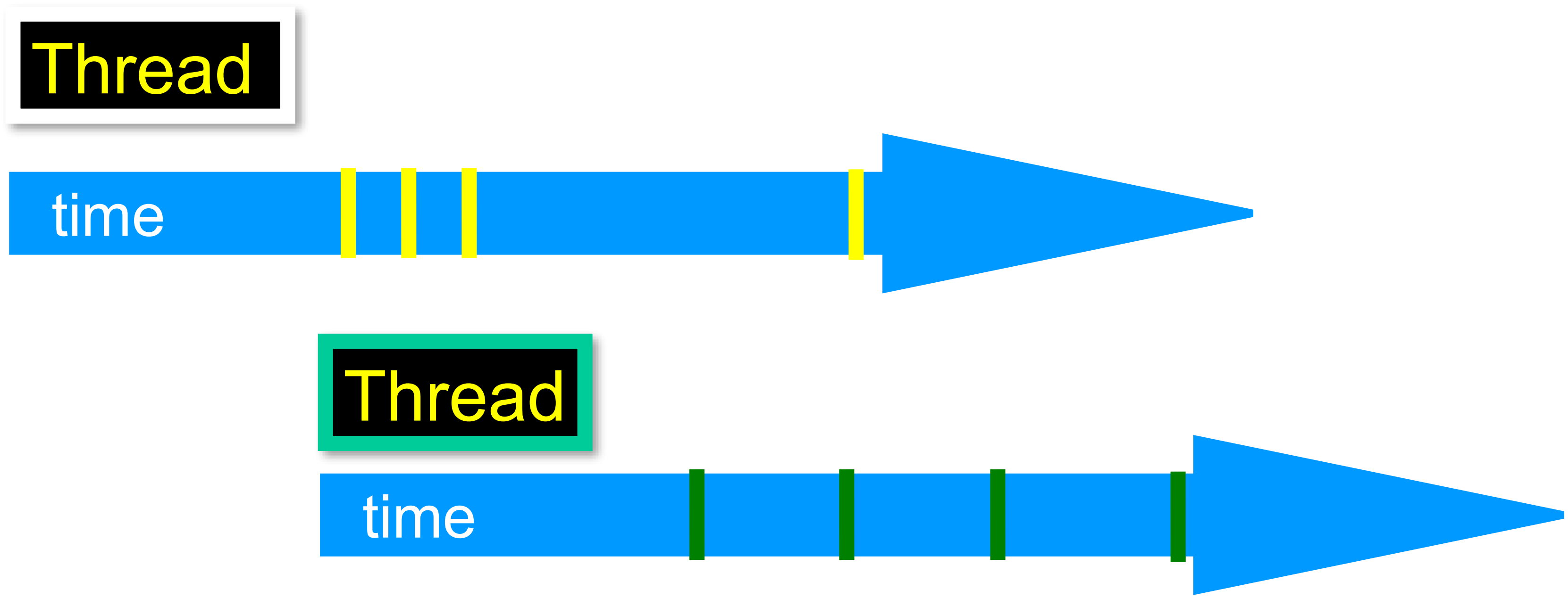
CSE 511: Operating Systems Design

Lectures 17,19

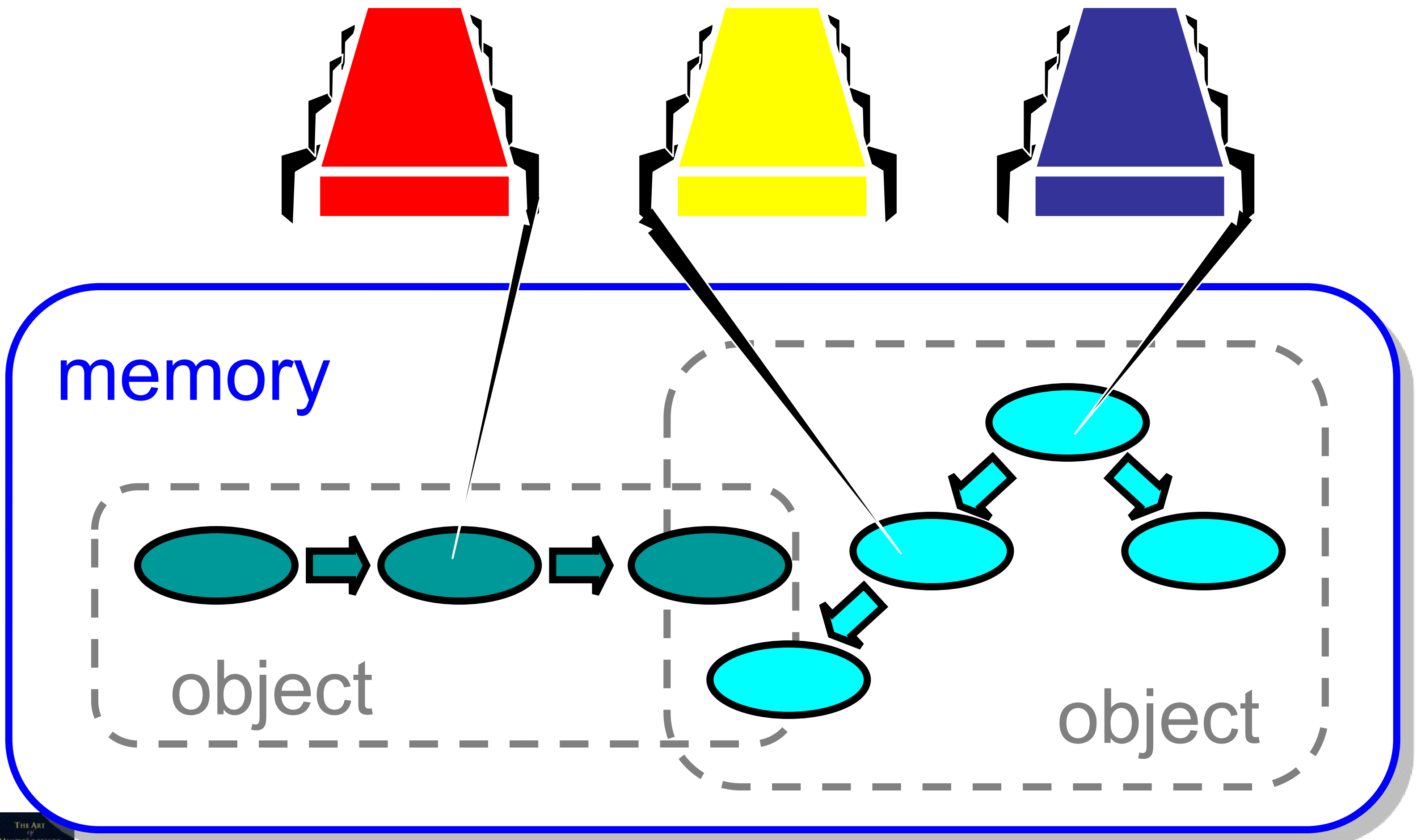
Concurrent Objects

The slides are partially based on Maurice Herlihy's slides from "The Art of Multiprocessing"

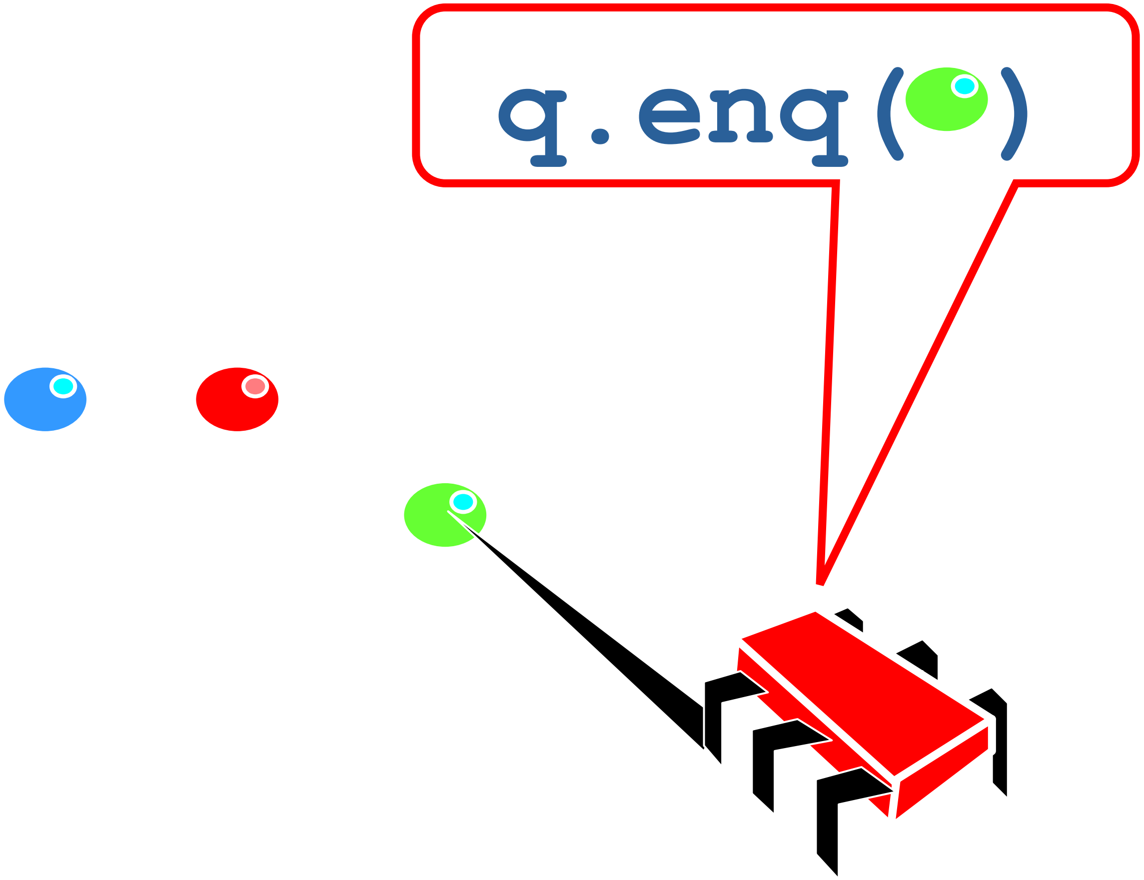
Concurrency



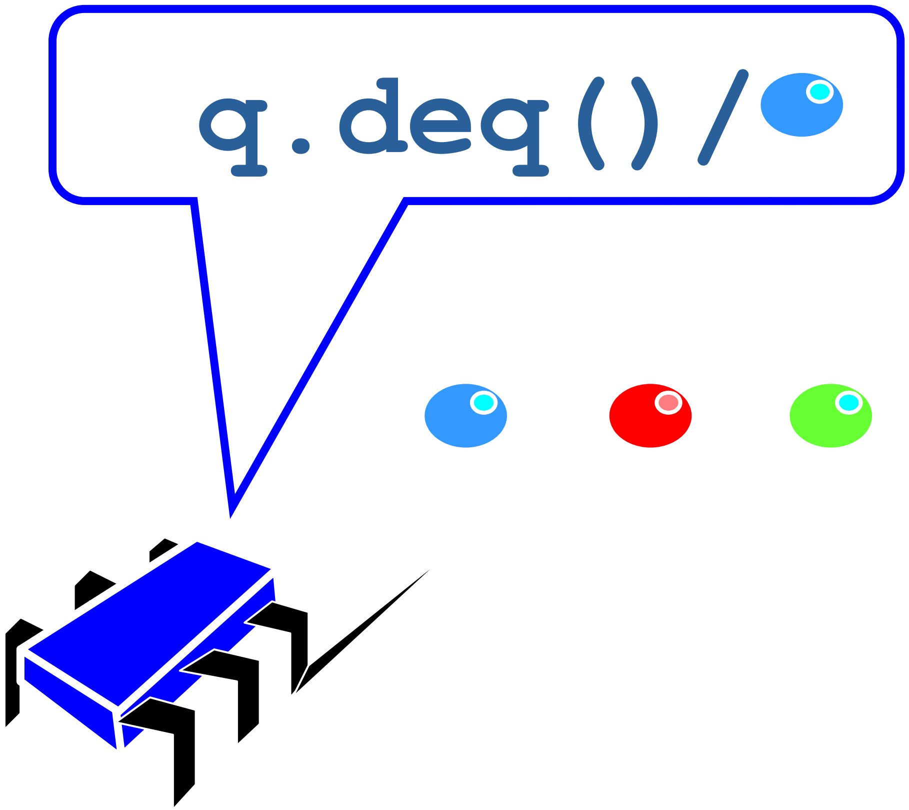
Concurrent Computation



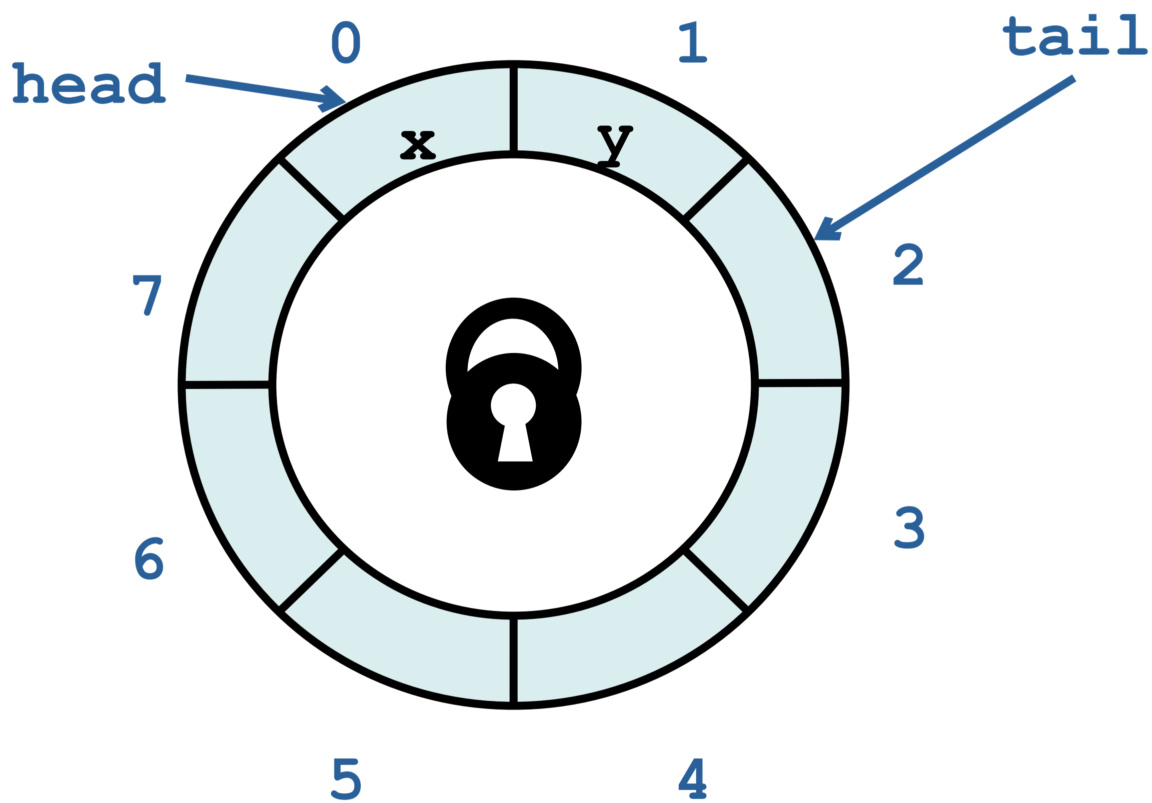
FIFO Queue: Enqueue Method



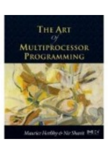
FIFO Queue: Dequeue Method



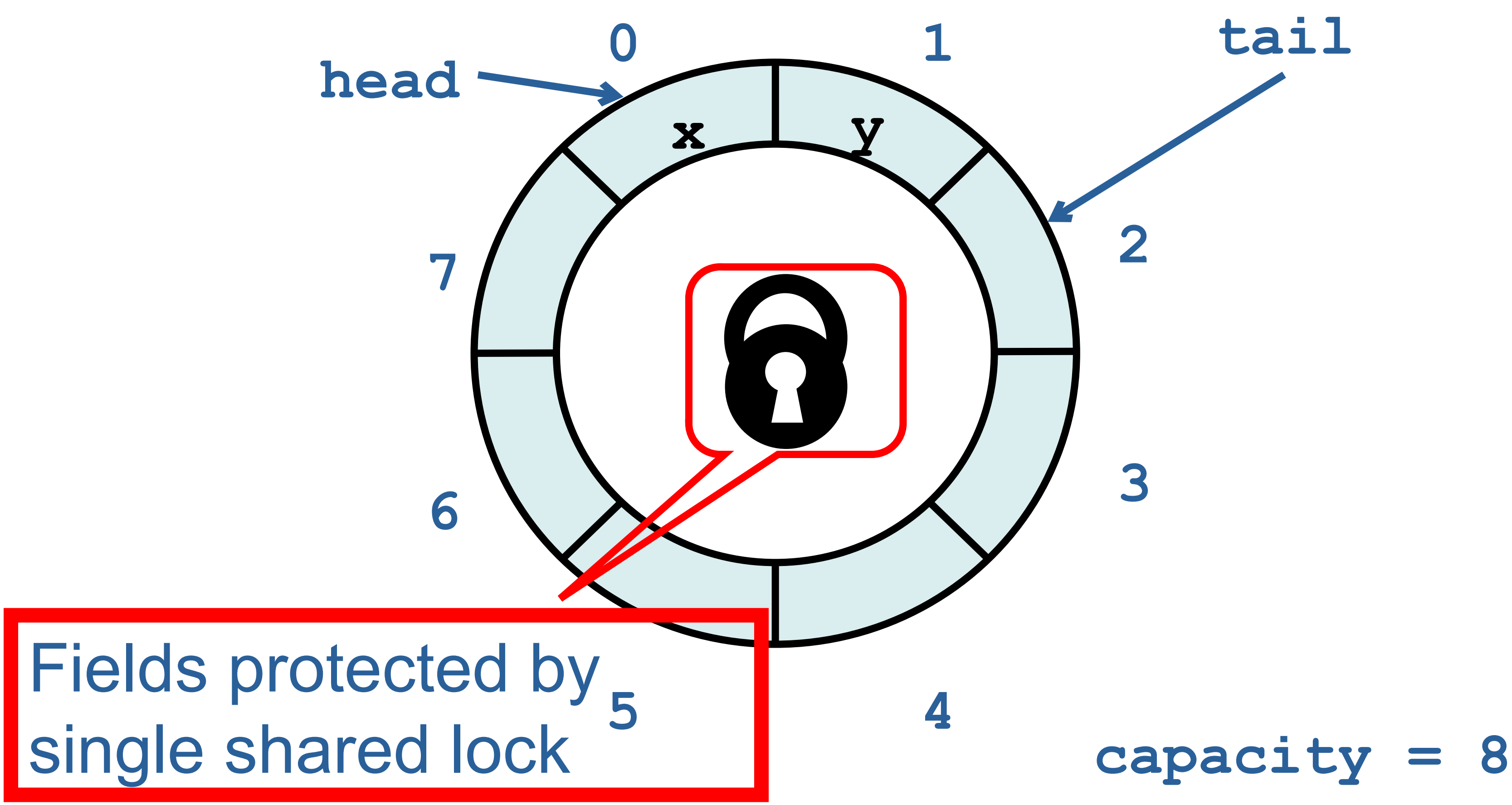
Lock-Based Queue

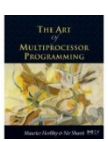


`capacity = 8`



Lock-Based Queue





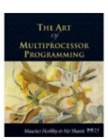
A Lock-Based Queue

```
int Head, Tail;
T items[capacity];
Lock lock;

mutex_init(&lock);
```

A Lock-Based Queue

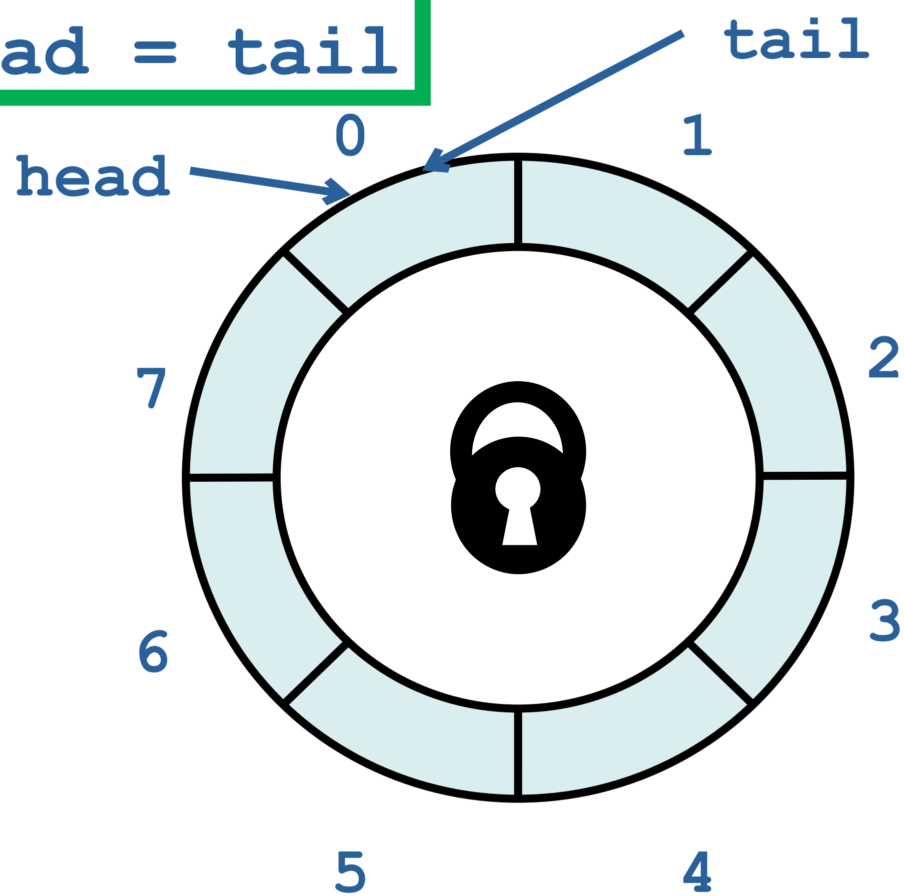
```
int Head, Tail;
T items[capacity];
Lock lock;

mutex_init(&lock);
```

Variables protected by single shared lock



Lock-Based Queue

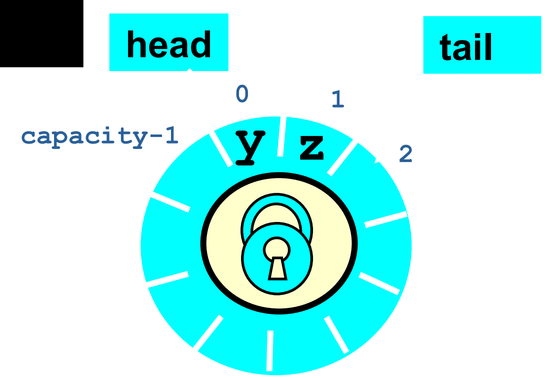
Initially **`head = tail`**

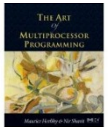


A Lock-Based Queue

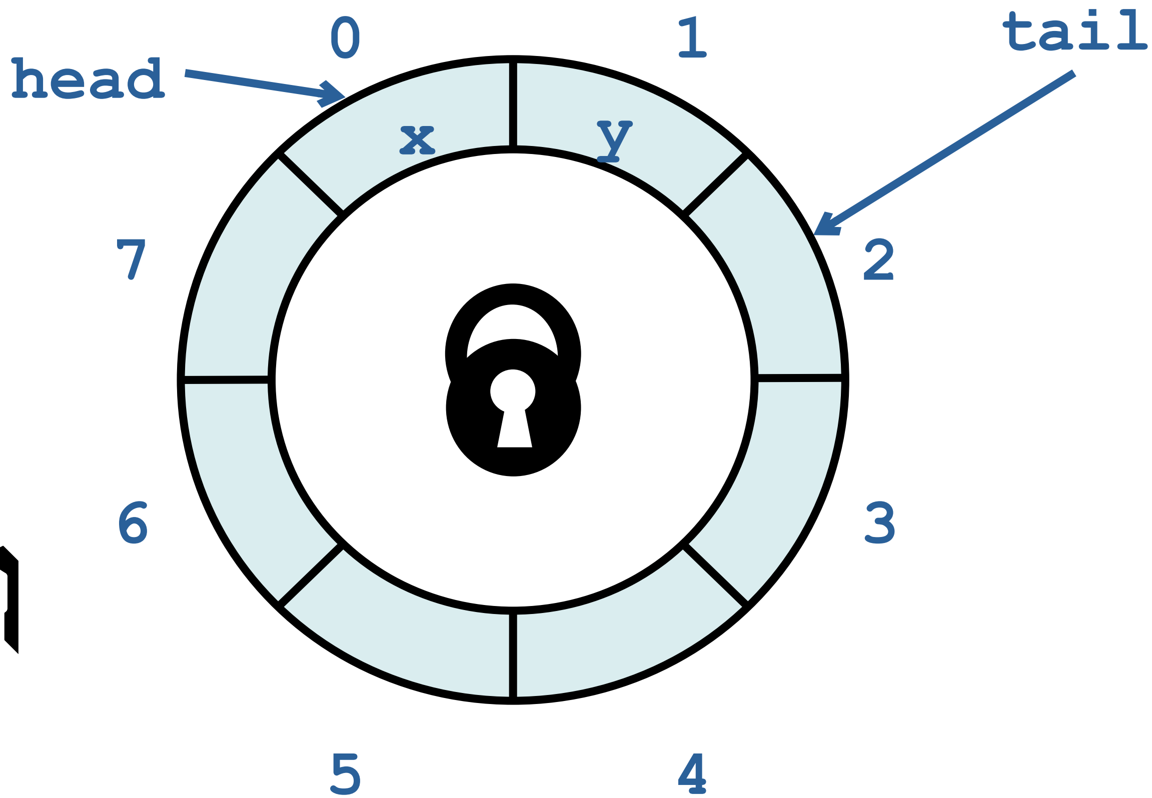
```
int Head = 0, Tail = 0;
T items[capacity];
Lock lock;

mutex_init(&lock);
```

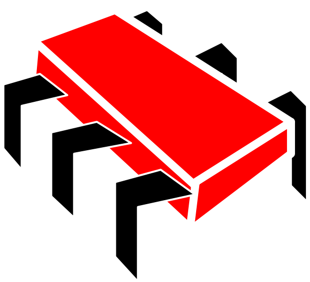
Initially **`head = tail`**

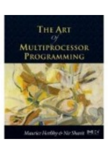




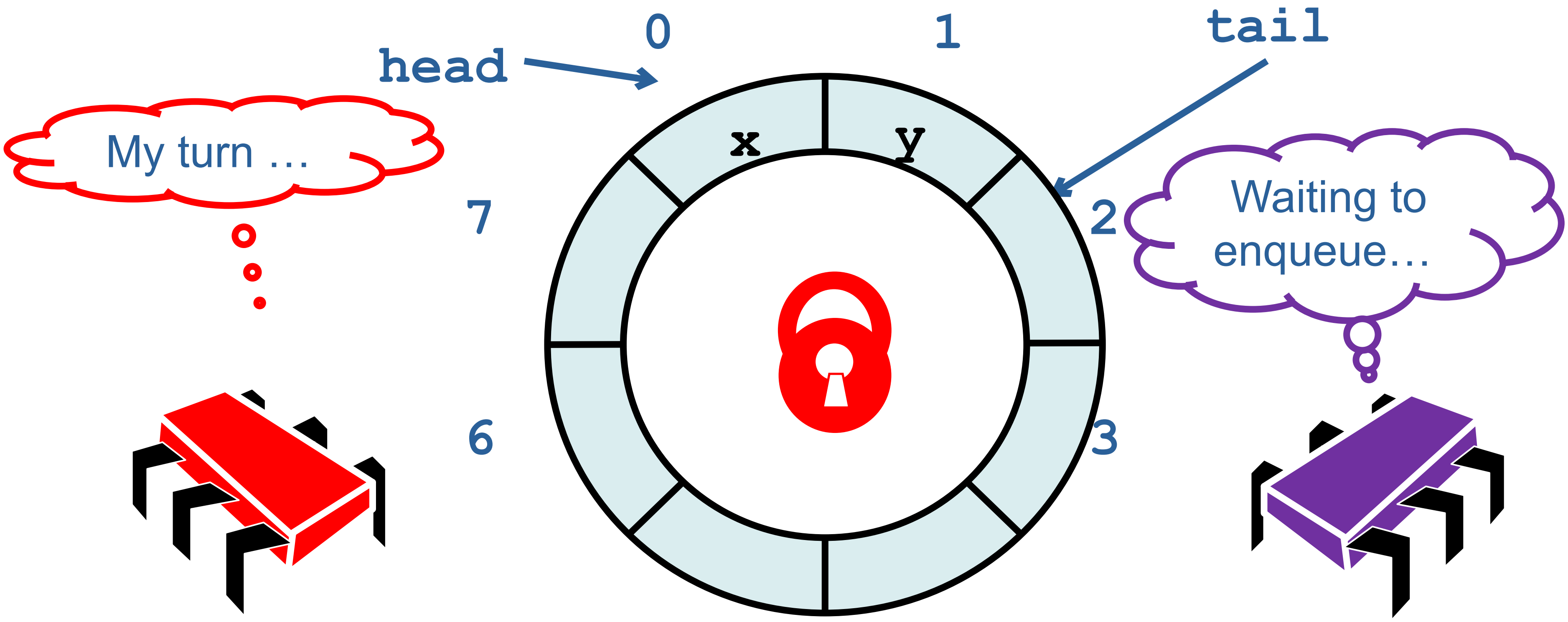
Lock-Based `deq()`

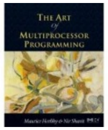






Acquire Lock

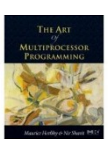




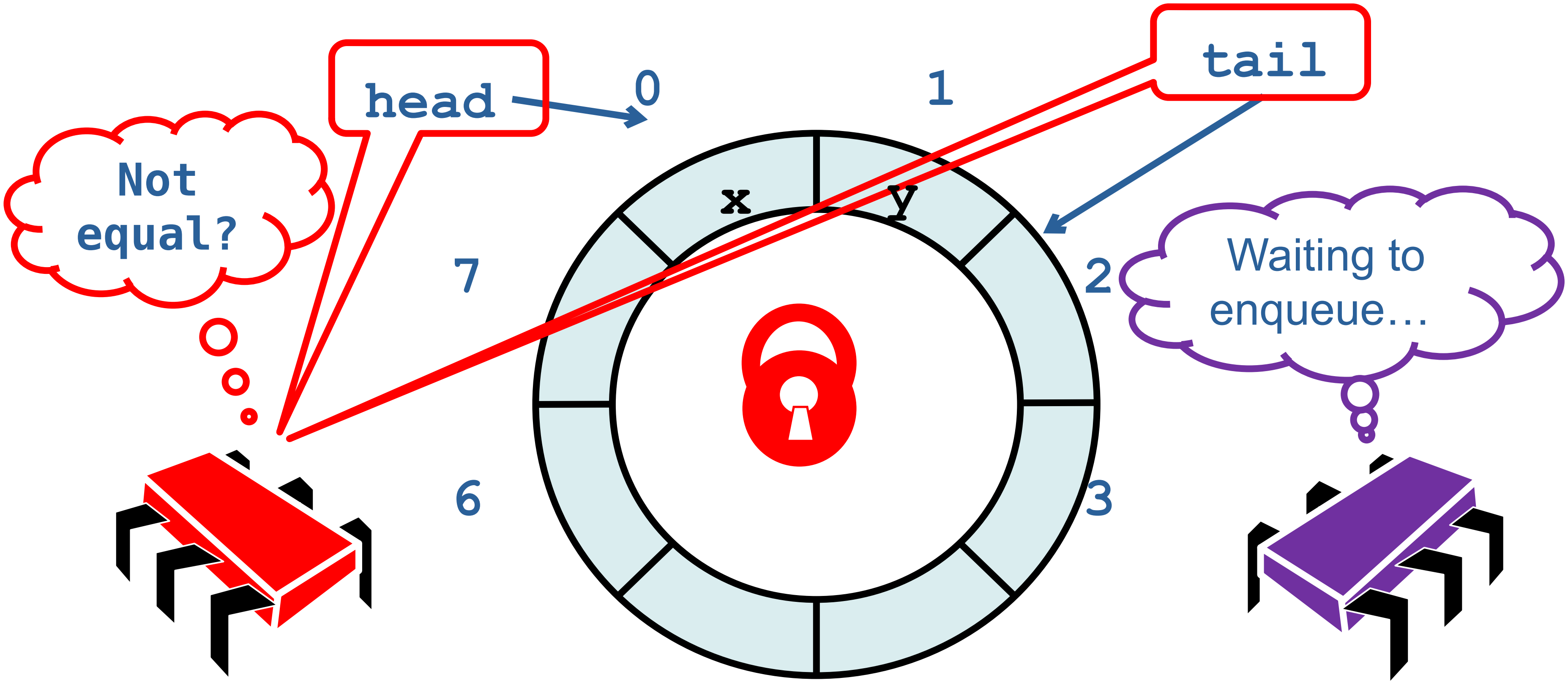
Implementation: **deq()**

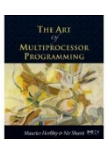
```
T deq() {
  mutex_lock(&lock);
  if (tail == head) {
    mutex_unlock(&lock);
    /* Error! */
  }
  T x = items[Head % capacity];
  Head++;
  mutex_unlock(&lock);
  return x;
}
```

Acquire lock



Check if Non-Empty

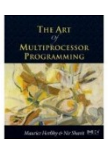




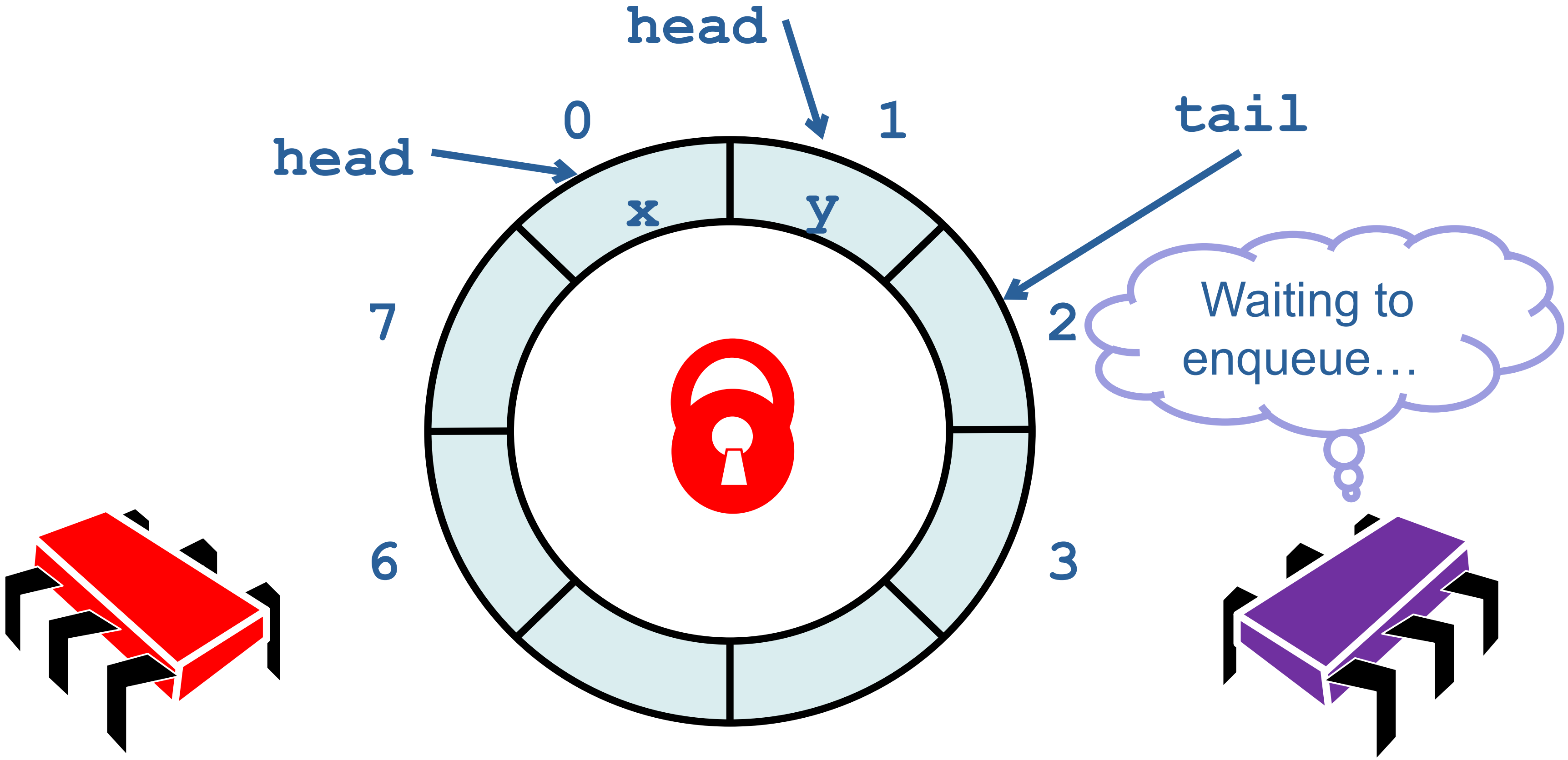
Implementation: deq()

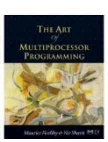
```
T deq() {
   mutex_lock(&lock);
   if (tail == head) {
     mutex_unlock(&lock);
     /* Error! */
   }
   T x = items[Head % capacity];
   Head++;
   mutex_unlock(&lock);
   return x;
 }
```

Return error
if queue is empty



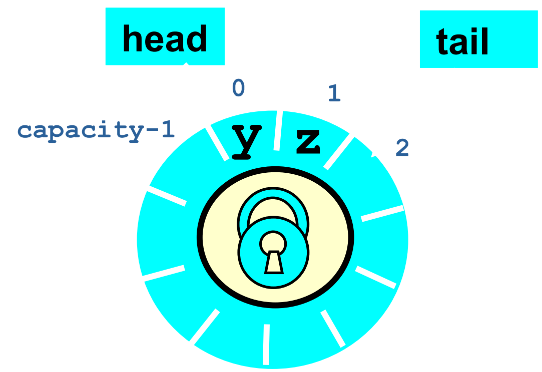
Modify the Queue



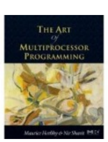


Implementation: deq()

```
T deq() {
    mutex_lock(&lock);
    if (tail == head) {
      mutex_unlock(&lock);
      /* Error! */
    }
    T x = items[Head % capacity];
    Head++;
    mutex_unlock(&lock);

    return x;
  }
```



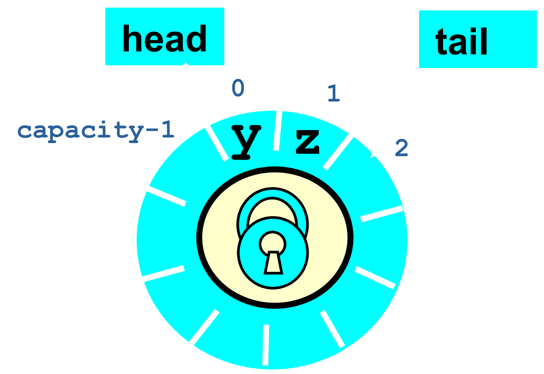
Queue not empty?
Remove item and update head

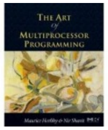


Implementation: deq()

```
T deq() {
   mutex_lock(&lock);
   if (tail == head) {
     mutex_unlock(&lock);
     /* Error! */
   }
   T x = items[Head % capacity];
   Head++;
   mutex_unlock(&lock);

   return x;
}
```

Return result

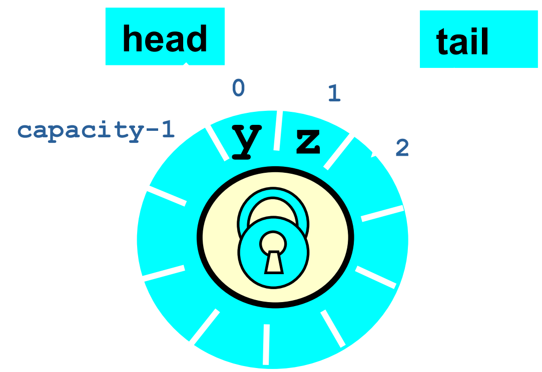


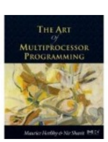


Implementation: deq()

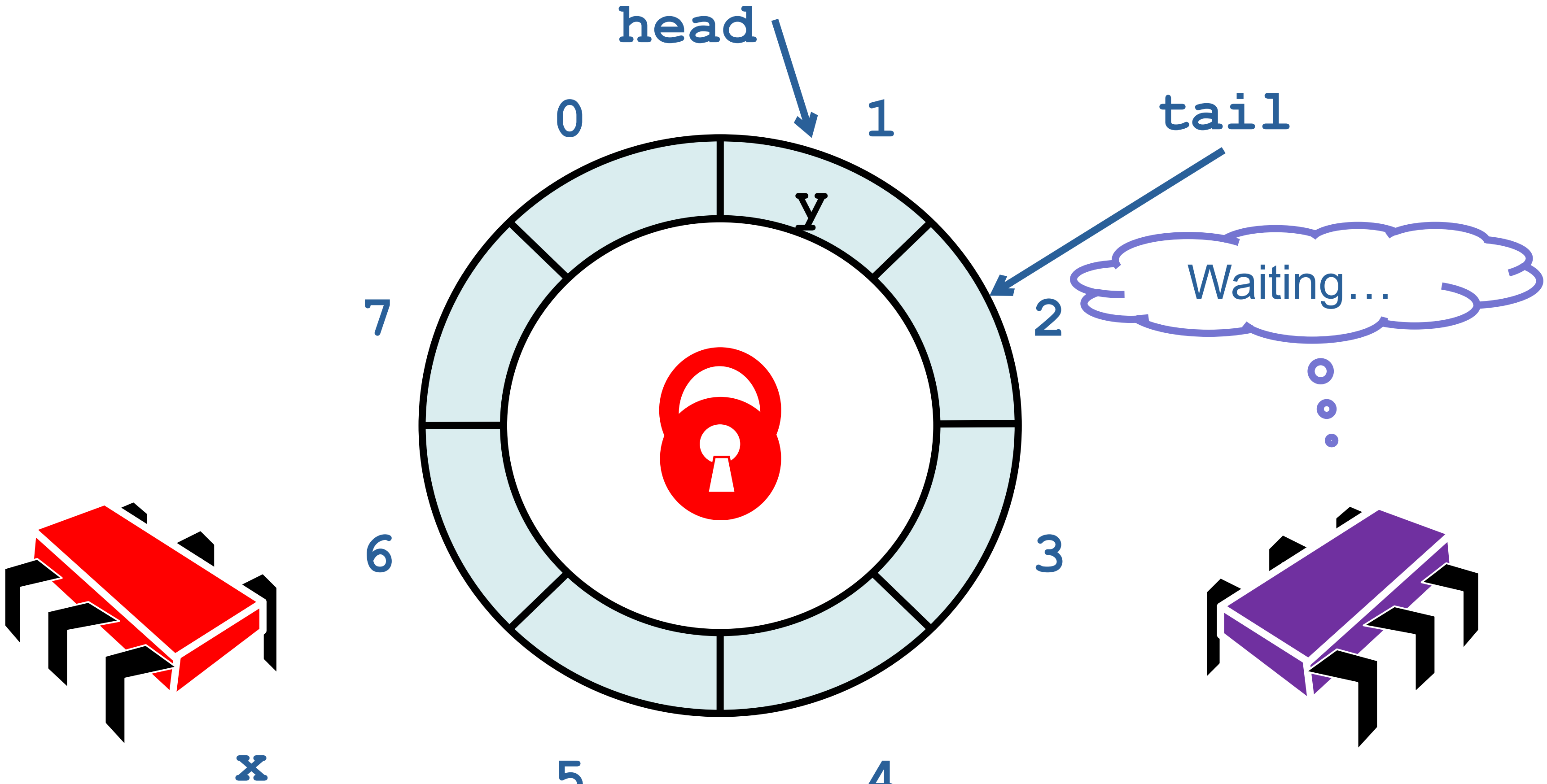
```
T deq() {
    mutex_lock(&lock);
    if (tail == head) {
      mutex_unlock(&lock);
      /* Error! */
    }
    T x = items[Head % capacity];
    Head++;
    mutex_unlock(&lock);

    return x;
  }
```

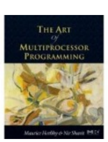
Return result



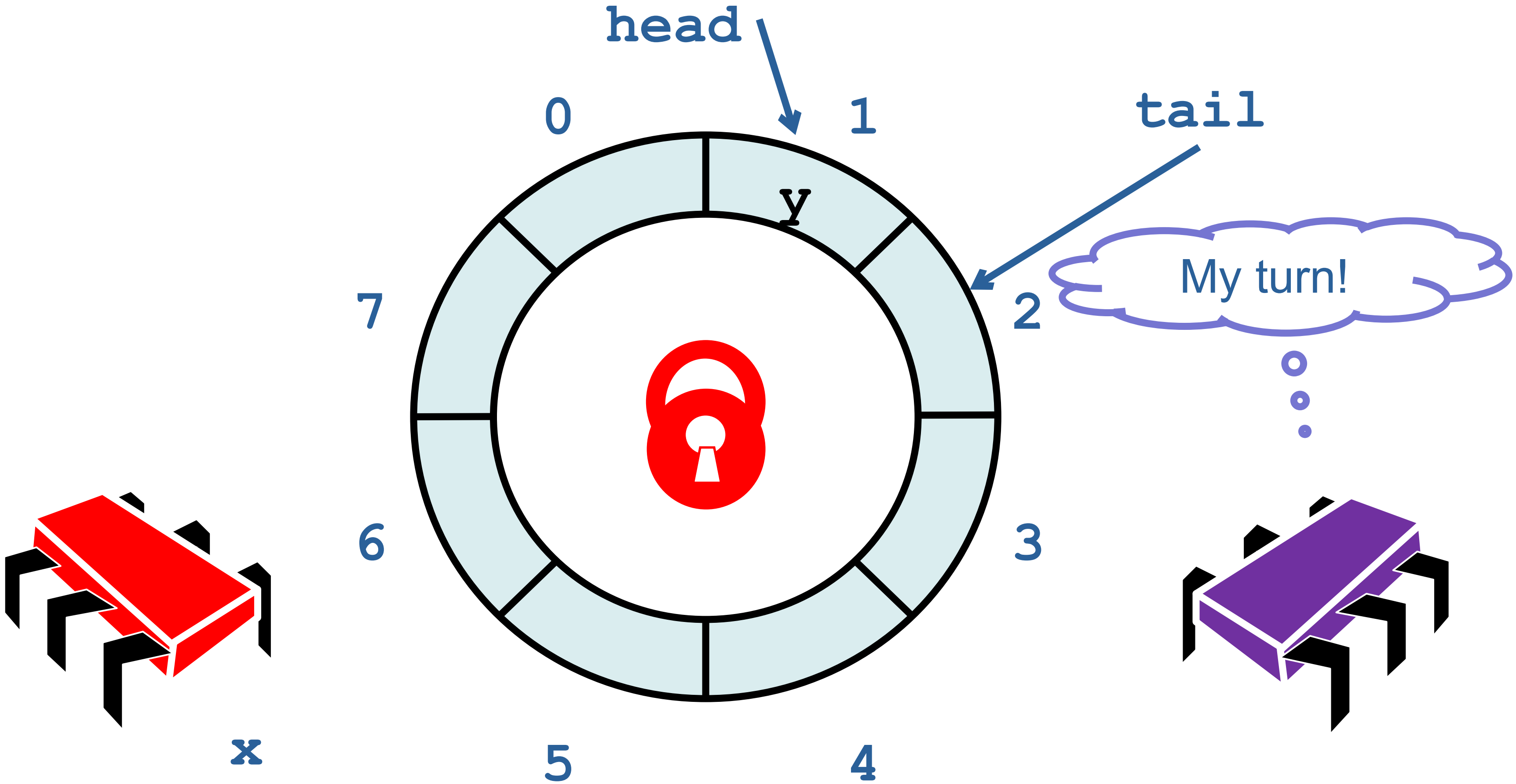


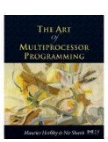
Release the Lock





Release the Lock

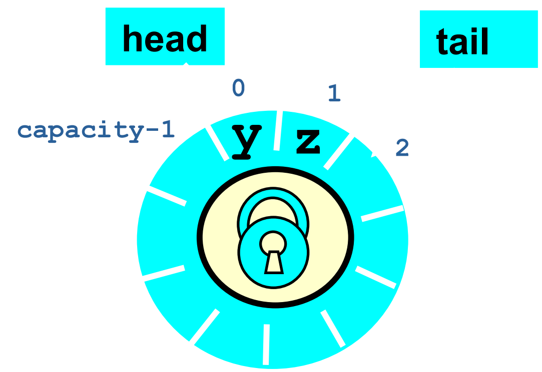


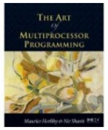


Implementation: deq()

```
T deq() {
    mutex_lock(&lock);
    if (tail == head) {
      mutex_unlock(&lock);
      /* Error! */
    }
    T x = items[Head % capacity];
    Head++;
    mutex_unlock(&lock);

    return x;
  }
```

Release lock no matter what!

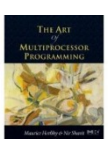




Implementation: **deq()**

```
T deq() {
    mutex_lock(&lock);
    if (tail == head) {
      mutex_unlock(&lock);
      /* Error! */
    }
    T x = items[Head % capacity];
    Head++;
    mutex_unlock(&lock);

    return x;
  }
```

Should be correct because modifications mutually exclusive…



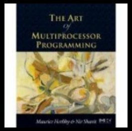
An alternative implementation

The same code without mutual exclusion

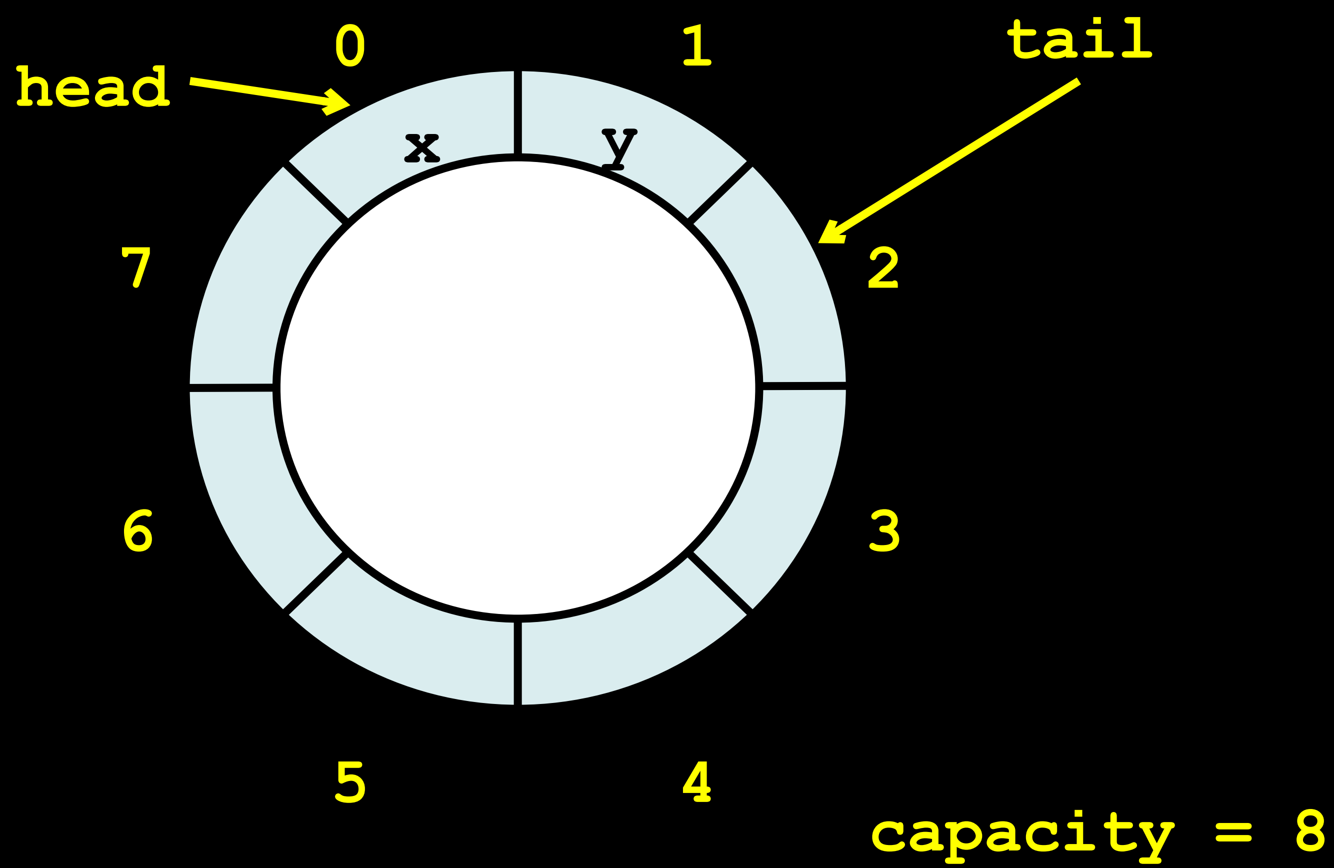
For simplicity, only *two* threads

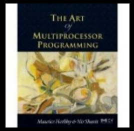
One thread enq only

The other deq only

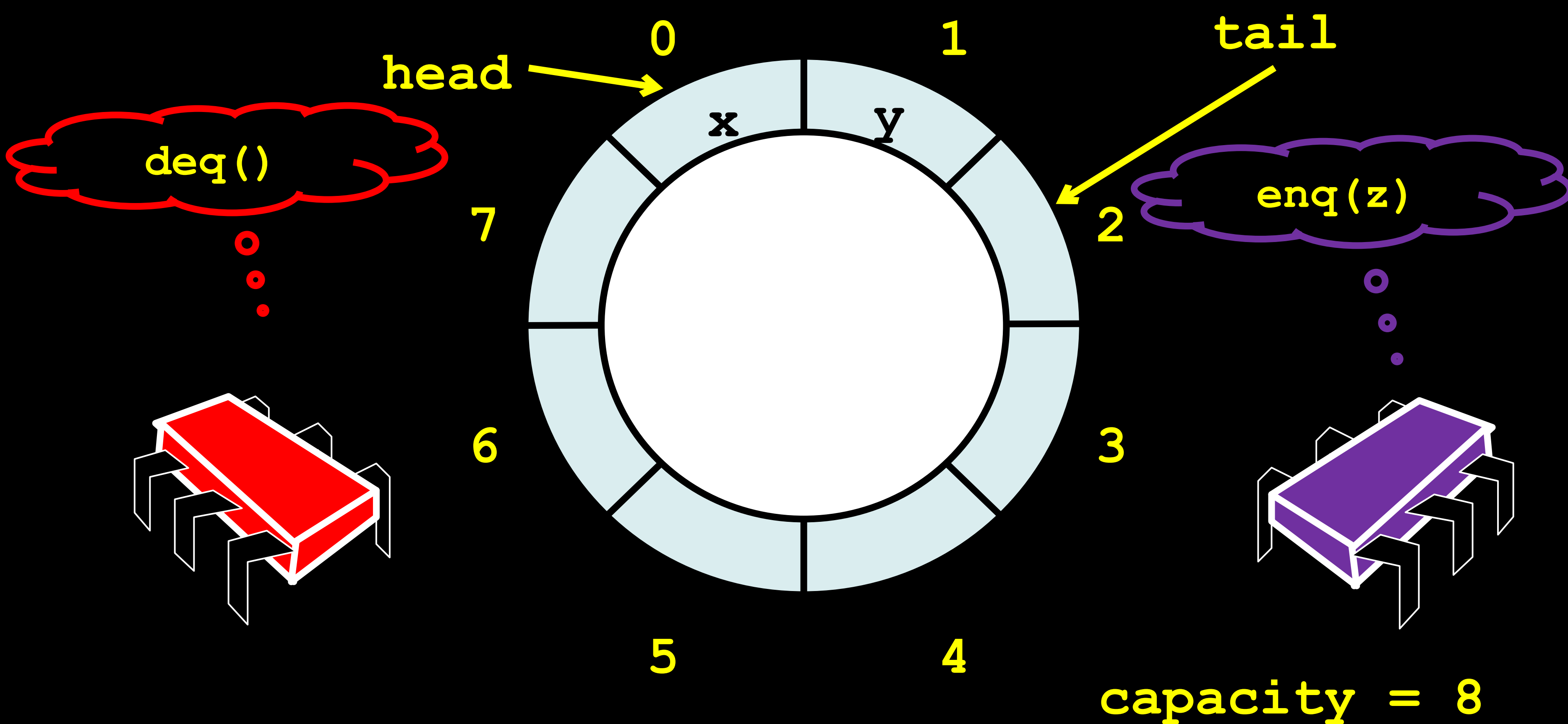


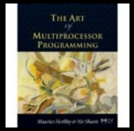
Wait-free 2-Thread Queue



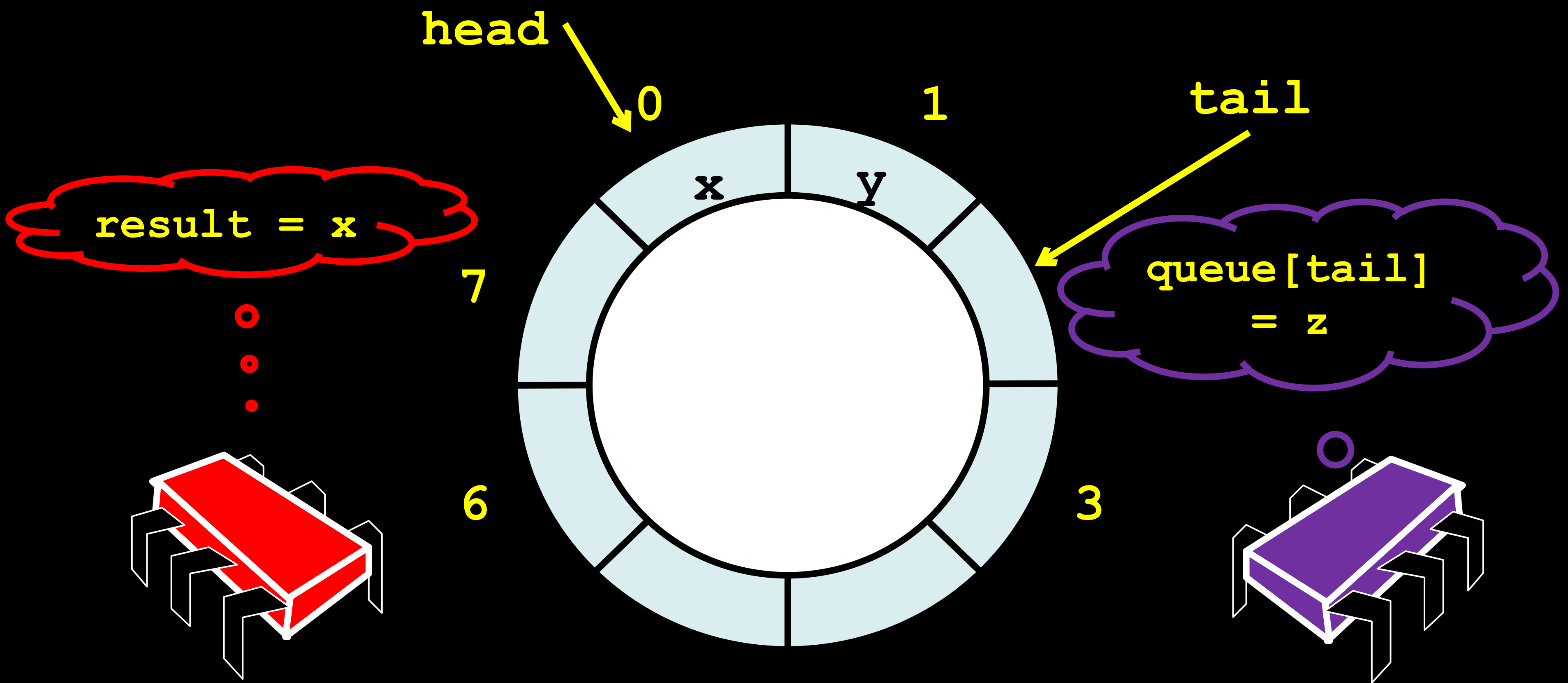


Wait-free 2-Thread Queue

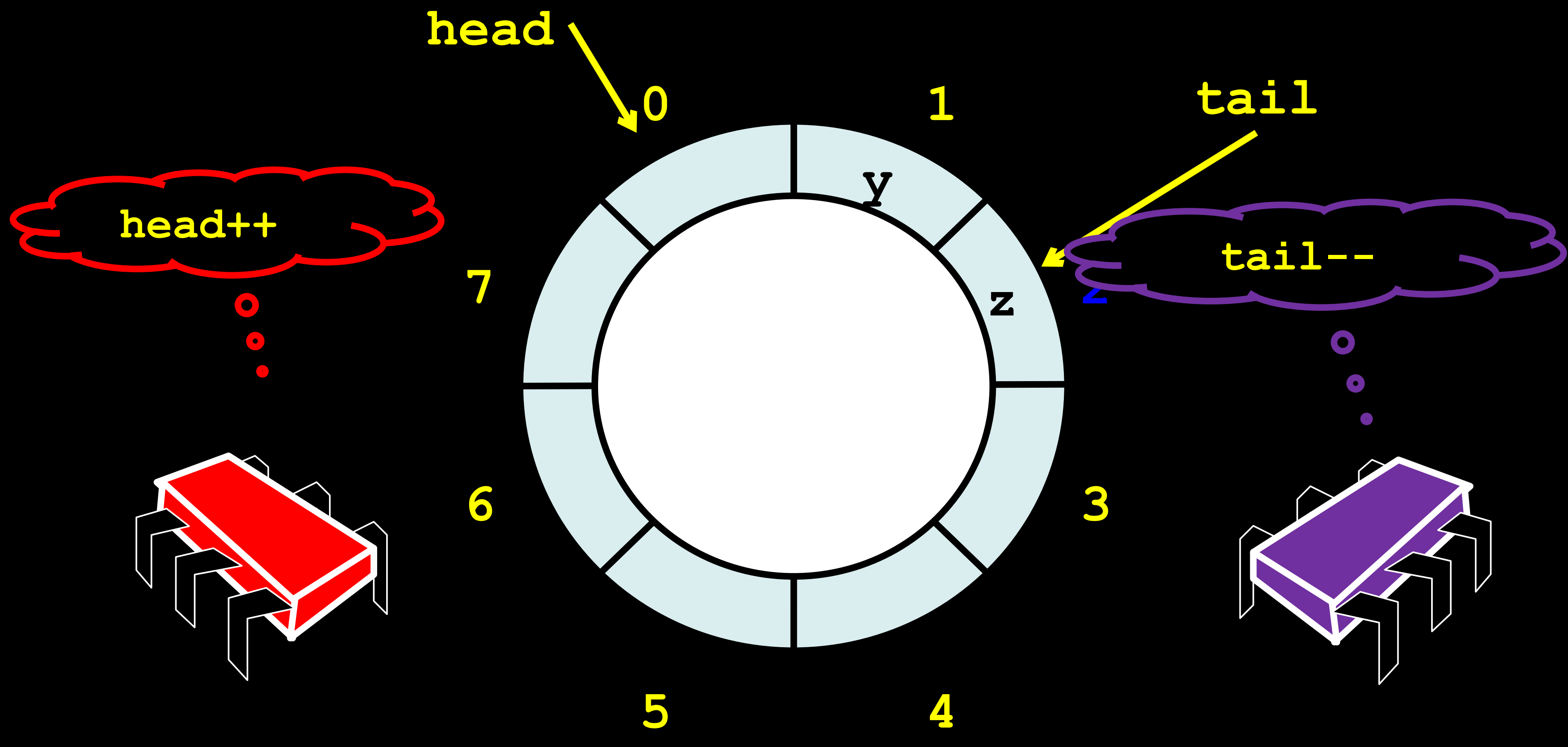




Wait-free 2-Thread Queue



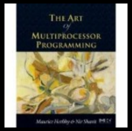
Wait-free 2-Thread Queue



Wait-free 2-Thread Queue

```
int head = 0, tail = 0;
Item items[capacity];

void enq(Item x) {
   if (tail-head == capacity) {
      /* Error */
   }
   items[tail % capacity] = x; tail++;
}
Item deq() {
   if (tail == head) {
     /* Error */
   }
   Item item = items[head % capacity]; head++;
   return item;
}
```

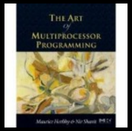
No lock needed!



Wait-free 2-Thread Queue

```
int head = 0, tail = 0;
Item items[capacity];

void enq(Item x) {
   if (tail-head == capacity) {
      /* Error */
   }
   items[tail %                                  ];
}
Item d
   if (          d) {
     /*        */
   }
   Item item = items[head % capacity]; head++;
   return item;
}
```

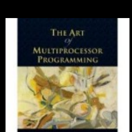
How do we define “correct” when modifications are not mutually exclusive?



What *is* a Concurrent Queue?

Need a way to *specify* a concurrent queue object

Need a way to prove that an algorithm implements the object's specification



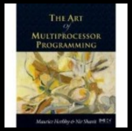
Correctness and Progress

In a concurrent setting, must specify both *safety* and *liveness* properties

Need a way to define

when an implementation is *correct*

When it guarantees *progress*



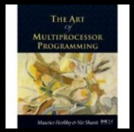
Sequential Objects

Each object has a *state*

- **Usually given by a set of *fields***
- **Queue example: sequence of items**

Each object has a set of *methods*

- **Only way to manipulate state**
- **Queue example: `enq` and `deq` methods**



Sequential Specifications

If (precondition)

the object is in such-and-such a state

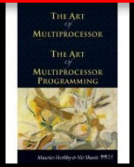
before you call the method,

Then (postcondition)

the method will return a particular value

or return an error.

and (also postcondition)



The object will be in some other state

Pre and Post Conditions for Dequeue

Precondition

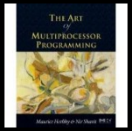
Queue is non-empty

Postcondition

Returns first item in queue

Postcondition

Removes first item in queue



Pre and Post Conditions for Dequeue

Precondition

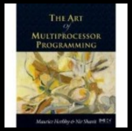
Queue is empty

Postcondition

Returns an error

Postcondition

Queue state unchanged

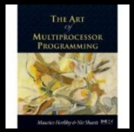


Why Are Sequential Specifications Useful?

Interactions among methods captured by side-effects on object state

Documentation size linear in number of methods

Can add new methods without combinatorial blow-up

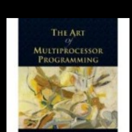


What About Concurrent Specifications ?

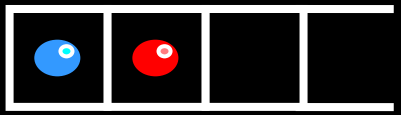
Methods?

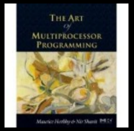
Documentation?

Adding new methods?

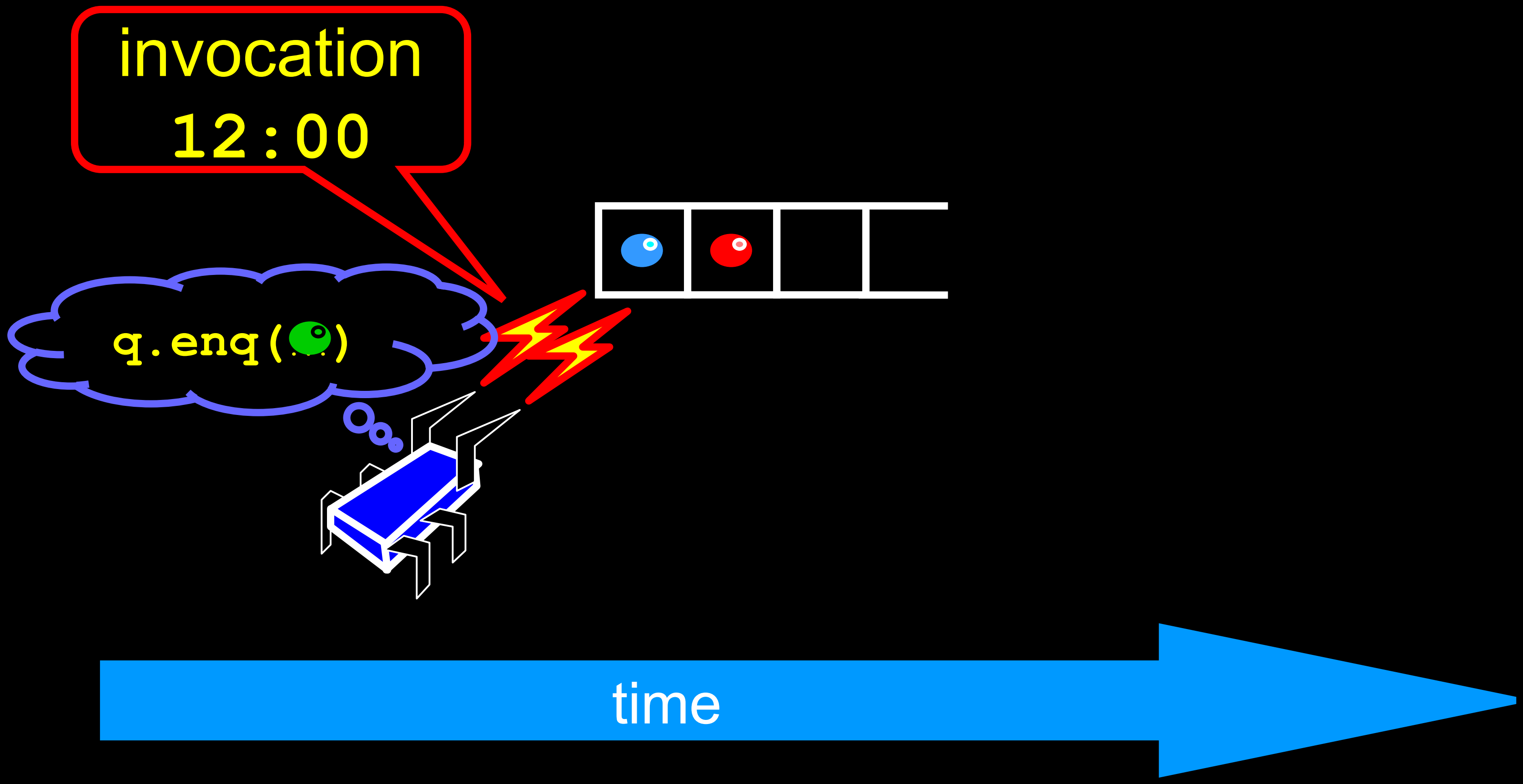


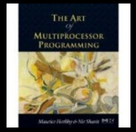
Methods Take Time



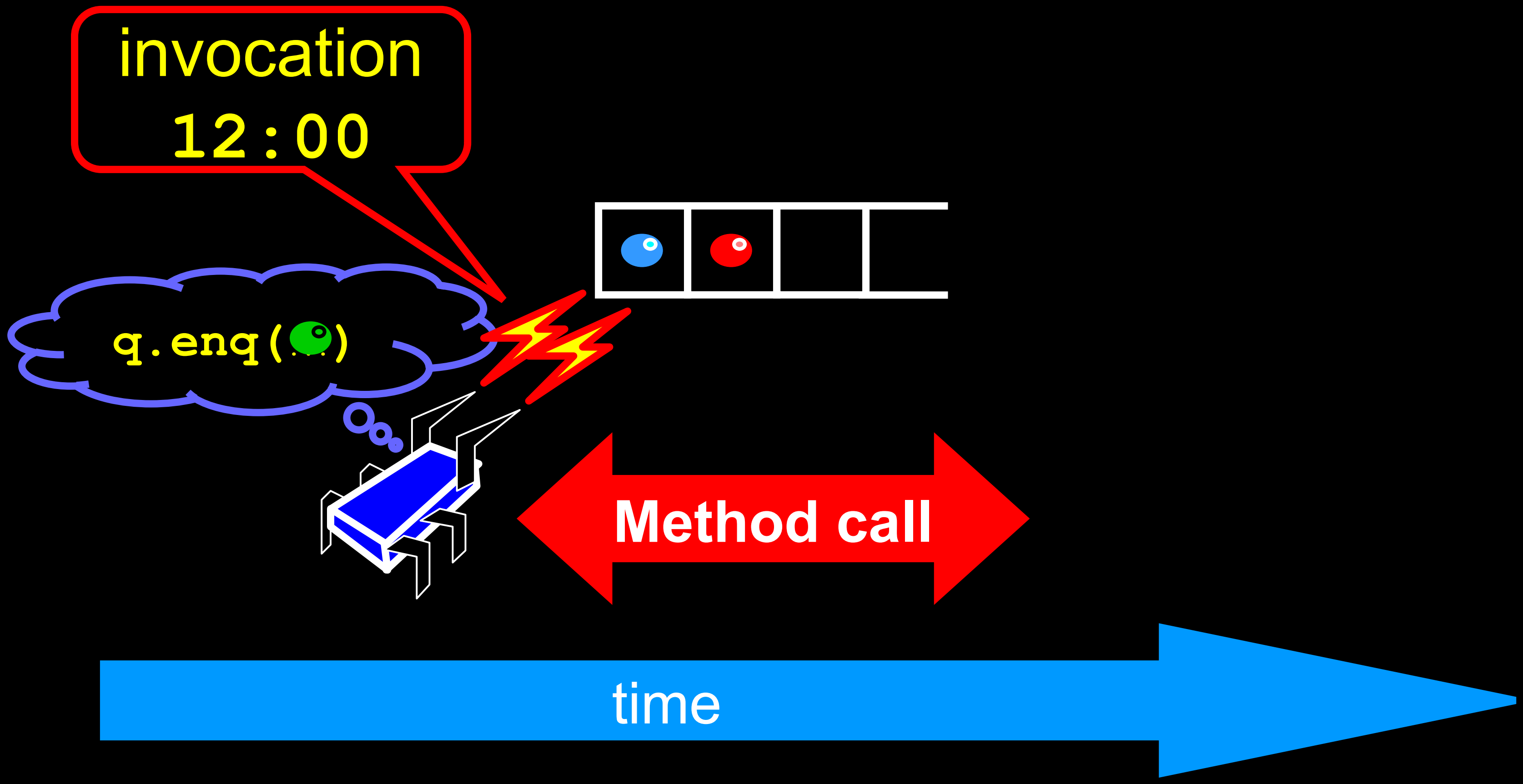


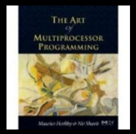
Methods Take Time



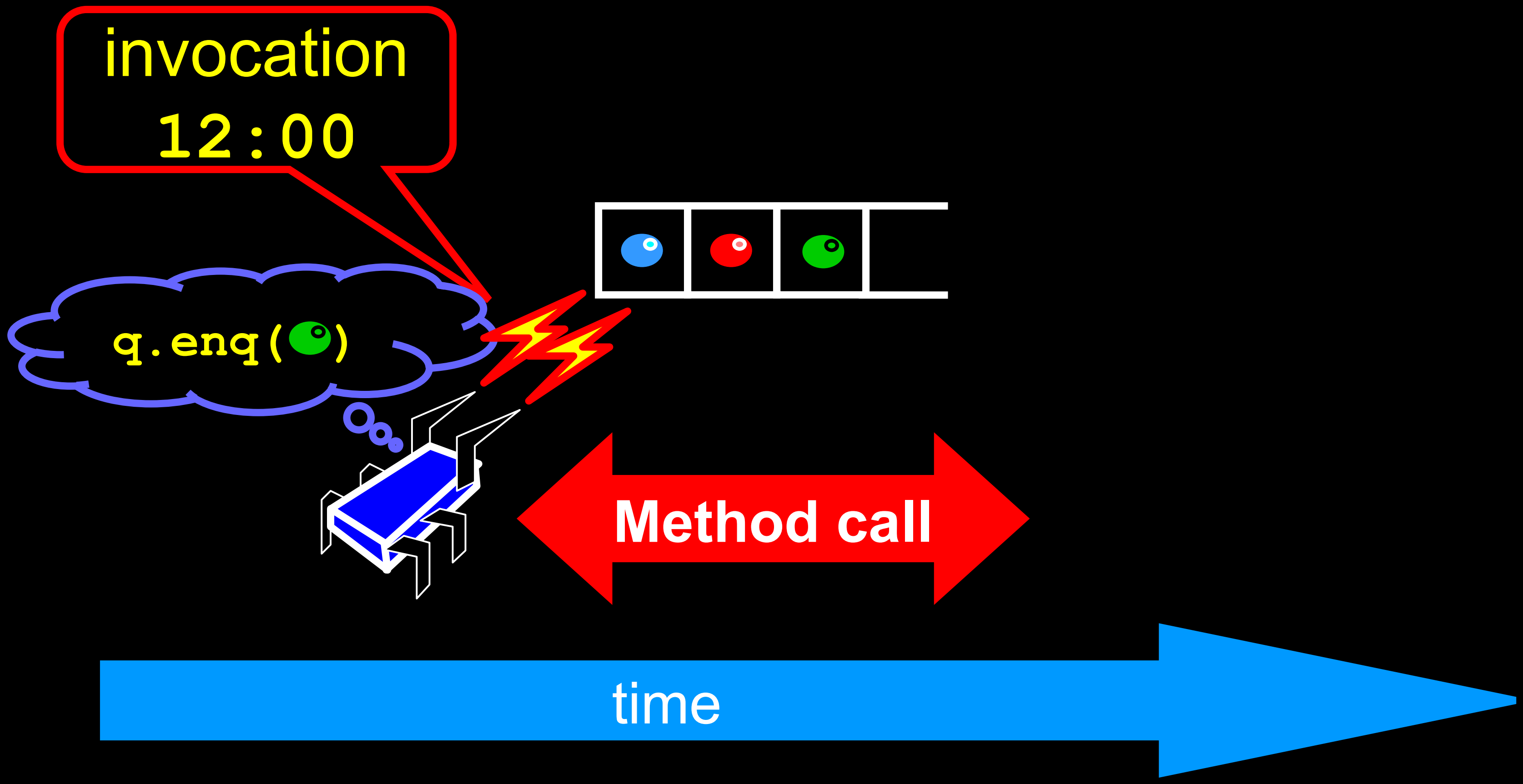


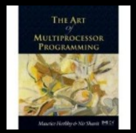
Methods Take Time



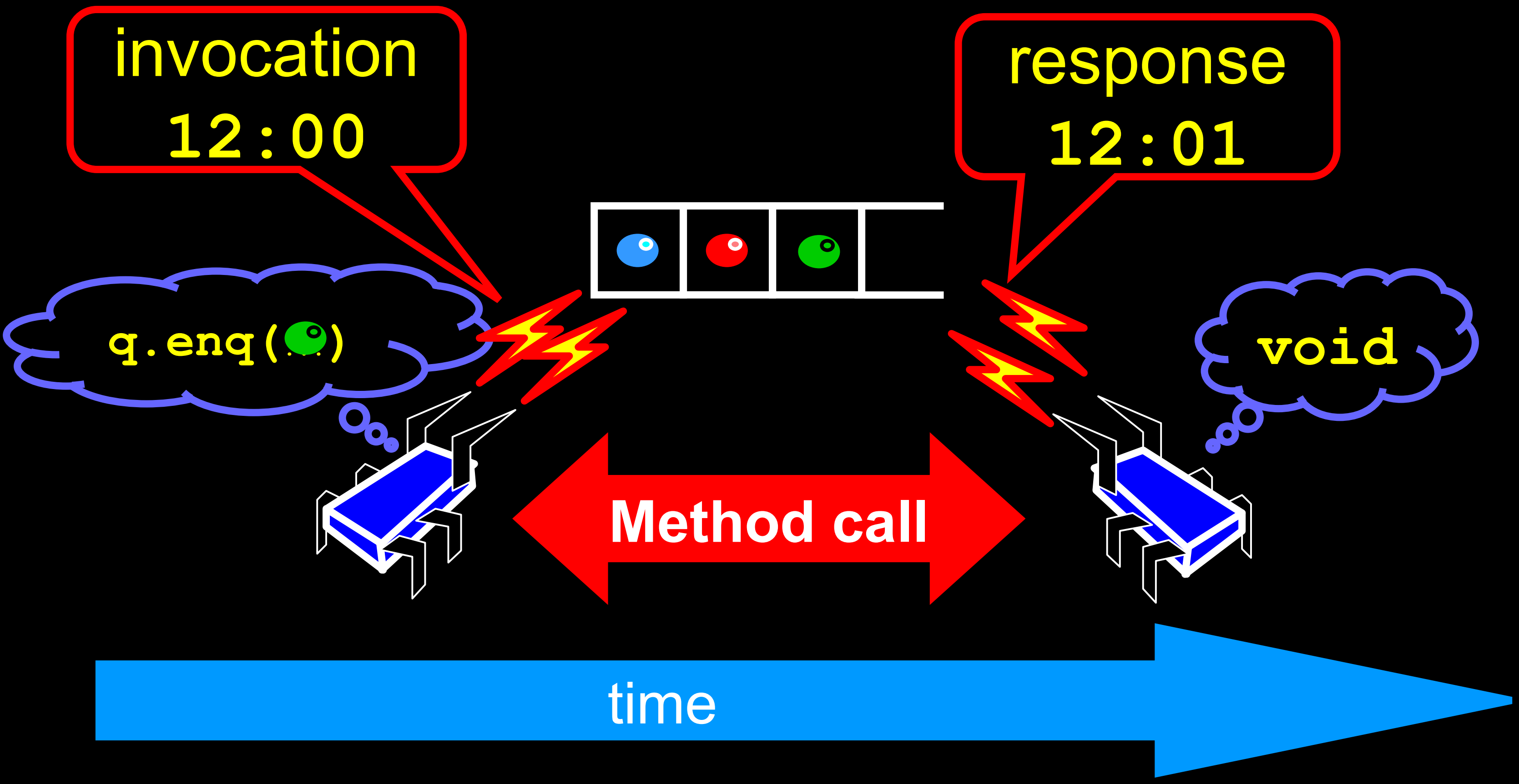


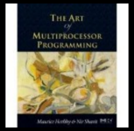
Methods Take Time





Methods Take Time





Sequential vs Concurrent

Sequential

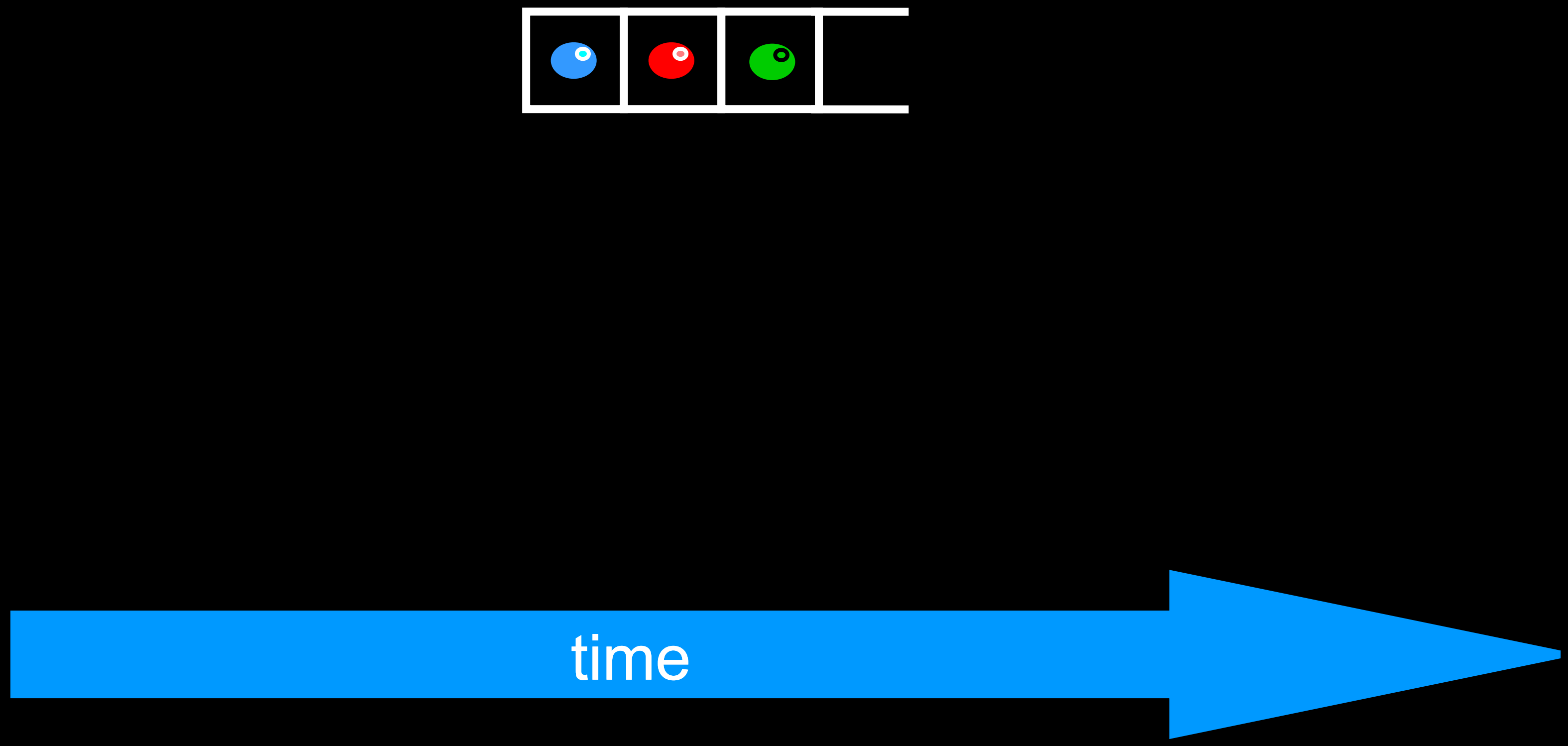
Method calls take time? Who knew?

Concurrent

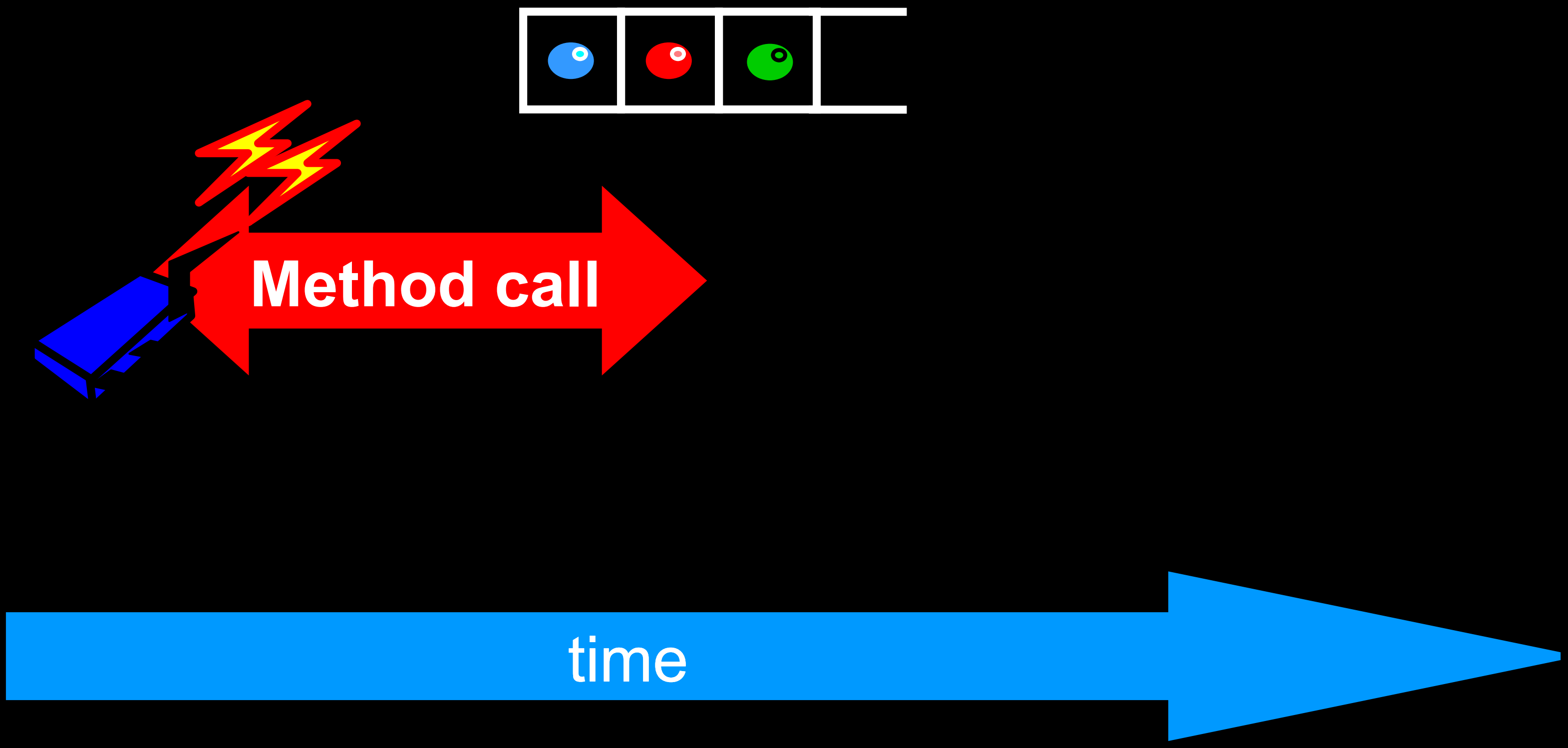
Method call is not an *event*

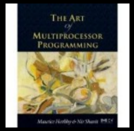
Method call is an *interval*.

Concurrent Methods Take *Overlapping* Time

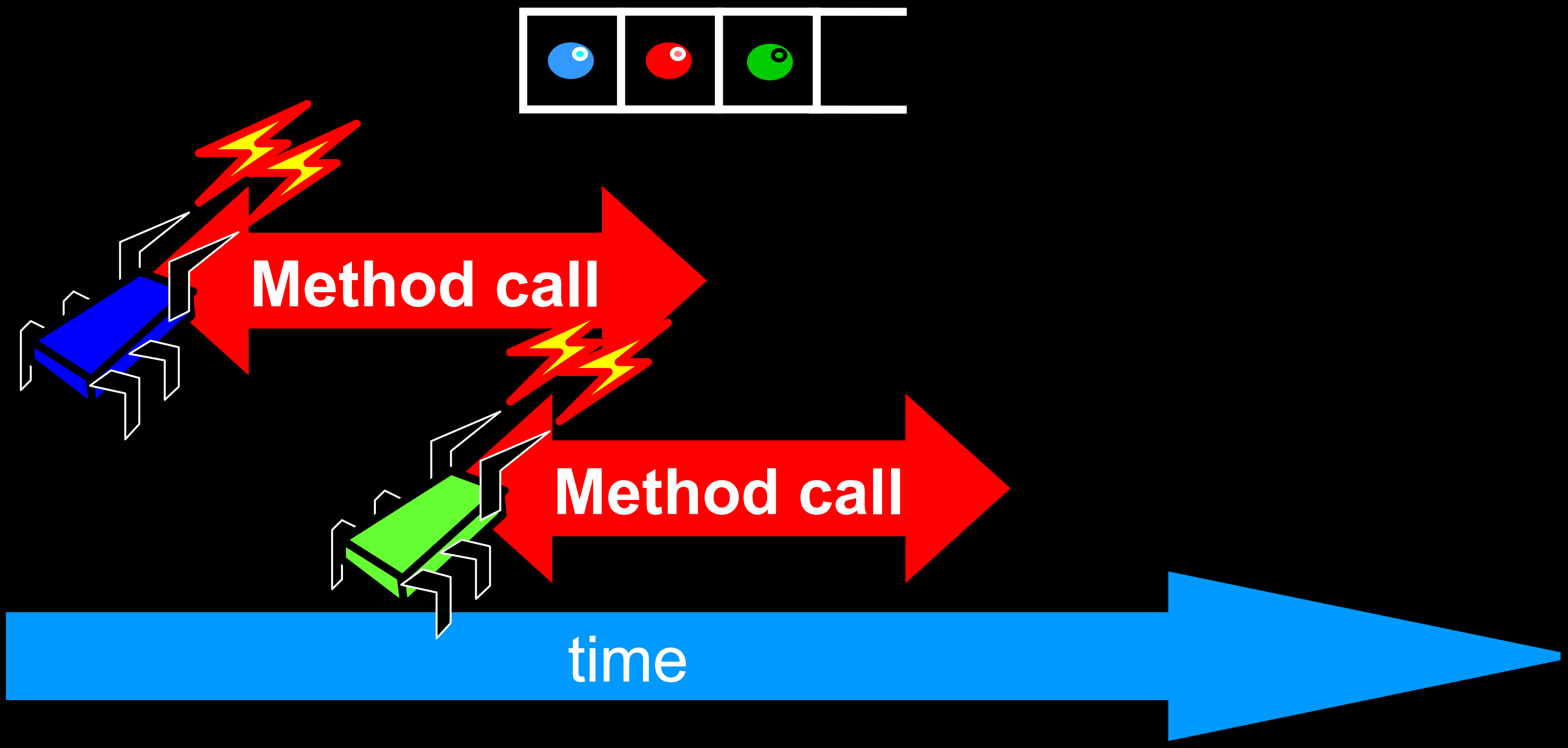


Concurrent Methods Take *Overlapping* Time

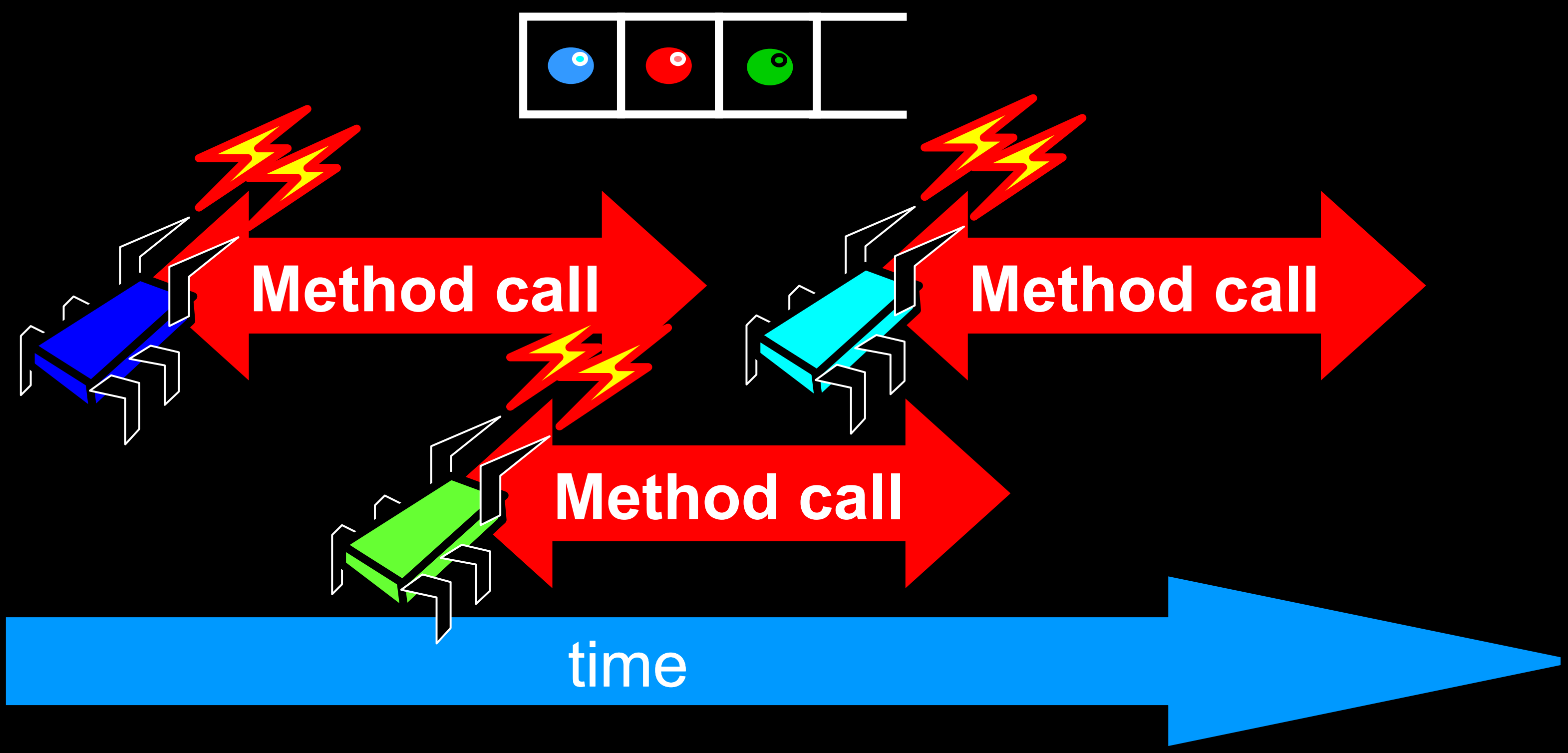


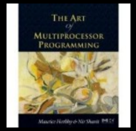


Concurrent Methods Take *Overlapping* Time



Concurrent Methods Take *Overlapping* Time





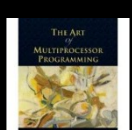
Sequential vs Concurrent

Sequential

Object needs meaningful state only *between* method calls

Concurrent

Because method calls overlap, object might *never* be between method calls



Sequential vs Concurrent

Sequential

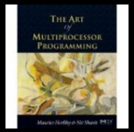
Each method described in isolation

Concurrent

Must characterize *all* possible interactions with concurrent calls

What if two **`enq`**`()` calls overlap?

Two **`deq`**`()` calls? **`enq`**`()` and **`deq`**`()`? …



Sequential vs Concurrent

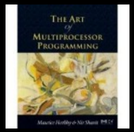
Sequential

Can add new methods without affecting older methods' specs

Concurrent

Adding a method can potentially rewrite every other method's spec

Panic!



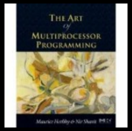
Today’s Big Question

What does it even *mean* for a concurrent object to be correct?

What *is* a concurrent FIFO queue?

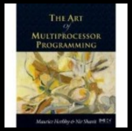
FIFO means strict temporal order

Concurrent means ambiguous temporal order



Intuitively…

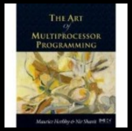
```
void deq()
{
    mutex_lock(&lock);
    if (tail == head) {
       /* Error */
       mutex_unlock(&lock);
    }
    T x = items[head % capacity];
    head++;
    mutex_unlock(&lock);
    return x;
}
```



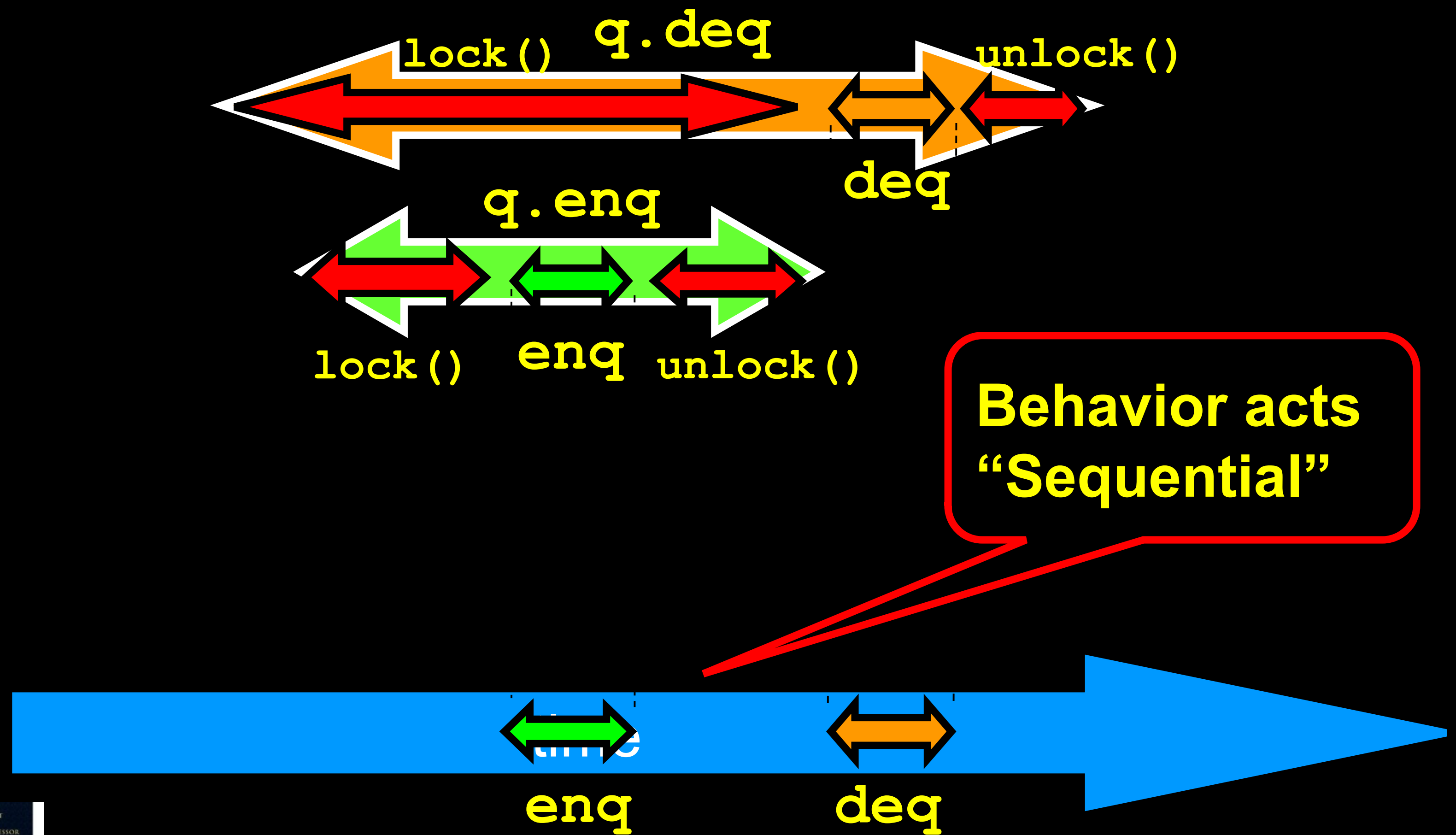
Intuitively…

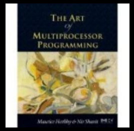
```
void deq()
{
    mutex_lock(&lock);
    if (tail == head) {
       /* Error */
       mutex_unlock(&lock);
    }
    T x = items[head % capacity];
    head++;
    mutex_unlock(&lock);
    return x;
}
```

All queue modifications are mutually exclusive



Intuitively





Linearizability

Each method call should

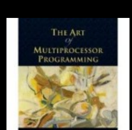
"take effect"

Instantaneously

Between invocation and response events

Object is correct if "sequential" behavior is correct

Any such concurrent object is *linearizable*



Linearizability

Each method call should

“take effect”

Instantaneously

Between invocation and response events

Actually a property of an execution

A *linearizable* object: one all of whose possible *executions* are linearizable