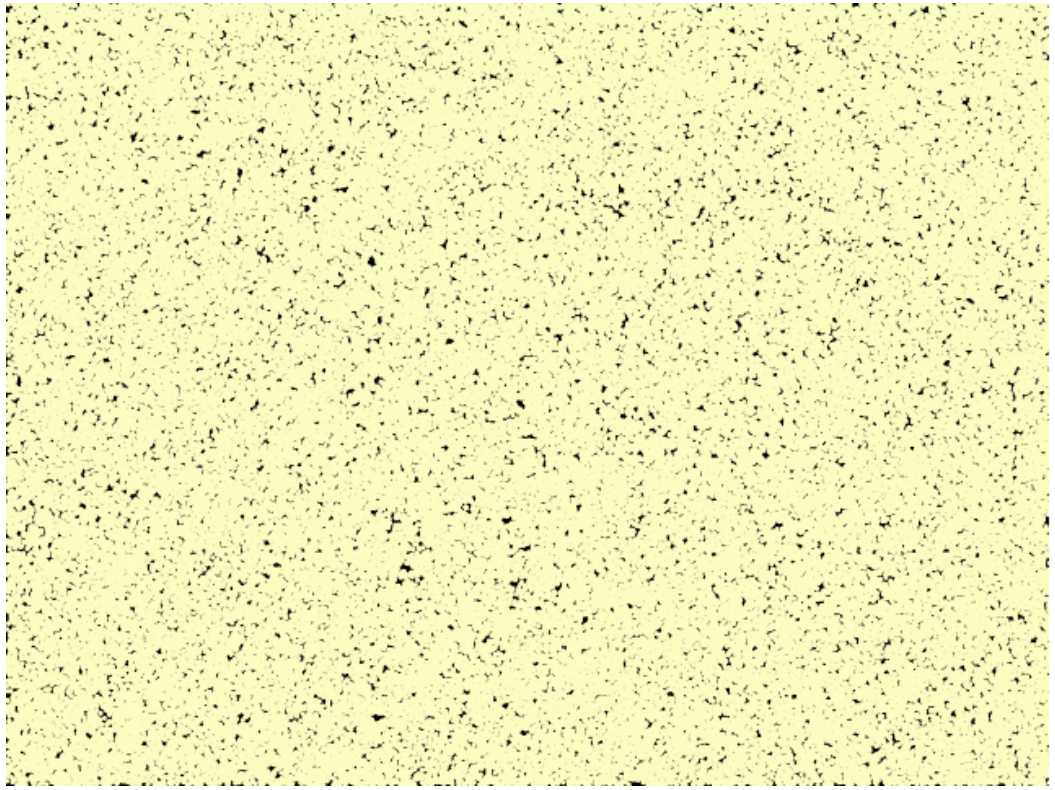
- Why do barriers fit into this module: “Reasoning About Parallel Computing”?
  - Relaxed Memory Models make reasoning about parallel computing HARD
  - Barriers make it EASIER (at the cost of performance potentially)

- A barrier is a concurrent object (like a mutex):
  - Only one method: `barrier` (called `await` in the book)

- Separates computational phases

Barrier Examples

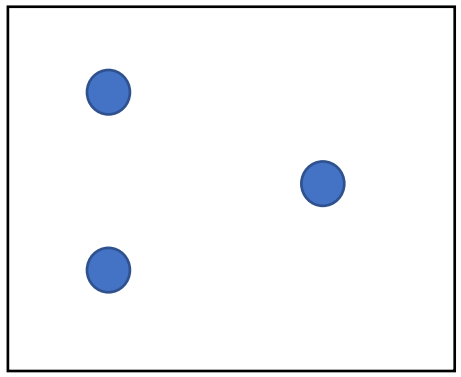
My current favorite: particle simulation



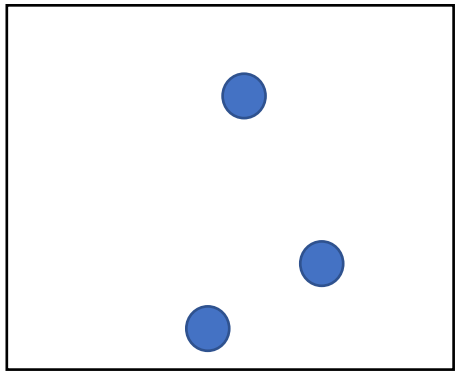
by Yanwen Xu

Barrier Examples

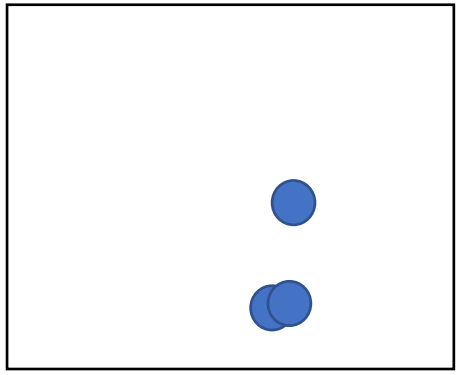
My current favorite: particle simulation



time = 0



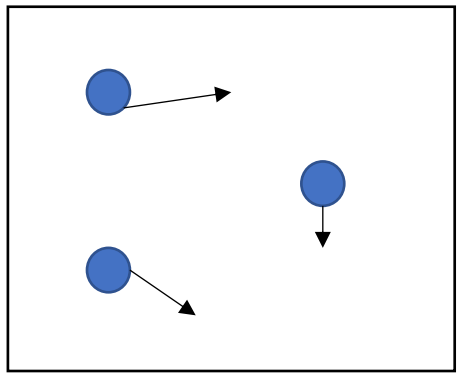
time = 1



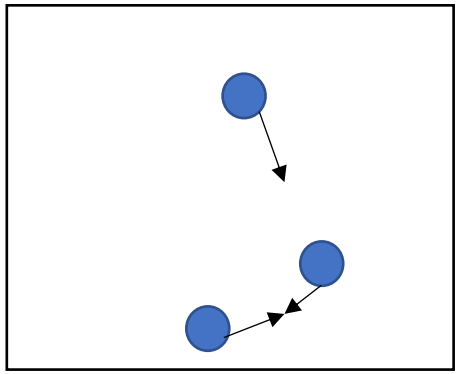
time = 2

Barrier Examples

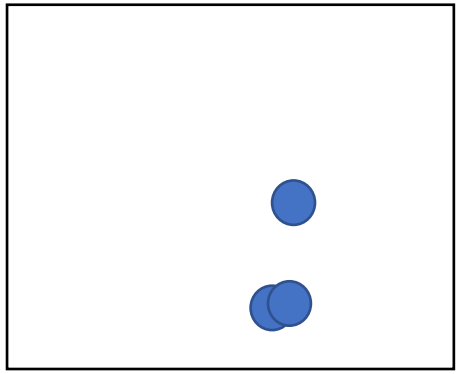
My current favorite: particle simulation



time = 0



time = 1

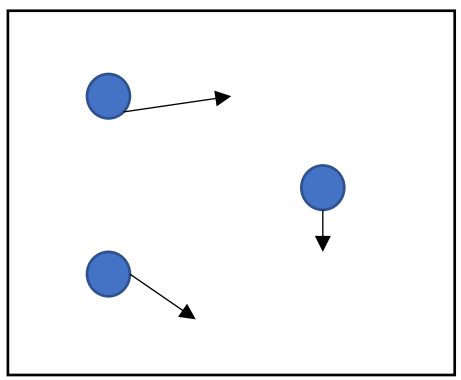


time = 2

at each time, compute
new positions for each particle
(in parallel)

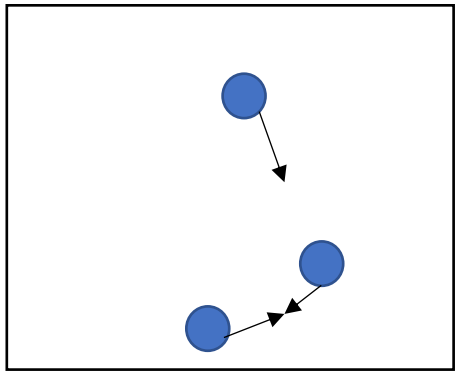
Barrier Examples

My current favorite: particle simulation



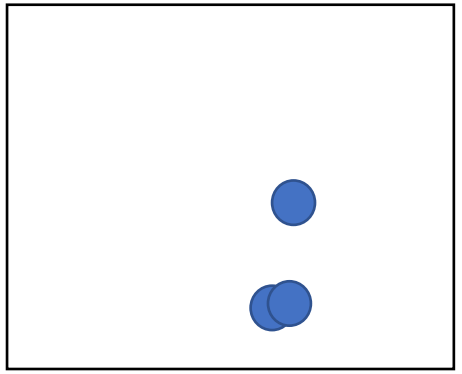
time = 0

```
barrier();
```



time = 1

```
barrier();
```



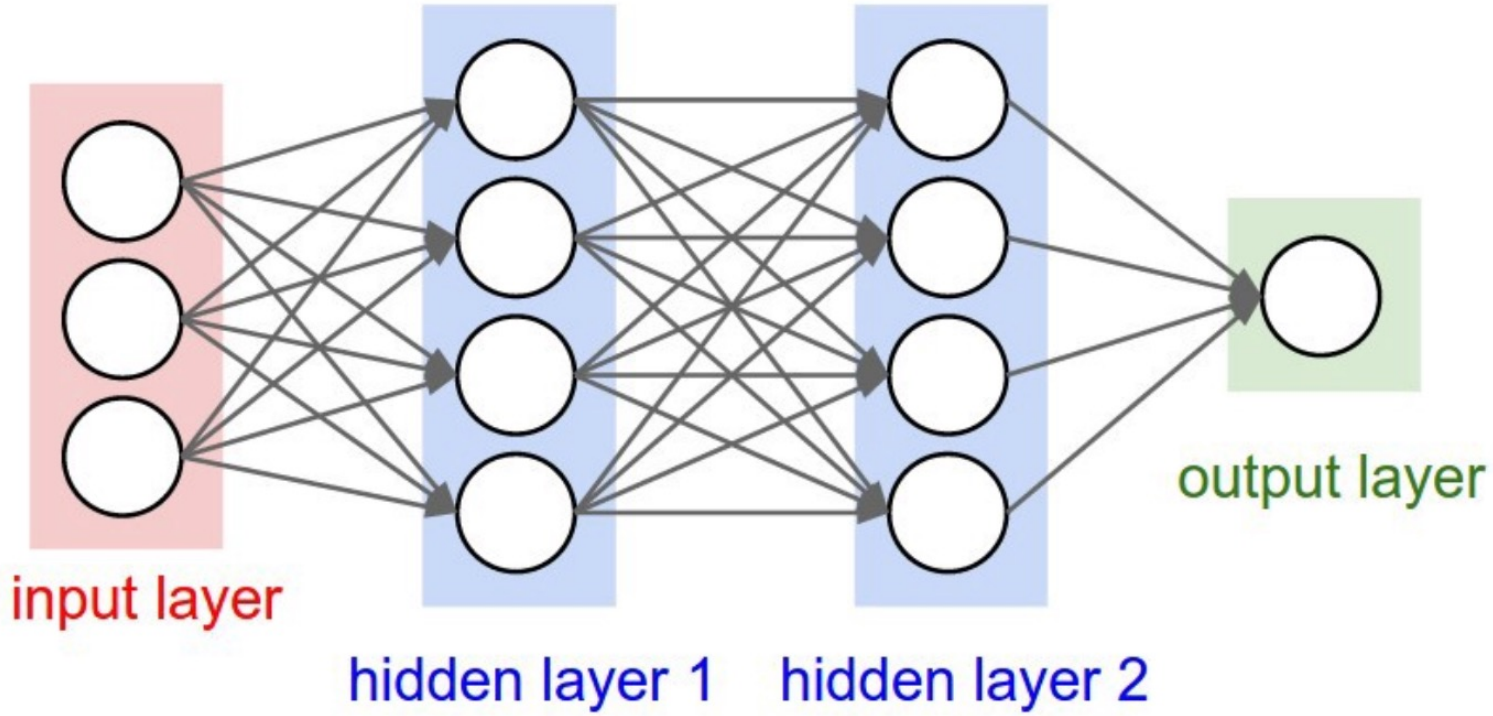
time = 2

at each time, compute
new positions for each particle
(in parallel)

But you need to wait for all particles to be
computed before starting the next time step

Barrier Examples

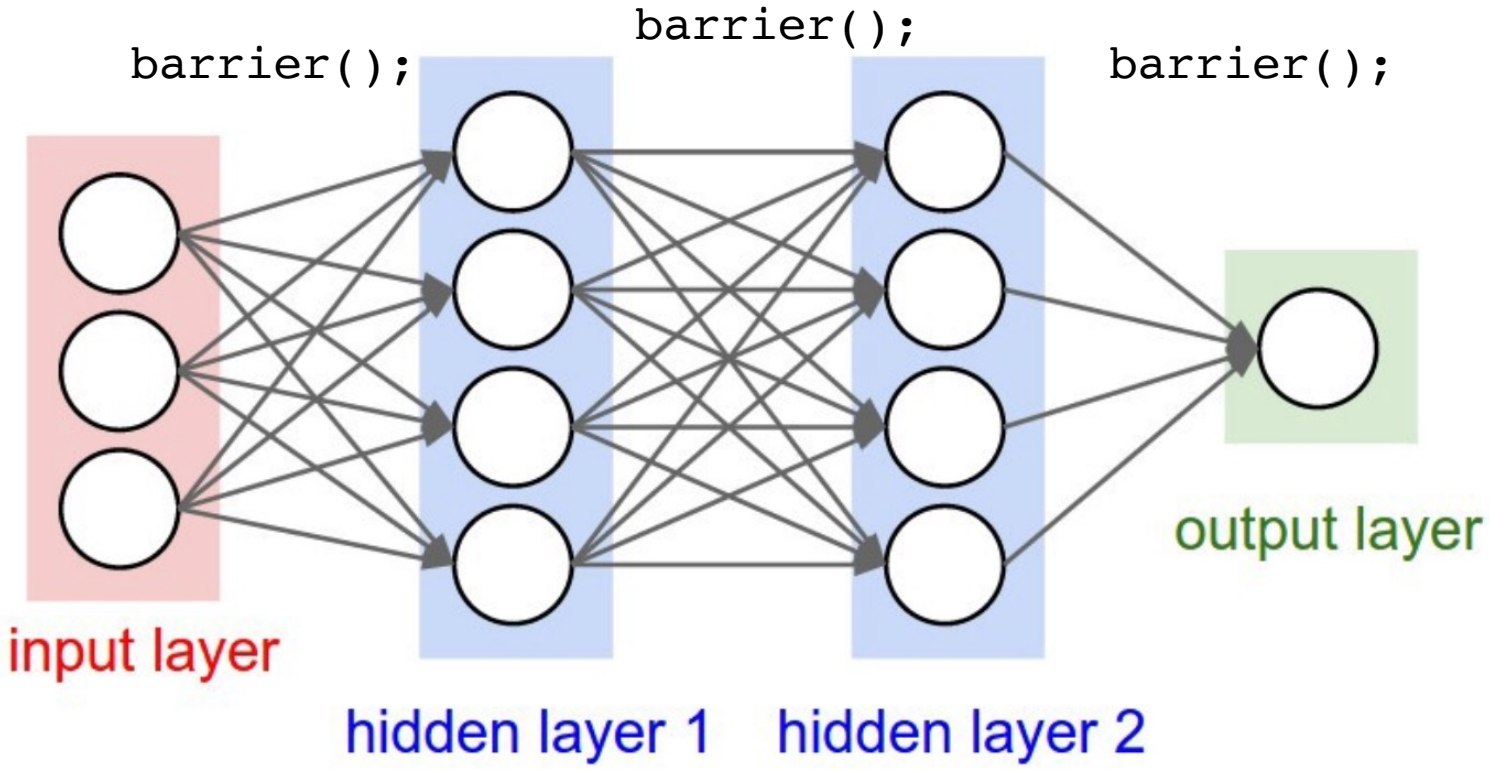
- Deep neural networks



from http://cs231n.stanford.edu/

Barrier Examples

- Deep neural networks



from http://cs231n.stanford.edu/

Barriers

- Intuition: threads stop and wait for each other:
  - Threads ***arrive*** at the barrier
  - Threads ***wait*** at the barrier
  - Threads ***leave*** the barrier once all other threads have arrived

Barriers

- Intuition: threads stop and wait for each other:
  - Threads ***arrive*** at the barrier
  - Threads ***wait*** at the barrier
  - Threads ***leave*** the barrier once all other threads have arrived

Example: say there are 4 threads:

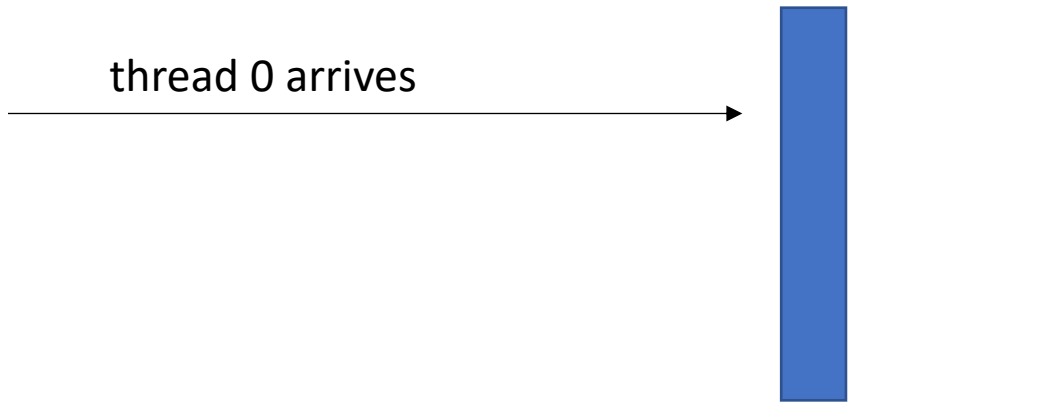
```
barrier();
```

Barriers

- Intuition: threads stop and wait for each other:
  - Threads ***arrive*** at the barrier
  - Threads ***wait*** at the barrier
  - Threads ***leave*** the barrier once all other threads have arrived

Example: say there are 4 threads:

```
barrier();
```

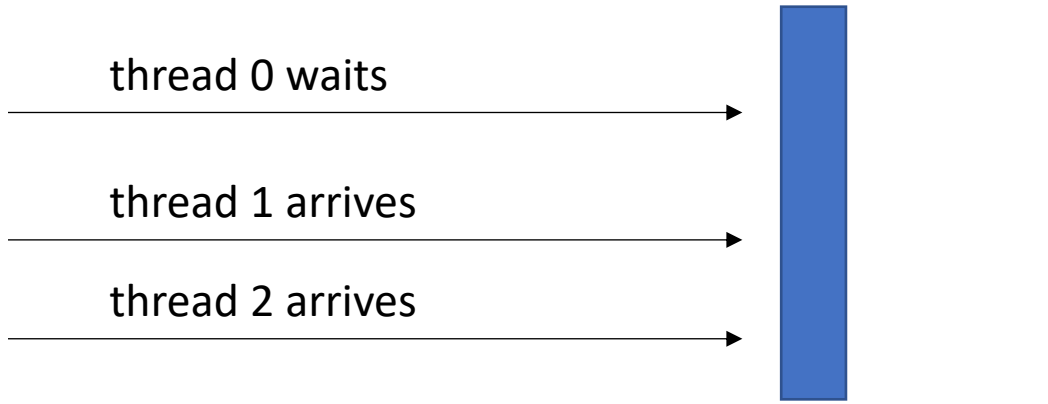


Barriers

- Intuition: threads stop and wait for each other:
  - Threads ***arrive*** at the barrier
  - Threads ***wait*** at the barrier
  - Threads ***leave*** the barrier once all other threads have arrived

Example: say there are 4 threads:

```
barrier();
```

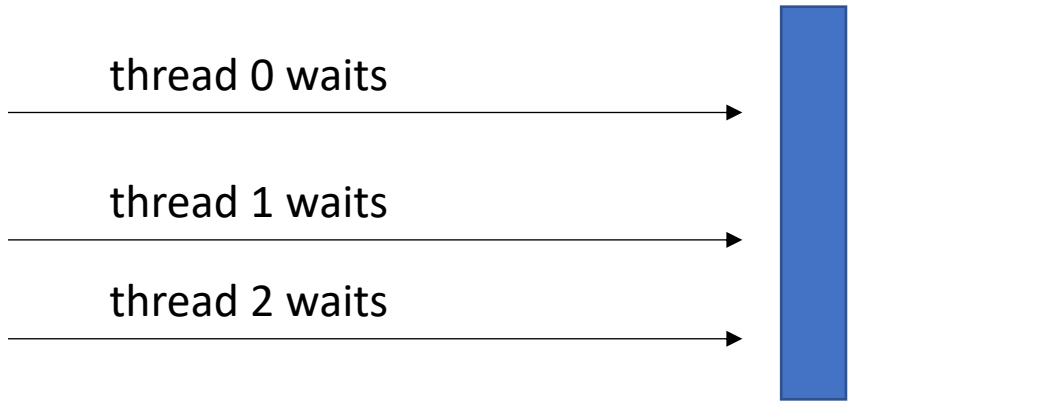


Barriers

- Intuition: threads stop and wait for each other:
  - Threads ***arrive*** at the barrier
  - Threads ***wait*** at the barrier
  - Threads ***leave*** the barrier once all other threads have arrived

Example: say there are 4 threads:

```
barrier();
```

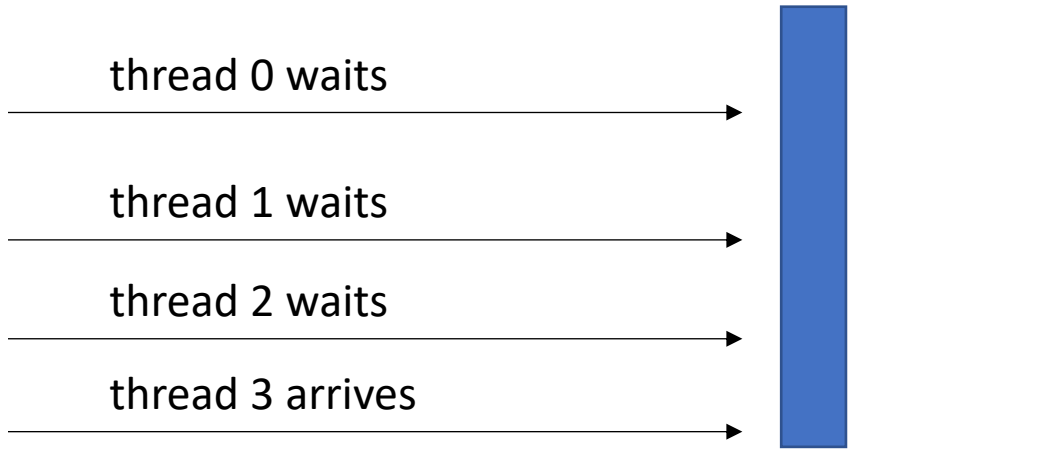


Barriers

- Intuition: threads stop and wait for each other:
  - Threads ***arrive*** at the barrier
  - Threads ***wait*** at the barrier
  - Threads ***leave*** the barrier once all other threads have arrived

Example: say there are 4 threads:

```
barrier();
```

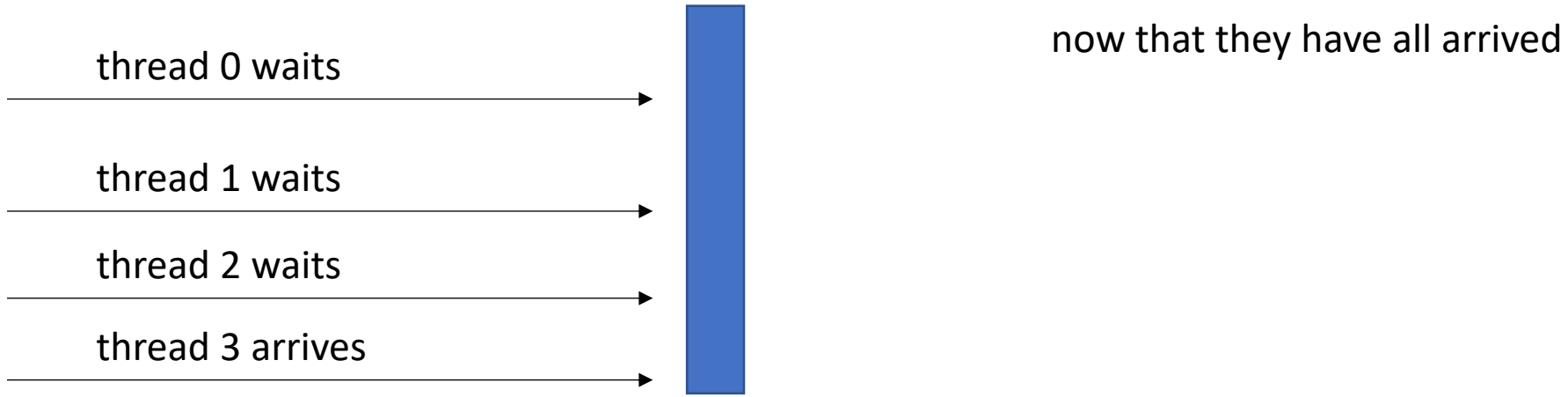


Barriers

- Intuition: threads stop and wait for each other:
  - Threads ***arrive*** at the barrier
  - Threads ***wait*** at the barrier
  - Threads ***leave*** the barrier once all other threads have arrived

Example: say there are 4 threads:

```
barrier();
```

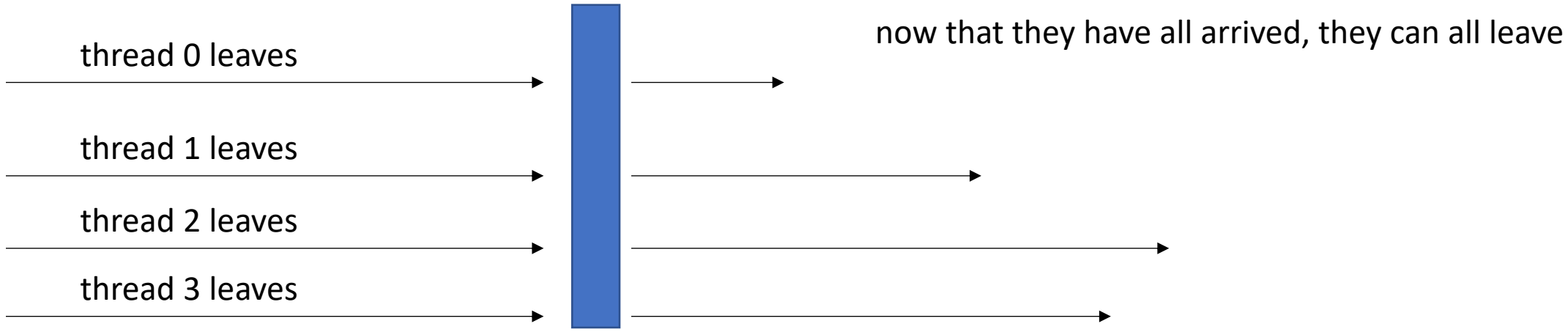


Barriers

- Intuition: threads stop and wait for each other:
  - Threads ***arrive*** at the barrier
  - Threads ***wait*** at the barrier
  - Threads ***leave*** the barrier once all other threads have arrived

Example: say there are 4 threads:

```
barrier();
```



A more formal specification

Given a global barrier `B`
and a global memory location x where
initially *x = 0;

First, what would we expect
var to be after this program?

Thread 0:

```
*x = 1;
B.barrier();
```

Thread 1:

```
B.barrier();
var = *x;
```

A more formal specification

Given a global barrier `B`
and a global memory location x where initially *x = 0;

Thread 0:

```
*x = 1;
B.barrier();
```

Thread 1:

```
B.barrier();
var = *x;
```

gives an event:
barrier arrive

A more formal specification

Given a global barrier `B`
and a global memory location x where
initially *x = 0;

Thread 0:

```
*x = 1;
B.barrier();
```

Thread 1:

```
B.barrier();
var = *x;
```

gives an event:
barrier arrive

barrier arrive needs to wait for all threads
to arrive (similar to how a mutex request must wait for
another to release)

A more formal specification

Given a global barrier `B`
and a global memory location x where initially *x = 0;

Thread 0:

```
*x = 1;
B.barrier();
```

Thread 1:

```
B.barrier();
var = *x;
```

A more formal specification

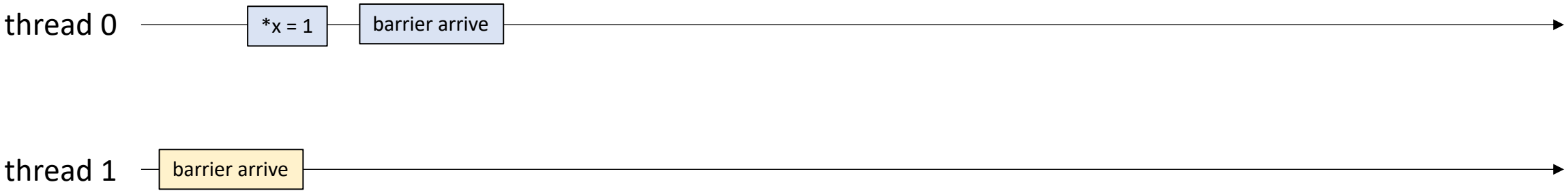
Given a global barrier `B`
and a global memory location x where initially *x = 0;

Thread 0:

```
*x = 1;
B.barrier();
```

Thread 1:

```
B.barrier();
var = *x;
```



A more formal specification

Given a global barrier `B`
and a global memory location x where
initially *x = 0;

Thread 0:

```
*x = 1;
B.barrier();
```

Thread 1:

```
B.barrier();
var = *x;
```

now that all threads have arrived:
They can leave (1 event at the same time)

A more formal specification

Given a global barrier `B`
and a global memory location x where initially *x = 0;

Thread 0:

```
*x = 1;
B.barrier();
```

Thread 1:

```
B.barrier();
var = *x;
```

This finishes the barrier execution

A more formal specification

Given a global barrier `B`
and a global memory location x where initially *x = 0;

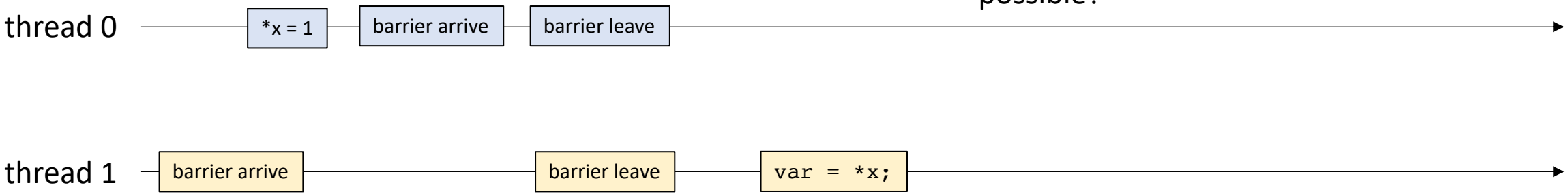
Thread 0:

```
*x = 1;
B.barrier();
```

Thread 1:

```
B.barrier();
var = *x;
```

what value must this read? Any other value possible?



One more example, assume initially `*x = *y = 0`

Thread 0:

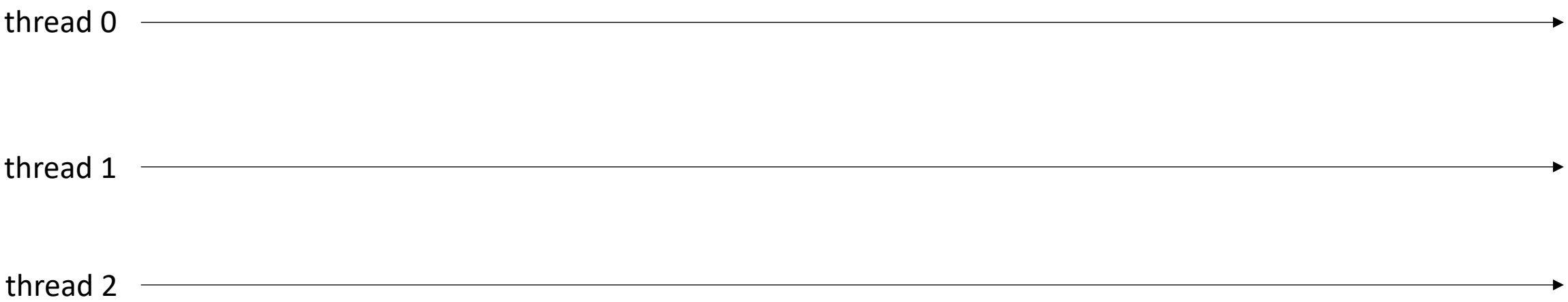
```
*x = 1;
B.barrier();
```

Thread 1:

```
*y = 2;
B.barrier();
```

Thread 2:

```
B.barrier();
var = *x + *y;
```



One more example, assume initially `*x = *y = 0`

Thread 0:

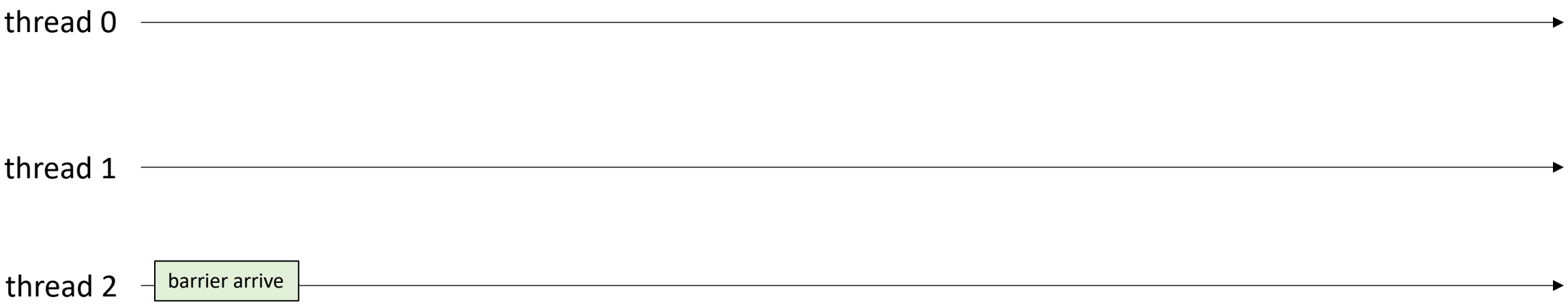
```
*x = 1;
B.barrier();
```

Thread 1:

```
*y = 2;
B.barrier();
```

Thread 2:

```
B.barrier();
var = *x + *y;
```



One more example, assume initially `*x = *y = 0`

Thread 0:

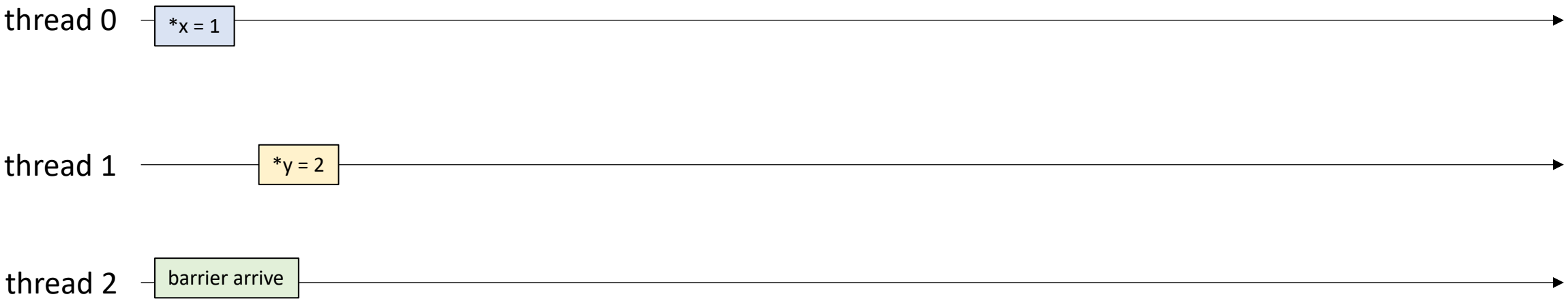
```
*x = 1;
B.barrier();
```

Thread 1:

```
*y = 2;
B.barrier();
```

Thread 2:

```
B.barrier();
var = *x + *y;
```



One more example, assume initially `*x = *y = 0`

Thread 0:

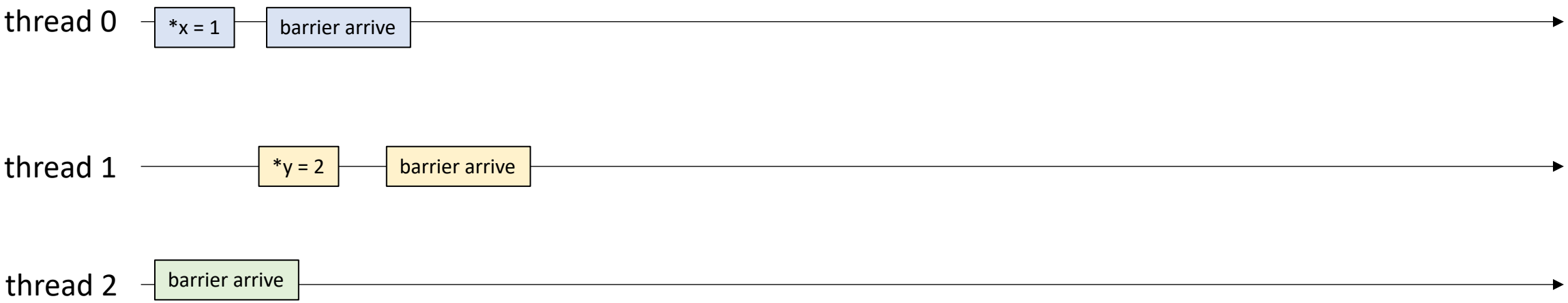
```
*x = 1;
B.barrier();
```

Thread 1:

```
*y = 2;
B.barrier();
```

Thread 2:

```
B.barrier();
var = *x + *y;
```



One more example, assume initially `*x = *y = 0`

Thread 0:

```
*x = 1;
B.barrier();
```

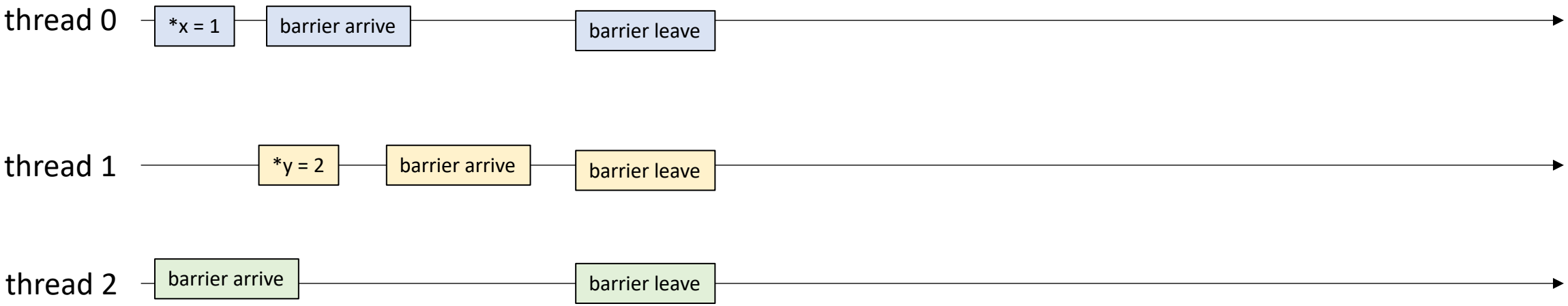
Thread 1:

```
*y = 2;
B.barrier();
```

Thread 2:

```
B.barrier();
var = *x + *y;
```

They've all arrived



One more example, assume initially `*x = *y = 0`

Thread 0:

```
*x = 1;
B.barrier();
```

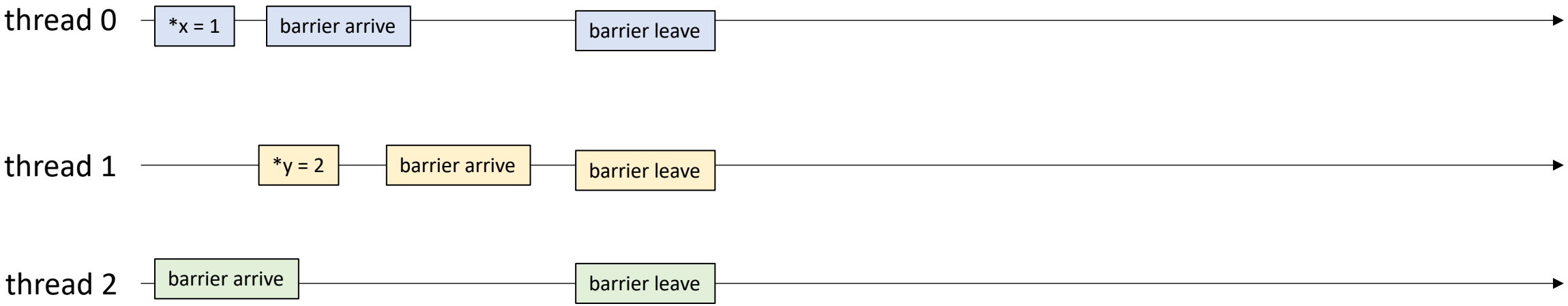
Thread 1:

```
*y = 2;
B.barrier();
```

Thread 2:

```
B.barrier();
var = *x + *y;
```

They've all arrived



One more example, assume initially `*x = *y = 0`

Thread 0:

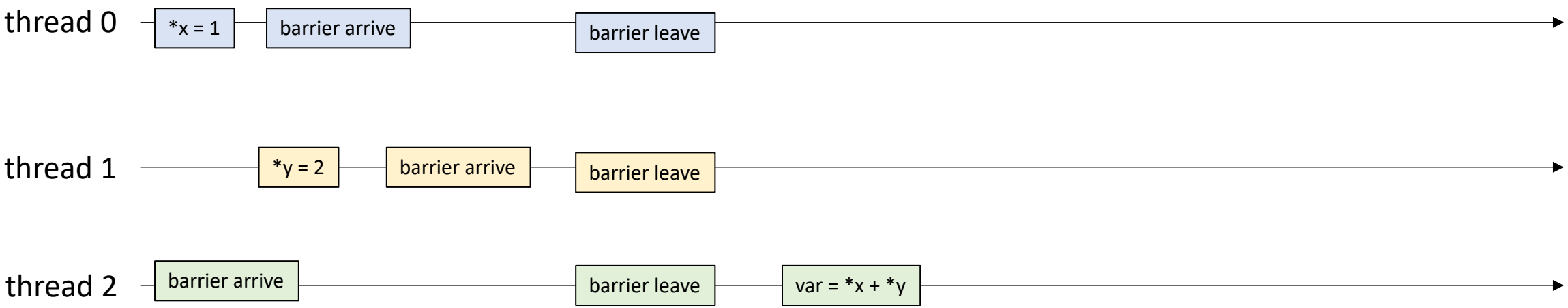
```
*x = 1;
B.barrier();
```

Thread 1:

```
*y = 2;
B.barrier();
```

Thread 2:

```
B.barrier();
var = *x + *y;
```



What is this guaranteed to be?

One more example, assume initially `*x = *y = 0`

Thread 0:

```
*x = 1;
B.barrier();
```

Thread 1:

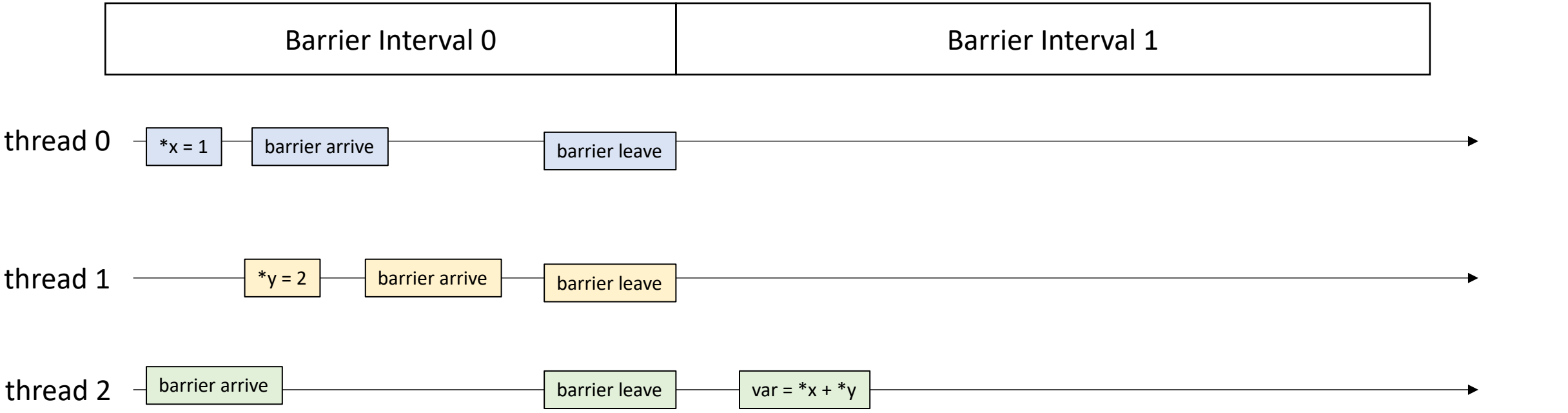
```
*y = 2;
B.barrier();
```

Thread 2:

```
B.barrier();
var = *x + *y;
```

sometimes called a *phase*

extending to the next *barrier leave*



Barriers

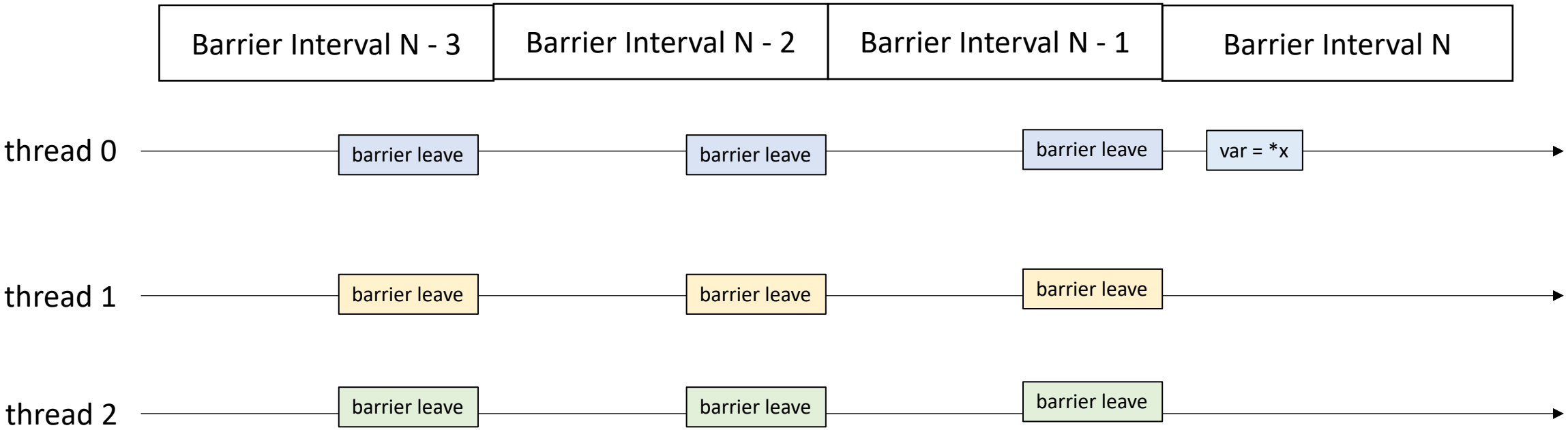
- Barrier Property:
  - If the only concurrent object you use in your program is a barrier (no mutexes, concurrent data-structures, atomic accesses)
  - If every barrier interval contains no data conflicts, then

    your program will be deterministic (only 1 outcome allowed)

  - much easier to reason about ☺

Assume we are reading from x

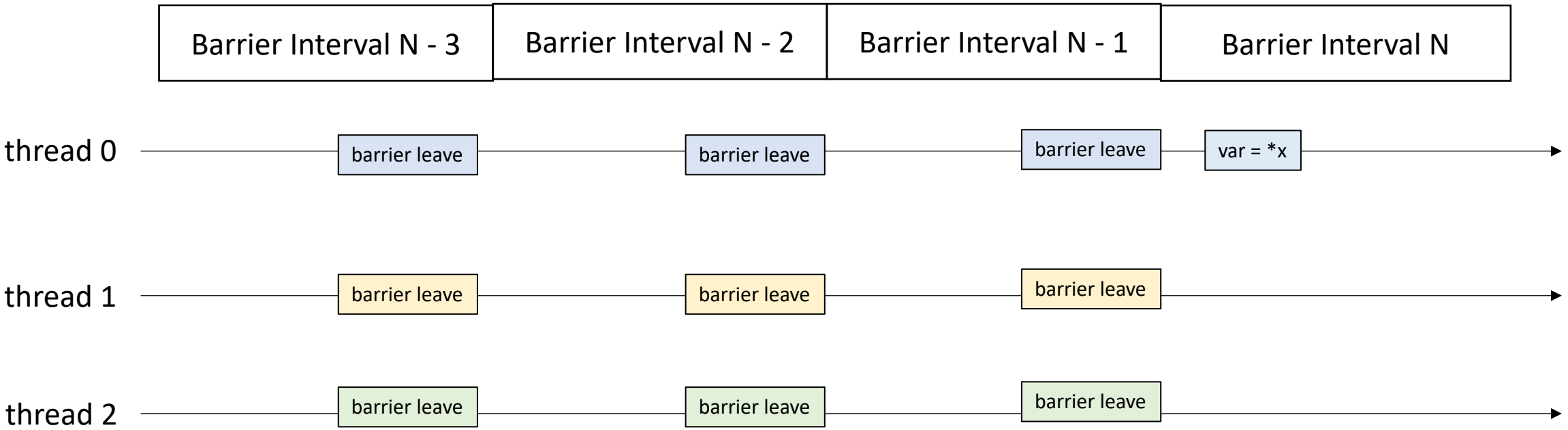
We are only allowed to return one possible value



no data conflicts means that x is written to at most once per barrier interval

Assume we are reading from x

We are only allowed to return one possible value

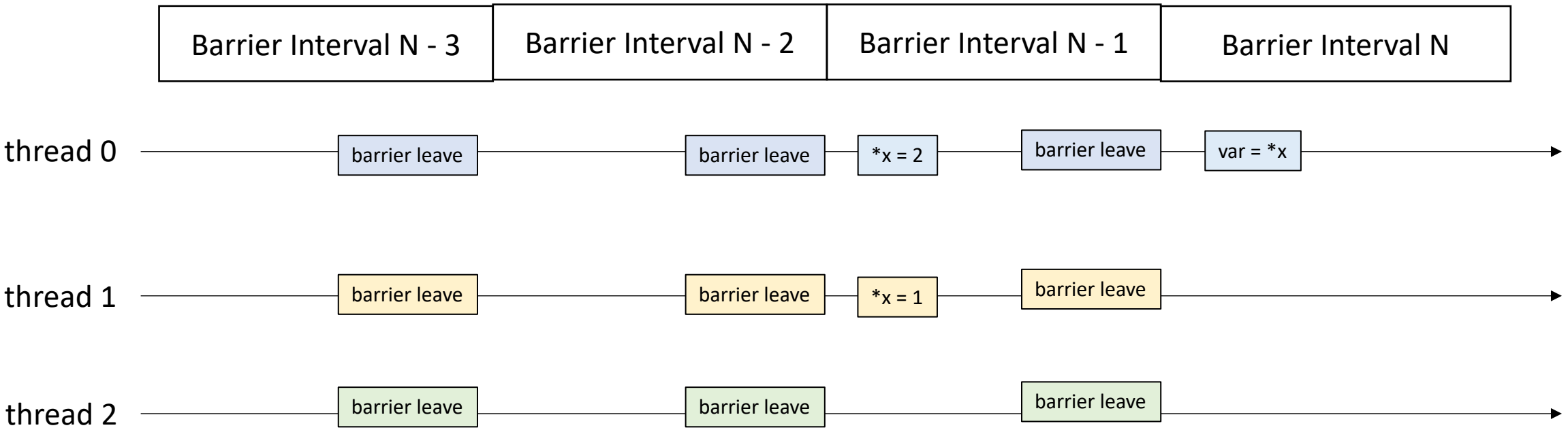


no data conflicts means that x is written to at most once per barrier interval

Assume we are reading from x

We are only allowed to return one possible value

not allowed

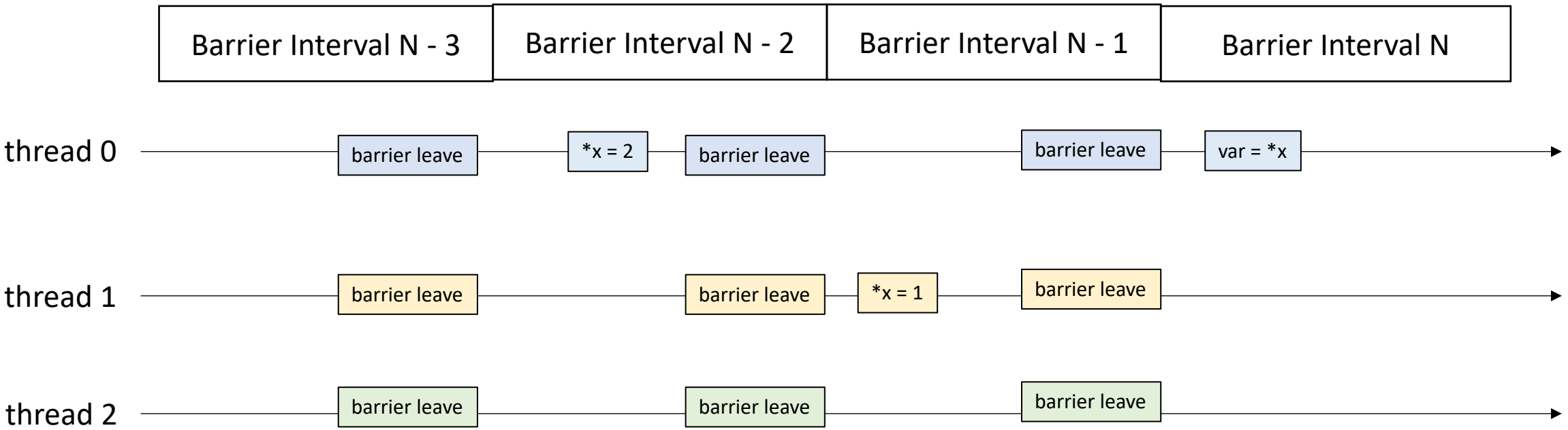


no data conflicts means that x is written to at most once per barrier interval

we will read from the write from the most recent barrier interval

Assume we are reading from x

We are only allowed to return one possible value



Schedule

- Module 4 introduction

- **Barriers**
  - Specification
  - **Implementation**

Barrier Implementation

- First attempt at implementation

```
class Barrier {
  private:
    atomic_int counter;
    int num_threads;
  public:
    Barrier(int num_threads) {
      counter = 0;
      this->num_threads = num_threads;
    }

     void barrier() {
        // ??
     }

}
```

Barrier Implementation

```
class Barrier {
  private:
    atomic_int counter;
    int num_threads;
  public:
    Barrier(int num_threads) {
      counter = 0;
      this->num_threads = num_threads;
    }

     void barrier() {
        int arrival_num = atomic_fetch_add(&counter, 1);
        // What next?
     }

}
```

Barrier Implementation

First handle the case where the thread is the last thread to arrive

```
class Barrier {
  private:
    atomic_int counter;
    int num_threads;
  public:
    Barrier(int num_threads) {
      counter = 0;
      this->num_threads = num_threads;
    }

     void barrier() {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads - 1) {
           counter.store(0);
        }
        // What next?
     }

}
```

Barrier Implementation

Spin while there is a thread waiting at the barrier

```
class Barrier {
  private:
    atomic_int counter;
    int num_threads;
  public:
    Barrier(int num_threads) {
      counter = 0;
      this->num_threads = num_threads;
    }

     void barrier() {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads - 1) {
           counter.store(0);
        }
        else {
          while (counter.load() != 0);
        }
     }

}
```

Barrier Implementation

Spin while there is a thread waiting at the barrier

Does this work?

```
class Barrier {
  private:
    atomic_int counter;
    int num_threads;
  public:
    Barrier(int num_threads) {
      counter = 0;
      this->num_threads = num_threads;
    }

     void barrier() {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads - 1) {
           counter.store(0);
        }
        else {
          while (counter.load() != 0);
        }
     }

}
```

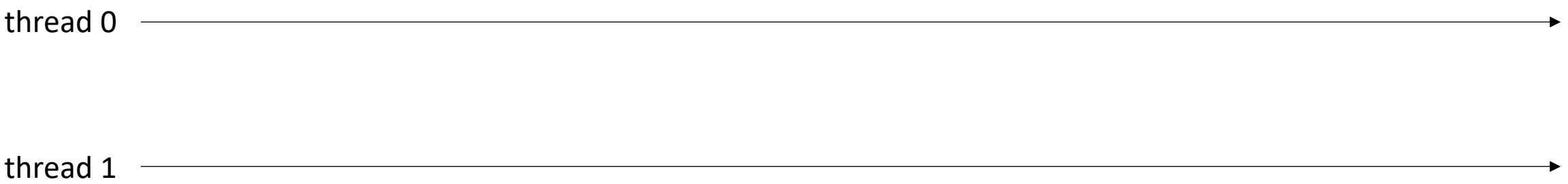
Thread 0:

```
B.barrier();
B.barrier();
```

```
void barrier() {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads) {
           counter.store(0);
        }
        else {
          while (counter.load() != 0);
        }
    }
```

Thread 1:

```
B.barrier();
B.barrier();
```



num_threads == 2

Thread 0:

```
B.barrier();
B.barrier();
```

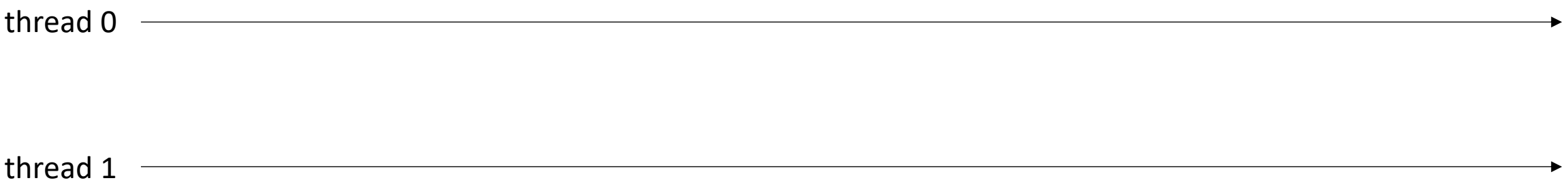
```
void barrier() {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads - 1) {
           counter.store(0);
        }
        else {
          while (counter.load() != 0);
        }
    }
```

Thread 1:

```
B.barrier();
B.barrier();
```

arrival_num = 1

arrival_num = 0



num_threads == 2
counter == 2

Thread 0:

```
B.barrier();
B.barrier();
```

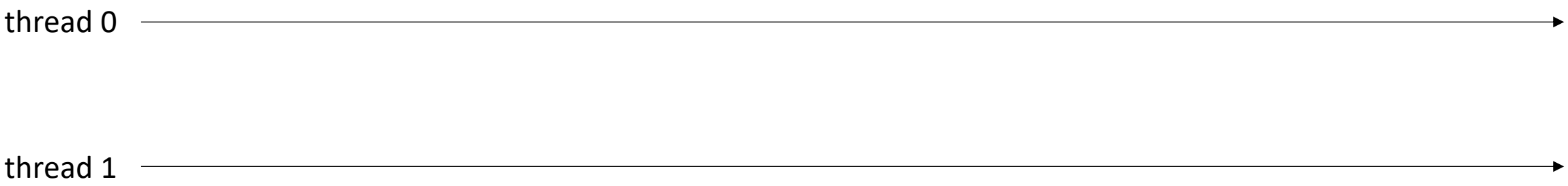
```
void barrier() {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads - 1) {
           counter.store(0);
        }
        else {
          while (counter.load() != 0);
        }
    }
```

Thread 1:

```
B.barrier();
B.barrier();
```

arrival_num = 1

arrival_num = 0



num_threads == 2
counter == 0

Thread 0:

```
B.barrier();
B.barrier();
```

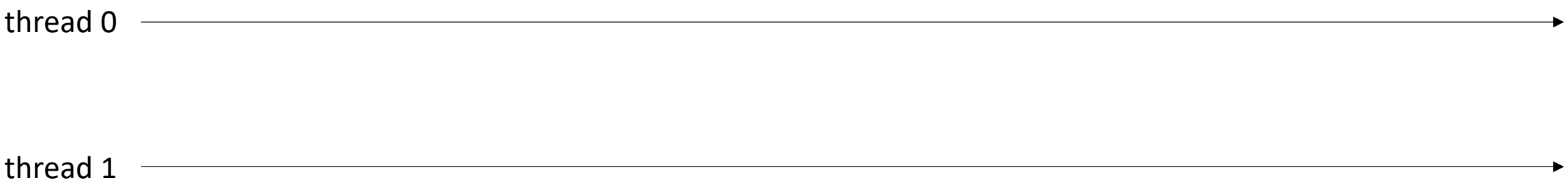
```
void barrier() {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads - 1) {
            counter.store(0);
        }
        else {
          while (counter.load() != 0);
        }
    }
```

Thread 1:

```
B.barrier();
B.barrier();
```

arrival_num = 1

arrival_num = 0



num_threads == 2
counter == 0

Thread 0:

```
B.barrier();
B.barrier();
```

```
void barrier() {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads - 1) {
           counter.store(0);
        }
        else {
          while (counter.load() != 0);
        }
    }
```

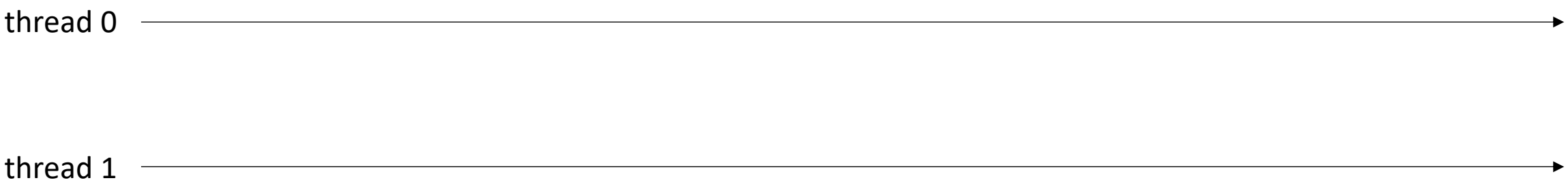
Thread 1:

```
B.barrier();
B.barrier();
```

Leaves barrier

arrival_num = 0

in a perfect world,
thread 1 executes now and leaves the barrier



```
num_threads == 2
counter == 0
```

Thread 0:

```
B.barrier();
B.barrier();
```

```
void barrier() {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads - 1) {
           counter.store(0);
        }
        else {
          while (counter.load() != 0);
        }
    }
```

Thread 1:

```
B.barrier();
B.barrier();
```

Leaves barrier

arrival_num = 0

in a perfect world,
thread 1 executes now and leaves the barrier

but what if the OS preempted thread 1? Or it was asleep?

num_threads == 2
counter == 0

Thread 0:

```
B.barrier();
B.barrier();
```

```
void barrier() {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads - 1) {
           counter.store(0);
        }
        else {
          while (counter.load() != 0);
        }
    }
```

Thread 1:

```
B.barrier();
B.barrier();
```

enters next barrier

arrival_num = 0

in a perfect world,
thread 1 executes now and leaves the barrier

but what if the OS preempted thread 1? Or it was asleep?

num_threads == 2
counter == 1

Thread 0:

```
B.barrier();
B.barrier();
```

```
void barrier() {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads - 1) {
           counter.store(0);
        }
        else {
          while (counter.load() != 0);
        }
    }
```

Thread 1:

```
B.barrier();
B.barrier();
```

arrival_num == 0

arrival_num = 0

in a perfect world,
thread 1 executes now and leaves the barrier

but what if the OS preempted thread 1? Or it was asleep?

num_threads == 2
counter == 1

Thread 0:

```
B.barrier();
B.barrier();
```

```
void barrier() {
       int arrival_num = atomic_fetch_add(&counter, 1);
       if (arrival_num == num_threads - 1) {
          counter.store(0);
       }
       else {
         while (counter.load() != 0);
       }
    }
```

Thread 1:

```
B.barrier();
B.barrier();
```

Thread 1 wakes up! Doesn't think its missed anything

arrival_num == 0

arrival_num = 0

in a perfect world,
thread 1 executes now and leaves the barrier

num_threads == 2
counter == 1

Thread 0:

```
B.barrier();
B.barrier();
```

```
void barrier() {
       int arrival_num = atomic_fetch_add(&counter, 1);
       if (arrival_num == num_threads - 1) {
          counter.store(0);
       }
       else {
         while (counter.load() != 0);
       }
    }
```

Thread 1:

```
B.barrier();
B.barrier();
```

Thread 1 wakes up! Doesn't think its missed anything

arrival_num == 0

arrival_num = 0

in a perfect world,
thread 1 executes now and leaves the barrier

Both threads get stuck here!

Thread 0:

```
B.barrier();
B.barrier();
```

```
void barrier() {
       int arrival_num = atomic_fetch_add(&counter, 1);
       if (arrival_num == num_threads - 1) {
          counter.store(0);
       }
       else {
         while (counter.load() != 0);
       }
    }
```

Thread 1:

```
B.barrier();
B.barrier();
```

Ideas for fixing?

Thread 0:

```
B.barrier();
B.barrier();
```

```
void barrier() {
       int arrival_num = atomic_fetch_add(&counter, 1);
       if (arrival_num == num_threads - 1) {
          counter.store(0);
       }
       else {
         while (counter.load() != 0);
       }
    }
```

Thread 1:

```
B.barrier();
B.barrier();
```

Ideas for fixing?

Two different barriers that alternate?

Thread 0:

```
B0.barrier();
B1.barrier();
```

```
void barrier() {
       int arrival_num = atomic_fetch_add(&counter, 1);
       if (arrival_num == num_threads - 1) {
          counter.store(0);
       }
       else {
         while (counter.load() != 0);
       }
    }
```

Thread 1:

```
B0.barrier();
B1.barrier();
```

Ideas for fixing?

Two different barriers that alternate?

Pros: simple to implement

Cons: user has to alternate barriers

Thread 0:

```
B0.barrier();
B1.barrier();
```

```
void barrier() {
       int arrival_num = atomic_fetch_add(&counter, 1);
       if (arrival_num == num_threads - 1) {
          counter.store(0);
       }
       else {
         while (counter.load() != 0);
       }
    }
```

Thread 1:

```
B0.barrier();
B1.barrier();
```

Ideas for fixing?

Two different barriers that alternate?

Pros: simple to implement

Cons: user has to alternate barriers

```
B.barrier();
if (...) {
  B.barrier();
}
B.barrier();
```

How to alternate these calls?

Sense Reversing Barrier

- Book Chapter 17
- Alternating ”sense” dynamically

Thread 0:

```
B.barrier();
B.barrier();
```

sync on sense = false

Thread 1:

```
B.barrier();
B.barrier();
```

Sense Reversing Barrier

- Book Chapter 17
- Alternating ”sense” dynamically

Thread 0:

```
B.barrier();
B.barrier();
```

sync on sense = true

Thread 1:

```
B.barrier();
B.barrier();
```

```
class SenseBarrier {
  private:
    atomic_int counter;
    int num_threads;
    atomic_bool sense;
    bool thread_sense[num_threads];
  public:
    Barrier(int num_threads) {
      counter = 0;
      this->num_threads = num_threads;
      sense = false;
      thread_sense = {true, ...};
    }

     void barrier(int tid) {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads) {
           counter.store(0);
           sense = thread_sense[tid];
        }
        else {
          while (sense != thread_sense[tid]);
        }
        thread_sense[tid] = !thread_sense[tid];
     }
}
```

thread_sense = true

```
num_threads == 2
counter == 0
sense = false
```

thread_sense = true

Thread 0:

```
B.barrier();
B.barrier();
```

```
void barrier(int tid) {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads-1) {
           counter.store(0);
           sense = thread_sense[tid];
        }
        else {
          while (sense != thread_sense[tid]);
        }
        thread_sense[tid] = !thread_sense[tid];
     }
```

Thread 1:

```
B.barrier();
B.barrier();
```

num_threads == 2
counter == 2
sense = false

thread_sense = true
arrival_num = 1

Thread 0:

```
B.barrier();
B.barrier();
```

```
void barrier(int tid) {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads-1) {
           counter.store(0);
           sense = thread_sense[tid];
        }
        else {
          while (sense != thread_sense[tid]);
        }
        thread_sense[tid] = !thread_sense[tid];
    }
```

thread_sense = true
arrival_num = 0

Thread 1:

```
B.barrier();
B.barrier();
```

num_threads == 2
counter == 2
sense = false

thread_sense = true
arrival_num = 1

Thread 0:

```
B.barrier();
B.barrier();
```

```
void barrier(int tid) {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads-1) {
            counter.store(0);
            sense = thread_sense[tid];
        }
        else {
          while (sense != thread_sense[tid]);
        }
        thread_sense[tid] = !thread_sense[tid];
    }
```

thread_sense = true
arrival_num = 0

Thread 1:

```
B.barrier();
B.barrier();
```

num_threads == 2
counter == 0
sense = true

thread_sense = false
arrival_num = 1

Thread 0:

```
B.barrier();
B.barrier();
```

```
void barrier(int tid) {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads-1) {
           counter.store(0);
           sense = thread_sense[tid];
        }
        else {
          while (sense != thread_sense[tid]);
        }
        thread_sense[tid] = !thread_sense[tid];
    }
```

thread_sense = true
arrival_num = 0

Thread 1:

```
B.barrier();
B.barrier();
```

num_threads == 2
counter == 0
sense = true

thread_sense = false
arrival_num = ?

Thread 0:

```
B.barrier();
B.barrier();
```

```
void barrier(int tid) {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads-1) {
           counter.store(0);
           sense = thread_sense[tid];
        }
        else {
          while (sense != thread_sense[tid]);
        }
        thread_sense[tid] = !thread_sense[tid];
    }
```

thread_sense = true
arrival_num = 0

Thread 1:

```
B.barrier();
B.barrier();
```

Remember the issue! Thread 1 went to sleep around this time and thread 0 went into the barrier again!

num_threads == 2
counter == 1
sense = true

thread_sense = false
arrival_num = 0

Thread 0:

```
B.barrier();
B.barrier();
```

```
void barrier(int tid) {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads-1) {
           counter.store(0);
           sense = thread_sense[tid];
        }
        else {
          while (sense != thread_sense[tid]);
        }
        thread_sense[tid] = !thread_sense[tid];
    }
```

thread_sense = true
arrival_num = 0

Thread 1:

```
B.barrier();
B.barrier();
```

num_threads == 2
counter == 1
sense = true

thread_sense = false
arrival_num = 0

Thread 0:

```
B.barrier();
B.barrier();
```

```
void barrier(int tid) {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads-1) {
           counter.store(0);
           sense = thread_sense[tid];
        }
        else {
          while (sense != thread_sense[tid]);
        }
        thread_sense[tid] = !thread_sense[tid];
    }
```

thread_sense = true
arrival_num = 0

Thread 1:

```
B.barrier();
B.barrier();
```

both are waiting!,
but thread 1 can leave

num_threads == 2
counter == 1
sense = true

thread_sense = false
arrival_num = 0

Thread 0:

```
B.barrier();
B.barrier();
```

```
void barrier(int tid) {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads-1) {
           counter.store(0);
           sense = thread_sense[tid];
        }
        else {
          while (sense != thread_sense[tid]);
        }
        thread_sense[tid] = !thread_sense[tid];
    }
```

thread_sense = false
arrival_num = 0

Thread 1:

```
B.barrier();
B.barrier();
```

both are waiting!,
but thread 1 can leave

num_threads == 2
counter == 1
sense = true

thread_sense = false
arrival_num = 0

Thread 0:

```
B.barrier();
B.barrier();
```

```
void barrier(int tid) {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads-1) {
           counter.store(0);
           sense = thread_sense[tid];
        }
        else {
          while (sense != thread_sense[tid]);
        }
        thread_sense[tid] = !thread_sense[tid];
      }
```

thread_sense = false
arrival_num = ?

Thread 1:

```
B.barrier();
B.barrier();
```

Thread 1 finishes the barrier

num_threads == 2
counter == 1
sense = true

thread_sense = false
arrival_num = 0

Thread 0:

```
B.barrier();
B.barrier();
```

```
void barrier(int tid) {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads-1) {
           counter.store(0);
           sense = thread_sense[tid];
        }
        else {
          while (sense != thread_sense[tid]);
        }
        thread_sense[tid] = !thread_sense[tid];
    }
```

thread_sense = false
arrival_num = ?

Thread 1:

```
B.barrier();
B.barrier();
```

Goes into the second barrier

num_threads == 2
counter == 2
sense = true

thread_sense = false
arrival_num = 0

Thread 0:

```
B.barrier();
B.barrier();
```

```
void barrier(int tid) {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads-1) {
           counter.store(0);
           sense = thread_sense[tid];
        }
        else {
          while (sense != thread_sense[tid]);
        }
        thread_sense[tid] = !thread_sense[tid];
    }
```

thread_sense = false
arrival_num = 1

Thread 1:

```
B.barrier();
B.barrier();
```

Goes into the second barrier

num_threads == 2
counter == 2
sense = true

thread_sense = false
arrival_num = 0

Thread 0:

```
B.barrier();
B.barrier();
```

```
void barrier(int tid) {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads-1) {
           counter.store(0);
           sense = thread_sense[tid];
        }
        else {
          while (sense != thread_sense[tid]);
        }
        thread_sense[tid] = !thread_sense[tid];
    }
```

thread_sense = false
arrival_num = 1

Thread 1:

```
B.barrier();
B.barrier();
```

Goes into the second barrier

```
num_threads == 2
counter == 0
sense = false
```

thread_sense = false

arrival_num = 0

Thread 0:

```
B.barrier();
B.barrier();
```

```
void barrier(int tid) {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads-1) {
           counter.store(0);
           sense = thread_sense[tid];
        }
        else {
          while (sense != thread_sense[tid]);
        }
        thread_sense[tid] = !thread_sense[tid];
    }
```

thread_sense = false

arrival_num = 1

Thread 1:

```
B.barrier();
B.barrier();
```

Goes into the second barrier

num_threads == 2
counter == 0
sense = false

thread_sense = false
arrival_num = 0

Thread 0:

```
B.barrier();
B.barrier();
```

```
void barrier(int tid) {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads-1) {
           counter.store(0);
           sense = thread_sense[tid];
        }
        else {
          while (sense != thread_sense[tid]);
        }
        thread_sense[tid] = !thread_sense[tid];
    }
```

thread_sense = false
arrival_num = 1

Thread 1:

```
B.barrier();
B.barrier();
```

thread 0 can leave, thread 1 can leave and the barrier works as expected!

See you on Wednesday!

- Starting on module 4

- Work on HW 3