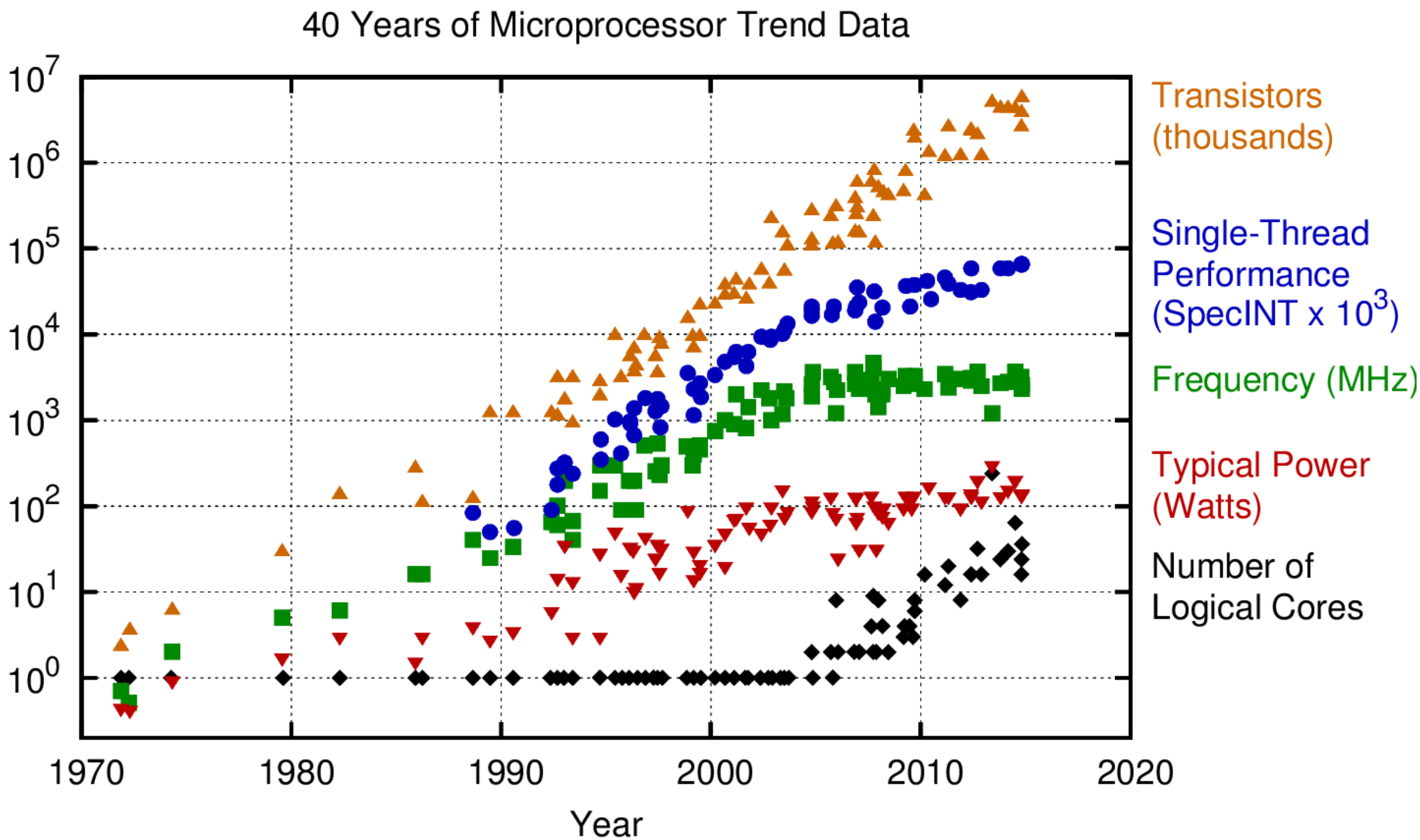
CSCI-GA.02110

Programming Languages

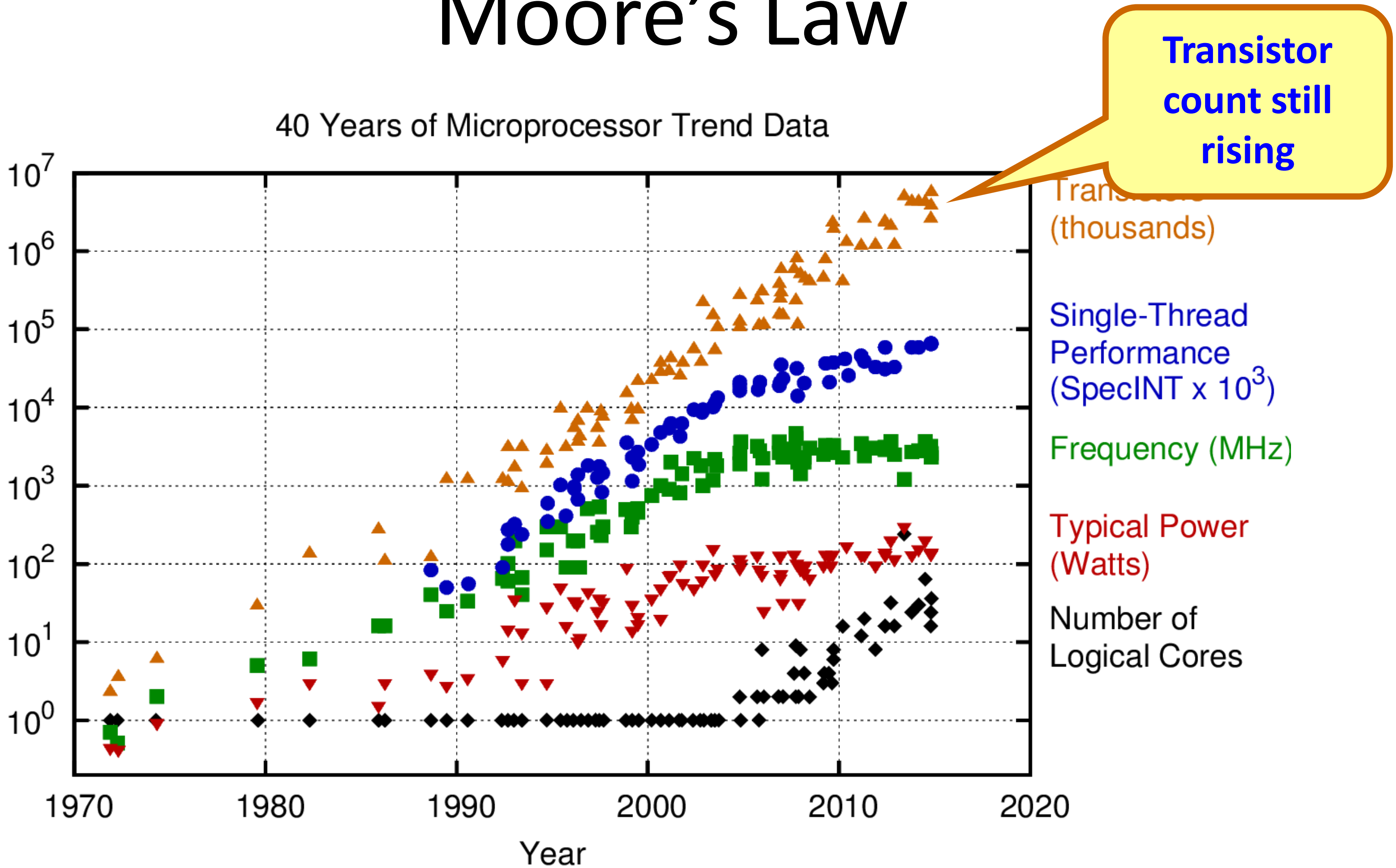
Concurrency

Moore’s Law



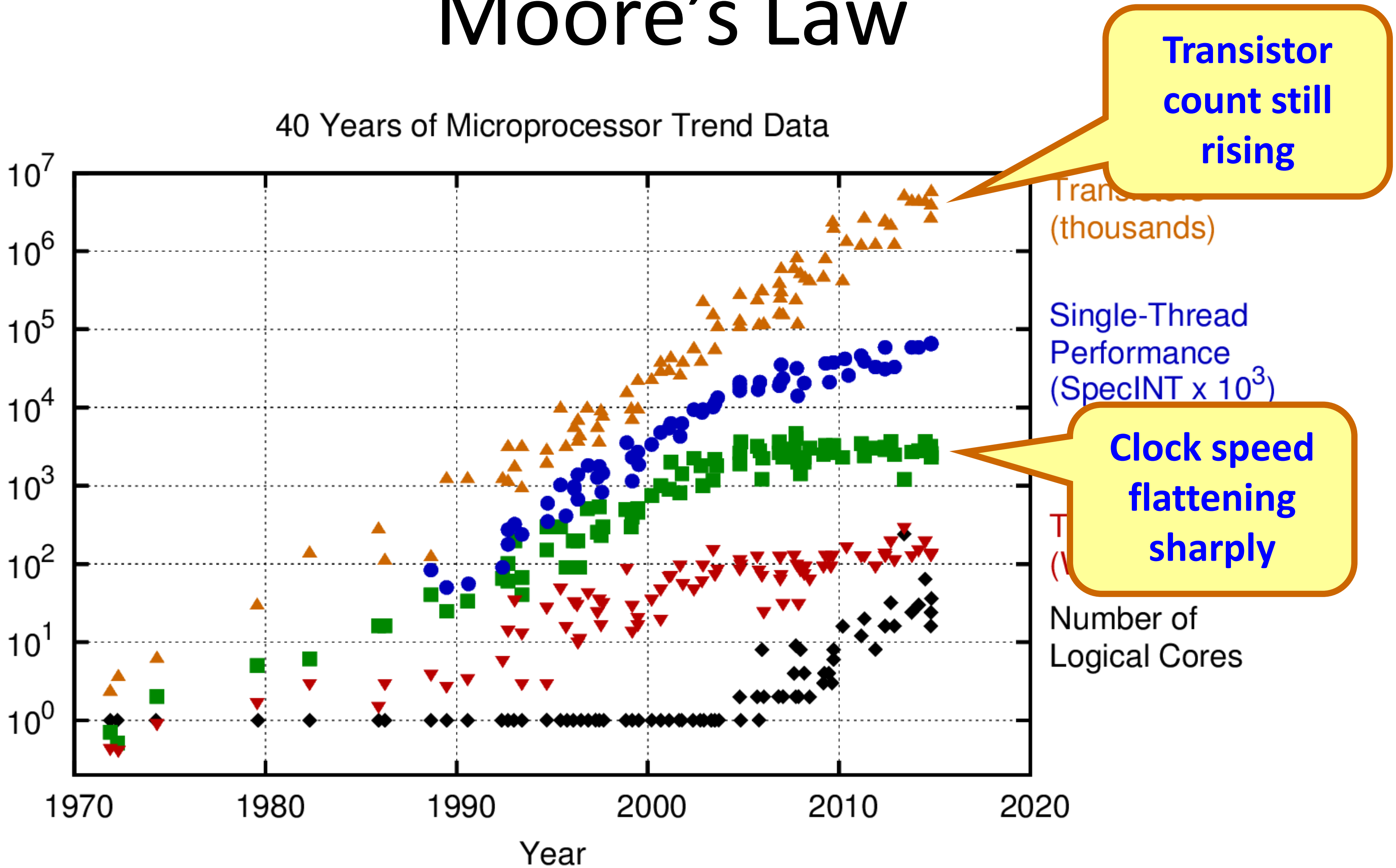
Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp

Moore's Law



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp

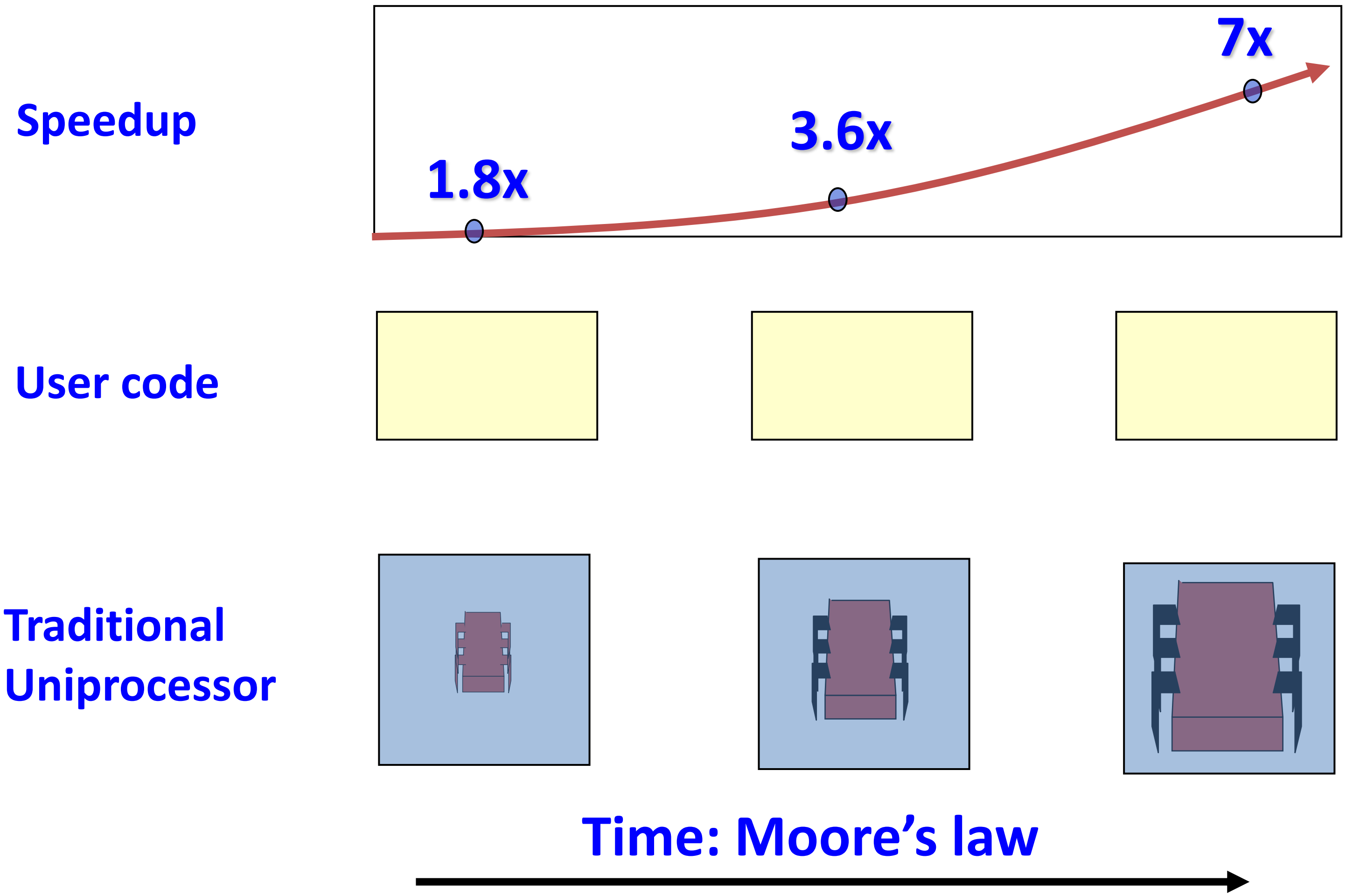
Moore’s Law



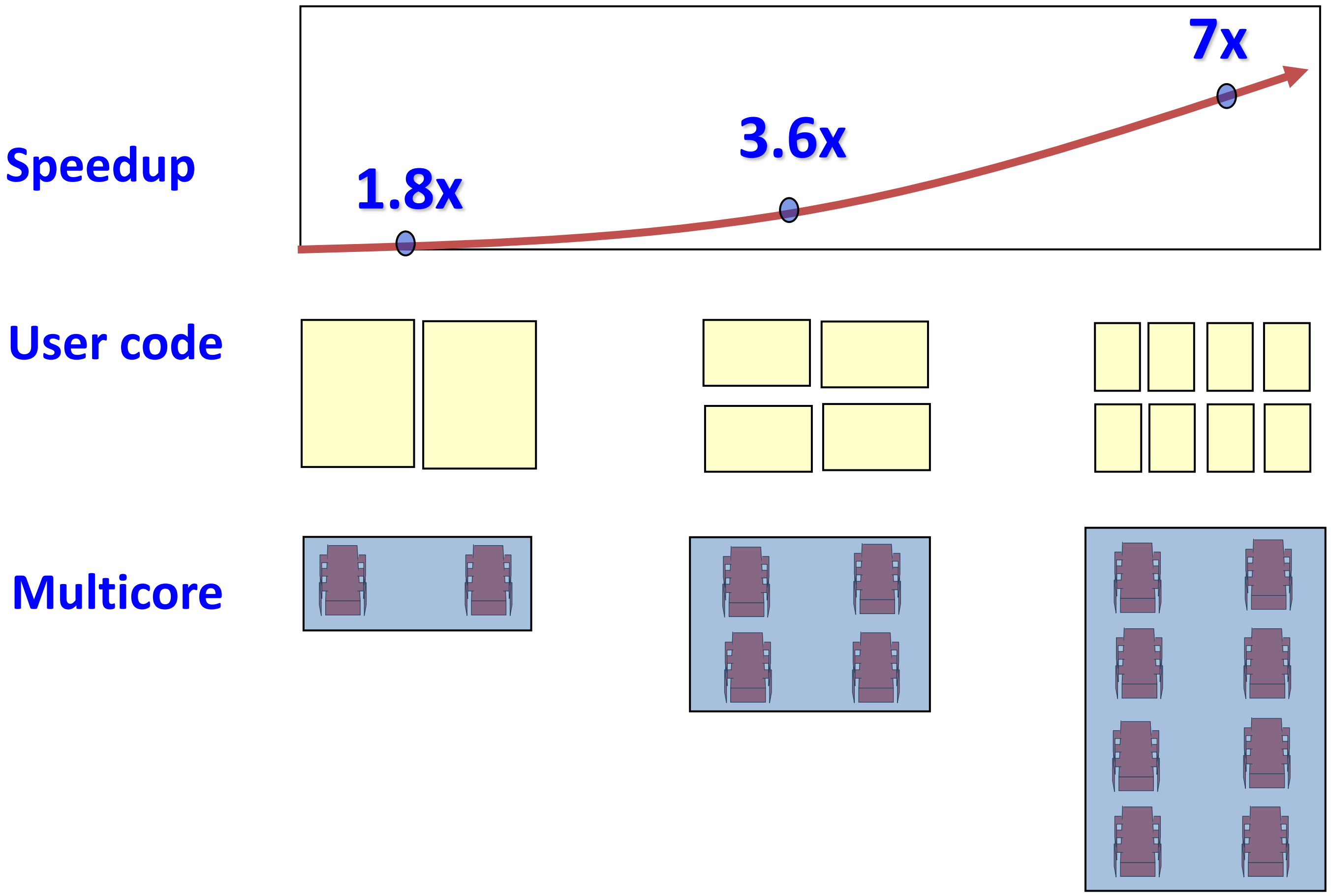
Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp

Moore’s Law (in practice)

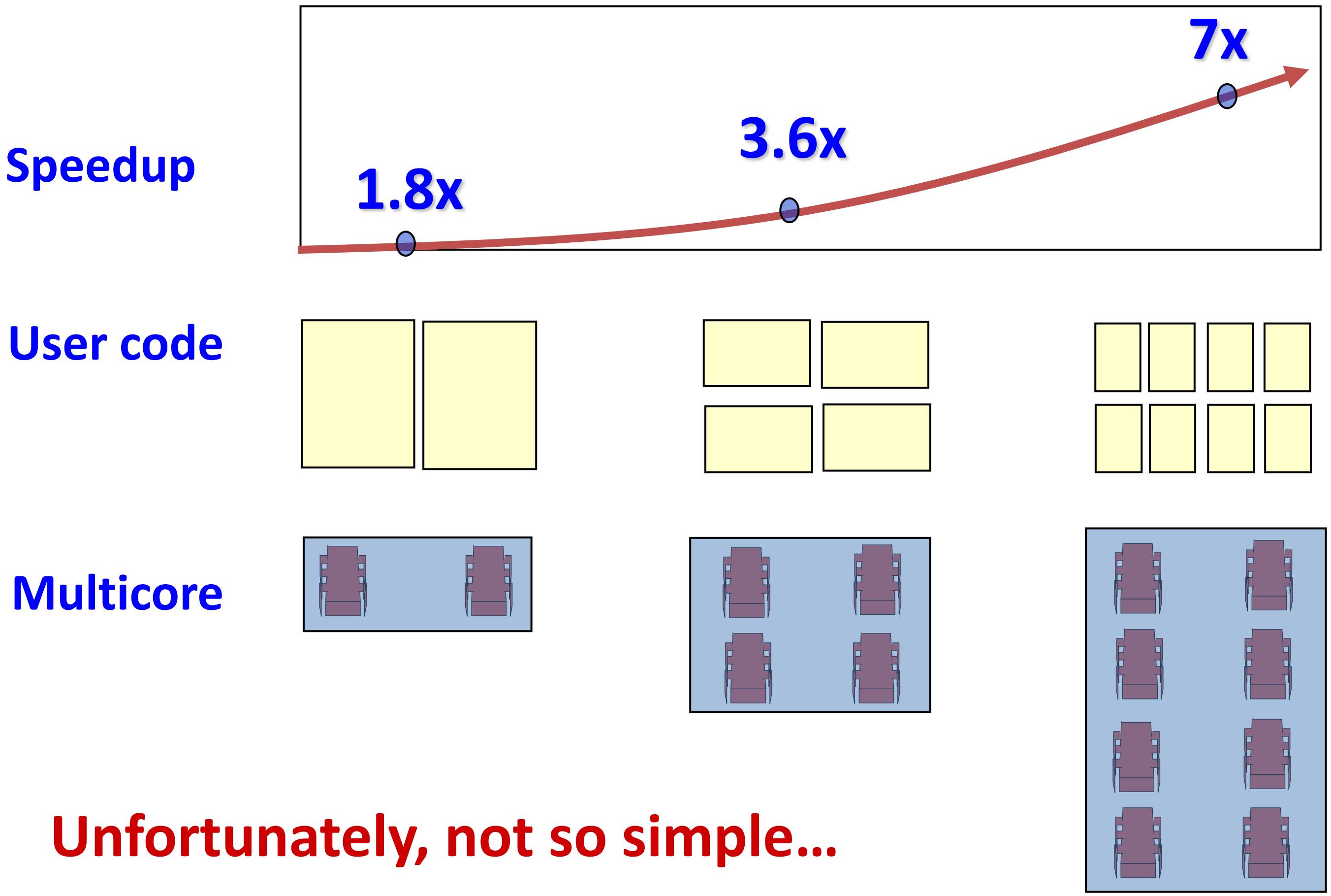
Traditional Scaling Process



Ideal Scaling Process

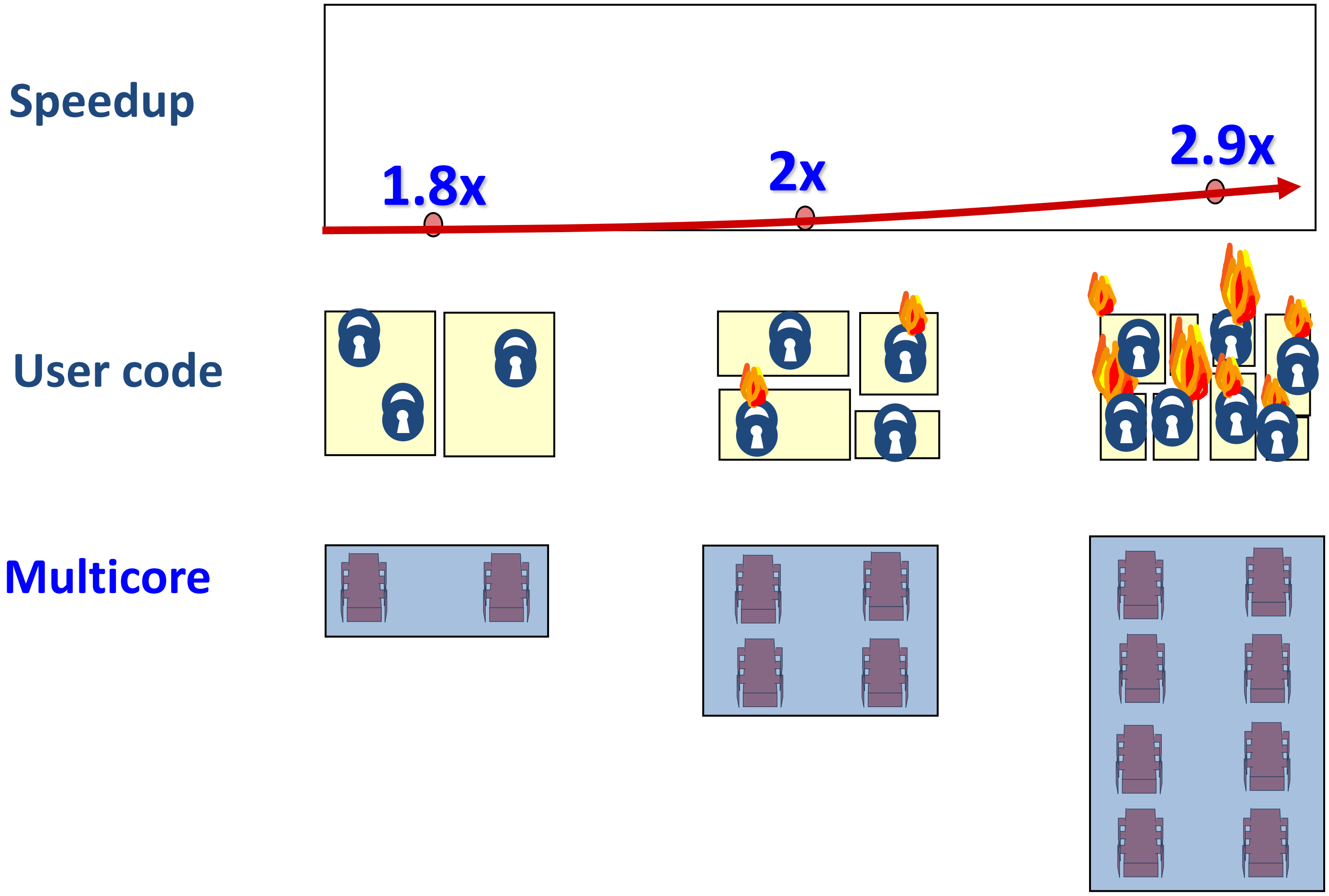


Ideal Scaling Process

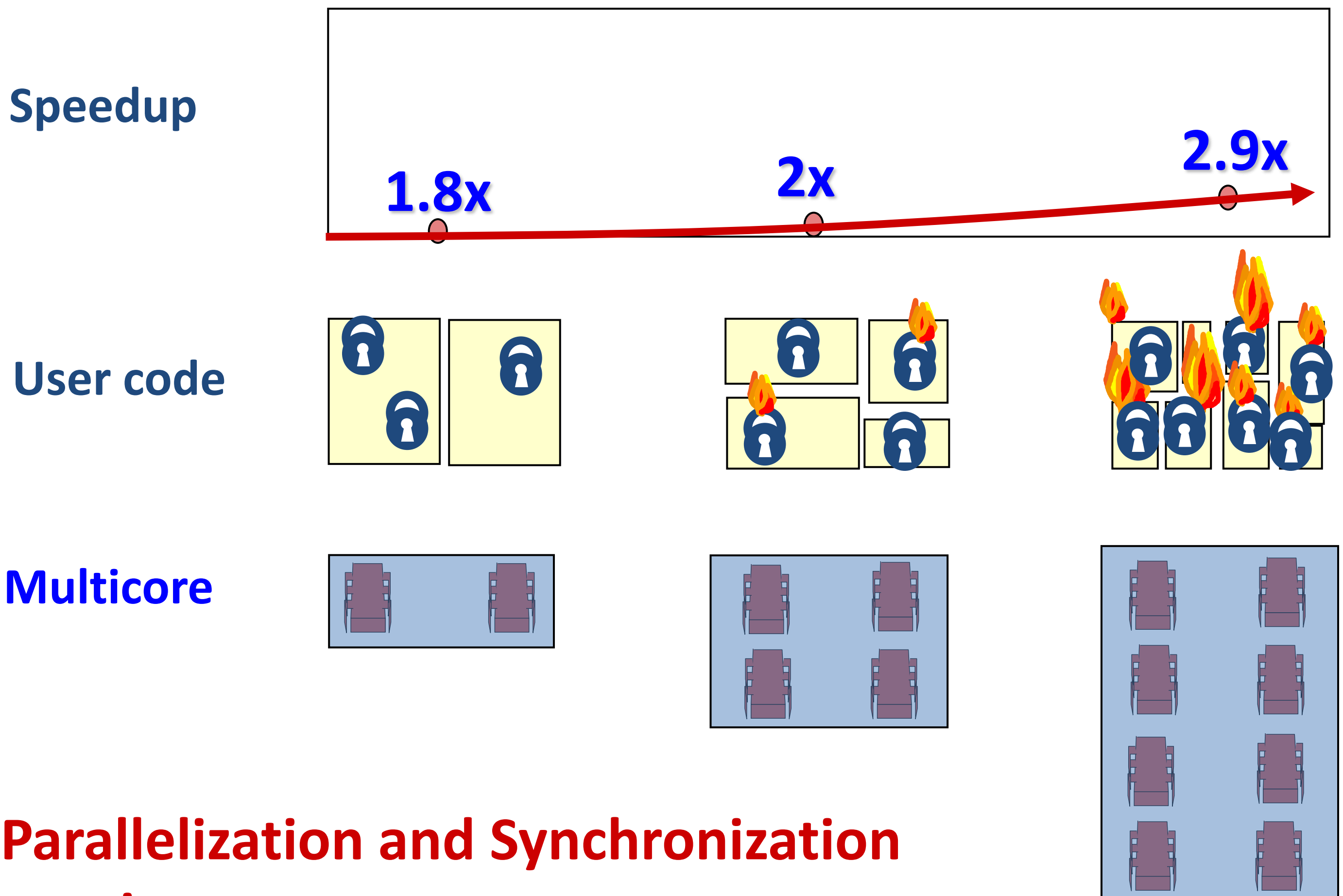


Unfortunately, not so simple…

Actual Scaling Process



Actual Scaling Process



Parallelization and Synchronization require great care…

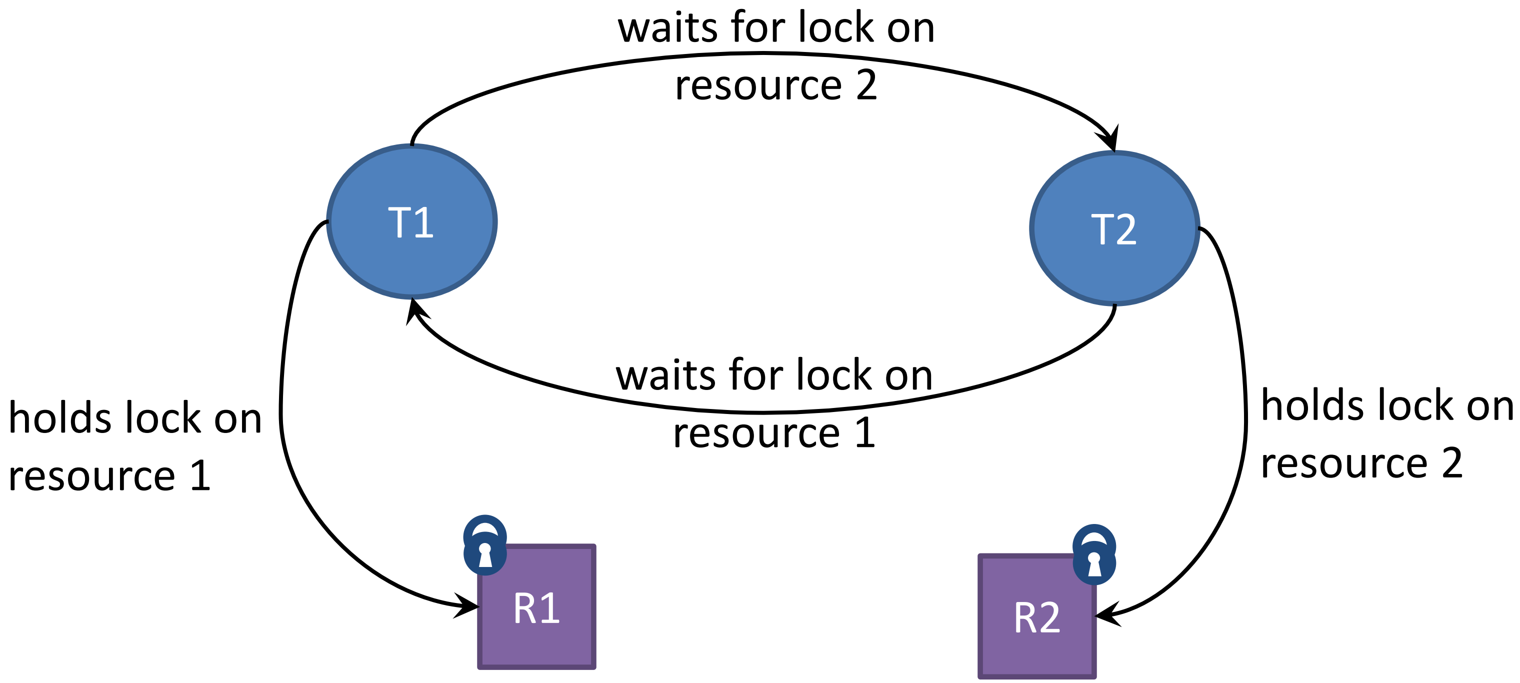
Concurrent Programming

- In order to keep scaling, programs now need to take advantage of parallelism
  - multi-core programming
  - distributed programming (aka cloud computing)

- What are good programming language abstractions for dealing with concurrency?
  - this is still an active area of research
  - let's look at some of the issues and some of the contenders

Issues with Concurrency

- Deadlock situation



Issues with Concurrency

- **Data race**: two instructions (at least one of which is a write) access the same memory location concurrently

  variable `x` shared by two threads T1 and T2

  ```
  T1: x = x + 1   ||   T2: y = x
  ```

- Data races can cause catastrophic software failures
  - Therac-25 radiation overdose
  - 2003 Northeast power blackout
  - …

- Need locks or similar synchronization mechanism to avoid data races

Issues with Concurrency

- **Weak consistency:** concurrent operations may appear to happen out-of-order.

```
T1: x = 1     ||  T2: y = 1
    print y   ||      print x
```

Can this program print `00` if the initial state is `x == y == 0`?

Issues with Concurrency

- **Weak consistency:** concurrent operations may appear to happen out-of-order.

```
T1: x = 1     ||  T2: y = 1
    print y   ||      print x
```

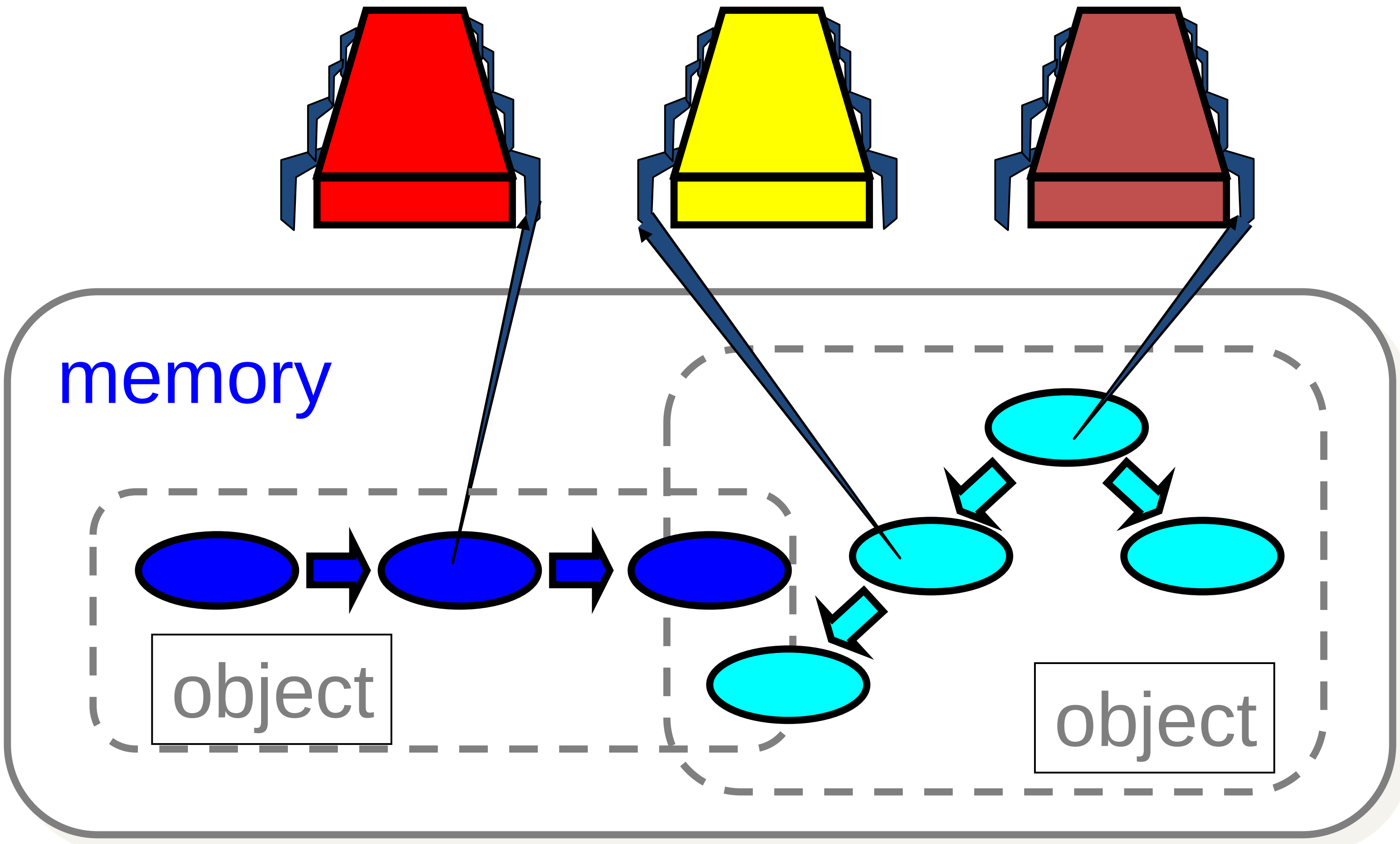
Can this program print `00` if the initial state is `x == y == 0`?

Yes!

Concurrent Programming Paradigms

- Shared memory concurrency (typically used for low-level code)
  - Shared linearizable objects
  - Monitors / Locks / Mutexes
  - Atomic machine-level instructions: compare-and-swap (CAS), fetch-and-add, fences/barriers, …
- Message passing concurrency
  - Actors (Erlang, Scala, Java, …)
  - Futures/Promises (Scala, OCaml, …)
  - Channels (Go)

Concurrent Computation



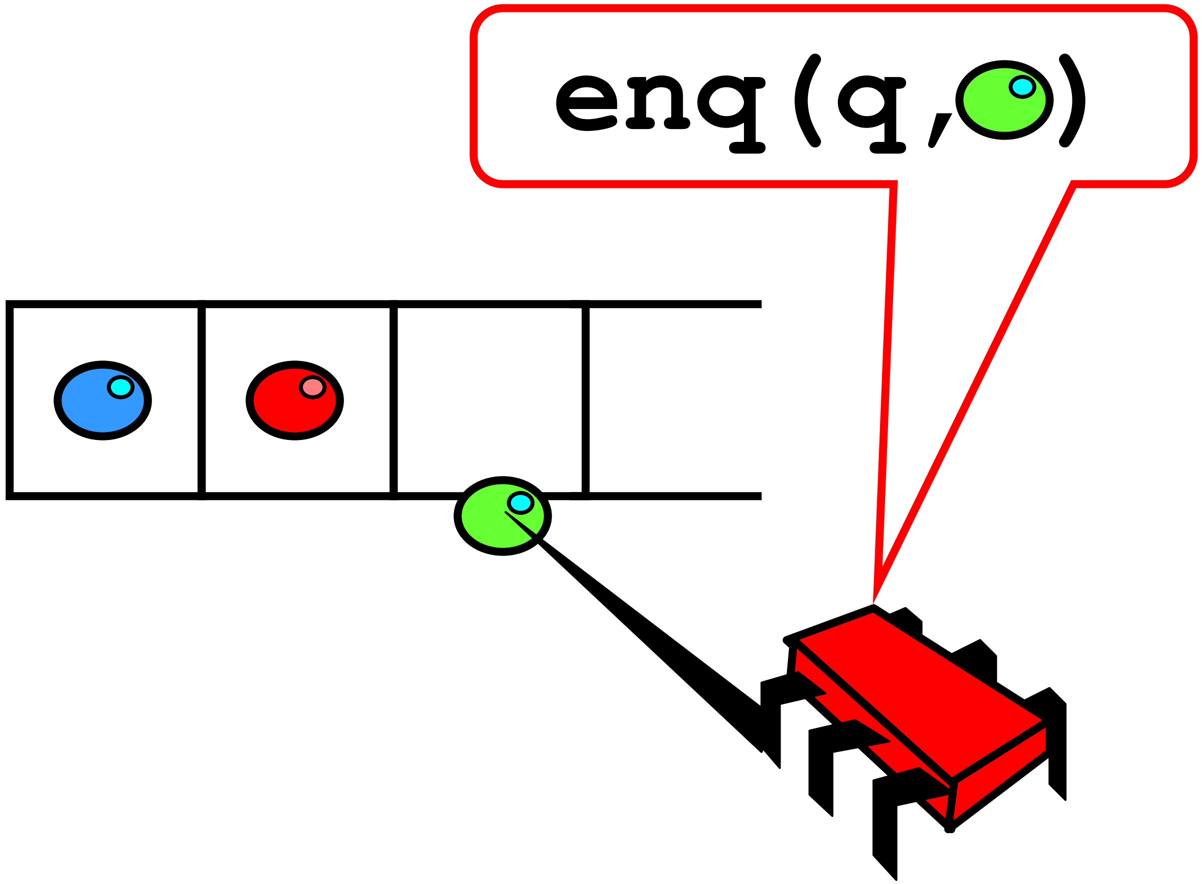
Objectivism

- What is a concurrent object?
  - How do we **describe** one?
  - How do we **implement** one?
  - How do we **tell if it is correct**?

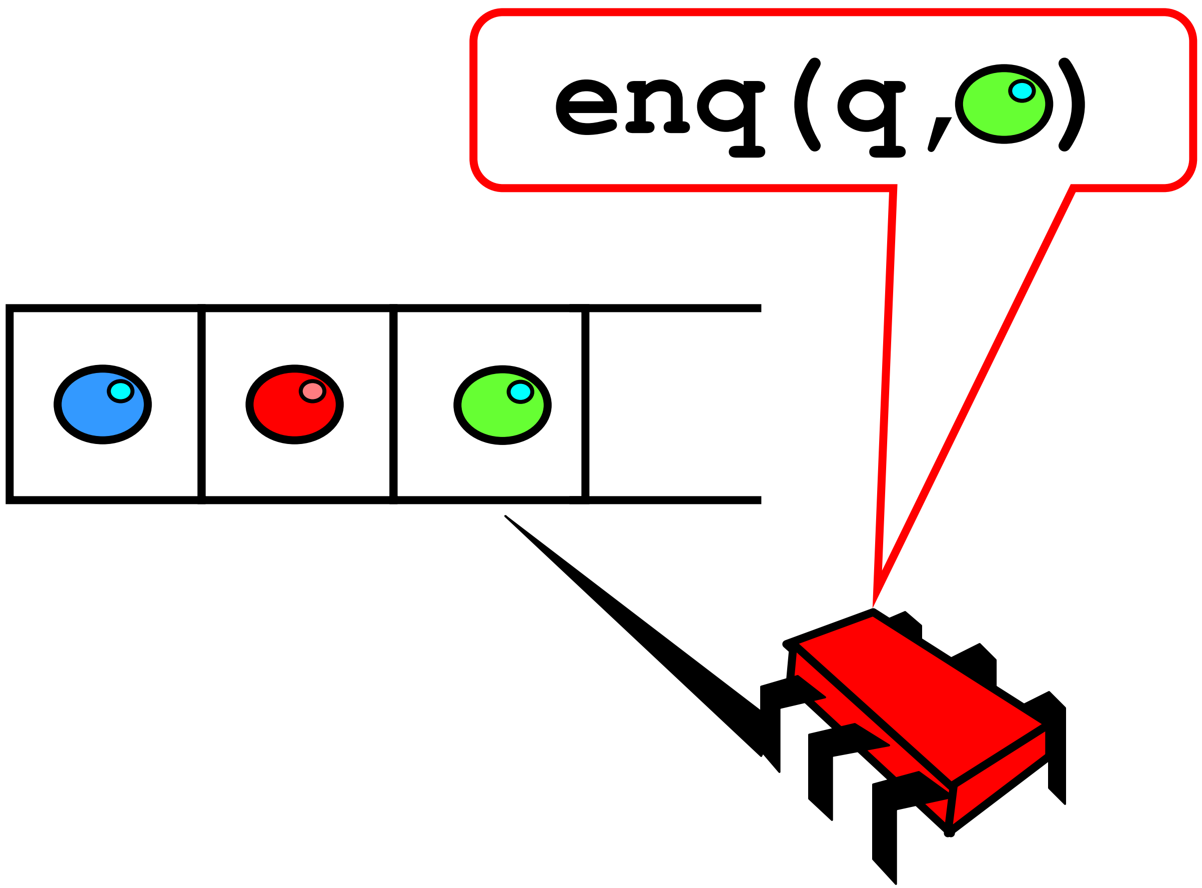
Objectivism

- What is a concurrent object?
  - How do we **describe** one?
  - How do we **tell if it is correct**?

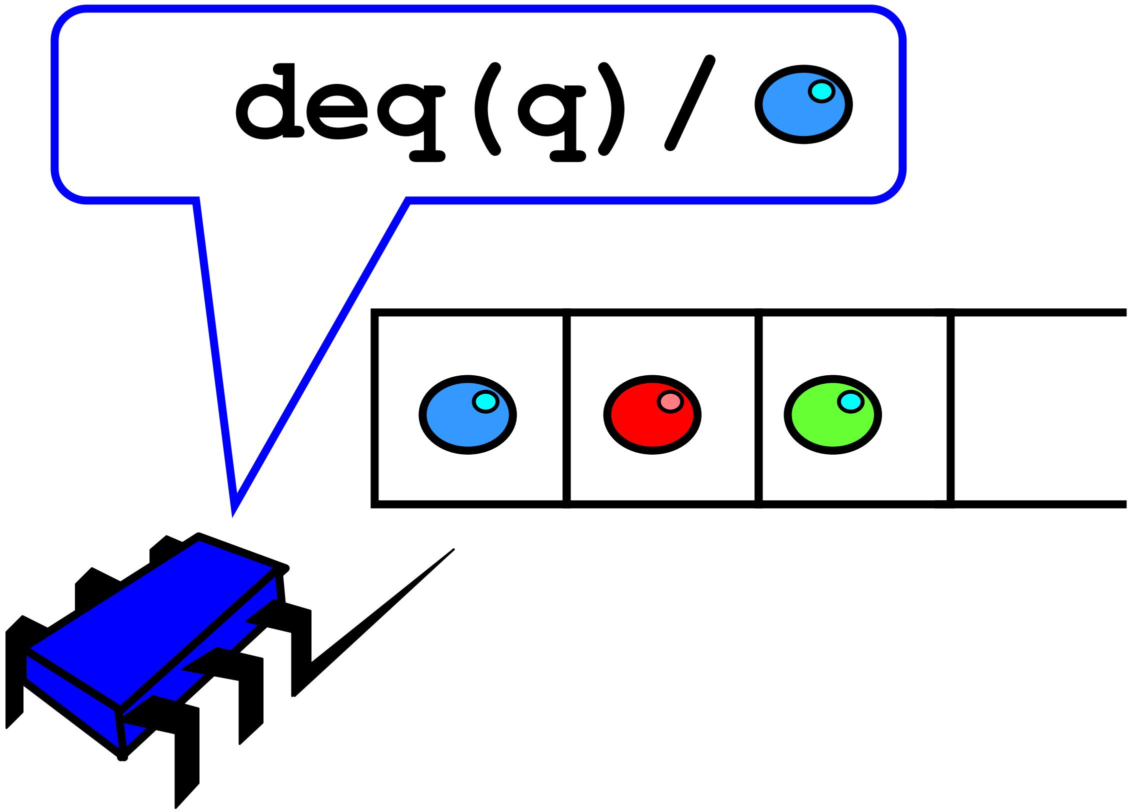
FIFO Queue: Enqueue Method



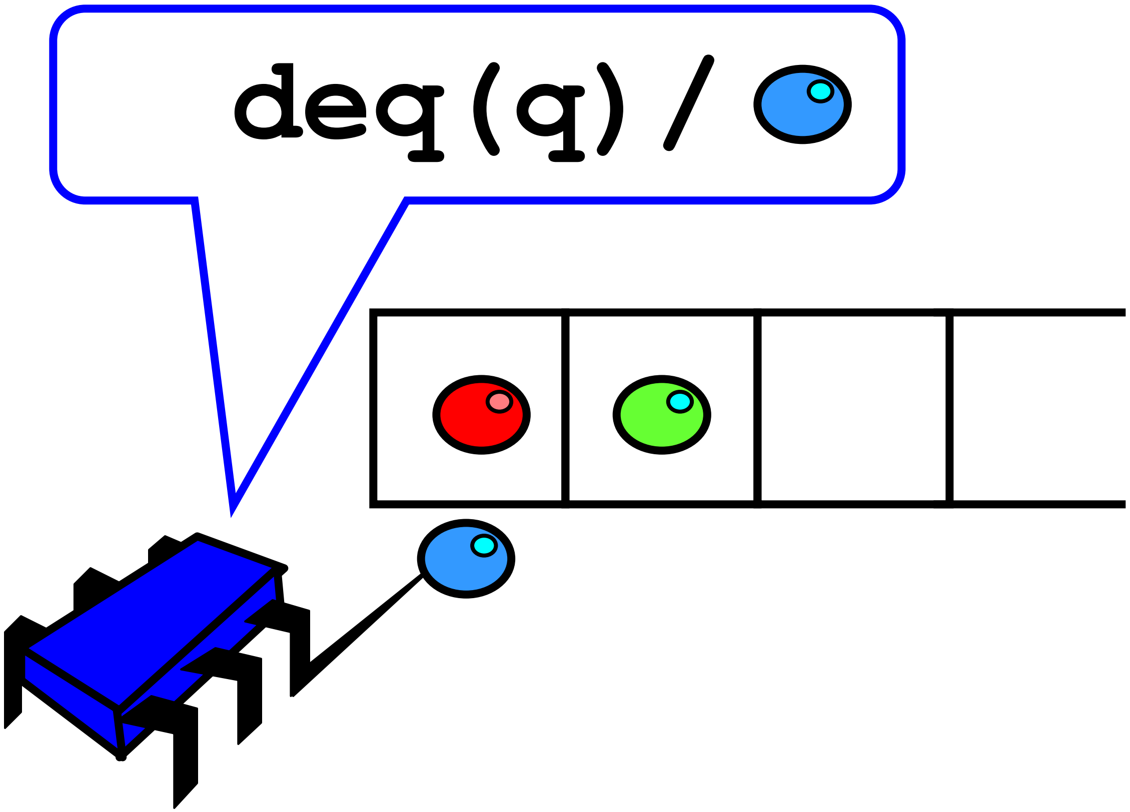
FIFO Queue: Enqueue Method



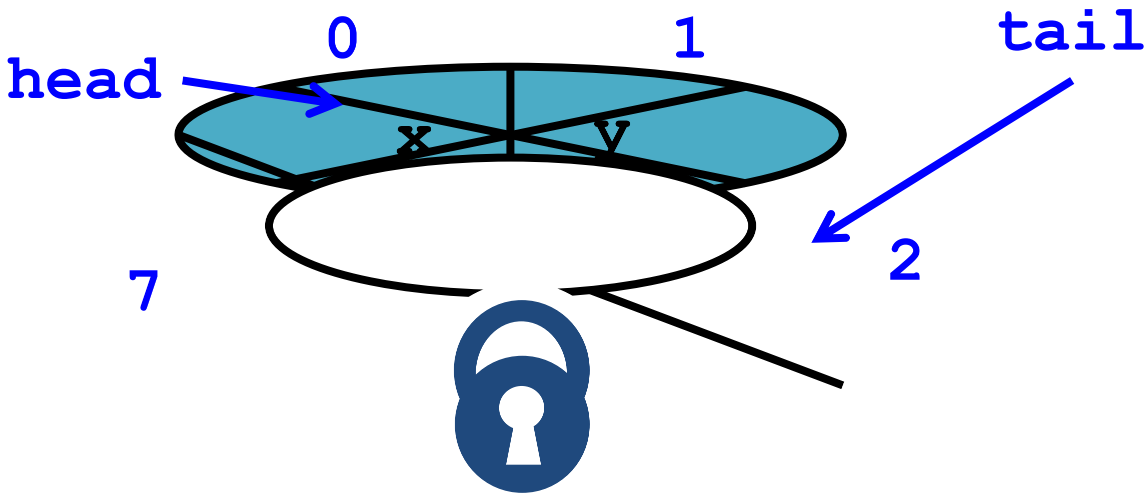
FIFO Queue: Dequeue Method



FIFO Queue: Dequeue Method

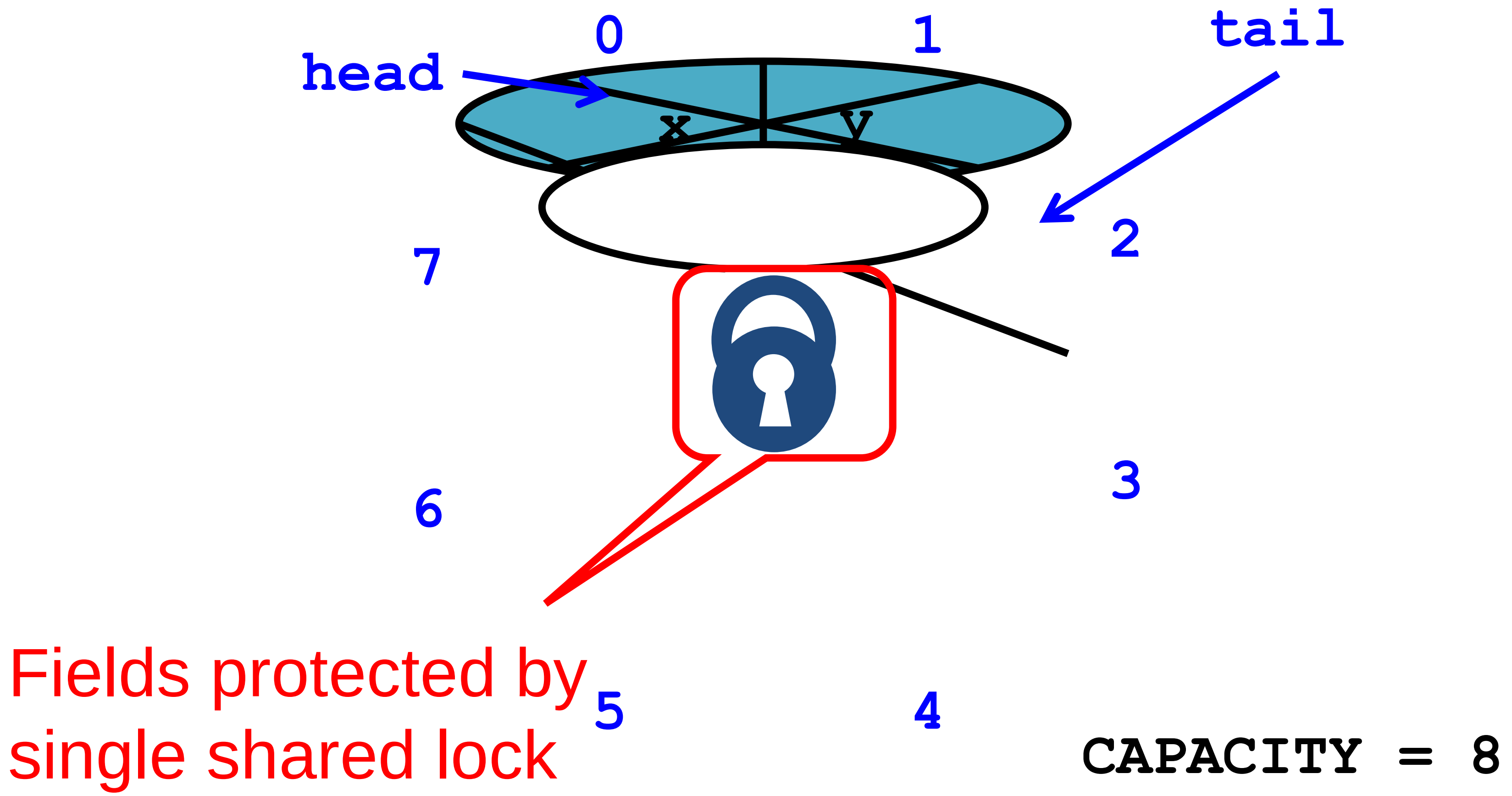


Lock-Based Queue



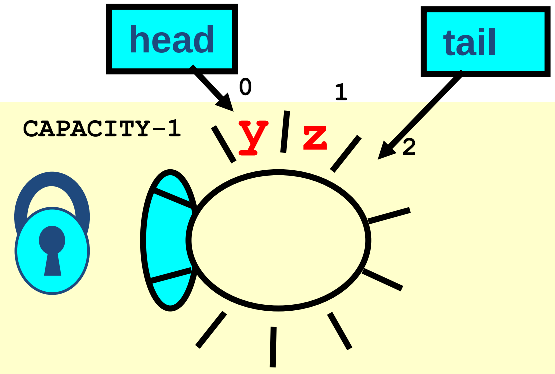
CAPACITY = 8

Lock-Based Queue



A Lock-Based Queue

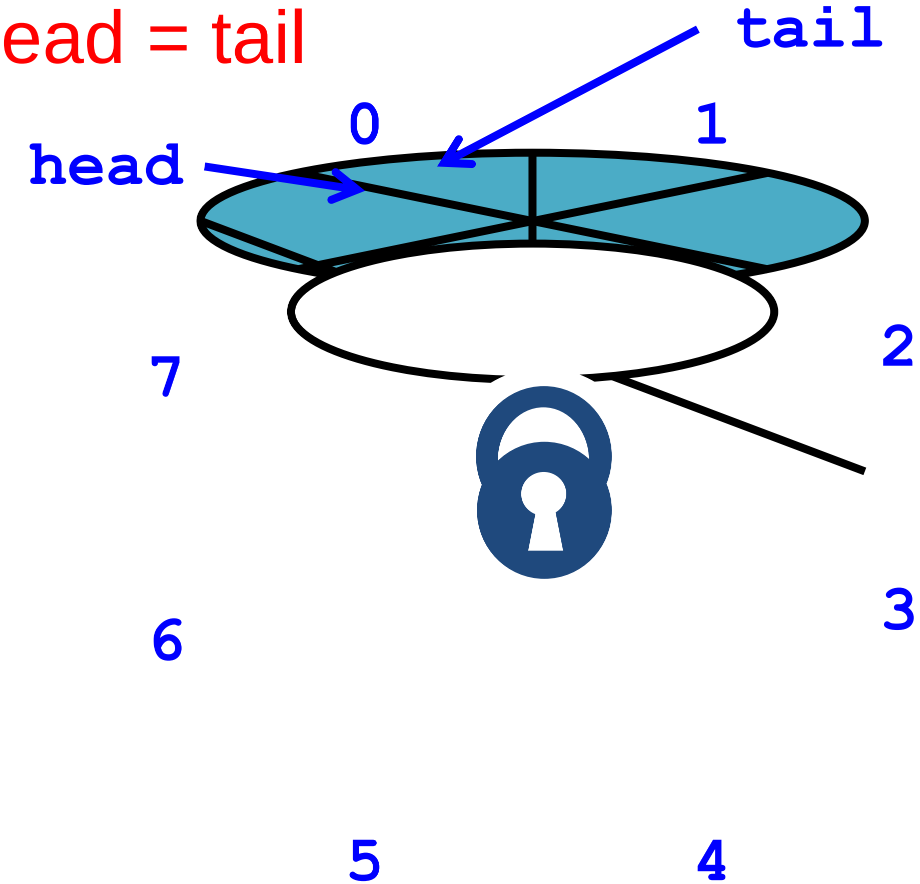
```
typedef struct {
  int head, tail;
  void* items[CAPACITY];
  phread_mutex_t lock;
} queue_t;
```



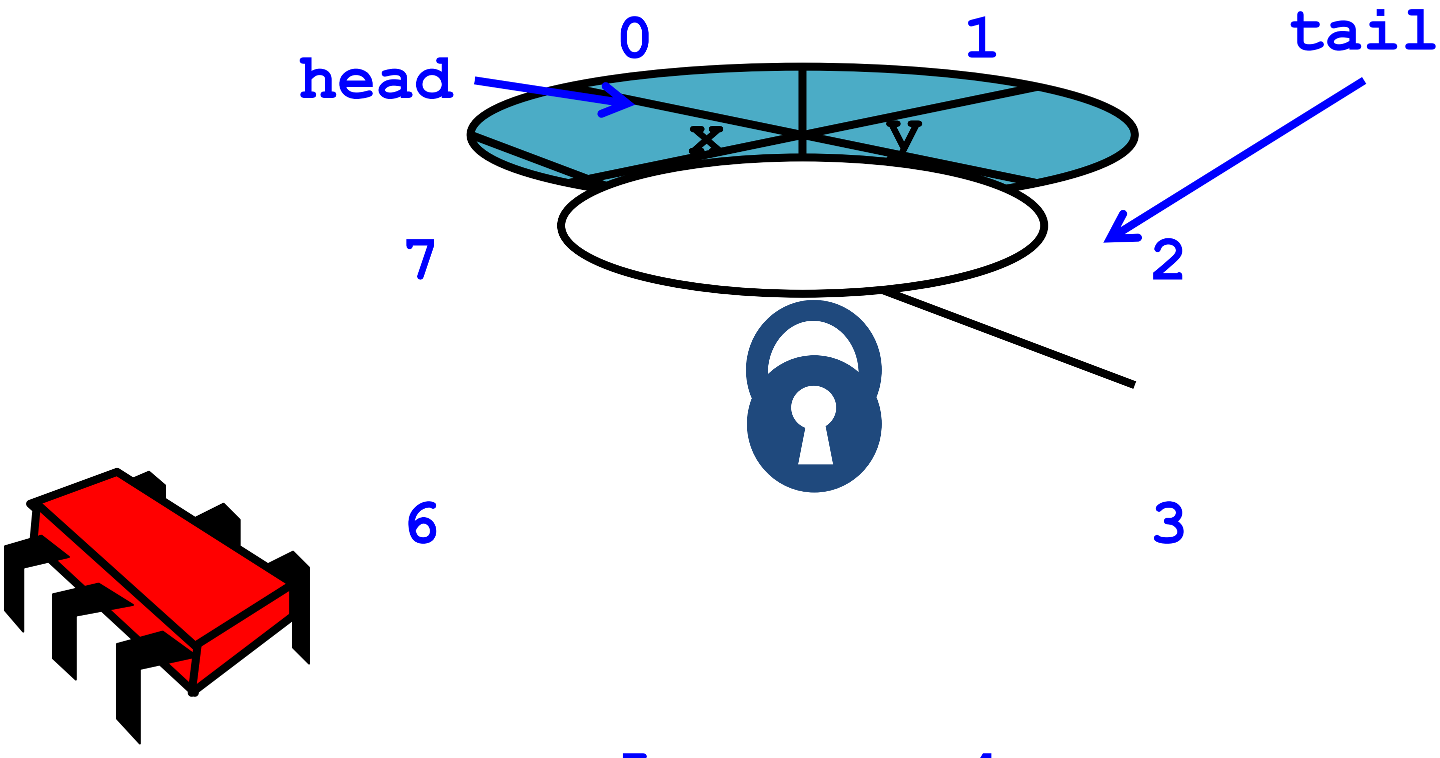
Fields protected by single shared lock

Lock-Based Queue

Initially head = tail

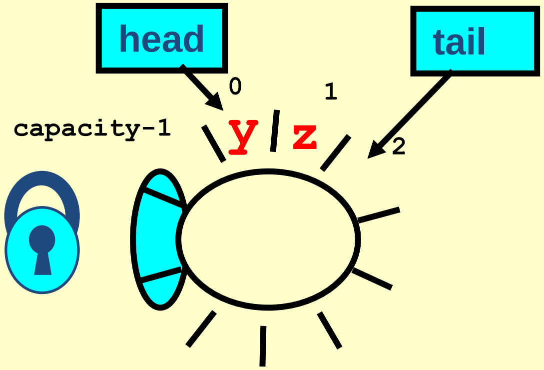


Lock-Based deq()

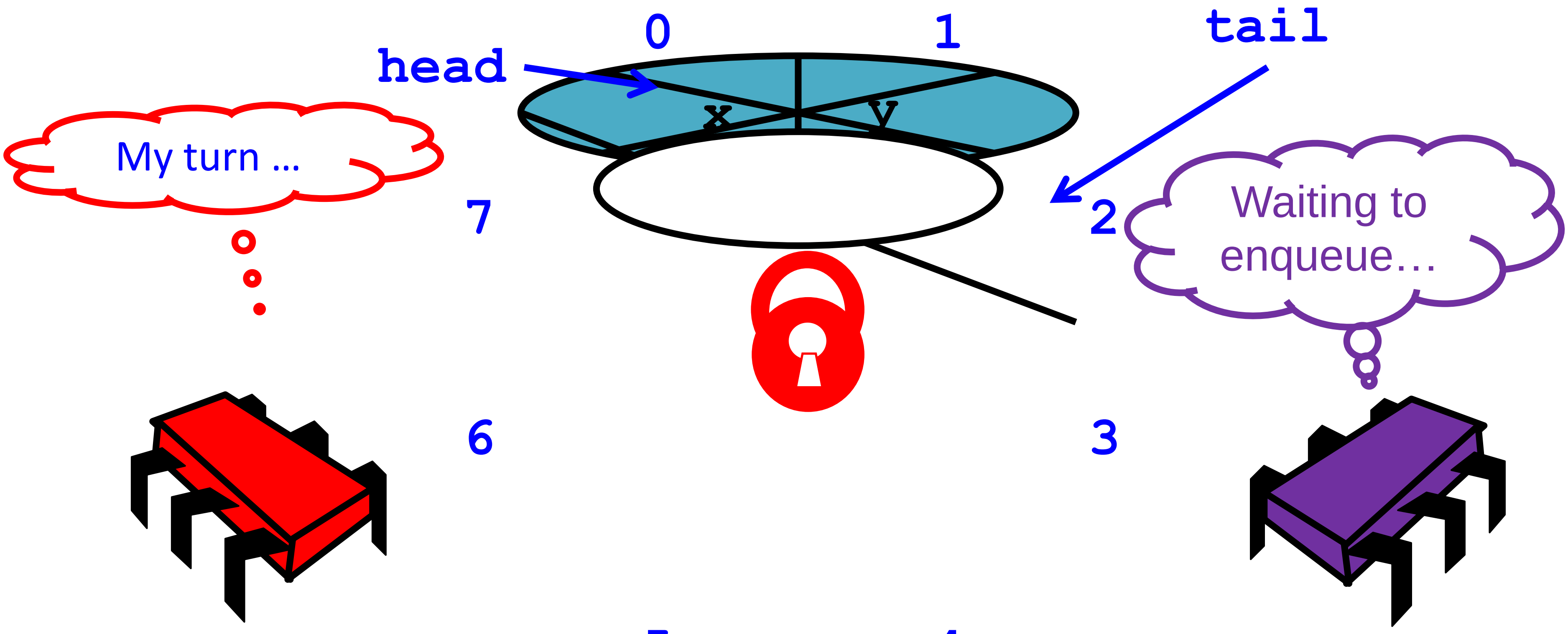


Implementation: deq()

```
int deq(queue_t q, void **elem) {
  int res;
  pthread_mutex_lock(&q->lock);
  if (q->tail == q->head) res = 0;
  else {
    *elem = q->items[q->head % CAPACITY];
    q->head++;
    res = 1;
  }
  pthread_mutex_unlock(&q->lock);
  return res;
}
```



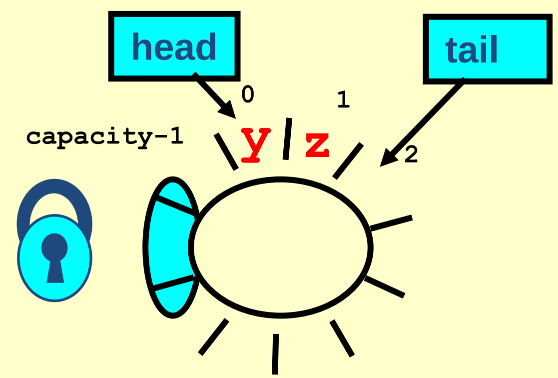
Acquire Lock



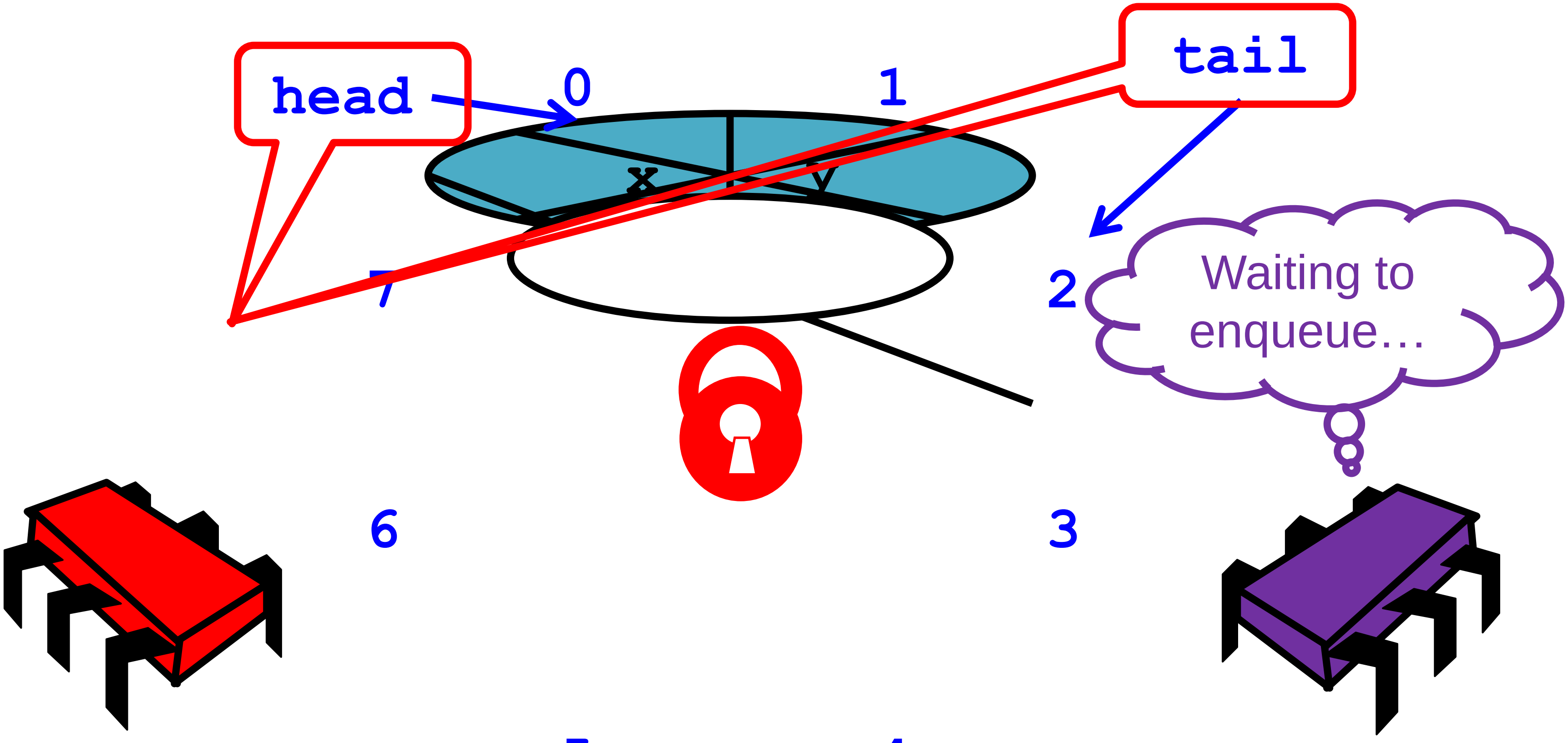
Implementation: deq()

```
int deq(queue_t q, void **elem) {
  int res;
  pthread_mutex_lock(&q->lock);
  if (q->tail == q->head) res = 0;
  else {
    *elem = q->items[q->head % CAPACITY];
    q->head++;
    res = 1;
  }
  pthread_mutex_unlock(&q->lock);
  return res;
}
```

Acquire lock at method start



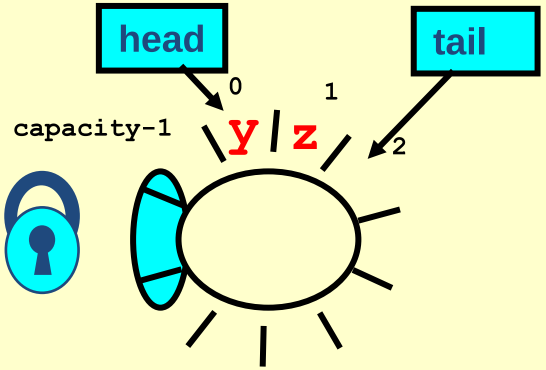
Check if Non-Empty



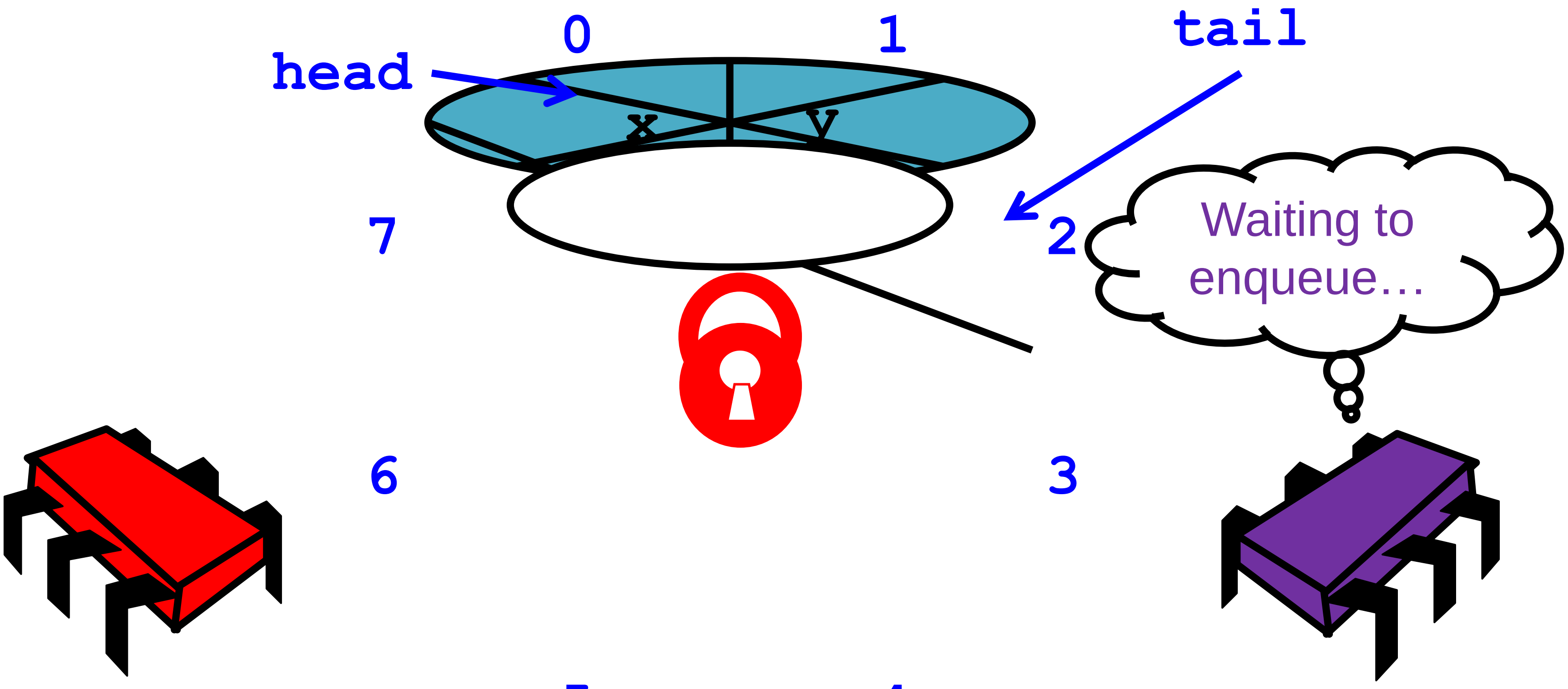
Implementation: deq()

```
int deq(queue_t q, void **elem) {
  int res;
  pthread_mutex_lock(&q->lock);
  if (q->tail == q->head) res = 0;
  else {
    *elem = q->items[q->head % CAPACITY];
    q->head++;
    res = 1;
  }
  pthread_mutex_unlock(&q->lock);
  return res;
}
```

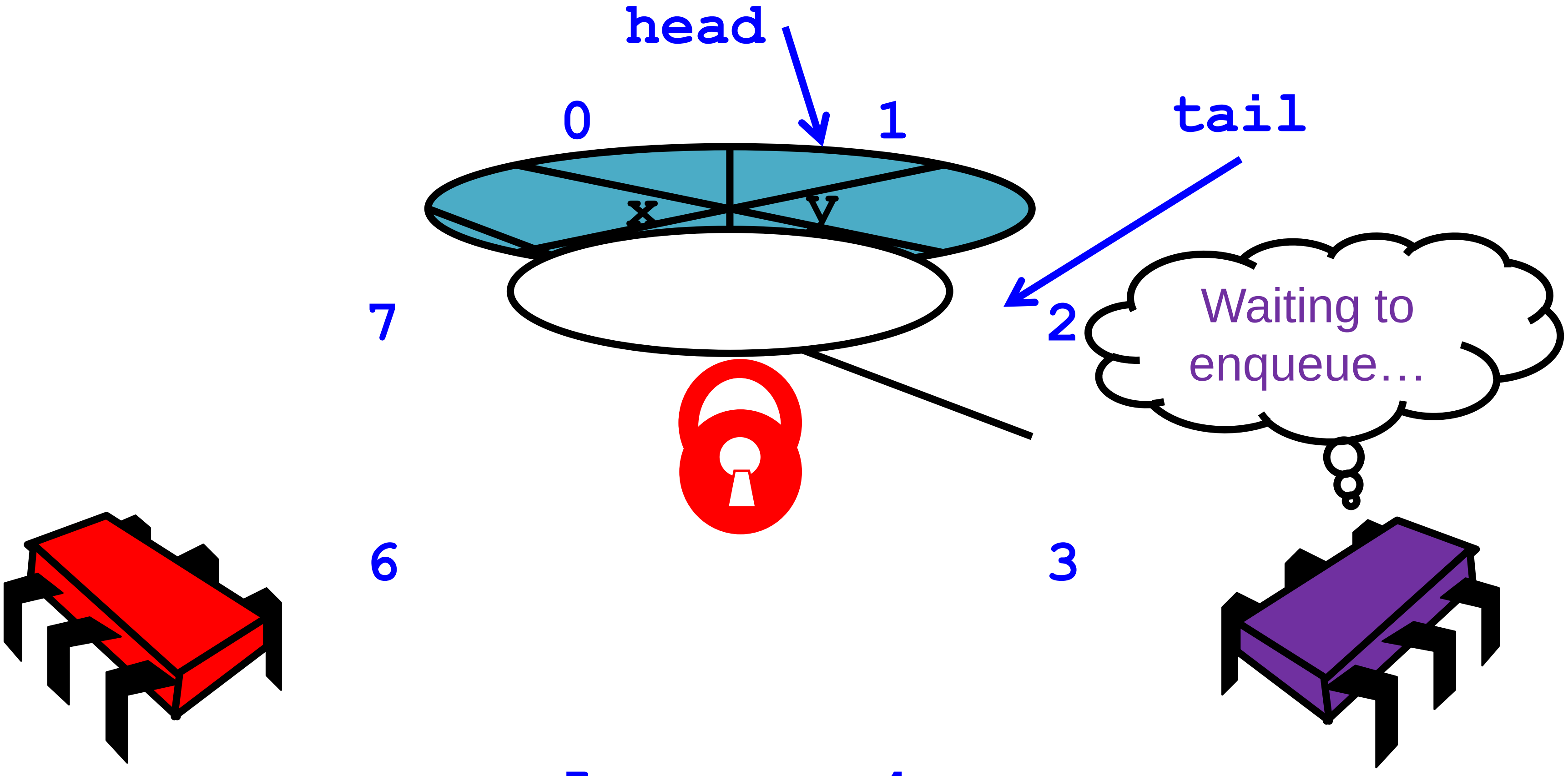
If queue empty
return "failure"



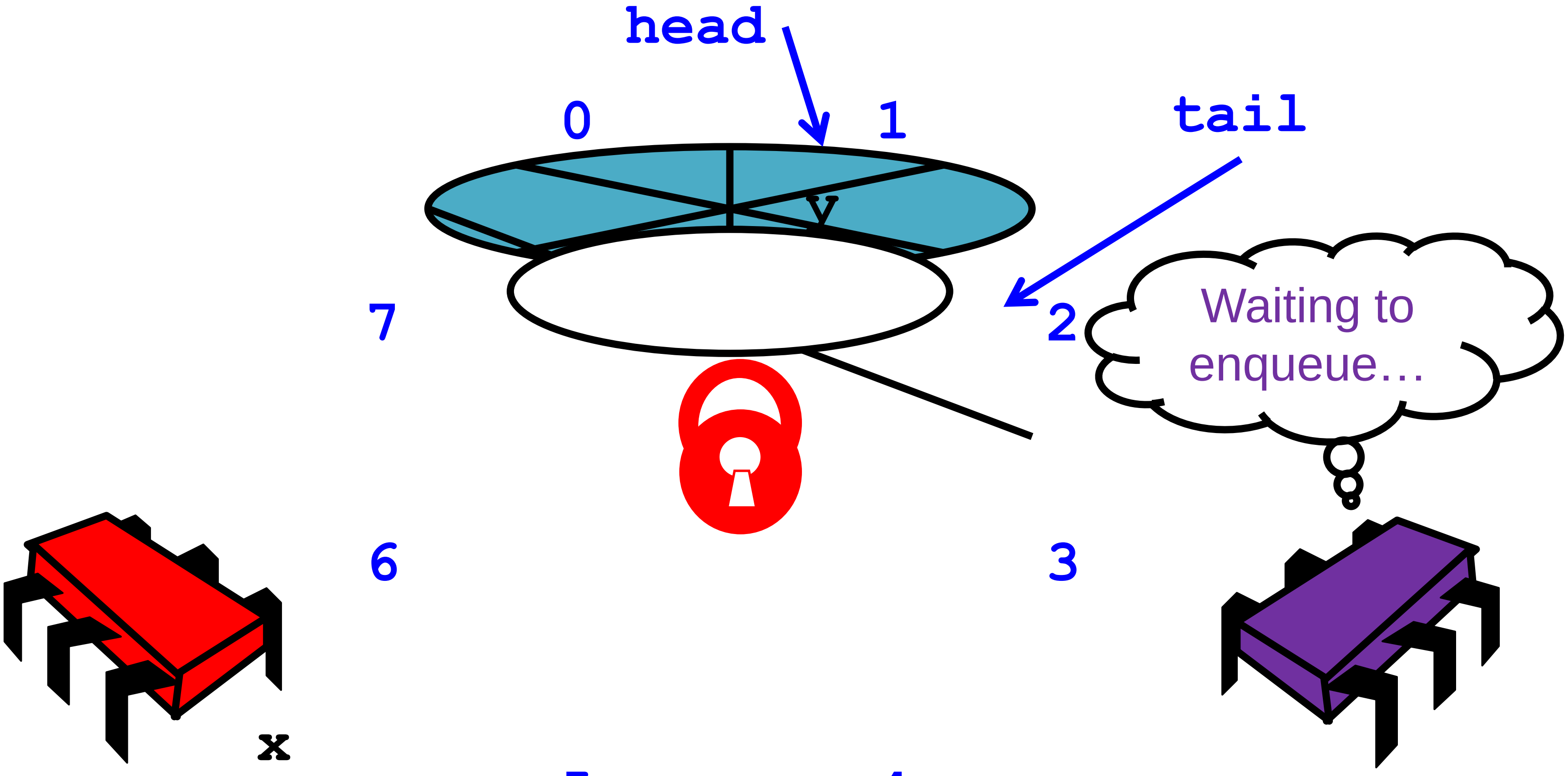
Modify the Queue



Modify the Queue

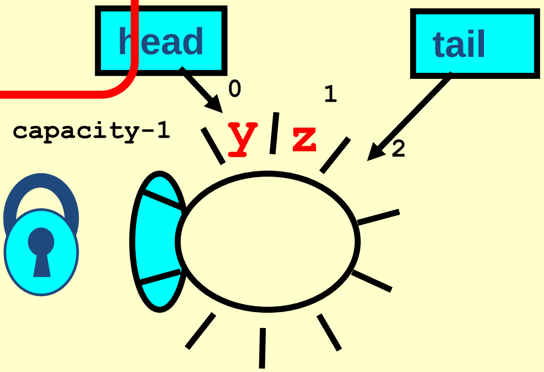


Modify the Queue



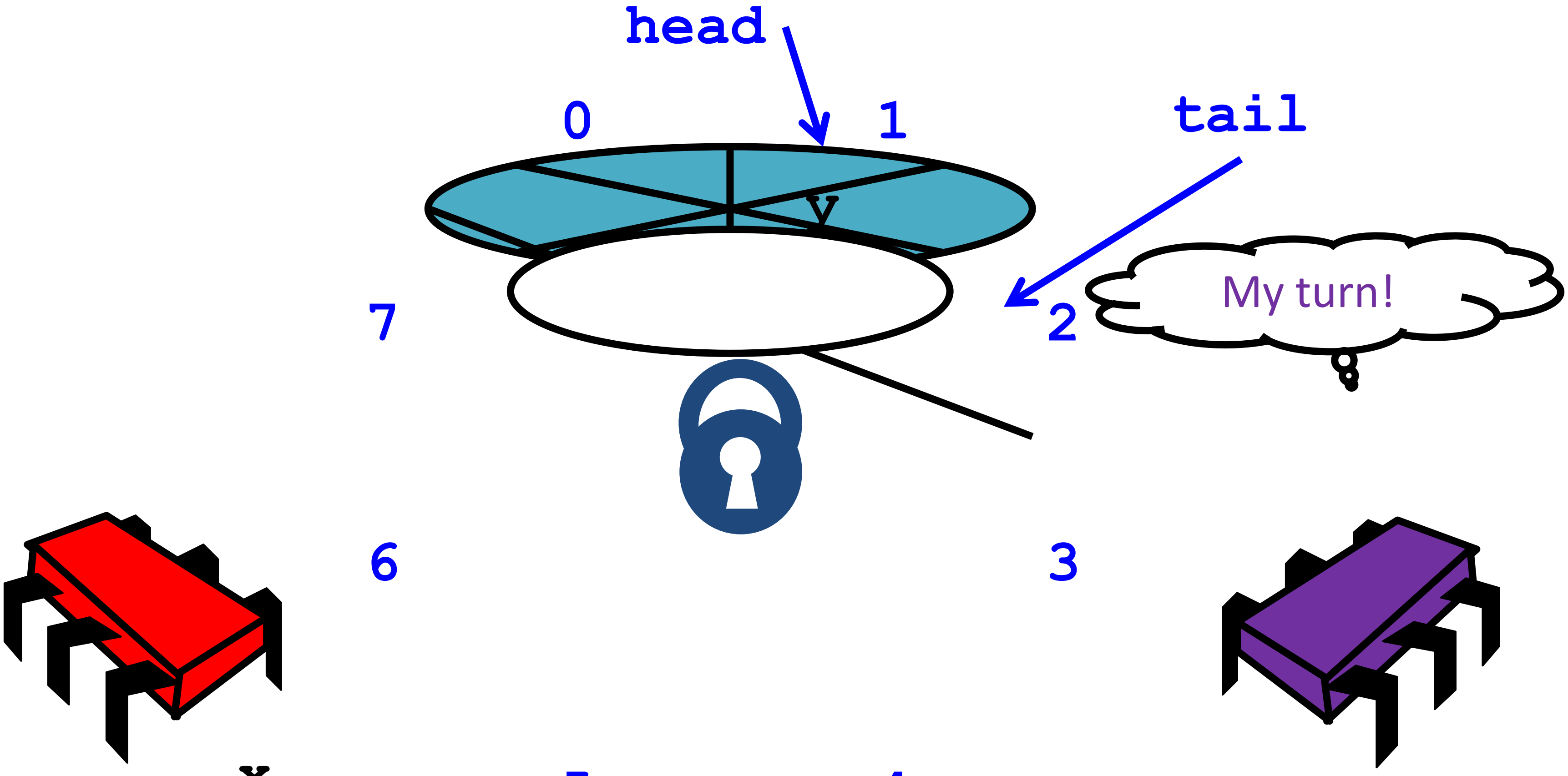
Implementation: deq()

```
int deq(queue_t q, void **elem) {
  int res;
  pthread_mutex_lock(&q->lock);
  if (q->tail == q->head) res = 0;
  else {
    *elem = q->items[q->head % CAPACITY];
    q->head++;
    res = 1;
  }
  pthread_mutex_unlock(&q->lock);
  return res;
}
```



Queue not empty?
Remove item and update head

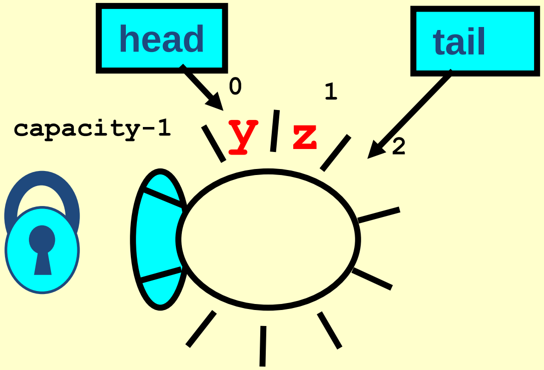
Release the Lock



Implementation: deq()

```
int deq(queue_t q, void **elem) {
  int res;
  pthread_mutex_lock(&q->lock);
  if (q->tail == q->head) res = 0;
  else {
    *elem = q->items[q->head % CAPACITY];
    q->head++;
    res = 1;
  }
  pthread_mutex_unlock(&q->lock);
  return res;
}
```

Release lock no matter what!



Implementation: deq()

```
int deq(queue_t q, void **elem) {
  int res;
  pthread_mutex_lock(&q->lock);
  if (q->tail == q->head) res = 0;
  else {
    *elem = q->items[q->head % CAPACITY];
    q->head++;
    res = 1;
  }
  pthread_mutex_unlock(&q->lock);
  return res;
}
```

Implementation: deq()

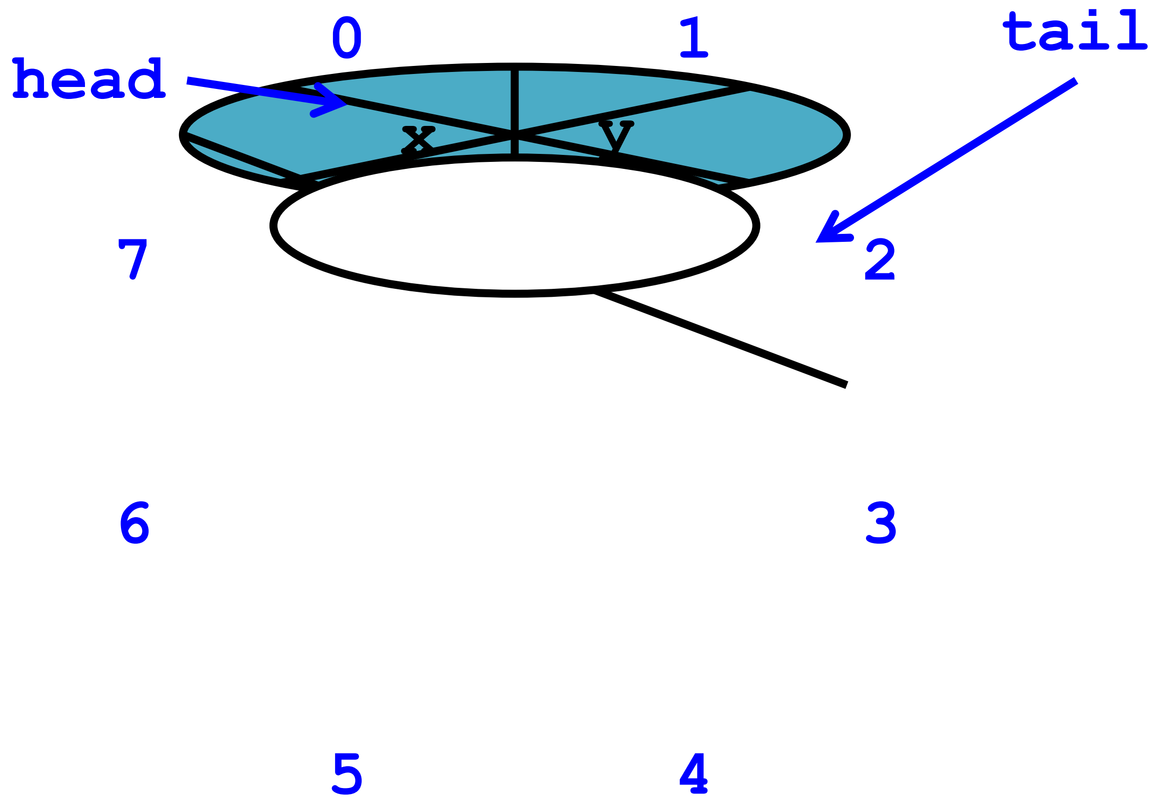
```
int deq(queue_t q, void **elem) {
  int res;
  pthread_mutex_lock(&q->lock);
  if (q->tail == q->head) res = 0;
  else {
    *elem = q->items[q->head % CAPACITY];
    q->head++;
    res = 1;
  }
  pthread_mutex_unlock(&q->lock);
  return res;
}
```

Should be correct because modifications are mutually exclusive…

Now consider the following implementation

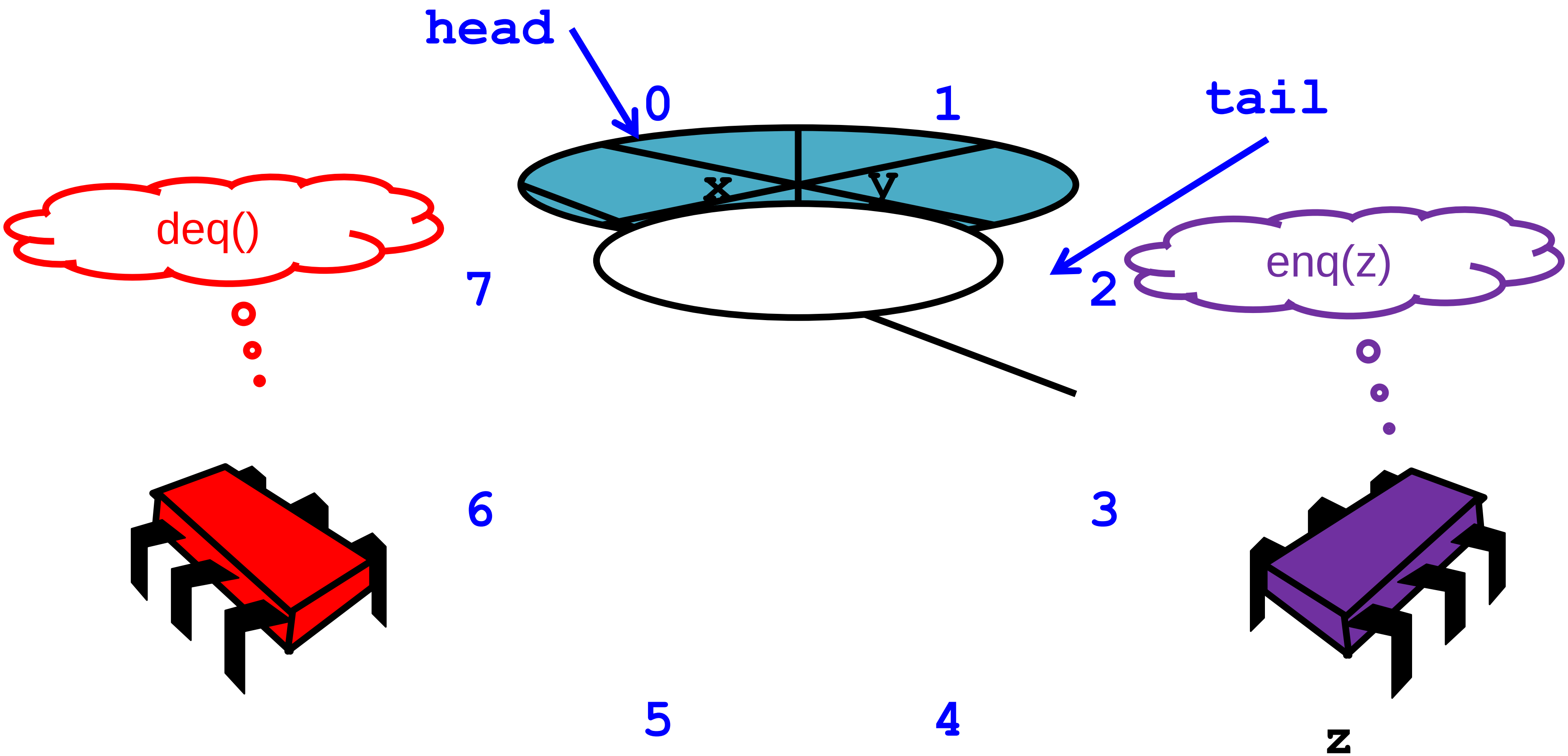
- The same thing without mutual exclusion
- For simplicity, only two threads
  - One thread enq only
  - The other deq only

Wait-free 2-Thread Queue

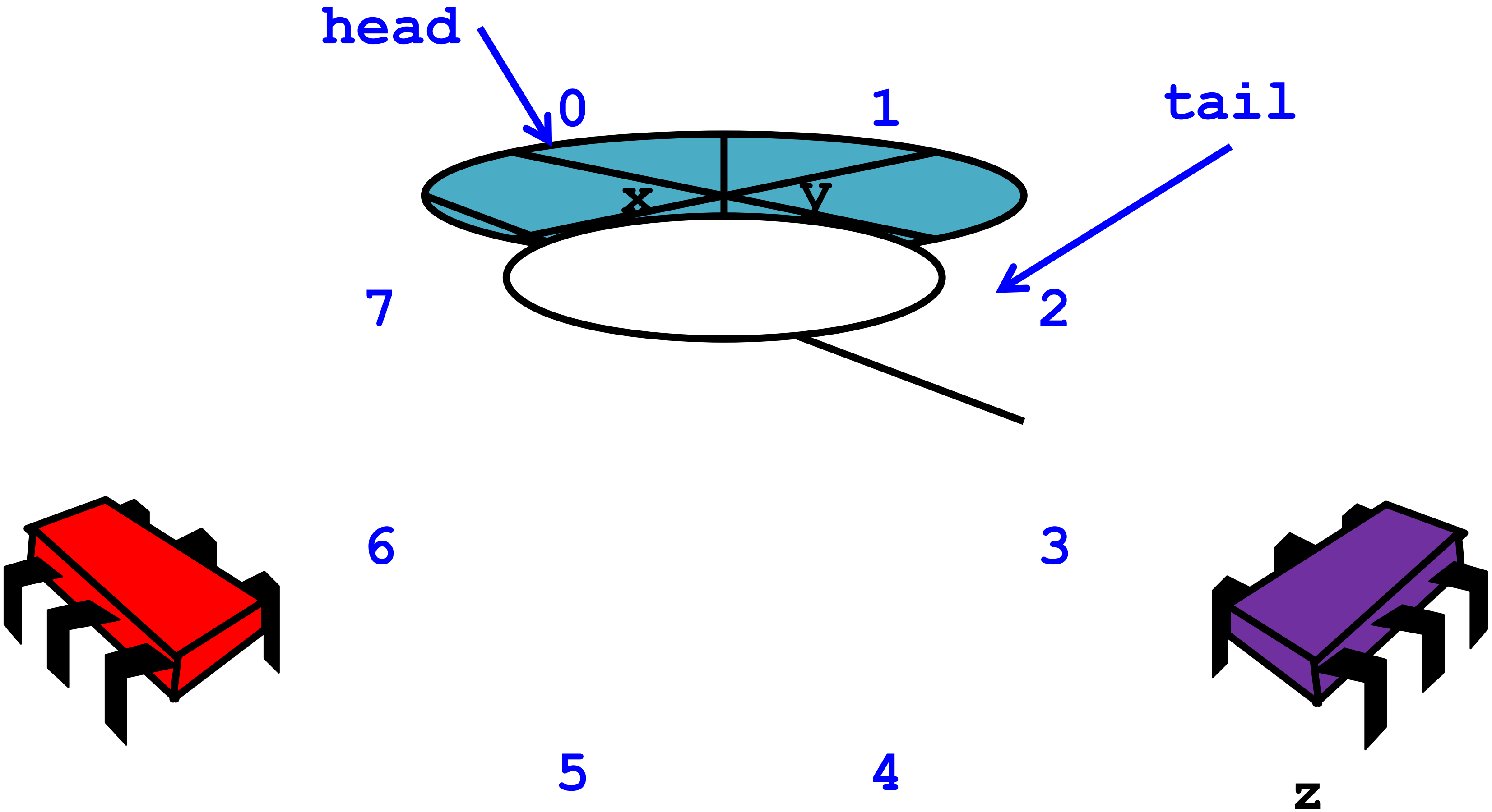


CAPACITY = 8

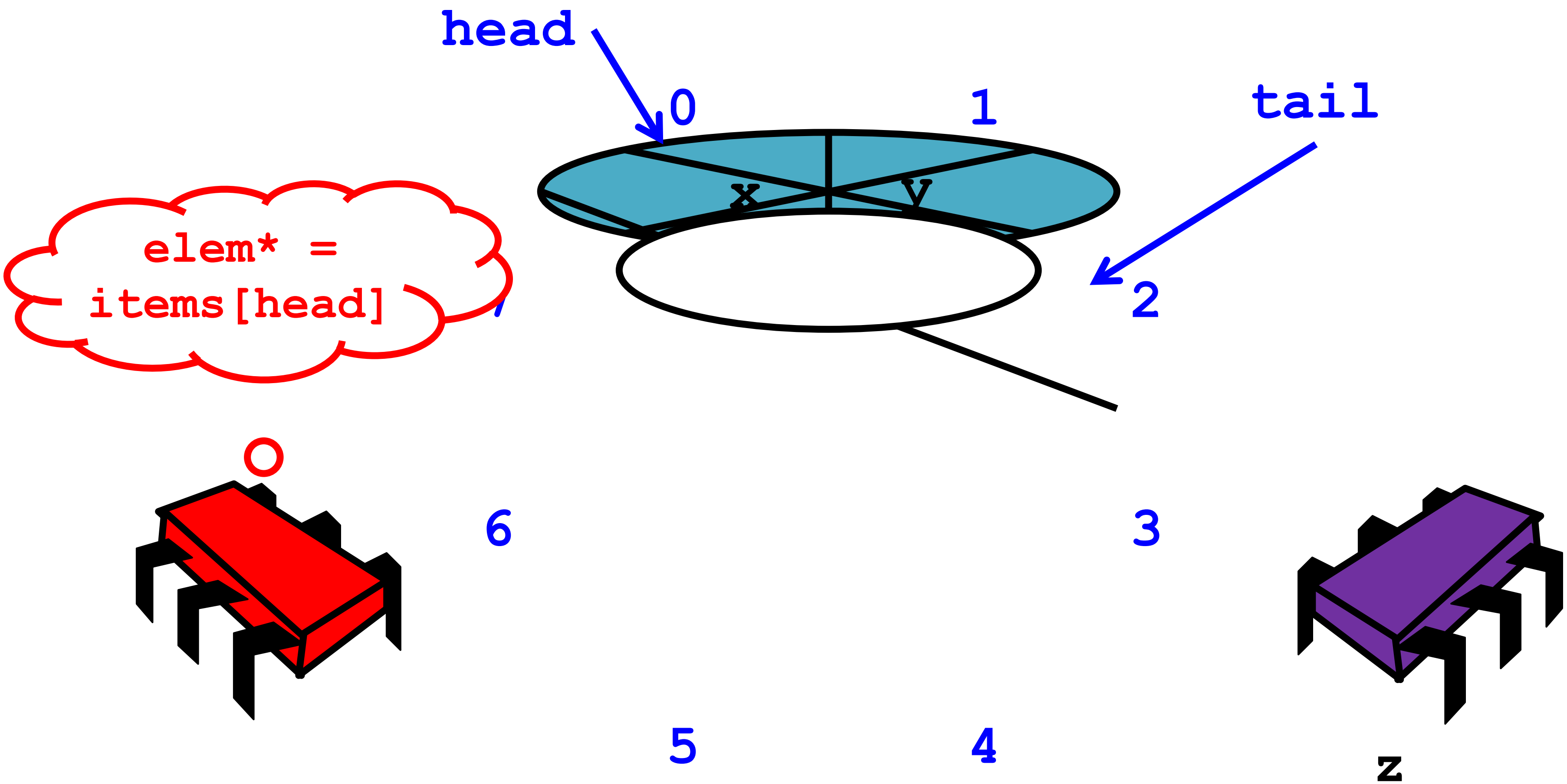
Wait-free 2-Thread Queue



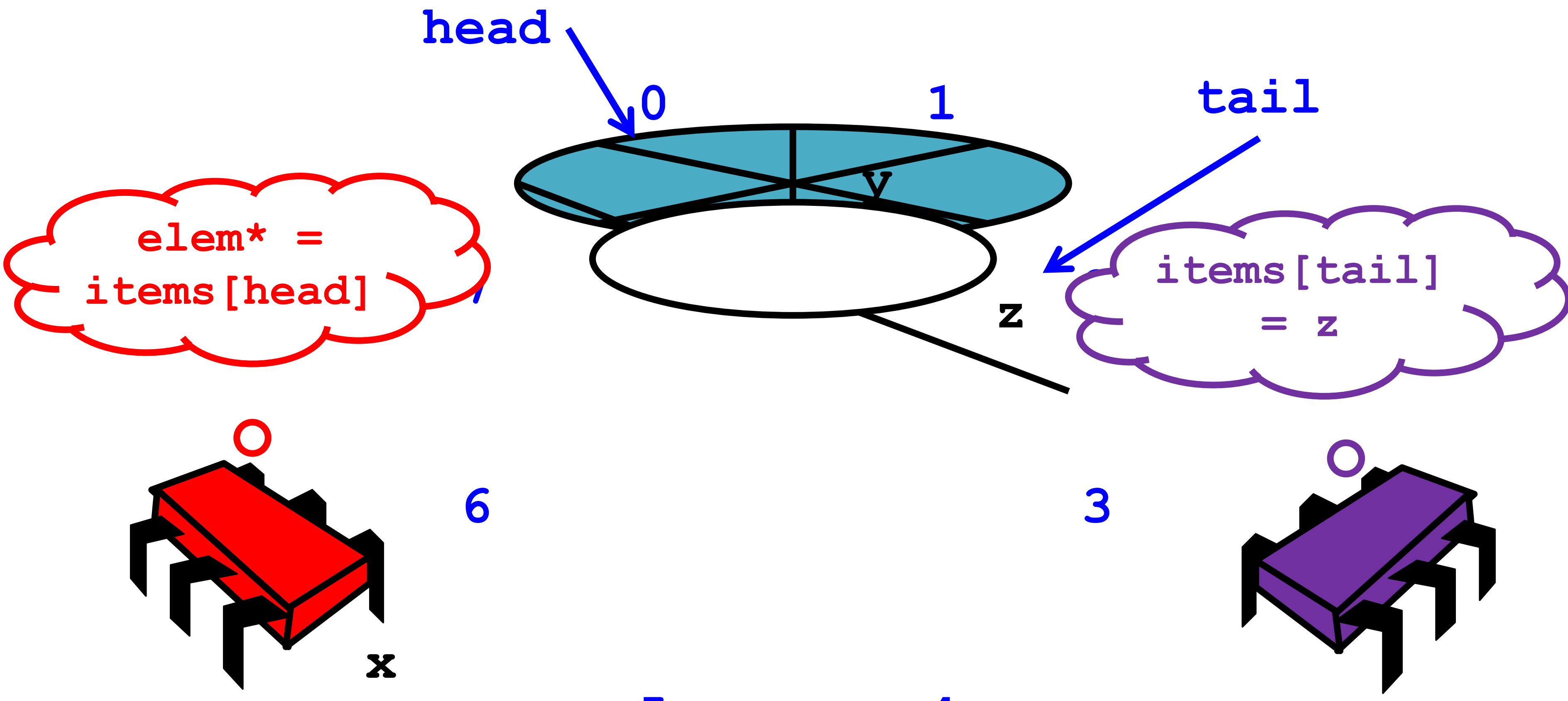
Wait-free 2-Thread Queue



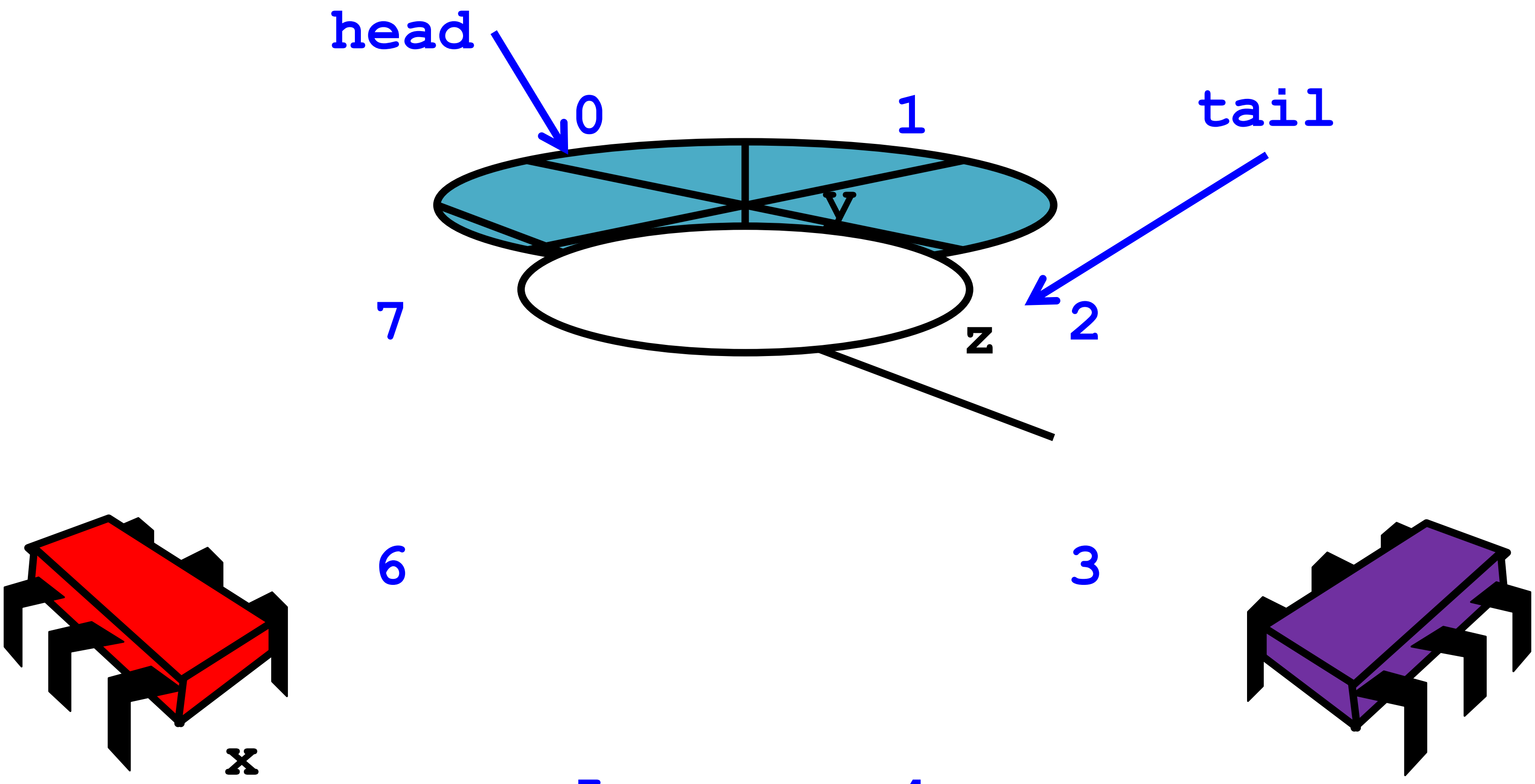
Wait-free 2-Thread Queue



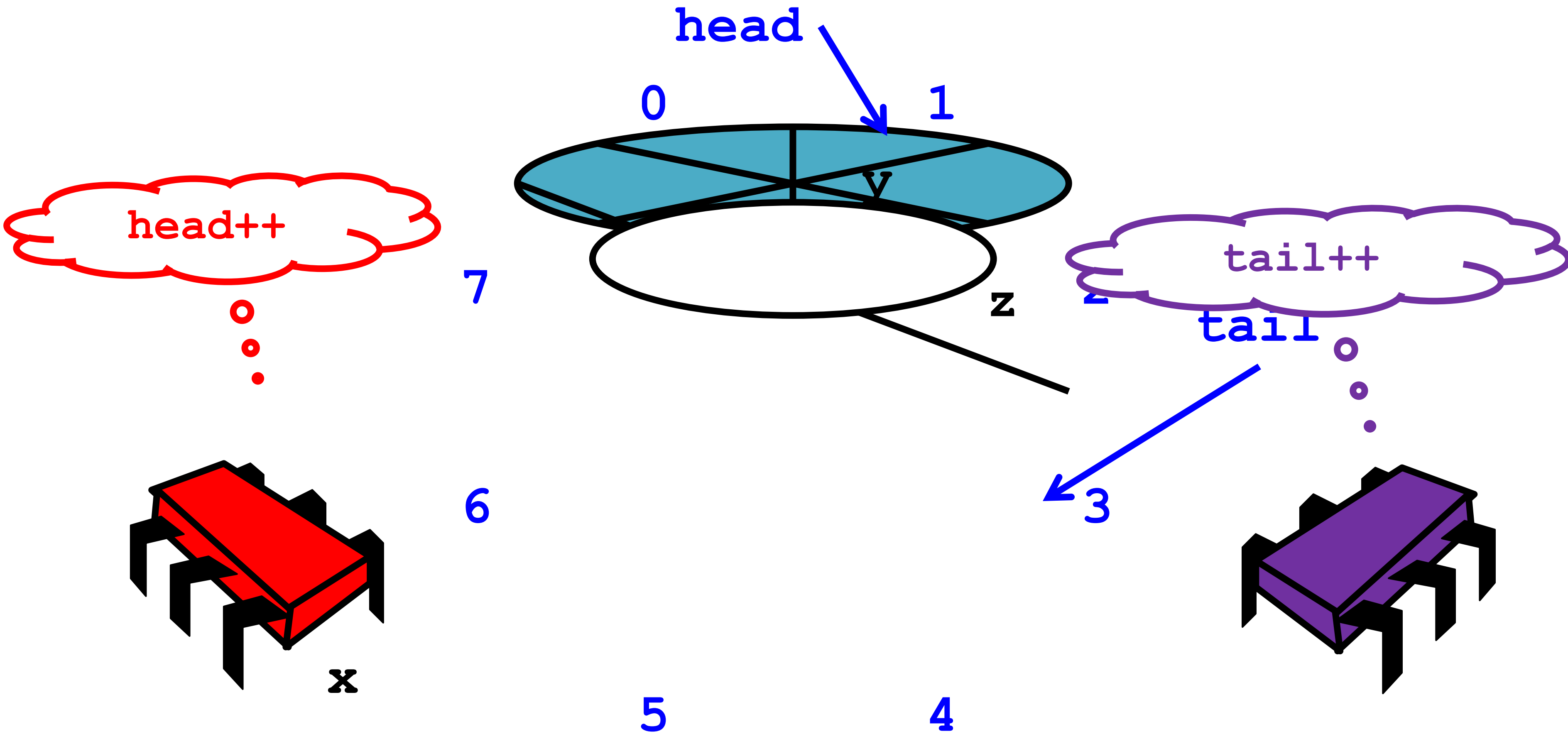
Wait-free 2-Thread Queue



Wait-free 2-Thread Queue



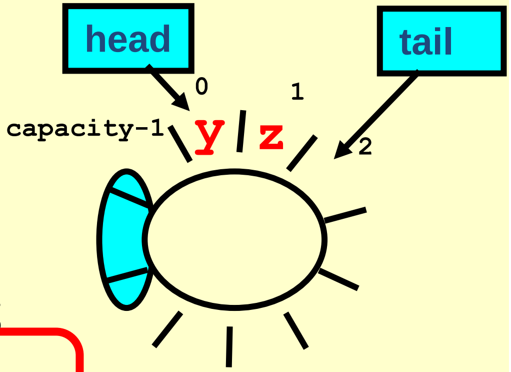
Wait-free 2-Thread Queue



Wait-free 2-Thread Queue

```
int deq(queue_t q, void **elem) {
  if (q->tail == q->head) return = 0;
  *elem = q->items[q->head % CAPACITY];
  q->head++;
  return 1;
}

int enq(queue_t q, void *x) {
  if (tail-head == CAPACITY) return 0;
  q->items[q->tail % CAPACITY] = x;
  q->tail++;
  return 1;
}
```



No lock needed !

Wait-free 2-Thread Queue

```
int deq(queue_t q, void **elem) {
  if (q->tail == q->head) return = 0;
  *elem = q->items[q->head % CAPACITY];
  q->head++;
  return 1;
}

int enq(queue_t q, void *x) {
  if (tail-head == CAPACITY) return 0;
  q->items[q->tail % CAPACITY] = x;
  q->tail++;
  return 1;
}
```

Wait-free 2-Thread Queue

```
int deq(queue_t q, void **elem) {
  if (q->tail == q->head) return = 0;
  *elem = q->items[q->head % CAPACITY];
  q->head++;
  return 1;
}

int enq(queue_t q, void *x) {
  if (tail-head == CAPACITY) return 0;
  q->items[q->tail % CAPACITY] =
  q->tail++;
  return 1;
}
```

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
- Let's talk about object specifications …

Correctness and Progress

- In a concurrent setting, we need to specify both the safety and the liveness properties of an object
- Need a way to define
  - when an implementation is correct
  - the conditions under which it guarantees progress

Correctness and Progress

- In a concurrent setting, we need to specify both the safety and the liveness properties of an object
- Need a way to define
  - when an implementation is correct
  - the conditions under which it guarantees progress

Let's begin with correctness

Sequential Objects

- Each object has a ***state***
  - Usually given by a set of ***fields***
  - Queue example: `items`, `head`, `tail`
- 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 object will be in some other state
  - and the method will return a particular value

Pre and Postconditions for Dequeue

- Precondition:
  – Queue is non-empty
- Postcondition:
  – Returns 1
- Postcondition:
  – Removes first item in queue

Pre and Postconditions for Dequeue

- Precondition:
  - Queue is empty
- Postcondition:
  - Returns 0
- Postcondition:
  - Queue state unchanged

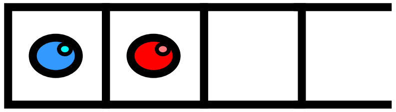
Why Sequential Specifications Totally Rock

- Interactions among methods captured by side-effects on object state
  - State meaningful between method calls
- Documentation size linear in number of methods
  - Each method described in isolation
- Can add new methods
  - Without changing descriptions of old methods

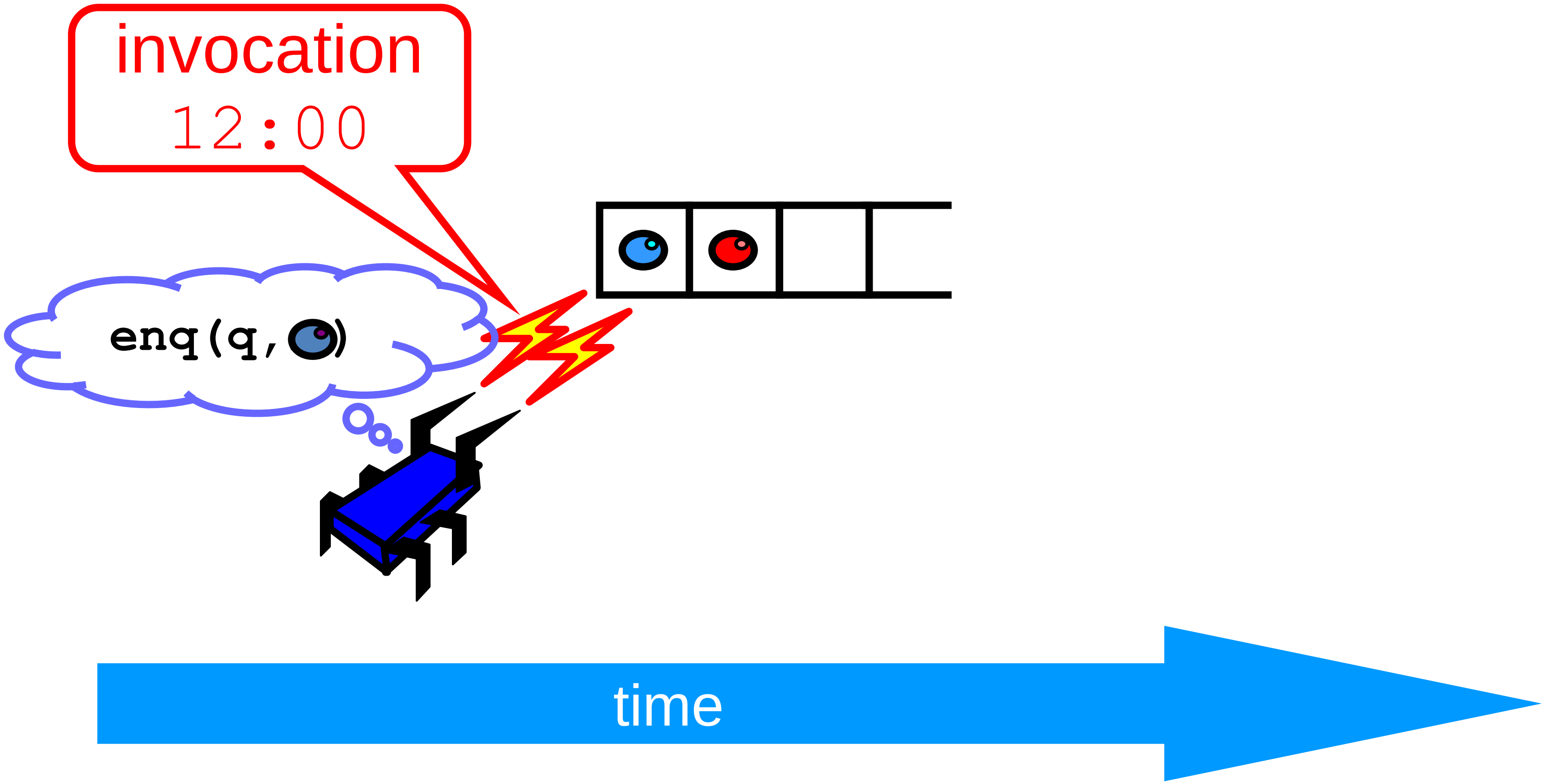
What About Concurrent Specifications ?

- Methods?
- Documentation?
- Adding new methods?

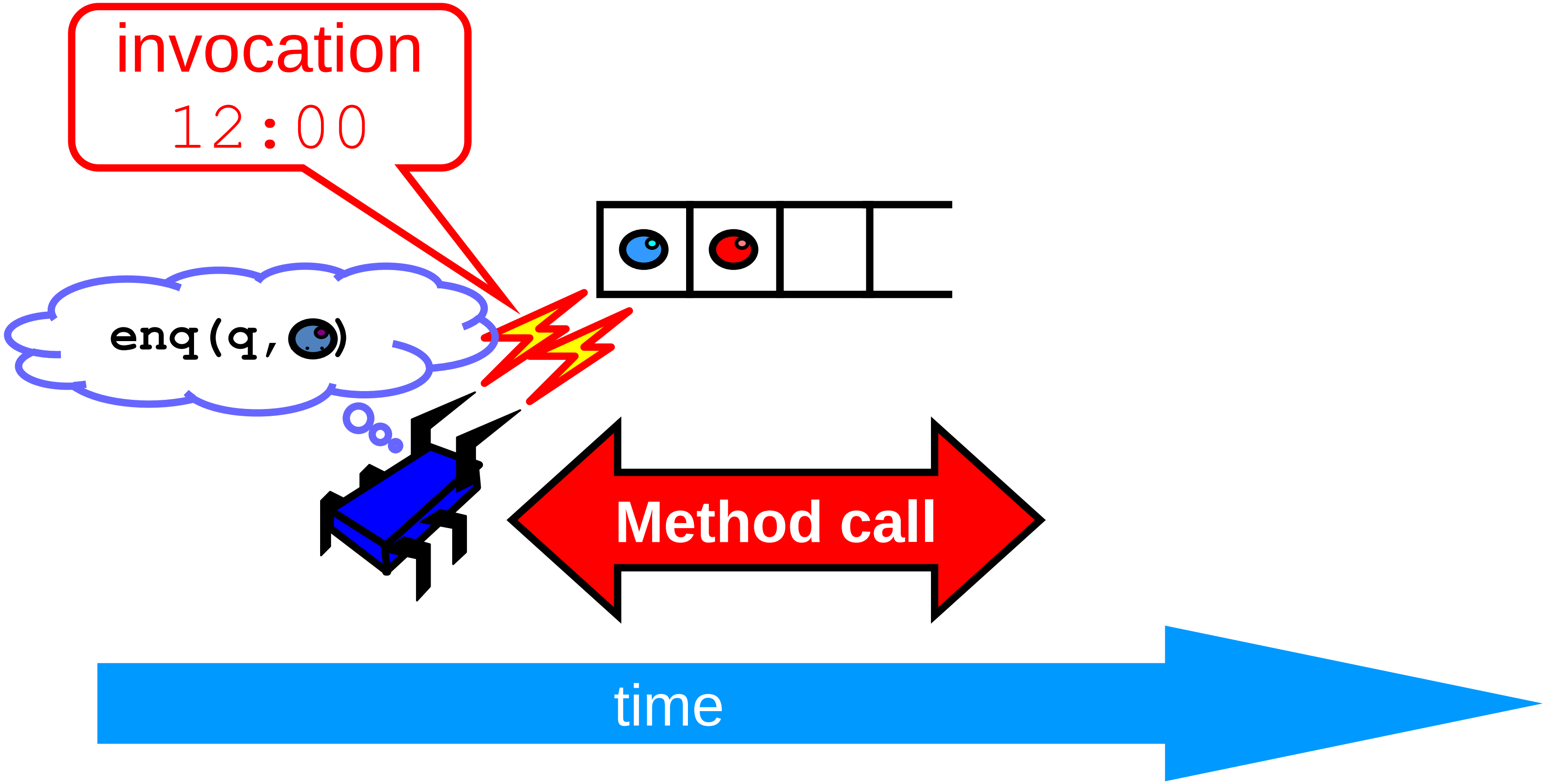
Methods Take Time



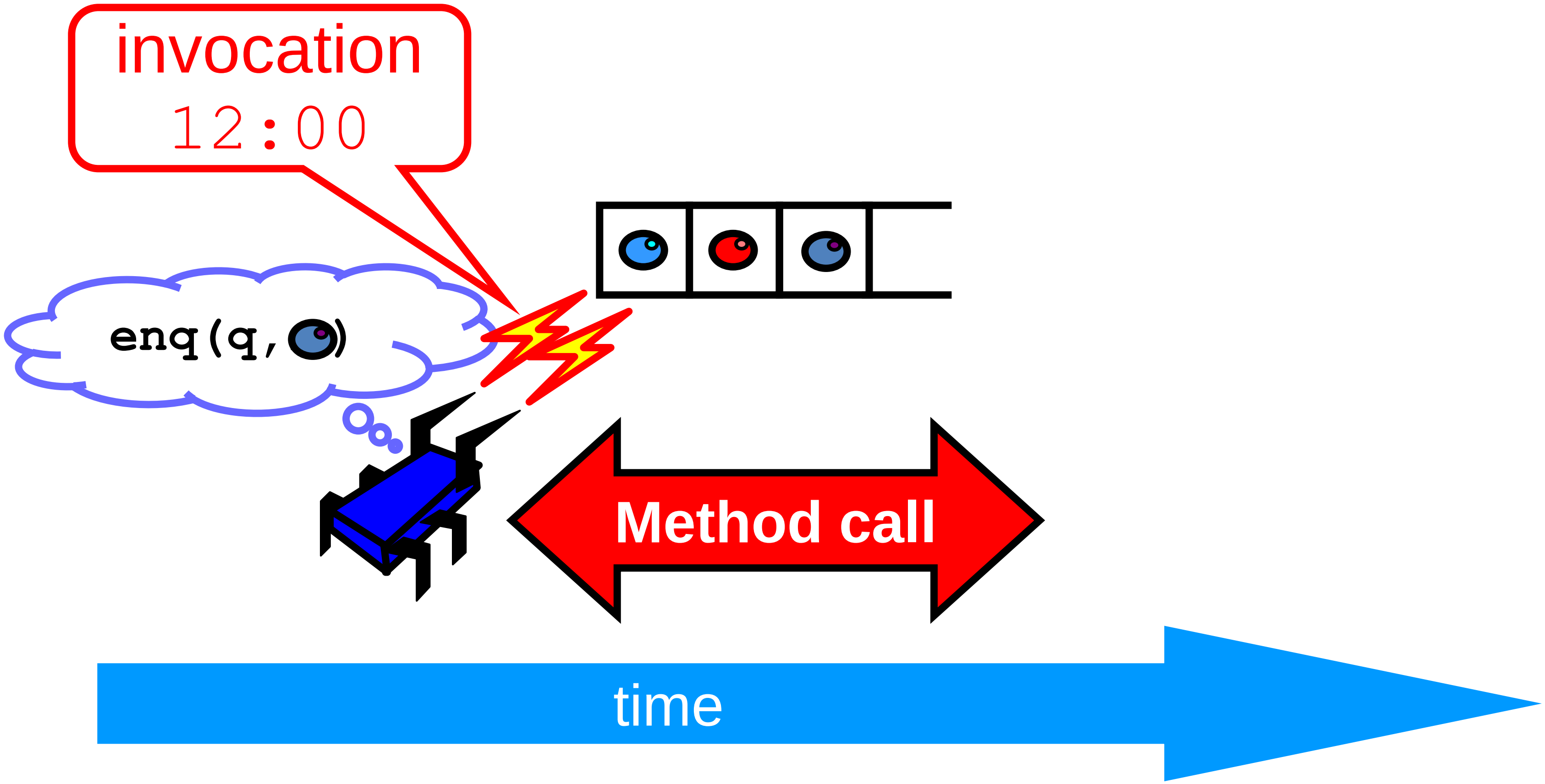
Methods Take Time



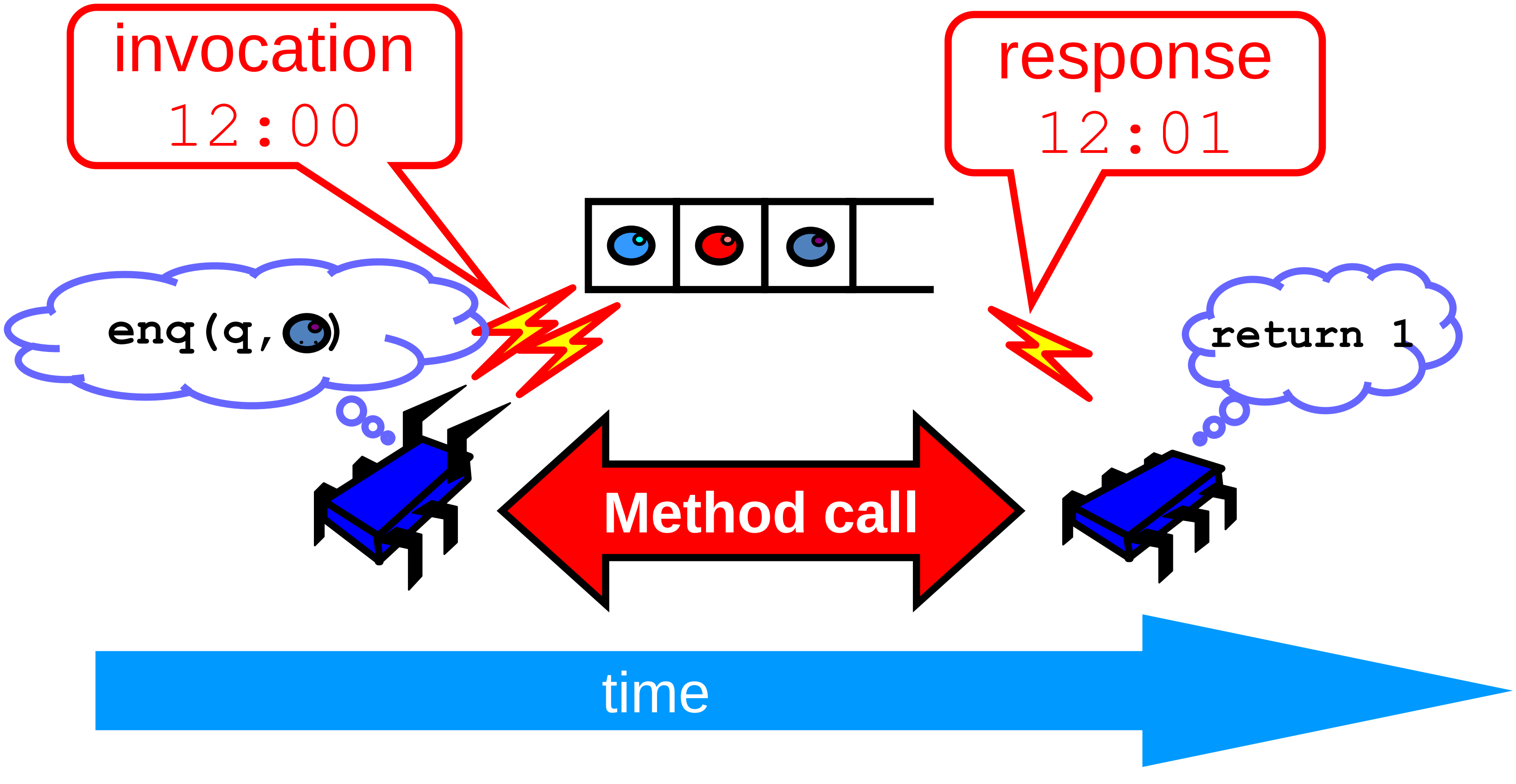
Methods Take Time



Methods Take Time



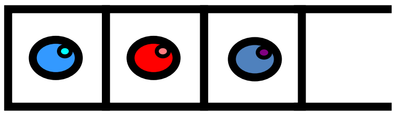
Methods Take Time



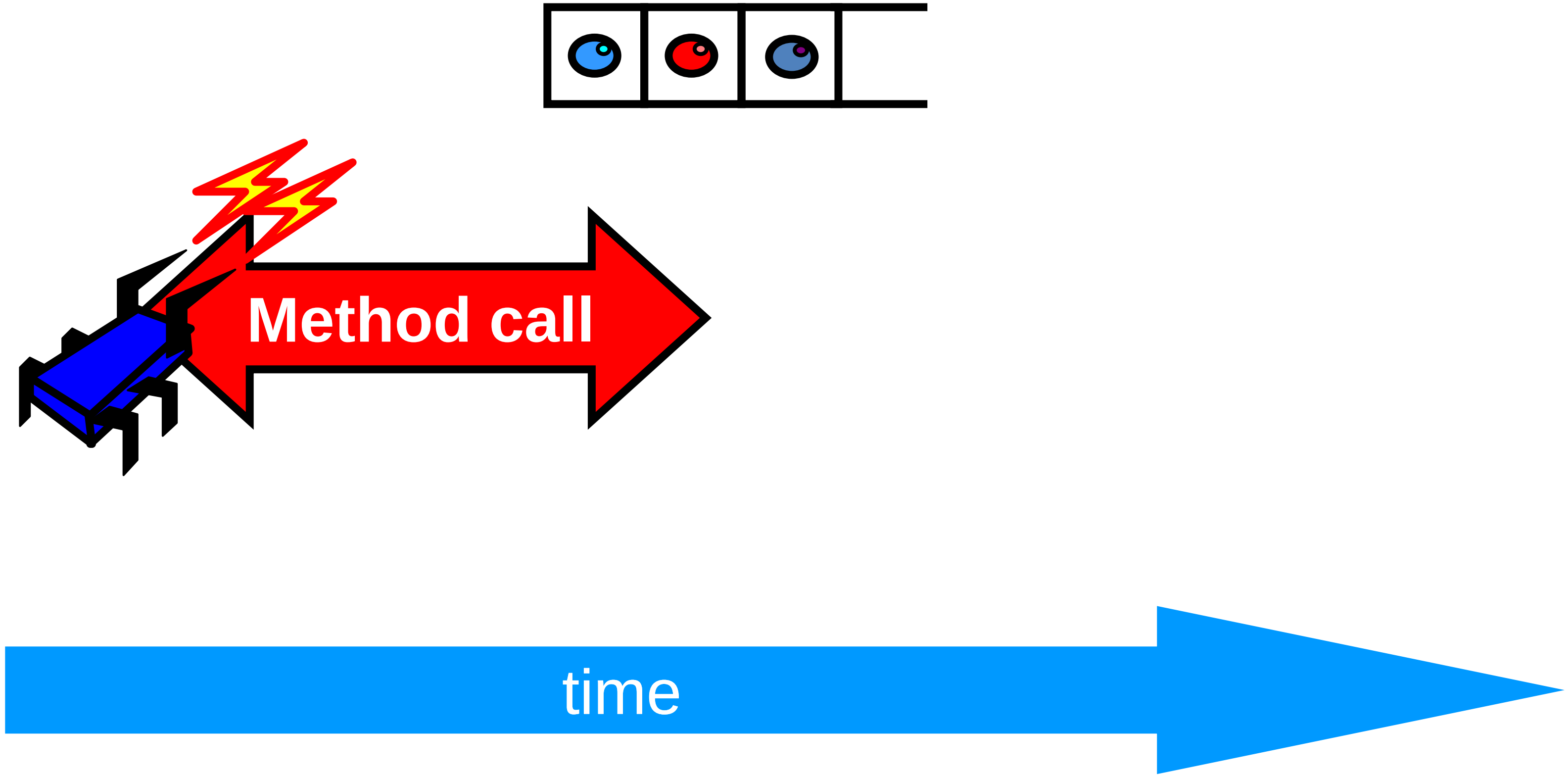
Sequential vs Concurrent

- Sequential
  - Methods take time? Who knew?
- Concurrent
  - Method call is not an event
  - Method call is an interval.

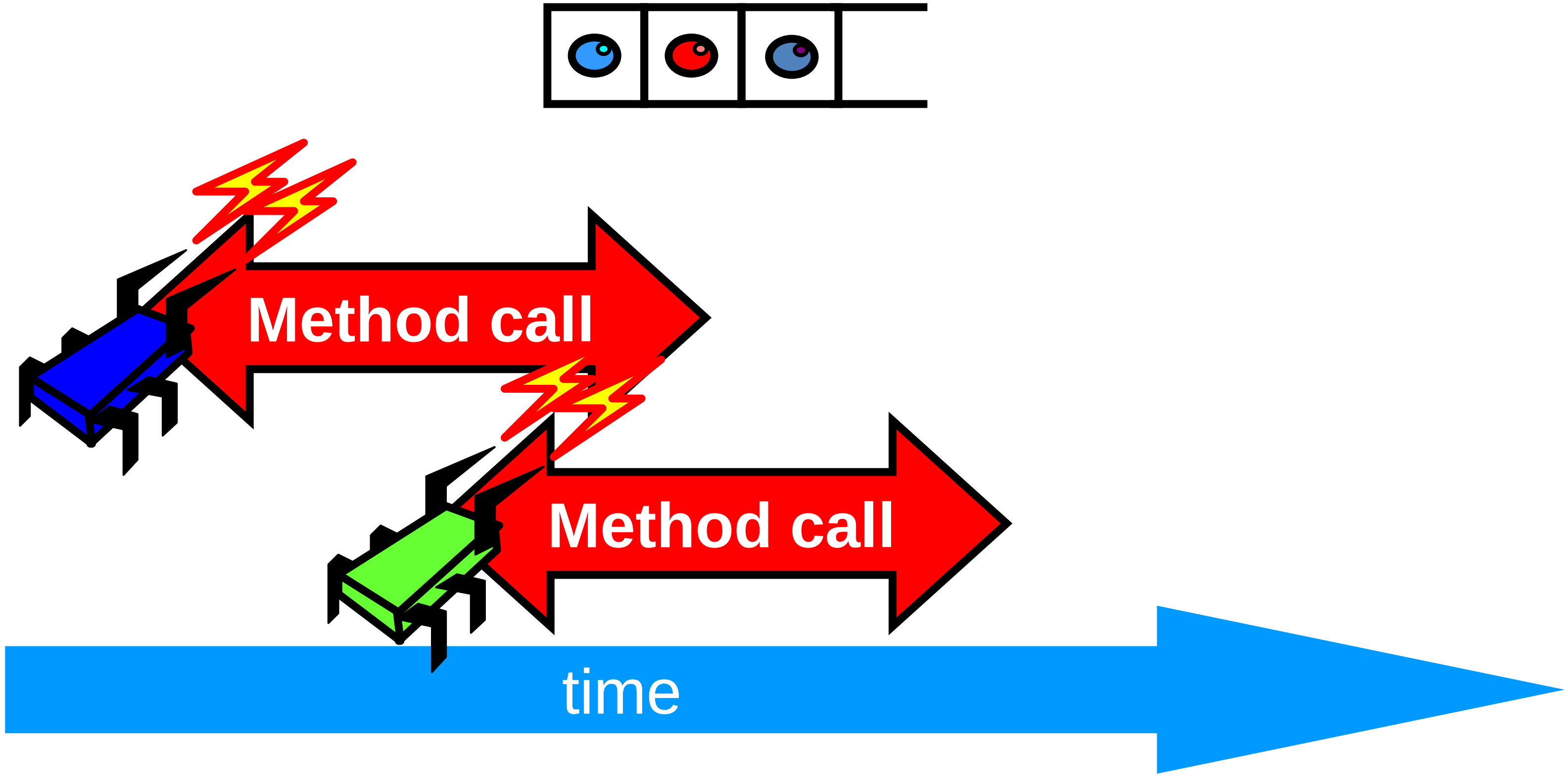
Concurrent Methods Take Overlapping Time



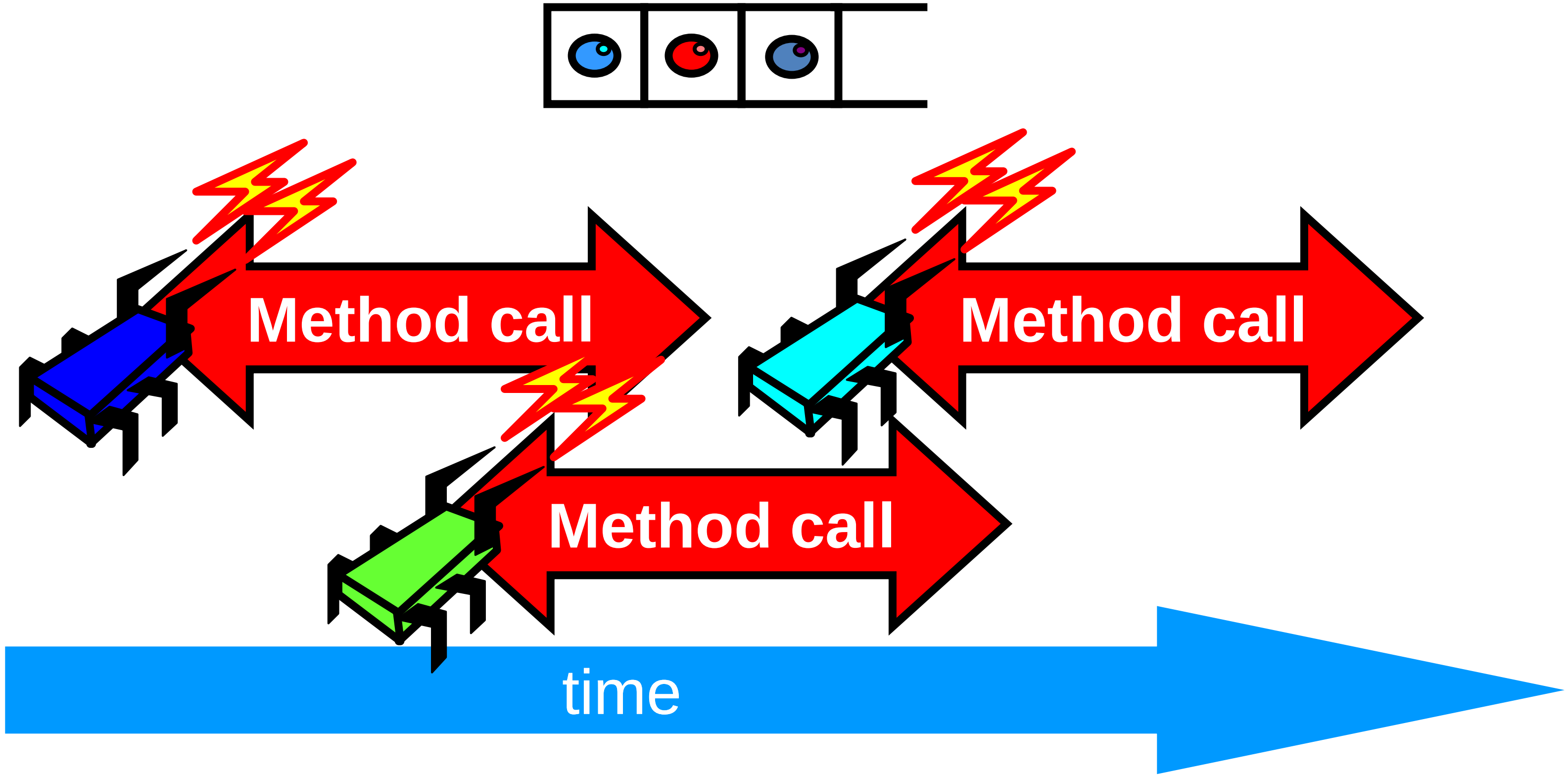
Concurrent Methods Take Overlapping Time



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 enqs overlap?
    - Two deqs? enq and deq? …

Sequential vs Concurrent

- Sequential:
  - Can add new methods without affecting older methods
- Concurrent:
  - Everything can potentially interact with everything else

Sequential vs Concurrent

- Sequential:
  - Can add new methods without affecting older methods
- Concurrent:
  - Everything can potentially interact with everything else

Panic!

The Big Question

- What does it 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...

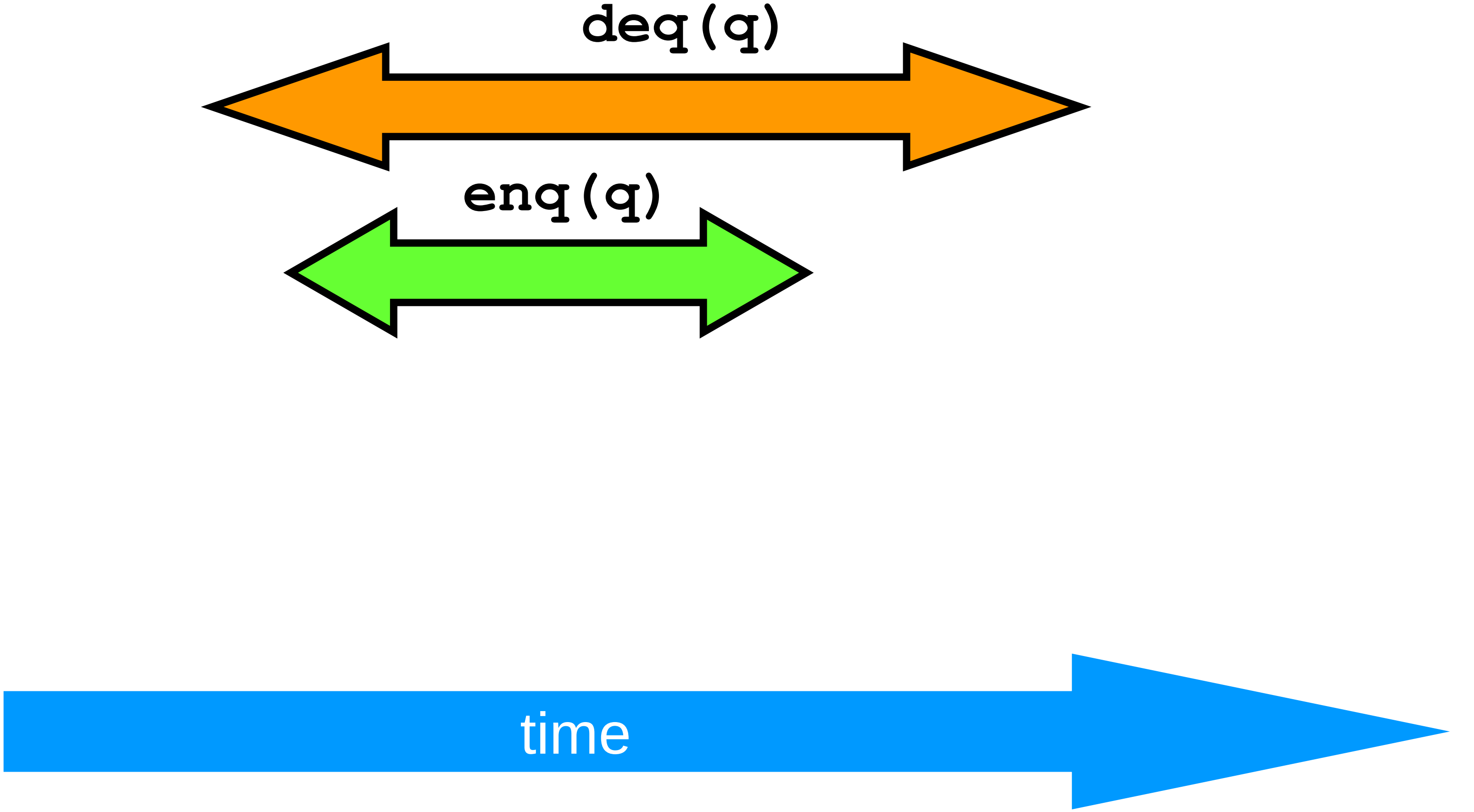
```
int deq(queue_t q, void **elem) {
  int res;
  pthread_mutex_lock(&q->lock);
  if (q->tail == q->head) res = 0;
  else {
    *elem = q->items[q->head % CAPACITY];
    q->head++;
    res = 1;
  }
  pthread_mutex_unlock(&q->lock);
  return res;
}
```

Intuitively...

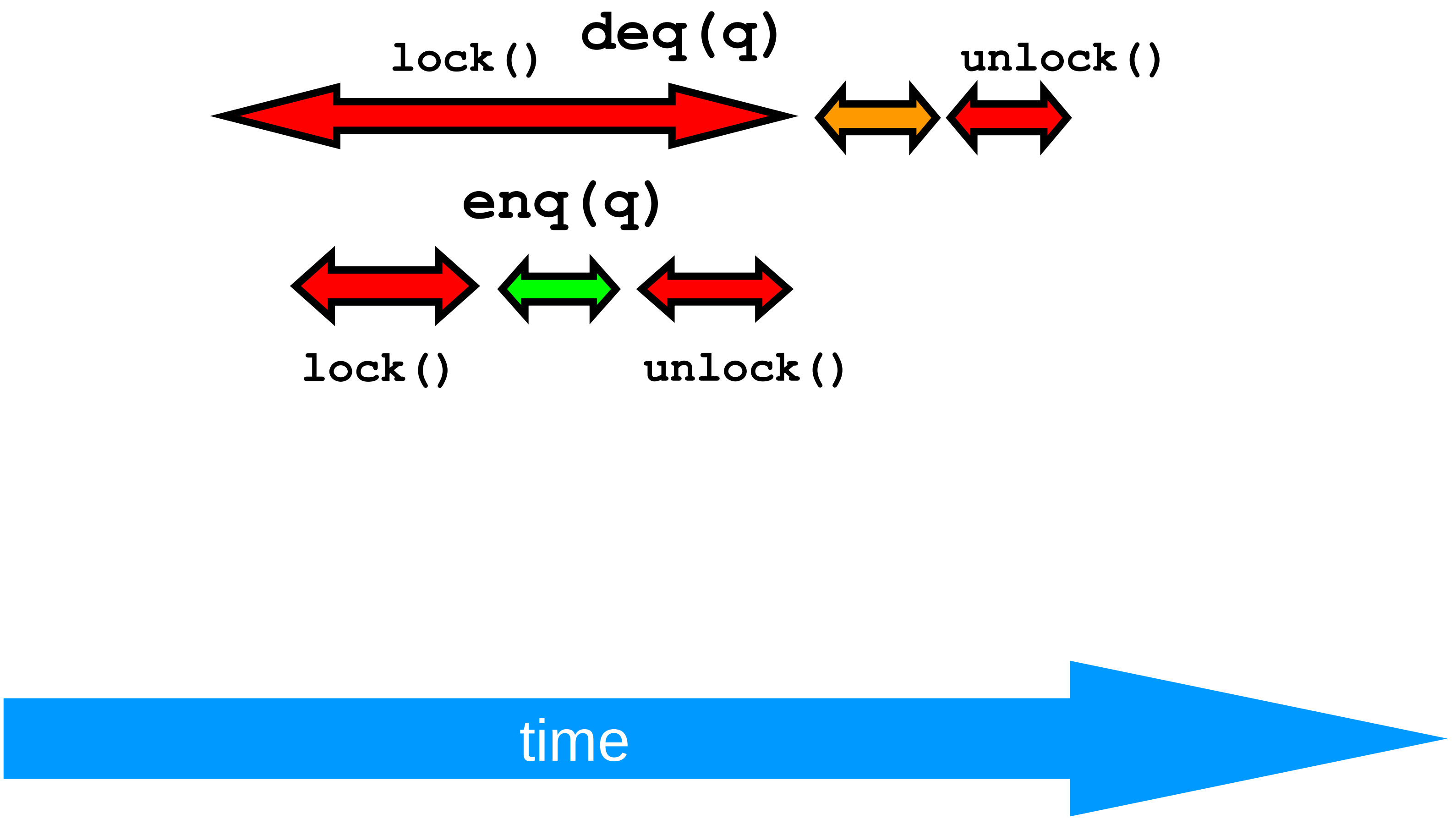
```
int deq(queue_t q, void **elem) {
  int res;
  pthread_mutex_lock(&q->lock);
  if (q->tail == q->head) res = 0;
  else {
    *elem = q->items[q->head % CAPACITY];
    q->head++;
    res = 1;
  }
  pthread_mutex_unlock(&q->lock);
  return res;
}
```

All queue modifications are mutually exclusive

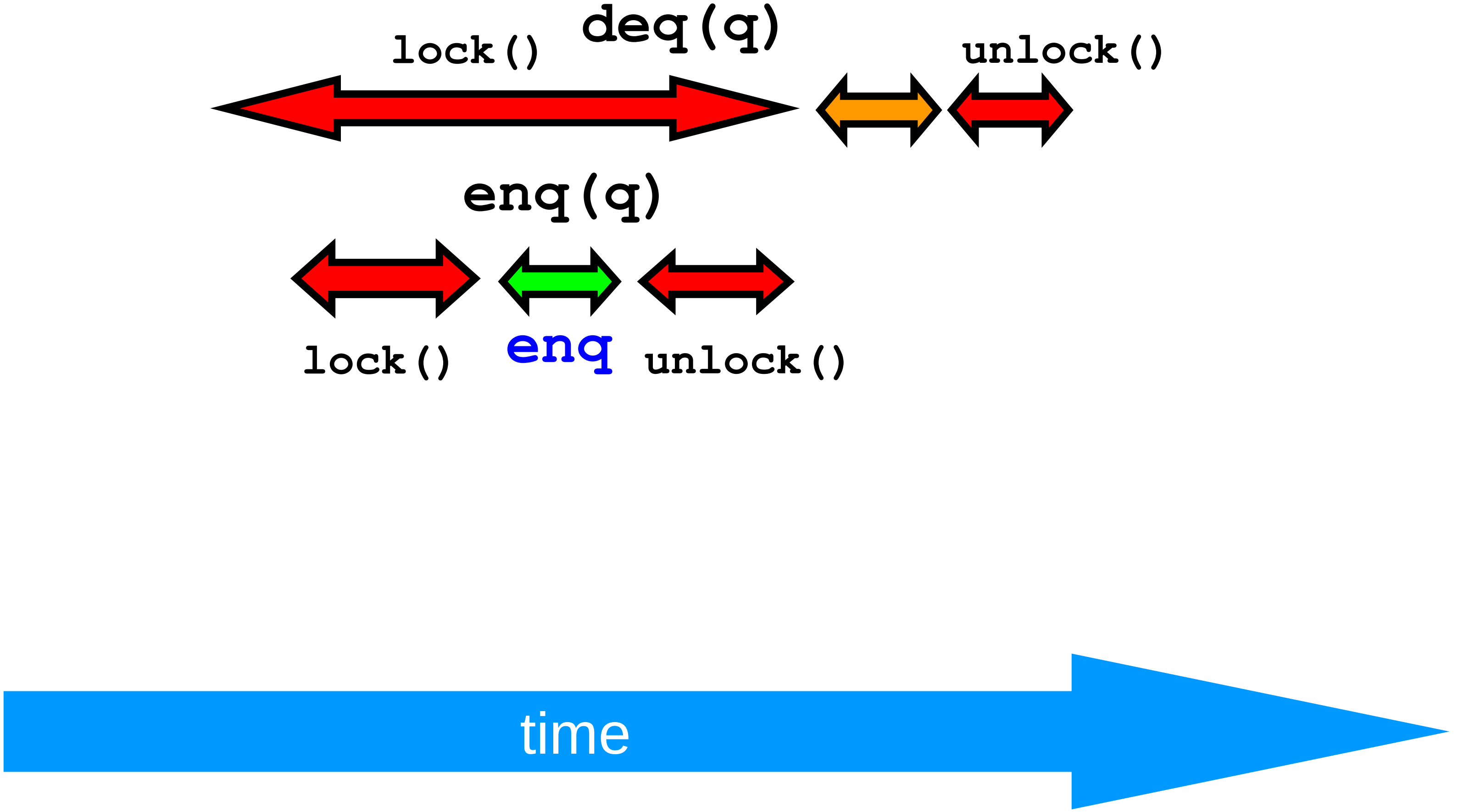
Intuitively



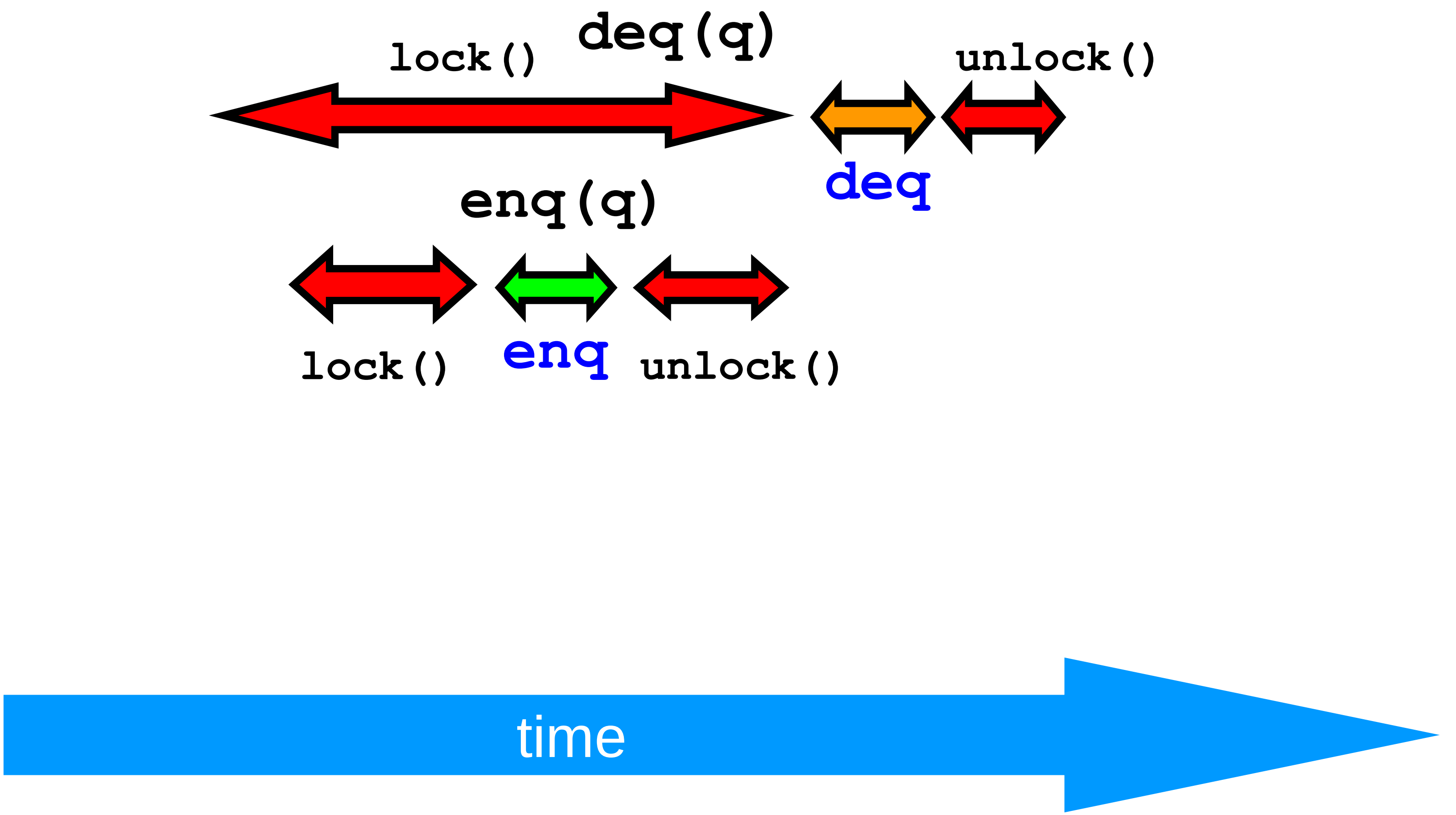
Intuitively



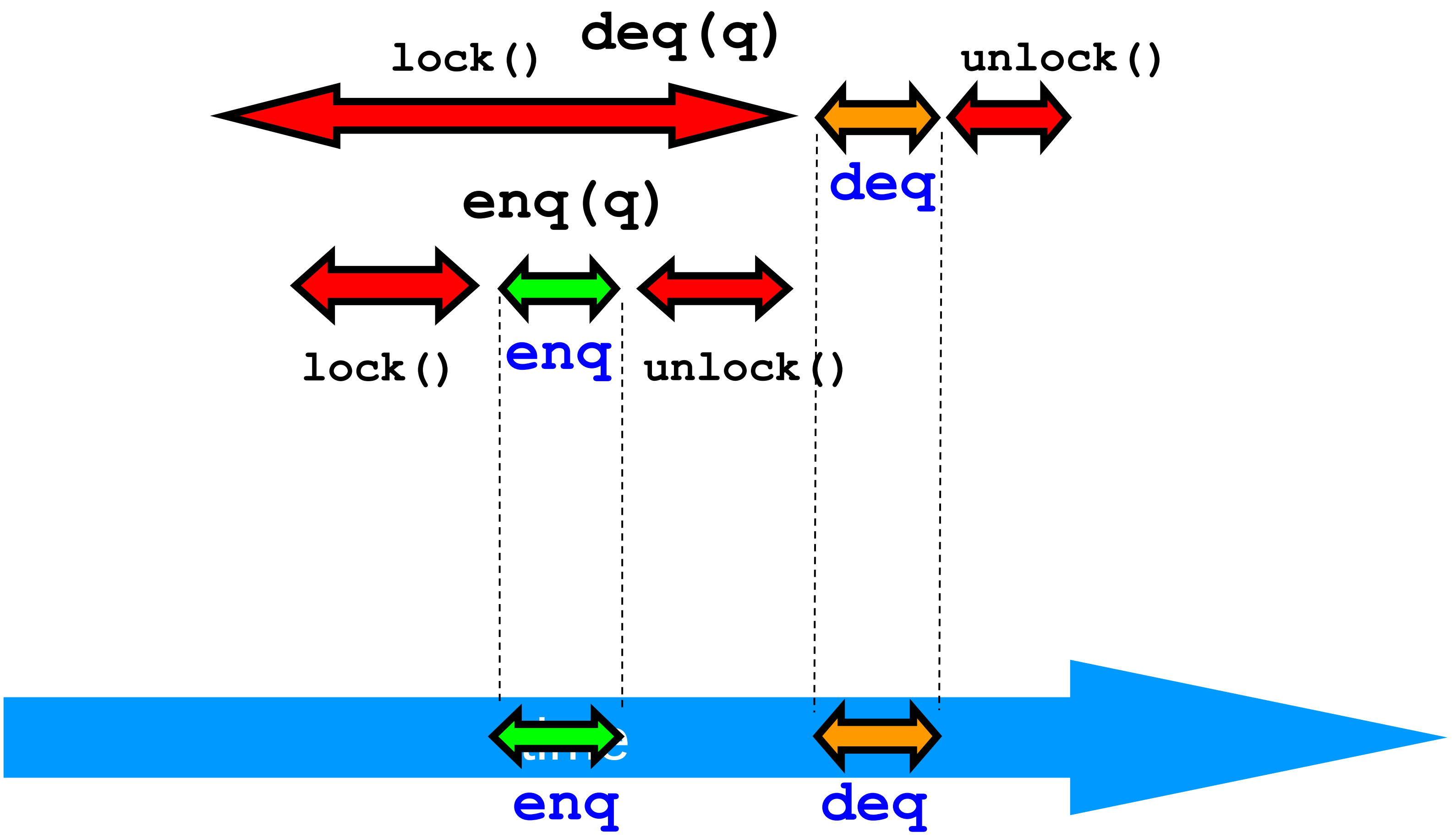
Intuitively



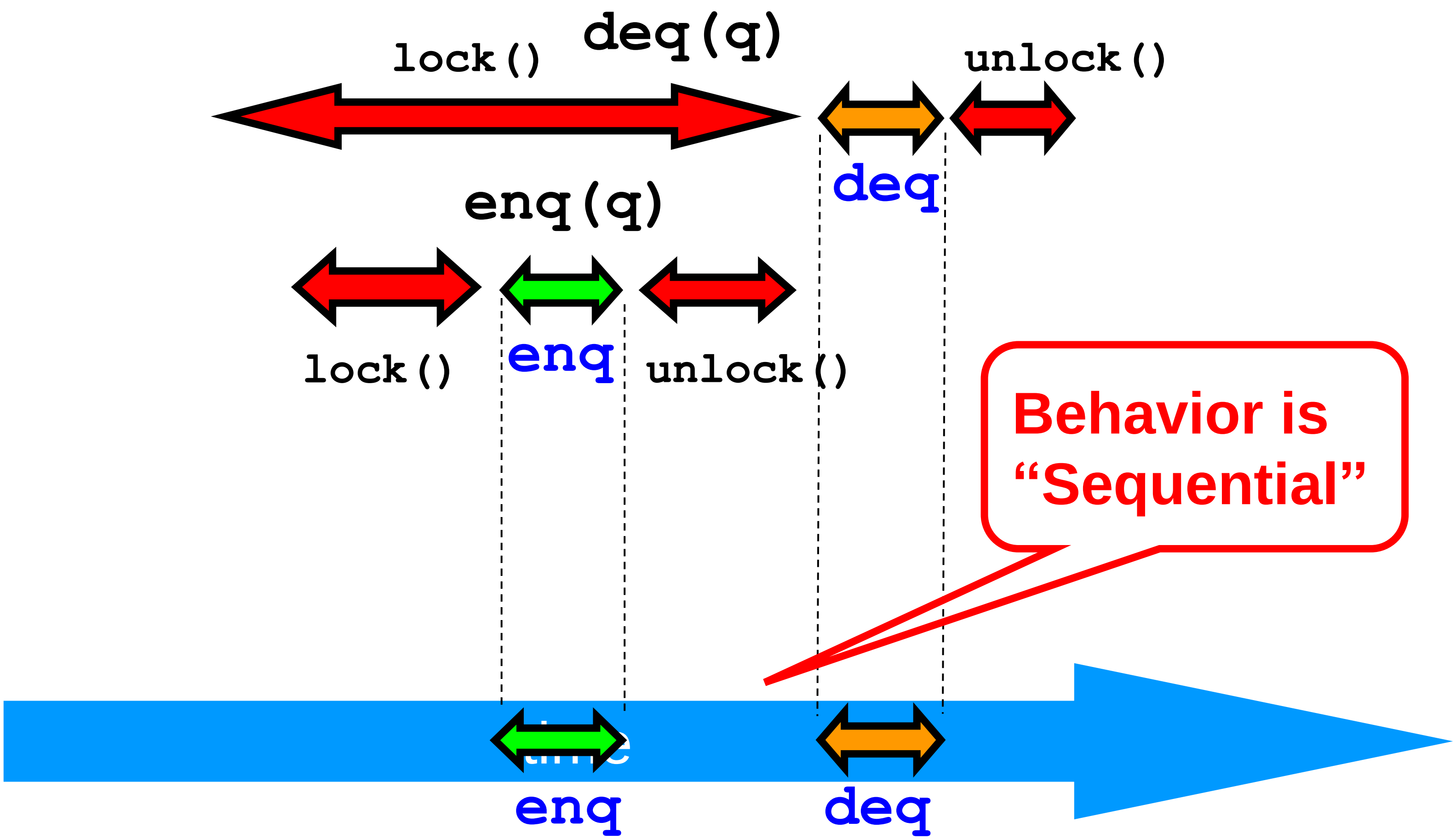
Intuitively



Intuitively

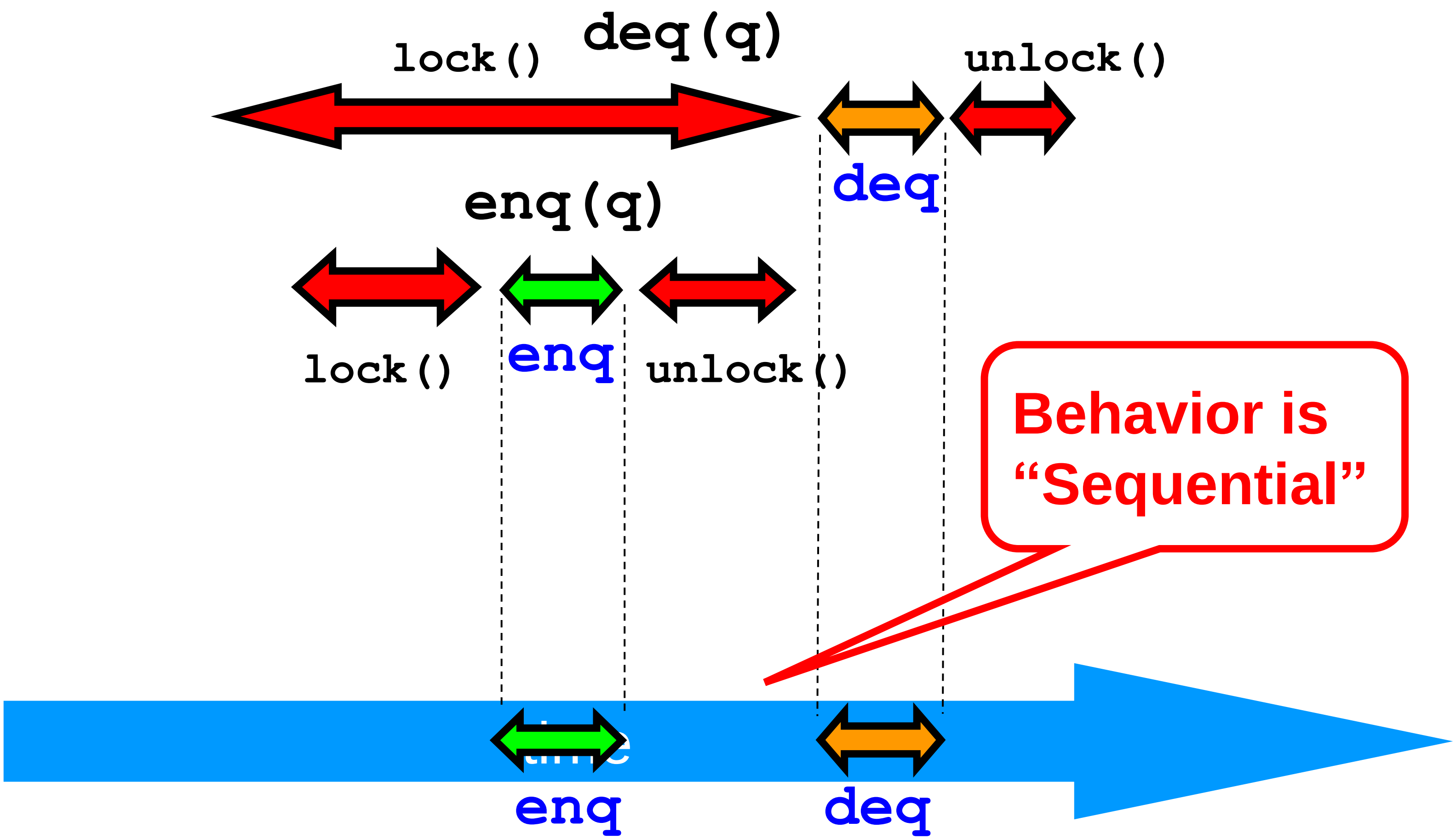


Intuitively



Intuitively

Lets capture the idea of describing the concurrent via the sequential



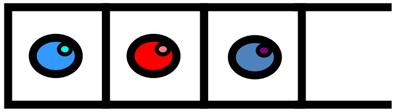
Linearizability

- Each method should
  - “take effect”
  - instantaneously
  - between invocation and response events
- Object is correct if this “sequential” behavior is correct
- Any such concurrent object is called
  - **Linearizable**

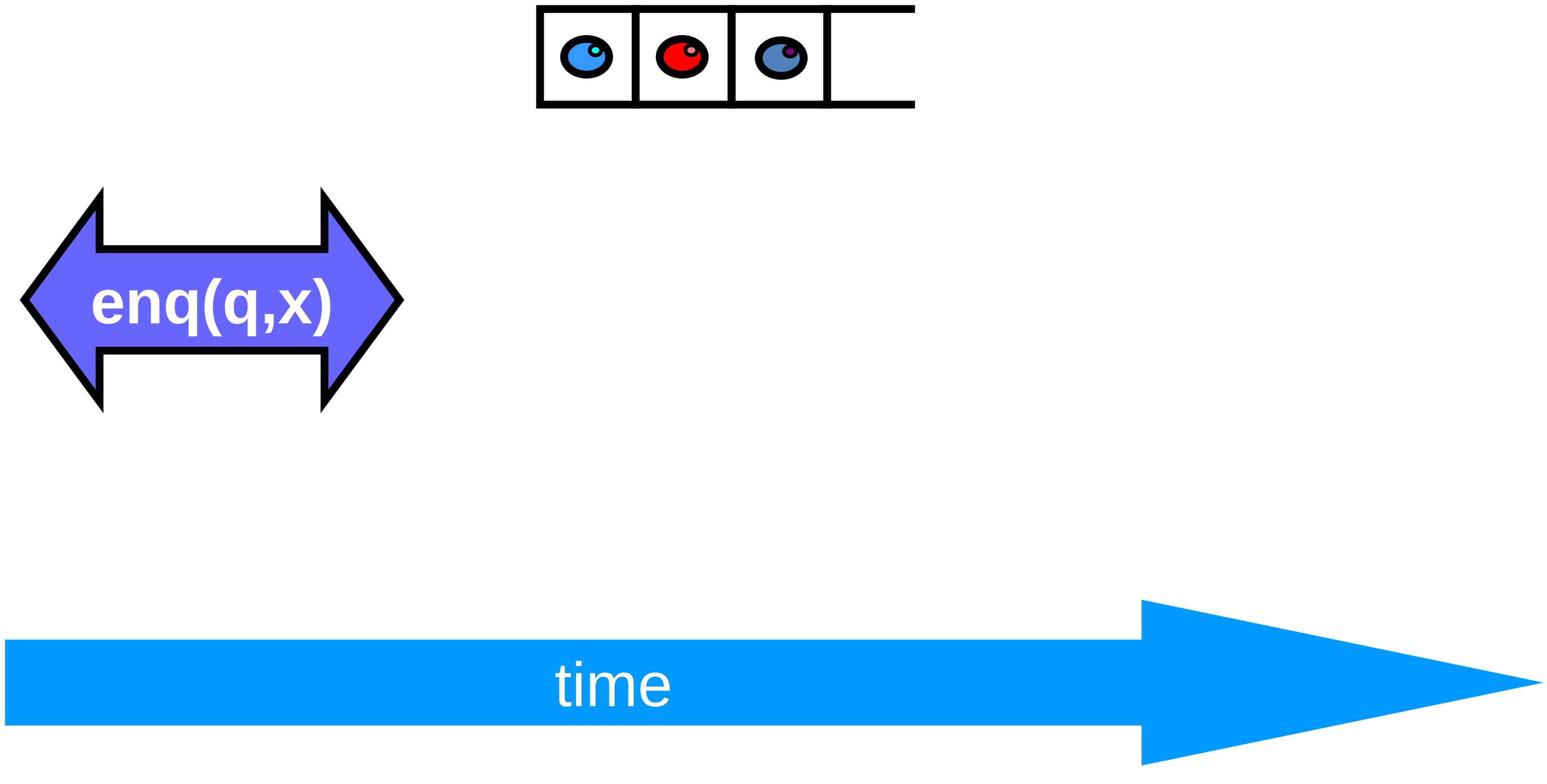
Is it really about the object?

- Each method should
  - “take effect”
  - instantaneously
  - between invocation and response events
- Sounds like a property of an execution…
- A linearizable object: one all of whose possible executions are linearizable

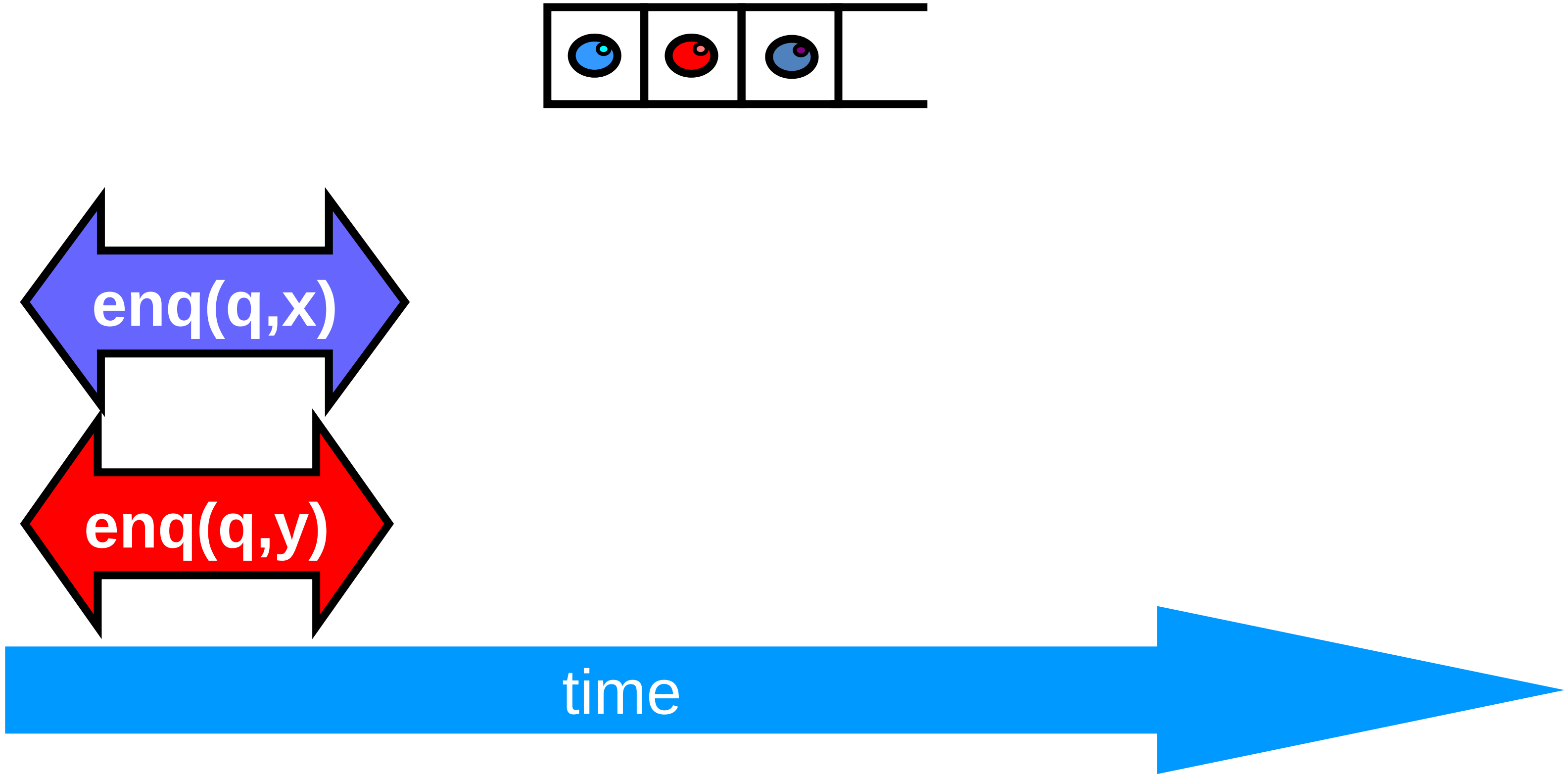
Example



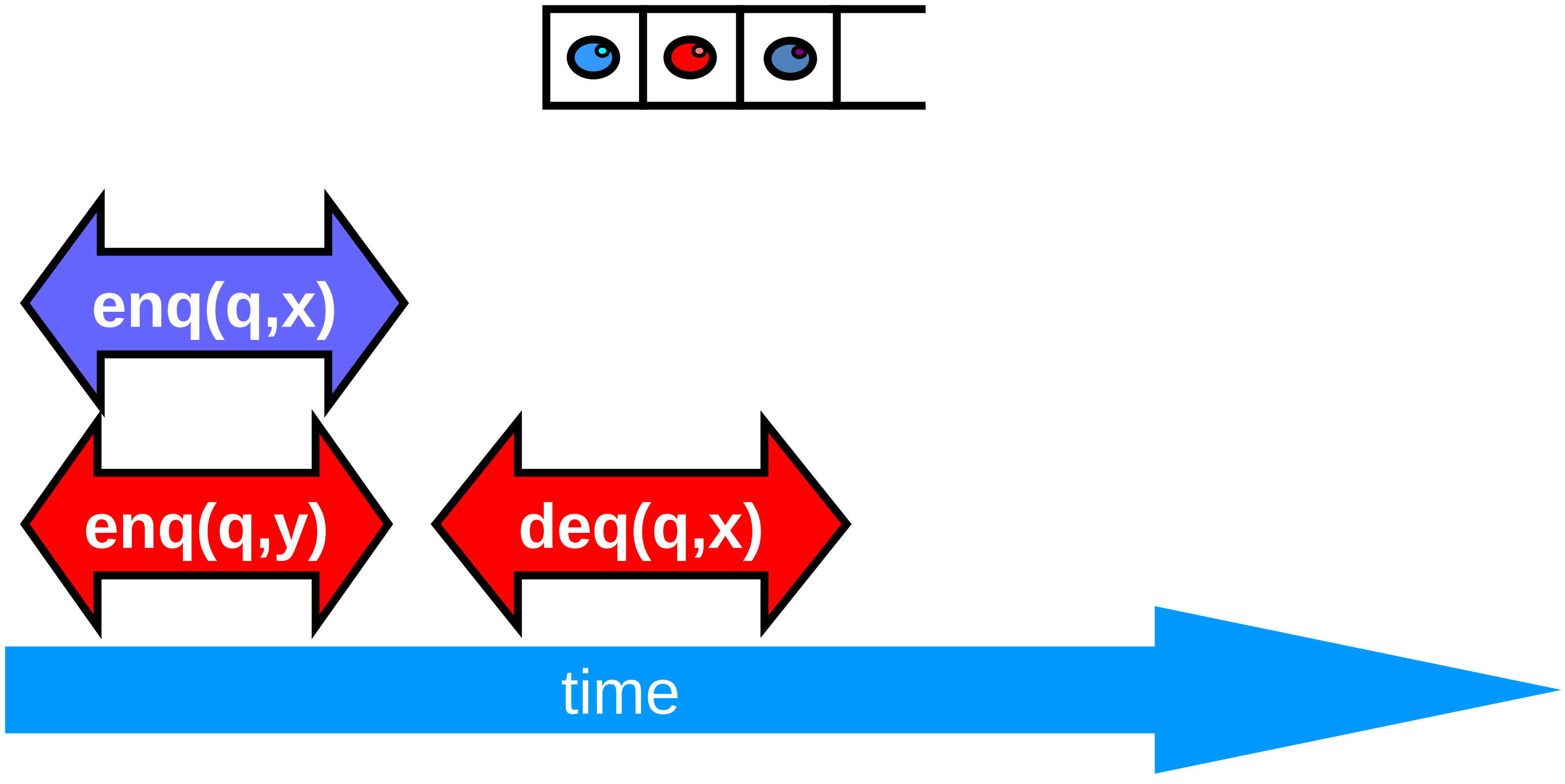
Example



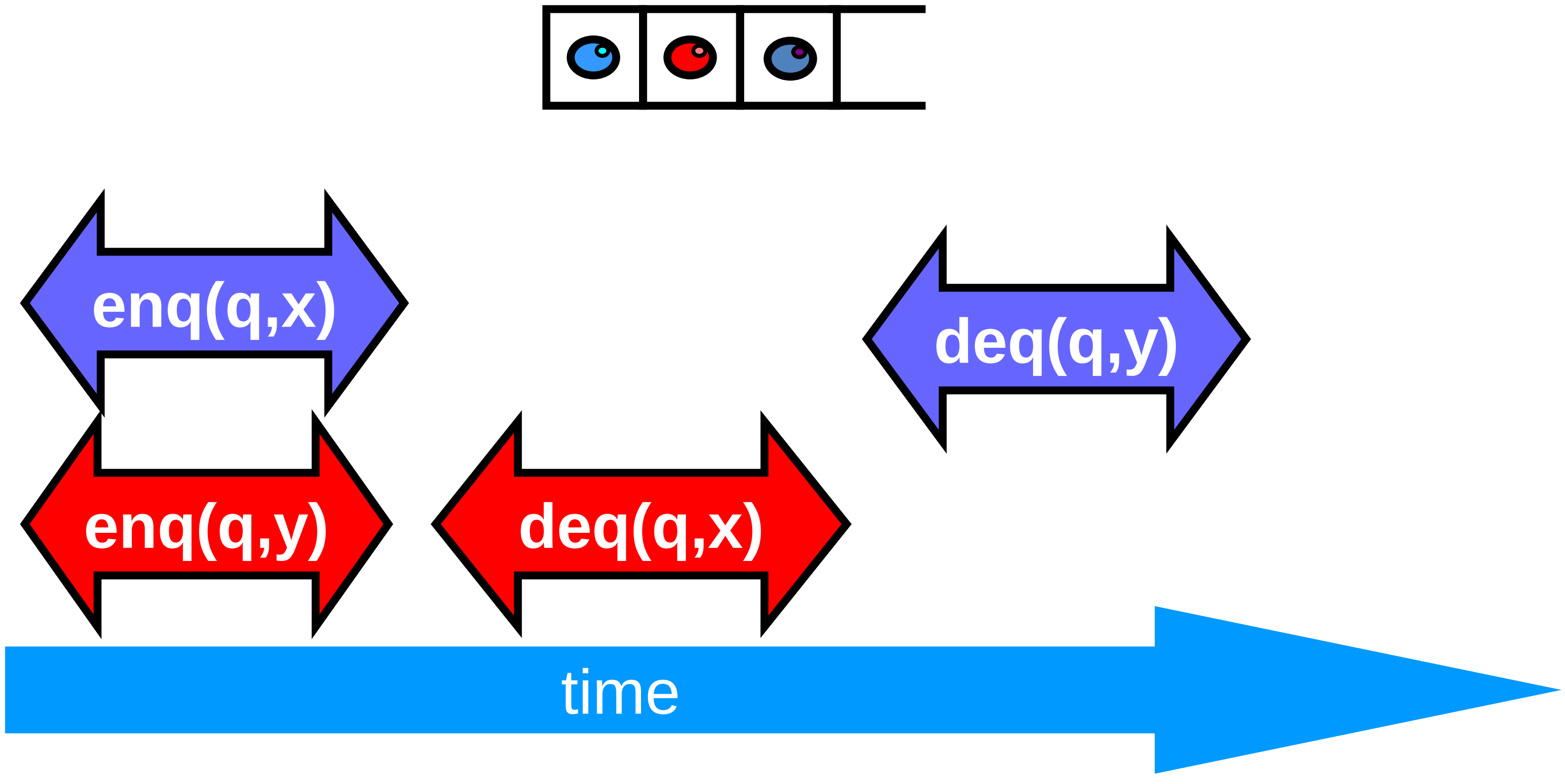
Example



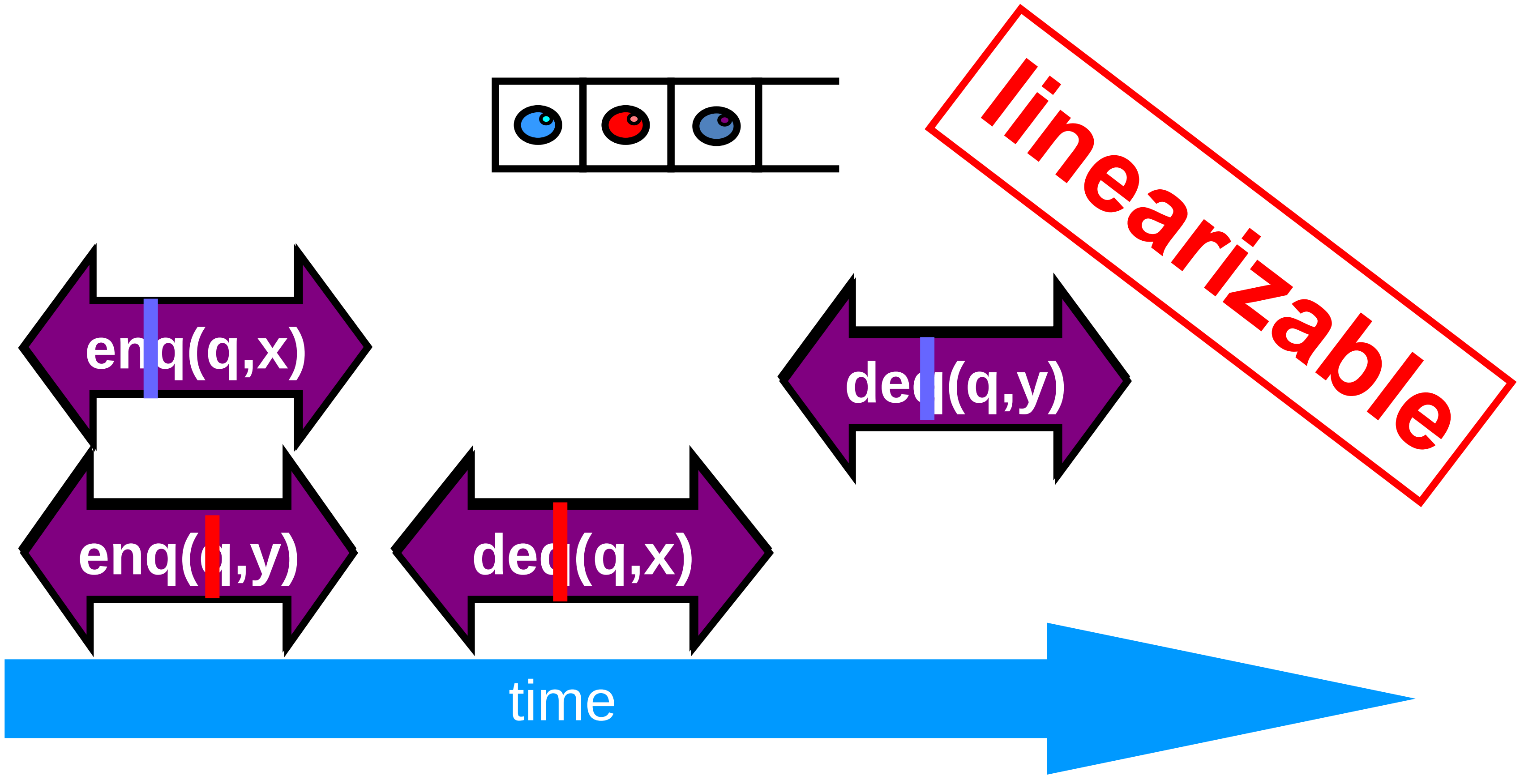
Example



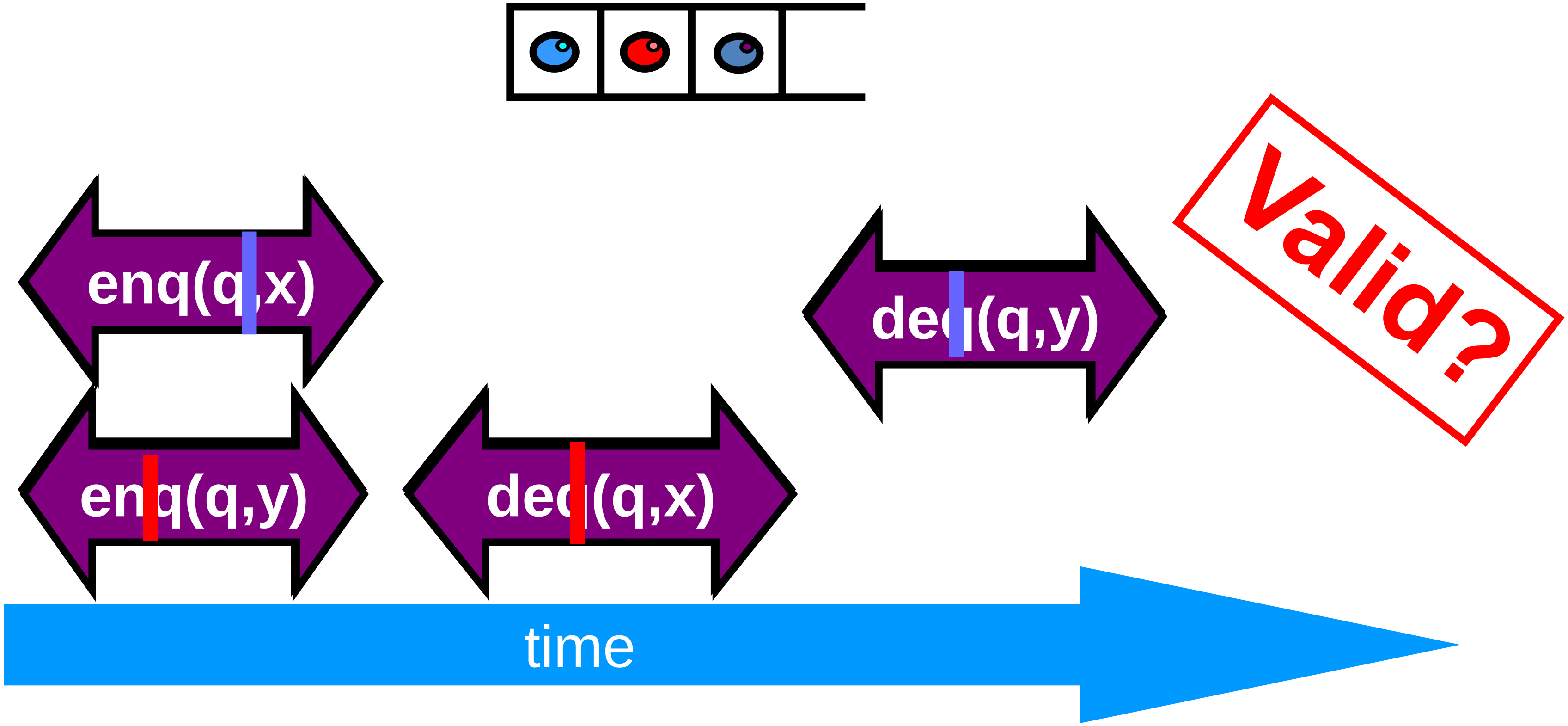
Example



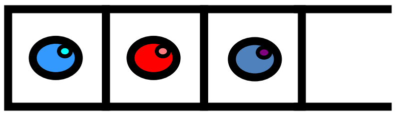
Example



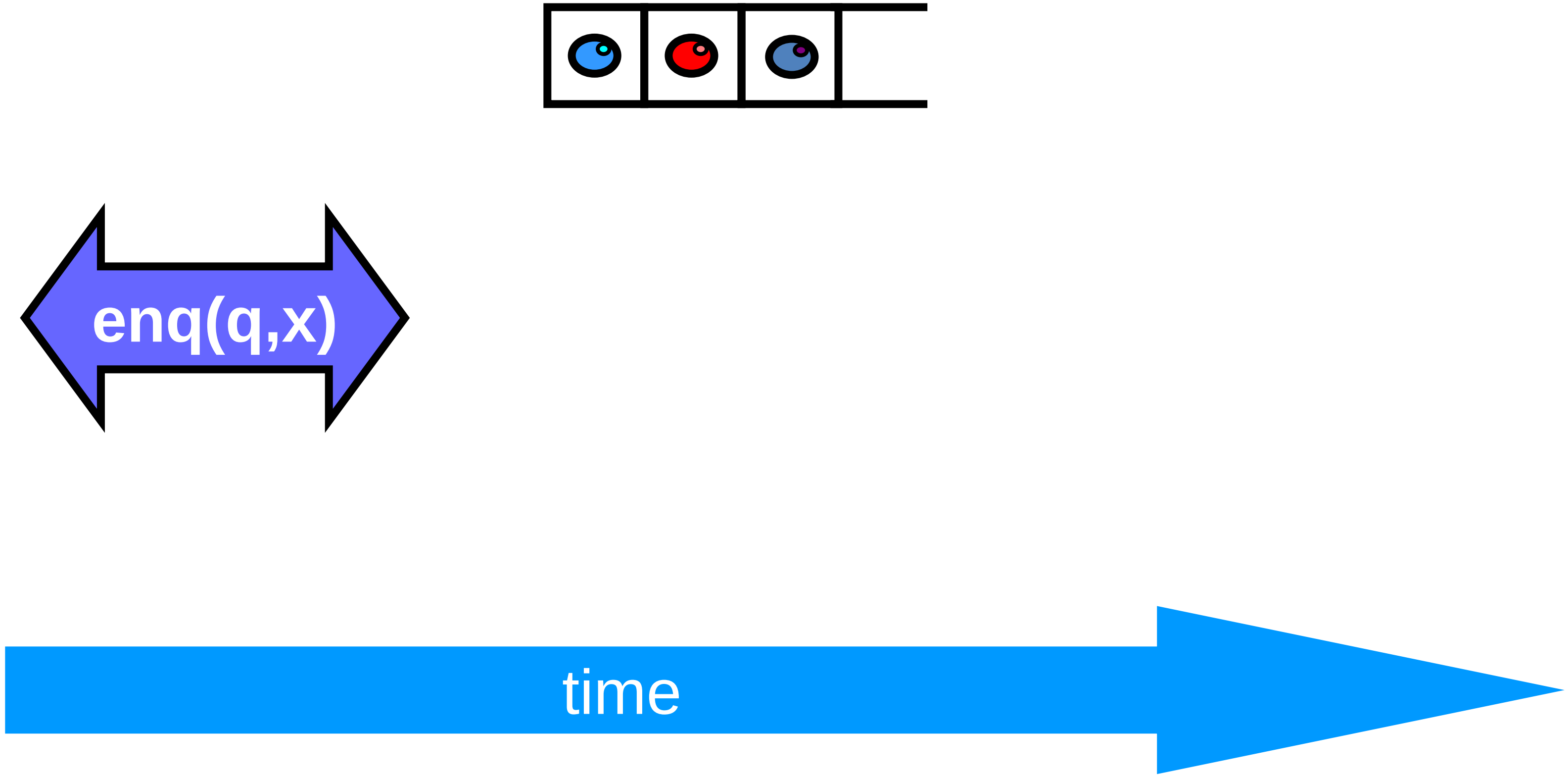
Example



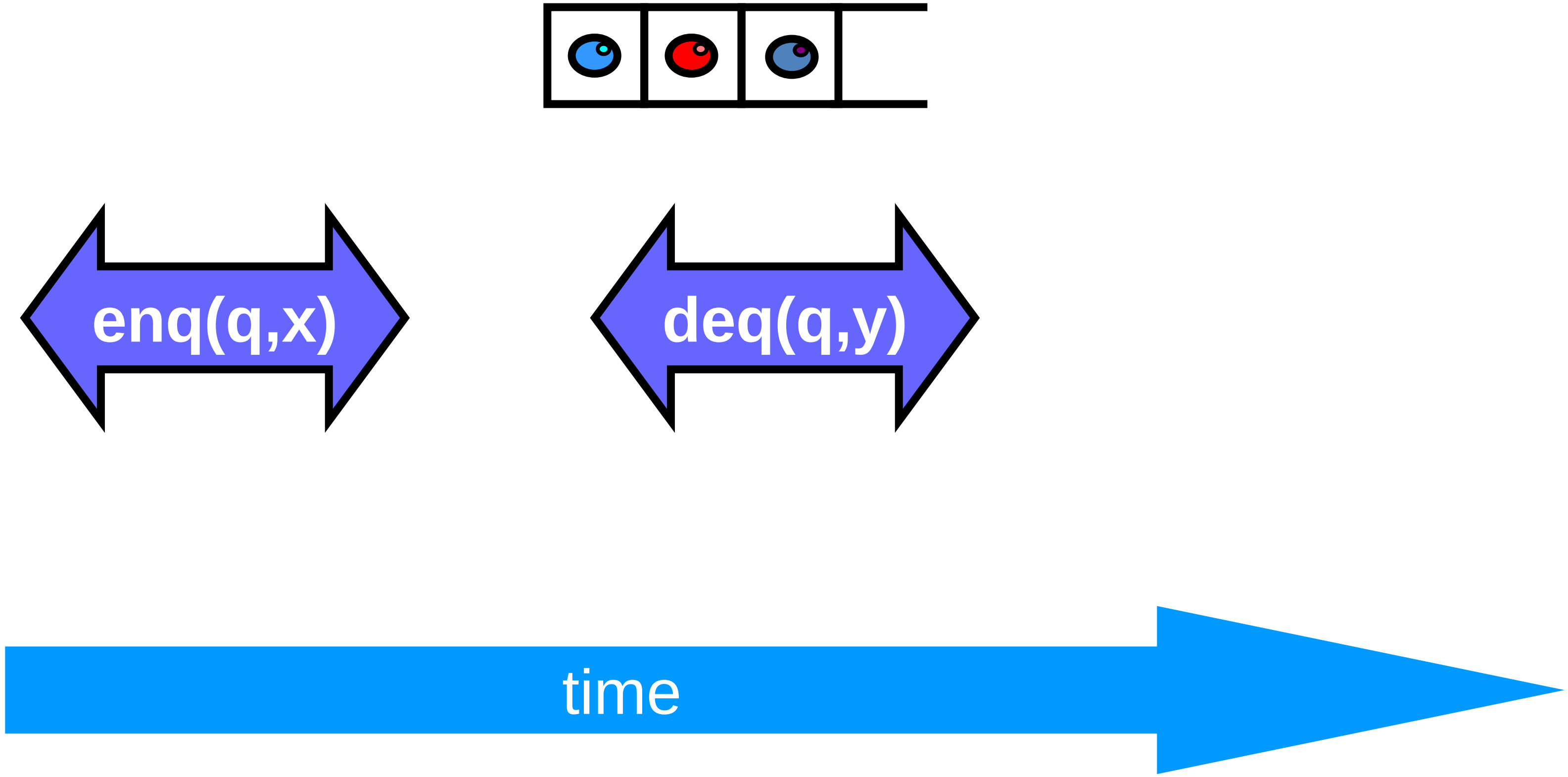
Example



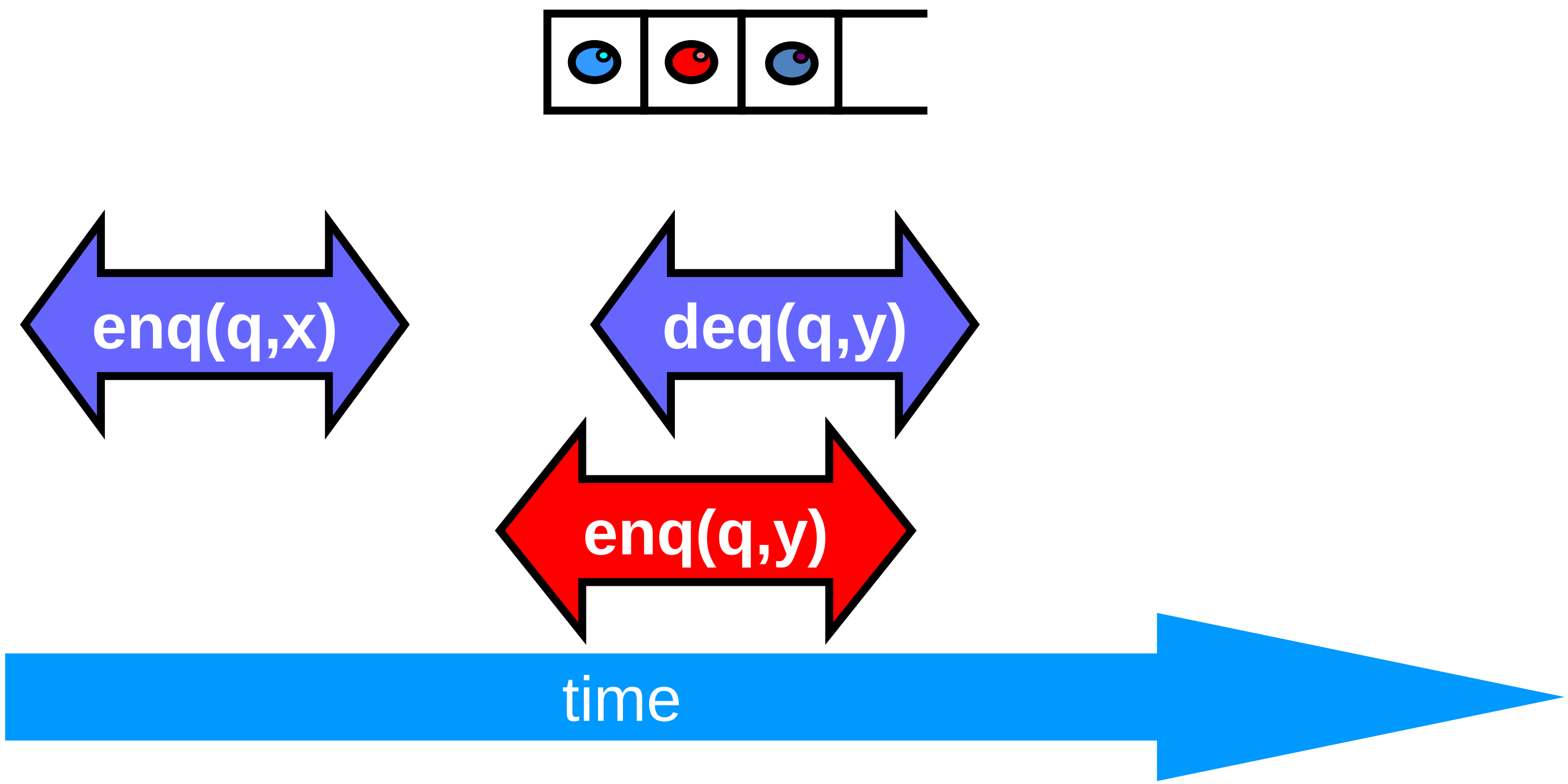
Example



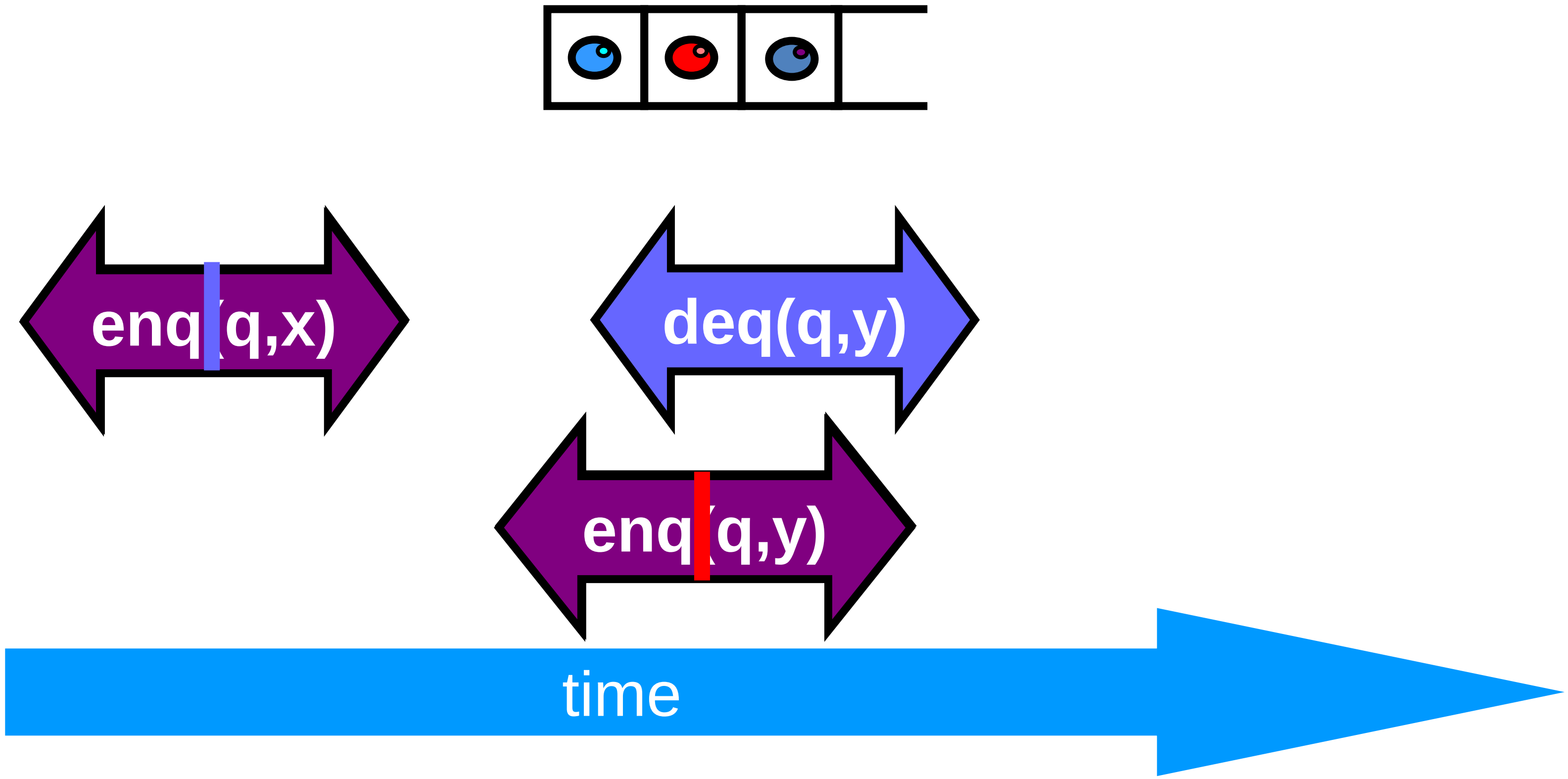
Example



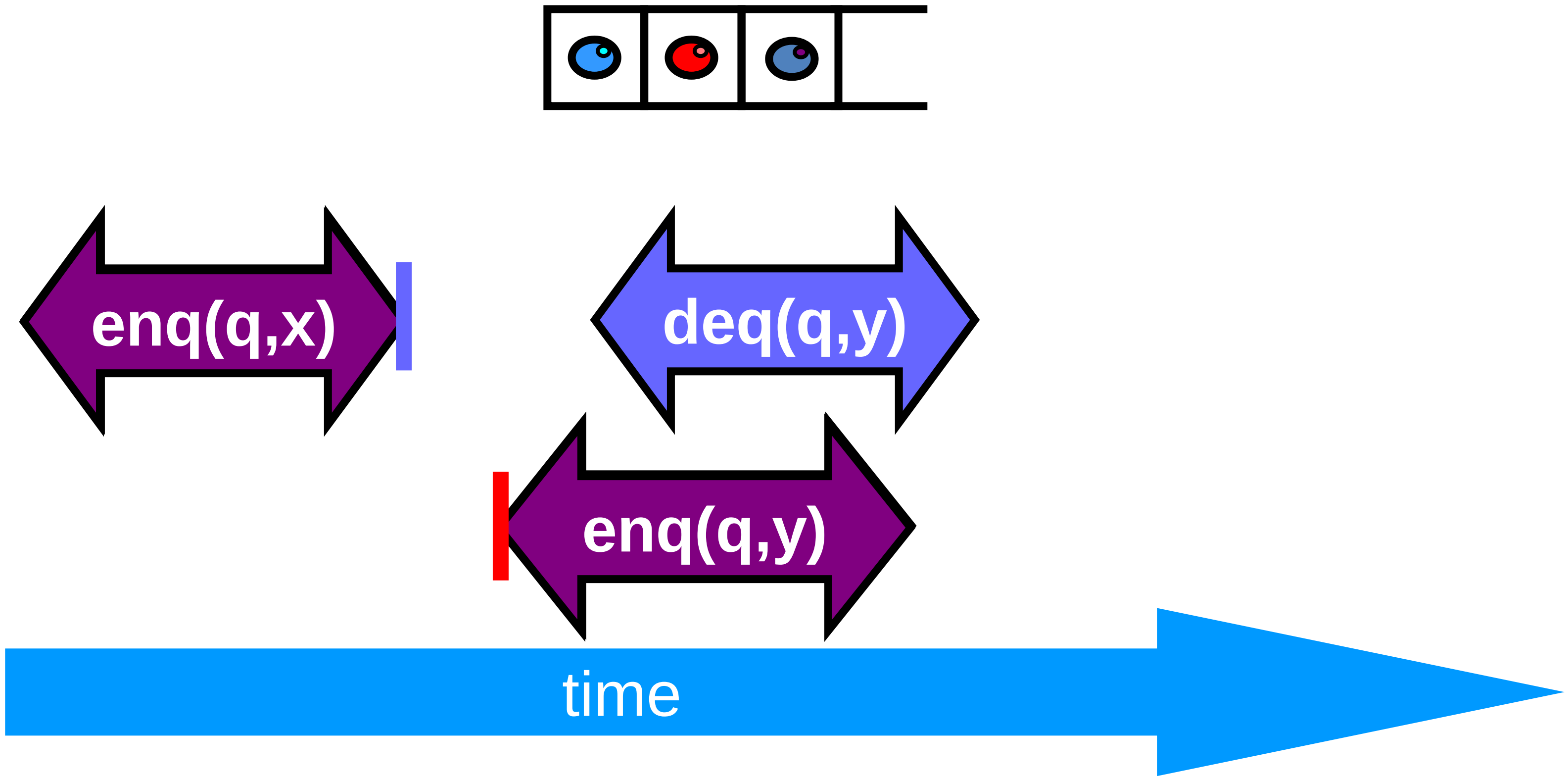
Example



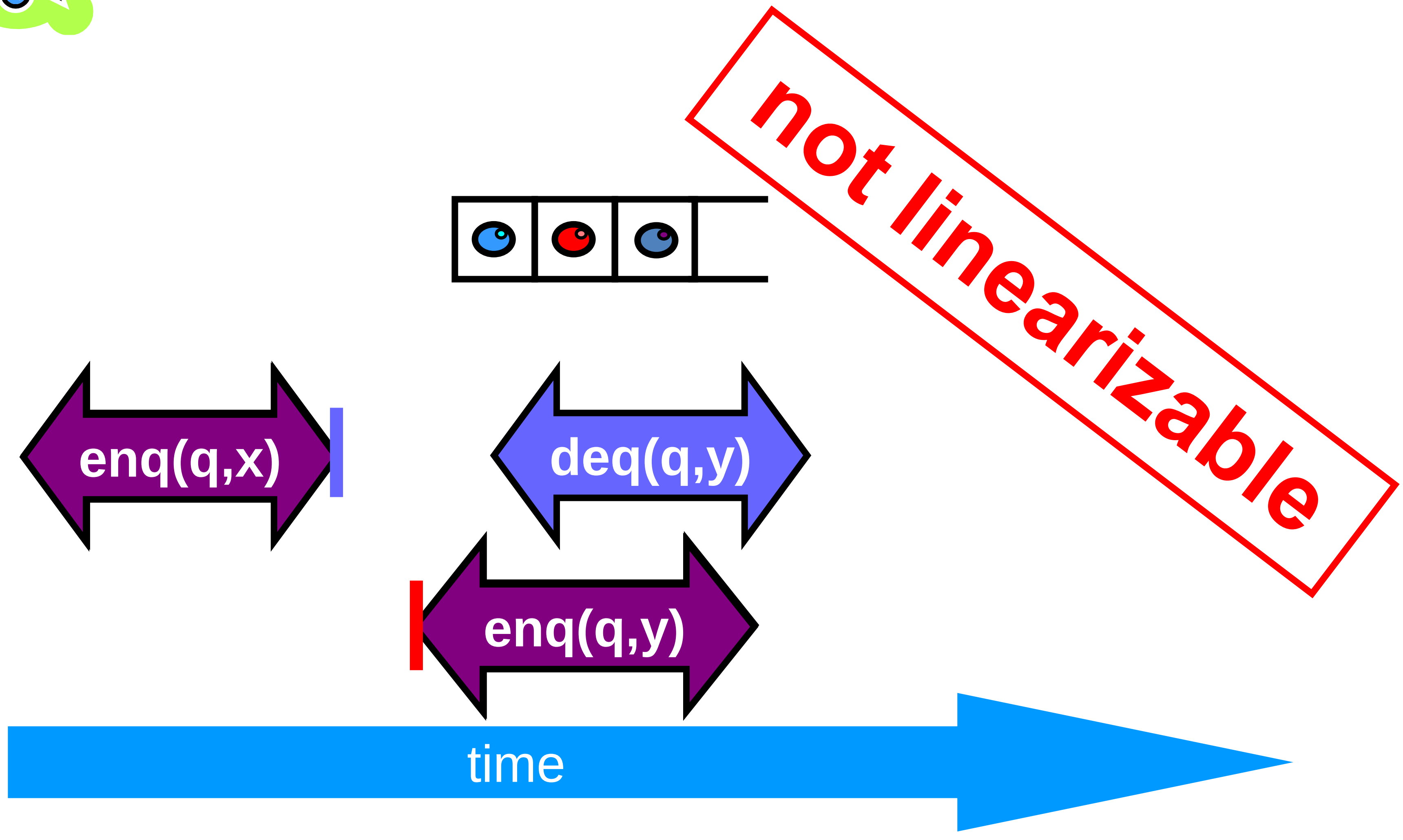
Example



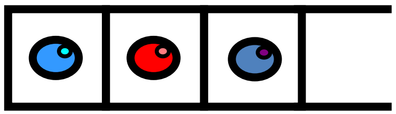
Example



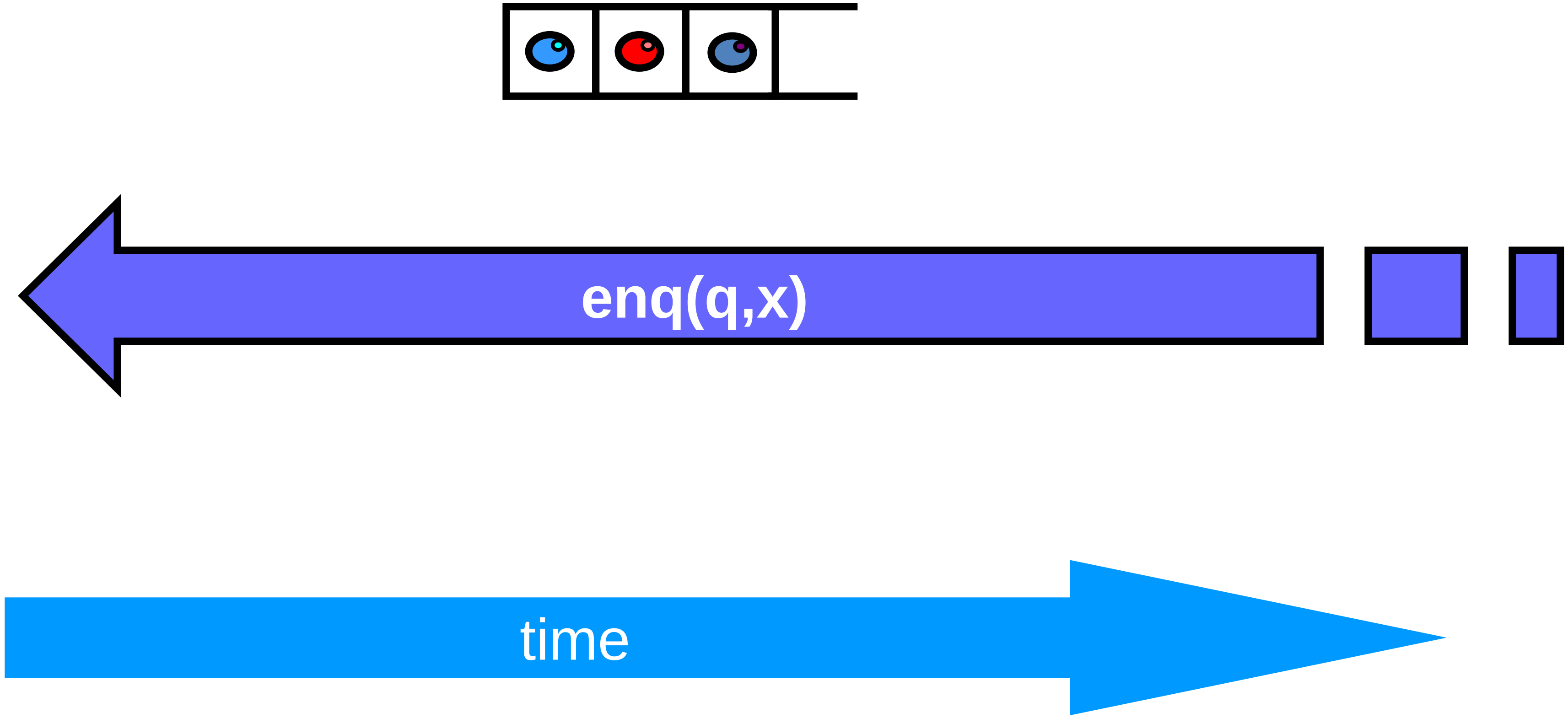
Example



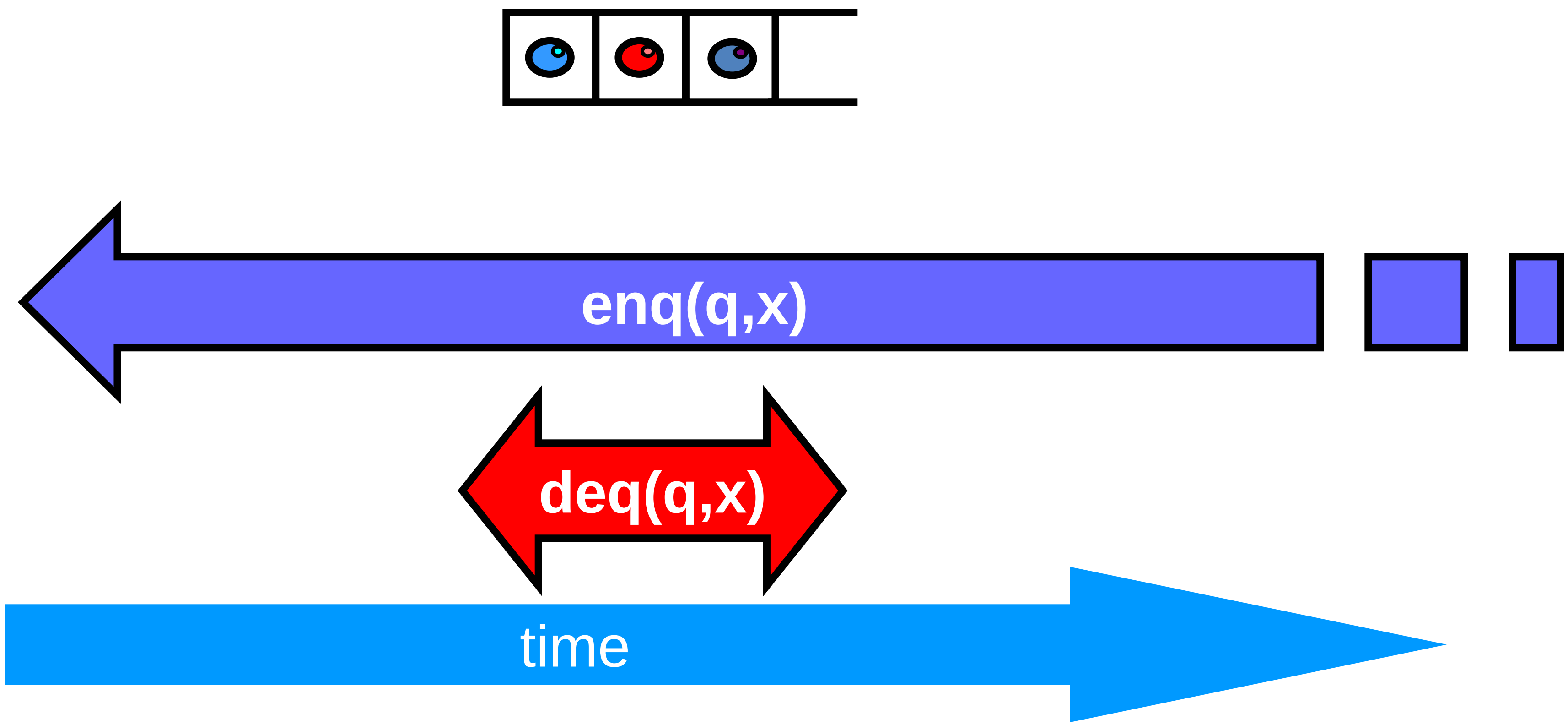
Example



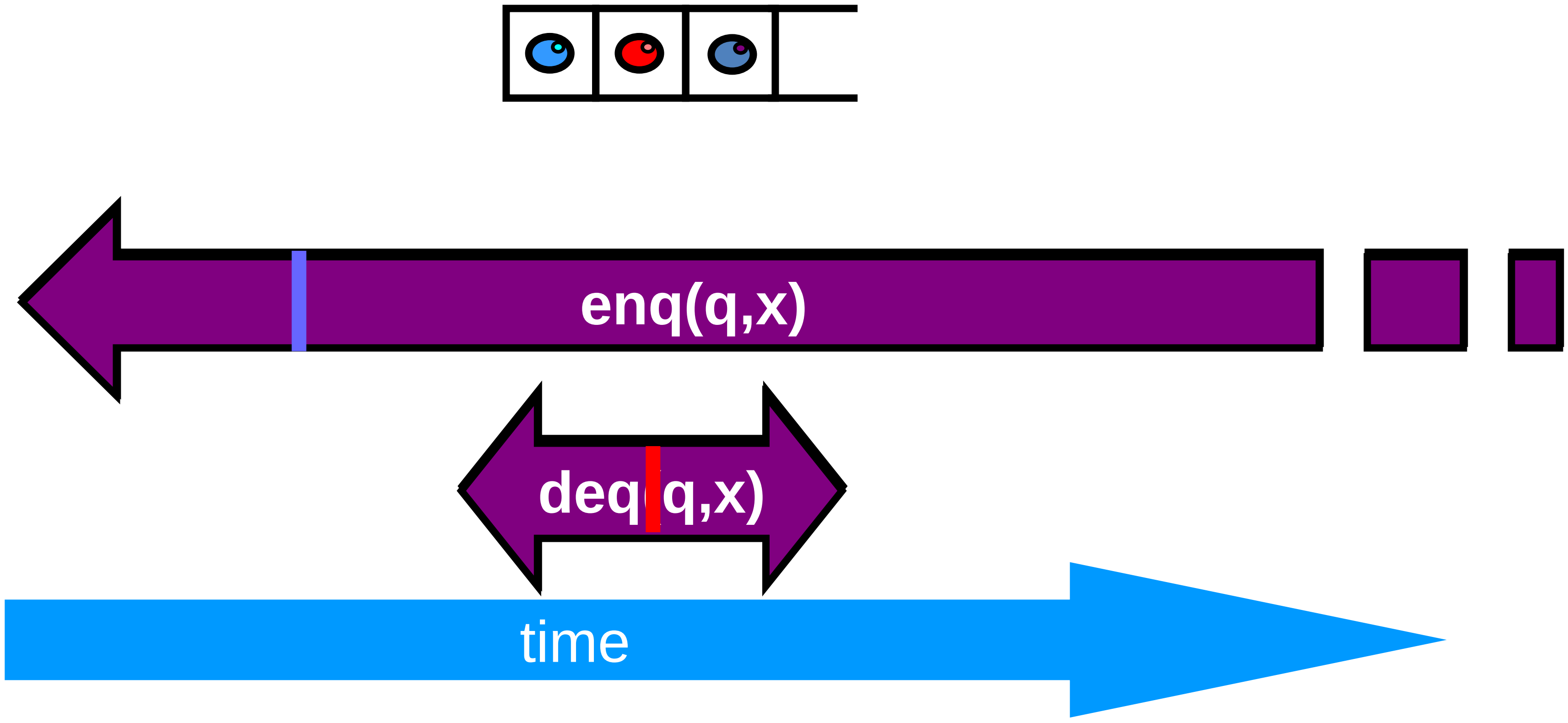
Example



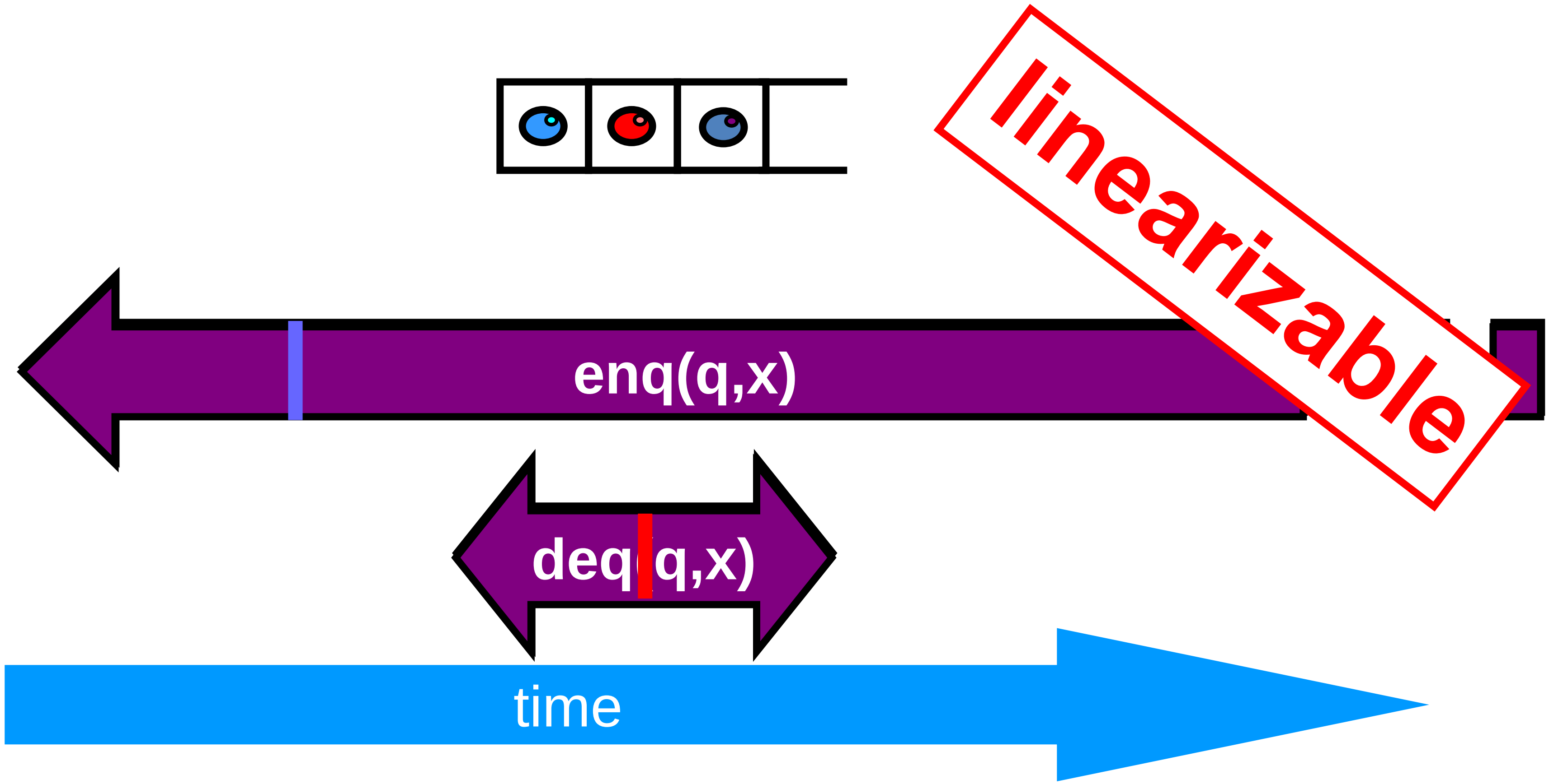
Example



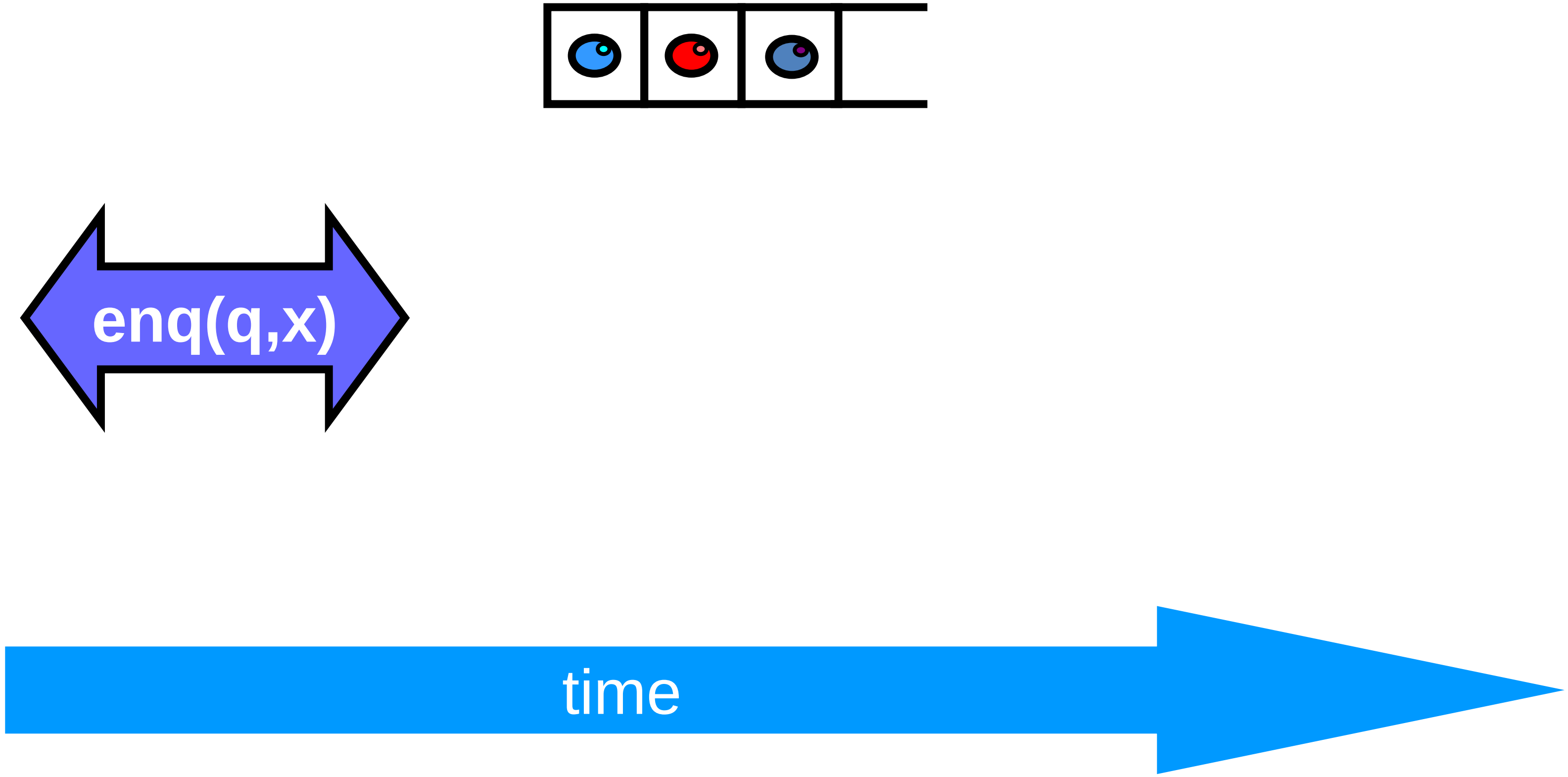
Example



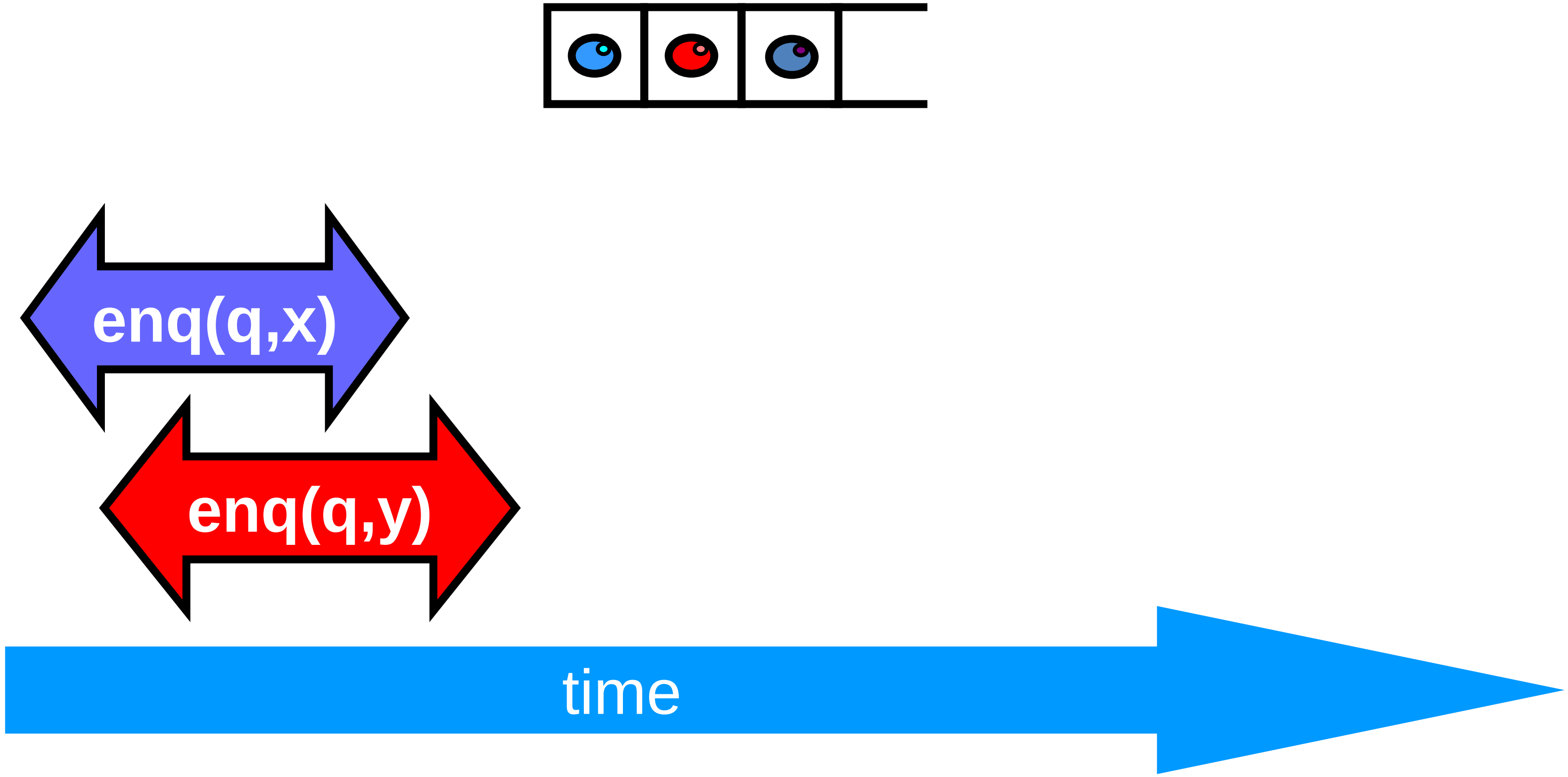
Example



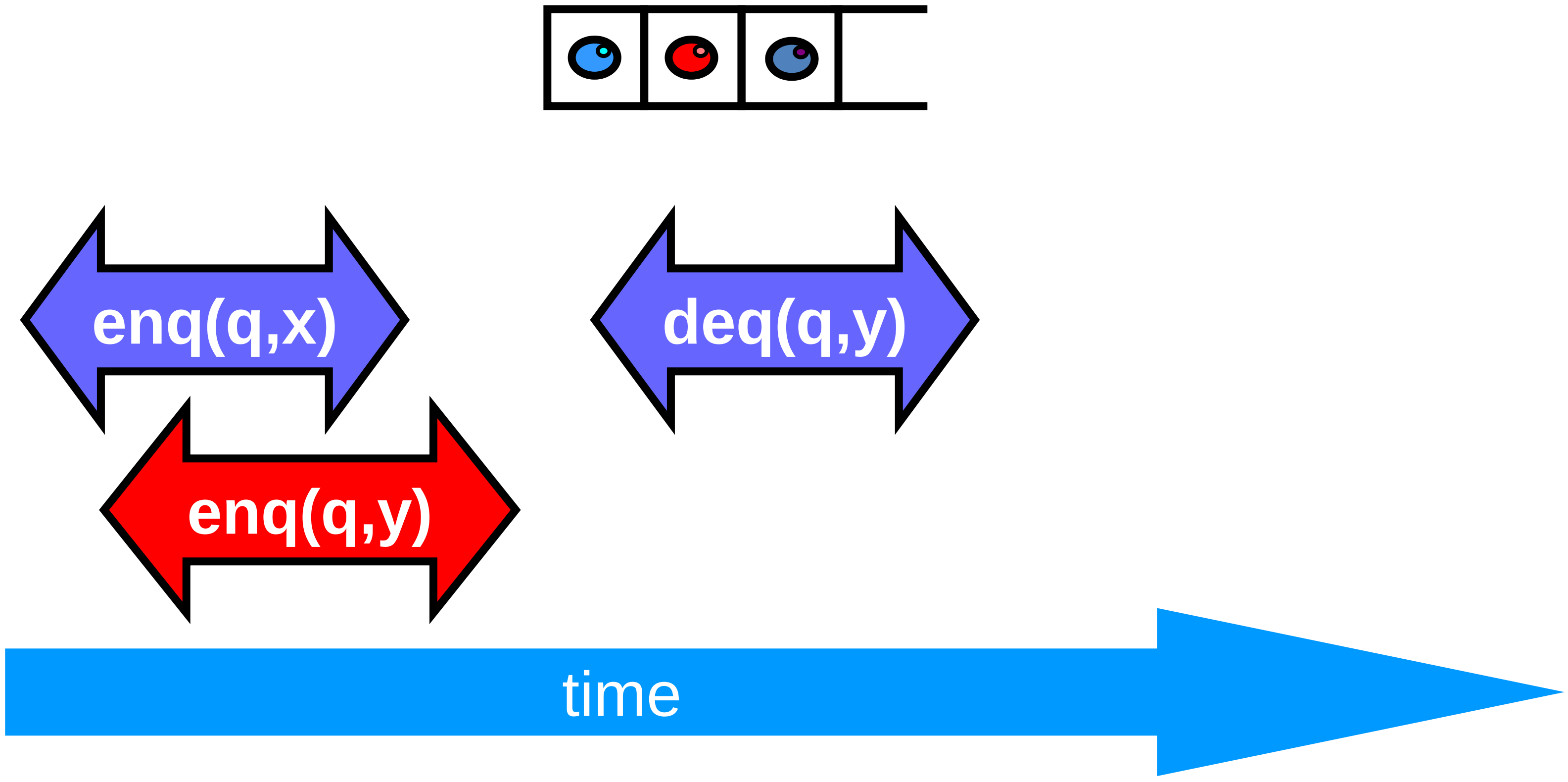
Example



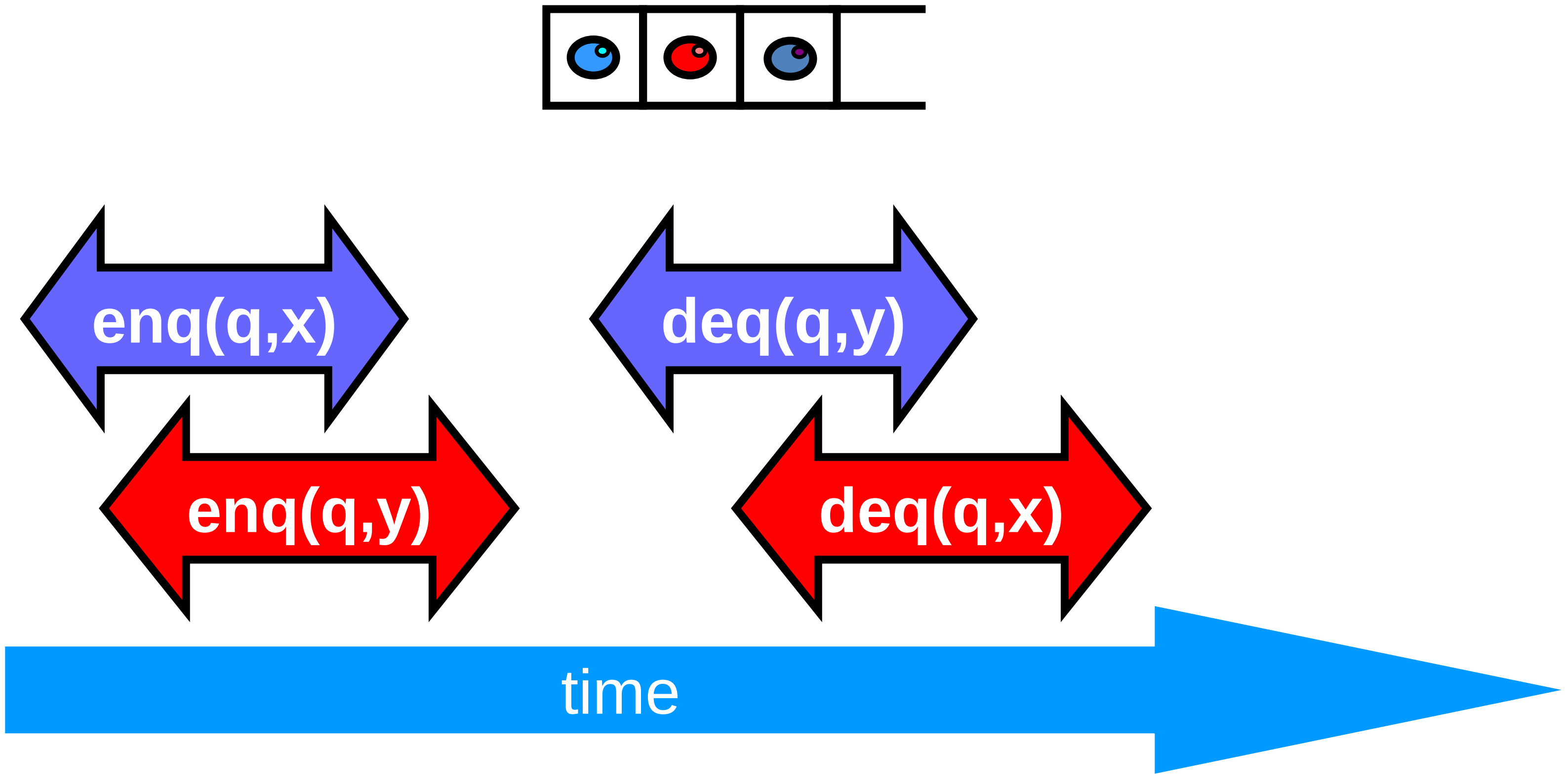
Example



Example

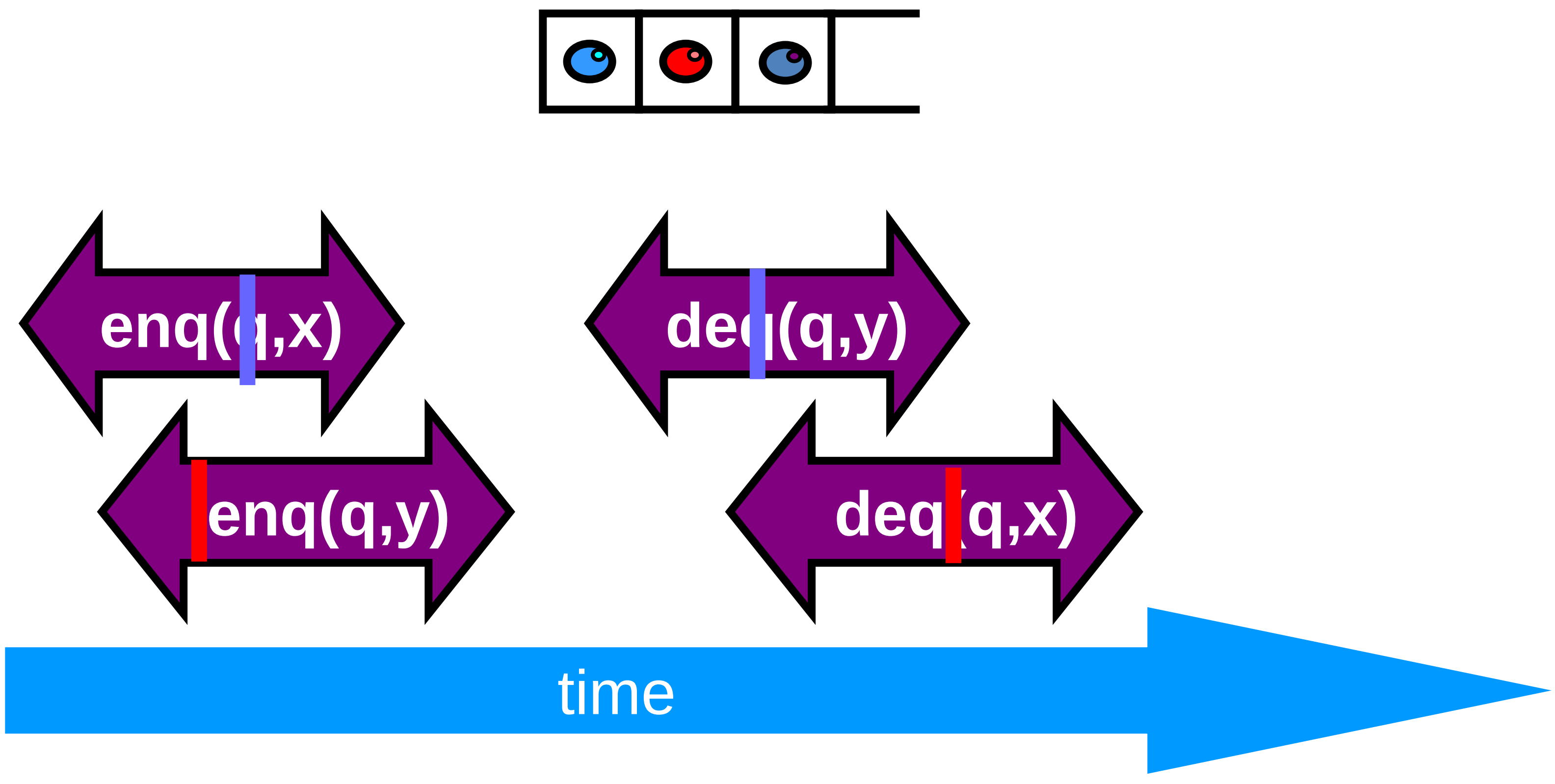


Example



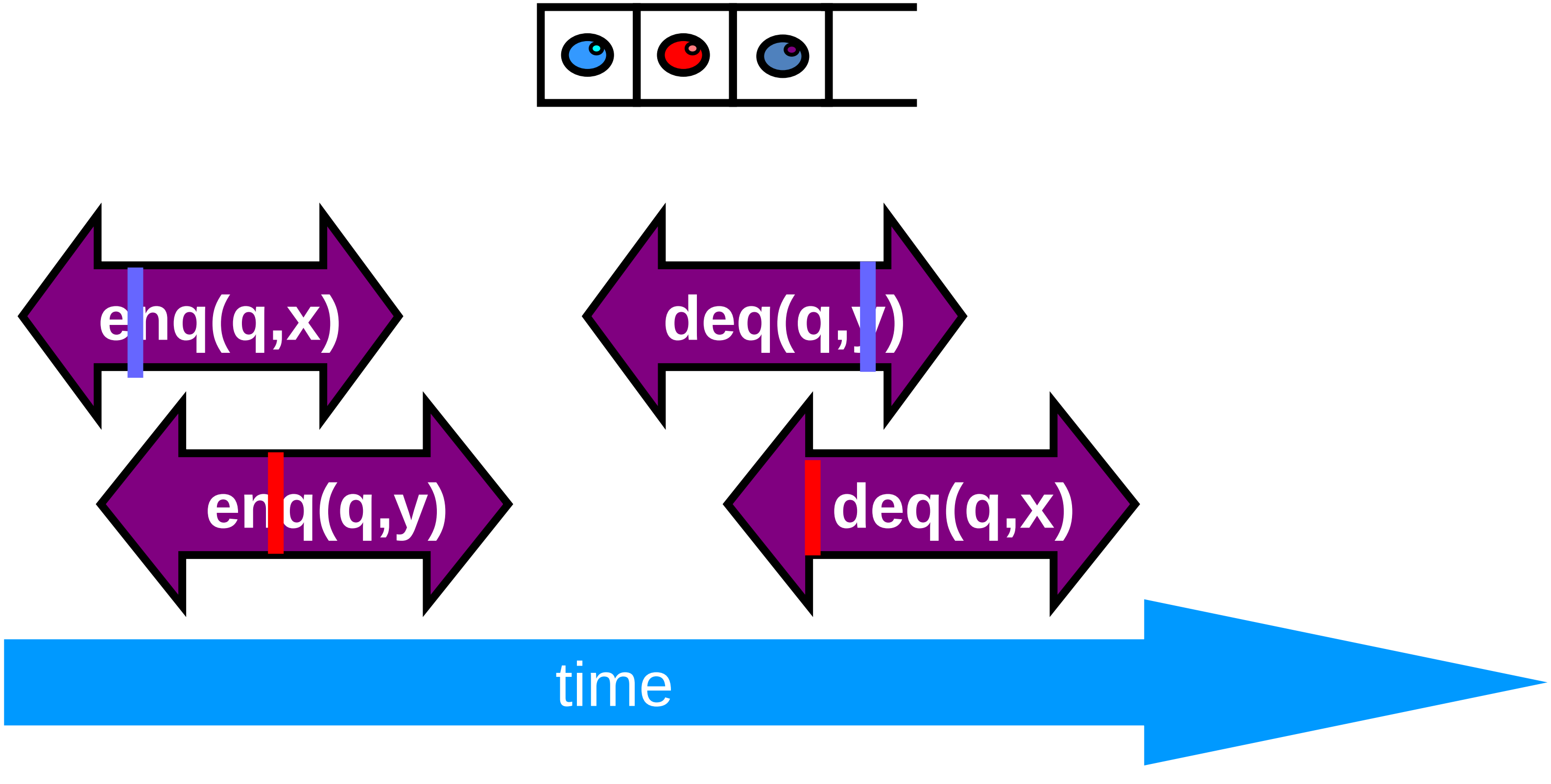
Comme ci

Example



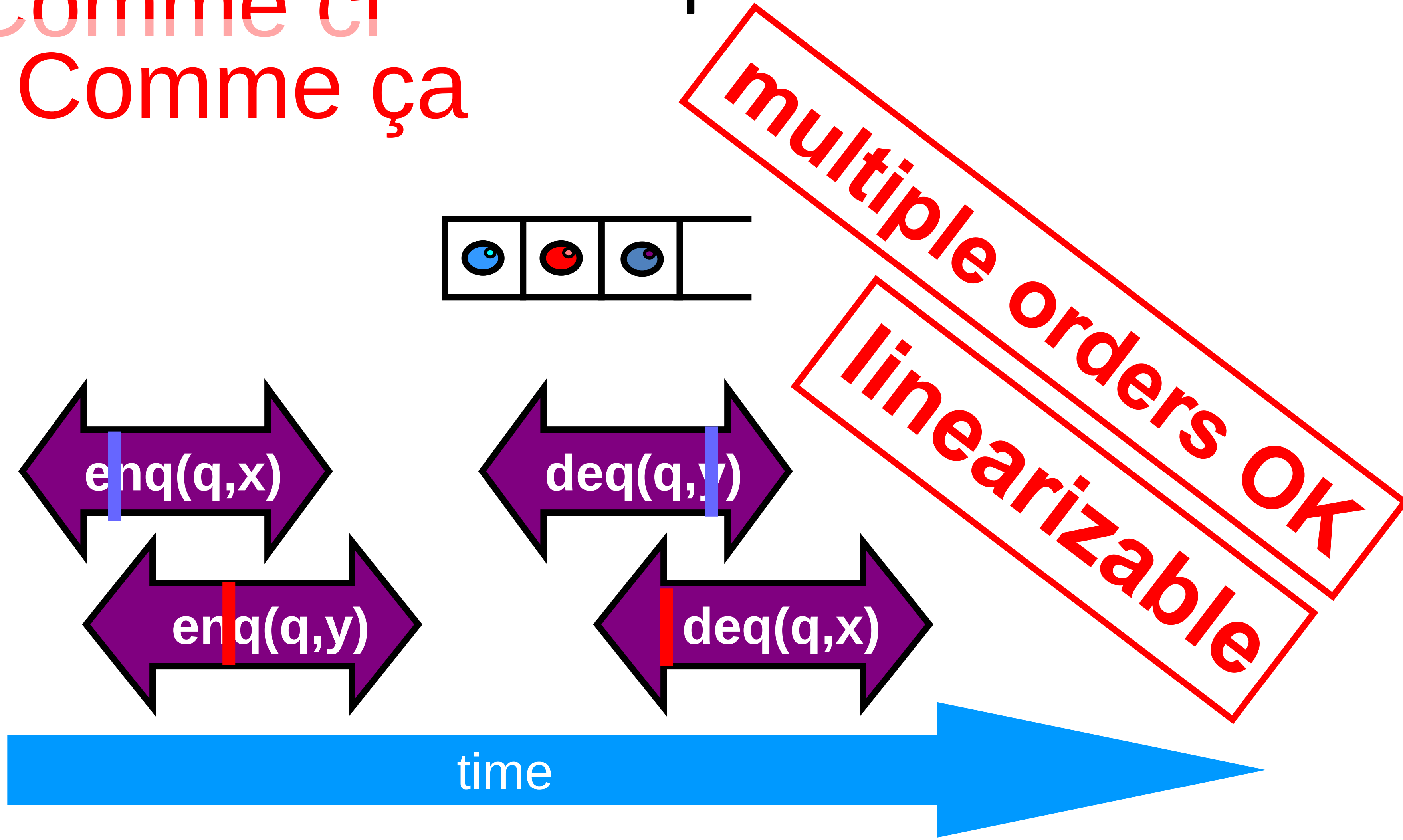
Example

Comme ci
Comme ça



Example

Comme ci
Comme ça



Talking About Executions

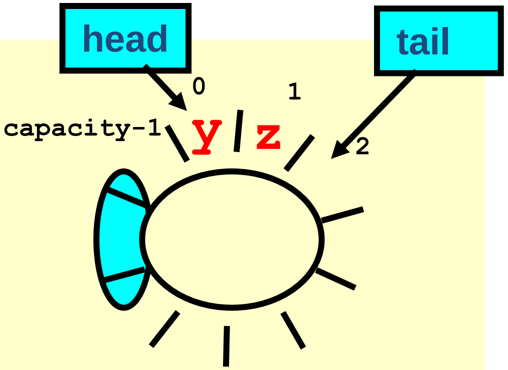
- Why executions?
  - Can't we specify the linearization point of each operation without describing an execution?
- Not Always
  - In some cases, linearization point depends on the execution

Linearizable Objects are Composable

- Modularity
- Can prove linearizability of objects in isolation
- Can compose independently-implemented objects

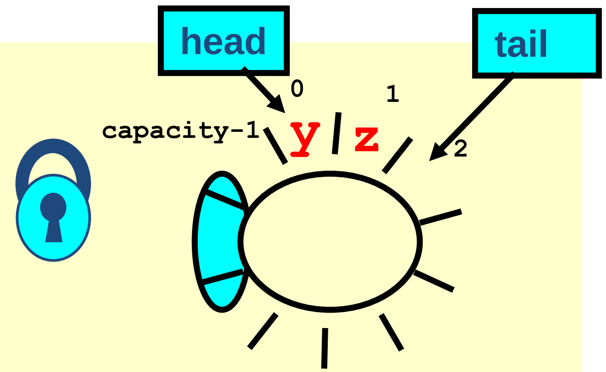
Reasoning About Linearizability: Locking

```
int deq(queue_t q, void **elem) {
  int res;
  pthread_mutex_lock(&q->lock);
  if (q->tail == q->head) res = 0;
  else {
    *elem = q->items[q->head % CAPACITY];
    q->head++;
    res = 1;
  }
  pthread_mutex_unlock(&q->lock);
  return res;
}
```



Reasoning About Linearizability: Locking

```
int deq(queue_t q, void **elem) {
  int res;
  pthread_mutex_lock(&q->lock);
  if (q->tail == q->head) res = 0;
  else {
    *elem = q->items[q->head % CAPACITY];
    q->head++;
    res = 1;
  }
  pthread_mutex_unlock(&q->lock);
  return res;
}
```

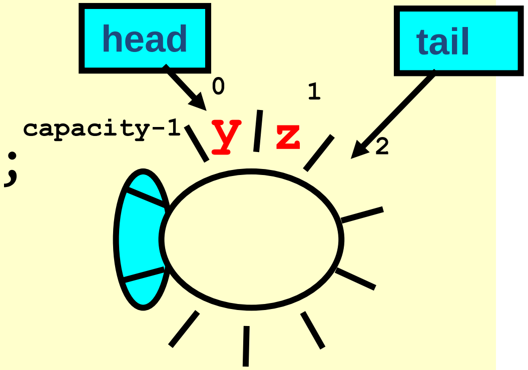


Linearization points are when locks are released

More Reasoning: Wait-free

```
int deq(queue_t q, void **elem) {
  if (q->tail == q->head) return = 0;
  *elem = q->items[q->head % CAPACITY];
  q->head++;
  return 1;
}

int enq(queue_t q, void *x) {
  if (tail-head == CAPACITY) return 0;
  q->items[q->tail % CAPACITY] = x;
  q->tail++;
  return 1;
}
```



More Reasoning: Wait-free

```
int deq(queue_t q, void **elem) {
  if (q->tail == q->head
  *elem = q->items[q->he
  q->head++;
  return 1;
}

int enq(queue_t q, void *x) {
  if (tail-head == CAPACITY) return 0;
  q->items[q->tail % CAPACITY] = x;
  q->tail++;
  return 1;
}
```

head

tail

Linearization order is order head and tail fields modified

More Reasoning: Wait-free

```
int deq(queue_t q, void **elem) {
  if (q->tail == q->head
  *elem = q->items[q->he
  q->head++;
  return 1;
}

int enq(queue_t q, void *x) {
  if (tail-head == CAPACITY) return 0;
  q->items[q->tail % CAPACITY] = x;
  q->tail++;
  return 1;
}
```

head

tail

Linearization order is order head and tail fields modified

Remember that there is only one enqueuer and only one dequeuer

Strategy

- Identify one atomic step where method "happens"
  - Critical section
  - Machine instruction
- Doesn't always work
  - Might need to define several different steps for a given method

Linearizability: Summary

- Powerful specification tool for shared objects
- Allows us to capture the notion of objects being "atomic"
- Don't leave home without it

Progress

- We saw an implementation whose methods were lock-based (deadlock-free)
- We saw an implementation whose methods did not use locks (lock-free)
- How do they relate?

Progress Conditions

- *Deadlock-free:* some thread trying to acquire the lock eventually succeeds.
- *Starvation-free:* every thread trying to acquire the lock eventually succeeds.
- *Lock-free:* some thread calling a method eventually returns.
- *Wait-free:* every thread calling a method eventually returns.

Progress Conditions

	Non-Blocking	Blocking
Everyone makes progress	Wait-free	Starvation-free
Someone makes progress	Lock-free	Deadlock-free