Practical Concurrent and Parallel Programming

Riko Jacob
IT University of Copenhagen

Friday 2018-08-31

Plan for today

- Why this course?
- Course contents, learning goals
- Practical information
- Mandatory exercises, examination

- Java threads
- Java locking, the **`synchronized`** keyword
  - Use **`synchronized`** on blocks, not on methods
- Visibility of memory writes

Based on slides by
Peter Sestoft

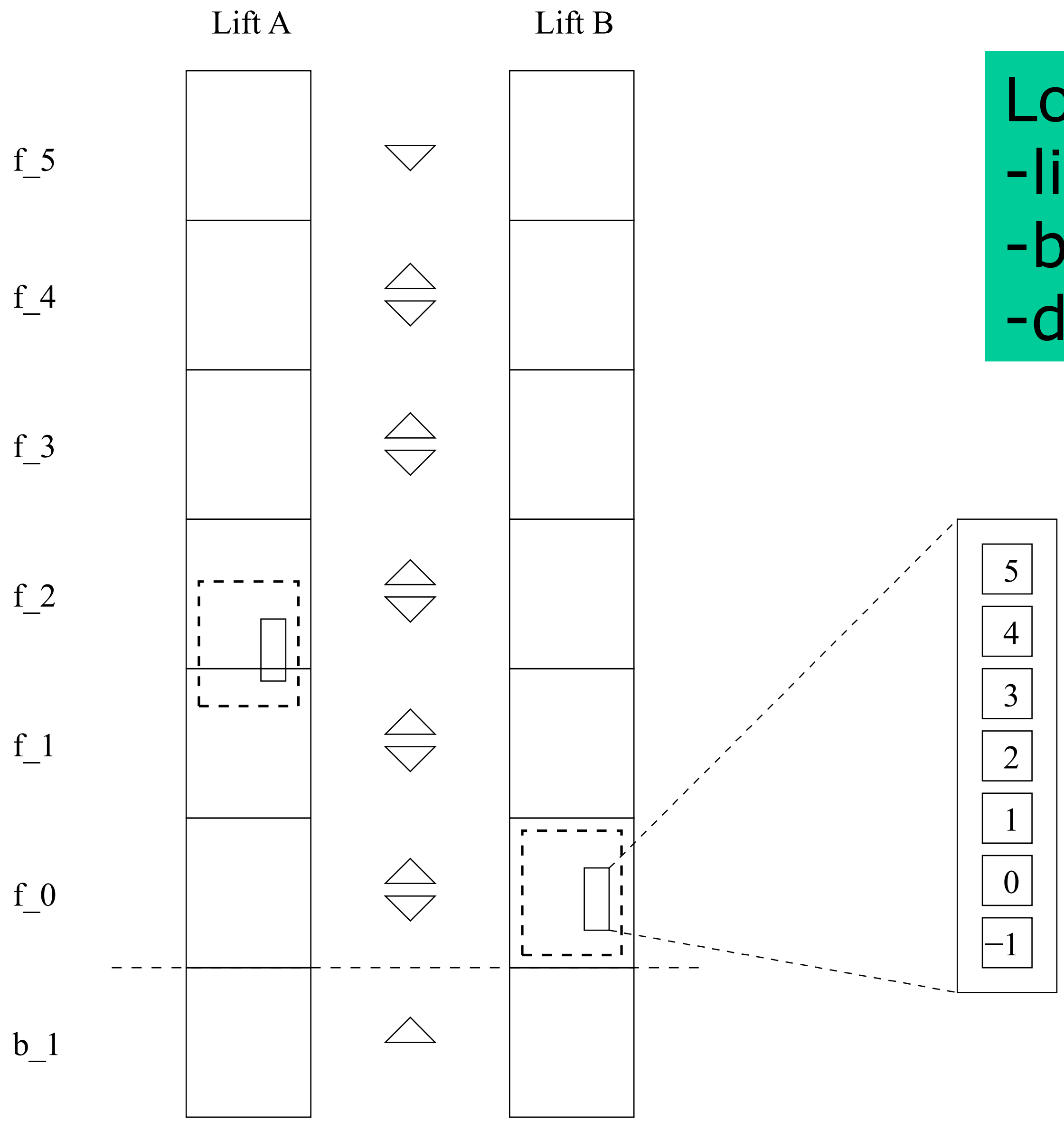
The teachers

- Course responsible: Riko Jacob
  - MSc 1998, PhD 2002 BRICS Aarhus University
  - Algorithms Engineering and other topics
  - Joined ITU in 2015
- Co-teachers:
  - Matteo Ceccarello
  - Claus Brabrand
- Material: Peter Sestoft, ’14, ’15, ’16, RJ ’17
- Exercises
  - Matteo Dusefante, ITU PhD student
  - Amund Lome, ITU MSc graduate

Why this course?

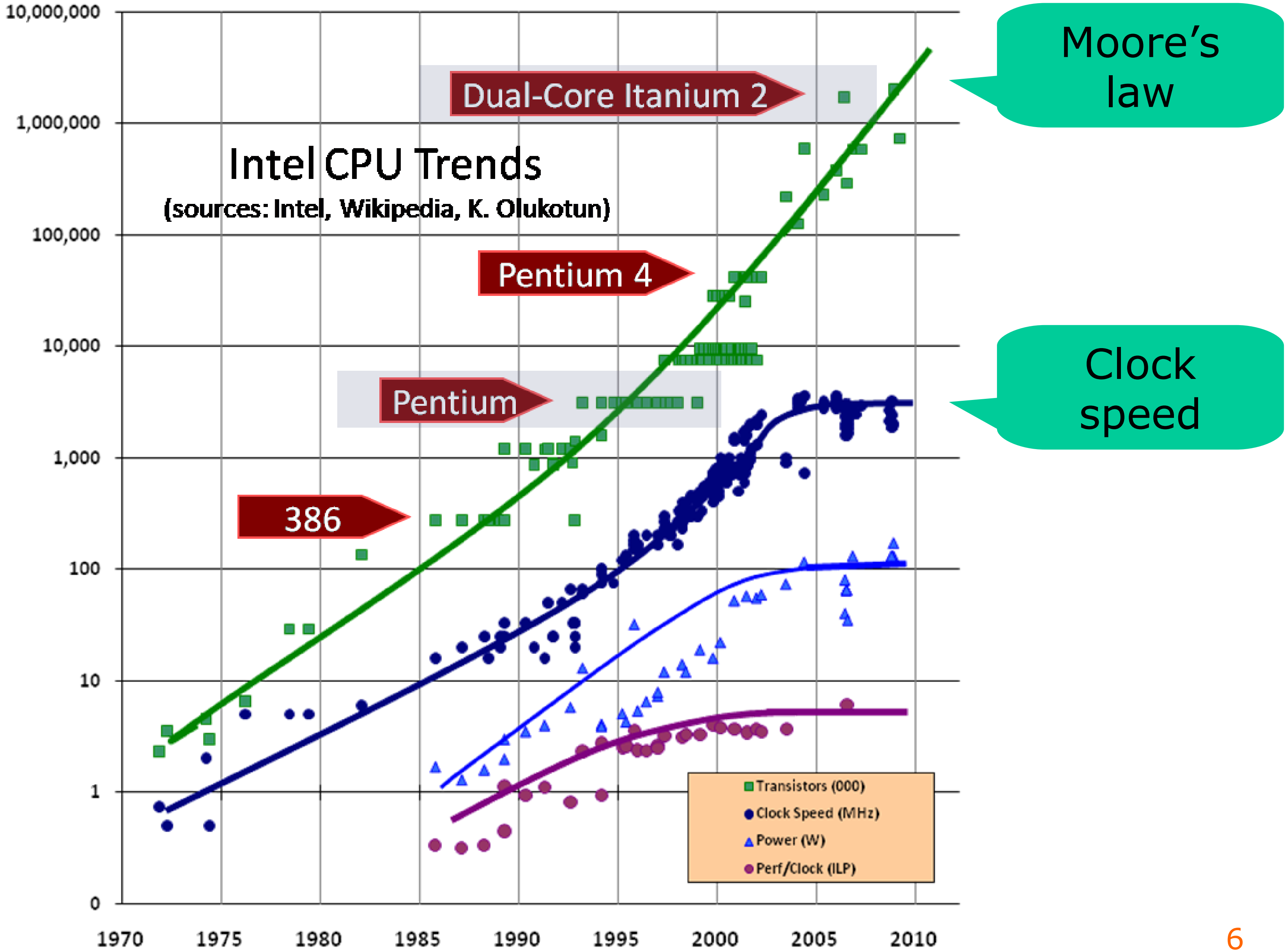
- Parallel programming is necessary
  - For responsiveness in user interfaces etc.
  - The real world is parallel
    - Think of the atrium lifts: lifts move, buttons are pressed
    - Think of handling a million online banking customers
  - For performance: *The free lunch is over*
- It is easy, and disastrous, to get it wrong
  - Testing is even harder than for sequential code
  - You should learn how to make correct parallel code
    - in a real language, used in practice
  - You should learn how to make fast parallel code
    - and measure whether one solution is faster than another
    - and understand why

Example: 2 lifts, 7 floors, 26 buttons



Lots of concurrency:
- lifts move
- buttons are pressed
- doors open & close

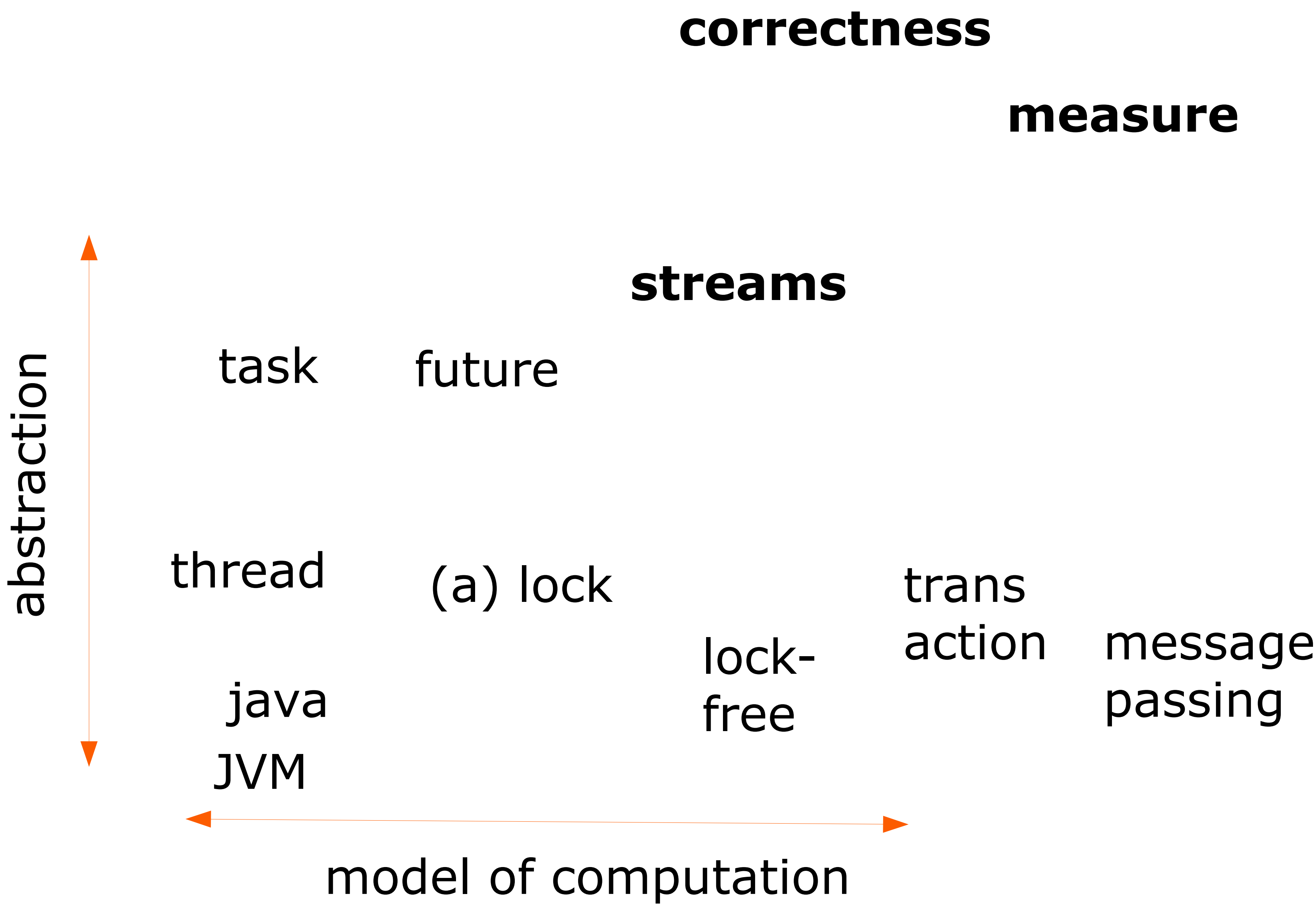
The free lunch is over: No more growth in single-core speed



Herb Sutter: The free lunch is over, Dr Dobbs, 2005.
Figure updated August 2009.
http://www.gotw.ca/publications/concurrency-ddj.htm

Course contents

- Threads, locks, mutual exclusion, scalability
- Java 8 streams, functional programming
- Performance measurements
- Tasks, the Java executor framework
- Safety, liveness, deadlocks
- Testing concurrent programs
- Transactional memory, Multiverse
- Lock-free data structures, Java mem. model
- Message passing, Akka


correctness
measure
streams
abstraction
task
future
thread
(a) lock
trans
action
lock-
free
message
passing
java
JVM
model of computation

Learning objectives

After the course, the successful student can:

- ANALYSE the correctness of concurrent Java software, and RELATE it to the Java memory model
- ANALYSE the performance of concurrent Java software
- APPLY Java threads and related language features (locks, final and volatile fields) and libraries (concurrent collections) to CONSTRUCT correct and well-performing concurrent Java software
- USE software tools for accelerated testing and analysis of concurrency problems in Java software
- CONTRAST different communication mechanisms (shared mutable memory, transactional memory, message passing)

Expected prerequisites

- From the ITU course base:
  "Students must know the Java programming language very well, including inner classes and a first exposure to threads and locks, and event-based GUIs as in Swing or AWT."
- Today we will briefly review the basics of
  - Java threads
  - Java synchronized methods and statements
  - Java's `final` keyword
  - Java inner classes and lambdas

Standard weekly plan

- Lectures Fridays in Auditorium 1
  Corresponding exercise assignment is ready
- Exercise Lab: Wednesdays, 2A12-14
  - Two slots: 14-16 and 16-18
  - First 15 minutes: Announcements wrt exercises
- Exercise hand-in: 6.5 days after lecture
  - That is, the following Thursday at 23:55
  - Feedback by 14 days after lecture
  - Retry-hand-in: 20.5 days after lecture
- Until December 7, Exam hand-in Dec 19
  (except fall break, Week 42, 15-19 Oct)

Course information online

- Course LearnIT page, restricted access: https://learnit.itu.dk/
  - Mandatory exercises and hand-ins, deadlines, feedback
  - Discussion forum
  - Non-public reading materials
- Course homepage, public access: http://www.itu.dk/people/rikj/PCPP2018/
  - Overview of lectures and exercises
  - Lecture slides and exercise sheets
  - Example code
  - List of all mandatory reading materials

Exercises

- There are 13 sets of weekly exercises
- At least 11 can be handed in towards the exam
- Hand in the solutions through LearnIT
- You can work in teams of 1,2 or 3 students
- The teaching assistants provide joint feedback
- Hand-ins: ≥6 must be submitted, ≥5 approved
  - otherwise you cannot take the course examination
  - failing to get 5 approved costs an exam attempt (!!)
- Exercise may be approved even if not fully solved
  - It is possible to resubmit
  - Make your best effort: two serious attempts=one solved
  - What is important is that **you learn**

The exam

- A 30 hour take-home written exam/project
  - Start at 0900 Tuesday 18 December 2017
  - End at 1500 Wednesday 19 December
  - Electronic submission in LearnIT
  - Followed by random sample “cheat check”
- Expected exam workload is 16 hours
  - Individual exam, no collaboration
  - All materials, including Internet, allowed
  - Always credit the sources you use
  - Plagiarism is **forbidden** – as always
- The old (2014 - 2017) exams are on the public homepage

Expected Time Usage

This course is 7.5 ECTS = 210 hours of work

For average student to get an average grade

Total hours	Weekly hours	
96	8	Solving / submitting exercise (12x)
42		Exam prep (2 old exams)
28	2	Reading
28	2	Lecture
16		Exam
Total: 210		

5 handins = preparation for barely pass

Stuff you need

- Buy Goetz et al: *Java Concurrency in Practice*
  - From 2006, still the best on Java concurrency
  - Most contents is relevant for C#/.NET too
- Free lecture notes and papers, see homepage
- A few other book chapters, see LearnIT

- **Java 8** SDK installed on your computer
  - Java 7 or earlier will **not** work
  - Java 9 or later should work, we will find out

- Various optional materials, see homepage:
  - Bloch: *Effective Java*, 2008, **highly recommended**
  - Sestoft: *Java Precisely*, 3rd edition 2016
  - more ...

What about other languages?

- .NET and C# are very similar to Java
  - We will point out differences on the way
- Clojure, Scala, F#, ... build on JVM or .NET
  - So thread concepts are very similar too
- C and C++ have some differences (ignore)
- Haskell has transactional memory
  - We will see this in Java too (Multiverse)
- Erlang, Scala, F# have message passing
  - We will see this in Java too (Akka)
- Dataflow, CSP, CCS, Pi-calculus, Join, Cω, ...
  - Zillions of other concurrency mechanisms

Other concurrency models

- Java threads interact via shared mutable fields
  - Shared: Visible to multiple threads
  - Mutable: The fields can be updated, assigned to
- This is a source of many problems
- **Alternatives** exist:
- No sharing: interact via message passing
  - Erlang, Scala, MPI, F#, Go ... and Java Akka library
- No mutability: use functional programming
  - Haskell, F#, ML, Google MapReduce, ...
- Allow shared mutable mem., but avoid locks
  - Transactional memory, optimistic concurrency
  - In Haskell, Clojure, ... and Java Multiverse library

Other parallel hardware

- We focus on multicore (standard) hardware
  - Typically 2-32 general cores on a CPU chip
  - (Instruction-level parallelism, invisible to software)
- Other types of parallel hardware exist
- Vector instructions (SIMD, SSE, AVX) on core
  - Typically 2-8 floating-point operations/CPU cycle
  - Claimed available through .NET JIT and hence C#
- General purpose graphics processors GPGPU
  - Such as Nvidia CUDA, up to 2500 cores on a chip
  - We're using those in a research project
- Clusters, cloud: servers connected by network

Threads and concurrency in Java

- A **thread** is
  - a sequential activity executing Java code
  - running at the same time as other activities
- Concurrent = at the same time = in parallel
- Threads communicate via fields
  - That is, by updating **shared mutable state**

A thread-safe class for counting

- A thread-safe long counter:

```
class LongCounter {
  private long count = 0;
  public synchronized void increment() {
    count = count + 1;
  }
  public synchronized long get() {
    return count;
  }
}
```

TestLongCounter.java

- The state (field **count**) is **private**
- Only **synchronized** methods read and write it

A thread that increments the counter

- A Thread **`t`** is created from a Runnable
- The thread's behavior is in the **`run`** method

```
final LongCounter lc = new LongCounter();
Thread t =
  new Thread(
    new Runnable() {
      public void run() {
        while (true)
          lc.increment();
      }
    }
  );
```

An anonymous inner class, and an instance of it

When started, the thread will do this: increment forever

- This only *creates* the thread, does not *start* it

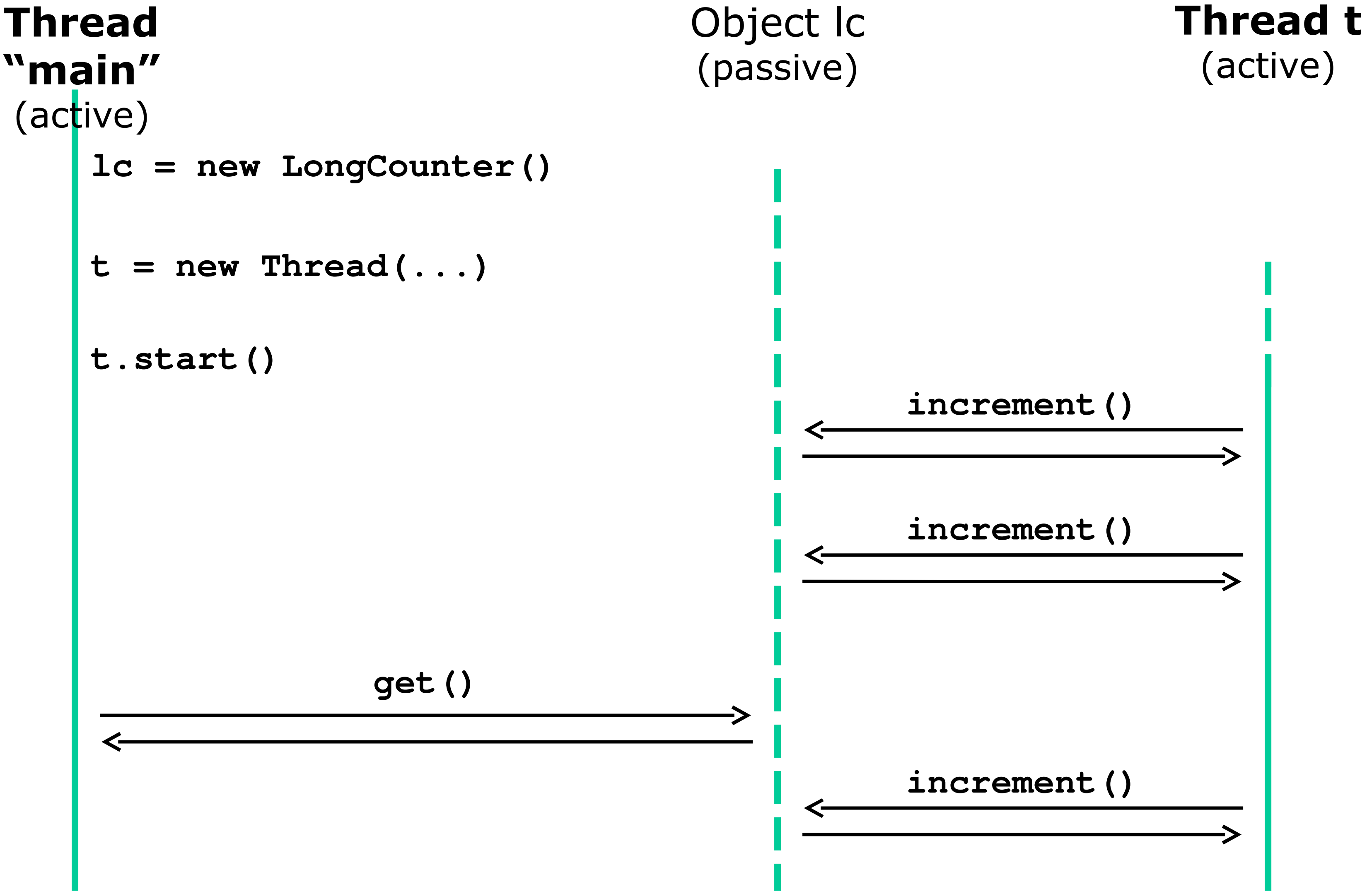
TestLongCounter7.java

Starting the thread in parallel with the main thread

```
public static void main(String[] args) ... {
  final LongCounter lc = new LongCounter();
  Thread t = new Thread(new Runnable() { ... });
  t.start();
  System.out.println("Press Enter ... ");
  while (true) {
    System.in.read();
    System.out.println(lc.get());
  }
}
```

```
Press Enter to get the current value:
60853639
103606384
263682708
...
```

Creating and starting a thread (and communicating via object)



Java 8 lambda expressions

- Instead of old anonymous inner classes:

```
Thread t = new Thread(
    new Runnable() {
      public void run() {
        while (true)
          lc.increment();
      }
    });
```

TestLongCounter7.java

- ... we use neat Java 8 lambda expressions:

```
Thread t = new Thread(() -> {
    while (true)
      lc.increment();
});
```

TestLongCounter.java

Locks and the synchronized statement

- Any Java object can be used for *locking*
- The **`synchronized`** statement

```
synchronized (obj) {
   ... body ...
}
```

  - Blocks until the lock on **`obj`** is available
  - Takes (acquires) the lock on **`obj`**
  - Executes the body block
  - Releases the lock, also on **`return`** or exception
- By consistently locking on the same object
  - one can obtain **mutual exclusion**, so
  - at most one thread can execute the code at a time

A synchronized method simply locks the "this" reference around body

- A synchronized instance method

```
class C {
  public synchronized void method() { ... }
}
```

  really uses a **synchronized** statement:

```
class C {
  public void method() {
    synchronized (this) { ... }
  }
}
```

- Q: What is being locked? (The entire class, the method, the instance, the Java system)?

What about synchronized *static* methods?

- A synchronized static method

```
class C {
  public synchronized static void method()
    { ... }
}
```

locks on the class runtime object C.class:

```
class C {
  public static void method() {
    synchronized (C.class) { ... }
  }
}
```

Use synchronized statements, not synchronized methods

- So it is clear what object is being locked on
- So only your methods lock on the object

```
class LongCounter {
  public synchronized void increment() { ... }
  public synchronized long get() { ... }
}
```

Good

```
class LongCounterBetter {
  private final Object myLock = new Object();
  public void increment() {
    synchronized (myLock) { ... }
  }
  public long get() {
    synchronized (myLock) { ... }
  }
}
```

Better

TestLongCounterBetter.java

Only these methods can lock on myLock

Clear what is locked on

Multiple threads, locking

- Two threads incrementing counter in parallel:

```
final int counts = 10_000_000;
Thread t1 = new Thread(() -> {
  for (int i=0; i<counts; i++)
    lc.increment();
});
Thread t2 = new Thread(() -> {
  for (int i=0; i<counts; i++)
    lc.increment();
});
```

TestLongCounterExperiments.java

- Q: How many threads are running now?

Starting the threads, and waiting for their completion

```
t1.start(); t2.start();
```

- A thread completes when the lambda returns
- To wait for thread **`t`** completing, call **`t.join()`**
- May throw InterruptedException

```
try { t1.join(); t2.join(); }
catch (InterruptedException exn) { ... }
```

```
System.out.println("Count is " + lc.get());
```

- What is **`lc.get()`** after threads complete?
  - Each thread calls **`lc.increment()`** ten million times
  - So it gets called 20 million times

Removing the locking

- Non-thread-safe counter class:

```
class LongCounter2 {
  private long count = 0;
  public void increment() {
    count = count + 1;
  }
  public long get() { return count; }
}
```

- Produces very wrong results, not 20 million:

```
Count is 10041965
Count is 19861602
Count is 18939813
```

- Q: Why?

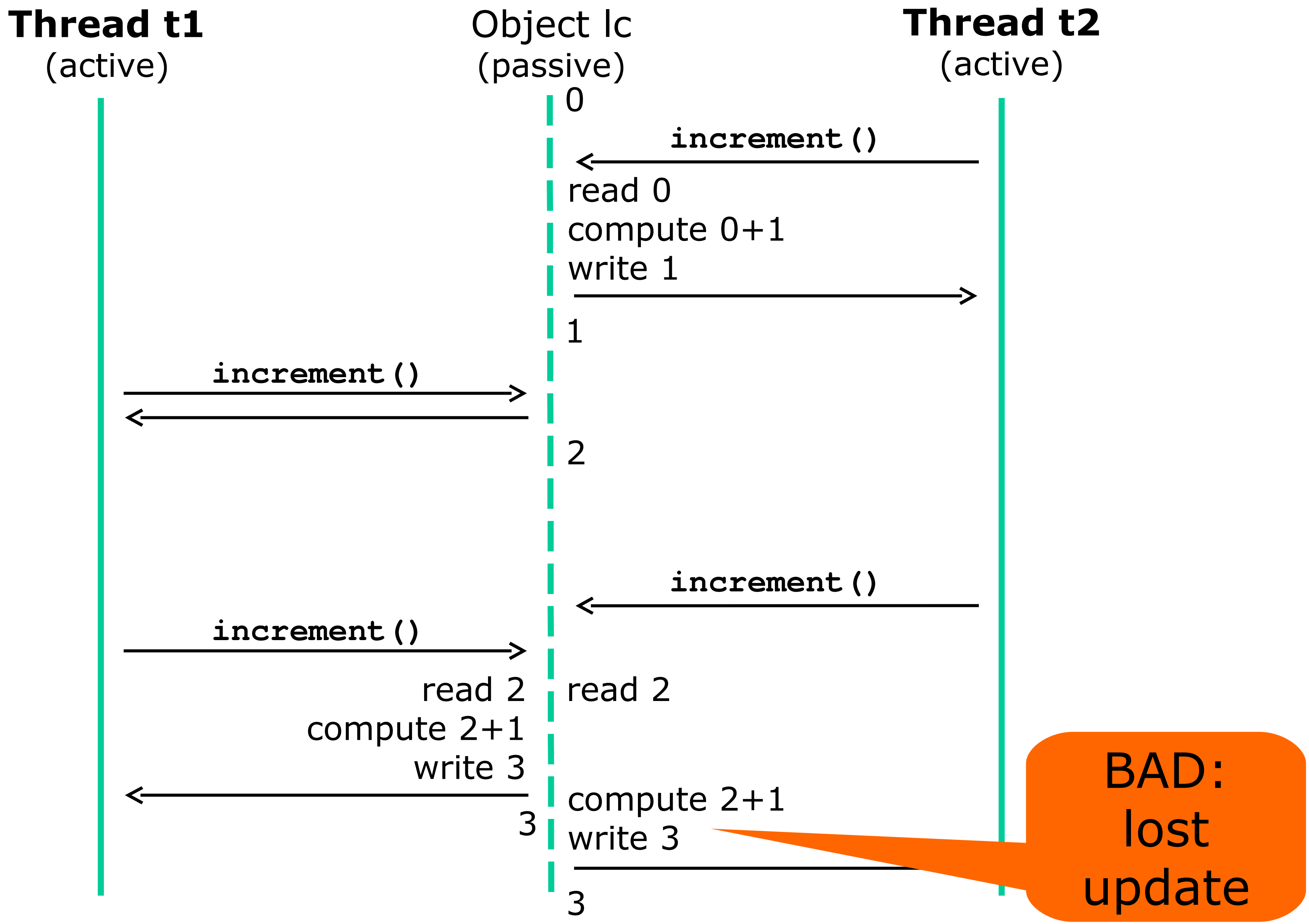
The operation
`count = count + 1` is not atomic

- What `count = count + 1` means:
  - read `count`
  - add 1
  - write result to `count`
- Hence *not atomic*
- So risk that two `increment()` calls will increase `count` by only 1

- NB: Same for `count += 1` and `count++`

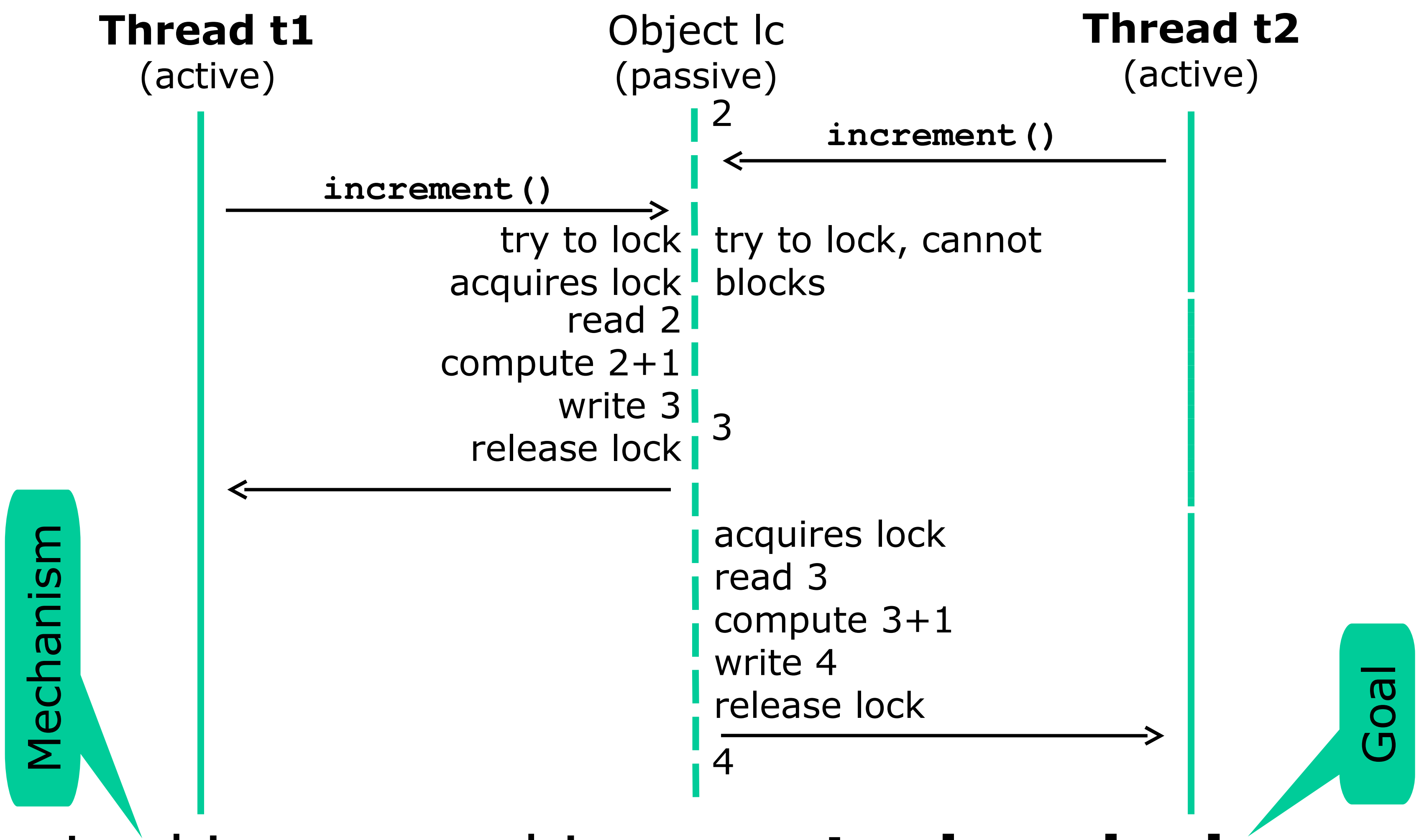
No locking: lost update

Without locking



How does locking help?

With locking



- Locking can achieve **mutual exclusion**
  - Lock on the same object before **all** state accesses
  - Unfortunately, quite easy to get it wrong

Why synchronize just to read data?

```
class LongCounter {
  private long count = 0;
  public synchronized void increment() {
    count = count + 1;
  }
  public synchronized long get() {
    return count;
  } }
```

Why needed?

TestLongCounter.java

- The **synchronized** keyword has **two** effects:
  - **Mutual exclusion**: only one thread can hold a lock (execute a synchronized method or block) at a time
  - **Visibility** of memory writes: All writes by thread A before releasing a lock (exit synchr) are visible to thread B after acquiring the lock (enter synchr)

Visibility is really important

WARNING: Useless

```
class MutableInteger {
  private int value = 0;
  public void set(int value) { this.value = value; }
  public int get() { return value; }
}
```

- Looks OK, no need for synchronization?
- But thread t may loop forever in this scenario:

```
final MutableInteger mi = new MutableInteger();
Thread t = new Thread(() -> {
  while (mi.get() == 0) { }
});
t.start();
mi.set(42);
```

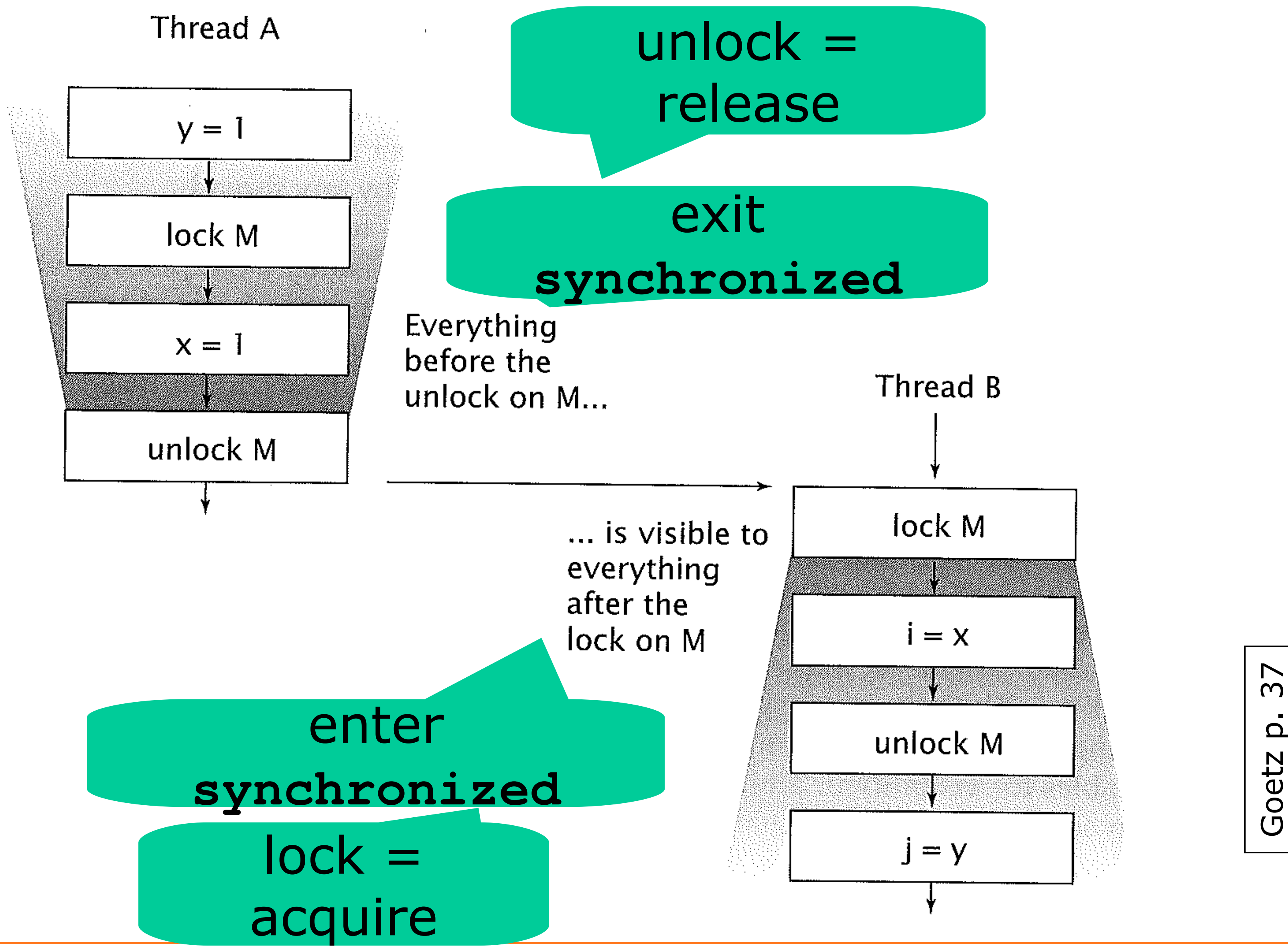
Loop while zero

This write by thread ”main” may be forever invisible to thread t

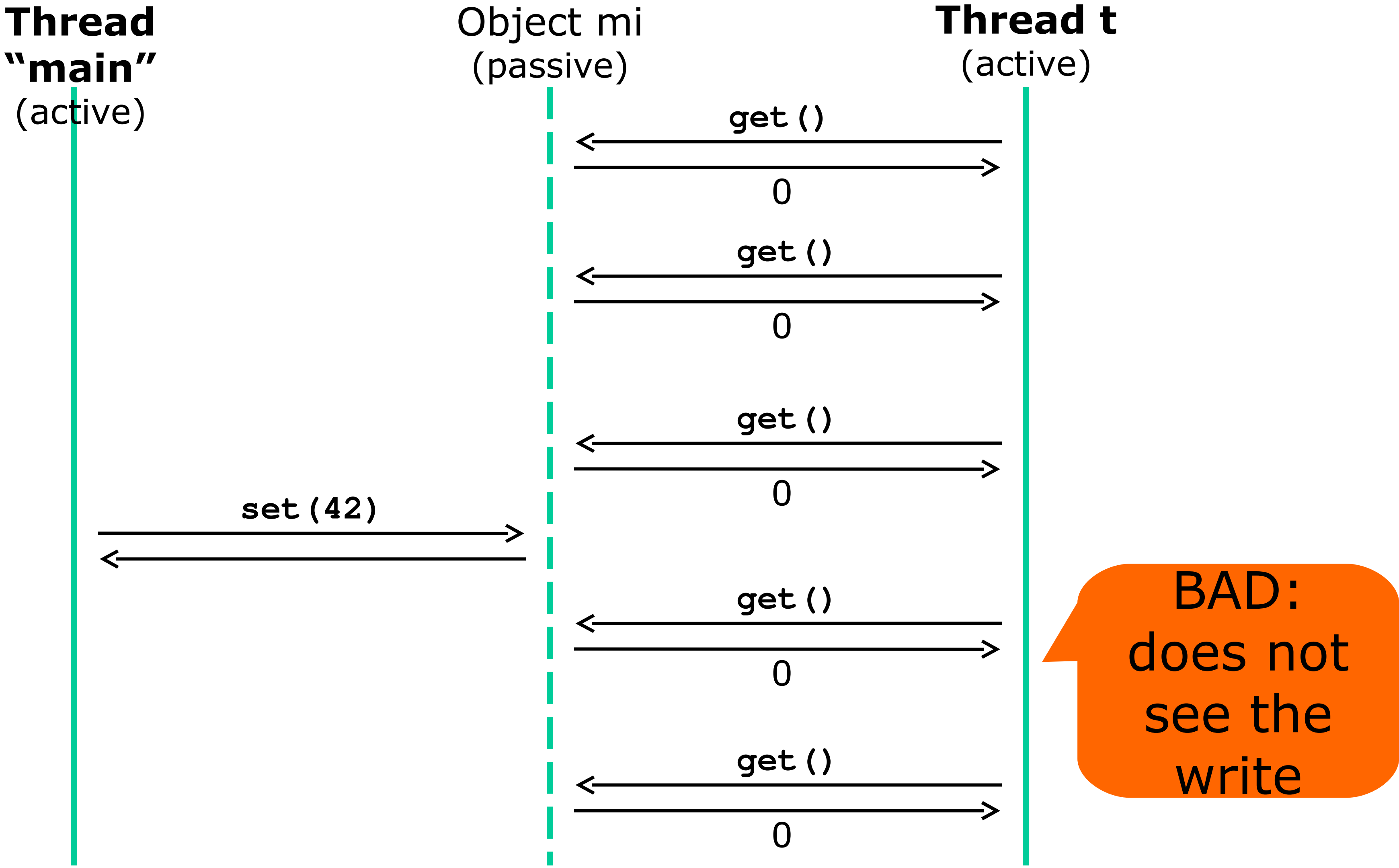
- Two possible fixes:
  - Add **synchronized** to methods **get** and **set**, OR
  - Add **volatile** to field **value**

TestMutableInteger.java

Visibility by synchronization



Communication through mutable shared state fails if no visibility



The volatile field modifier

- The **`volatile`** field modifier can be used to ensure visibility (but not mutual exclusion)

```
class MutableInteger {
  private volatile int value = 0;
  public void set(int value) { this.value = value; }
  public int get() { return value; }
}
```

OK

- All writes by thread A before writing a **`volatile`** field are visible to thread B when, and after, reading the **`volatile`** field
- Note: A single **`volatile`** write+read makes writes to all other fields visible also!
  - A bit mysterious, but a consequence of the implementation
  - This is Java semantics; C#, C, C++ volatile are different

Goetz advice on volatile

Use volatile variables only when they simplify your synchronization policy; avoid it when verifying correctness would require subtle reasoning about visibility.

Locking can guarantee both visibility and atomicity; volatile variables can only guarantee visibility.

Goetz p. 38, 39

- Rule 1: Use locks (**`synchronized`**)
- Rule 2: If circumstances are right, and you are an expert, maybe use **`volatile`** instead
- Rule 3: There are few experts

That was Java. What about C# and .NET?

- C# Language Spec. §17.3.4 *Volatile Fields*
- CLI Ecma-335 standard section §I.12.6.7:
  - "A volatile write has *release* semantics ... the write is guaranteed to happen *after* any memory references *prior* to the write instruction in the CIL instruction sequence"
  - "volatile read has *acquire* semantics ... the read is guaranteed to occur *prior* to any references to memory that occur *after* the read instruction in the CIL instruction sequence"

- C#'s **`volatile`** is weaker than Java's
  - And very unclearly described
  - Maybe use C# **`lock`** or MemoryBarrier() instead

Ways to ensure visibility

- Unlocking followed by locking the same lock
- Writing a volatile field and then reading it
- Calling one method on a concurrent collection and another method on same collection
  - java.util.concurrent.*
- Calling one method on an atomic variable and then another method on same variable
  - java.util.concurrent.atomic.*
- Finishing a constructor that initializes final or volatile fields
- Calling **`t.start()`** before anything in thread **`t`**
- Anything in thread **`t`** before **`t.join()`** returns

(Java Language Specification 8 §17.4, and the Javadoc for concurrent collection classes etc, give the full and rather complicated details)

Why "concurrent" and "parallel"?

- Informally both mean "at the same time"
- But some people distinguish
  - Concurrent: related to correctness
  - Parallel: related to performance
- Soccer (*fodbold*) analogy, by P. Panangaden
  - The referee (*dommer*) is concerned with concurrency: the soccer rules must be followed
  - The coach (*træner*) is concerned with parallelism: the best possible use of the team's 11 players
- This course is concerned with correctness as well as performance: concurrent and parallel

Processes, threads, and tasks

- An operating system **process** running Java is
  - a Java Virtual Machine that executes code
  - an object heap, managed by a garbage collector
  - one or more running Java threads
- A Java **thread**
  - has its own method call stack, takes much memory
  - shares the object heap with other threads
- A **task** (or future) (or actor)
  - does not have a call stack, so takes little memory
  - is run by an executor, using a thread pool, Week 5

This week

- Reading
  - Goetz chapters 1, 2 and 3
  - Sutter paper
  - Bloch item 66
- Exercises week 1, on homepage and LearnIT
  - Make sure you are familiar with Java threads and locks and inner classes
  - Make sure that you can compile, run and explain programs that use these features
- Read **before** next week's lecture
  - Goetz chapters 4 and 5
  - Bloch item 15