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 revisit to Raúl (raup@itu.dk) before Thursday Nov 28th**

- 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
- Sequential consistency & Linearizability
- Questions
- 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

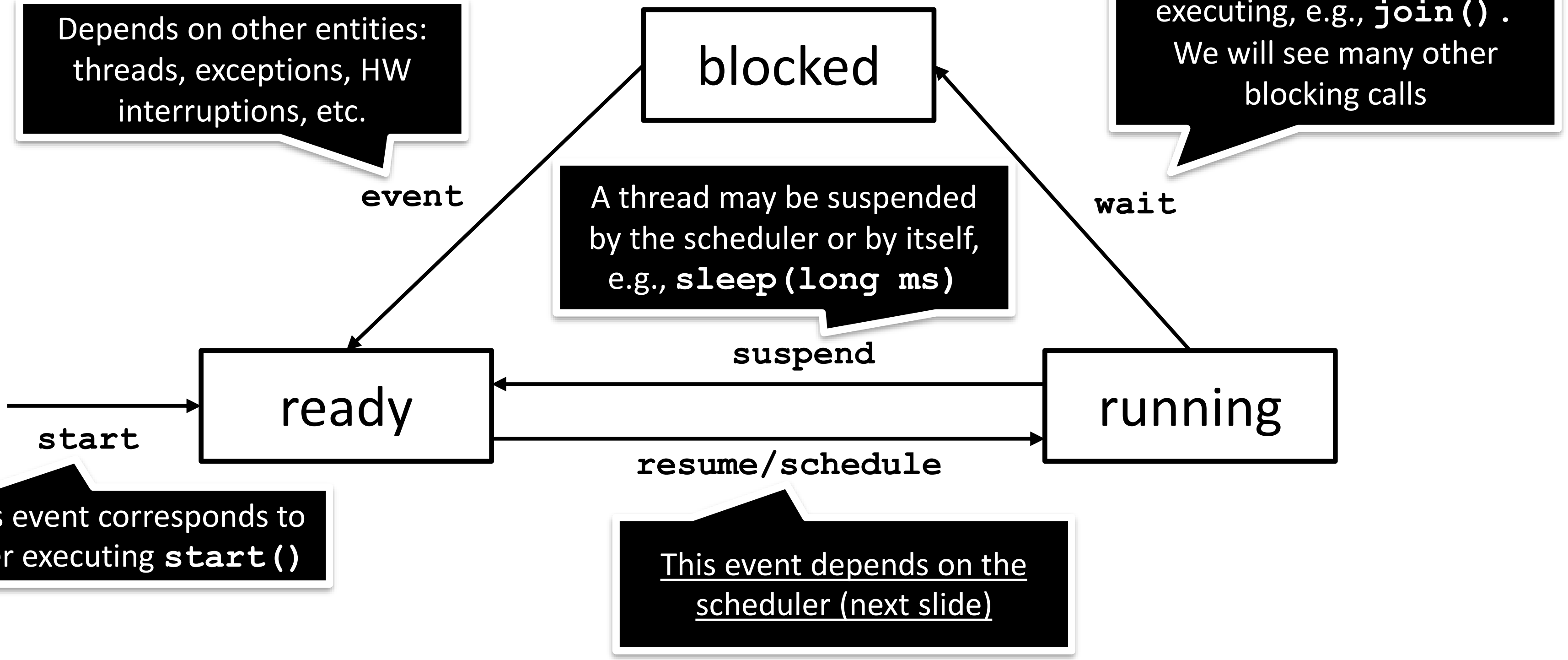
- The program statement **`counter++`** is not *atomic*
- *Atomic statements are executed as a single (indivisible) operation*

```
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)


blocked
ready
running
start
event
wait
suspend
resume/schedule
Depends on other entities: threads, exceptions, HW interruptions, etc.
This event corresponds to executing, e.g., **join()**. We will see many other blocking calls
A thread may be suspended by the scheduler or by itself, e.g., **sleep(long ms)**
This event corresponds to after executing **start()**
This event depends on the scheduler (next slide)

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: <u>a sequence of operations for a concurrent program</u>, 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)

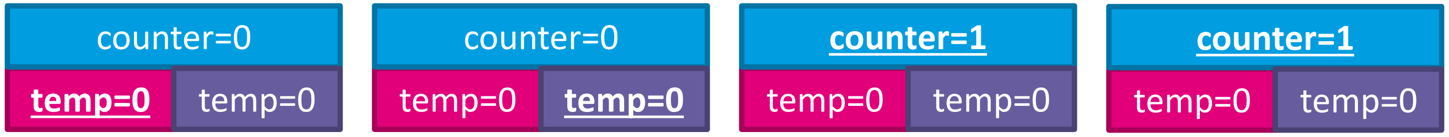
*Given the initial memory state on the right, provide an interleaving such that after two threads t1, t2 execute the program on the right the value of **counter==1***

Memory

counter=0	
i=0	i=0

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 <u>race conditions</u> are <u>data races</u>

- 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 <u>data races</u> result in <u>race conditions</u>

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

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

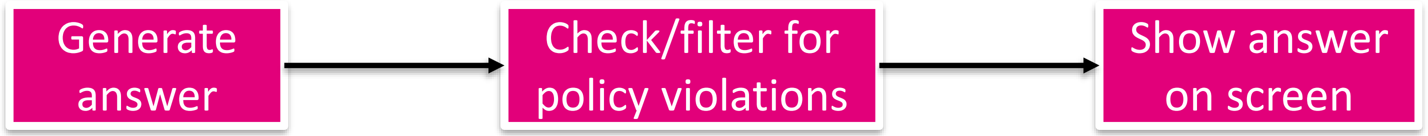
```
public void q() {
  x=1;    //Q1
}
```

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

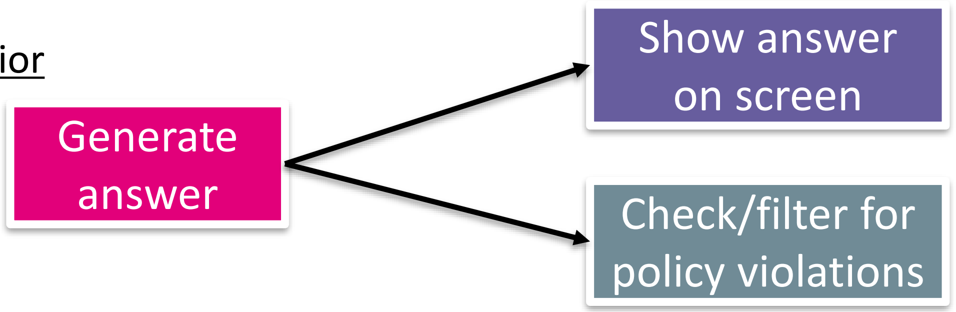
LLMs also in PCPP (curiosity)

- Race conditions also exist in LLMs, and may bypass their protection mechanisms (video)
  - https://www.schneier.com/blog/archives/2024/11/race-condition-attacks-against-llms.html

<u>Expected behavior</u>



<u>Observed behavior</u>



Since showing the answer and checking for violation are executed concurrently and without synchronization, the final output suffers from race conditions

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

Sequential consistency and Linearizability: What is their purpose and use case? How are they useful?

Linearizability – Intro

- Defining correctness of (non-blocking) concurrent objects is tricky
  - Recall correctness is defined by a specification

- ***The motivation behind linearizability is to use specifications of sequential objects as a basis for the correctness of concurrent objects***
  - Specifications of sequential objects are typically expressed as pre- and post-conditions for method calls

Queue sequential spec

<u>Enqueue(e)</u>

- *Pre-condition*: The state of the queue is q (where q denotes the sequence of elements stored in the queue or an empty queue)
- *Post-condition*: The state of the queue is $q \cdot e$ where $\cdot$ denotes concatenation

<u>Dequeue()</u>

- Case 1
  - *Pre-condition*: The state of the queue is empty
  - *Post-condition*: Return error
- Case 2
  - *Pre-condition*: The state of the queue is $h \cdot q$
  - *Post-condition*: The state of the queue is q and return h

Sequential consistency

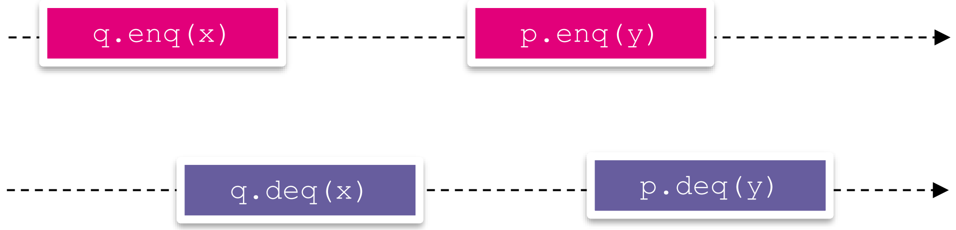
- For concurrent executions, we must define the conditions asserting that every thread is behaving consistently w.r.t. a sequential execution

- 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

What if we do not have sequential consistency

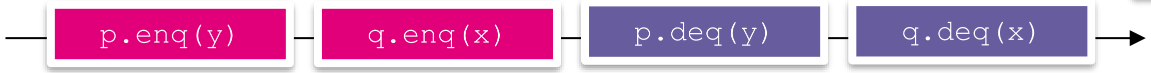
```
Thread pink = new Thread(() -> {
    q.enq(x);
    p.enq(y);
});
```

```
Thread purple = new Thread(() -> {
    q.deq(x);
    p.deq(y);
});
```



This is counterintuitive for most engineers, but it can occur due to intra-thread reordering

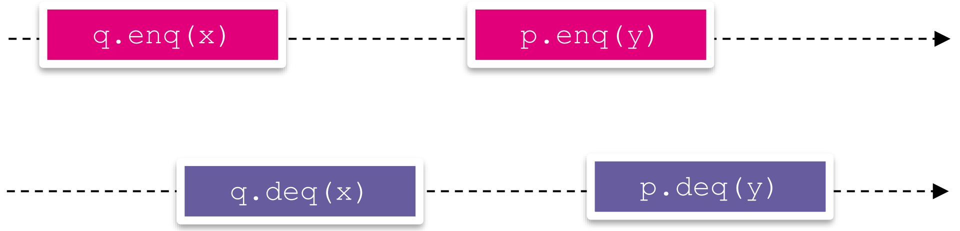
Without sequential consistency, the sequential execution below is possible



What if we do not have sequential consistency

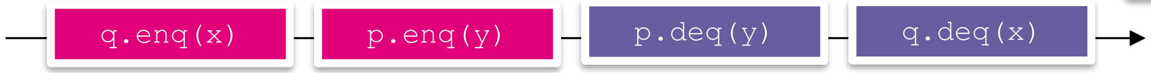
```
Thread pink = new Thread(() -> {
    q.enq(x);
    p.enq(y);
});
```

```
Thread purple = new Thread(() -> {
    q.deq(x);
    p.deq(y);
});
```



This is does not even satisfy the specification, but it can occur due to memory commands issued in wrong order

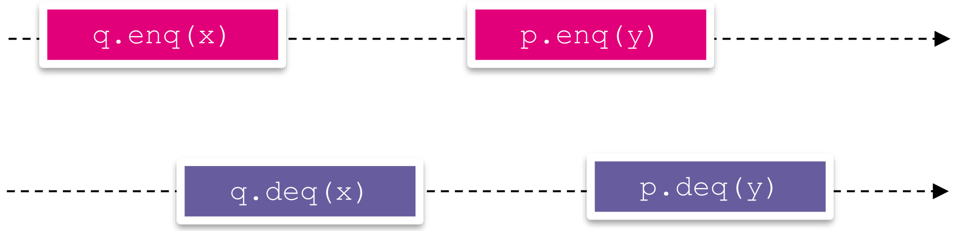
Without sequential consistency, the sequential execution below is possible



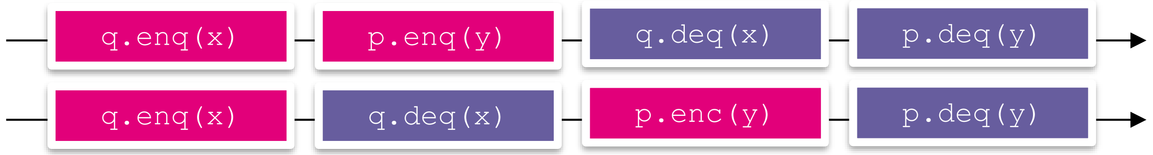
What if we do not have sequential consistency

```
Thread pink = new Thread(() -> {
    q.enq(x);
    p.enq(y);
});
```

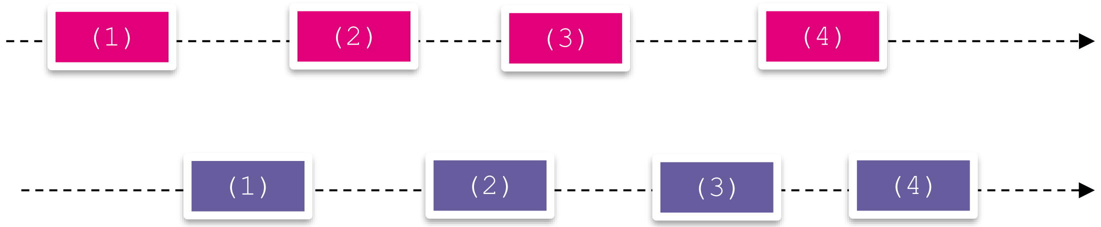
```
Thread purple = new Thread(() -> {
    q.deq(x);
    p.deq(y);
});
```



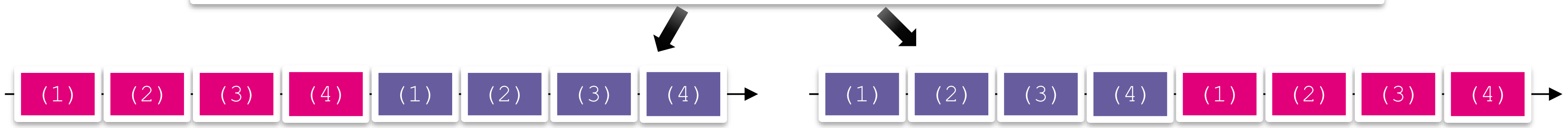
In summary, sequential consistency tries to ensure that only the two sequential executions below are possible, as they are considered intuitive or expected by most engineers



Sequential consistency in Java



As consequence of having a correctly synchronized program, the JVM is designed to only produce executions that exhibit any of the two sequential operation orderings below.
Most importantly, note that both are *sequentially consistent*
(we get sequential consistency---for free---if the program is correctly synchronized)



Sequential consistency and concurrent objects

- To show that an object is sequentially consistent, we must show that all possible concurrent executions are sequentially consistent

- Major drawback: sequentially consistent concurrent objects are not compositional
  - For instance, executions of two threads using two sequentially consistent queues are not guaranteed to be sequentially consistent
  - This prevents modular reasoning
  - This was the motivation to introduce *linearizability*

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

Proving this is hard

- 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

Benefits of linearizability

- It re-uses sequential specifications (like sequential consistency)
  - Specifying the sequential behaviour of an object is rather intuitive
- **It is a compositional property (unlike sequential consistency)**
  - If two objects are linearizable, any concurrent execution involving the two objects remains linearizable
  - Correctness proofs can be split into single objects that later can be safely composed
- It is a non-blocking property (like sequential consistency)
  - It can be used to reason about objects that do not use locks

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
- 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

Questions?

Questions?

Final remarks

Mandatory assignments

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

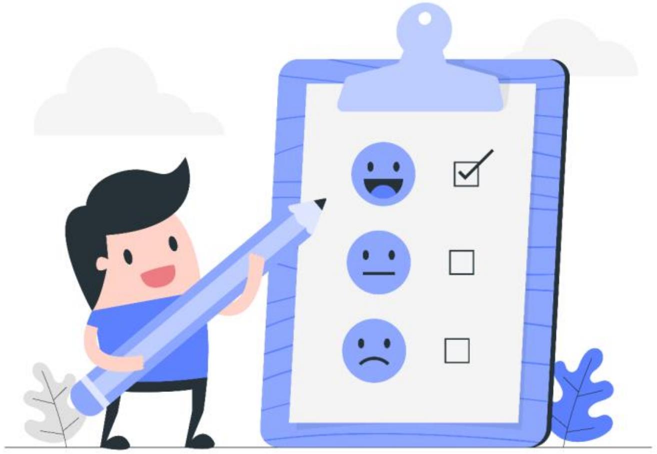
- You will get confirmation in the feedback for assignment 6
  - *"Your assignments have been approved and you may take the exam"*

- *<u>It is your responsibility to let us know if there are any errors in grading</u>*
  - For instance, missing grades, ungraded assignment, etc.

- There will be a final extra deadline on Dec 12th 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 2025)
  - Call in Spring 2025 (around March-April) ⟶ contact me (raup@itu.dk) directly if you are interested

MSc thesis in the SQUARE group

- Program analysis
- (Bayesian) data analysis
- Privacy (Data science software, ML, GDPR, etc.)
- Testing (concurrent/probabilistic programs)
- Robotics
- Reinforcement Learning
- Bayesian inference
- …

SOFTWARE QUALITY RESEARCH

Have a look at our projects at
https://square.itu.dk/student-projects/

Thank you for your attention

We hope you enjoyed the course

PCPP teaching team