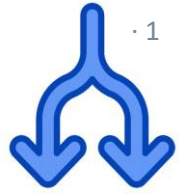
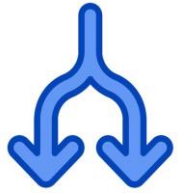


Practical Concurrent and Parallel Programming XIV
End of course / Exam

Raúl Pardo and
Jørgen Staunstrup

Examination – Material



- The folder `exam` in the GitHub repository contains
  - *The mandatory readings for the exam* (we can ask questions about any of these readings)
  - *Questions for the exam*

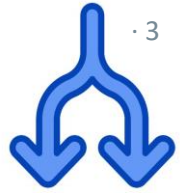
  Although *the list is preliminary and subject to change*, you can consider this an *almost final version*

- Please **read the list with mandatory reading and exam questions carefully** and ask for any clarifications/comments
  - **Send questions and/or topics to revisitto Raúl (raup@itu.dk) before Thursday Nov 30th**

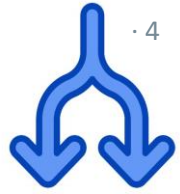
- Week 14 will be mostly about addressing your question/comments

- *Questions and answers in the LearnIT forum are not part of the mandatory readings*
  - The Q&A forum will be closed soon after we finish the course

Agenda

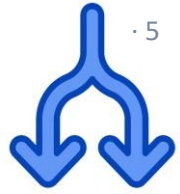


- Important concepts you need to know well (no matter the question)
- Race conditions vs data races, and thread-safety
- Checking that a class is thread-safe
- Testing concurrent programs
- Happens-before
- Linearizability
- Work-stealing queues
- Final remarks



Important concepts you need to know well (no matter the question)

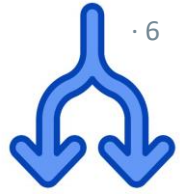
Threads in Java – Example II



- What was is the problem in the previous program?

- To answer this question we need to understand
  - Atomicity
  - States of a thread
  - Non-determinism
  - Interleavings

Atomicity



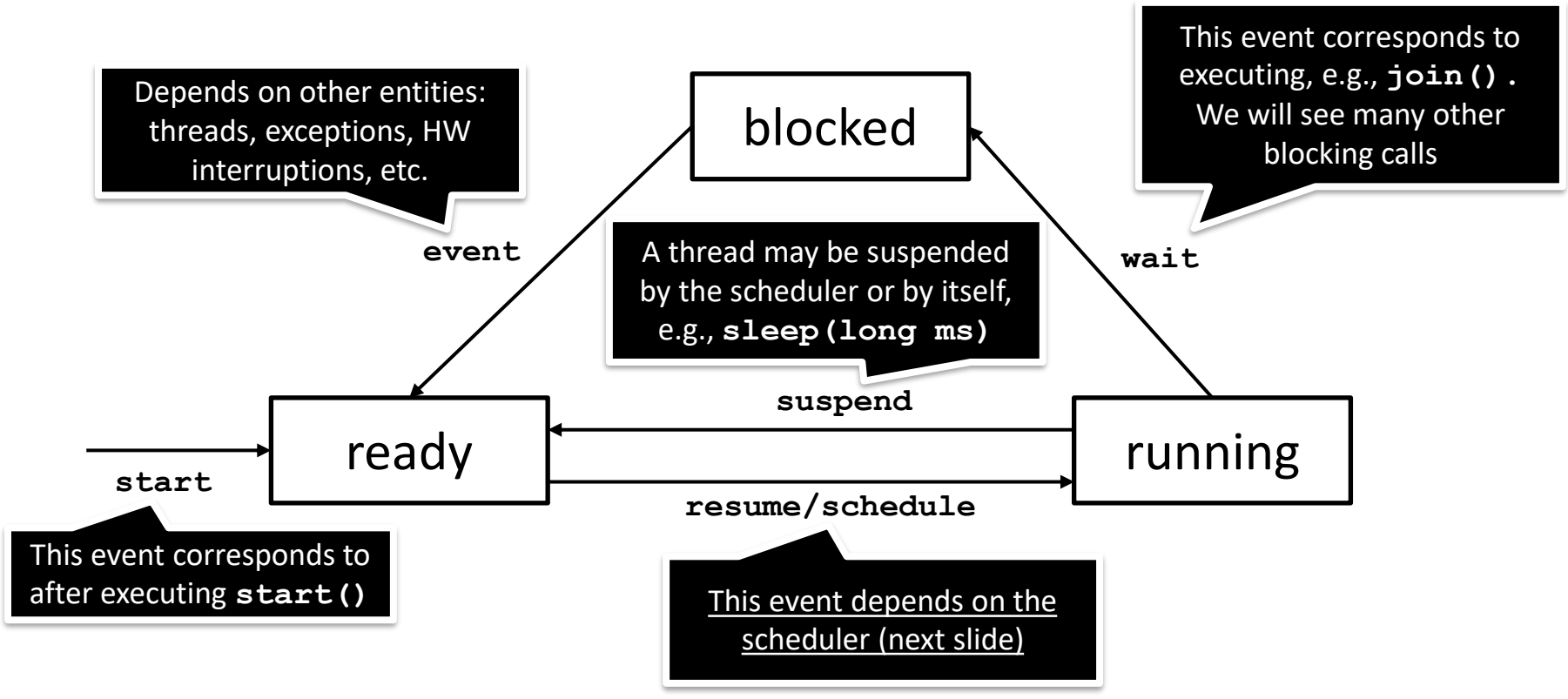
- The program statement **`counter++`** is not *atomic*
- *Atomic statements are executed as a single (indivisible) operation*

```
public class Turnstile extends Thread {
   public void run() {
        for (int i = 0; i < PEOPLE; i++) {
            counter++;
        }
   }
}
```

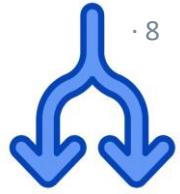
```
int temp = counter;
counter = temp + 1;
```

Watchout: Just because a program statement is a one-liner, it doesn´t mean that it is atomic

States of a thread (simplified)



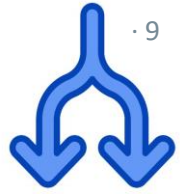
Non-determinism | Scheduler



- In all operating systems/executing environments a *scheduler* selects the processes/threads under execution
  - Threads are selected *non-deterministically*, i.e., no assumptions can be made about what thread will be executed next

- Consider two threads t1 and t2 in the ready state; *t1(ready)* and *t2(ready)*
  1. *t1(running) -> t1(ready) -> t1(running) -> t1(ready) -> …*
  2. *t2(running) -> t2(ready) -> t2(running) -> t2(ready) -> …*
  3. *t1(running) -> t1(ready) -> t2(running) -> t2(ready) -> …*
  4. Infinitely many different executions!

Interleaving



- The statements in a thread are executed when the thread is in its “running” state

- An *interleaving* is a possible sequence of operations for a concurrent program
  - Note this: a sequence of operations for a concurrent program, not for a thread. Concurrent programs are composed by 2 or more threads.

Interleaving syntax

- The drawings above are not suitable for thinking about possible interleavings
- When asked to provide an interleaving, use the following syntax

```
<thread>(<step>), <thread>(<step>), …
```

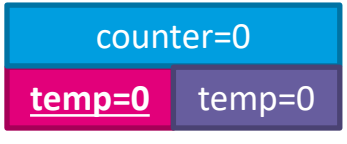
Interleaving – Example II (textual)

Given the initial memory state on the right, provide an interleaving such that after two threads t1, t2 execute the program on the right the value of ***counter==1***

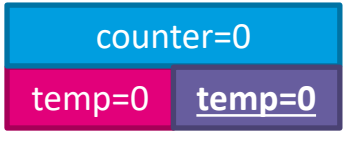
Memory

Program

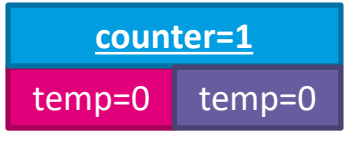
```
public void run() {
  int temp = counter;  // (1)
  counter = temp + 1;  // (2)
}
```



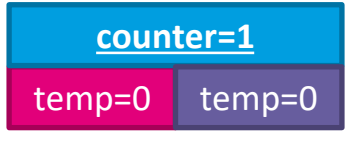
t1(1),



t2(1),



t1(2),



t2(2)

Race conditions, data races & Thread-safety (also very important no matter the question)

Race Conditions

- *A **race condition** occurs when the result of the computation depends on the interleavings of the operations*

Data Races (refined)

- *A **data race** occurs when two concurrent threads:*
  - *Access a shared memory location*
  - *At least one access is a write*
  - *There is no happens-before relation between the accesses*

Inspired by the Java memory model (JLS): *"A program is correctly synchronized if and only if all sequentially consistent executions are free of data races."*

Race Conditions vs Data Races

Not all race conditions are data races

- Threads may not access shared memory
- Threads may not write on shared memory

```
public void p() {
  print("P") //P1
}
```

```
public void q() {
  print("Q") //Q1
}
```

```
Interleaving 1: T1(P1)T2(Q1)
Output: P Q
Interleaving 2: T1(Q1)T2(P1)
Output: Q P
```

Not all data races result in race conditions

- The result of the program may not change based on the writes of threads

```
public void p() {
  x=1;   //P1
}
```

```
public void q() {
  x=1;   //Q1
}
```

```
Interleaving 1: T1(P1)T2(Q1)
Final state: x==1
Interleaving 2: T1(Q1)T2(P1)
Final state: x==1
```

Thread-safe program

*A concurrent **program** is said to be **thread-safe** if and only if it is race condition free*

Do not confuse thread-safe classes with thread-safe programs. Thread-safe programs are not defined in Goetz. But it is aligned with the definition of correctly synchronized programs in JLS

PCPP teaching team

Thread-safe class

IMPORTANT: In this course, *thread-safety* is not an umbrella term for code that seem to behave correctly in concurrent environments.

Inspired by the Java memory model (JLS): *"A program is correctly synchronized if and only if all sequentially consistent executions are free of data races."*

*A **class** is said to be **thread-safe** if and only if*
no concurrent execution of
method calls or field accesses (read/write)
result in data races on the fields of the class

Note that this definition is independent of class invariants as opposed to Goetz Chapter 4. This definition is more similar to Goetz Chapter 2, page 18.

PCPP teaching team

Critical sections

- A *critical section* is a part of the program that only one thread can execute at the same time
  - Useful to avoid race conditions in concurrent programs

```
public class Turnstile extends Thread {
   public void run() {
        for (int i = 0; i < PEOPLE; i++) {
            // start critical section
            int temp = counter;
            counter = temp + 1;
            // end critical section
        }
   }
}
```

Critical sections should cover the parts of the code handling shared memory

Mutual Exclusion

- An ideal solution to the mutual exclusion problem must ensure the following properties:
  - Mutual exclusion: at most one thread executing the critical section at the same time
  - Absence of *deadlock*: threads eventually exit the critical section allowing other threads to enter
  - Absence of *starvation*: if a thread is ready to enter the critical section, it must eventually do so

Thread-safe classes

- To analyse whether a class is thread-safe, we must identify/consider:
  - Class state
  - Escaping
  - (Safe) publication
  - Immutability
  - **Mutual exclusion**

Very important slide

Use as a reference when answering questions about thread-safety

Class state

- By definition, (uncontrolled) concurrent access to the shared state (variables) leads to data races
- So, the first thing we need to do is to identify the fields that may be shared by several threads
- The state of a class involves the fields defined in the class
  - In a nutshell, our goal is to ensure that concurrent access to class state is free from data races

```
class C {
     // class state (variables)
     T s1;
     T s2;
     T s3;
     T s4;
     …

     // class methods
     T m1(…) {…}
     T m2(…) {…}
     T m3(…) {…}
     …
}
```

Escaping

- It is important to not expose shared state variables
- Otherwise, threads may use them without ensuring mutual exclusion
  - Thus, we cannot enforce a happens-before relation
- Defining all (shared) class state (primitive) variables as private ensures that these variables will only be accessed through public methods.
  - Thus, it is easier to control and reason about concurrent access

```
class Counter {
     // class state (variables)
     int i=0;

     // class methods
     public synchronized void inc(){i++;}
}
```

```
// program using Counter

Counter c = new Counter();
new Thread(() -> {
     c.inc();
}).start();

new Thread(() -> {
     c.i++; // escaped the lock in inc()
}).start();
```

Escaping

- Remember that when a method returns an object, we get a *reference* to that object

```
class IntArrayList {
    // class state
    private List<Integer> a = new ArrayList<Integer>();

    public synchronized void set(Integer index, Integer elem)
        {  a.set(index,elem); }

    public synchronized List<Integer> get() { return a; }
}
```

- Therefore, even if obtain the reference using locks, later we can modify the content of the object without locks

```
IntArrayList array = new IntArrayList();
new Thread(() -> {
    array.set(0,1); // access state with lock
}).start();
new Thread(() -> {
    array.get().set(0,42); // access state without locks
}).start();
```

What about collections of non-primitive types?

- If the list contains non-primitive types (classes), thread-safety is delegated to the implementation of the class
  - The class (for non-primitive types) must be thread-safe

```
class TArrayList {
    // class state
    private List<Integer> a = new ArrayList<T>();

    public synchronized void get(T index)
        {  a.get(index); }
}
```

```
TArrayList array = new TArrayList();
new Thread(() -> {
    array.get(0).modify();
}).start();
new Thread(() -> {
    array.get(0).modify();
}).start();
```

Safe publication

- It is important to ensure that initialization *happens-before* publication
  - That is, before making accessible a reference to an object, all its fields must be correctly initialized

```
public class UnsafeLazyInitialization {
    private static Resource resource;

    public static Resource getInstance() {
        if (resource == null)
            resource = new Resource();
        return resource;
    }
}
```

Object initialization & visibility

- Visibility issues may appear during initialization of objects

```
public class UnsafeInitialization {
    private int x;
    private Object o;
    public UnsafeInitialization() {
        x = 42;
        o = new Object();
    }
}
```

- For the thread executing the constructor, there are no visibility issues, but if a reference to an instance of UnsafeInitialization object is accessible to another thread, it might not see **`x==42`** or **`o`** completely initialized

Object initialization & visibility

- We can address visibility issues during initialization as follows

For primitive types, we can:
- Declare them as **`volatile`**
- Declare them as **`final`** (only works if the content is never modified)
- Initialize as the default value: 0. (only works if the default value is acceptable)
- Use corresponding atomic class from Java standard library: **`AtomicInteger`**

```
public class UnsafeInitialization {
    private volatile int x;
    private final Object o;
    public UnsafeInitialization() {
        x = 42;
        o = new Object();
    }
}
```

For complex objects, we can:
- Declare them as **`final`**
- Initialize as the default value: null. (only works if the default value is acceptable)
- Use the **`AtomicReference`** class

Object initialization & visibility

NOTE: For clarity and simplicity, up to now, we did not take initialization concerns into account. But from now on we will.

- The previous suggestions ensure safe publication because:
  - They established a *happens-before* relation between initialization and access the object's reference (publication)
    - *A write to a volatile field happens-before every subsequent read of that field.*
    - *The default initialization (zero, false, or null) of any object happens-before any other actions of a program.*
    - *The initialization of a final field happens-before any other actions of a program (after the constructor has finished its execution)*

Defined by us from the JLS explanation. You can use for exercises in this course.

If the constructor of the class leaks a reference of the object being constructed before it has completed its execution, then there is no happen-before relation with the accesses to final field

  - At the JVM level, the reason is that
    - **`final`** fields cannot be cached or reordered during initialization
    - All fields are initialized with default values during class loading
    - writes on **`volatile`** are flushed to main memory and reordered (during initialization)

Immutability

- An immutable object is one whose state cannot be changed after initialization
  - You can think of it as a constant
  - The **`final`** keyword in Java prevents modification of fields
    - Remember that variables assigned to an object only hold a reference to the object
- Since immutable objects do not change the state after initialization, data races can only occur during initialization
- An immutable class is one whose instances are immutable objects

Immutability

- To ensure thread-safety of immutable classes you simply need to make sure:
  - No fields can be modified after publication
  - Objects are safely published
  - Access to inner mutable object do not escape

```
public final class ThreeStooges {
  private final Set<String> stooges = new HashSet<String>();

  public ThreeStooges () {
    stooges.add("Moe");
    stooges.add("Larry");
    stooges.add("Curly");
  }

  public Boolean isStooge(String name) {
    return stooges.contains(name)
  }
}
```

Goetz p. 47

Mutual exclusion

- Whenever shared mutable state is accessed by several threads, we must ensure mutual exclusion

Testing concurrent programs

Concurrent Correctness Test – Counter

Some strategies to take into account when developing a test:

1. Precisely define the property you want to test

2. If you are going to test multiple implementations, it is useful to define an *interface* for the class you are testing

3. Concurrent tests require a setup for starting and running multiple threads
   - Maximize contention to avoid a sequential execution of the threads
   - You may need to define thread classes

4. Run the tests multiple times and with different setups to try to maximize the number of interleavings tested

Concurrent Correctness Test – Counter

- Precisely define the property you want to test
  - Use assertions to test properties

- *"after N threads execute inc() X times, the value of the counter must be equal to N*X"*

```
Class CounterTest {

 Counter count;
 …
 public void testingCounterParallel(int nrThreads, int N) {
    // body of the test
    assert(N*nrThreads == count.get());
 }
 …
}
```

Concurrent Correctness Test – Counter

- If you are going to test multiple implementations, it is useful to define an *interface* for the class you are testing

```
public interface Counter {
    public void inc();
    public int get();
}
```

A thread-safe interger class, with methods to increase, decrease, etc. the intenger

```
class CounterDR implements Counter {

    private int count;

    public CounterDR() {
      count = 0;
    }

    public void inc() {
      count++;
    }

    public int get() {
      return count;
    }
}
```

```
class CounterSync implements Counter {

    private int count;

    public CounterSync() {
      count = 0;
    }

    public synchronized void inc() {
      count++;
    }

    public int get() {
      return count;
    }
}
```

```
class CounterAto ...lements Counter {

    private AtomicInteger count;

    public CounterAto() {
      count = new AtomicInteger(0);
    }

    public void inc() {
      count.incrementAndGet();
    }

    public int get() {
      return count.get();
    }}
```

Concurrent Correctness Test – Counter

- Maximize thread contention
  - Maximizing the number of threads running concurrently
- A cyclic barrier may be used to decrease the chance that threads are executed sequentially

```
class TestCounter {

  // Shared variable for the tests
  CyclicBarrier barrier;

  …

 public void testingCounterParallel(int nrThreads, int N) {
     …

     // init barrier
     barrier = new CyclicBarrier(nrThreads + 1);

     for (int i = 0; i < nrThreads; i++) {
       new Thread(() -> {
       barrier.await(); // wait until all threads are ready
       // thread execution
       barrier.await(); // wait until all threads are finished
      }).start();
     }

     try {
         barrier.await();
         barrier.await();
     } catch (InterruptedException | BrokenBarrierException e) {
         e.printStackTrace();
     }
     …
}
```

Concurrent Correctness Test – Counter

- You may need to define thread classes

```
class TestCounter {

 Counter count;
 …
 public class Turnstile extends Thread {

     private final int N;

     public Turnstile(int N) { this.N = N; }

     public void run() {
         try {
             barrier.await();
             for (int i = 0; i < N; i++) {
                 count.inc();
             }
             barrier.await();
         } catch (InterruptedException | BrokenBarrierException e) {
       e.printStackTrace();
         }
     }
    }
   …
}
```

Note that the thread includes the barrier.await()s

```
class TestCounter {

  // Shared variable for the tests
  CyclicBarrier barrier;

  …

 public void testingCounterParallel(int nrThreads,
                                    int N) {
     …

     // init barrier
     barrier = new CyclicBarrier(nrThreads + 1);

     for (int i = 0; i < nrThreads; i++) {
       new Turnstile(N).start();
     }

     try {
         barrier.await();
         barrier.await();
     } catch (InterruptedException |
     BrokenBarrierException e) {
         e.printStackTrace();
     }
     …
}
```

Now we can simply start the thread in the test

Concurrent Correctness Test – Counter

- You may need to define thread classes

```
class TestCounter {

 Counter count;
 …
 public class Turnstile extends Thread {

     private final int N;

     public Turnstile(int N) { this.N = N; }

     public void run() {
         try {
             barrier.await();
             for (int i = 0; i < N; i++) {
                 count.inc();
             }
             barrier.await();
         } catch (InterruptedException | BrokenBarrierException e) {
       e.printStackTrace();
         }
     }
    }
   …
}
```

Note that the thread includes the barrier.await()s

```
class TestCounter {

  // Shared variable for the tests
  CyclicBarrier barrier;
   private final static ExecutorService pool
      = Executors.newCachedThreadPool();
  …
 public void testingCounterParallel(int nrThreads,
                                    int N) {
     …

     // init barrier
     barrier = new CyclicBarrier(nrThreads + 1);

     for (int i = 0; i < nrThreads; i++) {
       pool.execute(new Turnstile(N));
     }

     try {
         barrier.await
         barrier.await
     } catch (Interru
     BrokenBarrierExc
         e.printStack
     }
     …
}
```

Alternatively, we can use a thread pool as in Goetz
We will cover ThreadPools in two weeks

Concurrent Correctness Test – Counter

- Run the test multiple times
  - @RepeatedTest()
- Optionally (though encouraged), one may generate input parameters using JUnit (@ParameterizedTest)
  - Note that the test method takes as input two integer parameters
  - Using @MethodSoucer we can specify a method that provides a collection of parameters (known as arguments)

```
class TestCounter {
 …
 @ParameterizedTest
 @MethodSource("argsGeneration")
 public void testingCounterParallel(int nrThreads,
                                    int N) {
     //body of the test
 }
 …
}
```

```
private static List<Arguments> argsGeneration() {

    // Max number of increments
    final int I = 50_000;
    final int iInit = 10_000;
    final int iIncrement = 10_000;

    // Max exponent number of threads (2^J)
    final int J = 6;
    final int jInit = 1;
    final int jIncrement = 1;

    // List to add each parameters entry
    List<Arguments> list = new
    ArrayList<Arguments>();

    // Loop to generate each parameter entry
    // (2^j, i) for i \in {10_000,20_000,...,J}
    //          and j \in {1,..,I}
    for (int i = iInit; i <= I; i += iIncrement) {
        for (int j = jInit; j < J; j += jIncrement) {
      list.add(Arguments.of((int) Math.pow(2,j),i));
        }
    }

    // Return the list
    return list;
 }
```

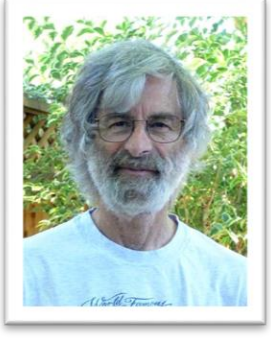
Arguments is a JUnit class that can be seen as a collection of objects of different type

Happens-before

Happens-before

We will focus on the Java happens-before relation (page 341 Goetz and JLS documentation)

- A *memory model* characterizes the set of *valid* executions of a concurrent program (interleavings) in terms of a *happens-before relation* (may be called differently in other languages)
- We say that an operation a *happens-before* an operation b, denoted as $a \rightarrow b$, iff
  - a and b belong to the same thread and a appears before b in the thread definition
  - a is an `unlock()` and b is a `lock()` on the same lock
- In the absence of *happens-before* relation between operations, the JVM is free to choose any execution order
  - In that case we say that operations are executed *concurrently*
  - Sometimes denoted as $a \mid\mid b$
- *Happens-before* is a *strict partial order* over operations of concurrent programs
  - Irreflexive, transitive and antisymmetric
- "Happened-before" was first introduced by Leslie Lamport for distributed systems
  - See optional readings



Java happens-before rules, Goetz Chp. 16

The rules for *happens-before* are:

Program order rule. Each action in a thread *happens-before* every action in that thread that comes later in the program order.

Monitor lock rule. An unlock on a monitor lock happens-before every subsequent lock on that same monitor lock.[3]

[3]. Locks and unlocks on explicit Lock objects have the same memory semantics as intrinsic locks.

Volatile variable rule. A write to a volatile field happens-before every subsequent read of that same field.[4]

[4]. Reads and writes of atomic variables have the same memory semantics as volatile variables.

Thread start rule. A call to Thread.start on a thread *happens-before* every action in the started thread.

Thread termination rule. Any action in a thread *happens-before* any other thread detects that thread has terminated, either by successfully return from Thread.join or by Thread.isAlive returning false.

Interruption rule. A thread calling interrupt on another thread happens-before the interrupted thread detects the interrupt (either by having InterruptedException thrown, or invoking isInterrupted or interrupted).

Finalizer rule. The end of a constructor for an object *happens-before* the start of the finalizer for that object.

Transitivity. If A *happens-before* B, and B *happens-before* C, then A *happens-before* C.

Happens-before | Interleavings

- The happens-before relation tell us that for all interleavings. Let $x, y \in \{1,2\}$. (By lock rule)

$$t_x(4) \rightarrow t_y(1)$$

- But also that (By sequential order rule)

$t_x(1) \rightarrow t_x(2)$ and $t_x(2) \rightarrow t_x(3)$ and $t_x(3) \rightarrow t_x(4)$

- Then we can derive that (By transitivity)

$$t_x(1) \rightarrow t_x(2) \rightarrow t_x(3) \rightarrow t_x(4) \rightarrow t_y(1) \rightarrow t_y(2) \rightarrow \cdots$$

- Thus, all the interleavings must include instructions 1-4 in a sequence of the form

$$[t_x(1), t_x(2), t_x(3), t_x(4)]^*$$

```
Lock l = new Lock();

public class Turnstile extends Thread {
  public void run() {
    for (int i=0; i < PEOPLE; i++) {
      l.lock()              // (1)
      int temp = counter;   // (2)
      counter = temp + 1;   // (3)
      l.unlock()            // (4)
    }
  }
}
```

This prevents that operations (2) and (3) are executed concurrently (which was the source of the race condition we saw above)

Linearizability

Sequential consistency

- For concurrent executions, we must define the conditions asserting that every thread is behaving consistently w.r.t. a sequential execution

- For executions of concurrent objects, an execution is *sequential consistency* iff
  1. Method calls appear to happen in a one-at-a-time, sequential order — Requires memory commands to be executed in order
  2. Method calls should appear to take effect in program order — Requires sequential program order for each thread

Linearizability

- Linearizability extends sequential consistency by requiring that the real time order of the execution is preserved

- Linearizability extends sequential consistency with the following condition:
  1. Each method call should appear to take effect instantaneously at some moment between its invocation and response

Note that the definition does not mention data races, race conditions or happens-before relation between operations

Linearizable Concurrent Objects

- Until now, linearizability is presented as a property of *executions*, not concurrent objects

Proving this is hard

- A concurrent object is linearizable iff
  - All executions are linearizable, and
  - All linearizations satisfy the sequential specification of the object

- To show that an object is linearizable first we must select its linearization points in the source code of the object class
  - Very often linearization points correspond to CAS operations

Linearizability – Source code

- To argue whether a (concurrent) object is linearizable one should:
  - Define clearly the sequential specification of the object
  - Identify linearization points
    - Explain the operation associated to each linearization point
  - Explain how (blocks of) programs statements in the same method or other methods in the class interact with the linearization point
    - The goal is to identify a concurrent execution that would produce an execution that does not satisfy the sequential specification of the object

Linearizability of MS Queue

```
class MSQueue<T> implements UnboundedQueue<T> {
…
 public void enqueue(T item) {
    Node<T> node = new Node<T>(item, null);
    while (true) {
      Node<T> last = tail.get();
      Node<T> next = last.next.get();
      if (last == tail.get()) { // E7
        if (next == null)  { // E8
          // In quiescent state, try inserting new node
          if (last.next.compareAndSet(next, node)) { // E9
            // Insertion succeeded, try advancing tail
            tail.compareAndSet(last, node);
            return;
          }
        } else
        // Queue in intermediate state, advance tail
        tail.compareAndSet(last, next);
      }
    }
 }
…
}
```

- Enqueue has one linearization point:
  - E9 – if successfully executed, the element has been enqueued
- Correctness (informal but systematic, tries to cover all branches):
  - If two threads execute enqueue concurrently before tail and next are updated, then only one of them succeeds in executing E9 (and possible update the tail). The other fails and repeats the enqueuing
  - If a thread executes enqueue after another thread updated the tail, then E7 fails and it repeats the enqueue
  - If a thread executes enqueue after another thread updated next, then E8 fails, the thread tries to advance the tail, and it restarts the enqueue

Linearizability of MS Queue

```
class MSQueue<T> implements UnboundedQueue<T> {
…

  public T dequeue() {
    while (true) {
      Node<T> first = head.get();
       Node<T> last = tail.get();
      Node<T> next = first.next.get(); // D3
       if (first == head.get()) { // D5
         if (first == last) { // D6
           if (next == null) // D7
             return null;
           else
             tail.compareAndSet(last, next);
         } else {
           T result = next.item;
           if (head.compareAndSet(first, next)) // D13
             return result;
         }
       }
    }
  }
…
}
```

- Dequeue has two linearization points
  - D3 - if the queue is empty. After its execution, the evaluation of D7 is determined and whether the method will return null.
  - D13 - if successfully executed, the element has been dequeued
- Correctness (informal but systematic, tries to cover all branches):
  - If two threads execute dequeue concurrently before the head is updated (D5 succeeds for both) and the queue is not empty (D6 fails), then D13 succeeds for only one of them. The other restarts the dequeue
  - If a thread executes dequeue after another thread updated the head, then D5 fails and it restarts the dequeue
  - If a thread execute dequeue while another thread executed enqueue (E9) and before the enqueuing thread updates the tail, then D7 fails, the dequeuing thread tries to update the tail and restarts the dequeue

Linearizability of objects with locks

- This is not a common use case for linearizability
  - Linearizability is better suited for programs using CAS operations
  - You may use happens-before reasoning and show mutual exclusion
- Herlihy, page 58
  - *"For lock-based implementations, any point within each method's critical section can serve as its linearization point."*
  - Intuitively, you may think of program statements within a critical section as one program statement (as they are effectively atomic)

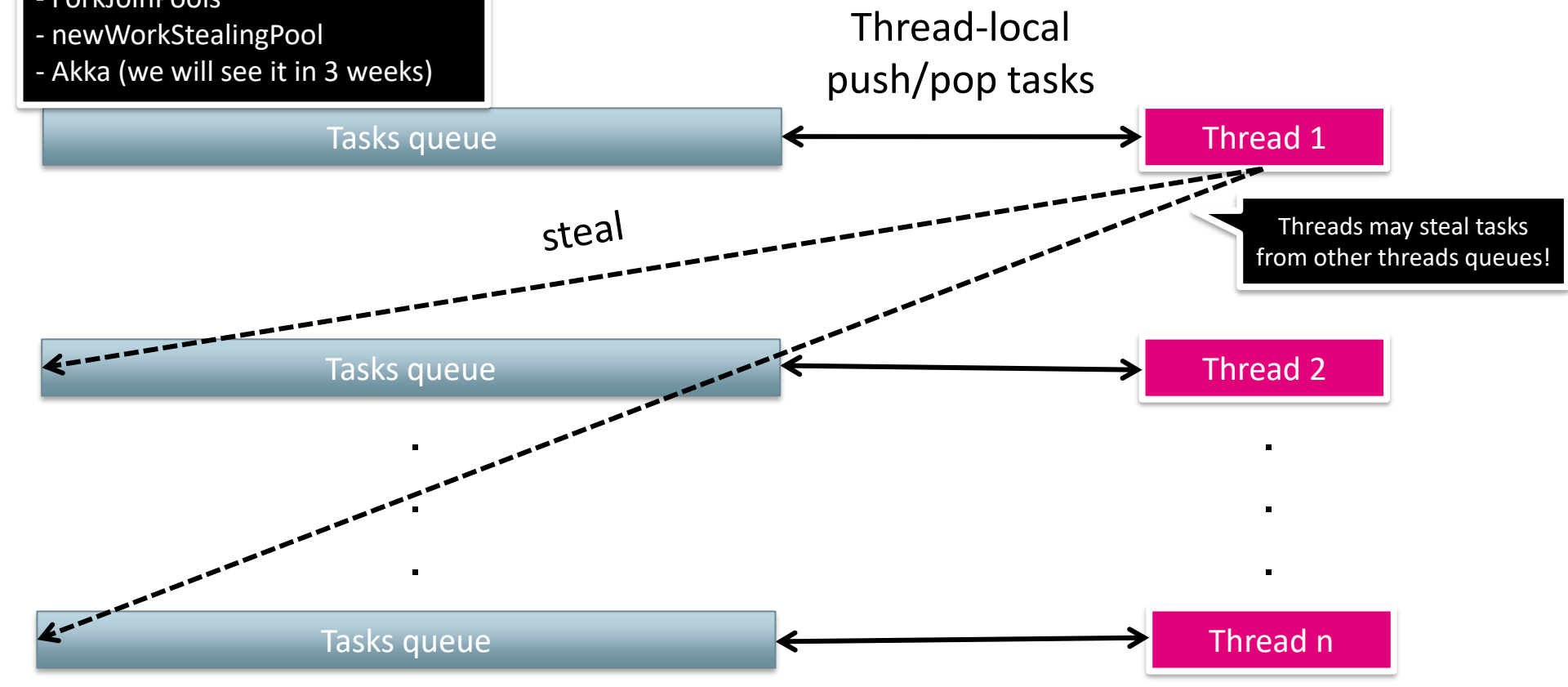
```
public class AtomicInteger {

  private Lock l = new ReentrantLock();
  private counter = 0;

  public void increment() {
    l.lock()              // (1)
    int temp = counter;   // (2)
    counter = temp + 1;   // (3)
    l.unlock()            // (4)
  }

}
```

Here (1),(2),(3),(4) could be mapped into a single program statement. This statement defines the linearization point of increment()

Note that the critical section needs not cover the entire body of the method

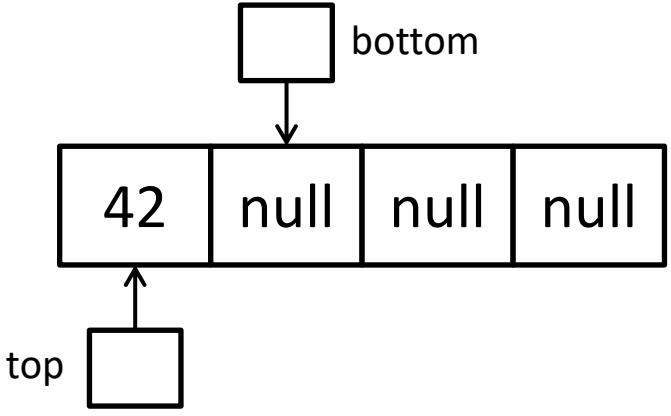
Executor framework work-stealing task queue

It is used in the implementation of:
- ForkJoinPools
- newWorkStealingPool
- Akka (we will see it in 3 weeks)



Work-stealing queues

- A work-stealing queue has the following methods
  - Push – adds an element at the bottom of the queue (thread-local)
  - Pop – removes an element from the bottom of the queue (thread-local)
  - Steal – removes an element from the top of the queue (concurrent)

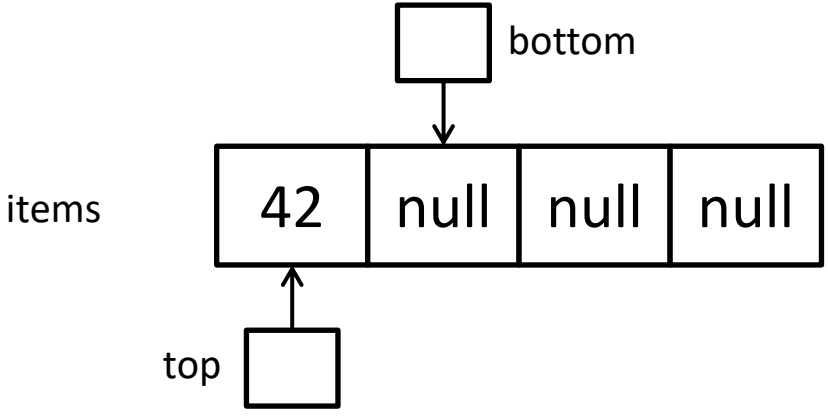



```
interface Deque<T> {
  void push(T item); // at bottom
  T pop();           // from bottom
  T steal();         // from top
}
```

We consider a simplified implementation with a fix size array

Chase-Lev work-stealing queue - state

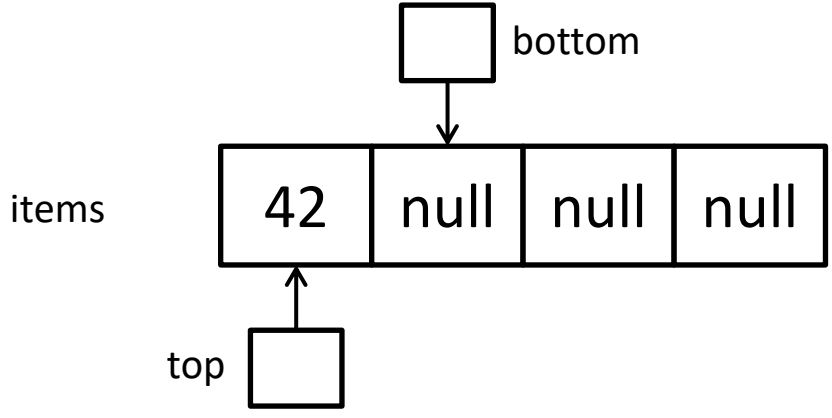
```
class ChaseLevDeque<T> implements Deque<T> {
  private volatile long bottom = 0;
  private final AtomicLong top = new AtomicLong();
  private final T[] items;
…
}
```



- The variable bottom is thread-local
  - Only the thread assigned to the queue can write it (other threads may read it)
- Any thread can read/write the variable top
  - We need an atomic variable to prevent data races
- For simplicity, we consider a fix-size array to store the elements of the queue
  - The array is used as a circular buffer

Chase-Lev work-stealing queue - push

```
public void push(T item) { // at bottom
  final long b = bottom, t = top.get(), size = b - t;
  if (size == items.length)
    throw new RuntimeException("queue overflow");
  items[index(b, items.length)] = item;
  bottom = b+1;
}
```

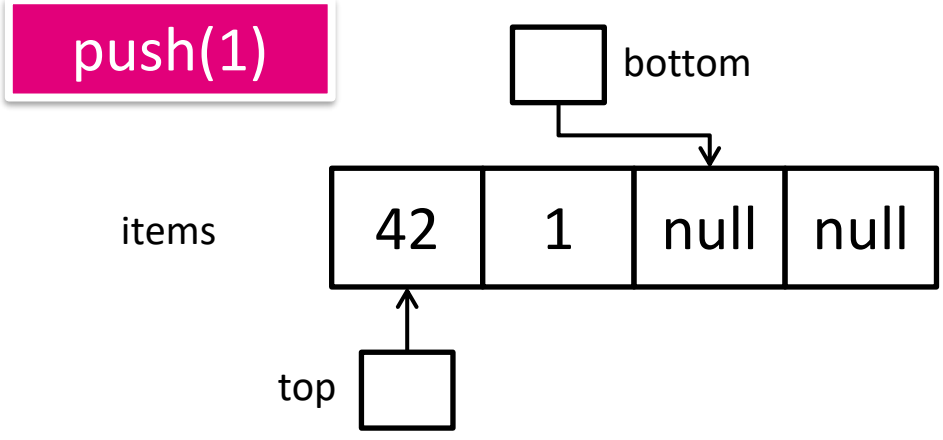


- Thread-safe because it is assumed to be thread-local
  - Always the same thread executes this method
  - Only writes bottom

Chase-Lev work-stealing queue - push

```
public void push(T item) { // at bottom
  final long b = bottom, t = top.get(), size = b - t;
  if (size == items.length)
    throw new RuntimeException("queue overflow");
  items[index(b, items.length)] = item;
  bottom = b+1;
}
```

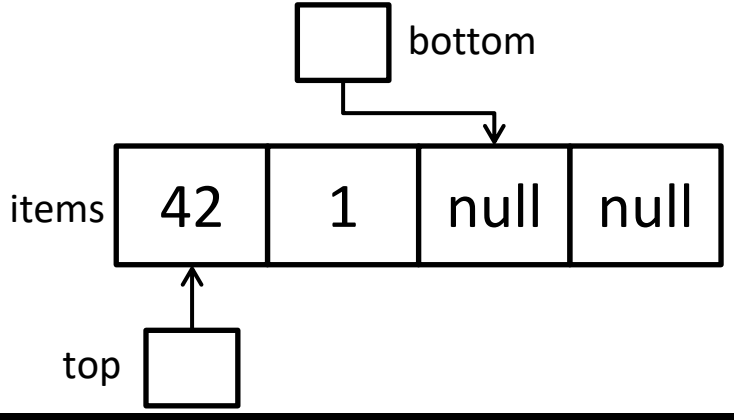
- Always the same thread executes this method
- Thread-safe because only writes bottom are thread-local (see other methods)



Chase-Lev work-stealing queue - steal

```
public T steal() { // from top
  final long t = top.get();
  final long b = bottom;
  final long size = b - t;
  if (size <= 0)
    return null;
  else {
    T result = items[index(t, items.length)];
    if (top.compareAndSet(t, t+1))
      return result;
    else
      return null;
  }
}
```

- It is executed by multiple threads
- Only reads bottom
- Performs a CAS on top to steal the top element
  - Only if not empty

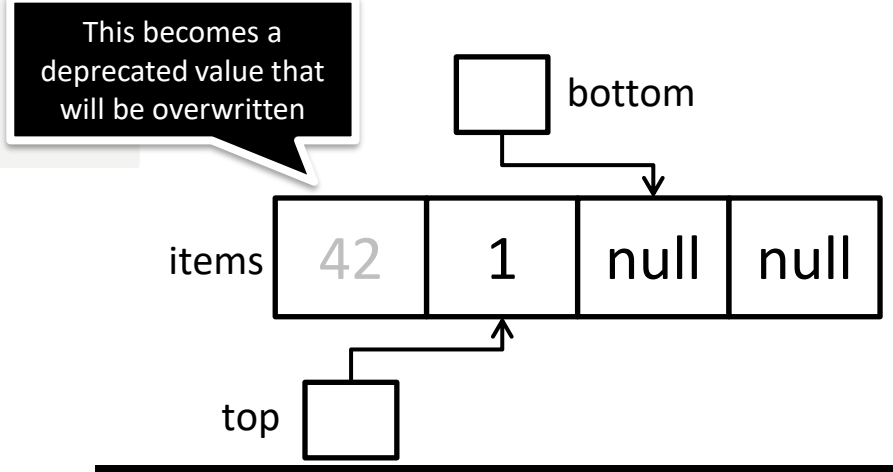


Chase-Lev work-stealing queue - steal

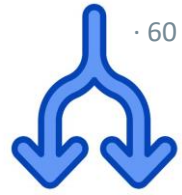
```
public T steal() { // from top
  final long t = top.get();
  final long b = bottom;
  final long size = b - t;
  if (size <= 0)
    return null;
  else {
    T result = items[index(t, items.length)];
    if (top.compareAndSet(t, t+1))
      return result;
    else
      return null;
  }
}
```

steal() -> 42

- It is executed by multiple threads
- Only reads bottom
- Performs a CAS on top to steal the top element
  - Only if not empty

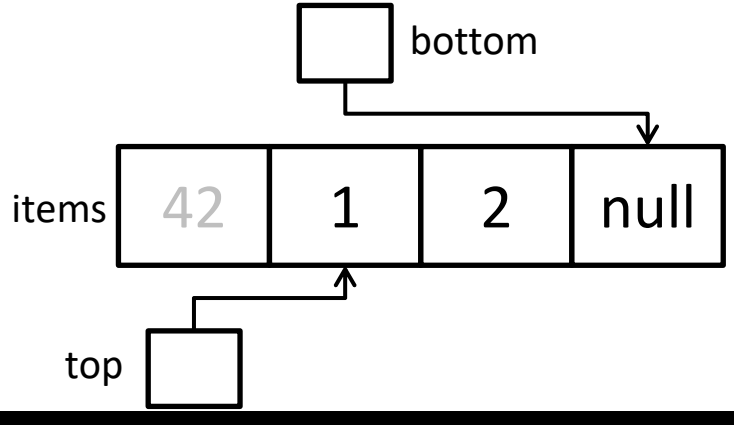


Chase-Lev work-stealing queue - pop



```
public T pop() { // from bottom
  final long b = bottom - 1;
  bottom = b;
  final long t = top.get(),
  final long afterSize = b - t;
  if (afterSize < 0) {
    bottom = t;
    return null;
  } else {
    T result = items[index(b, items.length)];
    if (afterSize > 0)
      return result;
    else {
      if (!top.compareAndSet(t, t+1))
          result = null;
      bottom = t+1;
      return result;
    }
  }
}
```

- Thread-local but more subtle than push
- It updates bottom (thread-local) and possibly top (concurrent)

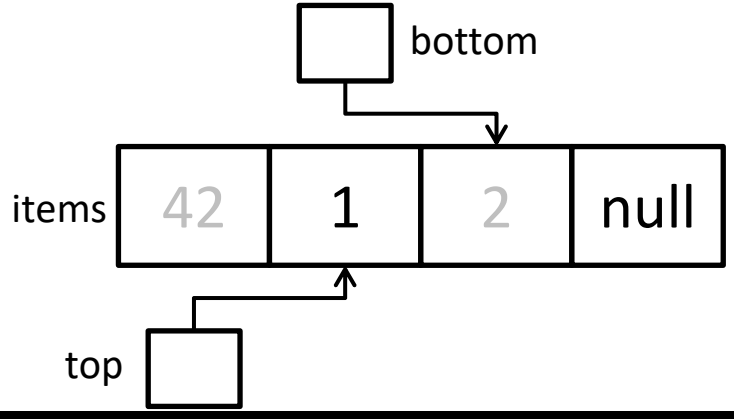


Chase-Lev work-stealing queue - pop

```
public T pop() { // from bottom
  final long b = bottom - 1;
  bottom = b;
  final long t = top.get(),
  final long afterSize = b - t;
  if (afterSize < 0) {
    bottom = t;
    return null;
  } else {
    T result = items[index(b, items.length)];
    if (afterSize > 0)
      return result;
    else {
      if (!top.compareAndSet(t, t+1))
          result = null;
      bottom = t+1;
      return result;
    }
  }
}
```

pop() -> 2

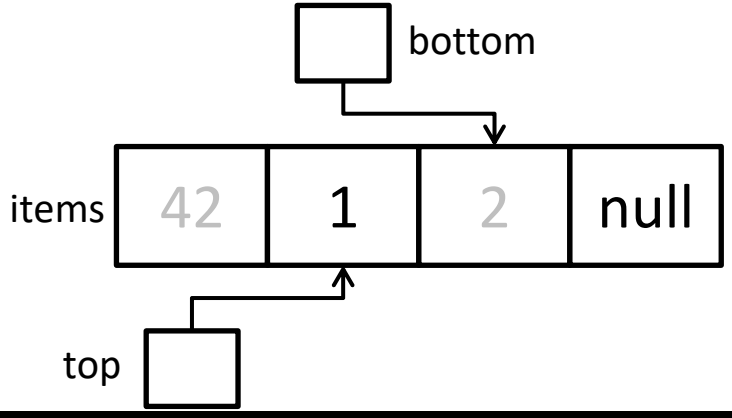
- When only the assign thread executes, then we simply update bottom and return the element



Chase-Lev work-stealing queue - pop

```
public T pop() { // from bottom
  final long b = bottom - 1;
  bottom = b;
  final long t = top.get(),
  final long afterSize = b - t;
  if (afterSize < 0) {
    bottom = t;
    return null;
  } else {
    T result = items[index(b, items.length)];
    if (afterSize > 0)
      return result;
    else {
      if (!top.compareAndSet(t, t+1))
          result = null;
      bottom = t+1;
      return result;
    }
  }
}
```

- What if we had pop() and steal() concurrently?

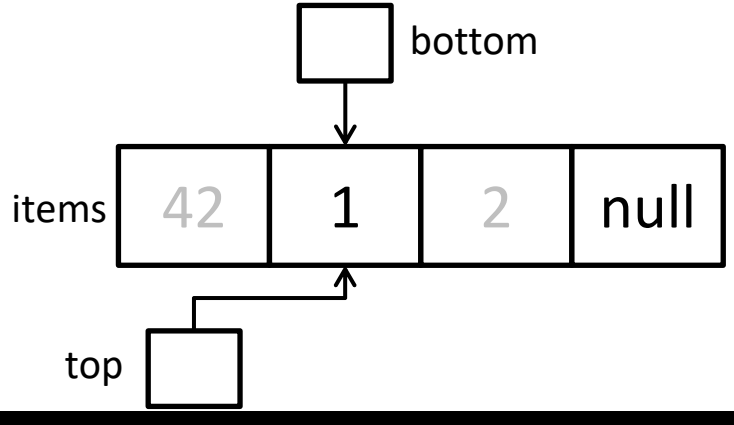


Chase-Lev work-stealing queue - pop

```
public T pop() { // from bottom
  final long b = bottom - 1;
  bottom = b;
  final long t = top.get(),
  final long afterSize = b - t;
  if (afterSize < 0) {
    bottom = t;
    return null;
  } else {
    T result = items[index(b, items.length)];
    if (afterSize > 0)
      return result;
    else {
      if (!top.compareAndSet(t, t+1))
          result = null;
      bottom = t+1;
      return result;
    }
  }
}
```

pop() -> ?

- What if we had pop() and steal() concurrently?



Chase-Lev ue - pop


```
public T steal() { // from top
  final long t = top.get();
  final long b = bottom;
  final long size = b - t;
  if (size <= 0)
    return null;
  else {
    T result = items[index(t, items.length)];
    if (top.compareAndSet(t, t+1))
      return result;
    else
      return null;
  }
}
```

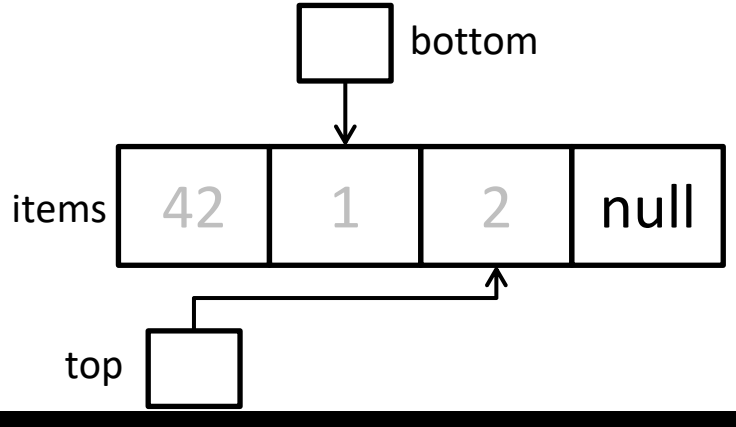
```
public T pop() {
  final long b = b
  bottom = b;
  final long t = t
  final long afterSize = b - t;
  if (afterSize < 0) {
    bottom = t;
    return null;
  } else {
    T result = items[index(b, items.length)];
    if (afterSize > 0)
      return result;
    else {
      if (!top.compareAndSet(t, t+1))
          result = null;
      bottom = t+1;
      return result;
    }
  }
}
```

pop() -> ?

steal() -> ?

What if we had pop() and steal() concurrently?

- Whatever thread succeeds in the CAS operation gets the element



Chase-Lev ue - pop

```
public T steal() { // from top
  final long t = top.get();
  final long b = bottom;
  final long size = b - t;
  if (size <= 0)
    return null;
  else {
    T result = items[index(t, items.length)];
    if (top.compareAndSet(t, t+1))
      return result;
    else
      return null;
  }
}
```

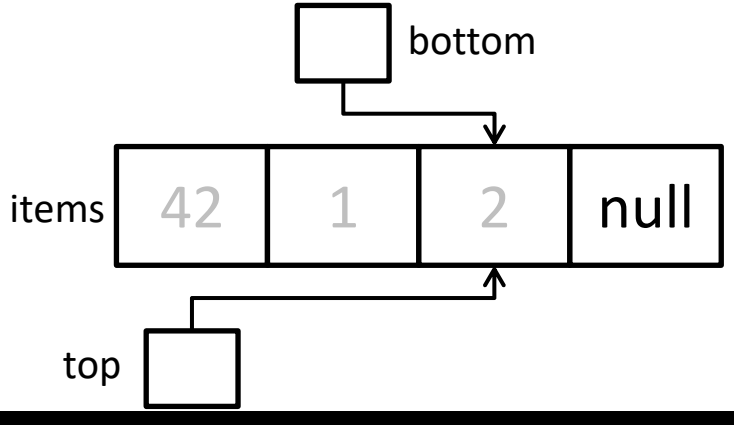
```
public T pop() {
  final long b = b
  bottom = b;
  final long t = t
  final long afterSize = b - t;
  if (afterSize < 0) {
    bottom = t;
    return null;
  } else {
    T result = items[index(b, items.length)];
    if (afterSize > 0)
      return result;
    else {
      if (!top.compareAndSet(t, t+1))
          result = null;
      bottom = t+1;
      return result;
    }
  }
}
```

pop() -> ?

steal() -> ?

What if we had pop() and steal() concurrently?

- Whatever thread succeeds in the CAS operation gets the element
- Afterwards, pop always fixes the bottom variable

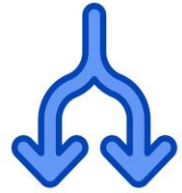


Anything we did not cover?

Questions?

Final remarks

Mandatory assignments

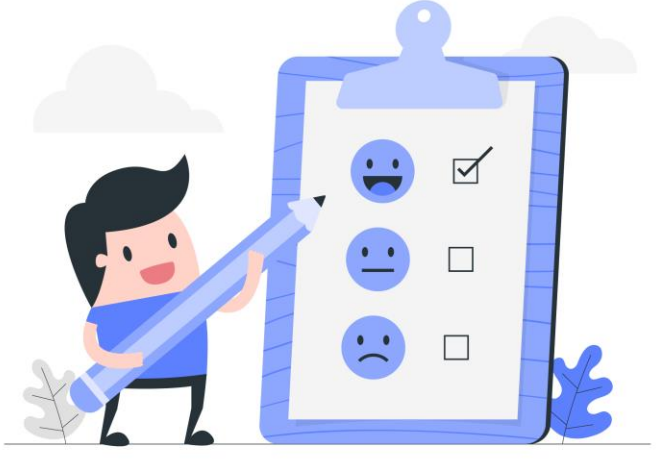


- To be eligible for the exam, 5 (or more) mandatory assignments must be approved

- You will get confirmation in the feedback for assignment 6
  - *"Your assignments have been approved and you may take the exam"*
- *It is your responsibility to let us know if there are any errors in grading*
  - For instance, missing grades, ungraded assignment, etc.
- There will be a final extra deadline on Dec 14th to hand-in assignments that have not yet been approved
  - With no possibility of re-submission and with written feedback

Course Evaluation Survey

Please participate in the course evaluation



Become a PCPP Teaching Assistant next year!

- Consider applying for a TA position in PCPP (Autumn 2024)
  - Call in Spring 2024 (around March-April) → contact me (raup@itu.dk) directly if you are interested

Thank you for you attention

We hope you enjoyed the course





PCPP teaching team