Practical Concurrent and Parallel Programming 8

Peter Sestoft

IT University of Copenhagen

Friday 2015-10-23**

Plan for today

- More synchronization primitives
  - Semaphore – resource control, bounded buffer
  - CyclicBarrier – thread coordination
- Testing concurrent programs
  - BoundedQueue (FIFO) example
- Coverage and interleaving
- Mutation and fault injection
- Java Pathfinder
- Concurrent correctness concepts

java.util.concurrent.Semaphore

- A semaphore holds zero or more *permits*
- **`void acquire()`**
  - Blocks till a permit is available, then decrements the permit count and returns
- **`void release()`**
  - Increments the permit count and returns; may cause another blocked thread to proceed
  - NB: a thread may call **`release()`** without preceding acquire, so a semaphore is different from a lock!
- A semaphore is used for resource control
  - Locking may be needed for data consistency
- Writes before **`release`** are *visible* after **`acquire`**

A bounded queue using semaphores

TestBoundedQueueTest.java

```
class SemaphoreBoundedQueue <T> implements BoundedQueue<T> {
  private final Semaphore availableItems, availableSpaces;
  private final T[] items;
  private int tail = 0, head = 0;
  public SemaphoreBoundedQueue(int capacity) {
    this.availableItems = new Semaphore(0);
    this.availableSpaces = new Semaphore(capacity);
    this.items = makeArray(capacity);
  }
  public void put(T item) throws InterruptedException { // tail
    availableSpaces.acquire();
    doInsert(item);
    availableItems.release();
  }
  public T take() throws InterruptedException {         // head
    availableItems.acquire();
    T item = doExtract();
    availableSpaces.release();
    return item;
  }
}
```

Wait for space

Signal new item

Wait for item

Signal new space

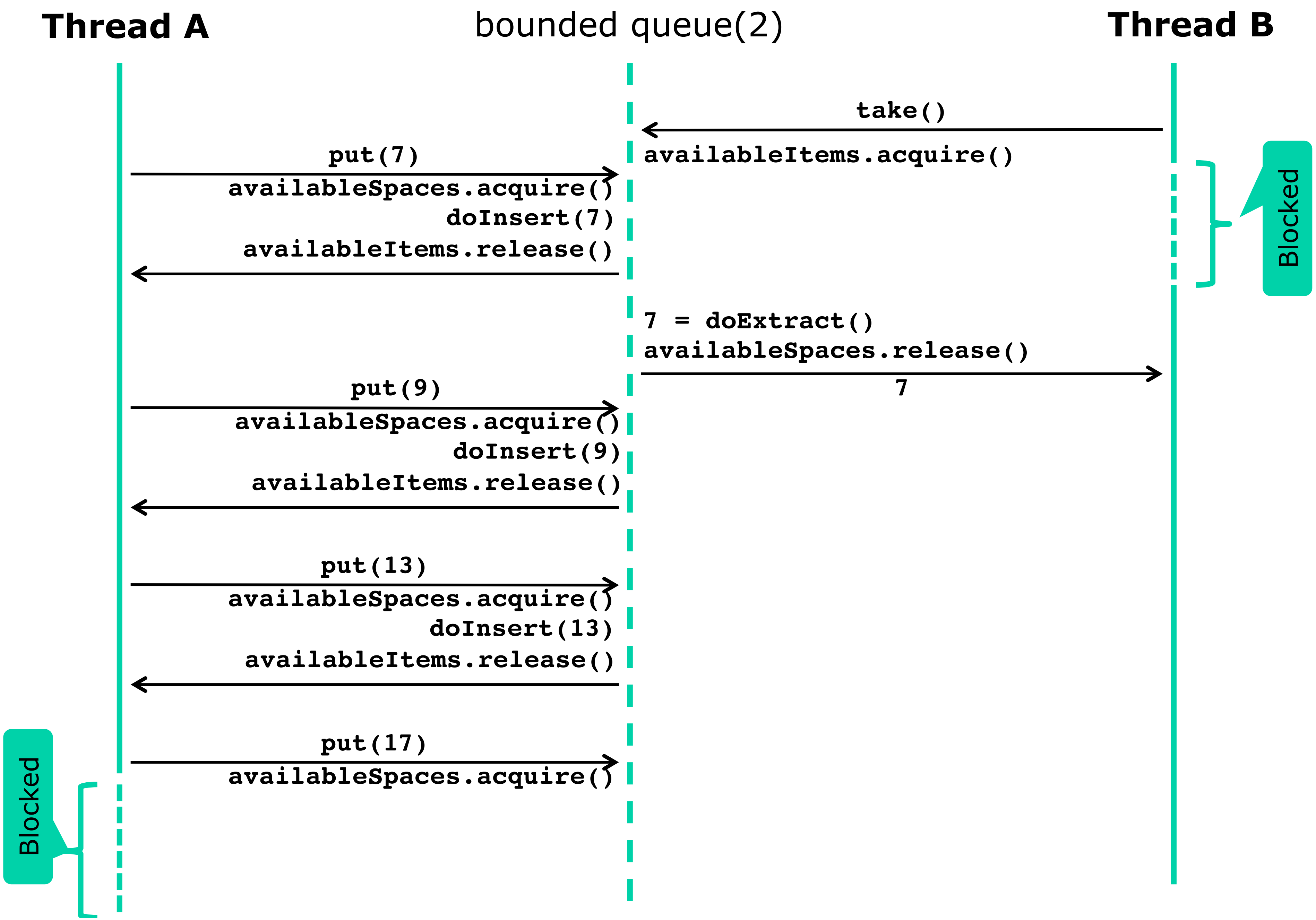
The doInsert and doExtract methods

```java
class SemaphoreBoundedQueue <T> implements BoundedQueue<T> {
  private final Semaphore availableItems, availableSpaces;
  private final T[] items;
  private int tail = 0, head = 0;
  public void put(T item) throws InterruptedException { ... }
  public T take() throws InterruptedException { ... }
  private synchronized void doInsert(T item) {
    items[tail] = item;
    tail = (tail + 1) % items.length;
  }
  private synchronized T doExtract() {
    T item = items[head];
    items[head] = null;
    head = (head + 1) % items.length;
    return item;
  }
}
```

TestBoundedQueueTest.java

- *Semaphores* to block waiting for “resources”
- *Locks* (synchronized) for atomic state mutation

Bounded queue with capacity 2



Thread A

bounded queue(2)

Thread B

take()

availableItems.acquire()

put(7)

availableSpaces.acquire()

doInsert(7)

availableItems.release()

Blocked

7 = doExtract()

availableSpaces.release()

7

put(9)

availableSpaces.acquire()

doInsert(9)

availableItems.release()

put(13)

availableSpaces.acquire()

doInsert(13)

availableItems.release()

put(17)

availableSpaces.acquire()

Blocked

Testing BoundedQueue

- Divide into
  - Sequential 1-thread test with *precise* results
  - Concurrent n-thread test with *aggregate* results
  - ... that make it plausible that invariants hold
- Sequential test for queue **bq** with capacity 3:

```
assertTrue(bq.isEmpty());
assertTrue(!bq.isFull());
bq.put(7); bq.put(9); bq.put(13);
assertTrue(!bq.isEmpty());
assertTrue(bq.isFull());
assertEquals(bq.take(), 7);
assertEquals(bq.take(), 9);
assertEquals(bq.take(), 13);
assertTrue(bq.isEmpty());
assertTrue(!bq.isFull());
```

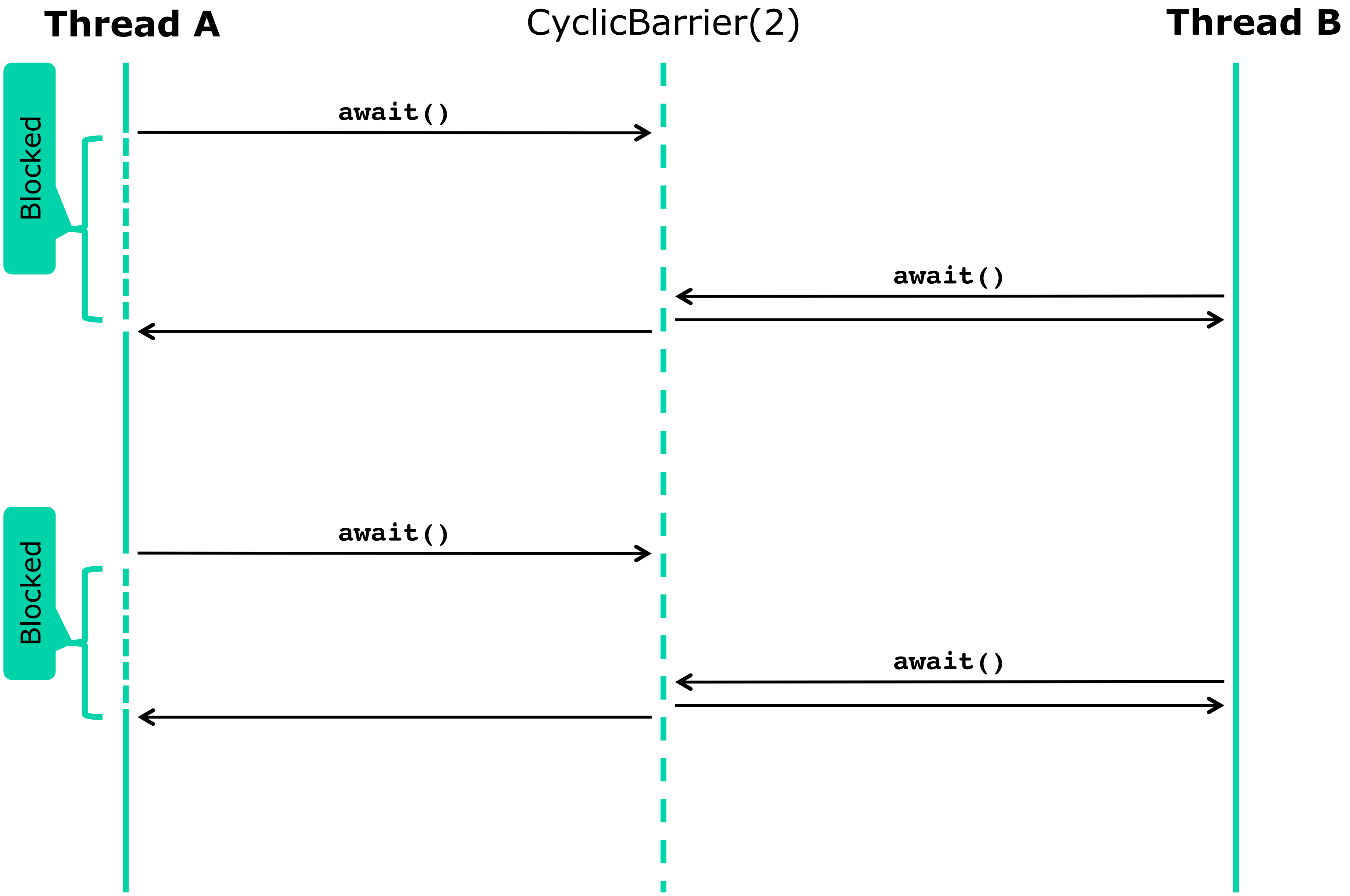
TestBoundedQueueTest.java

java.util.concurrent.CyclicBarrier

- A CyclicBarrier(N) allows N threads
  - to wait for each other, and
  - proceed at the same time when all are ready
- **`int await()`**
  - blocks until all N threads have called await
  - may throw InterruptedException
- Useful to start n test threads + 1 main thread at the same time, N = n + 1

- Writes before **`await`** is called are *visible* after it returns, in all threads passing the barrier

Cyclic barrier with count 2



Concurrent test of BoundedQueue

- Run 10 producer and 10 consumer threads
  - Each producer inserts 100,000 random numbers
    - Using a *thread-local* random number generator
  - Each consumer extracts 100,000 numbers
- Afterwards, check that
  - All consumers terminate, do not block on empty
  - The bounded queue is again empty
  - The sum of consumed numbers equals the sum of produced numbers
- Producers and consumers must sum numbers
  - Using a thread-local sum variable, and afterwards adding to a common AtomicInteger

The PutTakeTest class

```
class PutTakeTest extends Tests {
  protected CyclicBarrier barrier;
  protected final BoundedQueue<Integer> bq;
  protected final int nTrials, nPairs;
  protected final AtomicInteger putSum = new AtomicInteger(0);
  protected final AtomicInteger takeSum = new AtomicInteger(0);

  void test(ExecutorService pool) {
    try {
      for (int i = 0; i < nPairs; i++) {
        pool.execute(new Producer());
        pool.execute(new Consumer());
      }
      barrier.await(); // wait for all threads to be ready
      barrier.await(); // wait for all threads to finish
      assertTrue(bq.isEmpty());
      assertEquals(putSum.get(), takeSum.get());
    } catch (Exception e) {
      throw new RuntimeException(e);
    }
  }
```

Initialize to 2*nPairs+1

Being tested!

Make nPairs Producers and nPairs Consumers

Main: start, finish threads

Check that total effect is plausible

Goetz p. 255

TestBoundedQueueTest.java

A Producer test thread

```java
class Producer implements Runnable {
  public void run() {
    try {
      Random random = new Random();
      int sum = 0;
      barrier.await();
      for (int i = nTrials; i > 0; --i) {
        int item = random.nextInt();
        bq.put(item);
        sum += item;
      }
      putSum.getAndAdd(sum);
      barrier.await();
    } catch (Exception e) {
      throw new RuntimeException(e);
    }
  }
}
```

Thread-local Random

Wait till all are ready

Put 100,000 numbers

Add to global putSum

Signal I’m finished

A la Goetz p. 256

TestBoundedQueueTest.java

A Consumer test thread

```
class Consumer implements Runnable {
  public void run() {
    try {
      barrier.await();
      int sum = 0;
      for (int i = nTrials; i > 0; --i) {
        sum += bq.take();
      }
      takeSum.getAndAdd(sum);
      barrier.await();
    } catch (Exception e) {
      throw new RuntimeException(e);
    }
  }
}
```

Wait till all are ready

Take 100,000 numbers

Add to global takeSum

Signal I’m finished

Goetz p. 256

TestBoundedQueueTest.java

Reflection on the concurrent test

- Checks that *item count* and *item sum* are OK
- The sums say nothing about *item order*
  - Concurrent test would be satisfied by a *stack* also
  - But the sequential test would not
- Could we check better for *item order*?
  - Could use 1 producer, put'ting in increasing order; and 1 consumer take'ing and checking the order
    - But a 1-producer 1-consumer queue may be incorrect for multiple producers or multiple consumers
  - Could make test synchronize between producers and consumers, but
    - Reduces test thread interleaving and thus test efficacy
    - Risk of artificial deadlock because queue synchronizes also

Techniques and hints

- Create a *local random number generator* for each thread, or use ThreadLocalRandom
  - Else may limit concurrency, reduce test efficacy
- Do *no synchronization* between threads
  - May limit concurrency, reduce test efficacy
- Use CyclicBarrier(n+1) to *start* n threads
  - More likely to run at the same time, better testing
- Use it also to wait for the threads to *finish*
  - So main thread can check the results
- Test on a *multicore* machine, 4-16 cores
- Use *more test threads than cores*
  - So some threads occasionally get de-scheduled

Plan for today

- More synchronization primitives
  - Semaphore – resource control, bounded buffer
  - CyclicBarrier – thread coordination
- Testing concurrent programs
  - BoundedBuffer example
- **Coverage and interleaving**
- **Mutation and fault injection**
- Java Pathfinder
- Concurrent correctness concepts

Test coverage

- Sequential
  - *Method coverage*: has each method been called?
  - *Statement coverage*: has each statement been executed?
  - *Branch coverage*: have all branches of **`if`**, **`for`**, **`while`**, **`do-while`**, **`switch`**, **`try-catch`** been executed?
  - *Path coverage*: have all paths through the code been executed? (very unlikely)
- Concurrent
  - *Interleaving coverage*: have all interleavings of different methods’ execution paths been tried? (extremely unlikely)

Thread interleavings

Two threads both doing **`count = count + 1`**:

Thread A: read count; add 1; write count

Thread B: read count; add 1; write count

read count add 1 write count read count add 1 write count	read count add 1 read count write count add 1 write count	read count add 1 read count add 1 write count write count	read count add 1 read count add 1 write count write count	read count read count add 1 write count add 1 write count
read count read count add 1 add 1 write count write count	read count read count add 1 write count add 1 write count	read count read count add 1 add 1 write count write count	read count read count add 1 add 1 write count write count	read count read count add 1 add 1 write count write count

Plus 10 symmetric cases, swapping red and blue

Thread interleaving for testing

- To find concurrency bugs, we want to exercise all interesting thread interleavings
- How many: N threads each with M instructions have $(NM)!/(M!)^N$ possible interleavings
  - Zillions of test runs needed to cover interleavings
- PutTakeTest explores at most 1m of them
  - And JVM may be too deterministic and explore less
- One can increase interleavings using **`Thread.yield()`** or **`Thread.sleep(1)`**
  - But this requires modification of the tested code
  - Or special tools: Java Pathfinder, Microsoft CHESS

What is $(NM)!/(M!)^N$ in real money?

Scala

```
def fac(n: Int): BigInt = if (n==0) 1 else n*fac(n-1)
def power(M: BigInt, P: Int): BigInt = if (P == 0) 1 else M*power(M, P-1)
def interleaving(N : Int, M : Int) = fac(N*M) / power(fac(M), N)
```

Number of ways to interleave N threads each having M instructions

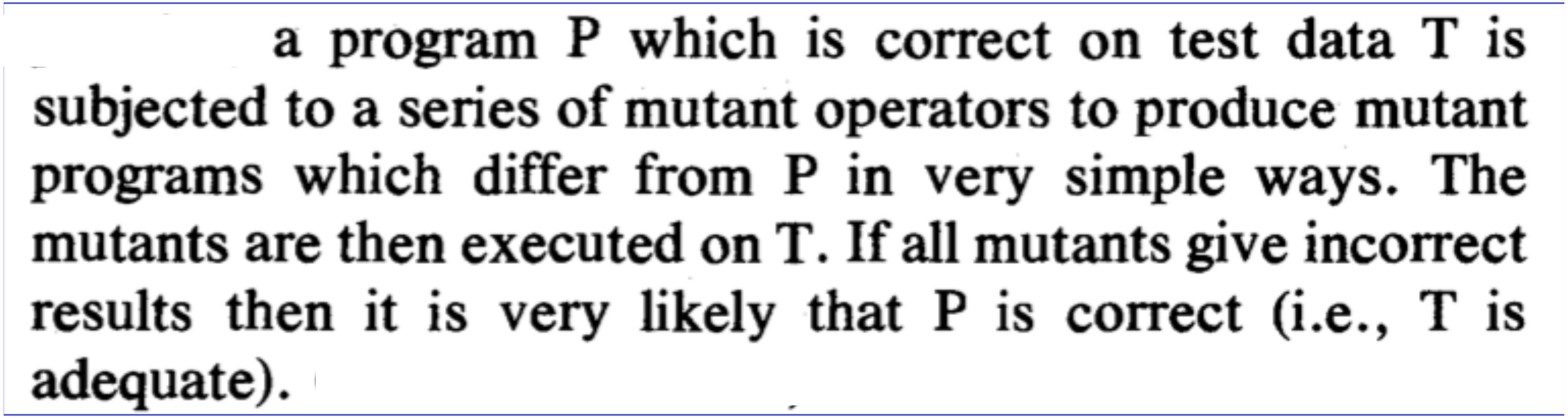
```
interleaving(1, 15) is 1

interleaving(5, 1) is 120

interleaving(5, 2) is 113400

interleaving(2,3) is 20

interleaving(5, 3) is 168168000

interleaving(5, 100) is
17234165594777008534148379284721996814952838615864289522194894697
40322151844673449823990180491172965116996270064140072158794074346
10748311946292872488592584004590960693662608800777663118272422394
64037292765889197732837222228396712117780290598829533989646231081
59928513983125529409127445230866953601595307305816729293520921681
3482694343474336000$
```

How good is that test? Mutation testing and fault injection

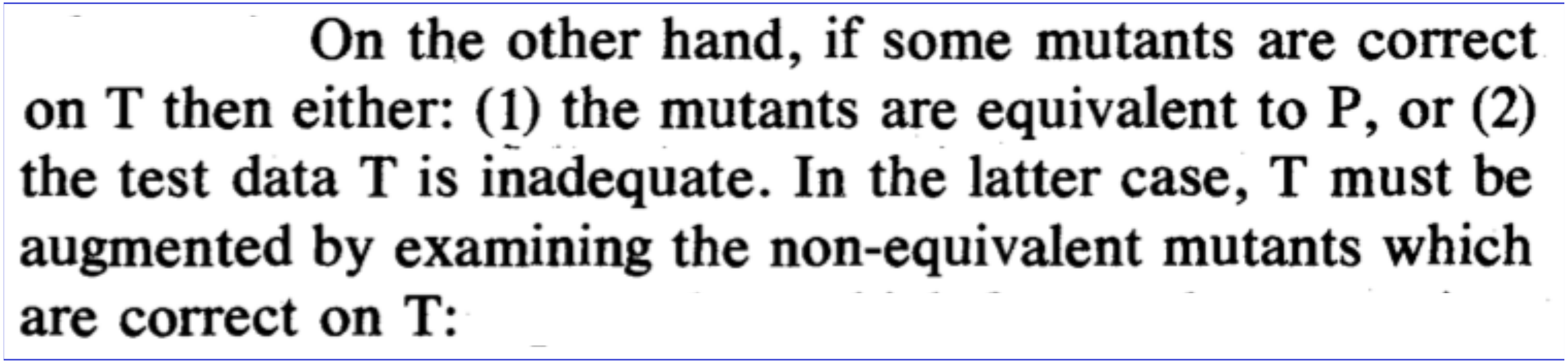
- If some code passes a test,
  - is that because the code is correct?
  - or because the test is too weak: bad coverage?
- To find out, *mutate* the **program**, *inject faults*
  - eg. remove synchronization
  - eg. lock on the wrong object
  - do anything that should make the code not work

- If it still passes the test, the **test** is too weak
  - Improve the test so it finds the code fault

Mutation testing quotes



a program P which is correct on test data T is subjected to a series of mutant operators to produce mutant programs which differ from P in very simple ways. The mutants are then executed on T. If all mutants give incorrect results then it is very likely that P is correct (i.e., T is adequate).



On the other hand, if some mutants are correct on T then either: (1) the mutants are equivalent to P, or (2) the test data T is inadequate. In the latter case, T must be augmented by examining the non-equivalent mutants which are correct on T:

Budd, Lipton, Sayward, DeMillo: The design of a prototype mutation system for software testing, 1978

Some mutations to BoundedQueue

```
public void put(T item) throws InterruptedException { // tail
   availableSpaces.acquire();
   doInsert(item);
   availableItems.release();
}
```

Delete

Insert

```
availableSpaces.release()
```

Delete

```
private synchronized void doInsert(T item) {
   items[tail] = item;
   tail = (tail + 1) % items.length;
}
```

Delete

```
private synchronized T doExtract() {
     T item = items[head];
     items[head] = null;
     head = (head + 1) % items.length;
     return item;
   }
```

Delete

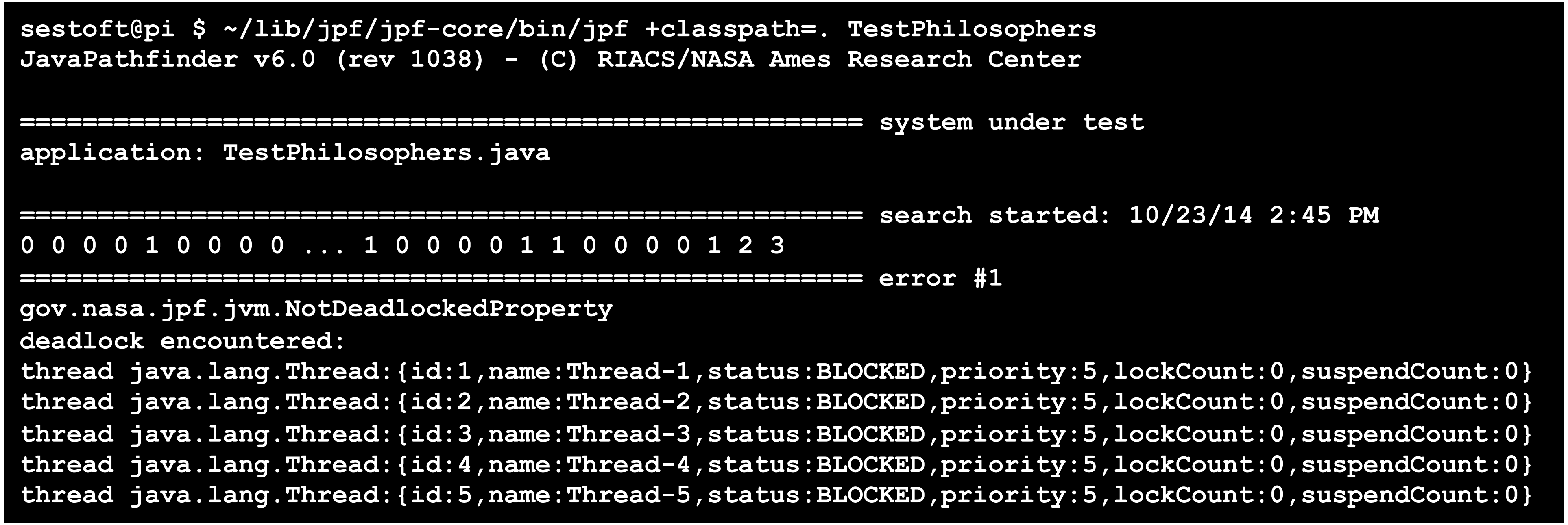
Delete

The Java Pathfinder tool

- NASA project at http://babelfish.arc.nasa.gov/trac/jpf
- A Java Virtual Machine that
  - can explore all computation paths
  - supervise the execution with “listeners”
  - generate test cases
- Properties of Java Pathfinder
  - a multifaceted research project
  - slow execution of code
  - much better test coverage, eg deadlock detection
  - only works for Java 7, not 8, so far

Java Pathfinder example

- TestPhilosophers on 1 core never deadlocks
  - at least not within the bounds of my patience ...
- But Java Pathfinder discovers a deadlock
  - because it explores many thread interleavings



```
sestoft@pi $ ~/lib/jpf/jpf-core/bin/jpf +classpath=. TestPhilosophers
JavaPathfinder v6.0 (rev 1038) - (C) RIACS/NASA Ames Research Center

======================================================= system under test
application: TestPhilosophers.java

======================================================= search started: 10/23/14 2:45 PM
0 0 0 0 1 0 0 0 0 ... 1 0 0 0 0 1 1 0 0 0 0 1 2 3
======================================================= error #1
gov.nasa.jpf.jvm.NotDeadlockedProperty
deadlock encountered:
thread java.lang.Thread:{id:1,name:Thread-1,status:BLOCKED,priority:5,lockCount:0,suspendCount:0}
thread java.lang.Thread:{id:2,name:Thread-2,status:BLOCKED,priority:5,lockCount:0,suspendCount:0}
thread java.lang.Thread:{id:3,name:Thread-3,status:BLOCKED,priority:5,lockCount:0,suspendCount:0}
thread java.lang.Thread:{id:4,name:Thread-4,status:BLOCKED,priority:5,lockCount:0,suspendCount:0}
thread java.lang.Thread:{id:5,name:Thread-5,status:BLOCKED,priority:5,lockCount:0,suspendCount:0}
```

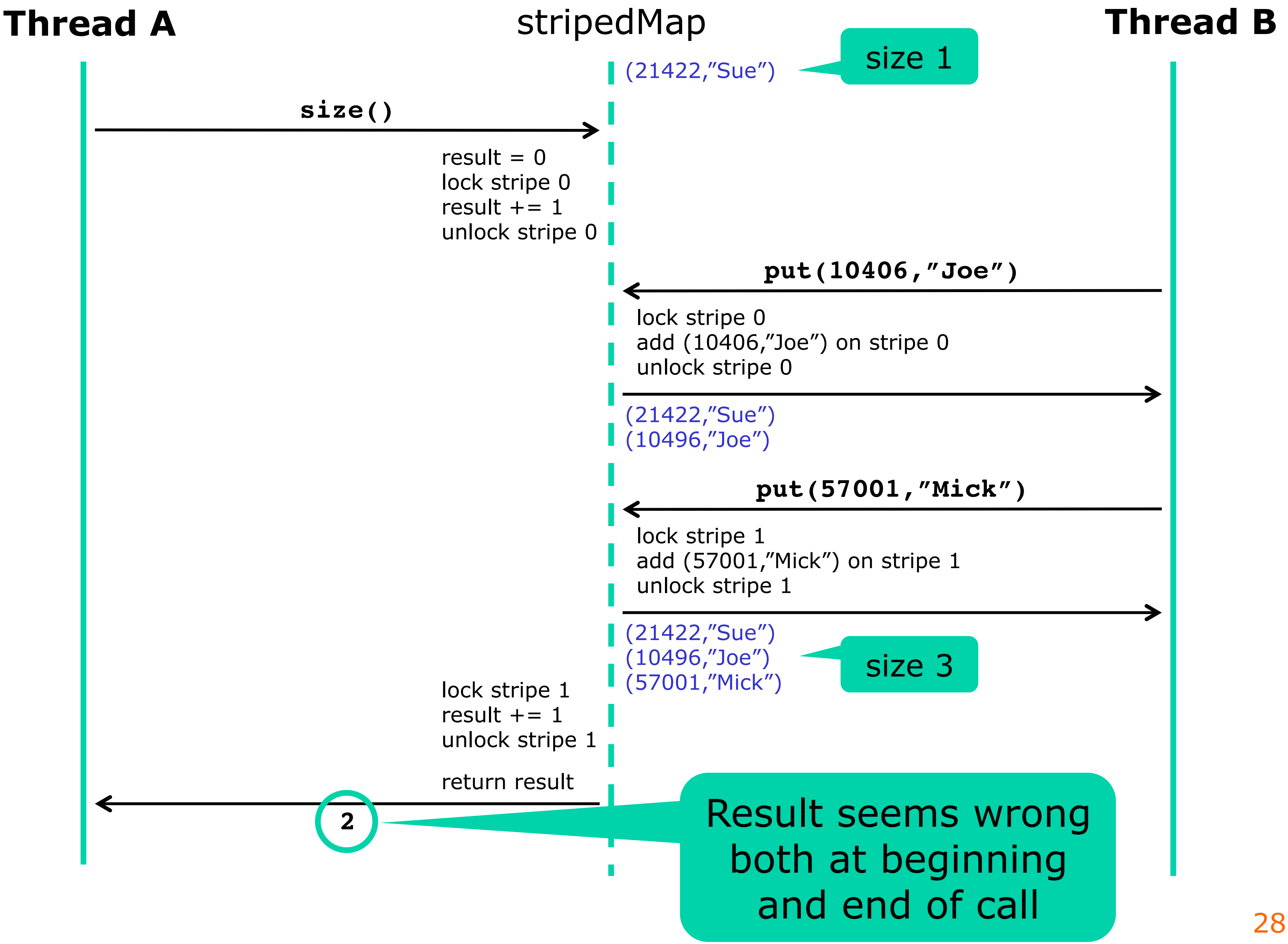
Plan for today

- More synchronization primitives
  - Semaphore – resource control, bounded buffer
  - CyclicBarrier – thread coordination
- Testing concurrent programs
  - BoundedBuffer example
- Coverage and interleaving
- Mutation and fault injection
- Java Pathfinder
- **Concurrent correctness concepts**

Correctness concepts

- Quiescent consistency
  - *Method calls separated by a period of quiescence should appear to take effect in their real-time order*
  - Says nothing about overlapping method calls
- Sequential consistency
  - *Method calls should appear to take effect in program order* – seen from each thread
- Linearizability
  - *A method call should appear to take effect at some point between its invocation and return*
  - This is called its *linearization point*

When is StripedMap.size() correct?



Result seems wrong both at beginning and end of call

Quiescent consistency

- Principle 3.3.2: *Method calls separated by a period of quiescence should appear to take effect in their real-time order*
  - This says nothing about overlapping method calls
  - This assumes we can observe inter-thread actions
- Java’s ConcurrentHashMap:

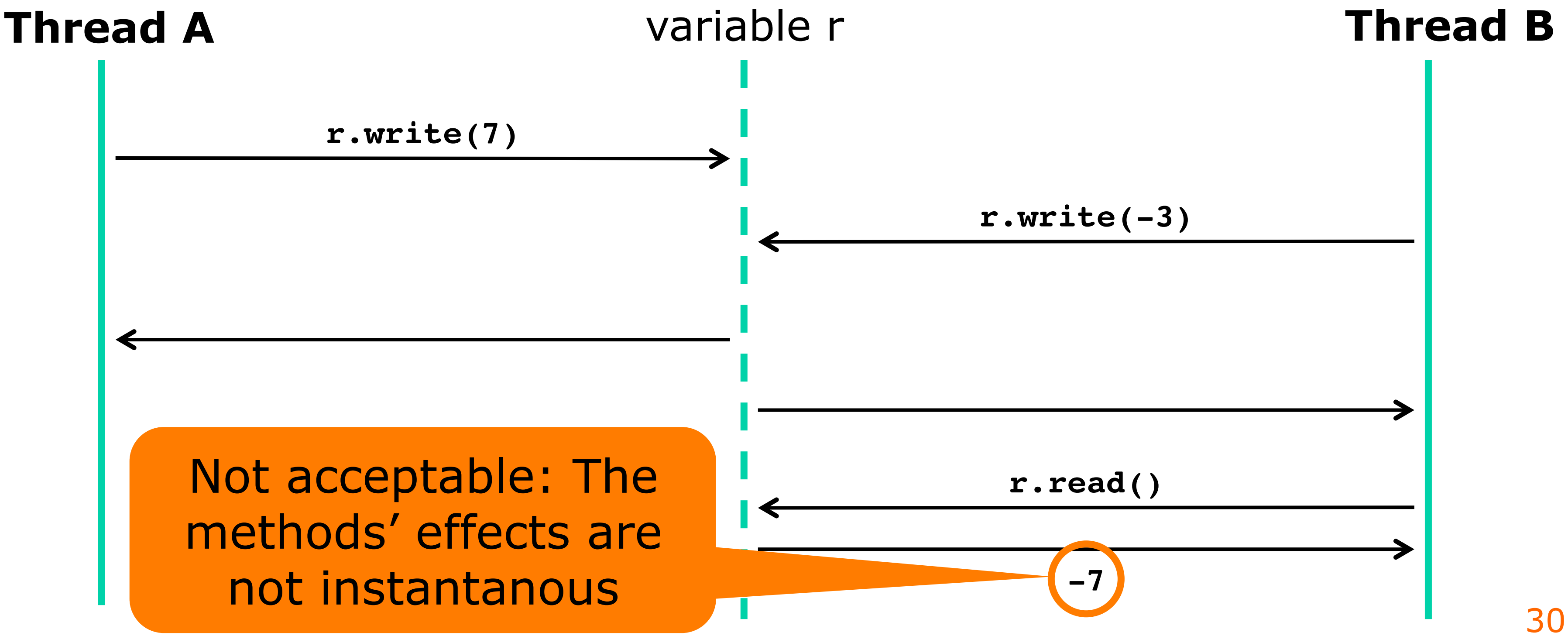
> “Bear in mind that the results of aggregate status methods including **`size`**, **`isEmpty`**, and **`containsValue`** are typically useful only when a map is not undergoing concurrent updates in other threads.
>
> Otherwise the results of these methods reflect transient states that may be adequate for monitoring or estimation purposes, but not for program control.”

Class java.util.concurrent.ConcurrentHashMap documentation

Method call effect must seem instantaneous

- Principle 3.3.1: *A method call should appear to take effect instantaneously*
  - Method calls take effect one at a time, even when they overlap

Herlihy & Shavit p. 50



Not acceptable: The methods' effects are not instantanous

Non-blocking queue example code

A la Herlihy & Shavit p. 46, 48

TestHSQueues.java

```
class LockBasedQueue<T> {
  private final T[] items;
  private           int tail = 0, head = 0;
  public synchronized boolean enq(T item) {
    if (tail - head == items.length)
      return false;
    else {
      items[tail % items.length] = item;
      tail++;
      return true;
  } }
  public synchronized T deq() {
    if (tail == head)
      return null;
    else {
      T item = items[head % items.length];
      head++;
      return item;
} } }
```

Cannot overlap

- With locking, state changes cannot overlap
  - Method call “takes effect” when releasing the lock, at return

Restricted-use queue without locks

A la Herlihy & Shavit p. 46, 48

TestHSQueues.java

```
class WaitFreeQueue<T> {
  private final T[] items;
  private volatile int tail = 0, head = 0;
  public              boolean enq(T item) {
    if (tail - head == items.length)
      return false;
    else {
      items[tail % items.length] = item;
      tail++;
      return true;
  } }
  public              T deq() {
    if (tail == head)
      return null;
    else {
      T item = items[head % items.length];
      head++;
      return item;
 } } }
```

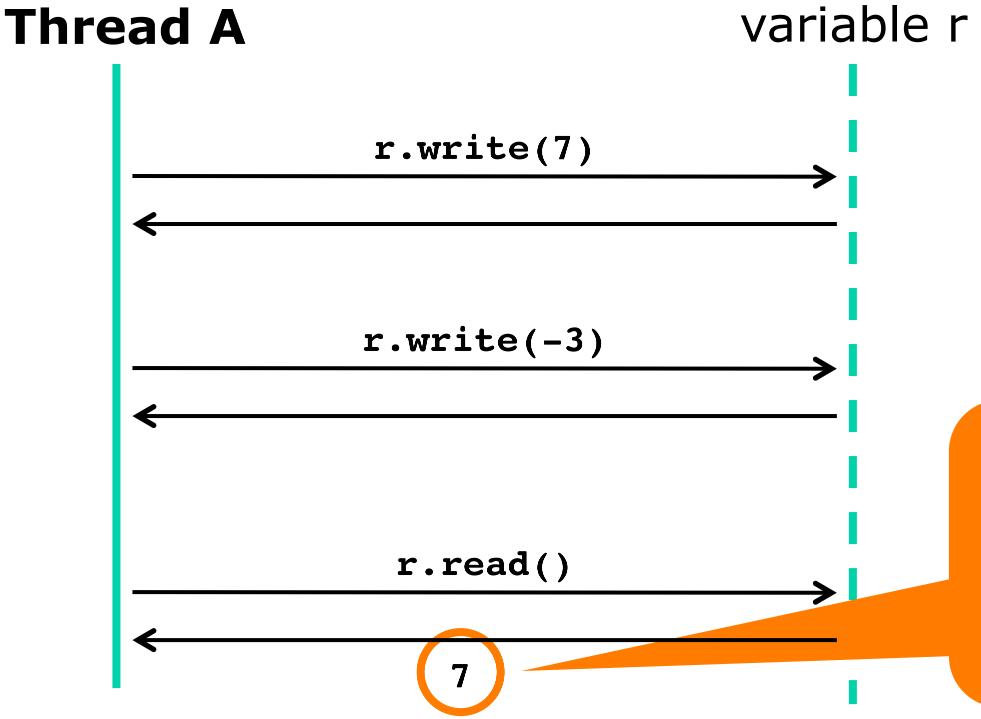
- Correct if there is only one enqueuer and one dequeuer!
- Only **enq** writes **tail**
- Only **deq** writes **head**
- **enq** and **deq** never write same **items[i]**
- Visibility ensured by **volatile**
- Subtle …

- No locking, so state updates may overlap!
  - One thread calling **enq**, another calling **deq**
  - Now what would it mean for WaitFreeQueue to be “correct”?

Sequential consistency

- Principle 3.4.1: *Method calls should appear to take effect in program order*
  - Program order is the *order within a single thread*
- The full execution is an interleaving of each thread's executions – seems natural!!

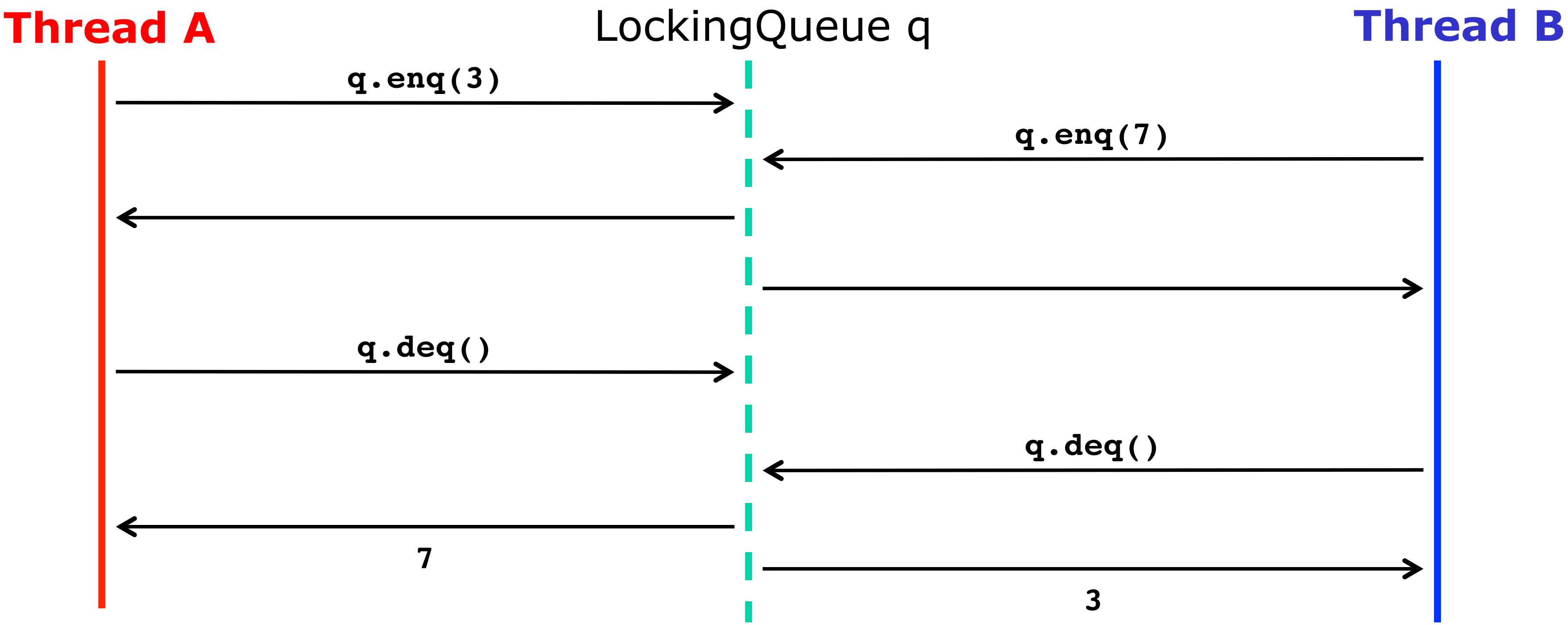
Herlihy & Shavit p. 52



Not acceptable: Calls do not take effect in program order (seen from thread A)

Sequentially consistent scenarios for LockingQueue

Herlihy & Shavit p. 52

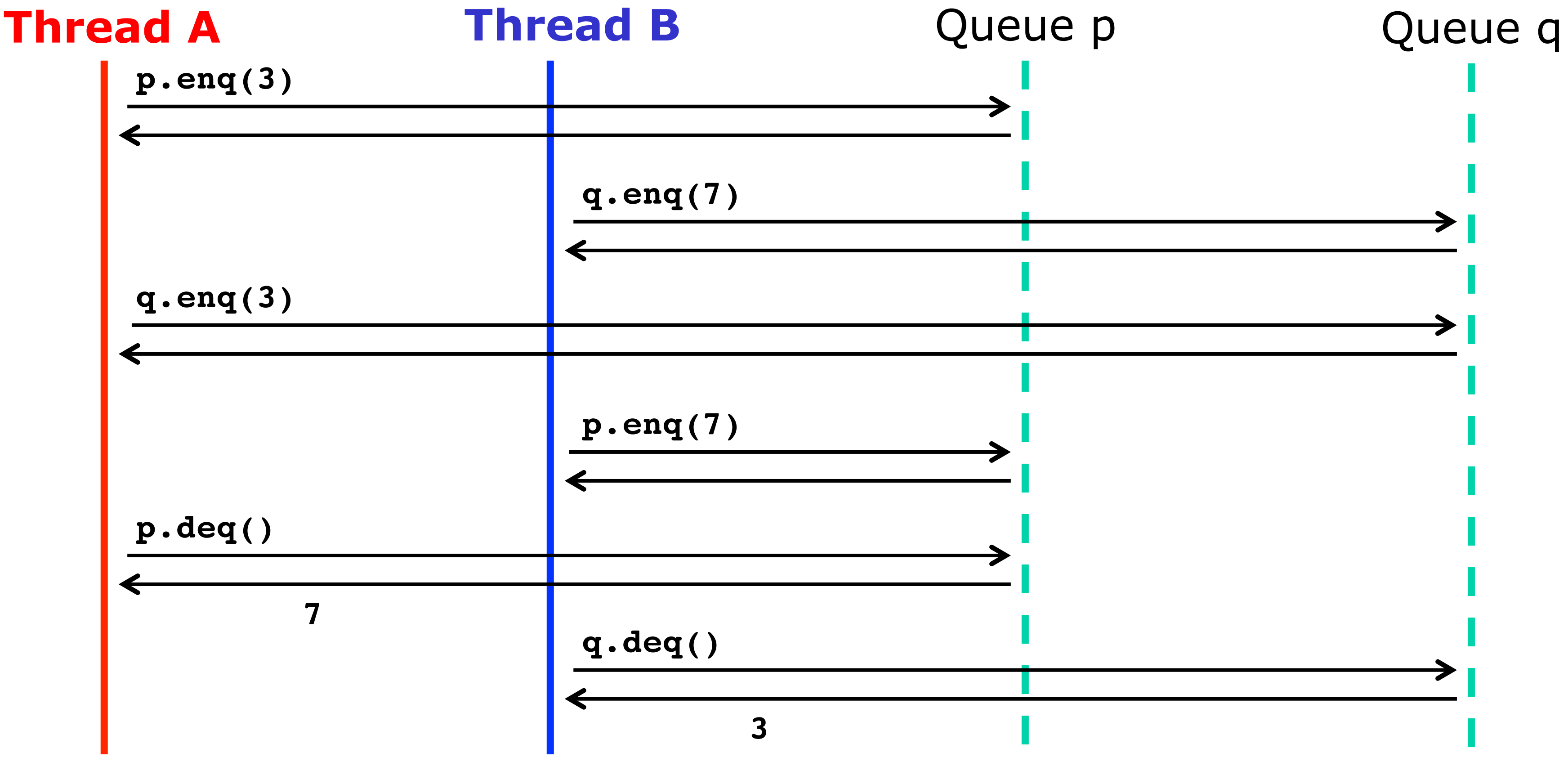


Two "global" scenarios showing seq cons:

A q.enq(3)
B q.enq(7)
B q.deq(3)
A q.deq(7)

B q.enq(7)
A q.enq(3)
A q.deq(7)
B q.deq(3)

Separately not jointly seq. cons.



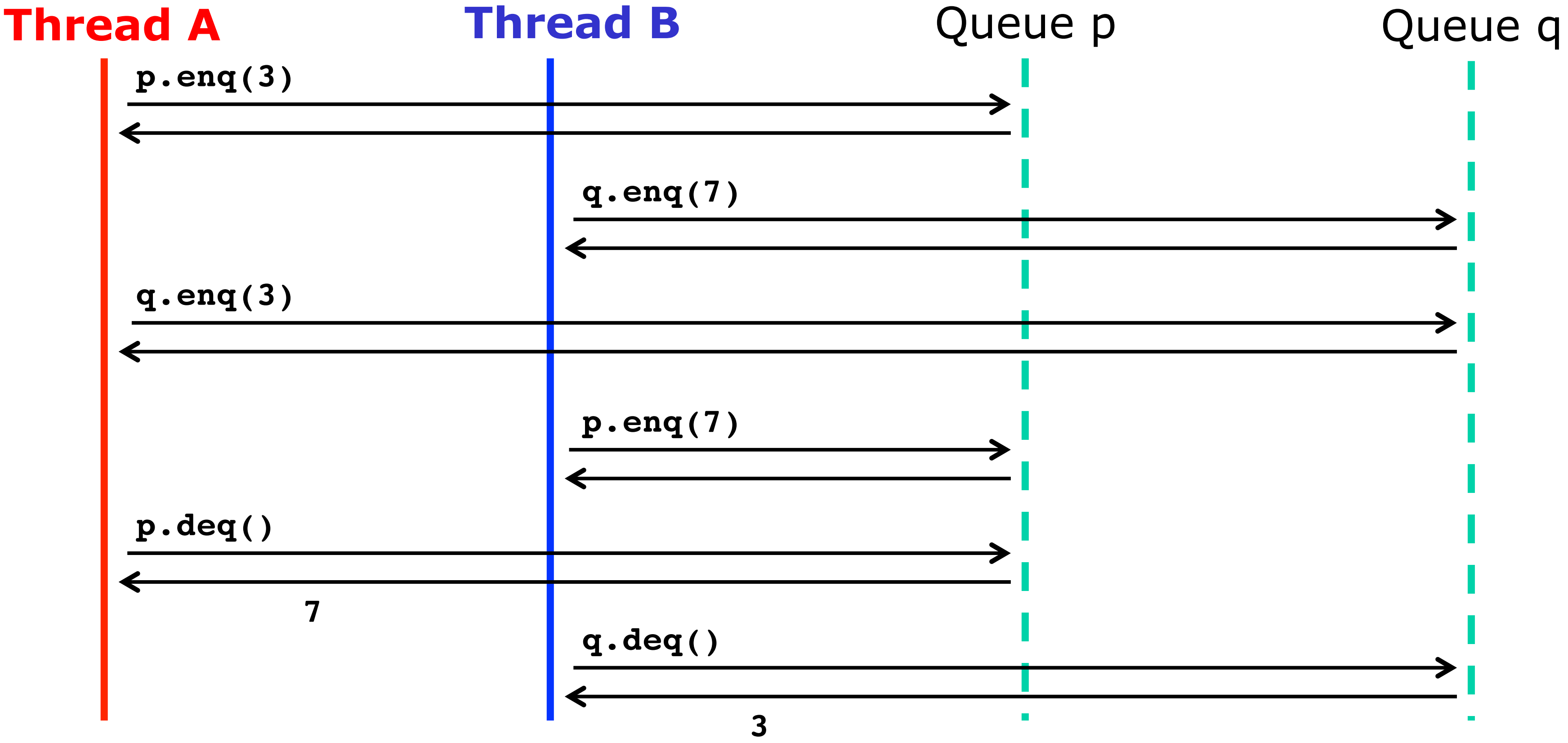
Herlihy & Shavit p. 54

- Sequentially consistent for each queue p, q:

B p.enq(7) A p.enq(3) A p.deq(7)	AND	A q.enq(3) B q.enq(7) B q.deq(3)

BUT assume different orders of p.enq(7) and p.enq(3)

Cannot be jointly seq. consistent



Herlihy & Shavit p. 54

- p.enq(7) must precede p.enq(3) because dequeues 7
  - which precedes q.enq(3) in thread A program order
- q.enq(3) must precede q.enq(7) because dequeues 3
  - which precedes p.enq(7) in thread B program order
- So p.enq(7) must precede p.enq(7), *impossible*

Reflection on sequential consistency

- Seems a natural expectation
- It is what synchronization tries to achieve
- If all (unsynchronized) code were to satisfy it, that would preclude optimizations:

Java, or C#, does not guarantee sequential consistency of accesses to non-synchronized non-volatile fields (eg. JLS §17.4.3)

- The lack of compositionality makes sequential consistency a poor reasoning tool
  - Using a bunch of sequentially consistent data structures together does not give seq. consistency

Linearizability

- Principle 3.5.1: *Each method call should appear to take effect instantaneously at some moment between its invocation and response.*
- Usually shown by identifying a *linearization point* for each method.

- In Java monitor pattern methods, the linearization point is typically at lock release
- In non-locking WaitFreeQueue<T>
  - linearization point of **`enq()`** is at **`tail++`** update
  - linearization point of **`deq()`** is at **`head++`** update
- Less clear in lock-free methods, week 11-12

TestHSQueues.java

Restricted-use queue without locks

A la Herlihy & Shavit p. 46, 48

TestHSQueues.java

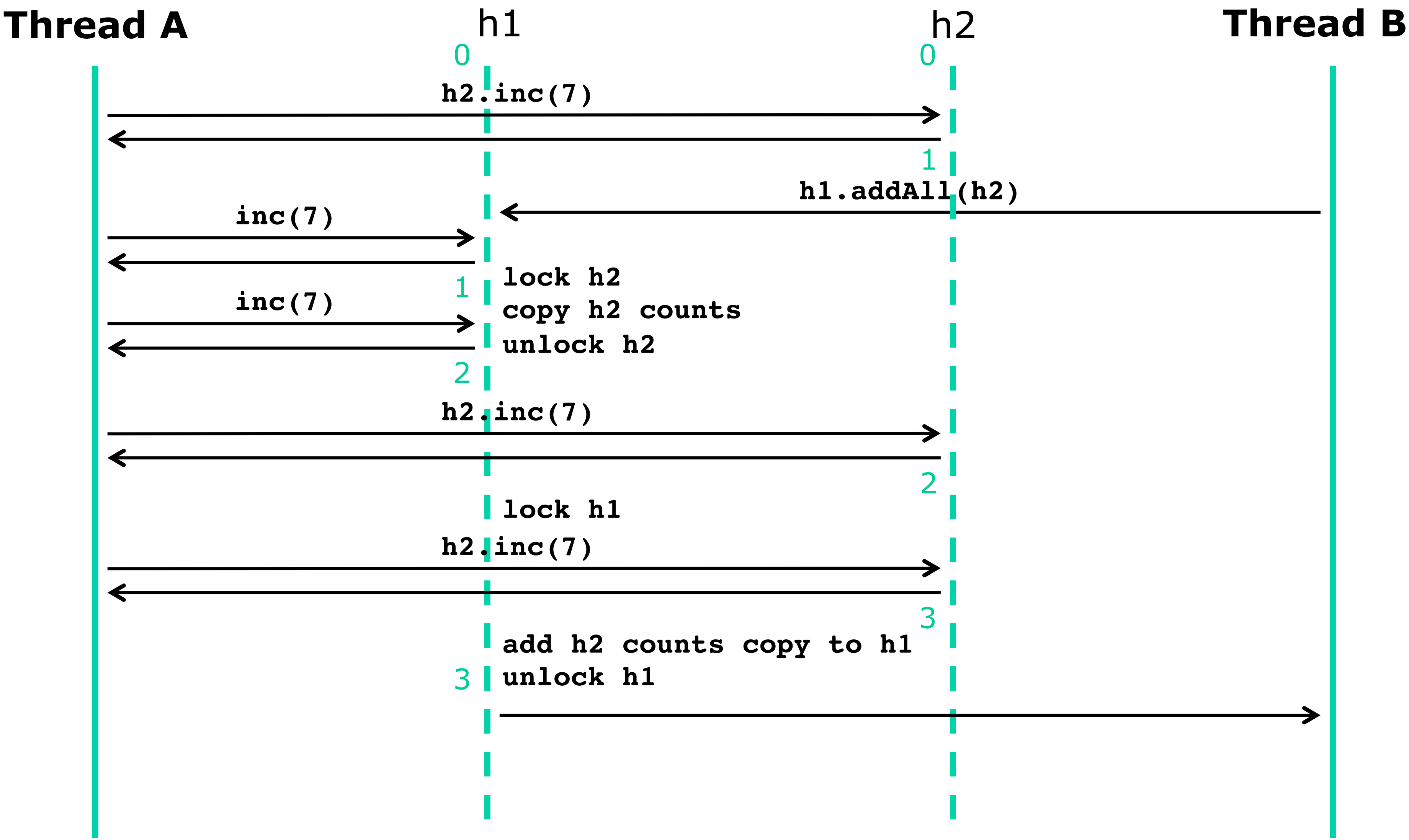
```
class WaitFreeQueue<T> {
  private final T[] items;
  private volatile int tail = 0, head = 0;
  public              boolean enq(T item) {
    if (tail - head == items.length)
      return false;
    else {
      items[tail % items.length] = item;
      tail++;
      return true;
  } }
  public              T deq() {
    if (tail == head)
      return null;
    else {
      T item = items[head % items.length];
      head++;
      return item;
 } } }
```

Linearization point

Linearization point

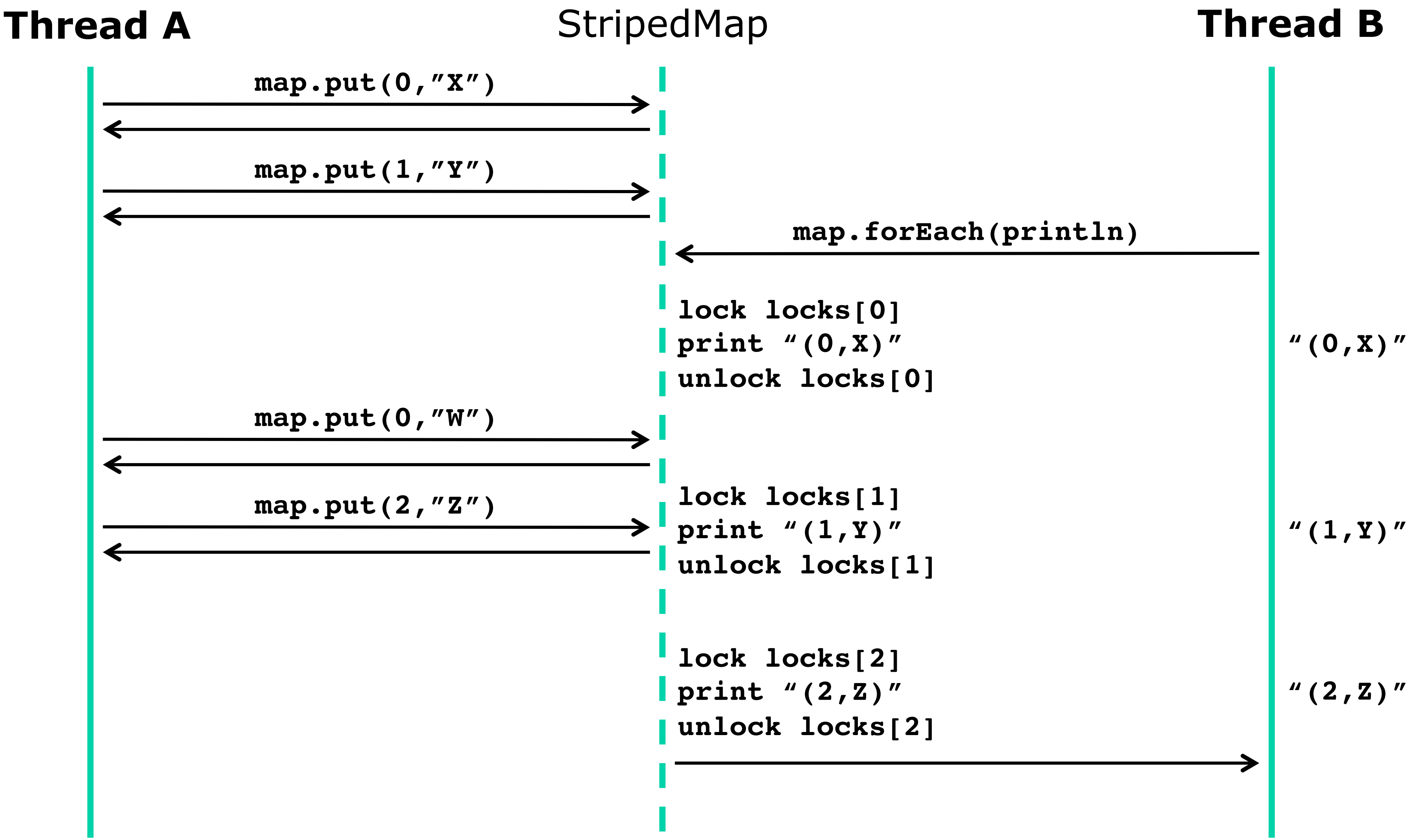
- NB: Only one enqueuer and one dequeuer thread!

A Histogram h1.addAll(h2) scenario



The result does not reflect the joint state of h1 and h2 at any point in time. (Because h1 may be updated while h2 is locked, and vice versa).

A StripedMap.forEach scenario



Thread A StripedMap **Thread B**

```
map.put(0,″X″)
map.put(1,″Y″)
map.forEach(println)
lock locks[0]
print “(0,X)″
unlock locks[0]
“(0,X)″
map.put(0,″W″)
map.put(2,″Z″)
lock locks[1]
print “(1,Y)″
unlock locks[1]
“(1,Y)″
lock locks[2]
print “(2,Z)″
unlock locks[2]
“(2,Z)″
```

Seen from Thread A it is strange that (2,Z) is in the map but not (0,W).
(Stripe 0 is enumerated before stripe 2, and stripe 1 updated in between).

Concurrent bulk operations

- These typically have rather vague semantics:

> "Iterators and Spliterators provide *weakly consistent* [...] traversal:
> - they may proceed concurrently with other operations
> - they will never throw ConcurrentModificationException
> - they are guaranteed to traverse elements as they existed upon construction exactly once, and may (but are not guaranteed to) reflect any modifications subsequent to construction"

Package java.util.concurrent documentation

- The three bullets hold for StripedMap.forEach
- Precise test only in quiescent conditions
  - But (a) it does not skip entries that existed at call time, and (b) it does not process an entry twice

This week

- Reading
  - Goetz et al chapter 12
  - Herlihy & Shavit chapter 3 (PDF on LearnIT)

- Exercises
  - Show you can test concurrent software with subtle synchronization mechanisms

- Read before next week's lecture
  - Goetz chapter 10, 13.1, 16
  - Bloch item 67