

The Art of Multiprocessor Programming

Chapter 3: Concurrent objects

3 Concurrent objects

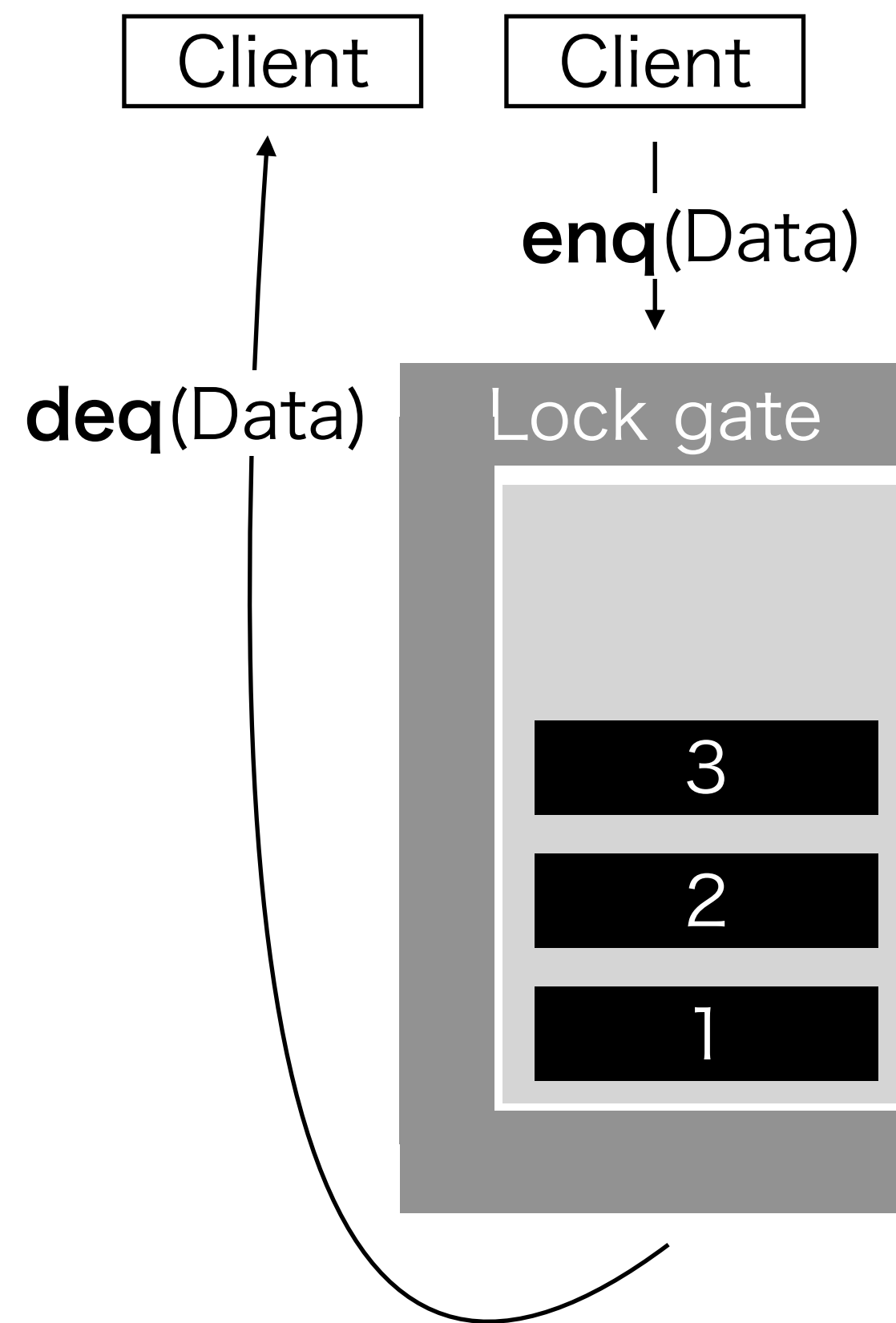
Overview

- **Correctness:**
 - Sequential Consistency
 - Linearizability
 - Quiescent Consistency
- **Progress:**
 - Blocking
 - Non-blocking

3.1 Concurrency and correctness

Simple lock-based concurrent FIFO queue

```
1 class LockBasedQueue<T> {
2     int head, tail;
3     T[] items;
4     Lock lock;
5     public LockBasedQueue(int capacity) {
6         head = 0; tail = 0;
7         lock = new ReentrantLock();
8         items = (T[])new Object[capacity];
9     }
10    public void enq(T x) throws FullException {
11        lock.lock();
12        try {
13            if (tail - head == items.length)
14                throw new FullException();
15            items[tail % items.length] = x;
16            tail++;
17        } finally {
18            lock.unlock();
19        }
20    }
21    public T deq() throws EmptyException {
22        lock.lock();
23        try {
24            if (tail == head)
25                throw new EmptyException();
26            T x = items[head % items.length];
27            head++;
28            return x;
29        } finally {
30            lock.unlock();
31        }
32    }
33 }
```

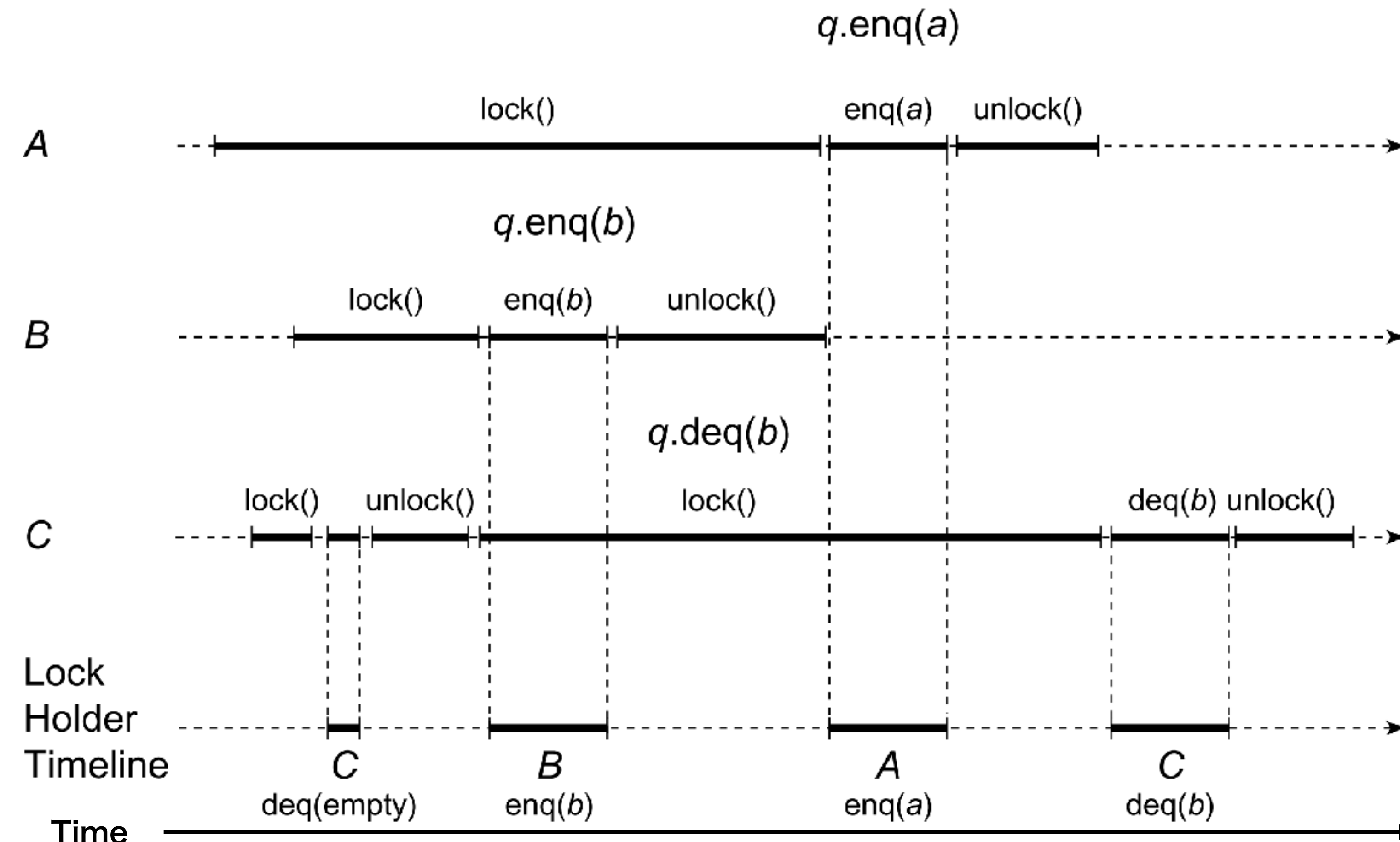


- Correct FIFO Queue
- 相互排他ロックによる同期
- FIFO: First in, First out
- Enqueue item: $q.enq(x)$
- Dequeue item: $q.deq(x)$

FIGURE 3.1

3.1 Concurrency and correctness

Simple lock-based concurrent FIFO queue



- 呼び出しは逐次的に実行される
- 一般的な逐次FIFO Queueの振る舞いと一致

FIGURE 3.2

3.1 Concurrency and correctness

Simple lock-based concurrent FIFO queue

- Amdahlの法則に照らすと, 排他的ロックを持つオブジェクトは性能が良くない

▶ Amdahl's law: $S = \frac{1}{(1 - p) + \frac{p}{n}}$ ← p is small, because there is locked time

- Need a way to specify the behavior required of concurrent objects, without method-level locking

3.1 Concurrency and correctness

Wait-free queue

```
1  class WaitFreeQueue<T> {
2      int head = 0, tail = 0;
3      T[] items;
4      public WaitFreeQueue(int capacity) {
5          items = (T[]) new Object[capacity];
6      }
7      public void enq(T x) throws FullException {
8          if (tail - head == items.length)
9              throw new FullException();
10         items[tail % items.length] = x;
11         tail++;
12     }
13     public T deq() throws EmptyException {
14         if (tail - head == 0)
15             throw new EmptyException();
16         T x = items[head % items.length];
17         head++;
18         return x;
19     }
20 }
```

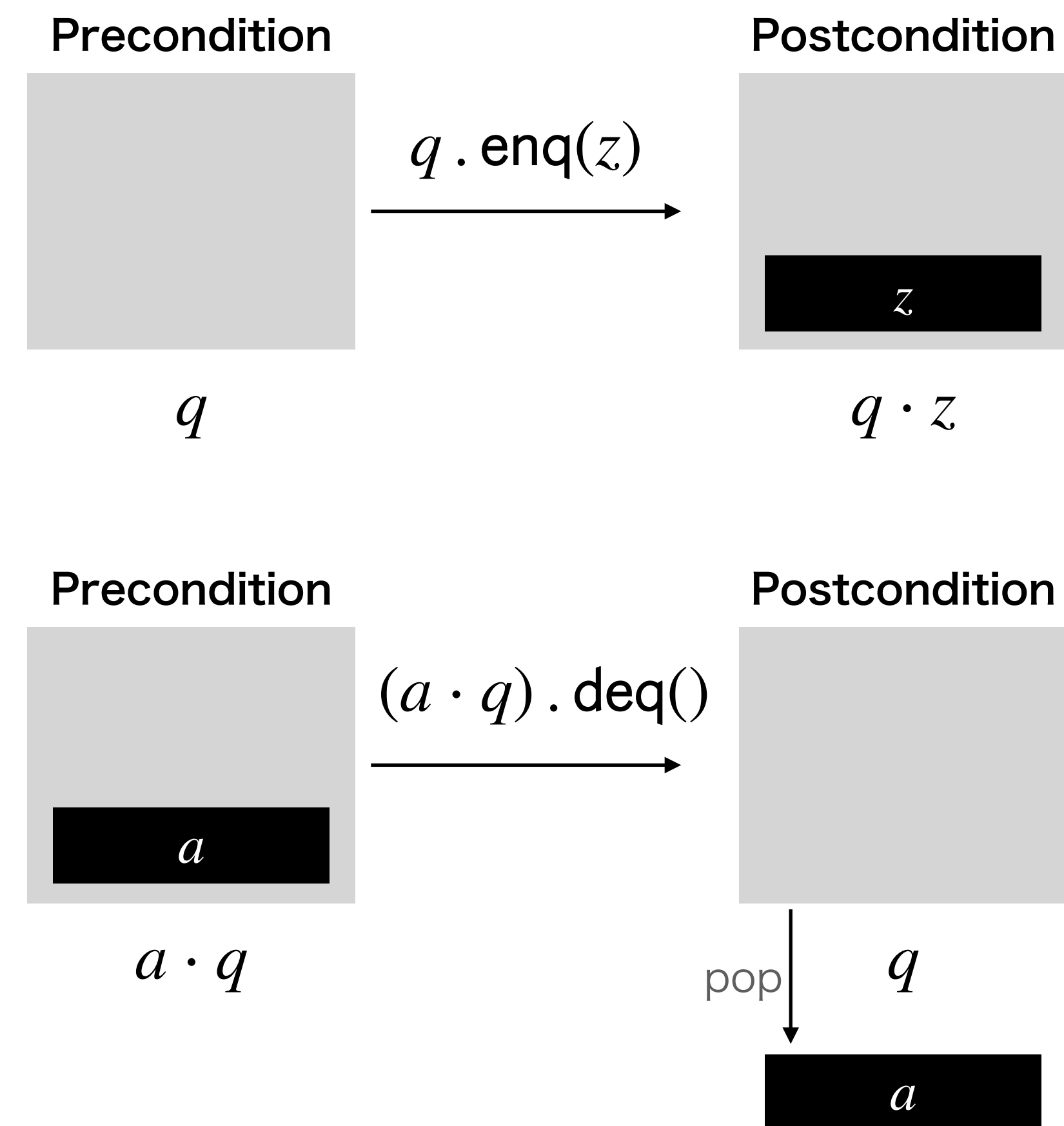
FIGURE 3.3

- Single-enqueuer/dequeuer FIFO queue

3.2 Sequential objects

Sequential specification

- **Precondition:** The object's state before invoking the method
- **Postcondition:** The object's state when the method returns (**side effect**) and return value
- It make sense in a sequential model of computation with single thread



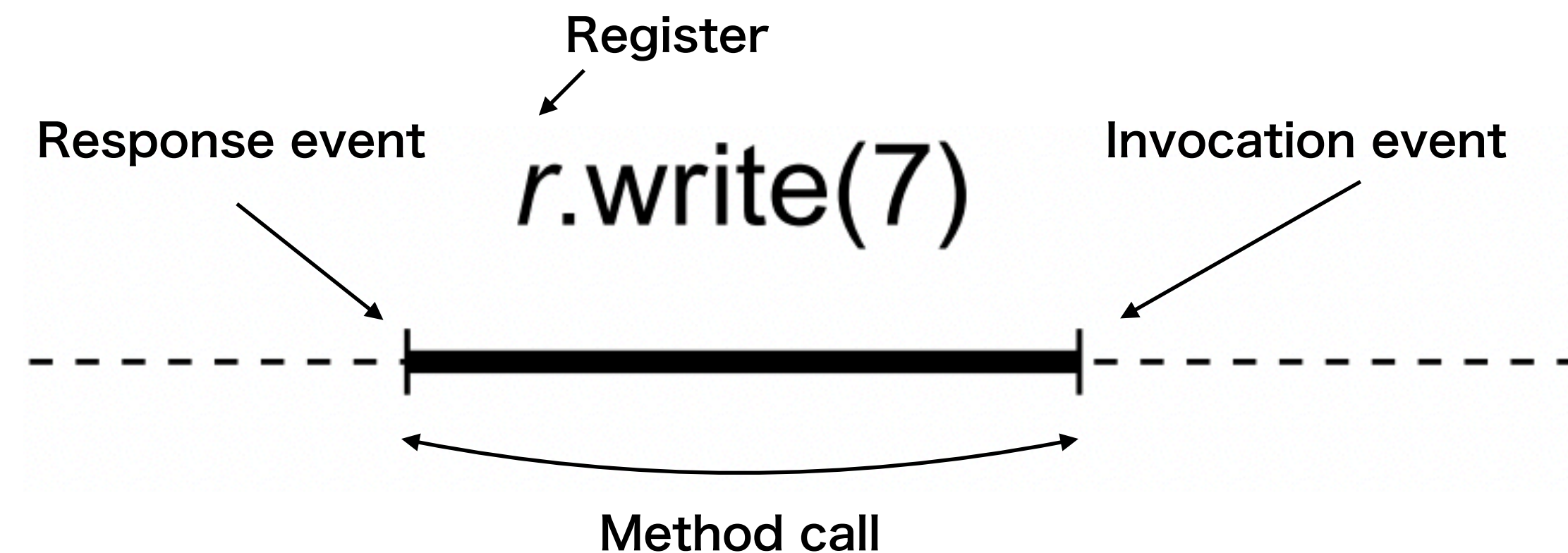
3.2 Sequential objects

Sequential specification with multiple threads

- For objects shared by multiple threads, the sequential specification is inappropriate.
 - Object calls may occur simultaneously
 - The order will be confusing.
 - Method call gets incomplete results

3.3 Sequential consistency

- **Method call**: Interval that starts with an **invocation event** and continues until the corresponding **response event**
 - **Pending**: If invocation event has occurred, but its response event has not



3.3 Sequential consistency

- Method calls should appear to happen in a one-at-a-time, sequential order
- Method calls should appear to take effect in program order

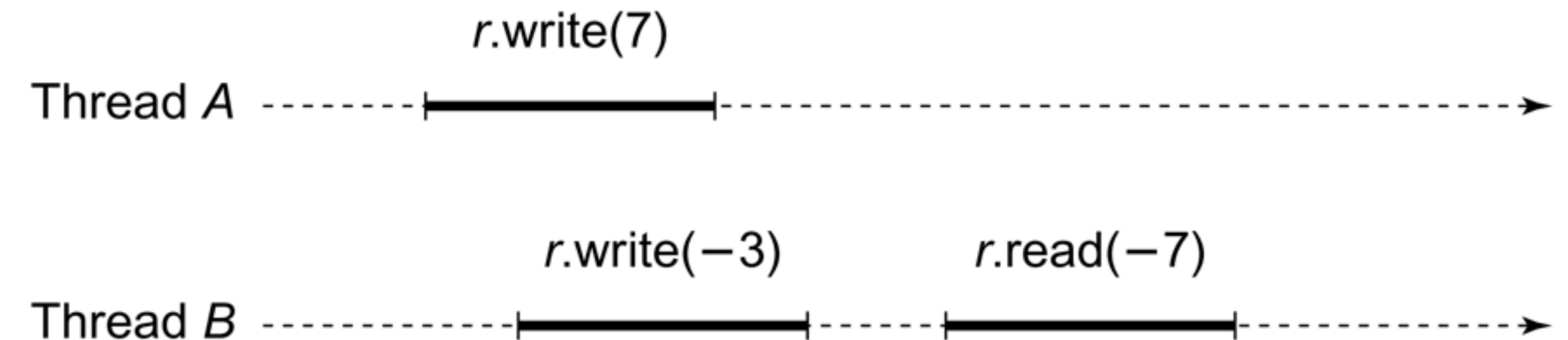


FIGURE 3.4

Why each method call should appear to take effect in one-at-a-time order. Two threads concurrently write `-3` and `7` to a shared register `r`. Later, one thread reads `r` and returns the value `-7`. We expect to find either `7` or `-3` in the register, not a mixture of both.

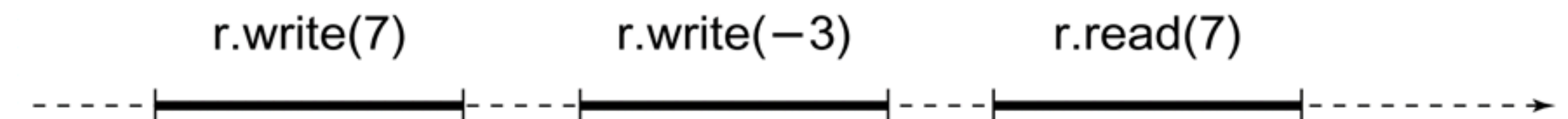


FIGURE 3.5

Why method calls should appear to take effect in program order. This behavior is not acceptable because the value the thread read is not the last value it wrote (and no other thread writes to the register).

3.3 Sequential consistency

Sequential consistency versus real-time order

- Since the order between threads can be interchanged, seemingly strange things can happen.
- Sequential consistency need not preserve the real-time order.

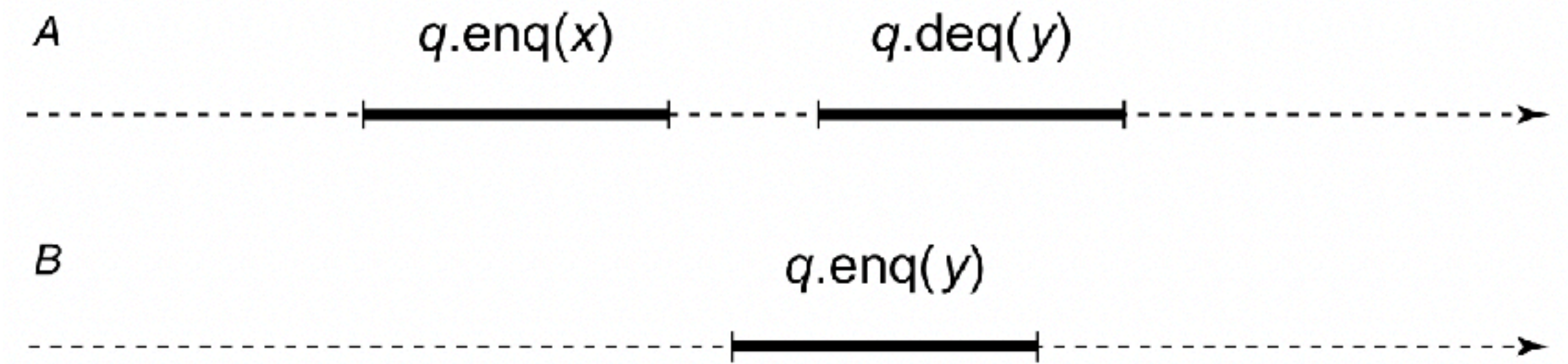


FIGURE 3.7

Sequential consistency versus real-time order. Thread *A* enqueues *x*, and later thread *B* enqueues *y*, and finally *A* dequeues *y*. This execution may violate our intuitive notion of how a FIFO queue should behave because the method call enqueueing *x* finishes before the method call enqueueing *y* starts, so *y* is enqueued after *x*. But it is dequeued before *x*. Nevertheless, this execution is sequentially consistent.

3.3 Sequential consistency

Sequential consistency is nonblocking

- In sequential consistency, there is no blocking such that a call to one method waits for another method to complete.

- ▶ **Nonblocking**

3.3 Sequential consistency

Compositionality

- A correctness property P is compositional if, whenever each object in the system satisfies P , the system as a whole satisfies P .
- Sequential consistency is not compositional.

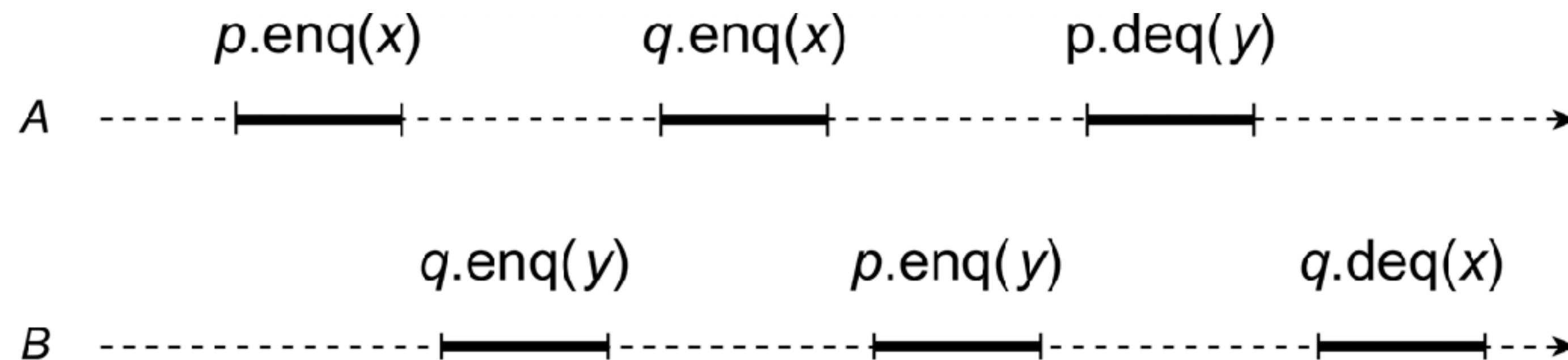


FIGURE 3.8

Sequential consistency is not compositional. Two threads, A and B , call enqueue and dequeue methods on two queue objects, p and q . It is not hard to see that p and q are each sequentially consistent, yet the execution as a whole is *not* sequentially consistent.

3.3 Sequential consistency

Compositionality

- Proof

- ▶ A dequeues y from p , y must have been enqueued at p before x :

$$\langle p . \text{enq}(y) \ B \rangle \rightarrow \langle p . \text{enq}(x) \ A \rangle$$

- ▶ x must have been enqueued onto q before y :

$$\langle q . \text{enq}(x) \ A \rangle \rightarrow \langle q . \text{enq}(y) \ B \rangle$$

- ▶ But program order implies that

$$\langle q . \text{enq}(x) \ A \rangle \rightarrow \langle q . \text{enq}(x) \ A \rangle \quad \text{and} \quad \langle q . \text{enq}(y) \ B \rangle \rightarrow \langle q . \text{enq}(y) \ B \rangle$$

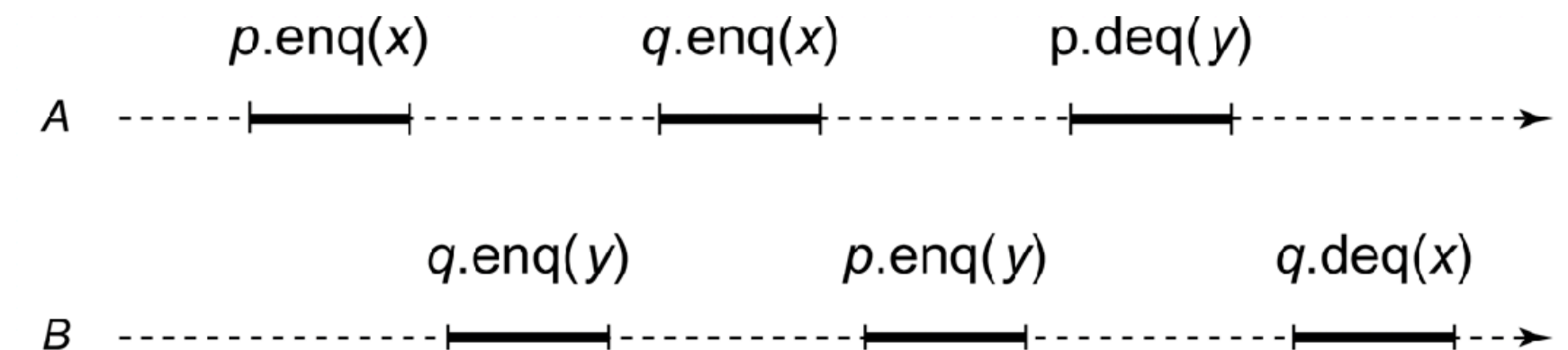


FIGURE 3.8

Sequential consistency is not compositional. Two threads, A and B , call enqueue and dequeue methods on two queue objects, p and q . It is not hard to see that p and q are each sequentially consistent, yet the execution as a whole is *not* sequentially consistent.

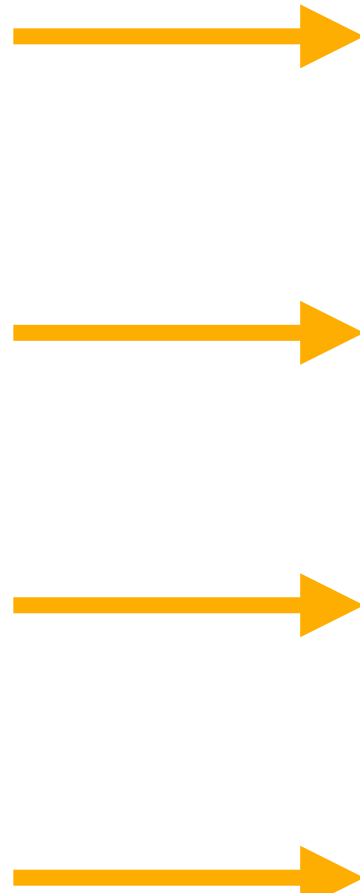
3.4 Linearizability

- Linearizability:
 - Each method call should appear to take effect instantaneously at some moment between its invocation and response

3.4 Linearizability

Linearization points

- Usual way to show that a concurrent object implementation is linearizable: to identify for each method a **linearization point**
 - Lock-based impl: critical section
 - Non-lock impl: single step



```
1  class WaitFreeQueue<T> {
2      int head = 0, tail = 0;
3      T[] items;
4      public WaitFreeQueue(int capacity) {
5          items = (T[]) new Object[capacity];
6      }
7      public void enq(T x) throws FullException {
8          if (tail - head == items.length)
9              throw new FullException();
10         items[tail % items.length] = x;
11         tail++;
12     }
13     public T deq() throws EmptyException {
14         if (tail - head == 0)
15             throw new EmptyException();
16         T x = items[head % items.length];
17         head++;
18         return x;
19     }
20 }
```

FIGURE 3.3

3.4 Linearizability

Linearizability versus sequential consistency

- Linearizability is nonblocking too
- Linearizability does not limit concurrency
- Threads are cannot distinguish between sequential consistency and linearizability
- **Linearizability is compositional**