Consistency

Robbert van Renesse

What is consistency?

- I know it when I see it…
    - US Supreme Court Justice Potter Stewart, 1964
        - praised as “realistic and gallant”
        - critiqued as “potentially fallacious, due to individualistic arbitrariness”
            - https://en.wikipedia.org/wiki/I_know_it_when_I_see_it
- An invariant?
    - What about “eventual consistency”?
- Many definitions and flavors
    - weak/relaxed consistency, strong consistency, entry consistency, lazy consistency, causal consistency, causal+ consistency, sequential consistency, FIFO (PRAM) consistency, serializability, strict serializability, linearizability, causal linearizability, and many more

Let’s start with a simple sequential object

- E.g., an integer, a queue, a stack, etc.
- An object has
  - a state
  - a set of methods
- State has an initial value
- Each method may change the state and returns a value of some sort
  - easy to specify usually through a pre-condition and a post-condition
  - we will only consider deterministic methods

Example, a queue specification

- state: sequence of values, initially [ ]
- methods:
    - enqueue(v: Value) -> r: ()
        - r := ()
        - state := state :: [v]
    - dequeue() -> r: Value or ERROR
        - if state == []
            - r := ERROR
            - state := state
        - if state <> []:
            - r := head(state)
            - state := tail(state)

Example, a queue specification

- state: sequence of values, initially [ ]
- methods:
    - enqueue(v: Value) -> r: ()
        - r := ()
        - state := state :: [v]
    - dequeue() -> r: Value or ERROR
        - if state == []
            - state := state
            - r := ERROR
        - if state <> []:
            - r := head(state)
            - state := tail(state)

Can easily be translated into TLA+ or Harmony, say

Some (nice) observations about sequential specifications

- State is meaningful only *between* method calls
- Methods only interact through passing state
- Sequence of operations (method calls) defines a behavior
- Specification is “linear” in the number of methods
- Can add new methods without having to change the old ones

- “inconsistency” simply is violating the spec:
  - enqueue(1)
  - enqueue(2)
  - dequeue() → 2

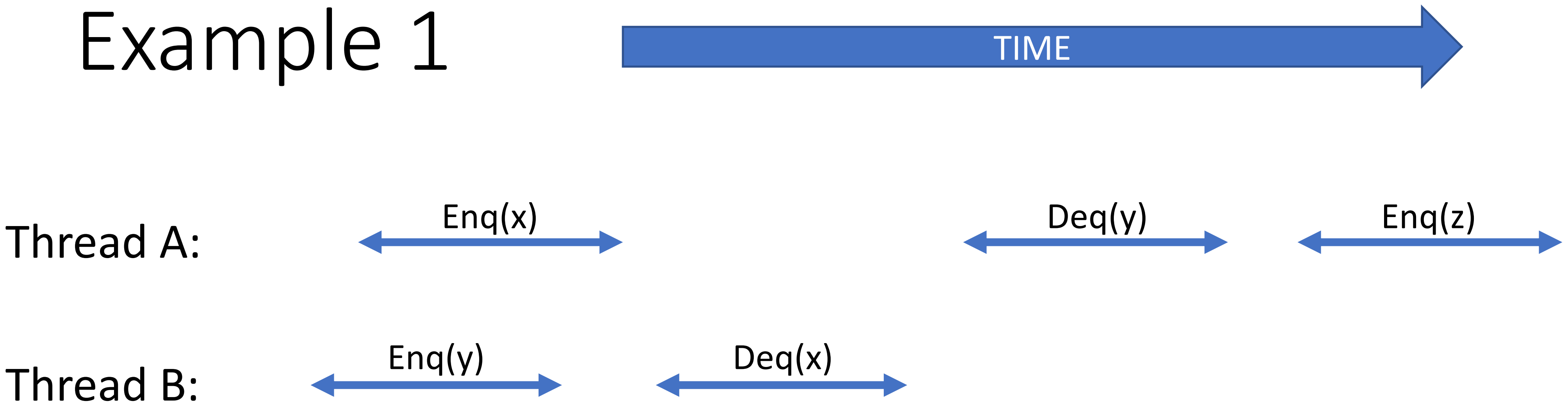
What happens when you add concurrency?

- E.g., what happens when two enqueue() operations are invoked at approximately the same time by two different threads (processes)?
- You have to consider what happens with state *during* an operation
  - all possible interactions…
- Operations take time! (Who knew?)
- Operations of different threads *overlap*
  - There many never be “between method calls”
- Many different cases to consider
  - Specification complicated and not linear in the number of methods ☹

What do we mean by consistency??

- Let’s look at some examples

Example 1



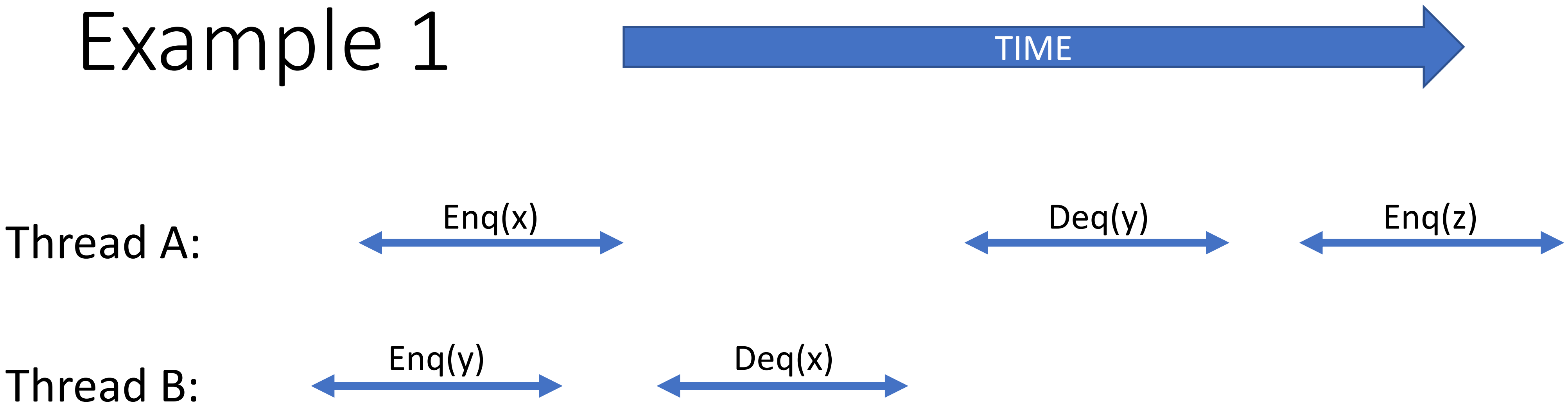
TIME

Thread A: Enq(x) Deq(y) Enq(z)

Thread B: Enq(y) Deq(x)

“Consistent” or not?

Example 1



TIME

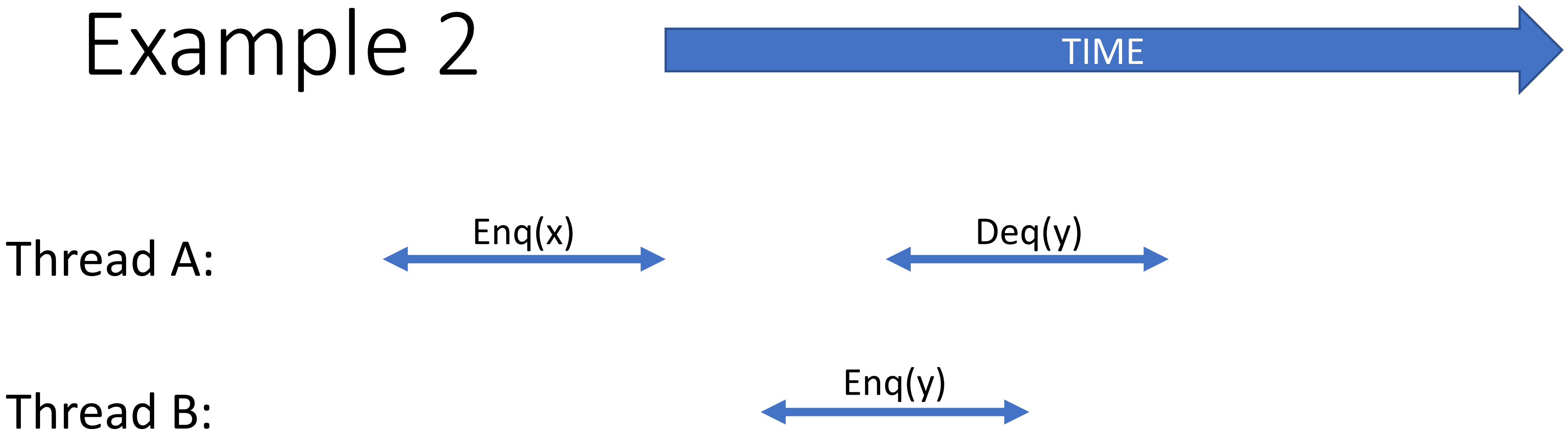
Thread A: Enq(x) Deq(y) Enq(z)

Thread B: Enq(y) Deq(x)

“Consistent” or not?

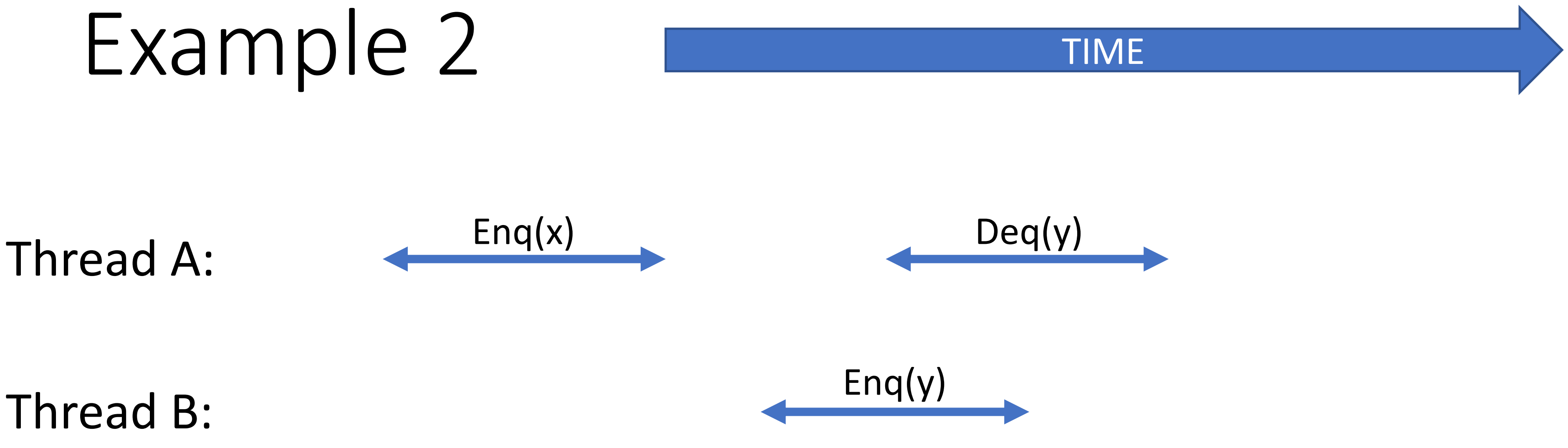
✓

Example 2



“Consistent” or not?

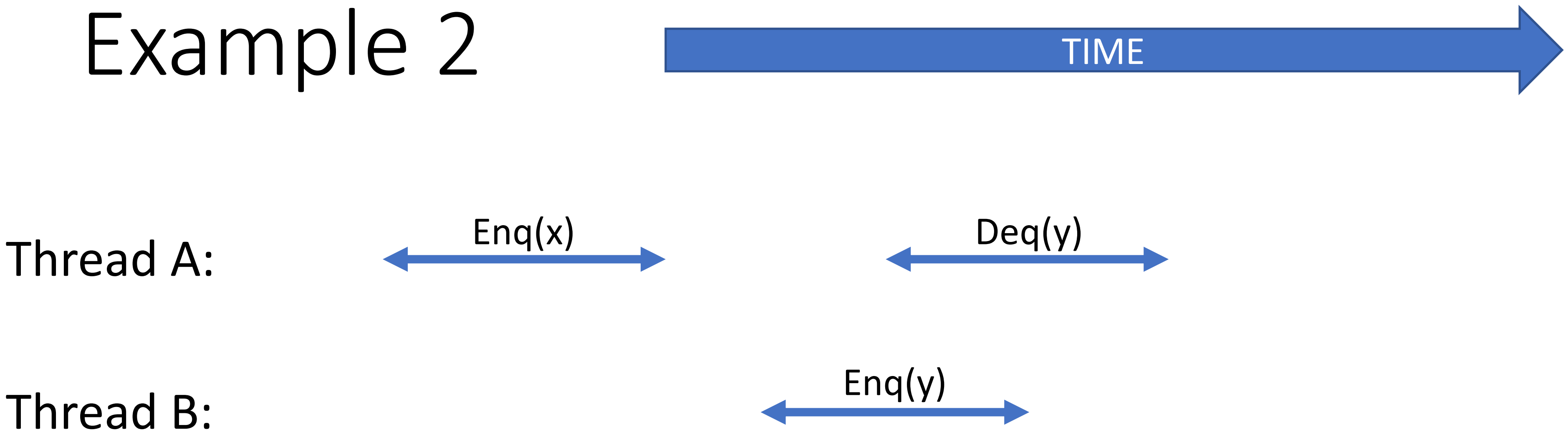
Example 2



“Consistent” or not?

✗

Example 2



TIME

Thread A: Enq(x) Deq(y)

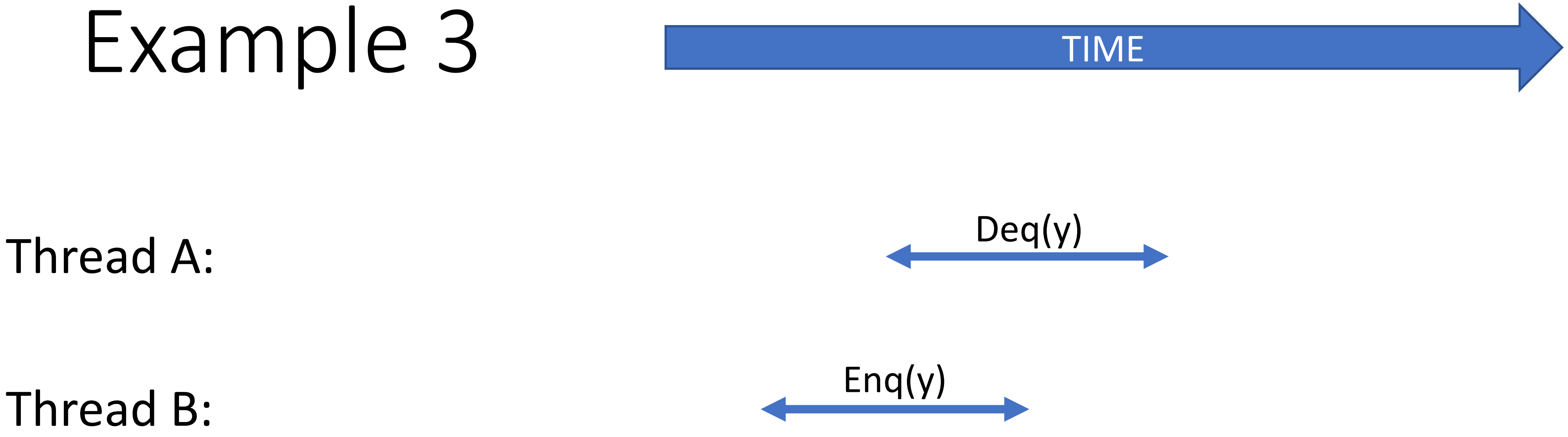
Thread B: Enq(y)

“Consistent” or not?

✗

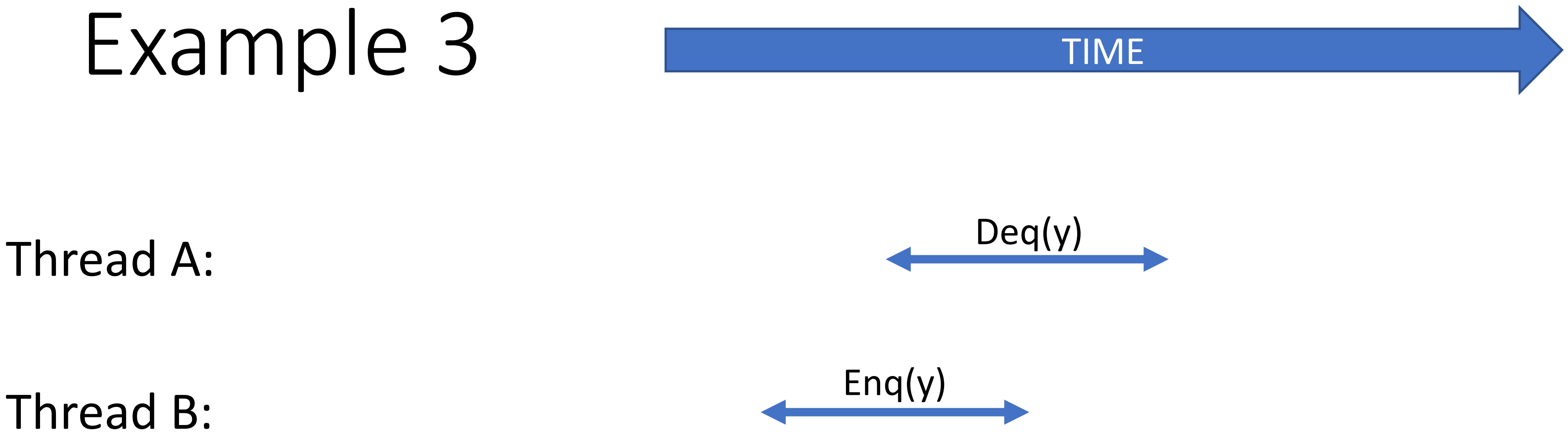
or maybe ✓

Example 3



“Consistent” or not?

Example 3



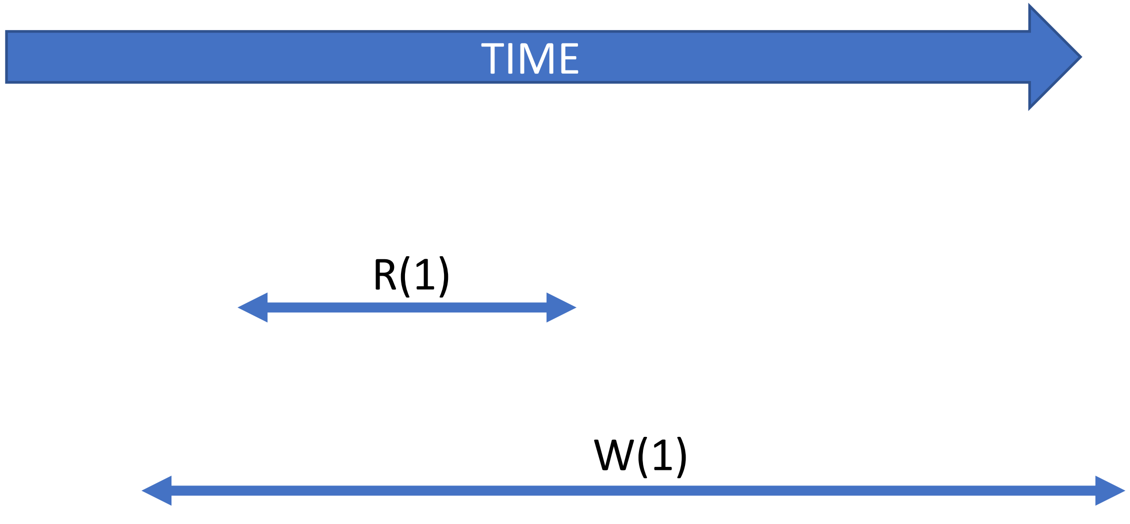
“Consistent” or not?

✓

What about simple register read/write?

- Operations: R(x), W(x)
- Initial value: 0
- Sequential spec:
  - read operation returns value of latest completed write operation
- But what if read and write operations can execute concurrently??

Example 4

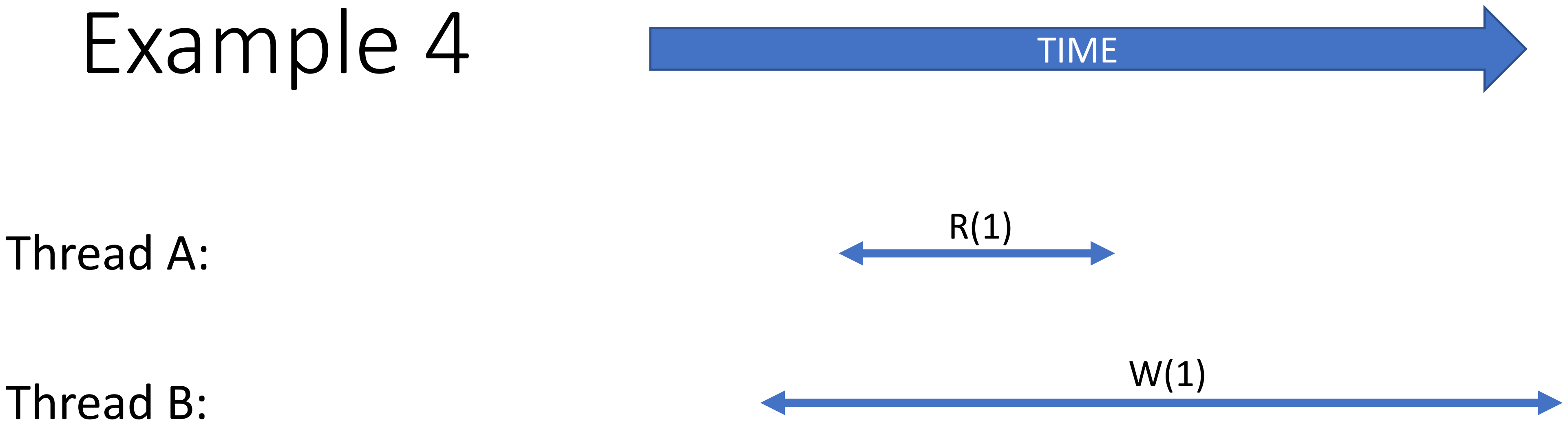


Thread A:

Thread B:

“Consistent” or not?

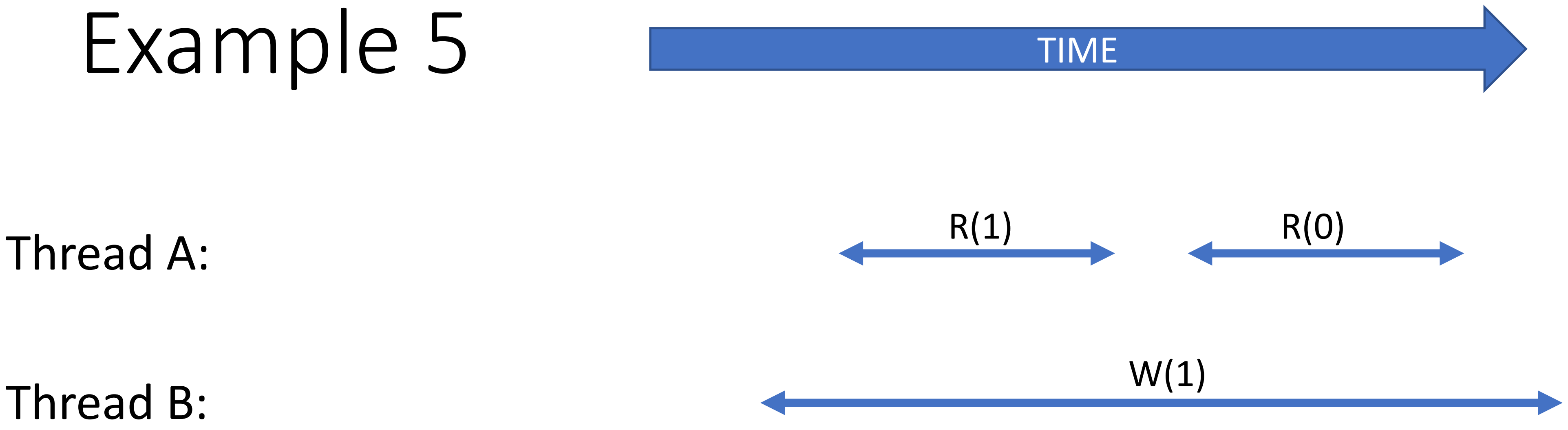
Example 4



“Consistent” or not?

✓

Example 5



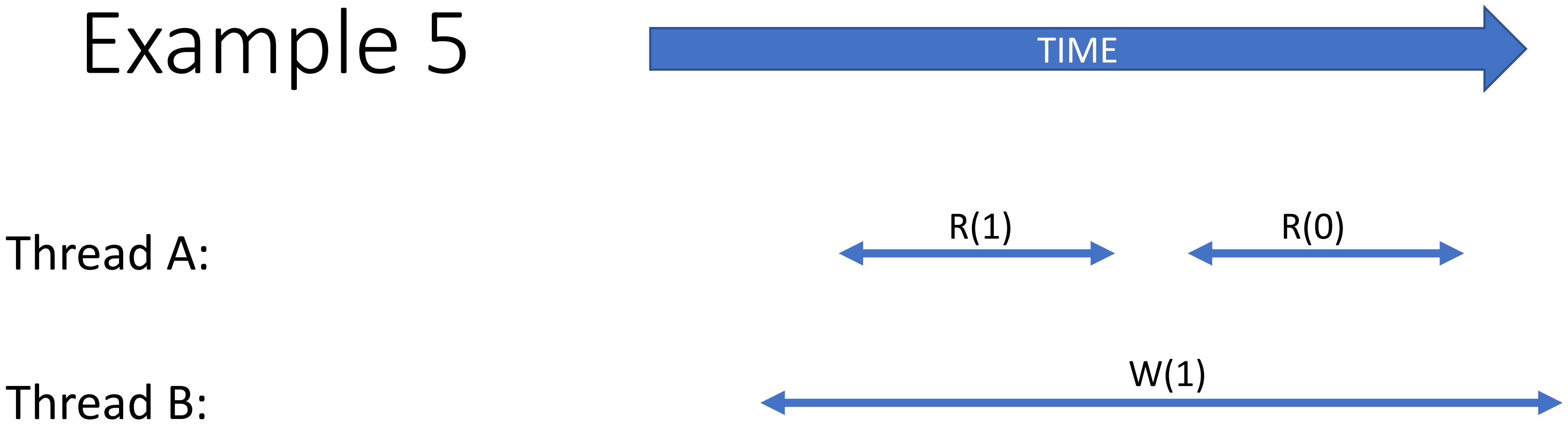
TIME

Thread A: R(1) R(0)

Thread B: W(1)

“Consistent” or not?

Example 5



TIME

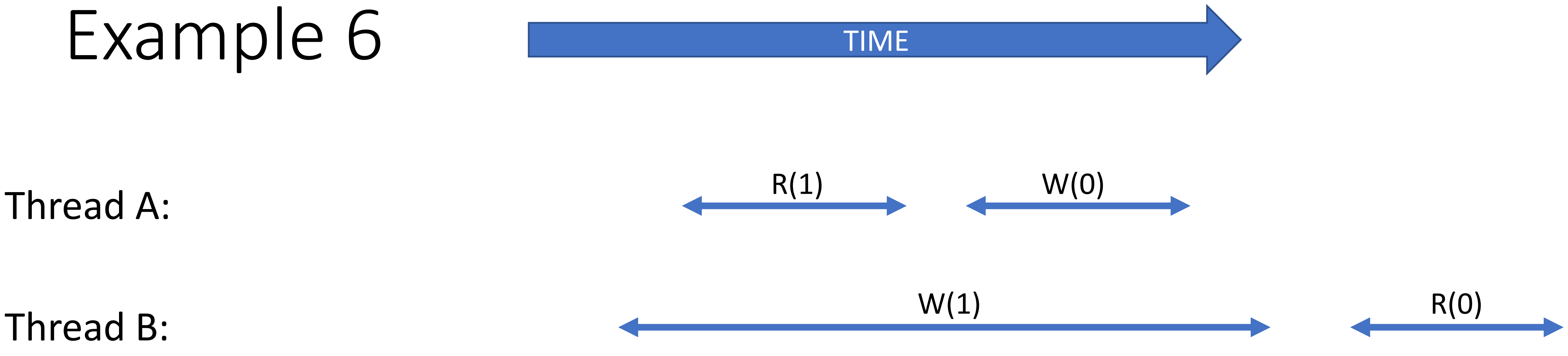
Thread A: R(1) R(0)

Thread B: W(1)

“Consistent” or not?

✗

Example 6



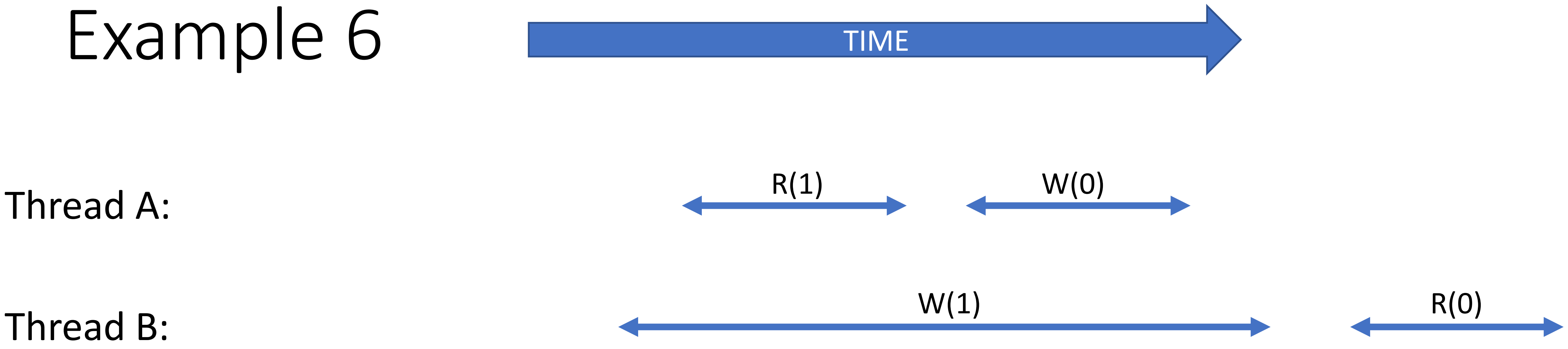
TIME

Thread A: R(1) W(0)

Thread B: W(1) R(0)

“Consistent” or not?

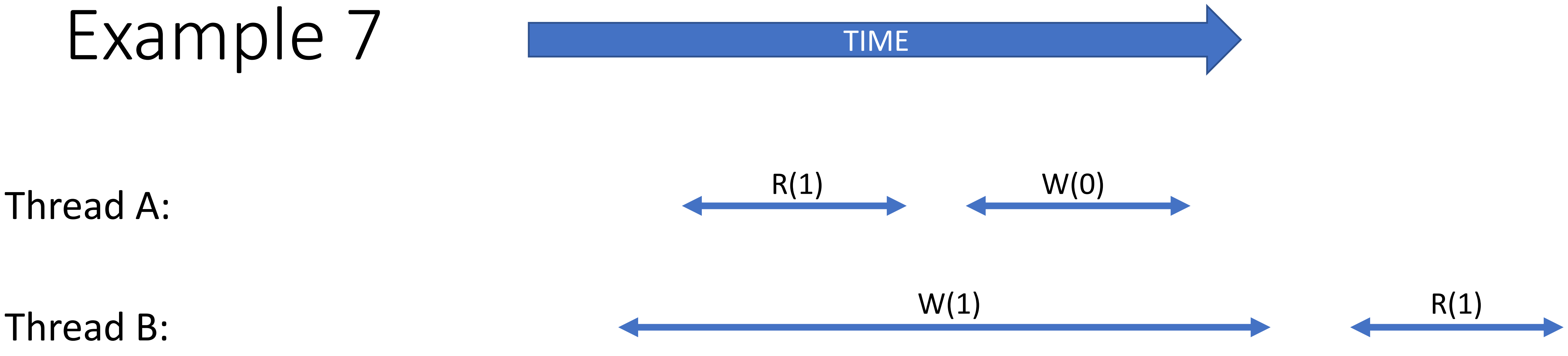
Example 6



“Consistent” or not?

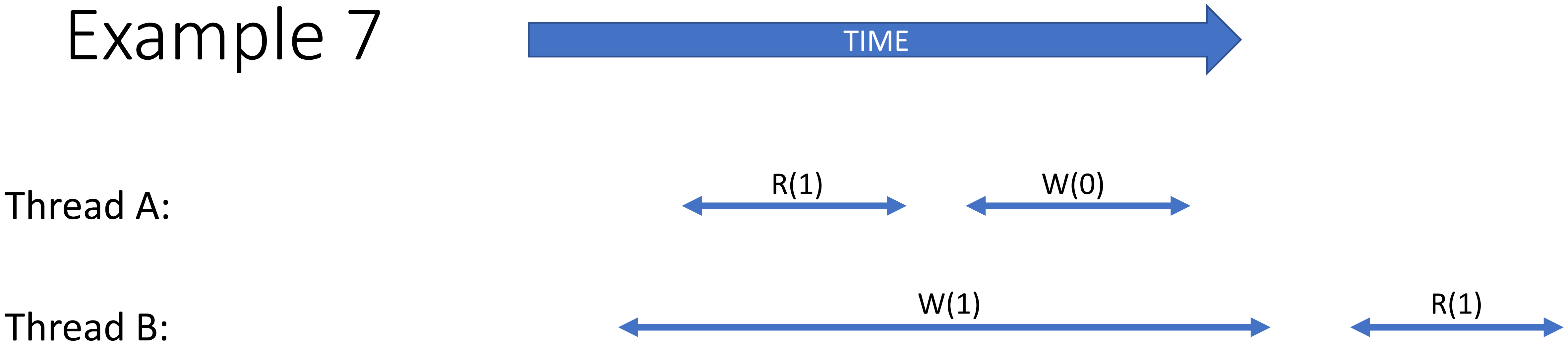
✓

Example 7



“Consistent” or not?

Example 7



TIME

Thread A: R(1) W(0)

Thread B: W(1) R(1)

“Consistent” or not?

✗

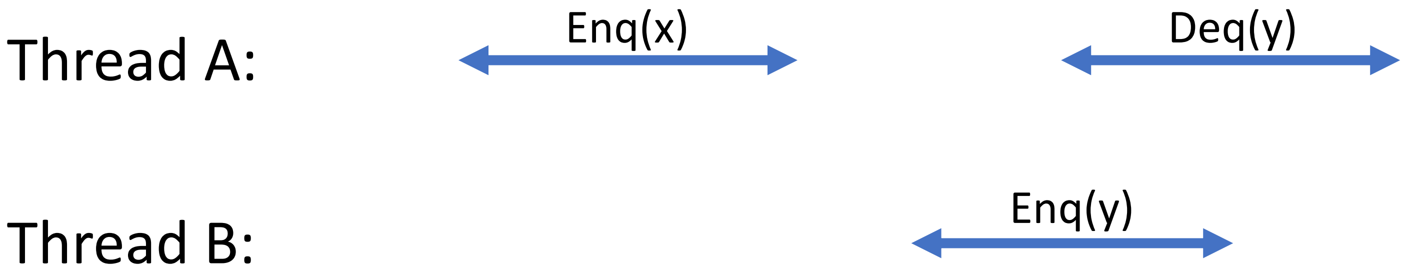
or maybe ✓

Linearizability (Herlihy and Wing 1990)

- Each operation appears to execute atomically (instantaneously) at some time between its invocation and completion
    - known as “linearization point”
- Implementation is linearizable iff for every behavior you can find a corresponding sequential behavior of linearization points

Sequential Consistency (Lamport 1979)

- The result of any execution is the same as if the operations of all processes were executed in some sequential order and the operations of each process appear in this sequence in the order specified by its program



Thread A:

Enq(x)

Deq(y)

Thread B:

Enq(y)

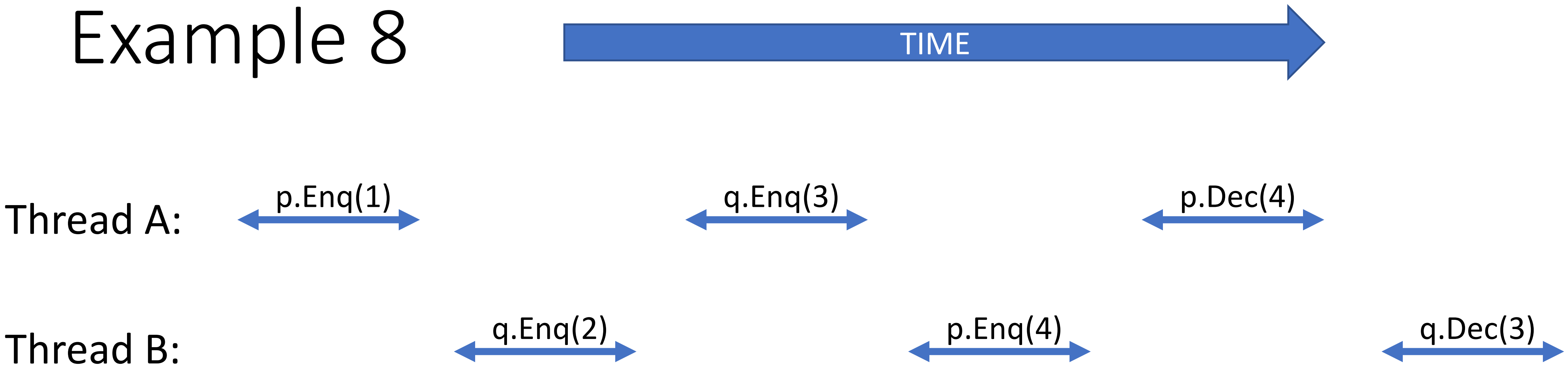
Example 2: sequentially consistent but not linearizable

Linearizability vs Sequential Consistency

- Linearizability implies Sequential Consistency (but not vice versa)
  - i.e., linearizability is a stronger consistency property than sequential consistency
  - sequential consistency allows more interleavings than linearizability

    →more concurrency, but harder to reason about
- Linearizability is a *local property*, but sequential consistency is not
  - Linearizability composes: a system of linearizable objects is linearizable
  - Vice versa: in a linearizable system each object is linearizable

Example 8



TIME

Thread A: p.Enq(1) q.Enq(3) p.Dec(4)

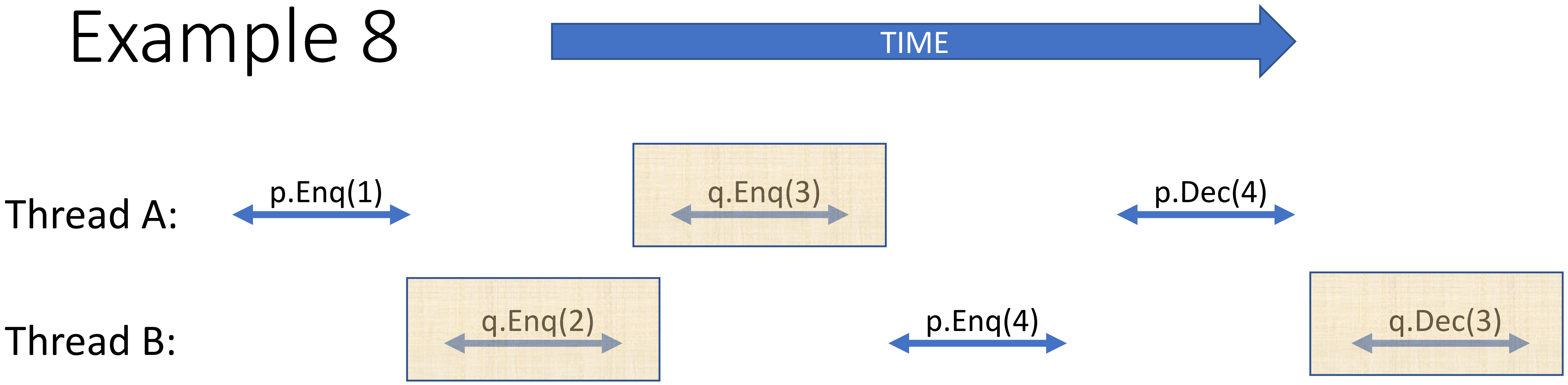
Thread B: q.Enq(2) p.Enq(4) q.Dec(3)

"Consistent" or not?

Linearizable?

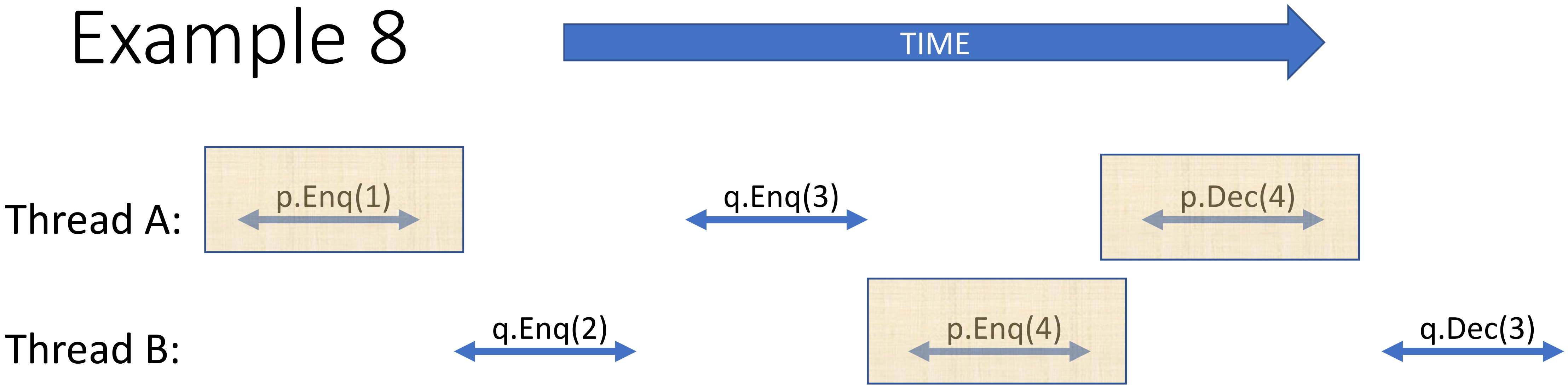
Sequentially consistent?

Example 8



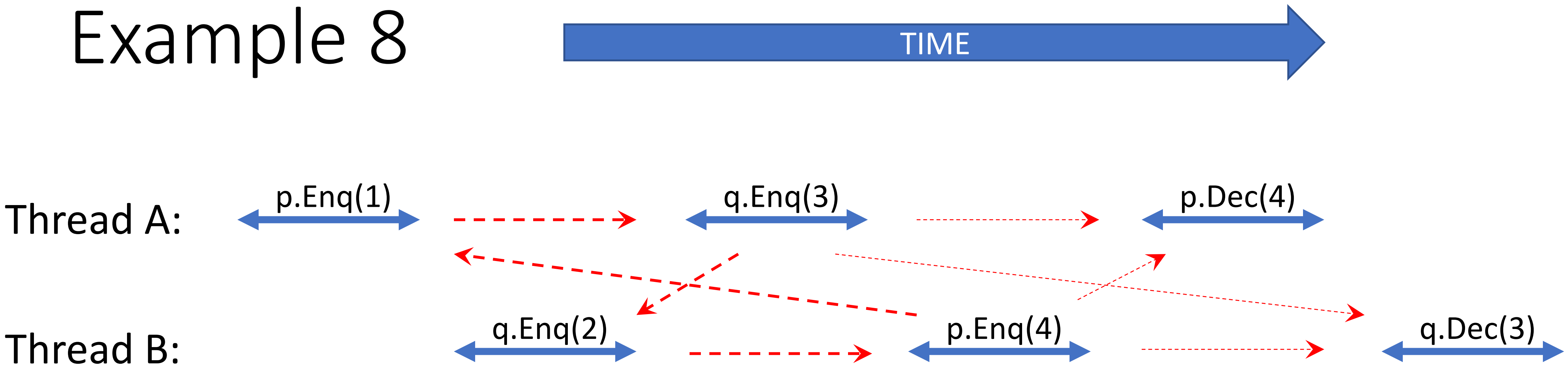
Operations on p are sequentially consistent

Example 8



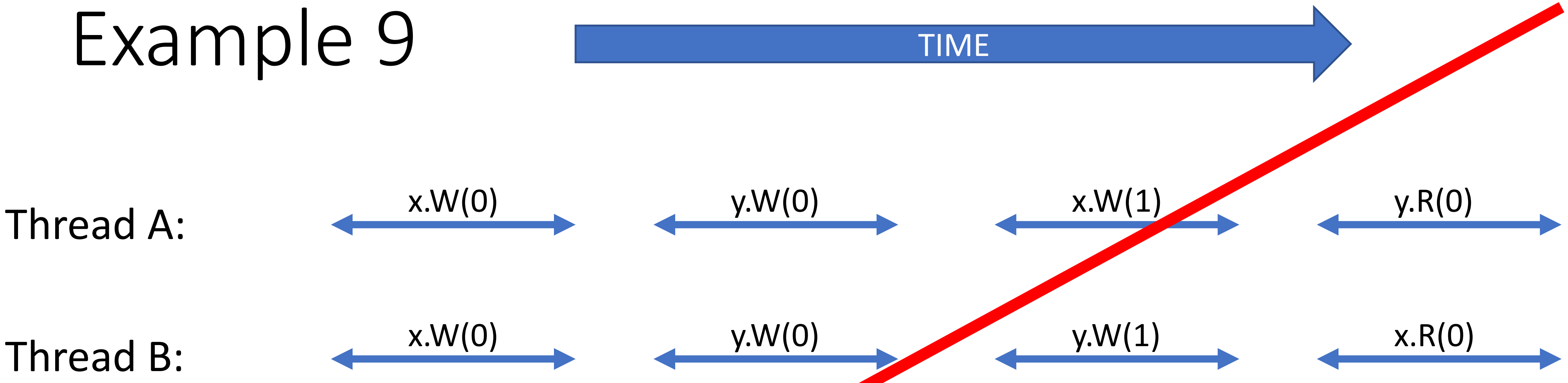
Operations on q are sequentially consistent

Example 8



But entire history is not sequentially consistent

Example 9



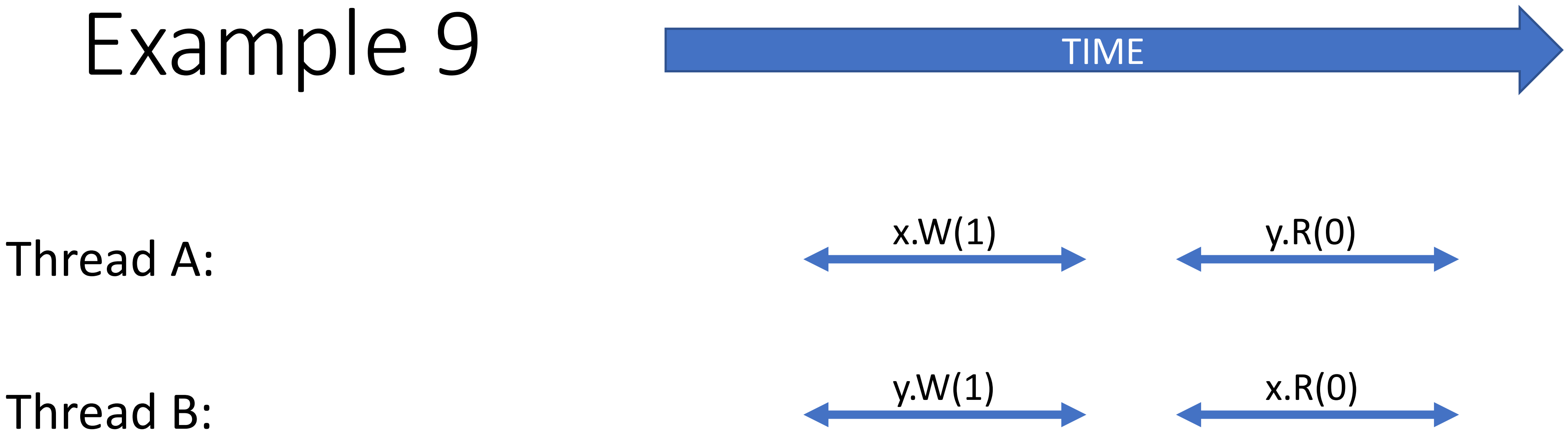
TIME

Thread A: x.W(0) y.W(0) x.W(1) y.R(0)

Thread B: x.W(0) y.W(0) y.W(1) x.R(0)

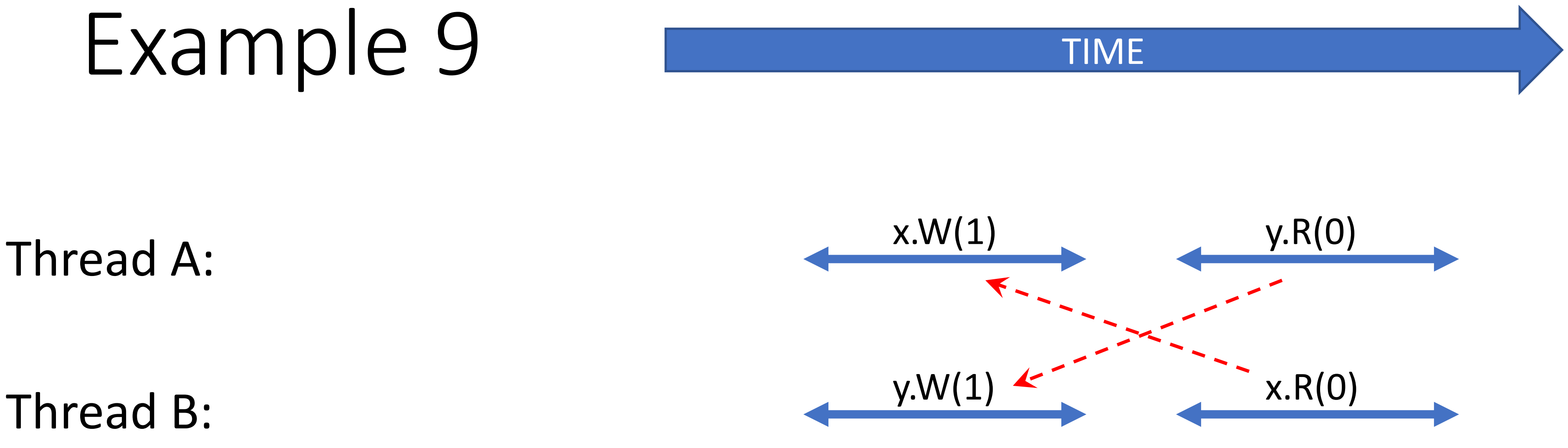
- Just Thread A: sequentially consistent
- Just Thread B: sequentially consistent
- Just location x: sequentially consistent
- Just location y: sequentially consistent
- Overall: not sequentially consistent

Example 9



- Just Thread A: sequentially consistent
- Just Thread B: sequentially consistent
- Just location x: sequentially consistent
- Just location y: sequentially consistent
- Overall: not sequentially consistent

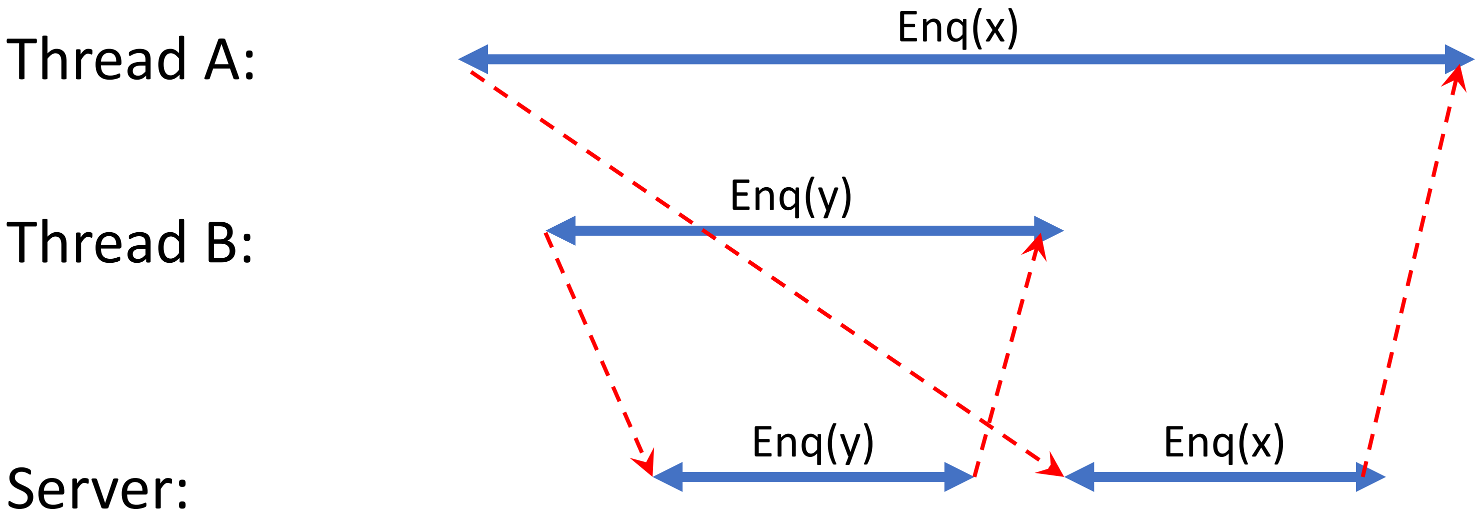
Example 9



- Just Thread A: sequentially consistent
- Just Thread B: sequentially consistent
- Just location x: sequentially consistent
- Just location y: sequentially consistent
- Overall: not sequentially consistent

A model of linearizability

- Each object implemented by a sequential server
- Communication is through sending requests and receiving responses



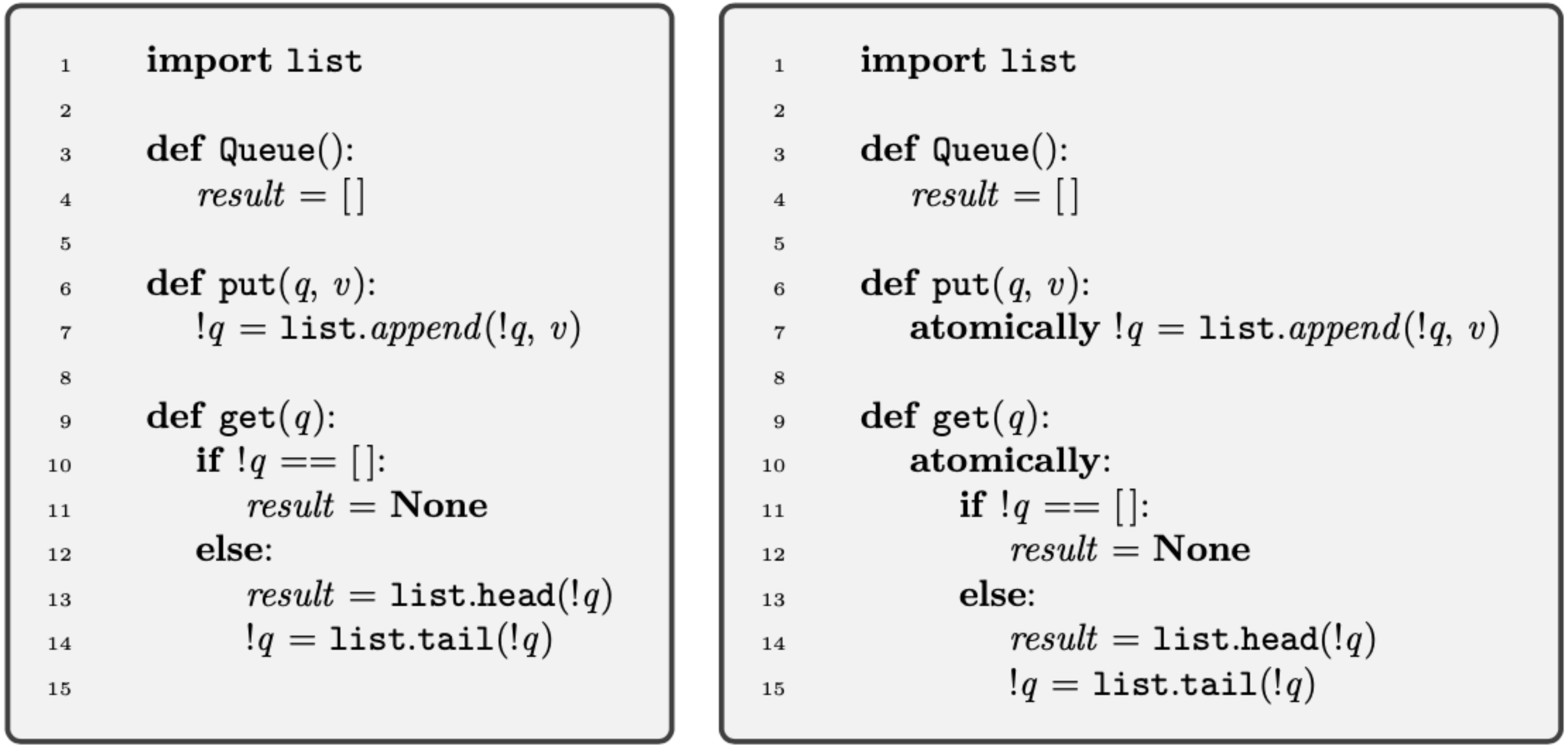
Verifying Linearizability

- In general, need to identify linearization points and show that they form a legal sequential history of operations

Building a Concurrent Queue in Harmony

- *q* = queue.Queue(): allocate a new queue
- queue.put(*q*, *v*): add *v* to the tail of queue *q*
- *v* = queue.get(*q*): returns None if *q* is empty or *v* if *v* was at the head of the queue

Specifying a concurrent queue



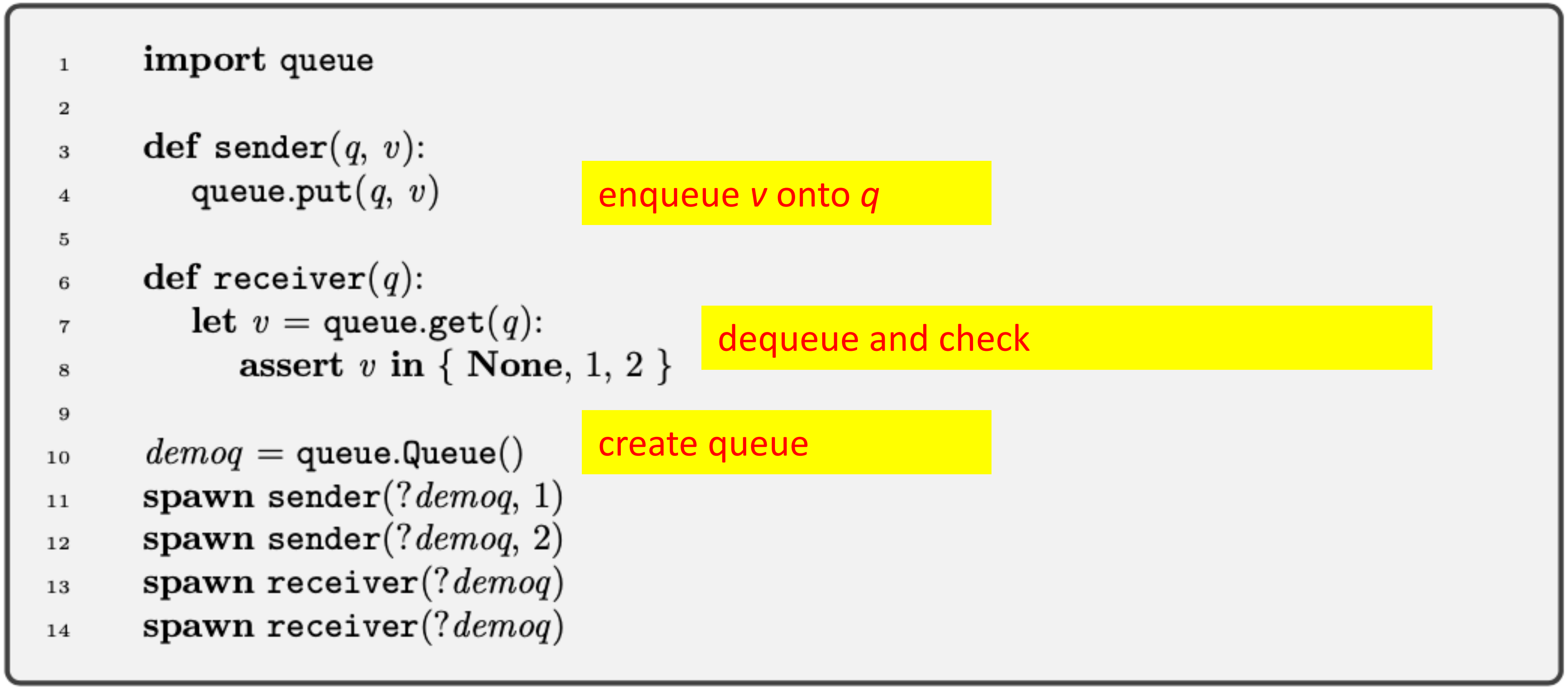
```
import list

def Queue():
    result = []

def put(q, v):
    !q = list.append(!q, v)

def get(q):
    if !q == []:
        result = None
    else:
        result = list.head(!q)
        !q = list.tail(!q)
```

(a) [code/queuespec.hny] Sequential

```
import list

def Queue():
    result = []

def put(q, v):
    atomically !q = list.append(!q, v)

def get(q):
    atomically:
        if !q == []:
            result = None
        else:
            result = list.head(!q)
            !q = list.tail(!q)
```

(b) [code/queue.hny] Concurrent

Example of using a queue



```
import queue

def sender(q, v):
    queue.put(q, v)

def receiver(q):
    let v = queue.get(q):
        assert v in { None, 1, 2 }

demoq = queue.Queue()
spawn sender(?demoq, 1)
spawn sender(?demoq, 2)
spawn receiver(?demoq)
spawn receiver(?demoq)
```

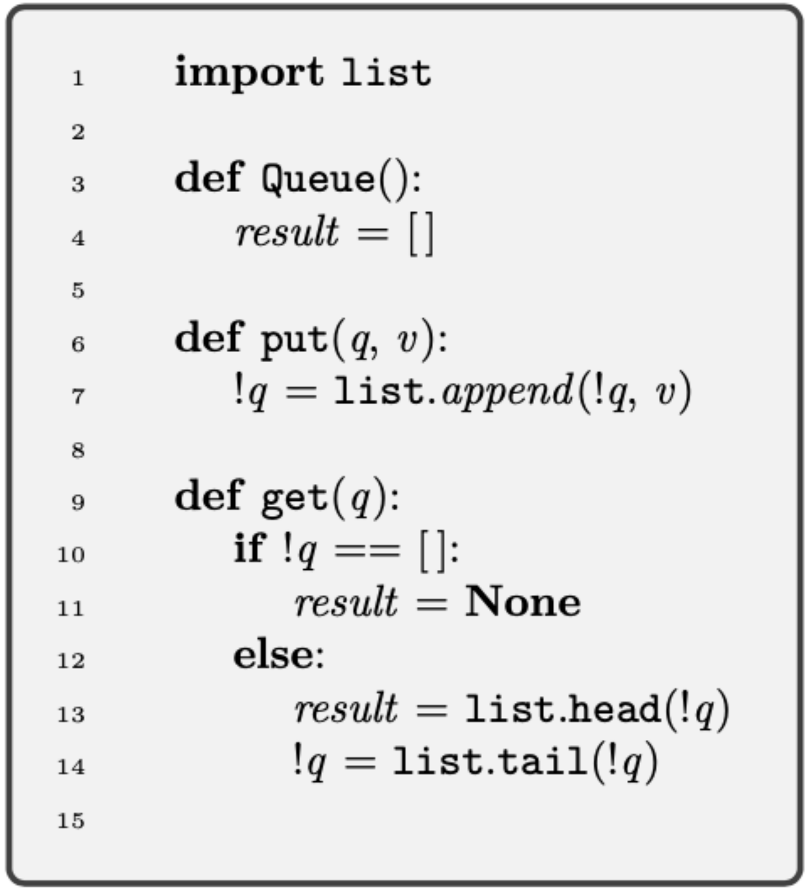
enqueue *v* onto *q*

dequeue and check

create queue

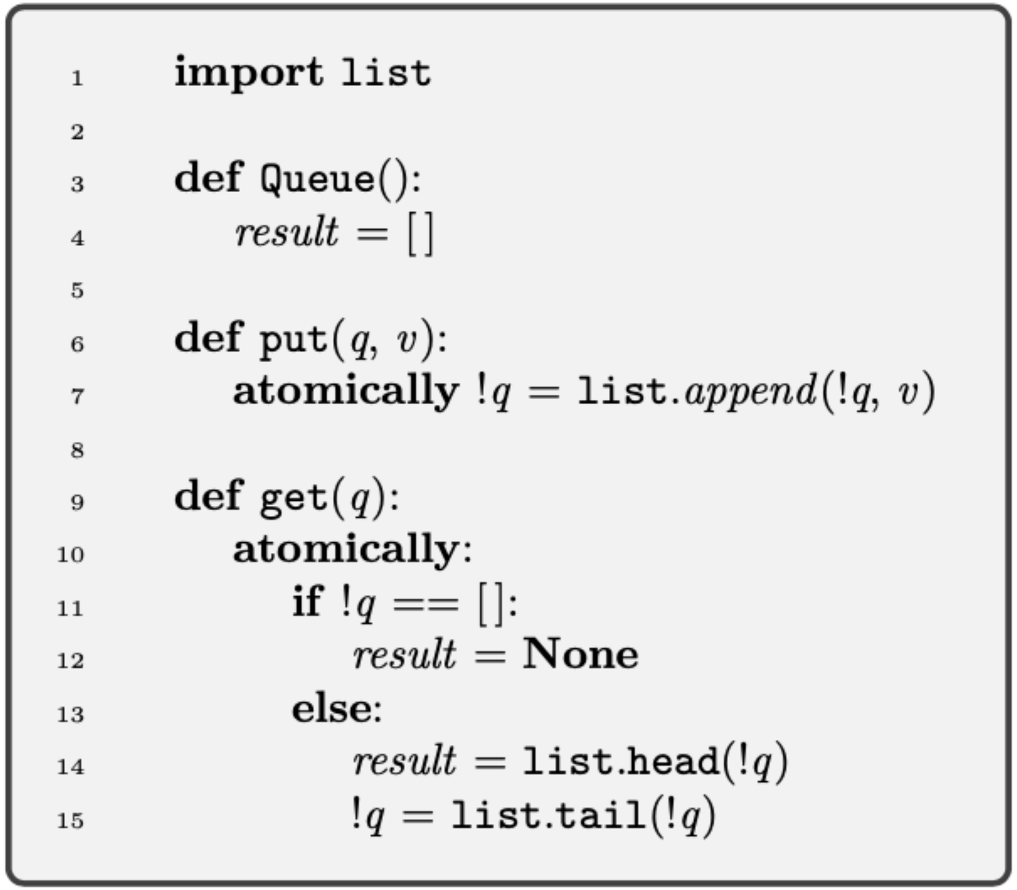
Figure 11.2: [code/queuedemo.hny] Using a concurrent queue

Specifying a concurrent queue



```
import list

def Queue():
    result = []

def put(q, v):
    !q = list.append(!q, v)

def get(q):
    if !q == []:
        result = None
    else:
        result = list.head(!q)
        !q = list.tail(!q)

```

(a) [code/queuespec.hny] Sequential

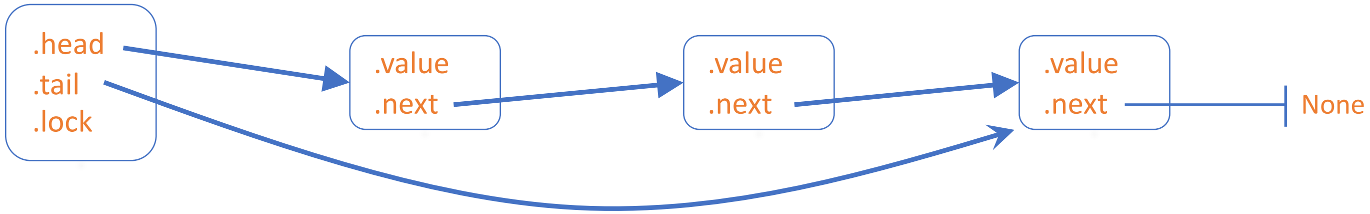


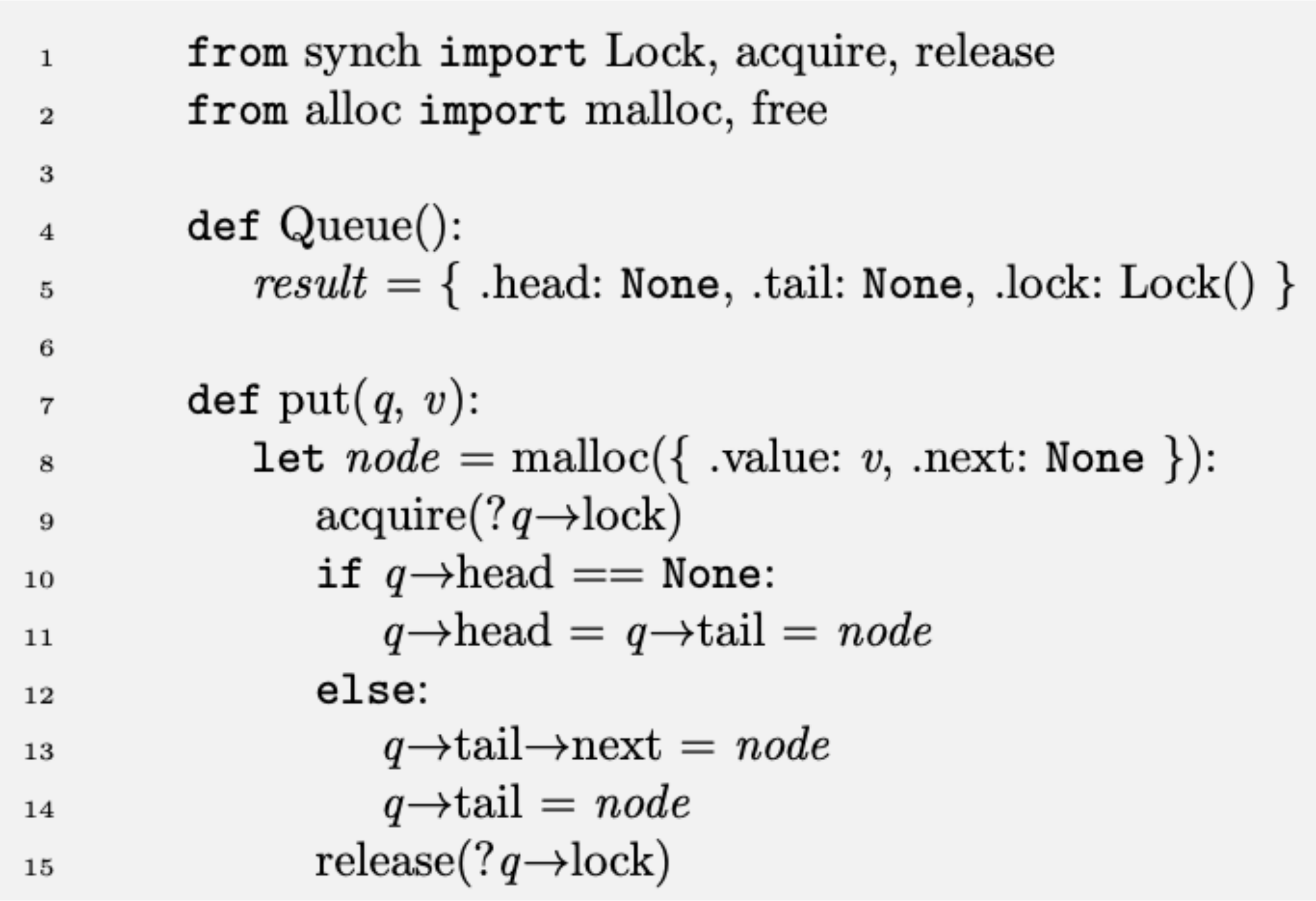
```
import list

def Queue():
    result = []

def put(q, v):
    atomically !q = list.append(!q, v)

def get(q):
    atomically:
        if !q == []:
            result = None
        else:
            result = list.head(!q)
            !q = list.tail(!q)
```

(b) [code/queue.hny] Concurrent

not a good implementation because (a) operations are O(n) and (b) compiler cannot generate code for “atomically”

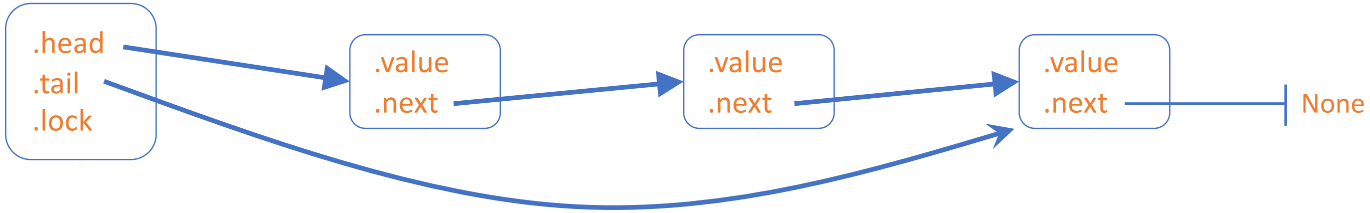
Queue implementation

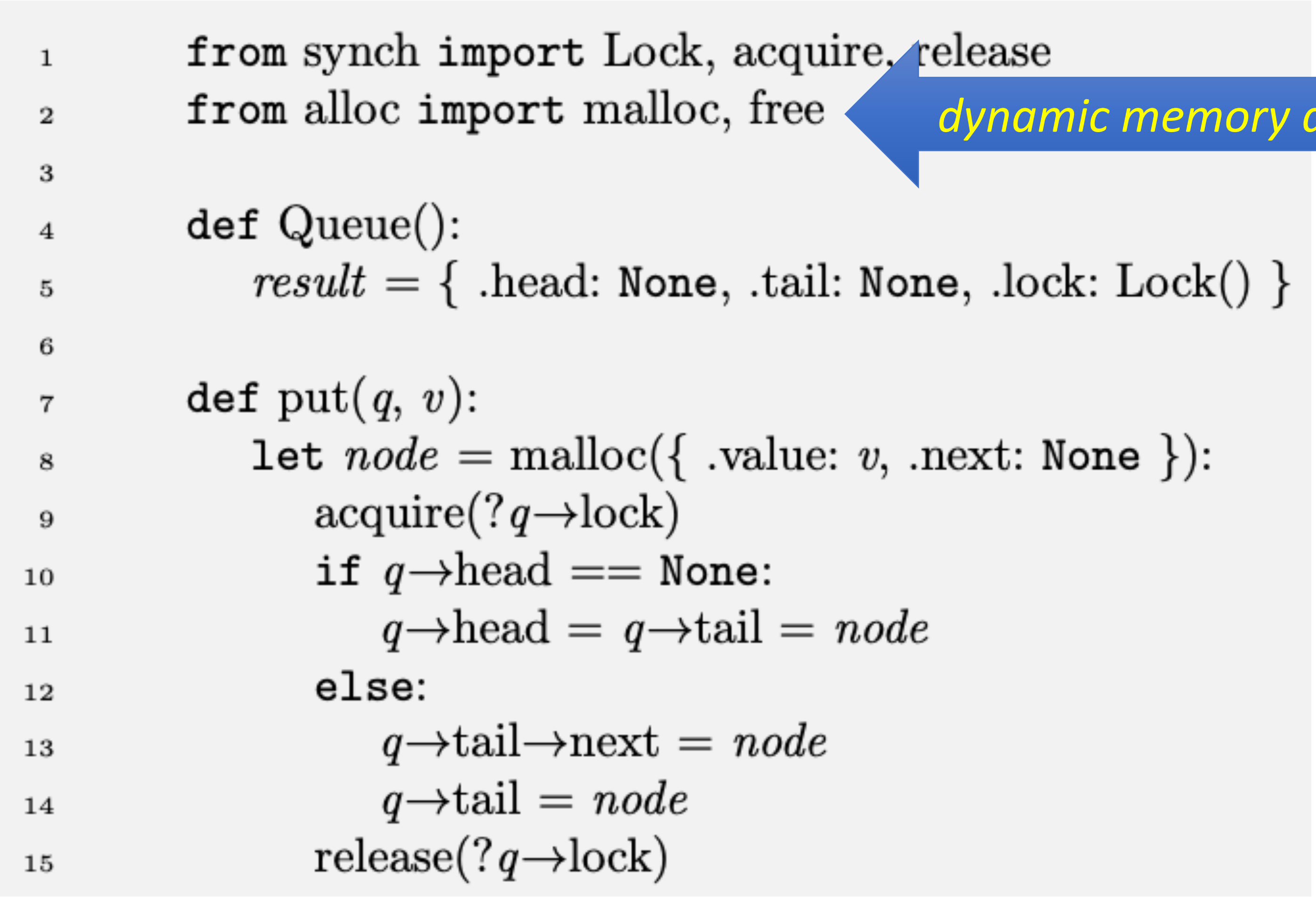




```
from synch import Lock, acquire, release
from alloc import malloc, free

def Queue():
    result = { .head: None, .tail: None, .lock: Lock() }

def put(q, v):
    let node = malloc({ .value: v, .next: None }):
        acquire(?q→lock)
        if q→head == None:
            q→head = q→tail = node
        else:
            q→tail→next = node
            q→tail = node
        release(?q→lock)
```

Queue implementation

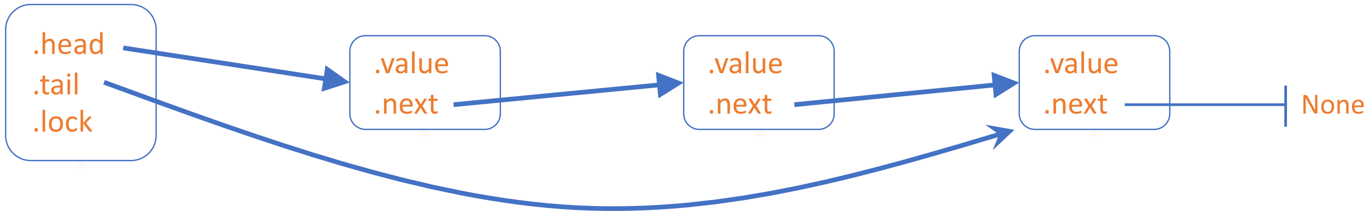


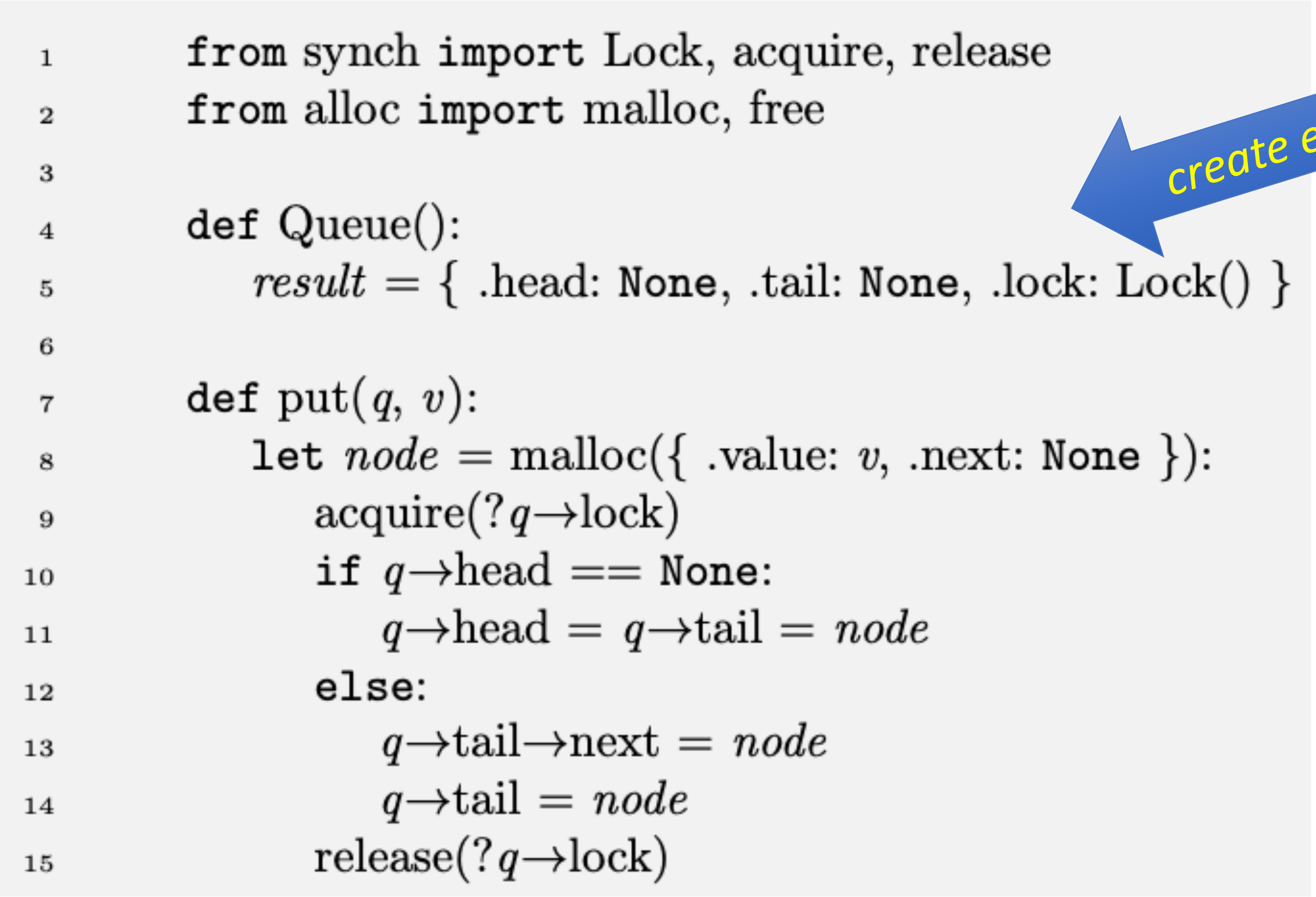


```
from synch import Lock, acquire, release
from alloc import malloc, free

def Queue():
    result = { .head: None, .tail: None, .lock: Lock() }

def put(q, v):
    let node = malloc({ .value: v, .next: None }):
        acquire(?q→lock)
        if q→head == None:
            q→head = q→tail = node
        else:
            q→tail→next = node
            q→tail = node
        release(?q→lock)
```

dynamic memory allocation

Queue implementation

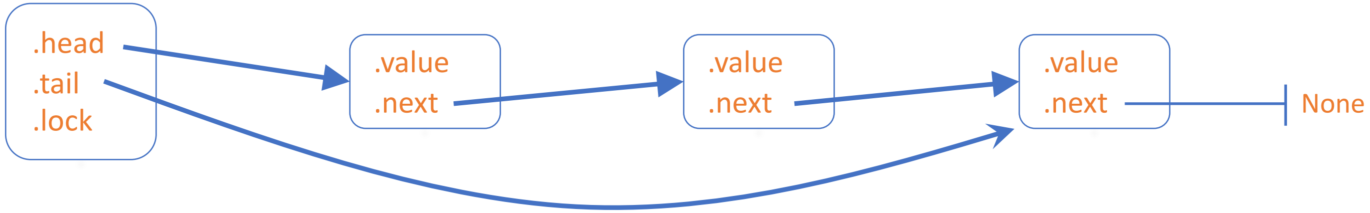


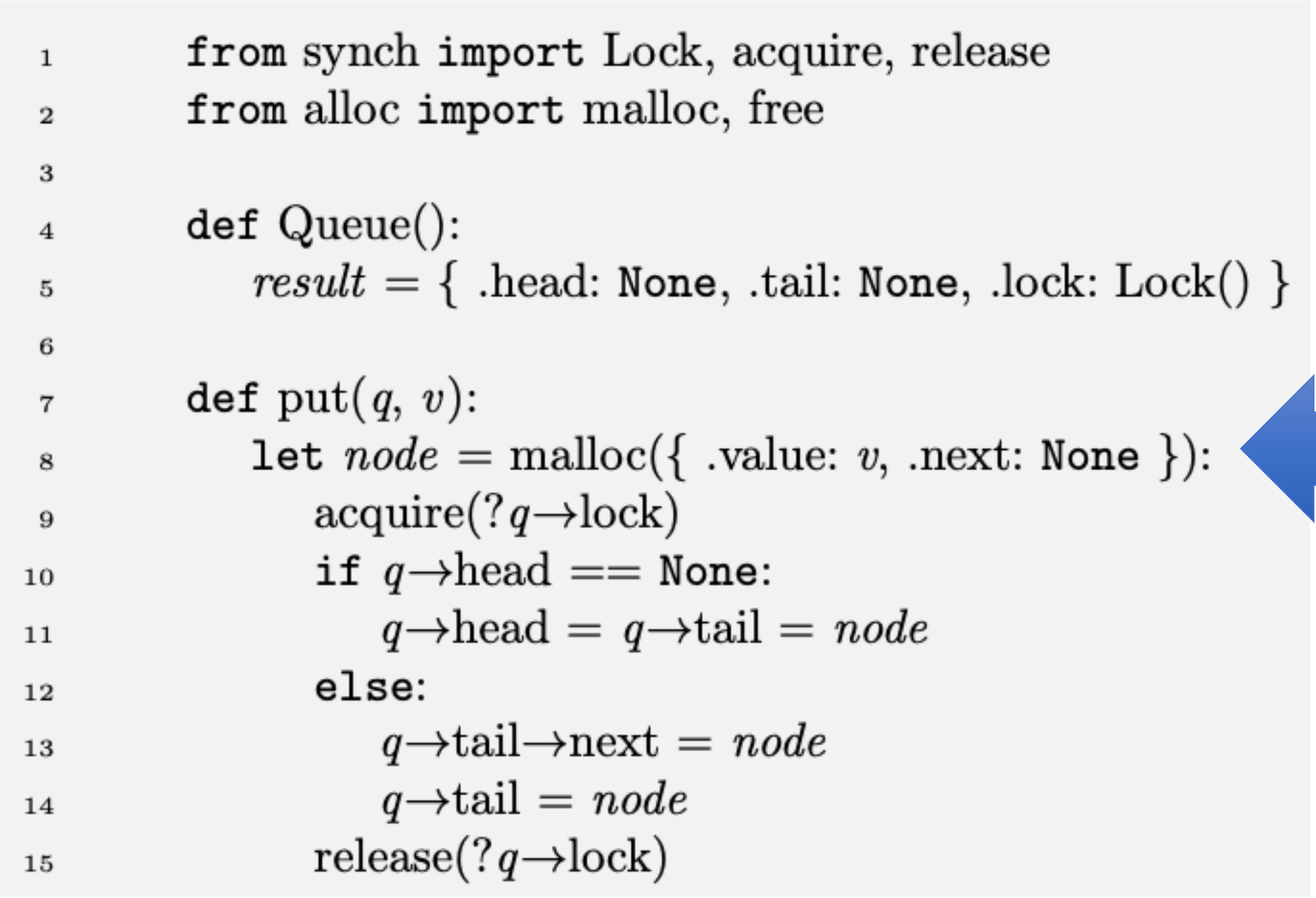


```
from synch import Lock, acquire, release
from alloc import malloc, free

def Queue():
    result = { .head: None, .tail: None, .lock: Lock() }

def put(q, v):
    let node = malloc({ .value: v, .next: None }):
        acquire(?q→lock)
        if q→head == None:
            q→head = q→tail = node
        else:
            q→tail→next = node
            q→tail = node
        release(?q→lock)
```

create empty queue

Queue implementation

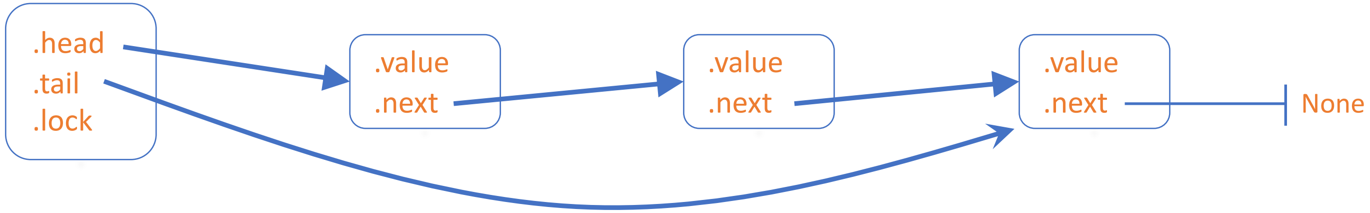


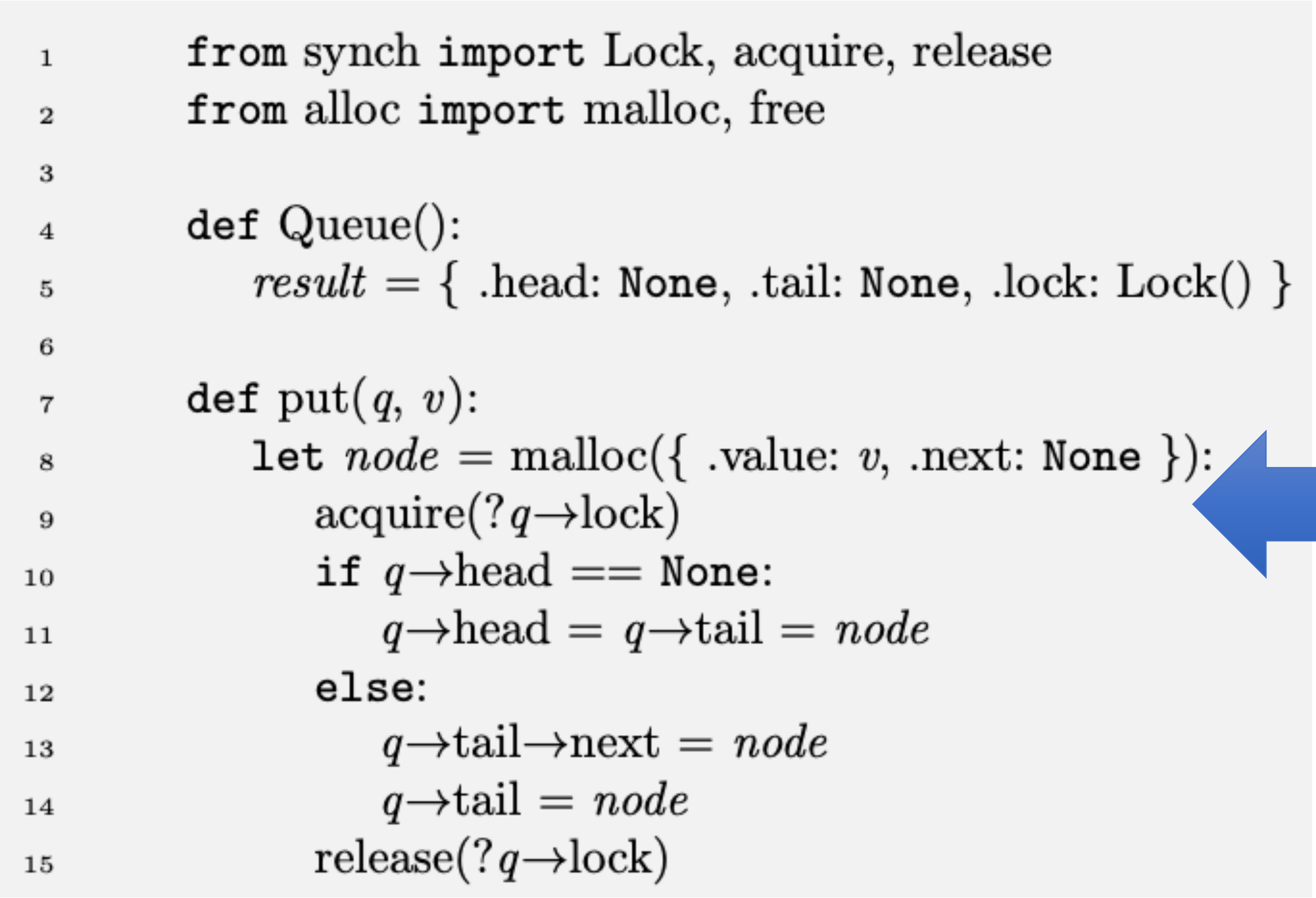


```
from synch import Lock, acquire, release
from alloc import malloc, free

def Queue():
    result = { .head: None, .tail: None, .lock: Lock() }

def put(q, v):
    let node = malloc({ .value: v, .next: None }):
        acquire(?q→lock)
        if q→head == None:
            q→head = q→tail = node
        else:
            q→tail→next = node
            q→tail = node
        release(?q→lock)
```

allocate node

Queue implementation

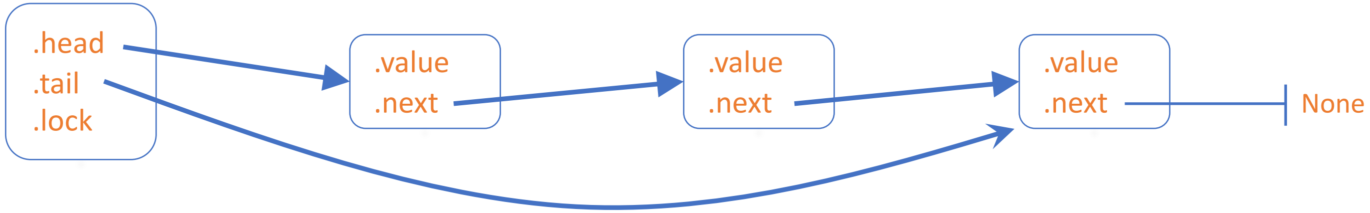


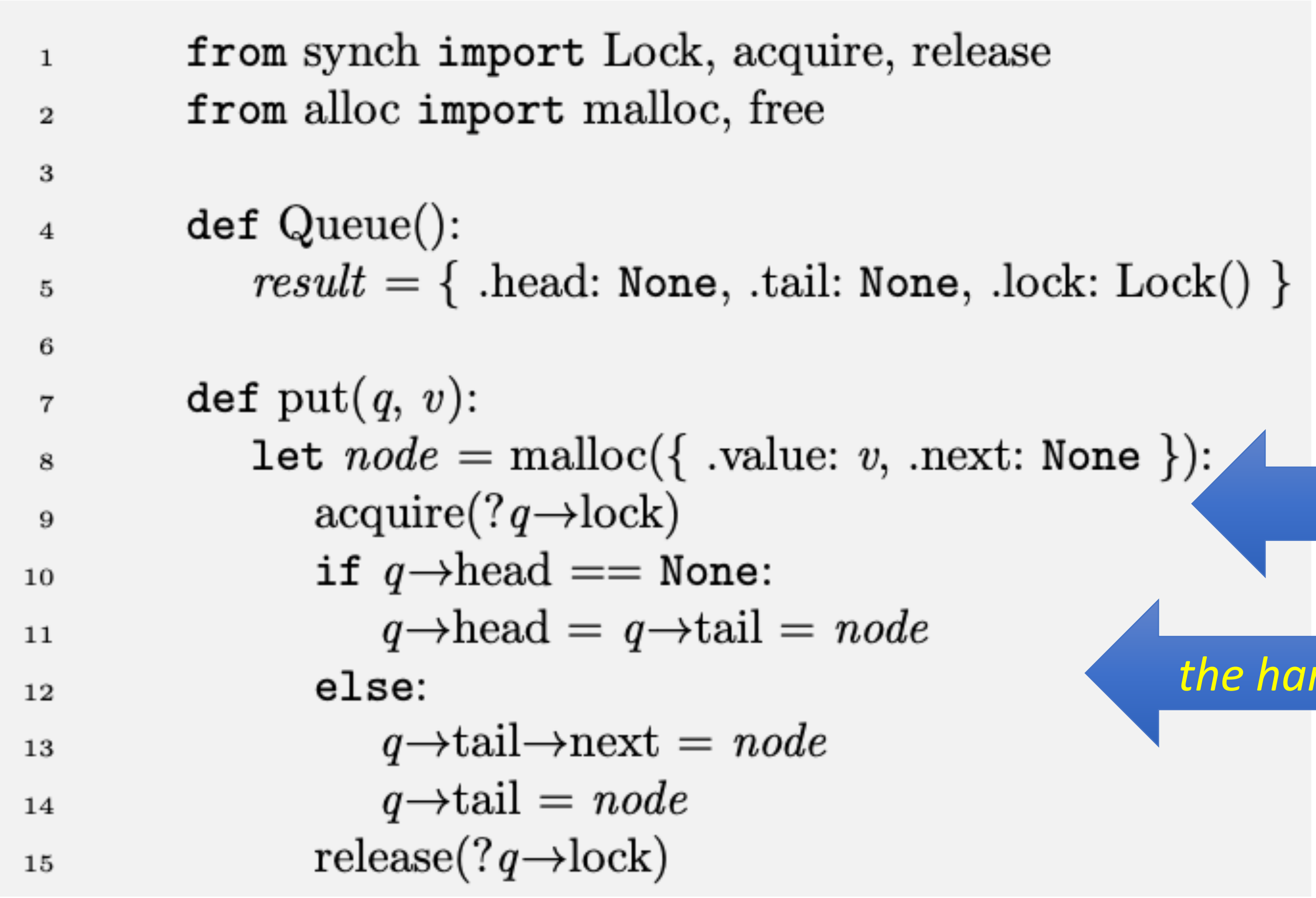


```
from synch import Lock, acquire, release
from alloc import malloc, free

def Queue():
    result = { .head: None, .tail: None, .lock: Lock() }

def put(q, v):
    let node = malloc({ .value: v, .next: None }):
        acquire(?q→lock)
        if q→head == None:
            q→head = q→tail = node
        else:
            q→tail→next = node
            q→tail = node
        release(?q→lock)
```

grab lock

Queue implementation



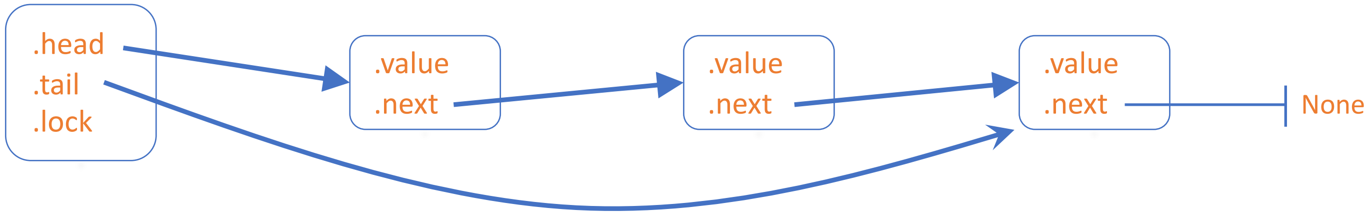


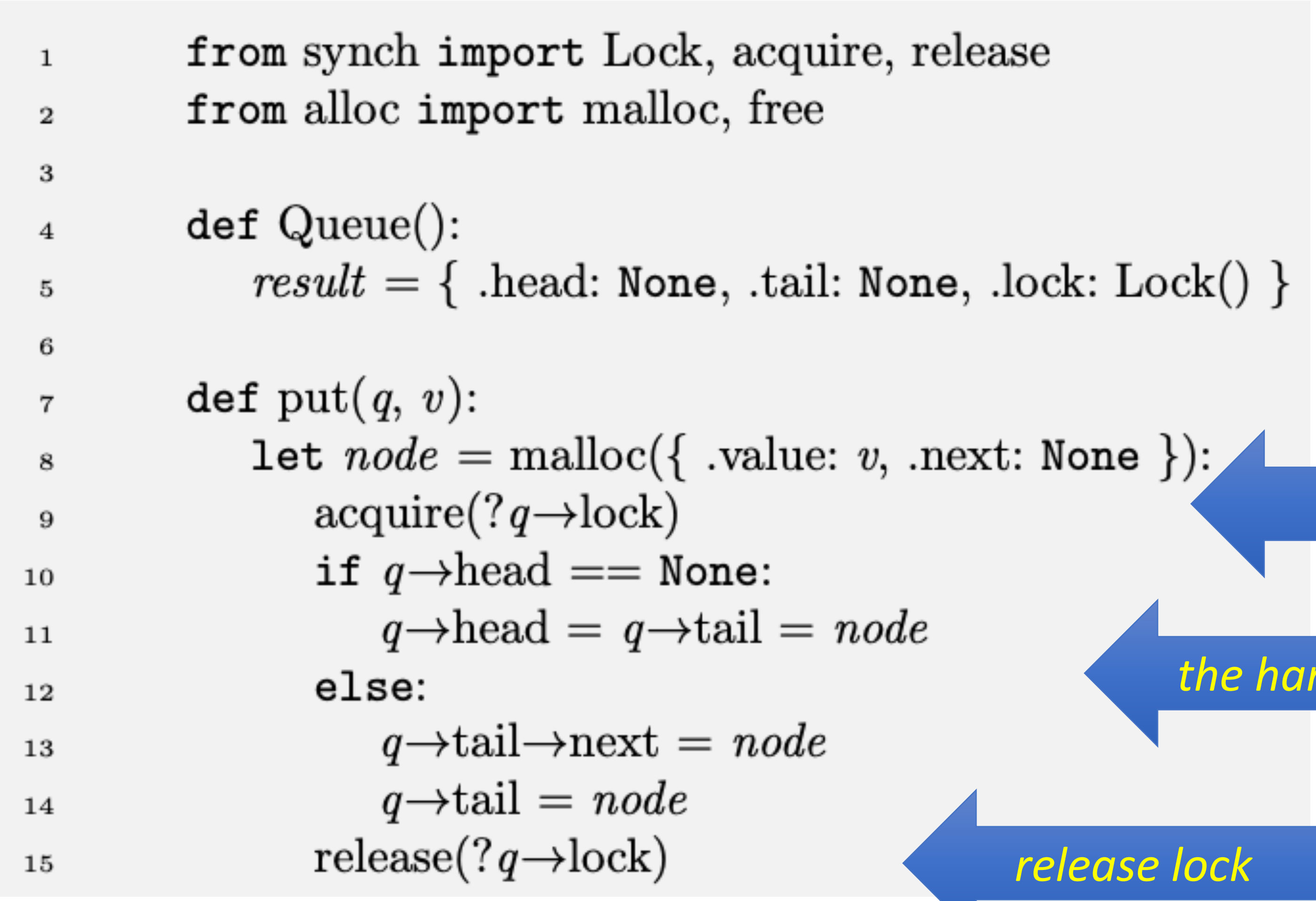
```
from synch import Lock, acquire, release
from alloc import malloc, free

def Queue():
    result = { .head: None, .tail: None, .lock: Lock() }

def put(q, v):
    let node = malloc({ .value: v, .next: None }):
        acquire(?q→lock)
        if q→head == None:
            q→head = q→tail = node
        else:
            q→tail→next = node
            q→tail = node
        release(?q→lock)
```

grab lock

the hard stuff

Queue implementation





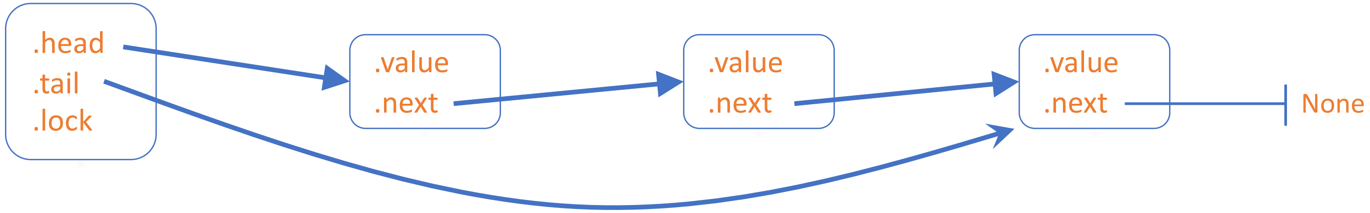
```
from synch import Lock, acquire, release
from alloc import malloc, free

def Queue():
    result = { .head: None, .tail: None, .lock: Lock() }

def put(q, v):
    let node = malloc({ .value: v, .next: None }):
        acquire(?q→lock)
        if q→head == None:
            q→head = q→tail = node
        else:
            q→tail→next = node
            q→tail = node
        release(?q→lock)
```

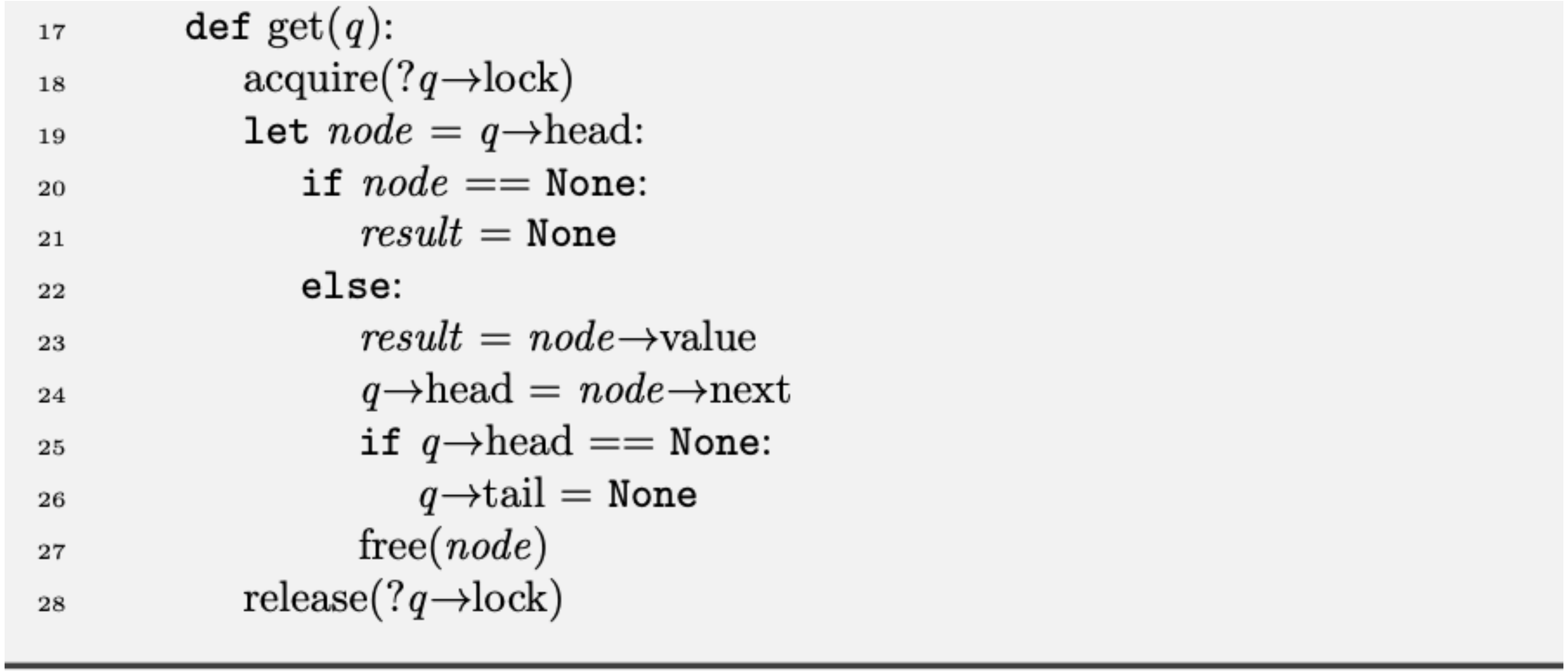
grab lock

the hard stuff

release lock

Queue implementation





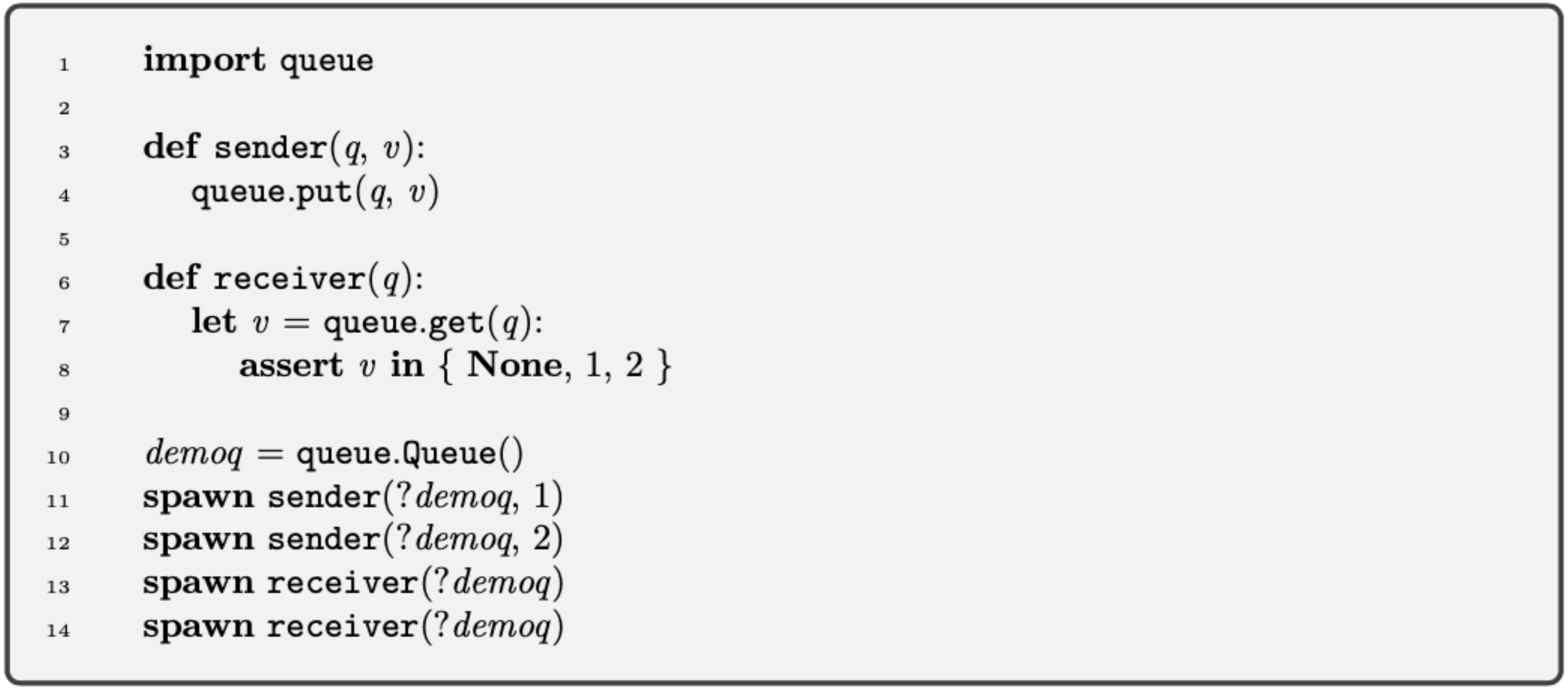
```
def get(q):
    acquire(?q→lock)
    let node = q→head:
        if node == None:
            result = None
        else:
            result = node→value
            q→head = node→next
            if q→head == None:
                q→tail = None
            free(node)
    release(?q→lock)
```

Figure 10.2: [code/queue.hny]A basic concurrent queue data structure.

How important are concurrent queues?

- Answer: all important
  - any resource that needs scheduling
    - CPU run queue
    - disk, network, printer waiting queue
    - lock waiting queue
  - inter-process communication
    - Posix pipes:
      - cat file | tr a-z A-Z | grep RVR
  - actor-based concurrency
  - …

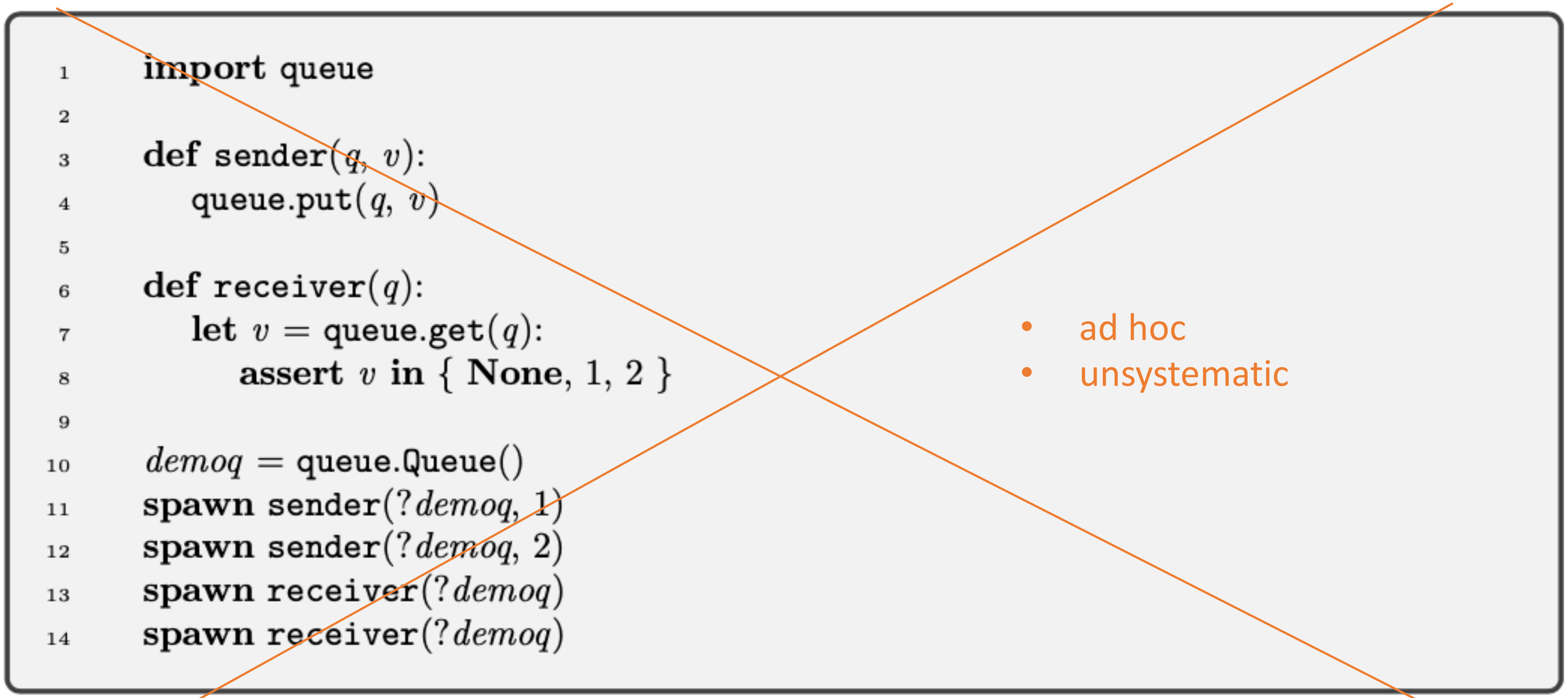
Testing a Concurrent Queue?



```
import queue

def sender(q, v):
    queue.put(q, v)

def receiver(q):
    let v = queue.get(q):
        assert v in { None, 1, 2 }

demoq = queue.Queue()
spawn sender(?demoq, 1)
spawn sender(?demoq, 2)
spawn receiver(?demoq)
spawn receiver(?demoq)
```

Figure 11.2: [code/queuedemo.hny] Using a concurrent queue

Testing a Concurrent Queue?



```
import queue

def sender(q, v):
    queue.put(q, v)

def receiver(q):
    let v = queue.get(q):
        assert v in { None, 1, 2 }

demoq = queue.Queue()
spawn sender(?demoq, 1)
spawn sender(?demoq, 2)
spawn receiver(?demoq)
spawn receiver(?demoq)
```

- ad hoc
- unsystematic

Figure 11.2: [code/queuedemo.hny] Using a concurrent queue

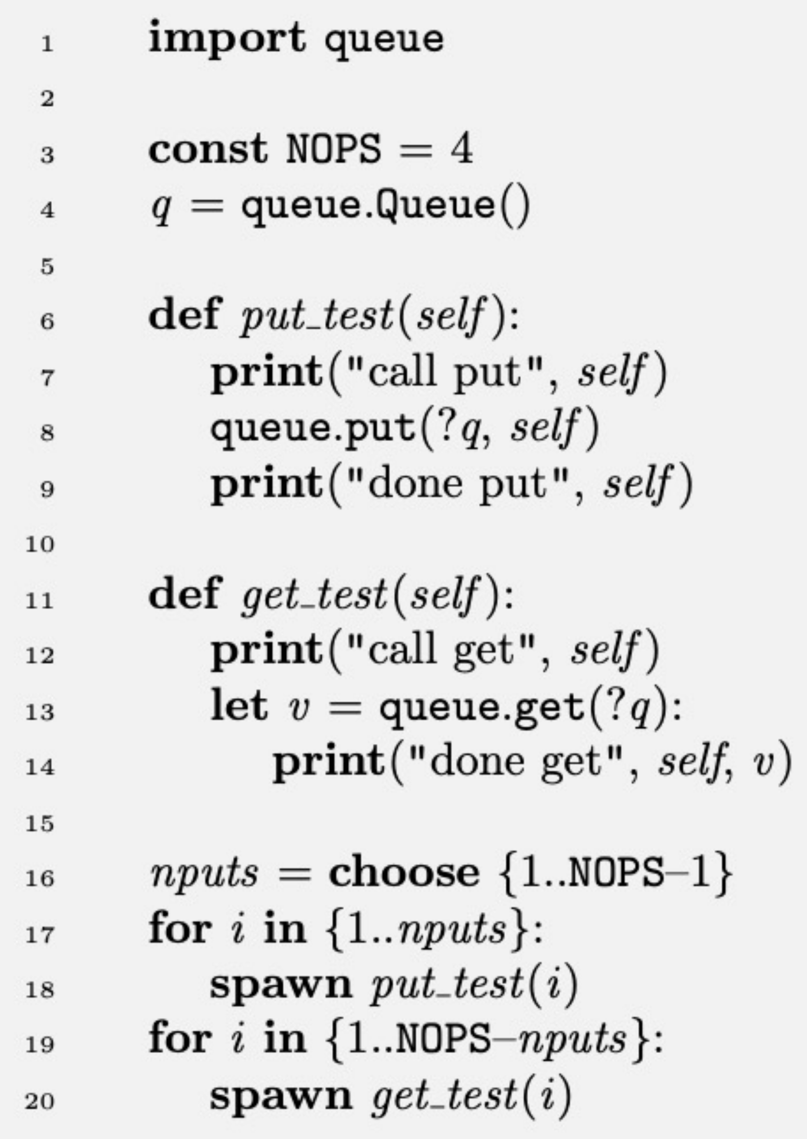
Systematic Testing

- Sequential case
  - try all “sequences” of 1 operation
    - put or get
  - try all sequences of 2 operations
    - put+put, put+get, get+put, get+get, …
  - try all sequences of 3 operations
  - …
- How do you know if a sequence is correct?
  - compare “behaviors” of running test against implementation with running test against the sequential specification

Systematic Testing

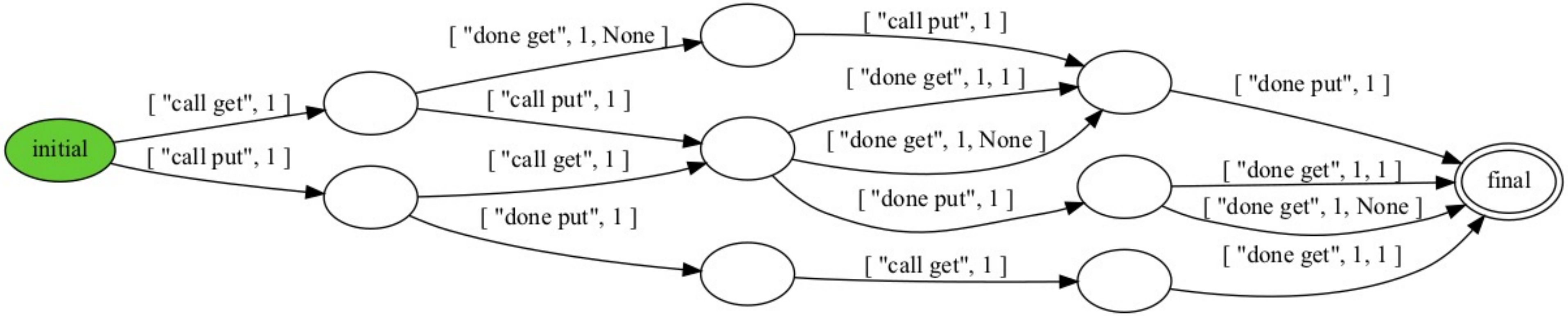
- Concurrent case
  - try all “interleavings” of 1 operation
  - try all interleavings of 2 operations
  - try all interleavings of 3 operations
  - …
- How do you know if a sequence is correct?
  - compare “behaviors” of running test against concurrent implementation with running test against the concurrent specification

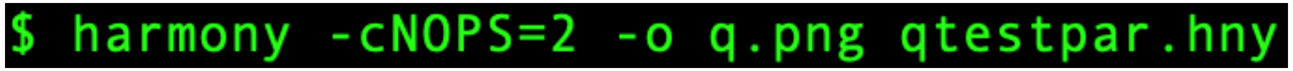
Queue test program



```
import queue

const NOPS = 4
q = queue.Queue()

def put_test(self):
    print("call put", self)
    queue.put(?q, self)
    print("done put", self)

def get_test(self):
    print("call get", self)
    let v = queue.get(?q):
        print("done get", self, v)

nputs = choose {1..NOPS-1}
for i in {1..nputs}:
    spawn put_test(i)
for i in {1..NOPS-nputs}:
    spawn get_test(i)
```

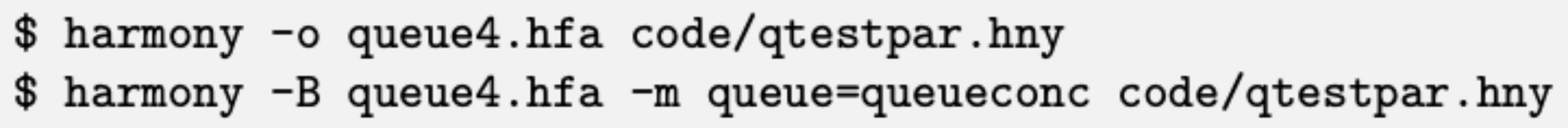
Behavior (NOPS=2: 1 get, 1 put)





```
$ harmony -cNOPS=2 -o q.png qtestpar.hny
```

Testing: comparing behaviors



```
$ harmony -o queue4.hfa code/qtestpar.hny
$ harmony -B queue4.hfa -m queue=queueconc code/qtestpar.hny
```

- The first command outputs the behavior of running the test program against the specification in file queue4.hfa
- The second command runs the test program against the implementation and checks if its behavior matches that stored in queue4.hfa

How about real memory?

- Registers: atomic (linearizable)
- Memory: not even sequentially consistent
    - write operations are buffered
    - processors, and even compilers, re-order operations in complex ways
        - or even remove operations that are deemed unnecessary but may not be
        - not usually a problem in sequential programs
    - big reads/big writes may be split across multiple instructions
        - i.e., 64-bit read/write on a 32-bit architecture
    - Note: Peterson's algorithm requires sequential consistency
- Modern processor has "memory barriers" or "fence" instructions to force data to memory
    - e.g. **mfence** instruction on x86

In high-level languages

- In Java, you can specify that a variable is “volatile”
    - Adds a memory barrier after each store
    - Inhibits compiler optimizations
- C++ offers various types of ”atomic variables” with various consistency guarantees
    - The “volatile” tag inhibits optimizations but does not add a memory barrier

What is Eventual Consistency?

- It is not really consistency at all
- Think instead of anti-entropy protocols

Further reading

- “Art of Multiprocessor Programming” by Maurice Herlihy et al.
- “On Concurrent Programming” by Fred B. Schneider

Atomic Transactions

- From database community:
  - an atomic transaction is a group of operations
    - e.g.:

```
begin_transaction
    if x < y:
        x := y
end_transaction
```

  - ACID properties
    - Atomicity: all or nothing (transactions may commit or abort)
    - Consistency: satisfies application-level invariants
    - Isolation: appears as if transactions are executed serially
    - Durable: effects of successful transactions are permanent

Transactions: commit or abort

- If a transaction commits then all its actions are permanent
- If a transaction aborts then there are no (visible) actions
- Typical usage: try until successful

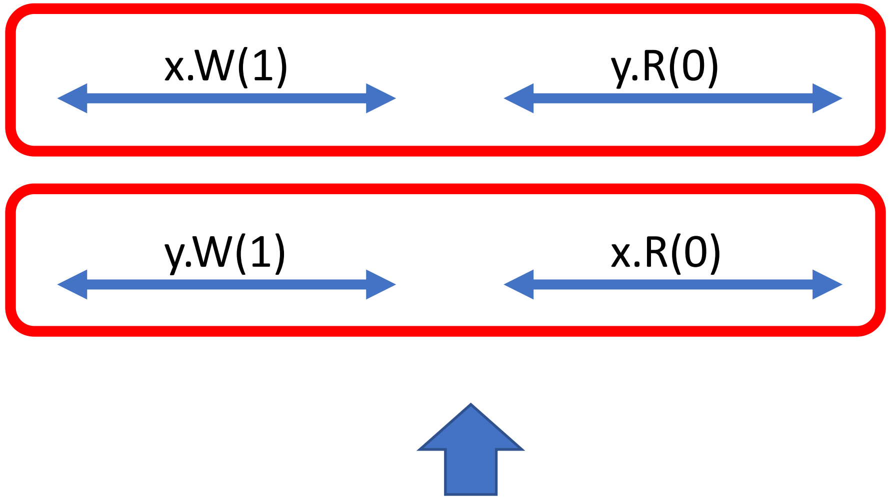
```
do
   begin_transaction()
         …
         …
while (end_transaction() == ABORT)
```

Serializability

- Serializability: (successful) transactions appear to execute sequentially
  - i.e., *isolation*
- Strict serializability: Consistent with real-time order
  - if transaction B starts after transaction A finishes, then B must be ordered after A
- Linearizability is a special case of strict serializability
  - transactions with a single operation each

Strict serializability is not local!

(Similar to Example 9)



Thread A:

x.W(1)

y.R(0)

Thread B:

y.W(1)

x.R(0)

Neither can go first; one must abort

Concurrency Control

- Ways to guarantee serializability: isolation between transactions
  - Pessimistic: grab read/write locks as the transaction is progressing
    - 2 phase locking
      - don’t release locks until end of transaction
      - acquire locks in some global order to prevent deadlock
  - Optimistic: keep track of read/write sets and check for conflicts at the end
    - abort transaction if its write set intersects with the read set or write set of a concurrent committed transaction or its read set intersects with the write set of a concurrent committed transaction

Atomic Commitment: 2 phase commit

- Actors: one coordinator and two or more participants
- Protocol:

  1A: coordinator broadcasts PREPARE to all participants

  1B: participants reply with either YES or NO

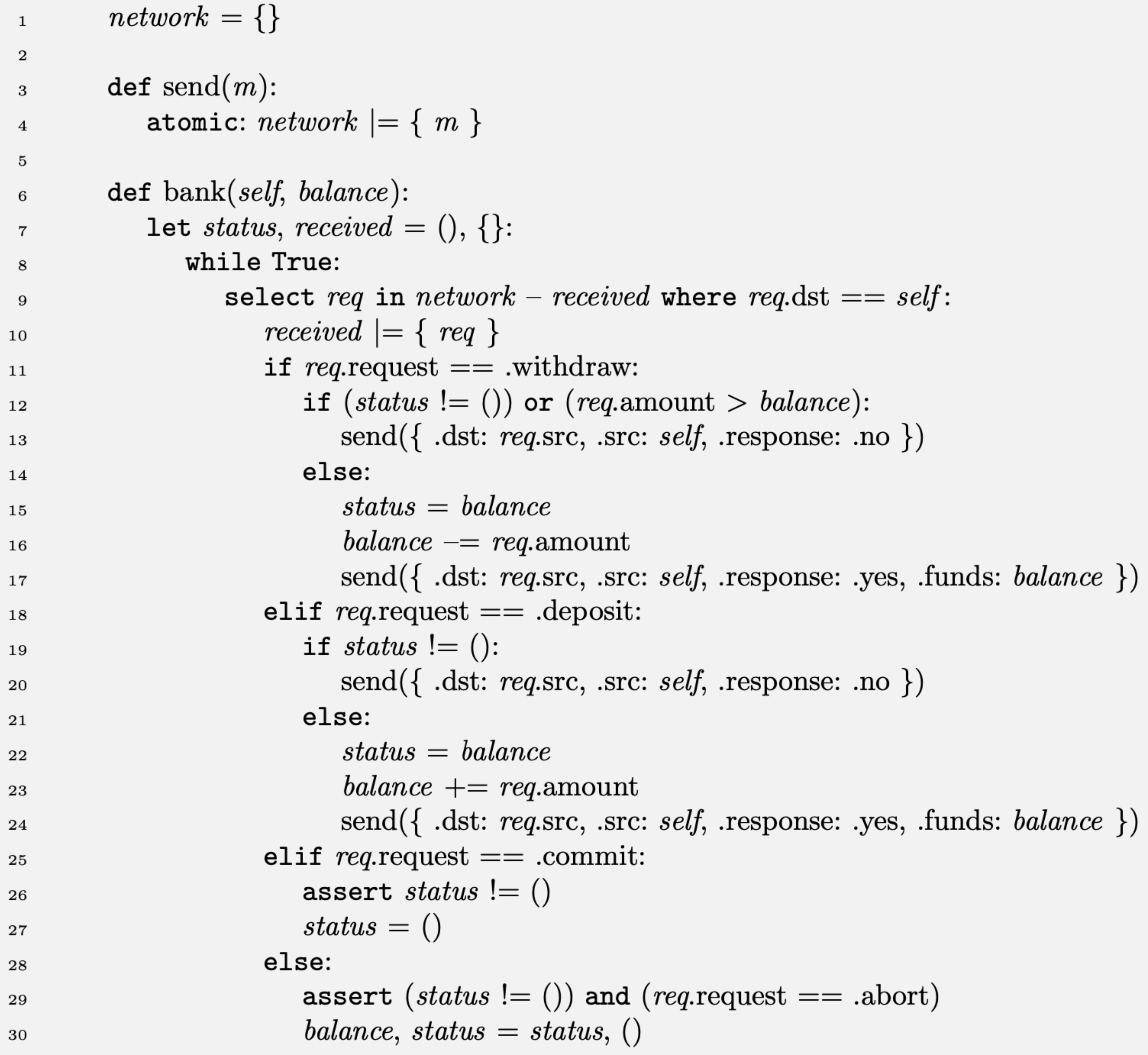
  if YES, participant promises to remain ready to move forward

  2A: coordinator broadcasts COMMIT *only if* all participants responded YES

  if some participant responds with NO, or does not respond,
  then coordinator broadcasts ABORT

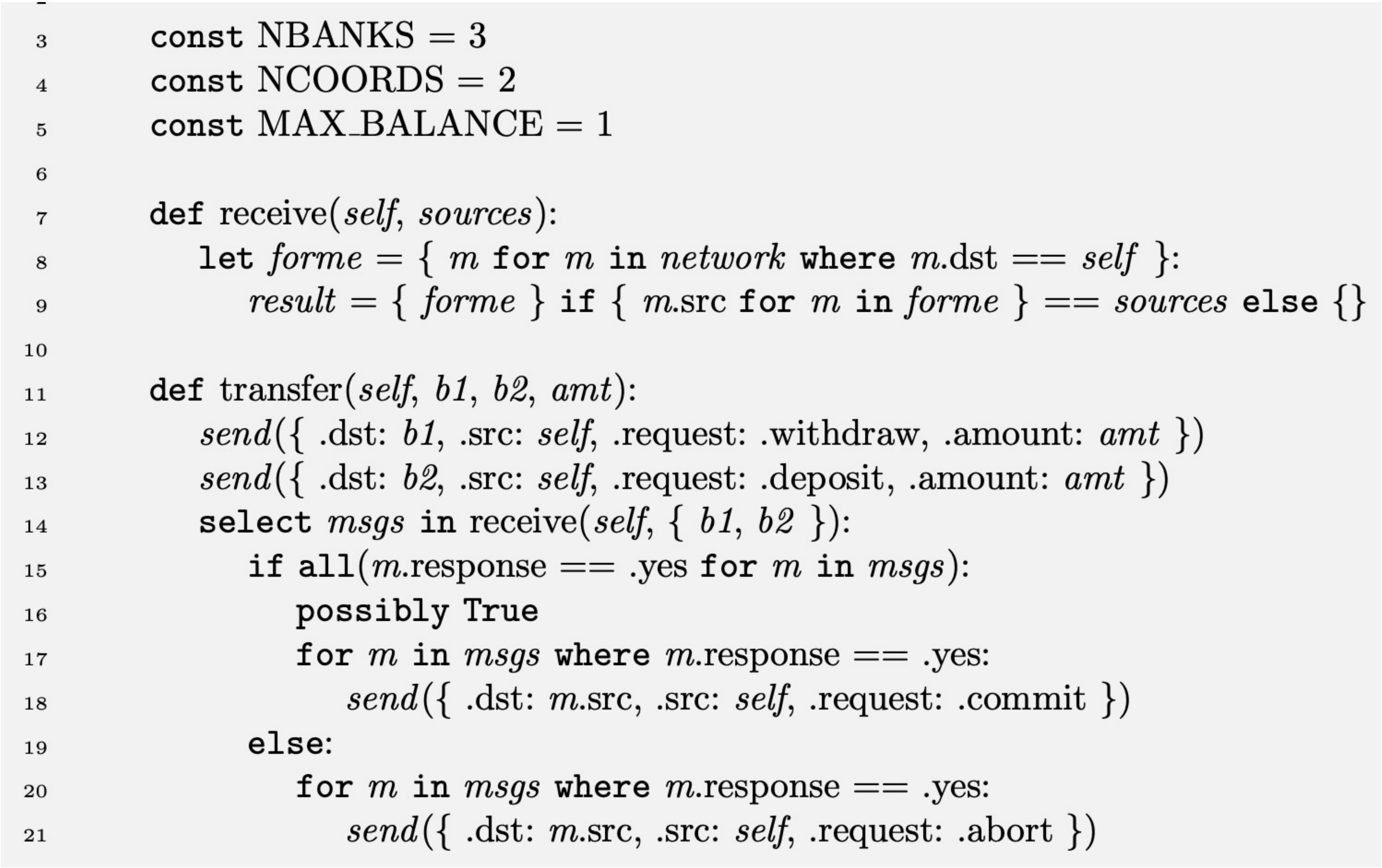
  2B: upon receiving COMMIT, participant finalizes local operations
  upon receiving ABORT, participant backs out of local operations
  release locks if any

Example: bank



```
network = {}

def send(m):
    atomic: network |= { m }

def bank(self, balance):
    let status, received = (), {}:
        while True:
            select req in network − received where req.dst == self:
                received |= { req }
                if req.request == .withdraw:
                    if (status != ()) or (req.amount > balance):
                        send({ .dst: req.src, .src: self, .response: .no })
                    else:
                        status = balance
                        balance −= req.amount
                        send({ .dst: req.src, .src: self, .response: .yes, .funds: balance })
                elif req.request == .deposit:
                    if status != ():
                        send({ .dst: req.src, .src: self, .response: .no })
                    else:
                        status = balance
                        balance += req.amount
                        send({ .dst: req.src, .src: self, .response: .yes, .funds: balance })
                elif req.request == .commit:
                    assert status != ()
                    status = ()
                else:
                    assert (status != ()) and (req.request == .abort)
                    balance, status = status, ()
```

Transfer:



```
const NBANKS = 3
const NCOORDS = 2
const MAX_BALANCE = 1

def receive(self, sources):
    let forme = { m for m in network where m.dst == self }:
        result = { forme } if { m.src for m in forme } == sources else {}

def transfer(self, b1, b2, amt):
    send({ .dst: b1, .src: self, .request: .withdraw, .amount: amt })
    send({ .dst: b2, .src: self, .request: .deposit, .amount: amt })
    select msgs in receive(self, { b1, b2 }):
        if all(m.response == .yes for m in msgs):
            possibly True
            for m in msgs where m.response == .yes:
                send({ .dst: m.src, .src: self, .request: .commit })
        else:
            for m in msgs where m.response == .yes:
                send({ .dst: m.src, .src: self, .request: .abort })
```

ACID revisited

ACID properties

- Atomicity: all or nothing (transactions may commit or abort)
  - 2PC protocol or some other atomic commitment protocol
- Consistency: satisfies application-level invariants
  - That’s up to the application (for example, prevent negative bank balances)
- Isolation: appears as if transactions are executed serially
  - Concurrency control protocol such as 2PL
- Durable: effects of successful transactions are permanent
  - Pragmatically: store data on disk ideally before you commit