Lecture 18: Sequential Consistency

COSC 273: Parallel and Distributed Computing

Spring 2023

Lecture 18: Sequential Consistency

COSC 273: Parallel and Distributed Computing

Spring 2023

Announcements

1. Lab 03 Due ~~Friday~~ MONDAY!!
    - Mandelbrot computations using `Vector` operations
    - Make sure your machine supports `Vector` ops **today**:

    ```
    > javac --add-modules jdk.incubator.vector SomeFile.java
    > java --add-modules jdk.incubator.vector SomeFile
    ```

    on HPC cluster, first run

    ```
    > module load amh-java/19.0.1
    ```

Last Time

Concurrent Objects!

- concurrent linked lists:
  1. lock the whole list to `insert`
  2. lock affected **nodes** to `insert`

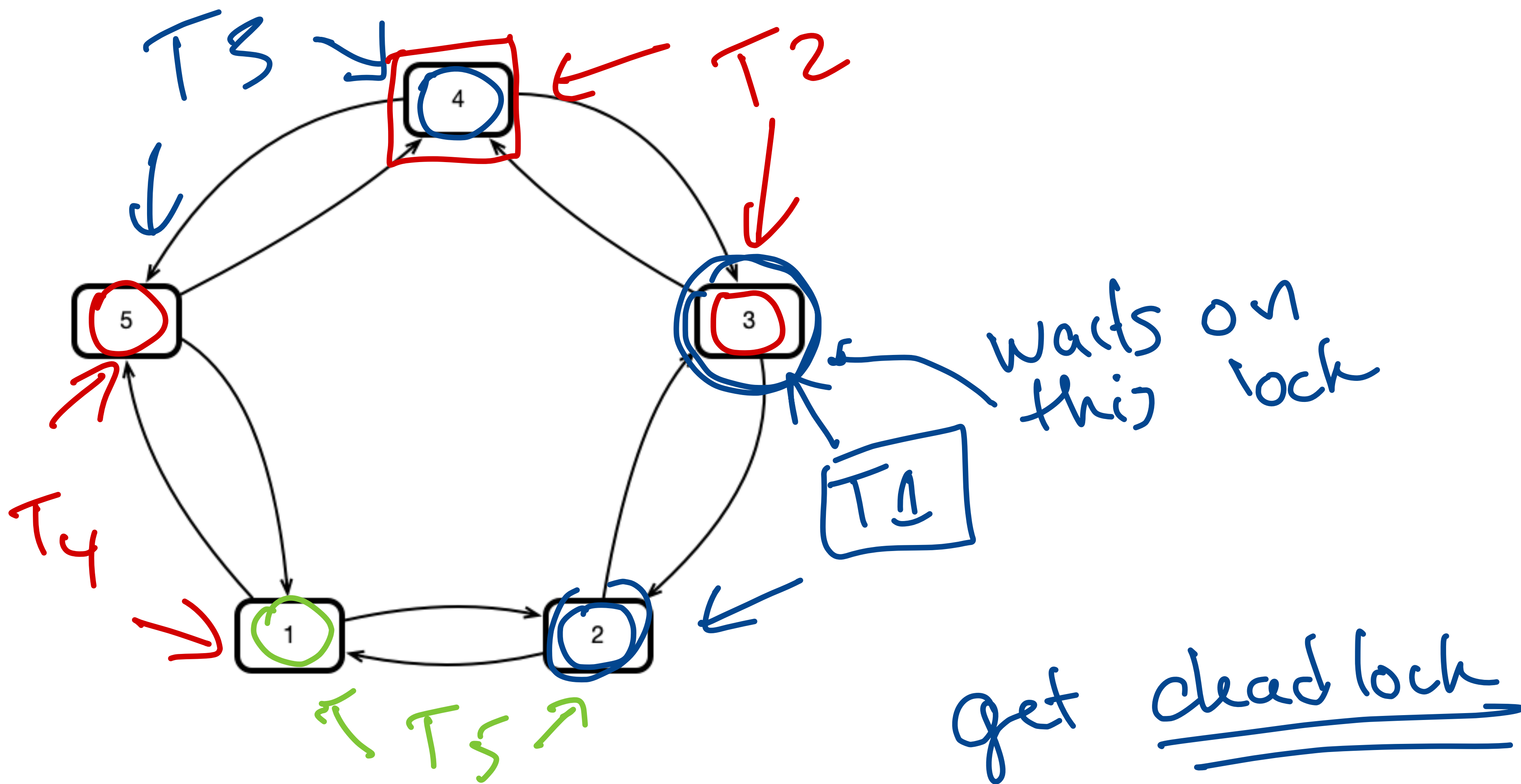
coarse locking

fine-grained locking

Option 1 is easy to reason about, but offers no benefit from parallelism

Option 2 may offer some performance benefit from parallelism, but reasoning about *correctness* is subtle

A Subtle Issue



Concurrent Queues

Question

What is a queue? "container"

Queue<T> q = ...

- "first in first out"
- add elts to "one side"
- remove from "other"

From Data Structures

An **abstract data type** (ADT) specifies:

1. allowed operations
2. effects of operations
   - return values
   - updates to internal state of object

Example. Queue ADT?

- enq(x) — enqueue the element x (add to queue)
- deq() — returns an elt. from queue (oldest elt)
- ordered collection of elts added, but not yet removed, in order they were added

What Does an ADT Give Us?

op_i is enq or deq for Queue

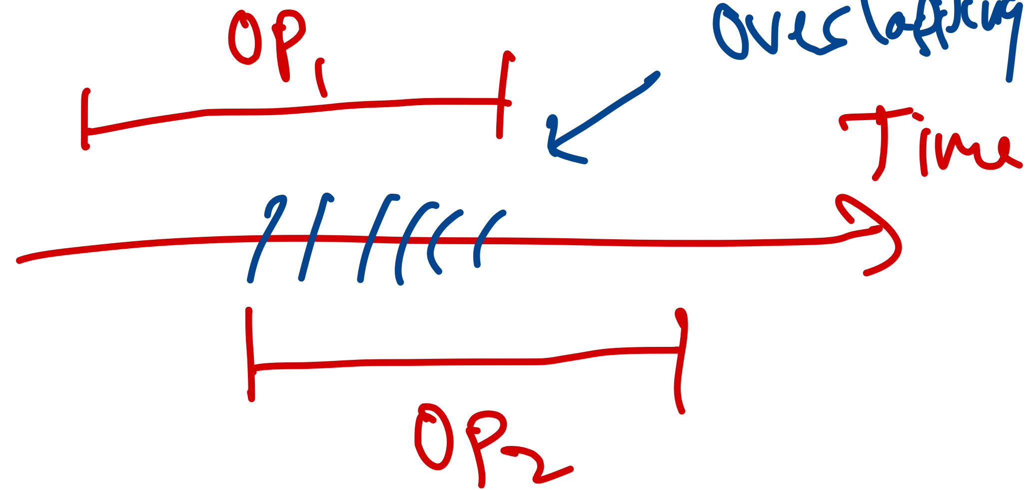
For any *sequence* of operations $op_1, op_2, op_3, \ldots, op_n$ an ADT specifies the results of these operations.

- this is a **sequential specification** of an object

Question. Why is a sequential specification insufficient for concurrent objects?

T1 : op_1

T2 : op_2



What about concurrent ops??

The Challenges of Concurrency

What if two or more operations are performed concurrently?

- What is the “correct” behavior?
- How can an implementation guarantee that the correct behavior occurs?
  - in general, each operation consists of several elemenatry steps
  - must guarantee correct behavior for all interleavings of elementary operations

Concurrent Queue Example

Thread 1:

1. `enq(x)`
2. `deq()`

Thread 2:

1. `enq(y)`

Question. What are "acceptable" results of `deq()`?

enq(x) happens "before" enq(y)

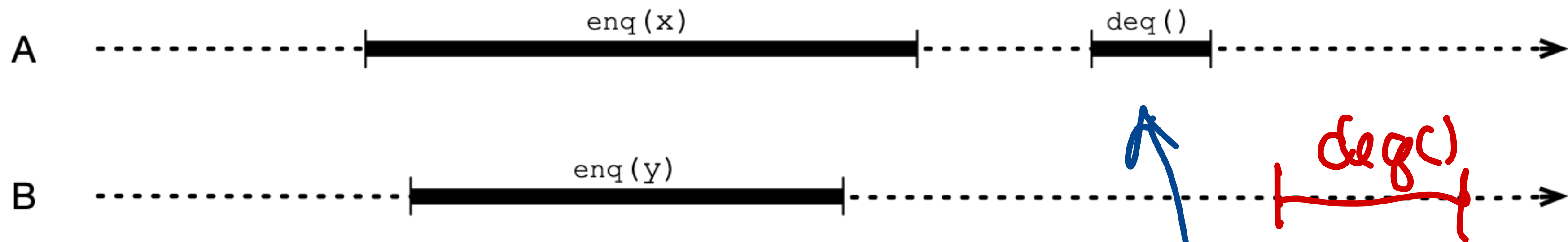
enq(y) happens "before" enq(x)

x or y

unacceptable: "empty queue exception"

b/c T1 enqued befor deq.

Concurrent Queue Timelines



- call to `deq()` could return either x or y
  - both reasonable!
- any other response seems "un-queue-ish"

should deque different elements

Sequential Consistency

A Sensible Feature

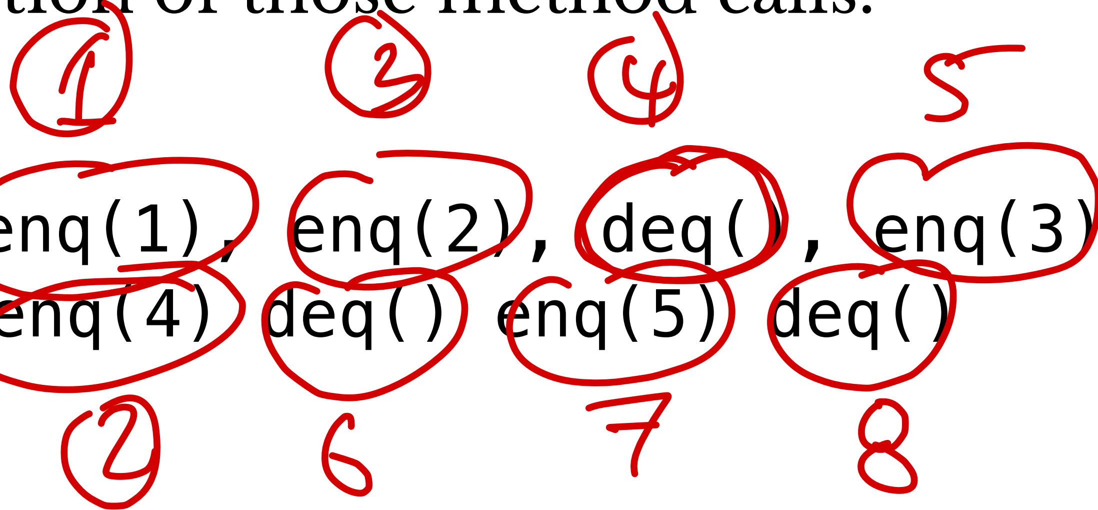
Consider all operations performed by all threads

- Each operation has some effect

Behavior of execution should be consistent with *some* **sequential** execution of those method calls.

Example.

1. Thread 1 calls `enq(1)`, `enq(2)`, `deq()`, `enq(3)`
2. Thread 2 calls `enq(4)`, `deq()`, `enq(5)`, `deq()`



Is This Enough?

Behavior of execution should be consistent with *some* sequential execution of the method calls.

1. Thread 1 calls `enq(1)`, `enq(2)`, `deq()`, `enq(3)`
2. Thread 2 calls `enq(4)` `deq()` `enq(5)` `deq()`

exception

Also must respect "program order" for each thread

Another Sensible Feature

Method calls should appear to take effect in **program order**

- if a single thread calls `op1()` before `op2()`, then `op1()` should take effect before `op2()` in sequential execution.

Sequential Consistency

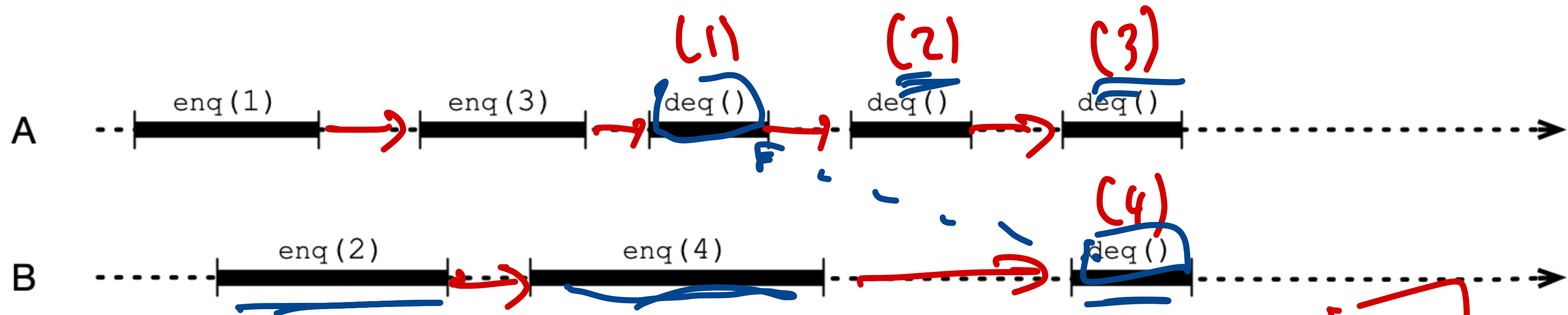
An execution is **sequentially consistent** if all method calls can be ordered such that:

1. they are consistent with program order
2. they meet object's sequential specification

An implementation of an object is sequentially consistent if

1. it guarantees *every* execution is sequentially consistent

Sequentially Consistent Outcomes?



1 : 1 4

2 : 2 1

3 : 3 3

4 : 4 2

and more

3
1

not possible

Example: A Sequentially Consistent Queue

An Array-Based Queue

```
public class LockedQueue<T> {
    int head, tail;
    T[] contents;
    Lock lock;
}
```

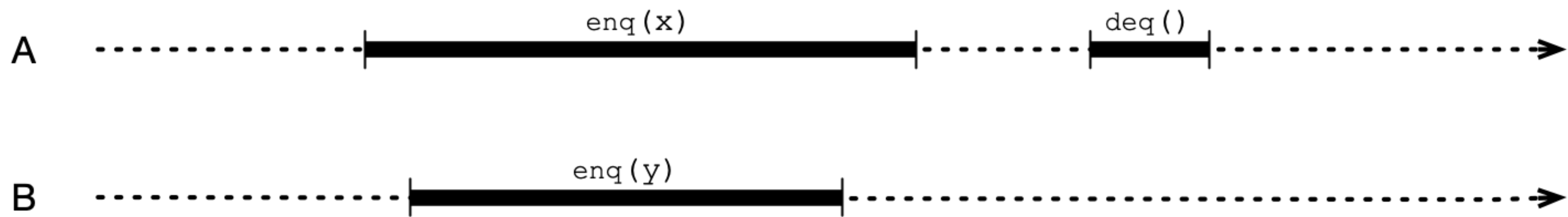
Enqueuing

```
public void enq(T x) {
    lock.lock();
    try {
        items[tail] = x;
        tail++;
    } finally {
        lock.unlock();
    }
}
```

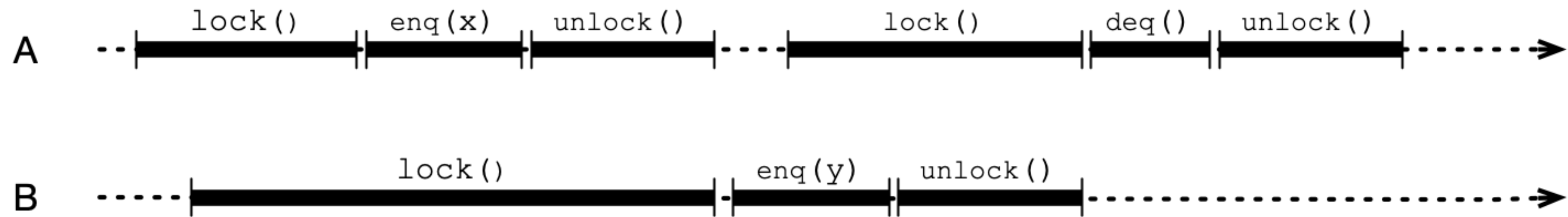
Dequeueing

```
public T deq() {
    lock.lock();
    try {
        T x = items[head];
        head++;
        return x;
    } finally {
        lock.unlock();
    }
}
```

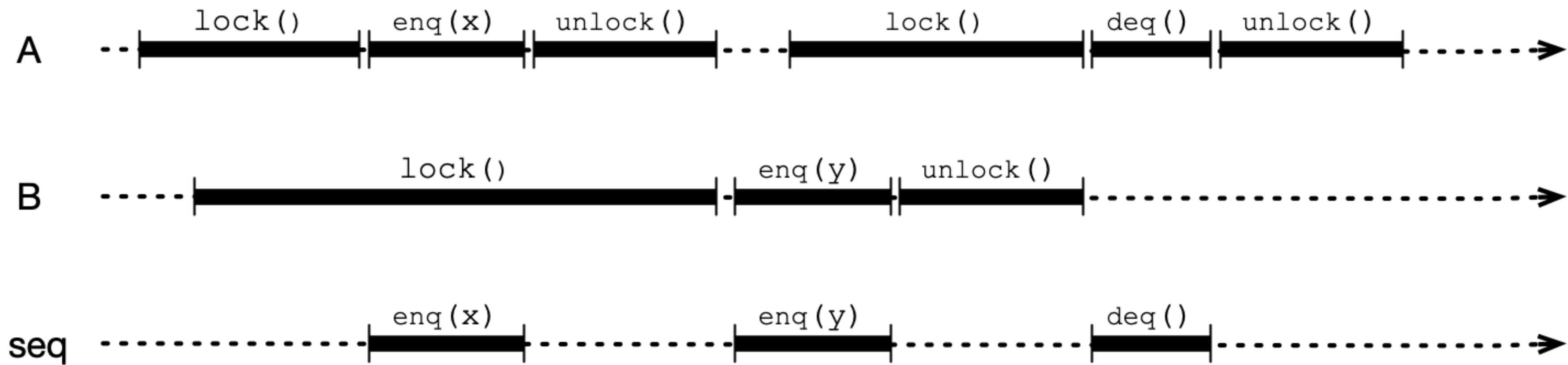
What Happens?



What Happens with Locks?



Equivalent Sequential Execution

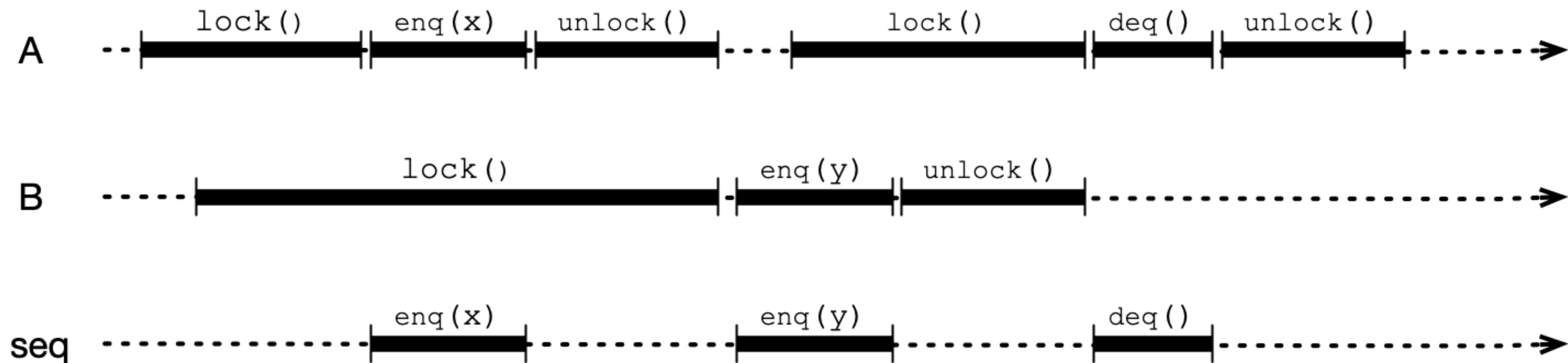


Why is Queue Sequentially Consistent?

Why is Queue Sequentially Consistent?

Locks!

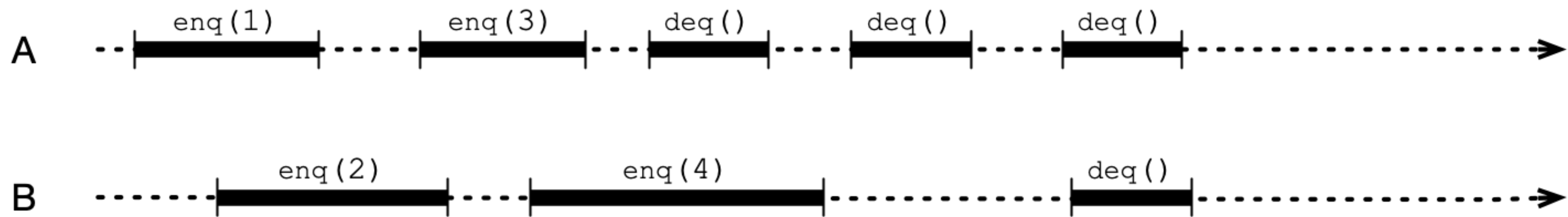
- mutual exclusion property of the `Lock` ensures that `enq`/`deq` operations are **not concurrent**
- calls to `enq`/`deq` can be ordered according to “wall clock” time of execution of critical sections



Questions

1. Can we achieve sequential consistency without resorting to locks?
    - again, this technique is essentially sequential
2. Is sequential consistency enough?

What are “Acceptable” Outcomes?



Next Time

Linearizability: A *stronger* notion of correctness for concurrent objects

- considers “wall clock” time in addition ot program order