Concurrent Queues and Stacks

Christof Fetzer, TU Dresden

Based on slides by Maurice Herlihy and Nir Shavit

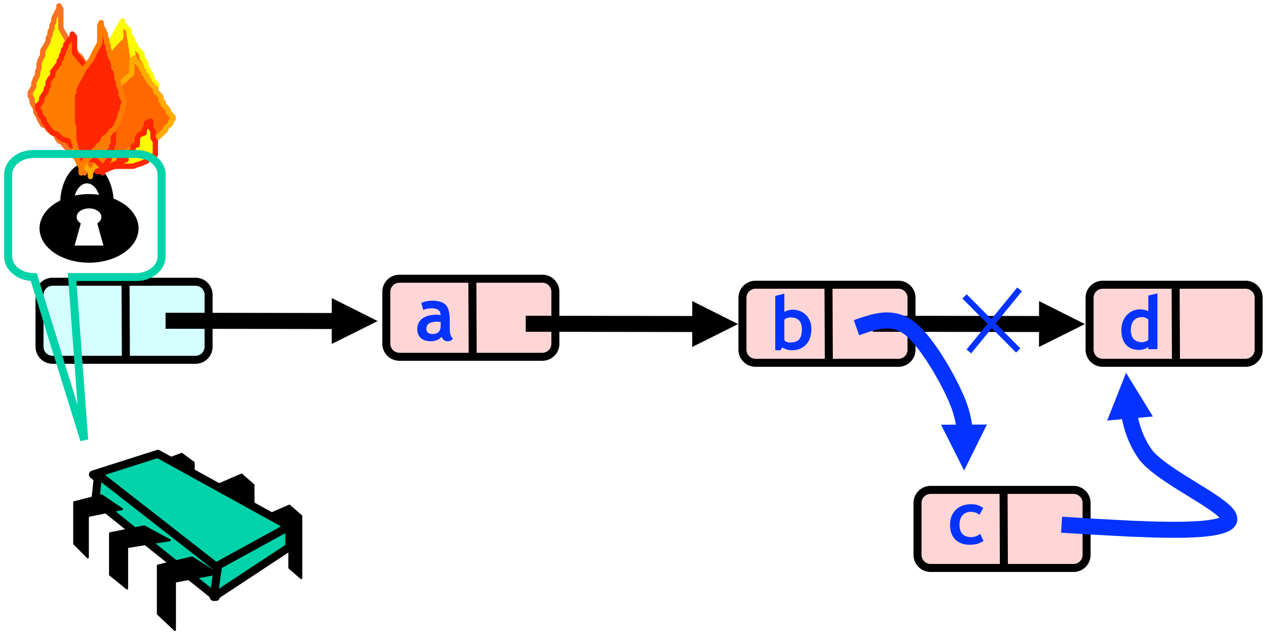
Linked List Lecture

- Five approaches to concurrent data structure design:
  - Coarse-grained locking
  - Fine-grained locking
  - Optimistic synchronization
  - Lazy synchronization
  - Lock-free synchronization

List-based Set

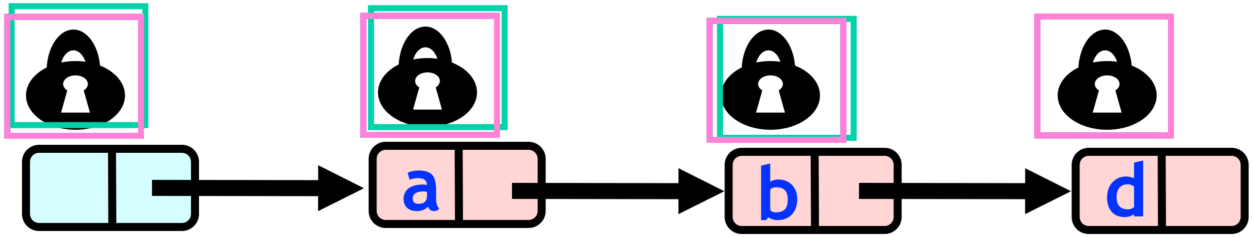
- We used an ordered list to implement a Set:
  - An unordered collection of objects
  - No duplicates
  - Methods:
    - add() a new object
    - remove() an object
    - Test if set contains() object

Course Grained Locking



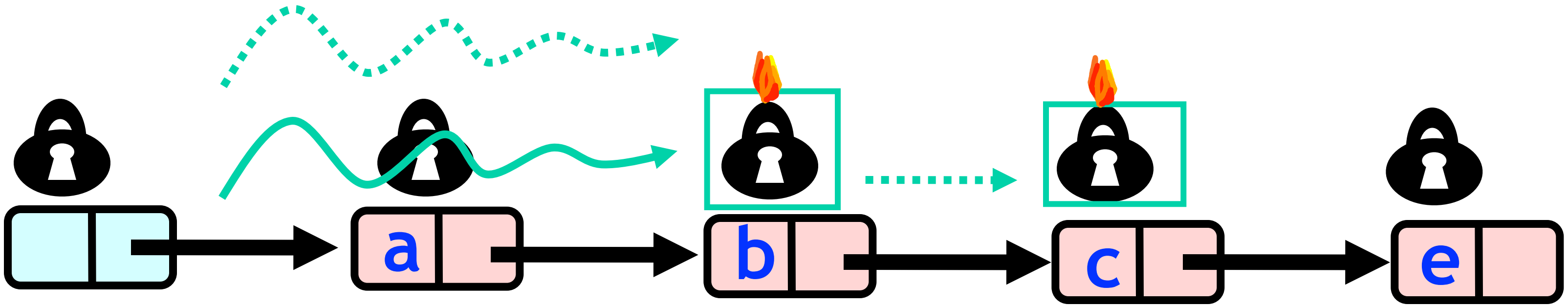
Simple but hotspot + bottleneck

Fine Grained Locking



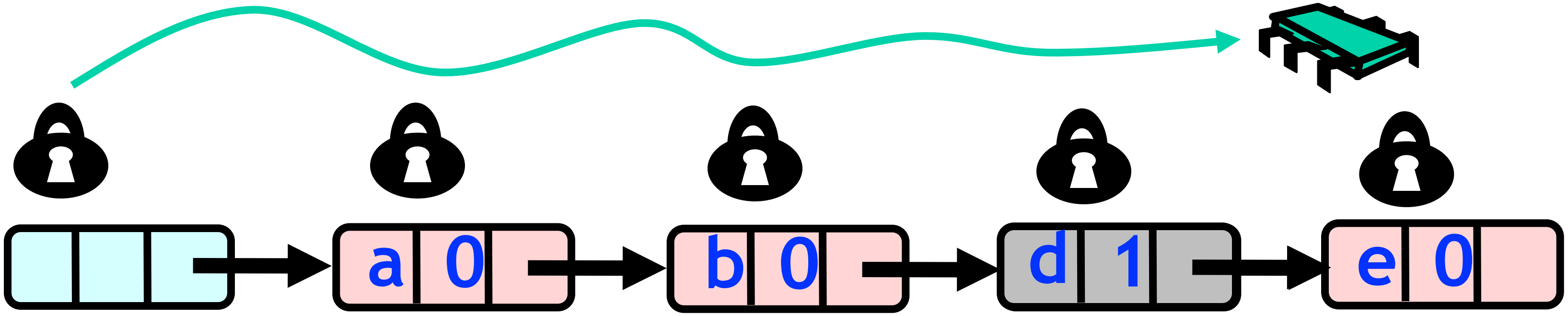
- Allows concurrency but everyone always delayed by front guy = bottleneck
- Lock acquisition overhead

Optimistic List



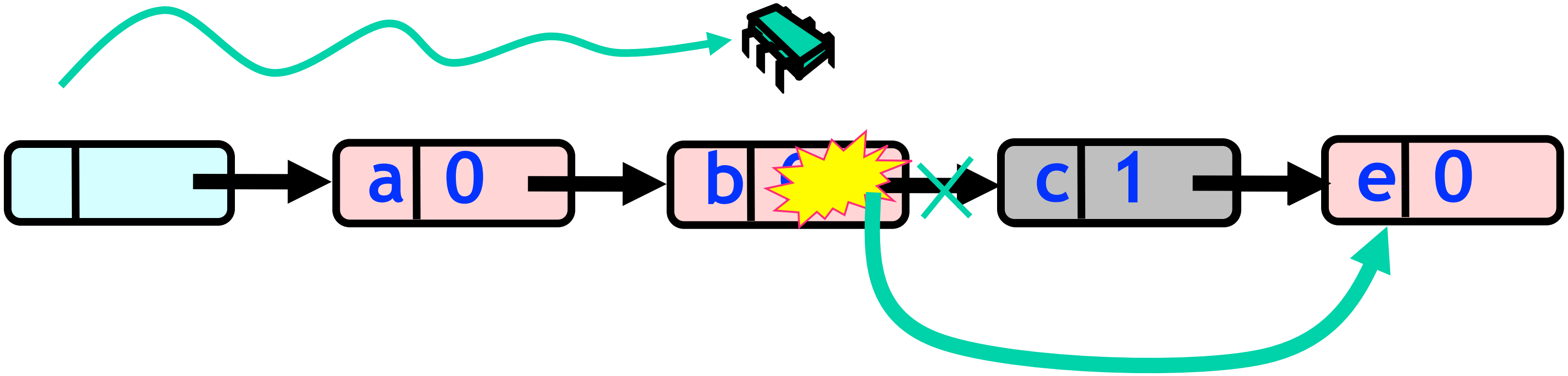
1. Limited Hotspots (Only at locked `Add()`, `Remove()`, `Find()` destination locations, not traversals)
2. But two traversals
3. Yet traversals are wait-free!

Lazy List



Lazy Add() and Remove() + Wait-free Contains()

Lock-free List



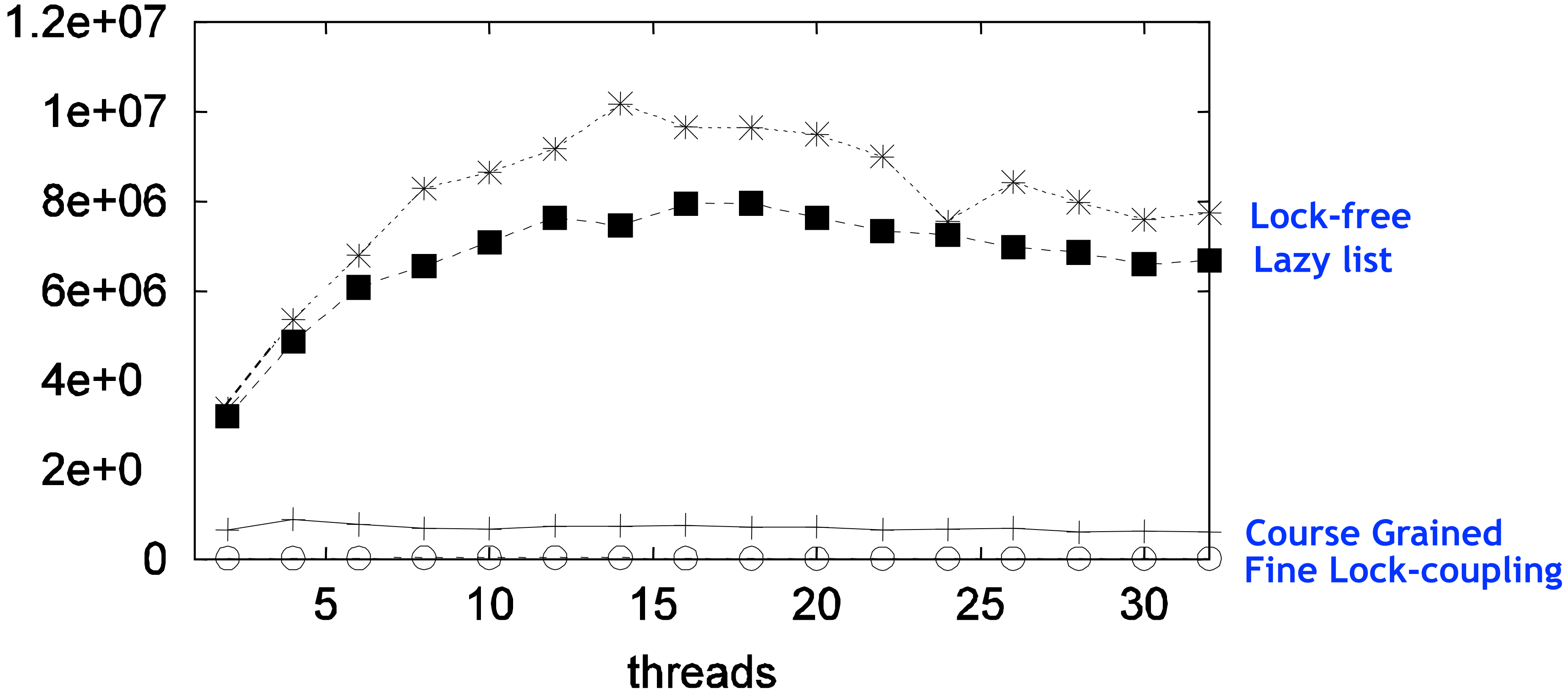
1. Add() and Remove() physically remove marked nodes
2. Wait-free contains() traverses both marked and removed nodes

Performance

On 16 node shared memory machine
Benchmark throughput of Java List-based Set algs. Vary % of Contains() method Calls.

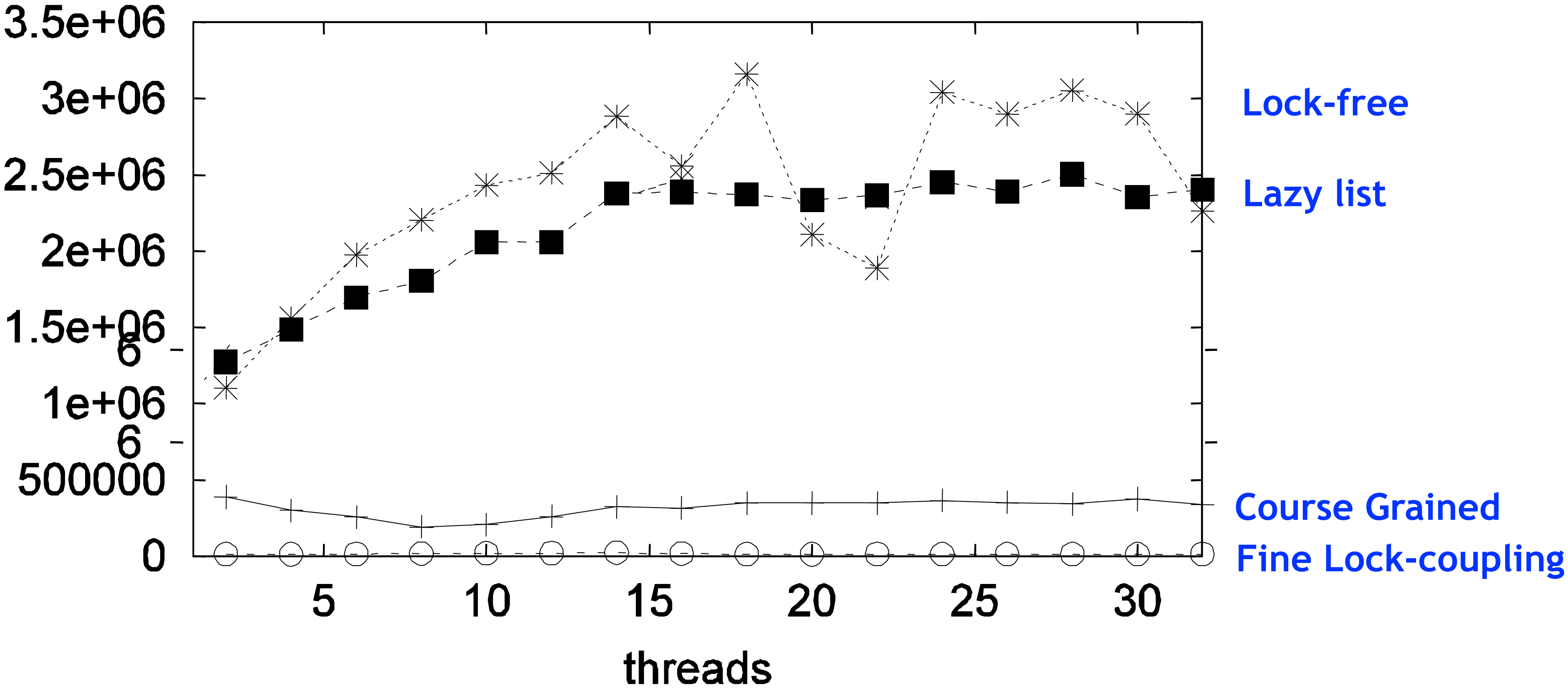
High Contains Ratio

Ops/sec (90% reads/ 10% updates)



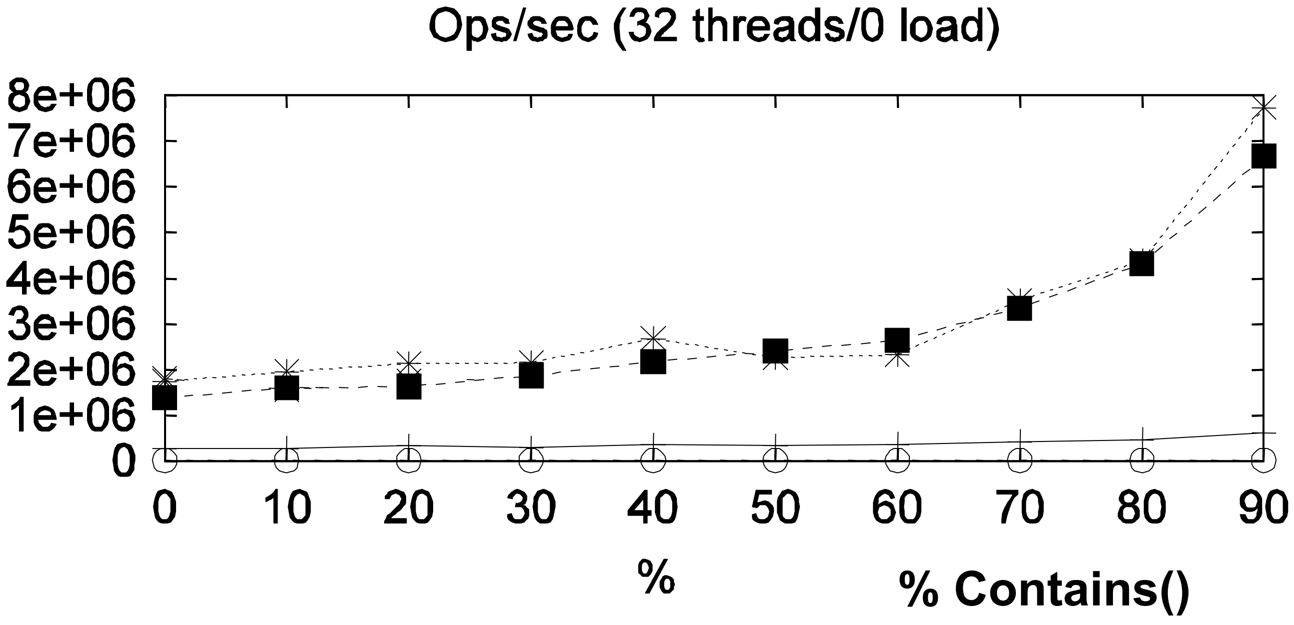
Low Contains Ratio

Ops/sec (50% reads/ 50% updates)



© Herlihy-Shavit

As Contains Ratio Increases



Lock-free

Lazy list

Course Grained

Fine Lock-coupling

Today: Another Fundamental Problem

- We told you about
  - Sets implemented using linked lists
- Next: **queues**
  - Ubiquitous data structure
  - Often used to buffer requests …

Shared Pools

- Queue belongs to broader pool class
- Pool: similar to Set but
  - Allows duplicates (it's a Multiset)
  - No membership test (no `contains()`)

Pool Flavors

- *Bounded*
  - Fixed capacity
  - Good when resources an issue
- *Unbounded*
  - Holds any number of objects

Pool Flavors

- Problem cases:
  - Removing from empty pool
  - Adding to full (bounded) pool
- Blocking
  - Caller waits until state changes
- Non-Blocking
  - Method throws exception

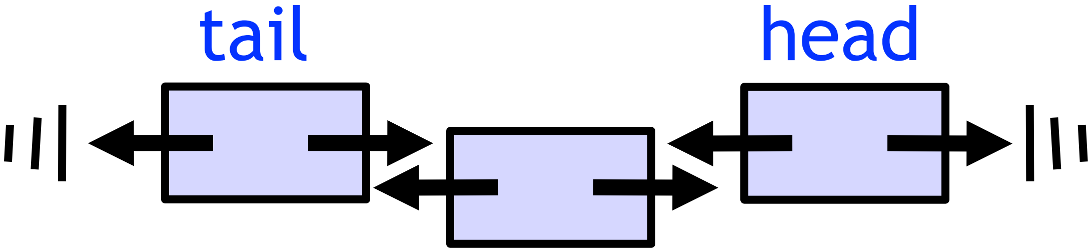
Queues & Stacks

- `Add()` and `Remove()`:
  - Queue enqueue (`Enq()`) and dequeue (`Deq()`)
  - Stack push (`push()`) and pop (`pop()`)
- A Queue is a pool with FIFO order on enqueues and dequeues
- A Stack is a pool with LIFO order on pushes and pops

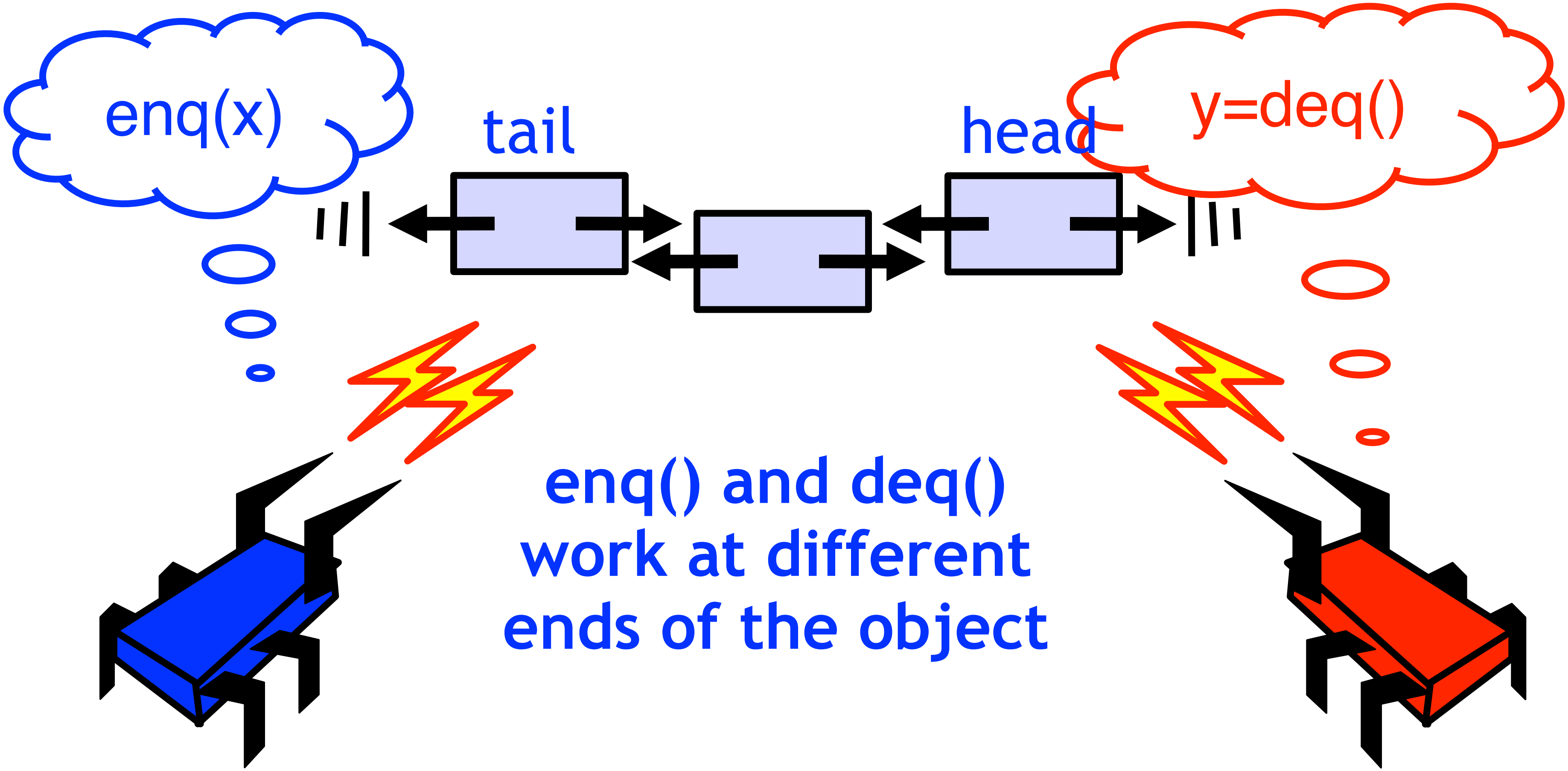
This and next Lectures…

- Bounded, Blocking, Lock-based Queue
- Unbounded, Non-Blocking, Lock-free Queue
- Examine effects of ABA problem
- Unbounded Non-Blocking Lock-free Stack
- Elimination-Backoff Stack

Queue: Concurrency

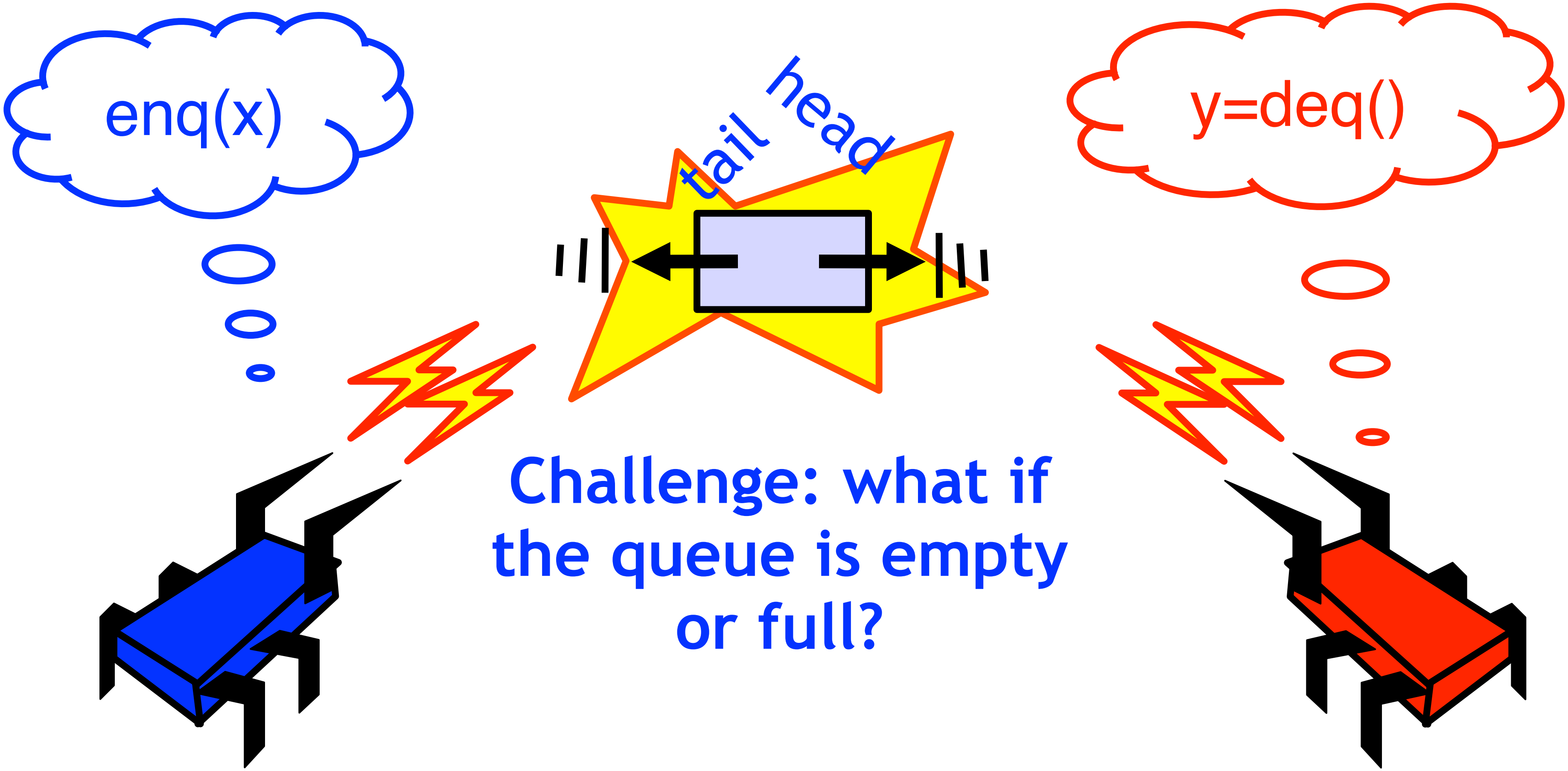


Queue: Concurrency



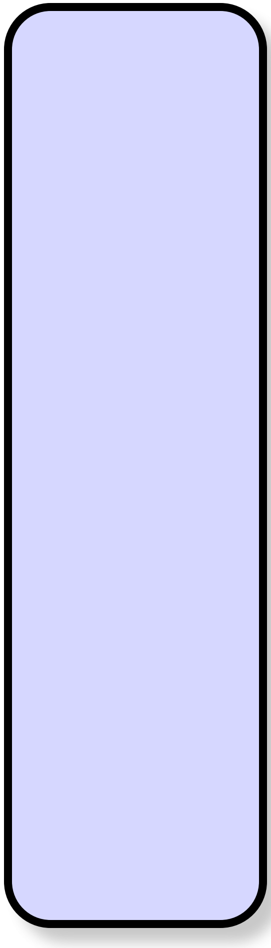
enq() and deq() work at different ends of the object

Concurrency

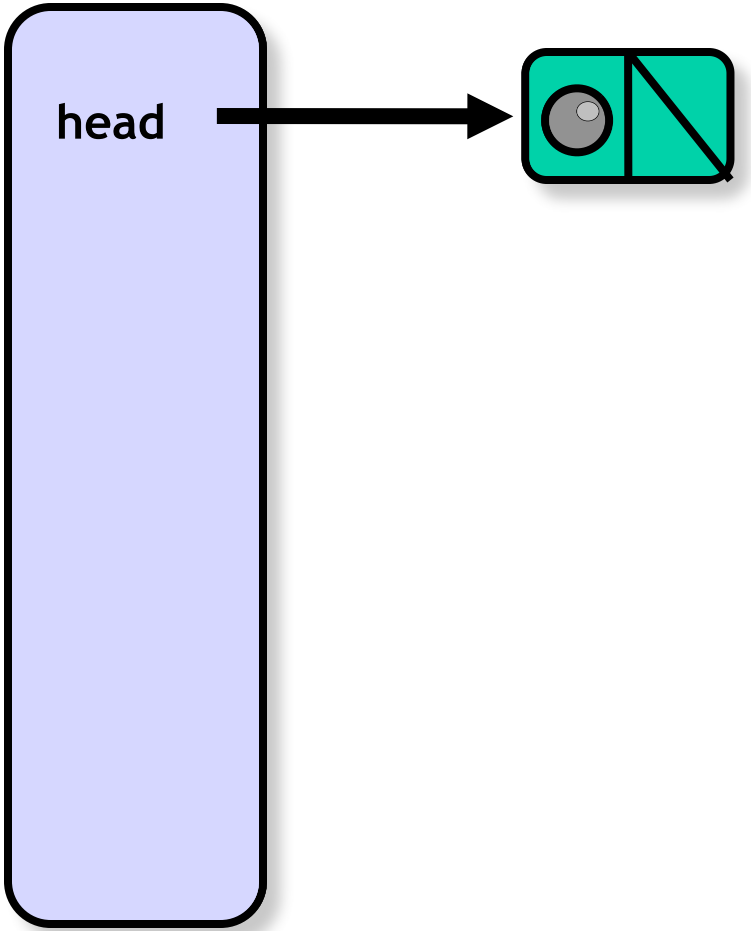


Challenge: what if the queue is empty or full?

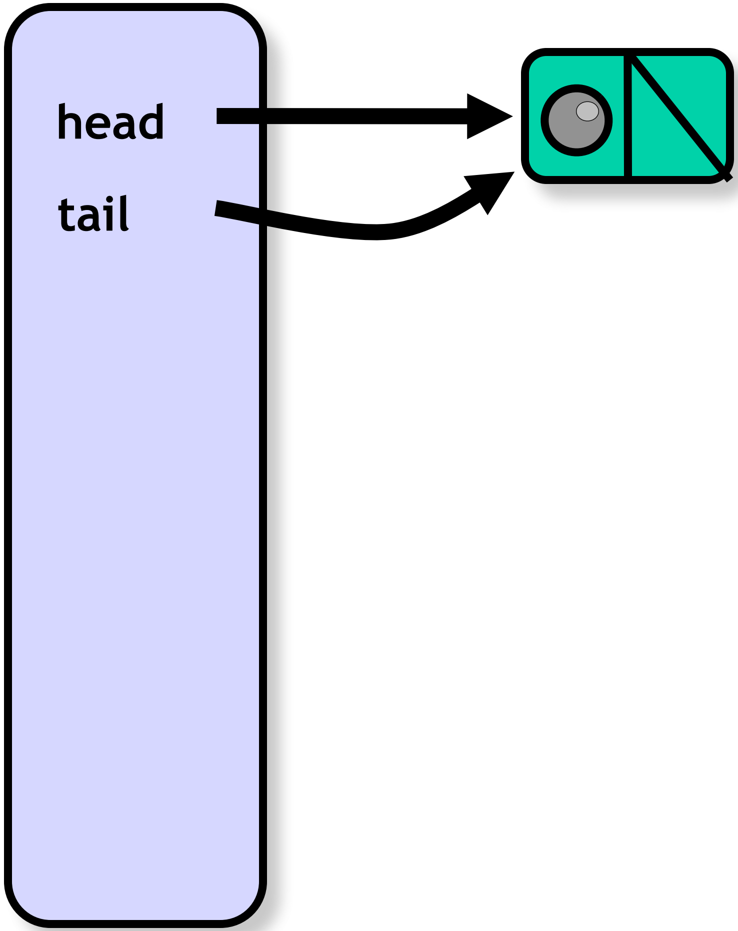
Ingredients: Bounded Queue



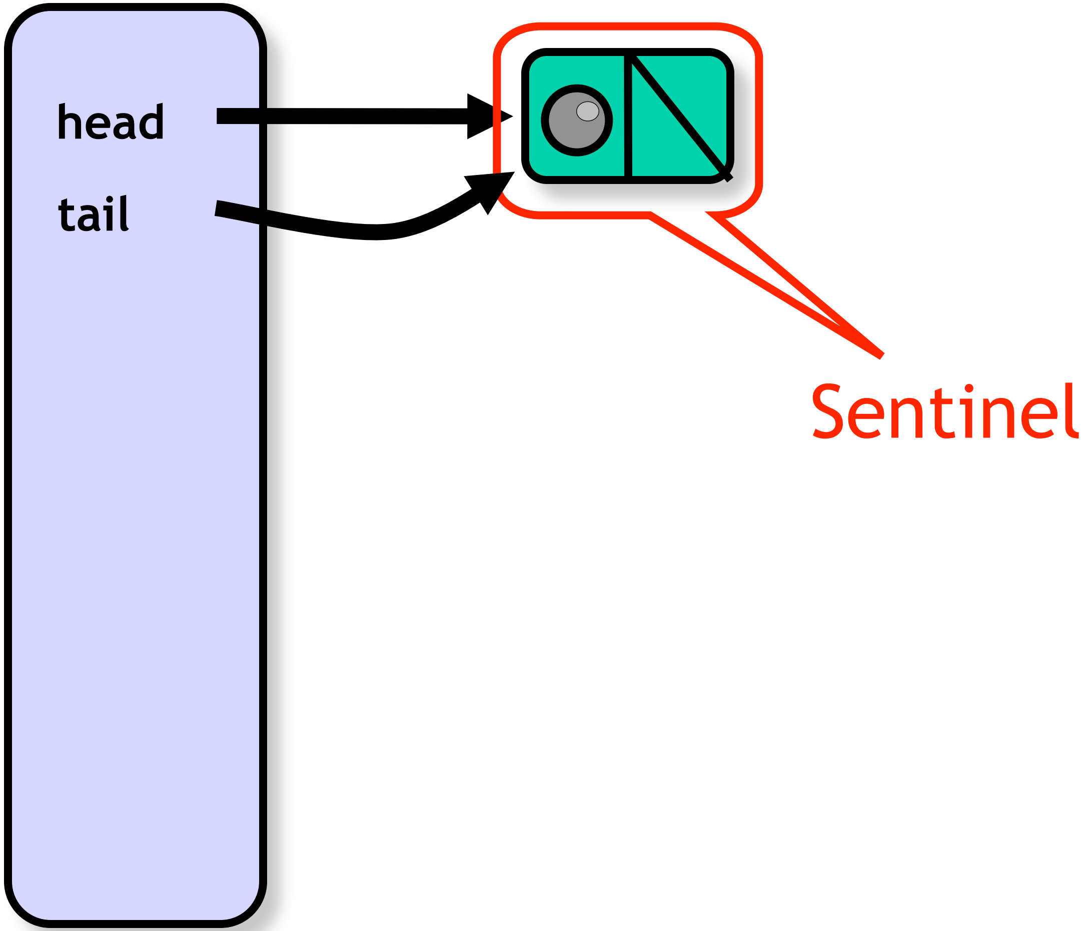
Ingredients: Bounded Queue



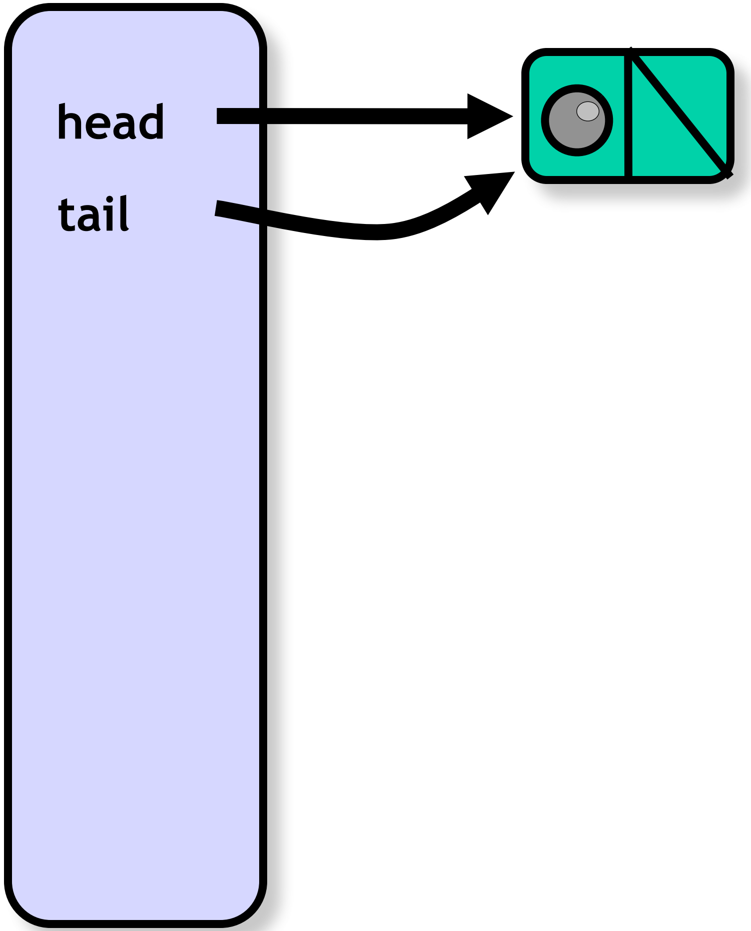
Ingredients: Bounded Queue



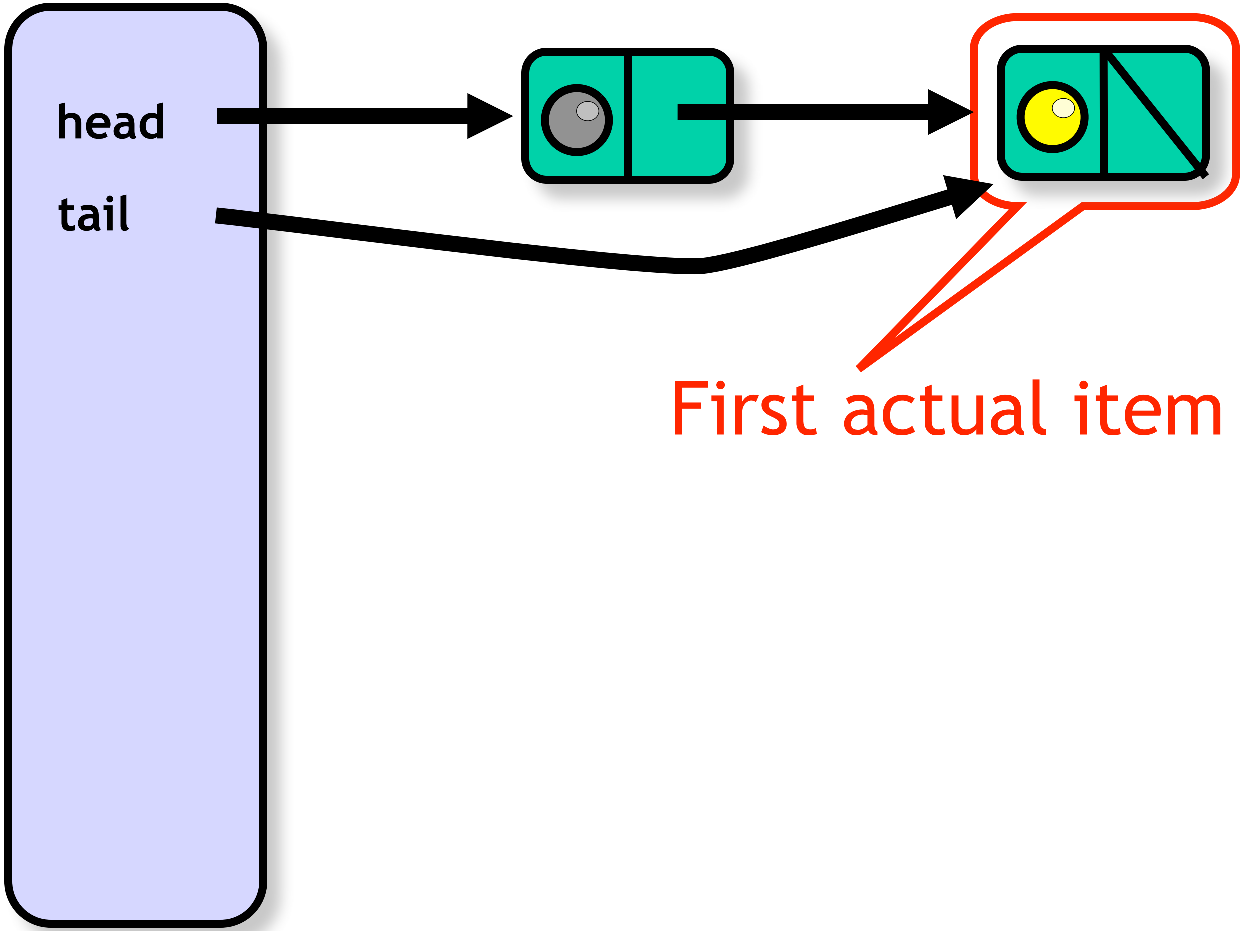
Ingredients: Bounded Queue



Ingredients: Bounded Queue

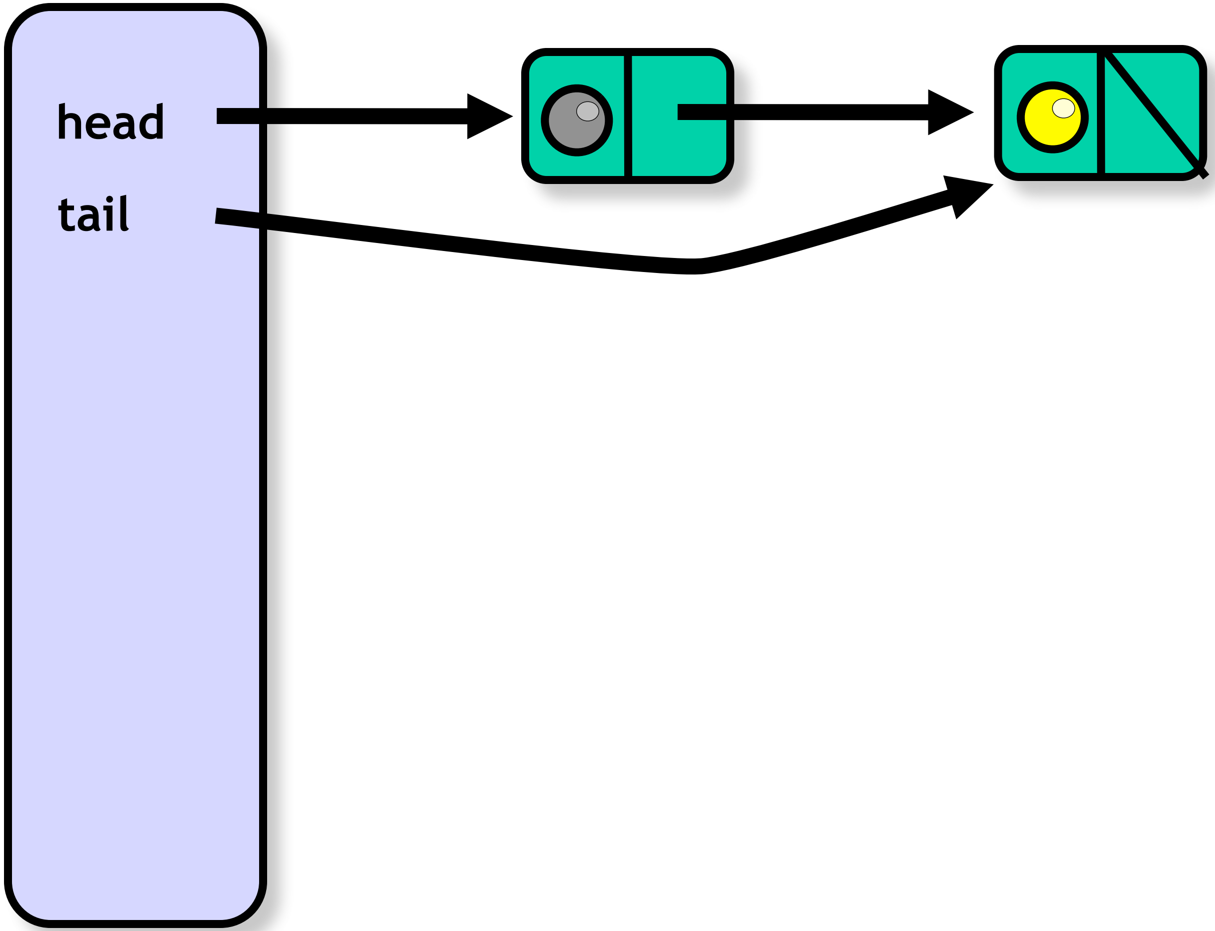


Ingredients: Bounded Queue

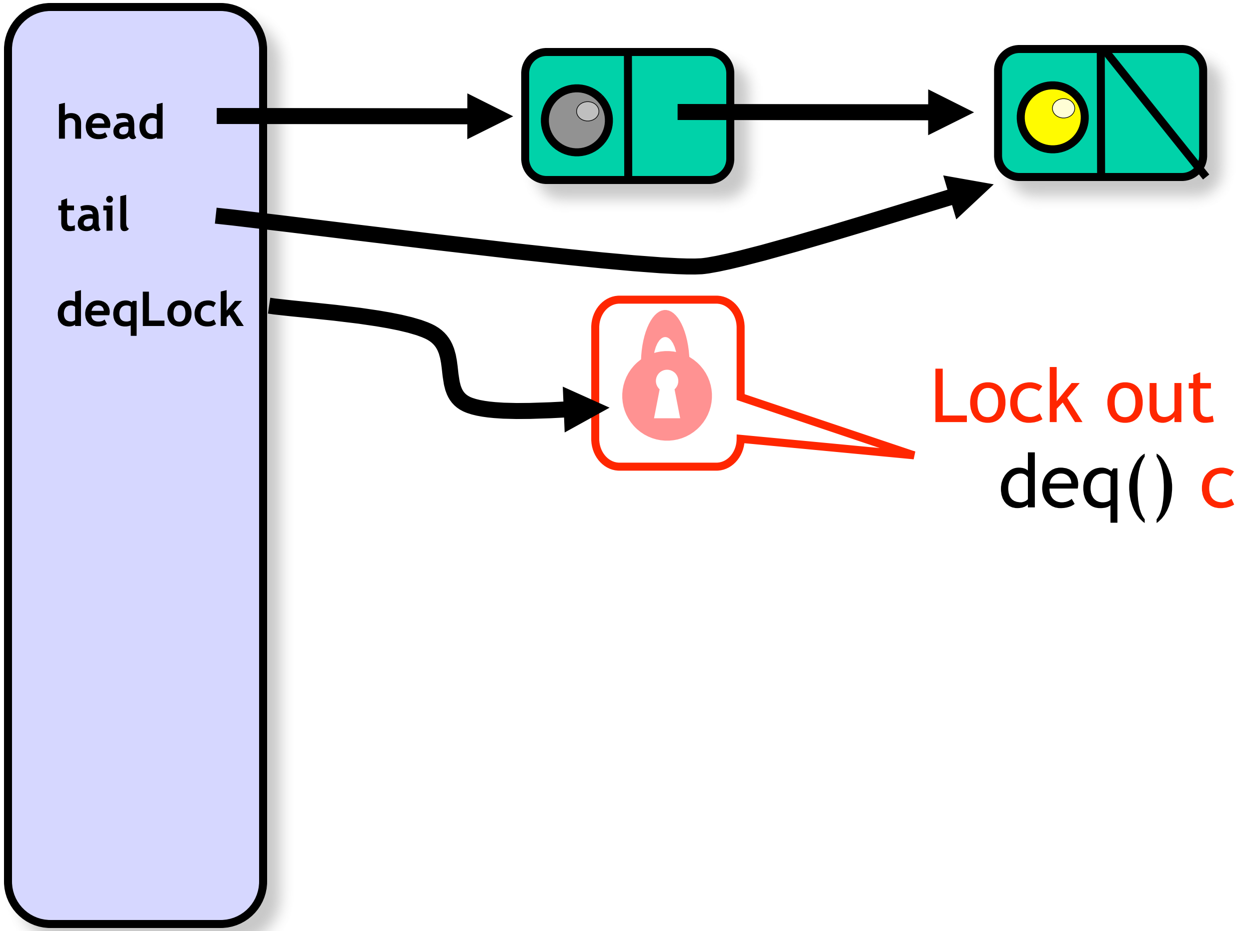


First actual item

Ingredients: Bounded Queue

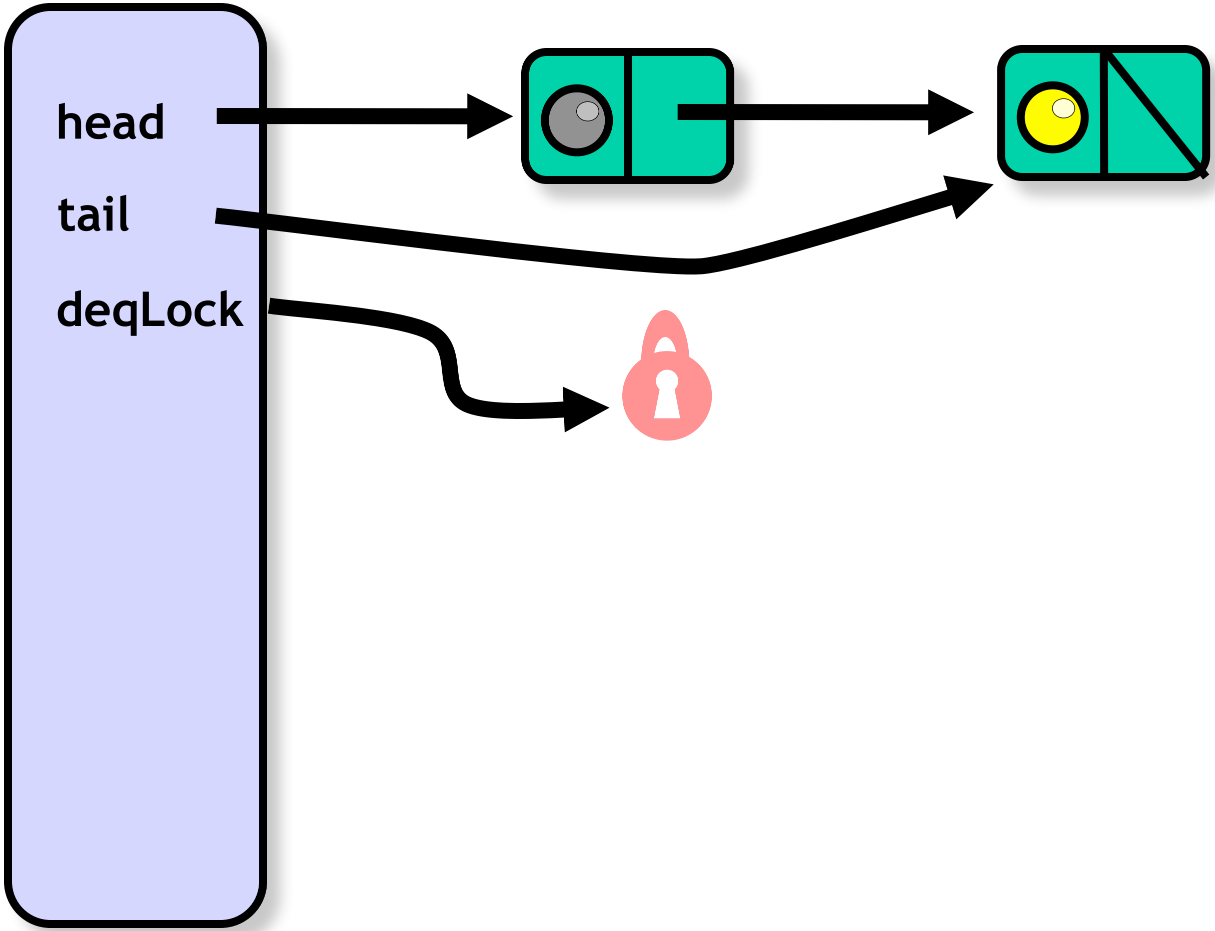


Ingredients: Bounded Queue

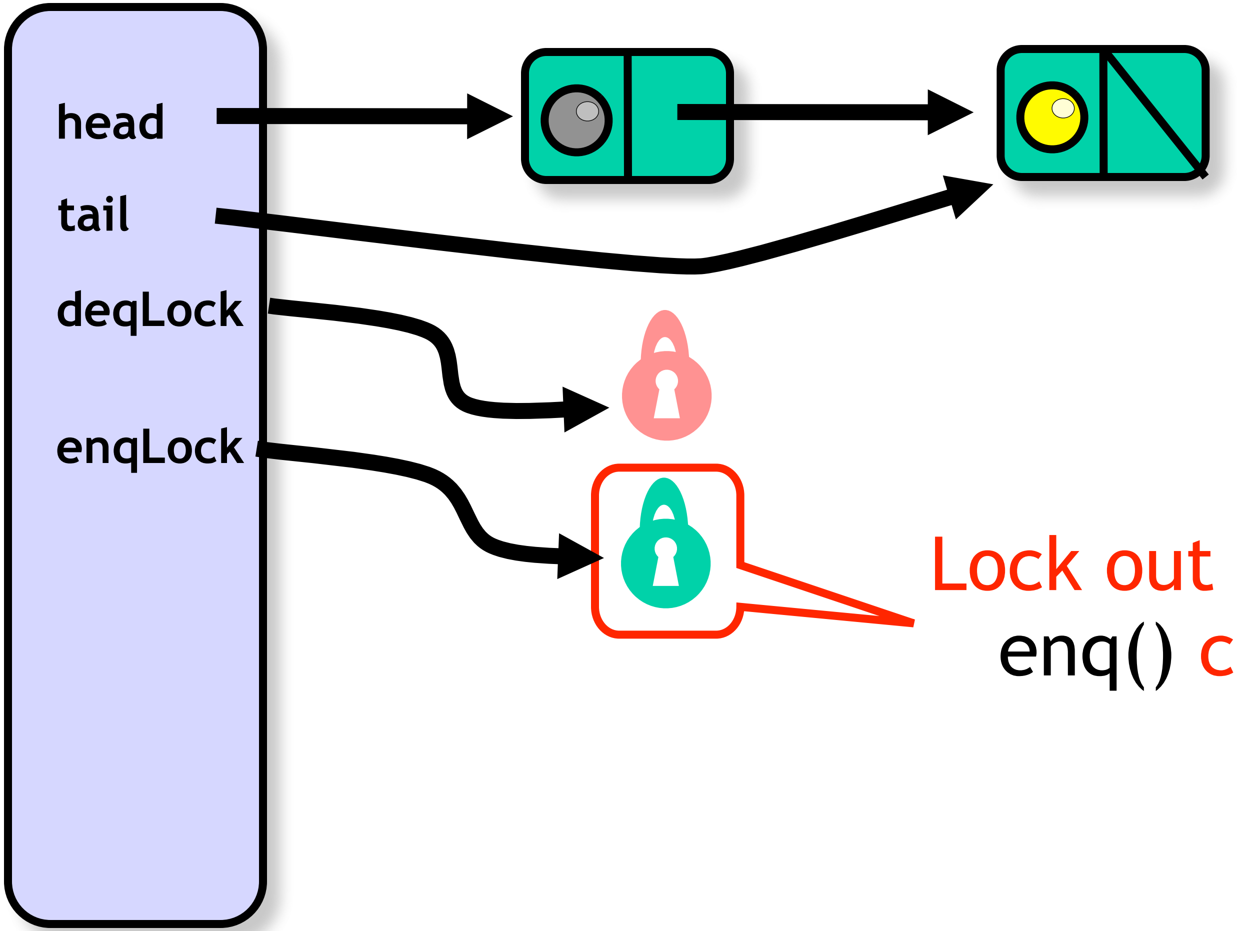


Lock out other deq() calls

Ingredients: Bounded Queue

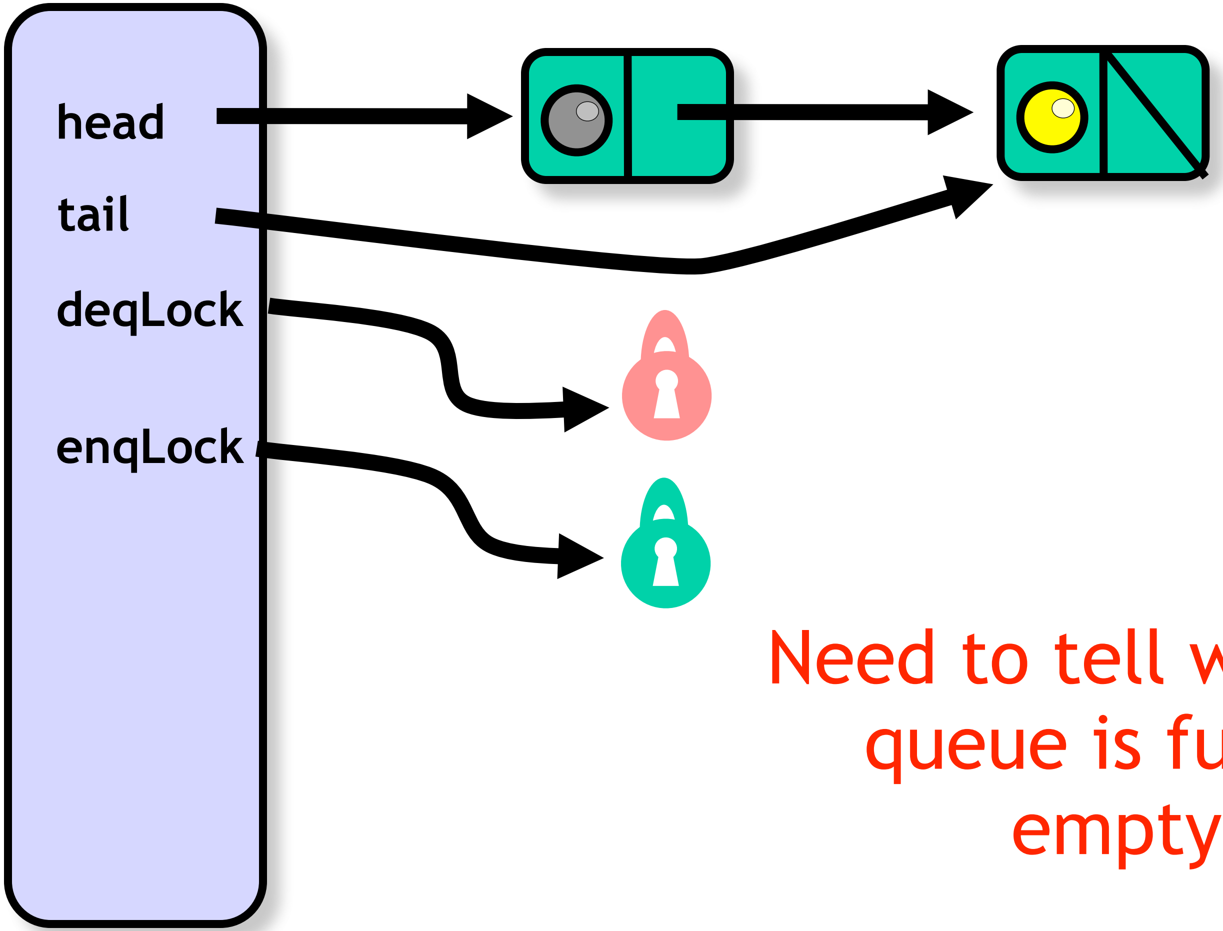


Ingredients: Bounded Queue



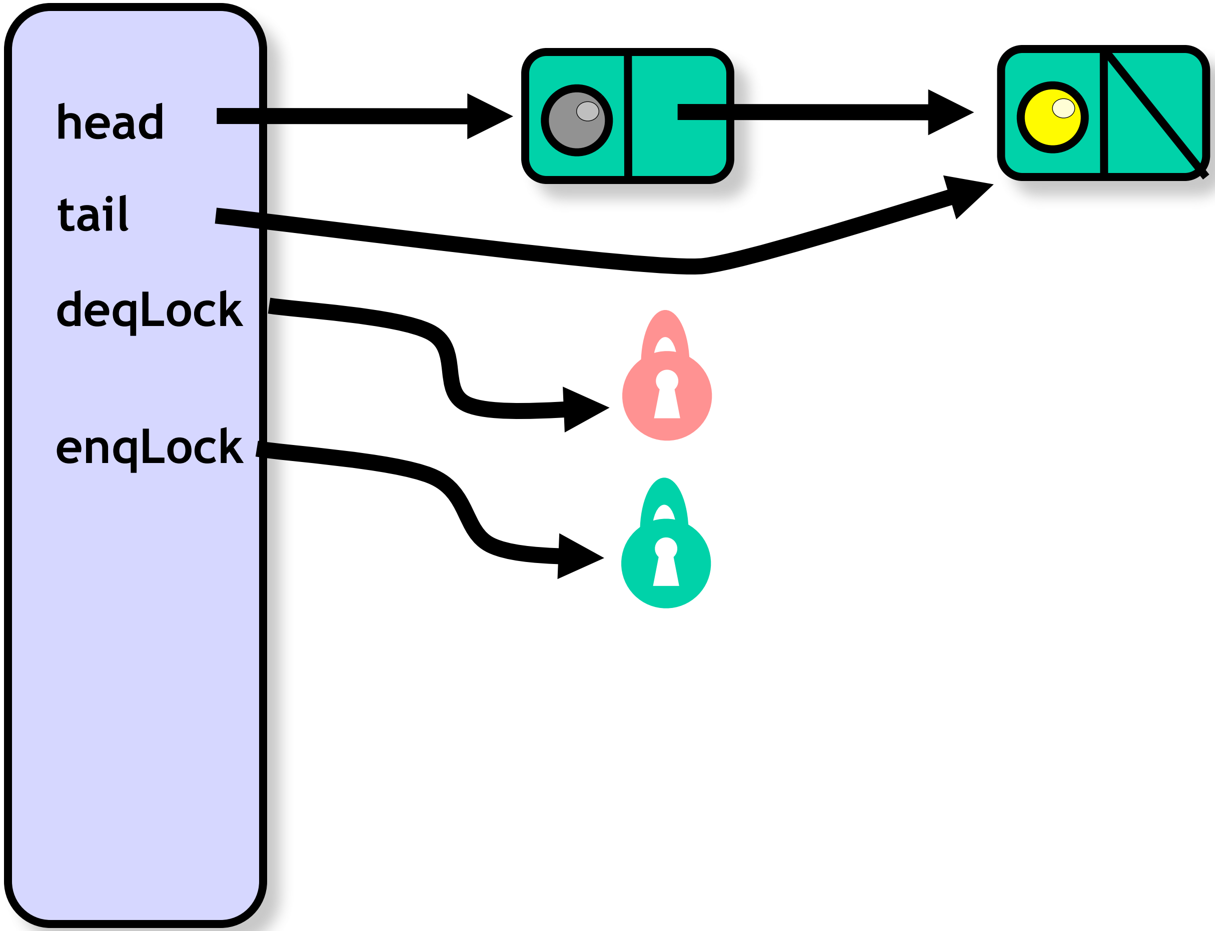
Lock out other enq() calls

Ingredients: Not Done Yet

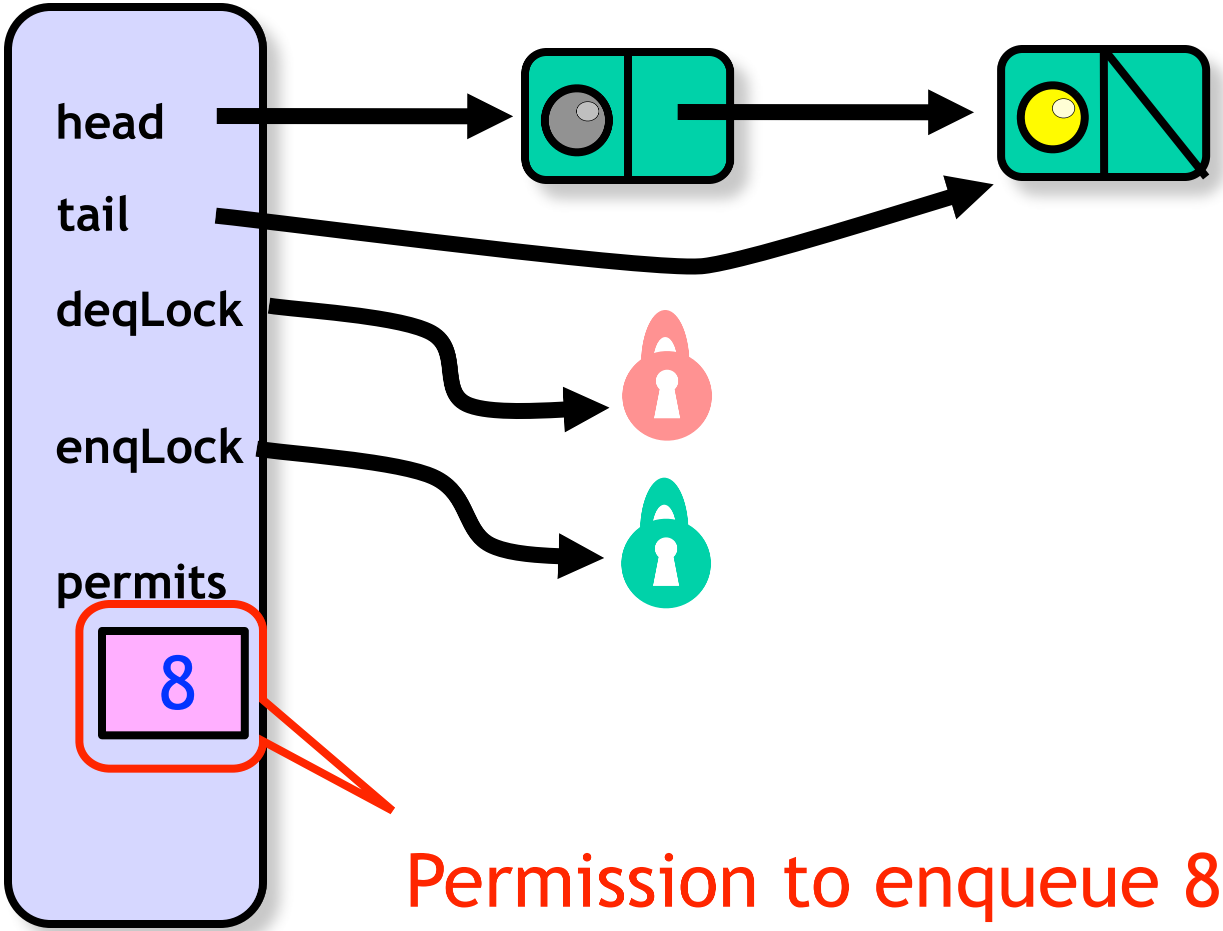


Need to tell whether queue is full or empty

Ingredients: Not Done Yet

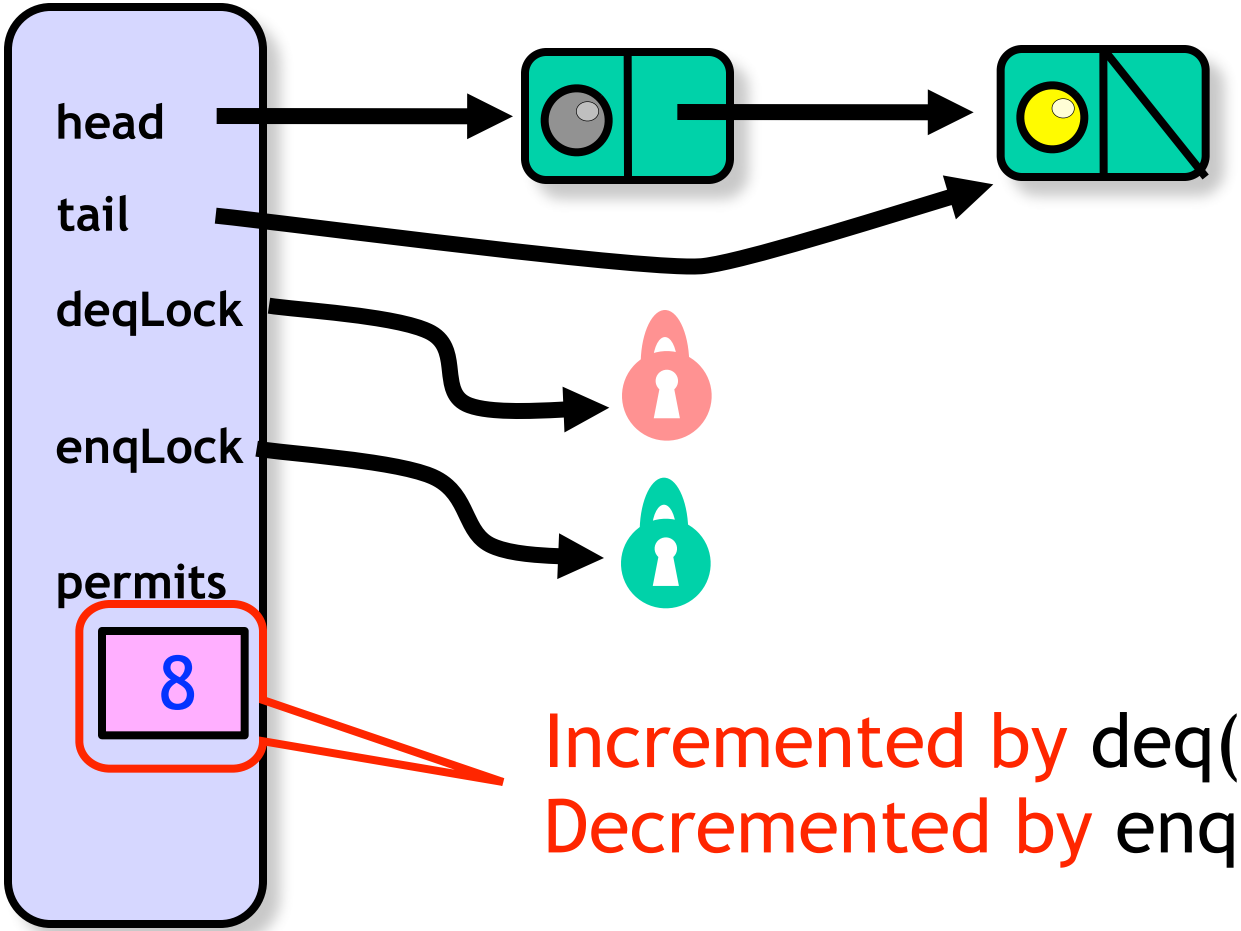


Ingredients: Not Done Yet



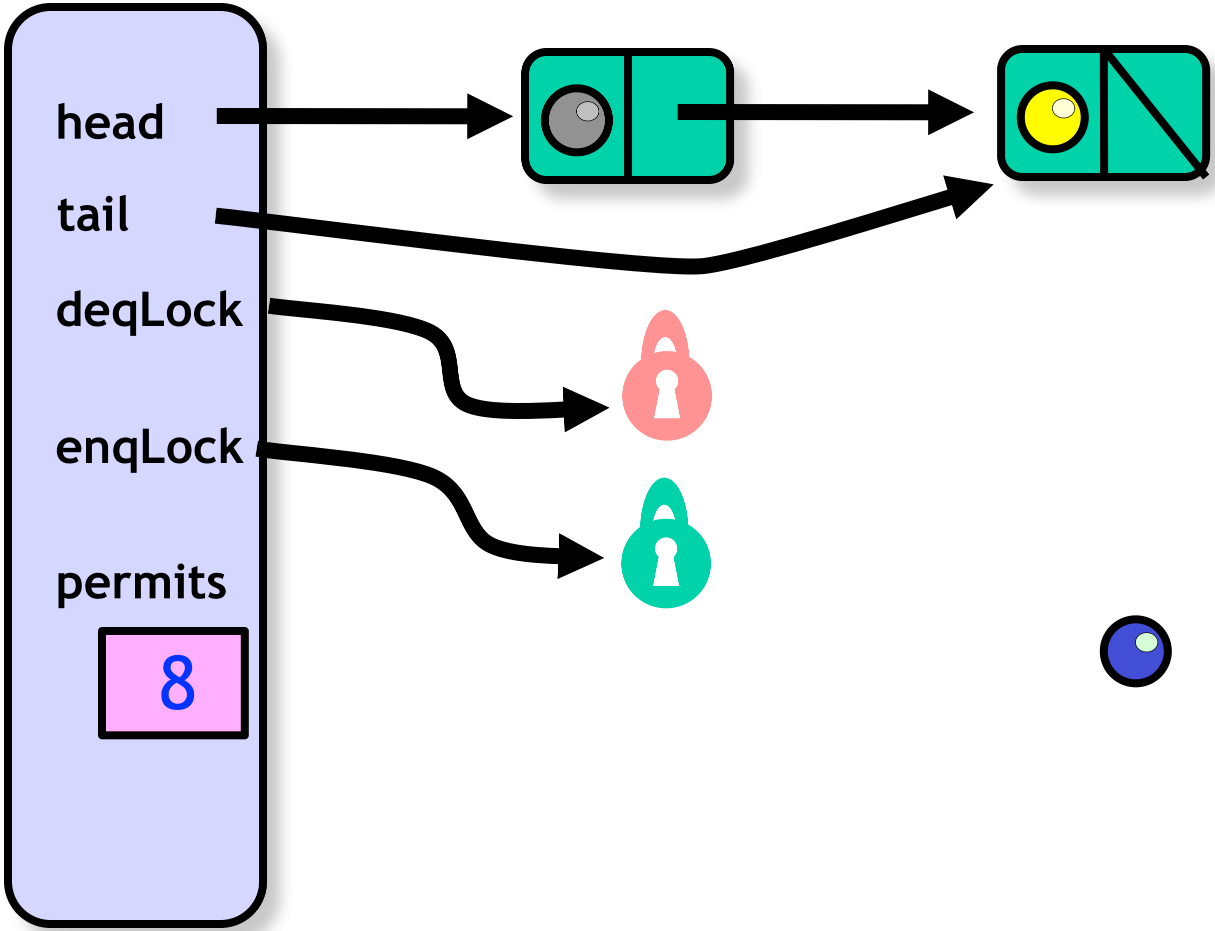
Permission to enqueue 8 items

Ingredients: Not Done Yet

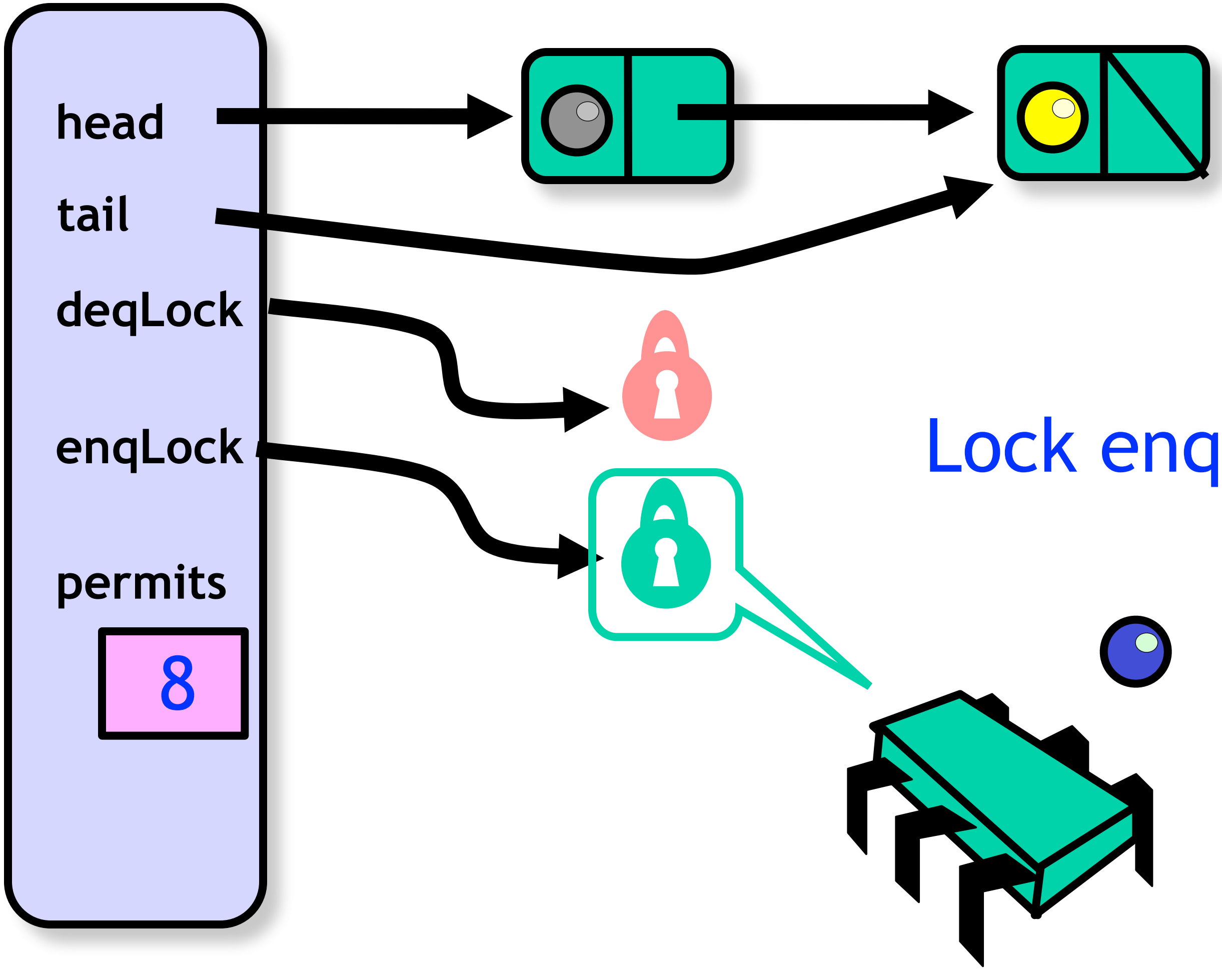


Incremented by deq()
Decremented by enq()

Enqueuer

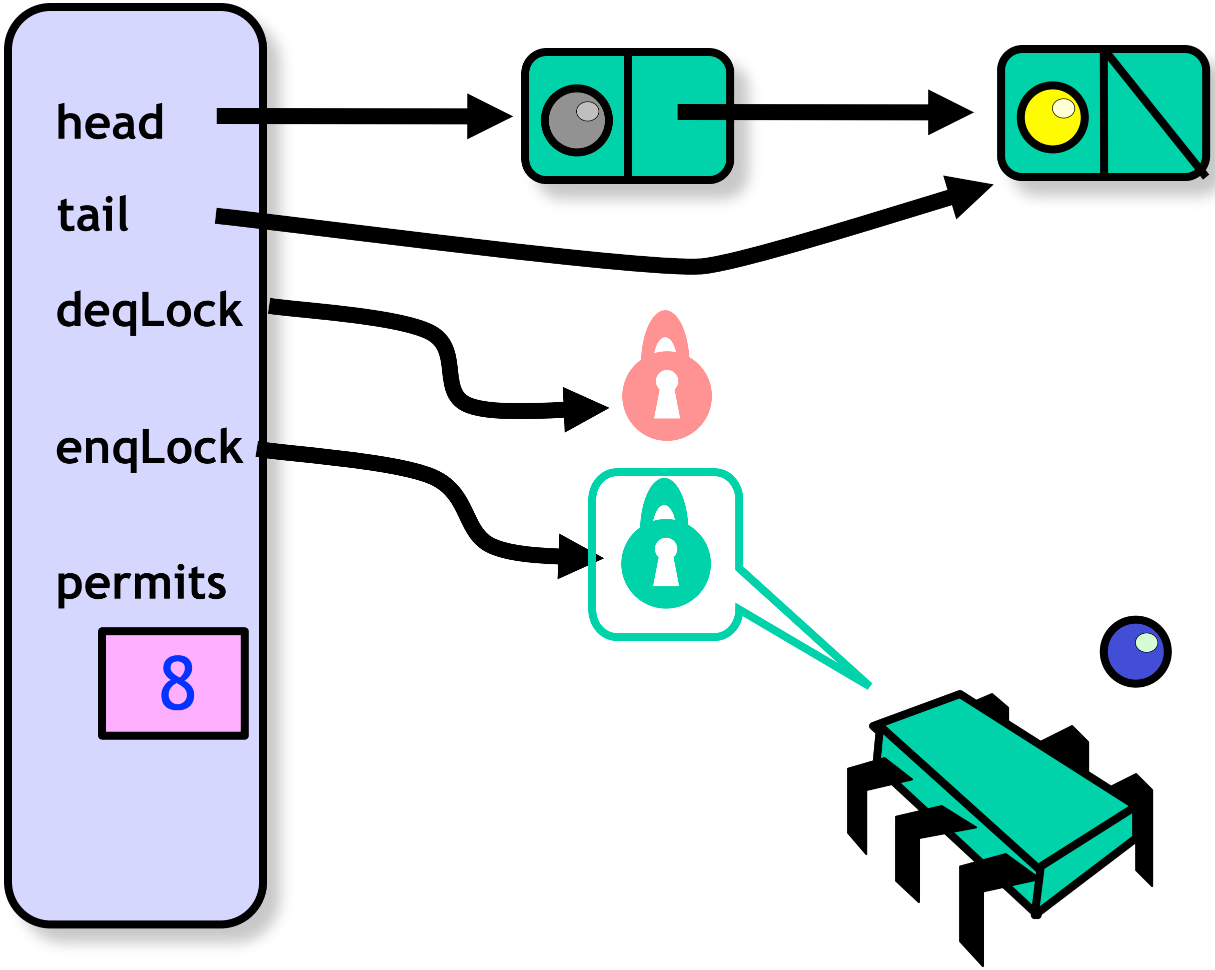


Enqueuer

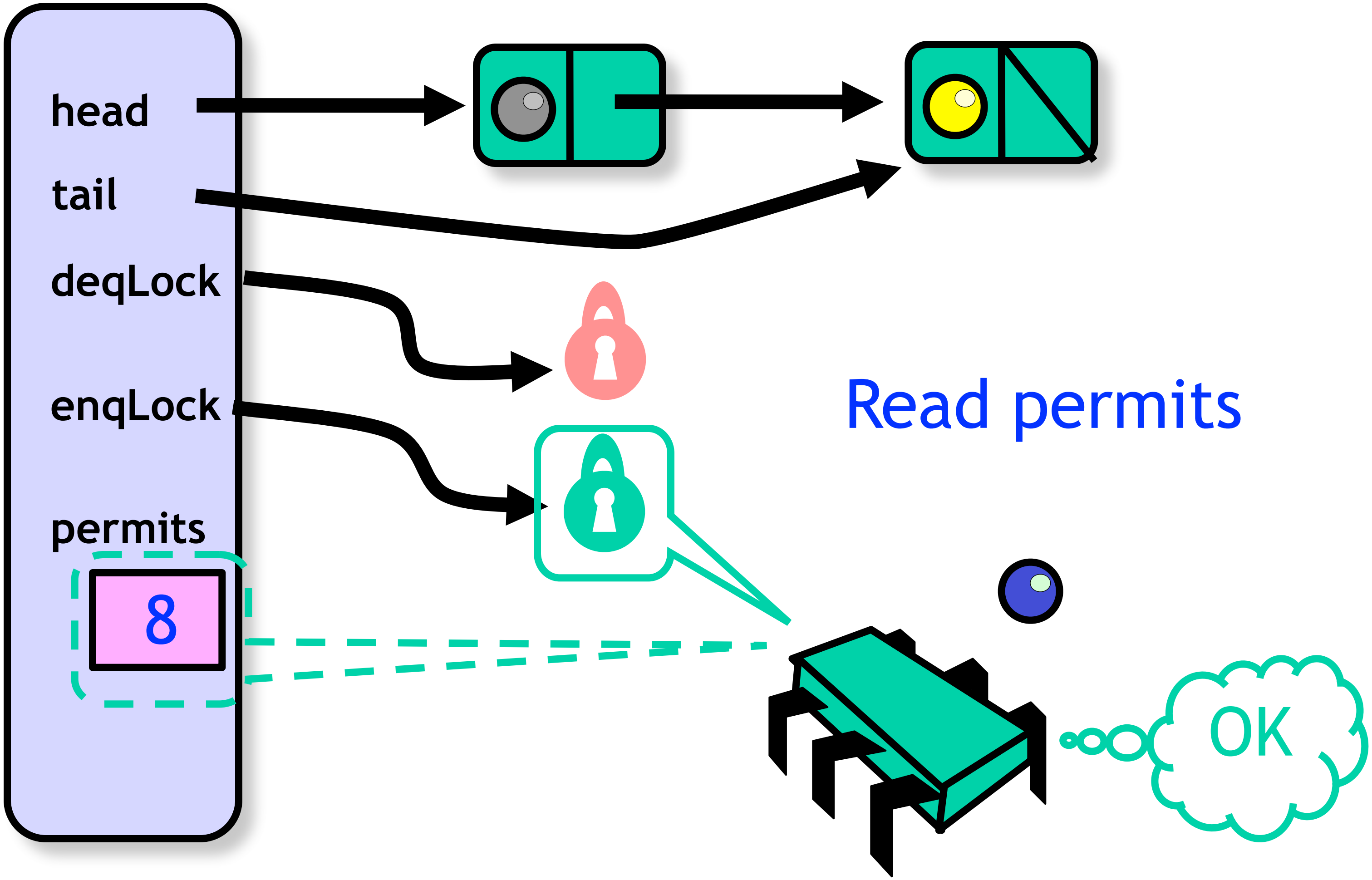


Lock enqLock

Enqueuer

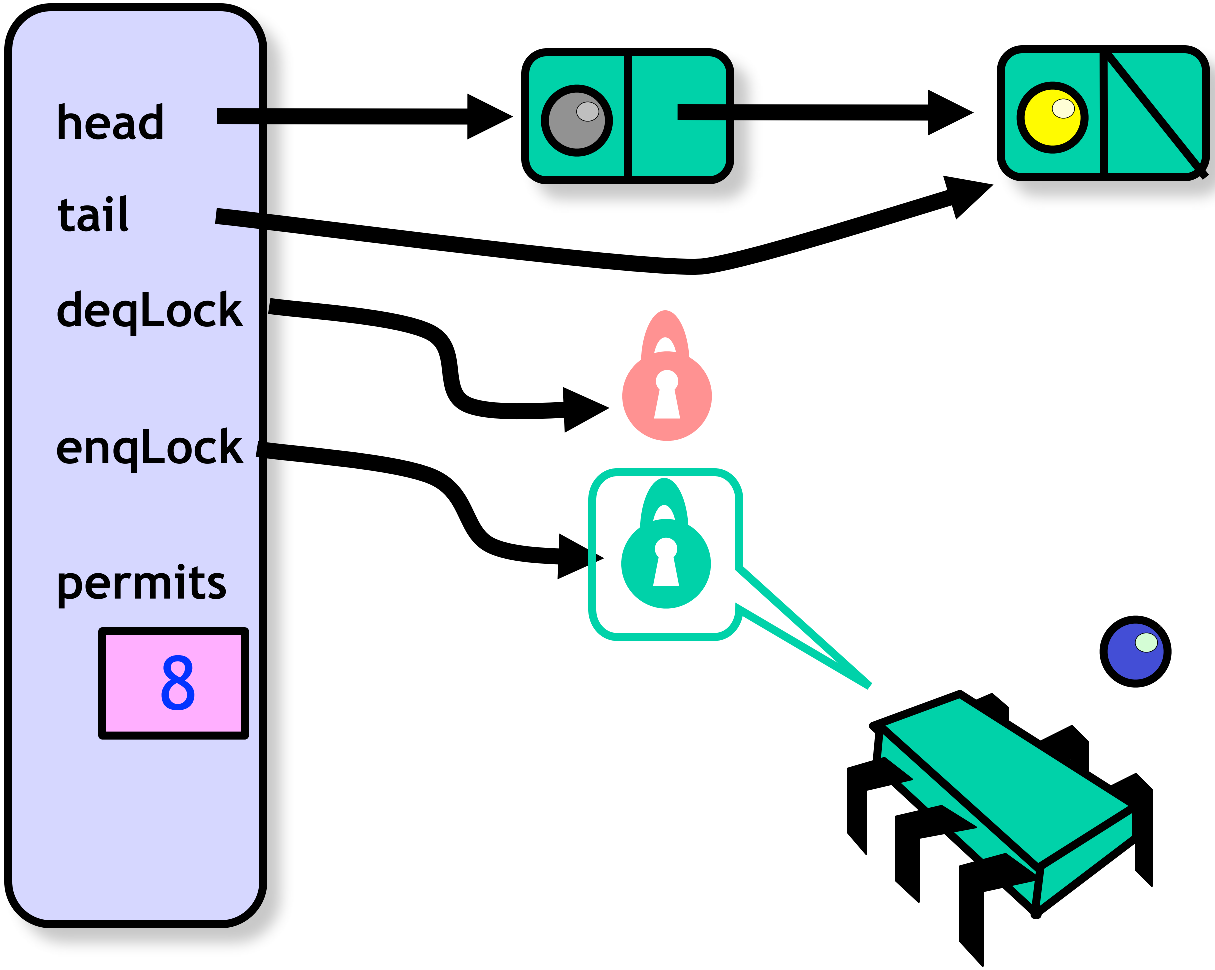


Enqueuer

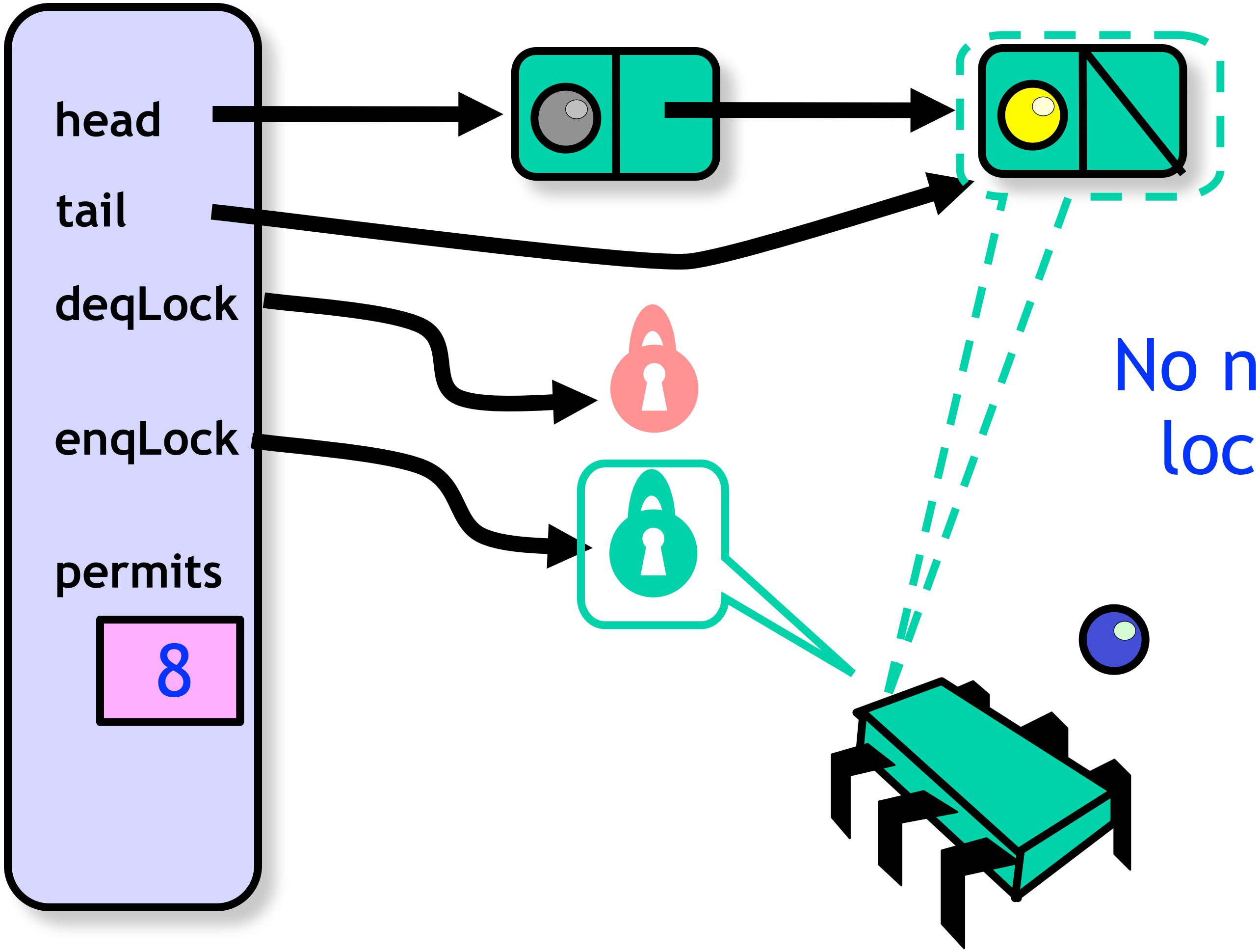


Read permits

Enqueuer

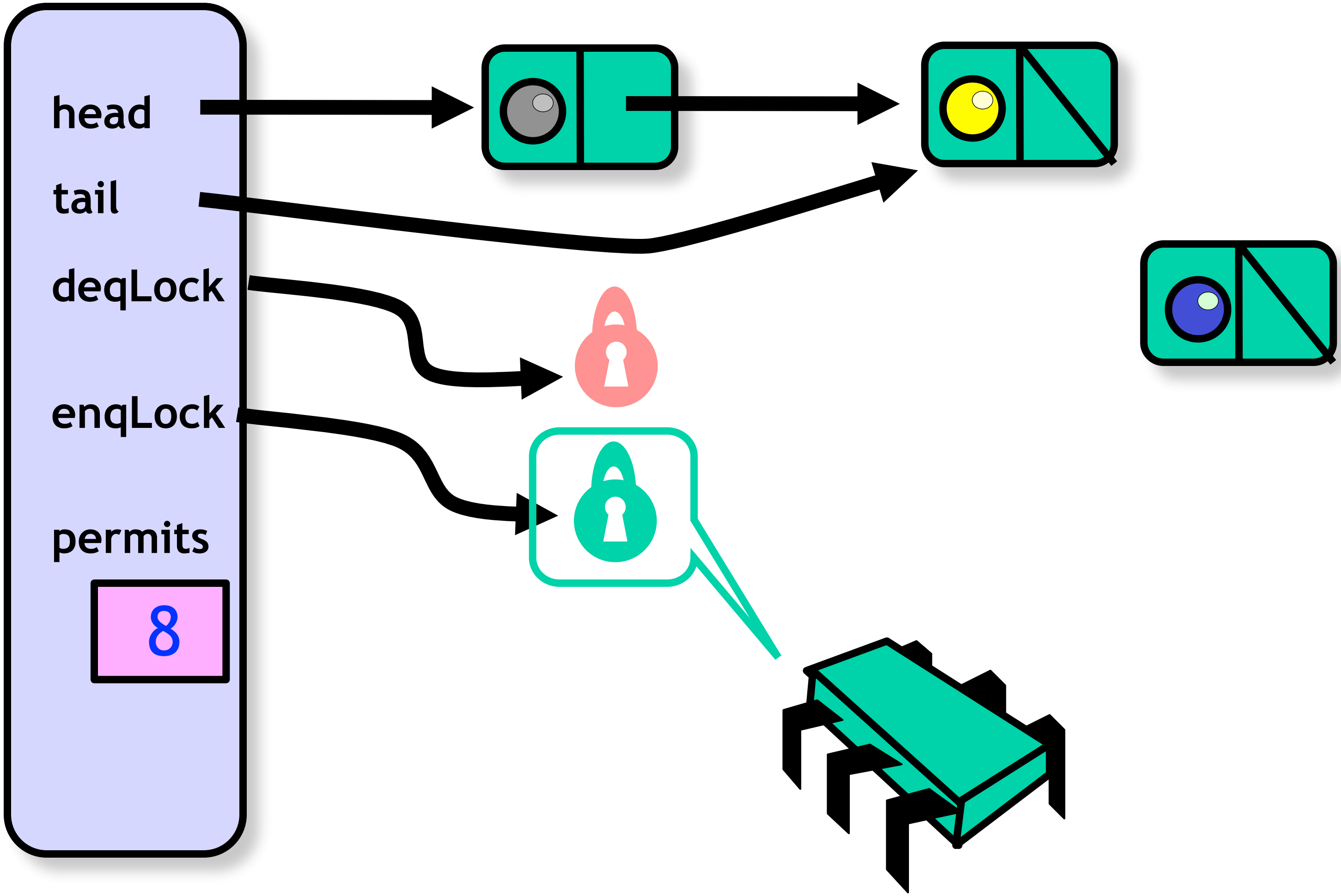


Enqueuer

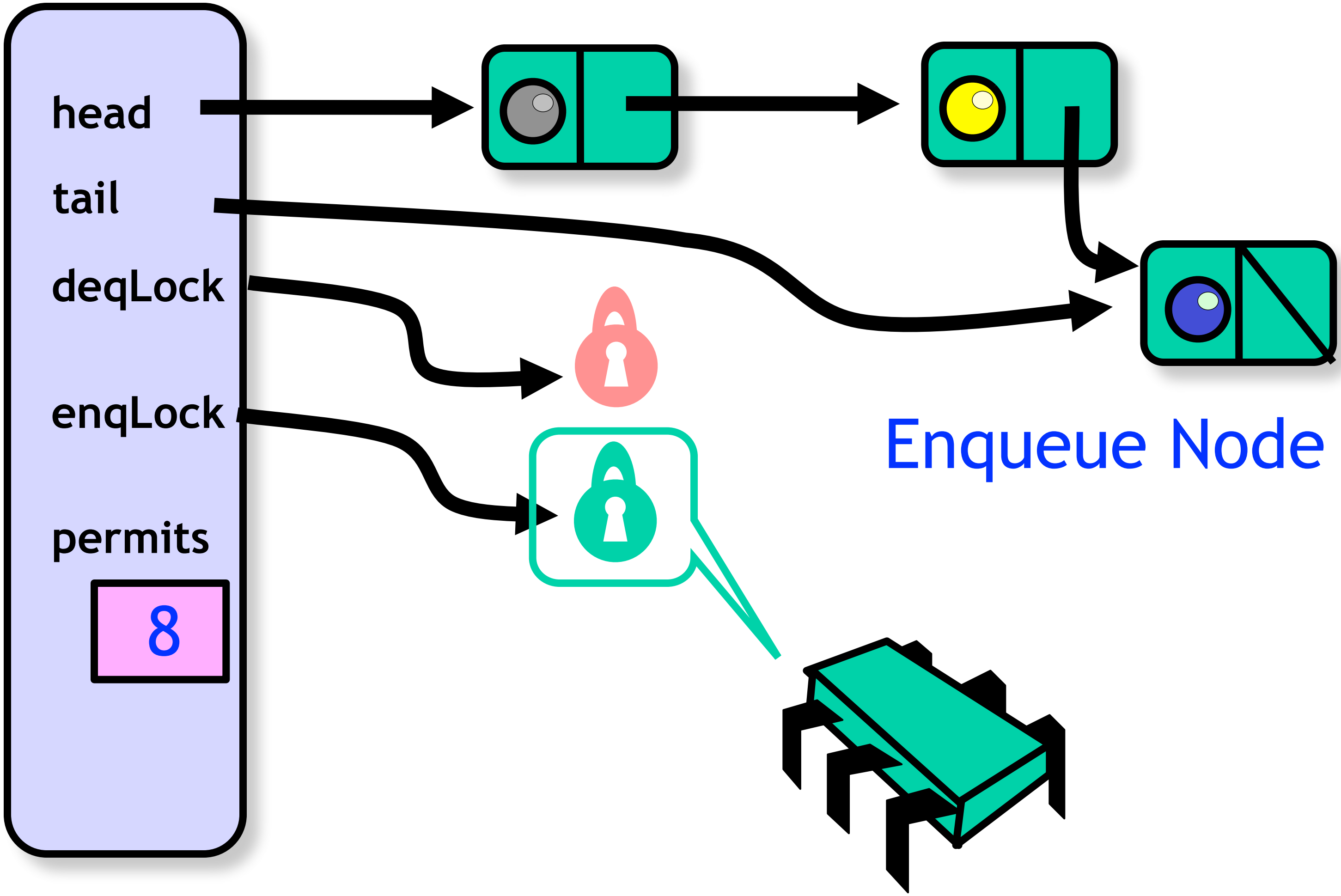


No need to lock tail

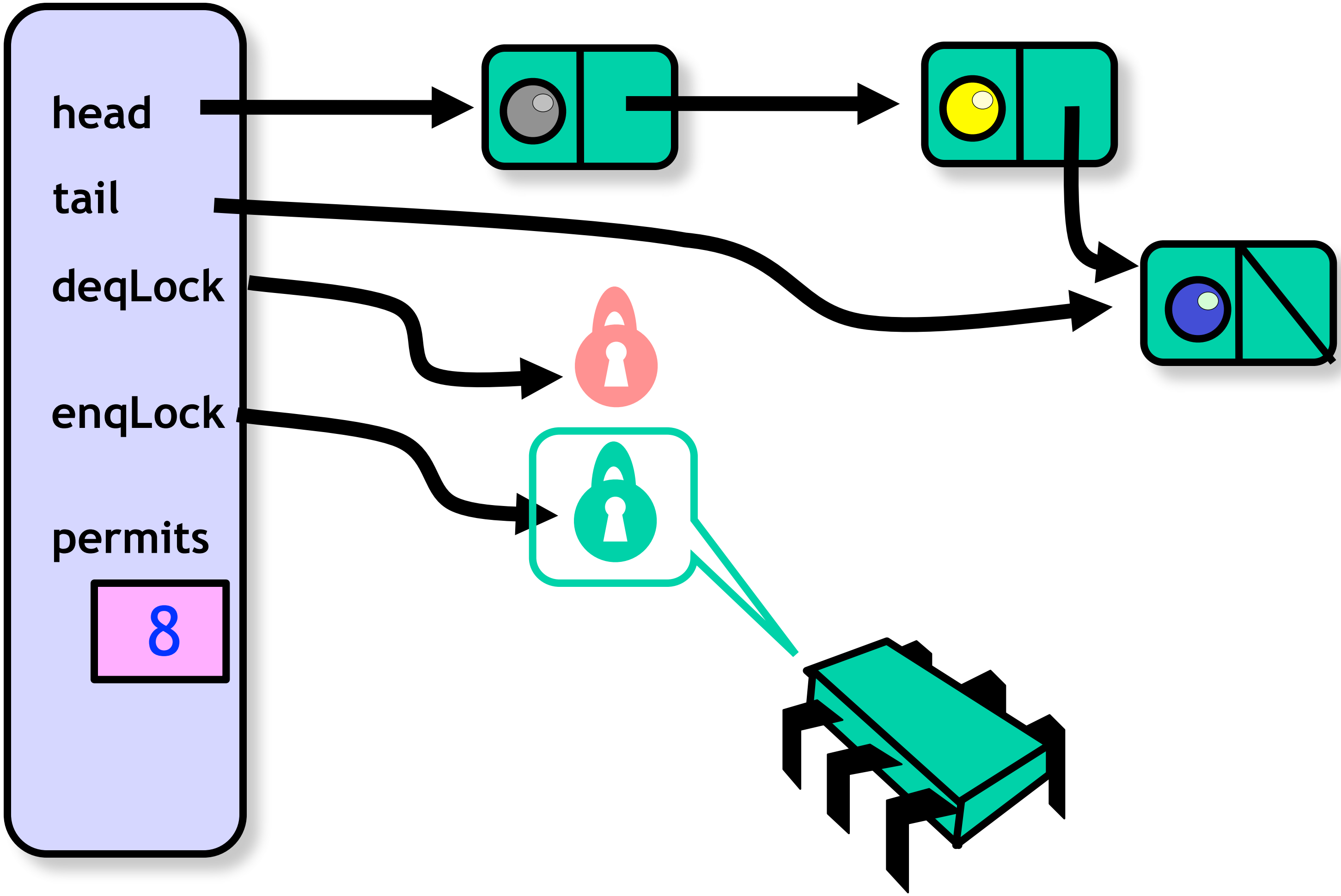
Enqueuer



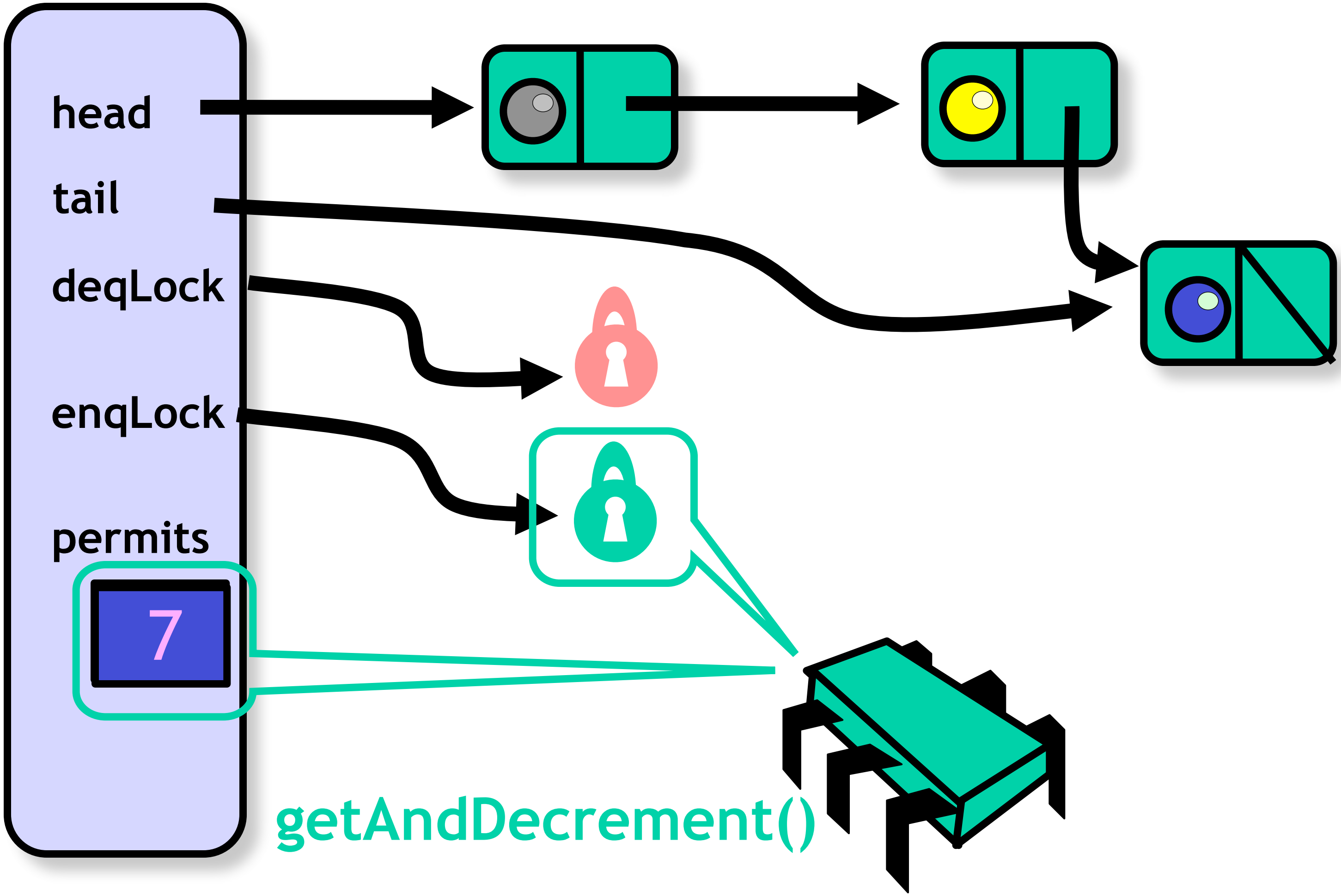
Enqueuer



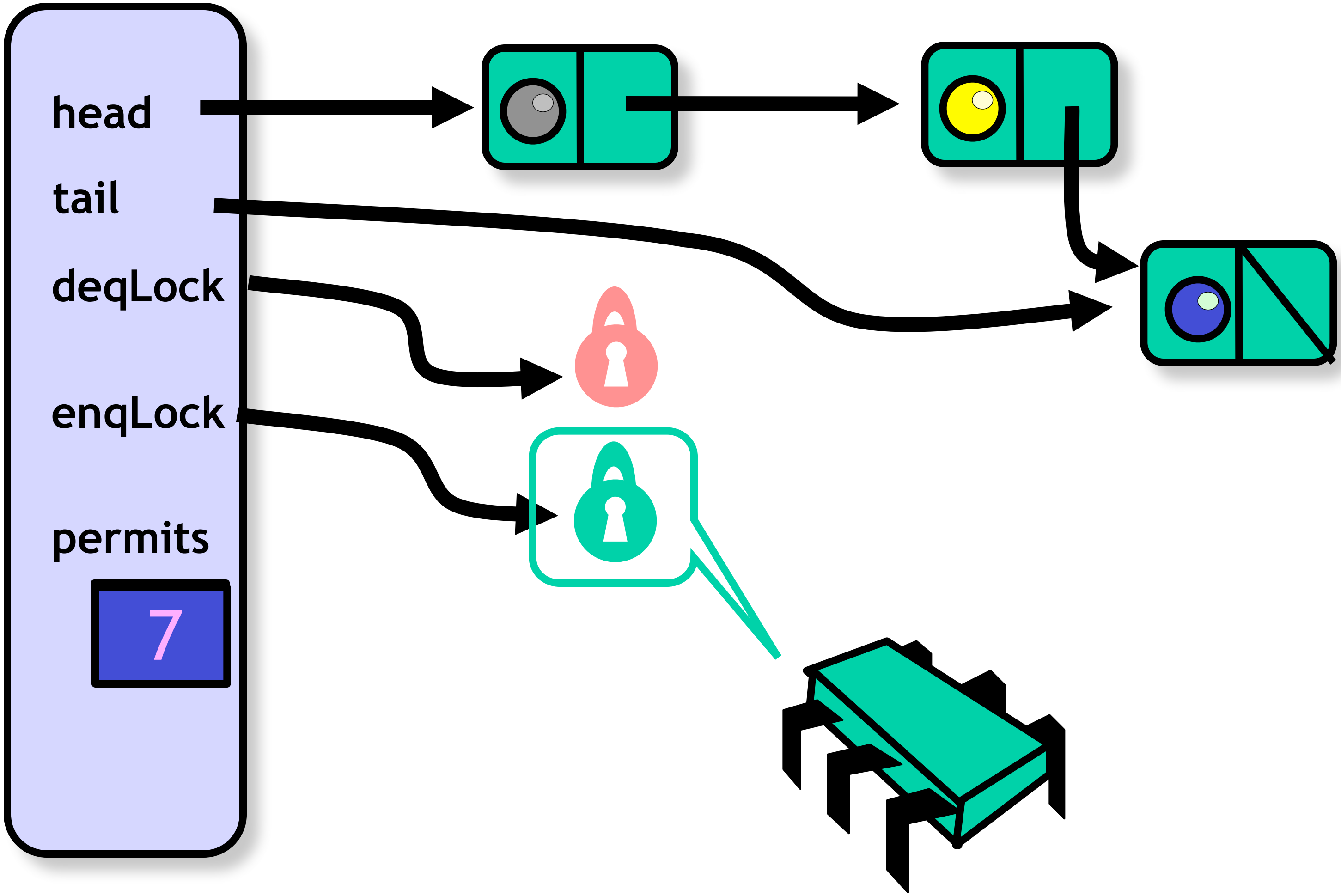
Enqueuer



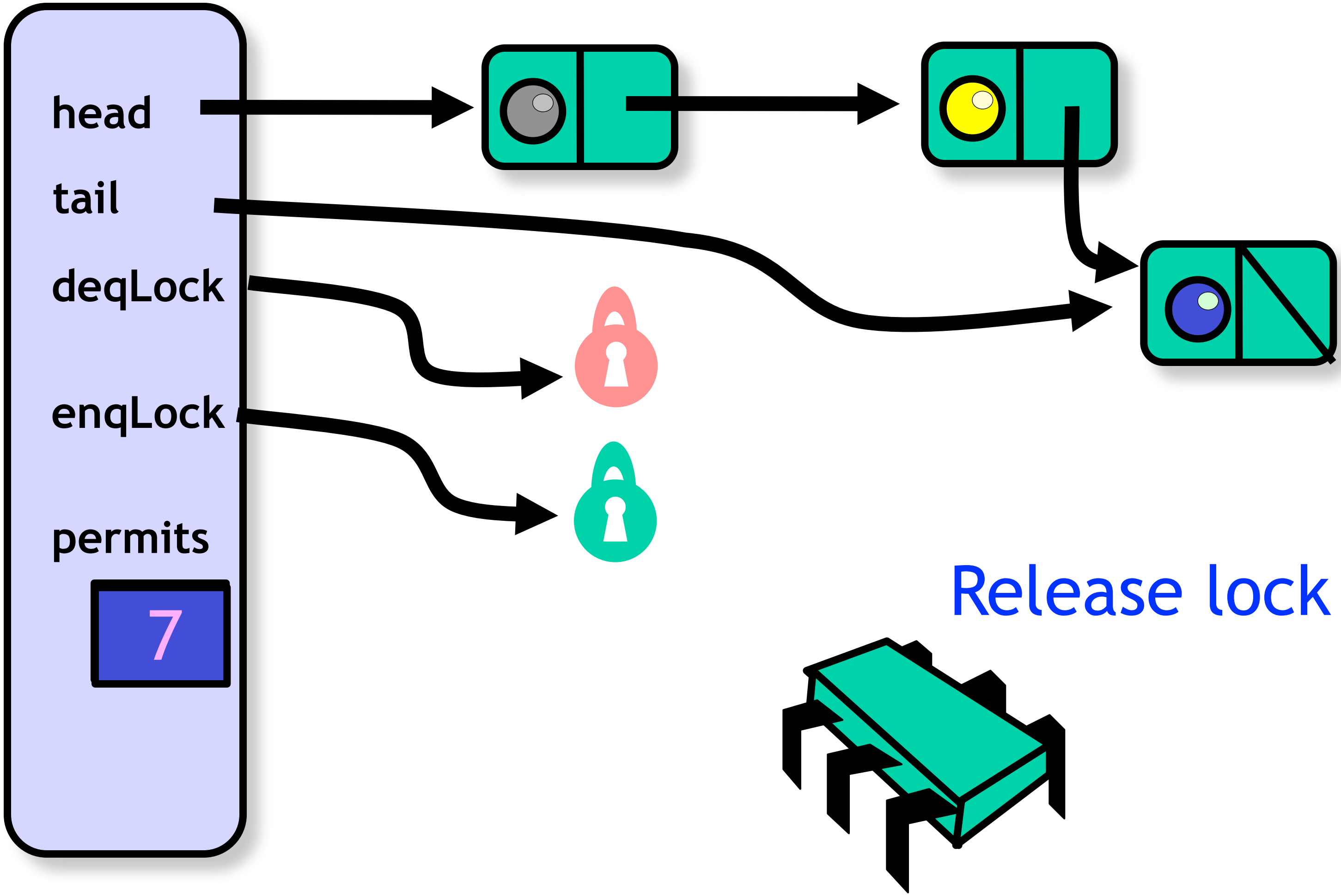
Enqueuer



Enqueuer

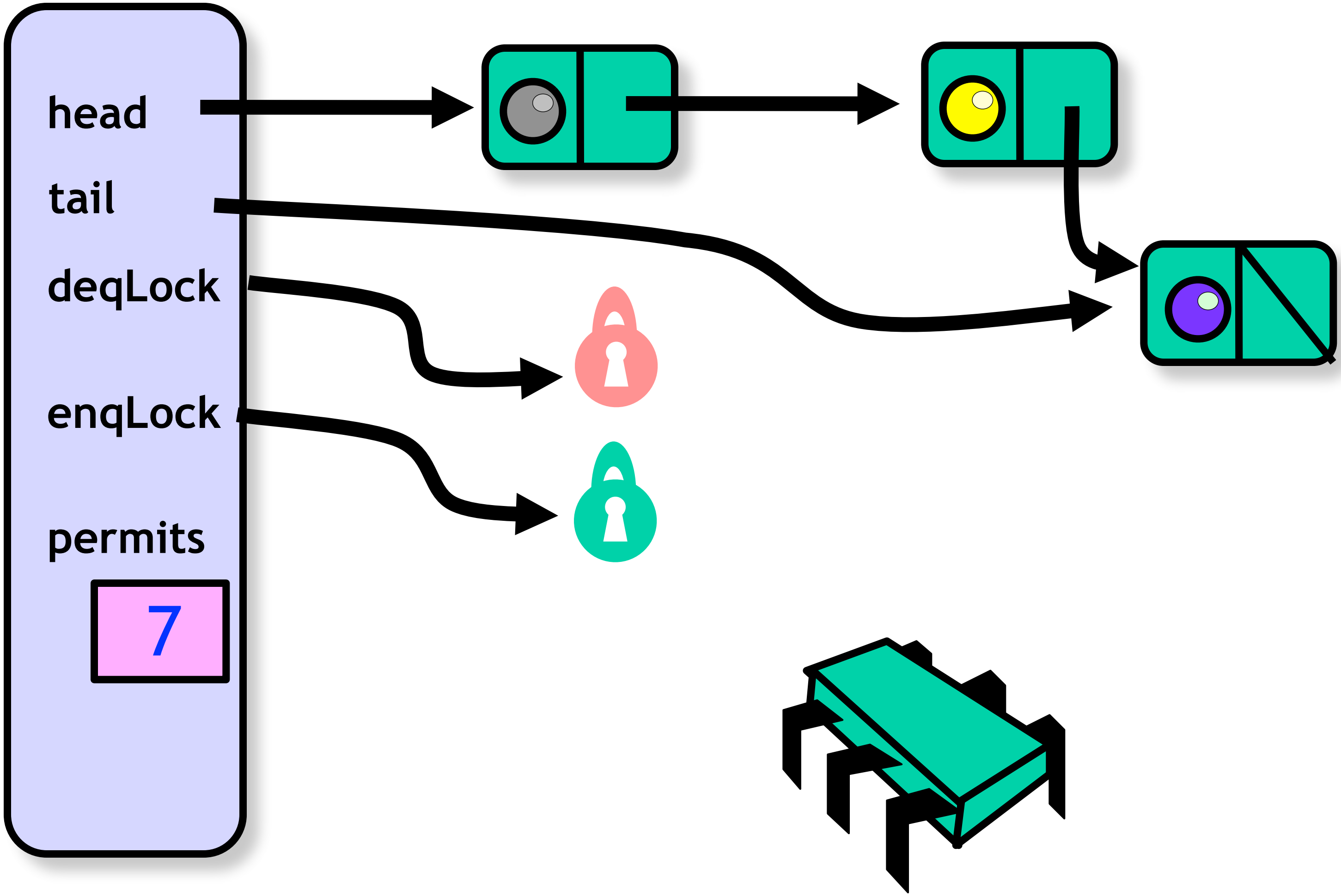


Enqueuer

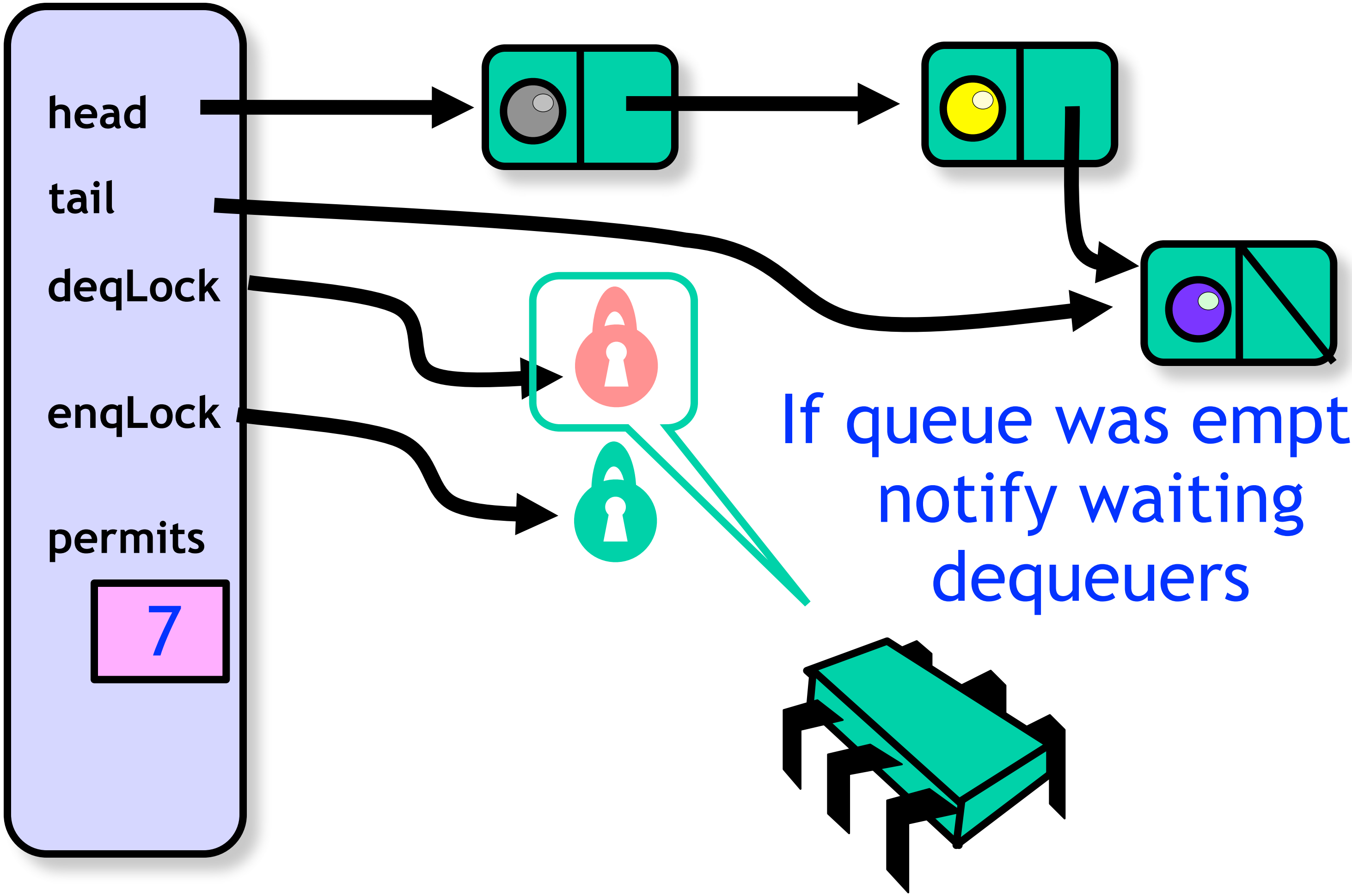


Release lock

Enqueuer

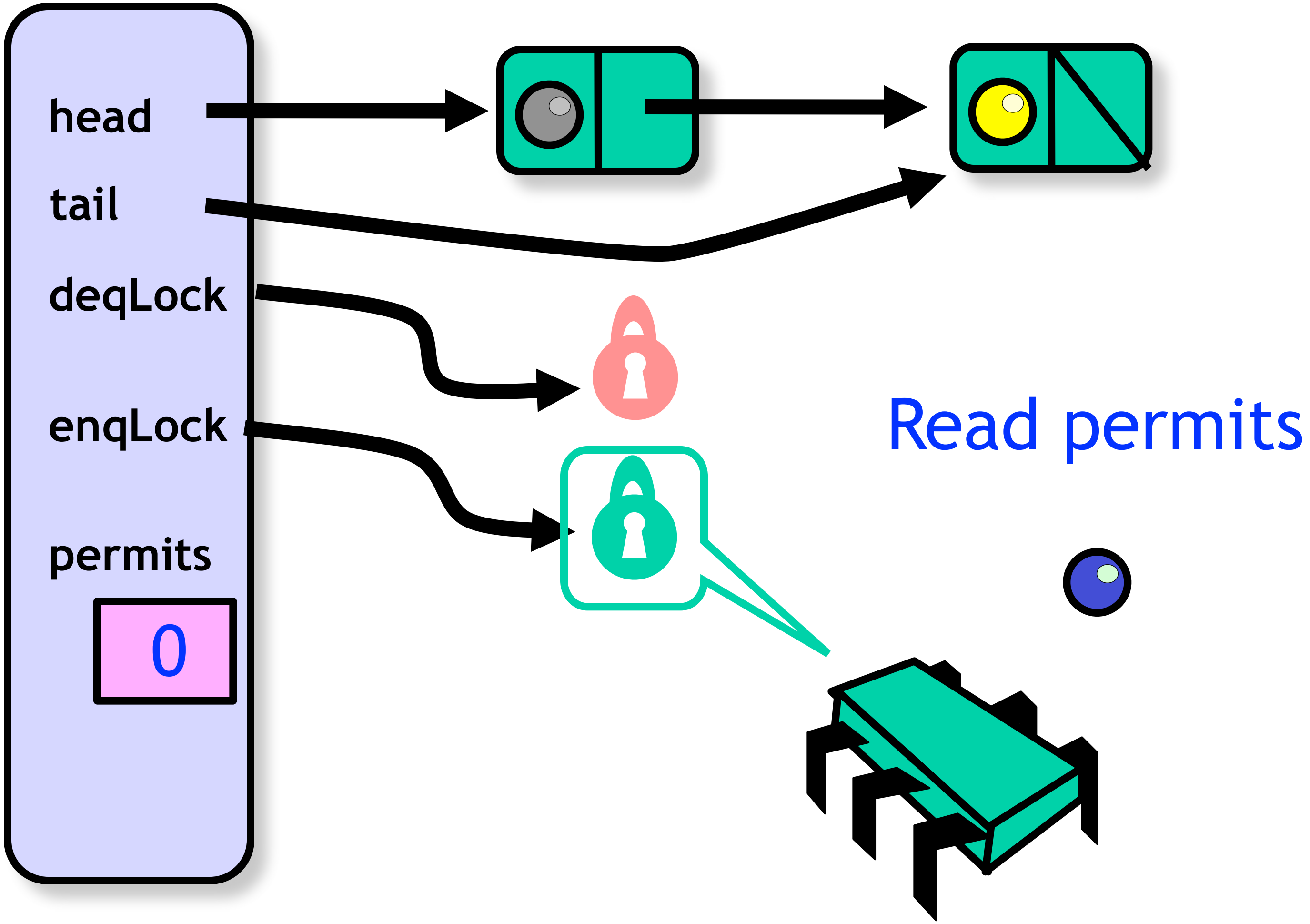


Enqueuer

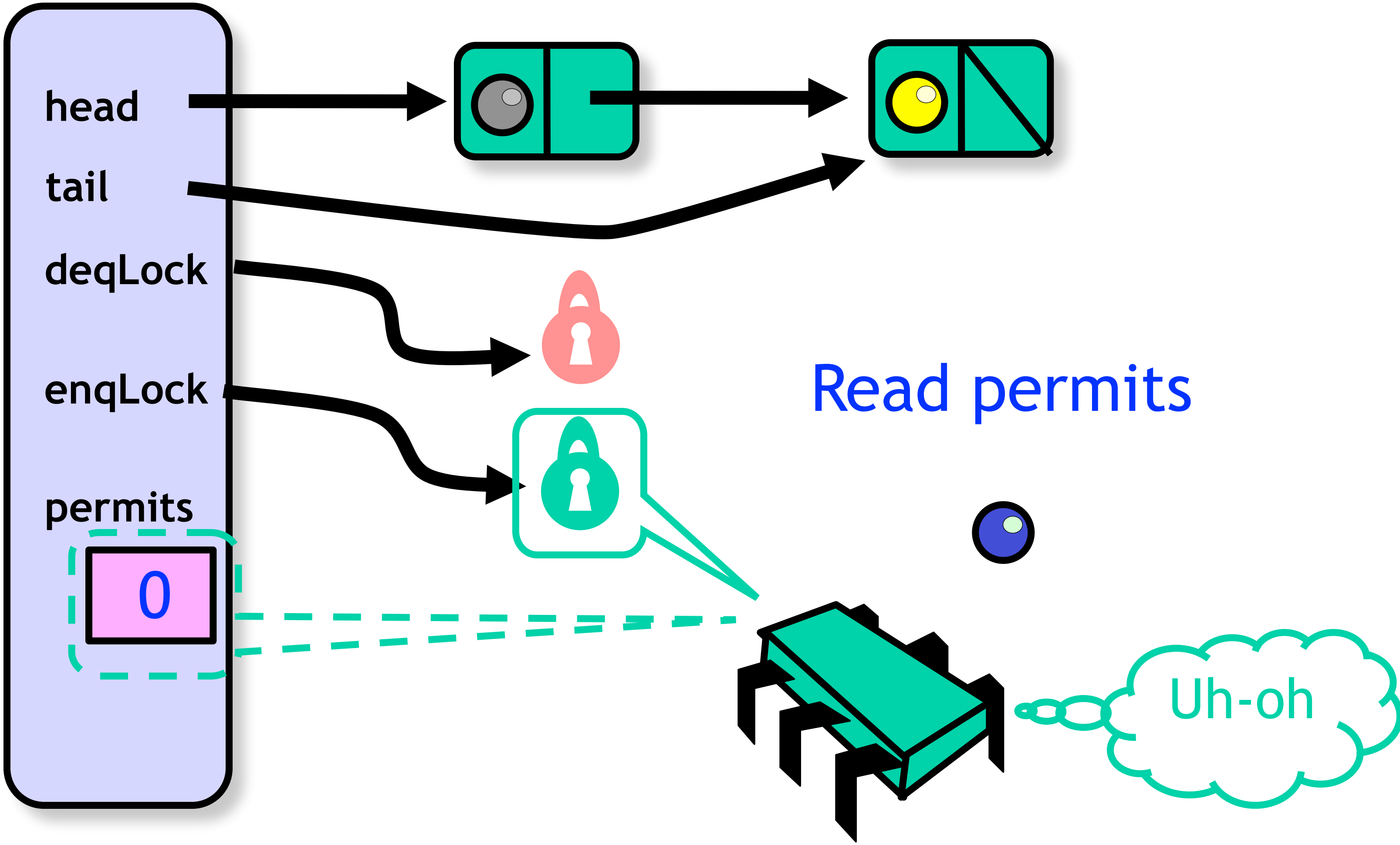


If queue was empty, notify waiting dequeuers

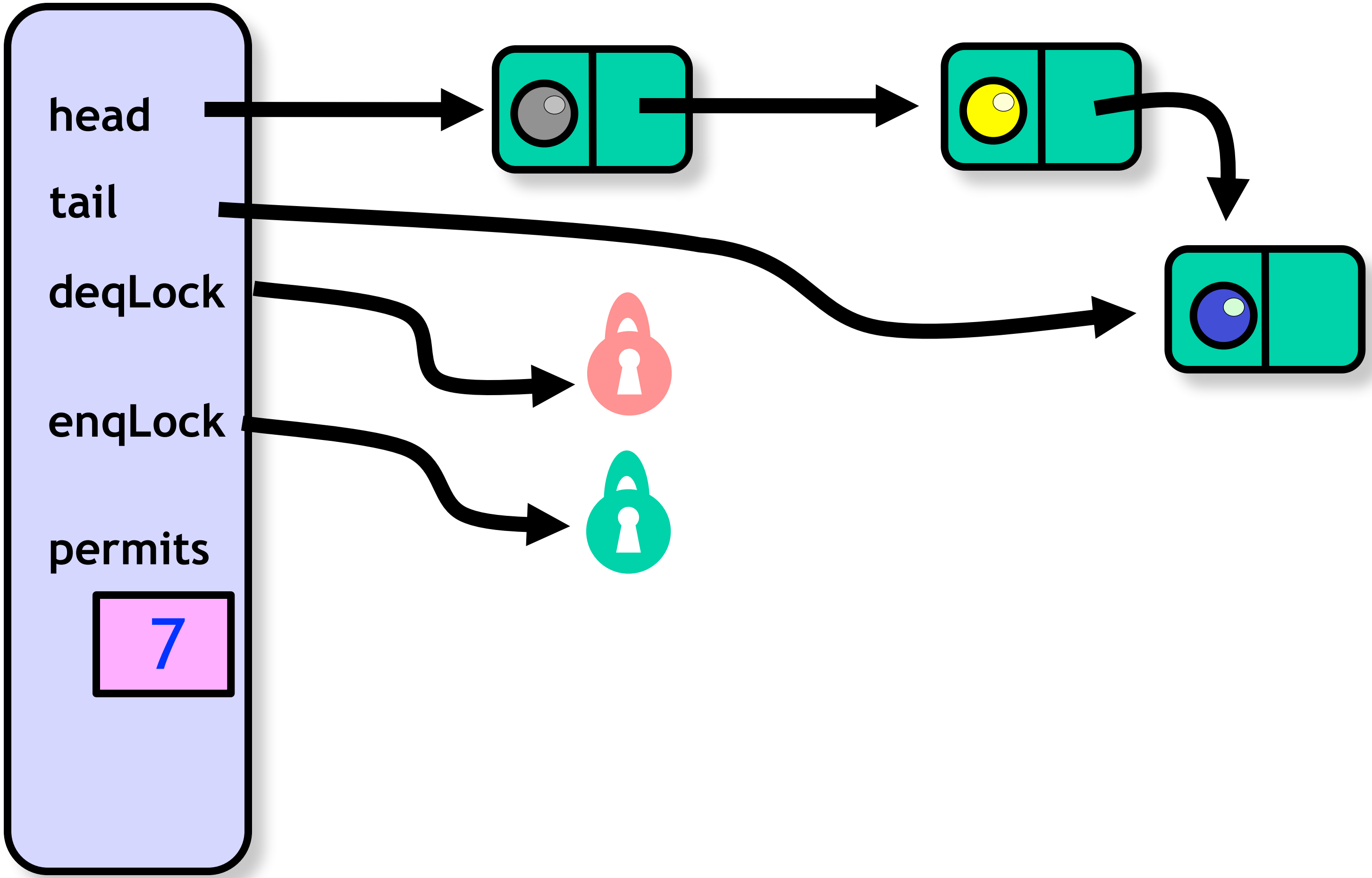
Unsuccesful Enqueuer



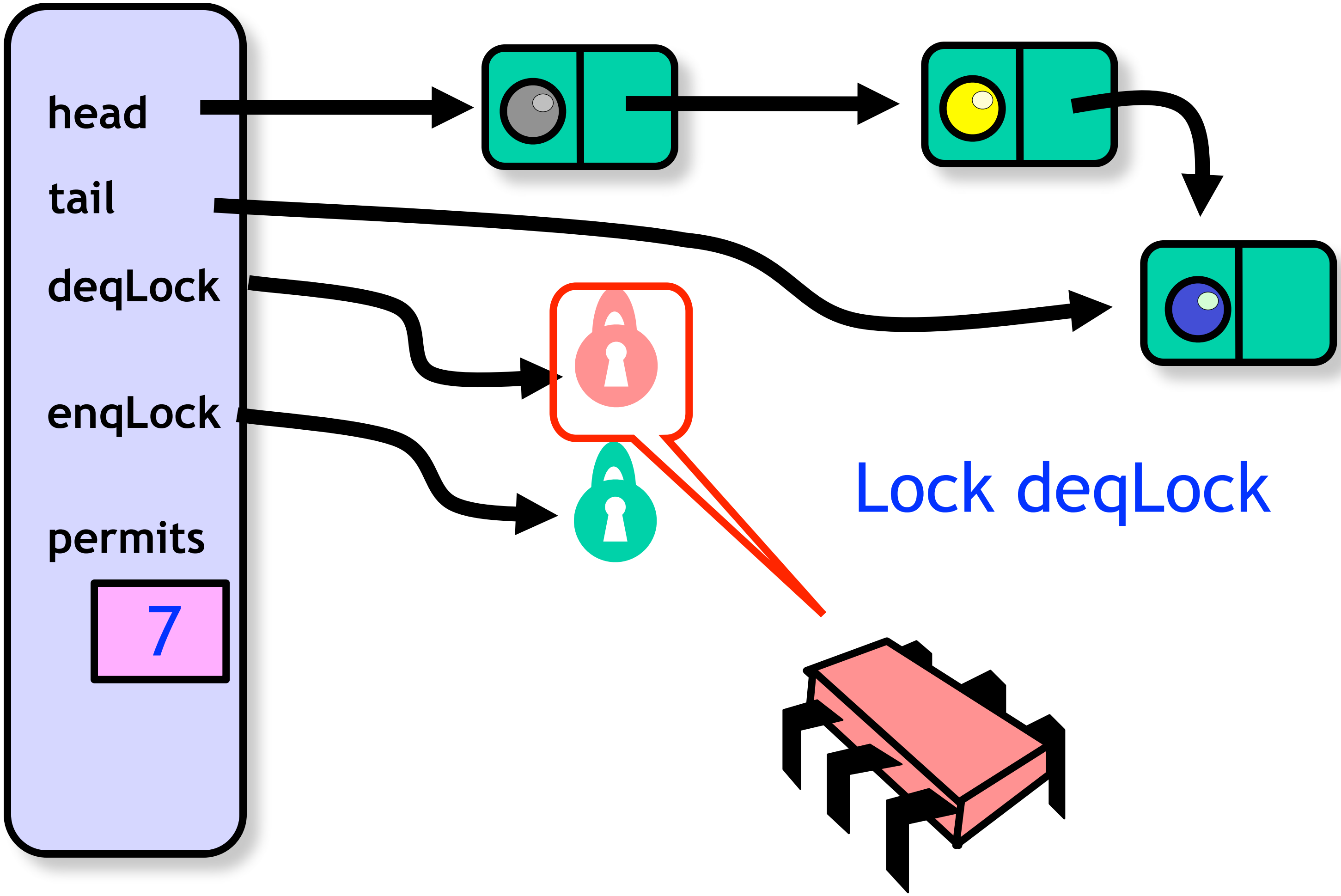
Unsuccesful Enqueuer



Dequeuer

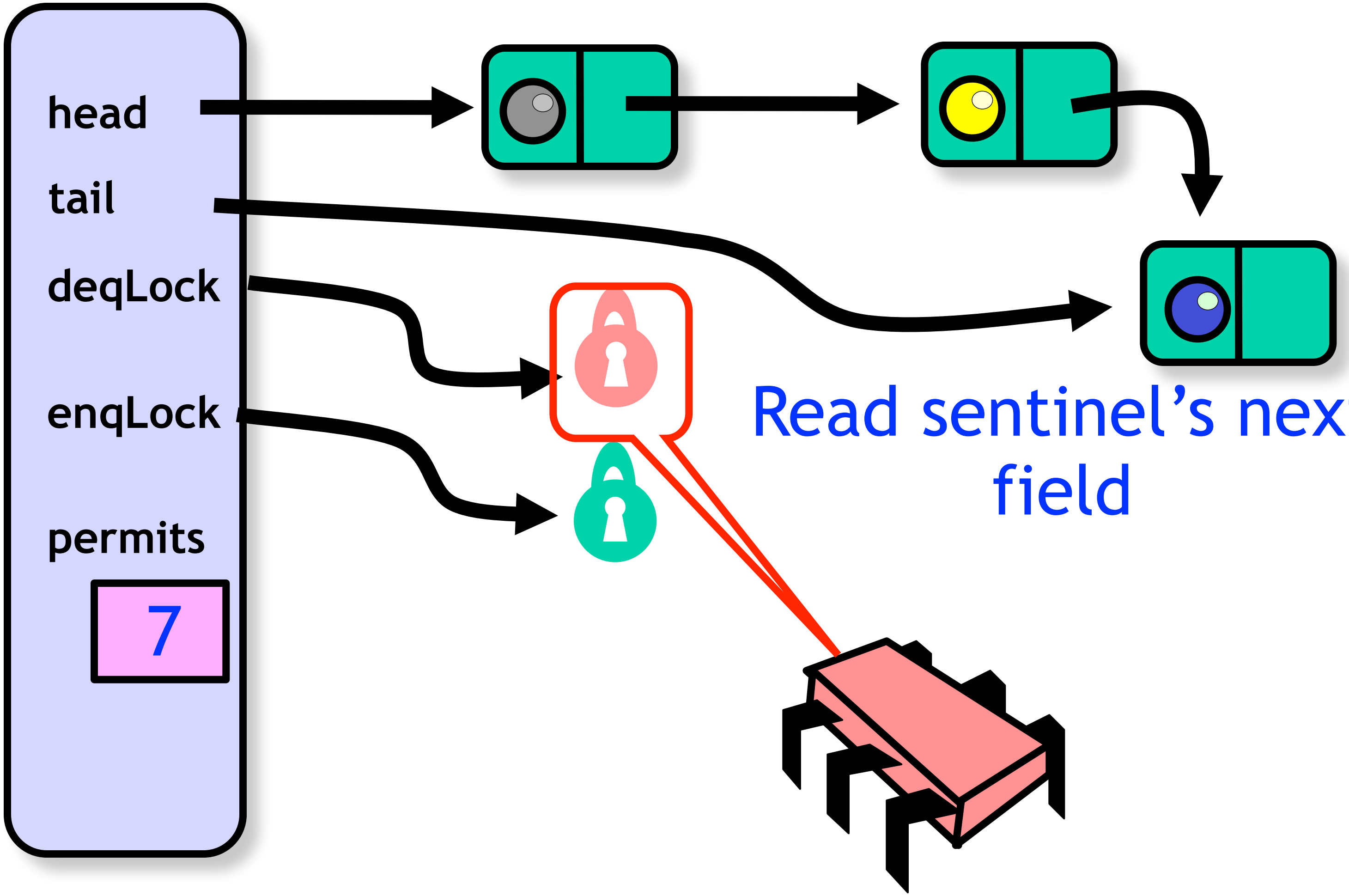


Dequeuer



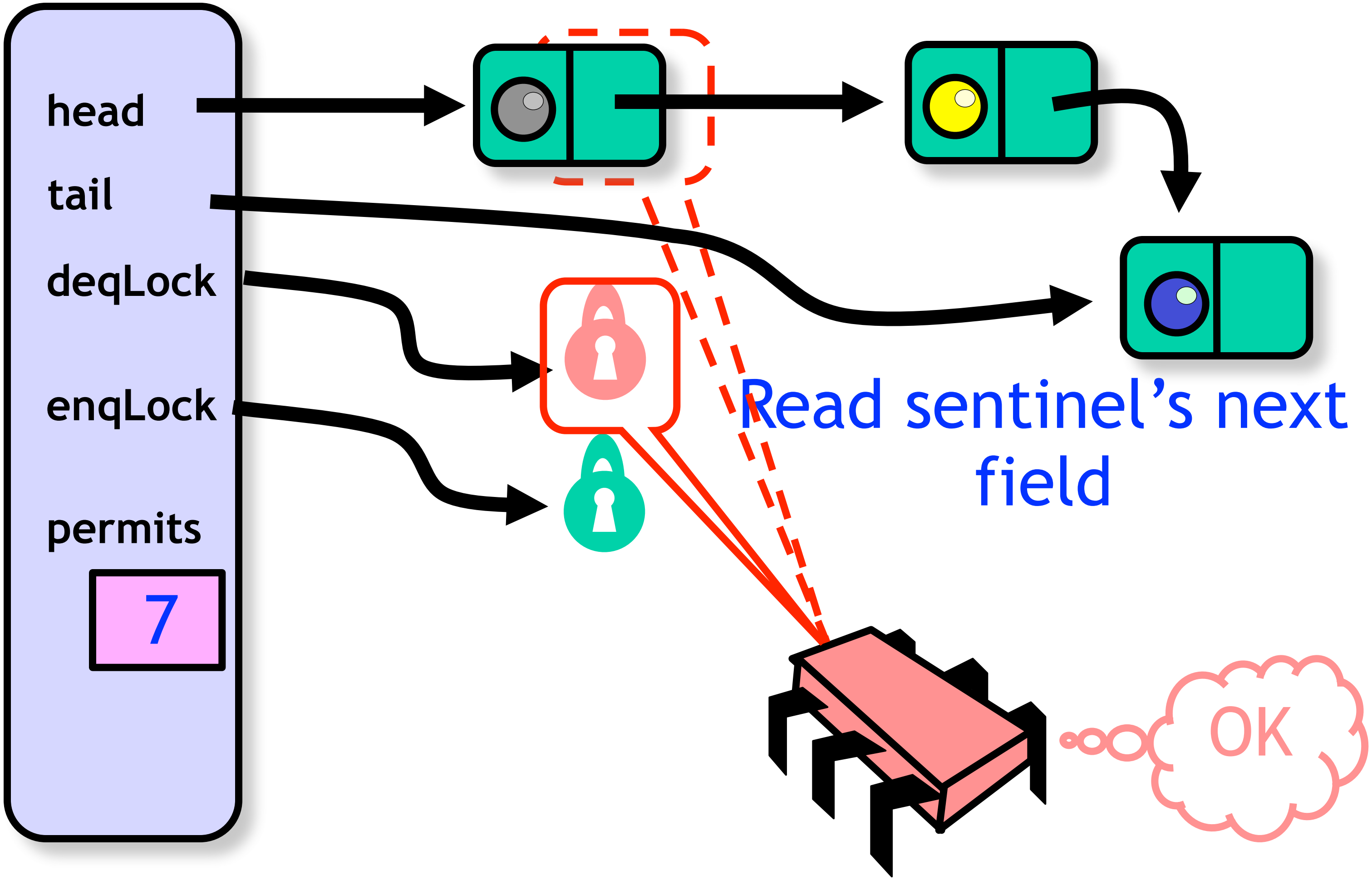
Lock deqLock

Dequeuer



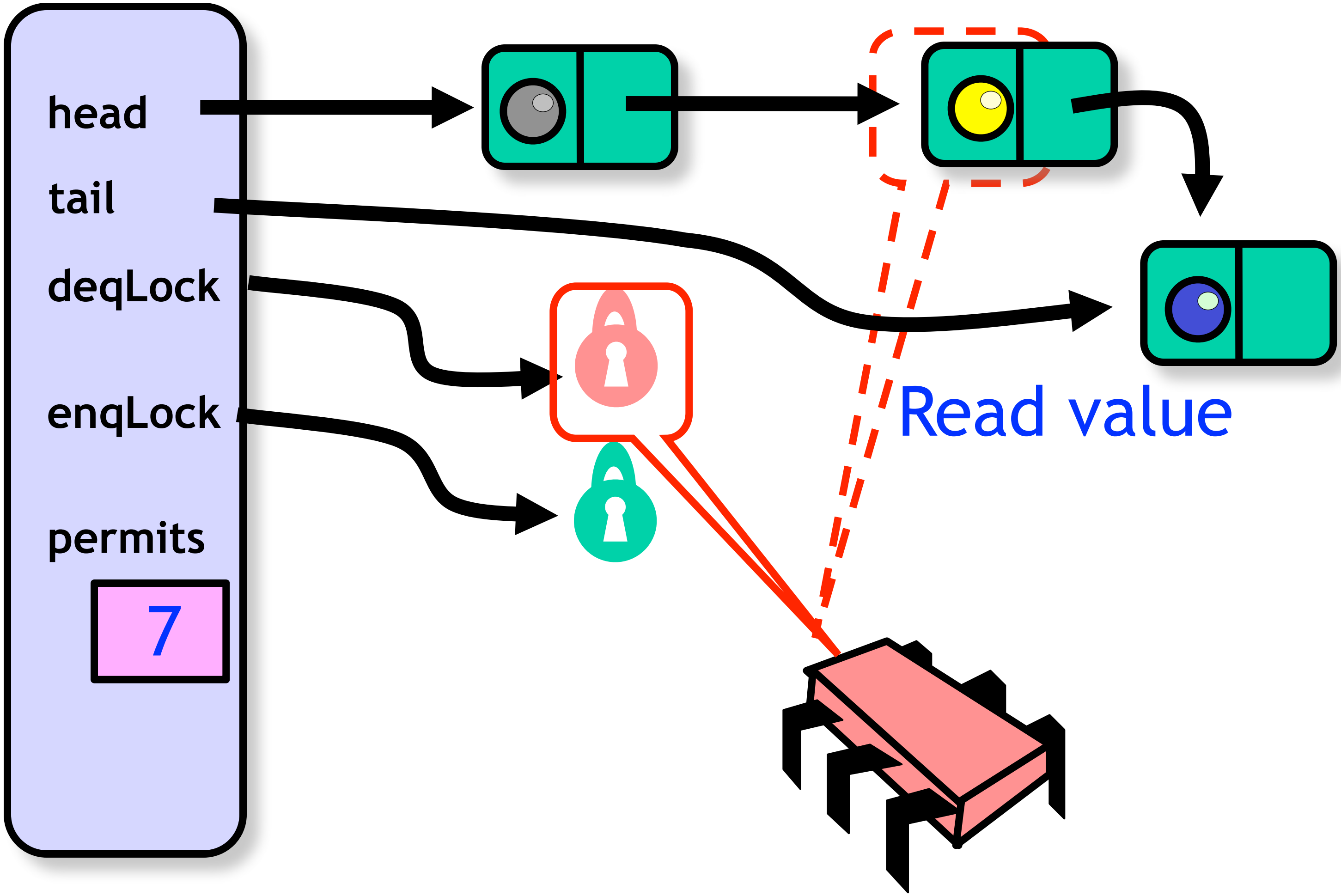
Read sentinel's next field

Dequeuer



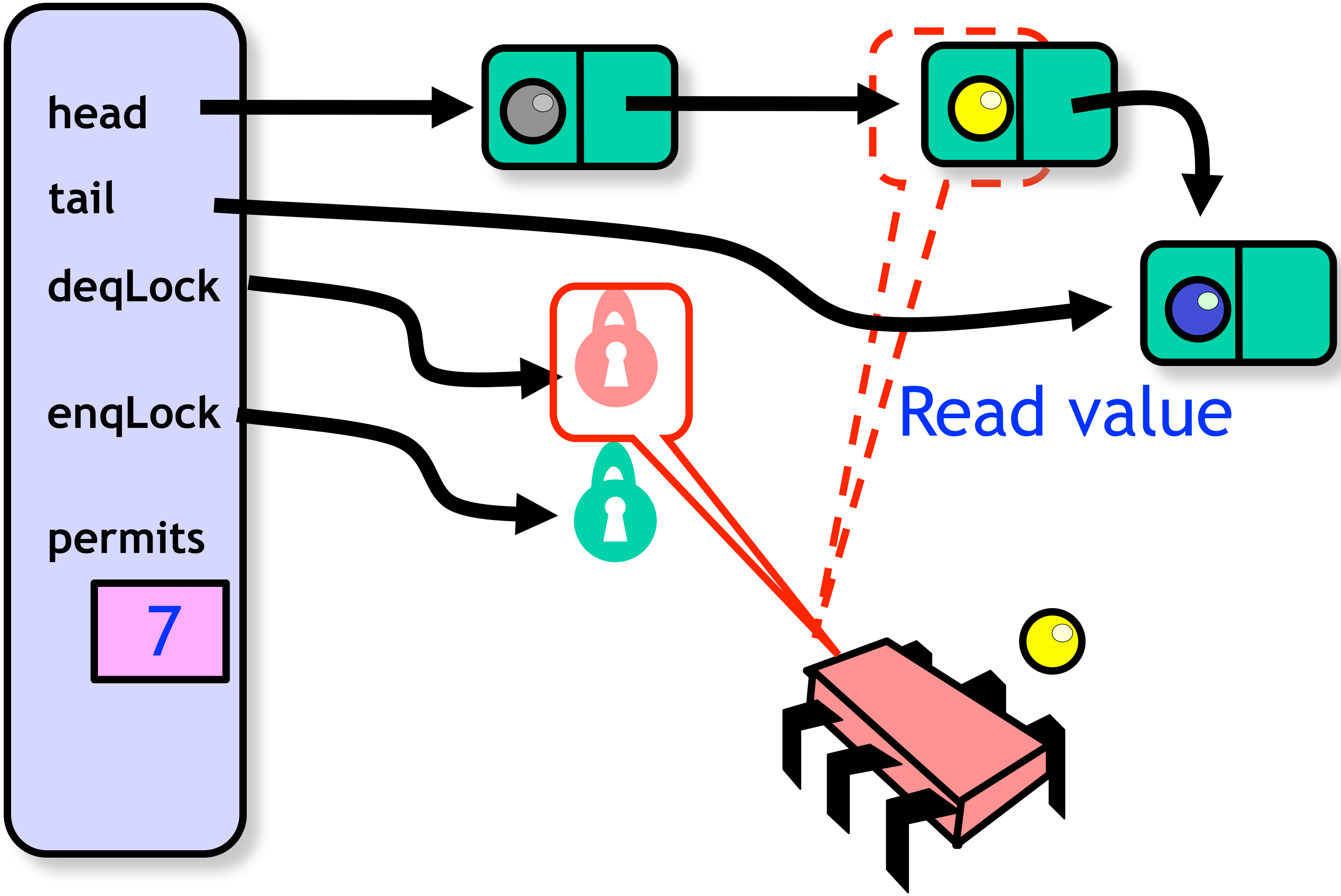
Read sentinel's next field

Dequeuer



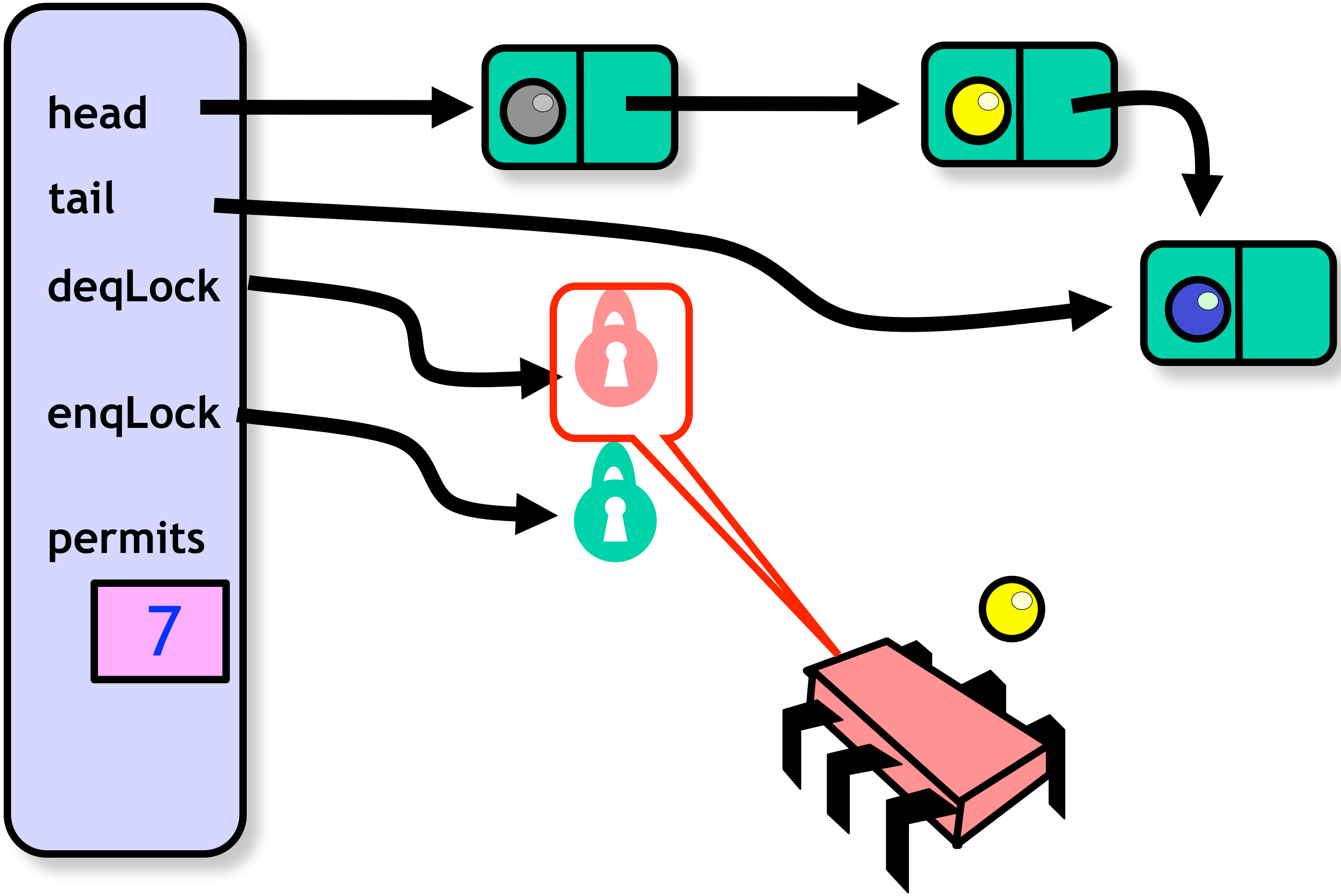
Read value

Dequeuer



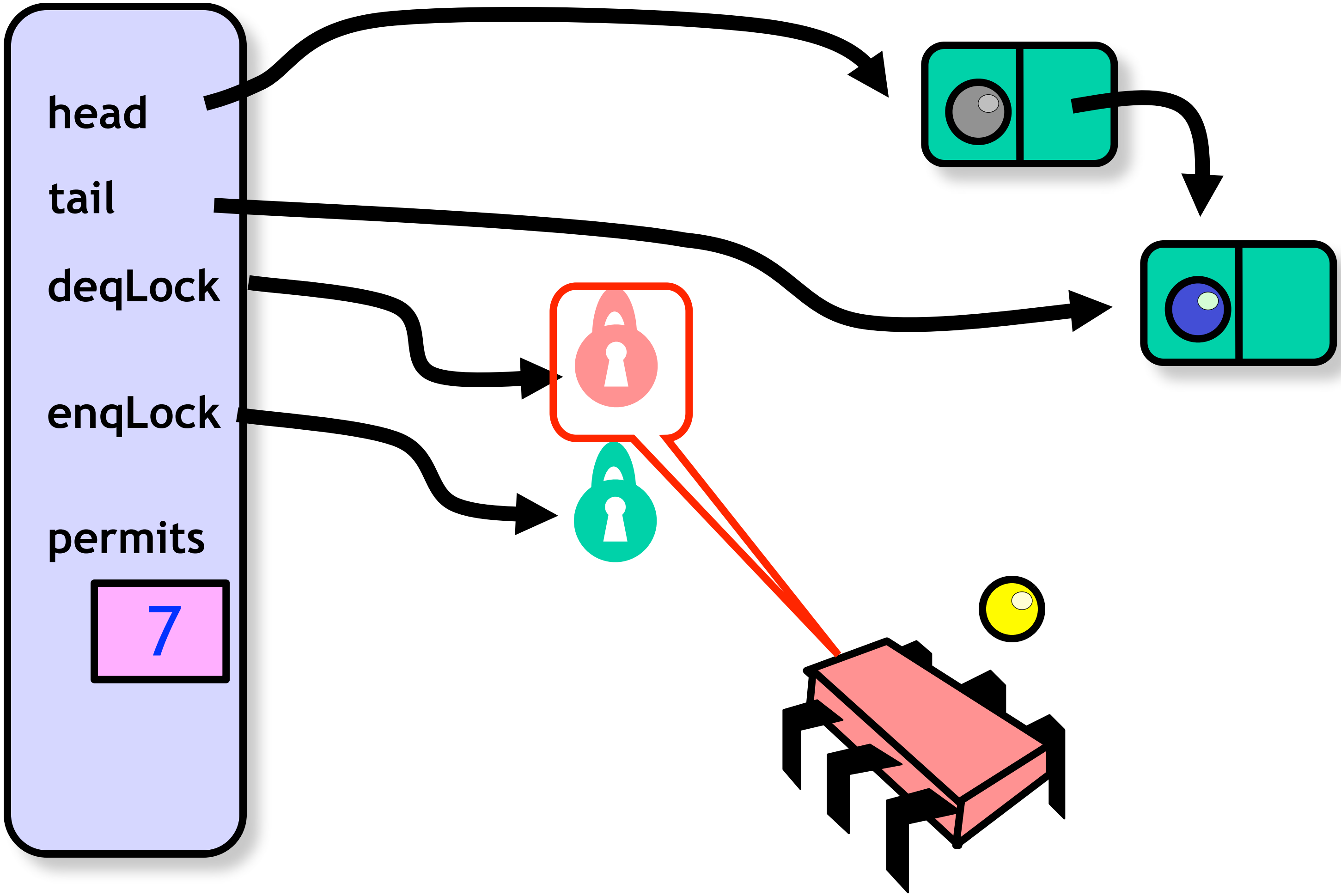
Read value

Dequeuer

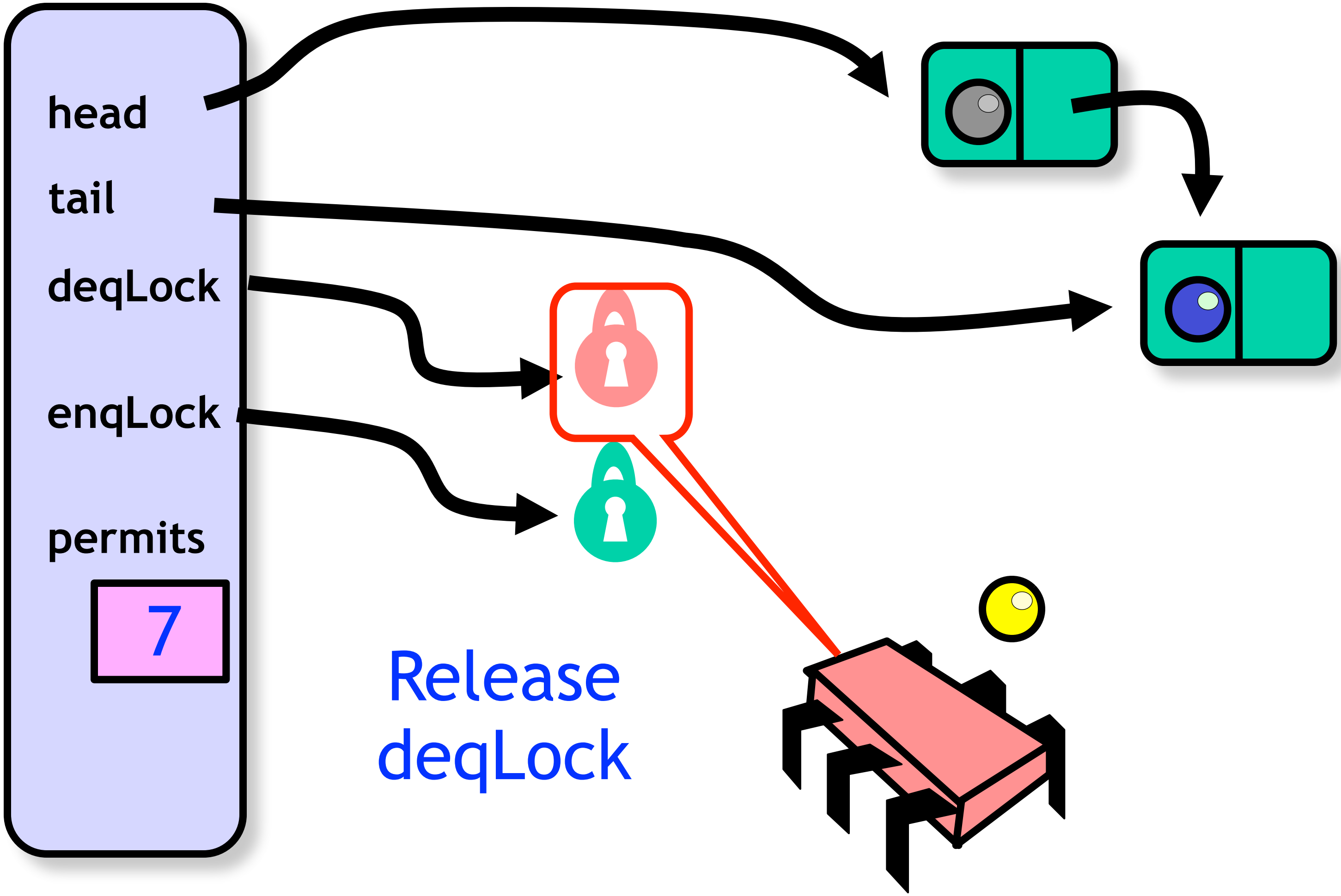


Dequeuer

Make first Node new sentinel

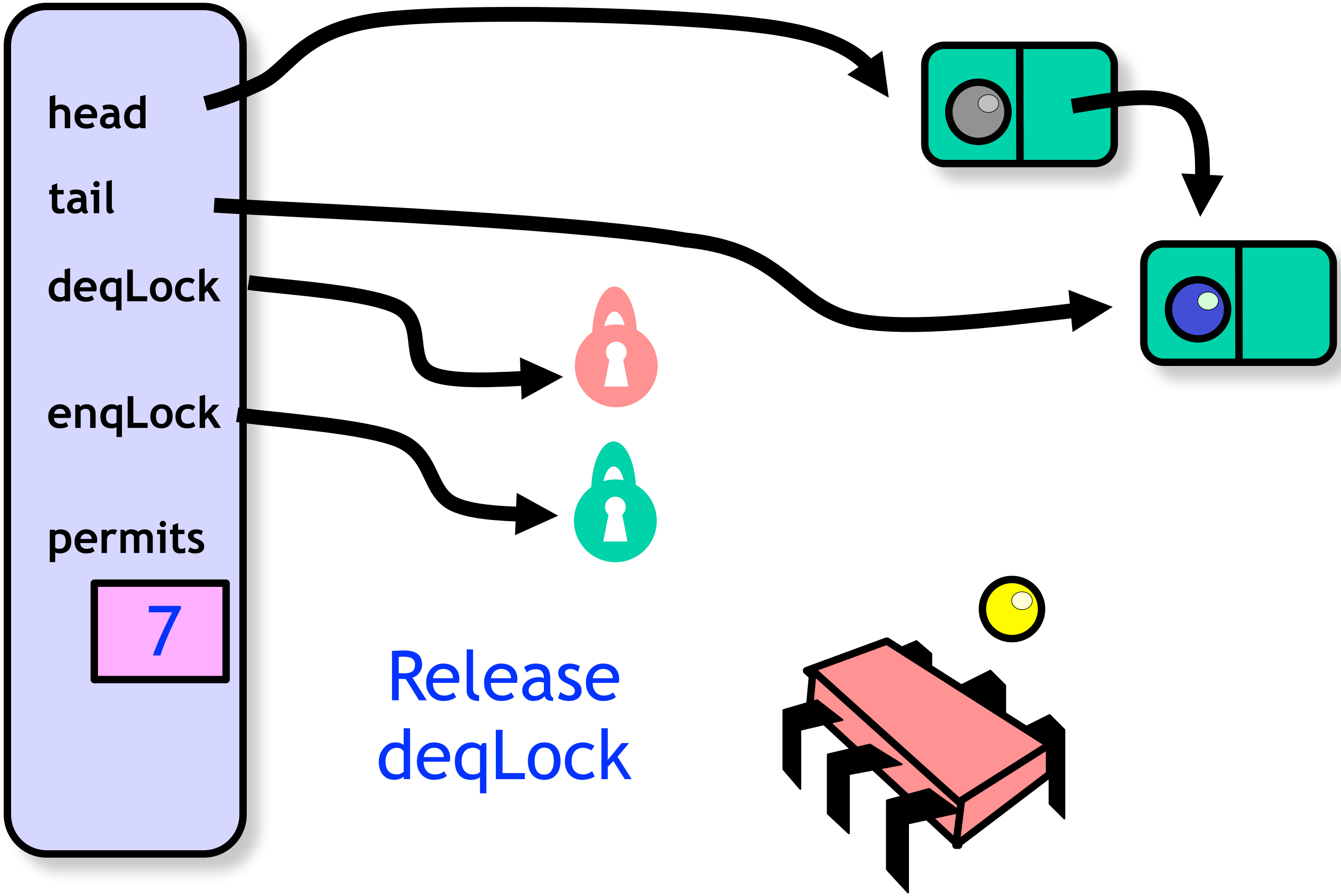


Dequeuer



Release deqLock

Dequeuer



head

tail

deqLock

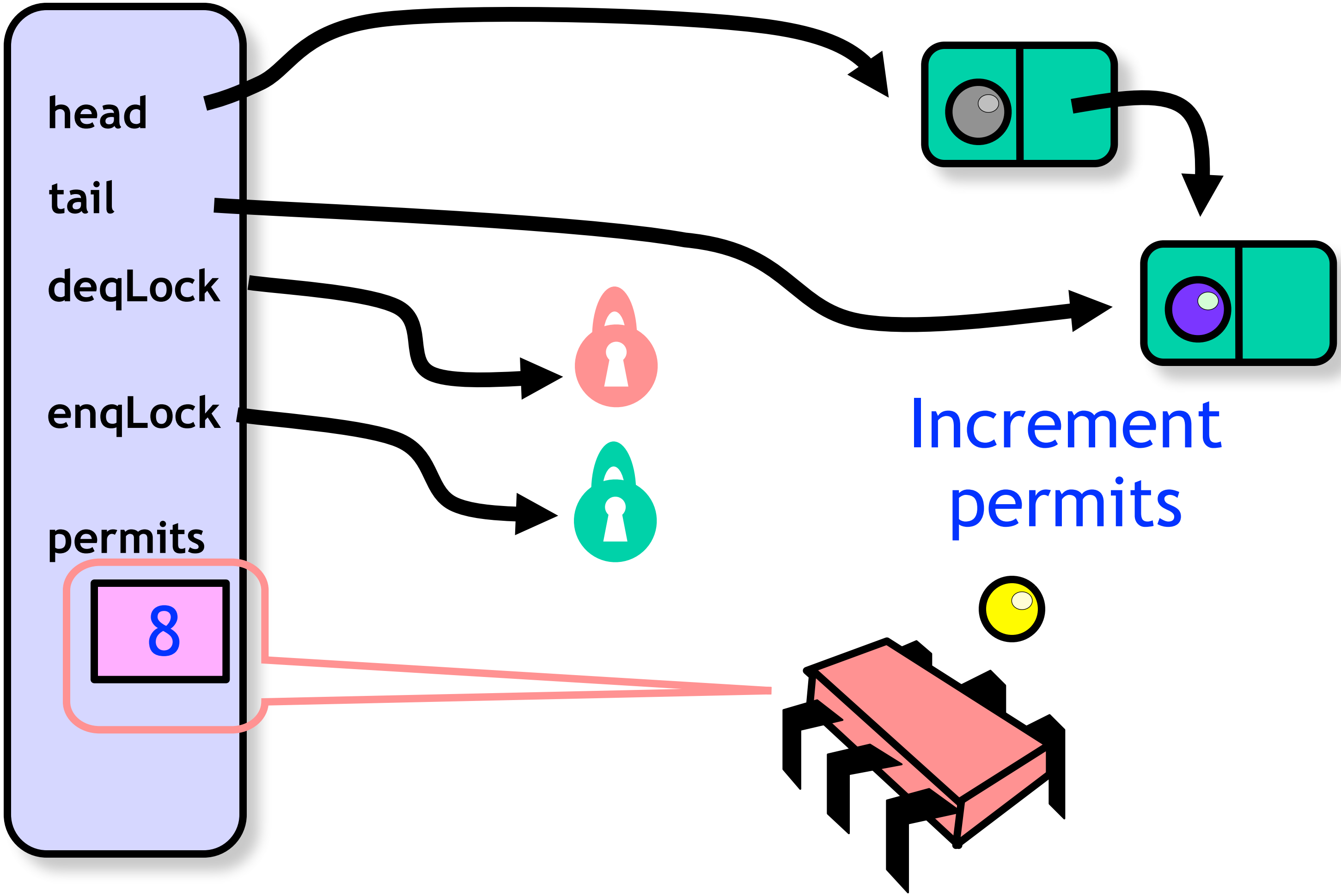
enqLock

permits

7

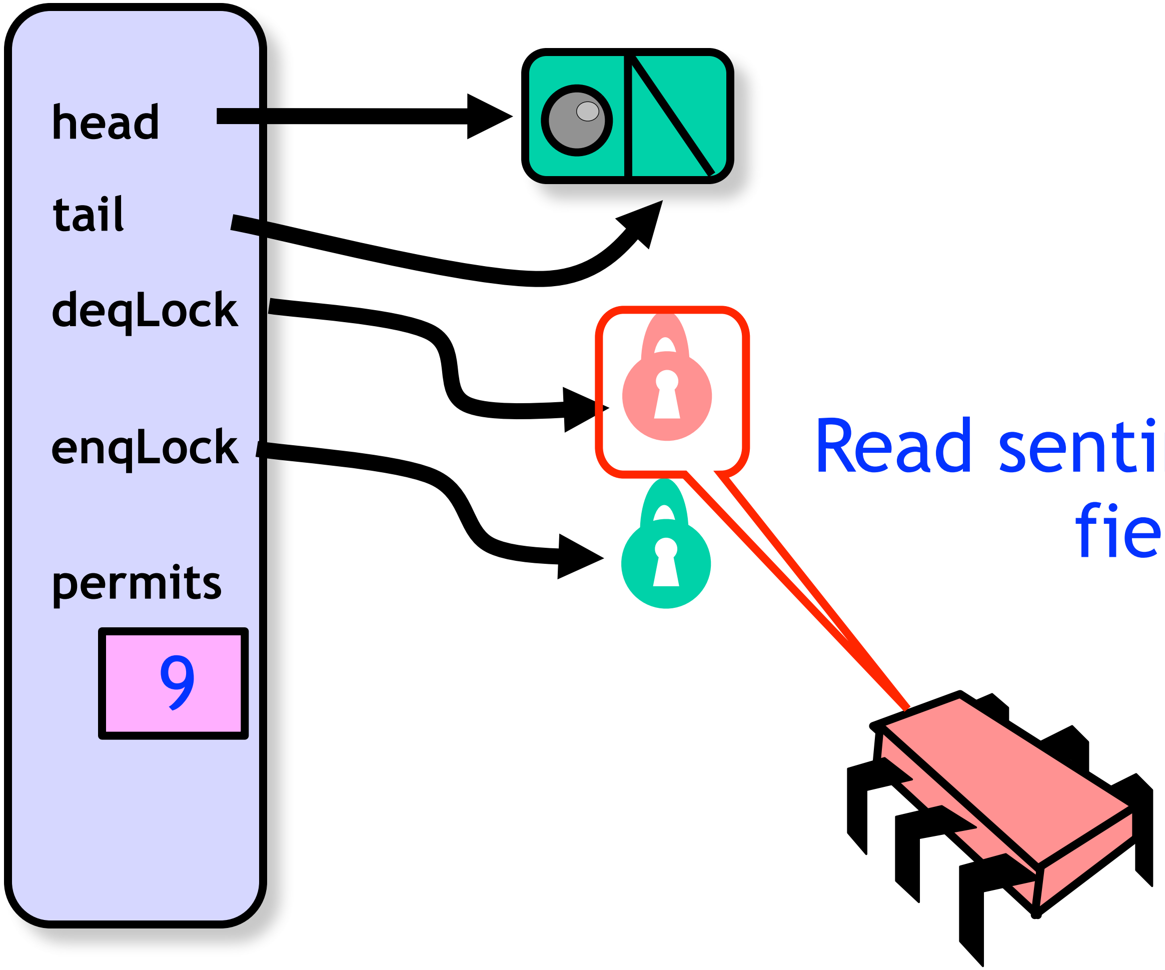
Release deqLock

Dequeuer



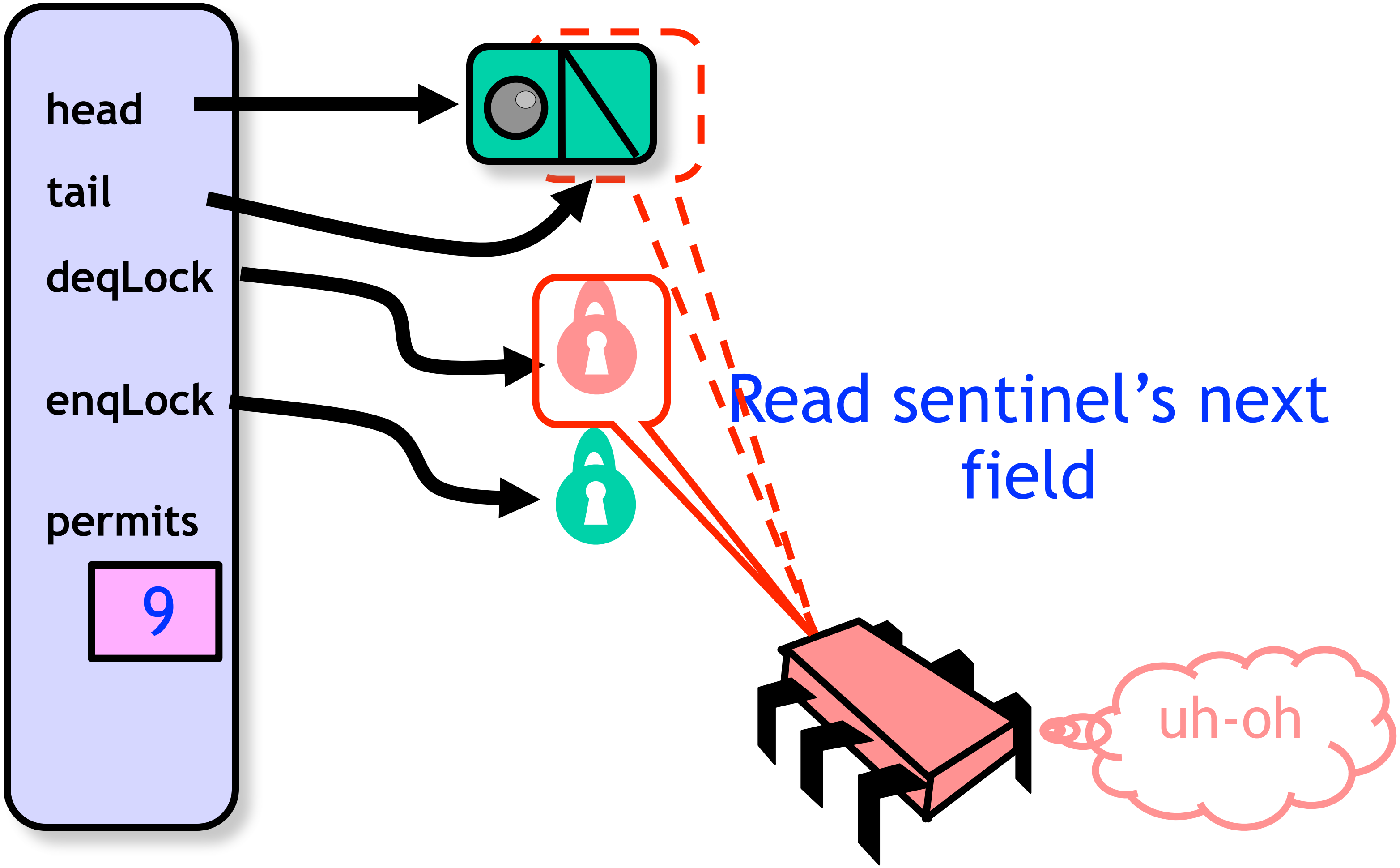
Increment permits

Unsuccesful Dequeuer



Read sentinel's next field

Unsuccesful Dequeuer



Read sentinel’s next field

© Herlihy-Shavit

Bounded Queue

```
public class BoundedQueue<T> {
  ReentrantLock enqLock, deqLock;
  Condition notEmptyCondition, notFullCondition;
  AtomicInteger permits;
  Node head;
  Node tail;
  int capacity;
  enqLock = new ReentrantLock();
  notFullCondition = enqLock.newCondition();
  deqLock = new ReentrantLock();
  notEmptyCondition = deqLock.newCondition();
}
```

Bounded Queue

```
public class BoundedQueue<T> {
  ReentrantLock enqLock, deqLock;
  Condition notEmptyCondition, notFullCondition;
  AtomicInteger permits;
  Node head;
  Node tail;
  int capacity;
  enqLock = new ReentrantLock();
  notFullCondition = enqLock.newCondition();
  deqLock = new ReentrantLock();
  notEmptyCondition = deqLock.newCondition();
}
```

Enq & deq locks

Monitor Locks

- The Reentrant Lock is a monitor
- Allows blocking on a condition rather than spinning
- Threads:
  - acquire and release lock
  - wait on a condition

Java Monitor Locks

```
public interface Lock {
 void lock();
 void lockInterruptibly() throw InterruptedException;
 boolean tryLock();
 boolean tryLock(long time, TimeUnit unit);
 Condition newCondition();
 void unlock();
}
```

Java Locks

```
public interface Lock {
  void lock();
  void lockInterruptibly() throws InterruptedException;
  boolean tryLock();
  boolean tryLock(long time, TimeUnit unit);
  Condition newCondition();
  void unlock();
}
```

Acquire lock

Java Locks

```
public interface Lock {
  void lock();
  void lockInterruptibly() throws InterruptedException;
  boolean tryLock();
  boolean tryLock(long time, TimeUnit unit);
  Condition newCondition();
  void unlock();
}
```

Release lock

Java Locks

```
public interface Lock {
  void lock();
  void lockInterruptibly() throws InterruptedException;
  boolean tryLock();
  boolean tryLock(long time, TimeUnit unit);
  Condition newCondition();
  void unlock();
}
```

Conditions to wait on

Lock Conditions

```
public interface Condition {
  void await()
    throws InterruptedException;
  boolean await(long time, TimeUnit unit)
    throws InterruptedException;
  …
  void signal();
  void signalAll();
 }
```

Lock Conditions

```
public interface Condition {
  void await()
    throws InterruptedException;
  boolean await(long time, TimeUnit unit)
    throws InterruptedException;
  …
  void signal();
  void signalAll();
}
```

Release lock and wait on condition

Lock Conditions

```
public interface Condition {
  void await()
    throws InterruptedException;
  boolean await(long time, TimeUnit unit)
    throws InterruptedException;
  …
  void signal();
  void signalAll();
 }
```

Signal release of next thread in line or all awaiting threads

The await() Method

```
q.await()
```

- Releases lock on q
- Sleeps (gives up processor)
- Awakens (resumes running)
- Reacquires lock & returns

The signal() Method

```
q.signal();
```

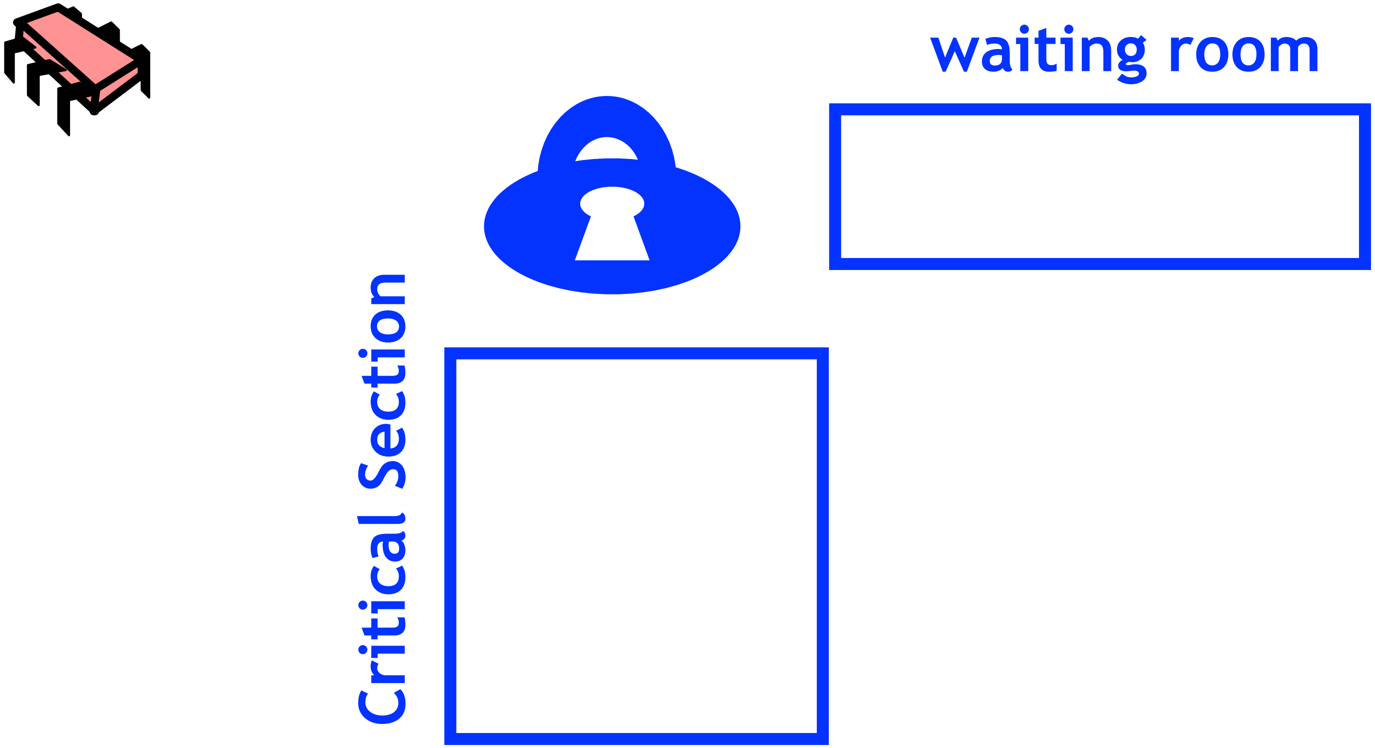
- Awakens one waiting thread
- Which will reacquire lock
- Then returns

The signalAll() Method

```
q.signalAll();
```

- Awakens all waiting threads
- Which will reacquire lock
- Then returns

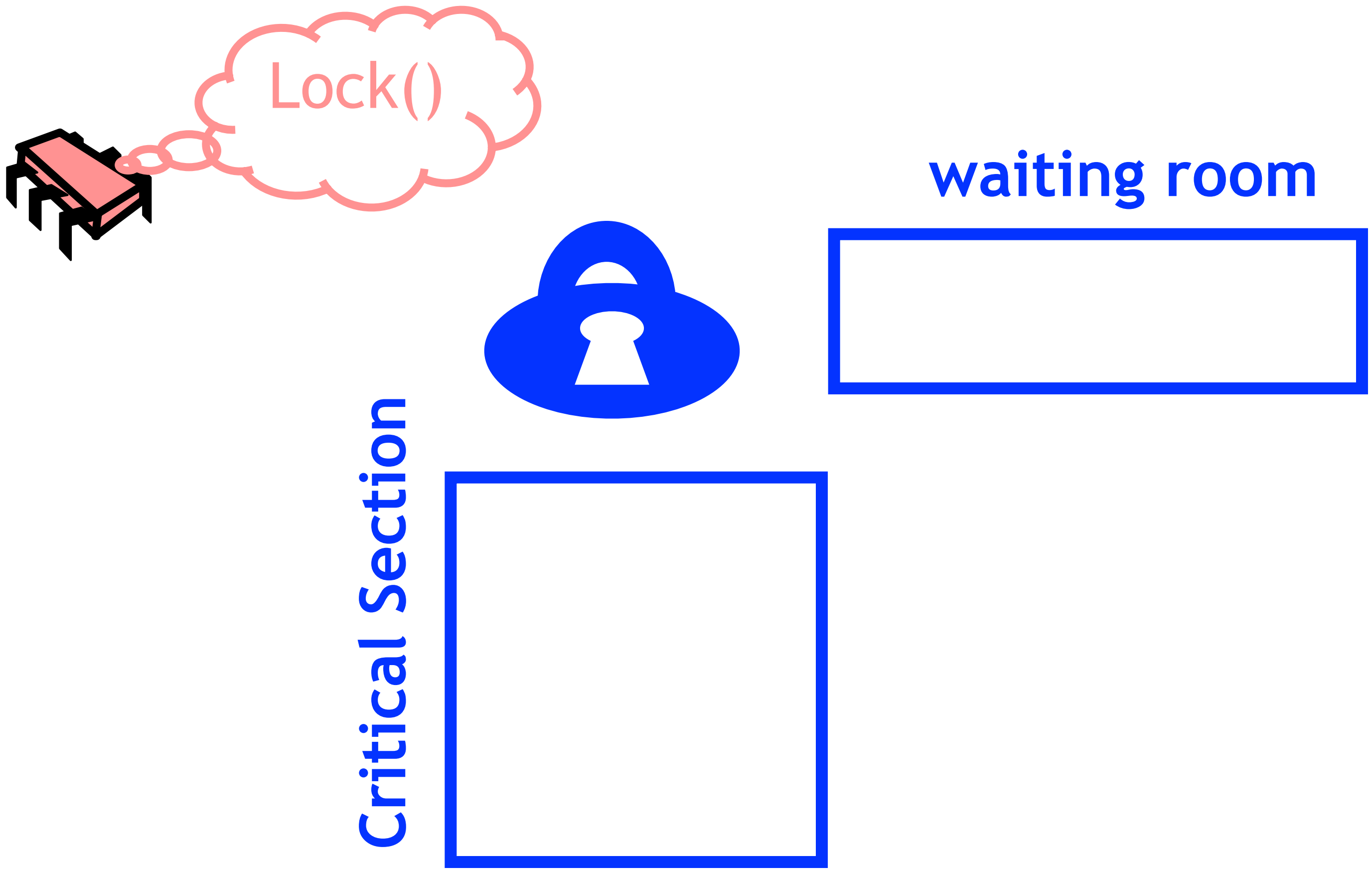
A Monitor Lock



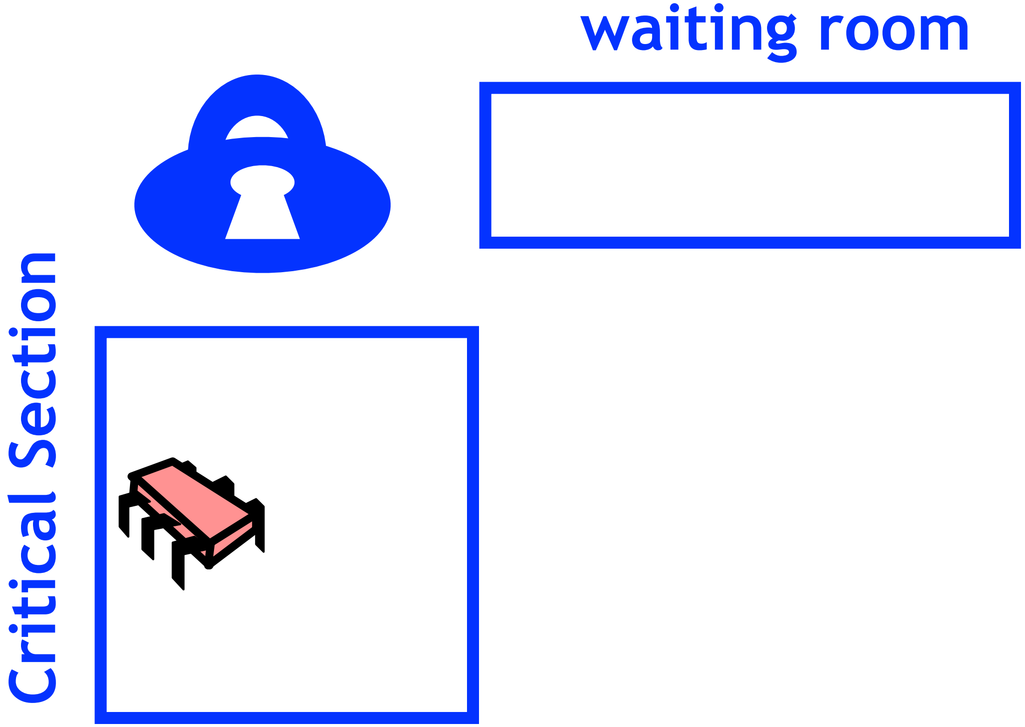
waiting room

Critical Section

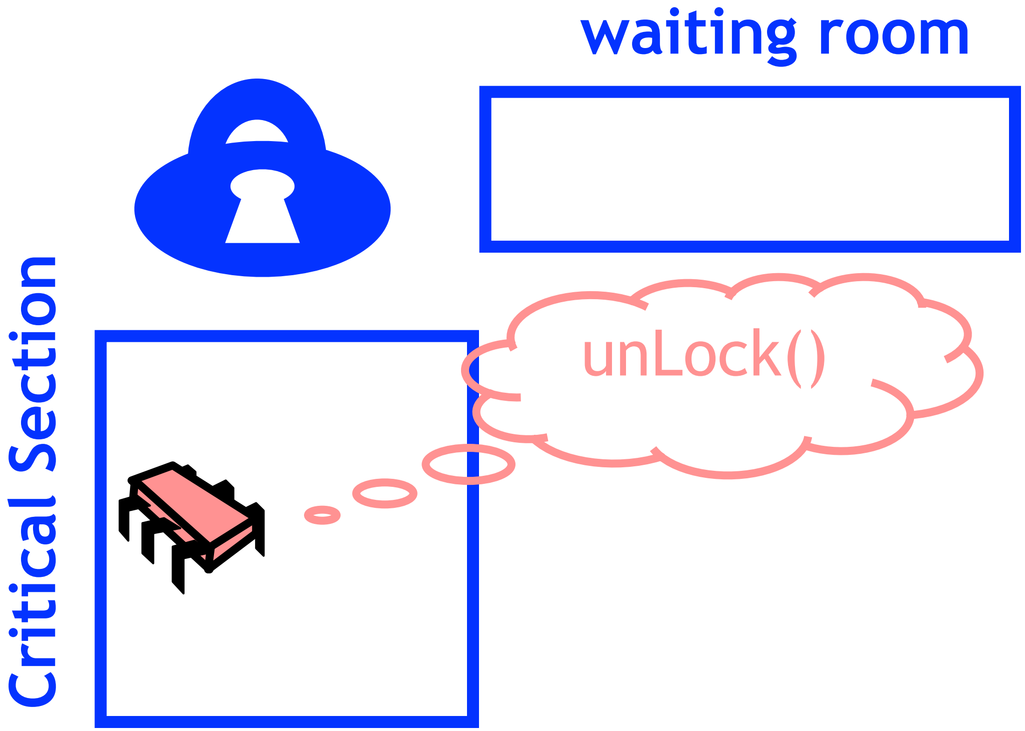
A Monitor Lock



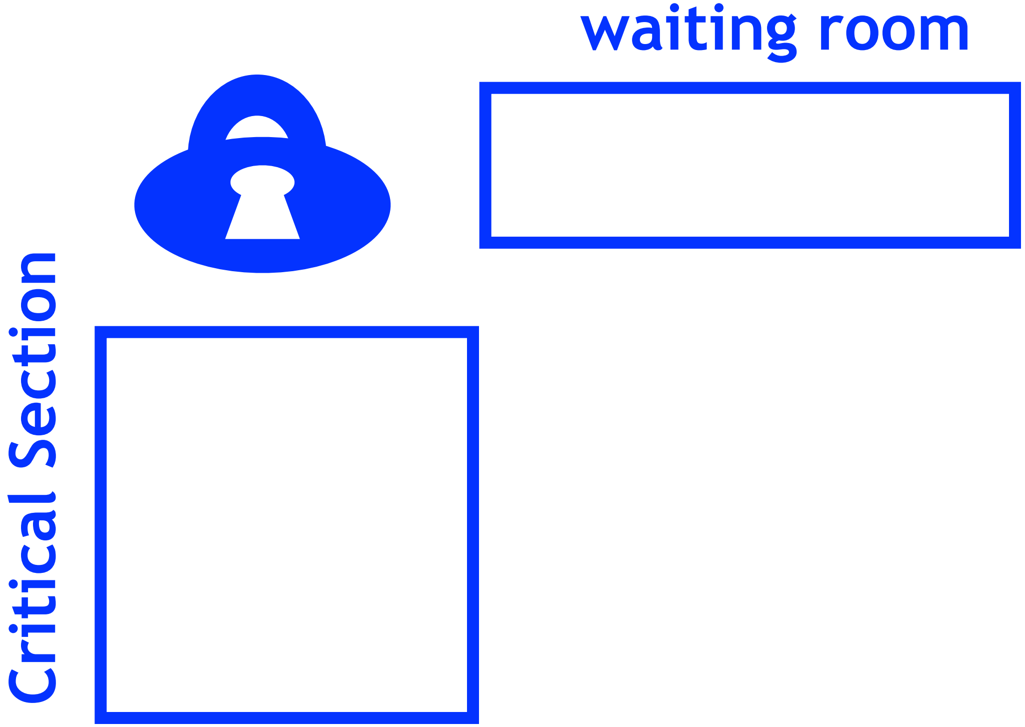
A Monitor Lock



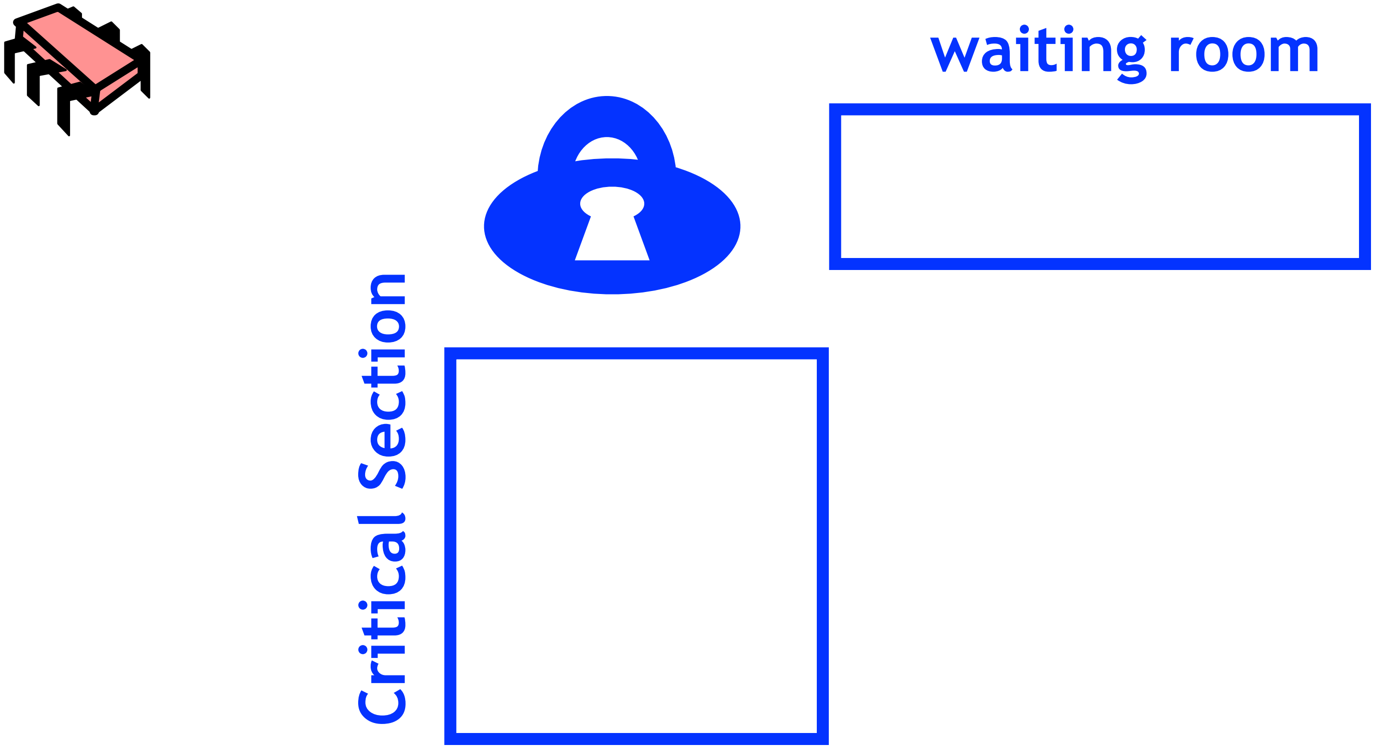
A Monitor Lock



A Monitor Lock



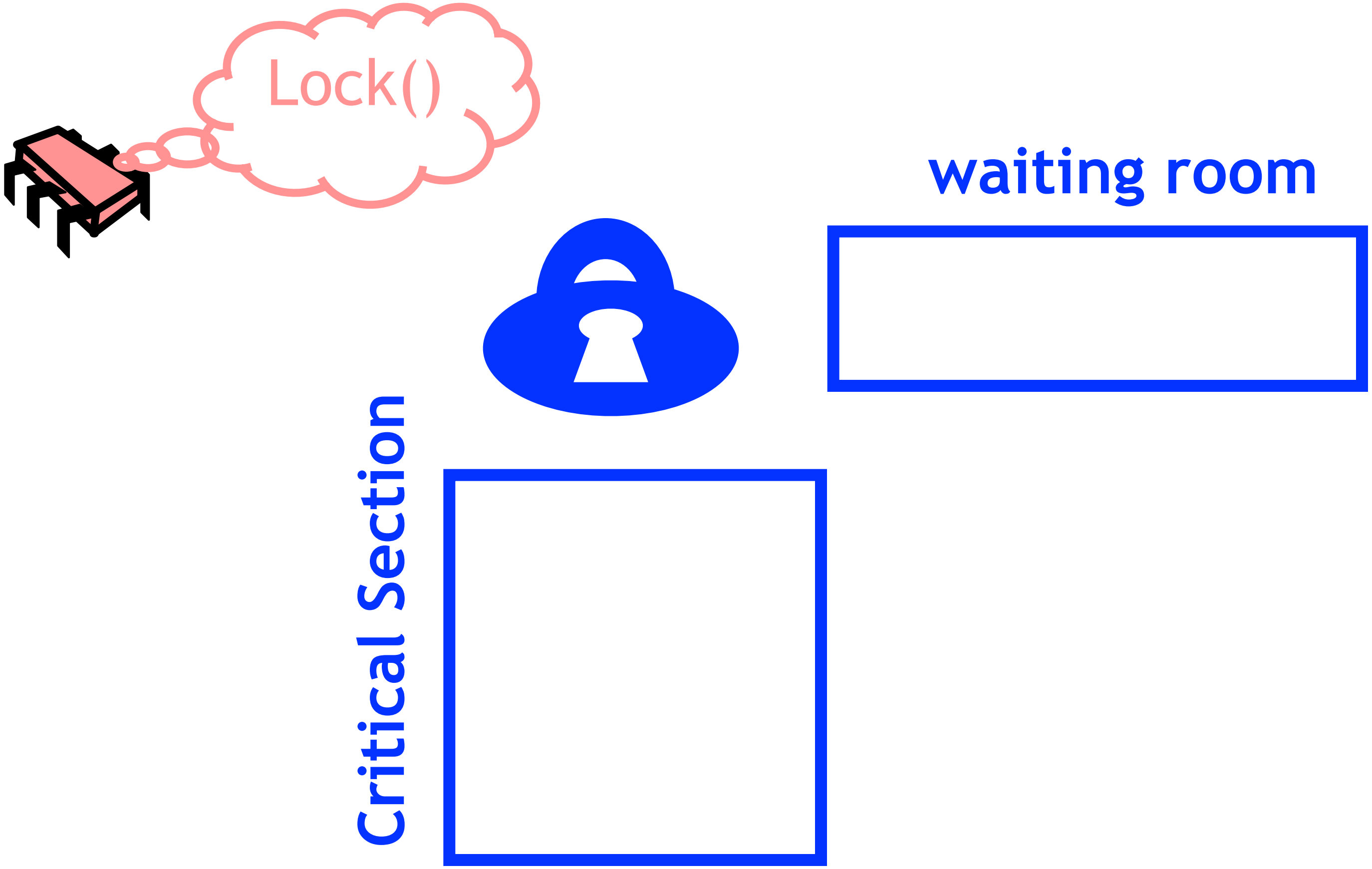
Awaiting a Condition



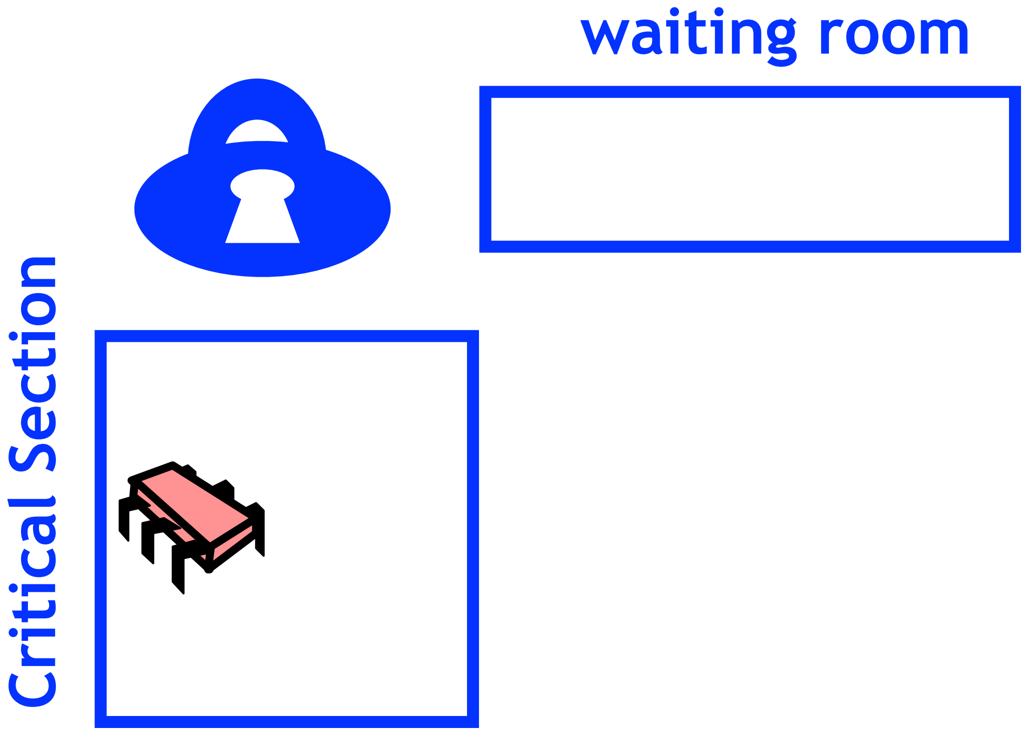
waiting room

Critical Section

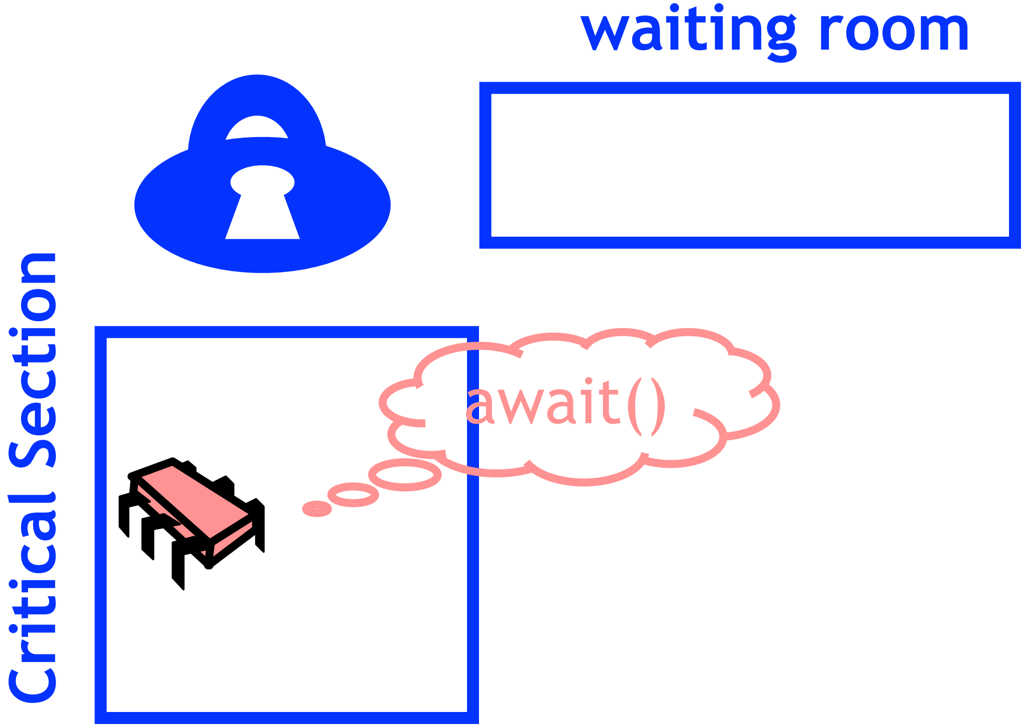
Awaiting a Condition



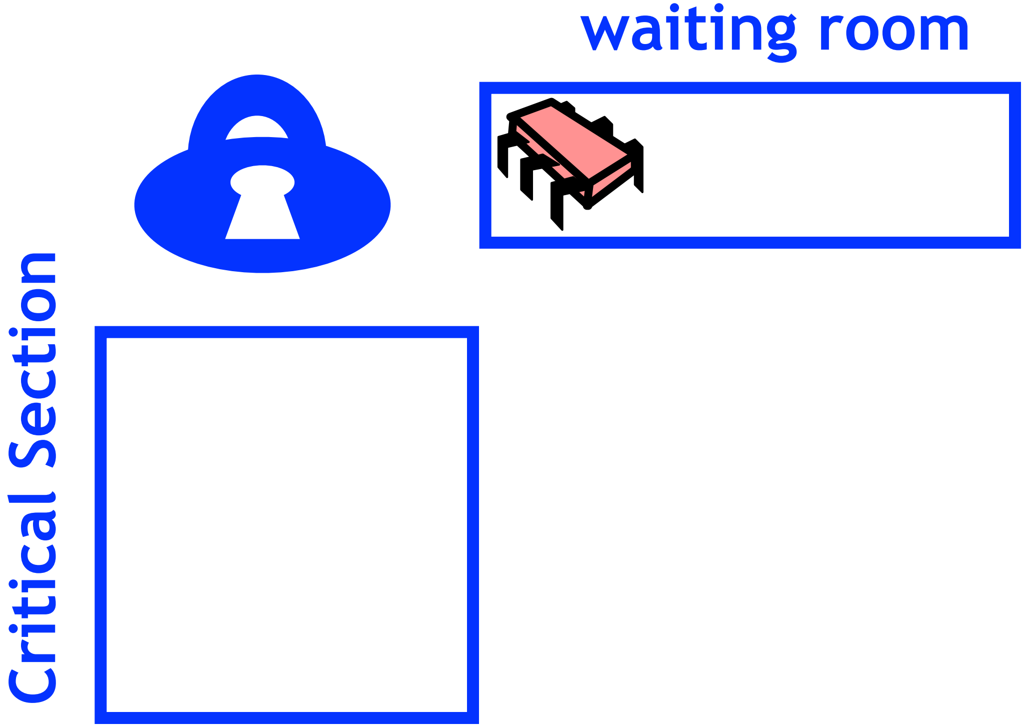
Awaiting a Condition



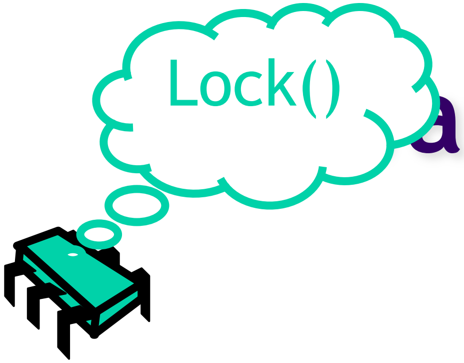
Awaiting a Condition

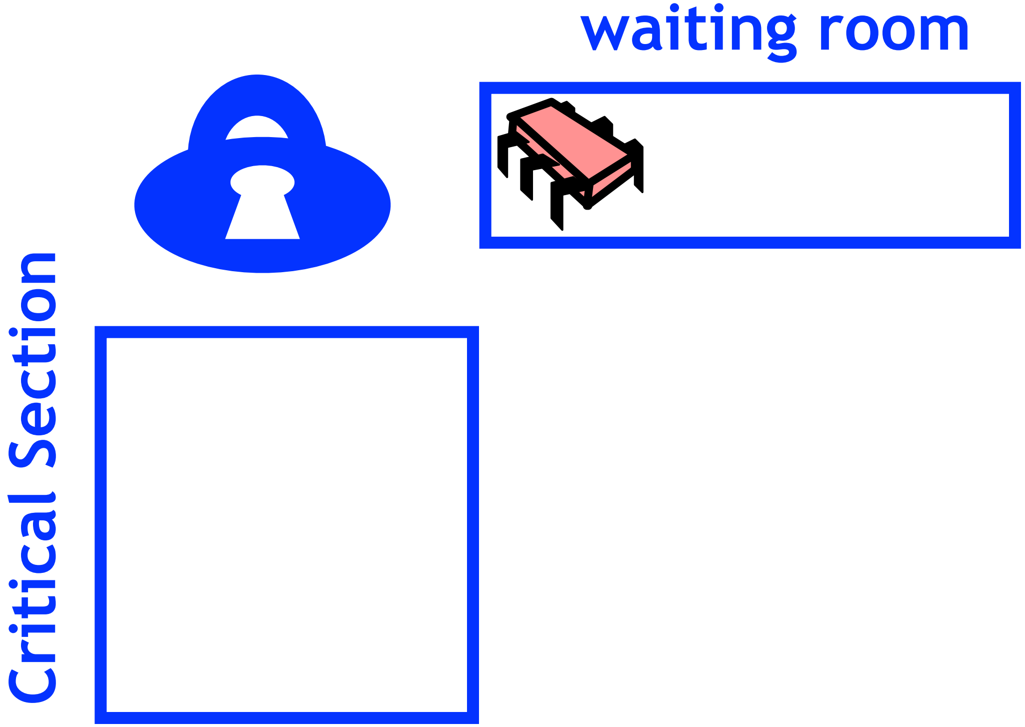


Awaiting a Condition

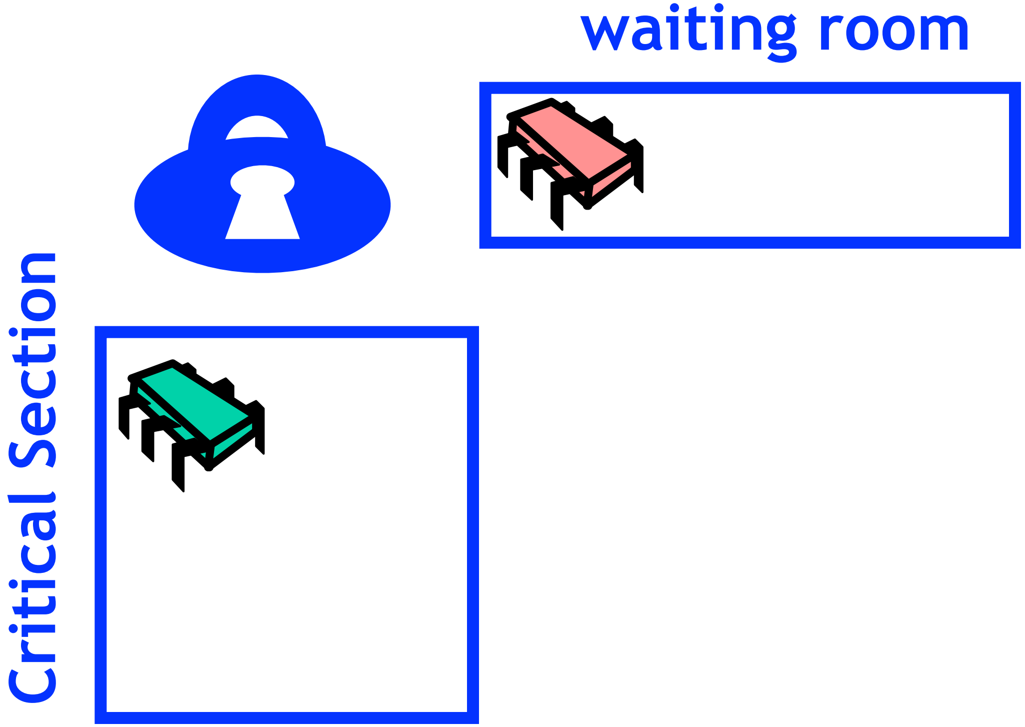


aiting a Condition





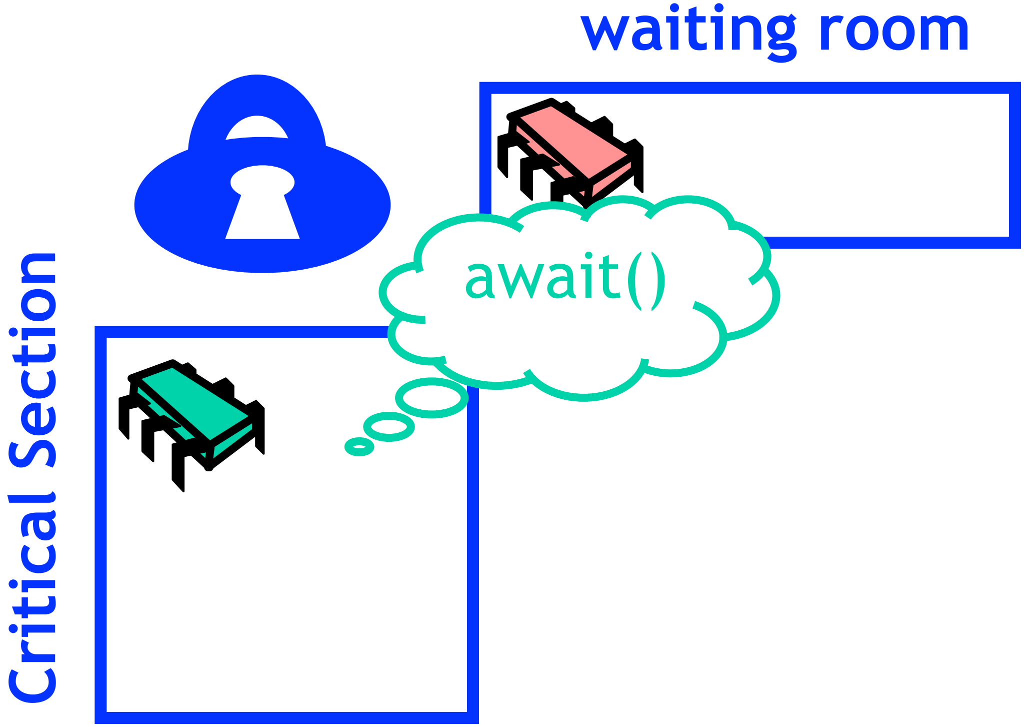
Awaiting a Condition



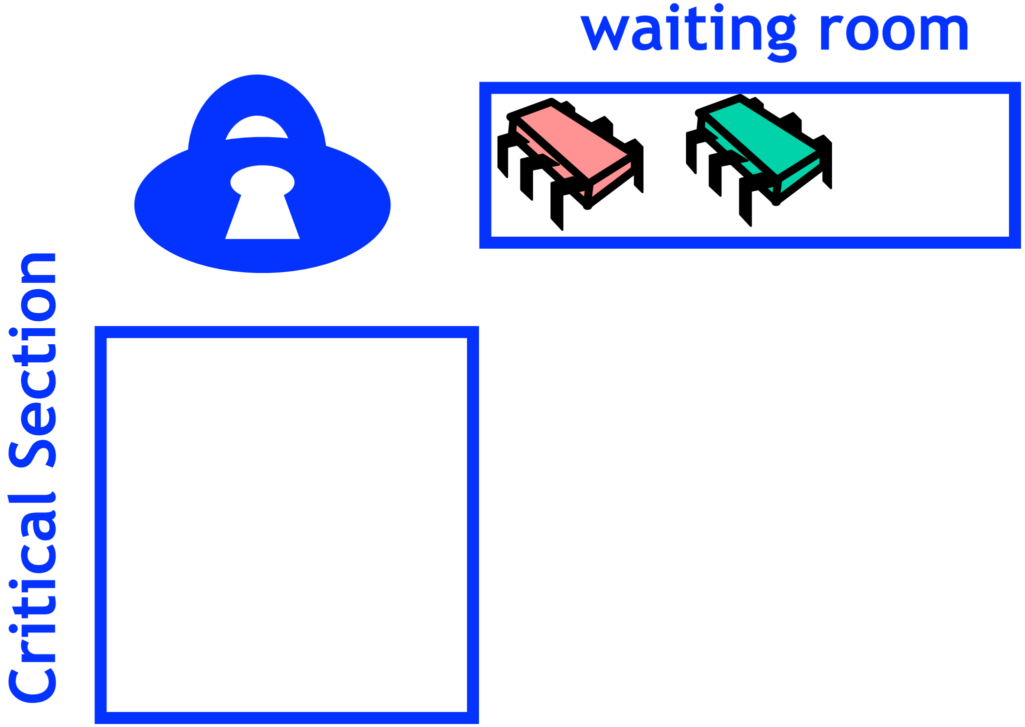
waiting room

Critical Section

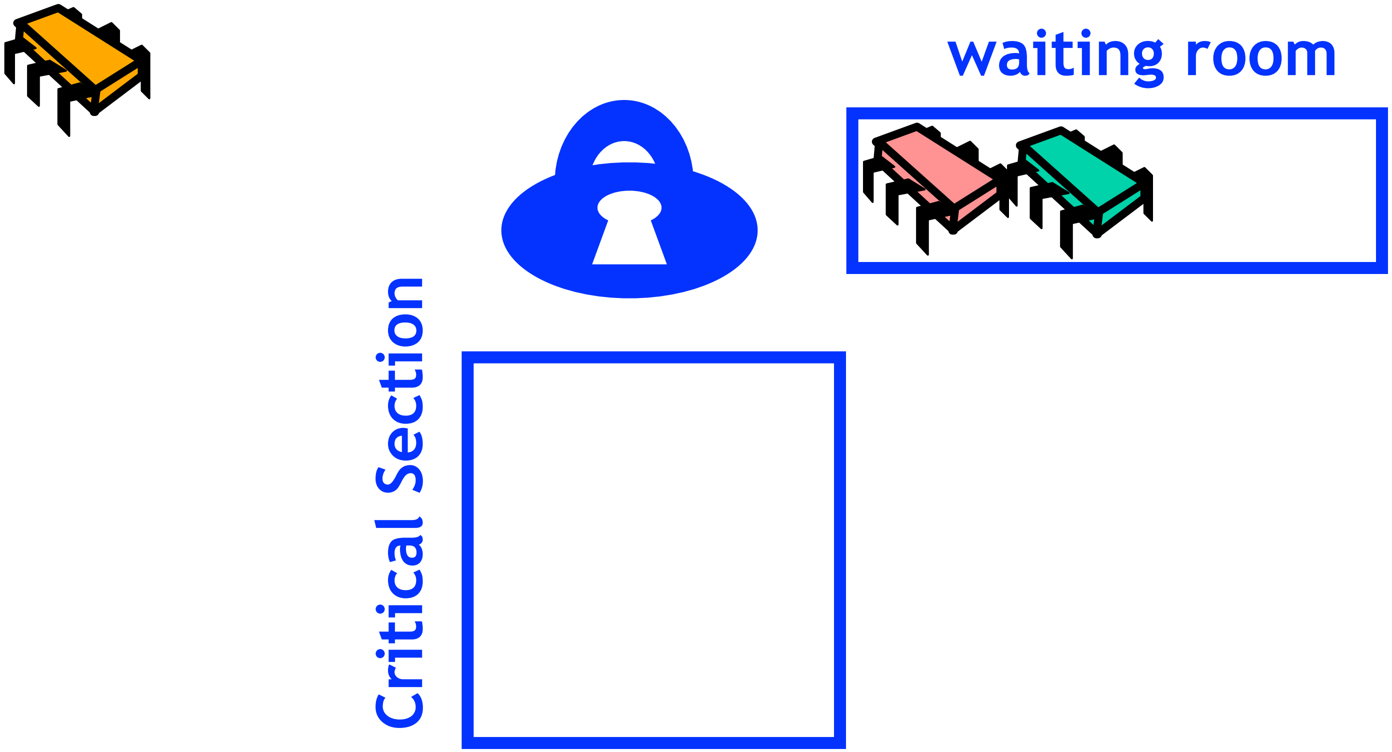
Awaiting a Condition



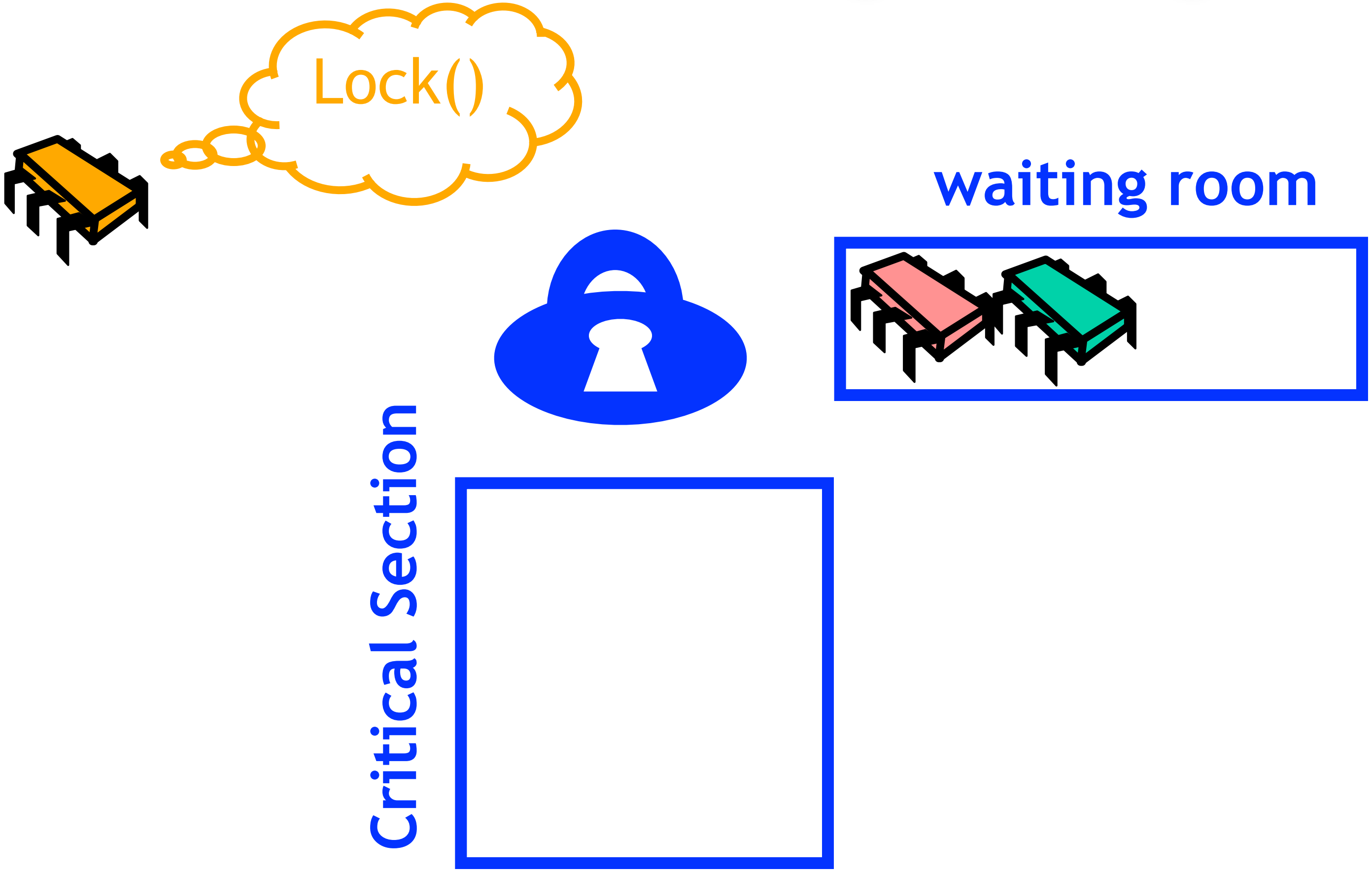
Awaiting a Condition



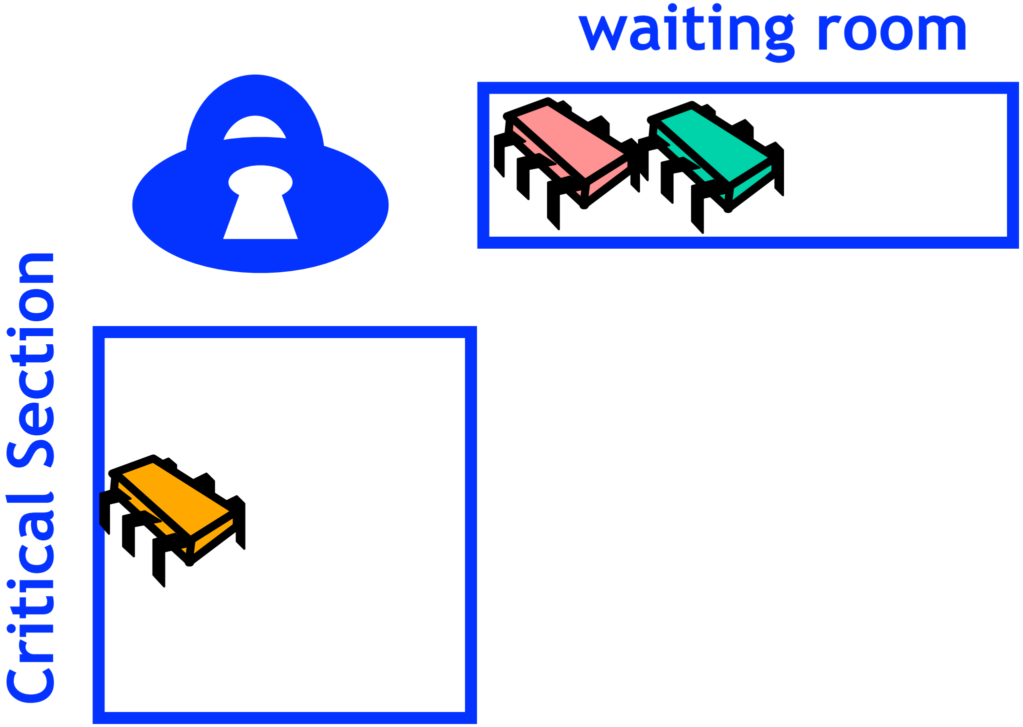
Monitor Signalling



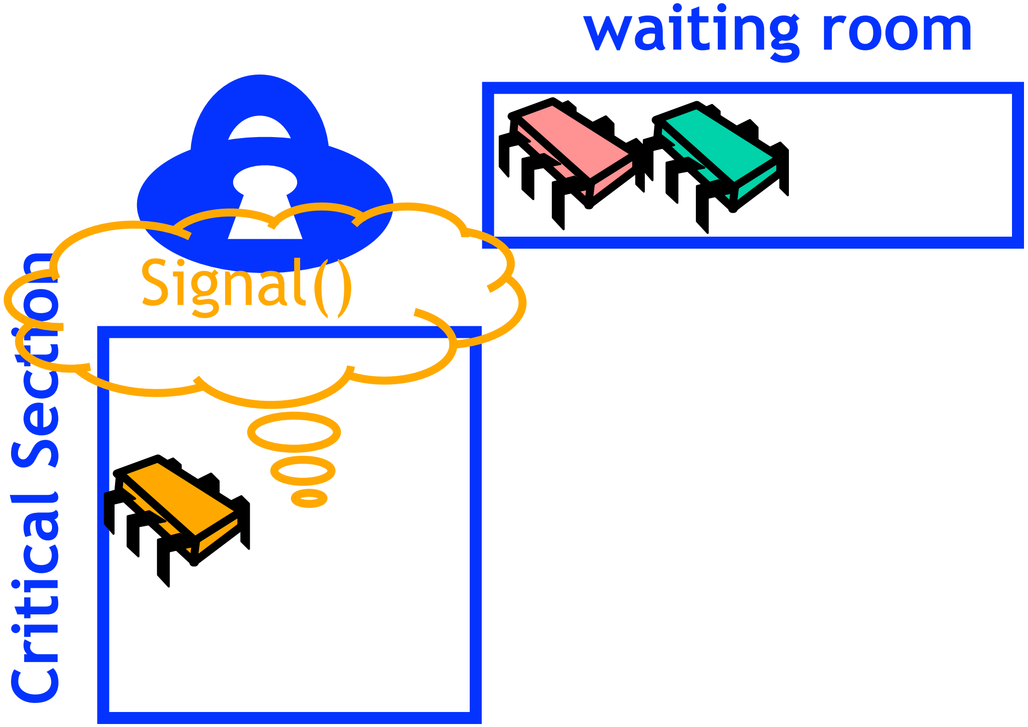
Monitor Signalling



Monitor Signalling

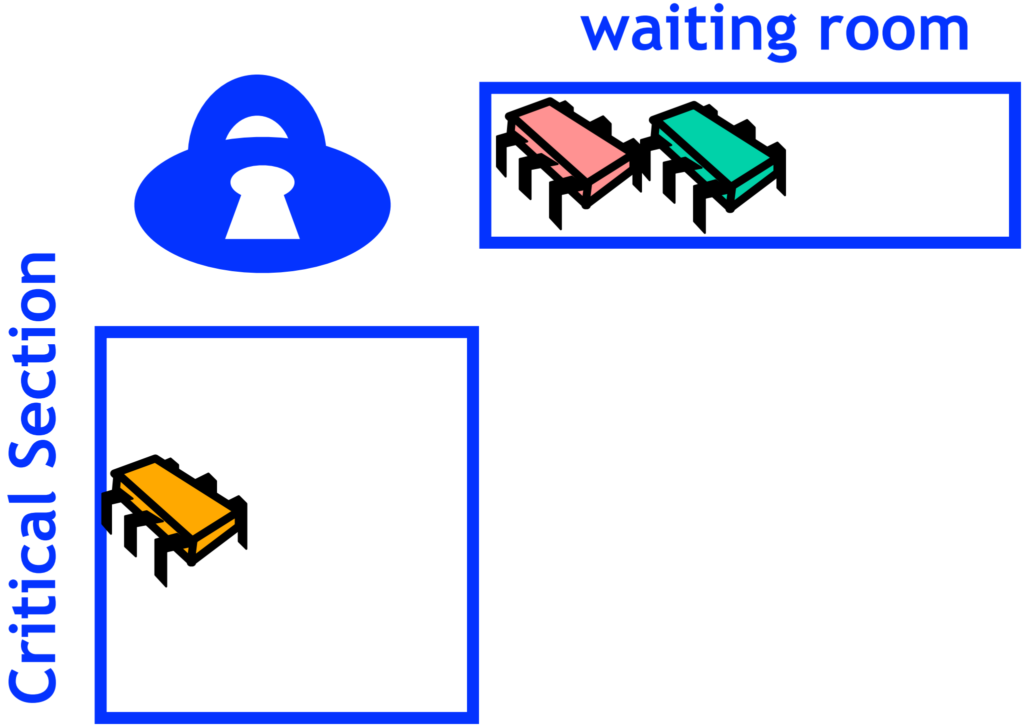


Monitor Signalling

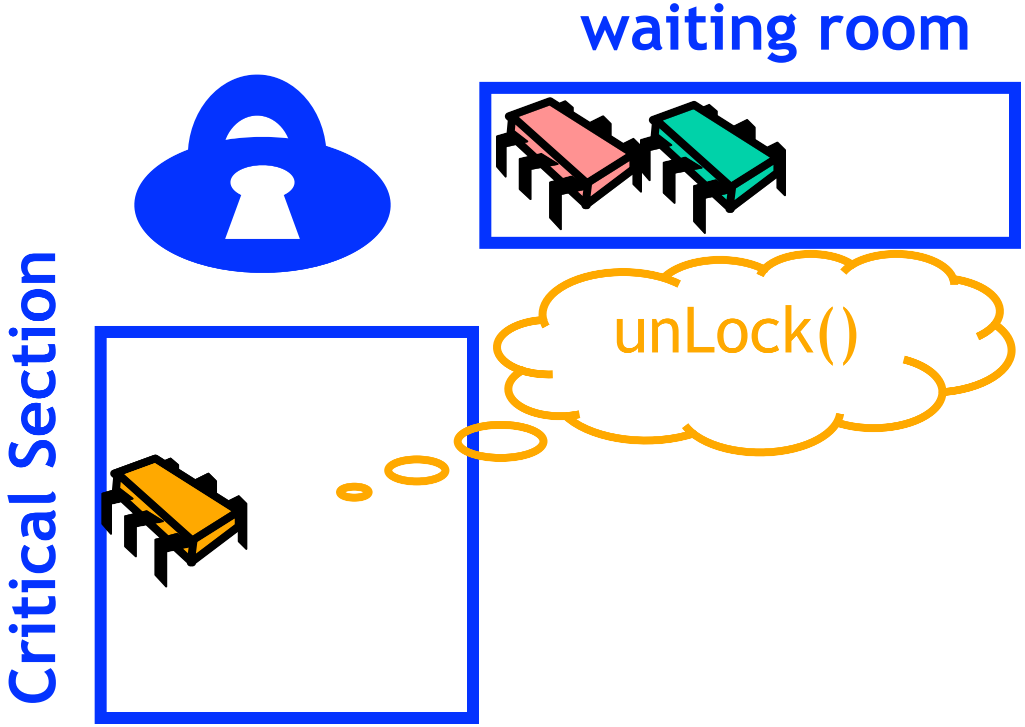


© Herlihy-Shavit

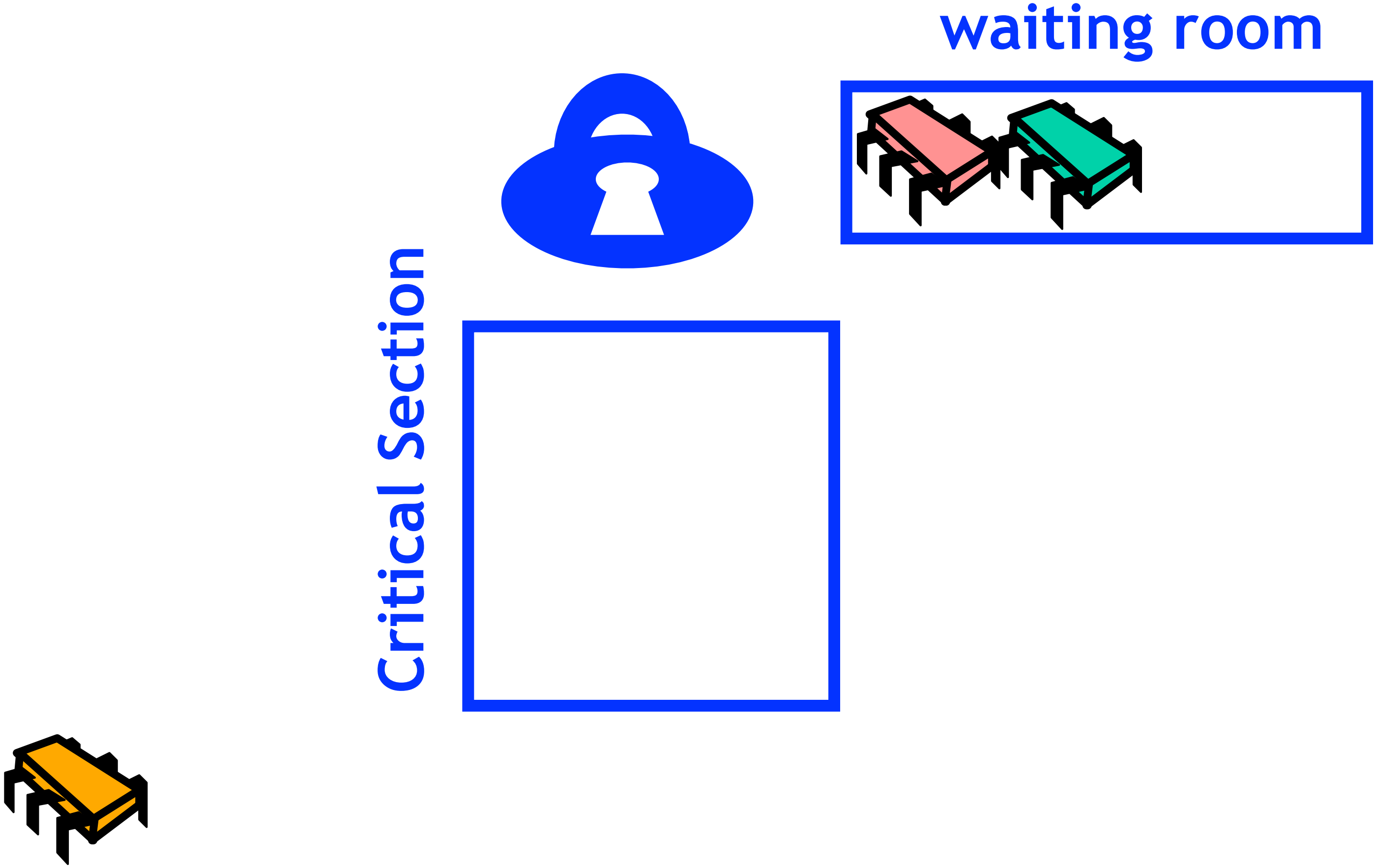
Monitor Signalling



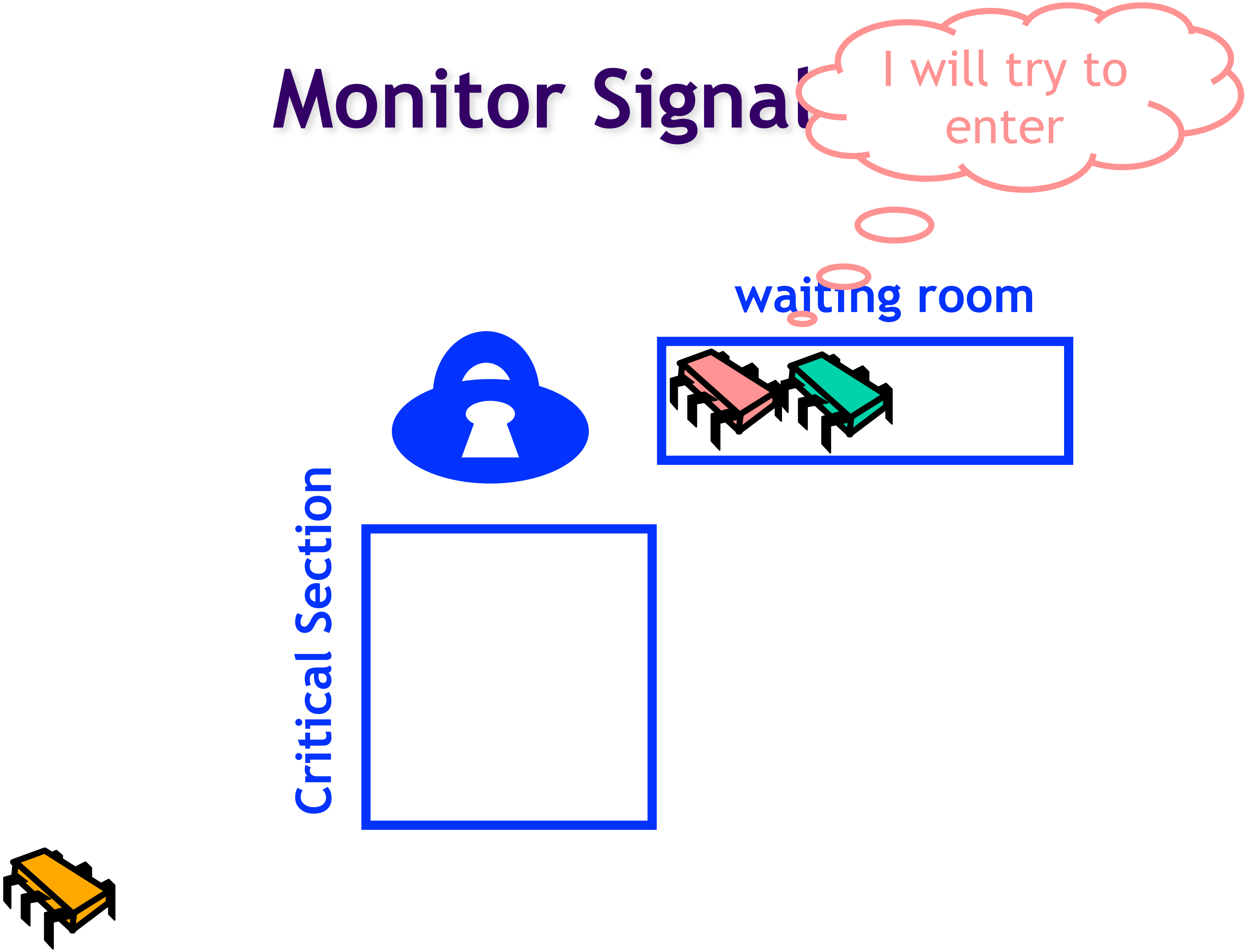
Monitor Signalling



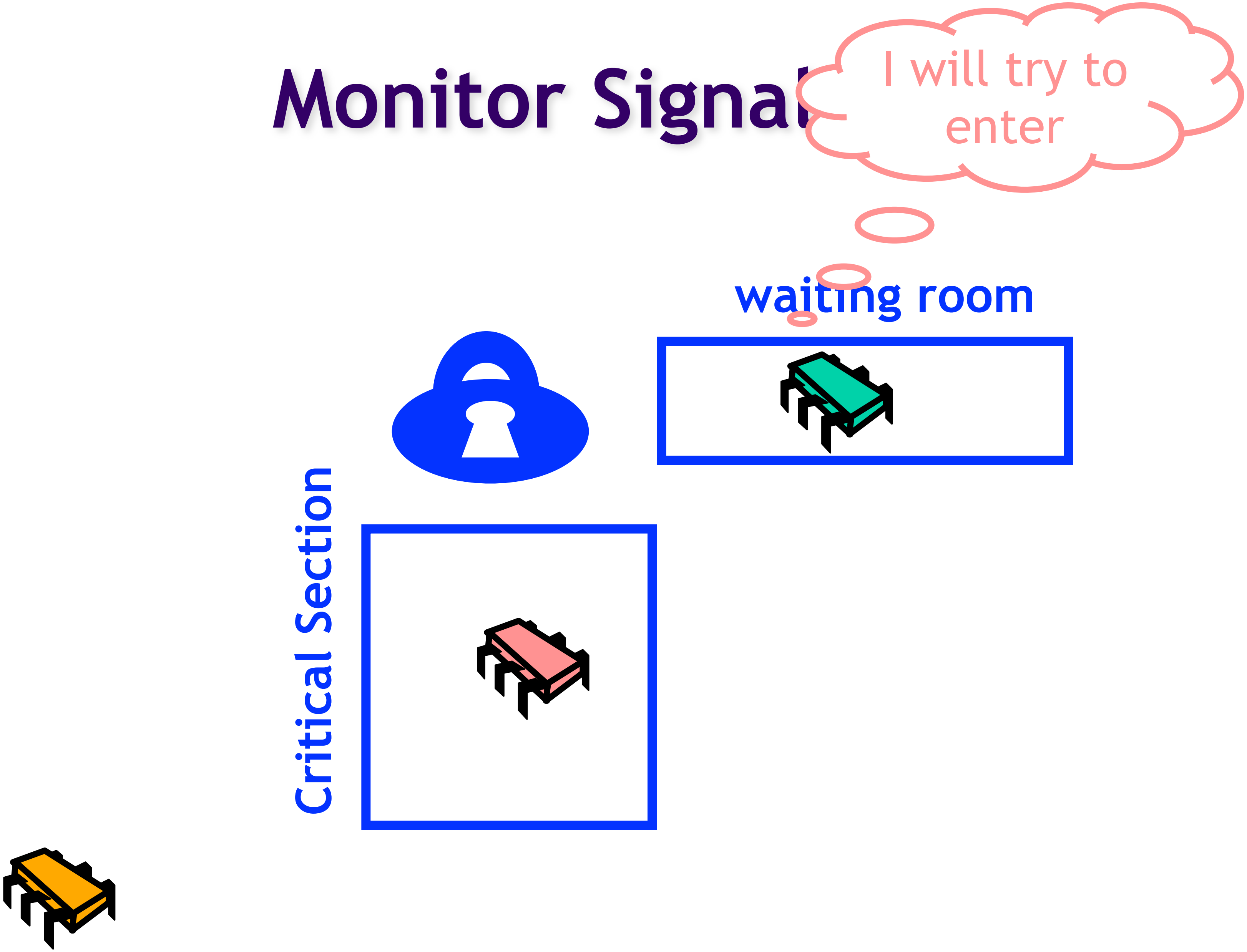
Monitor Signalling



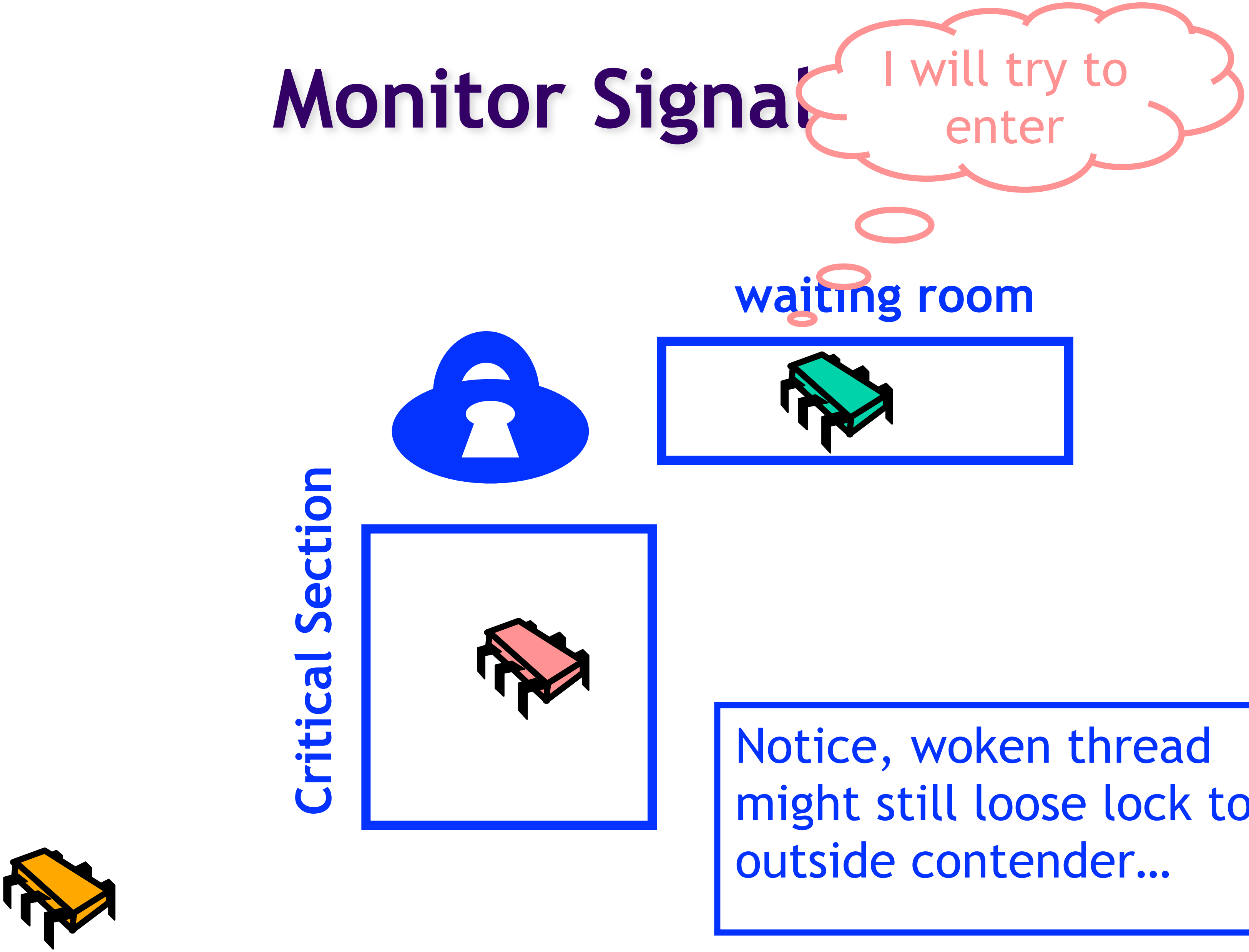
Monitor Signal



Monitor Signal

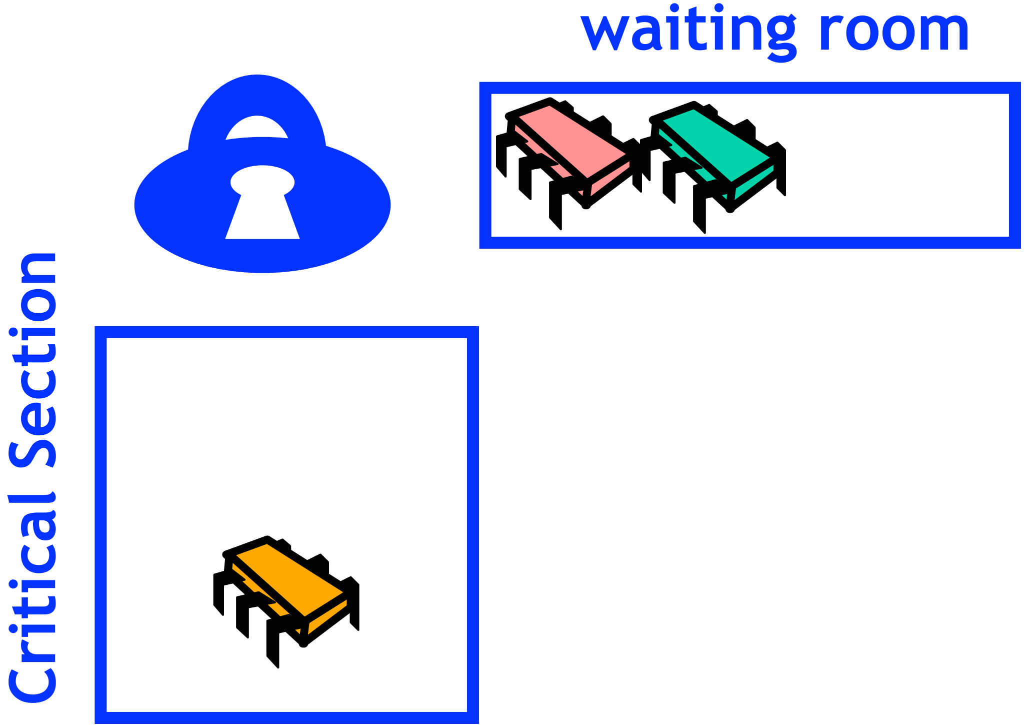


Monitor Signal

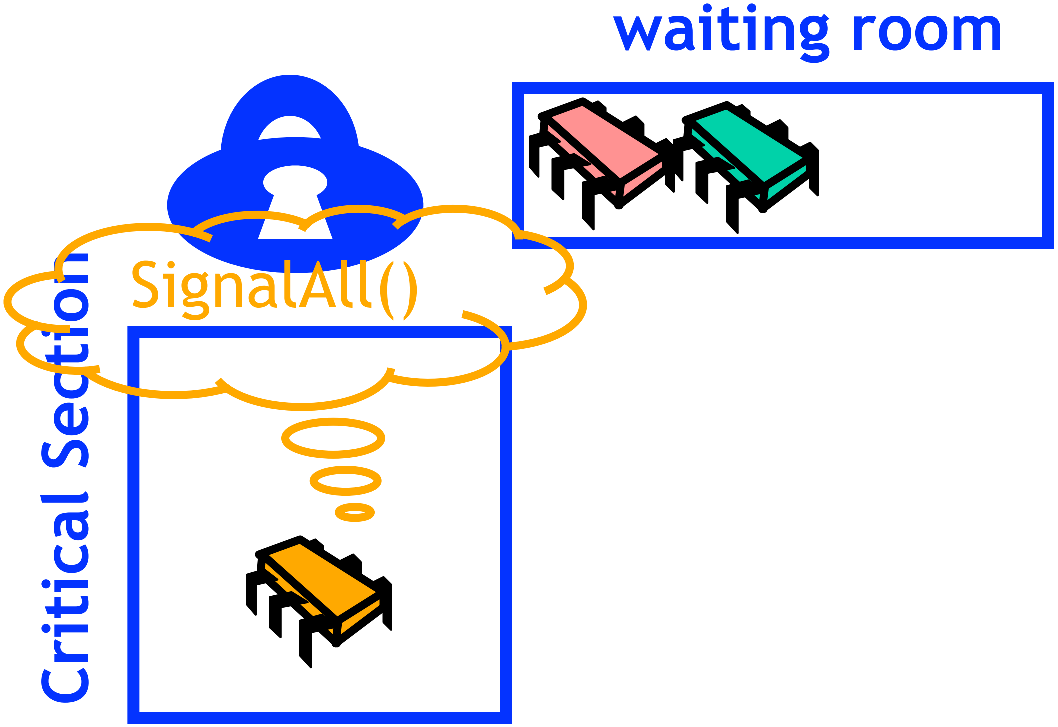


Notice, woken thread might still loose lock to outside contender…

Monitor Signaling All

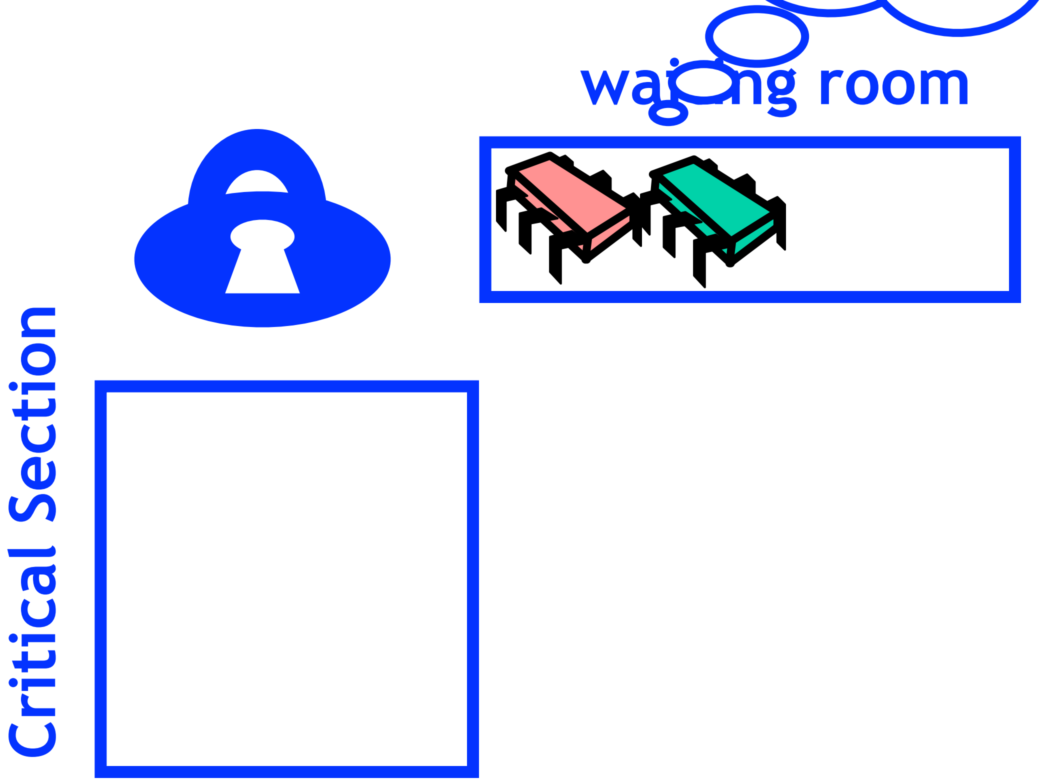


Monitor Signaling All



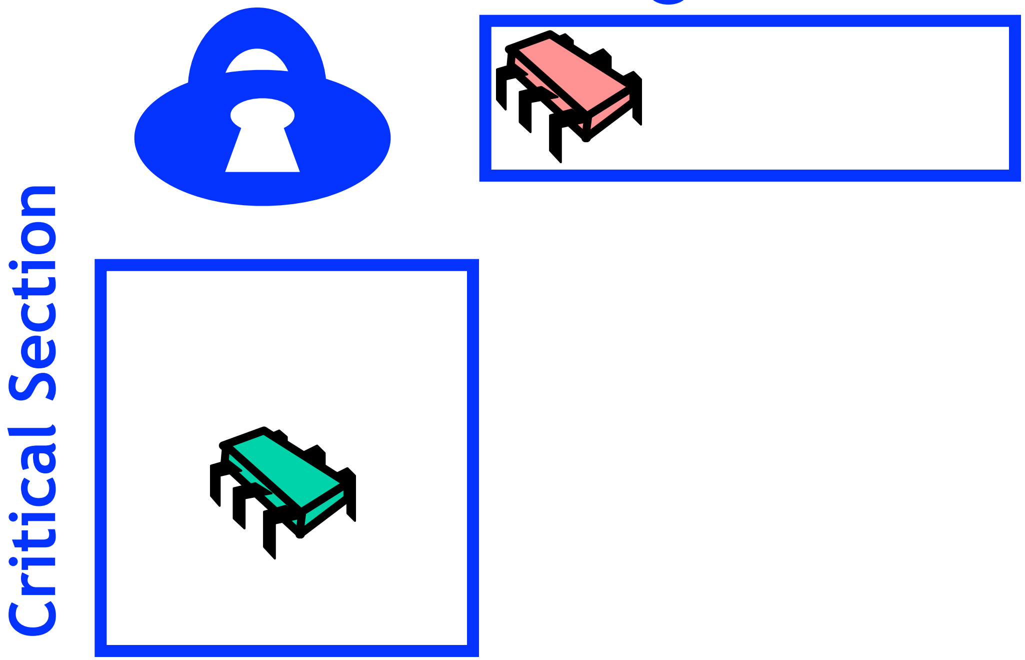
Monitor Signalin

Any one of us can try to enter



Monitor Signaling

Any one of us can try to enter



Java Synchronized Monitor

- await() is wait()
- signal() is notify()
- signalAll() is notifyAll()

Back to our Bounded Queue

```
public class BoundedQueue<T> {
  ReentrantLock enqLock, deqLock;
  Condition notEmptyCondition, notFullCondition;
  AtomicInteger permits;
  Node head;
  Node tail;
  int capacity;
  enqLock = new ReentrantLock();
  notFullCondition = enqLock.newCondition();
  deqLock = new ReentrantLock();
  notEmptyCondition = deqLock.newCondition();
}
```

Bounded Queue

```
public class BoundedQueue<T> {
  ReentrantLock enqLock, deqLock;
  Condition notEmptyCondition, notFullCondition;
  AtomicInteger permits;
  Node head;
  Node tail;
  int capacity;
  enqLock = new ReentrantLock();
  notFullCondition = enqLock.newCondition();
  deqLock = new ReentrantLock();
  notEmptyCondition = deqLock.newCondition();
}
```

Enq & deq locks

Bounded Queue

```
public class BoundedQueue<T> {
  ReentrantLock enqLock, deqLock;
  Condition notEmptyCondition, notFullCondition;
  AtomicInteger permits;
  Node head;
  Node tail;
  int capacity;
  enqLock = new ReentrantLock();
  notFullCondition = enqLock.newCondition();
  deqLock = new ReentrantLock();
  notEmptyCondition = deqLock.newCondition();
}
```

Reentrant lock can have a condition for threads to wait on

Bounded Queue

Num of permits ranges from 0 to capacity

```
public class BoundedQueue<T> {
  ReentrantLock enqLock, deqLock;
  Condition notEmptyCondition, notFullCondition;
  AtomicInteger permits;
  Node head;
  Node tail;
  int capacity;
  enqLock = new ReentrantLock();
  notFullCondition = enqLock.newCondition();
  deqLock = new ReentrantLock();
  notEmptyCondition = deqLock.newCondition();
}
```

Bounded Queue

```
public class BoundedQueue<T> {
  ReentrantLock enqLock, deqLock;
  Condition notEmptyCondition, notFullCondition;
  AtomicInteger permits;
  Node head;
  Node tail;
  int capacity;
  enqLock = new ReentrantLock();
  notFullCondition = enqLock.newCondition();
  deqLock = new ReentrantLock();
  notEmptyCondition = deqLock.newCondition();
}
```

Head and Tail

Bounded Queue `Enq()` Part 1

```
public void enq(T x) {
 boolean mustWakeDequeuers = false;
 enqLock.lock();
 try {
    while (permits.get() == 0){
          try {notFullCondition.await();}
    }
    Node e = new Node(x);
    tail.next = e;
    tail = e;
    if (permits.getAndDecrement() == capacity) {
       mustWakeDequeuers = true;
     }
  } finally {
     enqLock.unlock();
  }
   …
```

Bounded Queue `Enq()` Part 1

```
public void enq(T x) {
 boolean mustWakeDequeuers = false;
 enqLock.lock();
 try {
    while (permits.get() == 0){
          try {notFullCondition.await()}
    }
    Node e = new Node(x);
    tail.next = e;
    tail = e;
    if (permits.getAndDecrement() == capacity) {
      mustWakeDequeuers = true;
     }
   } finally {
    enqLock.unlock();
   }
   …
```

Lock enq lock

Bounded Queue `Enq()` Part 1

```
public void enq(T x) {
 boolean mustWakeDequeuers = false;
 enqLock.lock();
 try {
    while (permits.get() == 0){
          try {notFullCondition.await()}
    }
    Node e = new Node(x);
    tail.next = e;
    tail = e;
    if (permits.getAndDecrement() == capacity) {
      mustWakeDequeuers = true;
    }
  } finally {
    enqLock.unlock();
  }
  …
```

If permits = 0 wait till notFullCondition becomes true then check permits again…

Bounded Queue `Enq()` Part 1

```
public void enq(T x) {
 boolean mustWakeDequeuers = false;
 enqLock.lock();
 try {
    while (permits.get() == 0){
          try {notFullCondition.await()}
    }
    Node e = new Node(x);
    tail.next = e;
    tail = e;
    if (permits.getAndDecrement() == capacity) {
      mustWakeDequeuers = true;
     }
   } finally {
     enqLock.unlock();
   }
   …
```

Add a new node

Bounded Queue `Enq()` Part 1

```
public void enq(T x) {
 boolean mustWakeDequeuers = false;
 enqLock.lock();
 try {
    while (permits.get() == 0){
          try {notFullCondition.await}
    }
    Node e = new Node(x);
    tail.next = e;
    tail = e;
    if (permits.getAndDecrement() == capacity) {
      mustWakeDequeuers = true;
     }
   } finally {
     enqLock.unlock();
   }
  …
```

If I was the enqueuer that changed queue state from empty to none-empty will need to wake dequeuers

Bounded Queue `Enq()` Part 1

```
public void enq(T x) {
 boolean mustWakeDequeuers = false;
 enqLock.lock();
 try {
    while (permits.get() == 0){
          try {notFullCondition.await()}
    }
    Node e = new Node(x);
    tail.next = e;
    tail = e;
    if (permits.getAndDecrement() == capacity) {
      mustWakeDequeuers = true;
     }
   } finally {
     enqLock.unlock();
   }
   …
```

Release the enq lock

Bounded Queue `Enq()` Part 1

```
public void enq(T x) {
 boolean mustWakeDequeuers = false;
 enqLock.lock();
 try {
     while (permits.get() == 0){
            try {notFullCondition.await();}
     }
     Node e = new Node(x);
     tail.next = e;
     tail = e;
     if (permits.getAndDecrement() == capacity) {
        mustWakeDequeuers = true;
      }
    } finally {
      enqLock.unlock();
    }
    …
```

Bounded Queue `Enq()` Part 2

```
public void enq(T x) {
 …
   if (mustWakeDequeuers) {
     deqLock.lock();
     try {
       notEmptyCondition.signalAll();
     } finally {
       deqLock.unlock();
     }
   }
 }
```

Bounded Queue `Enq()` Part 2

```
public void enq(T x) {
 …
  if (mustWakeDequeuers) {
    deqLock.lock();
    try {
      notEmptyCondition.signalAll();
    } finally {
      deqLock.unlock();
    }
  }
}
```

To let the dequeuers know that the queue is non-empty, acquire deqLock

Bounded Queue `Enq()` Part 2

```
public void enq(T x) {
 …
   if (mustWakeDequeuers) {
     deqLock.lock();
     try {
      notEmptyCondition.signalAll();
     } finally {
      deqLock.unlock();
     }
   }
 }
```

Signal all dequeuers waiting that they can attempt to re-acquire deqLock

Bounded Queue `Enq()` Part 2

```
public void enq(T x) {
 …
   if (mustWakeDequeuers) {
     deqLock.lock();
     try {
       notEmptyCondition.signalAll();
     } finally {
       deqLock.unlock();
     }
   }
 }
```

Release deqLock

The Shared Counter

- The `enq()` and `deq()` methods
  - Don’t access the same lock concurrently
  - But they still share a counter
  - Which they both increment or decrement on every method call
  - Can we get rid of this bottleneck?

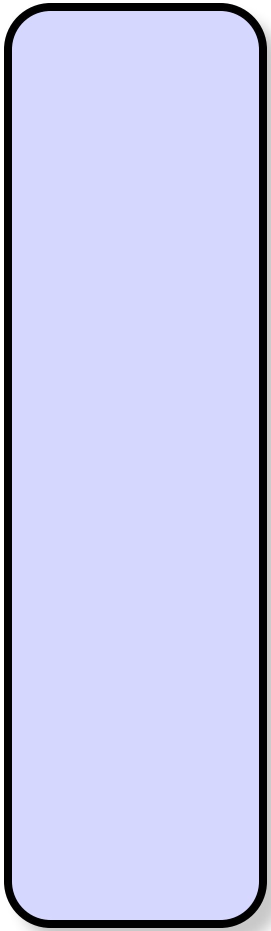
Split the Counter

- The `enq()` method
  - Decrements only
  - Cares only if value is zero
- The `deq()` method
  - Increments only
  - Cares only if value is capacity

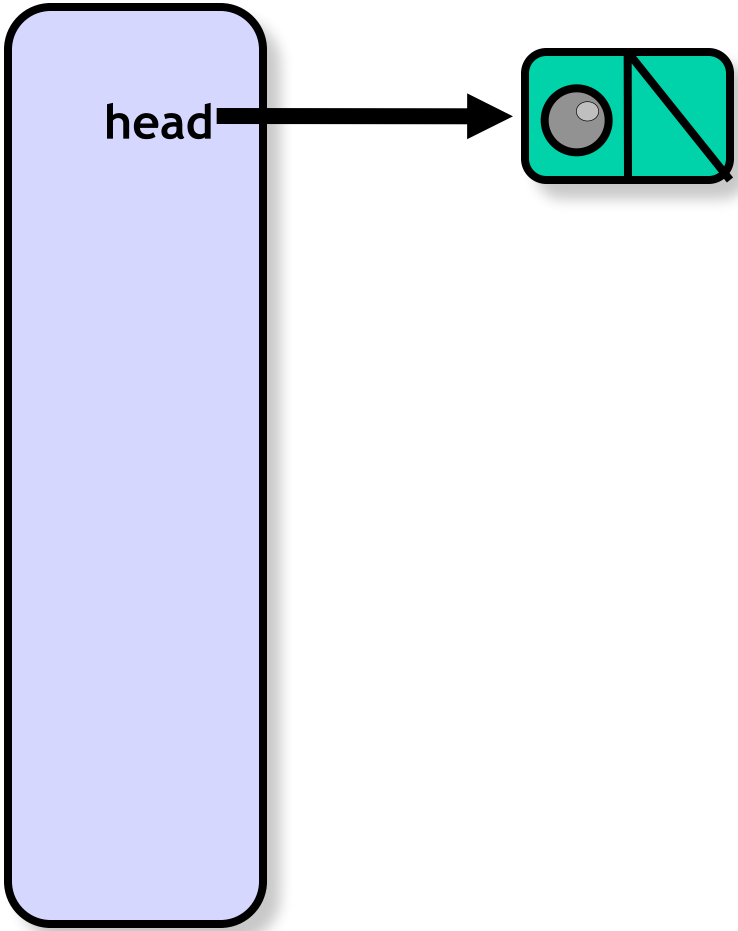
Split Counter

- Enqueuer decrements `enqSidePermits`
- Dequeuer increments `deqSidePermits`
- When enqueuer runs out of space
    - Locks `deqLock`
    - Transfers `permits`
- Intermittent synchronization
    - Not with each method call
    - Need both locks! (careful …)

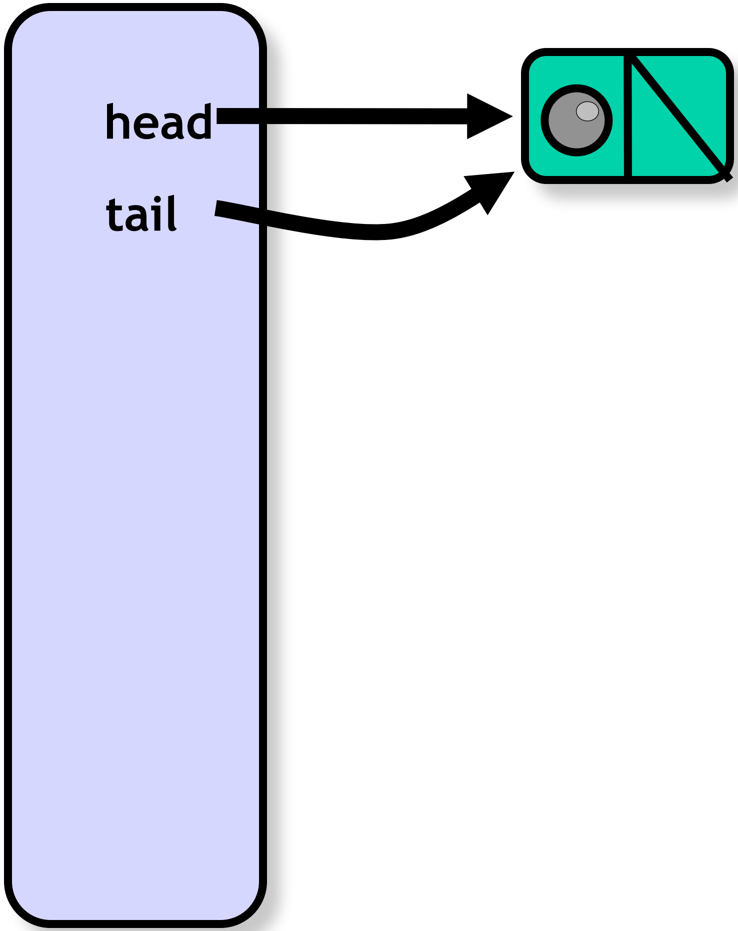
A Lock-Free Queue



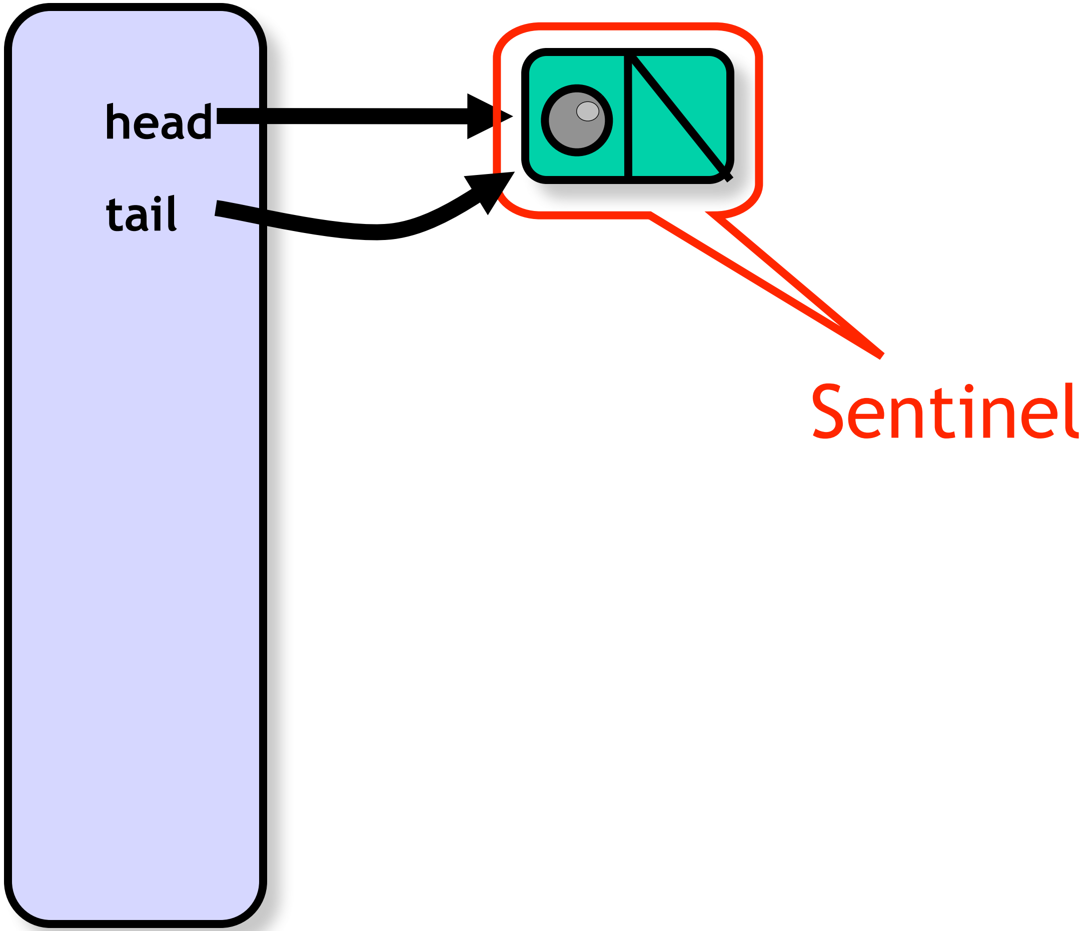
A Lock-Free Queue



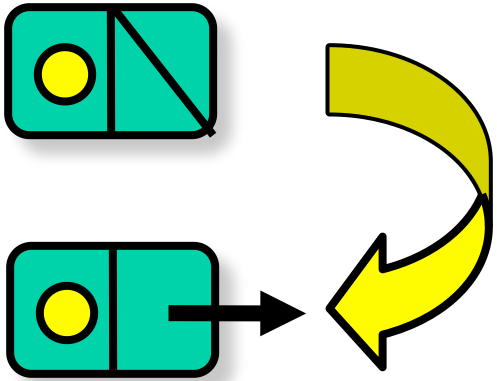
A Lock-Free Queue



A Lock-Free Queue

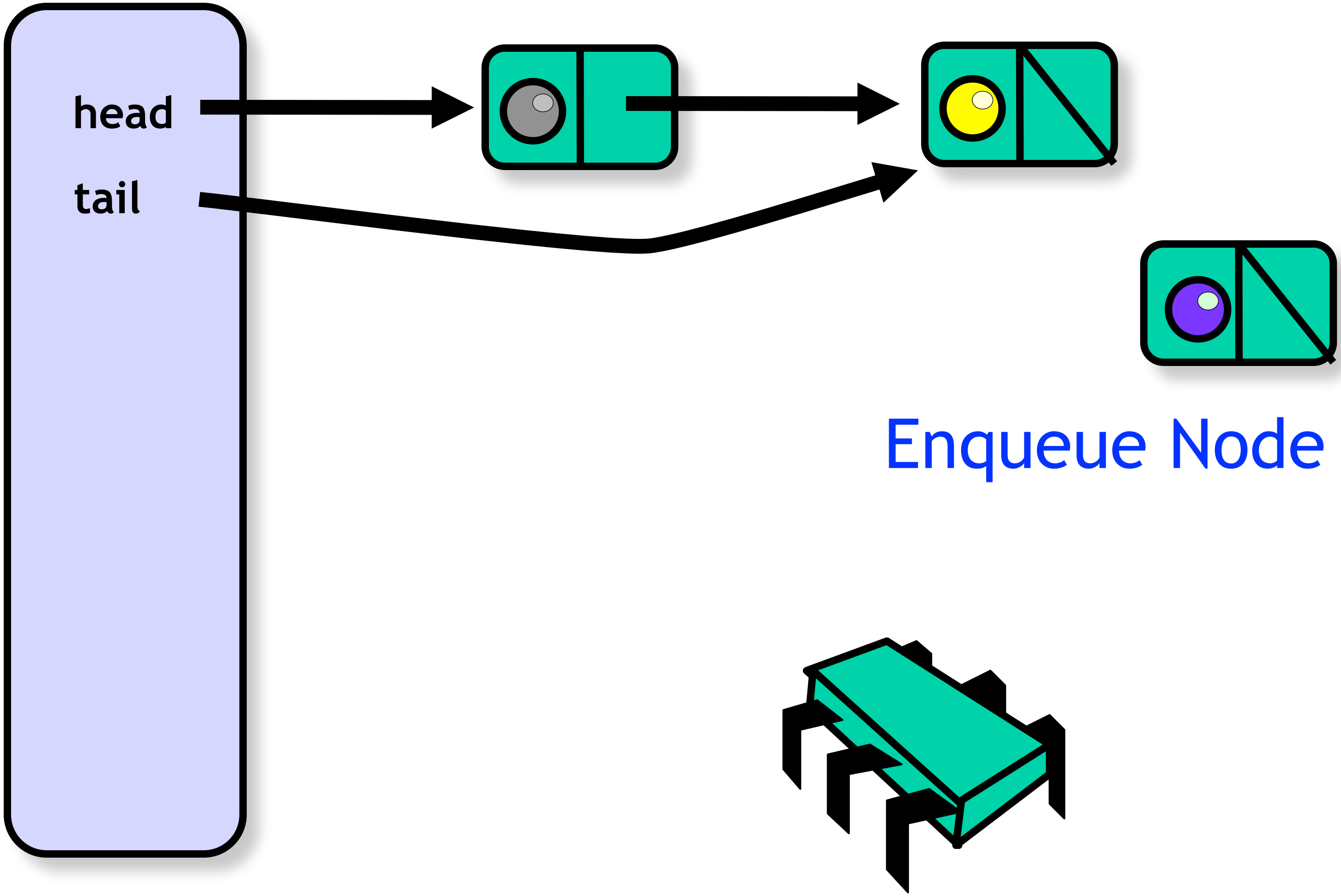


Compare and Set



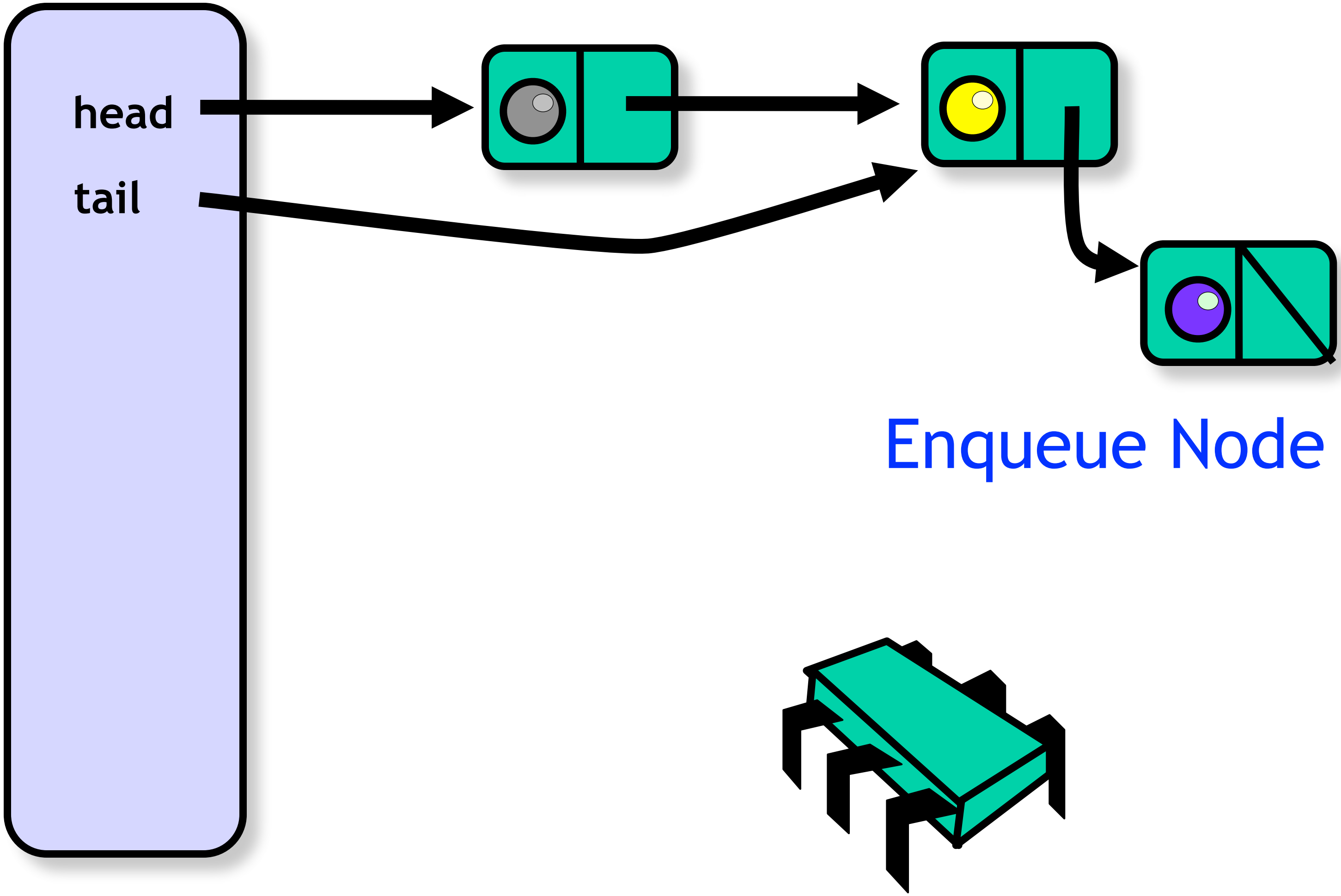
CAS

Enqueue Step One



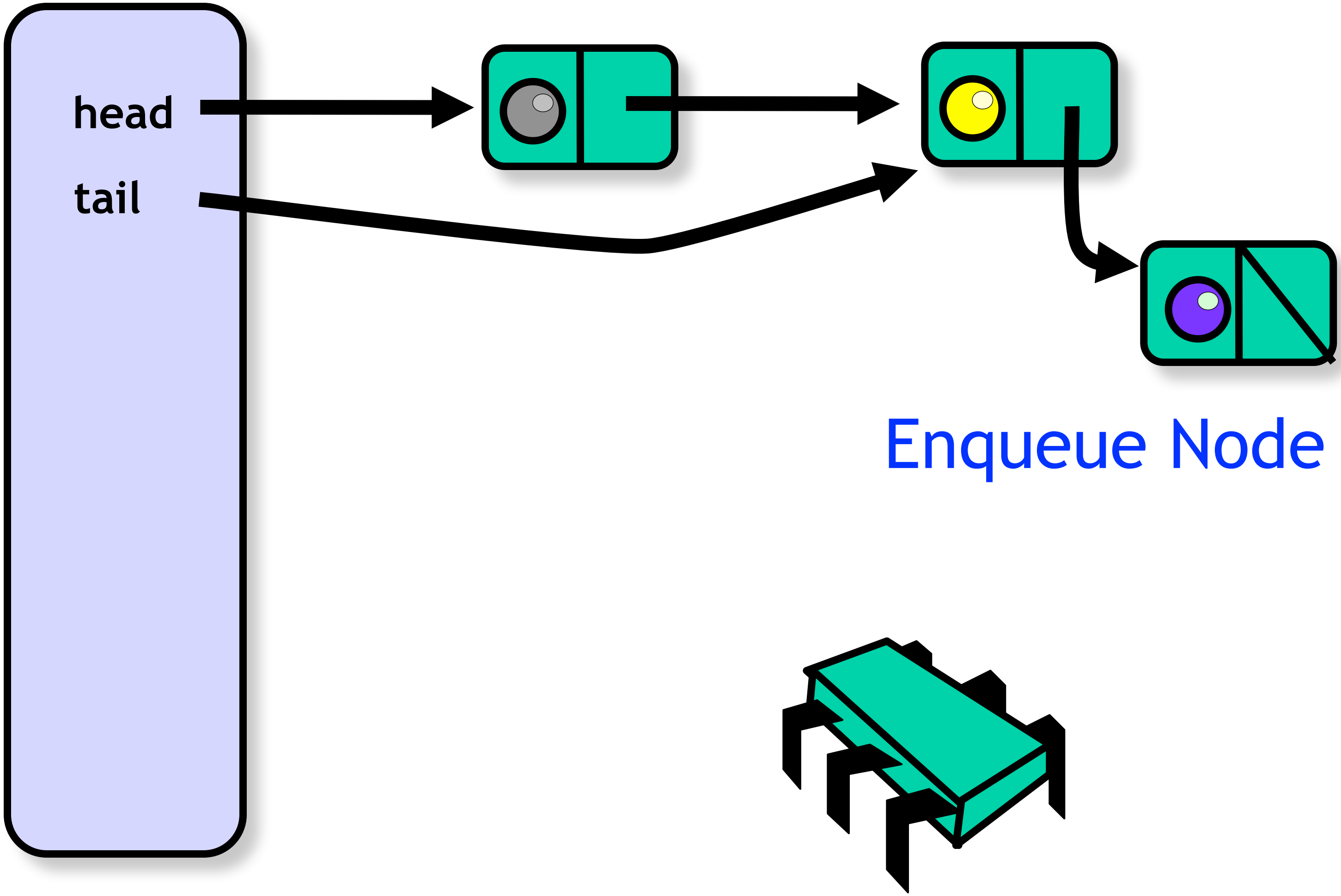
Enqueue Node

Enqueue Step One

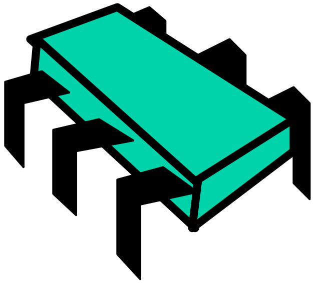


Enqueue Node

Enqueue Step Two

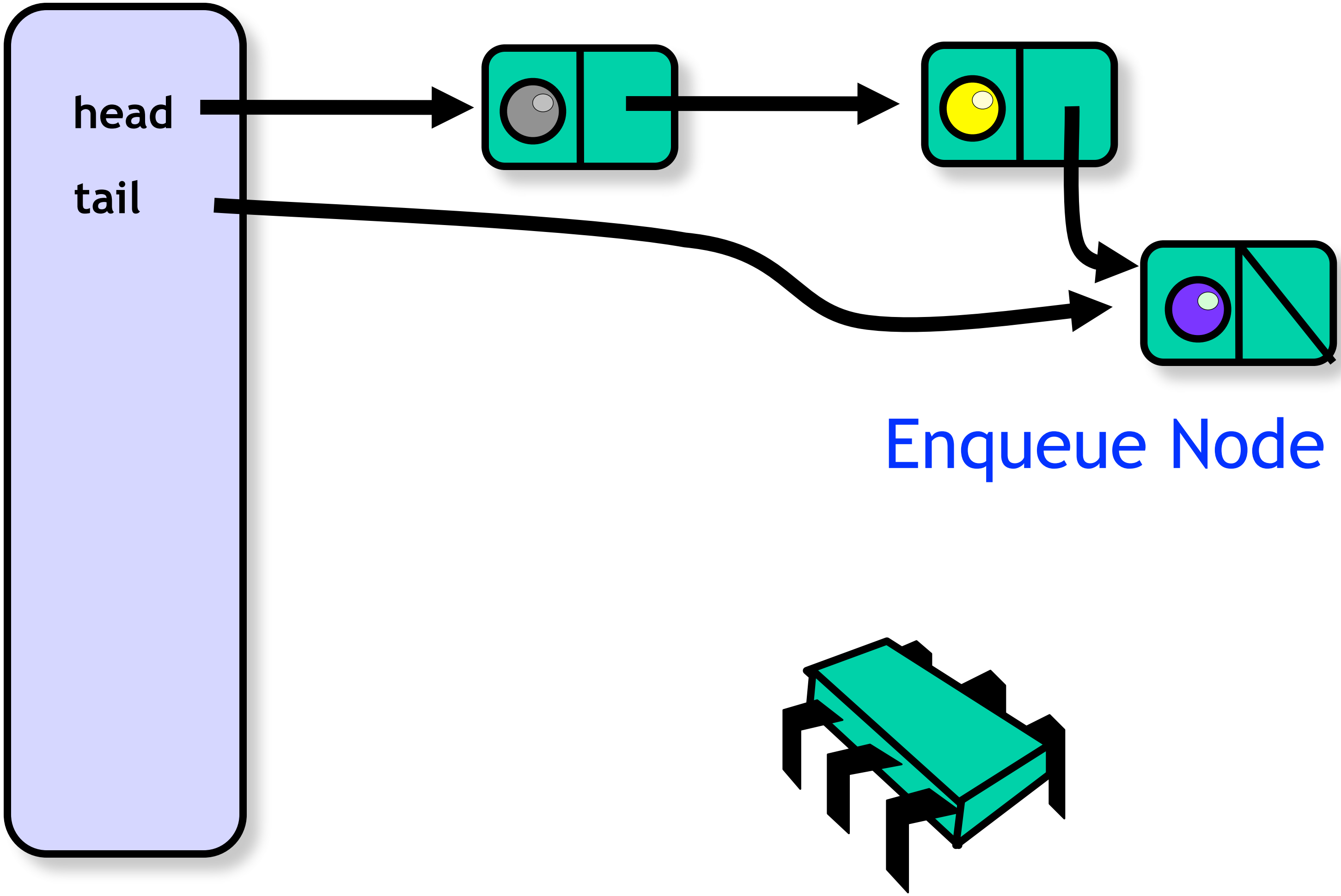


Enqueue Node



© Herlihy-Shavit

Enqueue Step Two



Enqueue Node

Enqueue

- These two steps are not atomic
- The tail field refers to either
  - Actual last Node (good)
  - Penultimate Node (not so good)

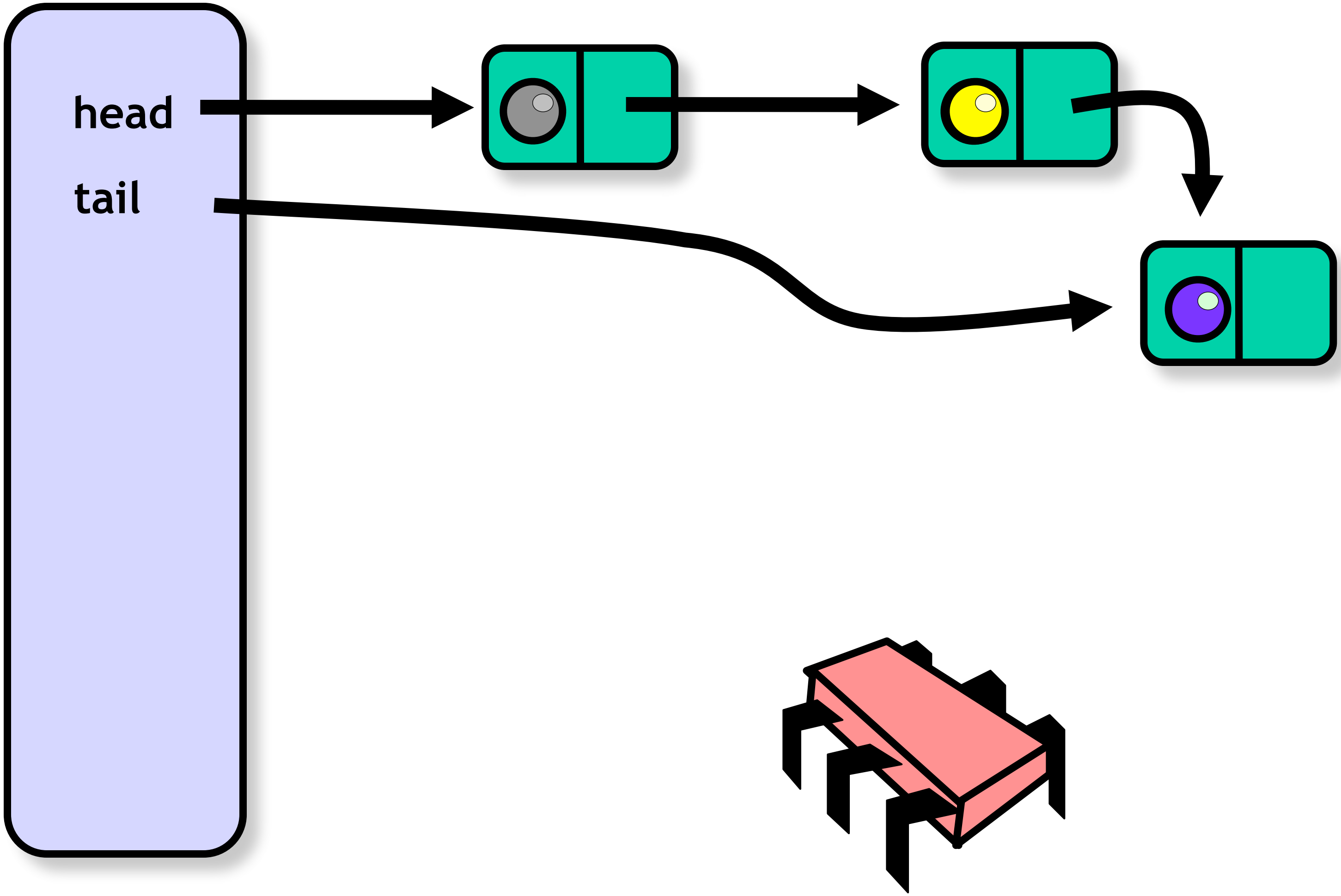
Enqueue

- What do you do if you find
  - A trailing tail?
- Stop and fix it
  - If node pointed to by tail has non-null next field
  - CAS the queue's tail field to tail.next

When CASs Fail

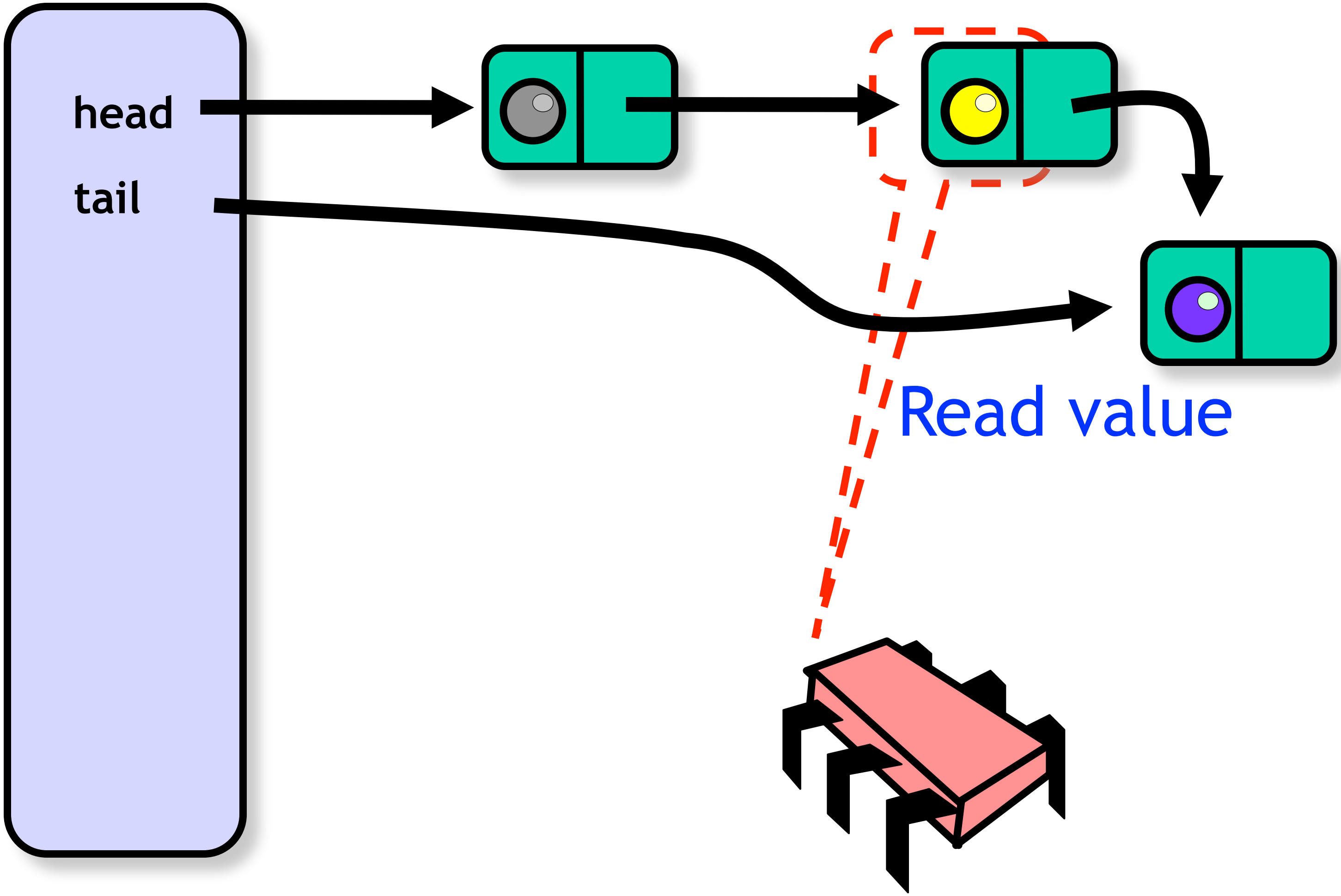
- In Step One (logical enqueue)
  - Retry loop
  - Method still lock-free (why?)
- In Step Two (physical enqueue)
  - Ignore it (why?)

Dequeuer

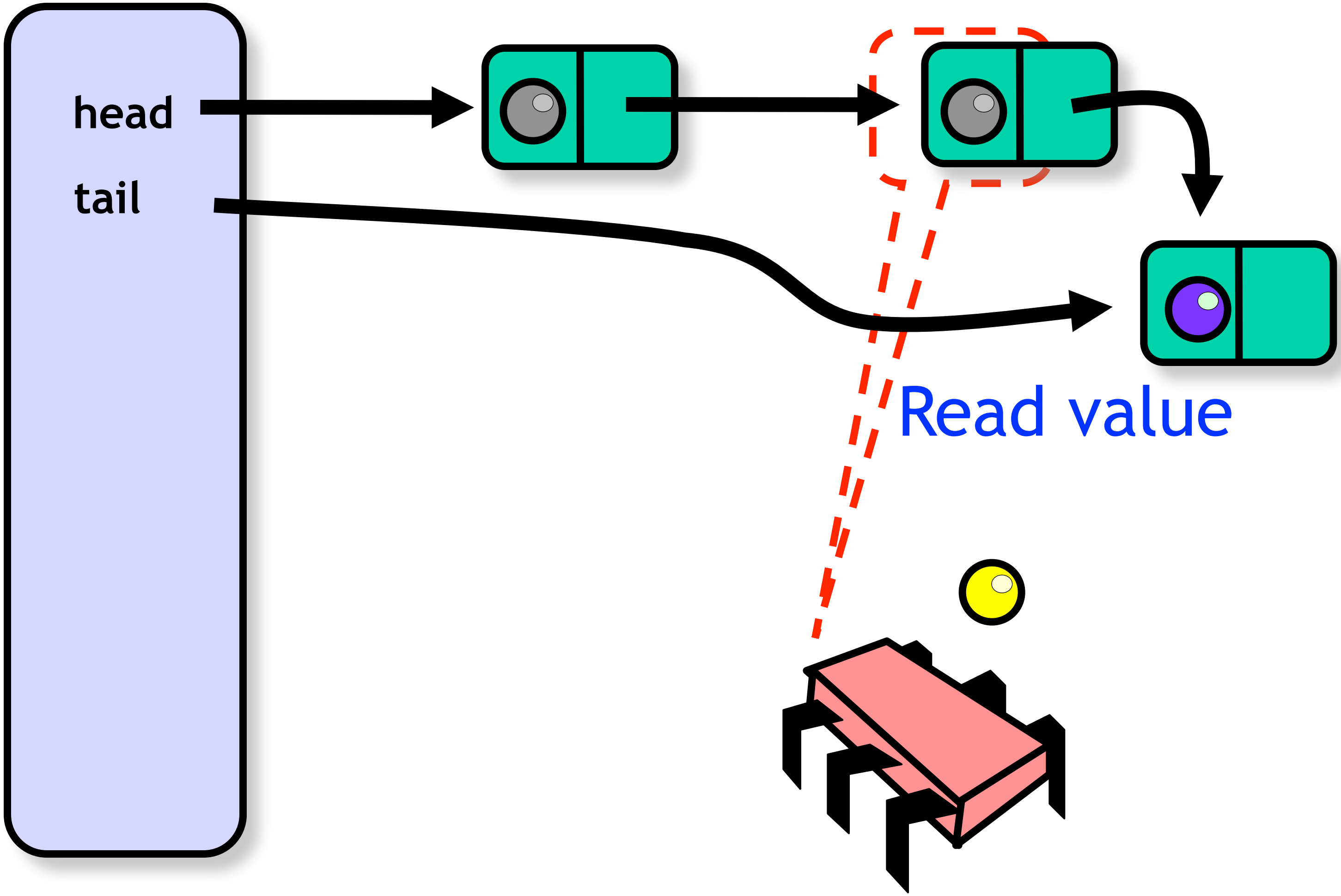


© Herlihy-Shavit

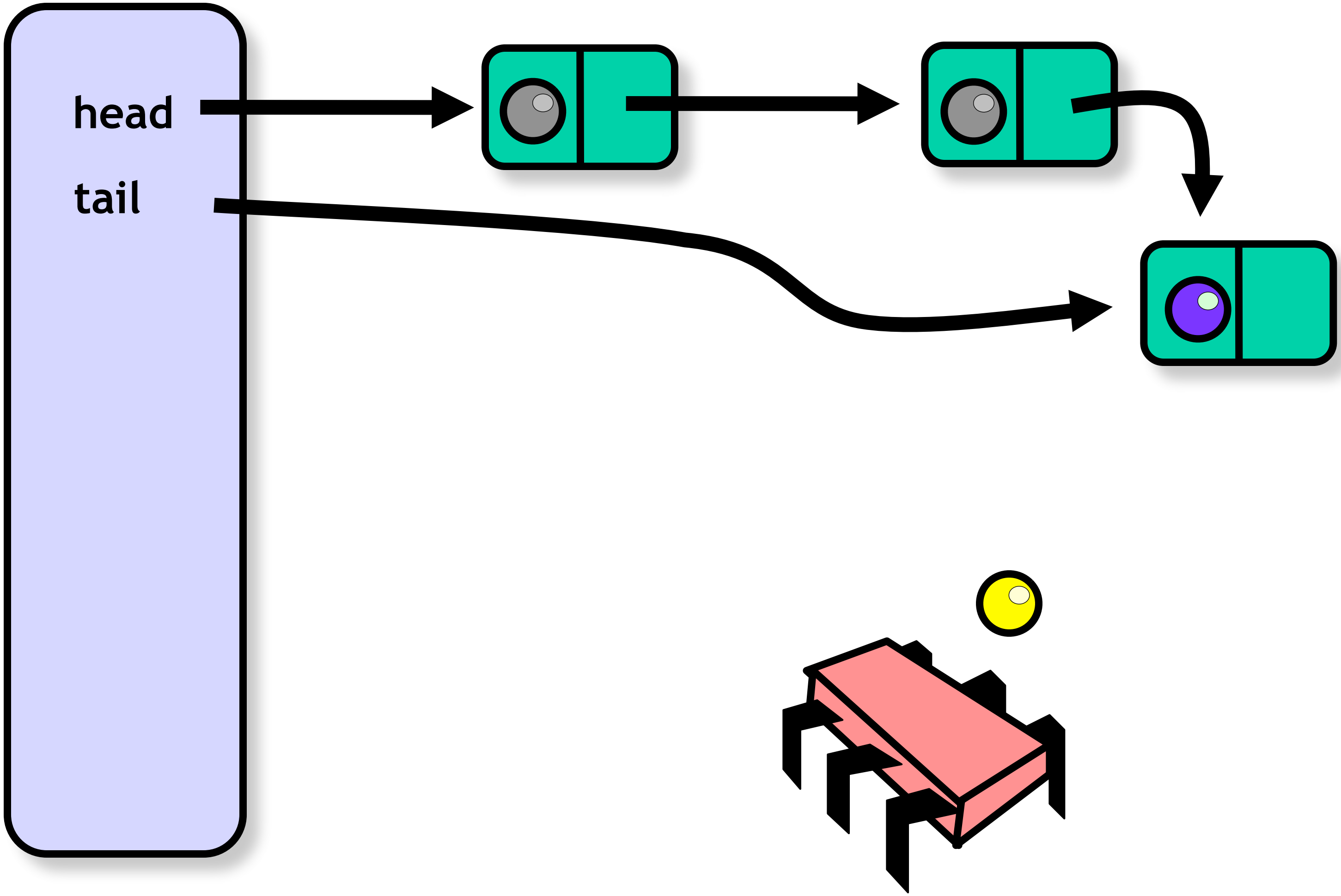
Dequeuer



Dequeuer

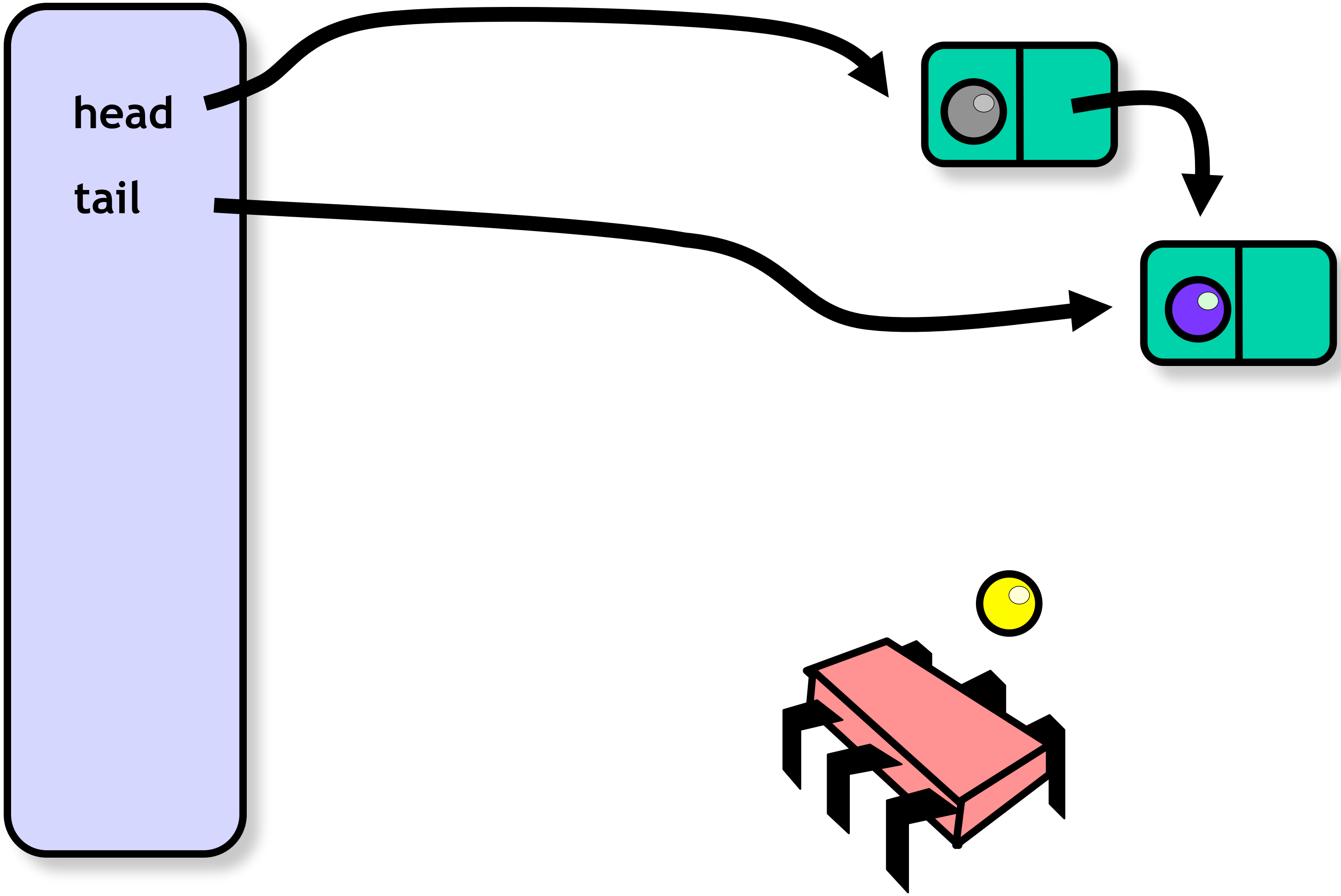


Dequeuer



Make first Node new sentinel

Dequeuer

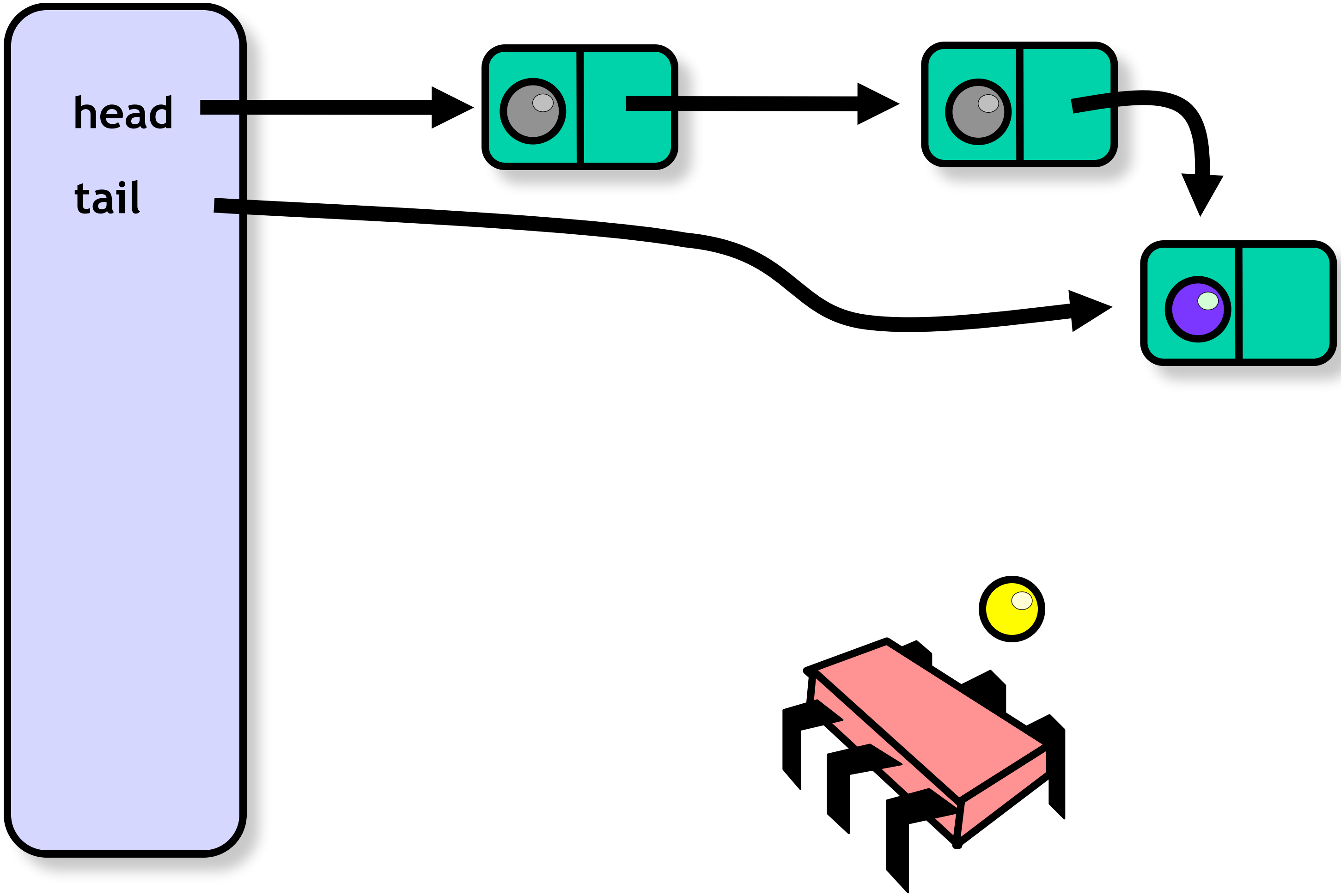


© Herlihy-Shavit

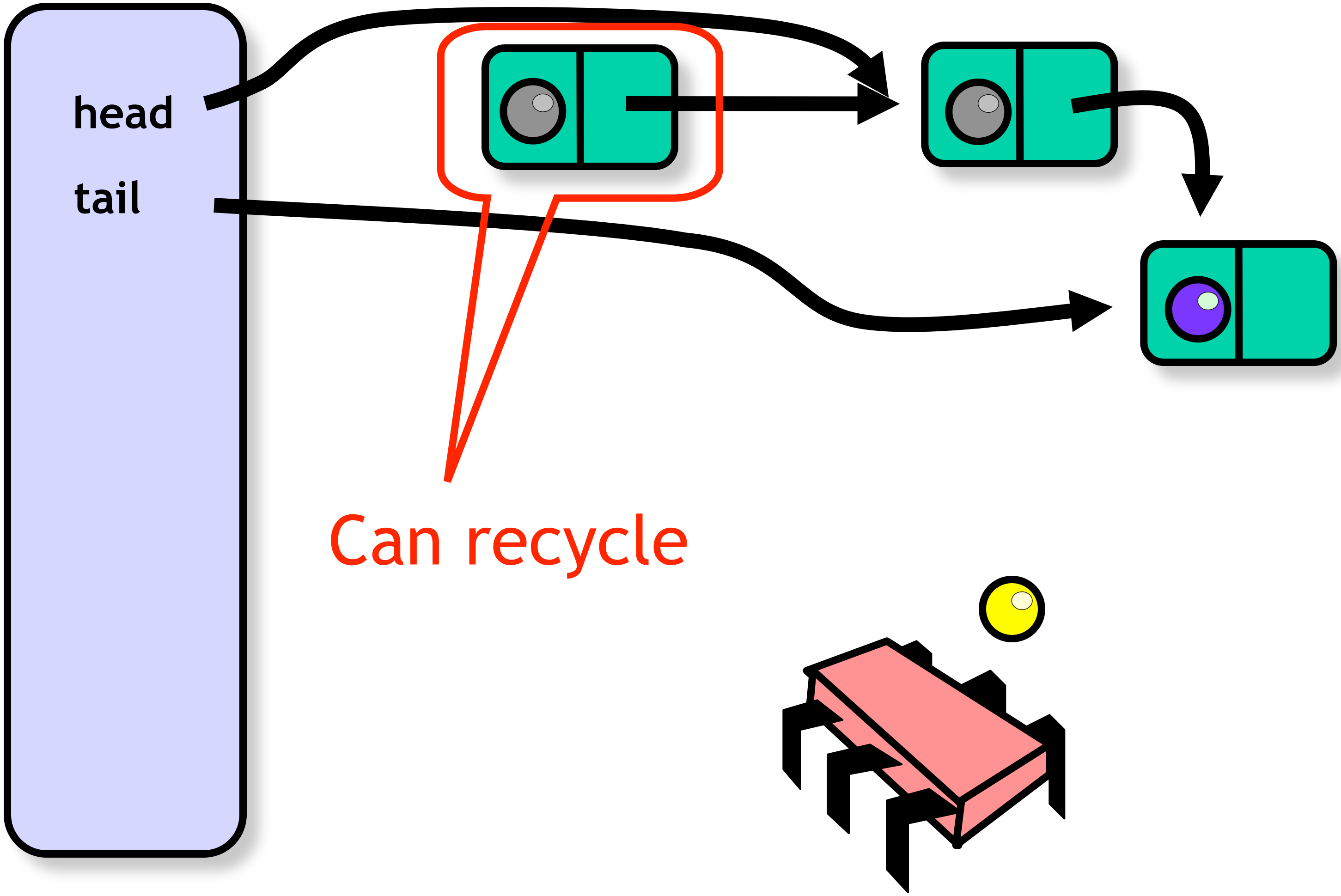
Memory Reuse?

- What do we do with nodes after we dequeue them?
- Java: let garbage collector deal?
- Suppose there isn’t a GC, or we don’t want to use it?

Dequeuer



Dequeuer

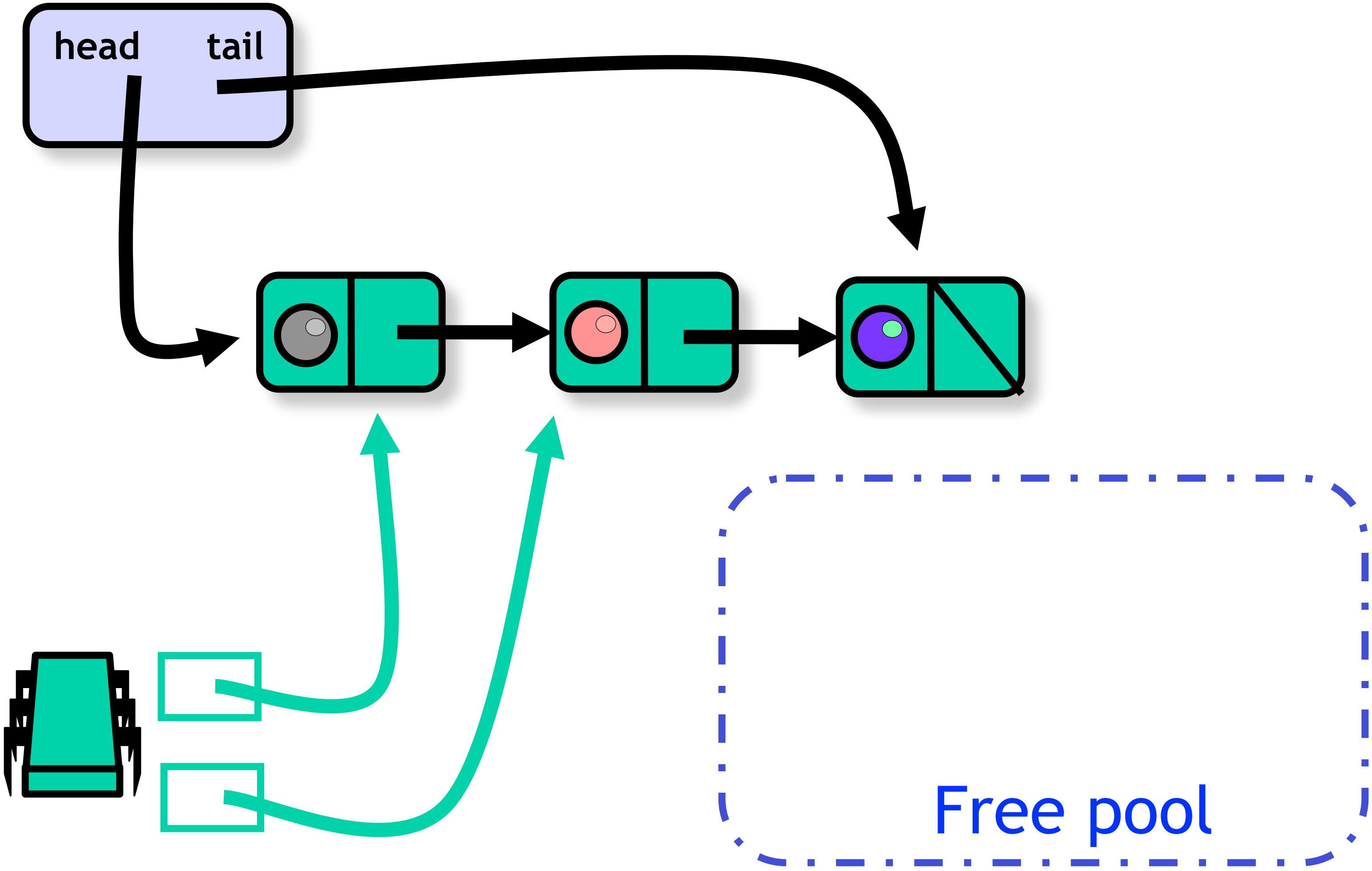


Can recycle

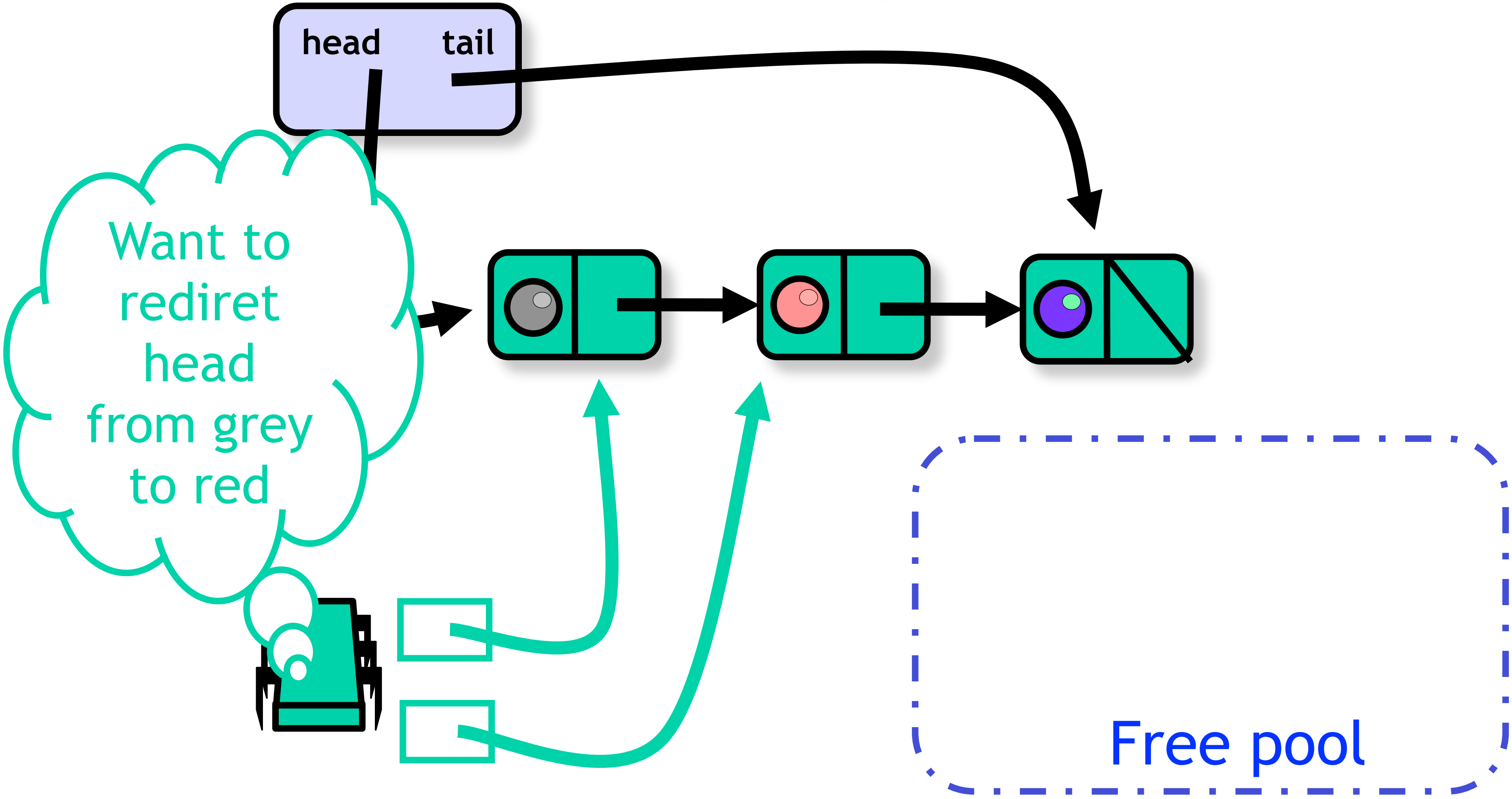
Simple Solution

- Each thread has a free list of unused queue nodes
- Allocate node: pop from list
- Free node: push onto list
- Deal with underflow somehow …

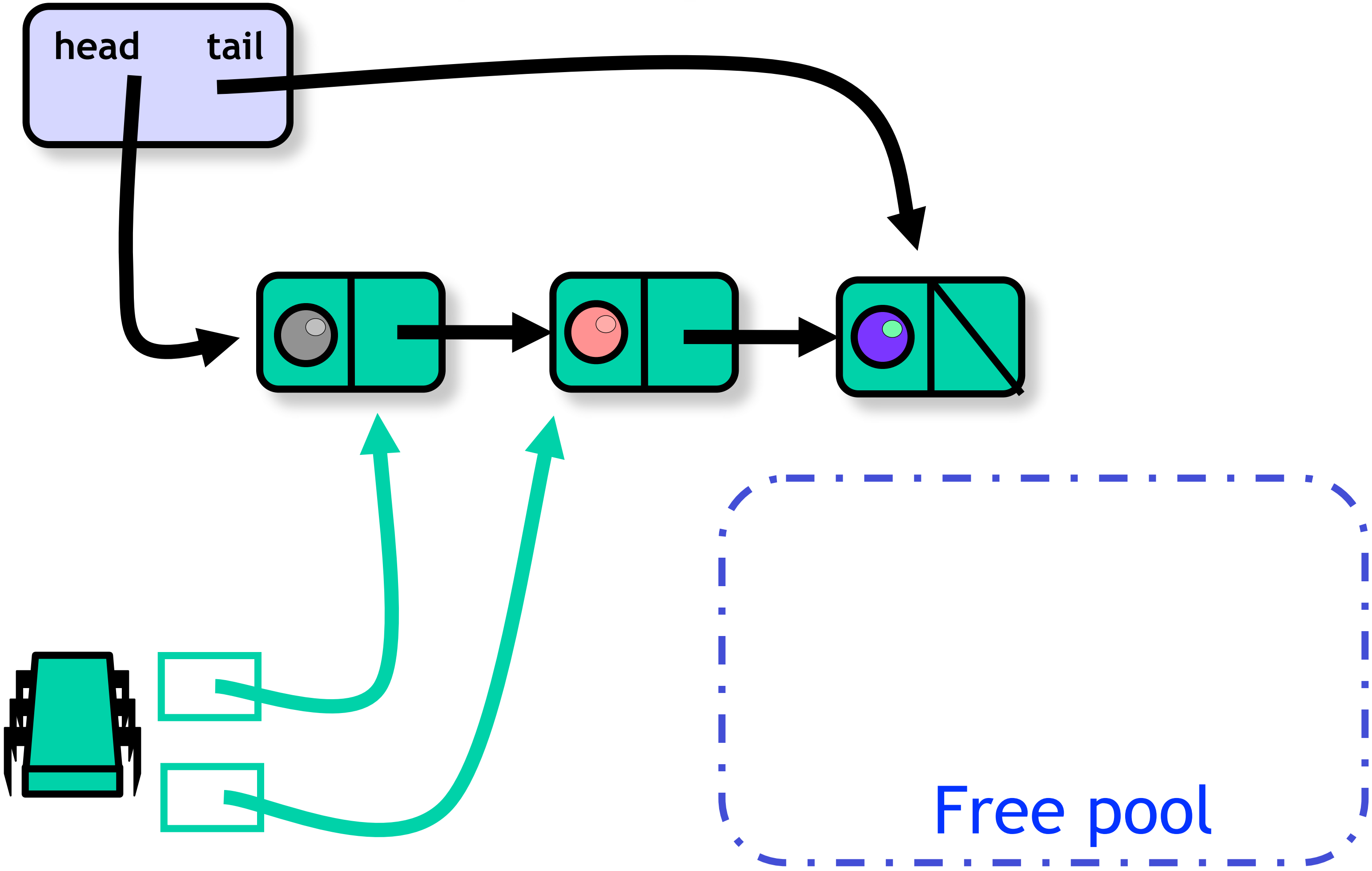
Why Recycling is Hard



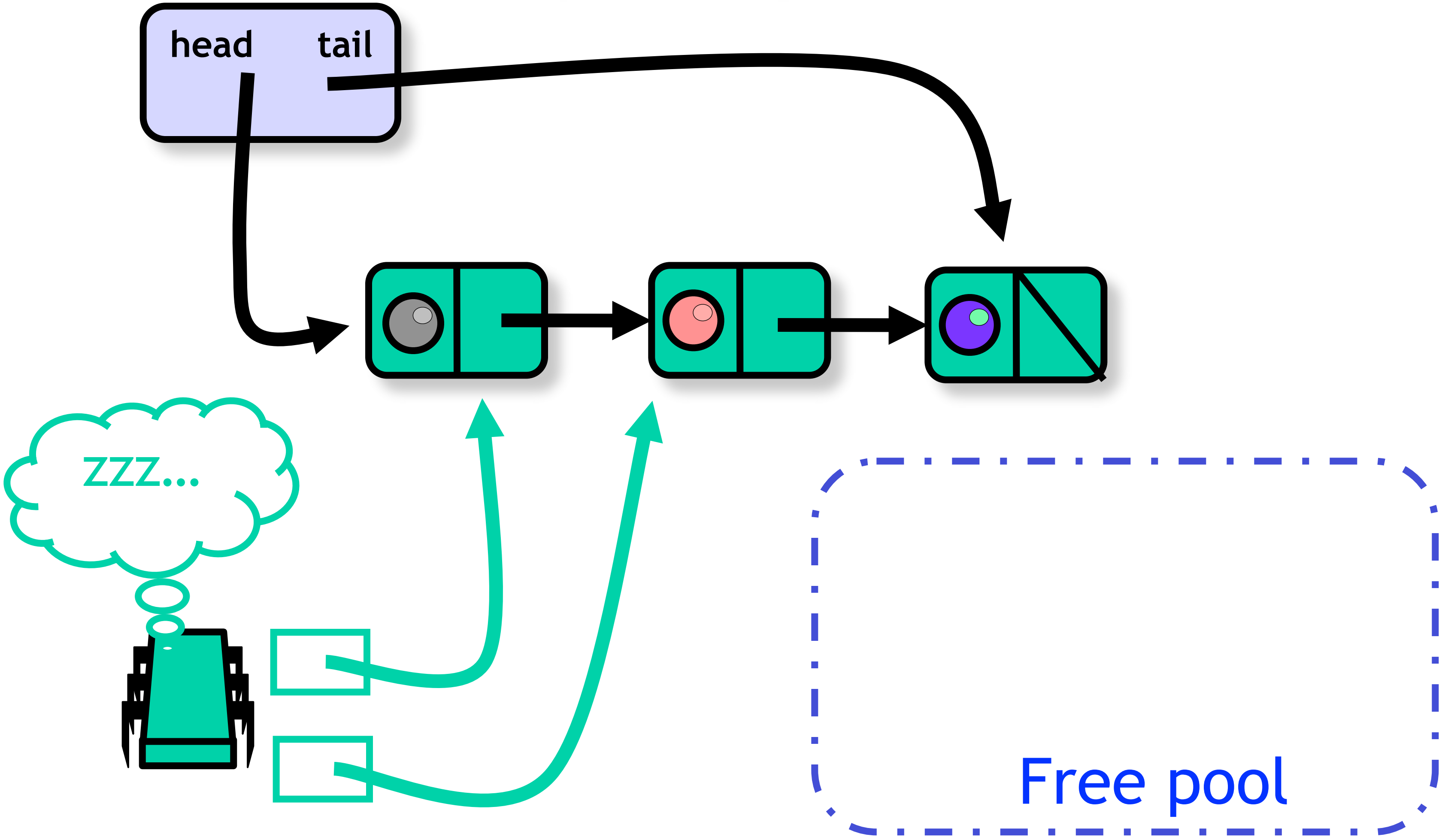
Why Recycling is Hard



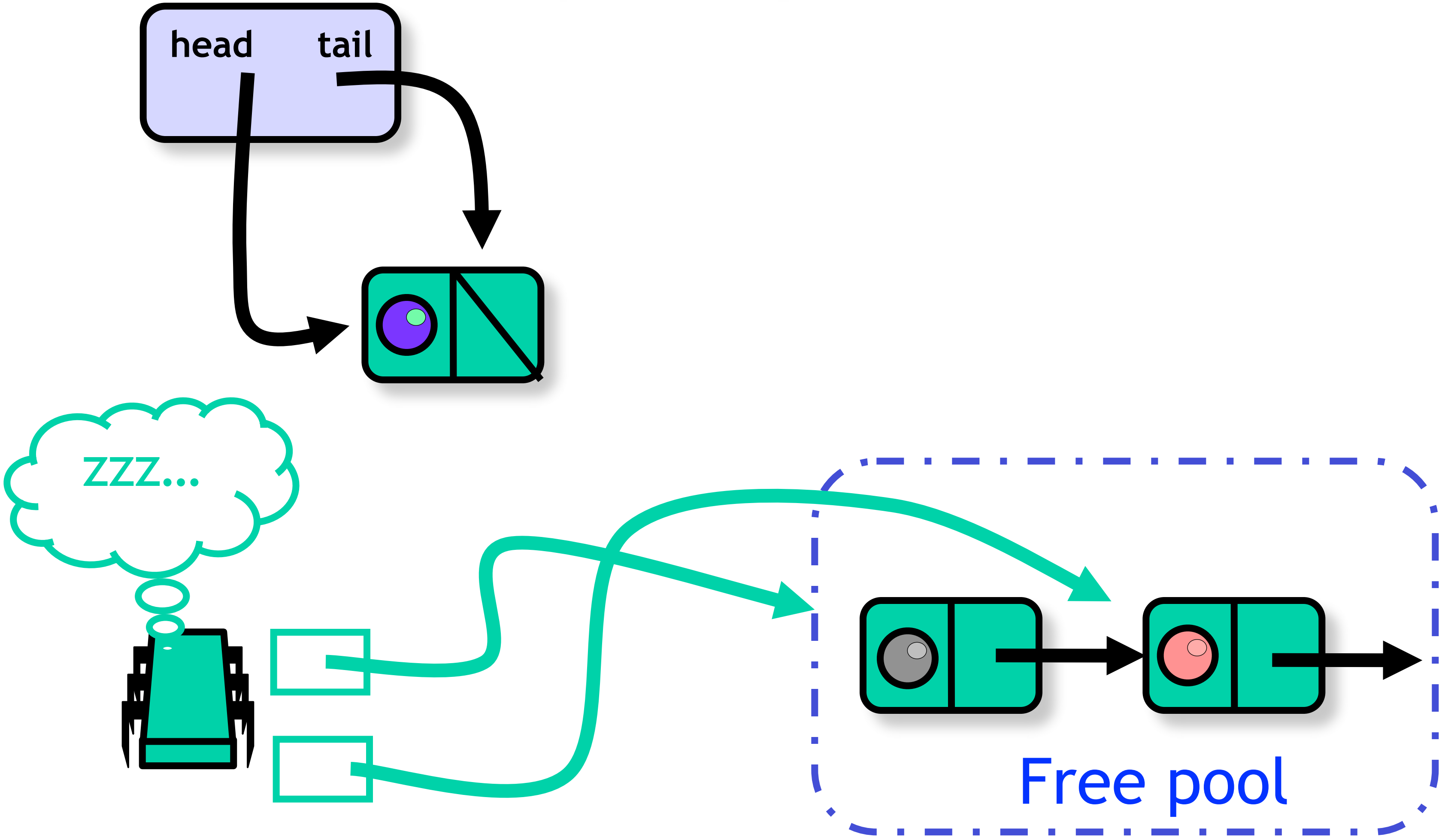
Why Recycling is Hard



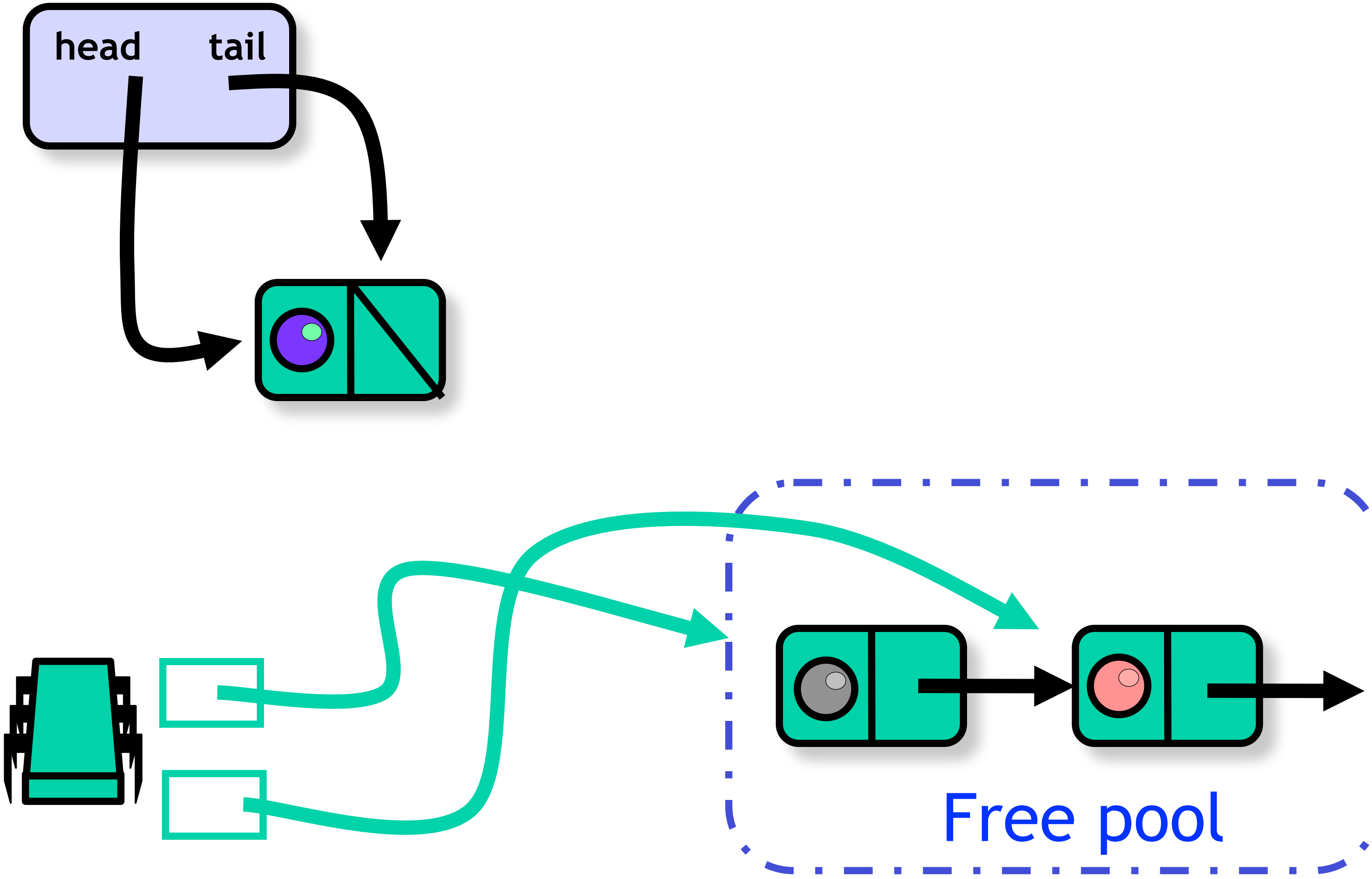
Why Recycling is Hard



Why Recycling is Hard

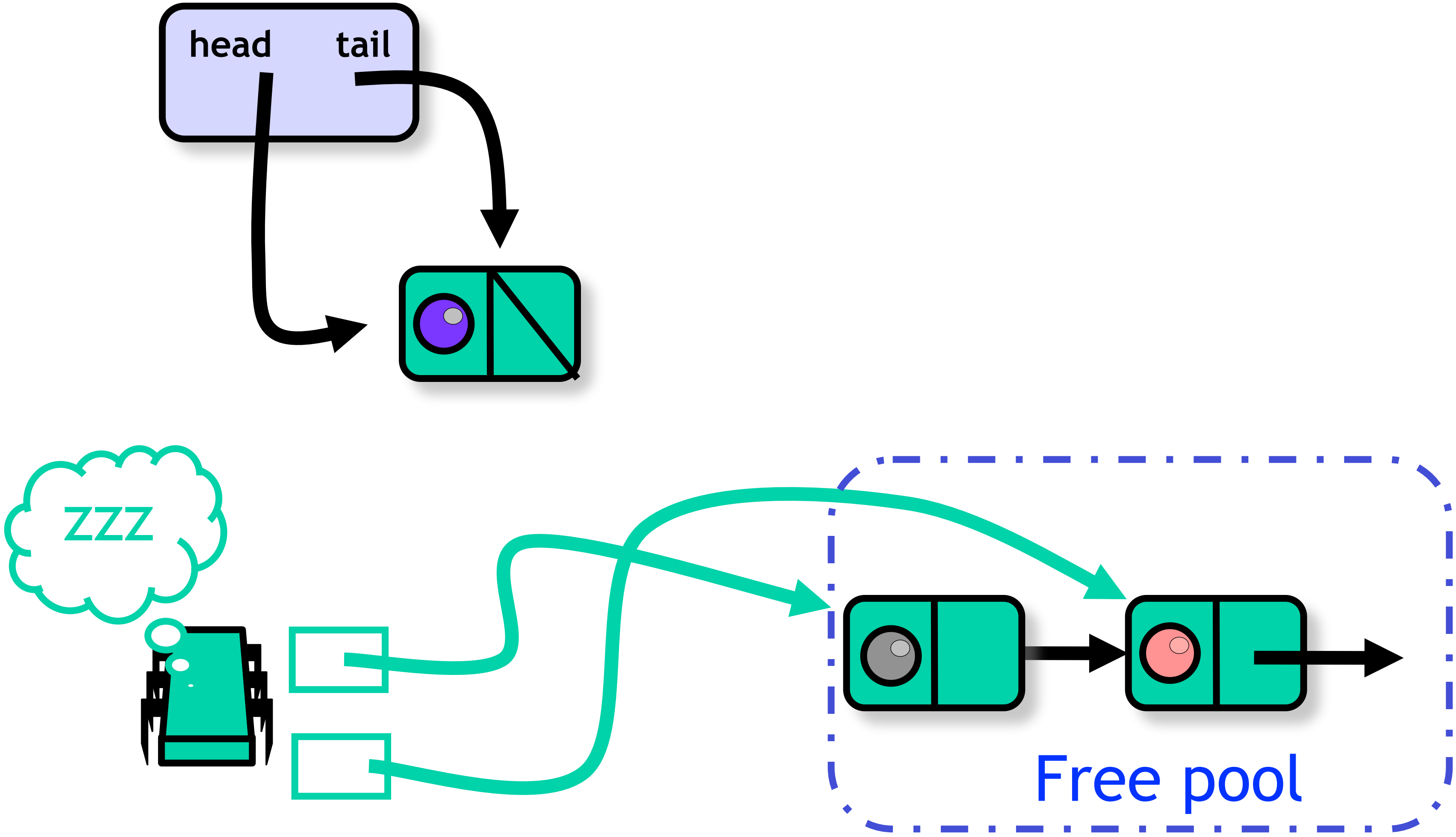


Why Recycling is Hard

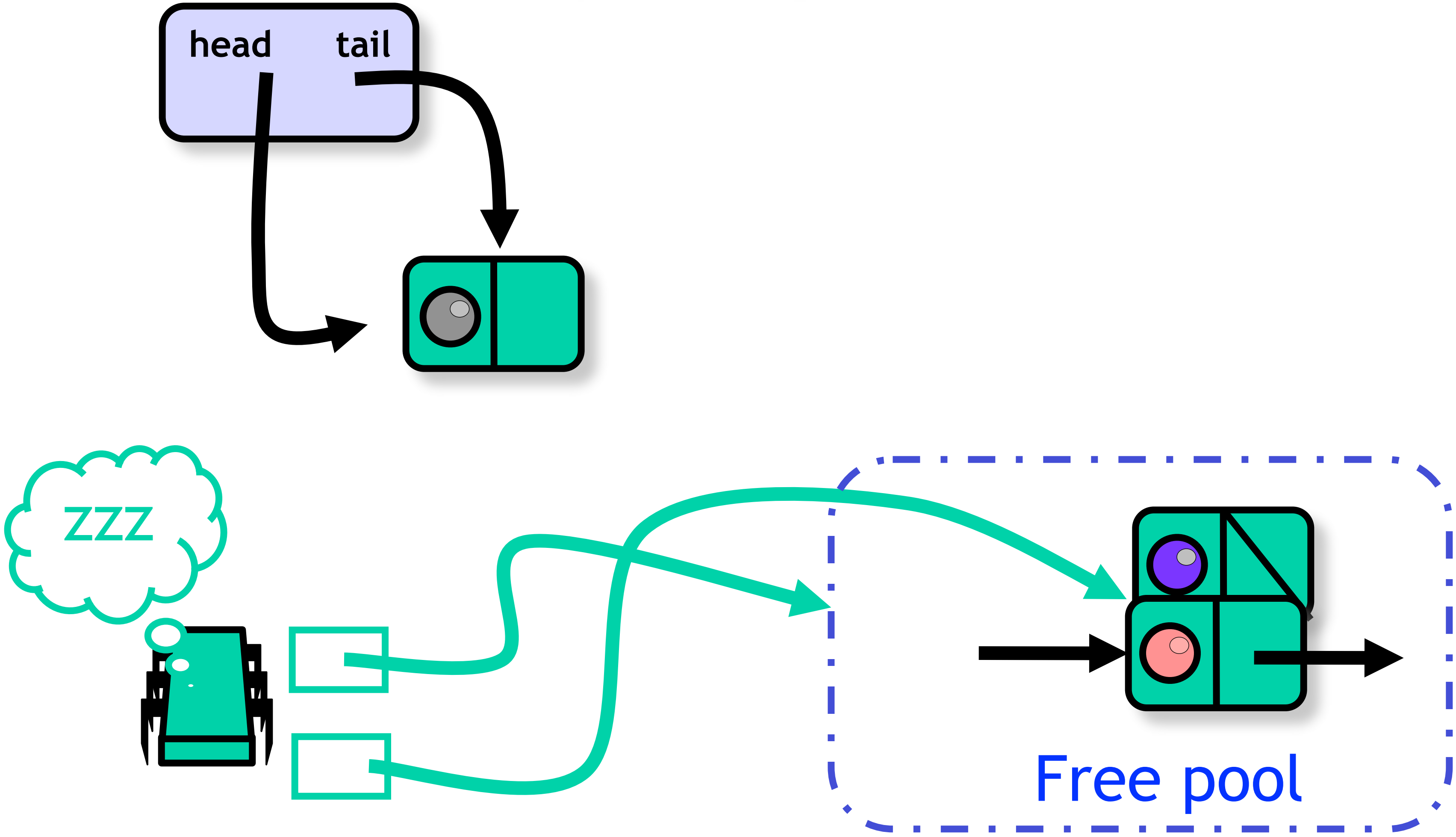


© Herlihy-Shavit

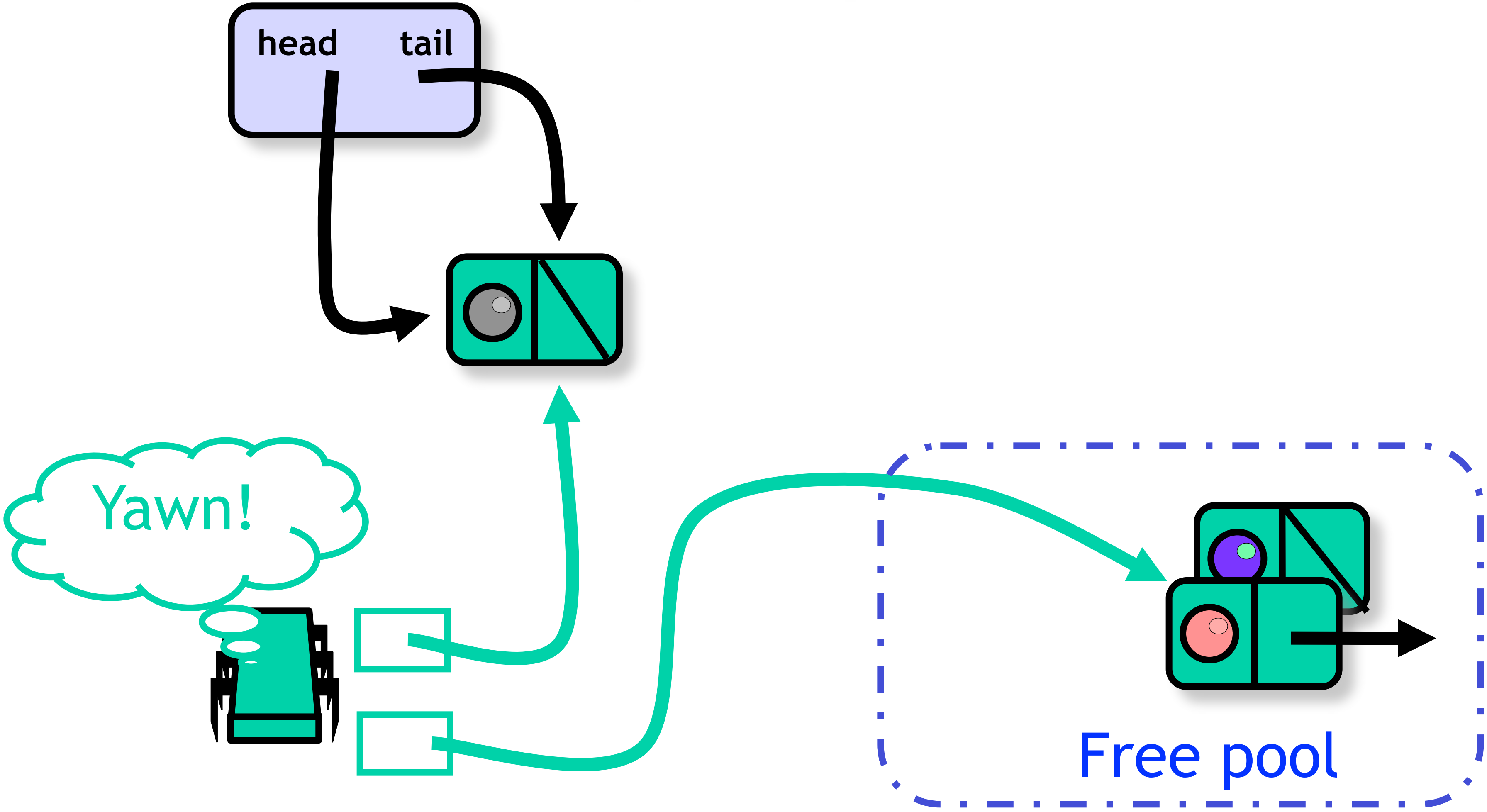
Why Recycling is Hard



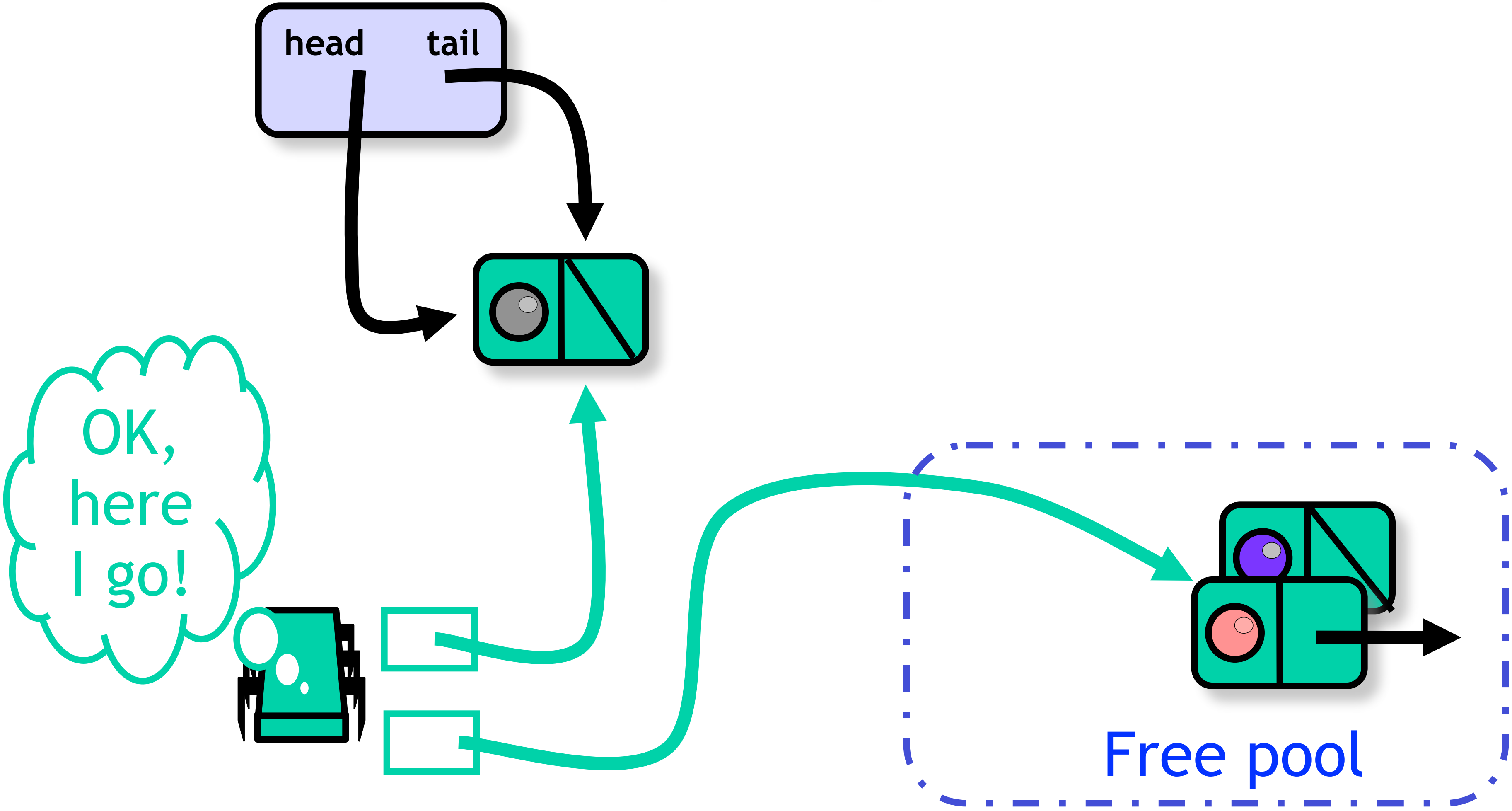
Why Recycling is Hard



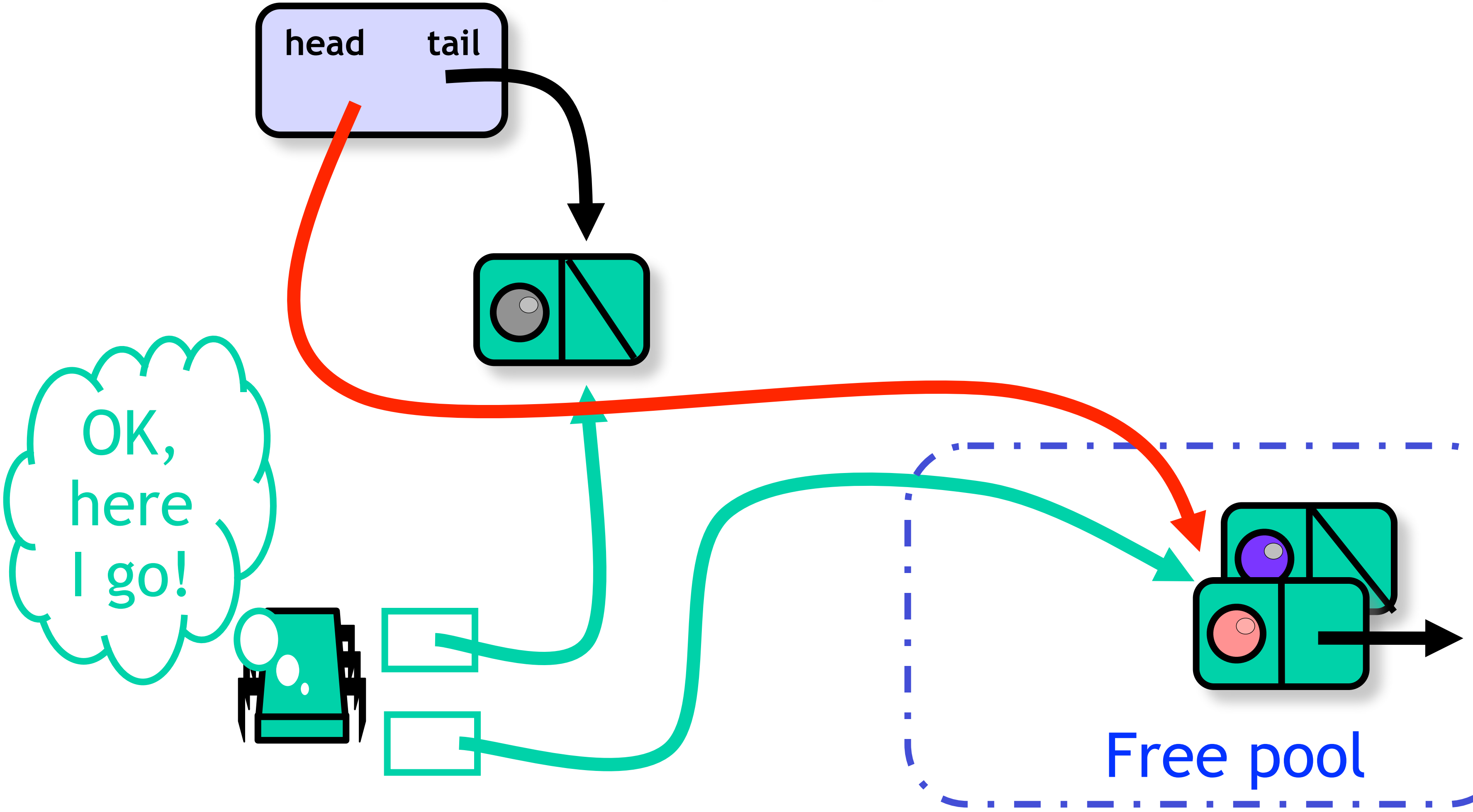
Why Recycling is Hard



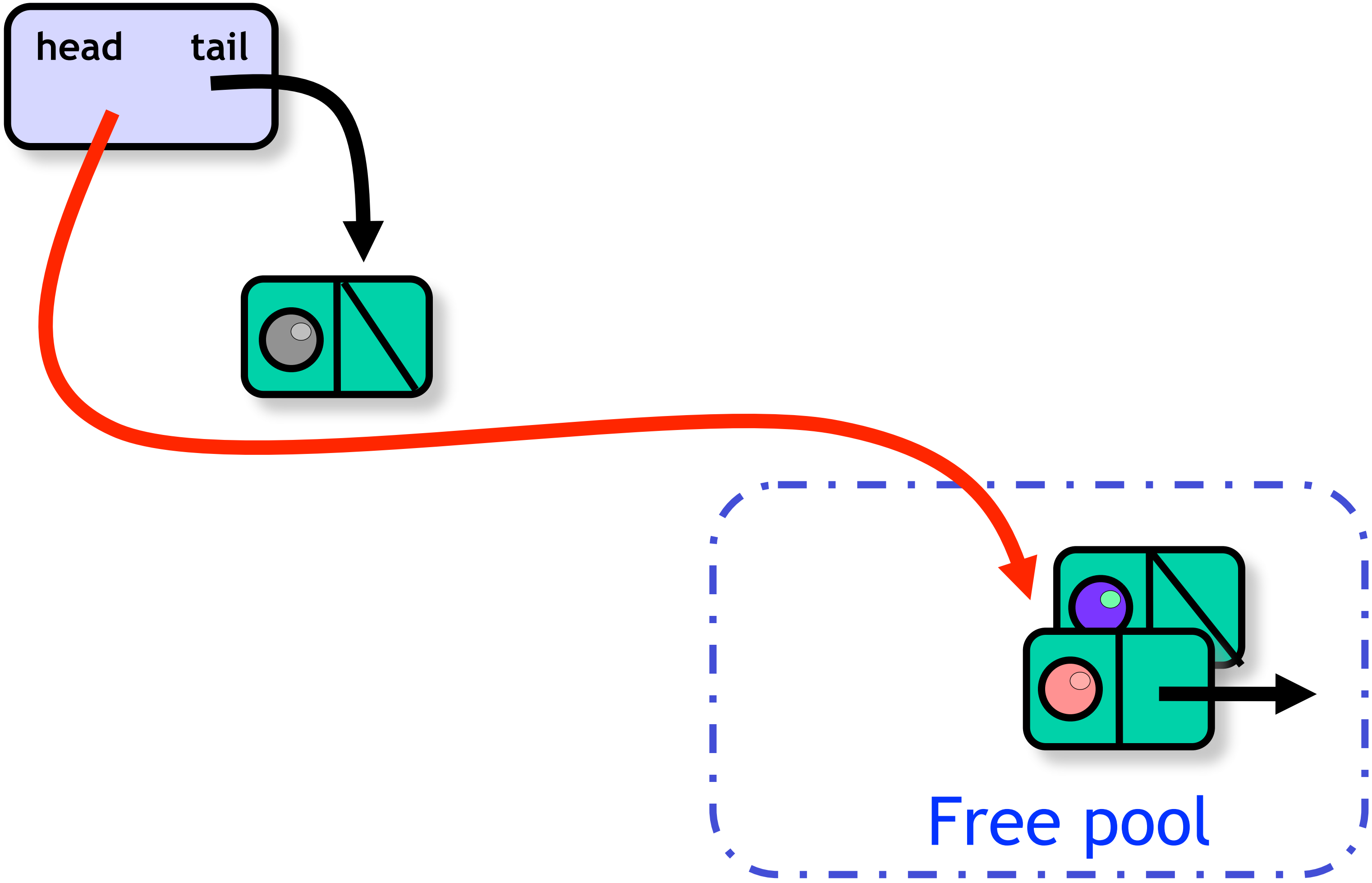
Why Recycling is Hard



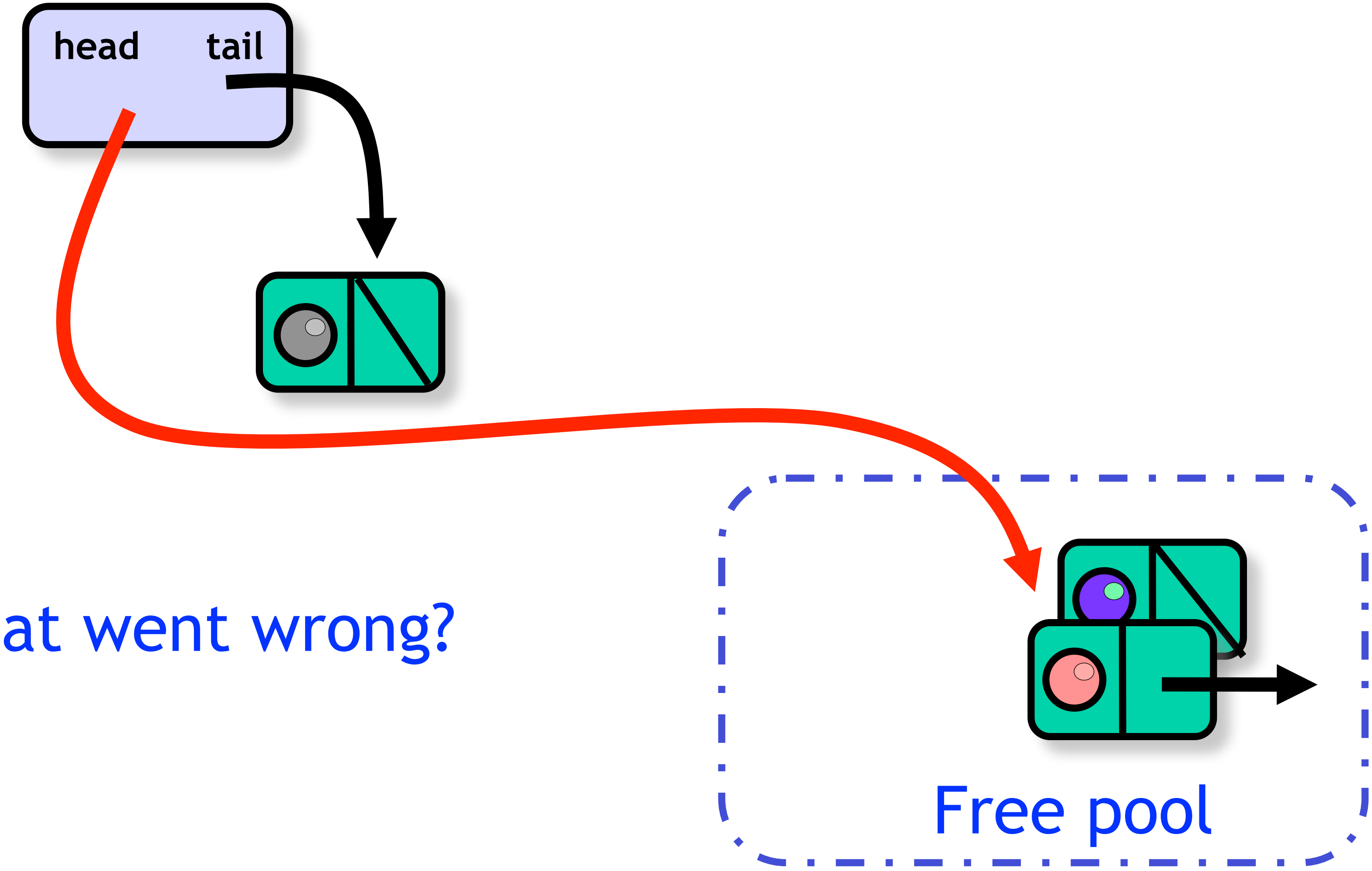
Why Recycling is Hard



Final State

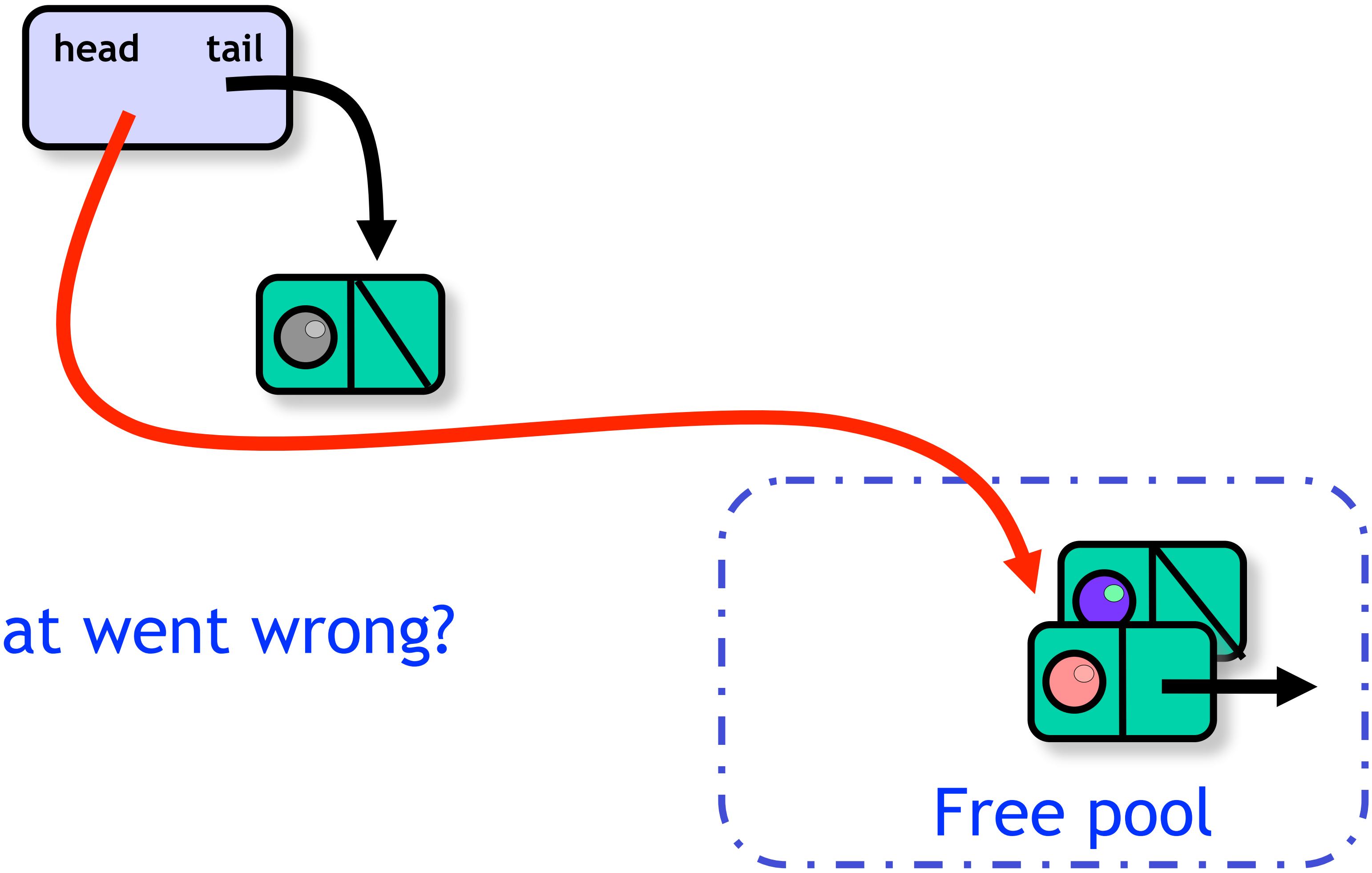


Final State



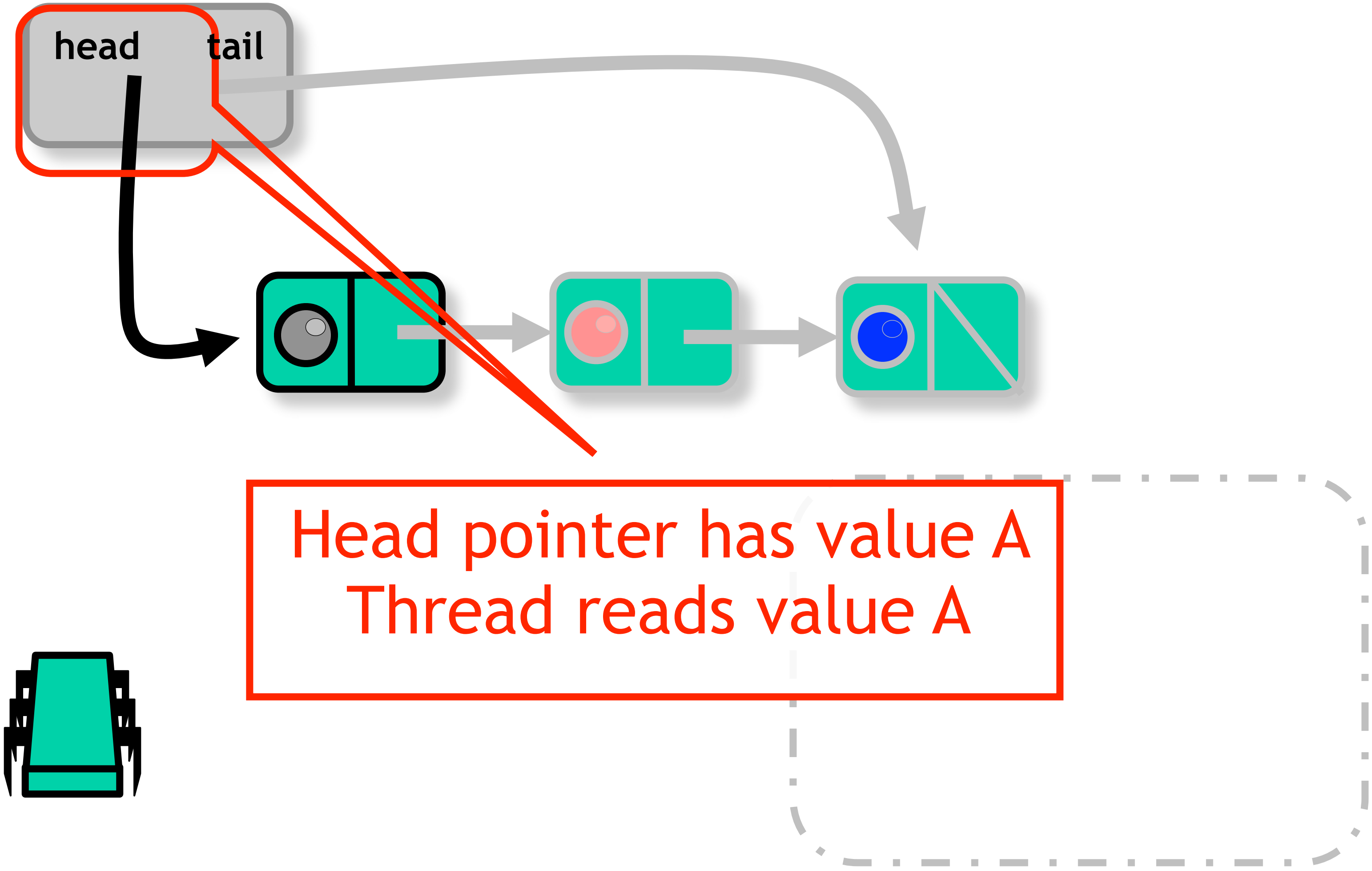
What went wrong?

Final State



What went wrong?

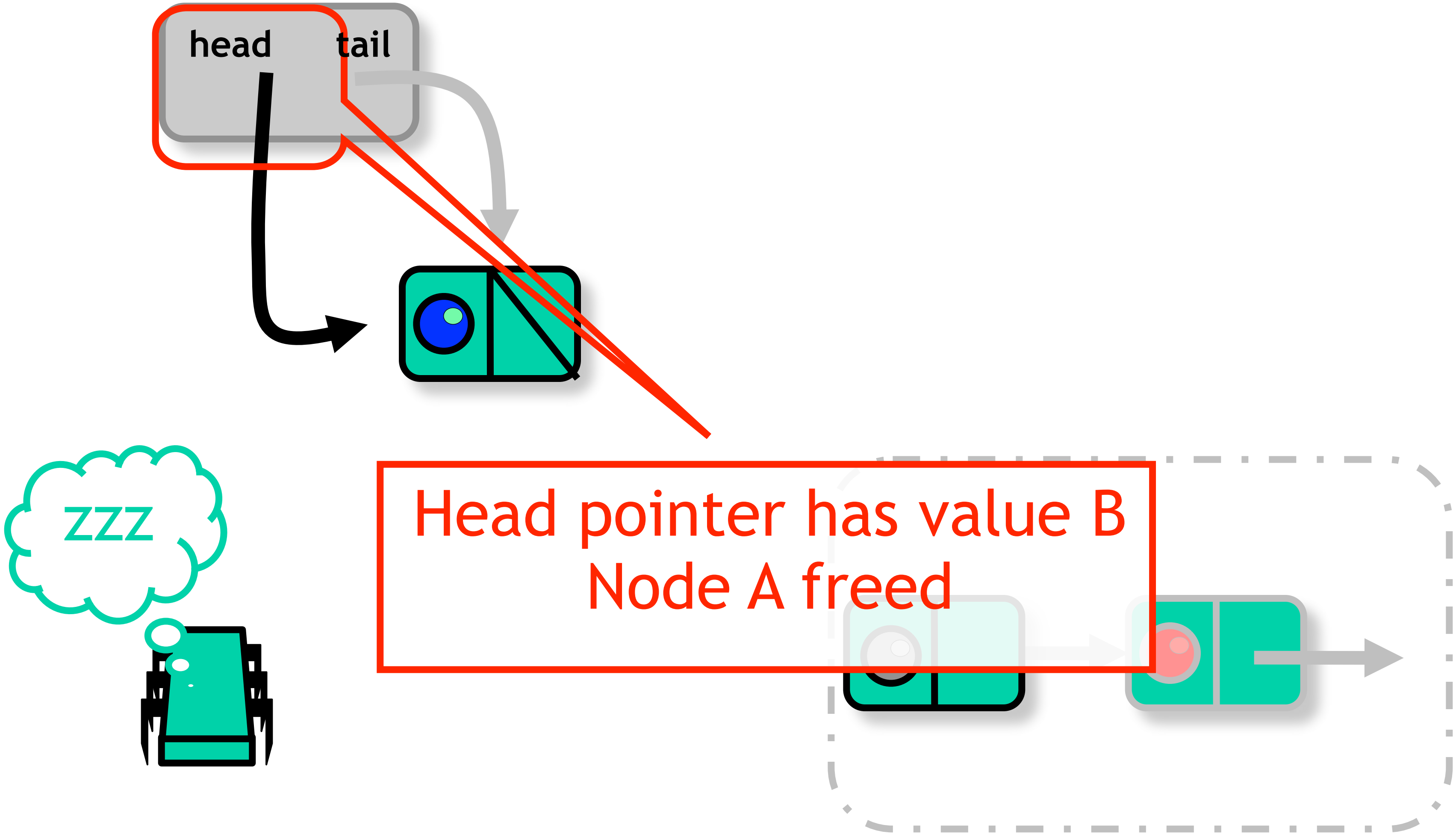
The Dreaded ABA Problem



Head pointer has value A
Thread reads value A

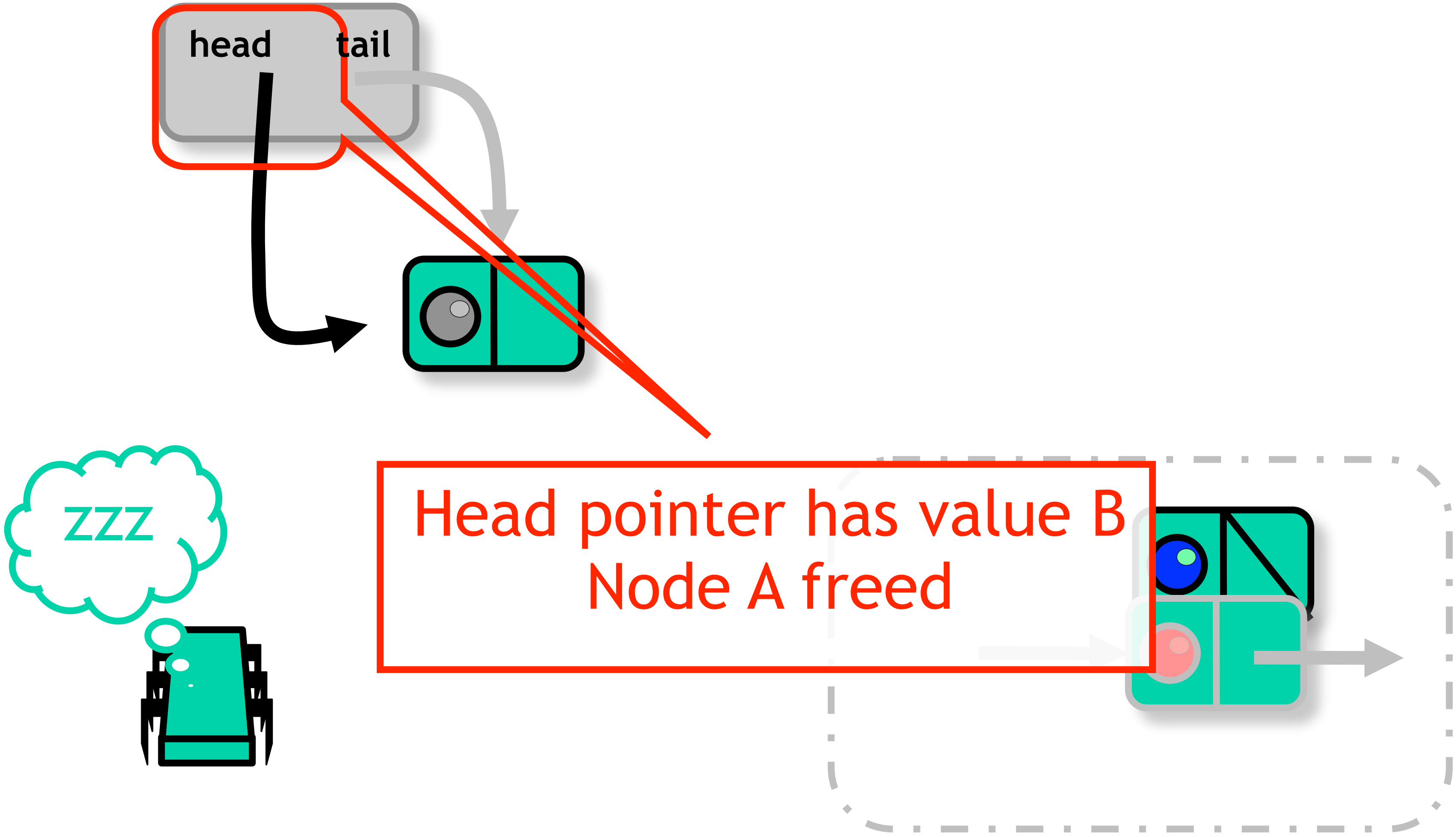
© Herlihy-Shavit

Dreaded ABA continued



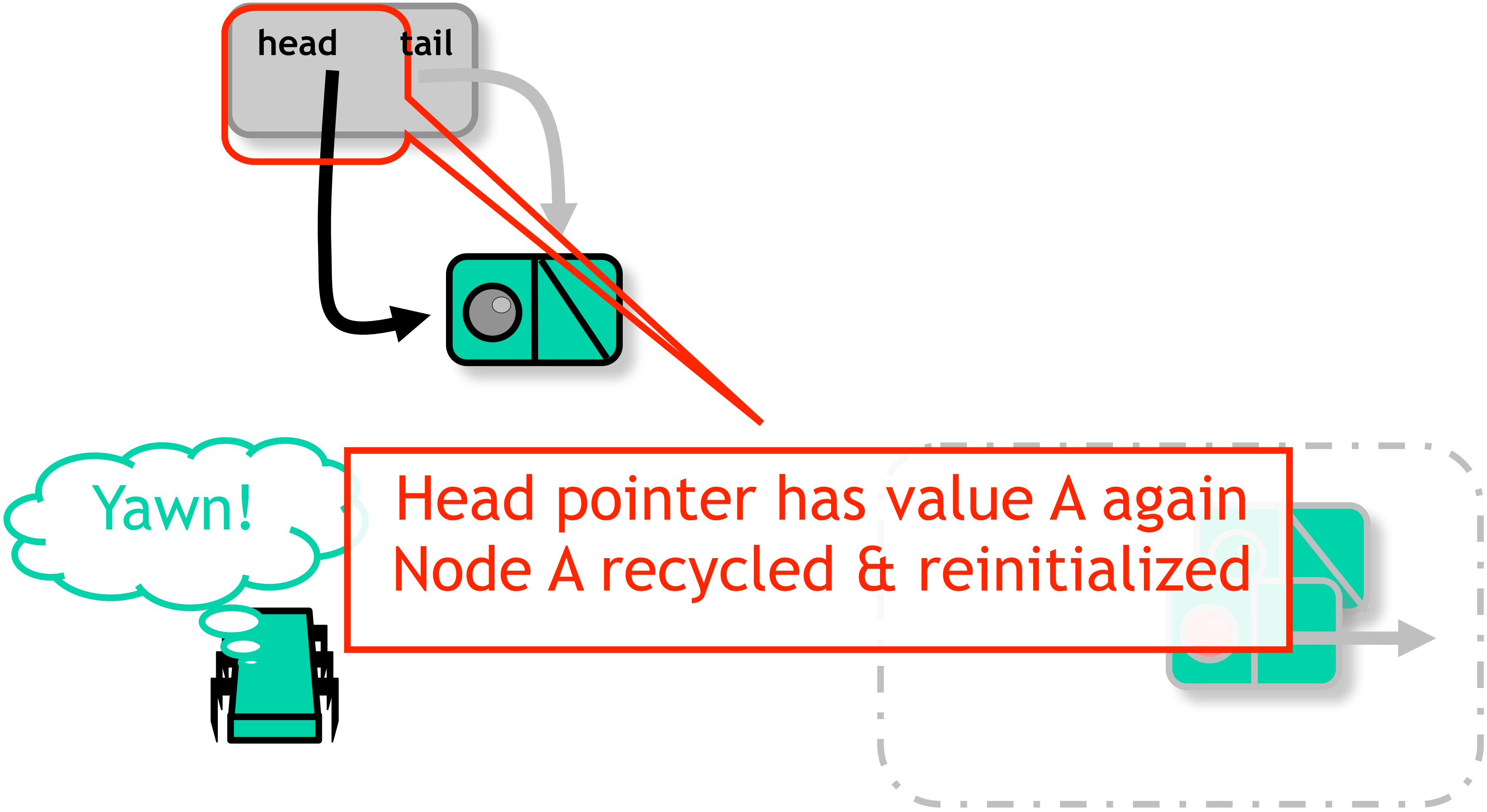
Head pointer has value B
Node A freed

Dreaded ABA continued



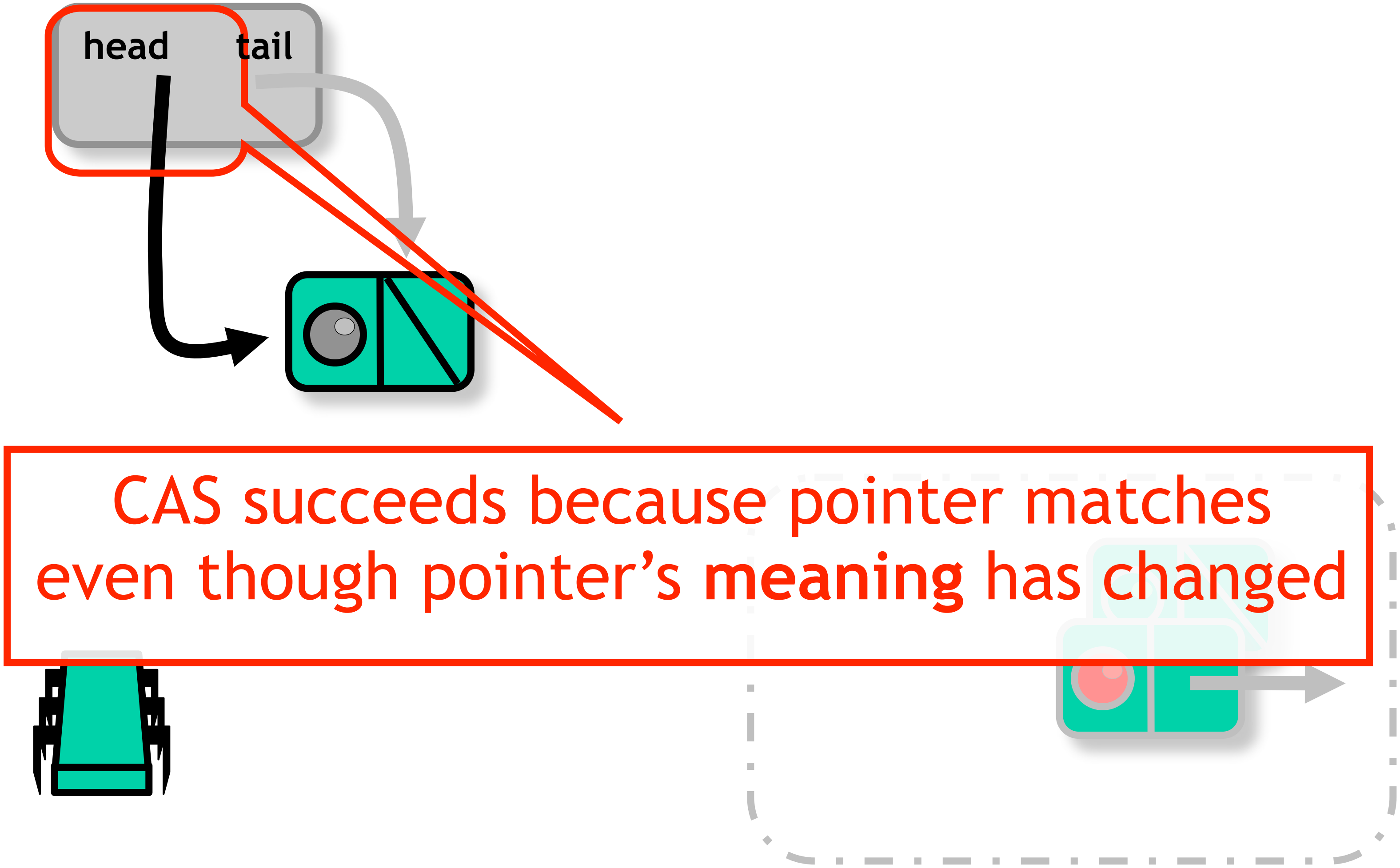
Head pointer has value B
Node A freed

Dreaded ABA continued



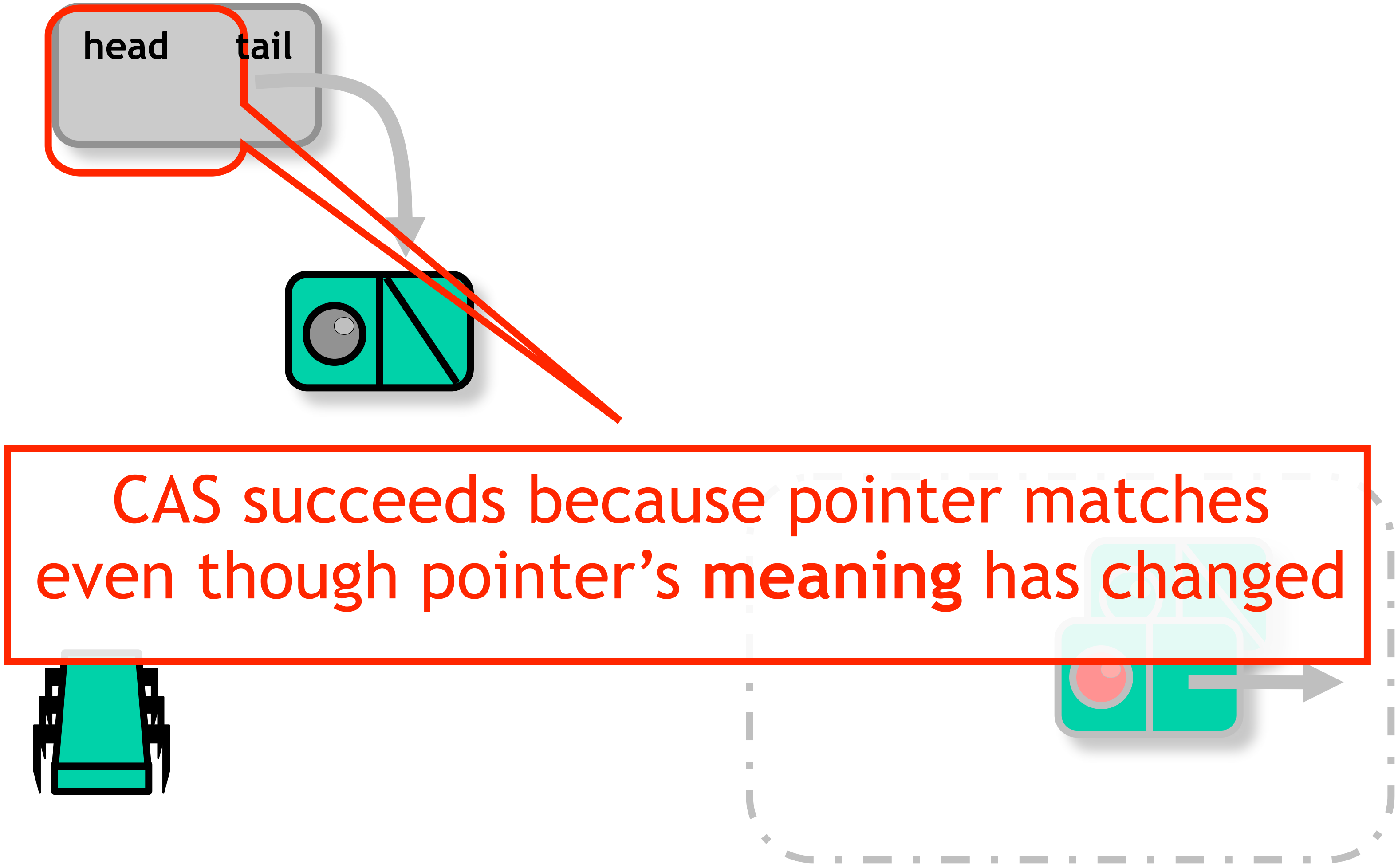
Head pointer has value A again
Node A recycled & reinitialized

Dreaded ABA continued



CAS succeeds because pointer matches even though pointer's **meaning** has changed

Dreaded ABA continued



CAS succeeds because pointer matches even though pointer's **meaning** has changed

The Dreaded ABA Problem

- Is a result of `CAS()` semantics (Sun, Intel, AMD)
- Does not arise with Load-Locked/Store-Conditional (IBM)
  - store conditional fails if memory location was updated since load-locked operation

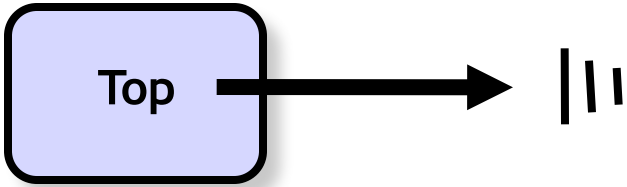
Dreaded ABA - A Solution

- Tag each pointer with a counter
- Unique over lifetime of node
- Pointer size vs word size issues
- Overflow?
  - Don’t worry be happy?
  - Bounded tags?
- AtomicStampedReference class

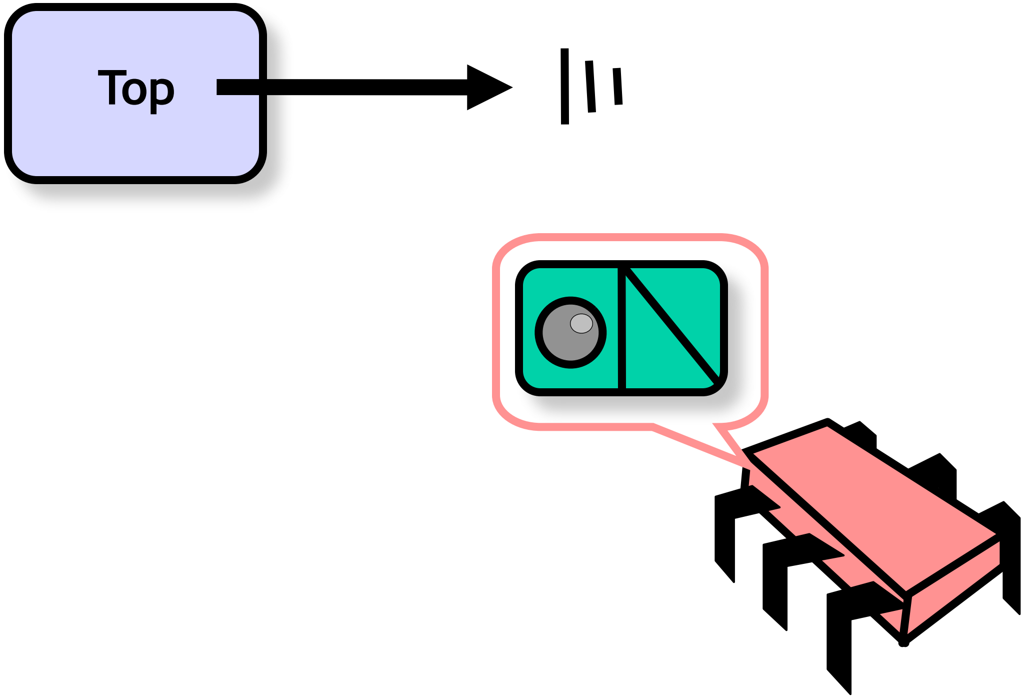
A Concurrent Stack

- Add() and Remove() of Stack are called `push()` and `pop()`
- A Stack is a pool with LIFO order on pushes and pops

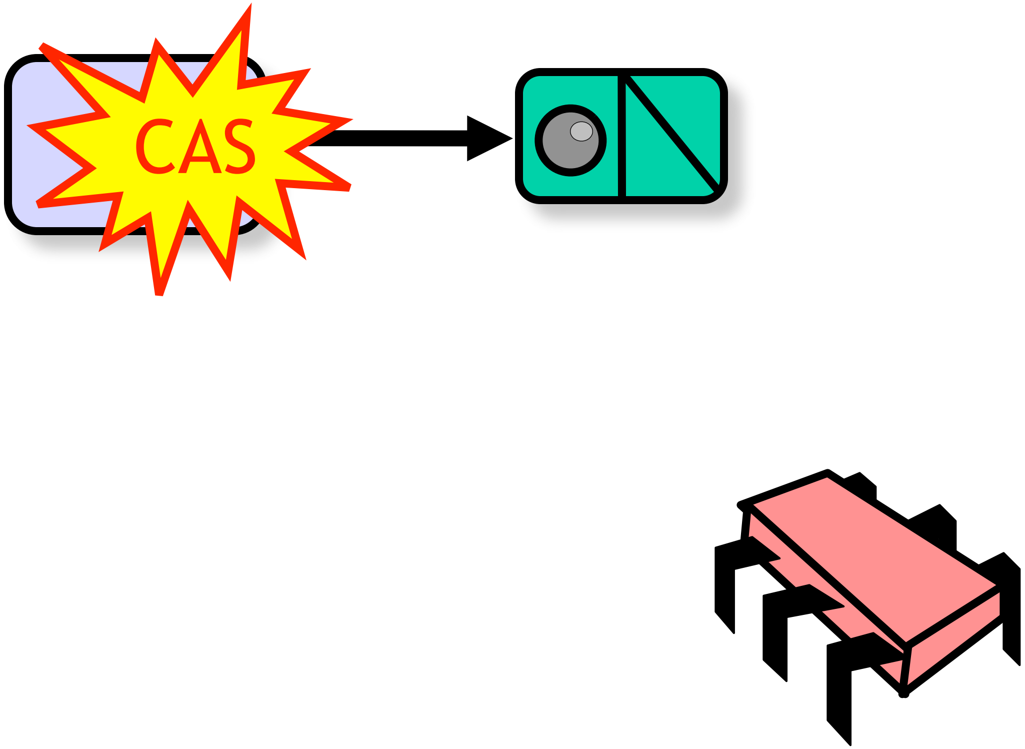
Unbounded Lock-free Stack



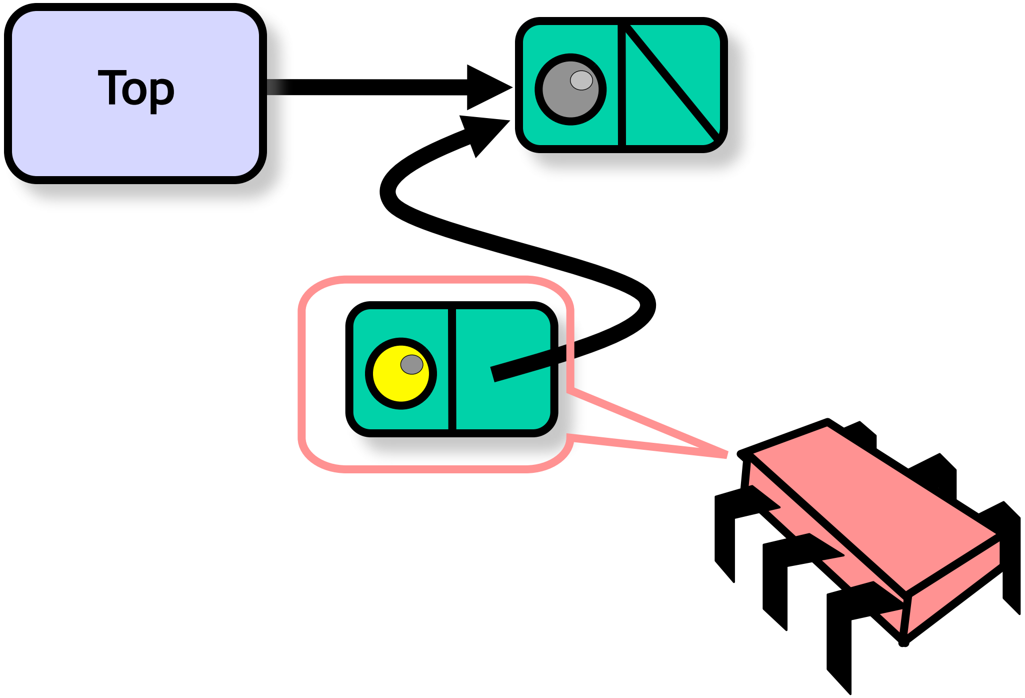
Unbounded Lock-free Stack



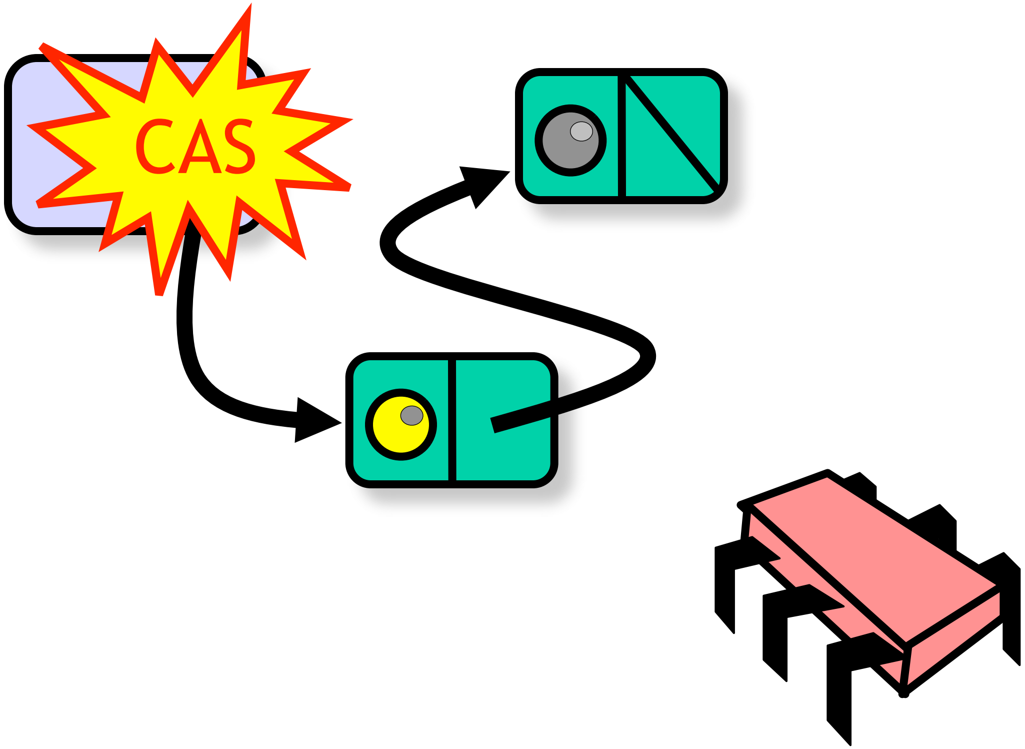
Push()



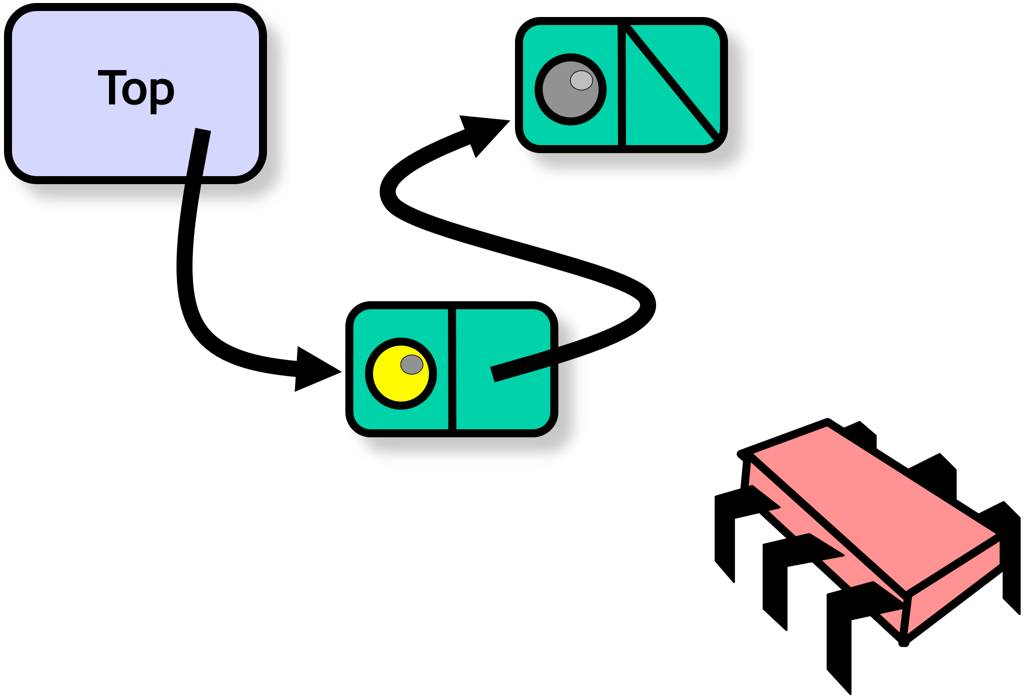
Push()



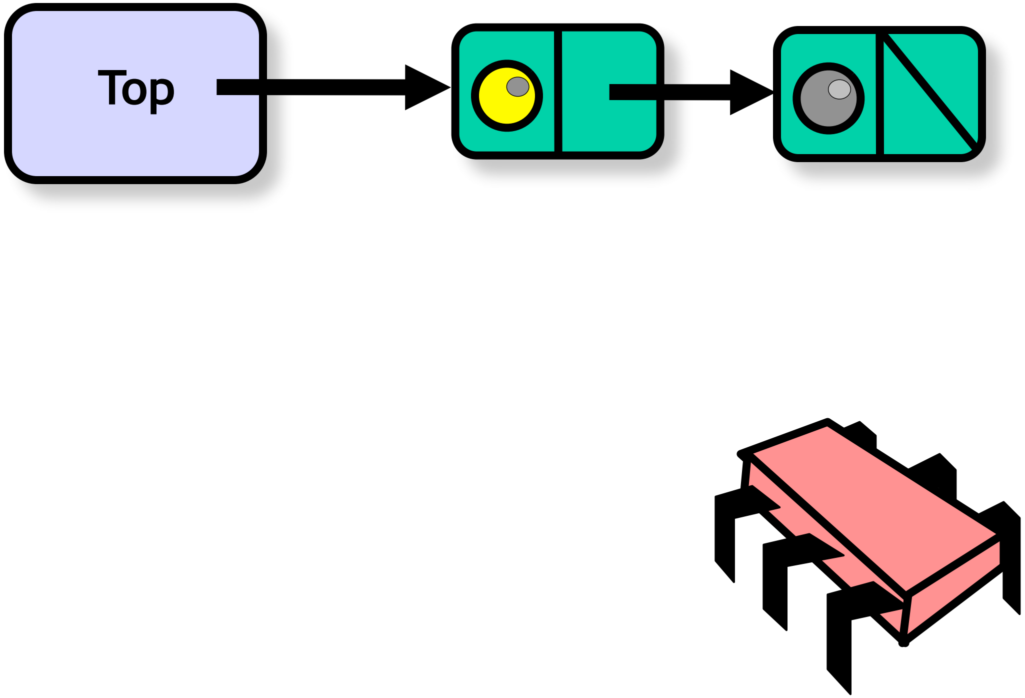
Push()



Push()

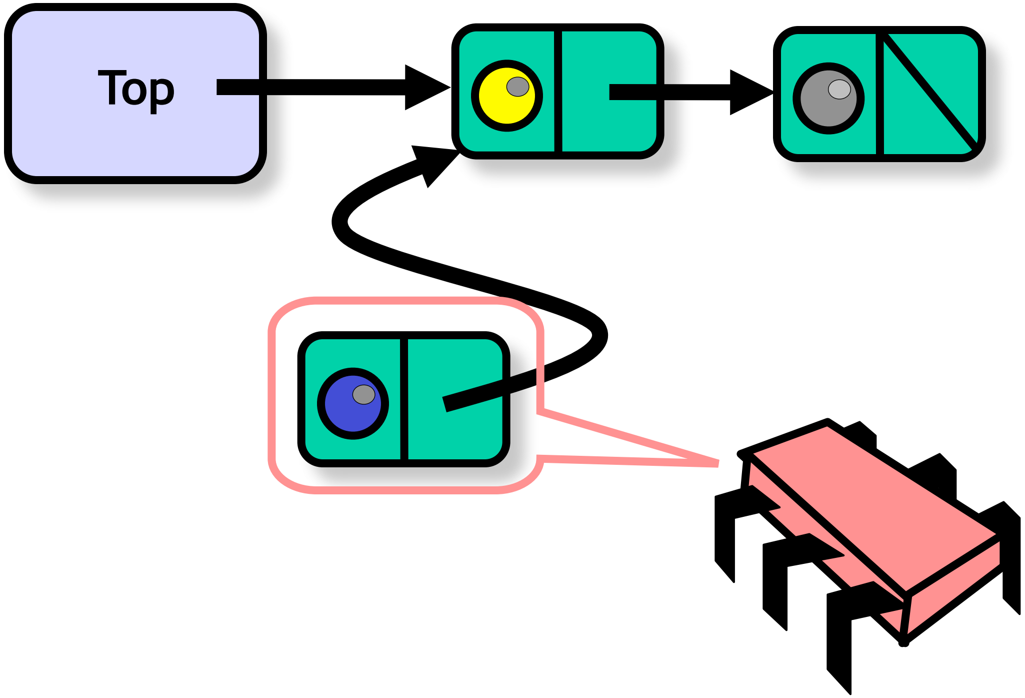


Push()

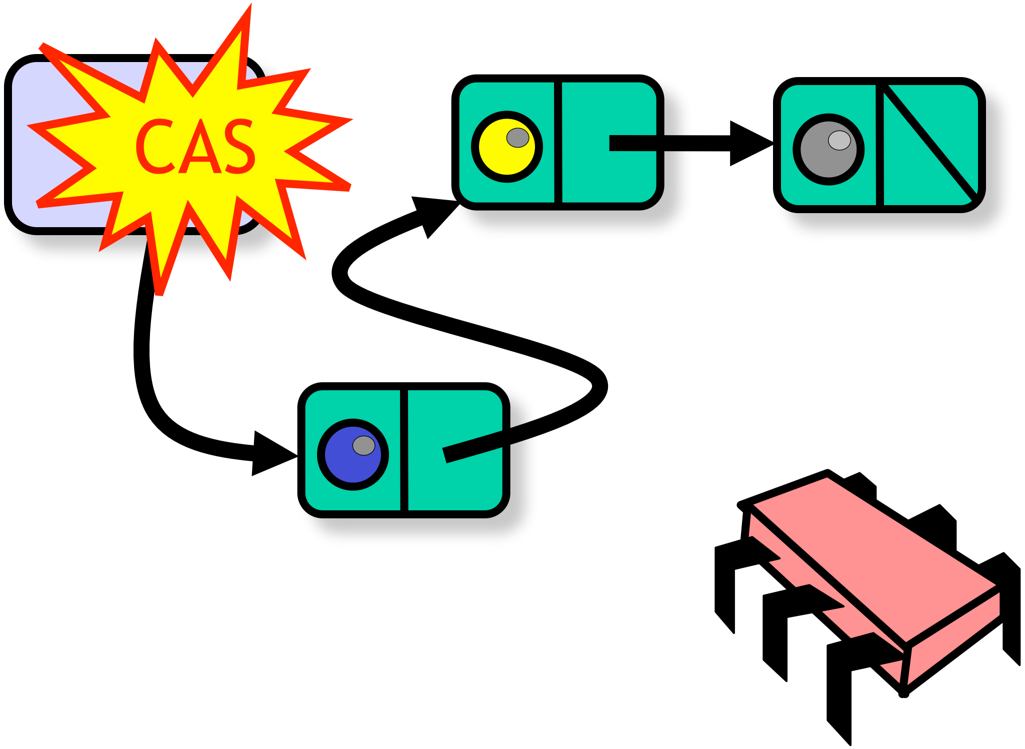


© Herlihy-Shavit

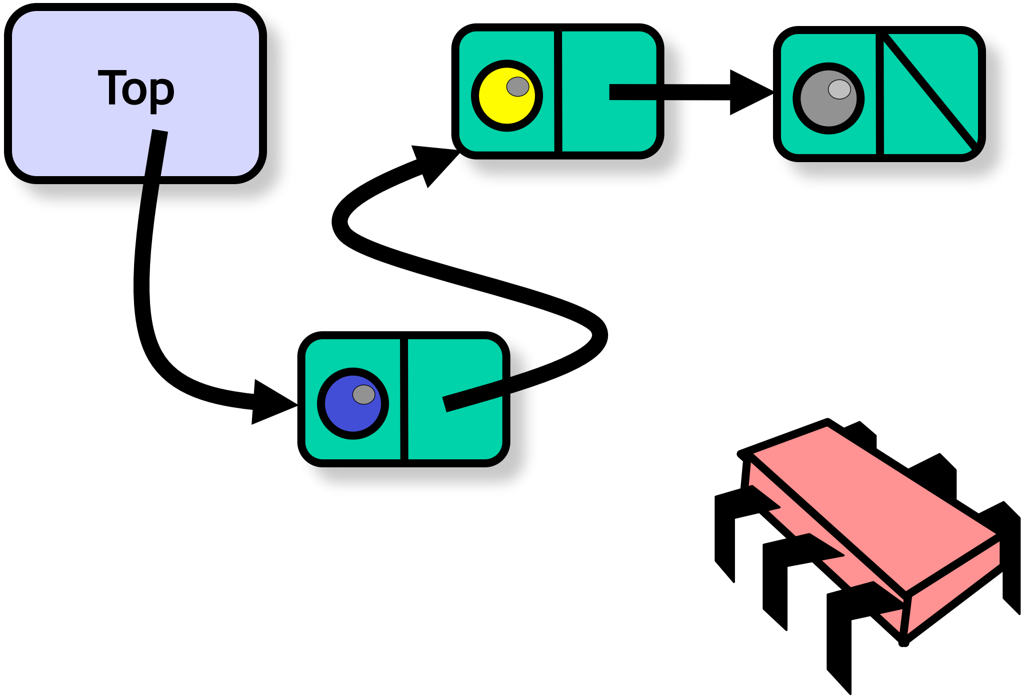
Push()



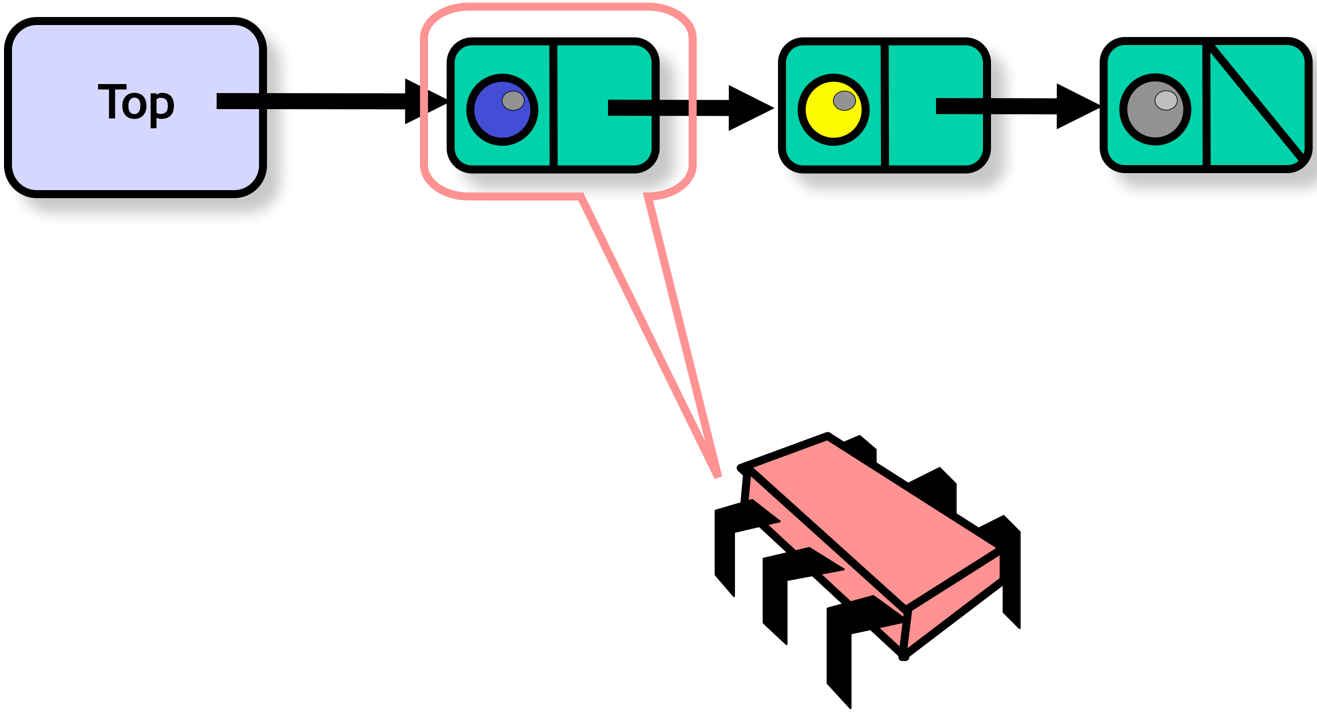
Push()



Push()

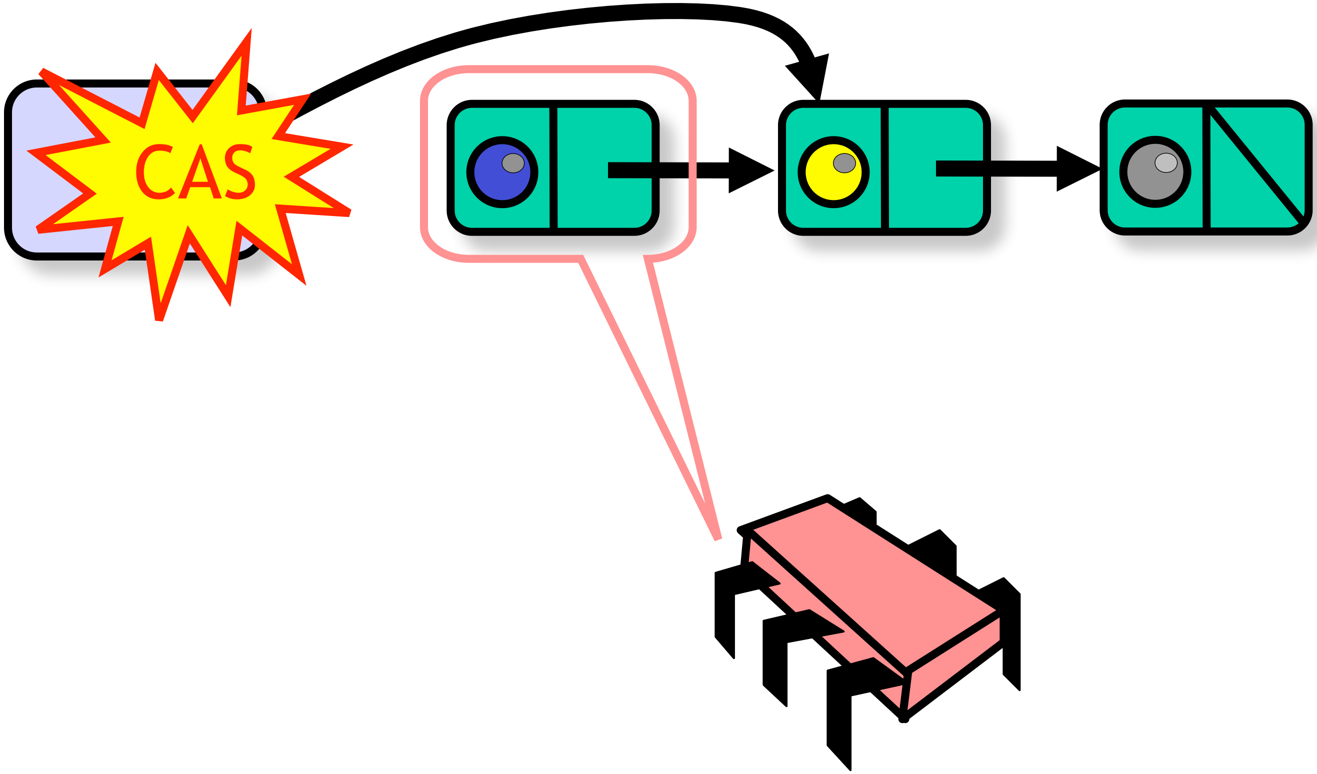


Pop()

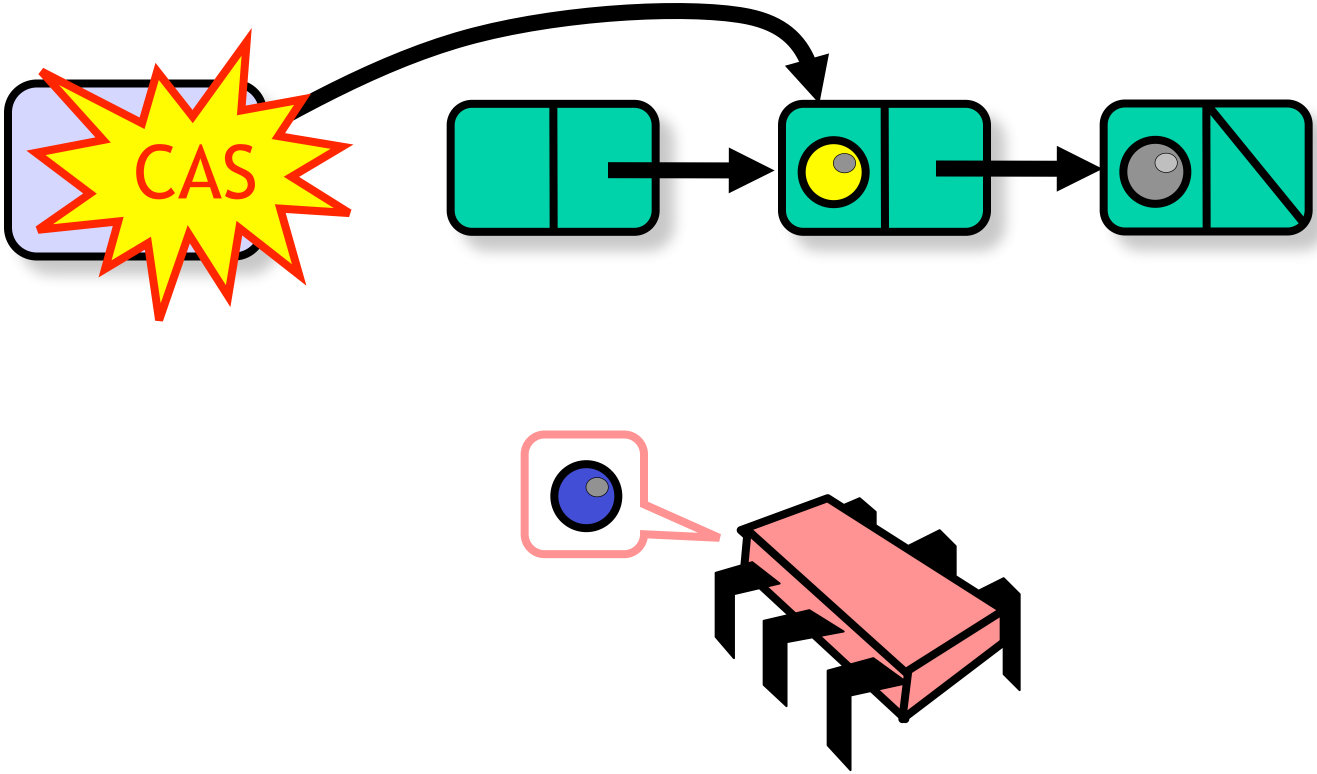


© Herlihy-Shavit

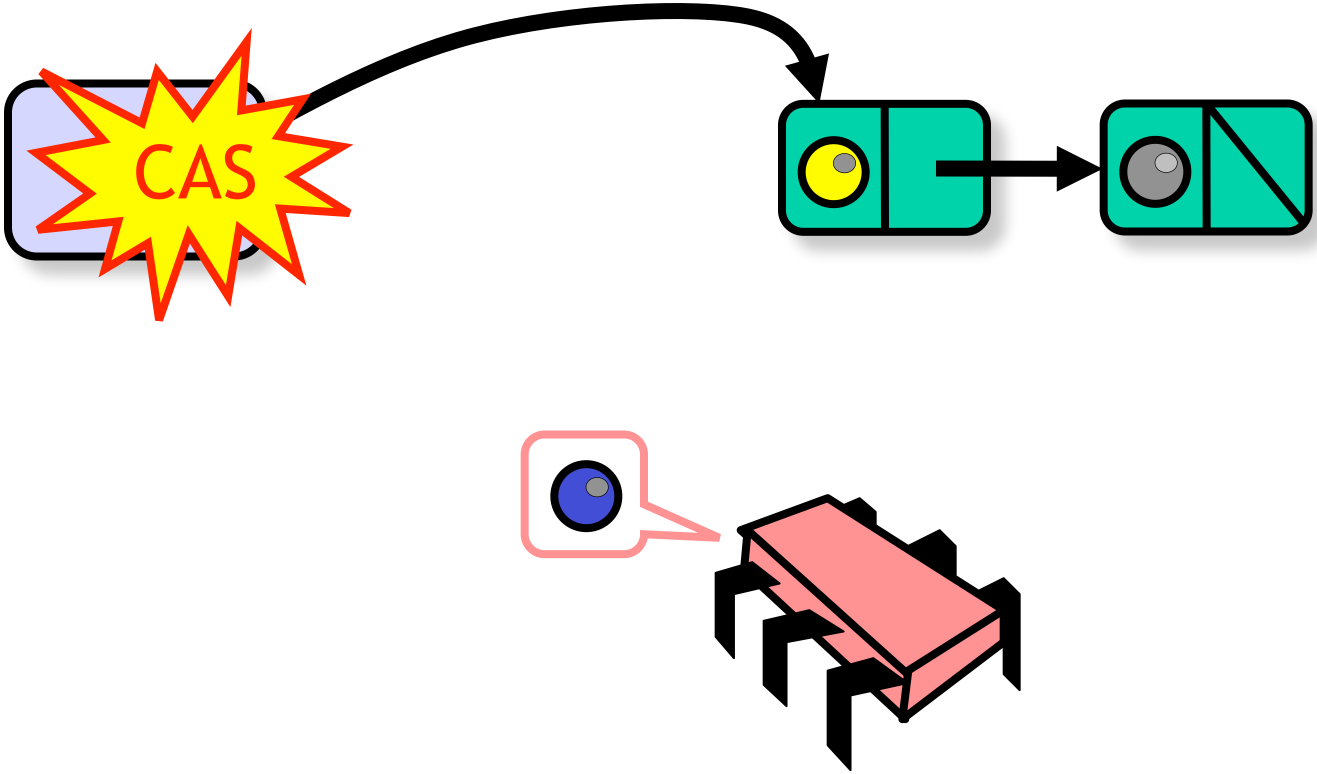
Pop()



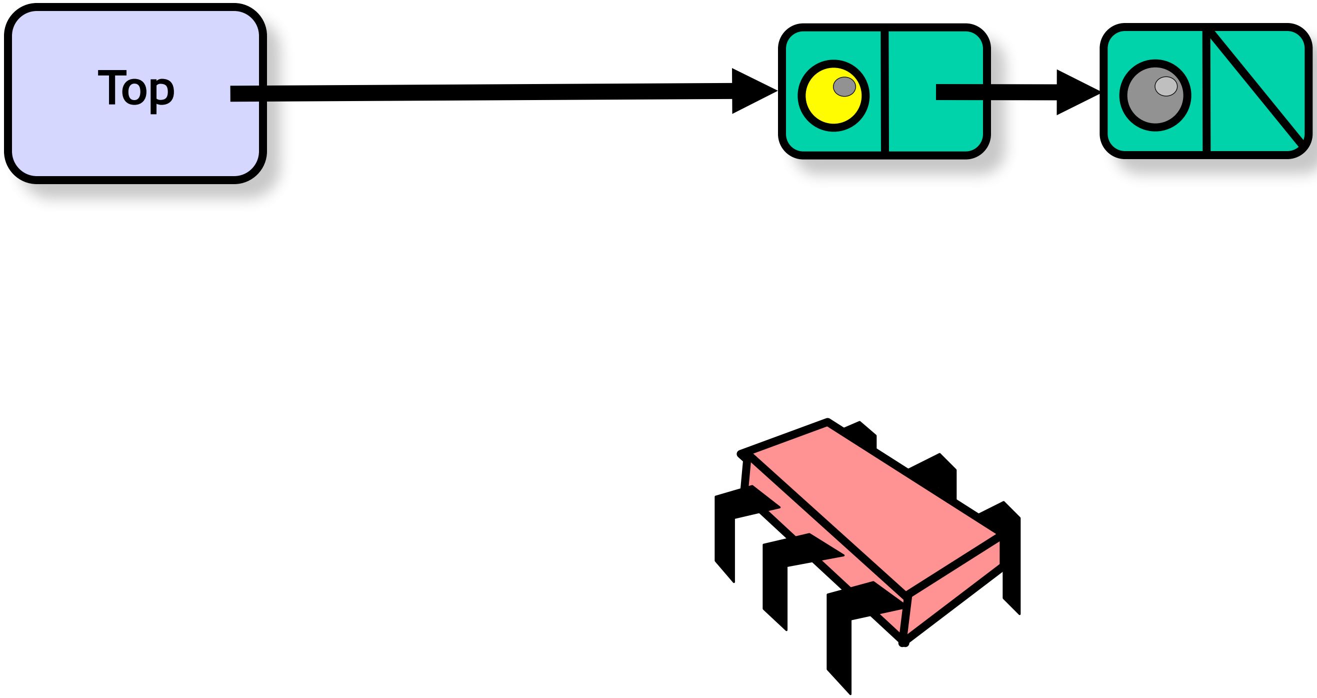
Pop()



Pop()



Pop()



Lock-free Stack

```
public class LockFreeStack {
  private AtomicReference top = new
    AtomicReference(null);

  public boolean tryPush(Node node){
    Node oldTop = top.get();
    node.next = oldTop;
    return(top.compareAndSet(oldTop, node))
  }
  public void push(T value) {
       Node node = new Node(value);
    while (true) {
      if (tryPush(node)) {
        return;
      } else
        backoff.backoff()
    }
```

Lock-free Stack

```
public class LockFreeStack {
 private AtomicReference top = new
  AtomicReference(null);

 public boolean tryPush(Node node){
  Node oldTop = top.get();
  node.next = oldTop;
  return(top.compareAndSet(oldTop, node))
 }
 public void push(T value) {
    Node node = new Node(value);
  while (true) {
   if (tryPush(node)) {
    return;
   } else
    backoff.backoff()
  }
```

Push uses `tryPush()` method

Lock-free Stack

```
public class LockFreeStack {
 private AtomicReference top = new
  AtomicReference(null);

 public boolean tryPush(Node node){
  Node oldTop = top.get();
  node.next = oldTop;
  return(top.compareAndSet(oldTop, node))
 }
 public void push(T value) {
  Node node = new Node(value);
  while (true) {
   if (tryPush(node)) {
    return;
   } else
    backoff.backoff()
  }
```

Create a new node

Lock-free Stack

```
public class LockFreeStack {
  private AtomicReference top = new
    AtomicReference(null);

  public boolean tryPush(Node node){
    Node oldTop = top.get();
    node.next = oldTop;
    return(top.compareAndSet(oldTop, node))
  }
  public void push(T value) {
       Node node = new Node(value);
    while (true) {
      if (tryPush(node)) {
        return;
      } else
        backoff.backoff()
    }
```

Then try to push: if `tryPush()` fails back-off before retrying

Lock-free Stack

```
public class LockFreeStack {
  private AtomicReference top = new
    AtomicReference(null);

  public boolean tryPush(Node node){
    Node oldTop = top.get();
    node.next = oldTop;
    return(top.compareAndSet(oldTop, node))
  }
  public void push(T value) {
      Node node = new Node(value);
    while (true) {
      if (tryPush(node)) {

      } else
       backoff.backoff()
    }
```

tryPush() attempts to push a node at top

Lock-free Stack

```
public class LockFreeStack {
 private AtomicReference top = new
  AtomicReference(null);

 public boolean tryPush(Node node){
  Node oldTop = top.get();
  node.next = oldTop;
  return(top.compareAndSet(oldTop, node))
 }
 public void push(T value) {
    Node node = new Node(value);
  while (true) {
   if (tryPush(node)) {
    return;
   } else
    backoff.backoff()
  }
```

Read top value

Lock-free Stack

```
public class LockFreeStack {
 private AtomicReference top = new
   AtomicReference(null);

 public boolean tryPush(Node node){
   Node oldTop = top.get();
   node.next = oldTop;
   return(top.compareAndSet(oldTop, node))
 }
 public void push(T value) {
     Node node = new Node(value);
   while (true) {
     if (tryPush(node)) {
       return;
     } else
       backoff.backoff()
   }
```

current top will be new node’s successor

Lock-free Stack

```
public class LockFreeStack {
  private AtomicReference top = new
   AtomicReference(null);

  public boolean tryPush(Node node){
   Node oldTop = top.get();
   node.next = oldTop;
   return(top.compareAndSet(oldTop, node))
  }
  public void push(T value) {
     Node node = new Node(value);
   while (true) {
    if (tryPush(node)) {
     return;
    } else
     backoff.backoff()
  }
```

Try to swing top to point at my new node

Lock-free Stack

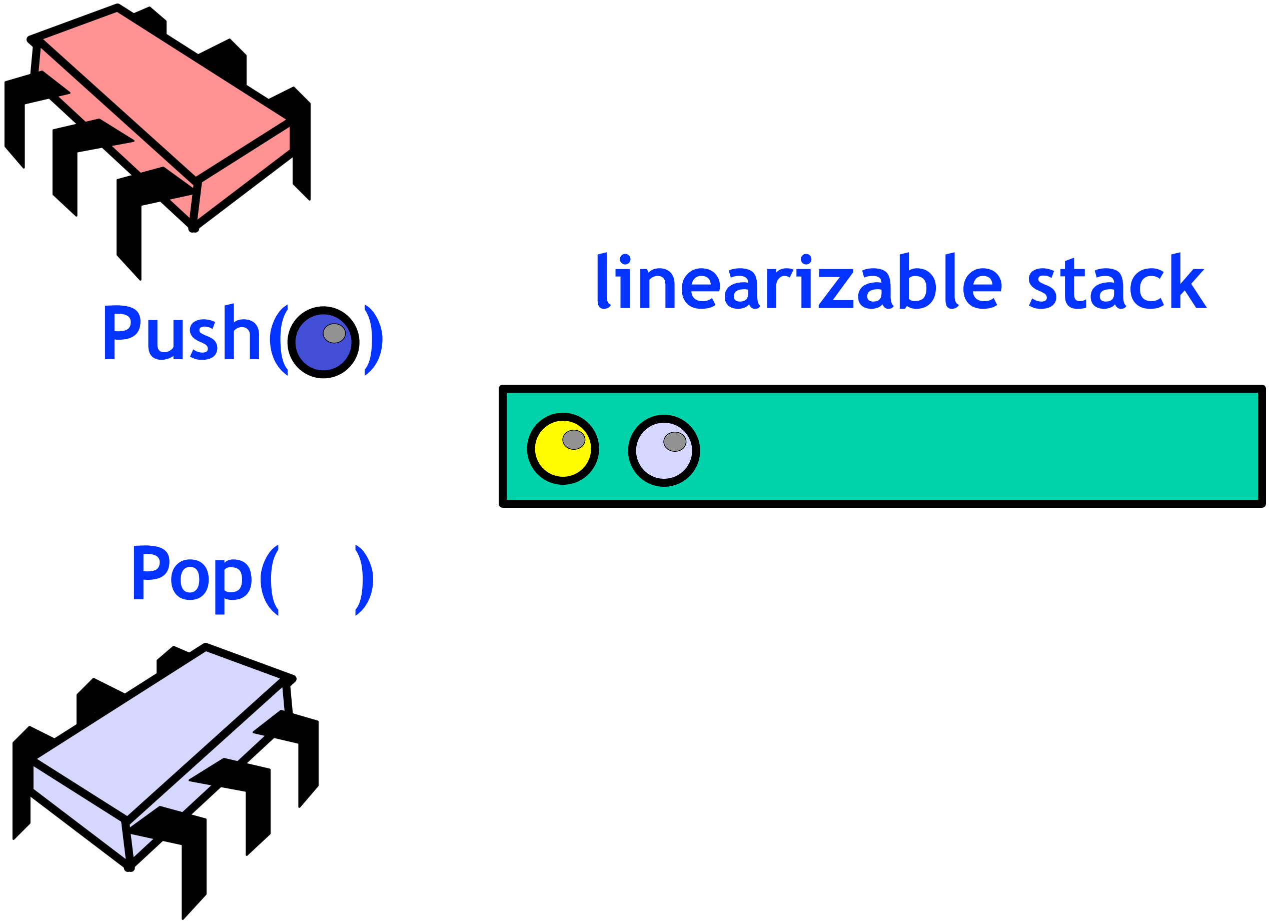
- Good: No locking
- Bad: if no GC then ABA as in queue (add time stamps)
- Bad: Contention on top (add backoff)
- Bad: No parallelism

- Is a stack inherently sequential?

Elimination-Backoff Stack

- How to “turn contention into parallelism”
- Replace regular exponential-backoff
- with an alternative elimination-backoff mechanism

Observation

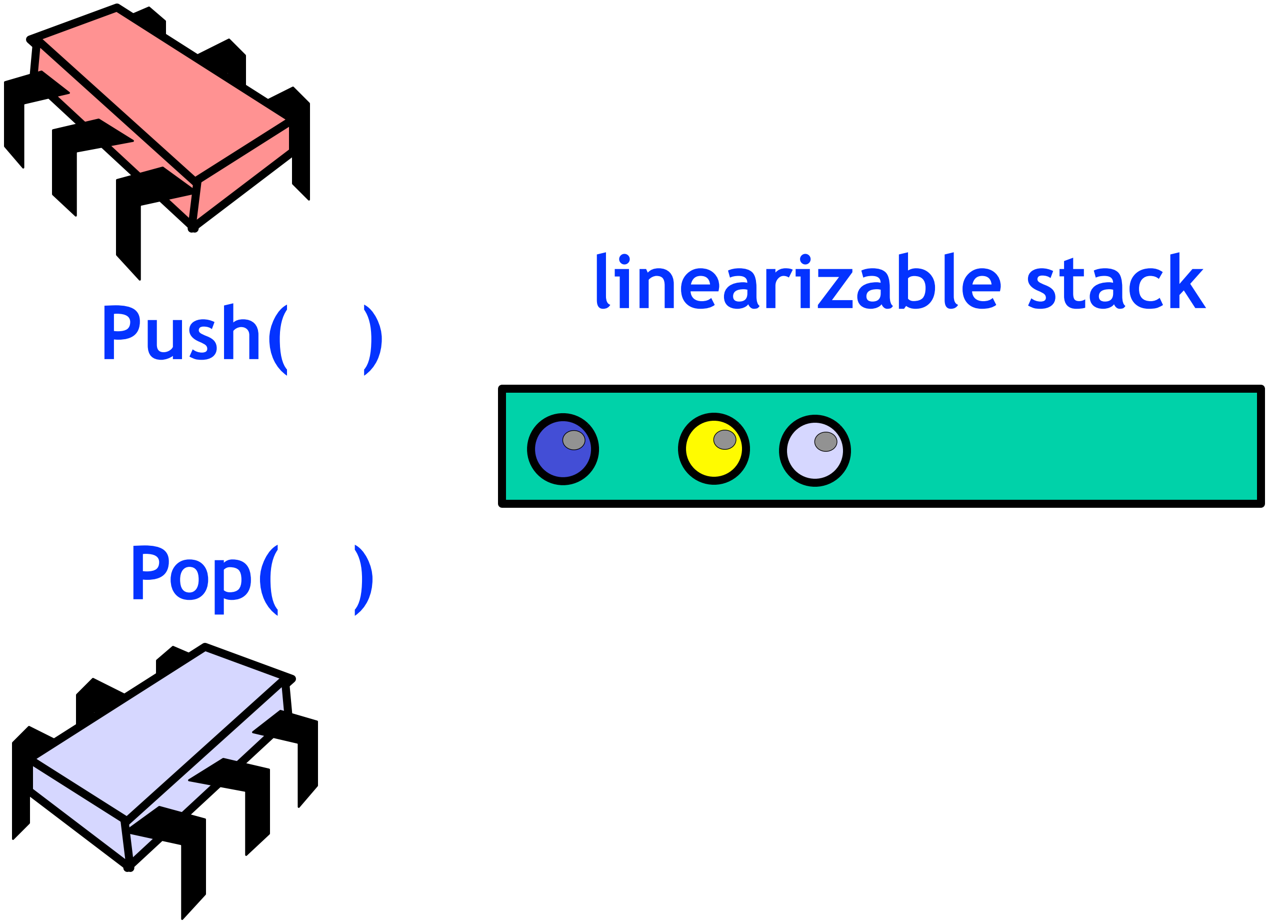


Push(●)

linearizable stack

Pop( )

Observation

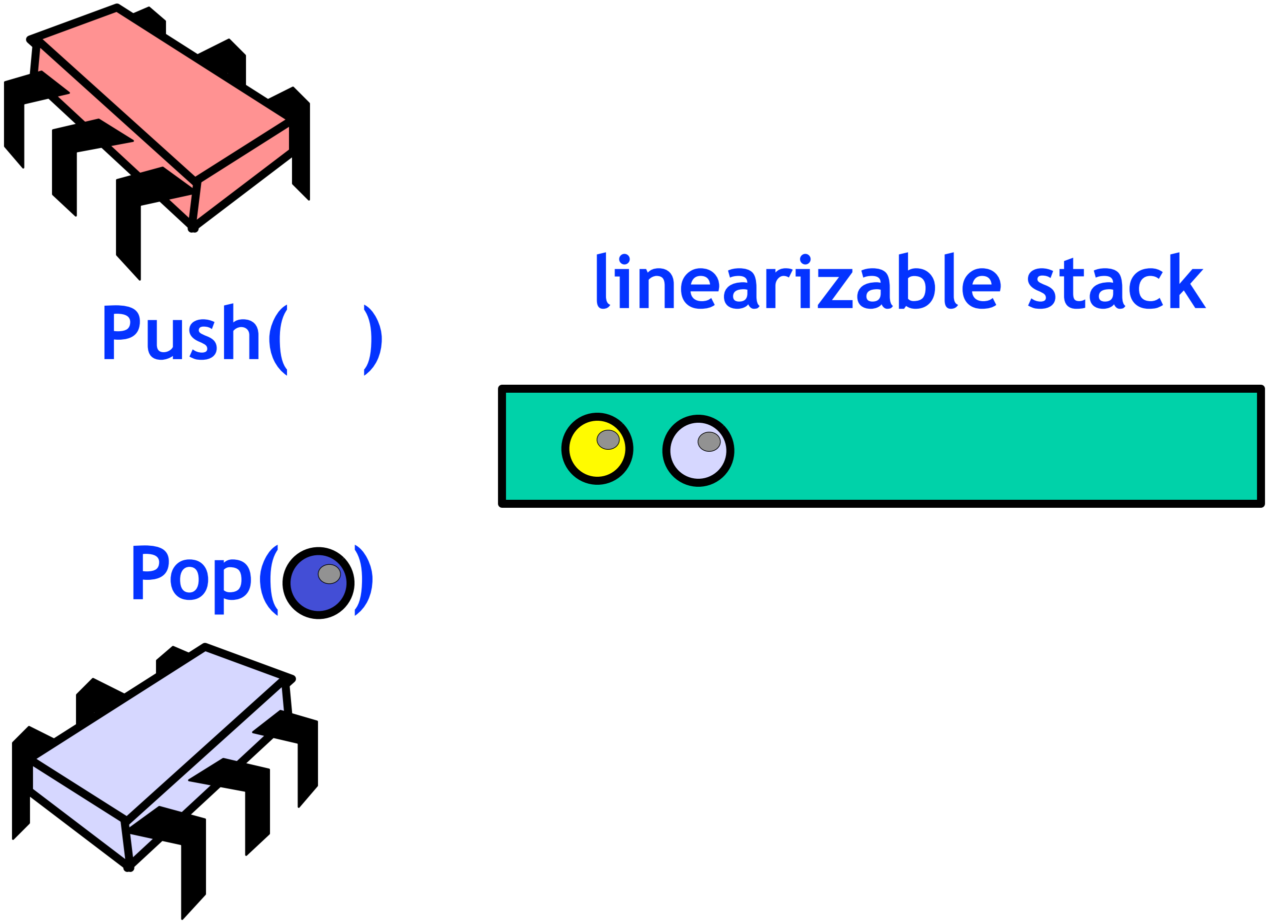


Push(   )

linearizable stack

Pop(   )

Observation

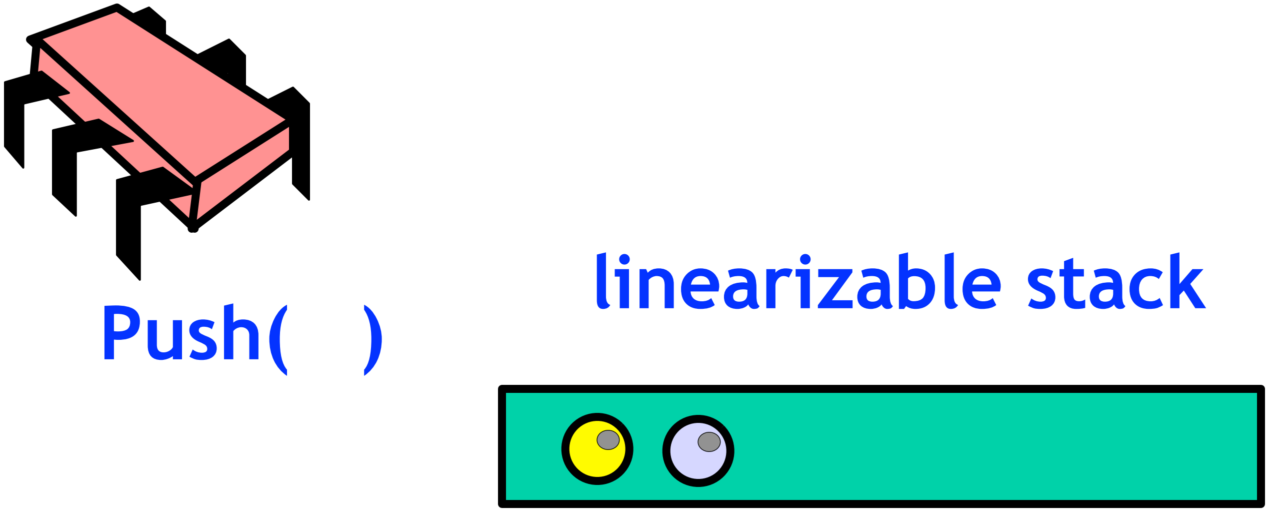


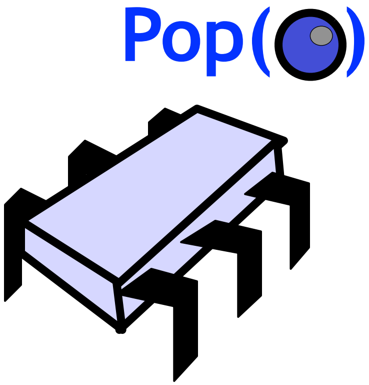
linearizable stack

Push( )

Pop(●)

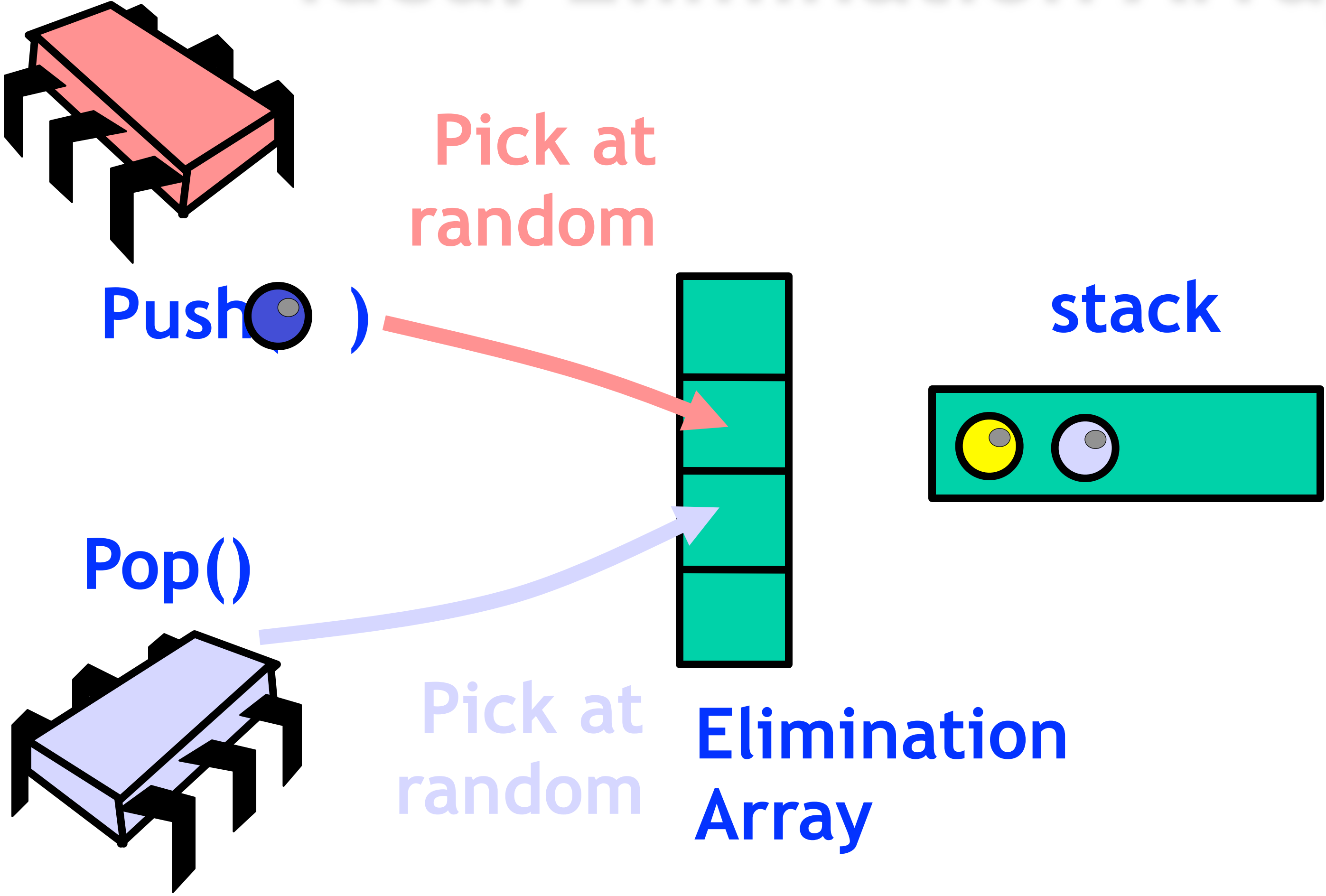
Observation





After any equal number of pushes and pops, stack stays the same

Idea: Elimination Array



Pick at random

Push( )

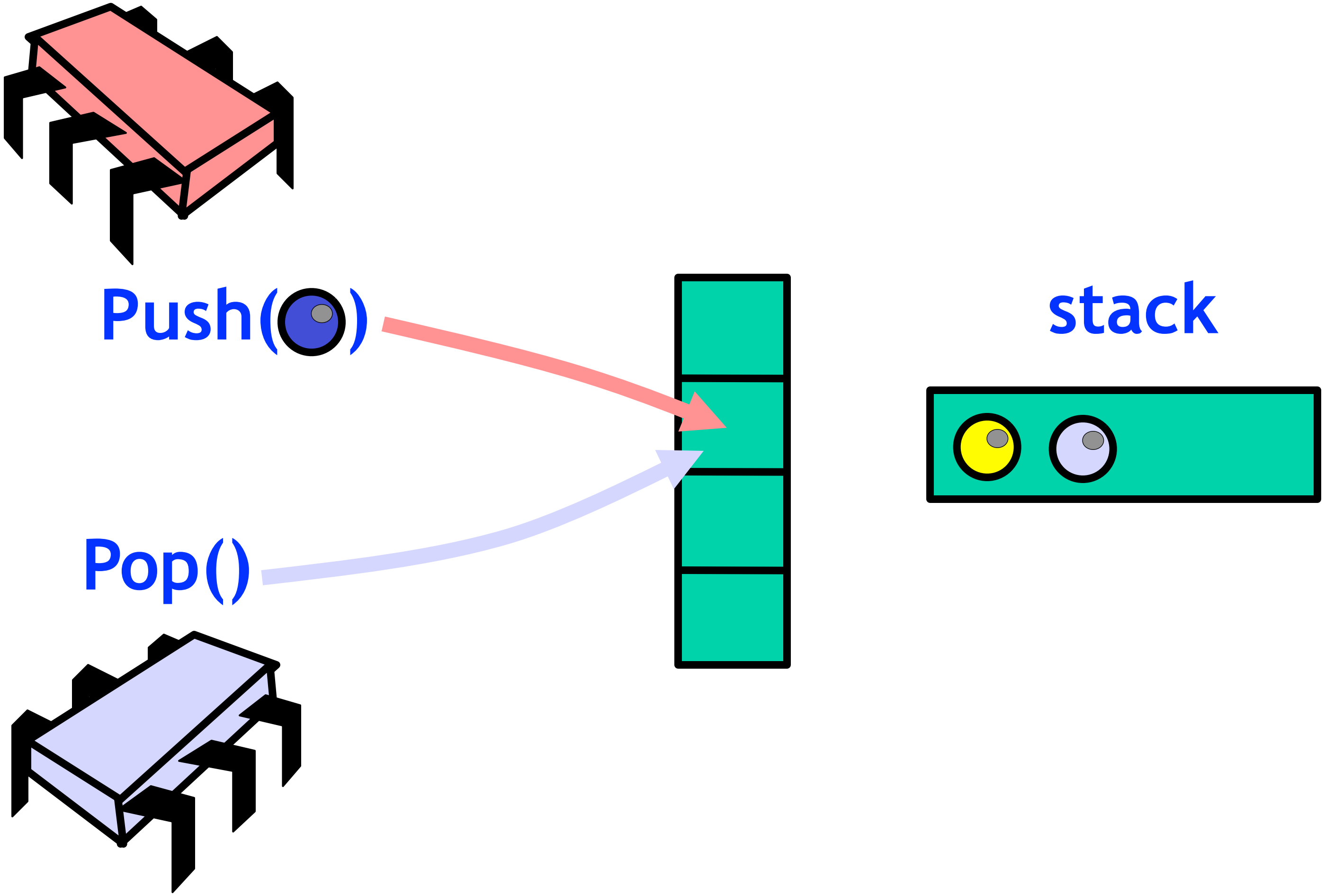
stack

Pop()

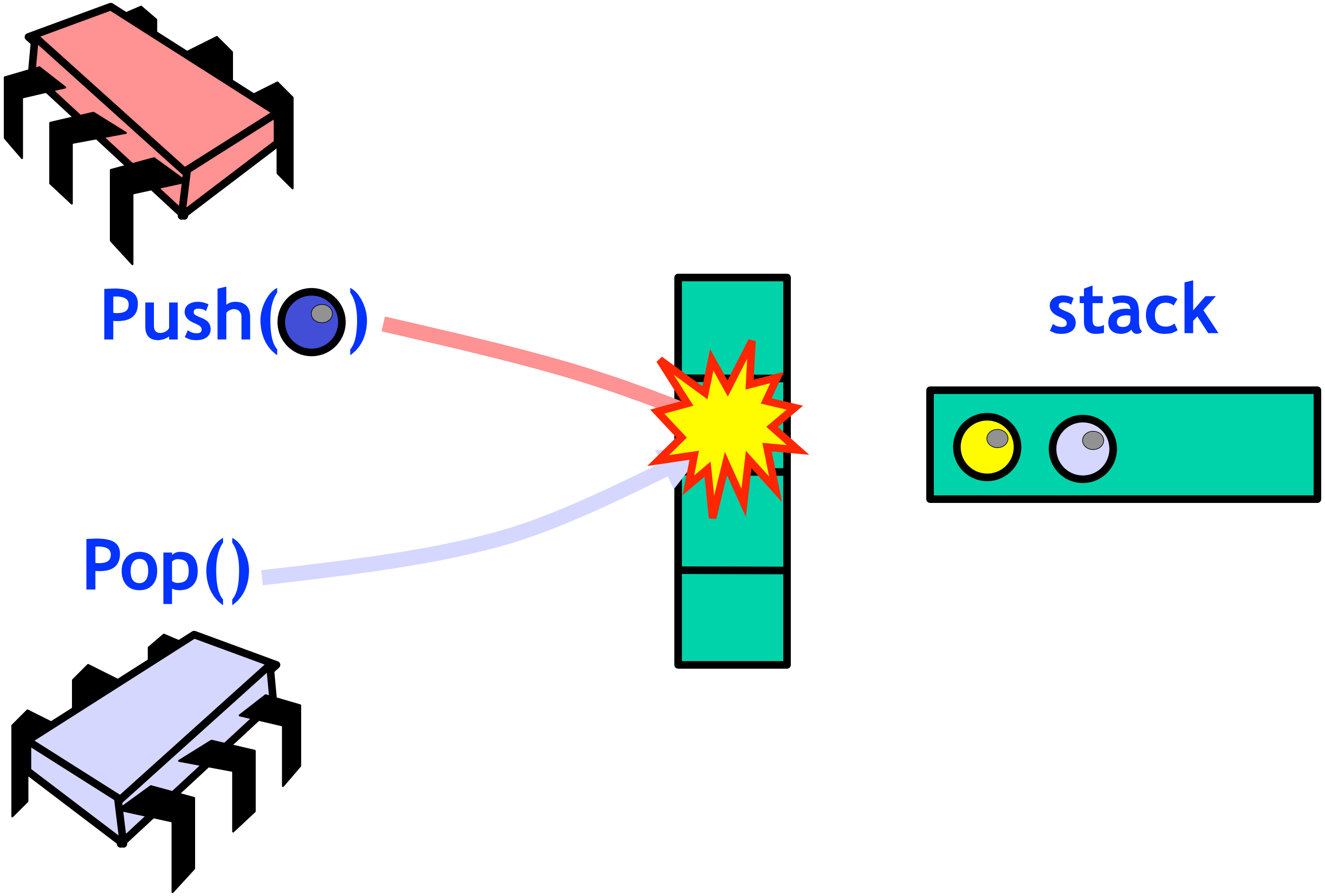
Pick at random

Elimination Array

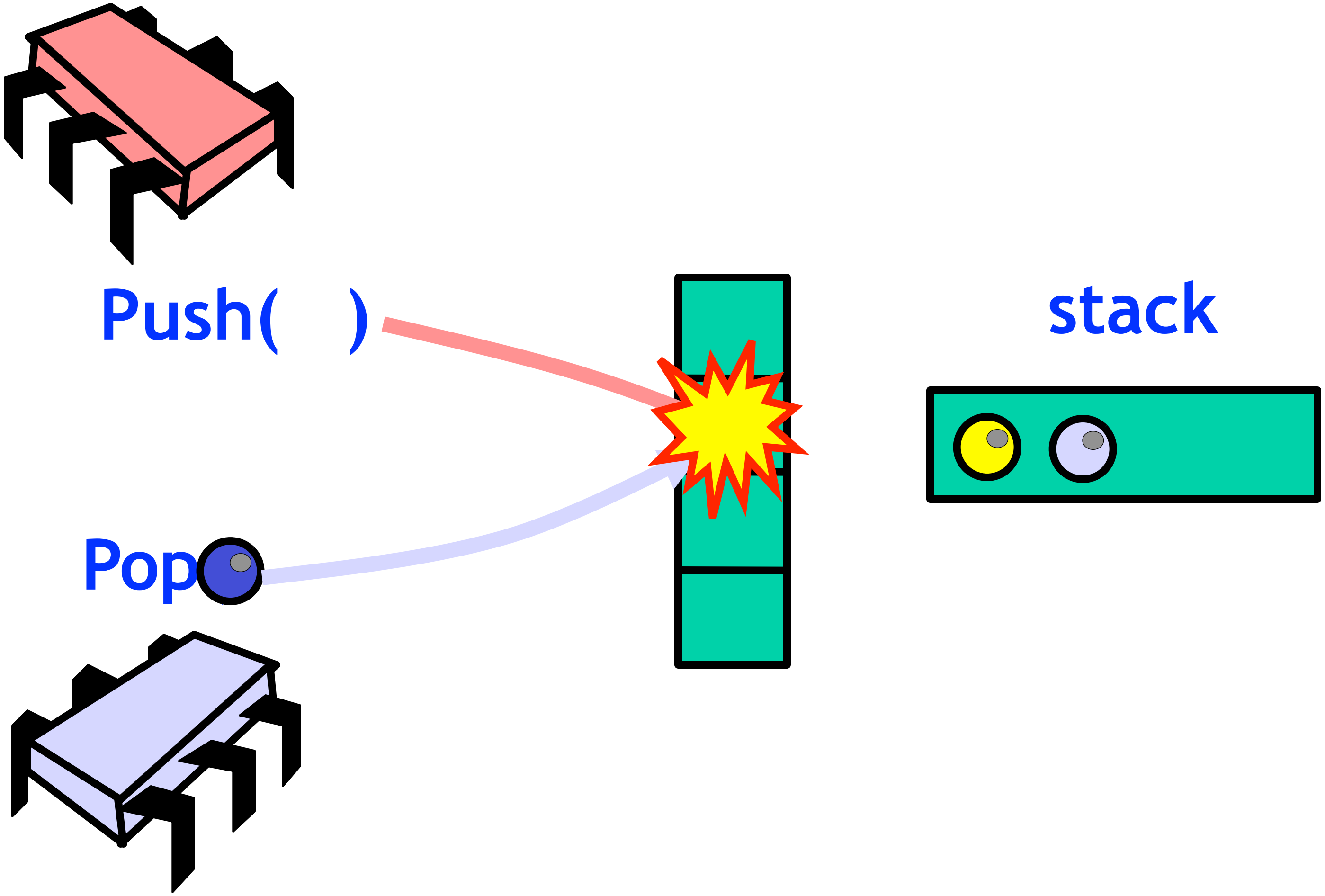
Push Collides With Pop



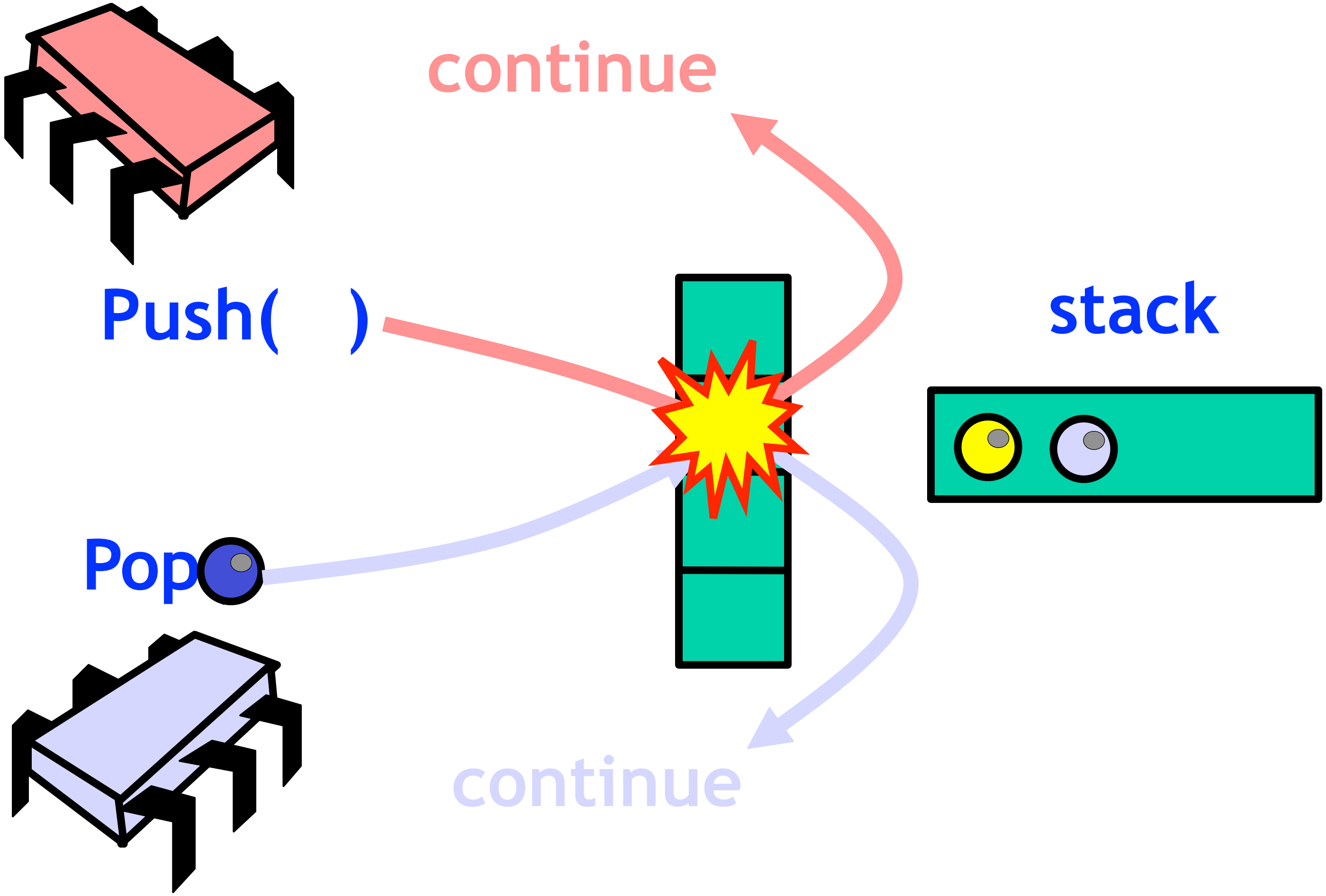
Push Collides With Pop



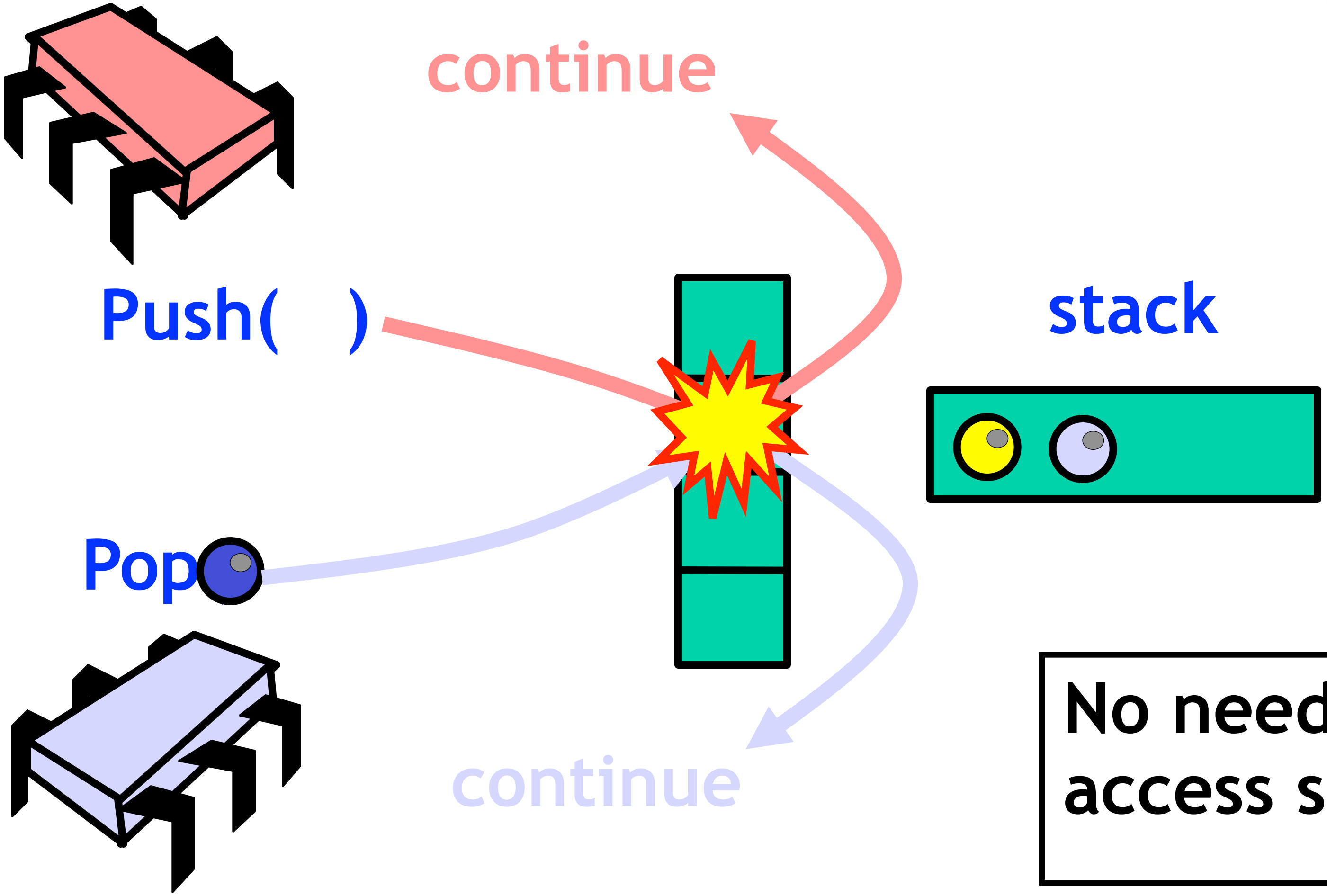
Push Collides With Pop



Push Collides With Pop

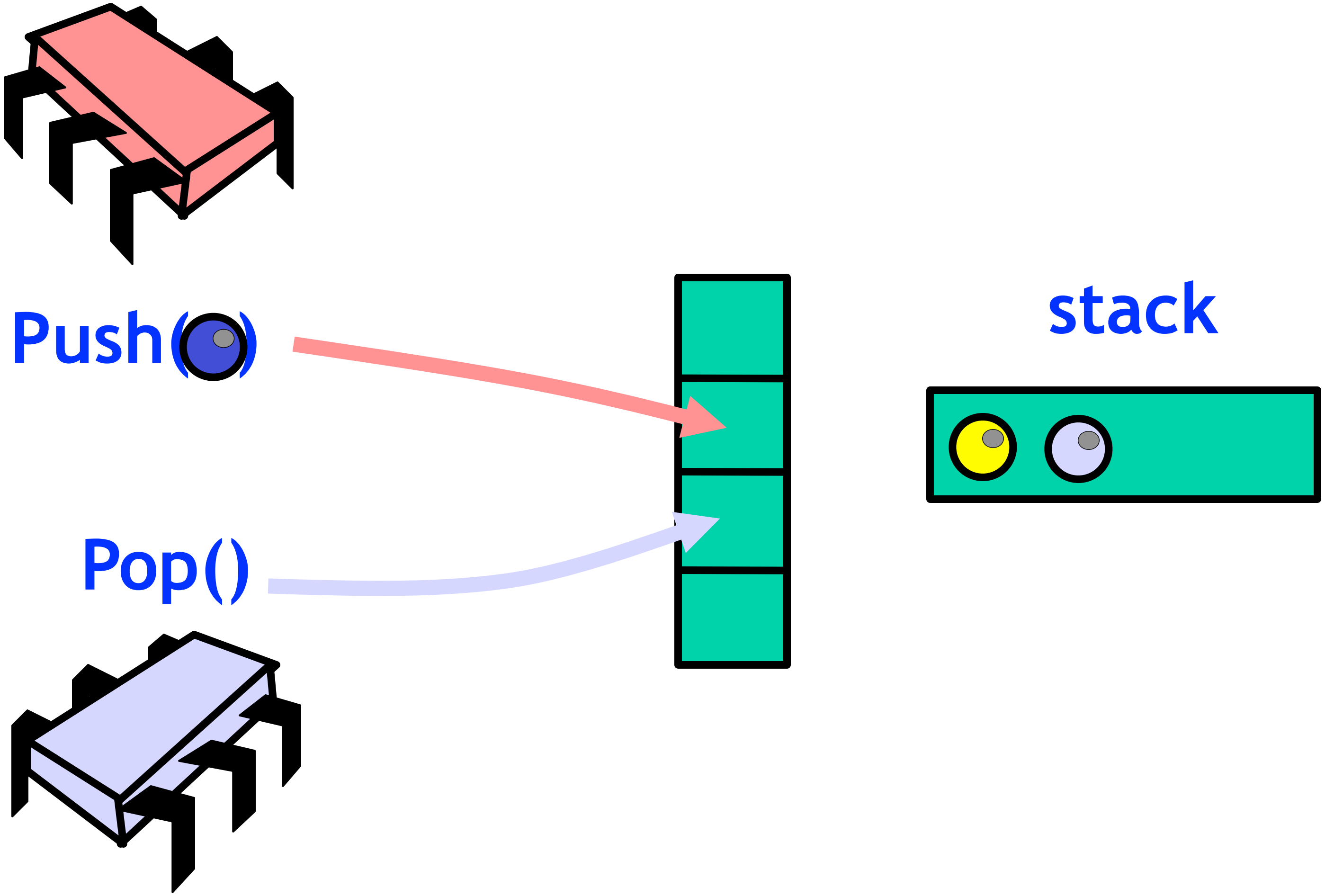


Push Collides With Pop



No need to access stack

No Collision

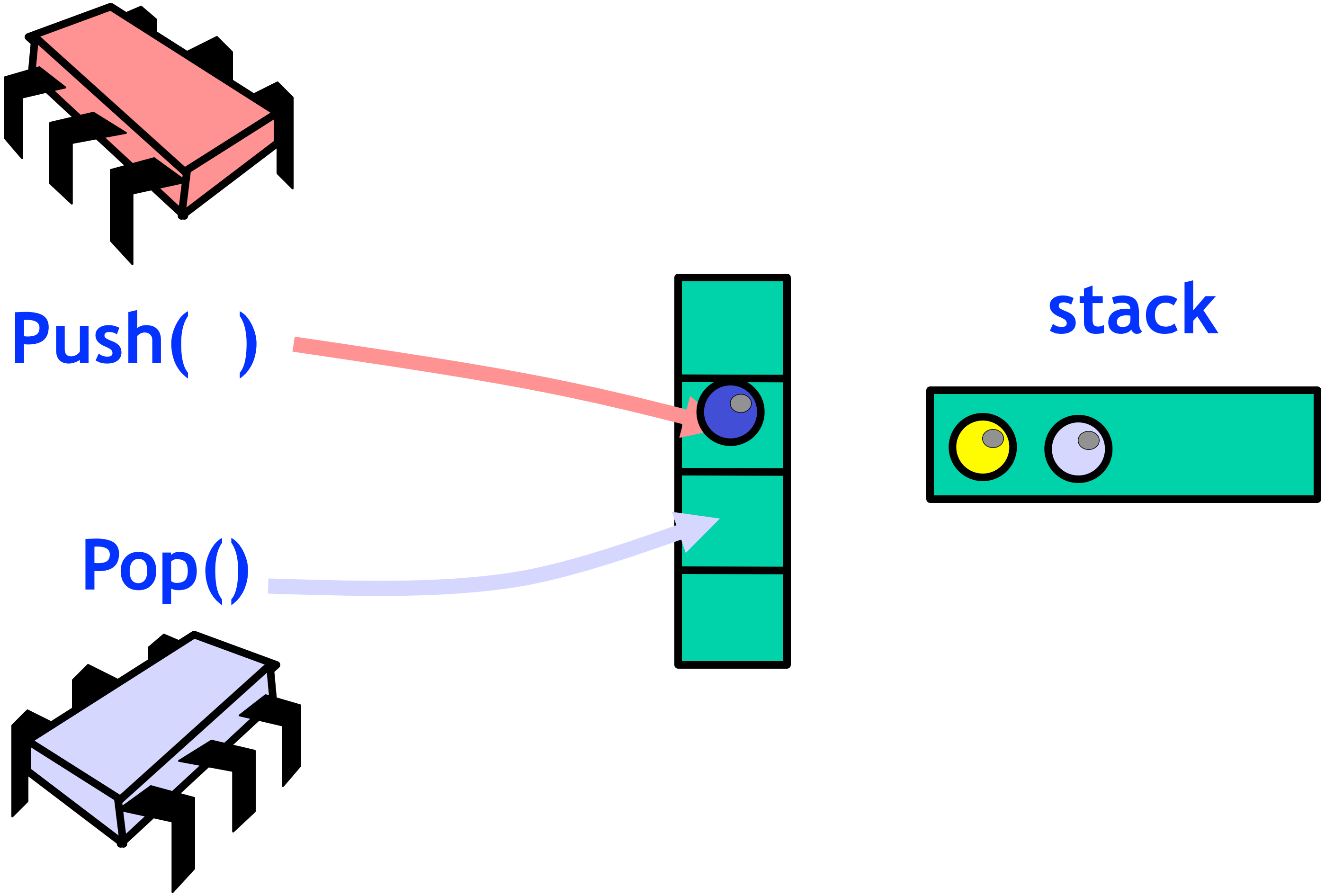


Push(●)

Pop()

stack

No Collision

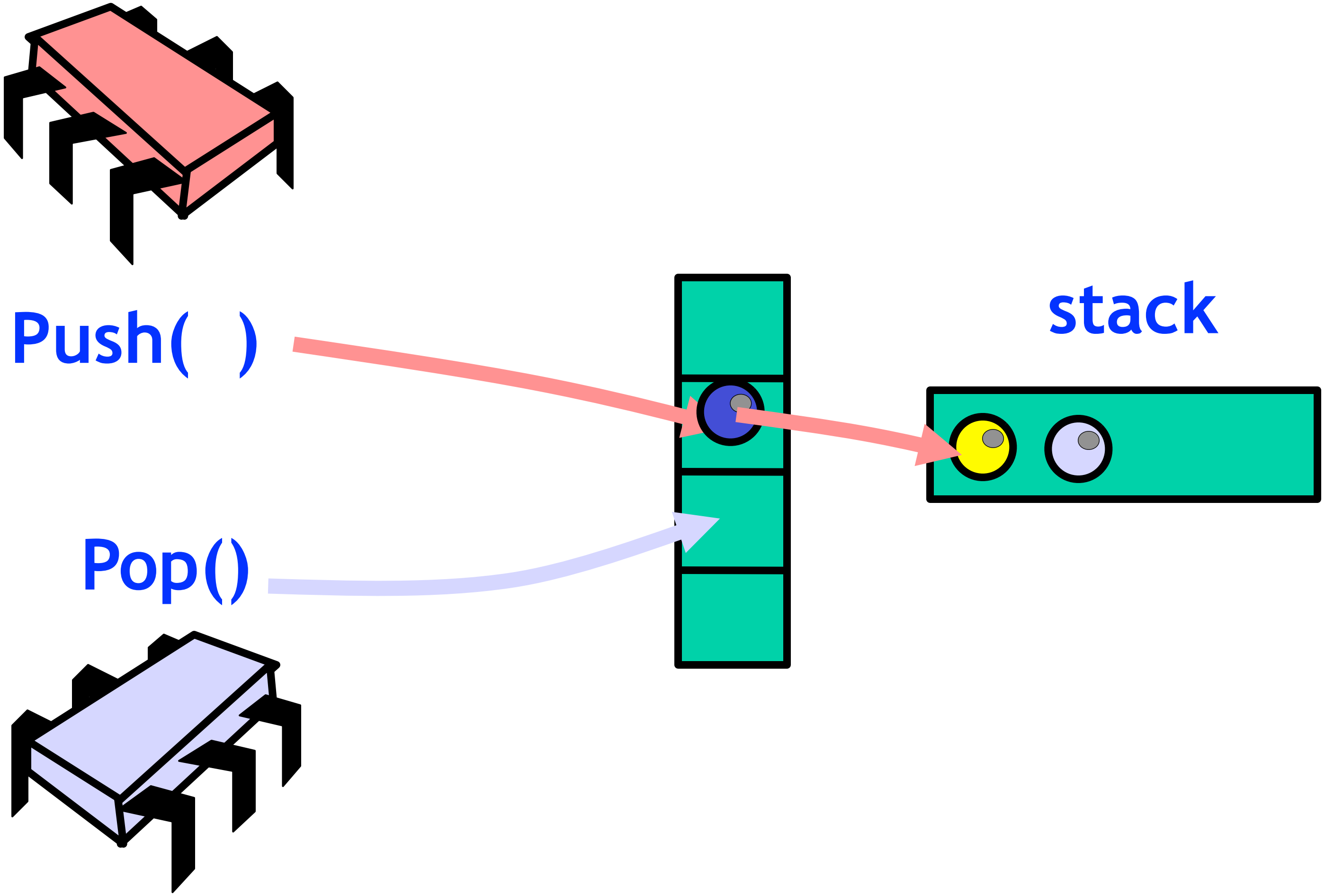


Push( )

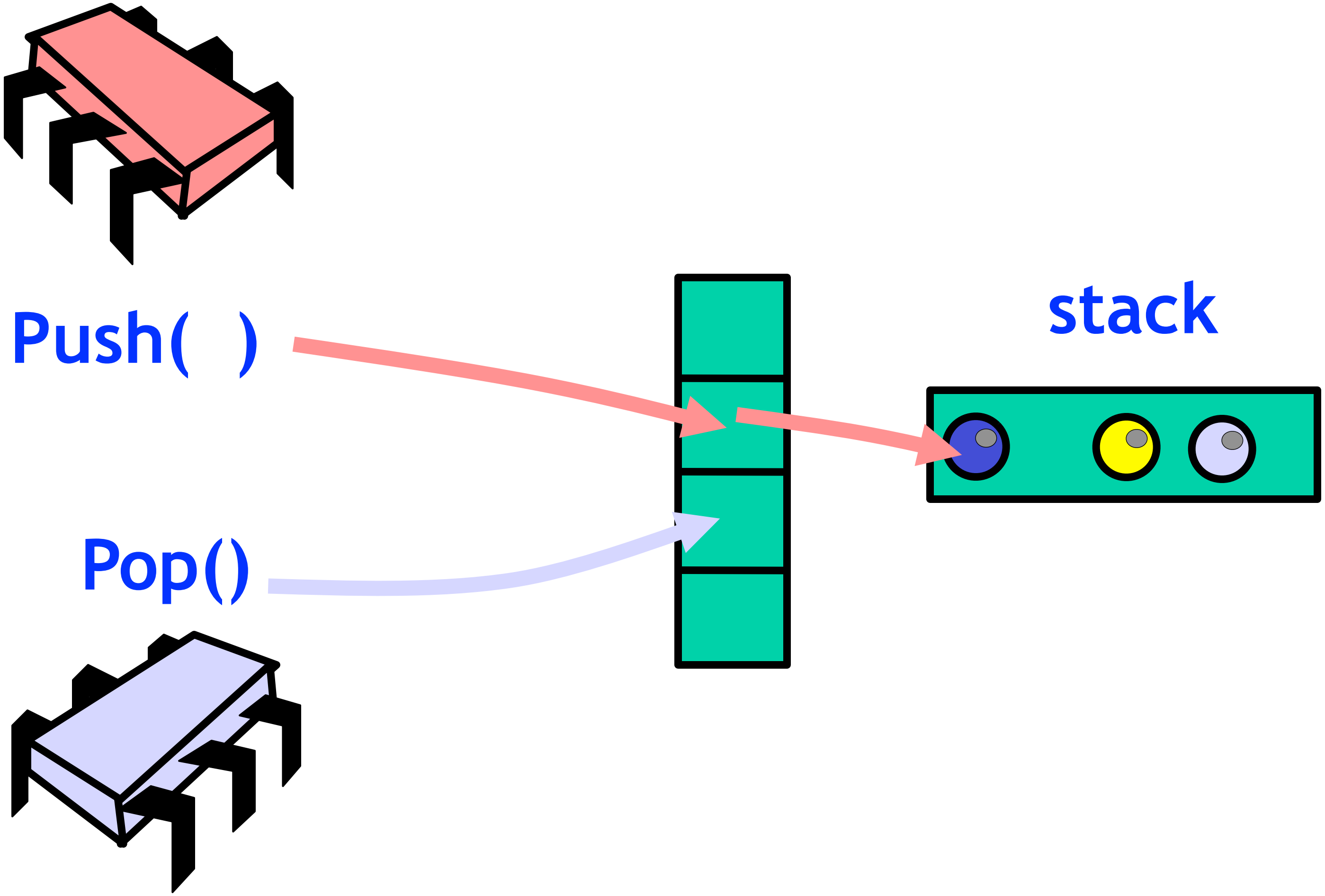
Pop()

stack

No Collision



No Collision

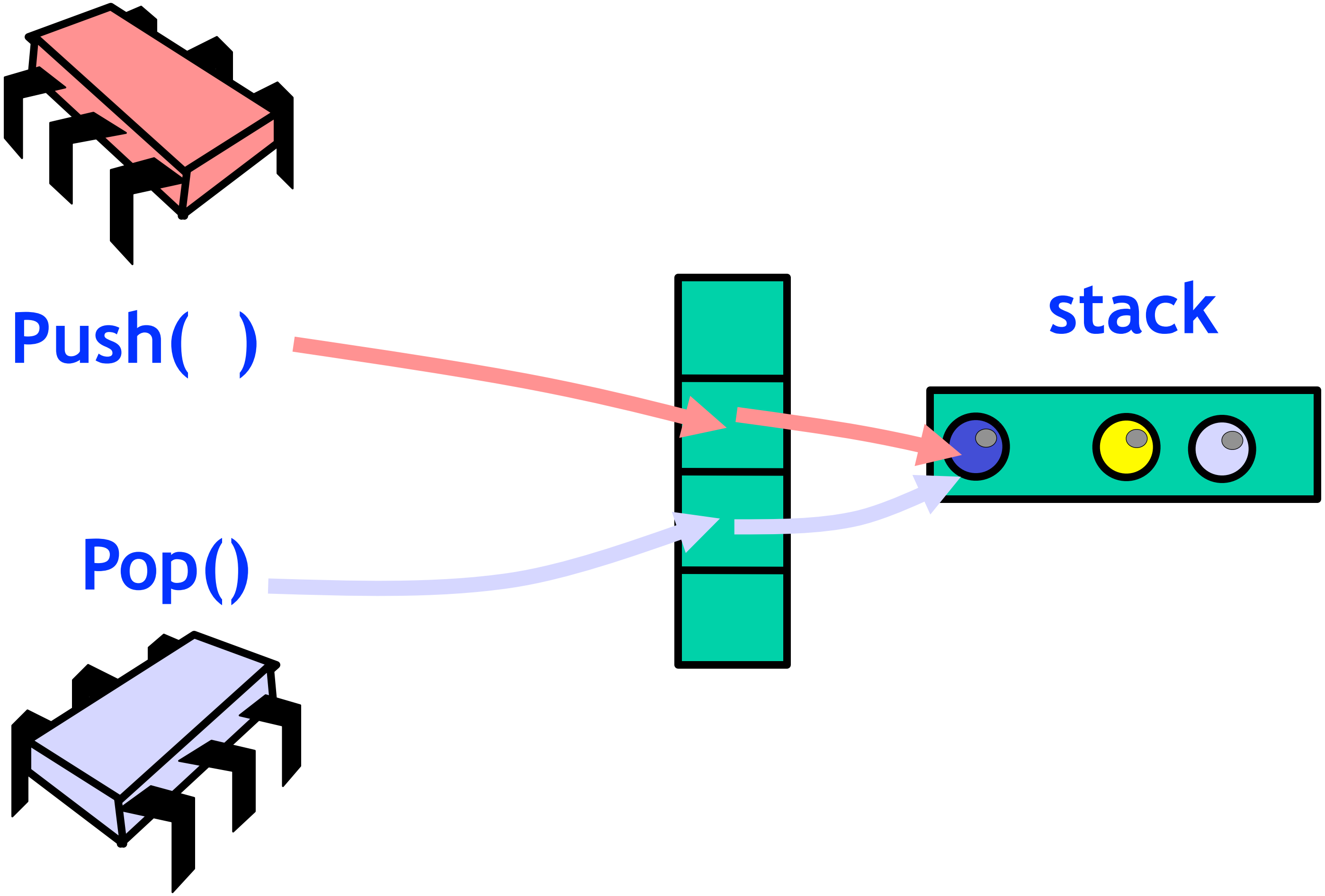


Push( )

Pop()

stack

No Collision

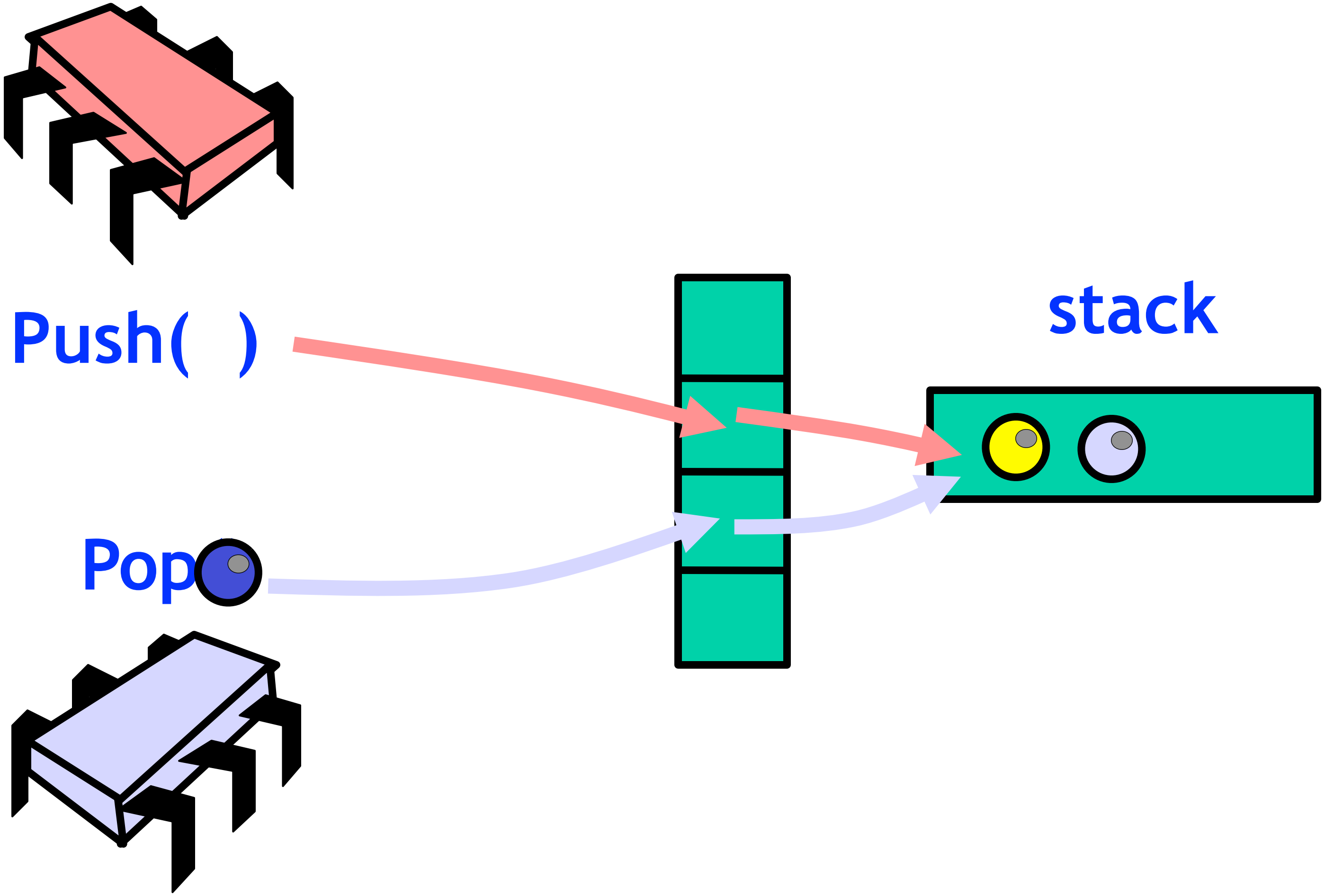


Push( )

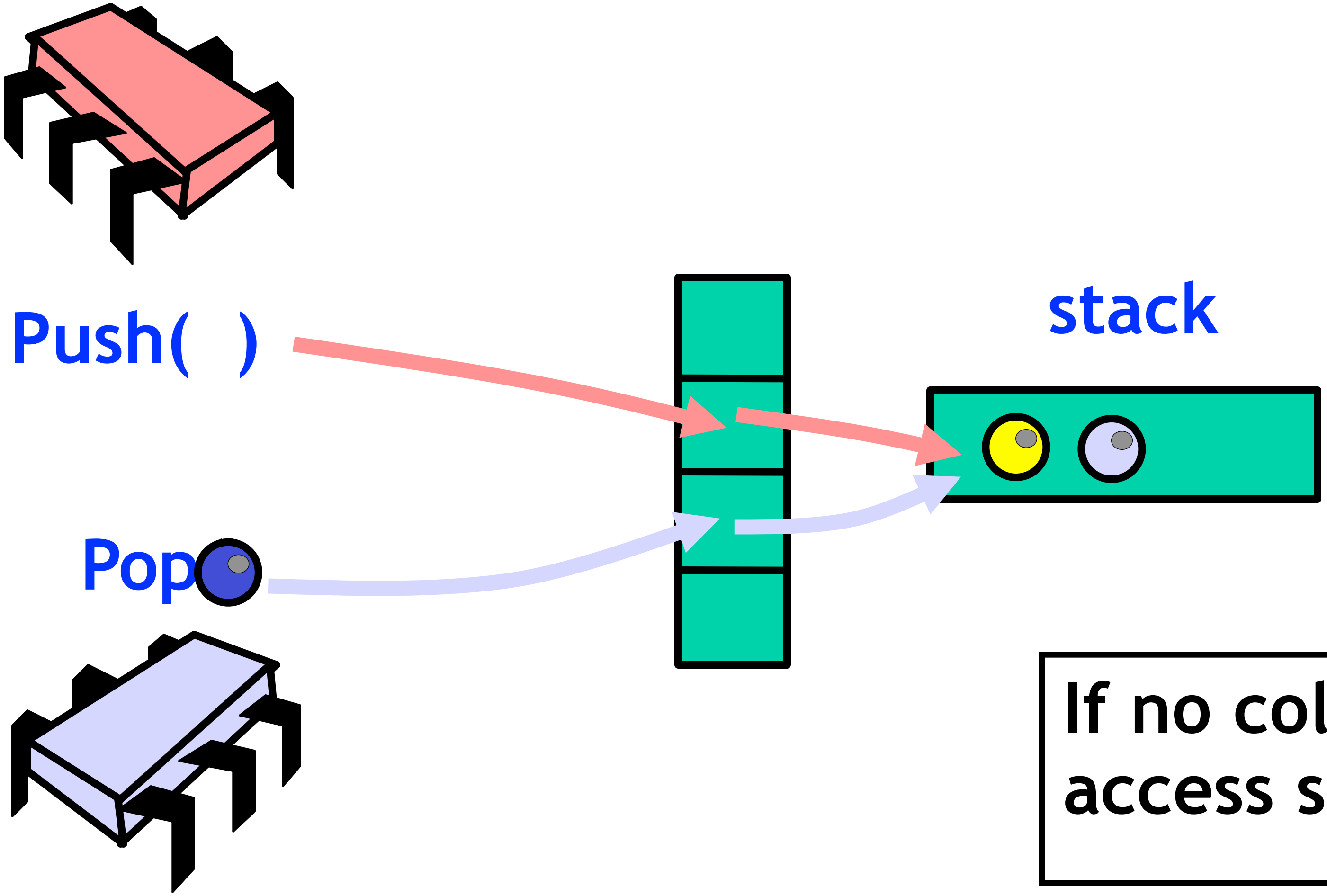
Pop()

stack

No Collision

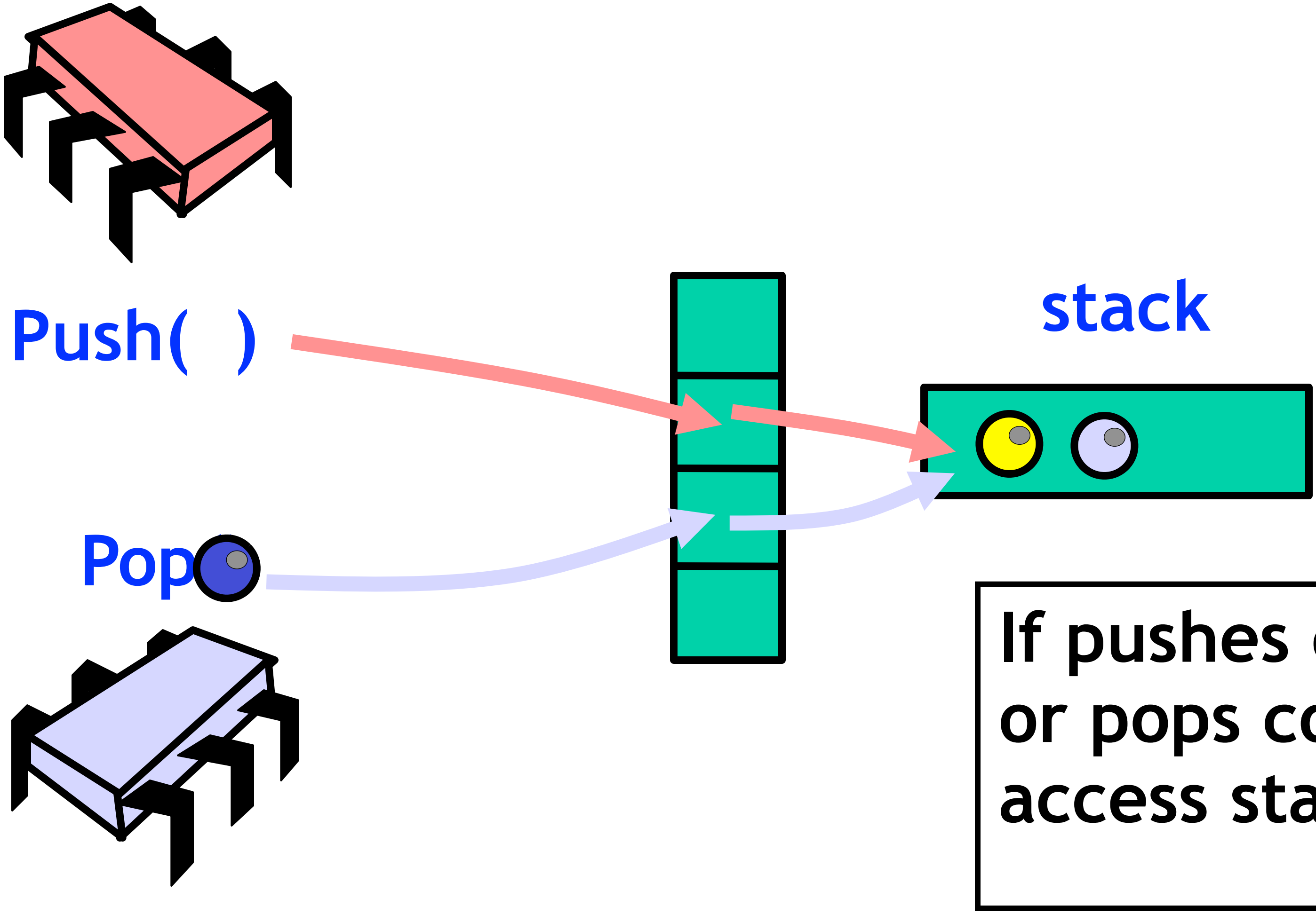


No Collision



If no collision, access stack

No Collision

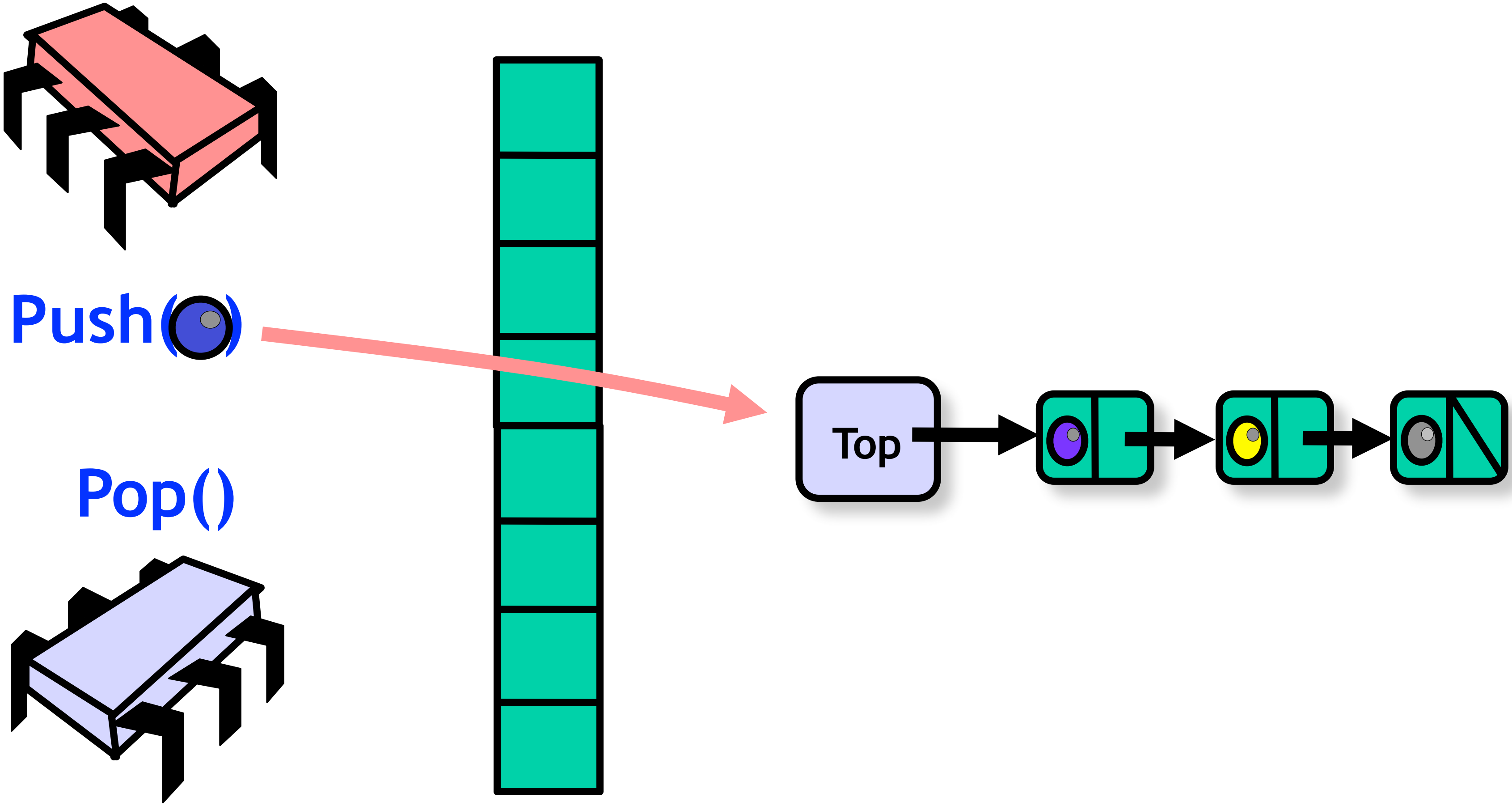


If pushes collide or pops collide access stack

Elimination-Backoff Stack

- Lock-free stack + elimination array
- Access Lock-free stack,
  - If uncontended, apply operation
  - if contended, back off to elimination array and attempt elimination

Elimination-Backoff Stack

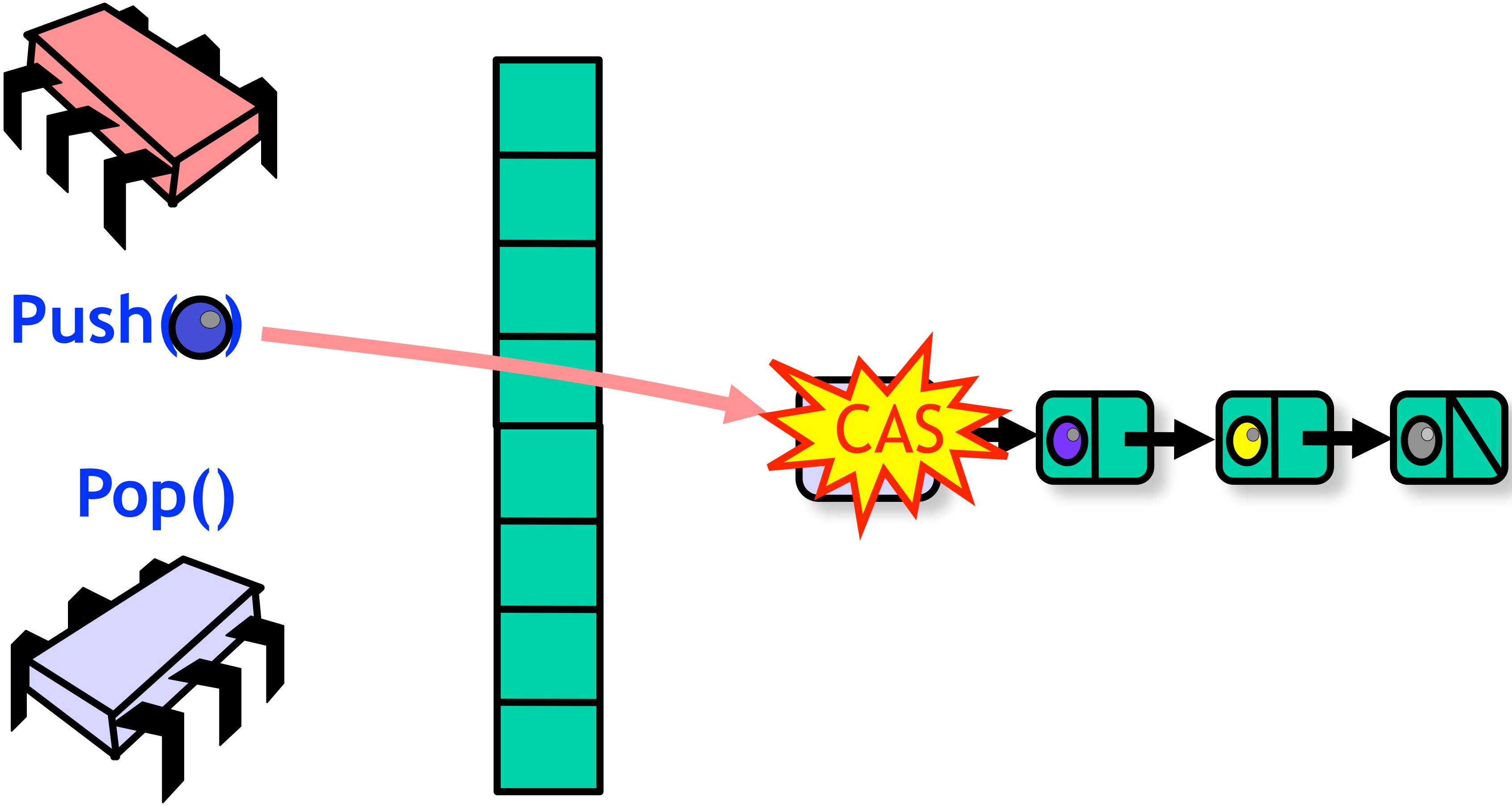


Push(●)

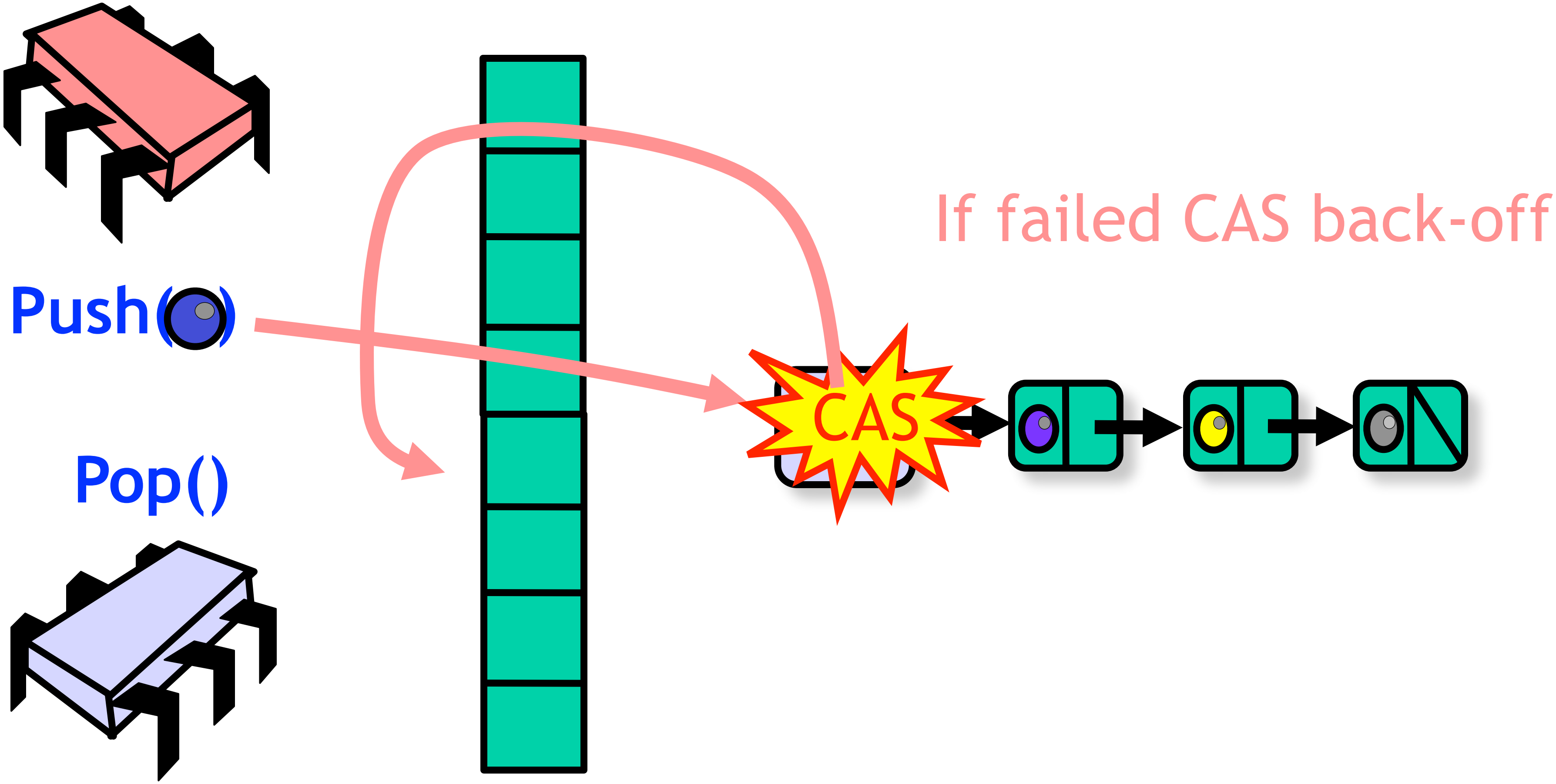
Pop()

Top

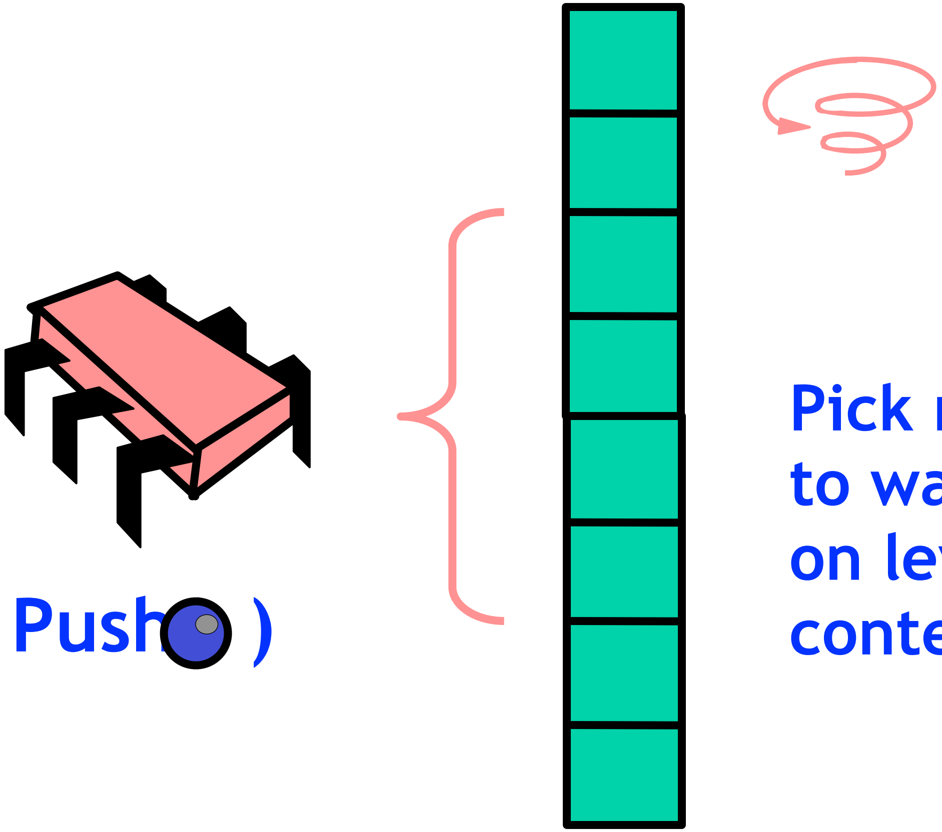
Elimination-Backoff Stack



Elimination-Backoff Stack



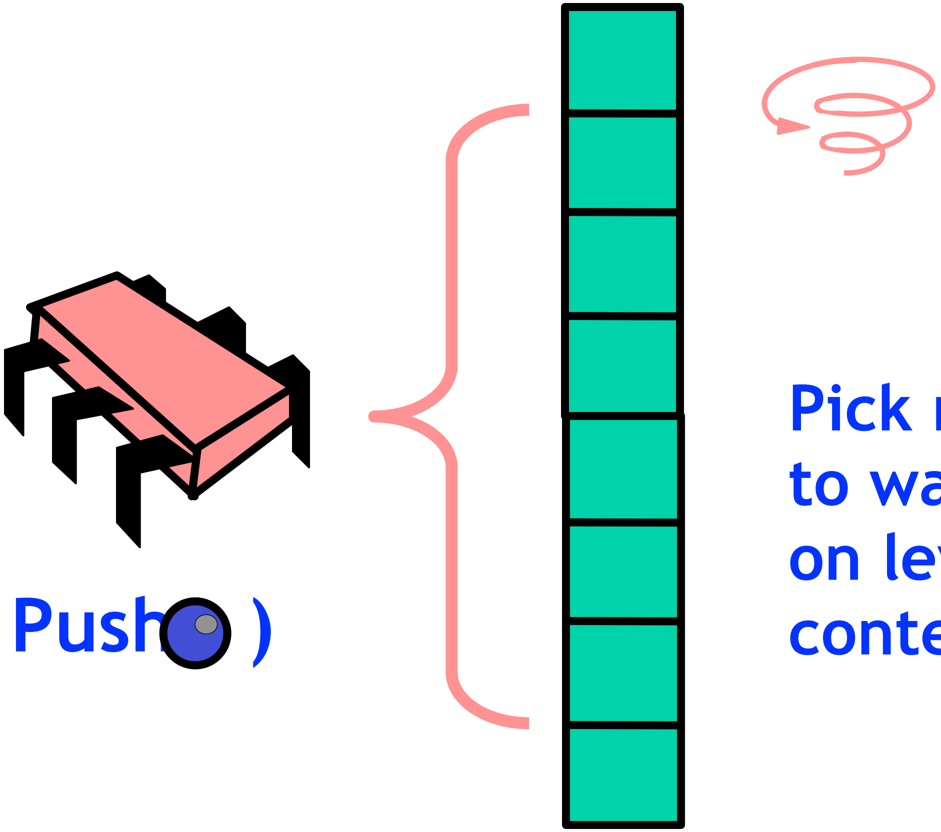
Dynamic Range and Delay



Push(●)

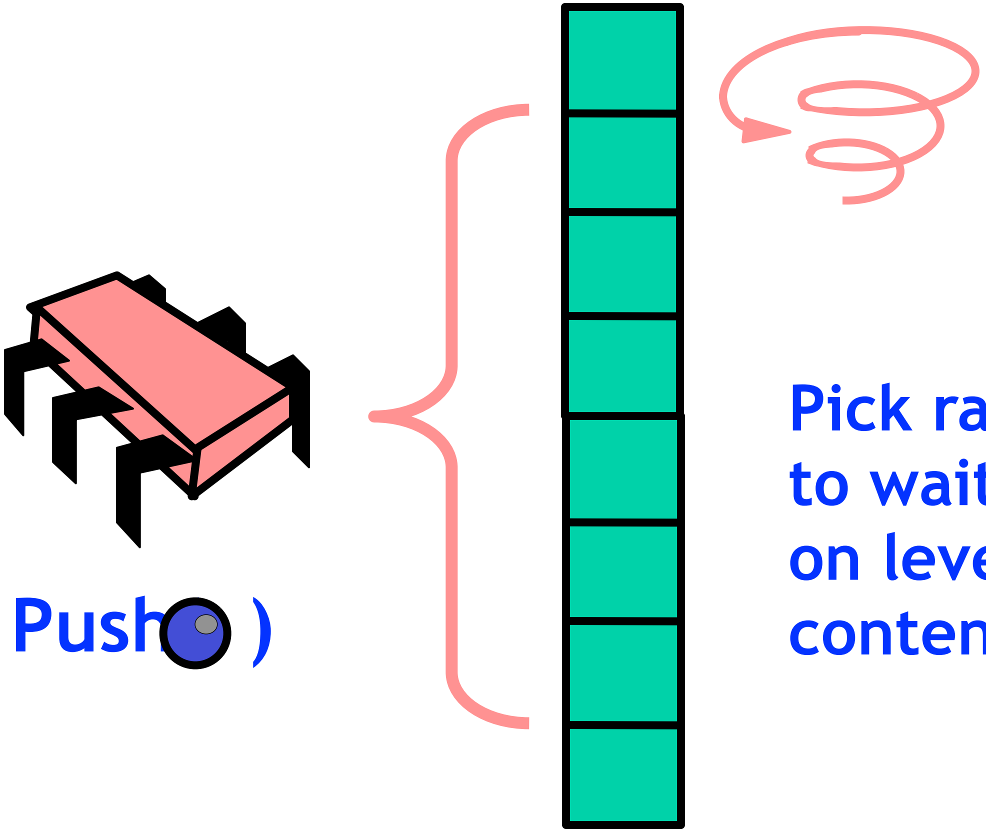
Pick range and max time to wait for collision based on level of contention encountered

Dynamic Range and Delay



Pick range and max time to wait for collision based on level of contention encountered

Dynamic Range and Delay



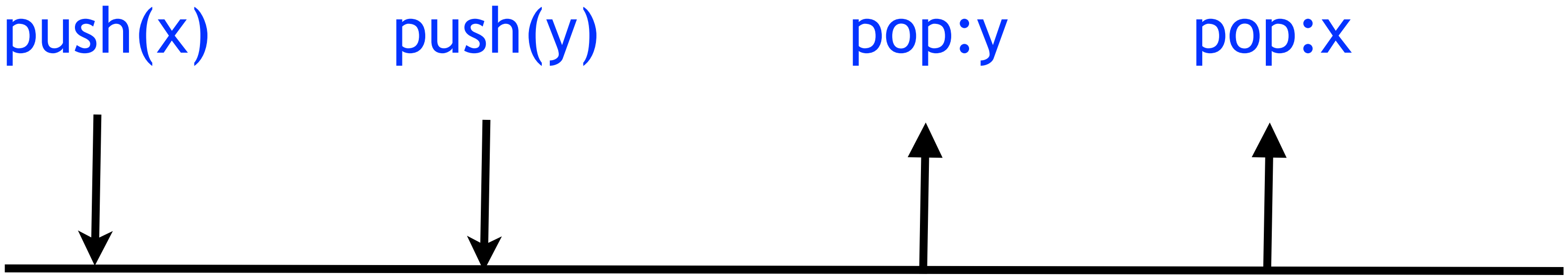
Push●)

Pick range and max time to wait for collision based on level of contention encountered

Linearizability

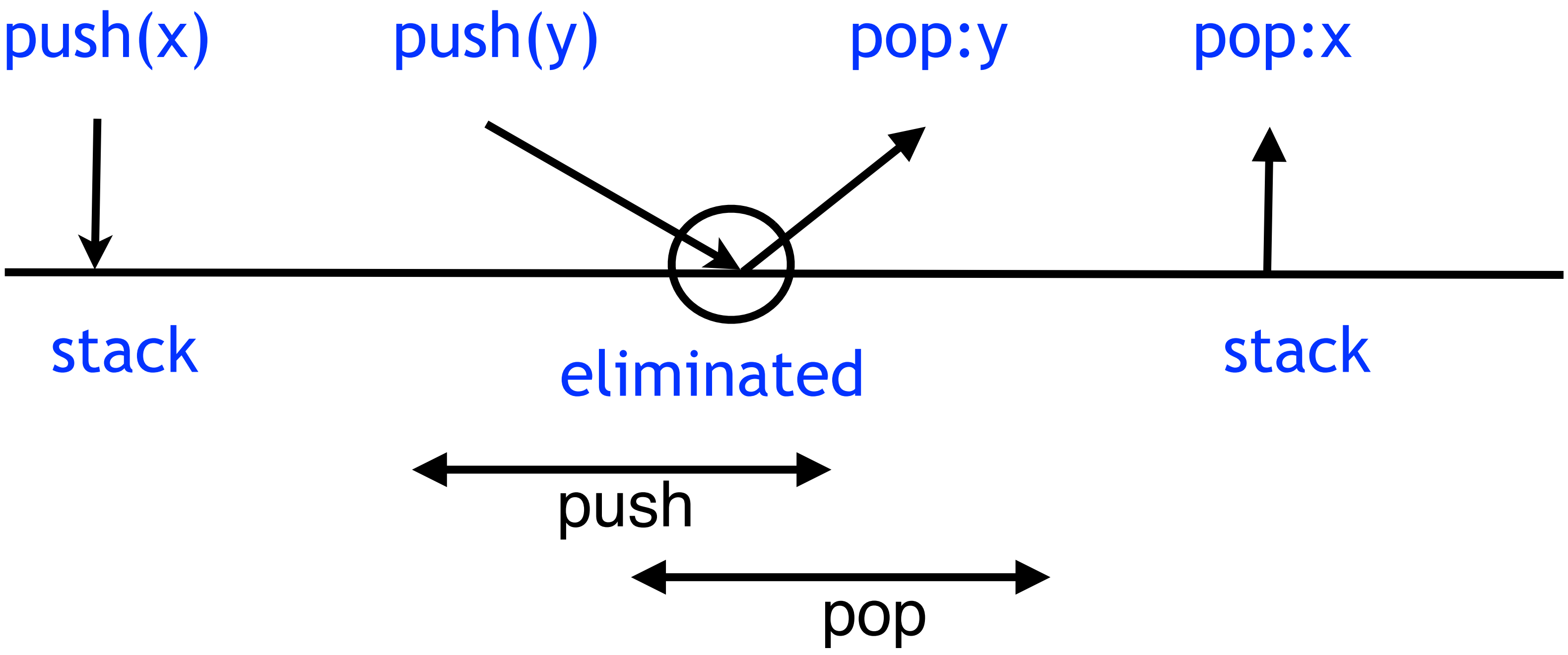
- Un-eliminated Lock-free stack calls:
  - linearized as before
- Eliminated calls:
  - linearize `push()` immediately before the `pop()` at the collision point
- Combination is a linearizable stack

Linearizability

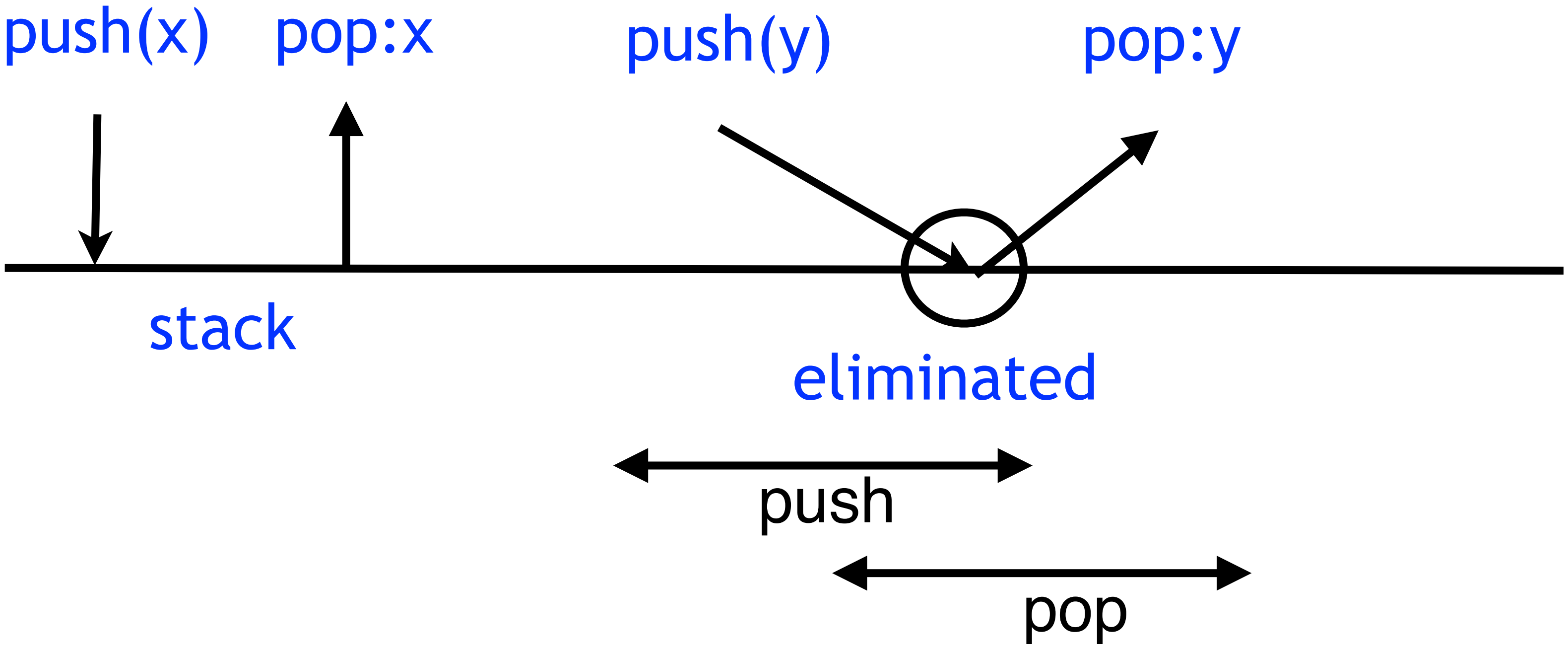


push(x) push(y) pop:y pop:x

Linearizability



Linearizability



push(x)

pop:x

push(y)

pop:y

stack

eliminated

Backoff Has Dual Effect

- Elimination introduces parallelism
- Backoff onto array cuts contention on lock-free stack
  - cuts down total number of threads ever accessing lock-free stack

Elimination Array

```
public class EliminationArray {
  private static final int duration = ...;
  private static final int timeUnit = ...;
  Exchanger<T>[] exchanger;
  Random random;
  public EliminationArray(int capacity) {
    exchanger = (Exchanger<T>[]) new
                                     Exchanger[capacity];
    for (int i = 0; i < capacity; i++) {
      exchanger[i] = new Exchanger<T>();
    }
    random = new Random();
  }
  …
}
```

Elimination Array

```
public class EliminationArray {
  private static final int duration = ...;
  private static final int timeUnit = ...;
  Exchanger<T>[] exchanger;
  Random random;
  public EliminationArray(int capacity) {
    exchanger = (Exchanger<T>[]) new
                Exchanger[capacity];
    for (int i = 0; i < capacity; i++) {
      exchanger[i] = new Exchanger<T>();
    }
    random = new Random();
  }
  …
}
```

An array of exchangers

A Lock-Free Exchanger

```
public class Exchanger<T> {
  AtomicStampedReference<T> slot = new
AtomicStampedReference<T>(null, 0);
```

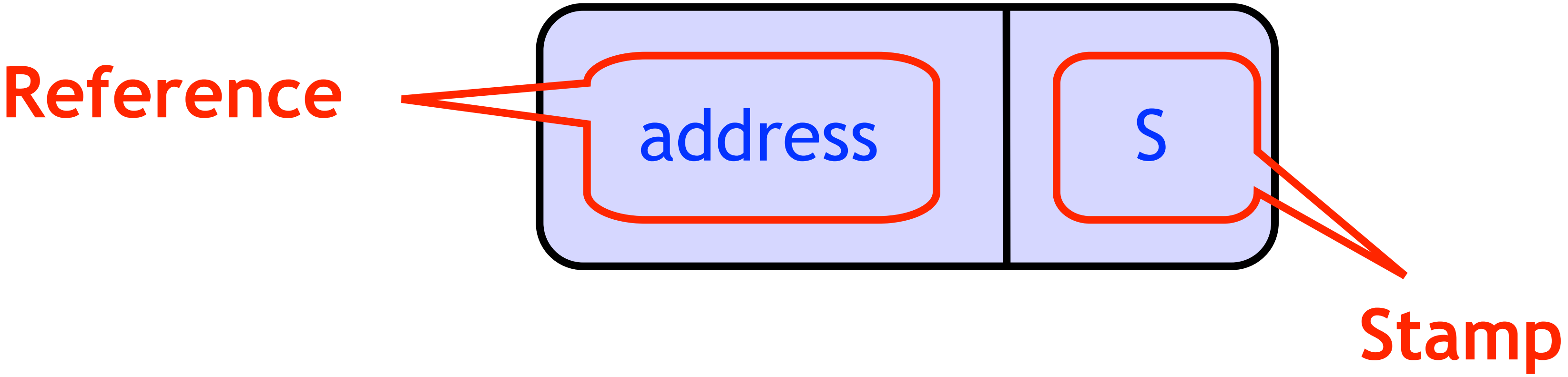
A Lock-Free Exchanger

```
public class Exchanger<T> {
 AtomicStampedReference<T> slot = new
AtomicStampedReference<T>(null, 0);
```

Slot holds atomically modifiable reference and time stamp

Atomic Stamped Reference

- AtomicStampedReference `class`
  - `Java.util.concurrent.atomic` package



Extracting Reference & Stamp

```
public T get(int[] stampHolder);
```

Extracting Reference & Stamp

```
Public T get(int[] stampHolder);
```

Returns reference to object of type T

Returns stamp at array index 0!

The Exchange

```
public T Exchange(T myItem, long nanos) throws
TimeoutException {
    long timeBound = System.nanoTime() + nanos;
    int[] stampHolder = {0};
    while (true) {
      if (System.nanoTime() > timeBound)
        throw new TimeoutException();
      T herItem = slot.get(stampHolder);
      int stamp = stampHolder[0];
      switch(stamp % 3) {
        case 0:  // slot is free
        case 1:  // someone waiting for me
        case 2:  // others exchanging
        }
    }}
```

The Exchange

```
public T Exchange(T myItem, long nanos) throws
TimeoutException {
   long timeBound = System.nanoTime() + nanos;
   int[] stampHolder = {0};
   while (true) {
     if (System.nanoTime() > timeBound)
       throw new TimeoutException();
     T herItem = slot.get(stampHolder);
     int stamp = stampHolder[0];
     switch(stamp % 3) {
       case 0:  // slot is free
       case 1:  // someone waiting for me
       case 2:  // others exchanging
       }
   }}
```

Input item and max time to wait for exchange before timing out

The Exchange

```
public T Exchange(T myItem, long nanos) throws
TimeoutException {
  long timeBound = System.nanoTime() + nanos;
  int[] stampHolder = {0};
  while (true) {
   if (System.nanoTime() > timeBound)
     throw new TimeoutException();
   T herItem = slot.get(stampHolder);
   int stamp = stampHolder[0];
   switch(stamp % 3) {
     case 0:  // slot is free
     case 1:  // someone waiting for me
     case 2:  // others exchanging
     }
  }}
```

Array to hold extracted timestamp

The Exchange

```
public T Exchange(T myItem, long nanos) throws
TimeoutException {
    long timeBound = System.nanoTime() + nanos;
    int[] stampHolder = {0};
    while (true) {
      if (System.nanoTime() > timeBound)
        throw new TimeoutException();
      T herItem = slot.get(stampHolder);
      int stamp = stampHolder[0];
      switch(stamp % 3) {
        case 0:  // slot is free
        case 1:  // someone waiting for me
        case 2:  // others exchanging
        }
    }}
```

Loop as long as time to attempt exchange does not run out

The Exchange

```
public T Exchange(T myItem, long nanos) throws
TimeoutException {
  long timeBound = System.nanoTime() + nanos;
  int[] stampHolder = {0};
  while (true) {
   if (System.nanoTime() > timeBound)
     throw new TimeoutException();
   T herItem = slot.get(stampHolder);
   int stamp = stampHolder[0];
   switch(stamp % 3) {
     case 0:  // slot is free
     case 1:  // someone waiting for me
     case 2:  // others exchanging
     }
  }}
```

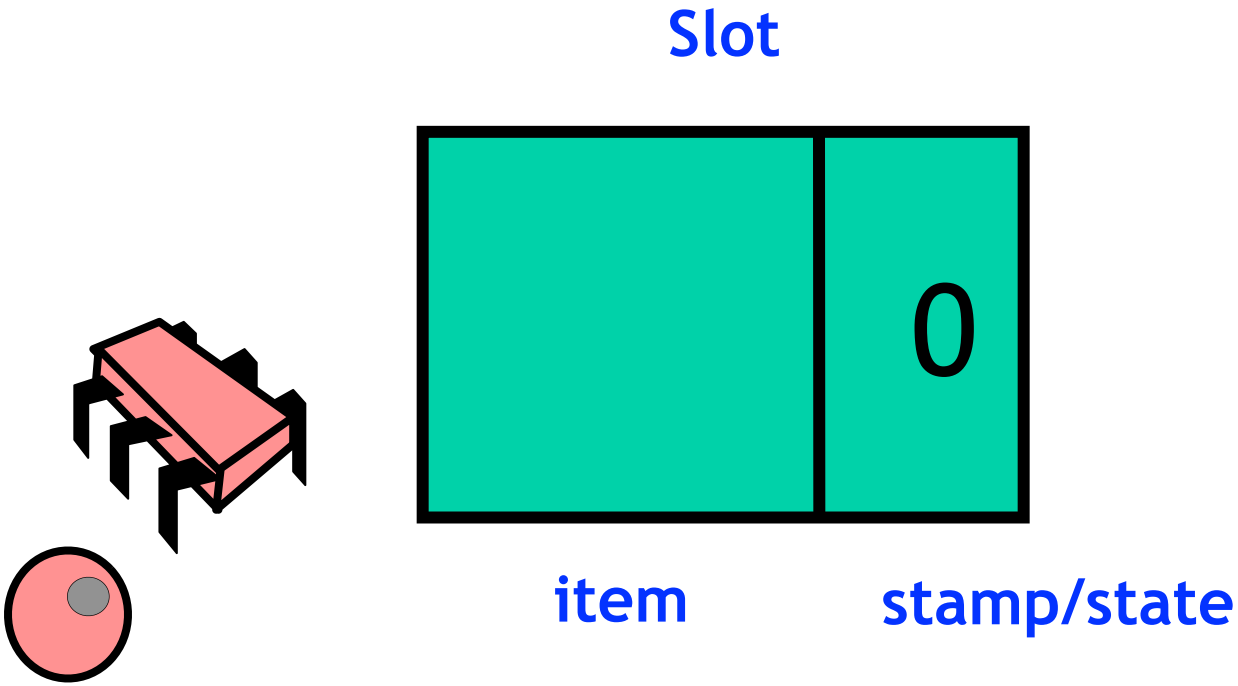
Get others item and time-stamp

The Exchange

```
public T Exchange(T myItem, long nanos) throws
TimeoutException {
  long timeBound = System.nanoTime() + nanos;
  int[] stampHolder = {0};
  while (true) {
   if (System.nanoTime() > timeBound)
     throw new TimeoutException();
   T herItem = slot.get(stampHolder);
   int stamp = stampHolder[0];
   switch(stamp % 3) {
     case 0:  // slot is free
     case 1:  // someone waiting for me
     case 2:  // others exchanging
     }
  }}
```

Exchanger slot has three states determined by the timestamp mod 3

Lock-free Exchanger



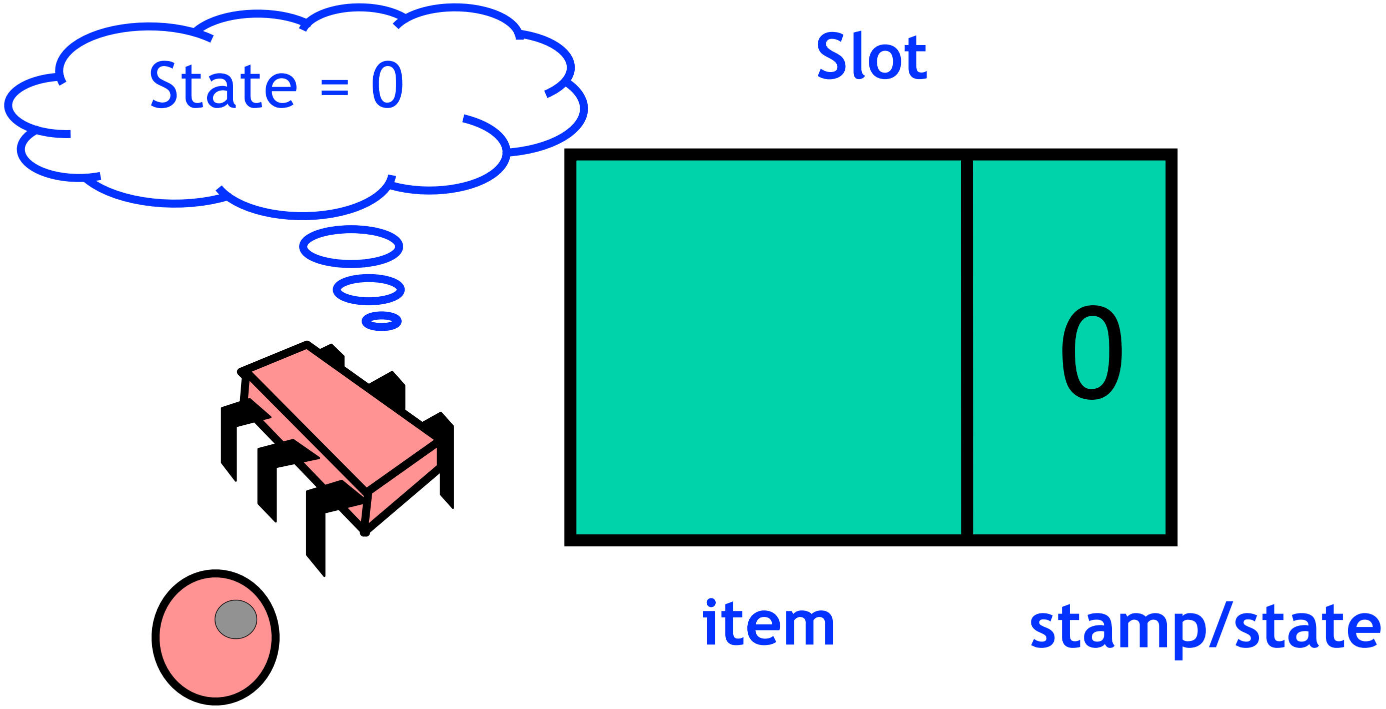
Slot

0

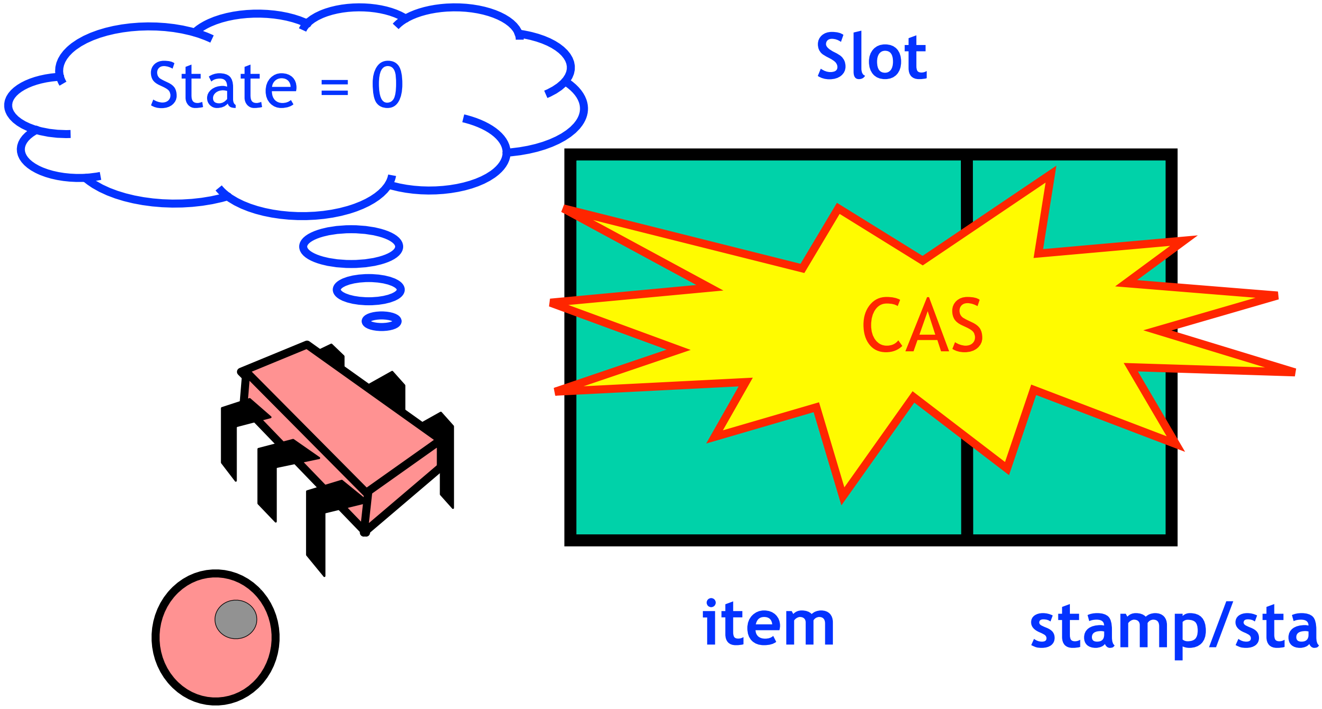
item

stamp/state

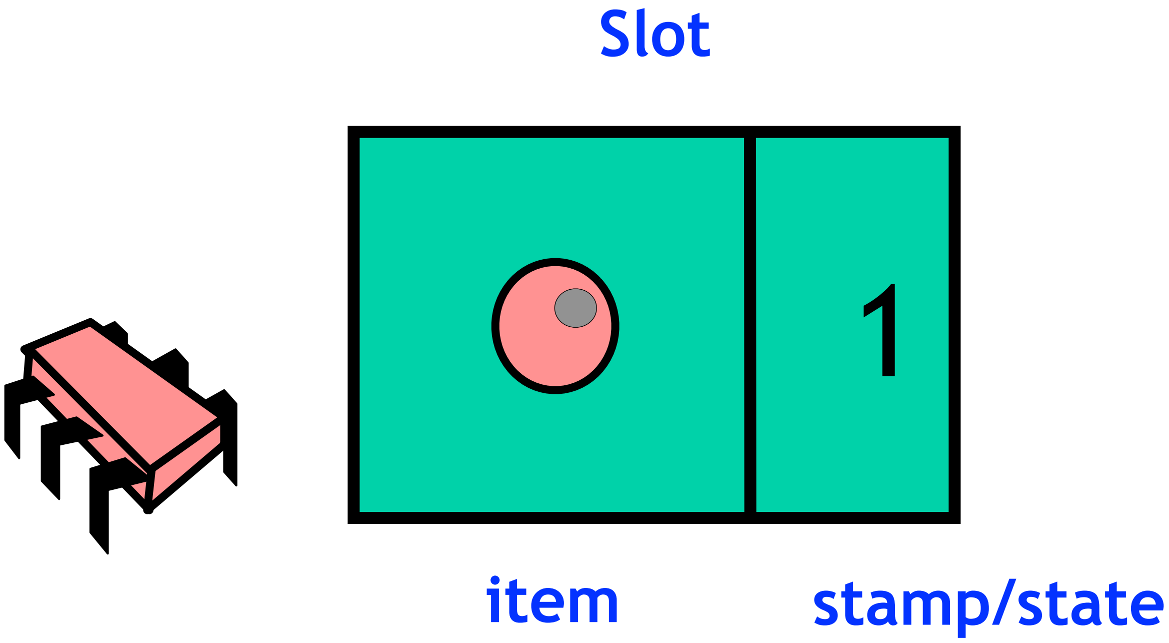
Lock-free Exchanger



Lock-free Exchanger



Lock-free Exchanger



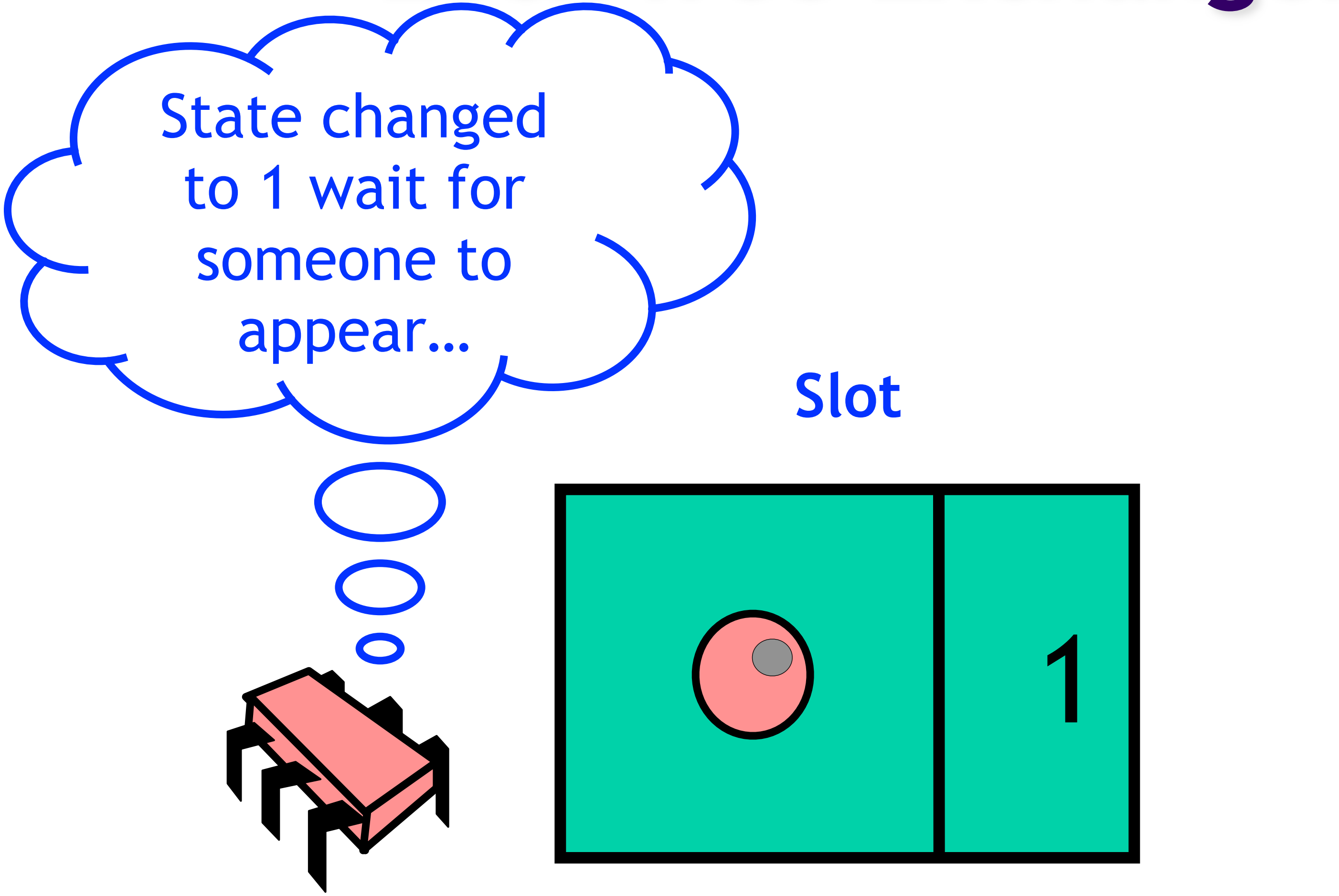
Slot

1

item

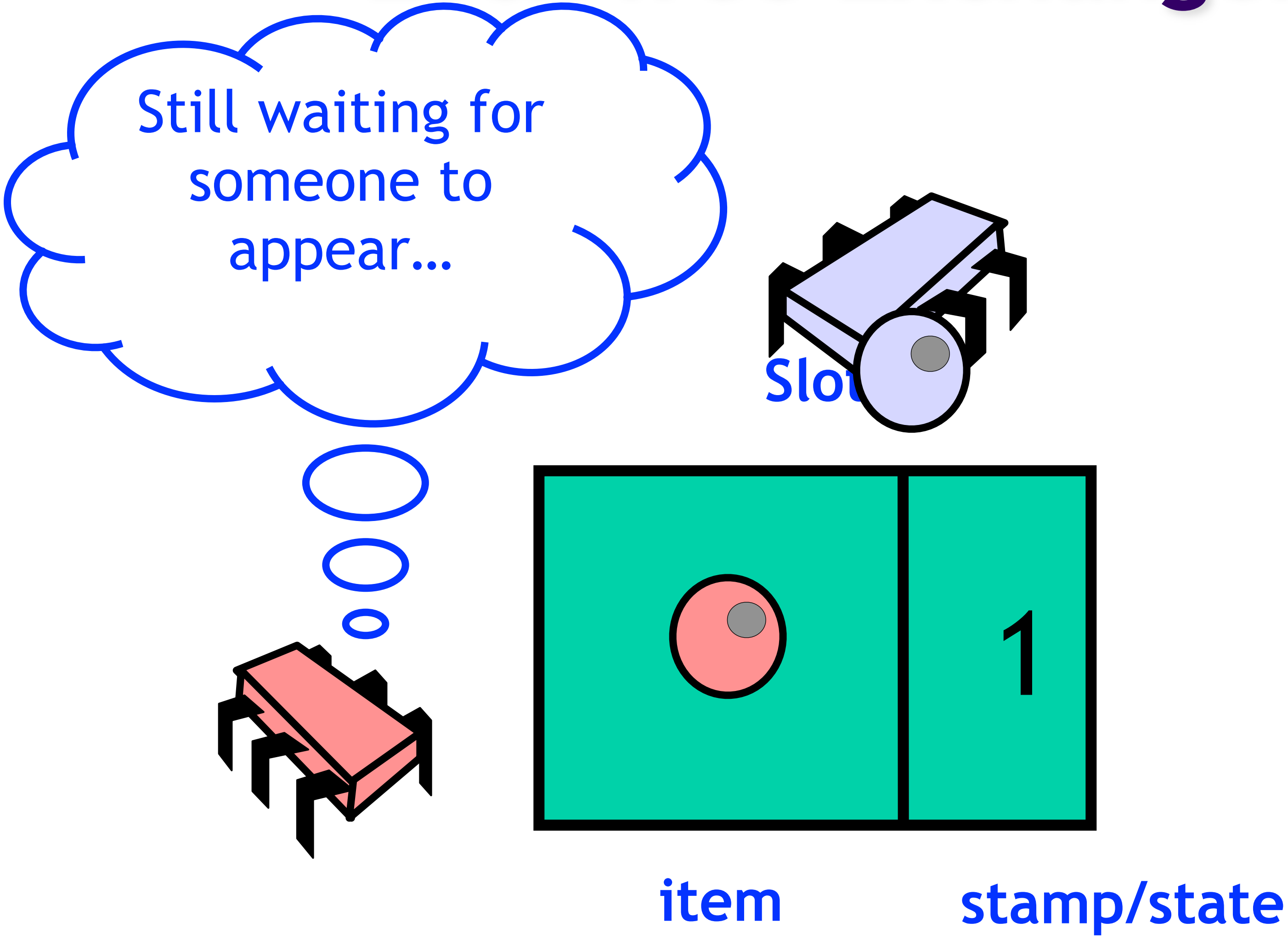
stamp/state

Lock-free Exchanger



State changed to 1 wait for someone to appear...

Lock-free Exchanger



Still waiting for someone to appear…

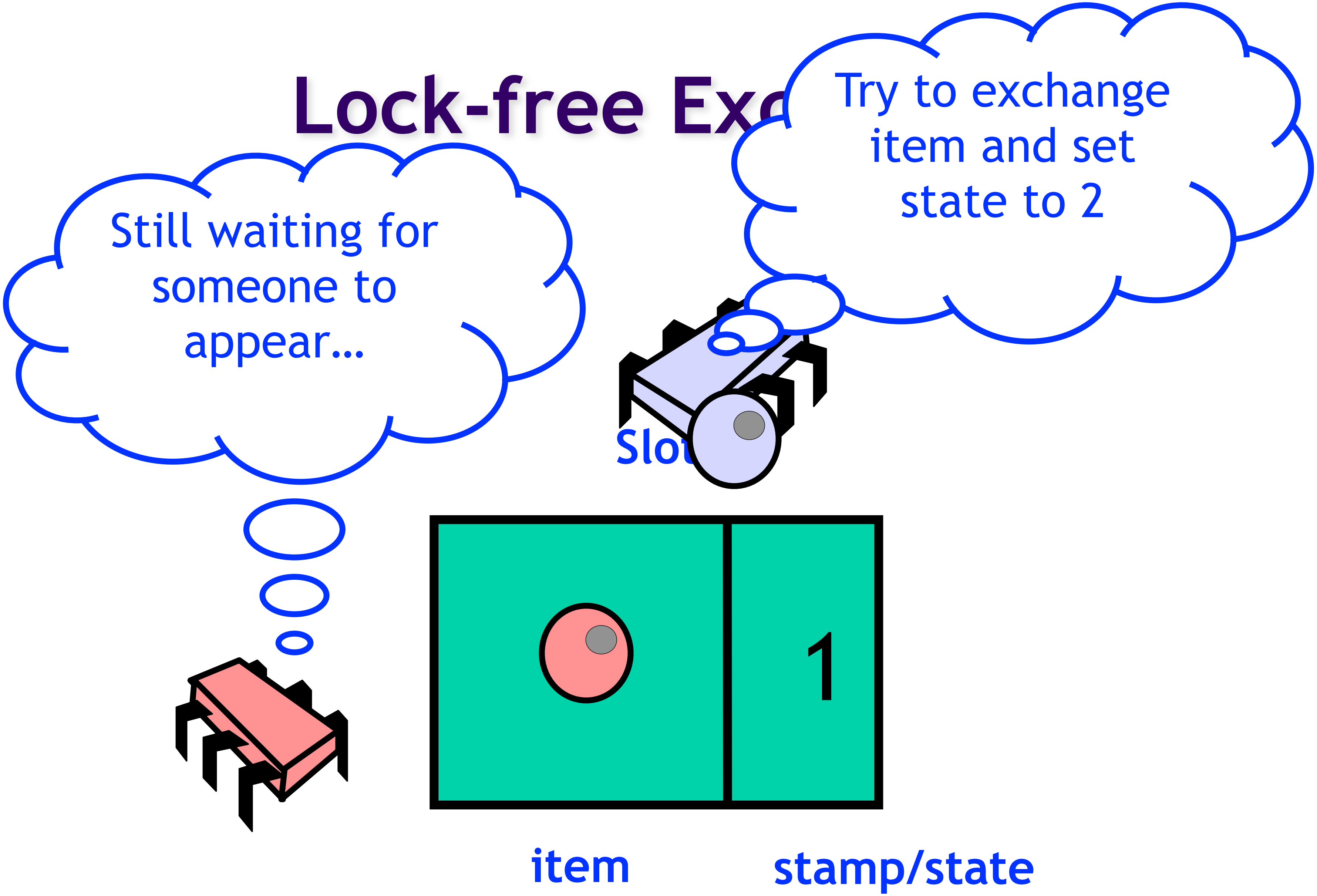
Slot

1

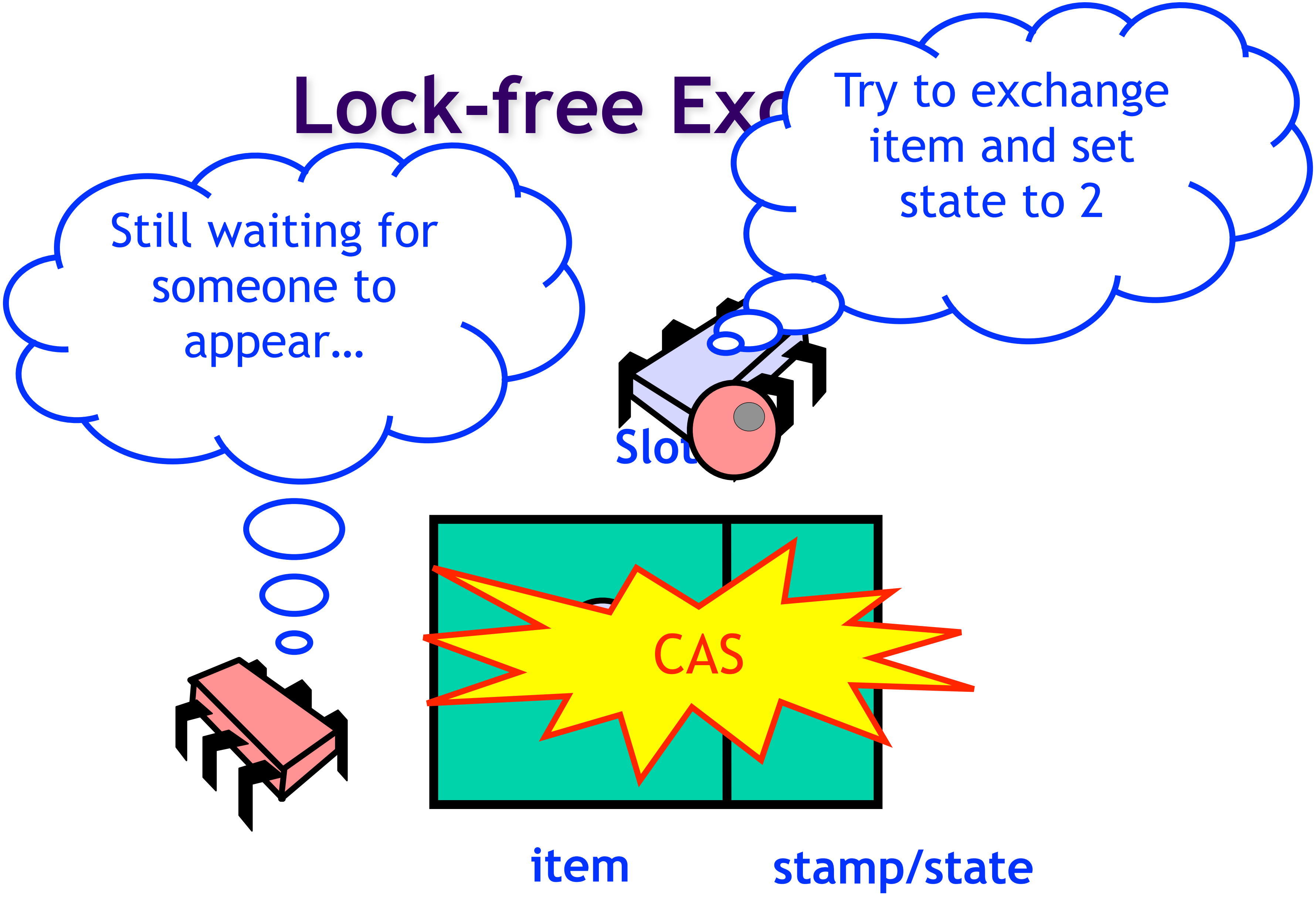
item

stamp/state

Lock-free Exc


Still waiting for someone to appear…
Try to exchange item and set state to 2
Slot
1
item
stamp/state

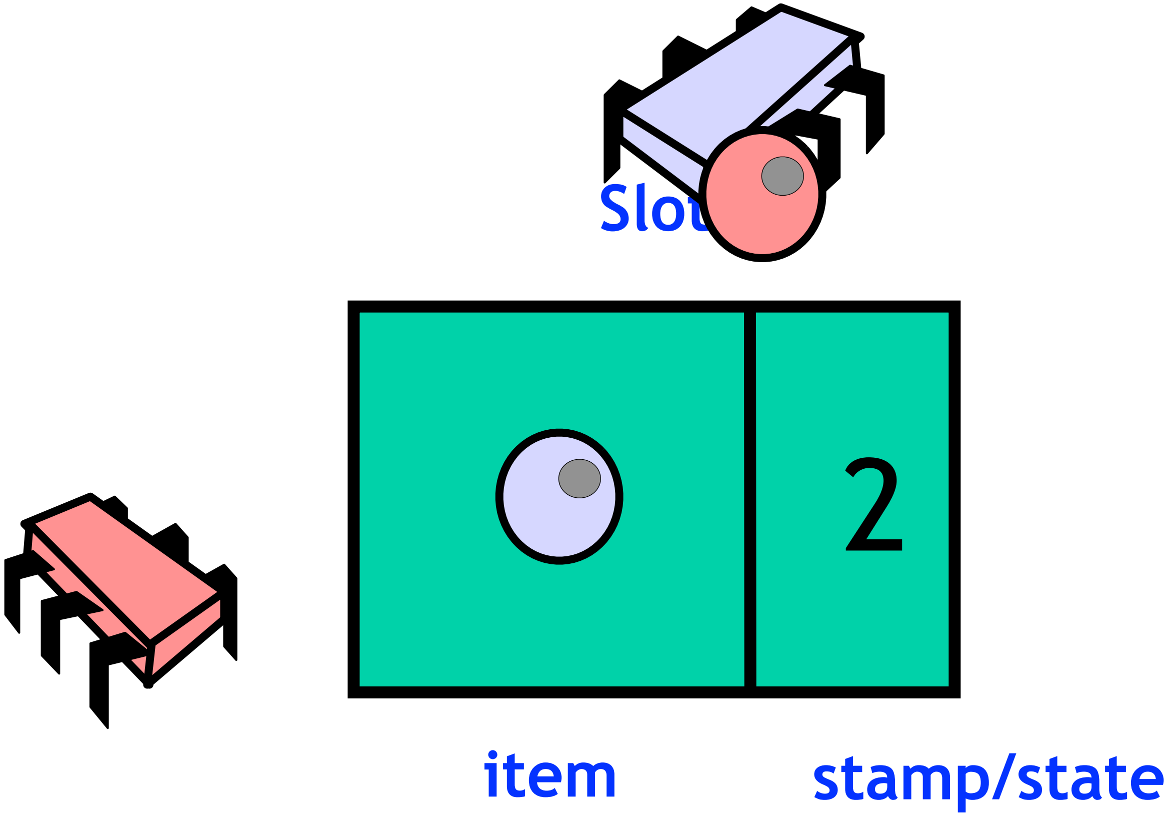
Lock-free Exc



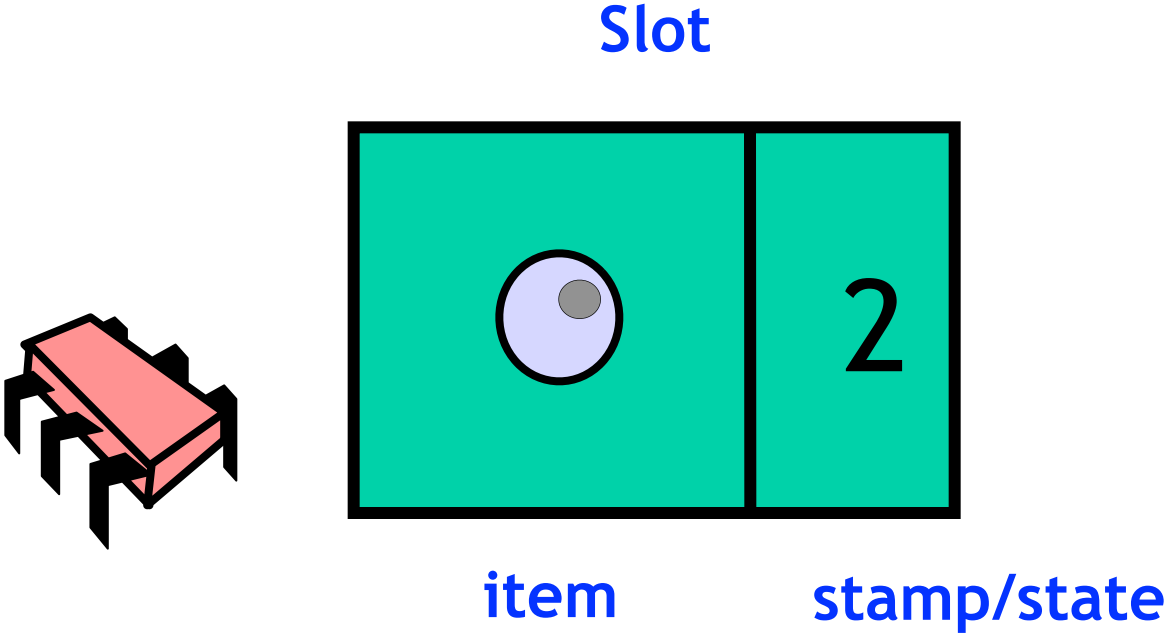
Still waiting for someone to appear…

Try to exchange item and set state to 2

Lock-free Exchanger



Lock-free Exchanger



Slot

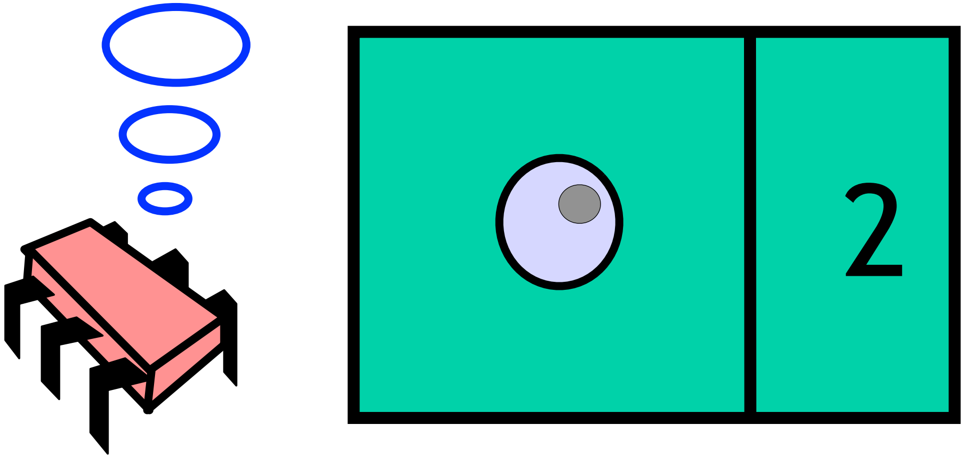
2

item

stamp/state

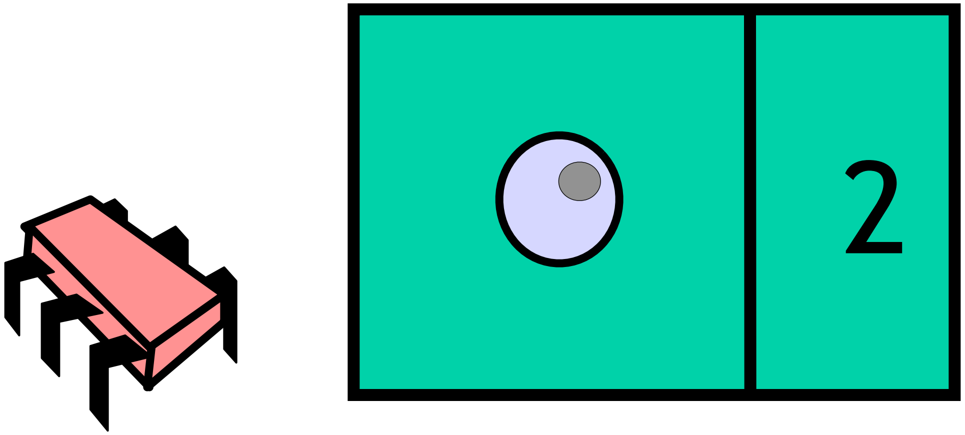
Lock-free Exchanger

2 means someone showed up, take item and reset to 0



Lock-free Exchanger

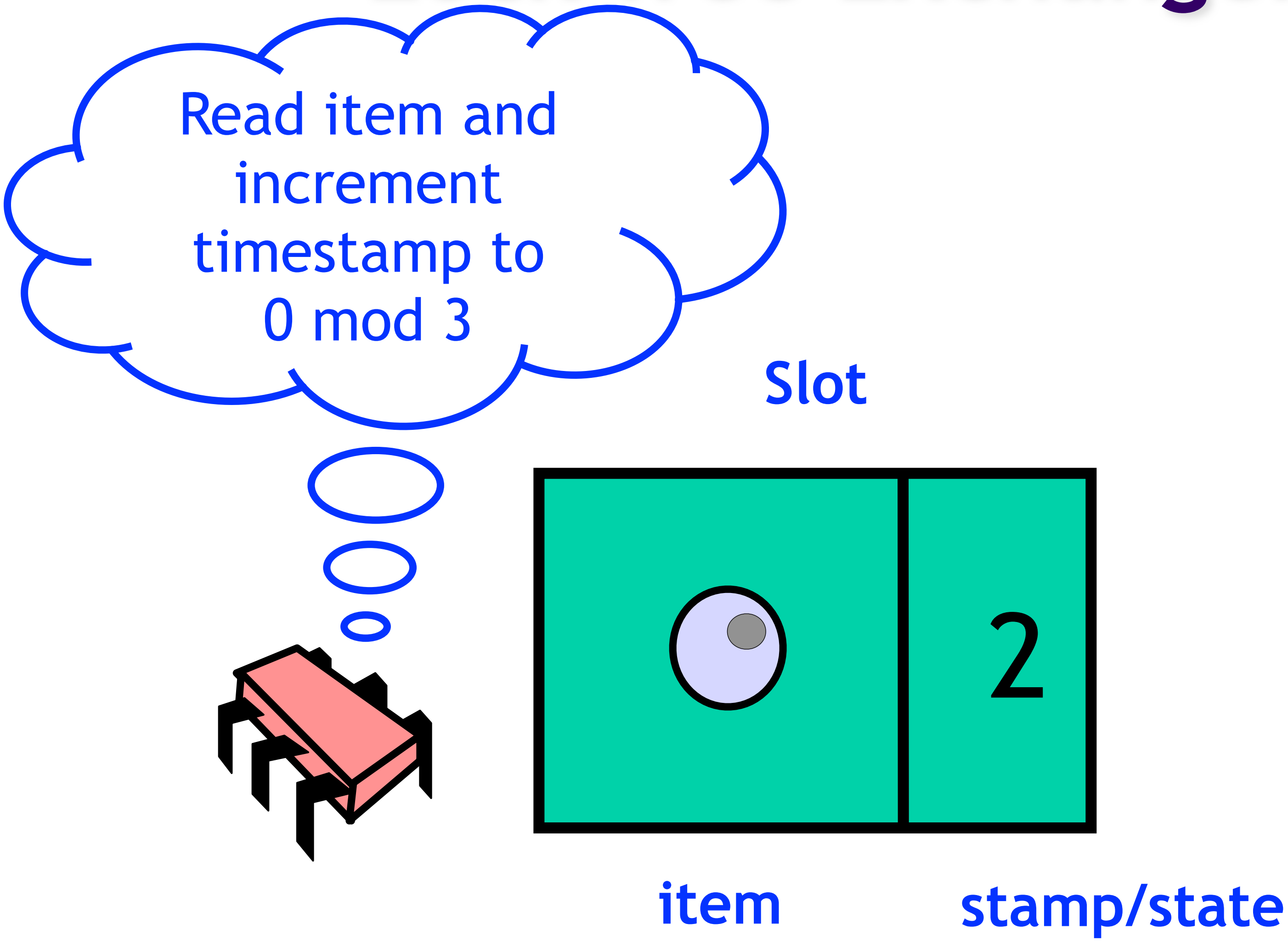
Slot



item

stamp/state

Lock-free Exchanger



Read item and increment timestamp to 0 mod 3

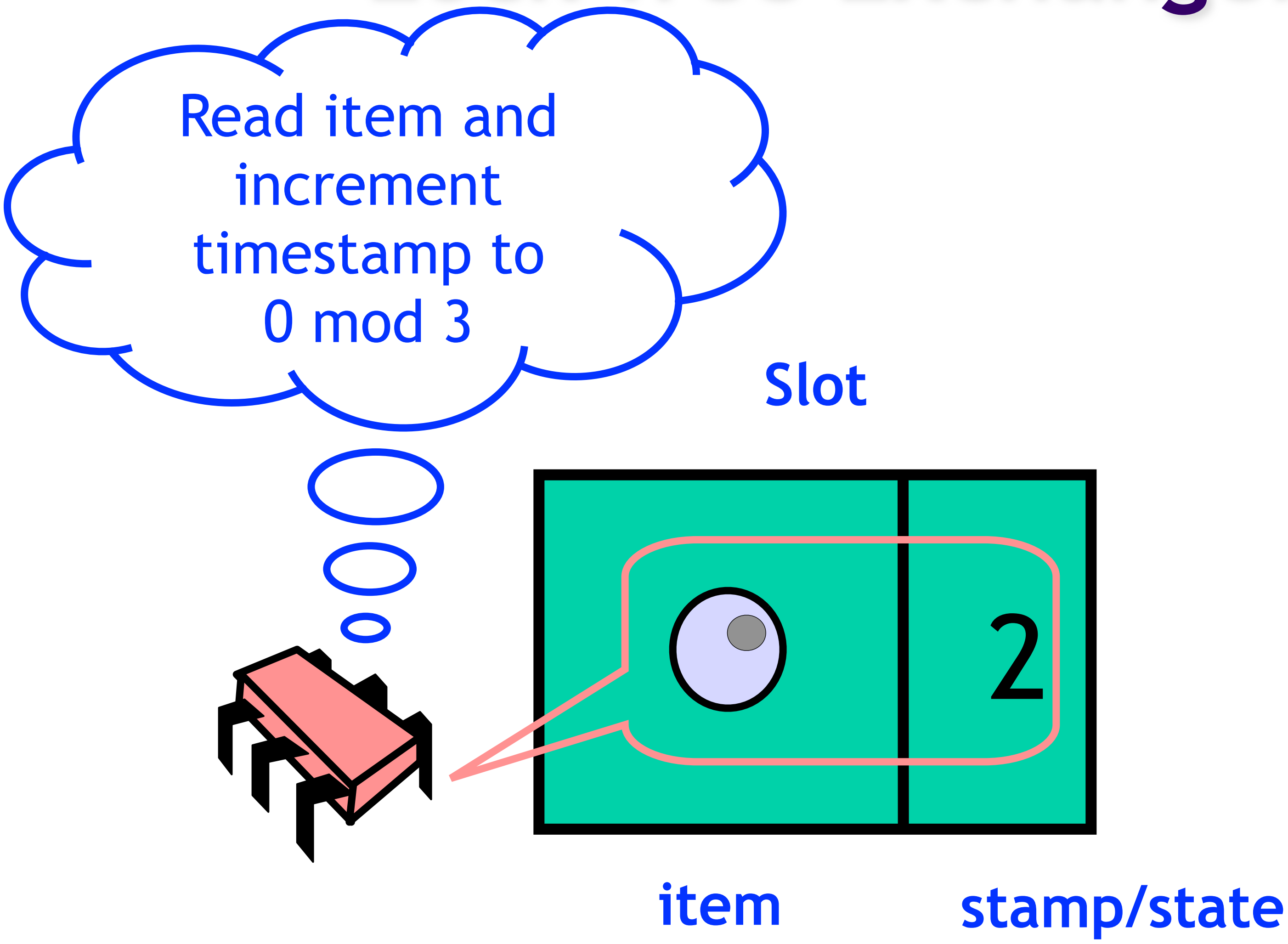
Slot

2

item

stamp/state

Lock-free Exchanger



Read item and increment timestamp to 0 mod 3

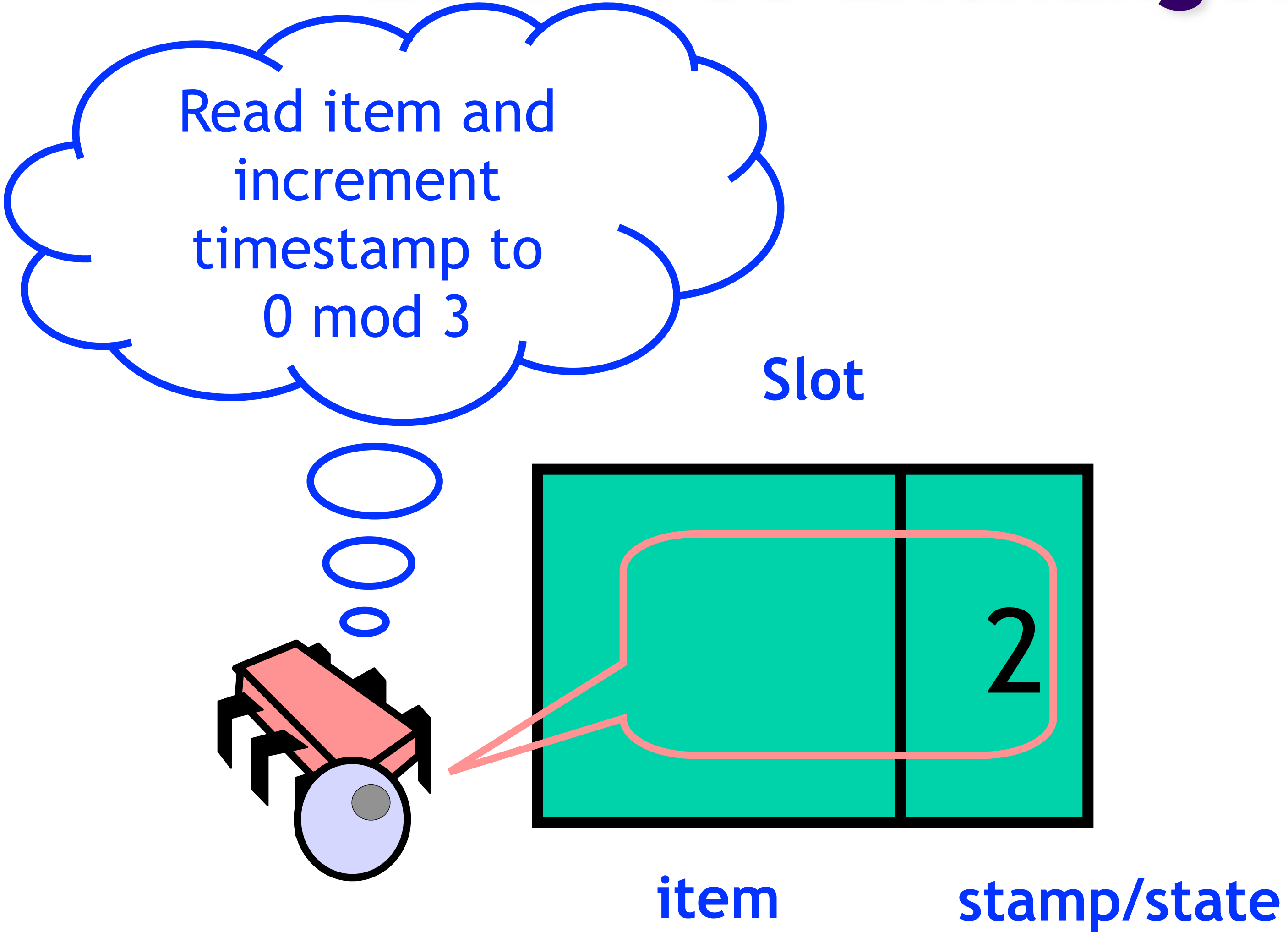
Slot

2

item

stamp/state

Lock-free Exchanger



Read item and increment timestamp to 0 mod 3

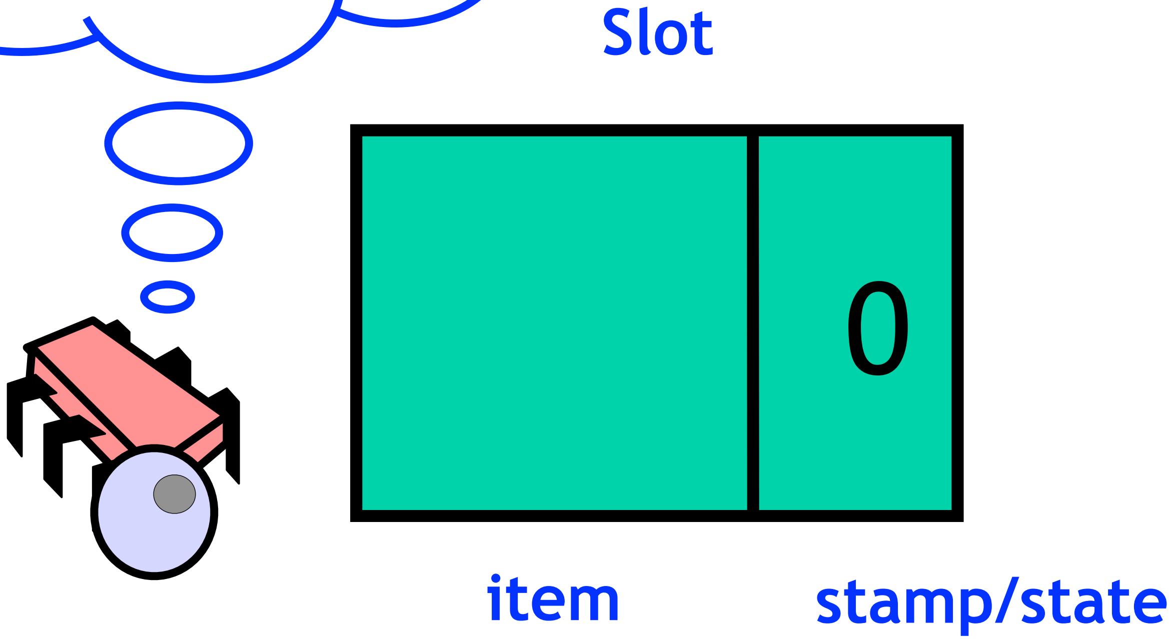
Slot

item

stamp/state

Lock-free Exchanger

Read item and increment timestamp to 0 mod 3



Exchanger State 0

```
case 0: // slot is free
    if (slot.compareAndSet(herItem, myItem, stamp, stamp + 1)) {
    while (System.nanoTime() < timeBound){
      herItem = slot.get(stampHolder);
      if (stampHolder[0] == stamp + 2) {
        slot.set(null, stamp + 3);
        return herItem;
      }}
    if (slot.compareAndSet(myItem, null, stamp + 1, stamp)) {throw new
TimeoutException();
     } else {
        herItem = slot.get(stampHolder);
        slot.set(null, stamp + 3);
        return herItem;
    }
} break;
```

Exchanger State 0

```
case 0: // slot is free
    if (slot.compareAndSet(herItem, myItem, stamp, stamp + 1)) {
    while (System.nanoTime() < timeBound){
      herItem = slot.get(stampHolder);
      if (stampHolder[0] == stamp + 2) {
        slot.set(null, stamp + 3);
        return herItem;
      }}
    if (slot.compareAndSet(myItem, null, stamp + 1, stamp)) {throw new
TimeoutException();
     } else {
        herItem = slot.get(stampHolder);
        slot.set(null, stamp + 3);
        return herItem;
    }
} break;
```

Slot is free, try and insert myItem and change state to 1

Exchanger State 0

```
case 0: // slot is free
    if (slot.compareAndSet(herItem, myItem, stamp, stamp + 1)) {
    while (System.nanoTime() < timeBound){
      herItem = slot.get(stampHolder);
      if (stampHolder[0] == stamp + 2) {
        slot.set(null, stamp + 3);
        return herItem;
      }}
    if (slot.compareAndSet(myItem, null, stamp + 1, stamp)) {throw new
TimeoutException();
     } else {
        herItem = slot.get(stampHolder);
        slot.set(null, stamp + 3);
        return herItem;
    }
} break;
```

Loop while still time left to try and exchange

Exchanger State 0

```
case 0: // slot is free
     if (slot.compareAndSet(herItem, myItem, stamp, stamp + 1)) {
     while (System.nanoTime() < timeBound){
       herItem = slot.get(stampHolder);
       if (stampHolder[0] == stamp + 2) {
         slot.set(null, stamp + 3);
         return herItem;
       }}
     if (slot.compareAndSet(myItem, null, stamp + 1, stamp)) {throw new
TimeoutException();
      } else {
         herItem = slot.get(stampHolder);
         slot.set(null, stamp + 3);
         return herItem;
     }
} break;
```

Get item and stamp in slot and check if state changed to 2

Exchanger State 0

```
case 0: // slot is free
    if (slot.compareAndSet(herItem, myItem, stamp, stamp + 1)) {
    while (System.nanoTime() < timeBound){
      herItem = slot.get(stampHolder);
      if (stampHolder[0] == stamp + 2) {
        slot.set(null, stamp + 3);
        return herItem;
      }}
    if (slot.compareAndSet(myItem, null, stamp + 1, stamp)) {throw new
TimeoutException();
     } else {
        herItem = slot.get(stampHolder);
        slot.set(null, stamp + 3);
        return herItem;
    }
} break;
```

If successful reset slot state to 0

Exchanger State 0

```
case 0: // slot is free
     if (slot.compareAndSet(herItem, myItem, stamp, stamp + 1)) {
     while (System.nanoTime() < timeBound){
       herItem = slot.get(stampHolder);
       if (stampHolder[0] == stamp + 2) {
         slot.set(null, stamp + 3);
         return herItem;
       }}
     if (slot.compareAndSet(myItem, null, stamp + 1, stamp)) {throw new
TimeoutException();
      } else {
         herItem = slot.get(stampHolder);
         slot.set(null, stamp + 3);
         return herItem;
     }
} break;
```

and return item found in slot

Exchanger State 0

```
case 0: // slot is free
    if (slot.compareAndSet(herItem, myItem, stamp, stamp + 1)) {
    while (System.nanoTime() < timeBound){
      herItem = slot.get(stampHolder);
      if (stampHolder[0] == stamp + 2) {
        slot.set(null, stamp + 3);
        return herItem;
      }}
    if (slot.compareAndSet(myItem, null, stamp + 1, stamp)) {throw new
TimeoutException();
     } else {
        herItem = slot.get(stampHolder);
        slot.set(null, stamp + 3);
        return herItem;
    }
} break;
```

Otherwise we ran out of time, try and reset state to 0, if successful time out

Exchanger State 0

```
case 0: // slot is free
    if (slot.compareAndSet(herItem, myItem, stamp, stamp + 1)) {
    while (System.nanoTime() < timeBound){
      herItem = slot.get(stampHolder);
      if (stampHolder[0] == stamp + 2) {
        slot.set(null, stamp + 3);
        return herItem;
      }}
    if (slot.compareAndSet(myItem, null, stamp + 1, stamp)) {throw new
TimeoutException();
    } else {
        herItem = slot.get(stampHolder);
        slot.set(null, stamp + 3);
        return herItem;
    }
} break;
```

If reset failed can only be that someone showed up after all, take her item

Exchanger State 0

```
case 0: // slot is free
    if (slot.compareAndSet(herItem, myItem, stamp, stamp + 1)) {
    while (System.nanoTime() < timeBound){
      herItem = slot.get(stampHolder);
      if (stampHolder[0] == stamp + 2) {
        slot.set(null, stamp + 3);
        return herItem;
      }}
    if (slot.compareAndSet(myItem, null, stamp + 1, stamp)) {throw new
TimeoutException();
     } else {
        herItem = slot.get(stampHolder);
        slot.set(null, stamp + 3);
        return herItem;
    }
} break;
```

Set slot to 0 with new time stamp and return the item found

Exchanger State 0

```
case 0: // slot is free
     if (slot.compareAndSet(herItem, myItem, stamp, stamp + 1)) {
     while (System.nanoTime() < timeBound){
       herItem = slot.get(stampHolder);
       if (stampHolder[0] == stamp + 2) {
         slot.set(null, stamp + 3);
         return herItem;
       }}
     if (slot.compareAndSet(myItem, null, stamp + 1, stamp)) {throw new
TimeoutException();
      } else {
         herItem = slot.get(stampHolder);
         slot.set(null, stamp + 3);
         return herItem;
      }
} break;
```

If initial CAS failed then someone else changed slot from 0 to 1 so retry from start

Exchanger States 1 and 2

```
case 1:   // someone waiting for me
        if (slot.compareAndSet(herItem, myItem, stamp, stamp + 1))
          return herItem;
        break;
case 2:   // others in middle of exchanging
        break;
default:  // impossible
        break;
      }
    }
  }
}
```

Exchanger States 1 and 2

```
case 1:   // someone waiting for me
      if (slot.compareAndSet(herItem, myItem, stamp, stamp + 1))
        return herItem;
      break;
case 2:   // others in middle of exchanging
      break;
default:  // impossible
      break;
     }
    }
  }
}
```

state 1 means someone is waiting for an exchange, so attempt to CAS my Item in and change state to 2

Exchanger States 1 and 2

```
case 1:   // someone waiting for me
      if (slot.compareAndSet(herItem, myItem, stamp, stamp + 1))
        return herItem;
      break;
case 2:   // others in middle of exchanging
      break;
default:  // impossible
      break;
     }
   }
  }
}
```

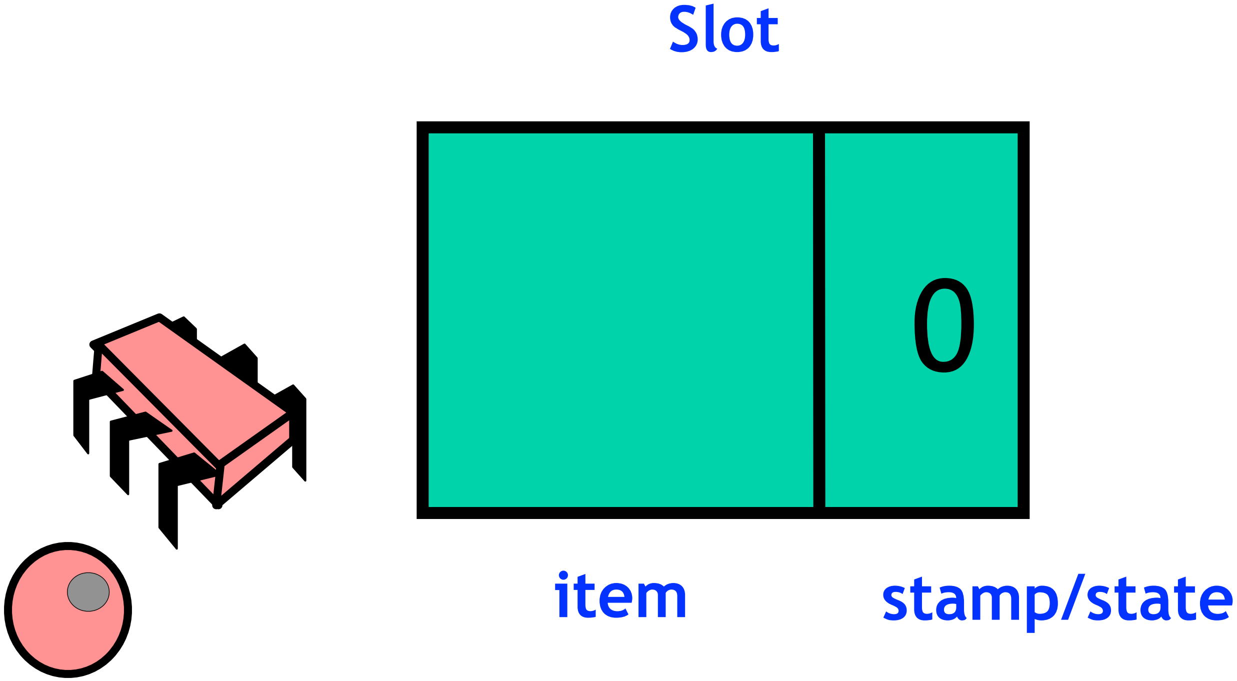
If successful return her item, state is now 2, otherwise someone else took her item so try again from start

Exchanger States 1 and 2

```
case 1:   // someone waiting for me
        if (slot.compareAndSet(herItem, myItem, stamp, stamp + 1))
          return herItem;
        break;
case 2:   // others in middle of exchanging
        break;
default:  // impossible
        break;
      }
    }
  }
}
```

If state is 2 then some other threads are using slot to exchange so start again

Lock-free Exchanger



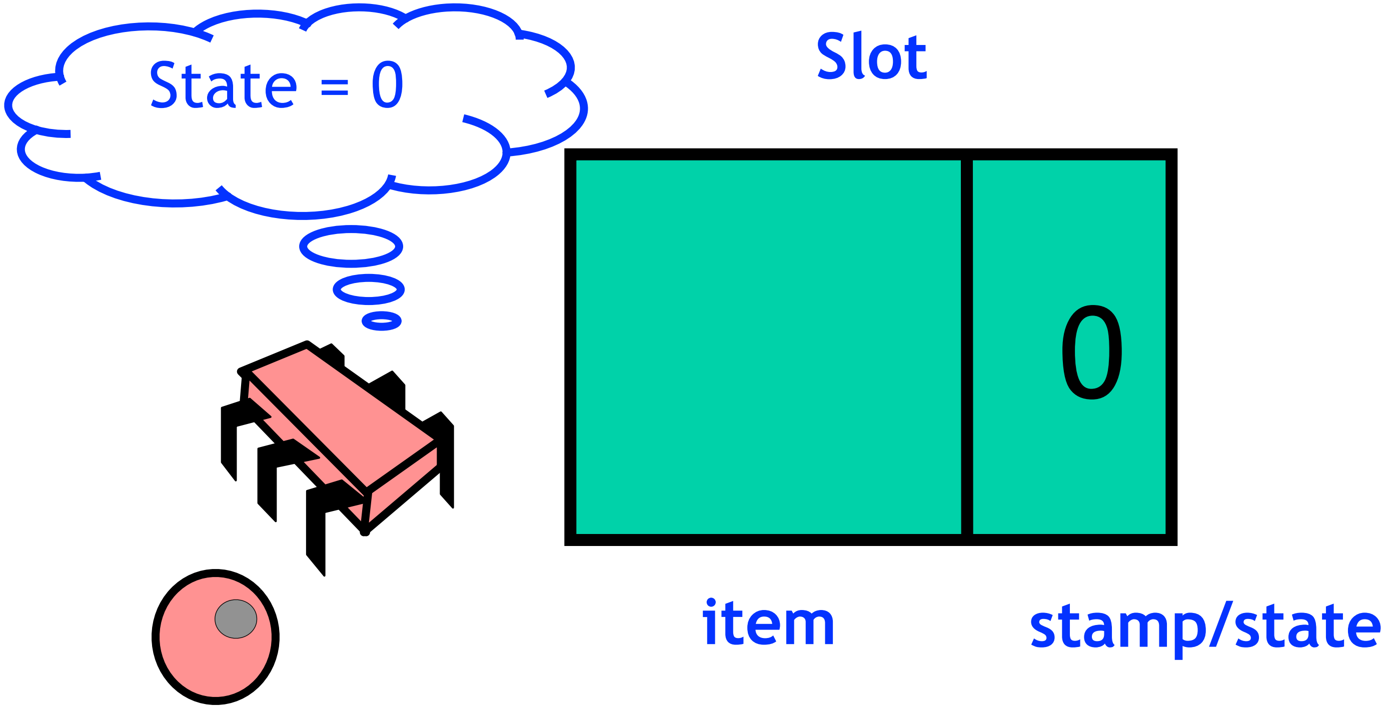
Slot

0

item

stamp/state

Lock-free Exchanger



State = 0

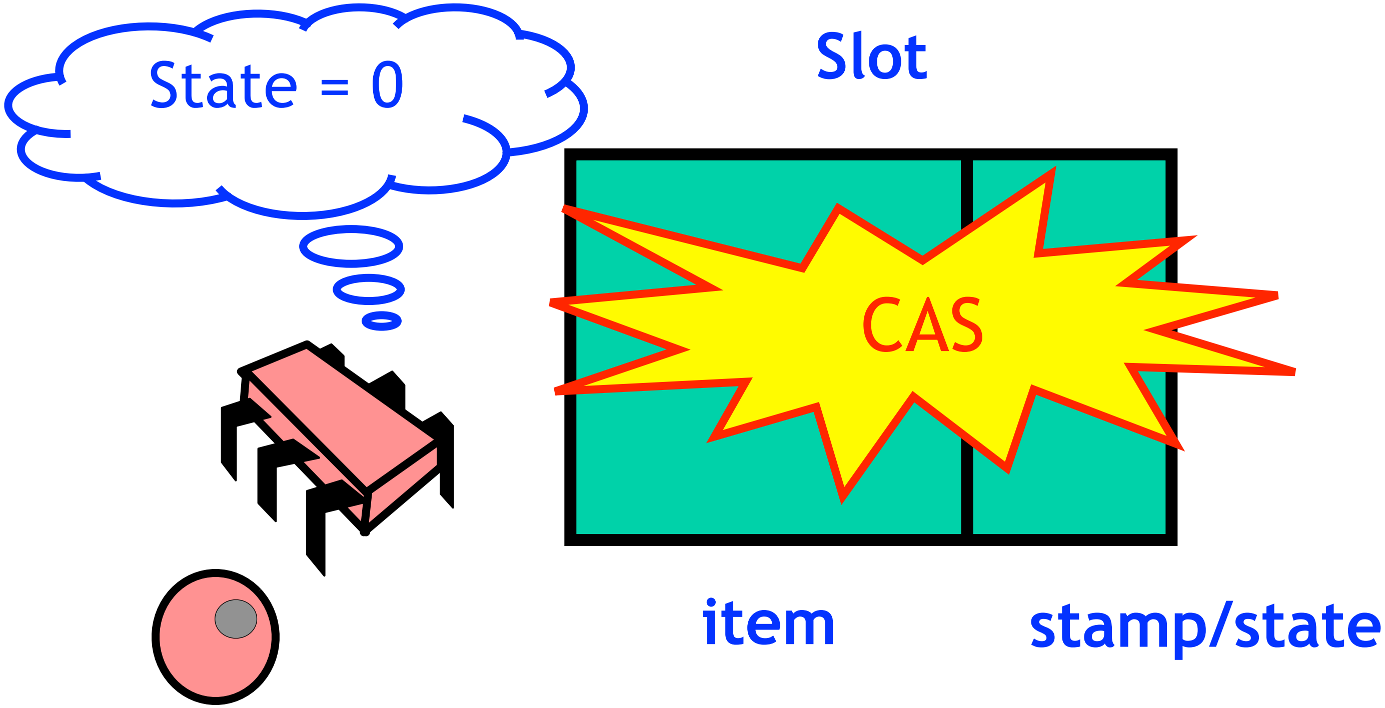
Slot

0

item

stamp/state

Lock-free Exchanger



State = 0

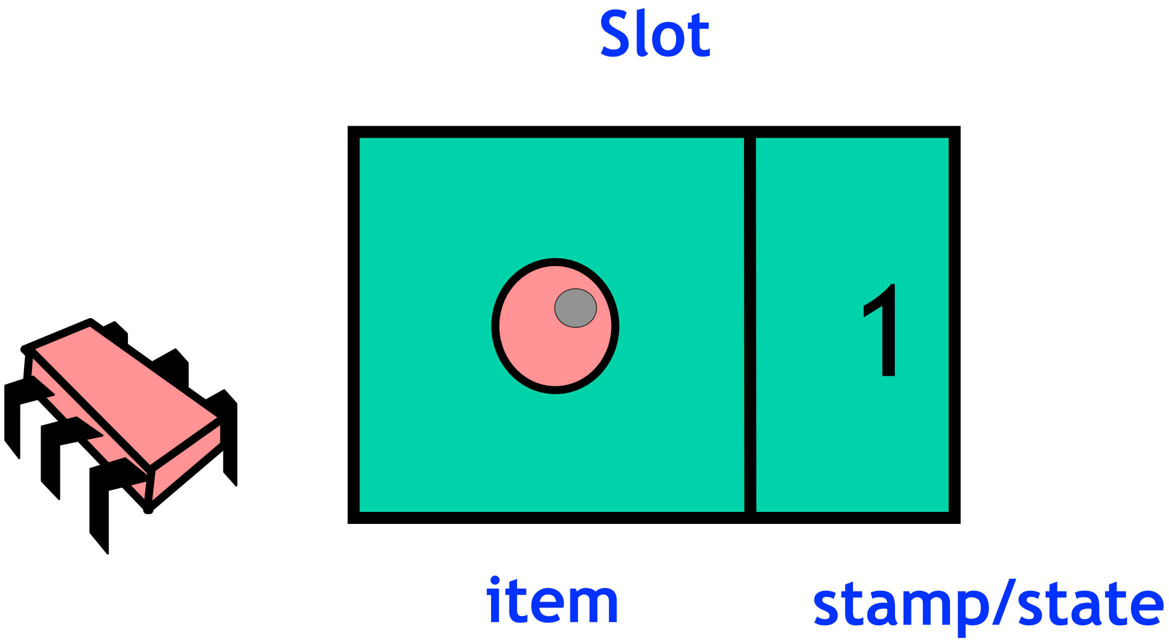
Slot

CAS

item

stamp/state

Lock-free Exchanger



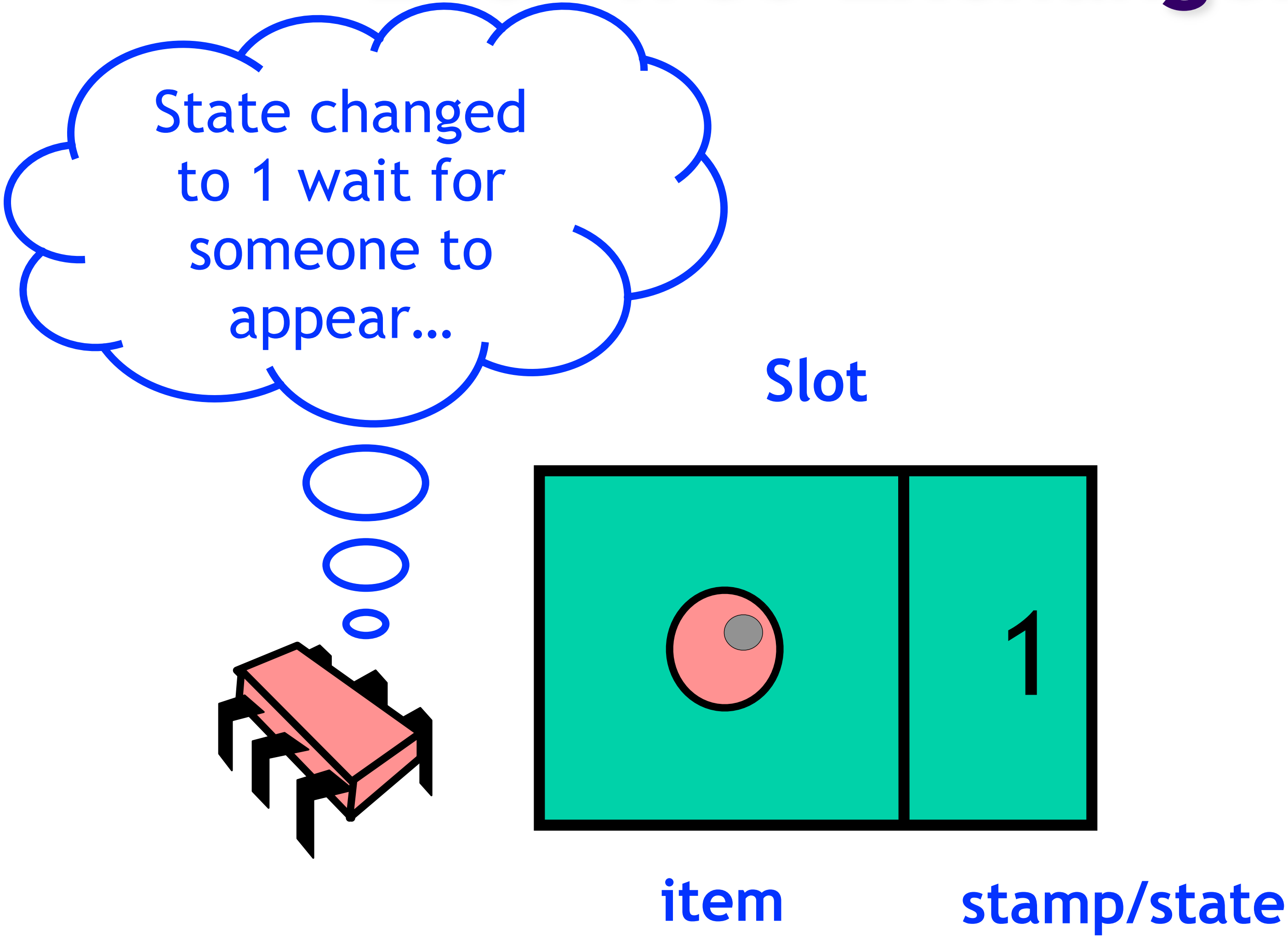
Slot

1

item

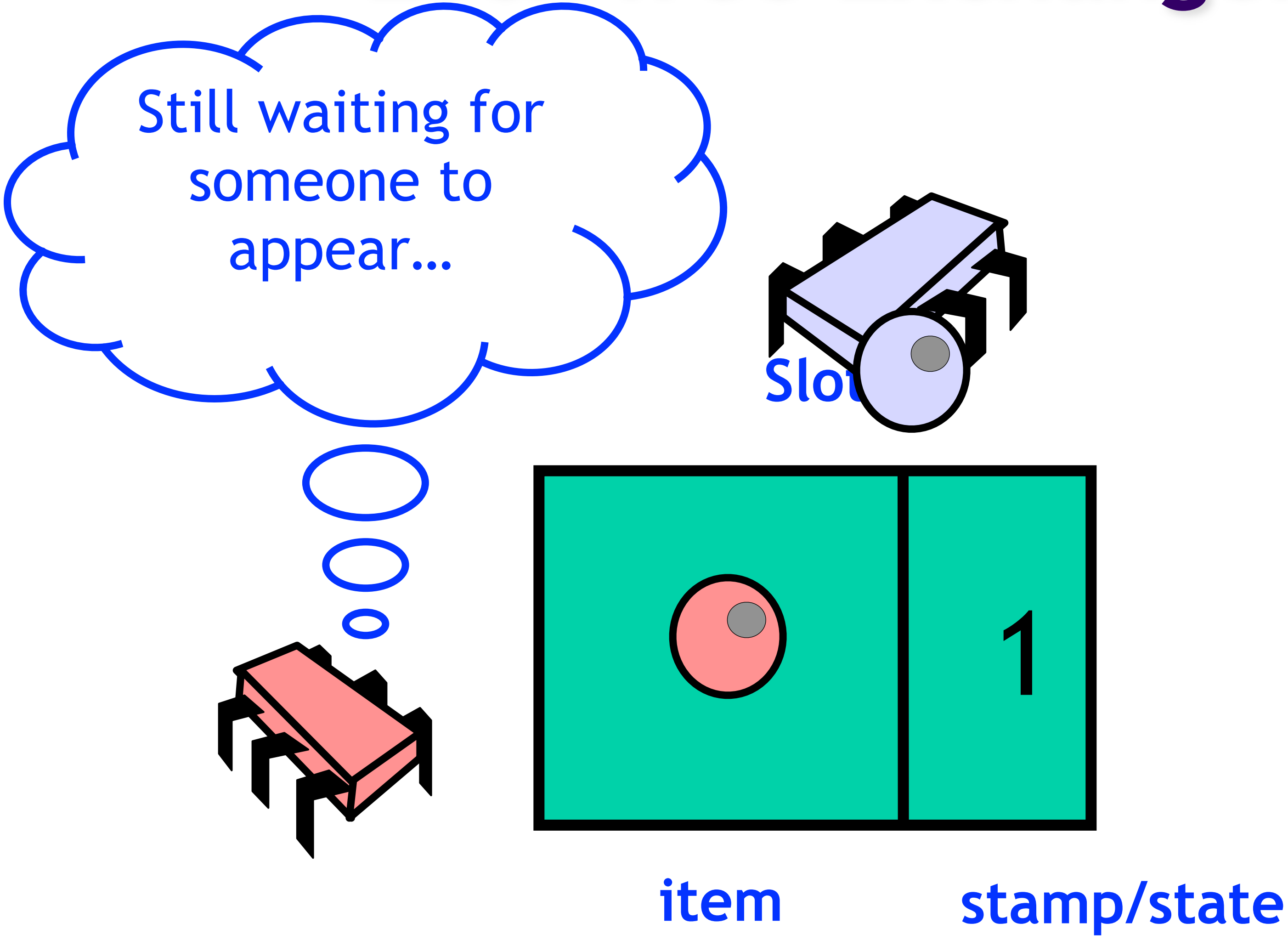
stamp/state

Lock-free Exchanger



State changed to 1 wait for someone to appear…

Lock-free Exchanger



Still waiting for someone to appear…

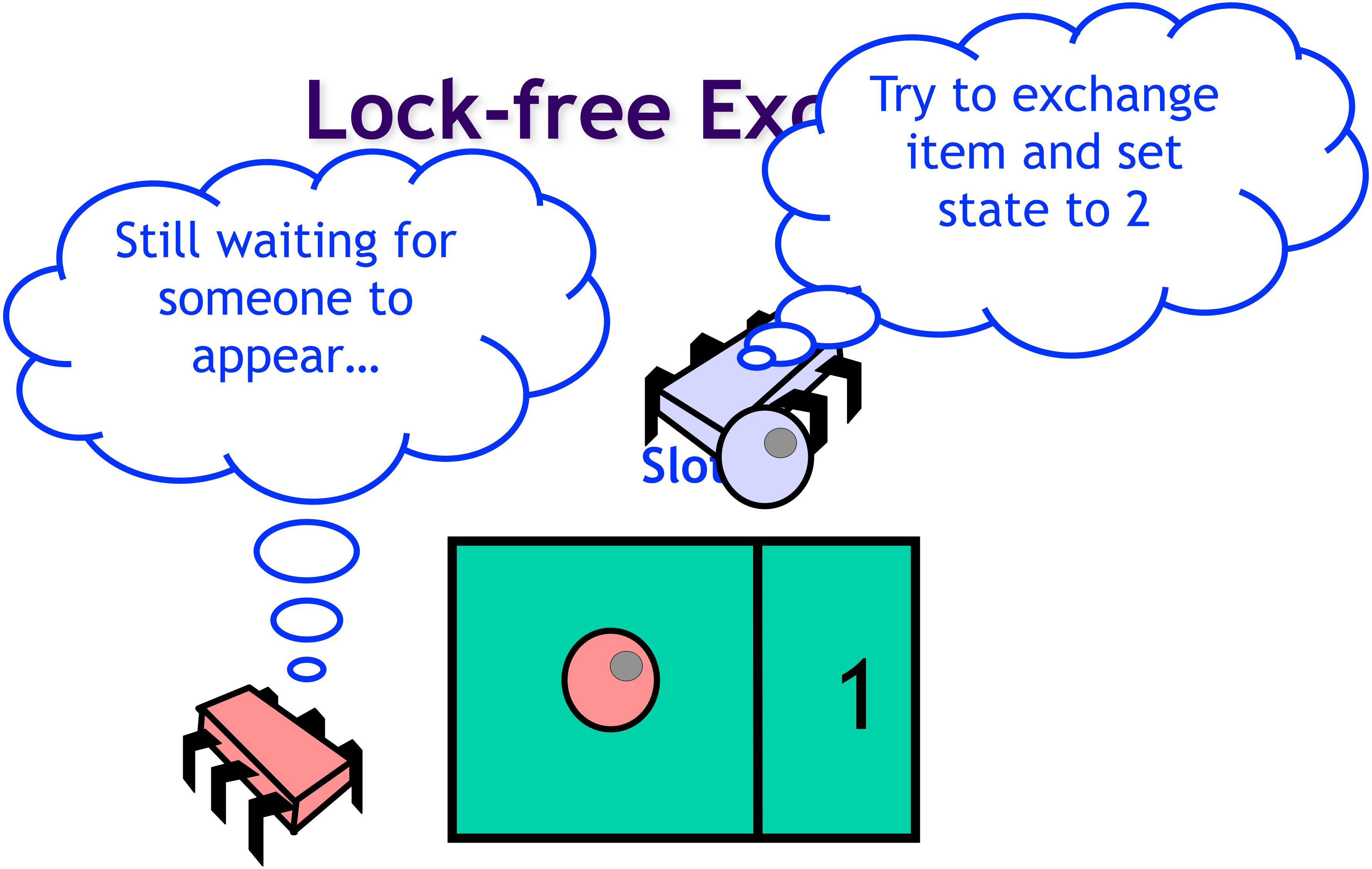
Slot

1

item

stamp/state

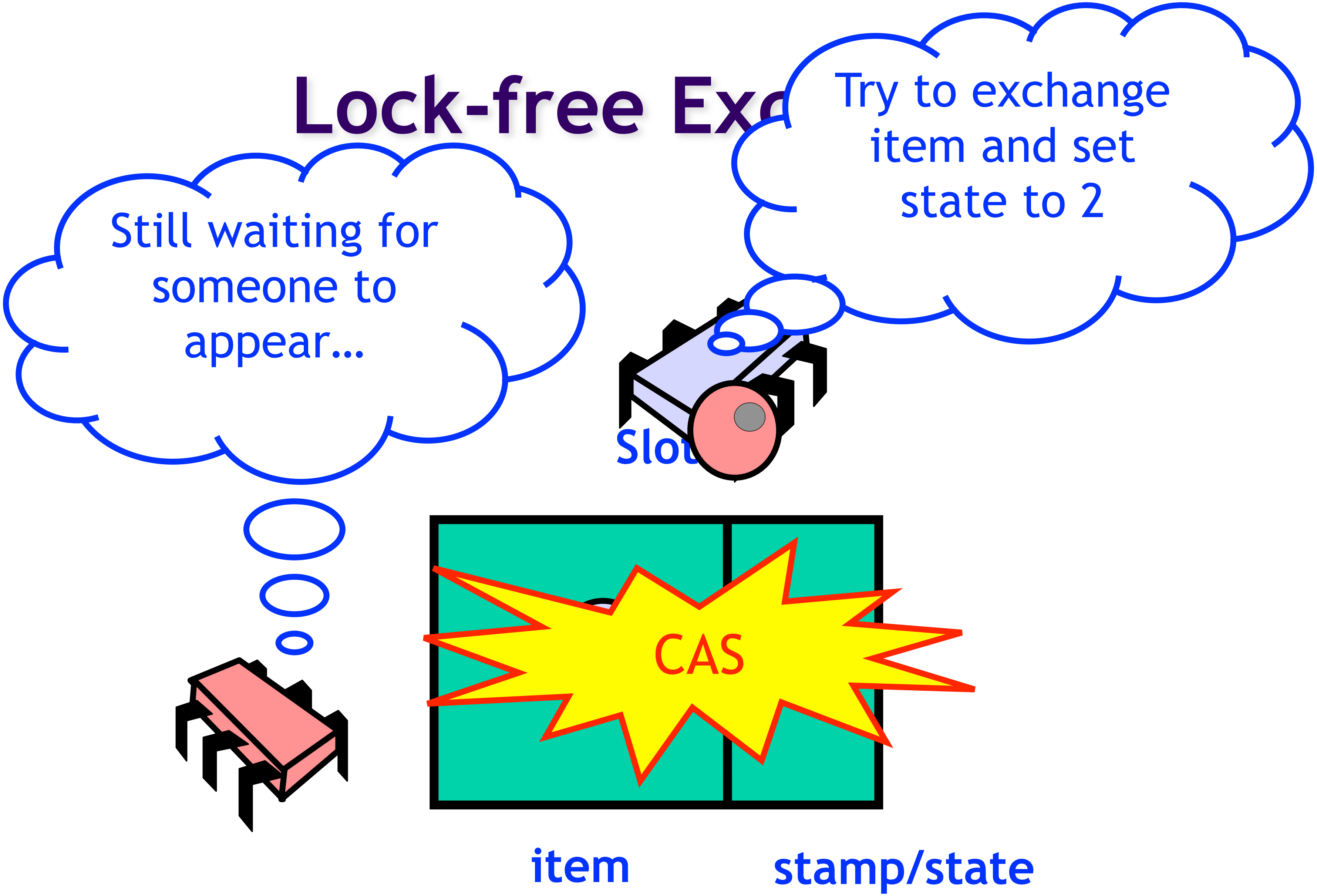
Lock-free Exc



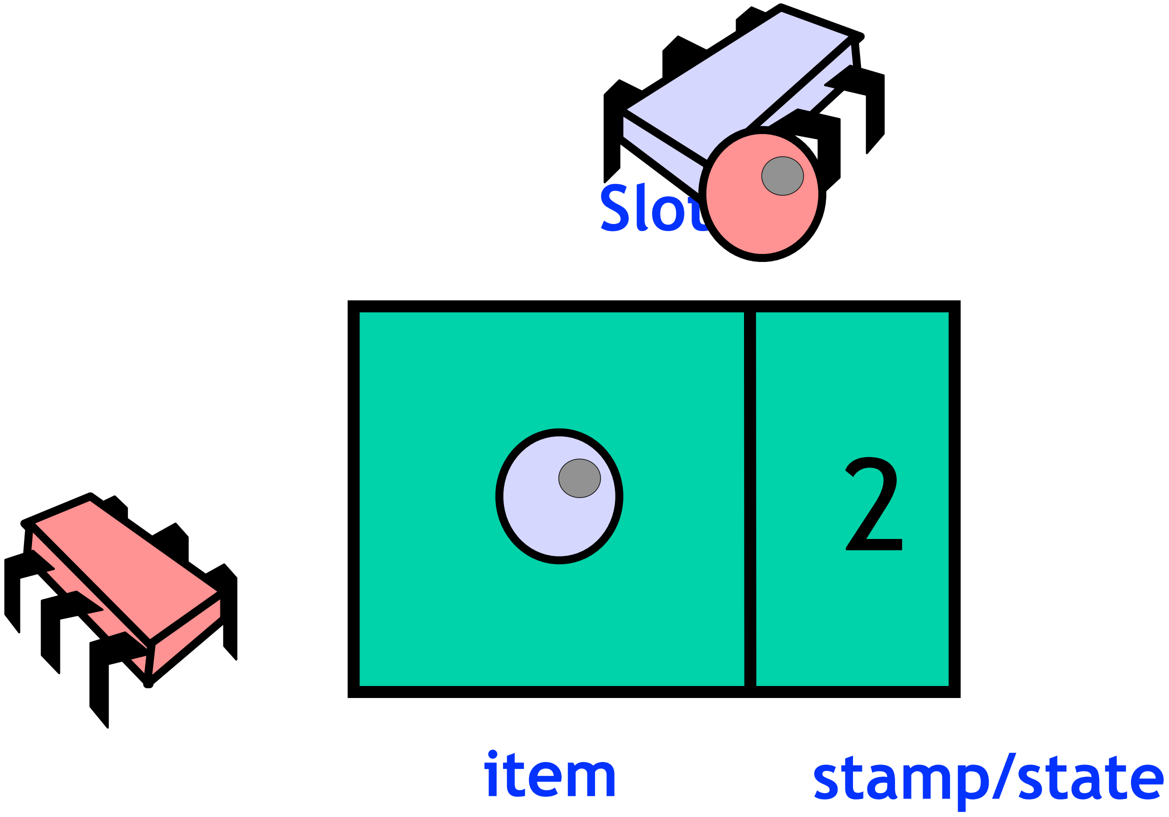
Still waiting for someone to appear…

Try to exchange item and set state to 2

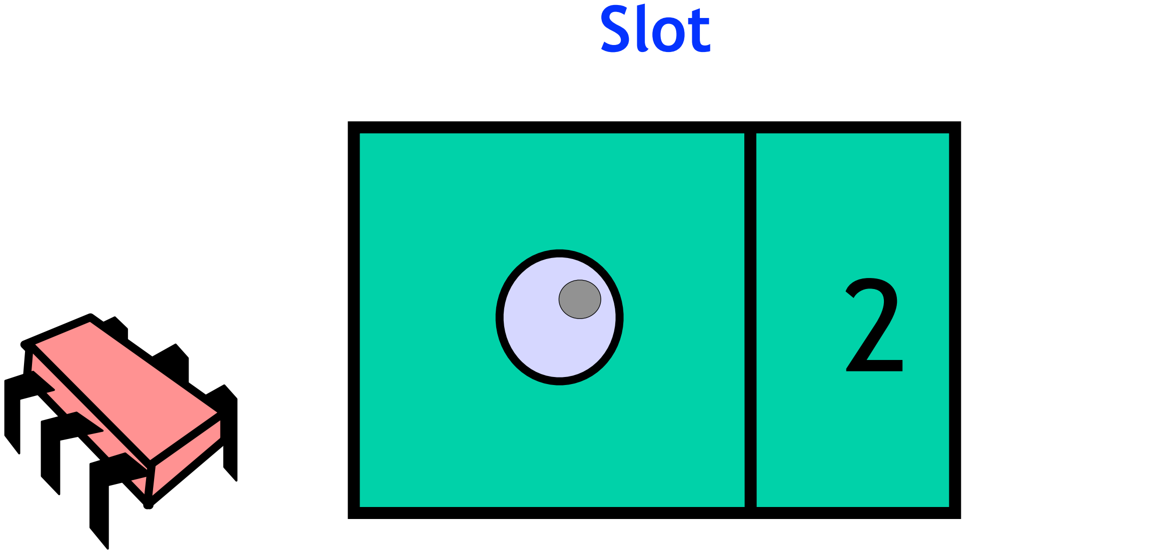
Lock-free Exc


Still waiting for someone to appear…
Try to exchange item and set state to 2
Slot
CAS
item
stamp/state

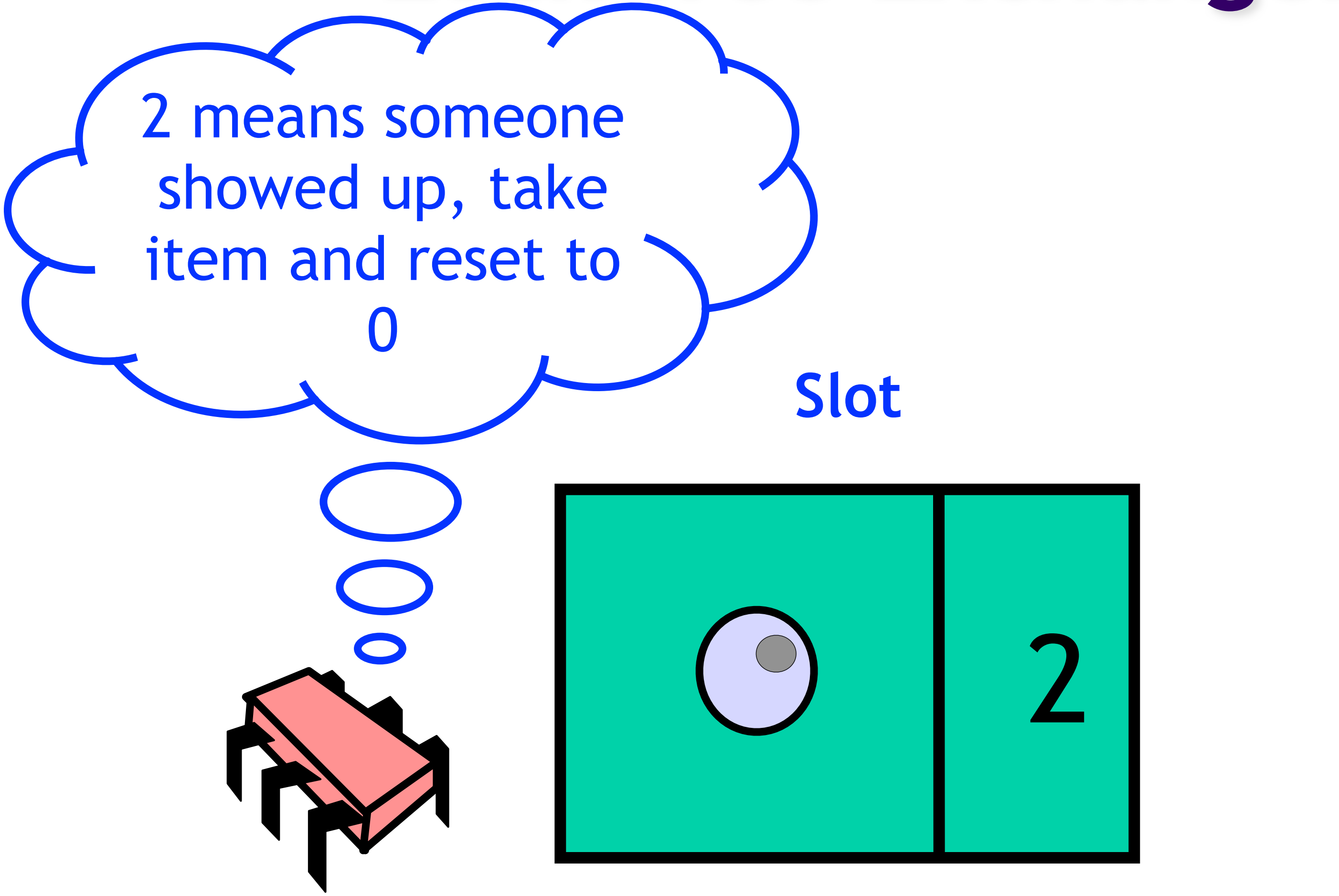
Lock-free Exchanger



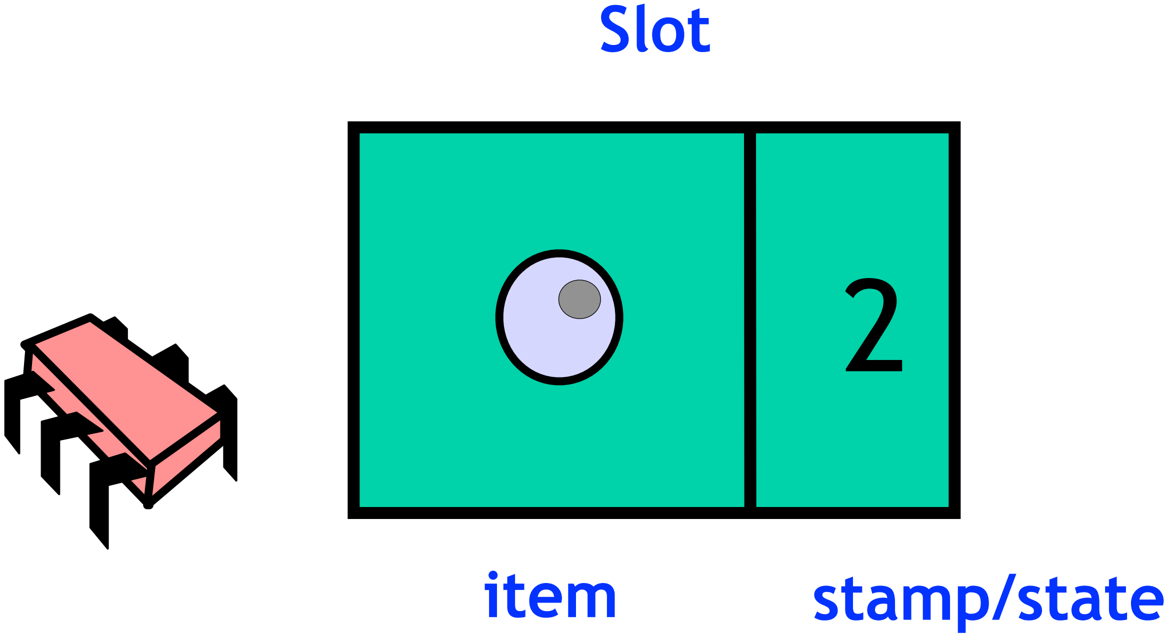
Lock-free Exchanger



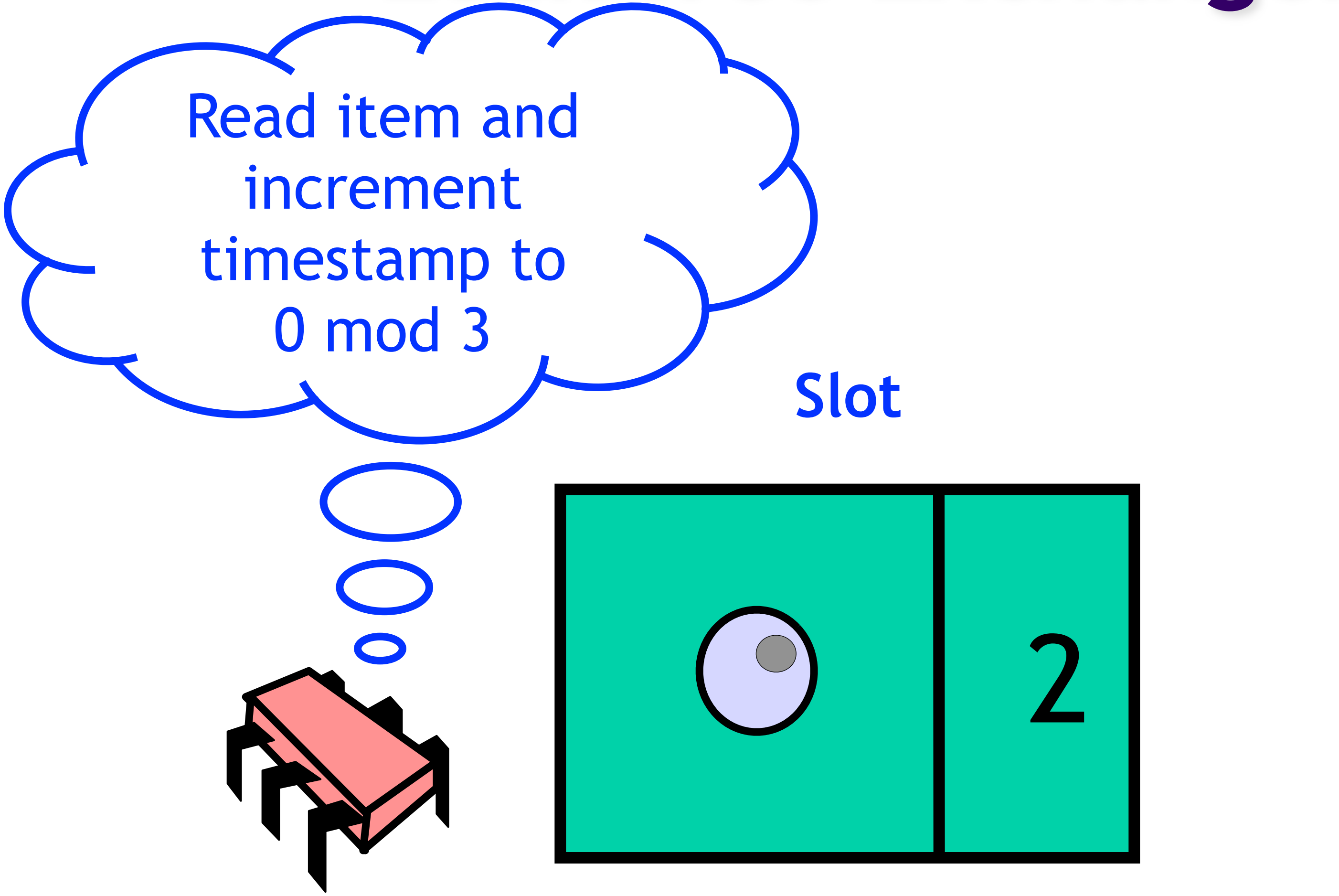
Lock-free Exchanger



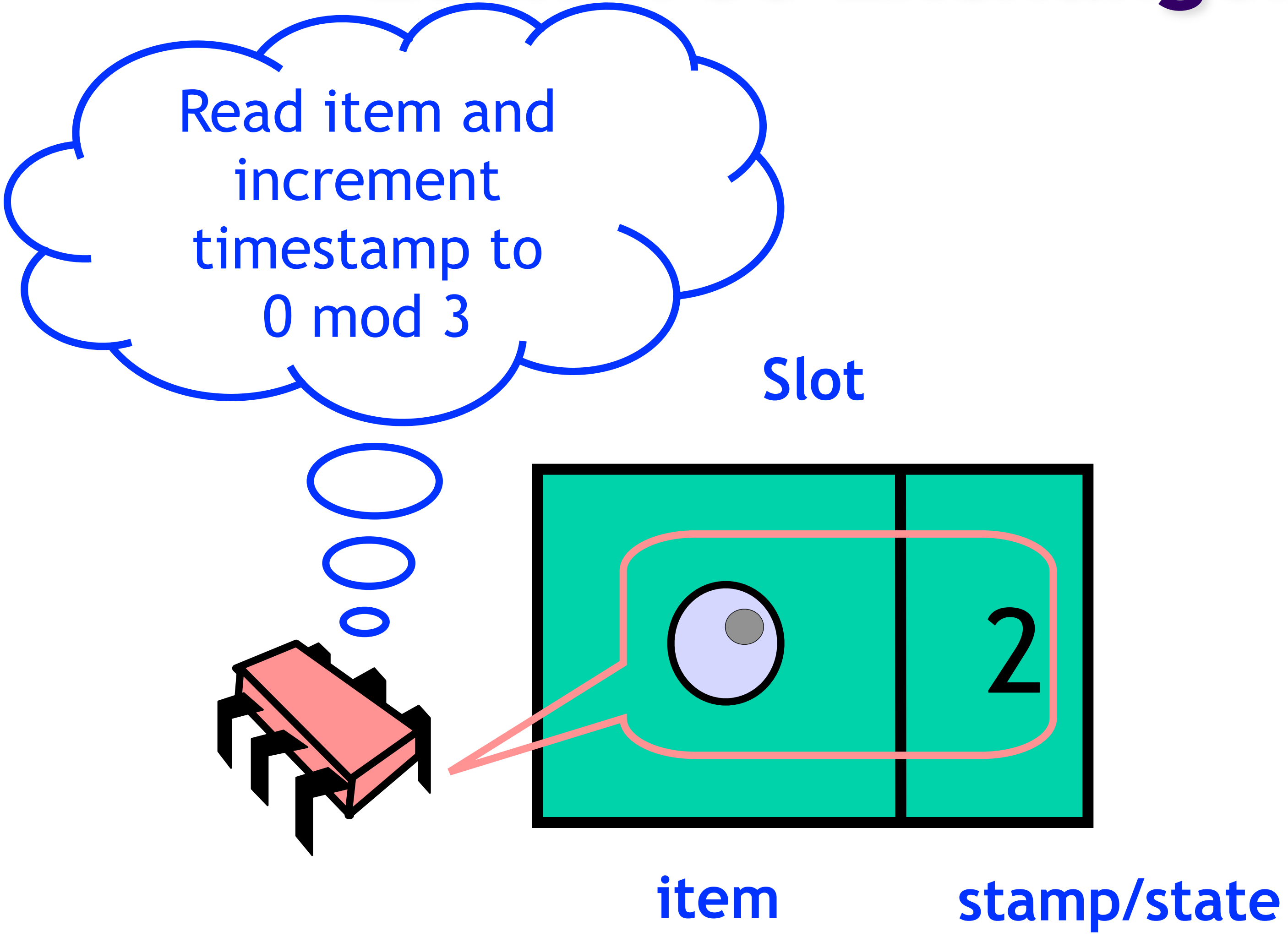
Lock-free Exchanger



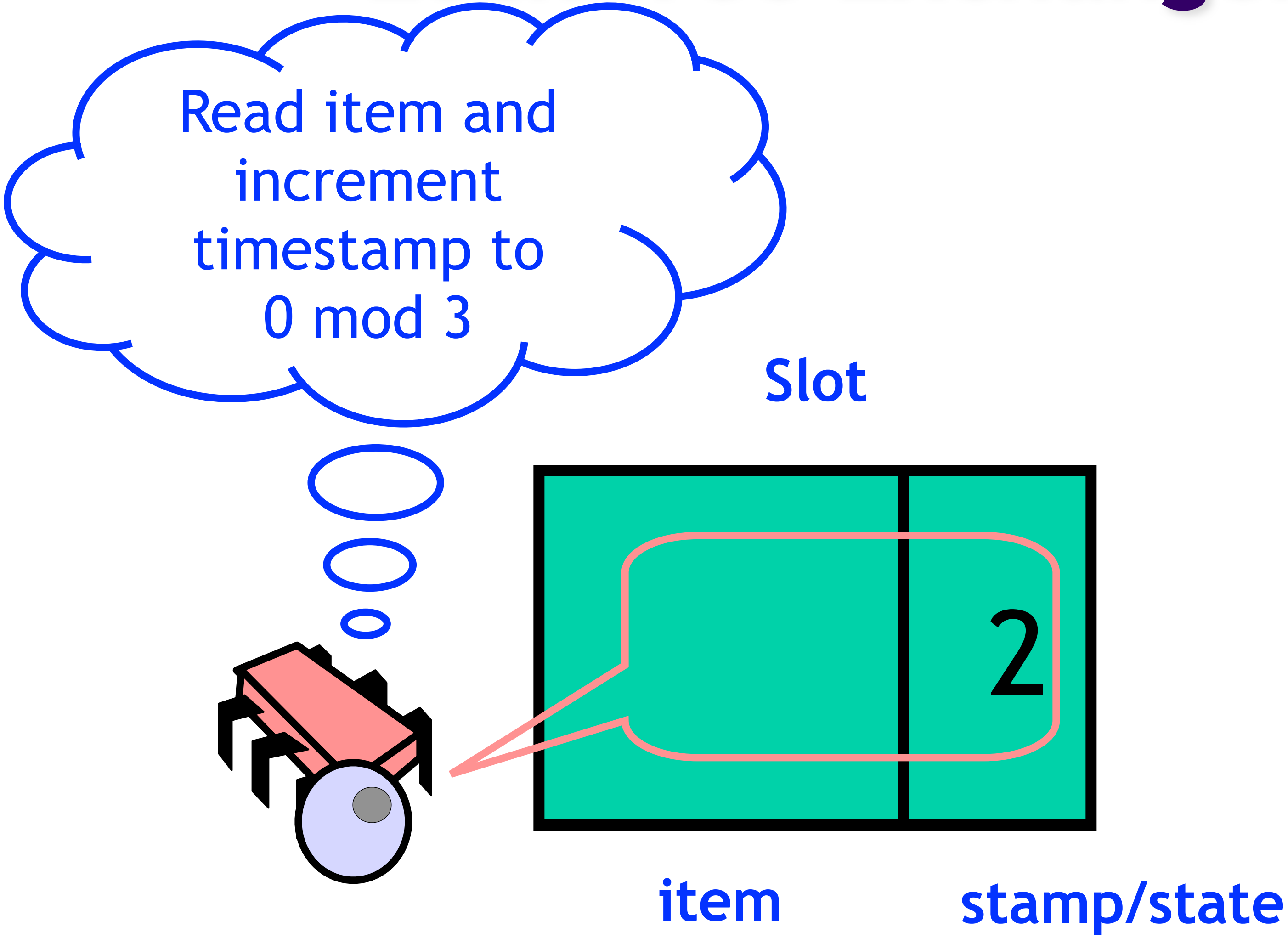
Lock-free Exchanger



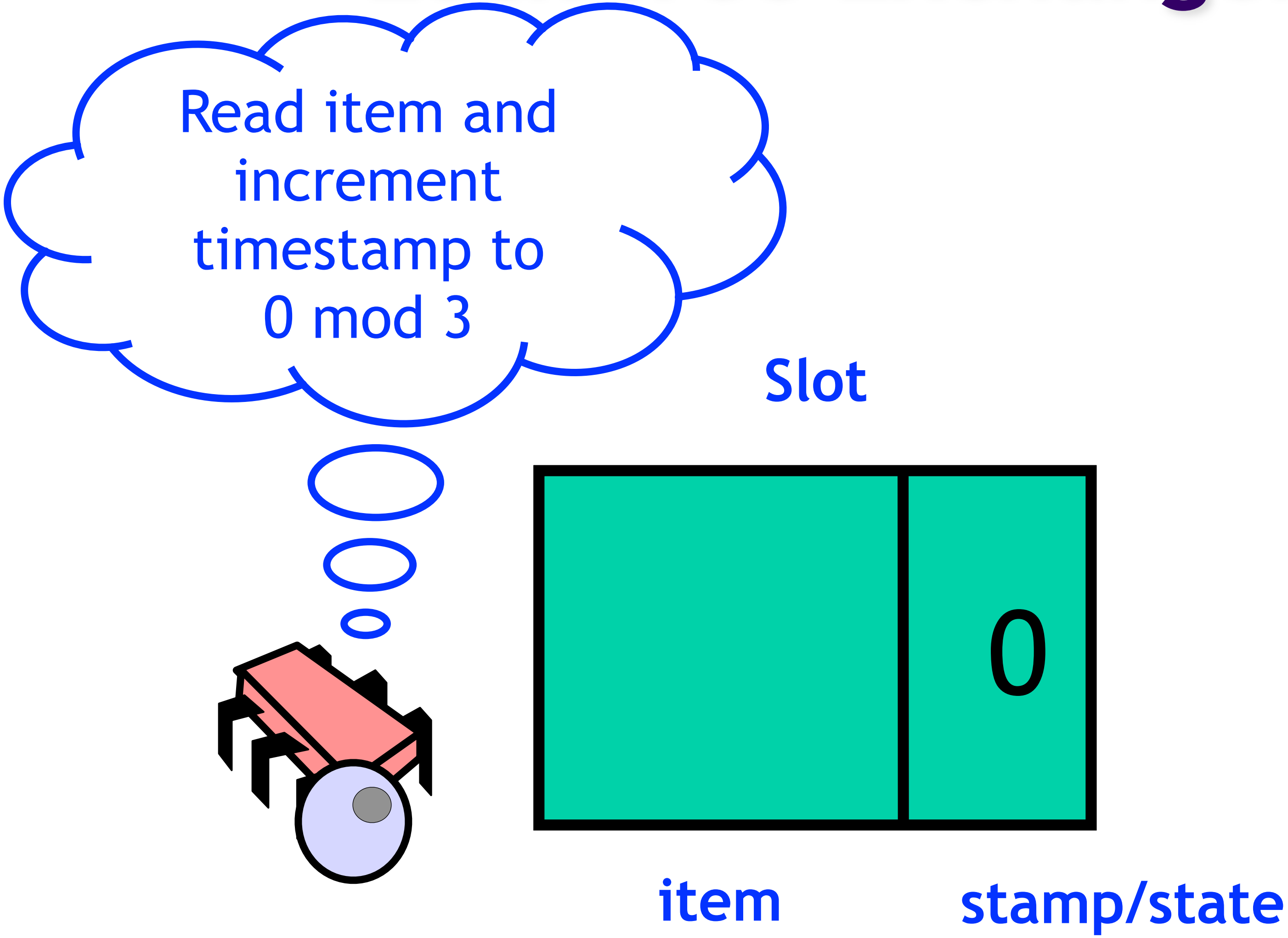
Lock-free Exchanger



Lock-free Exchanger



Lock-free Exchanger



Read item and increment timestamp to 0 mod 3

Slot

0

item

stamp/state

Our Exchanger Slot

- Notice that we showed a general lock-free exchanger
- Its lock-free because the only way an exchange can fail is if others repeatedly succeeded or no-one showed up
- The slot we need does not require symmetric exchange

Elimination Array

```
public class EliminationArray {
…
public T visit(T value, int Range) throws TimeoutException {
    int slot = random.nextInt(Range);
    int nanodur = convertToNanos(duration, timeUnit))
    return (exchanger[slot].exchange(value, nanodur )
}}
```

Elimination Array

```
public class EliminationArray {
...
public T visit(T value, int Range) throws TimeoutException {
    int slot = random.nextInt(Range);
    return (exchanger[slot].exchange(value, nanodur))
}}
```

visit the elimination array with a value and a range (duration to wait is not dynamic)

Elimination Array

Pick a random array entry

```
public class EliminationArray {
…
public T visit(T value, int Range) throws TimeoutException {
   int slot = random.nextInt(Range)
   return (exchanger[slot].exchange(value, nanodur))
}}
```

Elimination Array

```
public class EliminationArray {
…
public T visit(T value, int Range) throws TimeoutException {
    int slot = random.nextInt(Range)
    return (exchanger[slot].exchange(value, nanodur))
}}
```

Exchange value or time out

Elimination Stack Push

```
public void push(T value) {
...
 while (true) {
  if (tryPush(node)) {
    return;
  } else try {
      T otherValue =
eliminationArray.visit(value,policy.Range);
      if (otherValue == null) {
        return;
    }
}
```

Elimination Stack Push

```
public void push(T value) {
...
 while (true) {
  if (tryPush(node)) {
    return;
  } else try {
      T otherValue =
eliminationArray.visit(value,policy.Range);
      if (otherValue == null) {
         return;
    }
}
```

First try to push

Elimination Stack Push

```
public void push(T value) {
...
 while (true) {
  if (tryPush(node)) {
    return;
  } else try {
      T otherValue =
eliminationArray.visit(value,policy.Range);
      if (otherValue == null) {
        return;
    }
}
```

If failed back-off to try and eliminate

Elimination Stack Push

```
public void push(T value) {
...
 while (true) {
  if (tryPush(node)) {
    return;
  } else try {
      T otherValue =
eliminationArray.visit(value,policy.Range);
      if (otherValue == null) {
         return;
    }
}
```

Value being pushed and range to try

Elimination Stack Push

```
public void push(T value) {
...
 while (true) {
  if (tryPush(node)) {
    return;
  } else try {
      T otherValue =
eliminationArray.visit(value,policy.Range);
      if (otherValue == null) {
        return;
    }
}
```

Only a pop has null value
so elimination was successful

Elimination Stack Push

```
public void push(T value) {
...
 while (true) {
  if (tryPush(node)) {
    return;
  } else try {
      T otherValue =
eliminationArray.visit(value,policy.Range);
      if (otherValue == null) {
         return;
    }
}
```

Else retry push on lock-free stack

Elimination Stack Pop

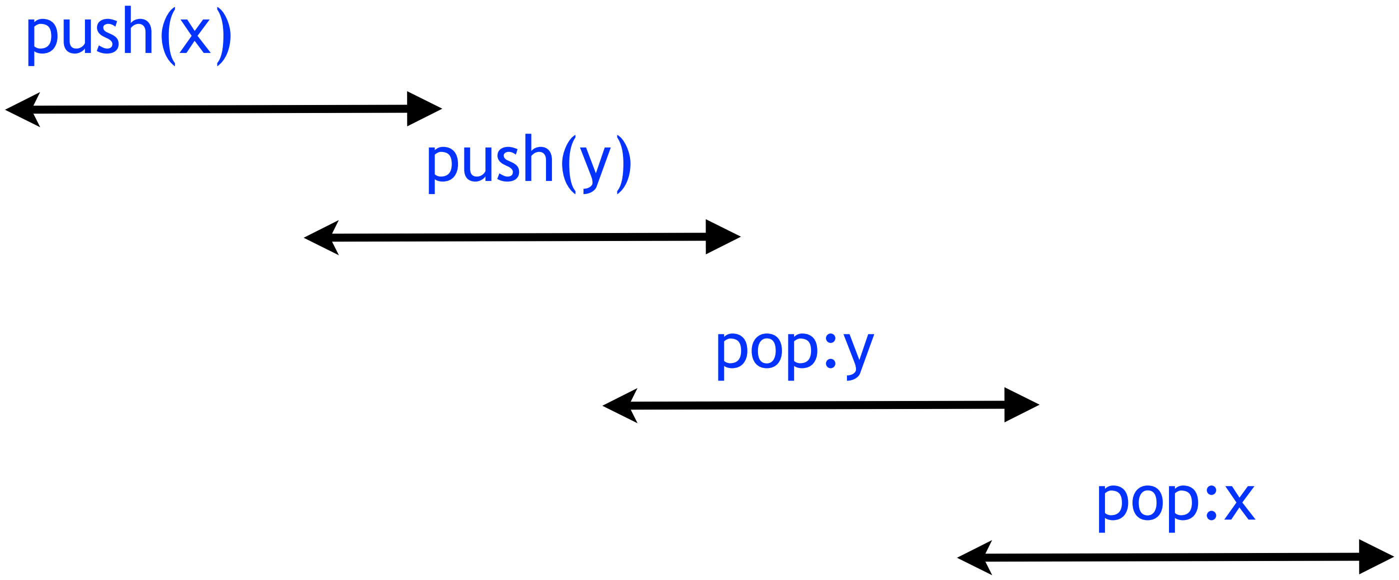
```
public T pop() {
 ...
 while (true) {
  if (tryPop()) {
   return returnNode.value;
   } else
      try {
        T otherValue = eliminationArray.visit(null,policy.Range);
        if ( otherValue != null) {
         return otherValue;
         }
      }
}}
```

Elimination Stack Pop

```
public T pop() {
 ...
 while (true) {
  if (tryPop()) {
   return returnNode.value;
   } else
      try {
        T otherValue = eliminationArray.visit(null,policy.Range;
        if ( otherValue != null) {
         return otherValue;
         }
      }
}}
```

If non-null other thread must have pushed, so elimination succeeds

Linearizability



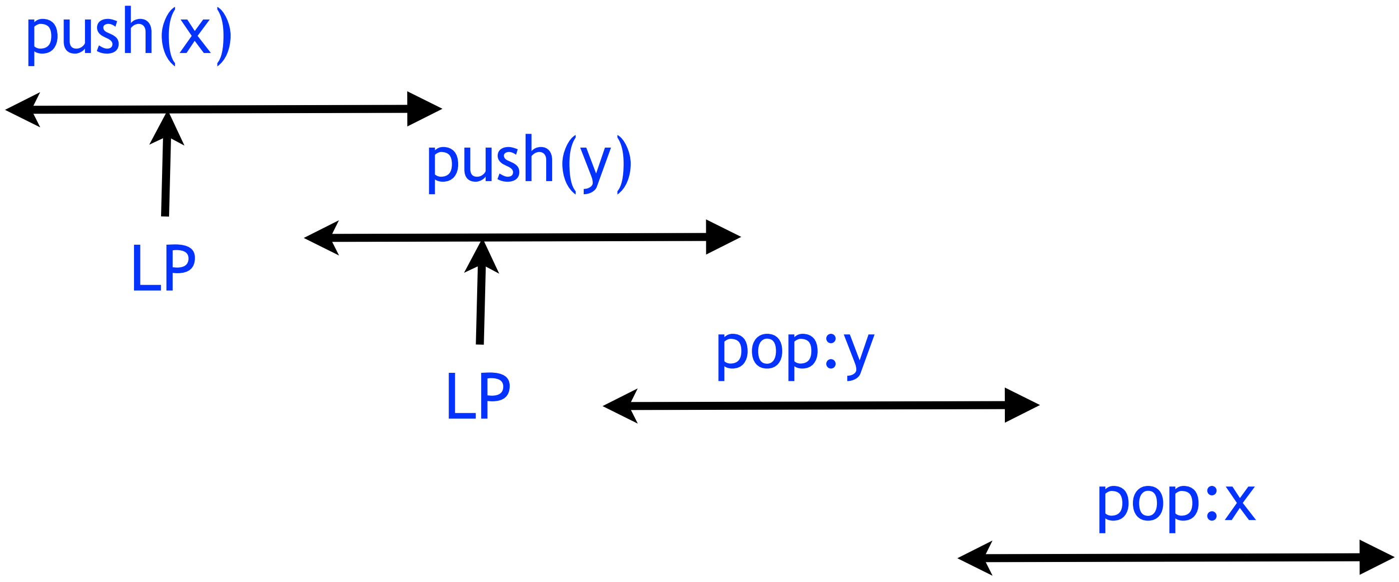
push(x)

push(y)

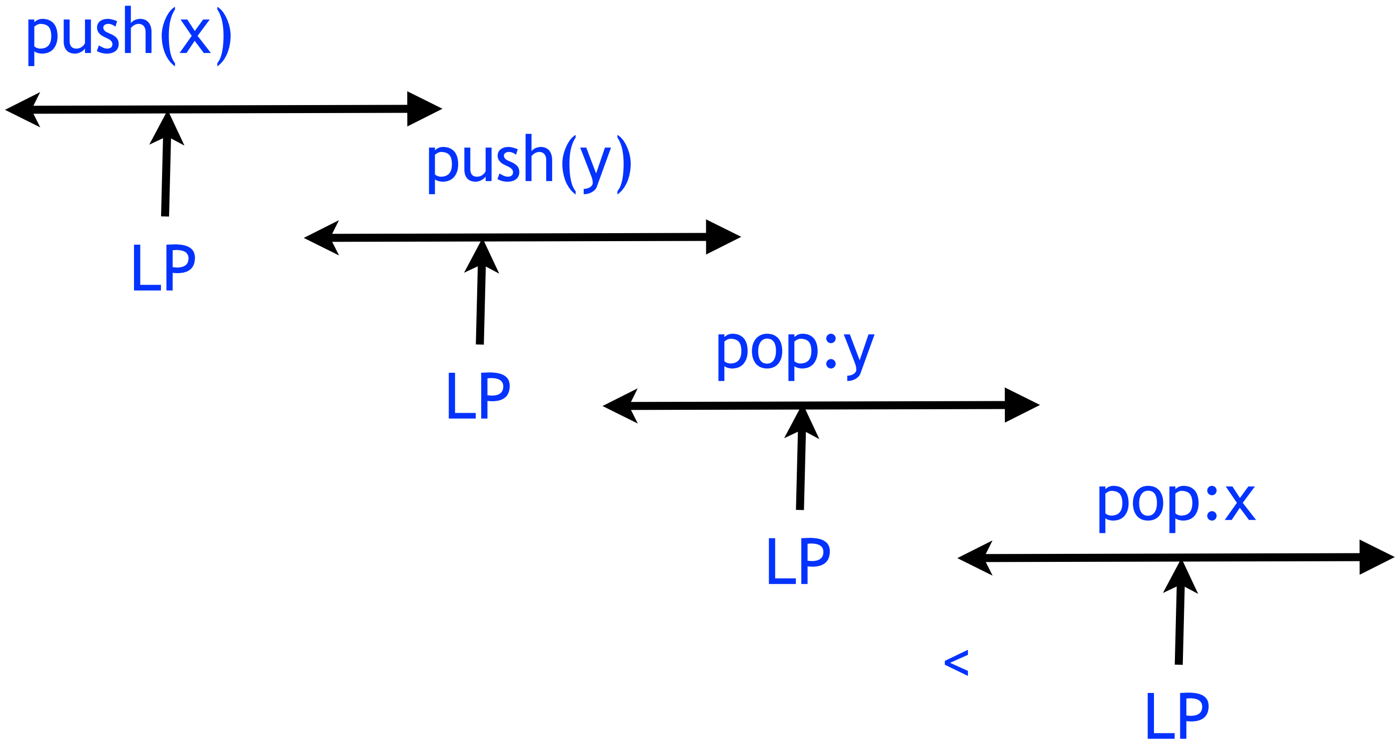
pop:y

pop:x

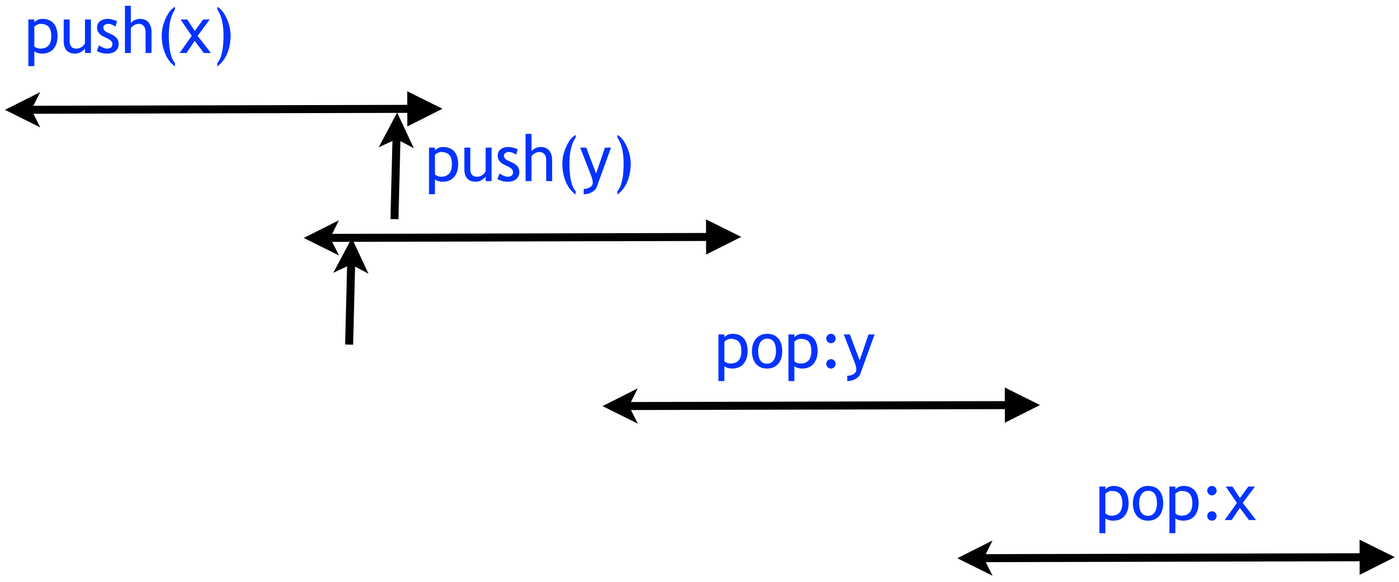
Linearizability



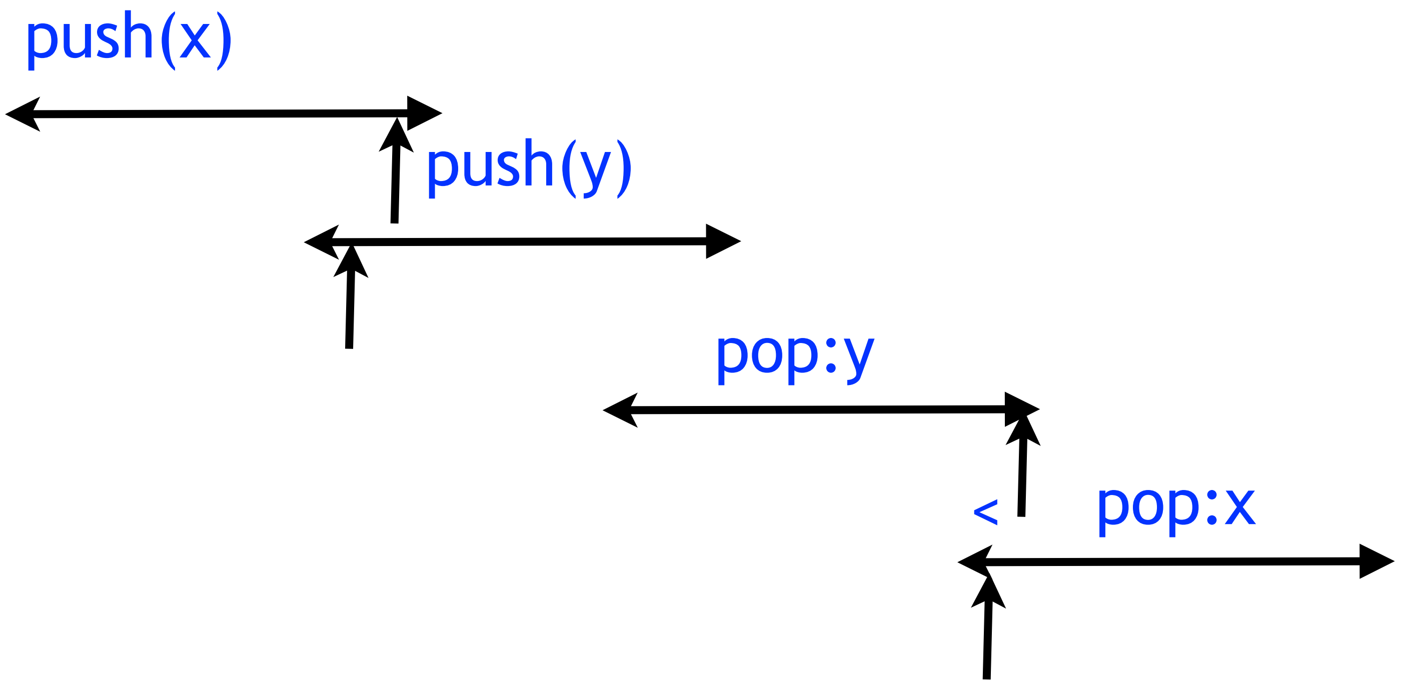
Linearizability



Linearizability



Linearizability



push(x)

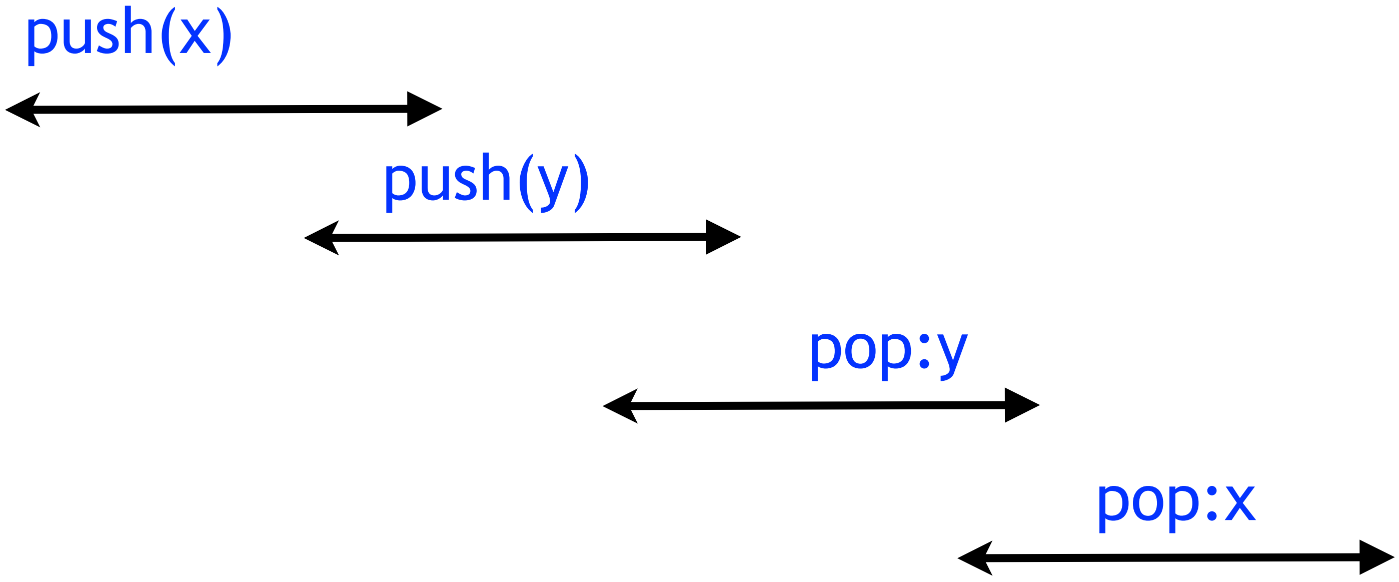
push(y)

pop:y

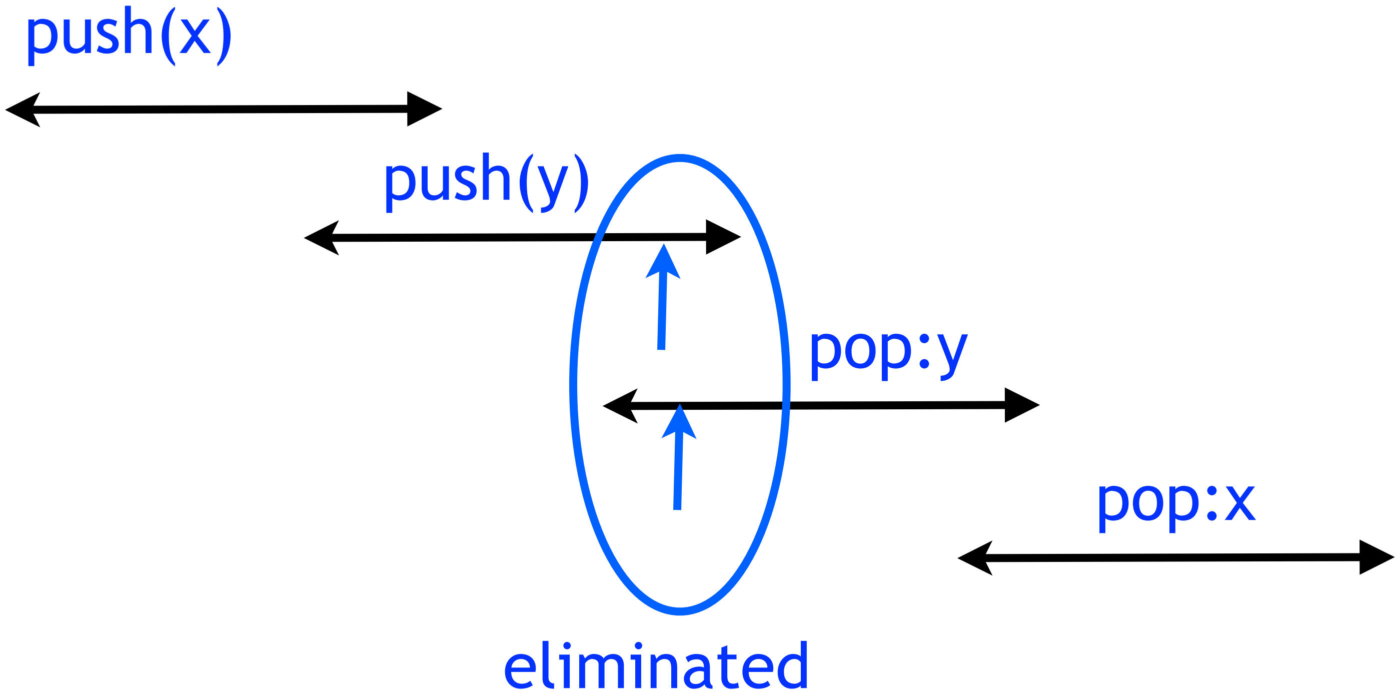
<

pop:x

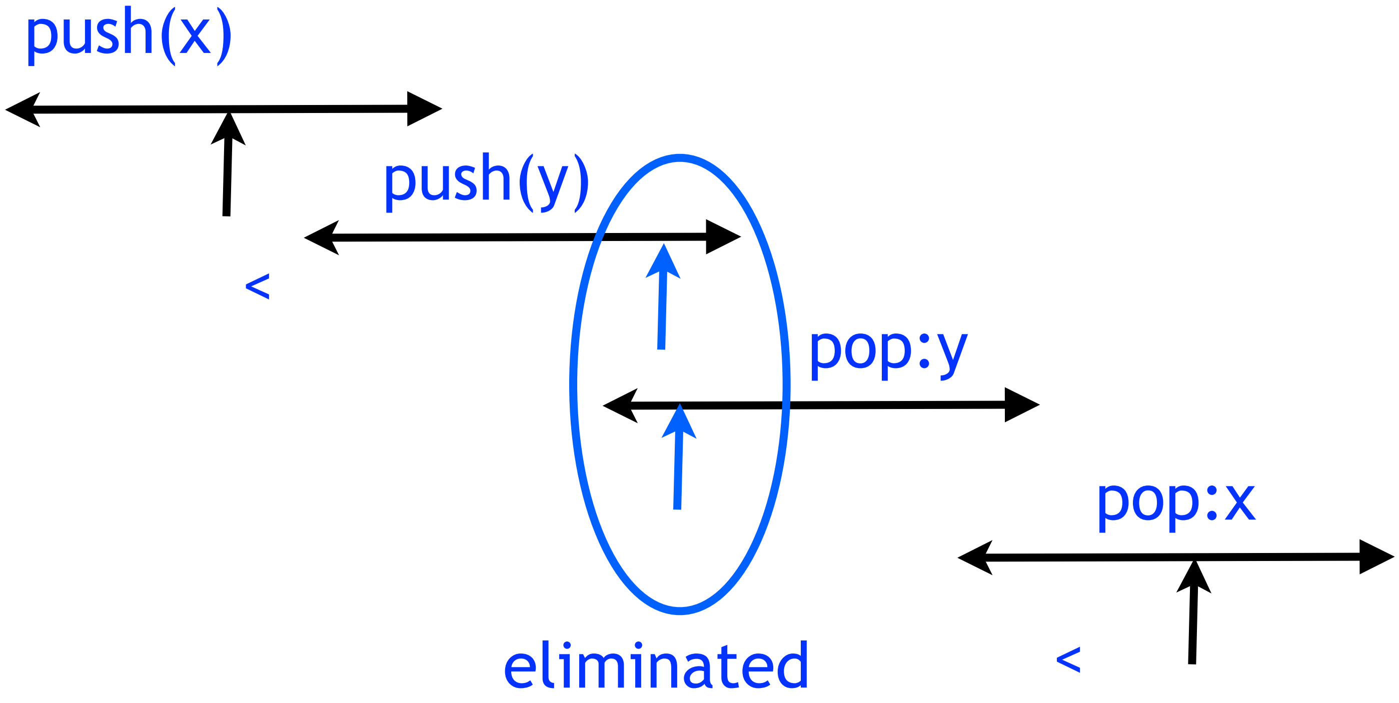
Elimination



Elimination



Elimination



push(x)

push(y)

<

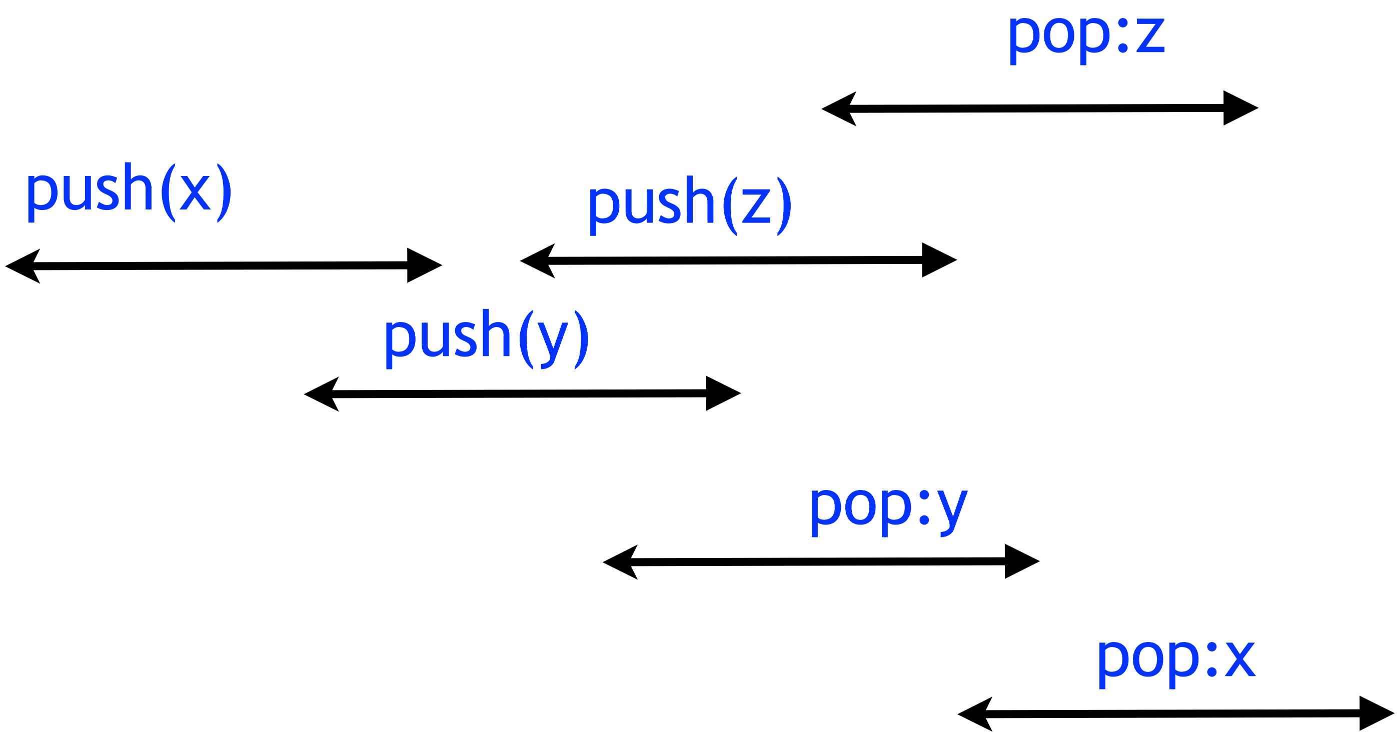
pop:y

pop:x

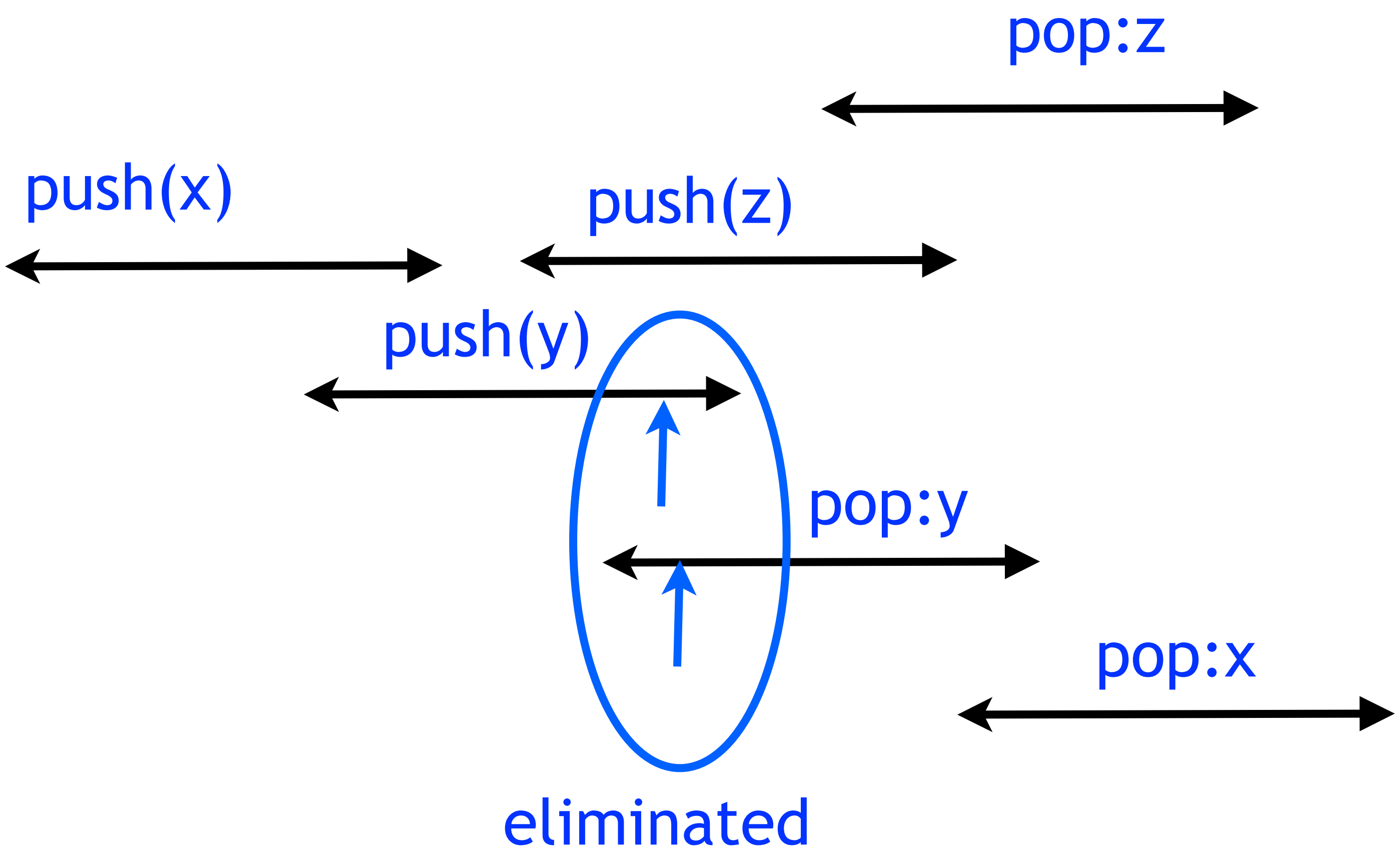
eliminated

<

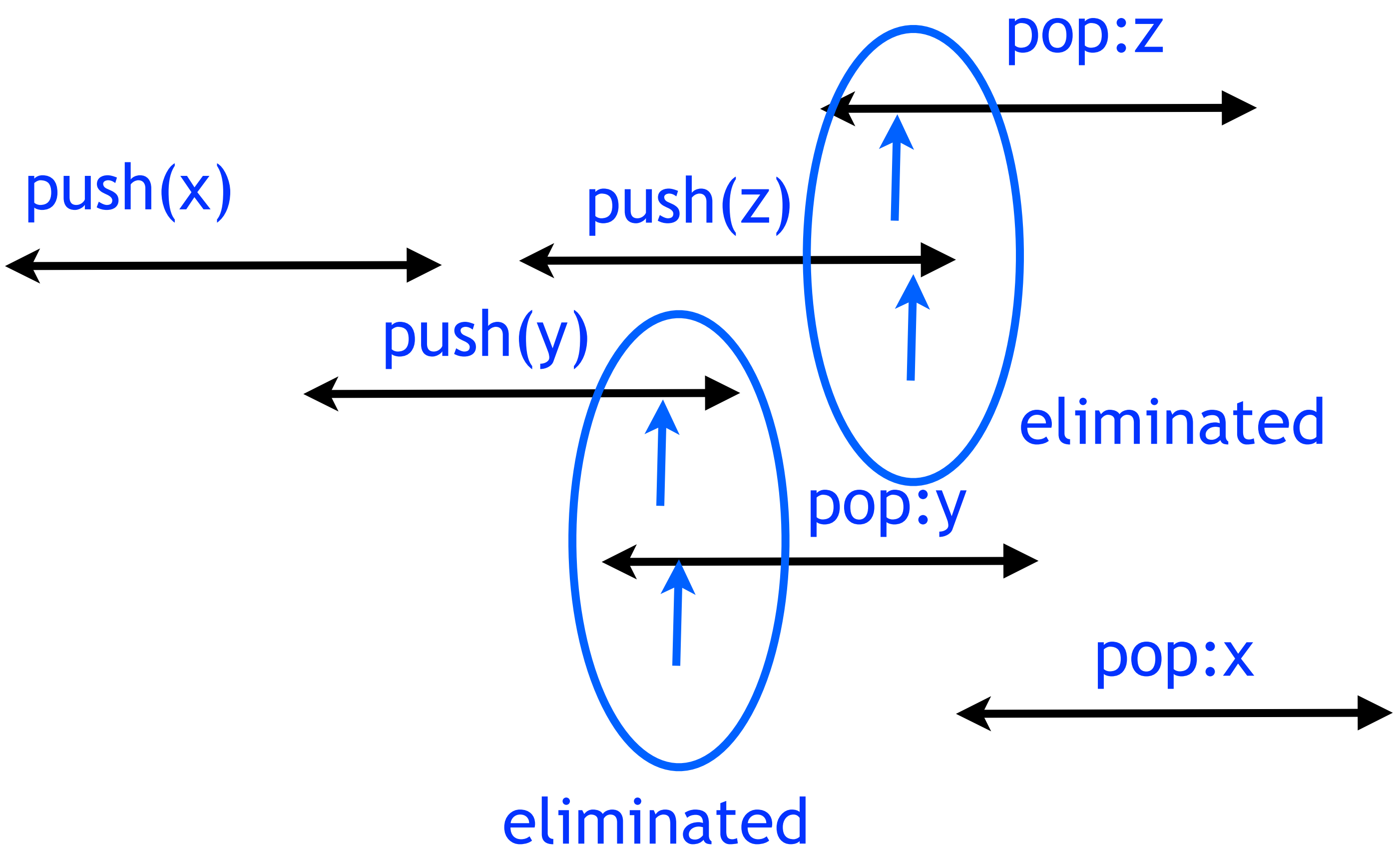
Elimination



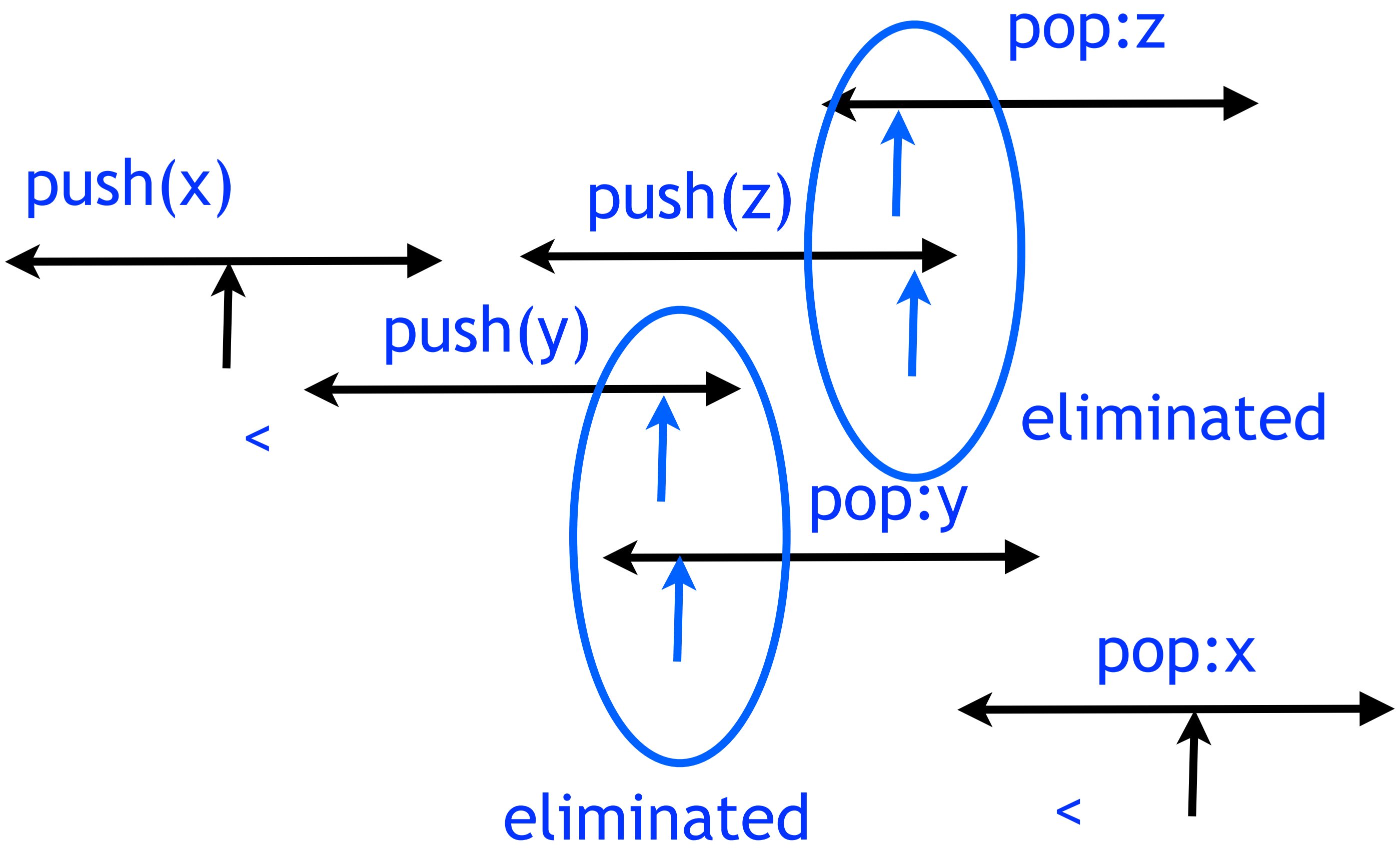
Elimination



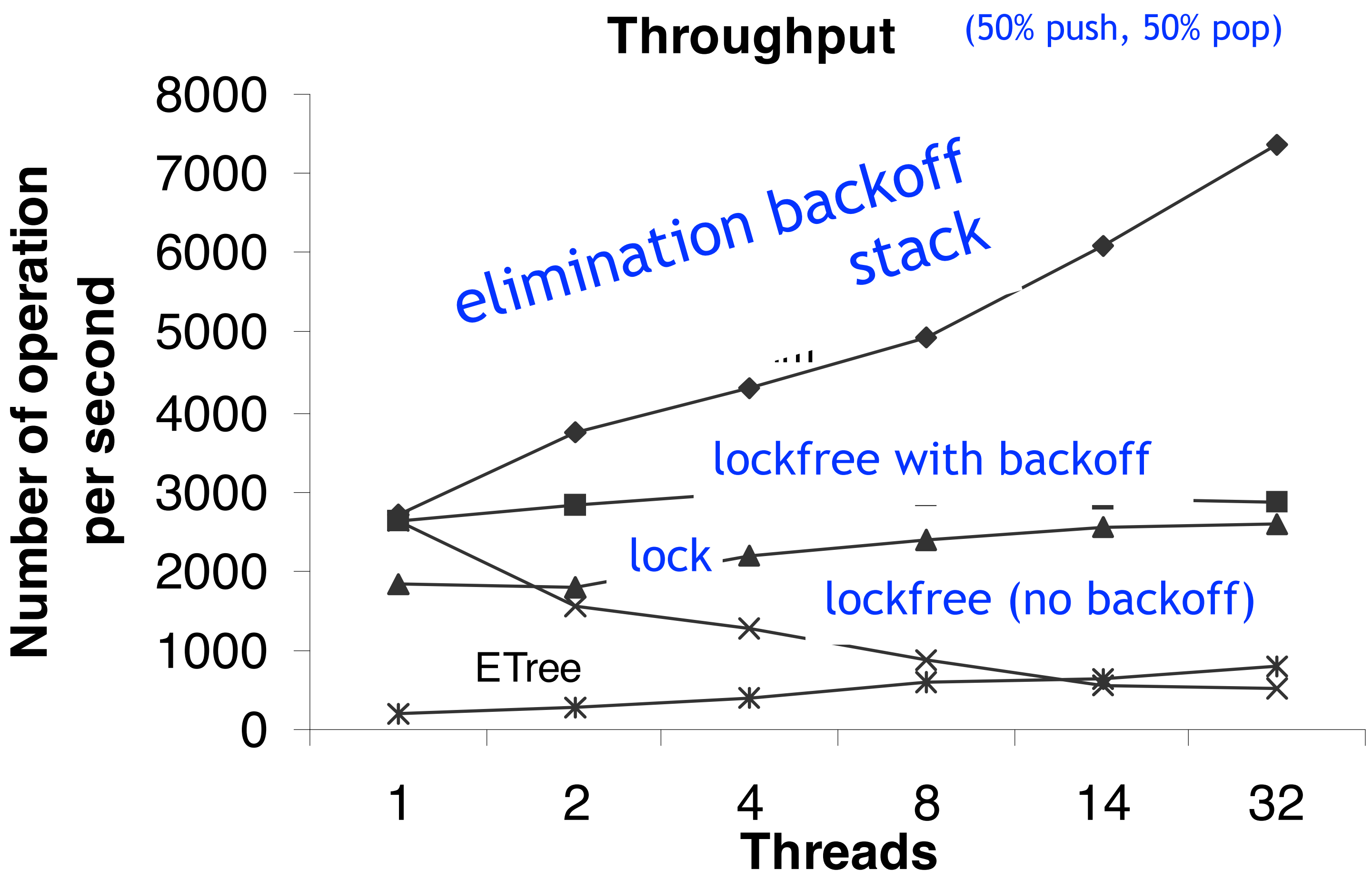
Elimination



Elimination



Measurements



(14 processor sun)

Summary

- We saw both lock-based and lock-free implementations of
  - queues and stacks
- Don’t be quick to declare a data structure inherently sequential
  - Linearizable stack is not inherently sequential
- ABA is a real problem, pay attention