Lecture 20: Linearizability I

COSC 273: Parallel and Distributed Computing

Spring 2023

Announcements

1. Lab 03 due tonight
2. Quiz this Friday
    - sequential consistency
    - linearizability

Previously

An execution of a concurrent object 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

ADT

An implementation of an object is sequentially consistent if

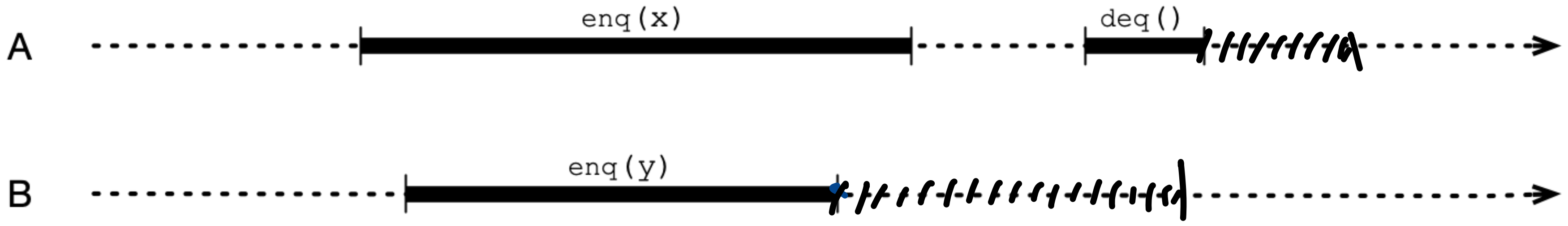
1. it guarantees *every* execution is sequentially consistent

Example: A Queue with Locks

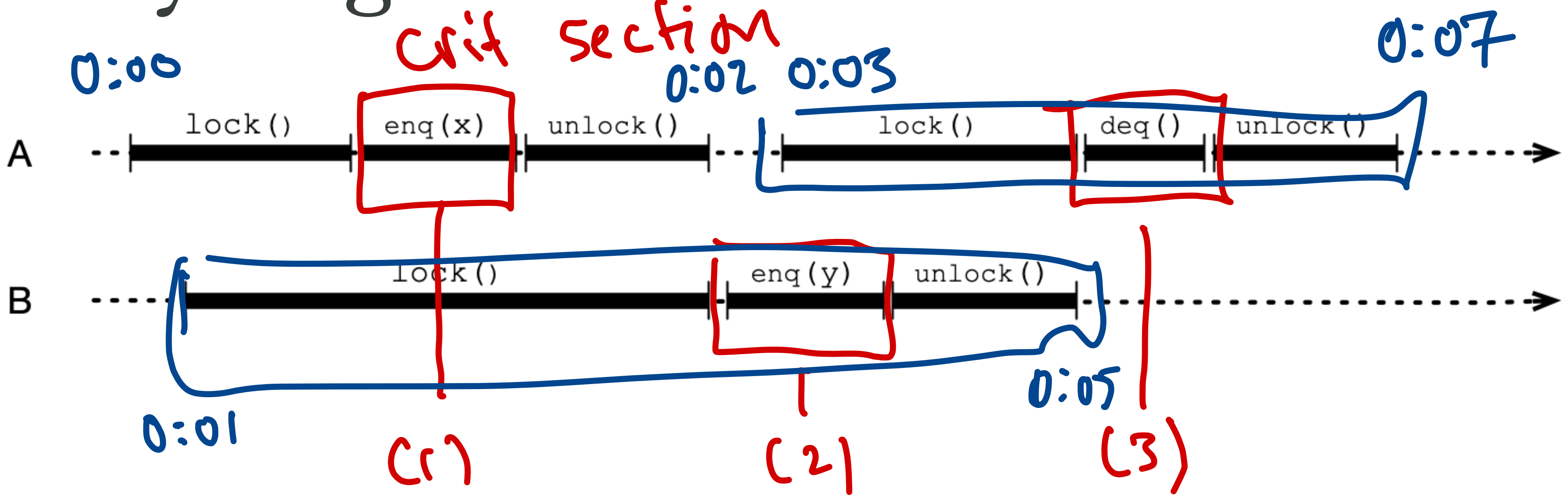
Queue supports `enq(x)` and `deq()` operations

- each instance stores a lock
- wrap `enq` and `deq` operations with `lock/unlock`
  - modifications are in critical section

Sample Concurrent Calls

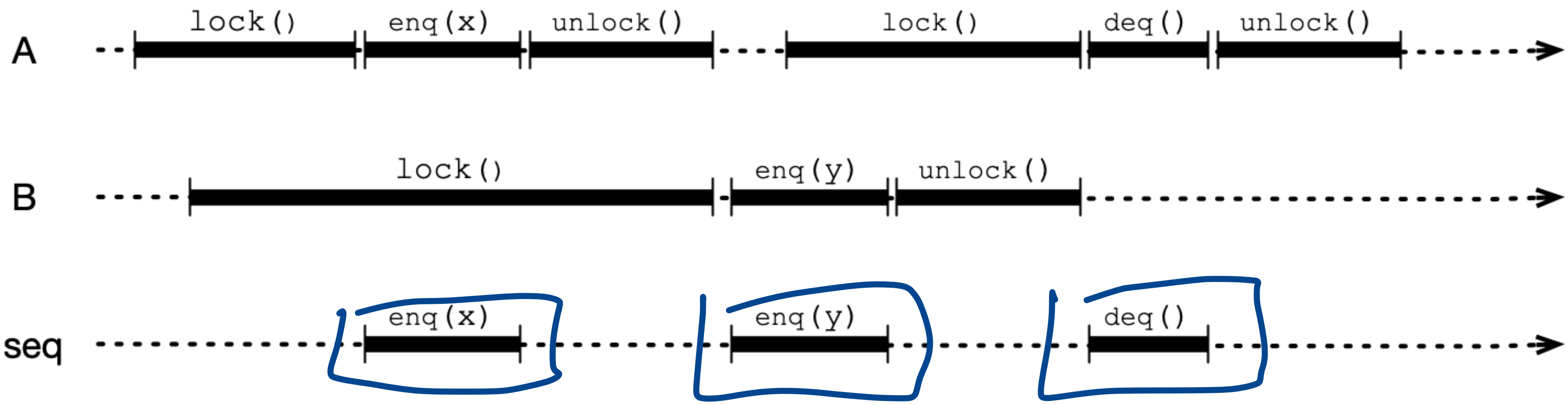


Analyzing lock/unlock Calls



Mutual exclusion ⇒
critical sections don't overlap

Equivalent Sequential Execution

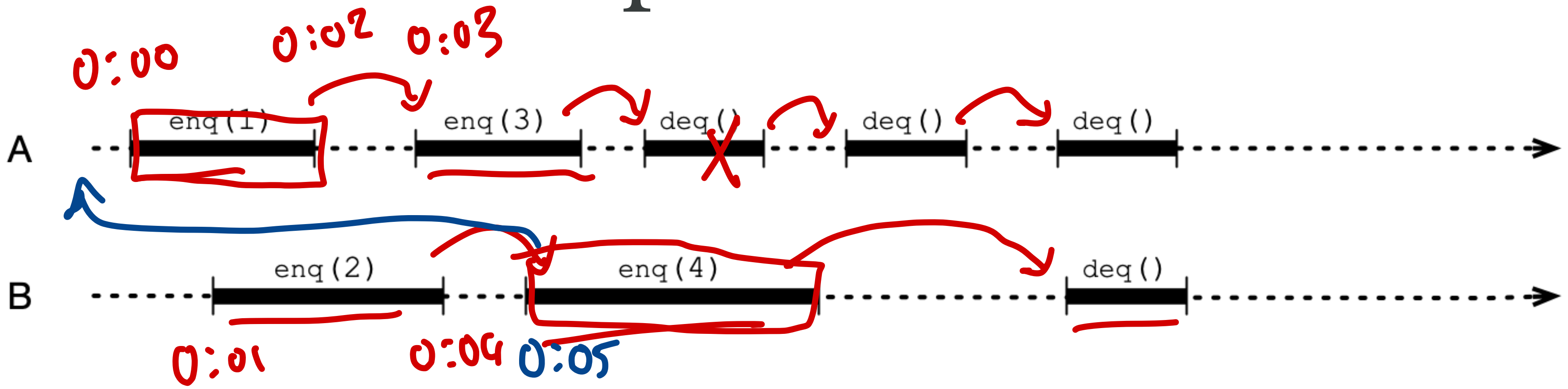


inherent to locks

Two Issues

1. Calls to enq/deq are **blocking**
   - if thread A enters critical section, other threads are blocked from making progress until A unlocks
2. Sequential consistency is a “weak” notion of correctness
   - does not necessarily respect “wall clock” order of method calls

What are "Acceptable" Outcomes?



seq. consistent outcome:

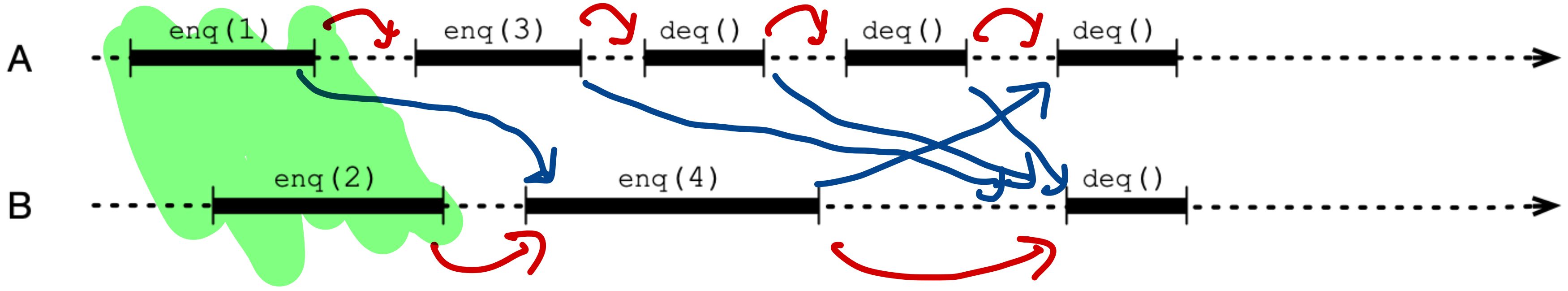
enq(2) → enq(4) → enq(1) → enq(3) → deq...

Another idea

absolute "wall clock" timing

- Make sure execution is consistent with timing of method calls
- Consider sequential executions consistent with each method call taking effect at some *instant* during the method call

Same Example, Fewer Options



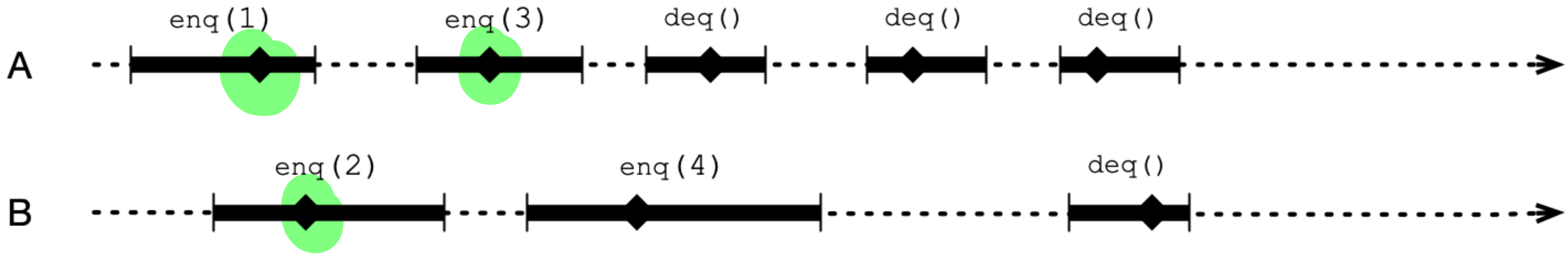
Can only change relative order of method calls if they overlap

Linearization Points

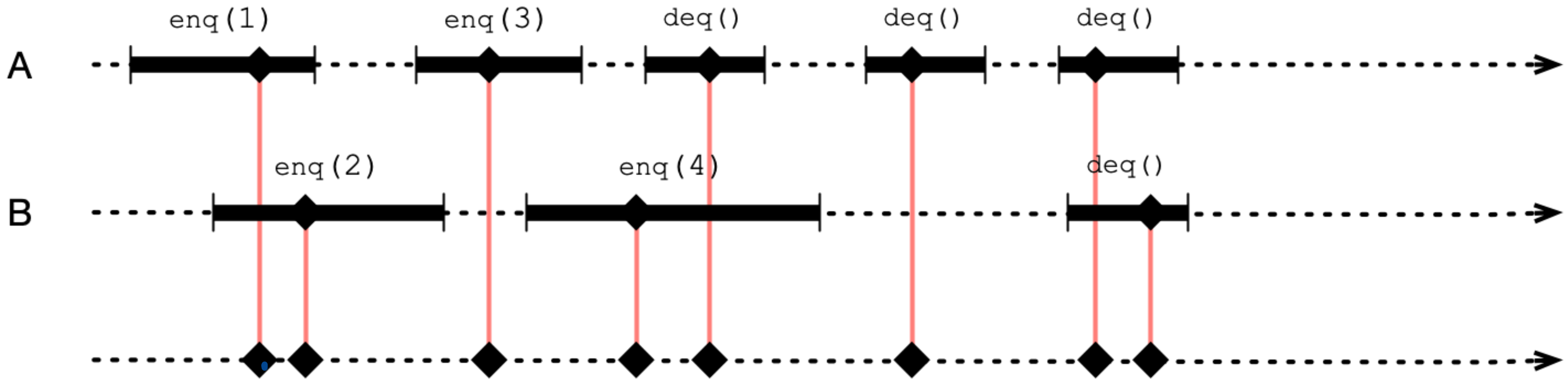
A **linearization point** is a point in a method call where method “takes effect”

- all events after linearization point see effect of method call
- linearization points must be distinct (correspond to some atomic operation)

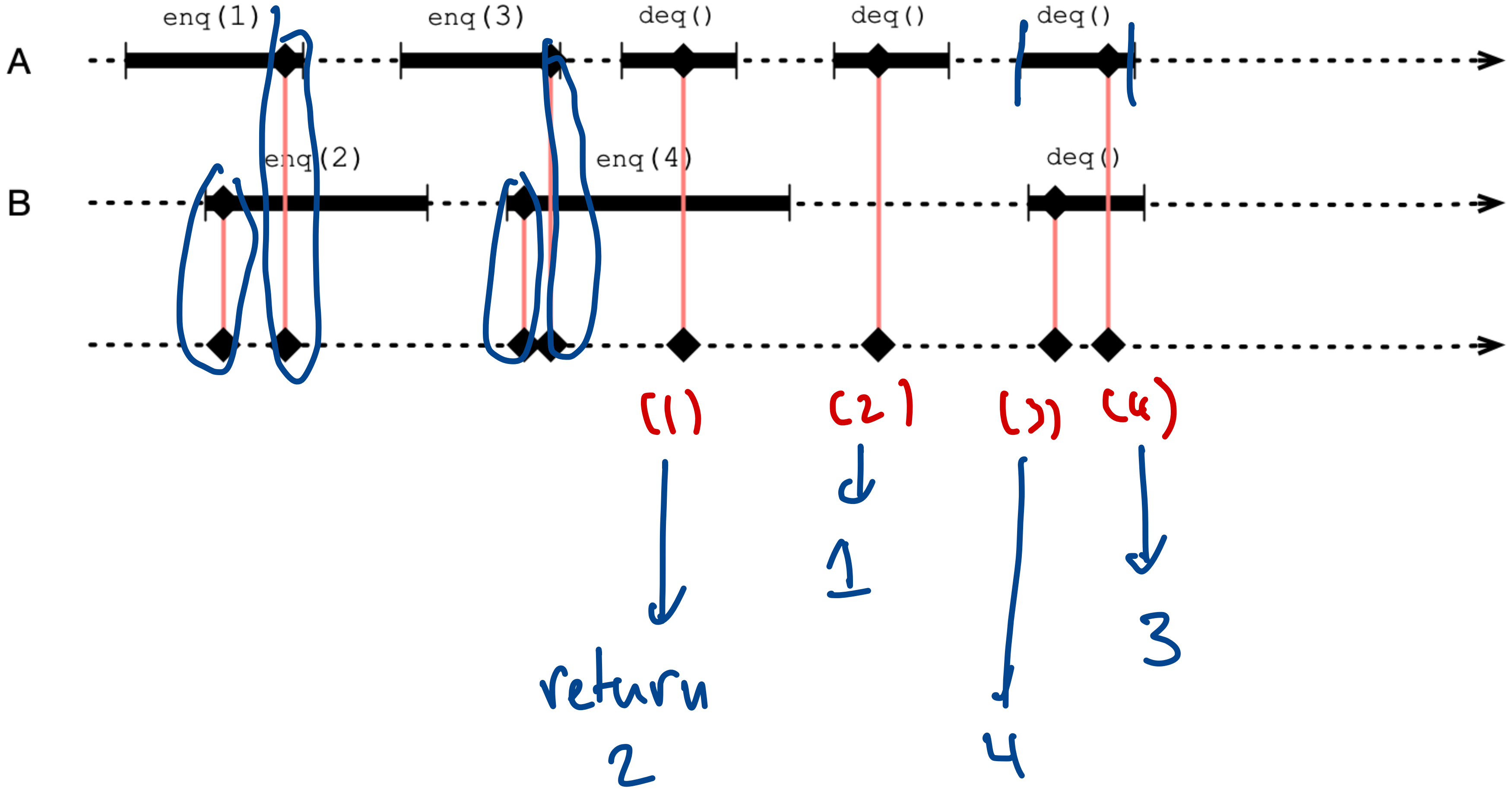
Example of Linearization Points



Equivalent Sequential Execution



An Alternative Sequential Execution



Linearizability

A concurrent execution is **linearizable** if:

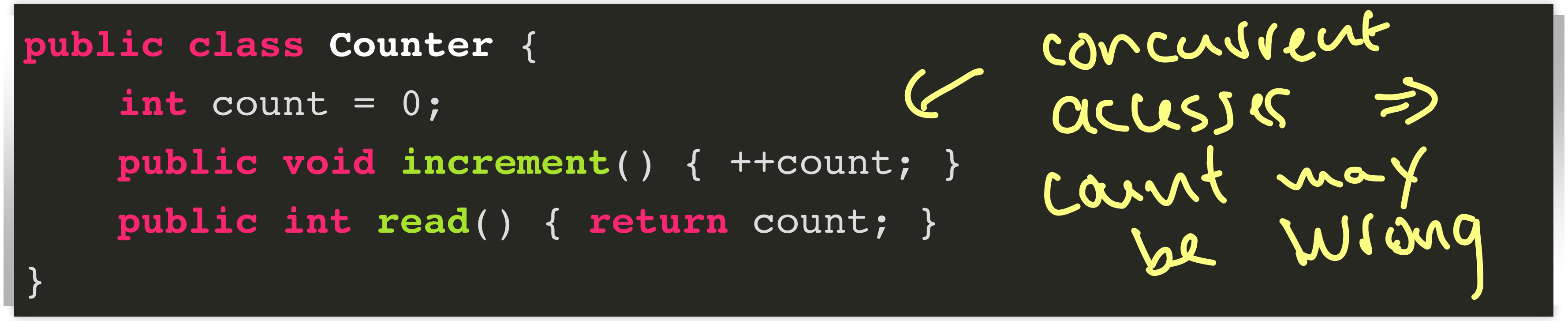
- exists a linearization point in each method call such that execution is consistent with sequential execution where method calls occur in order of corresponding linearization points

An implementation of an object is linearizable if:

- it guarantees every execution is linearizable

Back to the Counter

An incorrect (concurrent) counter



```
public class Counter {
    int count = 0;
    public void increment() { ++count; }
    public int read() { return count; }
}
```

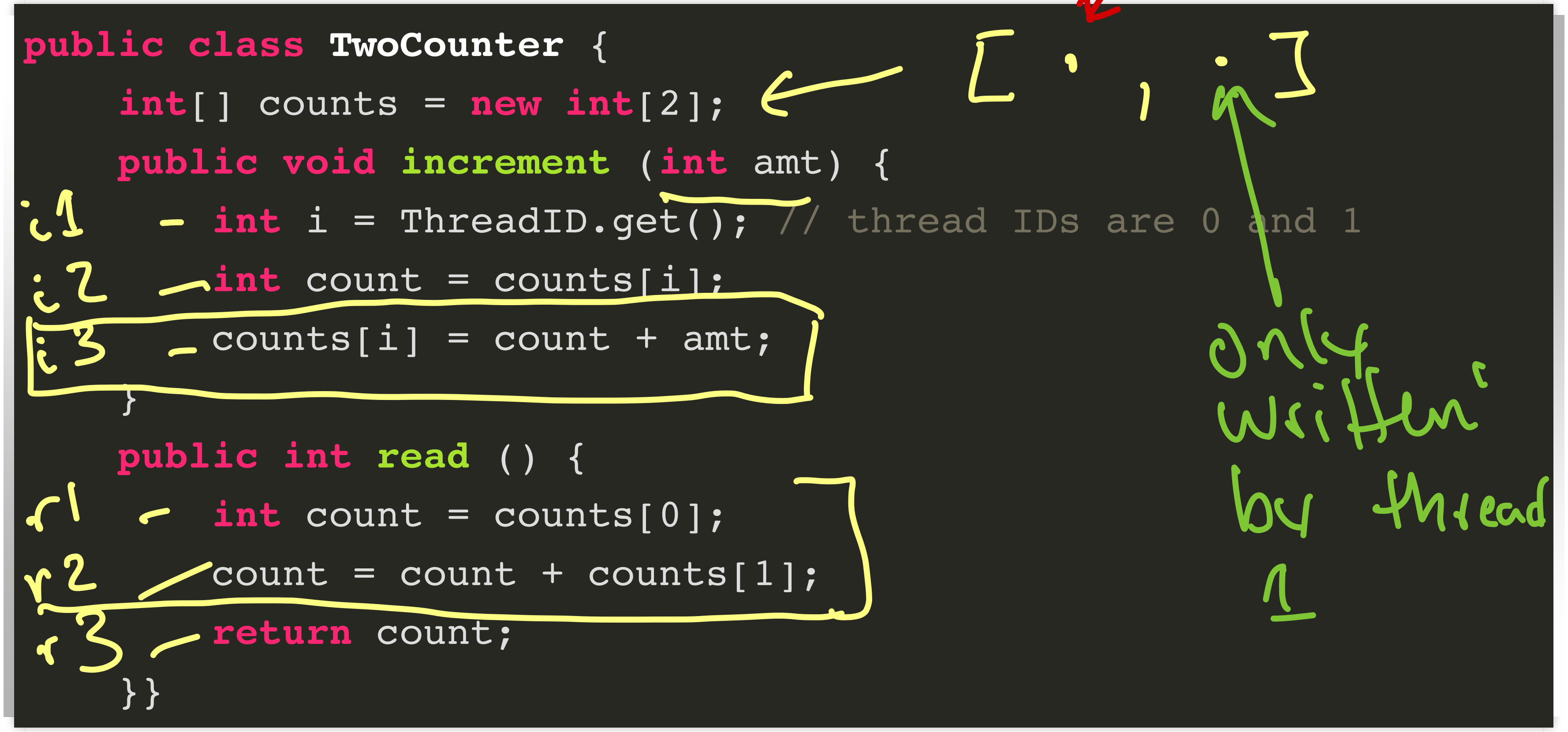
concurrent accesses ⇒ count may be wrong

Better strategy (e.g., from lab 1)?

- each thread had own "counter"
- to get final count:
  sum local counts

A Counter for Two Threads

only written by thread 0



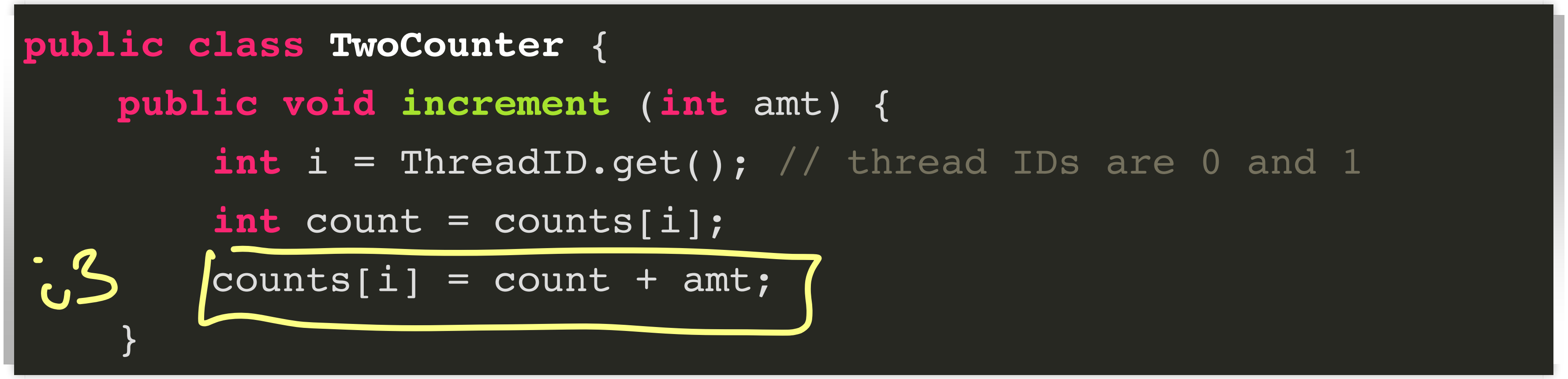
```
public class TwoCounter {
    int[] counts = new int[2];
    public void increment (int amt) {
        int i = ThreadID.get(); // thread IDs are 0 and 1
        int count = counts[i];
        counts[i] = count + amt;
    }
    public int read () {
        int count = counts[0];
        count = count + counts[1];
        return count;
}}
```

Is `TwoCounter` Linearizable?

- if not, find a non-linearizable execution
- if so, what are the linearization points for the execution

Linearizing increment

What is the linearization point of increment?



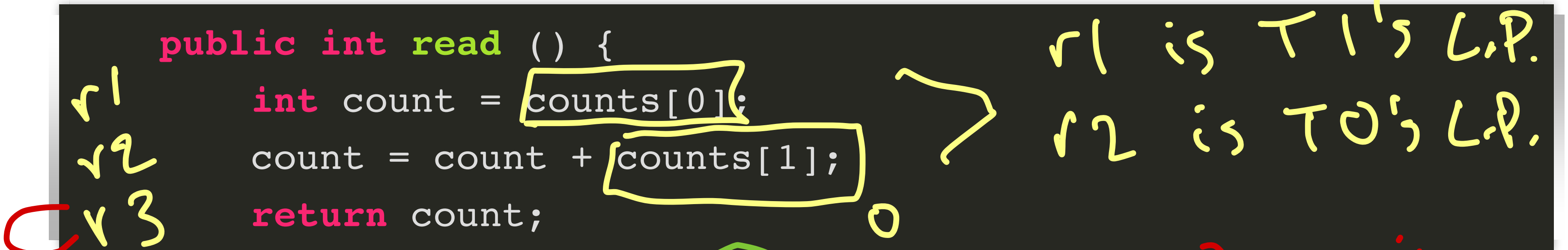
```
public class TwoCounter {
    public void increment (int amt) {
        int i = ThreadID.get(); // thread IDs are 0 and 1
        int count = counts[i];
        counts[i] = count + amt;
    }
```

i3

Linearizing read

What is the linearization point of read?

initially: count[0] count[1] are both 0



```
public int read () {
    int count = counts[0];
    count = count + counts[1];
    return count;
```

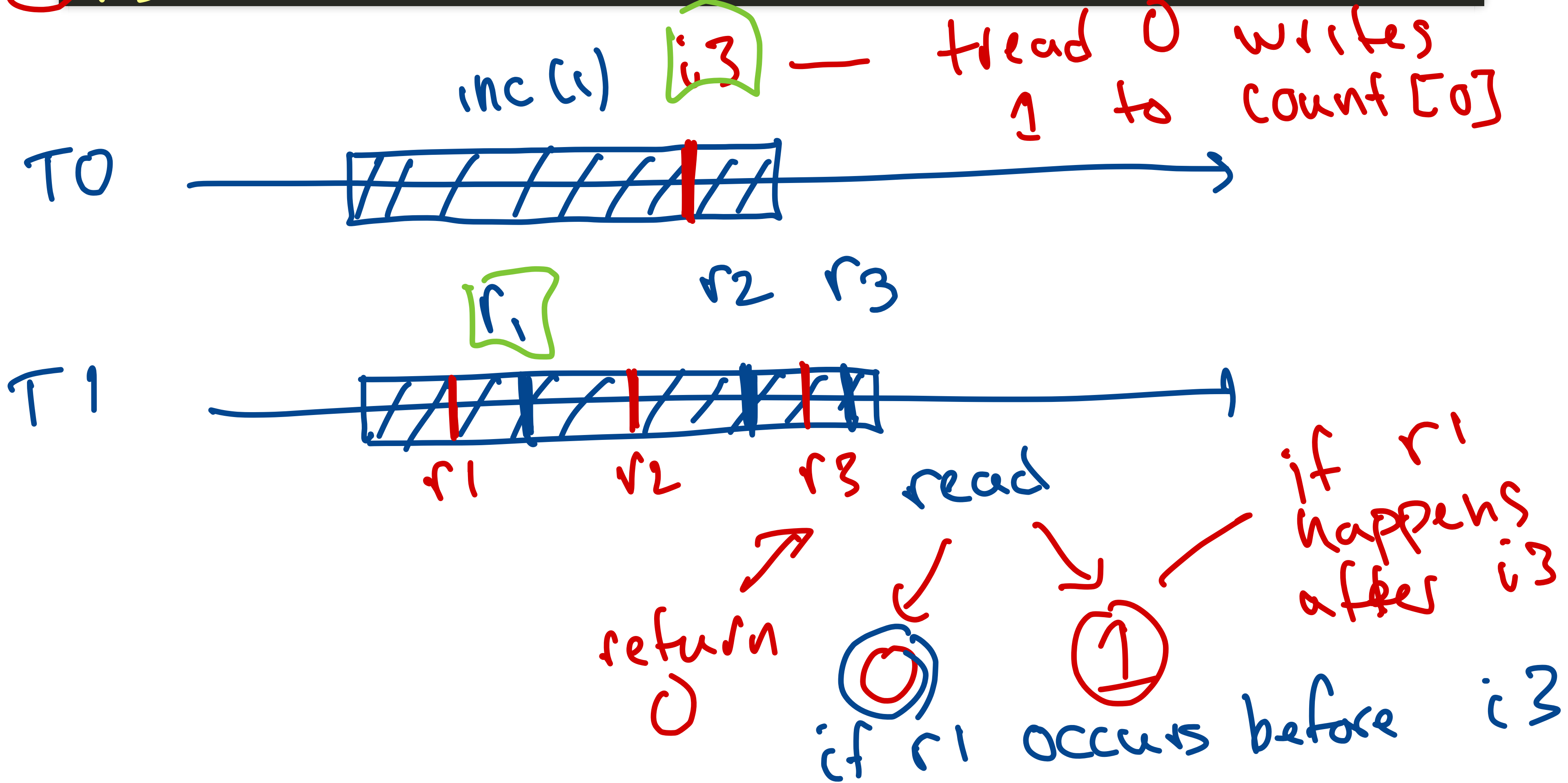
r1

r2

r3

0

r1 is T1's L.P.
r2 is T0's L.P.



i3 — thread 0 writes 1 to count[0]

if r1 happens after i3

if r1 occurs before i3

First Moral

```
    public int read () {
        int count = counts[0];
        count = count + counts[1];
        return count;
```

The linearization point may depend on

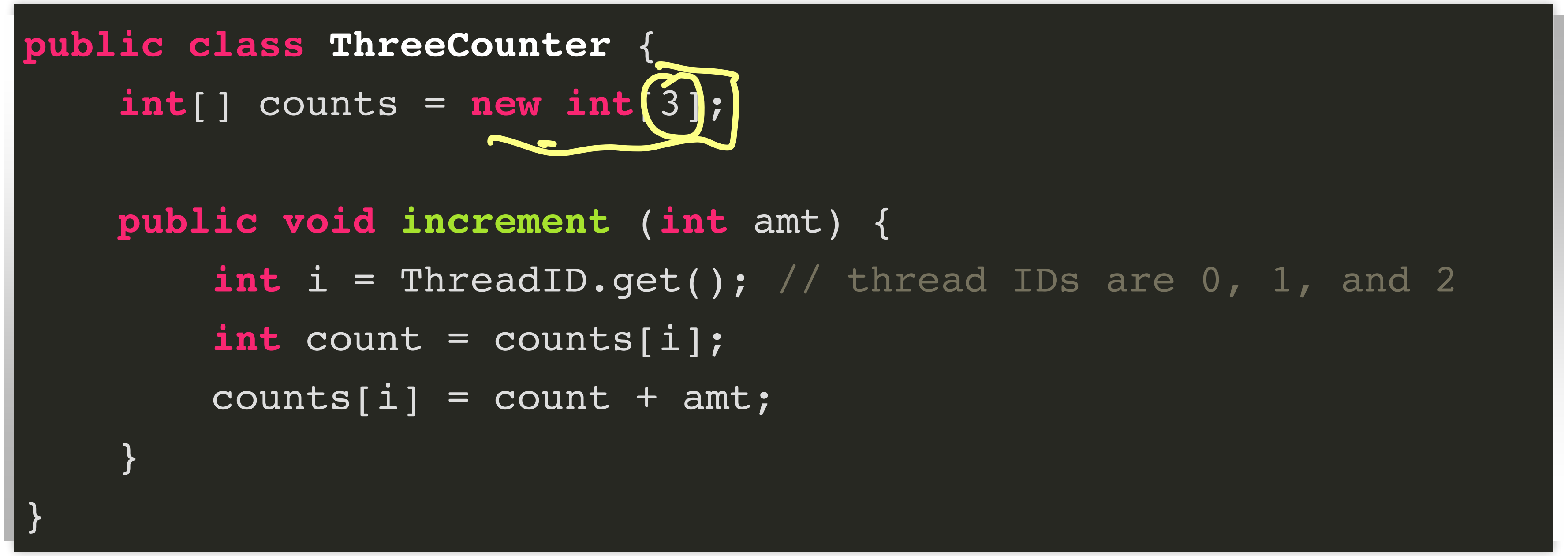
- which thread calls the method
- method calls of other threads

Three Threaded Counter?

How to generalize `TwoCounter` to three threads?

Three Threaded Counter?

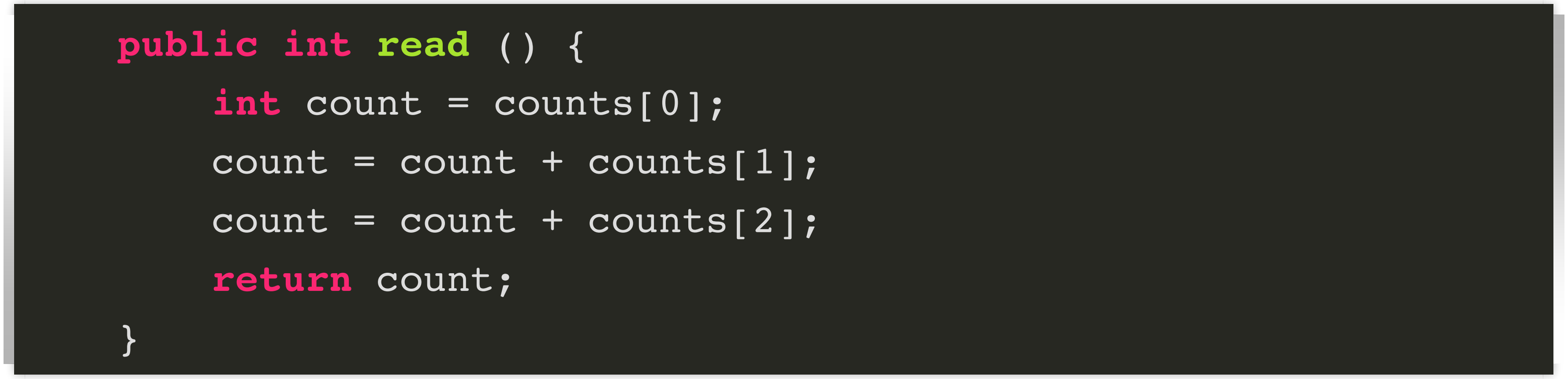
How to generalize `TwoCounter` to three threads?



```
public class ThreeCounter {
    int[] counts = new int[3];

    public void increment (int amt) {
        int i = ThreadID.get(); // thread IDs are 0, 1, and 2
        int count = counts[i];
        counts[i] = count + amt;
    }
}
```

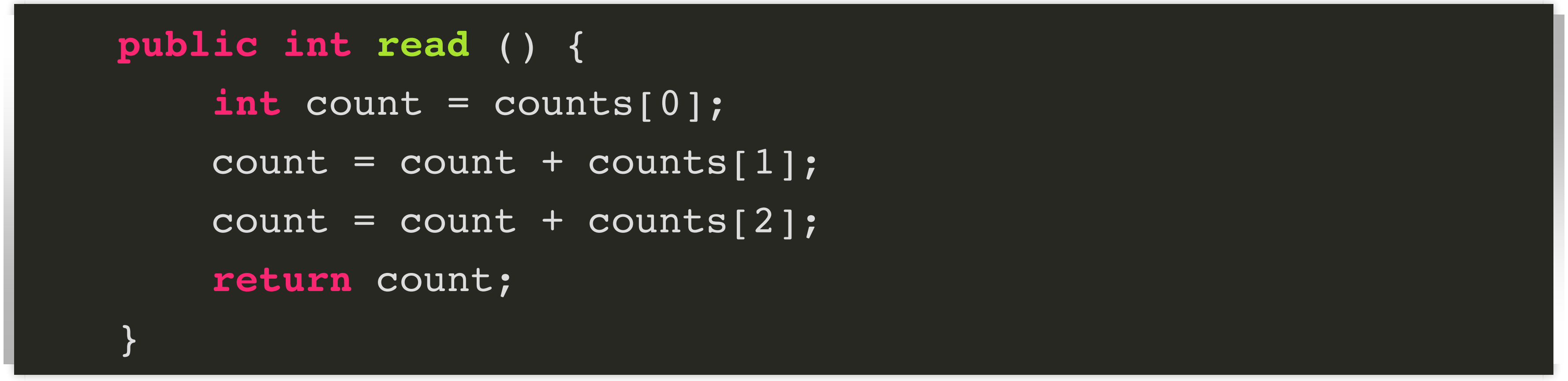
A read Method



```
public int read () {
     int count = counts[0];
     count = count + counts[1];
     count = count + counts[2];
     return count;
}
```

Is ThreeCounter Linearizable?

Writing Between the Lines



```
public int read () {
    int count = counts[0];
    count = count + counts[1];
    count = count + counts[2];
    return count;
}
```

Sequentially Consistency

Questions.

1. Is the previous execution sequentially consistent?
2. Is `ThreeCounter` sequentially consistent?

Next Time

Linearizable Queues!