

Practical Concurrent and Parallel Programming 1

Peter Sestoft
IT University of Copenhagen

Friday 2014-08-29

Plan for today

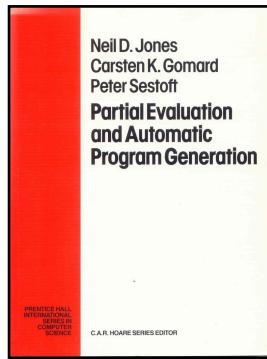
- Why this course?
- Course contents, learning goals
- Practical information
- Mandatory exercises, examination

- Java threads
- Java inner classes
- Java locking, the **synchronized** keyword

- Quizzes

The teachers

- Course responsible: Peter Sestoft
 - MSc 1988 and PhD 1991, Copenhagen University



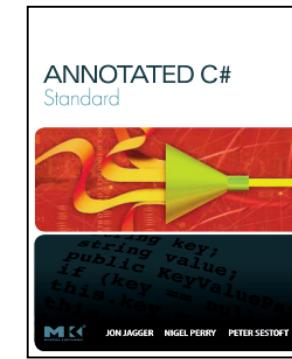
1993



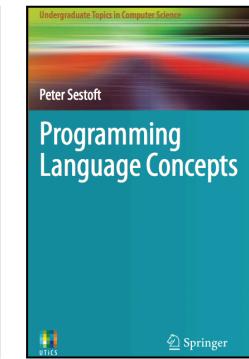
2002 & 2005



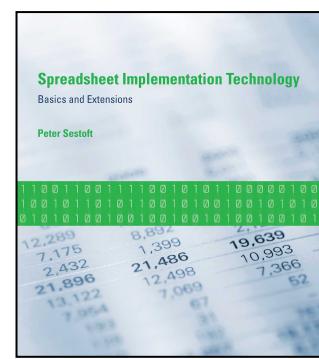
2004 & 2012



2007



2012



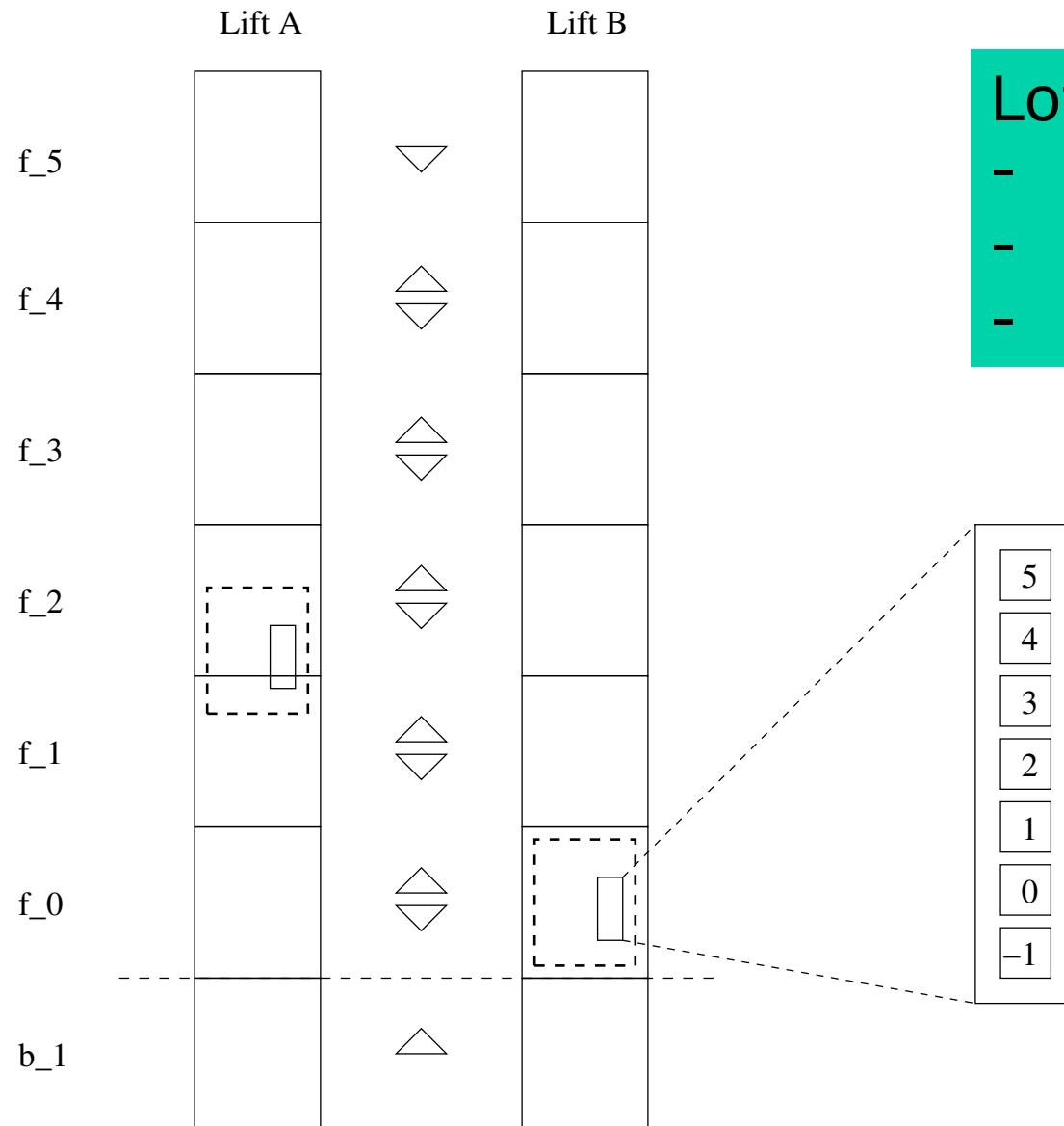
2014

- Co-teacher: Claus Brabrand
- Exercises
 - Iago Abal Rivas, ITU PhD student
 - Florian Biermann, research assistant, ex-ITU MSc
 - Håkan Lane, PhD, external teaching assistant

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

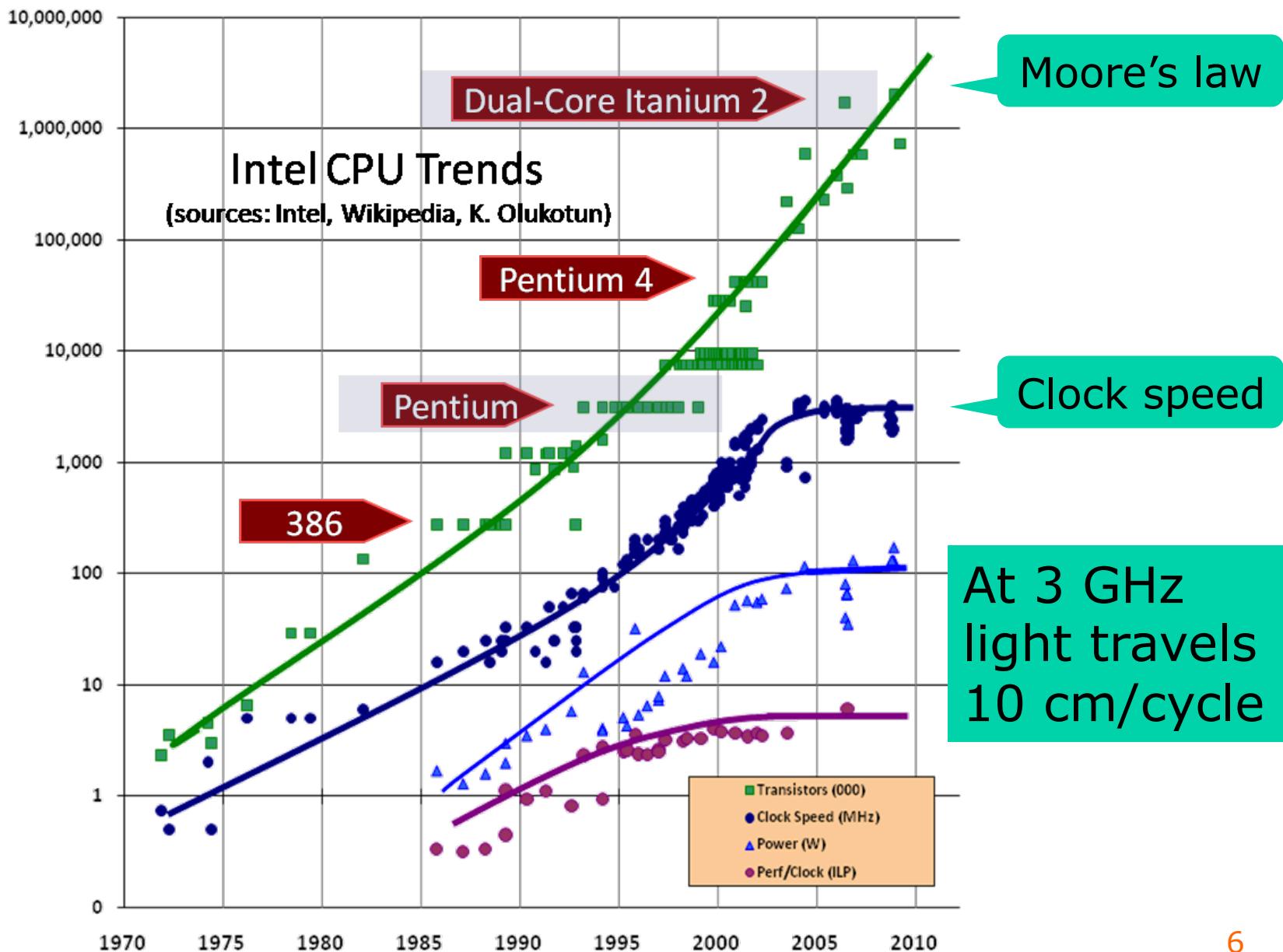


Lots of concurrency:

- 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
- Performance measurements
- Tasks, the Java executor framework
- Safety, liveness, deadlocks
- Testing concurrent programs, ThreadSafe
- Transactional memory, Multiverse
- Lock-free data structures, Java mem. model
- Message passing, Akka

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 review the basics of
 - Java threads
 - Java synchronized methods and statements
 - Java’s final field modifier
 - Java inner classes

Standard Friday plan

- Fridays until 5 December (except 17 Oct)
- Lectures 0800-1000
- Exercise startup
 - either 1000-1200 (Iago, Florian)
 - or 1200-1400 (Håkan)
- Exercise hand-in: 6.5 days after lecture
 - That is, the following Thursday at 23:55

Course information online

- Course LearnIT page, restricted access:
<https://learnit.itu.dk/course/view.php?id=3000701>
 - Exercises and hand-ins, deadlines, feedback
 - Mandatory exercises and hand-ins, deadlines, feedback
 - Discussion forum
 - Non-public reading materials
- Course homepage, public access:
<http://www.itu.dk/people/sestoft/itu/SPPP/E2014/>
 - 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
- Hand in the solutions through LearnIT
- You can work in teams of 1 or 2 students
- The teaching assistants will provide feedback
- Six of the 13 weekly exercise sets are mandatory
- At least five of those must be 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
 - What is important is that **you learn**

The exam

- A 53 hour take-home written exam/project
 - Start at 0900 on Wednesday 7 January 2015
 - End at 1400 on Friday 9 January
 - Electronic submission
- Expected exam workload is 22 hours
- Individual exam, no collaboration
- All materials, including Internet, allowed
- Always credit the sources you use
- Plagiarism is **forbidden** – as always

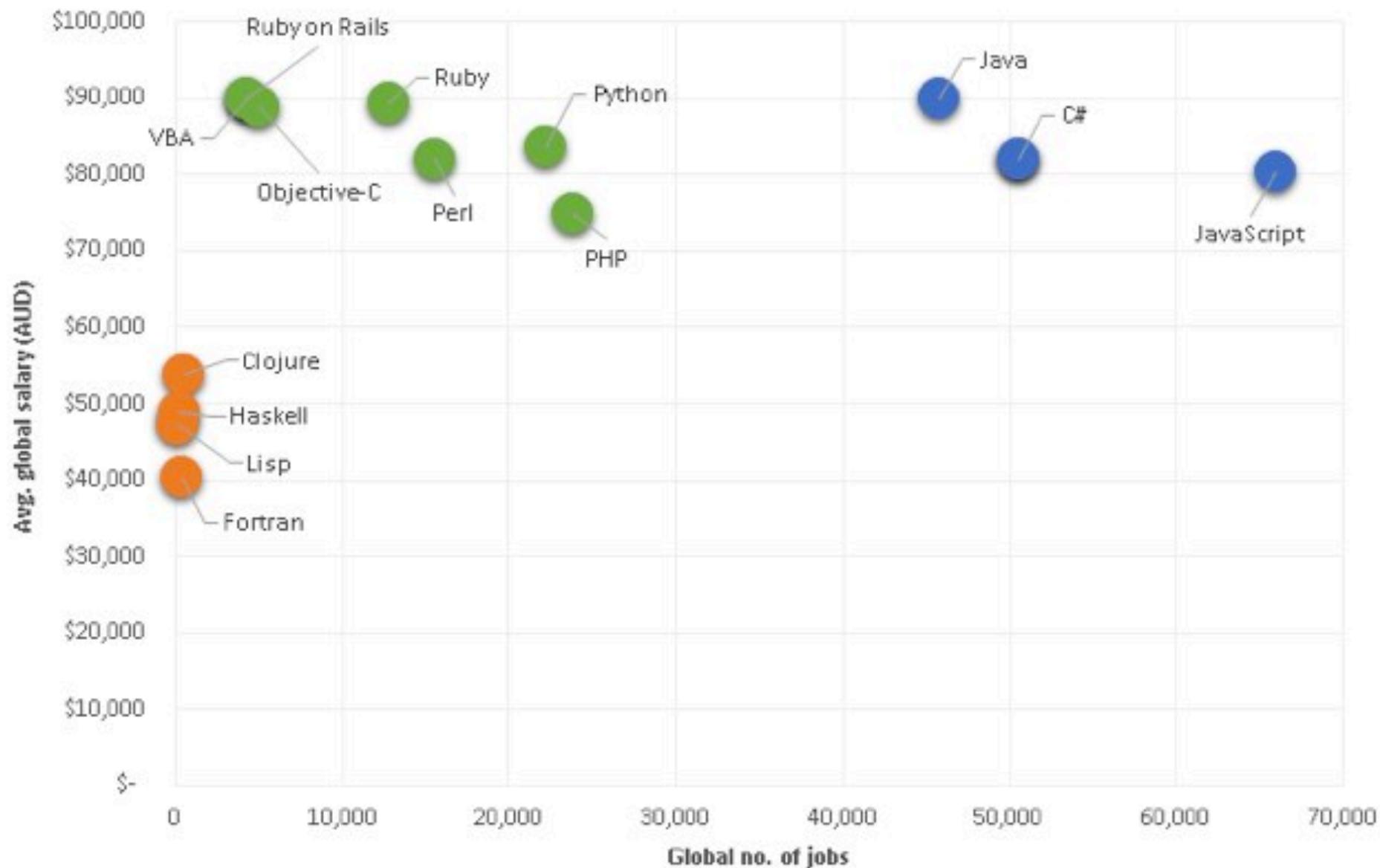
Stuff you need

- Goetz et al: *Java Concurrency in Practice*
 - From 2006, still the best on Java concurrency
 - Almost everything is relevant for C#/.NET too
- Free lecture notes and papers, see homepage
- A few other book chapters, see LearnIT
- Java 7 or 8 SDK installed on your computer
- Various optional materials, see homepage:
 - Bloch: *Effective Java*, 2008, **highly recommended**
 - Sestoft: *Java Precisely*, 2005
 - 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, C ω , ...
 - Zillions of other concurrency mechanisms

Salary and jobs by language



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

NB!

```
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

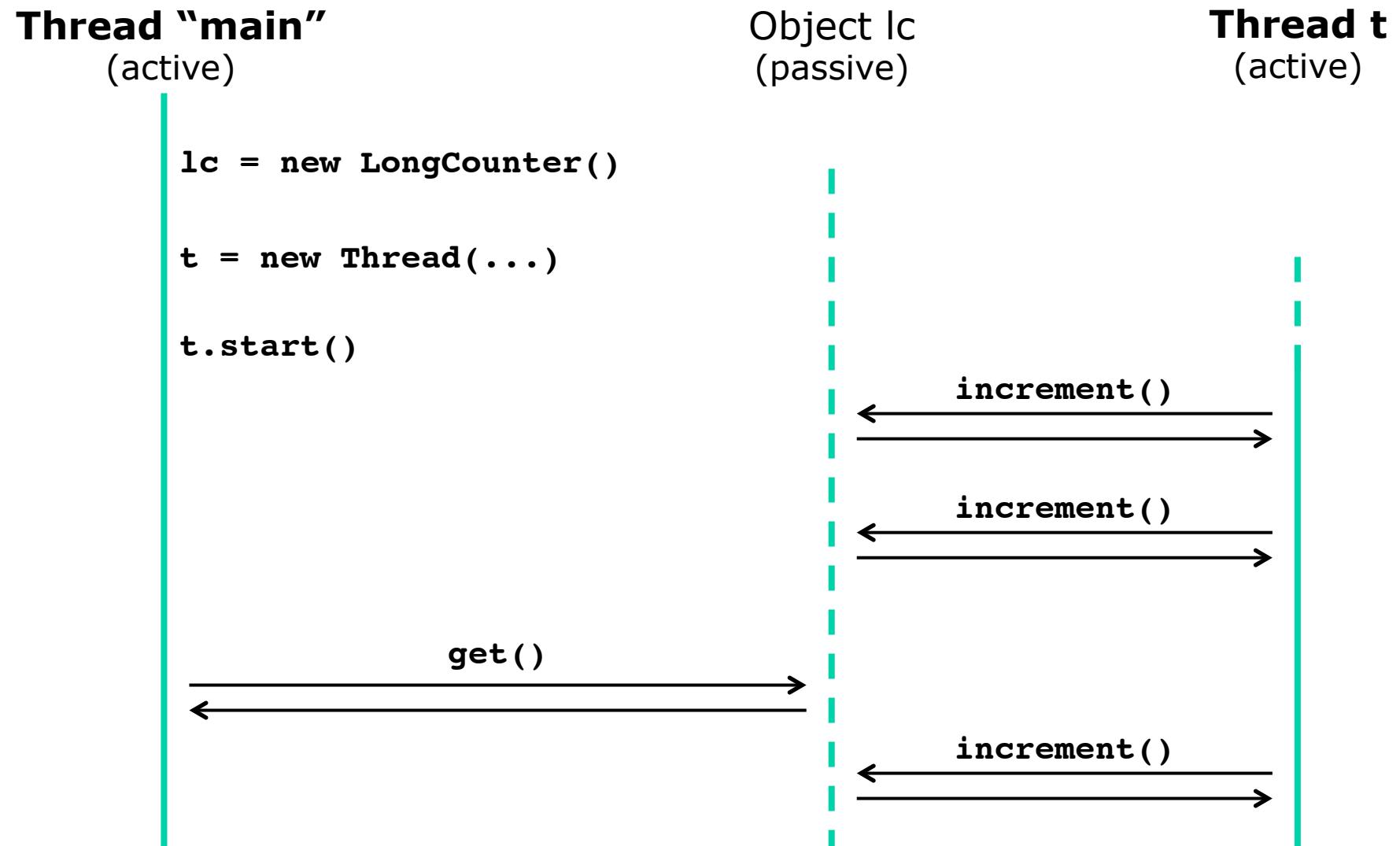
- This only *creates* the thread, does not *start* it
- Q: What does **final** mean?

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



Java (1-7) anonymous inner classes

- A statement must be in a method, eg. **run**

```
public interface Runnable {  
    public void run();  
}
```

- An anonymous inner class is a quick way to
 - create a class that implements Runnable *and*
 - create an instance of it

```
Runnable r =
```

```
new Runnable() {  
    public void run() {  
        ... some code we want to execute ...  
    }  
};
```

Anonymous inner class
and instance

Locks and the `synchronized` keyword

- 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 is just one with a synchronized 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 **static synchronized methods?**

- A synchronized static method in class C

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

locks on the reflected Class object for C:

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

- Not important to understand **c.class** because
 - Dangerous to share static fields between threads
 - Except possibly in factory methods

Multiple threads, locking

- Two threads incrementing counter in parallel:

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

count = count + 1 means:

read **count**, add 1, write result to **count**

- Possible scenario when **count** is 42, and two threads call **lc.increment()** at the same time

Thread 1	Thread 2
Read 42 from count	
	Read 42 from count
Add 1, giving 43	
	Add 1, giving 43
Write 43 to count	
	Write 43 to count

- Two increments but **count** only increased by 1
- So-called *lost update*

Why does locking help?

Thread 1	Thread 2
Try to lock, get lock	
Read 42 from count	
	Try to lock, cannot, must block
Add 1, giving 43	(blocked)
Write 43 to count	(blocked)
Release lock	(blocked)
	Get lock
	Read 42 from count
	Add 1, giving 43
	Write 43 to count
	Release lock

- Locking can achieve **mutual exclusion**
 - When used on **all** state accesses
 - Unfortunately, quite easy to get it wrong

Reads must be synchronized also

- A very small bank with two accounts:

```
class Bank {  
    private long account1 = 3000, account2 = 2000;  
    public synchronized void transfer(long amount) {  
        account1 -= amount;  
        account2 += amount;  
    }  
    public synchronized long getSum() {  
        return account1 + account2;  
    }  
}
```

Transfer, should preserve sum

Count sum of bank's money

TestAccountTransfer.java

- Common mistake to believe that only writes, method **transfer**, must be synchronized
- But we need **synchronized** on **both** methods!

Transferring money, and counting it, at the same time

- Transfer money, and concurrently count it:

```
final Bank bank = new Bank();
final int transfers = 10_000_000;
final Thread clerk = new Thread(new Runnable() {
    public void run() {
        for (int i=0; i<transfers; i++)
            bank.transfer(rnd.nextInt(10000));
    }
});
clerk.start();
for (int i=0; i<100; i++)
    System.out.println(bank.getSum());
```

Make many transfers

Print sum

- Q: Why must **both** Bank methods be synchr.?
- Even a single field read must be synchronized
 - But for another reason, see next week's lecture

Transferring money concurrently

- With a single clerk, final sum is 5000 always
- With two clerks, the final sum may be wrong
 - When **transfer** is not synchronized

```
final Thread clerk1 = new Thread(new Runnable() {  
    public void run() {  
        for (int i=0; i<transfers; i++)  
            bank.transfer(rnd.nextInt(10000));  
    } });  
final Thread clerk2 = ... exact same code ...  
clerk1.start(); clerk2.start();
```

- Q: What scenario may give wrong final sum?
- Simplified take-home message:
 - **All** reads and writes must be synchronized

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
 - Soon 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

This week

- Reading
 - Goetz chapters 1 and 2
 - 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
- Reading for next week
 - Goetz chapters 2 and 3
 - Bloch item 15

Practical Concurrent and Parallel Programming 2

Peter Sestoft
IT University of Copenhagen

Friday 2014-09-05**

Plan for today

- “concurrent” and “parallel”, what difference?
- Using threads for performance
- Processes, threads, tasks
- Atomically updating multiple fields
- Visibility of writes between threads
- `java.util.concurrent.atomic.AtomicLong`
- Safe publication
- Thread and stack confinement
- Immutability

Exercises

- Last week, problems with LearnIT, now fixed
- Hand-ins this week:
 - Must put yourself into a group, maybe 1-person
 - Your hand-in will automatically count for the group
- Last week's exercises:
 - Too easy?
 - Too hard?
 - Too time-consuming?
 - Too confusing?
 - Any particular problems?

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

Recall: Creating a thread, Java 1-7

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

NB!

```
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

New: Java 8 allows simpler syntax

- Java 8 anonymous functions may look better

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

The diagram shows the Java 8 anonymous function syntax with several annotations:

- A callout bubble points to the `() ->` part of the lambda expression, labeled "An anonymous **void** function".
- A callout bubble points to the block of code within the lambda, labeled "Function body".

- Use this if you want, else forget about it
- In Java 8, the **final** is sometimes not needed
 - If the captured variable (`lc`) is *effectively final*
 - That is, not assigned after initialization

Using threads for performance

Example: Count primes 2 3 5 7 11 ...

- Count primes in 0...9999999

```
static long countSequential(int range) {  
    long count = 0;  
    final int from = 0, to = range;  
    for (int i=from; i<to; i++)  
        if (isPrime(i))  
            count++;  
    return count;  
}
```

TestCountPrimes.java

Result is 664579

- Takes 6.4 sec to compute on 1 CPU core
- Why not use all my computer's 4 (x 2) cores?
 - Eg. use two threads t1 and t2 and divide the work:
t1: 0...4999999 and t2: 5000000...9999999

Using two threads to count primes

```
final LongCounter lc = new LongCounter();
final int from1 = 0, to1 = perThread;
Thread t1 = new Thread(new Runnable() { public void run() {
    for (int i=from1; i<to1; i++)
        if (isPrime(i))
            lc.increment();
}});

final int from2 = perThread, to2 = perThread * 2;
Thread t2 = new Thread(new Runnable() { public void run() {
    for (int i=from2; i<to2; i++)
        if (isPrime(i))
            lc.increment();
}});

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

Same code twice,
bad practice

- Takes 4.2 sec real time, so already faster
- Q: Why not just use a **long count** variable?
- Q: What if we want to use 10 threads?

Using N threads to count primes

```
final LongCounter lc = new LongCounter();
Thread[] threads = new Thread[threadCount];
for (int t=0; t<threadCount; t++) {
    final int from = perThread * t,
        to = (t+1==threadCount) ? range : perThread * (t+1);
    threads[t] = new Thread(new Runnable() { public void run()
        for (int i=from; i<to; i++)
            if (isPrime(i))
                lc.increment();
    });
}
for (int t=0; t<threadCount; t++)
    threads[t].start();
```

Last thread has
to==range

Thread processes
segment [from,to)

- Takes 1.8 sec real time with **threadCount** 10
 - Approx 3.3 times faster than sequential solution
 - Q: Why not 4 times, or 10 times faster?
 - Q: What if we just put to=perThread * (t+1)?

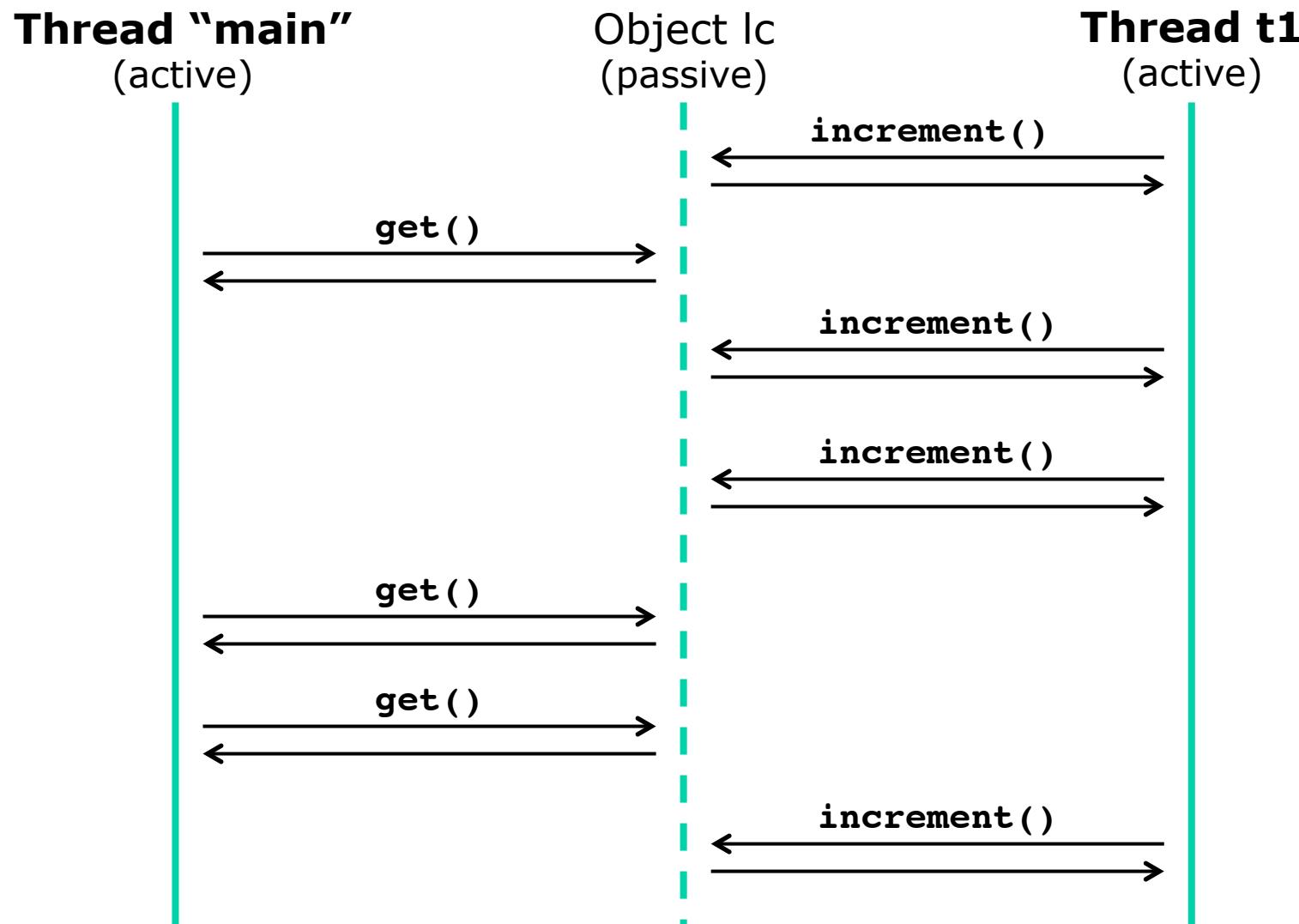
Reflections: threads for performance

- This code can be made better in many ways
 - Eg better distribution of work on the 10 threads
 - Eg less use of the synchronized LongCounter
- Proper performance measurements, **week 4**
- Very bad idea to use many (> 500) threads
 - Each thread takes much memory for the stack
 - Each thread slows down the garbage collector
- Better use *tasks* and Java “executors”, **week 5**
- More advice on scalability, **week 7**
- How to avoid locking, **week 11 and 12**
- (Prime numbers used as example for simplicity)

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

Java threads communicate through mutable shared state



Last week's LongCounter

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

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

WARNING: Useless

TestMutableInteger.java

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

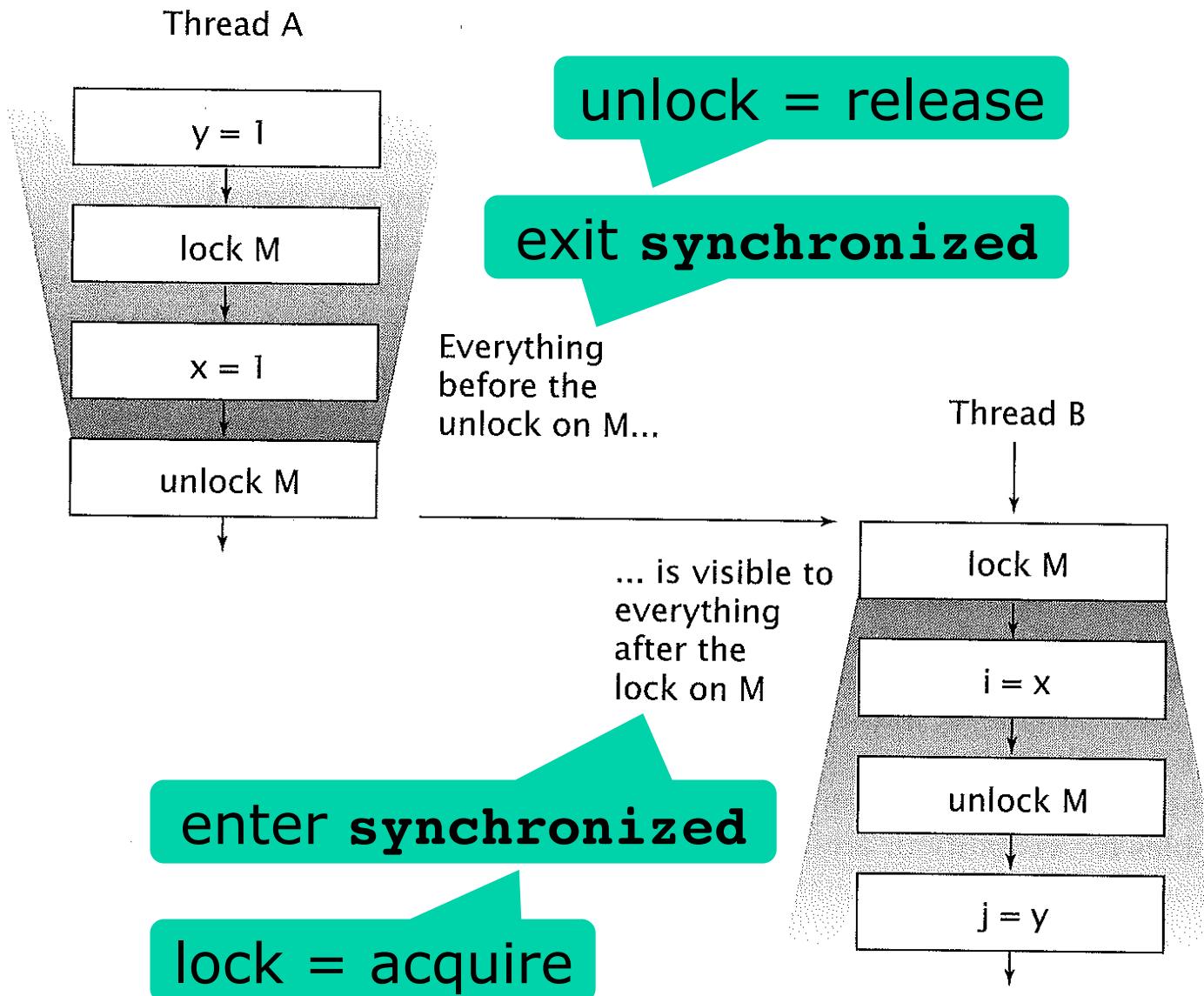
```
Thread t = new Thread(new Runnable() { public void run() {  
    while (mi.get() == 0) { }  
}});  
t.start();  
...  
mi.set(42);
```

Loop while zero

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

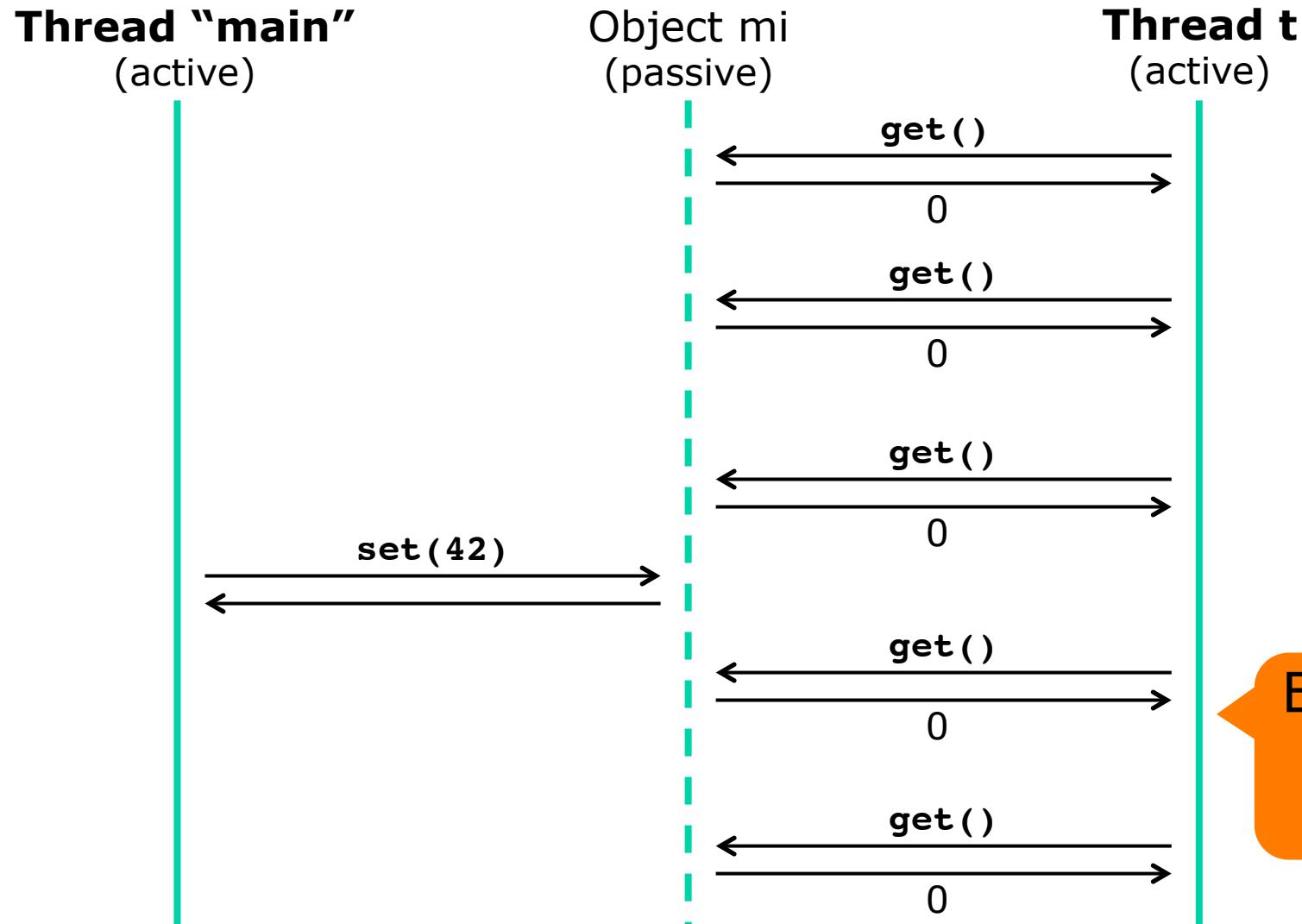
- Two possible fixes:
 - Add **synchronized** to methods **get** and **set**, OR
 - Add **volatile** to field **value**

Visibility by synchronization



Goetz p. 37

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 and C++ volatile is 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 **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 Specification 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"
- So same as Java: volatile write+read has the visibility effect of lock release+acquire
 - (but not the mutual exclusion effect, of course)

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 coll.
 - `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; week 11)

Goetz examples use servlets

```
public class StatelessFactorizer implements Servlet {  
    public void service(ServletRequest req, ServletResponse resp) {  
        BigInteger i = extractFromRequest(req);  
        BigInteger[] factors = factor(i);  
        encodeIntoResponse(resp, factors);  
    }  
}
```

Goetz p. 19

- Because a webserver is naturally concurrent
 - So servlets should be thread-safe
- We use similar, simpler examples:

```
class StatelessFactorizer implements Factorizer {  
    public long[] getFactors(long p) {  
        long[] factors = PrimeFactors.compute(p);  
        return factors;  
    }  
}
```

TestFactorizer.java

A “server” for computing prime factors 2 3 5 7 11 ... of a number

- Could replace the example by this

```
interface Factorizer {  
    public long[] getFactors(long p);  
    public long getCount();  
}
```

- Call the server from multiple threads:

```
final Factorizer factorizer = new StatelessFactorizer();  
for (int t=0; t<threadCount; t++) {  
    threads[t] = new Thread(new Runnable() { public void run()  
        for (int i=2; i<range; i++) {  
            long[] result = factorizer.getFactors(i);  
        }  
    } );
```

Stateless objects are thread-safe

```
class StatelessFactorizer implements Factorizer {  
    public long[] getFactors(long p) {  
        long[] factors = PrimeFactors.compute(p);  
        return factors;  
    }  
    public long getCount() { return 0; }  
}
```

Like Goetz p. 18

- Local variables are never shared btw threads
 - two getFactors calls can execute at the same time

Bad attempt to count calls

```
class UnsafeCountingFactorizer implements Factorizer {  
    private long count = 0;  
    public long[] getFactors(long p) {  
        long[] factors = PrimeFactors.compute(p);  
        count++;  
        return factors;  
    }  
    public long getCount() { return count; }  
}
```

Like Goetz p. 19

- Not thread-safe
- Q: Why?
- Q: How could we repair the code?

Thread-safe server counting calls

```
class CountingFactorizer implements Factorizer {  
    private final AtomicLong count = new AtomicLong(0);  
    public long[] getFactors(long p) {  
        long[] factors = PrimeFactors.compute(p);  
        count.incrementAndGet();  
        return factors;  
    }  
    public long getCount() { return count.get(); }  
}
```

Like Goetz p. 23

- `java.util.concurrent.atomic.AtomicLong` supports atomic thread-safe arithmetics
- Similar to an improved `LongCounter` class

Bad attempt to cache last factorization

```
class TooSynchronizedCachingFactorizer implements Factorizer {
    private long lastNumber = 1;
    private long[] lastFactors = new long[0];
    // Invariant: product(lastFactors) == lastNumber }
```

cache

```
public synchronized long[] getFactors(long p) {
    if (p == lastNumber)
        return lastFactors.clone();
    else {
        long[] factors = PrimeFactors.compute(p);
        lastNumber = p;
        lastFactors = factors;
        return factors;
    }
}
```

Like Goetz p. 26

Without synchronized the two fields could be written by different threads

- Bad performance: no parallelism at all
- Q: Why? Q: What is an invariant?

Atomic operations

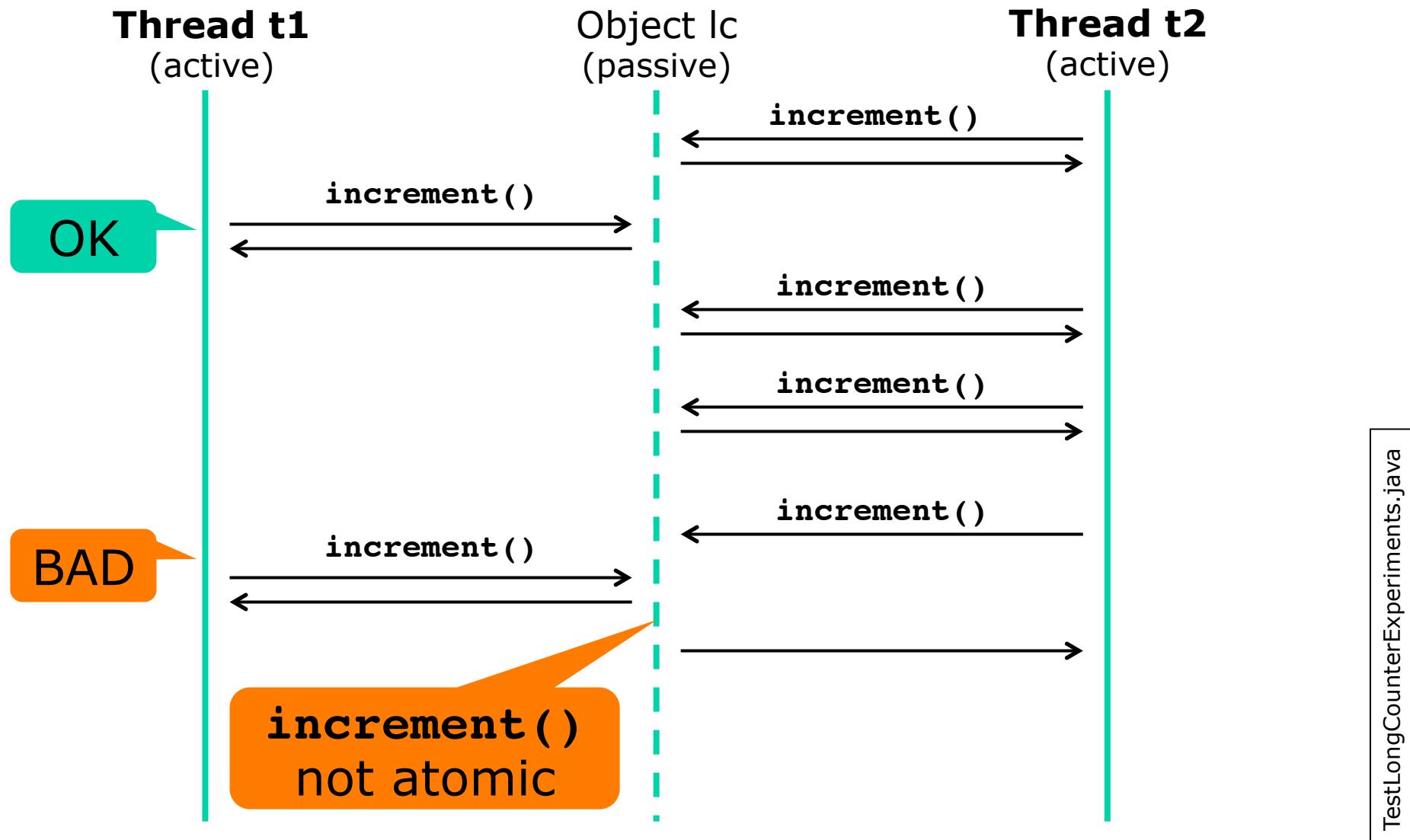
- We want to *atomically* update `lastNumber` and `lastFactors`

Operations A and B are *atomic* with respect to each other if, from the perspective of a thread executing A, when another thread executes B, either all of B has executed or none of it has.

An *atomic operation* is one that is atomic with respect to all operations, including itself, that operate on the same state.

Goetz p. 22, 25

Lack of atomicity: overlapping reads and writes



Atomic update without excess locking

```
class CachingFactorizer implements Factorizer {  
    private long lastNumber = 0;  
    private long[] lastFactors = new long[0];  
    public long[] getFactors(long p) {  
        long[] factors = null;  
        synchronized (this) {  
            if (p == lastNumber)  
                factors = lastFactors.clone();  
        }  
        if (factors == null) {  
            factors = PrimeFactors.compute(p);  
            synchronized (this) {  
                lastNumber = p;  
                lastFactors = factors.clone();  
            }  
        }  
        return factors;  
    } }  
}
```

Atomic
test-then-act

Atomic write
of both fields

Like Goetz p. 31

- Correct but subtle

Using locks for atomicity

For each mutable state variable that may be accessed by more than one thread, *all* accesses to that variable must be performed with the *same* lock held. Then the variable is *guarded* by that lock.

Goetz p. 28, 29

For every invariant that involves more than one variable, *all* the variables involved in that invariant must be guarded by the *same* lock.

- Common mis-reading and mis-reasoning:
 - The *purpose* of **synchronized** is to get atomicity
 - So **synchronized** roughly means “**atomic**”
 - True only if **all other** accesses are **synchronized**!!!

Wrapping the state in an immutable object

NB!

```
class OneValueCache {  
    private final long lastNumber;  
    private final long[] lastFactors;  
    public OneValueCache(long p, long[] factors) {  
        this.lastNumber = p;  
        this.lastFactors = factors.clone();  
    }  
    public long[] getFactors(long p) {  
        if (lastFactors == null || lastNumber != p)  
            return null;  
        else  
            return lastFactors.clone();  
    }  
}
```

Nothing can
change between
test and return

Q: Why?

- Immutable, so automatically thread-safe

Make the state a single field, referring to an immutable object

NB!

```
class VolatileCachingFactorizer implements Factorizer {  
    private volatile OneValueCache cache  
        = new OneValueCache(0, null);  
    public long[] getFactors(long p) {  
        long[] factors = cache.getFactors(p);  
        if (factors == null) {  
            factors = PrimeFactors.compute(p);  
            cache = new OneValueCache(p, factors);  
        }  
        return factors;  
    }  
}
```

Single-field state,
atomic assignment

Atomic assignment

Like Goetz p. 50

- Only one mutable field, atomic assignment
- Easy to implement, easy to see it is correct
- Drawback: cost of creating cache objects
 - Not a problem with modern garbage collectors

Immutability

- OOP: An object has state, held by its fields
 - Fields should be **private** for encapsulation
 - It is common to define getters and setters
- But mutable state causes lots of problems
 - So make fields **final** and remove the setters

Immutable objects are always thread-safe.

An object is *immutable* if:

- Its state cannot be modified after construction
- All its fields are **final**
- It is properly constructed (**this** does not escape)

Goetz p. 46, 47

Bloch: Effective Java, item 15

Item 15: Minimize mutability

An immutable class is simply a class whose instances cannot be modified. All of the information contained in each instance is provided when it is created and is fixed for the lifetime of the object. The Java platform libraries contain many immutable classes, including `String`, the boxed primitive classes, and `BigInteger` and `BigDecimal`. There are many good reasons for this: Immutable classes are easier to design, implement, and use than mutable classes. They are less prone to error and are more secure.

To make a class immutable, follow these five rules:

1. **Don't provide any methods that modify the object's state** (known as *mutators*).
2. **Ensure that the class can't be extended.** This prevents careless or malicious subclasses from compromising the immutable behavior of the class by behaving as if the object's state has changed. Preventing subclassing is generally accepted as a good idea, so this rule is often summarized as "final".
3. **Make all fields private.** Immutable classes provide many advantages, and their only disadvantage is that they are less flexible than mutable classes under certain circumstances. You are forced by the system. Also, it is necessary to ensure correct behavior if a reference to a newly created instance is passed from one thread to another without synchronization, as spelled out in the *memory model* [JLS, 17.5; Goetz06 16].
4. **Make all fields private.** This prevents clients from obtaining access to muta-

Josh Bloch
designed the Java
collection classes

A serious Java (or
C#) developer
should own and
use this book

Safe publication: visibility

- The `final` field modifier has two effects
 - **Un-updatability** can be checked by the compiler
 - **Visibility** from other threads of the fields' values after the `OneValueCache` constructor returns
- So `final` has visibility effect like `volatile`
- Without `final` or synchronization, another thread may not see the given field values
- That was Java. What about C#/.NET?
 - No visibility effect of `readonly` field modifier
 - So must be ensured by `volatile` or synchronization
 - Seems a little dangerous?

Avoiding shared mutable state

- Avoiding sharing between threads:
 - Ad hoc thread confinement: Swing GUI components are accessed only by the GUI thread
 - Thread confinement via ThreadLocal objects
 - Stack confinement: Local variables are never shared between threads
- Avoiding mutable state:
 - Make fields final as far as possible
 - Replace multiple mutable fields by a single mutable reference to an immutable object

Why `.clone()` in the factorizers?

```
public long[] getFactors(long p) {  
    ...  
    factors = lastFactors.clone();  
    ...  
    lastFactors = factors.clone();  
    ...  
}
```

- Because Java array elements are mutable
- So unsafe to share an array with anybody
- Must defensively clone the array when passing a reference to some other part of the program
- This is a problem in sequential code too, only much worse in concurrent code
 - Minimize Mutability! More about this next week ...

This week

- Reading
 - Goetz et al chapters 2 and 3
 - Bloch item 15
- Exercises
 - Mandatory hand-in Thursday at 23:55
 - Goals: Understand and use multiple threads for performance; visibility of concurrent writes; atomicity by locking; advantages of immutability
- Reading for next week
 - Goetz chapters 4 and 5

Practical Concurrent and Parallel Programming 3

Peter Sestoft
IT University of Copenhagen

Friday 2014-09-12*

Plan for today

- Java Monitor pattern
- Defensive copying, VehicleTracker
- Standard collection classes not thread-safe
- Extending collection classes
- ConcurrentModificationException
- FutureTask<T>, asynchronous execution
- (Silly complications of checked exceptions)
- Building a scalable result cache

Comments on exercises

- Exercise schedule:
 - 1000-1200: 2A14
 - 1200-1400: **2A14 ← change!**
- True:
 - If program p fails when tested, then it is not thread-safe
- **False:**
 - If program p does not fail when tested, then it is thread-safe

NEVER reason like that

Java monitor pattern

An object following the *Java monitor pattern* encapsulates all its mutable state (in **private** fields) and guards it with the object's own intrinsic lock (**synchronized**).

Goetz p. 60

- Monitors invented 1974 by Hansen and Hoare
 - A way to encapsulate mutable state in concurrency
- Java monitor pattern implements monitors
 - If you use care and discipline!
 - Per Brinch Hansen critical of Java, 1999 paper
- Modern (Java) data structures are subtler ...
 - Illustrated by Goetz VehicleTracker example

LongCounter as monitor, and documenting thread-safety

- Use the @GuardedBy annotation on fields:

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

ThreadsafelongCounter.java

- Compile files with

```
javac -cp ~/lib/jsr305-3.0.0.jar ThreadsafelongCounter.java
```

- Annotations show the programmer's *intent*
 - Annotations are **not** checked by the Java compiler
 - Week 6 we see a tool for checking @GuardedBy

A class of mutable points

- MutablePoint, like `java.awt.Point`

Design mistake

```
class MutablePoint {  
    public int x, y;  
    public MutablePoint() {  
        x = 0; y = 0;  
    }  
    public MutablePoint(MutablePoint p) {  
        this.x = p.x; this.y = p.y;  
    }  
}
```

Not thread-safe

TestVehicleTracker.java

Goetz p. 64

- Q: Why not thread-safe?

Vehicle tracker as a monitor class

V1

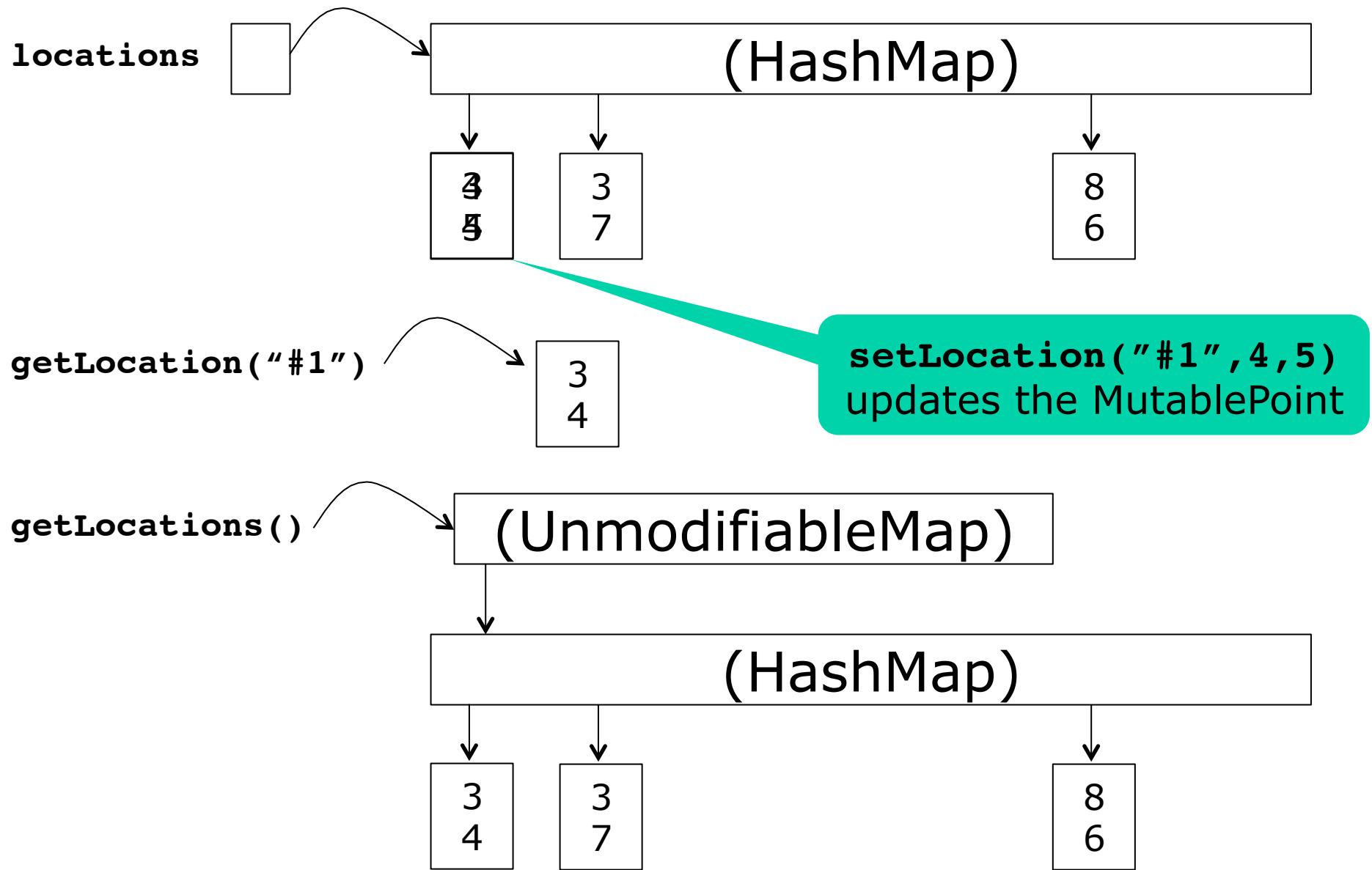
```
class MonitorVehicleTracker {  
    private final Map<String, MutablePoint> locations;  
    public MonitorVehicleTracker(Map<String, MutablePoint> locations) {  
        this.locations = deepCopy(locations);  
    }  
    public synchronized Map<String, MutablePoint> getLocations() {  
        return deepCopy(locations);  
    }  
    public synchronized MutablePoint getLocation(String id) {  
        MutablePoint loc = locations.get(id);  
        return loc == null ? null : new MutablePoint(loc);  
    }  
    public synchronized void setLocation(String id, int x, int y) {  
        MutablePoint loc = locations.get(id);  
        loc.x = x;  
        loc.y = y;  
    }  
    private static Map<String, MutablePoint> deepCopy(Map<String, MutablePoint> m) {  
        Map<String, MutablePoint> result = new HashMap<String, MutablePoint>();  
        for (String id : m.keySet())  
            result.put(id, new MutablePoint(m.get(id)));  
        return Collections.unmodifiableMap(result);  
    }  
}
```

Goetz p. 63

TestVehicleTracker.java

- Protects its state in field locations
- But why all that copying?

MonitorVehicleTracker memory



A class of immutable points

- Immutable Point class:

```
class Point {  
    public final int x, y;  
    public Point(int x, int y) {  
        this.x = x; this.y = y;  
    }  
}
```

TestVehicleTracker.java

Goetz p. 64

- Automatically thread-safe

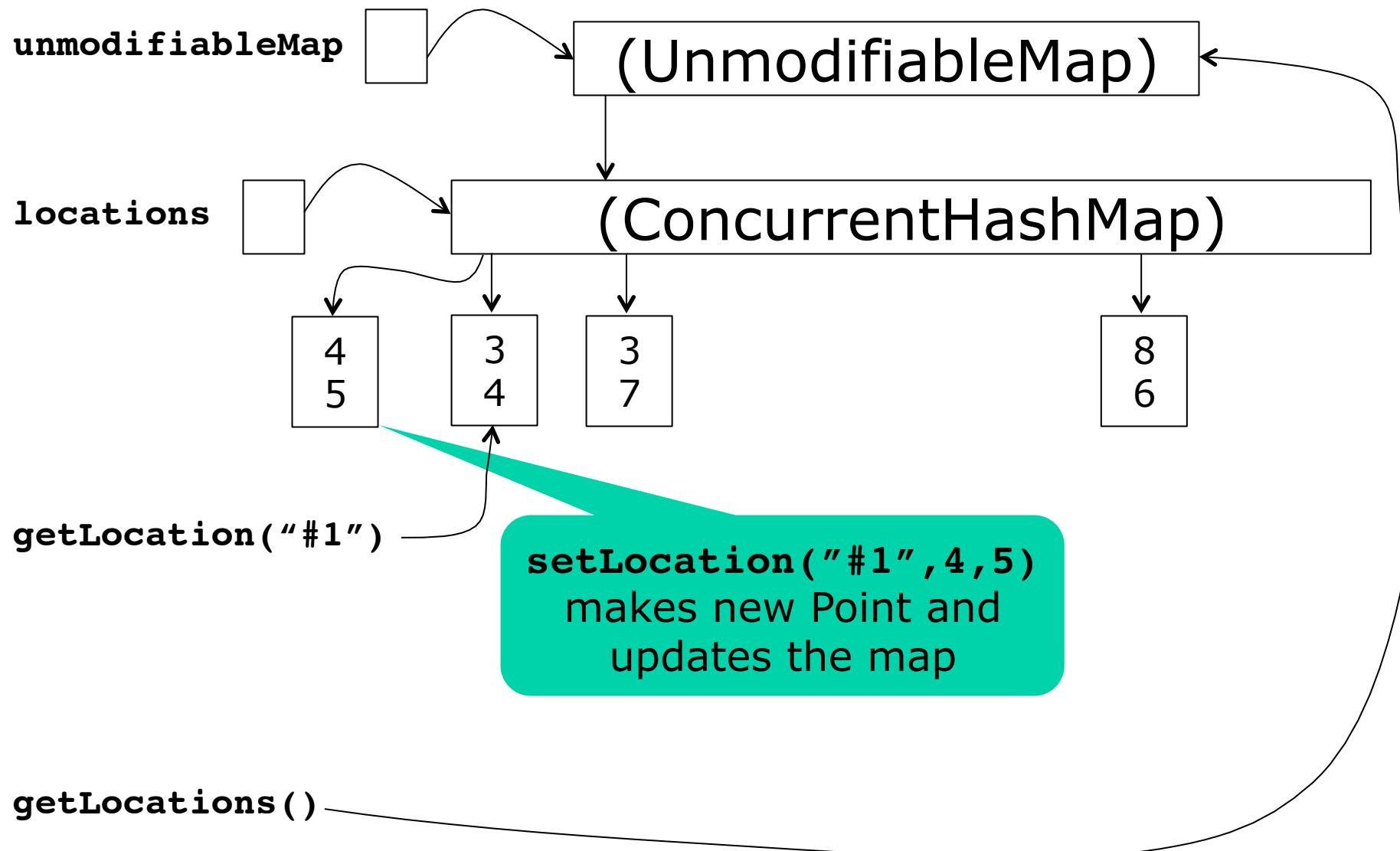
Thread safety by delegation and immutable points

```
class DelegatingVehicleTracker {  
    private final ConcurrentHashMap<String, Point> locations;  
    private final Map<String, Point> unmodifiableMap;  
    public DelegatingVehicleTracker(Map<String, Point> points) {  
        locations = new ConcurrentHashMap<String, Point>(points);  
        unmodifiableMap = Collections.unmodifiableMap(locations);  
    }  
    public Map<String, Point> getLocations() {  
        return unmodifiableMap;  
    }  
    public Point getLocation(String id) {  
        return locations.get(id);  
    }  
    public void setLocation(String id, int x, int y) {  
        locations.replace(id, new Point(x, y));  
    }  
}
```

Goetz p. 65

- No defensive copying any longer
 - Less mutability can give better performance!
- Q: Why not just cast **locations** to an interface without setters?

DelegatingVehicleTracker memory



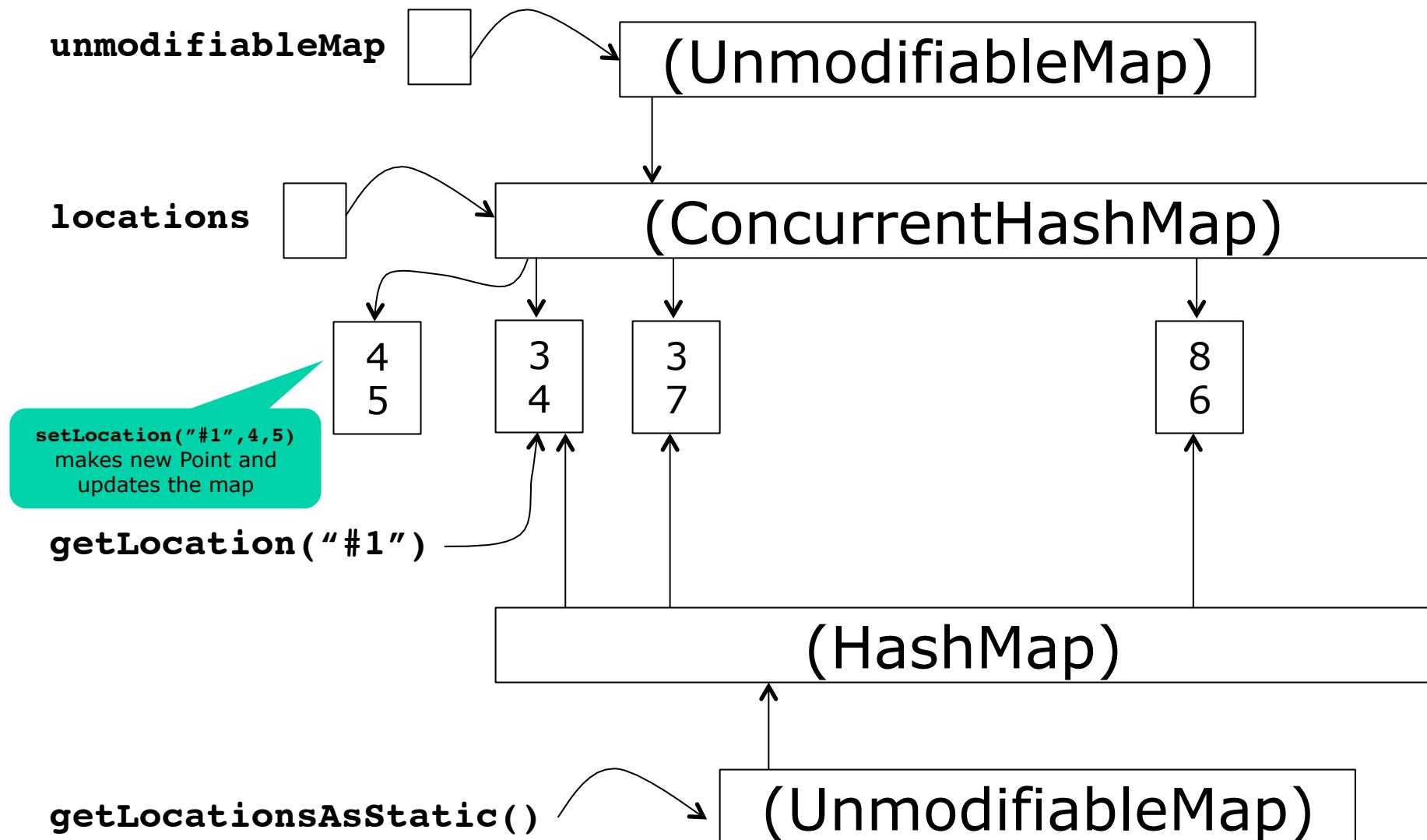
Alternative `getLocations()`

- Returns unmodifiable view
 - of static copy of hashmap,
 - referring to the existing immutable points

```
public Map<String, Point> getLocationsAsStatic() {  
    return Collections.unmodifiableMap(new HashMap<String, Point>(locations));  
}
```

Goetz p. 66

DelegatingVehicleTracker memory with static getLocations result



Immutability is GOOD

- Can speed up some operations
- Can simplify thread-safety
- Microsoft .NET has new immutable collections
 - <http://msdn.microsoft.com/en-us/library/dn385366%28v=vs.110%29.aspx>
 - <http://blogs.msdn.com/b/bclteam/archive/2012/12/18/preview-of-immutable-collections-released-on-nuget.aspx>
- Different from unmodifiable collections
 - No underlying modifiable collection
 - Enumeration is safe, including thread-safe
- Java 8 does not have immutable collections

Safe mutable point class

- Mutable point as monitor

```
public class SafePoint {  
    private int x, y;  
    private SafePoint(int[] a) { this(a[0], a[1]); }  
    public SafePoint(SafePoint p) { this(p.get()); }  
    public SafePoint(int x, int y) { this.set(x, y); }  
    public synchronized int[] get() {  
        return new int[]{x, y};  
    }  
    public synchronized void set(int x, int y) {  
        this.x = x; this.y = y;  
    }  
}
```

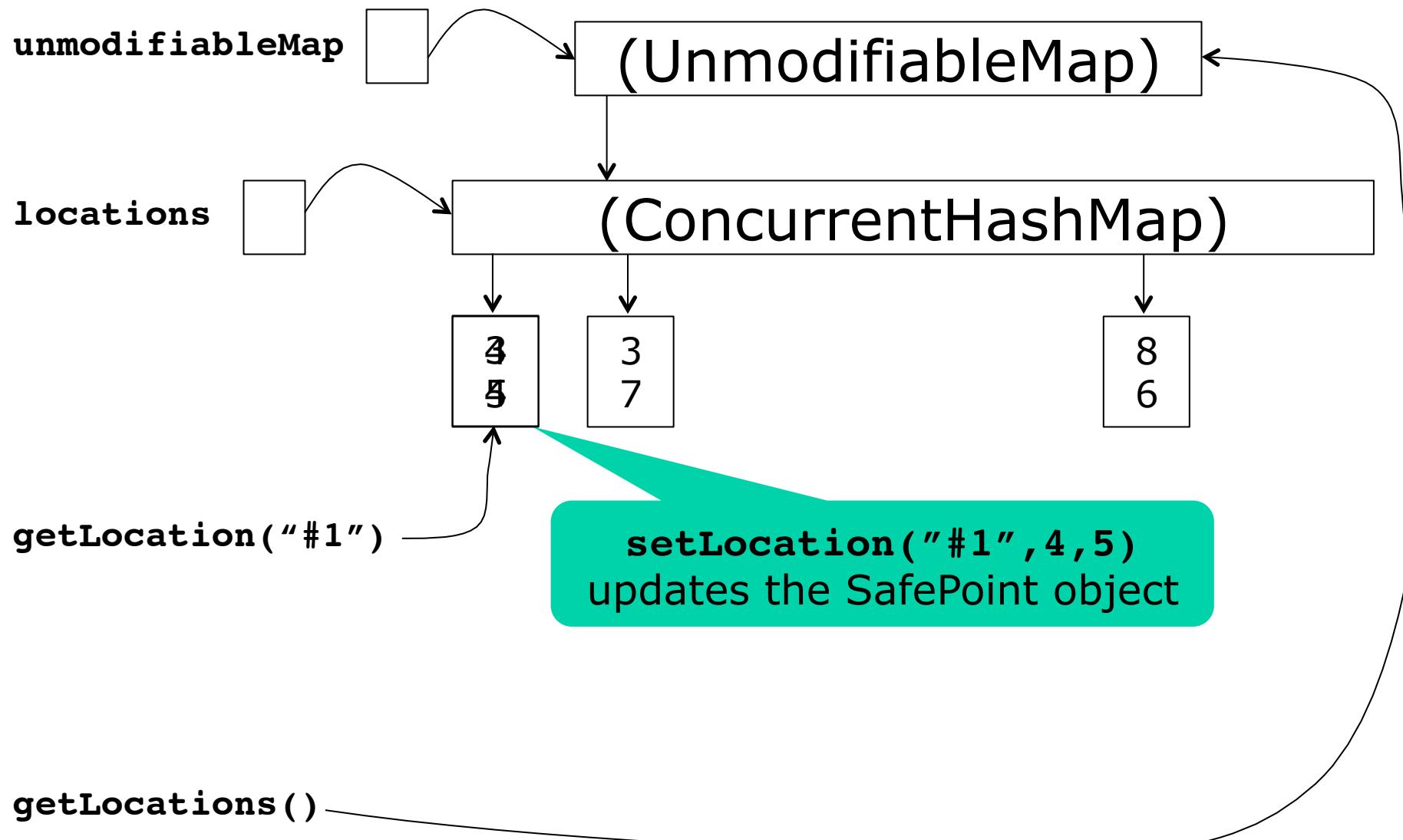
Goetz p. 69

Safe publishing vehicle tracker

```
public class PublishingVehicleTracker {  
    private final Map<String, SafePoint> locations;  
    private final Map<String, SafePoint> unmodifiableMap;  
  
    public PublishingVehicleTracker(Map<String, SafePoint> locations) {  
        this.locations  
            = new ConcurrentHashMap<String, SafePoint>(locations);  
        this.unmodifiableMap = Collections.unmodifiableMap(this.locations);  
    }  
    public Map<String, SafePoint> getLocations() {  
        return unmodifiableMap;  
    }  
    public SafePoint getLocation(String id) {  
        return locations.get(id);  
    }  
    public void setLocation(String id, int x, int y) {  
        locations.get(id).set(x, y);  
    }  
}
```

Goetz p. 70

SafePublishingVehicleTracker memory



Which VehicleTracker is best?

- All are thread-safe
 - Some due to defensive copying
 - Some due to immutability and unmodifiability
- Different meanings of setLocation:
 - setLocation **does not** affect prior getLocation/s:
 - MonitorVehicleTracker (V1)
 - DelegatingVehicleTracker with getLocationsStatic (V3)
 - setLocation **does** affect prior getLocation/s:
 - DelegatingVehicleTracker (V2)
 - SafePublishingVehicleTracker (V4)
- Performance depends on the usage
 - More setLocation calls than getLocations calls
 - Number of results returned by getLocations

Plan for today

- Java Monitor pattern
- Defensive copying, VehicleTracker
- **Standard collection classes not threadsafe**
- **Extending collection classes**
- ConcurrentModificationException
- FutureTask<T> and asynchronous execution
- (Silly complications of checked exceptions)
- Building a scalable result cache

The classic collection classes are not threadsafe

```
final Collection<Integer> coll = new HashSet<Integer>();  
final int itemCount = 100_000;  
Thread addEven = new Thread(new Runnable() { public void run() {  
    for (int i=0; i<itemCount; i++)  
        coll.add(2 * i);  
}});  
Thread addOdd = new Thread(new Runnable() { public void run() {  
    for (int i=0; i<itemCount; i++)  
        coll.add(2 * i + 1);  
}});
```

TestCollections.java

- May give wrong results or obscure exceptions:

There are 169563 items, should be 200000

"Thread-0" ClassCastException: java.util.HashMap\$Node cannot be cast to java.util.HashMap\$TreeNode

- Wrap as synchronized coll. for thread safety

```
final Collection<Integer> coll  
= Collections.synchronizedCollection(new HashSet<Integer>());
```

Adding `putIfAbsent` to `ArrayList<T>`

```
class FirstBadListHelper<E> {  
    public List<E> list = Collections.synchronizedList(new ArrayList<E>());  
    public boolean putIfAbsent(E x) {  
        boolean absent = !list.contains(x);  
        if (absent)  
            list.add(x);  
        return absent;  
    }  
}
```

Not thread-safe

test, then ...

... act

- Non-atomic test-then-act is not thread-safe
- But this is not thread-safe either. Q: Why?

```
class SecondBadListHelper<E> {  
    public List<E> list = Collections.synchronizedList(new ArrayList<E>());  
    public synchronized boolean putIfAbsent(E x) {  
        boolean absent = !list.contains(x);  
        if (absent)  
            list.add(x);  
        return absent;  
    }  
}
```

Not thread-safe

Goetz p. 72

Client side locking for putIfAbsent

```
class GoodListHelper<E> {  
    public List<E> list = Collections.synchronizedList(new ArrayList<E>());  
  
    public boolean putIfAbsent(E x) {  
        synchronized (list) {  
            boolean absent = !list.contains(x);  
            if (absent)  
                list.add(x);  
            return absent;  
        }  
    }  
}
```

Atomic test-
then-act

Goetz p. 72

- Discuss:
 - Is the test-then-act guaranteed atomic?
 - What could undermine the atomicity?

Using composition is safer – and more work

```
final class BetterArrayList<E> implements List<E> {  
    private List<E> list = new ArrayList<E>();  
  
    public synchronized boolean putIfAbsent(E x) {  
        boolean absent = !list.contains(x);  
        if (absent)  
            list.add(x);  
        return absent;  
    }  
  
    public synchronized boolean add(E item) {  
        return list.add(item);  
    }  
  
    ... approx. 30 other ArrayList<E> methods with synchronized added ...  
}
```

TestListHelper.java

- Q: Are operations now guaranteed atomic?
- Better use `java.util.concurrent.*` collections
 - If you need to make updates concurrently

Plan for today

- Java Monitor pattern
- Defensive copying, VehicleTracker
- Standard collection classes not thread-safe
- Extending collection classes
- **ConcurrentModificationException**
- FutureTask<T> and asynchronous execution
- (Silly complications of checked exceptions)
- Building a scalable result cache

ConcurrentModificationException

```
ArrayList<String> universities = new ArrayList<String>();  
universities.add("Copenhagen University");  
universities.add("KVL");  
universities.add("Aarhus University");  
for (String name : universities) {  
    System.out.println(name);  
    if (name.equals("KVL"))  
        universities.remove(name);  
}
```

Should not change the collection while iterating

Even when no thread concurrency

Copenhagen University
KVL

Exception ... java.util.ConcurrentModificationException

- The “fail-early” mechanism is not thread-safe!
- Do not rely on it in a concurrent context
 - ... instead ...

Java 8 documentation on iteration

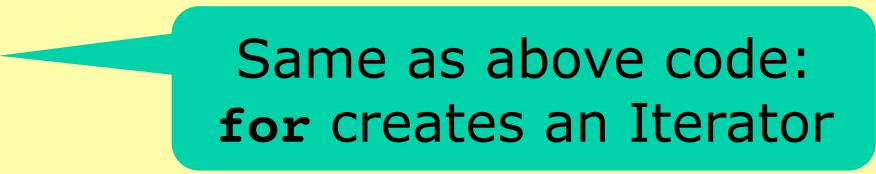
- `Collections.synchronizedList()` says:

It is imperative that the user manually synchronize on the returned collection when traversing it via `Iterator`, `Spliterator` or `Stream`:

```
Collection c = Collections.synchronizedCollection(myCollection);
...
synchronized (c) {
    Iterator i = c.iterator(); // Must be in the synchronized block
    while (i.hasNext())
        foo(i.next());
}
```

Failure to follow this advice may result in non-deterministic behavior.

```
Collection c = Collections.synchronizedCollection(myCollection);
synchronized (c) {
    for (T item : c)
        foo(item);
}
```



Same as above code:
`for` creates an `Iterator`

- All access to `myCollection` must be through `c`

Collections in a concurrent context

- Preferably use a modern concurrent collection in `java.util.concurrent.*`
 - Iterators and `for` are *weakly consistent*:
 - 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.
- Or else wrap collection as synchronized
- Or synchronize accesses yourself
- Or make a thread-local copy of the collection and iterate over that

Plan for today

- Java Monitor pattern
- Defensive copying, VehicleTracker
- Standard collection classes not thread-safe
- Extending collection classes
- ConcurrentModificationException
- **FutureTask<T>, asynchronous execution**
- (Silly complications of checked exceptions)
- Building a scalable result cache

Callable<T> versus Runnable

- A Runnable contains a **void** method:

```
public interface Runnable {  
    public void run();  
}
```

unit -> unit

- A `java.util.concurrent.Callable<T>` returns a `T`:

```
public interface Callable<T> {  
    public T call() throws Exception;  
}
```

unit -> T

```
Callable<String> getWiki = new Callable<String>() {  
    public String call() throws Exception {  
        return getContents("http://www.wikipedia.org/", 10);  
    }  
    // Call the Callable, block till it returns:  
    try { String homepage = getWiki.call(); ... }  
    catch (Exception exn) { throw new RuntimeException(exn); }
```

TestCallable.java

Synchronous FutureTask<T>

```
Callable<String> getWiki = new Callable<String>() {  
    public String call() throws Exception {  
        return getContents("http://www.wikipedia.org/", 10);  
    } };  
FutureTask<String> fut = new FutureTask<String>(getWiki);  
fut.run(); Run call() on "main" thread  
try {  
    String homepage = fut.get(); Get result of call()  
    System.out.println(homepage);  
}  
catch (Exception exn) { throw new RuntimeException(exn); }
```

- A FutureTask<T>

- Produces a T
- Is created from a Callable<T>
- Above we run it synchronously on the main thread
- More useful to run asynchronously on other thread
- Possible because it implements Runnable

Similar to .NET
System.Threading.Tasks.Task<T>

Asynchronous FutureTask<T>

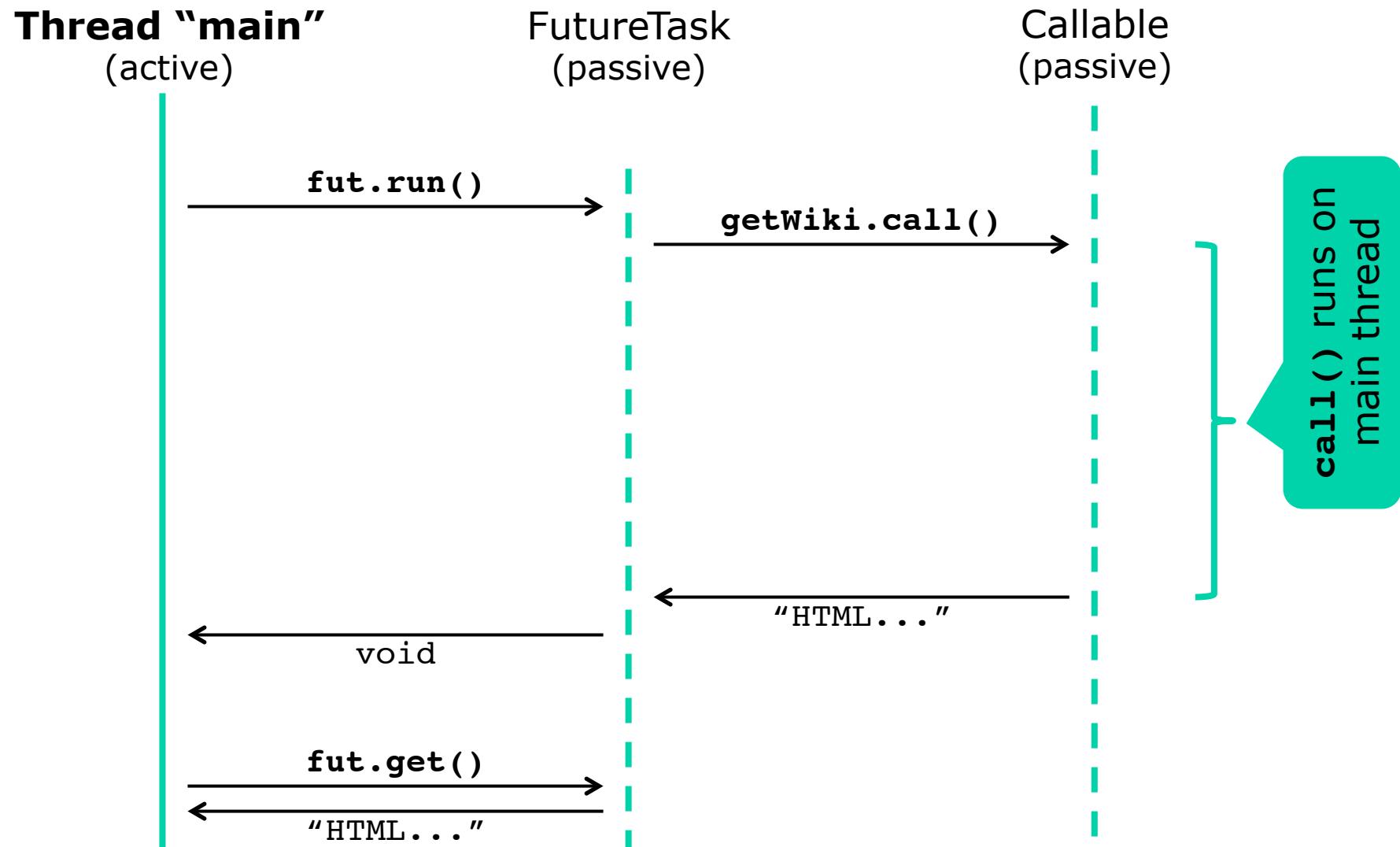
```
Callable<String> getWiki = new Callable<String>() {  
    public String call() throws Exception {  
        return getContents("http://www.wikipedia.org/", 10);  
    } };  
FutureTask<String> fut = new FutureTask<String>(getWiki);  
Thread t = new Thread(fut);  
t.start();  
try {  
    String homepage = fut.get();  
    System.out.println(homepage);  
}  
catch (Exception exn) { throw new RuntimeException(exn); }
```

Create and start
thread running **call()**

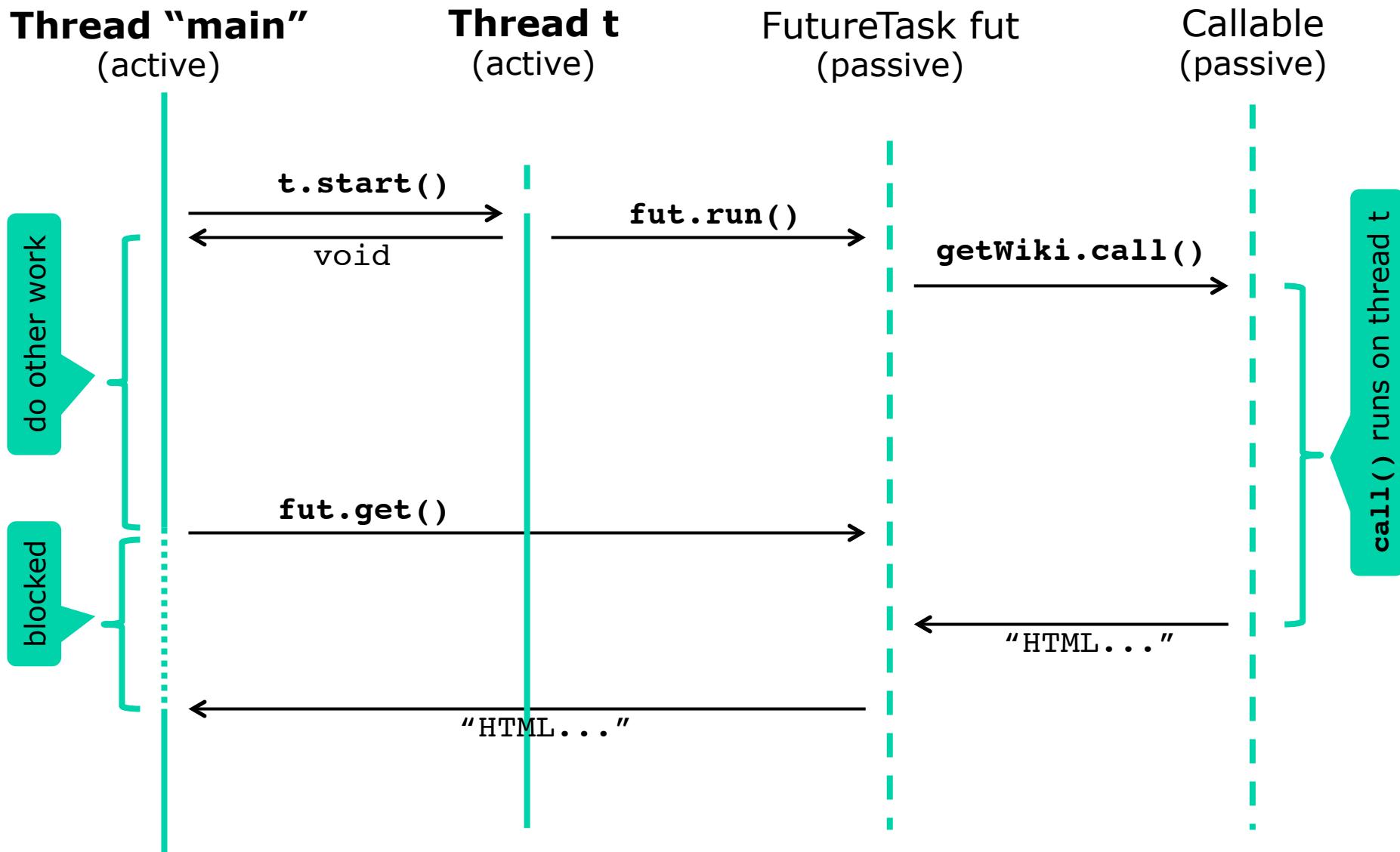
Block until **call()**
completes

- The “main” thread can do other work between **t.start()** and **fut.get()**
- FutureTask can also be run as a *task*, week 5

Synchronous FutureTask



Asynchronous FutureTask



Those @\$%&!!! checked exceptions

- Our exception handling is simple but gross:

If `call()` throws `exn`, then `get()` throws `ExecutionException(exn)`

... and then we further wrap a `RuntimeException(...)` around that

```
try { String homepage = fut.get(); ... }
catch (Exception exn) { throw new RuntimeException(exn); }
```

- Goetz has a better, more complex, approach:

```
try { String homepage = fut.get(); ... }
catch (ExecutionException exn) {
    Throwable cause = exn.getCause();
    if (cause instanceof IOException)
        throw (IOException)cause;
    else
        throw launderThrowable(cause);
}
```

Rethrow “expected”
`call()` exceptions

Turn others into
unchecked exceptions

Like Goetz p. 97

Goetz's launderThrowable method

unchecked

checked

```
public static RuntimeException launderThrowable(Throwable t) {  
    if (t instanceof RuntimeException)  
        return (RuntimeException) t;  
    else if (t instanceof Error)  
        throw (Error) t;  
    else  
        throw new IllegalStateException("Not unchecked", t);  
}
```

Goetz p. 98

- Make a checked exception into an unchecked
 - without adding unreasonable layers of wrapping
 - cannot just **throw cause;** in previous slide's code
- Mostly an administrative mess
 - caused by the Java's “checked exceptions” design
 - thus not a problem in C#/.NET

Plan for today

- Java Monitor pattern
- Defensive copying, VehicleTracker
- Standard collection classes not threadsafe
- Extending collection classes
- ConcurrentModificationException
- FutureTask<T>, asynchronous execution
- (Silly complications of checked exceptions)
- **Building a scalable result cache**

Goetz's “scalable result cache”

- Interface representing functions from A to V

```
interface Computable <A, V> {  
    V compute(A arg) throws InterruptedException;  
}
```

A → V

Goetz p. 103

- Example 1: Our prime factorizer

```
class Factorizer implements Computable<Long, long[]> {  
    public long[] compute(Long wrappedP) {  
        long p = wrappedP;  
        ...  
    } }
```

TestCache.java

- Example 2: Fetching a web page

```
class FetchWebpage implements Computable<String, String> {  
    public String compute(String url) {  
        ... create Http connection, fetch webpage ...  
    } }
```

Thread-safe but non-scalable cache

```

class Memoizer1 <A, V> implements Computable<A, V> {
    private final Map<A, V> cache = new HashMap<A, V>();
    private final Computable<A, V> c;

    public Memoizer1(Computable<A, V> c) { this.c = c; }

    public synchronized V compute(A arg) throws InterruptedException {
        V result = cache.get(arg);
        if (result == null) {
            result = c.compute(arg);
            cache.put(arg, result);
        }
        return result;
    }
}

```

If not in cache,
compute and put

Goetz p. 103

```

Computable<Long, long[]> factorizer = new Factorizer(),
    cachingFactorizer = new Memoizer1<Long, long[]>(factorizer);
long[] factors = cachingFactorizer.compute(7182763656381322L);

```

- Q: Why not scalable?

Thread-safe scalable cache, using concurrent hashmap

```
class Memoizer2 <A, V> implements Computable<A, V> {  
    private final Map<A, V> cache = new ConcurrentHashMap<A, V>();  
    private final Computable<A, V> c;  
  
    public Memoizer2(Computable<A, V> c) { this.c = c; }  
  
    public V compute(A arg) throws InterruptedException {  
        V result = cache.get(arg);  
        if (result == null) {  
            result = c.compute(arg);  
            cache.put(arg, result);  
        }  
        return result;  
    }  
}
```

Goetz p. 105

- But large risk of computing same thing twice
 - Argument put in cache only after computing result
 - so cache may be updated long after **compute(arg)** call

How Memoizer2 can duplicate work

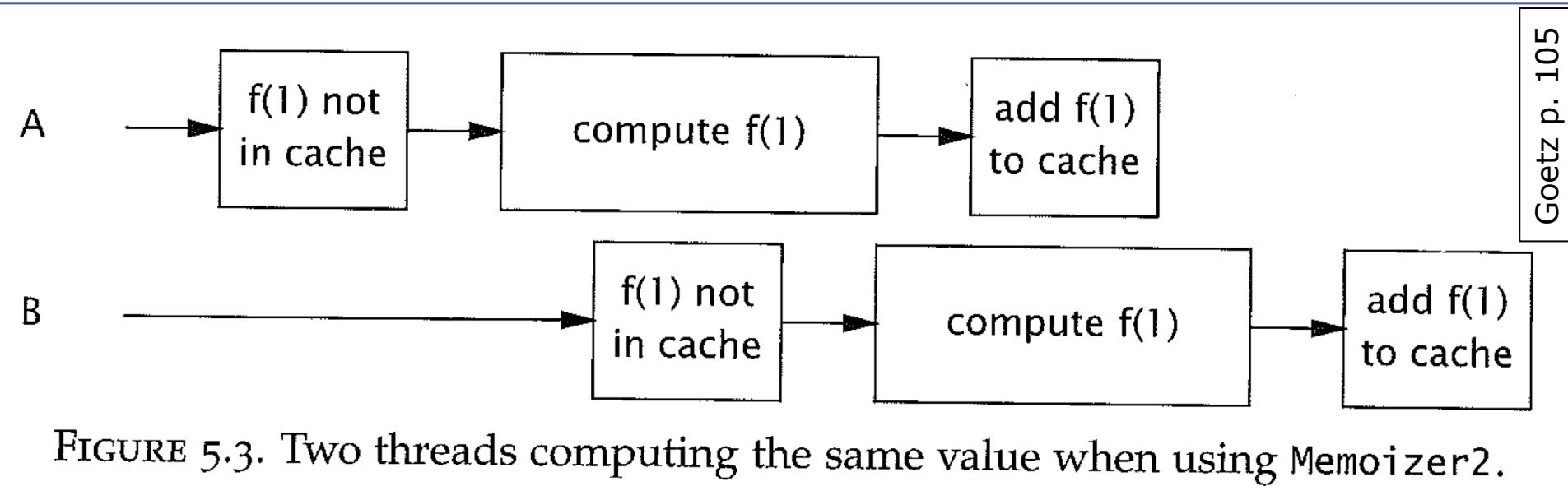


FIGURE 5.3. Two threads computing the same value when using Memoizer2.

- Better approach, Memoizer3:
 - Create a FutureTask for `arg`
 - Add the FutureTask to cache immediately at `arg`
 - Run the future on the calling thread
 - Return `fut.get()`

Thread-safe scalable cache using FutureTask<V>, v. 3

```

class Memoizer3<A, V> implements Computable<A, V> {
    private final Map<A, Future<V>> cache
        = new ConcurrentHashMap<A, Future<V>>();
    private final Computable<A, V> c;

    public V compute(final A arg) throws InterruptedException {
        Future<V> f = cache.get(arg);
        if (f == null) { If arg not in cache ...
            Callable<V> eval = new Callable<V>() {
                public V call() throws InterruptedException
                    return c.compute(arg);
            };
            FutureTask<V> ft = new FutureTask<V>(eval);
            cache.put(arg, ft);
            f = ft;
            ft.run(); ... run it on calling thread
        }
        try { return f.get(); } Block until completed
        catch (ExecutionException e) { throw launderThrowable(...); }
    }
}

```

Goetz p. 106

Memoizer3 can still duplicate work

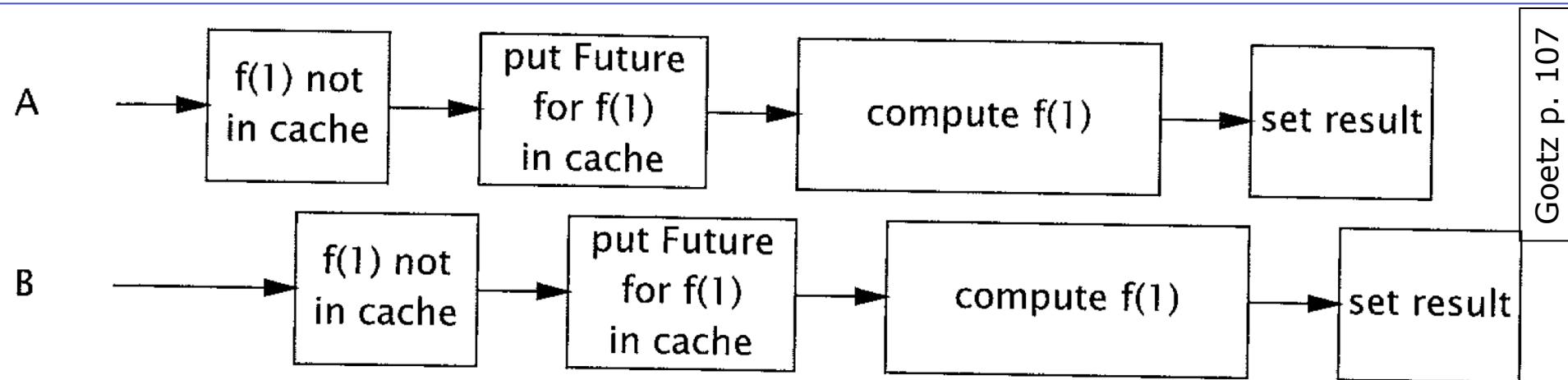


FIGURE 5.4. Unlucky timing that could cause Memoizer3 to calculate the same value twice.

- Better approach, Memoizer4:
 - Fast initial check for `arg` cache
 - If not, create a future for the computation
 - Atomic put-if-absent may add future to cache
 - Run the future on the calling thread
 - Return `fut.get()`

Thread-safe scalable cache using FutureTask<V>, v. 4

```

class Memoizer4<A, V> implements Computable<A, V> {
    private final Map<A, Future<V>> cache
        = new ConcurrentHashMap<A, Future<V>>();
    private final Computable<A, V> c;
    public V compute(final A arg) throws InterruptedException {
        Future<V> f = cache.get(arg);
        if (f == null) { Fast test: If arg not in cache ...
            Callable<V> eval = new Callable<V>() {
                public V call() throws InterruptedException {
                    return c.compute(arg);
                }
            };
            FutureTask<V> ft = new FutureTask<V>(eval);
            f = cache.putIfAbsent(arg, ft);
            if (f == null) { ... atomic put-if-absent
                f = ft; ft.run();
            }
        }
        try { return f.get(); }
        catch (ExecutionException e) { throw launderThrowable(...); }
    }
}

```

... make future

... atomic put-if-absent

... run on calling thread if not added to cache before

The technique used in Memoizer4

- (Before Java 8) one cannot atomically test-then-create-future-and-add-it
- Hence, suggested by Bloch item 69:
 - Make a fast (non-atomic) test for arg in cache
 - If not there, create future object
 - Then atomically put-if-absent (arg, future)
 - If the arg was added in the meantime, do not add
 - Otherwise, add (arg, future) and run the future
- May wastefully create a future, but only rarely
 - The garbage collector will remove it
- Java 8 has computeIfAbsent, can avoid the two-stage test, but looks complicated

Thread-safe scalable cache using FutureTask<V>, v. 5 (Java 8)

```

class Memoizer5<A, V> implements Computable<A, V> {
    private final Map<A, Future<V>> cache
        = new ConcurrentHashMap<A, Future<V>>();
    private final Computable<A, V> c;
    public V compute(final A arg) throws InterruptedException {
        final AtomicReference<FutureTask<V>> ftr = new ...();
        Future<V> f = cache.computeIfAbsent(arg, new Function<...>() {
            public Future<V> apply(final A arg) {
                Callable<V> eval = new Callable<V>() {
                    public V call() throws InterruptedException {
                        return c.compute(arg);
                    }
                };
                ftr.set(new FutureTask<V>(eval));
                return ftr.get();
            }
        });
        if (ftr.get() != null) {
            ftr.get().run();
        }
        try { return f.get(); }
        catch (ExecutionException e) { throw launderThrowable(...); }
    }
}

```

TestCache.java

make future

... run on calling thread if not already in cache

Parts of Java are 20 years old, have some design mistakes

- Never use these Thread methods (API):
 - **destroy()**
 - **countStackFrames()**
 - **resume()**
 - **stop()**
 - **suspend()**
- Avoid thread groups (Bloch item 73)
- Prefer non-synchronized
 - `StringBuilder` over `StringBuffer`
 - `ArrayList` or `CopyOnWriteArrayList` over `Vector`
 - `HashMap` or `ConcurrentHashMap` over `HashTable`

This week

- Reading
 - Goetz et al chapters 4 and 5
- Exercises
 - Build a threadsafe class, use built-in collection classes, use the future concept
- Read before next week's lecture
 - Sestoft: Microbenchmarks in Java and C#
 - Optional: McKenney chapter 3

Practical Concurrent and Parallel Programming 4

Peter Sestoft
IT University of Copenhagen

Friday 2014-09-19

Plan for today

- Performance measurements
- Back-of-the envelope estimates
- A class for measuring elapsed wall-clock time
- Mark0-5: Towards reliable measurements
- Mark6-7: Automated general measurements
- Cost of thread creation, start, exec, volatile ...
- Measuring the prime counting example
- Scatter charts, or x-y plots
- General advice, warnings and pitfalls

How long does this method take?

```
private static double multiply(int i) {  
    double x = 1.1 * (double) (i & 0xFF);  
    return x * x * x * x * x * x * x * x * x * x *  
          x * x * x * x * x * x * x * x * x * x * x;  
}
```

- There are 20 floating-point multiplications
- An integer op. and an int-double conversion
- Takes at least $20 * 0.4 = 8$ ns
- Tricks used in this code:
 - Make result depend on **i** to avoid caching (C only)
 - The **i & 0xFF** is in range 0–255 to avoid overflow
 - Multiply **i & 0xFF** by 1.1 to make it floating-point

Back-of-the envelope calculations

- 2.4 GHz processor = 0.4 ns/cycle = 0.4×10^{-9} s
- Throughput:
 - Addition or multiplication takes 1 cycle
 - Division maybe 15 cycles
 - Transcendental functions, `sin(x)` maybe 100-200?
- Instruction-level parallelism
 - 2-3 integer operations/cycle, not always possible
- Memory latency
 - Registers: 1 cycle
 - L1 cache: a few cycles
 - L2 cache: many cycles
 - RAM: hundreds of cycles – expensive cache misses!

A Timer class for Java

- We measure elapsed wall-clock time
 - This is what matters in reality
 - Measured uniformly on Linux, MacOS, Windows
 - Enables comparison Java/C#/C/Scala/F# etc

```
public class Timer {  
    private long start, spent = 0;  
    public Timer() { play(); }  
    public double check()  
    { return (System.nanoTime()-start+spent)/1e9; }  
    public void pause() { spent += System.nanoTime()-start; }  
    public void play() { start = System.nanoTime(); }  
}
```

Benchmark.java

- Alternatives: total CPU time, or user + kernel
- Never use imprecise, slow `new Date().getTime()`
- Q: Reasons to measure total CPU time?

Mark0: naïve attempt

```
public static void Mark0() {           Useless
    Timer t = new Timer();
    double dummy = multiply(10);
    double time = t.check() * 1e9;
    System.out.printf("%6.1f ns\n", time);
}
```

- Useless because
 - Timer resolution too poor, likely 100 ns
 - Runtime start-up costs larger than execution time
 - So result are unrealistic and vary a lot

5000.0 ns
6000.0 ns
4500.0 ns

Mark1: Measure many operations

```
public static void Mark1() { Quite useless
    Timer t = new Timer();
    Integer count = 1_000_000;
    for (int i=0; i<count; i++) {
        double dummy = multiply(i);
    }
    double time = t.check() * 1e9 / count;
    System.out.printf("%6.1f ns%n", time);
}
```

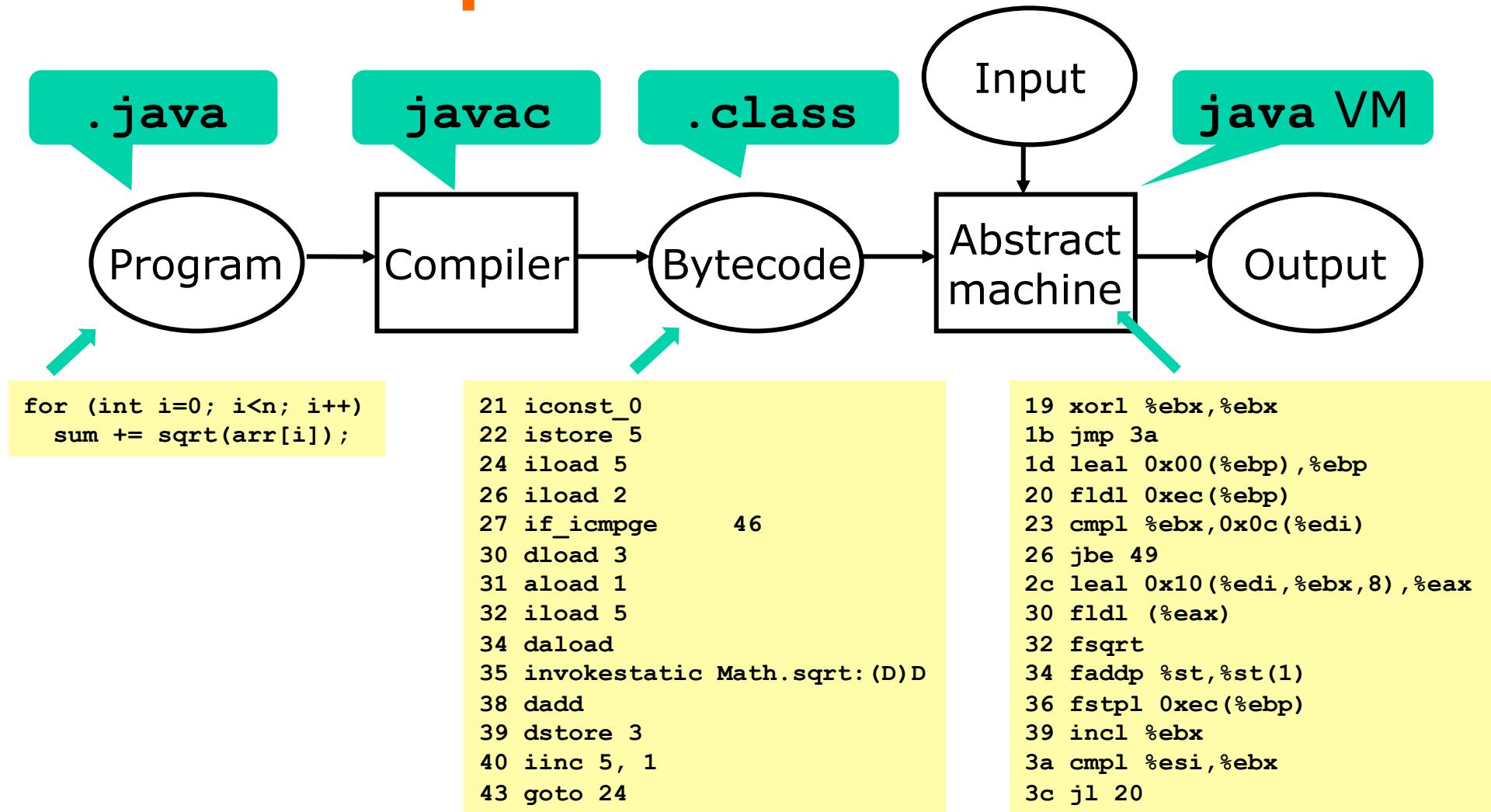
- Measure 1 million calls; better but fragile:

- If **count** is larger, optimizer may notice that result of **multiply** is not used, and remove call
 - So-called “dead code elimination”
 - May give completely unrealistic results

5.0 ns
5.5 ns
5.0 ns

0.1 ns
0.1 ns
0.0 ns

Java compiler and virtual machine



- The **javac** compiler is simple, makes no optimizations
- The **java** runtime system (JIT) is clever, makes many

Mark2: Avoid dead code elimination

```
public static double Mark2() {  
    Timer t = new Timer();  
    int count = 100_000_000;  
    double dummy = 0.0;  
    for (int i=0; i<count; i++)  
        dummy += multiply(i);  
    double time = t.check() * 1e9 / count;  
    System.out.printf("%6.1f ns\n", time);  
    return dummy;  
}
```

30.5 ns
30.4 ns
30.3 ns

- Much more reliable

Mark3: Automate multiple samples

```
int n = 10;  
int count = 100_000_000;  
double dummy = 0.0;  
for (int j=0; j<n; j++) {  
    Timer t = new Timer();  
    for (int i=0; i<count; i++)  
        dummy += multiply(i);  
    double time = t.check() * 1e9 / count;  
    System.out.printf("%6.1f ns%n", time);  
}
```

Number of samples

Iterations per sample

30.7 ns
30.3 ns
30.1 ns
30.7 ns
30.5 ns
30.4 ns
30.9 ns
30.3 ns
30.5 ns
30.8 ns

- Multiple samples gives an impression of reproducibility

Mark4: Compute standard deviation

```
int count = 100_000_000;
double st = 0.0, sst = 0.0;
for (int j=0; j<n; j++) {
    Timer t = new Timer();
    for (int i=0; i<count; i++)
        dummy += multiply(i);
    double time = t.check() * 1e9 / count;
    st += time;
    sst += time * time;
}
double mean = st/n, sdev = Math.sqrt(sst/n - mean*mean);
System.out.printf("%6.1f ns +/- %6.3f %n", mean, sdev);
```

Is this a reasonable iteration count?

- The standard deviation σ summarizes the variation around the mean, in a single number

30.3 ns +/- 0.137

Statistics: Central limit theorem

- The average of n independent identically distributed observations t_1, t_2, \dots, t_n tends to follow the normal distribution $N(\mu, \sigma^2)$ where

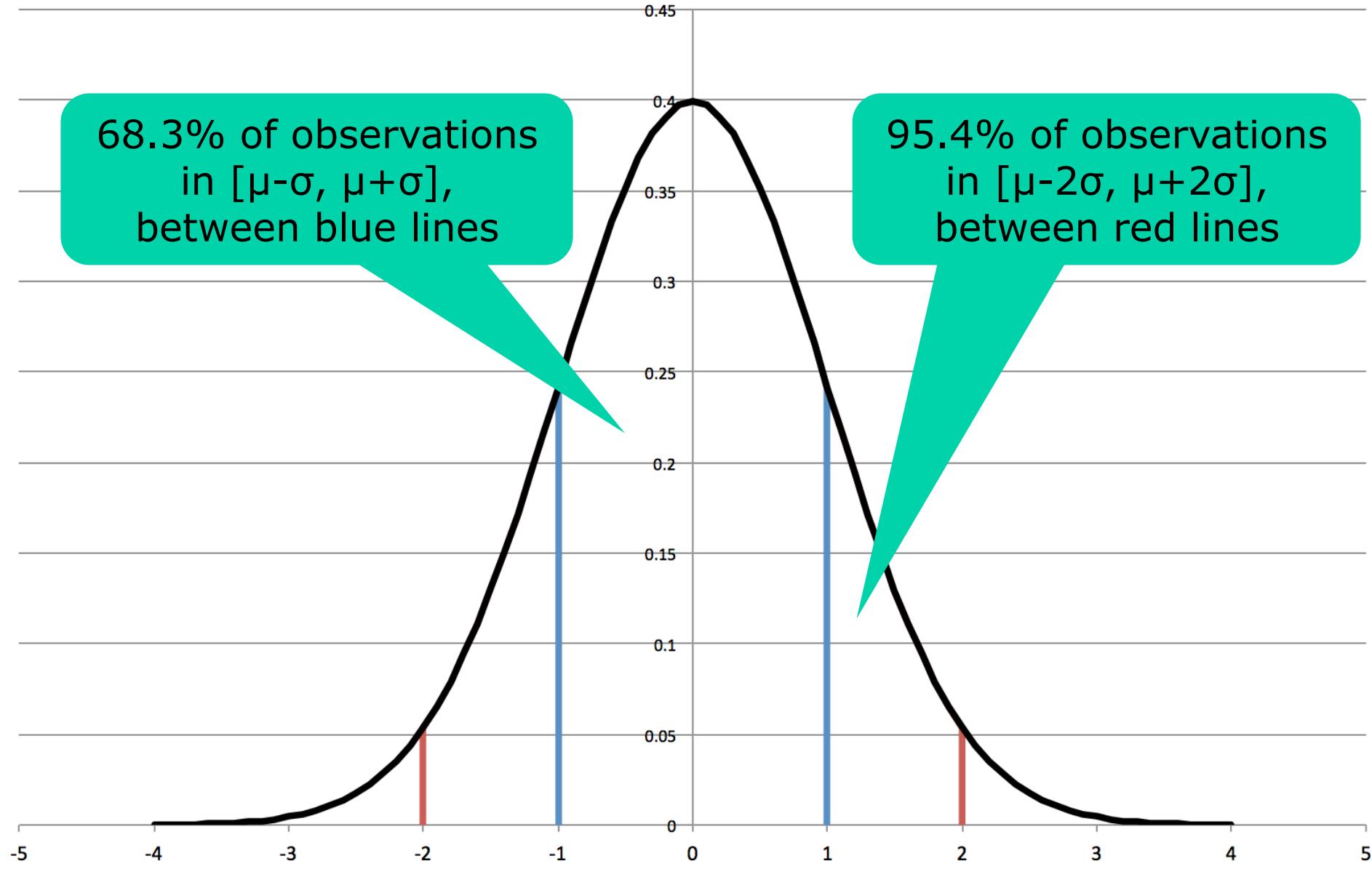
$$\mu = \frac{1}{n} \sum_{j=1}^n t_j$$

$$\sigma = \sqrt{\frac{1}{n} \sum_{j=1}^n t_j^2 - \mu^2}$$

when n tends to infinity

- Eg with probability 68.3% the “real” result is between 30.163 ns and 30.437 ns

The normal distribution $N(\mu, \sigma^2)$



Mark5: Auto-choose iteration count

```
int n = 10, count = 1, totalCount = 0;
double dummy = 0.0, runningTime = 0.0;
do {
    count *= 2; Double count until ...
    double st = 0.0, sst = 0.0;
    for (int j=0; j<n; j++) {
        Timer t = new Timer();
        for (int i=0; i<count; i++)
            dummy += multiply(i);
        runningTime = t.check();
        double time = runningTime * 1e9 / count;
        st += time;
        sst += time * time;
        totalCount += count;
    }
    double mean = st/n, sdev = Math.sqrt(sst/n - mean*mean);
    System.out.printf("%6.1f ns +/- %8.2f %10d%n", mean, ...);
} while (runningTime < 0.25 && count < Integer.MAX_VALUE/2);
return dummy / totalCount; ... loop runs at least 0.25 sec
```

Example results from Mark5

mean time	sdev	count
100.0 ns +/-	200.00	2
100.0 ns +/-	122.47	4
62.5 ns +/-	62.50	8
50.0 ns +/-	37.50	16
46.9 ns +/-	15.63	32
40.6 ns +/-	10.36	64
39.8 ns +/-	2.34	128
36.3 ns +/-	1.79	256
36.5 ns +/-	1.25	512
35.6 ns +/-	0.49	1024
111.1 ns +/-	232.18	2048
36.1 ns +/-	1.75	4096
33.7 ns +/-	0.84	8192
32.5 ns +/-	1.07	16384
35.6 ns +/-	4.84	32768
30.4 ns +/-	0.26	65536
33.1 ns +/-	5.06	131072
30.3 ns +/-	0.49	262144
...

Outlier, maybe due to
other program activity

Advantages of Mark5

- The early rounds (2, 4, ...) serve as warm-up
 - Make sure the code is in memory and cache
- Measured code loop runs at least 0.25 sec
 - Roughly 500 million CPU cycles
 - Lessen impact of other activity on computer
 - Makes sure code has been JIT compiled
- Still, total time spent measuring at most 1 sec
 - Because last measurement runs at most 0.5 sec
 - and sum of previous times is same time as last one
 - because $2 + 4 + 8 + \dots + 2^n < 2^{n+1}$
- Independent of problem and hardware

Development of the benchmarking method

- Mark0: Measure one call, useless
- Mark1: Measure many calls, nearly useless
- Mark2: Avoid dead code elimination
- Mark3: Automate multiple samples
- Mark4: Compute standard deviation
- Mark5: Automate choice of iteration count
- But need to measure not just **multiply!**

Mark6: Generalize to any function

```
public interface IntToDouble {  
    double call(int i);  
}
```

Interface describes
multiply-like methods

```
public static double Mark6(String msg, IntToDouble f) {  
    ...  
    do {  
        ...  
        for (int j=0; j<n; j++) {  
            ...  
            for (int i=0; i<count; i++)  
                dummy += f.call(i);  
        ...  
    }  
    ...  
    System.out.printf("%-25s %15.1f ns %10.2f %10d%n", ...);  
    } while (runningTime<0.25 && count<Integer.MAX_VALUE/2);  
    return dummy / totalCount;  
}
```

Call given method f

Example use of Mark6

Function to be measured is given as anonymous inner class

```
Mark6("multiply", new Int.ToDouble() {  
    public double call(int i) { return multiply(i); } });
```

multiply	800.0 ns	1435.27	2
multiply	250.0 ns	0.00	4
multiply	212.5 ns	80.04	8
multiply	187.5 ns	39.53	16
multiply	200.0 ns	82.92	32
multiply	57.8 ns	24.26	64
multiply	46.9 ns	4.94	128
...			
multiply	30.6 ns	0.61	2097152
multiply	30.0 ns	0.10	4194304
multiply	30.1 ns	0.15	8388608

Mark7: Print only last measurement

```
public static double Mark7(String msg, IntToDouble f) {  
    ...  
    do {  
        ...  
        } while (runningTime<0.25 && count<Integer.MAX_VALUE/2);  
    double mean = st/n, sdev = Math.sqrt(sst/n - mean*mean);  
    System.out.printf("%-25s %15.1f ns %10.2f %10d%n", ...);  
    return dummy / totalCount;  
}
```

Printing moved from here to outside loop

```
Mark7("pow", new IntToDouble() {  
    public double call(int i) { return Math.pow(10.0, 0.1 * (i & 0xFF)); } } );  
Mark7("exp", new IntToDouble() {  
    public double call(int i) { return Math.exp(0.1 * (i & 0xFF)); } } );  
Mark7("log", new IntToDouble() {  
    public double call(int i) { return Math.log(0.1 + 0.1 * (i & 0xFF)); } } );  
Mark7("sin", new IntToDouble() {  
    public double call(int i) { return Math.sin(0.1 * (i & 0xFF)); } } );  
Mark7("cos", new IntToDouble() {  
    public double call(int i) { return Math.cos(0.1 * (i & 0xFF)); } } );
```

Mark 7 benchmarking results for Java mathematical functions

pow	75.5 ns	0.43	4194304
exp	54.9 ns	0.19	8388608
log	31.4 ns	0.16	8388608
sin	116.3 ns	0.41	4194304
cos	116.6 ns	0.33	4194304
tan	143.6 ns	0.48	2097152
asin	229.7 ns	2.24	2097152
acos	217.0 ns	2.46	2097152
atan	54.3 ns	0.84	8388608

- 2.4 GHz Intel i7; MacOS 10.9.4; 64-bit JVM 1.8.0_11
- So **sin(x)** takes $116.3 \text{ ns} \times 2.4 \text{ GHz} = 279$ cycles
 - approximately

Saving measurements to a text file

- Command line in Linux, MacOS, Windows

```
java Benchmark > benchmark-20140831.txt
```

- In Linux, MacOS get both file and console

```
java Benchmark | tee benchmark-20140831.txt
```

Platform identification

```
public static void SystemInfo() {  
    System.out.printf("# OS: %s; %s; %s%n",  
                      System.getProperty("os.name"),  
                      System.getProperty("os.version"),  
                      System.getProperty("os.arch"));  
    System.out.printf("# JVM: %s; %s%n",  
                      System.getProperty("java.vendor"),  
                      System.getProperty("java.version"));  
    // This line works only on MS Windows:  
    System.out.printf("# CPU: %s%n", System.getenv("PROCESSOR_IDENTIFIER"))  
    java.util.Date now = new java.util.Date();  
    System.out.printf("# Date: %s%n",  
                      new java.text.SimpleDateFormat("yyyy-MM-dd'T'HH:mm:ssZ").format(now));  
}
```

- Output information about date and platform:

```
# OS: Mac OS X; 10.9.4; x86_64  
# JVM: Oracle Corporation; 1.8.0_11  
# CPU: null  
# Date: 2014-08-31T14:46:56+0200
```

31 August 2014 at
14:46 in UTC+2h

Measuring task creation, start-up etc

- First: how long to create an ordinary object?

```
class Point {  
    public final int x, y;  
    public Point(int x, int y) { this.x = x; this.y = y; }  
}
```

```
Mark6("Point creation",  
    new IntToDouble() { public double call(int i) {  
        Point p = new Point(i, i);  
        return p.hashCode();  
    } } );
```

TestTimeThreads.java

- Result on i7, approximately 80 ns
- Q: Why return **p.hashCode()**?
If not it would maybe only allocate the memory and not compute anything
- Computing the hash code takes 3.3 ns
 - Q: How can I know that?

Measure it by testing

Cost of thread create

```
final AtomicInteger ai = new AtomicInteger();
Mark6("Thread create",
    new IntToDouble() { public double call(int i) {
        Thread t = new Thread(new Runnable() { public void run() {
            for (int j=0; j<1000; j++)
                ai.getAndIncrement();
        } });
        return t.hashCode();
    } });
}

```

TestTimeThreads.java

Actual work,
not run

- Takes 1030 ns, or 13 x slower than a Point
 - So a Thread object must be somewhat complicated

Cost of thread create + start

```
Mark6("Thread create start",
  new InttoDouble() { public double call(int i) {
    Thread t = new Thread(new Runnable() { public void run() {
      for (int j=0; j<1000; j++)
        ai.getAndIncrement();
    } });
    t.start();
    return t.hashCode();
  } });
}
```

TestTimeThreads.java

Actual work,
not run

- Takes 49000 ns
- So a lot of work goes into setting up a task
 - Even after creating it
- Note: does **not** include executing the loop

Cost of thread create+start+run+join

```
Mark6("Thread create start join",
  new IntToDouble() { public double call(int i) {
    Thread t = new Thread(new Runnable() { public void run() {
      for (int j=0; j<1000; j++)
        ai.getAndIncrement();
    } });
    t.start();
    try { t.join(); }
    catch (InterruptedException exn) { }
    return t.hashCode();
  } });
}
```

TestTimeThreads.java

Actual work
is done

because of join()

- Takes 72700 ns
- Of this, the actual work is 6580 ns, in loop
- Thus 1080 ns to create; 48000 ns to start;
13000 ns run and join; 6580 ns actual work
- Never create threads for small computations

Cost of taking a free lock

```
final Object obj = new Object();  
Mark6("Uncontented lock",  
    new IntToDouble() { public double call(int i) {  
        synchronized (obj) {  
            return i;  
        }  
    }});
```

Succeeds immediately
because only one
thread is running

TestTimeThreads.java

- Takes 4.1 ns although sometime 20 ns instead
- Both are very fast
 - The result of much engineering on the Java VM
 - Taking a free lock was much slower in early Java
 - Today no need to use “double-checked-locking”, Goetz antipattern p. 349
- Q: Possible to measure time to take a lock already held by another thread?

Wouldn't make sense as you can't control the time it takes for them to release it

Cost of volatile

```
class IntArrayVolatile {  
    private volatile int[] array;  
    public IntArray(int length) { array = new int[length]; ... }  
    public boolean isSorted() {  
        for (int i=1; i<array.length; i++)  
            if (array[i-1] > array[i])  
                return false;  
        return true;  
    }  
}
```

TestVolatileCost.java

IntArray	3.4 us	0.01	131072
IntArrayVolatile	17.2 us	0.14	16384

- Volatile read is 5 x slower in this case
- Because
 - Memory barriers at runtime for cache consistency
 - JIT compiler cannot perform certain optimizations

Full measurements on two platforms

hashCode()	3.3 ns	0.02	134217728
Point creation	80.9 ns	1.06	4194304
Thread's work	6581.5 ns	37.64	65536
Thread create	1030.3 ns	20.17	262144
Thread create start	48929.6 ns	320.94	8192
Thread create start join	72758.9 ns	1204.68	4096
Uncontented lock	4.1 ns	0.06	67108864

Intel i7, 2.4 GHz, 4 core
45 W, Sep 2012, \$378

hashCode()	15.5 ns	0.01	16777216
Point creation	184.1 ns	0.43	2097152
Thread's work	30802.5 ns	18.65	8192
Thread create	3690.2 ns	7.99	131072
Thread create start	153097.2 ns	11142.30	2048
Thread create start join	165992.8 ns	3916.62	2048
Uncontented lock	16.9 ns	0.01	16777216

AMD 6386 SE, 2.8 GHz, 16 core
140 W, Nov 2012, \$1392

Measuring TestCountPrimes

```
final int range = 100_000;
Mark6("countSequential", new IntToDouble() {
    public double call(int i) {
        return countSequential(range);
    }
});
Mark6("countParallel", new IntToDouble() {
    public double call(int i) {
        return countParallelN(range, 10);
    }
});
```

TestCountPrimesThreads.java

- Include Mark6 and Mark7 in source file
 - Modified to show microseconds not nanoseconds
- Reduce range to 100,000
- Threads must be join()'ed to measure time
 - Else you just measure the time to create and start, not the time to actually compute

TestCountPrimes results, 10 threads

countSequential	11117.3 us	501.25	2
countSequential	10969.3 us	82.93	4
countSequential	10935.4 us	52.34	8
countSequential	10936.0 us	32.76	16
countSequential	10970.5 us	142.69	32
countParallel	3944.9 us	764.30	2
countParallel	3397.5 us	166.58	4
countParallel	3218.1 us	59.62	8
countParallel	3224.4 us	62.28	16
countParallel	3261.4 us	65.42	32
countParallel	3379.1 us	224.53	64
countParallel	3239.2 us	111.56	128

- So 10 threads is $10970/3239 = 3.4 \times$ faster
- What about 1 thread, 2, ..., 32 threads?

Measuring different thread counts

```
final int range = 100_000;
Mark7("countSequential", new IntToDouble() {
    public double call(int i) {
        return countSequential(range);
    } });
for (int c=1; c<=32; c++) {
    final int threadCount = c;
    Mark7(String.format("countParallel %6d", threadCount),
        new IntToDouble() {
            public double call(int i) {
                return countParallelN(range, threadCount);
            } });
}
```

TestCountPrimesThreads.java

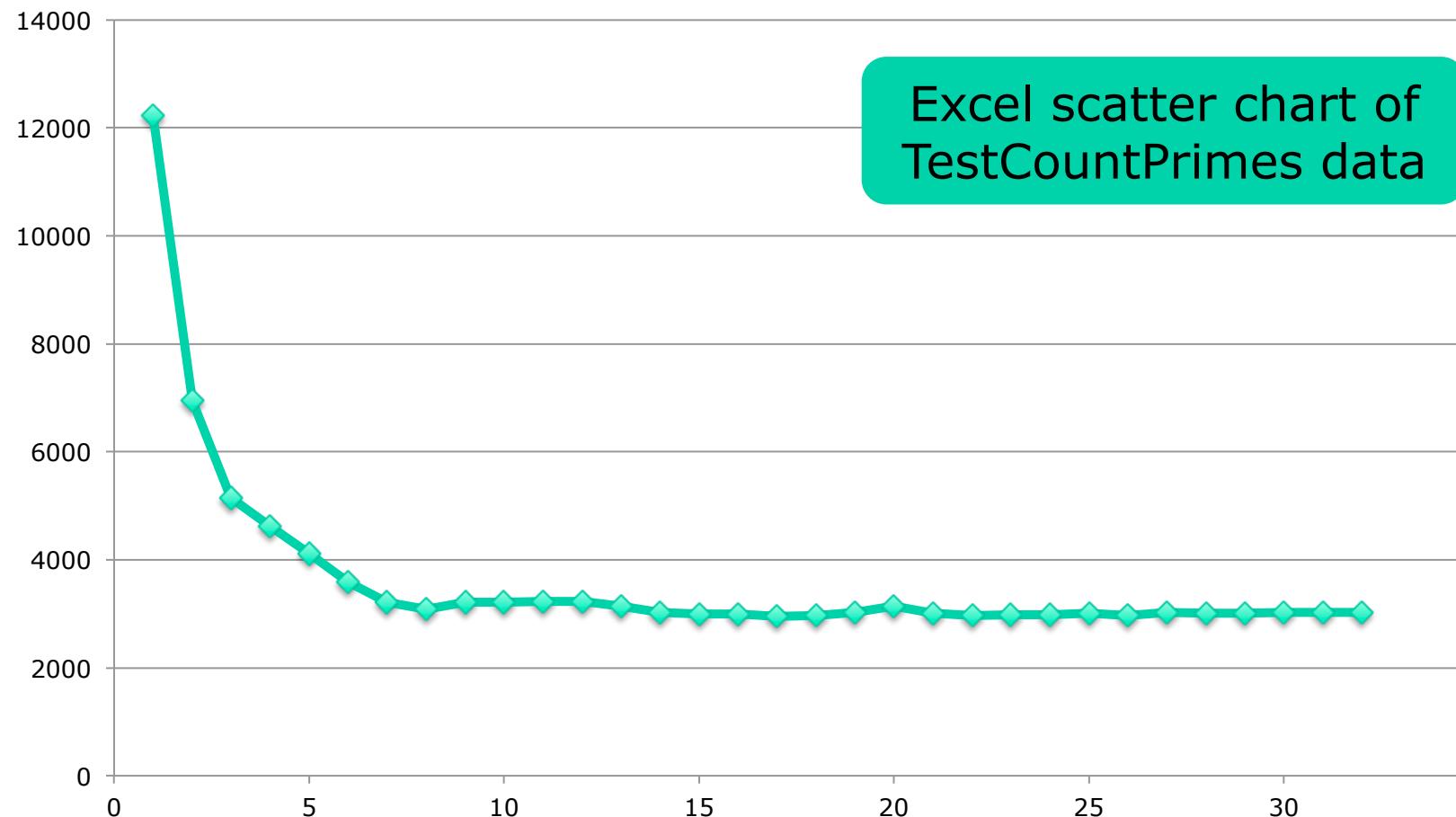
TestCountPrimes results

countParallel	1	11887.9 us	513.02	32
countParallel	2	7313.4 us	792.47	32
countParallel	3	5085.8 us	67.75	64
countParallel	4	4697.3 us	76.39	64
countParallel	5	4042.7 us	40.06	64
countParallel	6	3577.5 us	19.87	128
countParallel	7	3233.1 us	8.28	128
countParallel	8	3149.4 us	77.59	128
countParallel	9	3196.3 us	11.66	128
countParallel	10	3203.0 us	8.49	128
countParallel	11	3198.5 us	15.70	128
countParallel	12	3263.3 us	27.53	128
countParallel	13	3128.0 us	16.66	128
countParallel	14	3021.6 us	19.58	128
countParallel	15	2960.8 us	11.23	128
countParallel	16	3033.4 us	65.49	128
countParallel	17	2926.2 us	5.94	128
countParallel	18	2972.6 us	21.47	128
countParallel	19	3001.7 us	6.40	128
countParallel	20	3051.9 us	37.81	128
countParallel	21	2992.3 us	8.10	128
countParallel	22	2978.9 us	20.45	128
countParallel	23	2957.3 us	5.70	128
countParallel	24	2978.5 us	7.67	128
countParallel	25	3006.8 us	38.01	128
countParallel	26	2972.0 us	19.80	128
countParallel	27	2993.0 us	63.53	128
countParallel	28	3008.0 us	24.42	128
countParallel	29	2997.7 us	5.80	128
countParallel	30	3019.1 us	21.74	128
countParallel	31	2998.5 us	2.80	128
countParallel	32	3000.7 us	2.38	128

- One thread slower than sequential
- Max speedup 4.1x
- From some point, more threads are worse
- How choose best thread count?
- Tasks and executors are better than threads, week 5

Making plots of measurements

- Zillions of plotting and charting programs, including Excel, Gnuplot, R, Ploticus, ...
- Always use scatter (x-y) plots, no smoothing



General advice

- To avoid interference with measurements, shut down other programs: mail, Skype, browsers, Dropbox, iTunes, MS Office ...
- Disable logging and debugging messages
- Compile with optimizations enabled
- Never measure inside IDEs such as Eclipse
- Turn off power-savings modes
- Run on mains power, not on battery
- Lots of differences between
 - Runtime systems: Oracle, IBM Java; Mono, .NET
 - CPUs: Intel i5, i7, Xeon, AMD, ARM, ...

Mistakes and pitfalls

- Windows Upgrade etc may ruin measurements
 - Runs at unpredictable times, and is slow
- Some CPUs have a temporary “turbo mode”
 - May increase clock speed, will ruin measurements
- Some CPUs do “thermal throttling” if too hot
 - May reduce clock speed, will ruin measurements
- Measure the right thing
 - Eg when measuring binary search, do not search for the same item repeatedly (notes §11)
- Beware of irrelevant overheads
 - For instance random number generation
 - (But now you know how to measure the overhead!)

Timing threads à la Goetz & Bloch

- A countdown N-latch is a use-once gate
 - When `latch.countDown()` has been called N times, all threads blocked on `latch.await()` are unblocked
- Can use it to measure thread wall-clock time
 - **excluding** thread creation and start-up
- But thread start costs seems relevant too...

Timing threads à la Goetz & Bloch

```
final CountDownLatch startGate = new CountDownLatch(1);
final CountDownLatch endGate = new CountDownLatch(threadCount);
for (int i = 0; i < threadCount; i++) {
    Thread t = new Thread(new Runnable() { public void run() {
        try {
            startGate.await();
            try { task.run(); }
            finally { endGate.countDown(); }
        } catch (InterruptedException ignored) { }
    } });
    t.start();
}
Timer timer = new Timer();
startGate.countDown();
endGate.await();
double time = timer.check();
```

The diagram illustrates the timing of threads. The main thread (right) sends a signal to startGate (left). Worker threads (left) then await start, do work, and finally signal end. The main thread then checks the timer.

worker threads

main thrd

Goetz p. 96

See also Bloch p. 275

- All threads start nearly at the same time
- Measure excludes thread creation overhead

This week

- Reading
 - Sestoft: Microbenchmarks in Java and C#
 - (Optional) McKenney chapter 3
- Exercises week 4 = Mandatory hand-in 2
 - Conduct meaningful performance measurements and comparisons, and discuss the results
- Read before next week's lecture
 - Goetz chapters 6 and 8
 - Bloch items 68, 69

Practical Concurrent and Parallel Programming 5

Peter Sestoft
IT University of Copenhagen

Friday 2014-09-26*

Plan for today

- Tasks and the Java executor framework
 - Executors, Runnables, Callables, Futures
- The states of a task
- Task creation overhead
- Using tasks to count prime numbers
- Java versus the .NET Task Parallel Library
- Producer-consumer pipelines
- Bounded queues, thread wait() and notify()
- The states of a thread

Exercises ...

- The 1200-1400 time slot is in **3A18** again
- Hand-ins: Submit a zip-file containing
 - explanation as a text file answers.txt
 - a subdirectory src/ with source code
 - graphs as image files *.jpg or *.png
 - NO Netbeans projects, Eclipse workspaces or other junk

Prefer executors and tasks to threads

- We have used *threads* to parallelize work
 - But creating many threads takes time and memory
- Better divide work into small *tasks*
 - Then submit the tasks to an executor
 - This uses a pool of (few) threads to run the tasks
- Goetz chapters 6-8 and Bloch item 68

should generally refrain from working directly with threads. The key abstraction is no longer `Thread`, which served as both the unit of work and the mechanism for executing it. Now the unit of work and mechanism are separate. The key abstraction is the unit of work, which is called a *task*. There are two kinds of tasks: `Runnable` and its close cousin, `Callable` (which is like `Runnable`, except that it returns a value). The general mechanism for executing tasks is the *executor service*.

Bloch item 68

Executors and tasks

- A task is just a Runnable or Callable<T>
- Submitting it to an executor gives a Future

```
Future<?> fut
    = executor.submit(new Runnable() { public void run() {
        System.out.println("Task ran!");
    } });

```

TestTaskSubmit.java

- The executor has a pool of threads and uses one of them to run the task
- Use the Future to wait for task completion

```
try { fut.get(); }
catch (InterruptedException exn) { System.out.println(exn); }
catch (ExecutionException exn) { throw new RuntimeException(exn); }
```

A task that produces a result

- Make the task from a Callable<T>

Future's result type

... same a Callable's

```
Future<String> fut
    = executor.submit(new Callable<String>() {
        public String call() throws IOException {
            return getPage("http://www.wikipedia.org", 10);
    }});
```

TestTaskSubmit.java

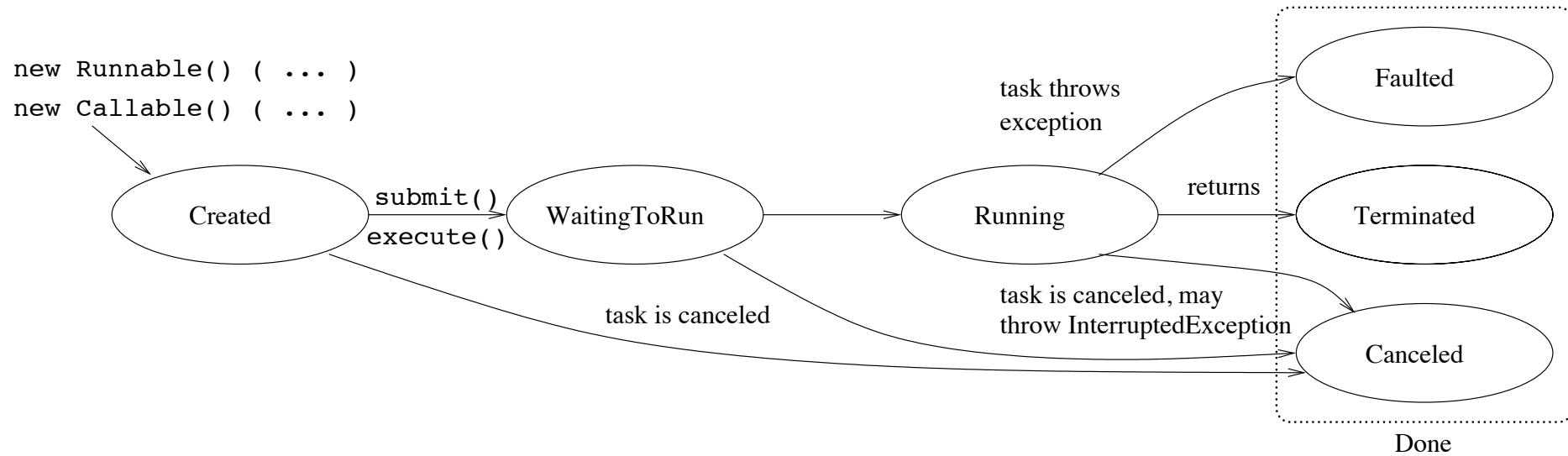
- Use the Future to get the task's result:

```
try {
    String webpage = fut.get();
    System.out.println(webpage);
} catch (InterruptedException exn) { System.out.println(exn); }
catch (ExecutionException exn) { throw new RuntimeExcep...; }
```

Task rules

- Different tasks may run on different threads
 - Objects accessed from tasks must be thread-safe
- A thread running a task can be interrupted
 - So a task can be interrupted
 - So `fut.get()` can throw `InterruptedException`
- Creating a task is fast, takes little memory
- Creating a thread is slow, takes much mem.

The states of a task



- After **submit** or **execute**
 - a task may be running immediately or much later
 - depending on the executor and available threads

Thread creation vs task creation

- Task creation is faster than thread creation

	Thread	Task	
Work	6581 ns	6612 ns	
Create	1030 ns	77 ns	
Create+start/(submit+cancel)	48929 ns	835 ns	
Create+(start/submit)+complete	72759 ns	21226 ns	Intel i7 2.4 GHz JVM 1.8

- A task also uses much less memory

Various Java executors

- In class `java.util.concurrent.Executors`:
- `newFixedThreadPool(n)`
 - Fixed number n of threads; automatic restart
- `newCachedThreadPool()`
 - Dynamically adapted number of threads, no bound
- `newSingleThreadExecutor()`
 - A single thread; so tasks need not be thread-safe
- `newScheduledThreadPool()`
 - Delayed and periodic tasks; eg clean-up, reporting
- `newWorkStealingPool()`  New in Java 8. Use it
 - Adapts thread pool to number of processors, uses multiple queues; therefore better scalability

Plan for today

- Tasks and the Java executor framework
 - Executors, Runnables, Callables, Futures
- The states of a task
- Task creation overhead
- **Using tasks to count prime numbers**
- Java versus the .NET Task Parallel Library
- Producer-consumer pipelines
- Bounded queues, thread wait and notify
- The states of a thread

Week 2 flashback: counting primes in multiple threads

```
final LongCounter lc = new LongCounter();
Thread[] threads = new Thread[threadCount];
for (int t=0; t<threadCount; t++) {
    final int from = perThread * t,
        to = (t+1==threadCount) ? range : perThread * (t+1);
    threads[t] = new Thread(new Runnable() { public void run()
        for (int i=from; i<to; i++)
            if (isPrime(i))
                lc.increment();
    } );
}
for (int t=0; t<threadCount; t++)
    threads[t].start();
```

Last thread has
to==range

Thread t processes
segment [from,to)

TestCountPrimes.java

- Creates one thread for each segment

Counting primes in multiple tasks

```

final LongCounter lc = new LongCounter();
List<Future<?>> futures = new ArrayList<Future<?>>();
for (int t=0; t<taskCount; t++) {
    final int from = perTask * t,
        to = (t+1 == taskCount) ? range : perTask * (t+1);
    futures.add(executor.submit(new Runnable() { public void run() {
        for (int i=from; i<to; i++)
            if (isPrime(i))
                lc.increment();
    } }));
```

} });

try {

for (Future<?> fut : futures)

fut.get();

} catch (...) { ... }

Add to shared

Create task, submit to executor, save a future

Wait for all tasks to complete

TestCountPrimesTasks.java

- Creates a task for each segment
- The tasks execute on a thread pool

Tasks that return task-local counts

```

List<Callable<Long>> tasks = new ArrayList<Callable<Long>>();
for (int t=0; t<taskCount; t++) {
    final int from = perTask * t,
        to = (t+1 == taskCount) ? range : perTask * (t+1);
    tasks.add(new Callable<Long>() { public Long call() {
        long count = 0;
        for (int i=from; i<to; i++)
            if (isPrime(i))
                count++;
        return count;
    } });
}
long result = 0;
try {
    List<Future<Long>> futures = executor.invokeAll(tasks);
    for (Future<Long> fut : futures)
        result += fut.get();
} catch (...) { ... }

```

Create task

Add to local

Submit tasks, wait for all to complete, get futures

Add local task results

TestCountPrimesTasks.java

Callable<Void> is like Runnable

```
final LongCounter lc = new LongCounter();
List<Callable<Void>> tasks = new ArrayList<Callable<Void>>()
for (int t=0; t<taskCount; t++) {
    final int from = perTask * t,
        to = (t+1 == taskCount) ? range : perTask * (t+1);
tasks.add(new Callable<Void>() { public Void call() {
    for (int i=from; i<to; i++)
        if (isPrime(i))
            lc.increment();
return null;
}});
}
try {
    executor.invokeAll(tasks);
} catch (...) { ... }
```

Annotations:

- Create task** (green callout, points to `tasks.add(new Callable<Void>() { public Void call() {`)
- Add to shared** (purple callout, points to `lc.increment();`)
- Submit tasks, wait for all to complete** (green callout, points to `executor.invokeAll(tasks);`)

TestCountPrimesTasks.java

Type parameters <Void> and <?>

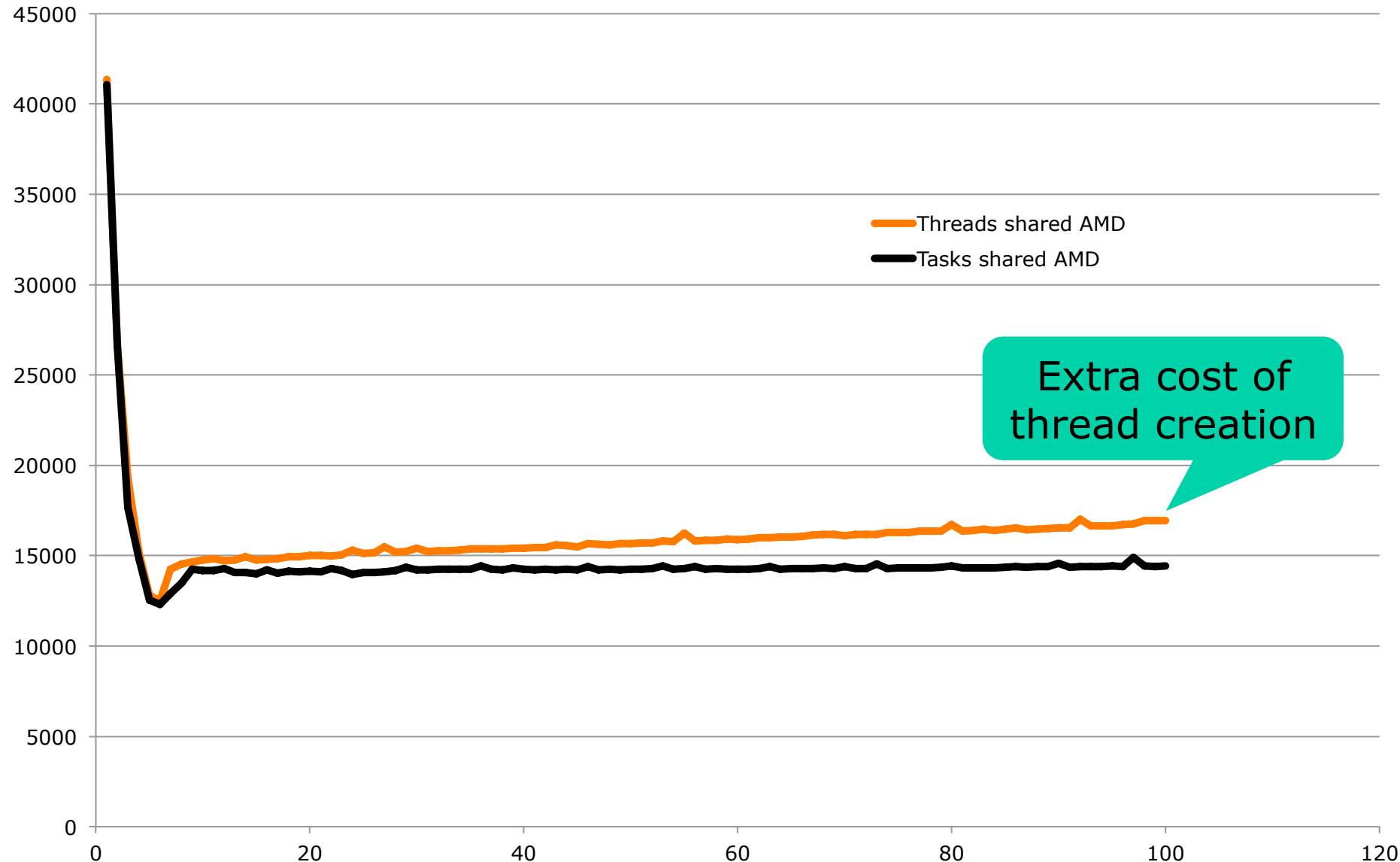
- The type `java.lang.Void` contains only `null`
- `Callable<Void>` requires `Void call() { ... }`
 - Similar to `Runnable`'s `void run() { ... }`
 - With `Future<Void>` the `get()` returns `null`
- `Future<?>` has an unknown type of value
 - With `Future<?>` the `get()` returns `null` also
- Java's type system is a somewhat muddled
 - Will not allow this assignment:

Type `Future<?>`

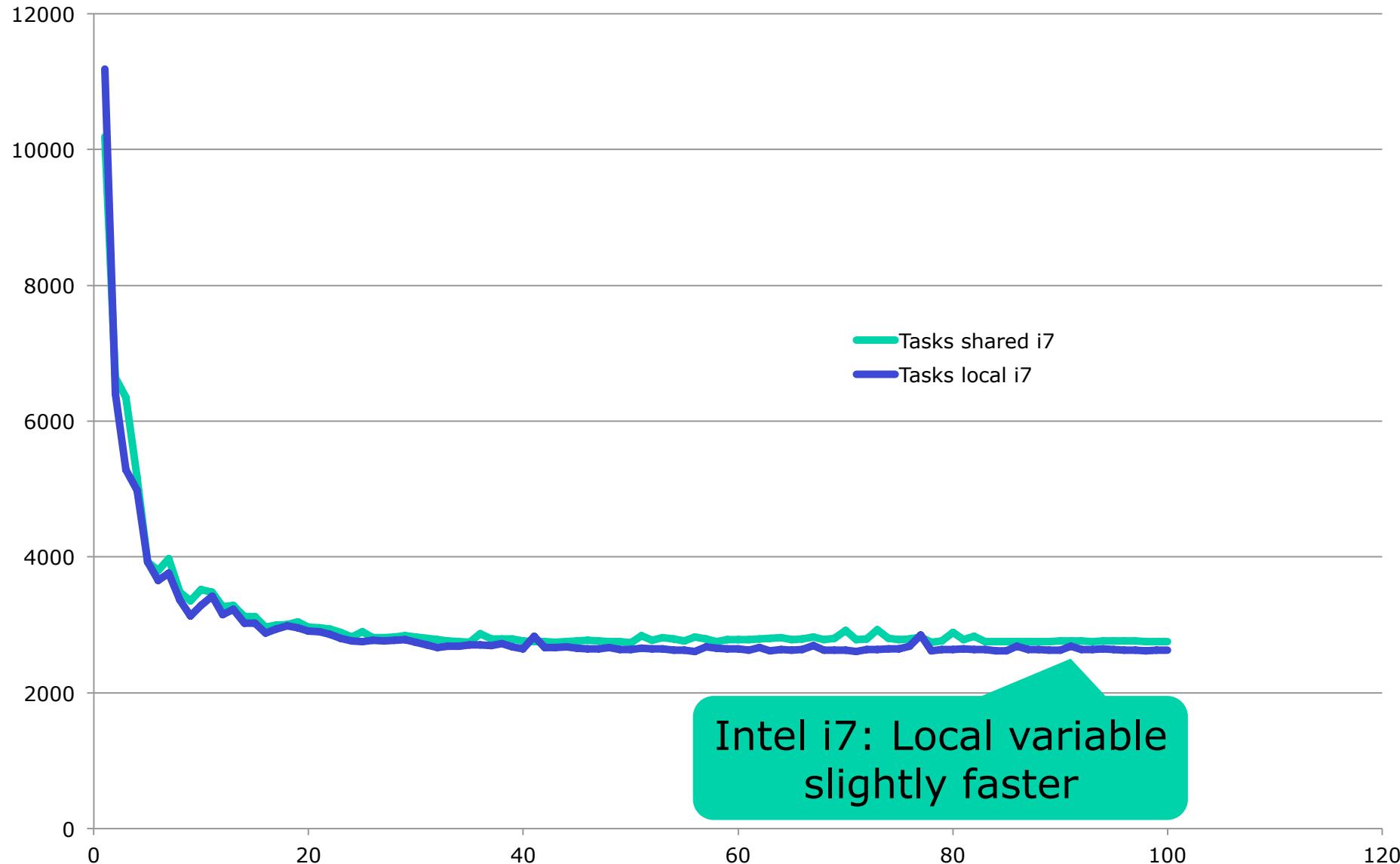
```
Future<Void> future;  
future = executor.submit(new Runnable() { ... });
```

Not
same

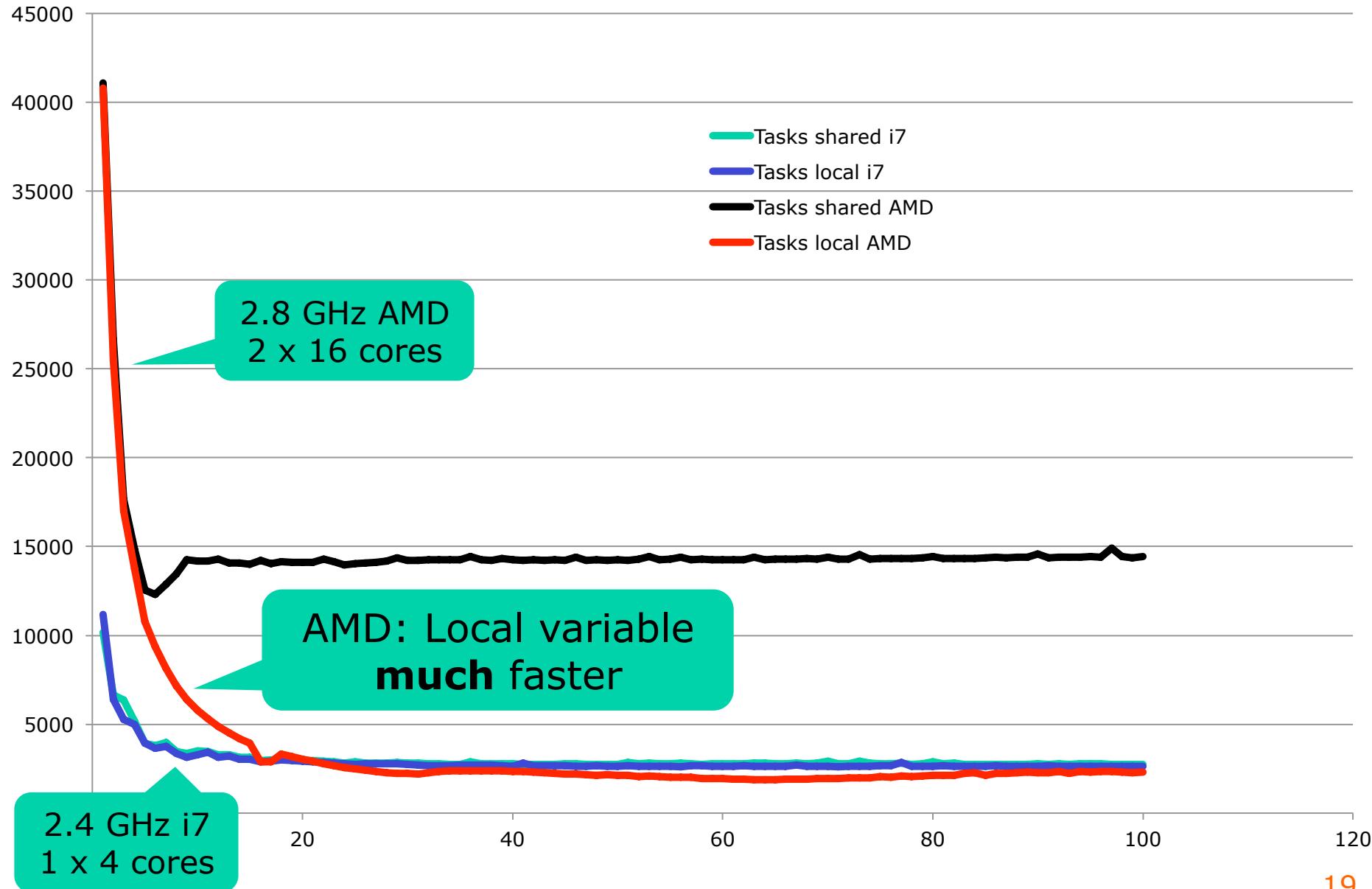
Overhead of creating many threads



Shared counter vs local counter



Computers differ a lot



Plan for today

- Tasks and the Java executor framework
 - Executors, Runnables, Callables, Futures
- The states of a task
- Task creation overhead
- Using tasks to count prime numbers
- **Java versus the .NET Task Parallel Library**
- Producer-consumer pipelines
- Bounded queues, thread wait and notify
- The states of a thread

The .NET Task Parallel Library

- Since C#/.NET 4.0, 2010
- Easier to use and better language integration
 - `async` and `await` keywords in C#
 - .NET class library has more non-blocking methods
 - Java may get them in version 9 (2016)
- Namespace `System.Threading.Tasks`
- Class `Task` combines `Runnable` & `Future`
- Class `Task<T>` combines `Callable<T>` and `Future<T>`
- See *C# Precisely* chapters 22 and 23

Parallel prime counts in C#, shared

```
int perTask = range / taskCount;
LongCounter lc = new LongCounter();
Parallel.For(0, taskCount, t =>
{
    int from = perTask * t,
        to = (t+1 == taskCount) ? range : perTask * (t+1);
    for (int i=from; i<to; i++)
        if (isPrime(i))
            lc.increment();
});
return lc.get();
```

Create tasks, submit to standard executor, run

TestCountPrimesTasks.cs

- Same concepts as in Java
 - much leaner notation
 - easier to use out of the box
- The tasks are executed on a thread pool
 - in an unknown order

Create task t

Parallel prime counts in C#, local

```
long[] results = new long[taskCount];
Parallel.For(0, taskCount, t =>
{ int from = perTask * t,
  to = (t+1 == taskCount) ? range : perTask * (t+1);
  long count = 0;
  for (int i=from; i<to; i++)
    if (isPrime(i))
      count++;
  results[t] = count;
} );
return results.Sum();
```

TestCountPrimesTasks.cs

- Q: Why safe to write to **results** array?

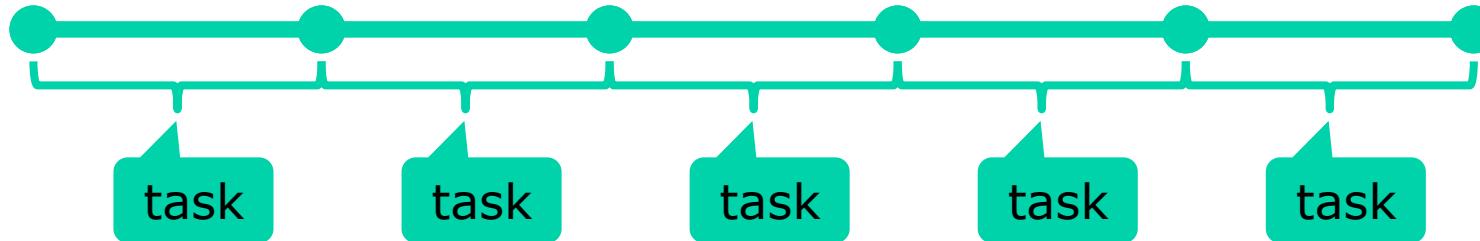
Every thread writes to a different field in results

Plan for today

- Tasks and the Java executor framework
 - Executors, Runnables, Callables, Futures
- The states of a task
- Task creation overhead
- Using tasks to count prime numbers
- Java versus the .NET Task Parallel Library
- **Producer-consumer pipelines**
- Bounded queues, thread wait and notify
- The states of a thread

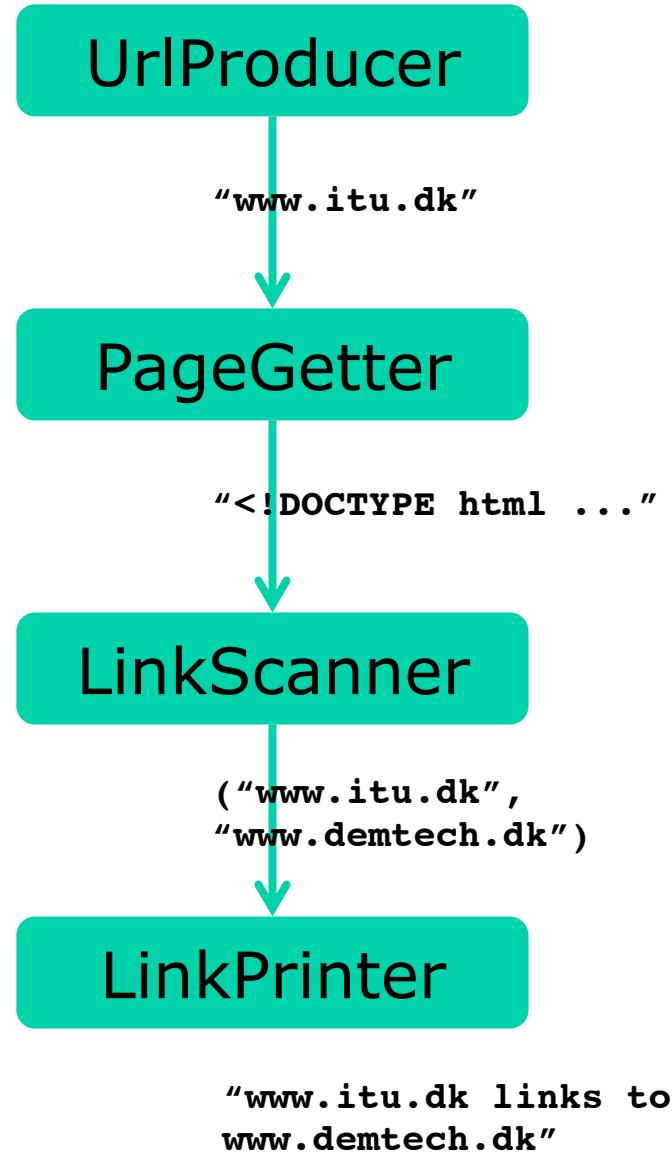
Concurrent pipelines (Goetz §5.3)

- We parallelized prime counting by splitting the work into chunks:



- A different way is to create a *pipeline*
- Example problem: Given long list of URLs,
 - For each URL,
 - download the webpage at that URL
 - scan the webpage for links ` ...`
 - for each link, print “`url links to link`”

Pipeline to produce URL, get webpage, scan for links, and print them



- There are four stages
- They can run in parallel
 - On four threads
 - Or as four tasks
- Each does a simple job
- Two stages communicate via a blocking queue
 - `queue.put(item)` sends data item to next stage; blocks until room for data
 - `queue.take()` gets data item from previous stage; blocks until data available

Sketch of a one-item queue

TestPipeline.java

```
interface BlockingQueue<T> {  
    void put(T item);  
    T take();  
}
```

```
class OneItemQueue<T> implements BlockingQueue<T> {  
    private T item;  
    private boolean full = false;  
    public void put(T item) {  
        synchronized (this) {  
            full = true;  
            this.item = item;  
        }  
    }  
    public T take() {  
        synchronized (this) {  
            full = false;  
            return item;  
        }  
    }  
}
```

Java monitor pattern, good

But: what if already full?

If queue full, we must wait for **another** thread to **take()** first

But: What if queue empty?

Other thread can **take()** only if we release lock first

Useless

Using `wait()` and `notifyAll()`

```
public void put(T item) {  
    synchronized (this) {  
        while (full) {  
            try { this.wait(); }  
            catch (InterruptedException exn) { }  
        }  
        full = true;  
        this.item = item;  
        this.notifyAll();  
    }  
}
```

If queue full, wait for
notify from other thread

When non-full, save item,
notify all waiting threads

TestPipeline.java

- **`this.wait()`**: release lock on `this`; do nothing until notified, then acquire lock and continue
 - Must hold lock in `this` before call
- **`this.notifyAll()`**: tell all threads `wait()`ing on `this` to wake up
 - Must hold lock on `this`, and keeps holding it

The `take()` method is similar

```
public T take() {  
    synchronized (this) {  
        while (!full) {  
            try { this.wait(); }  
            catch (InterruptedException exn) { }  
        }  
        full = false;  
        this.notifyAll();  
        return item;  
    }  
}
```

If queue empty, wait for
notify from other thread

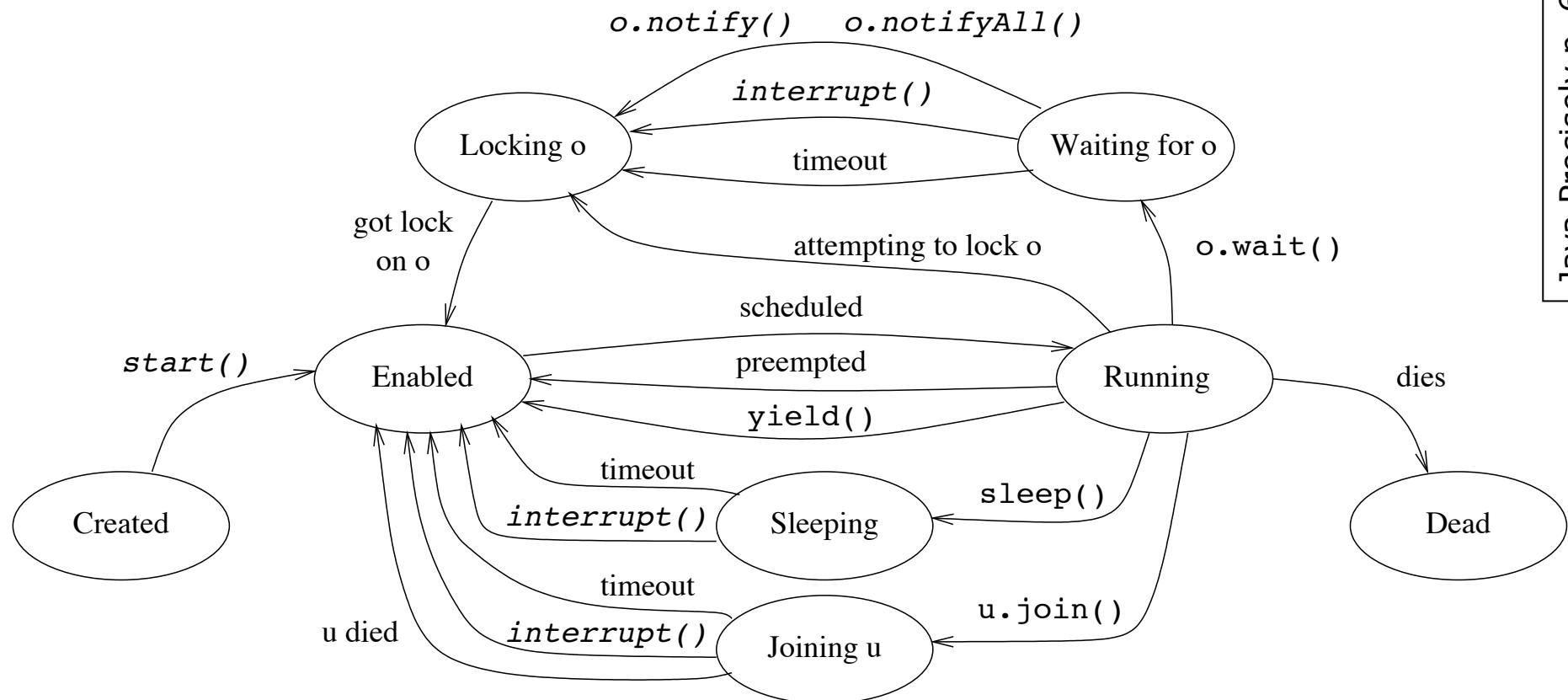
When non-empty, take item,
notify all waiting threads

TestPipeline.java

- Only works if **all** methods locking on the queue are written correctly
- MUST do the `wait()` in a while loop; Q: Why?

Always use the `wait` loop idiom to invoke the `wait` method; never invoke it outside of a loop. The loop serves to test the condition before and after waiting.

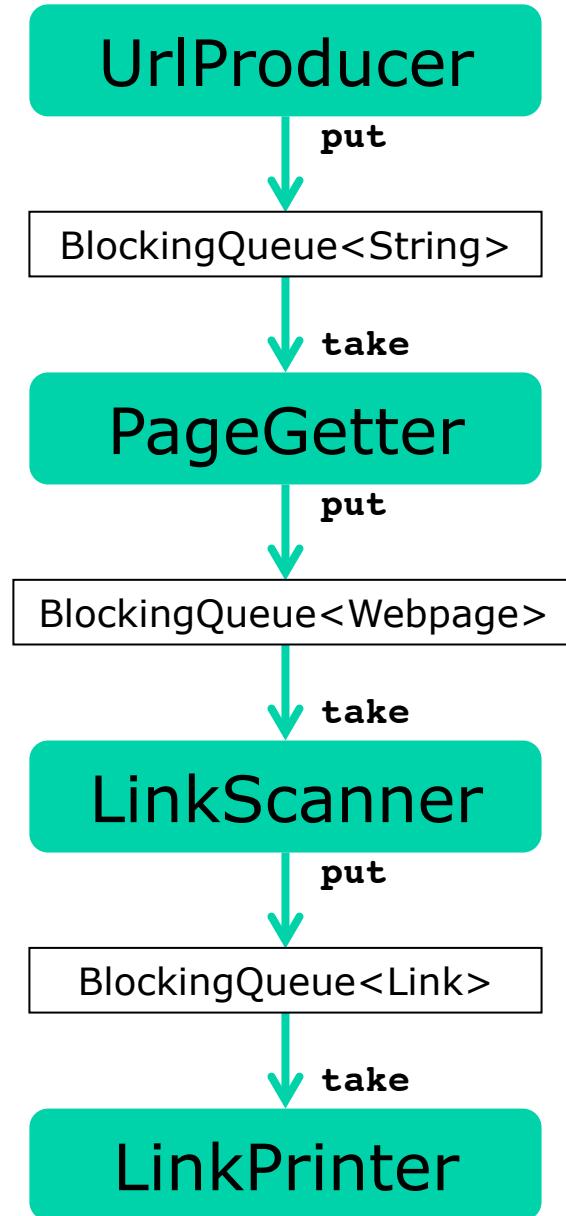
Java Thread states



Java Precisely p. 67

- `o.wait()` is an action of the running thread itself
- `o.notify()` is an action by another thread, on the waiting one
- `scheduled`, `preempted`, ... are actions of the system

Producer-consumer pattern: Pipeline stages and connecting queues



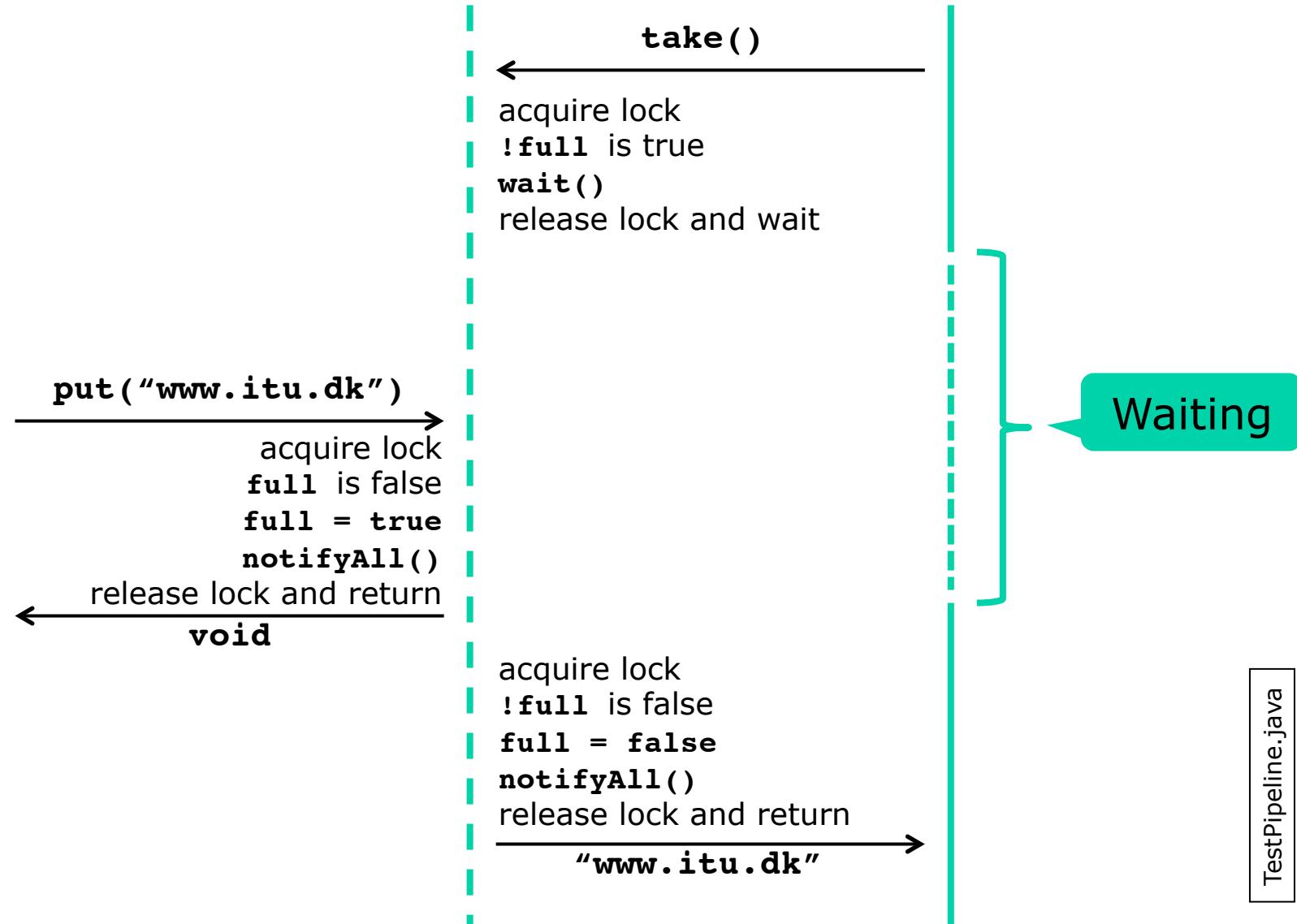
- The first stage is a producer only
- The middle stages are both consumers and producers
- The last stage is only a consumer
- A queue connects producer(s) to consumer(s) in a thread-safe way

How wait and notifyAll collaborate

UrlProducer
(active thread)

OneItemQueue
(passive object)

PageGetter
(active thread)



TestPipeline.java

Stages 1 and 2

```
class UrlProducer implements Runnable {  
    private final BlockingQueue<String> output;  
    public UrlProducer(BlockingQueue<String> output) {  
        this.output = output;  
    }  
    public void run() {  
        for (int i=0; i<urls.length; i++)  
            output.put(urls[i]);  
    }  
}
```

Produce URLs

TestPipeline.java

```
class PageGetter implements Runnable {  
    ...  
    public void run() {  
        while (true) {  
            String url = input.take();  
            try {  
                String contents = getPage(url, 200);  
                output.put(new Webpage(url, contents));  
            } catch (IOException exn) { System.out.println(exn); }  
        }  
    }  
}
```

Transform URL
to webpage

Stages 3 and 4

```
class LinkScanner implements Runnable {  
    ...  
    private final static Pattern urlPattern  
        = Pattern.compile("a href=\"(\\p{Graph}*)\"");  
    public void run() {  
        while (true) {  
            Webpage page = input.take();  
            Matcher urlMatcher = urlPattern.matcher(page.contents);  
            while (urlMatcher.find()) {  
                String link = urlMatcher.group(1);  
                output.put(new Link(page.url, link));  
            }  
        }  
    }  
}
```

Transform
web page to
link stream

```
class LinkPrinter implements Runnable {  
    ...  
    public void run() {  
        while (true) {  
            Link p = input.take();  
            System.out.printf("%s links to %s%n", p.from, p.to);  
        }  
    }  
}
```

Consume links
and print them

Putting stages and queues together

```
final BlockingQueue<String> urls = new OneItemQueue<String>();  
final BlockingQueue<Webpage> pages = new OneItemQueue<Webpage>();  
final BlockingQueue<Link> refPairs = new OneItemQueue<Link>();  
Thread t1 = new Thread(new UrlProducer(urls));  
Thread t2 = new Thread(new PageGetter(urls, pages));  
Thread t3 = new Thread(new LinkScanner(pages, refPairs));  
Thread t4 = new Thread(new LinkPrinter(refPairs));  
t1.start(); t2.start(); t3.start(); t4.start();
```

TestPipeline.java

- Each stage does *one* job
 - Simple to implement and easy to modify
 - Separation of concerns, simple control flow
- Easy to add new stages
 - For instance, discard duplicate links
- Can achieve high throughput
 - May run multiple copies of a slow stage

“Prefer concurrency utilities to `wait` and `notify`”

Bloch item 69

- It's instructive to use `wait` and `notify`
- ... but easy to do it wrong
- Package `java.util.concurrent` has
 - `BlockingQueue<T>` interface
 - `ArrayBlockingQueue<T>` class and much more
- Better use those in practice
- Next week: same pipeline with Java 8 streams
 - Simpler, and *very* easy to parallelize
 - (But of course still needs to be thread-safe)

This week

- Reading
 - Goetz et al chapters 5.3, 6 and 8
 - Bloch items 68, 69
- Exercises week 5
 - Show that you can use tasks and the executor framework, and modify a concurrent pipeline
- Read before next week's lecture
 - Goetz chapter 10
 - Bloch item 67

Practical Concurrent and Parallel Programming 6

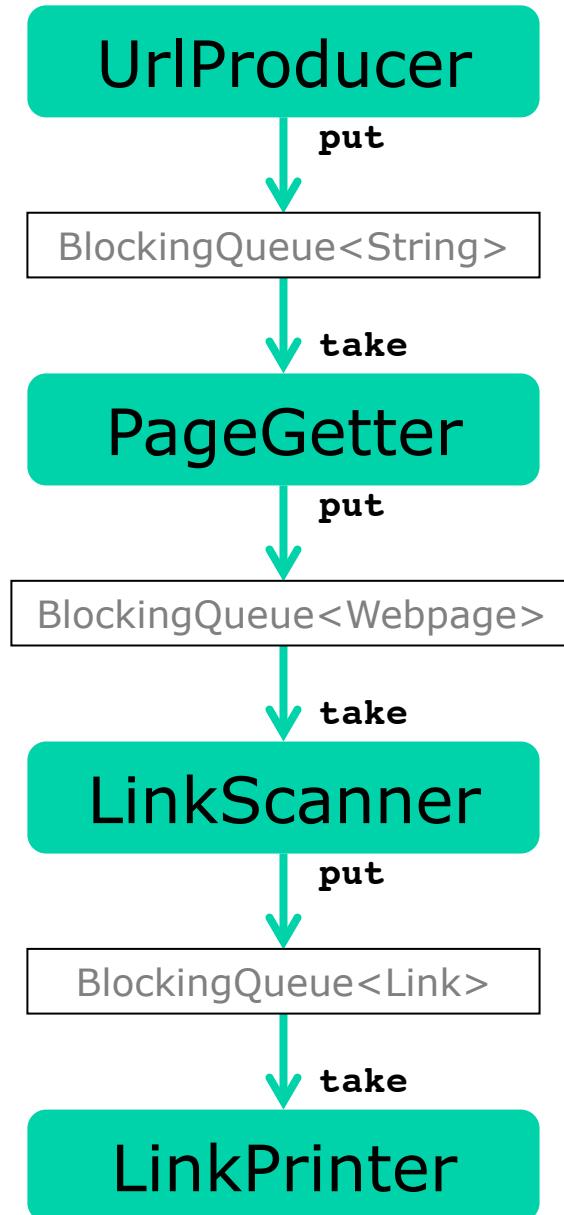
Peter Sestoft
IT University of Copenhagen

Friday 2014-10-03

Plan for today

- Pipelines with Java 8 streams
 - Easy and efficient parallelization
- Locking on multiple objects
- Deadlock and locking order
- Tool: jvisualvm, a JVM runtime visualizer
- Explicit locks, `lock.tryLock()`
- Liveness
- Concurrent correctness: safety + liveness
- Tool: ThreadSafe, static checking

Recall from last week: A pipeline connected by queues



- All stages run in parallel
- Two stages communicate via a blocking queue

```
BlockingQueue<String> urls
    = new OneItemQueue<String>();
BlockingQueue<Webpage> pages
    = new OneItemQueue<Webpage>();
BlockingQueue<Link> refPairs
    = new OneItemQueue<Link>();
Thread
    t1 = new Thread(new UrlProducer(urls)),
    t2 = new Thread(new PageGetter(urls, pages)),
    t3 = new Thread(new LinkScanner(pages, refPairs)),
    t4 = new Thread(new LinkPrinter(refPairs));
t1.start(); t2.start(); t3.start(); t4.start();
```

TestPipeline.java

Using Java 8 streams instead

- Package `java.util.stream`
- A `Stream<T>` is a source of `T` values
 - Lazily generated
 - Can be transformed with `map(f)` and `flatMap(f)`
 - Can be filtered with `filter(p)`
 - Can be consumed by `forEach(action)`
- Generally simpler than concurrent pipeline

```
Stream<String> urlStream
  = Stream.of(urls);
Stream<Webpage> pageStream
  = urlStream.flatMap(url -> makeWebPageOrNone(url, 200));
Stream<Link> linkStream
  = pageStream.flatMap(page -> makeLinks(page));
linkStream.forEach(link ->
  System.out.printf("%s links to %s%n", link.from, link.to));
```

TestStreams.java

Making the stages run in parallel

```
Stream<String> urlStream
    = Stream.of(urls).parallel();
Stream<Webpage> pageStream
    = urlStream.flatMap(url -> makeWebPageOrNone(url, 200));
Stream<Link> linkStream
    = pageStream.flatMap(page -> makeLinks(page));
linkStream.forEach(link ->
    System.out.printf("%s links to %s%n", link.from, link.to));
```

TestStreams.java

- Magic? No!
- Divides streams into substream chunks
- Evaluates the chunks in tasks
- Runs tasks on an executor called ForkJoinPool
 - Using a thread pool and work stealing queues
 - More precisely ForkJoinPool.commonPool()

So easy. Why learn about threads?

- Parallel streams use tasks, run on threads
- Should be **side effect free** and take no locks
- Otherwise all the usual thread problems:
 - updates must be made atomic (by locking)
 - updates must be made visible (by locking, volatile)
 - deadlock risk if locks are taken

Side-effects

Side-effects in behavioral parameters to stream operations are, in general, discouraged, as they can often lead to unwitting violations of the statelessness requirement, as well as other thread-safety hazards.

If the behavioral parameters do have side-effects, unless explicitly stated, there are no guarantees as to the *visibility* of those side-effects to other threads, nor are there any guarantees that different operations on the "same" element within the same stream pipeline are executed in the same thread. Further, the ordering of those effects may be surprising.

Counting primes on Java 8 streams

- Our old standard Java for loop:

```
int count = 0;  
for (int i=0; i<range; i++)  
    if (isPrime(i))  
        count++;
```

Classical efficient
imperative loop

- Sequential Java 8 stream:

```
IntStream.range(0, range)  
    .filter(i -> isPrime(i))  
    .count()
```

Pure functional
programming ...

- Parallel Java 8 for loop

```
IntStream.range(0, range)  
    .parallel()  
    .filter(i -> isPrime(i))  
    .count()
```

... and thus
parallelizable and
thread-safe

Performance results (!!)

- Counting the primes in 0 ... 99,999

Method	Intel i7 (us)	AMD Opteron (us)
Sequential for-loop	9962	40548
Sequential stream	9933	40772
Parallel stream	2752	1673
Best thread-parallel	2969	
Best task-parallel	2631	1874

- Functional streams give the simplest solution
- Nearly as fast as tasks, or faster:
 - Intel i7 (4 cores) speed-up: 3.6 x
 - AMD Opteron (32 cores) speed-up: 24.2 x
- The future is parallel – and functional ☺

Plan for today

- Pipelines with Java 8 streams
 - Easy and efficient parallelization
- **Locking on multiple objects**
- **Deadlock and locking order**
- **Tool: jvisualvm, a JVM runtime visualizer**
- Explicit locks, `lock.tryLock()`
- Liveness
- Concurrent correctness: safety + liveness
- Tool: ThreadSafe, static checking

Bank accounts and transfers

- An Account object à la Java monitor pattern:

```
class Account {  
    private long balance = 0;  
    public synchronized void deposit(long amount) {  
        balance += amount;  
    }  
    public synchronized long get() {  
        return balance;  
    }  
}
```

TestAccountUnsafe.java

- Naively add method for transfers:

```
public synchronized void transferA(Account that, long amount) {  
    this.balance = this.balance - amount;  
    that.balance = that.balance + amount;  
}
```

Bad

Two clerks working concurrently

```

account1.deposit(3000); account2.deposit(2000);
Thread clerk1 = new Thread(new Runnable() { public void run() {
    for (int i=0; i<transfers; i++)
        account1.transferA(account2, rnd.nextInt(10000));
} });
Thread clerk2 = new Thread(new Runnable() { public void run() {
    for (int i=0; i<transfers; i++)
        account2.transferA(account1, rnd.nextInt(10000));
} });
clerk1.start(); clerk2.start();

```

Transfer
ac1 to ac2

Transfer
ac2 to ac1

- Main thread occasionally prints balance sum:

```

for (int i=0; i<40; i++) {
    try { Thread.sleep(10); } catch (InterruptedException exn) { }
    System.out.println(account1.get() + account2.get());
}

```

- Method **transferA** may seem OK, but is not
- Why?

Losing updates with transferA

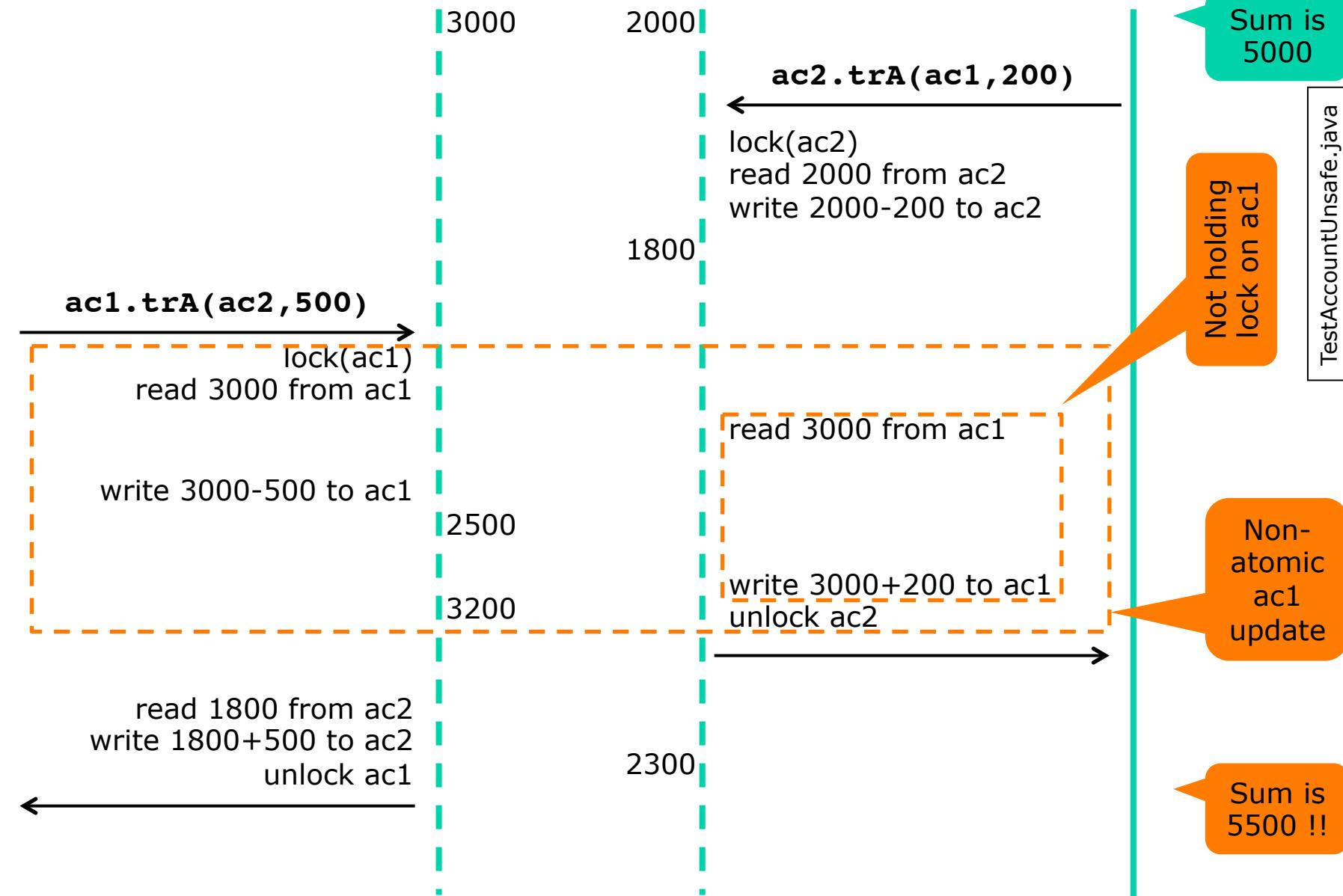
Acc A

Clerk 1

Account 1

Account 2

Clerk 2



TestAccounts version B

- Only one thread locks ac1
 - This does not achieve atomic update
- This would give atomic update of each account

```
public void transferB(Account that, long amount) {  
    this.deposit(-amount);  
    that.deposit(+amount);  
}
```

TestAccountUnsafe.java

- But a transfer is still non-atomic
 - so wrong, non-5000, account sums are observed:

```
...  
12919  
-8826  
-11648  
-10716  
Final sum is 5000
```

Must lock both accounts

- Atomic transfers and account sums require **all** accesses to lock on **both** account objects:

```
public void transferC(Account that, long amount) {  
    synchronized (this) { synchronized(that) {  
        this.balance = this.balance - amount;  
        that.balance = that.balance + amount;  
    } }  
}
```

Bad

TestAccountDeadlock.java

- But this may deadlock:
 - Clerk1 gets lock on ac1
 - Clerk2 gets lock on ac2
 - Clerk1 waits for lock on ac2
 - Clerk2 waits for lock on ac1
 - ... forever

Deadlocking with transferC

Acc C

Clerk 1

Account 1

Account 2

Clerk 2

`ac1.trA(ac2, 500)`

acquire lock on ac1

try to get lock on ac2

`ac2.trA(ac1, 200)`

acquire lock on ac2

try to get lock on ac1

Blocked forever

Deadlock

Blocked forever

TestAccountDeadlock.java

Avoiding deadlock, serial no.

Acc D

- Always take multiple locks **in the same order**
 - Give each account a unique serial number:

```
class Account {  
    private static final AtomicInteger intSequence = new AtomicInteger();  
    private final int serial = intSequence.getAndIncrement();  
    ...  
}
```

TestAccountLockOrder.java

- Take locks in serial number order:

```
public void transferD(Account that, final long amount) {  
    Account ac1 = this, ac2 = that;  
    if (ac1.serial <= ac2.serial)  
        synchronized (ac1) { synchronized (ac2) { // ac1 < ac2  
            ac1.balance = ac1.balance - amount;  
            ac2.balance = ac2.balance + amount;  
        } }  
    else  
        synchronized (ac2) { synchronized (ac1) { // ac2 < ac1  
            ac1.balance = ac1.balance - amount;  
            ac2.balance = ac2.balance + amount;  
        } }  
}
```

Atomic
and
deadlock
free

Avoiding deadlock, lock order

Acc D
Acc F

- **All** accesses must lock in the same order

```
public static long balanceSumD(Account ac1, Account ac2) {  
    if (ac1.serial <= ac2.serial)  
        synchronized (ac1) { synchronized (ac2) { // ac1 < ac2  
            return ac1.balance + ac2.balance;  
        } }  
    else  
        synchronized (ac2) { synchronized (ac1) { // ac2 < ac1  
            return ac1.balance + ac2.balance;  
        } }  
}
```

TestAccountLockOrder.java

- Cumbersome, we may encapsulate lock-taking

```
static void lockBothAndRun(Account ac1, Account ac2, Runnable action) {  
    if (ac1.serial <= ac2.serial)  
        synchronized (ac1) { synchronized (ac2) { action.run(); } }  
    else  
        synchronized (ac2) { synchronized (ac1) { action.run(); } }  
}
```

Avoiding deadlock, hashCode

Acc E

- Every object has an almost-unique hashCode
 - Hence no need to give accounts a serial number:
 - Instead take locks in hashCode order:

```
public void transferE(Account that, final long amount) {  
    Account ac1 = this, ac2 = that;  
    if (System.identityHashCode(ac1) <= System.identityHashCode(ac2))  
        synchronized (ac1) { synchronized (ac2) { // ac1 < ac2  
            ac1.balance = ac1.balance - amount;  
            ac2.balance = ac2.balance + amount;  
        } }  
    else  
        synchronized (ac2) { synchronized (ac1) { // ac2 < ac1  
            ac1.balance = ac1.balance - amount;  
            ac2.balance = ac2.balance + amount;  
        } }  
}
```

TestAccountLockOrder.java

Almost unbad

- Small risk of equal hashcodes and so deadlock
- See Goetz 10.1.2 + exercise how to eliminate

jvisualvm: Runtime Java thread state visualization

- Included with Java JDK since version 6
- Command-line tool: `jvisualvm`
- Can give graphical overview of thread history
 - As in `TestCountPrimes.java` (50m, 4 threads)
- Can display and diagnose most deadlocks
 - As in `TestAccountDeadlock.java`
- But not that in `TestPipelineSolution.java`
 - The tasks are blocked in Waiting, not in Locking
- Can produce much other information

Using jvisualvm on TestAccountDeadlock.java

○ TestAccountDeadlock (pid 10862)



Thread dump points to deadlock scenario

```
Found one Java-level deadlock:
```

```
=====
```

```
"Thread-1":
```

```
  waiting to lock monitor 0x00007fc43a010b48 (object 0x000000740088b40, a Account),  
  which is held by "Thread-0"
```

```
"Thread-0":
```

```
  waiting to lock monitor 0x00007fc43a010d58 (object 0x000000740088b28, a Account),  
  which is held by "Thread-1"
```

```
Java stack information for the threads listed above:
```

```
=====
```

```
"Thread-1":
```

```
  at Account.transferC(TestAccountDeadlock.java:61)  
  - waiting to lock <0x000000740088b40> (a Account)  
  - locked <0x000000740088b28> (a Account)  
  at TestAccountDeadlock$2.run(TestAccountDeadlock.java:29)  
  at java.lang.Thread.run(Thread.java:745)
```

transferC
method is
involved

```
"Thread-0":
```

```
  at Account.transferC(TestAccountDeadlock.java:61)  
  - waiting to lock <0x000000740088b28> (a Account)  
  - locked <0x000000740088b40> (a Account)  
  at TestAccountDeadlock$1.run(TestAccountDeadlock.java:23)  
  at java.lang.Thread.run(Thread.java:745)
```

Sources of deadlock

- Taking multiple locks in different orders
 - TestAccounts example
- Dependent tasks on too-small thread pool
 - Eg running last week's 4-stage pipeline on a FixedThreadPool with only 3 threads
- Synchronizing on too much
 - Use synchronized on statements, not methods
 - The reason C# has **lock** on statement, not methods
- When possible, use only *open calls*
 - Don't hold a lock when calling an unknown method

Deadlocks may be hard to spot

Taxi A

```
class Taxi {  
    private Point location, destination;  
    private final Dispatcher dispatcher;  
    public synchronized Point getLocation() { return location; }  
    public synchronized void setLocation(Point location) {  
        this.location = location;  
        if (location.equals(destination))  
            dispatcher.notifyAvailable(this);  
    }  
}  
  
class Dispatcher {  
    private final Set<Taxi> taxis;  
    private final Set<Taxi> availableTaxis;  
    public synchronized void notifyAvailable(Taxi taxi) {  
        availableTaxis.add(taxi);  
    }  
    public synchronized Image getImage() {  
        Image image = new Image();  
        for (Taxi t : taxis)  
            image.drawMarker(t.getLocation());  
        return image;  
    }  
}
```

Bad

Lock taxi

Call **notify...**, locks dispatcher

Deadlock risk!

Lock dispatcher

Call **getLocation**, locks taxi

Goetz p. 212

Locking less to remove deadlock

Taxi B

```
class Taxi {  
    public synchronized Point getLocation() { return location; }  
    public void setLocation(Point location) {  
        boolean reachedDestination;  
        synchronized (this) {  
            this.location = location;  
            reachedDestination = location.equals(destination);  
        }  
        if (reachedDestination)  
            dispatcher.notifyAvailable(this);  
    }  
}  
class Dispatcher {  
    public synchronized void notifyAvailable(Taxi taxi) { ... }  
    public Image getImage() {  
        Set<Taxi> copy;  
        synchronized (this) {  
            copy = new HashSet<Taxi>(taxis);  
        }  
        Image image = new Image();  
        for (Taxi t : copy)  
            image.drawMarker(t.getLocation());  
        return image;  
    } }
```

Lock taxi, make test, release lock

Call **notify...**
with no lock held

Lock dispatcher, copy
set, release lock

Call **getLocation**
with no lock held

Goetz p. 214

Locks for atomicity do not compose

- We use locks and synchronized for atomicity
 - when working with *mutable shared* data
- But this is not compositional
 - Atomic access of each of ac1 and ac2 does not mean atomic access to their combination, eg. sum
- Locks are pessimistic, there are alternatives:
- No mutable data
 - immutable data, functional programming
- No shared data
 - message passing, Akka library, week 13-14
- Accept mutable shared data, but avoid locks
 - optimistic concurrency, transactional memory, Multiverse library, week 10

Plan for today

- Pipelines with Java 8 streams
 - Easy and efficient parallelization
- Locking on multiple objects
- Deadlock and locking order
- Tool: jvisualvm, a JVM runtime visualizer
- **Explicit locks, lock.tryLock()**
- **Liveness**
- Concurrent correctness: safety + liveness
- Tool: ThreadSafe, static checking

Using explicit (and tryable) locks

- Namespace `java.util.concurrent.locks`
- New `Account` class with explicit locks:

```
class Account {  
    private final Lock lock = new ReentrantLock();  
  
    public void deposit(long amount) {  
        try {  
            lock.lock();  
            balance += amount;  
        } finally {  
            lock.unlock();  
        }  
    }  
  
    public long get() {  
        try {  
            lock.lock();  
            return balance;  
        } finally {  
            lock.unlock();  
        }  
    }  
}
```

Acquire lock

Always
release it

Acquire lock

Always
release it

Avoiding deadlock by retrying

- The Java runtime does not discover deadlock
- Unlike database servers
 - They typically lock tables automatically
 - In case of deadlock, abort and retry
- Similar idea can be used in Java
 - Try to take lock ac1
 - If successful, try to take lock on ac2
 - If successful, do action, release both locks, we are done
 - Else release lock on ac1, and start over
 - Else start over
 - Main (small) risk: may forever “start over”
 - Related to optimistic concurrency
 - and to software transactional memory, week 10

Taking two locks, using tryLock()

```

public void transferG(Account that, final long amount) {
    Account ac1 = this, ac2 = that;
    while (true) {
        if (ac1.lock.tryLock()) {
            try {
                if (ac2.lock.tryLock()) {
                    try {
                        ac1.balance = ac1.balance - amount;
                        ac2.balance = ac2.balance + amount;
                        return;
                    } finally {
                        ac2.lock.unlock();
                    }
                }
            } finally {
                ac1.lock.unlock();
            }
        }
        try { Thread.sleep(0, (int)(500 * Math.random())); }
        catch (InterruptedException exn) { }
    }
}

```

Try locking ac1

Try locking ac2

Actual work

If success, do work
and exit; else retryIn any case, release
acquired locksSleep 0-500 ns
before retry to
save CPU time

Livelock: nobody makes progress

- The `transferG` method never deadlocks
- In principle it can *livelock*:
 - Thread 1 locks ac1
 - Thread 2 locks ac2
 - Thread 1 tries to lock ac2 but discovers it cannot
 - Thread 2 tries to lock ac1 but discovers it cannot
 - Thread 1 releases ac1, sleeps, starts over
 - Thread 2 releases ac2, sleeps, starts over
 - ... forever ...
- Extremely unlikely
 - requires the sleep periods to be the same always
 - requires the operation interleaving to be the same

Correctness = Safety + Liveness

- Safety: nothing bad happens
 - Invariants are preserved, no updates lost, etc
- Liveness: something happens
 - No deadlock, no livelock
- You must be able to use these concepts:

Testing the condition before waiting and skipping the wait if the condition already holds are necessary to ensure liveness. If the condition already holds and the `notify` (or `notifyAll`) method has already been invoked before a thread waits, there is no guarantee that the thread will *ever* wake from the wait.

Testing the condition after waiting and waiting again if the condition does not hold are necessary to ensure safety. If the thread proceeds with the action when the condition does not hold, it can destroy the invariant guarded by the lock. There

```
while (<condition> is false) {  
    try { this.wait(); }  
    catch (InterruptedException exn) { }  
} // Now <condition> is true
```

Bloch p. 276

Remember
lecture 5

Plan for today

- Pipelines with Java 8 streams
 - Easy and efficient parallelization
- Locking on multiple objects
- Deadlock and locking order
- Tool: jvisualvm, a JVM runtime visualizer
- Explicit locks, `lock.tryLock()`
- Liveness
- Concurrent correctness: safety + liveness
- **Tool: ThreadSafe, static checking**

The ThreadSafe tool

- Download zip file, put files somewhere, eg. `~/lib/ts/`
- Download license file `threadsafe.properties` from LearnIT, put it the same place
- You may use ThreadSafe
 - from the command line (as we do here)
 - as Eclipse plugin (more convenient)
- Interpreting ThreadSafe's reports
- Apply ThreadSafe to Accounts
 - with `@GuardedBy` and no locking
 - with inadequate locking on transfers

Compiling `@GuardedBy` annotations

- Download `jsr305-3.0.0.jar`, link on homepage
- Put it somewhere, eg `~/lib/jsr305-3.0.0.jar`

```
import javax.annotation.concurrent.GuardedBy;

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

Defined in jar file

ts/guardedby/TestGuardedBy.java

- Compile like this:

```
$ javac -g -cp ~/lib/jsr305-3.0.0.jar TestGuardedBy.java
```

Emit debug info

Class path of jar file

- NB: `javac` does NOT check `@GuardedBy`

Checking `@GuardedBy` annotations

- Run `ThreadSafe` to check `@GuardedBy`
- Put a `threadsafe-project.properties` file in same directory:

```
projectName=counterTest
sources=.
binaries=.
outputDirectory=threadsafe-html
```

- Compile, run `ThreadSafe`, inspect report:

```
$ javac -g -cp ~/lib/jsr305-3.0.0.jar TestGuardedBy.java
$ java -jar ~/lib/ts/threadsafe.jar
INFO: Running analysis...
INFO: Analysis completed
$ open threadsafe-html/index.html
```

ts/guardedby/threadsafe-project.properties

Add method, forget synchronized

The screenshot shows a Java code editor with a sidebar and a main content area. The sidebar on the left lists 'Findings' for 'TestGuardedBy.java', including 'Problem location', 'Synchronized read', 'Synchronized write', 'Unsynchronized read', 'Unsynchronized write', and 'Synchronized read'. The main content area displays the following Java code:

```
// (see lecture 6) in ~/lib/ts/ and run it AFTER compiling as above
// java -jar ~/lib/ts/threadsafe.jar
// Then read ThreadSafe's report in a browser:
// open threadsafe-html/index.html

// Or do the whole thing in Eclipse, where it works more smoothly.

// From JSR 305 jar file jsr305-3.0.0.jar:
import javax.annotation.concurrent.GuardedBy;

import java.io.IOException;

public class TestGuardedBy {
    public static void main(String[] args) throws IOException {
        final LongCounter lc = new LongCounter();
        Thread t = new Thread(new Runnable() {
            public void run() {
                while (true)          // Forever call increment
                    lc.increment();
            }
        });
        t.start();
        System.out.println("Press Enter to get the current value:");
        while (true) {
            System.in.read();      // Wait for enter key
            System.out.println(lc.get());
        }
    }
}

class LongCounter {
    @GuardedBy("this")
    private long count = 0;
    public synchronized void increment() {
        count++;
    }
    public void decrement() {
        count++;
    }
    public synchronized long get() {
        return count;
    }
}
```

A green callout bubble with the word 'Violation' points to the line 'count++' in the 'decrement' method of the 'LongCounter' class. The line 'count++' is highlighted in blue, indicating it is a violation of the 'GuardedBy' annotation.

Analysing unsafe account transfer

Acc A

- Problem found, but message is subtle:

```
24
25  public synchronized void transferA(Account that, long amount) {
26      this.balance = this.balance - amount;
27      that.balance = that.balance + amount;
28  }
29
30  // This (wrongly) allows observation in the middle of a transfer
31  public void transferB(Account that, long amount) {
32      this.deposit(-amount);
33      that.deposit(+amount);
34  }
35 }
```

22 Synchronized read
26 Synchronized read
26 Synchronized write
27 Synchronized read
27 Synchronized write

Accesses

Rule description

Category: Locking

Severity: Major

Type: CCE_RA_GUARDED_BY_VIOLATED

Guards for access to field Account.balance:

Account.this <unknown>

UnsafeAccount.java: 18 Always Held Not Held

UnsafeAccount.java: 18 Always Held Not Held

UnsafeAccount.java: 22 Always Held Not Held

UnsafeAccount.java: 26 Always Held Not Held

UnsafeAccount.java: 26 Always Held Not Held

UnsafeAccount.java: 27 Not Held Always Held

UnsafeAccount.java: 27 Not Held Always Held

ts/accounts/UnsafeAccount.java

Using ThreadSafe

- Use ThreadSafe to check @GuardedBy
- Does a rather admirable job
 - Better on large projects than on small examples
- Is not perfect; Java is very difficult to analyse
 - False negatives: may fail to spot real unsafe code
 - False positives: may spot complain on safe code
- Rarely identifies actual deadlock risks
- Does not understand higher-order code well:

```
public static void lockBothAndRun(Account ac1, Account ac2, Runnable action) {  
    if (ac1.serial <= ac2.serial)  
        synchronized (ac1) { synchronized (ac2) { action.run(); } }  
    else  
        synchronized (ac2) { synchronized (ac1) { action.run(); } }  
}
```

TestAccountLockOrder.java

Thread scheduler, priorities, ...

- Controls the “scheduled” and “preempted” arcs in *Java Thread states* diagram, lecture 5

Item 72: Don't depend on the thread scheduler

Bloch p. 286

When many threads are runnable, the thread scheduler determines which ones get to run, and for how long. Any reasonable operating system will try to make this determination fairly, but the policy can vary. Therefore, well-written programs shouldn't depend on the details of this policy. **Any program that relies on the thread scheduler for correctness or performance is likely to be nonportable.**

- Thread priorities: Don't use them
 - except to make GUIs responsive by giving background worker threads lower priority
- Don't fix liveness or performance problems using `.yield()` and `.sleep(0)`; not portable

This week

- Reading
 - Goetz et al chapter 10 + 13.1
 - Bloch item 67
- Exercises week 6 = mandatory hand-in 3
 - Show that you can write non-deadlocking code, and that you can use tools such as jvisualvm and ThreadSafe
- Read before next week's lecture
 - Goetz et al chapter 11

Practical Concurrent and Parallel Programming 7

Peter Sestoft
IT University of Copenhagen

Friday 2014-10-10

Plan for today

- Performance and scalability
- Reduce lock duration, use lock splitting
- Hash maps, a scalability case study
 - (A) Hash map à la Java monitor
 - (B) Hash map with lock striping
 - (C) Ditto with lock striping and non-blocking reads
- An atomic long with “thread striping”
- Shared mutable state is slow on multicore

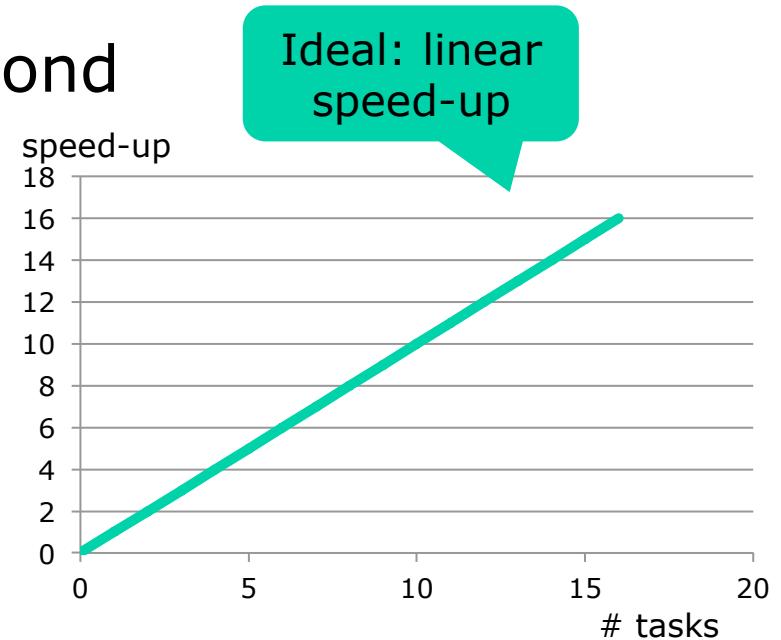
Performance versus scalability

- Performance

- Latency: time till first result
 - Throughput: results per second

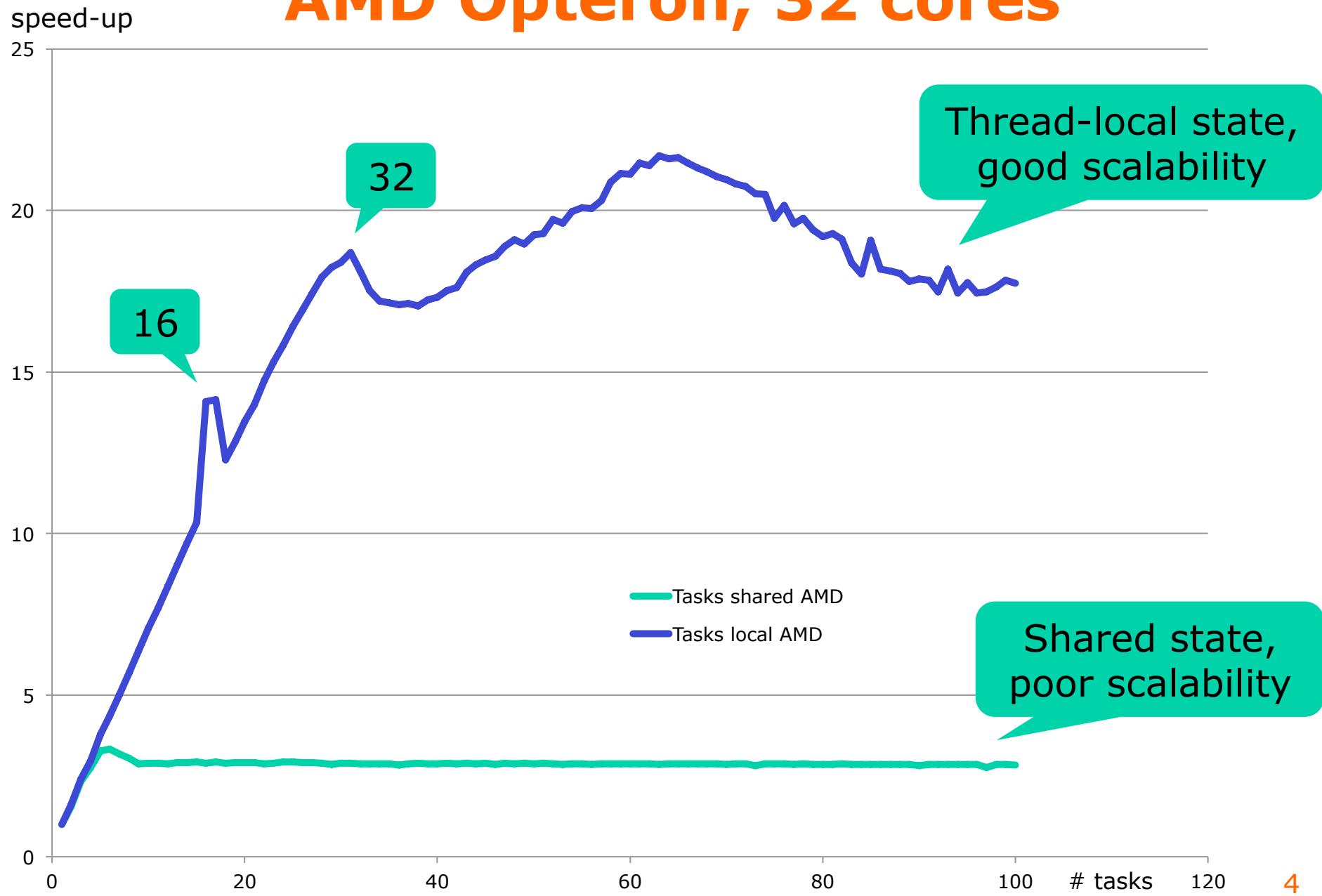
- Scalability

- Improved throughput when more resources are added
 - Speed-up as function of number of threads or tasks



- One may sacrifice performance for scalability
 - OK to be slower on 1 core if faster on 2 or 4 or ...
 - Requires rethinking our “best” sequential code

Scalability of prime counting AMD Opteron, 32 cores



What limits throughput?

- CPU-bound
 - Eg. counting prime numbers
 - To speed up, add more CPUs (cores)
- Memory-bound
 - Eg. make color histograms of images
 - To speed up, improve data locality; recompute more
- Input/output-bound
 - Eg. fetching webpages and finding links
 - To speed up, use more tasks
- Synchronization-bound
 - Eg. image segmentation using shared data structure
 - To speed up, improve shared data structure. How?

Much of this
lecture

What limits scalability?

- Sequentiality of *problem*
 - Example: growing a crop
 - 4 months growth + 1 month harvest if done by 1 person
 - Growth (sequential) cannot be speeded up
 - Using 30 people to harvest, takes $1/30$ month = 1 day
 - Maximal speed-up factor, using many many harvesters: $5/(4+1/30) = 1.24$ times faster
 - Amdahl's law
 - F = sequential fraction of problem = $4/5 = 0.8$
 - N = number of parallel resources = 30
 - Speed-up $\leq 1/(F+(1-F)/N) = 1/(0.8+0.2/30) = 1.24$
- Sequentiality of *solution*
 - Solution slower than necessary because shared resources, eg. locking, sequentialize solution

Reduce lock duration

```
public class AttributeStore {  
    private final Map<String, String> attributes = ...;  
    public synchronized boolean userLocationMatches(String name,  
                                                   String regexp)  
{  
    String key = "users." + name + ".location";  
    String location = attributes.get(key);  
    return location != null && Pattern.matches(regexp, location);  
}  
}
```

Must lock

May be slow, holds lock unnecessarily

- Better:

```
public class BetterAttributeStore {  
    private final Map<String, String> attributes = ...;  
    public boolean userLocationMatches(String name, String regexp) {  
        String key = "users." + name + ".location";  
        String location;  
        synchronized (this) {  
            location = attributes.get(key);  
        }  
        return location != null && Pattern.matches(regexp, location);  
    }  
}
```

Lock only here

Does not hold lock

Lock splitting

```
public class ServerStatusBeforeSplit {  
    @GuardedBy("this") public final Set<String> users = ...;  
    @GuardedBy("this") public final Set<String> queries = ...;  
    public synchronized void addUser(String u) {  
        users.add(u);  
    }  
    public synchronized void addQuery(String q) {  
        queries.add(q);  
    }  
    public synchronized void removeUser(String u) { . . .  
    }  
}
```

Lock server status object

Lock server status object

- Better, (addUser and addQuery can run concurrently)

```
public class ServerStatusAfterSplit {  
    @GuardedBy("users") public final Set<String> users = ...;  
    @GuardedBy("queries") public final Set<String> queries = ...;  
    public void addUser(String u) {  
        synchronized (users) { users.add(u); }  
    }  
    public void addQuery(String q) {  
        synchronized (queries) { queries.add(q); }  
    }  
    ...  
}
```

Lock only users set

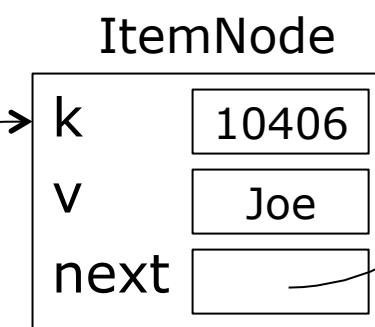
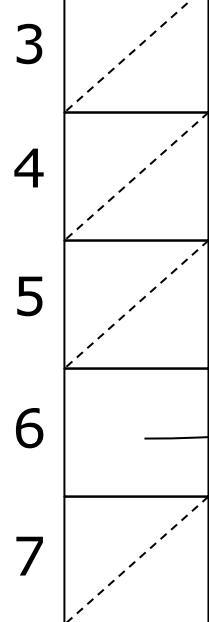
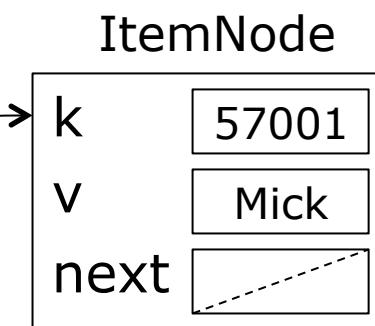
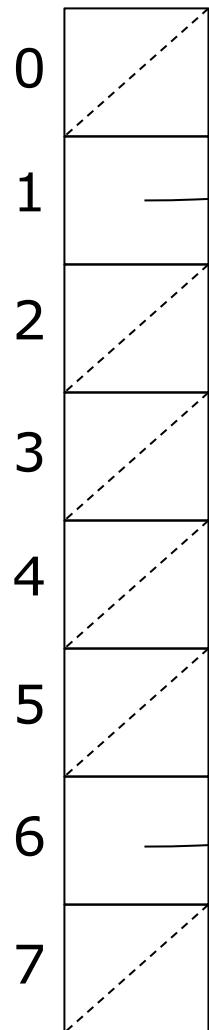
Lock only queries set

Plan for today

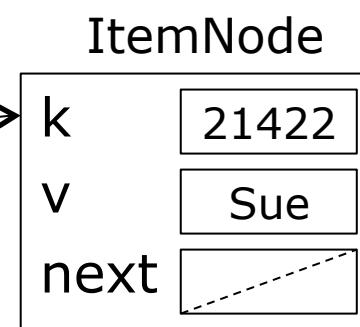
- Performance and scalability
- Reduce lock duration, use lock splitting
- **Hash maps, a scalability case study**
 - (A) Hash map à la Java monitor
 - (B) Hash map with lock striping
 - (C) Ditto with lock striping and non-blocking reads
- An atomic long with “thread striping”
- Shared mutable state is slow on multicore

A hash map = buckets table + item node lists

buckets

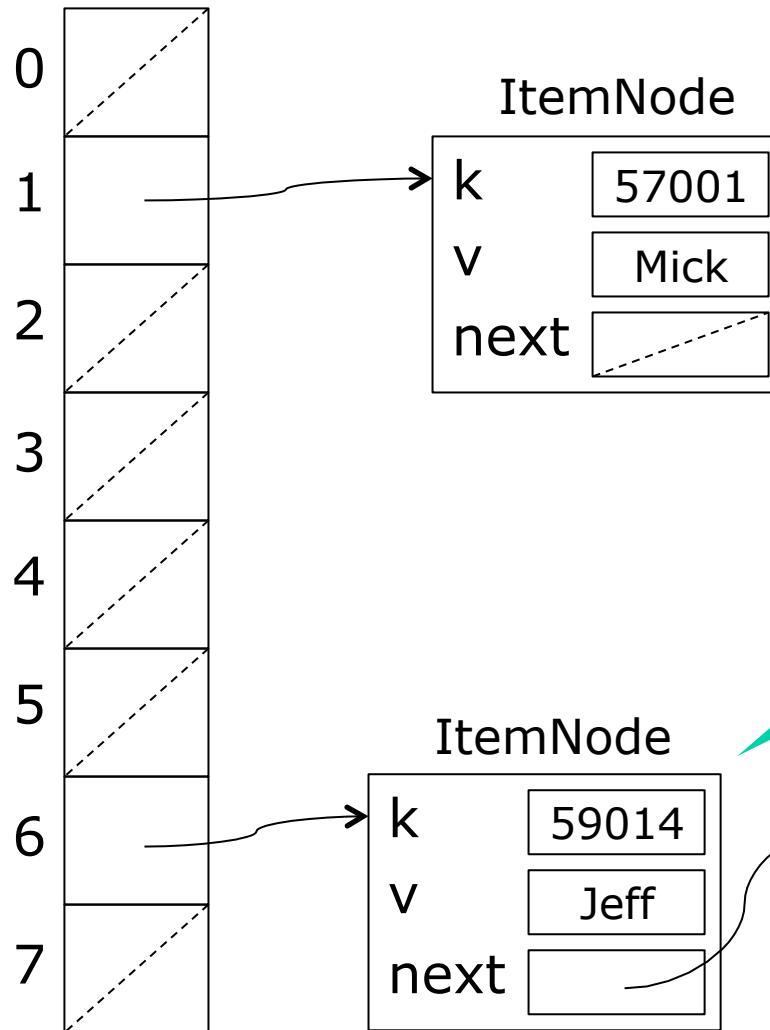


Example `get(10406)`
key k is 10406
`k.hashCode()` is 406
bucket $406 \% 8$ is 6



Insertion into the hashmap

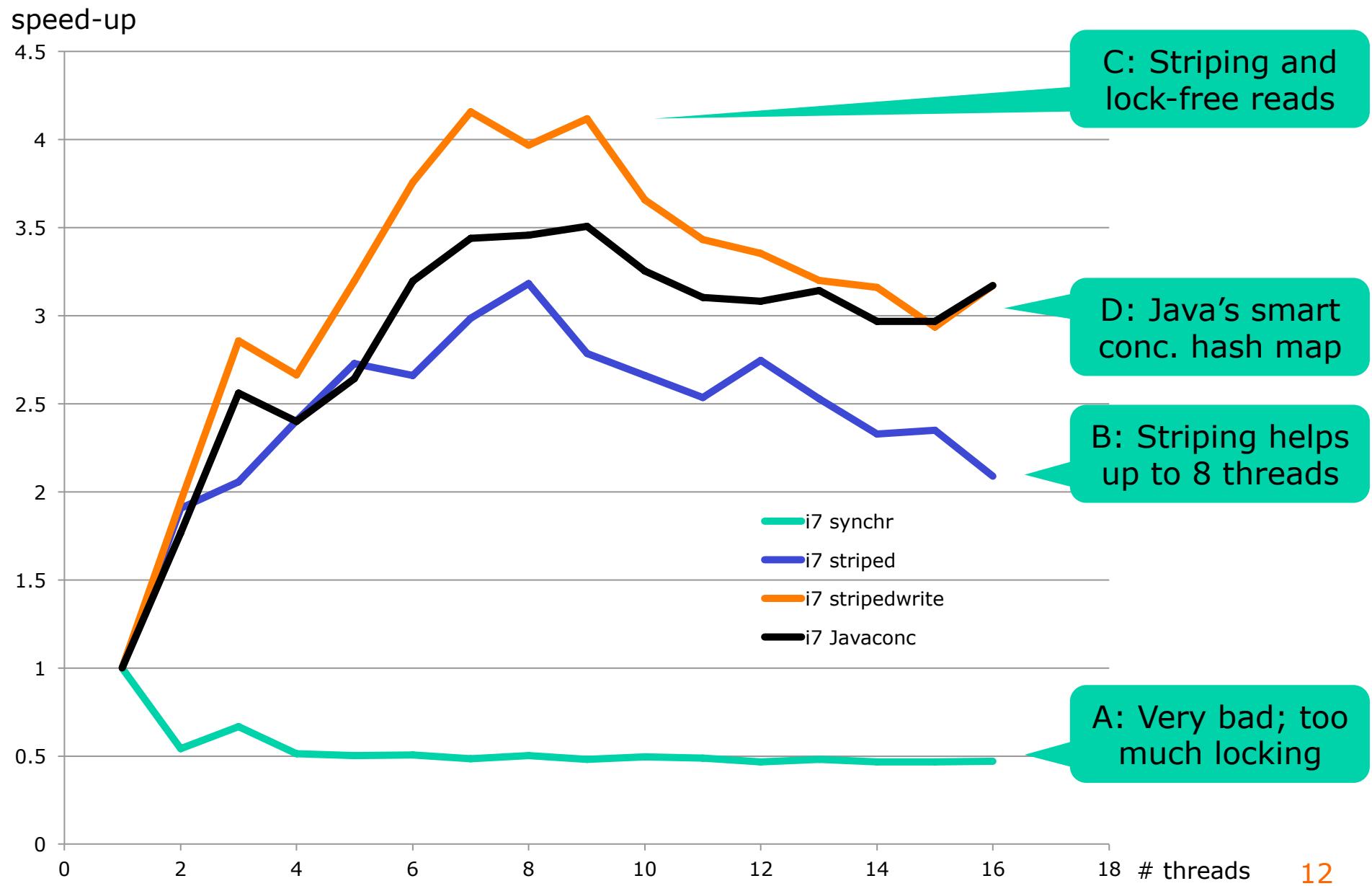
buckets



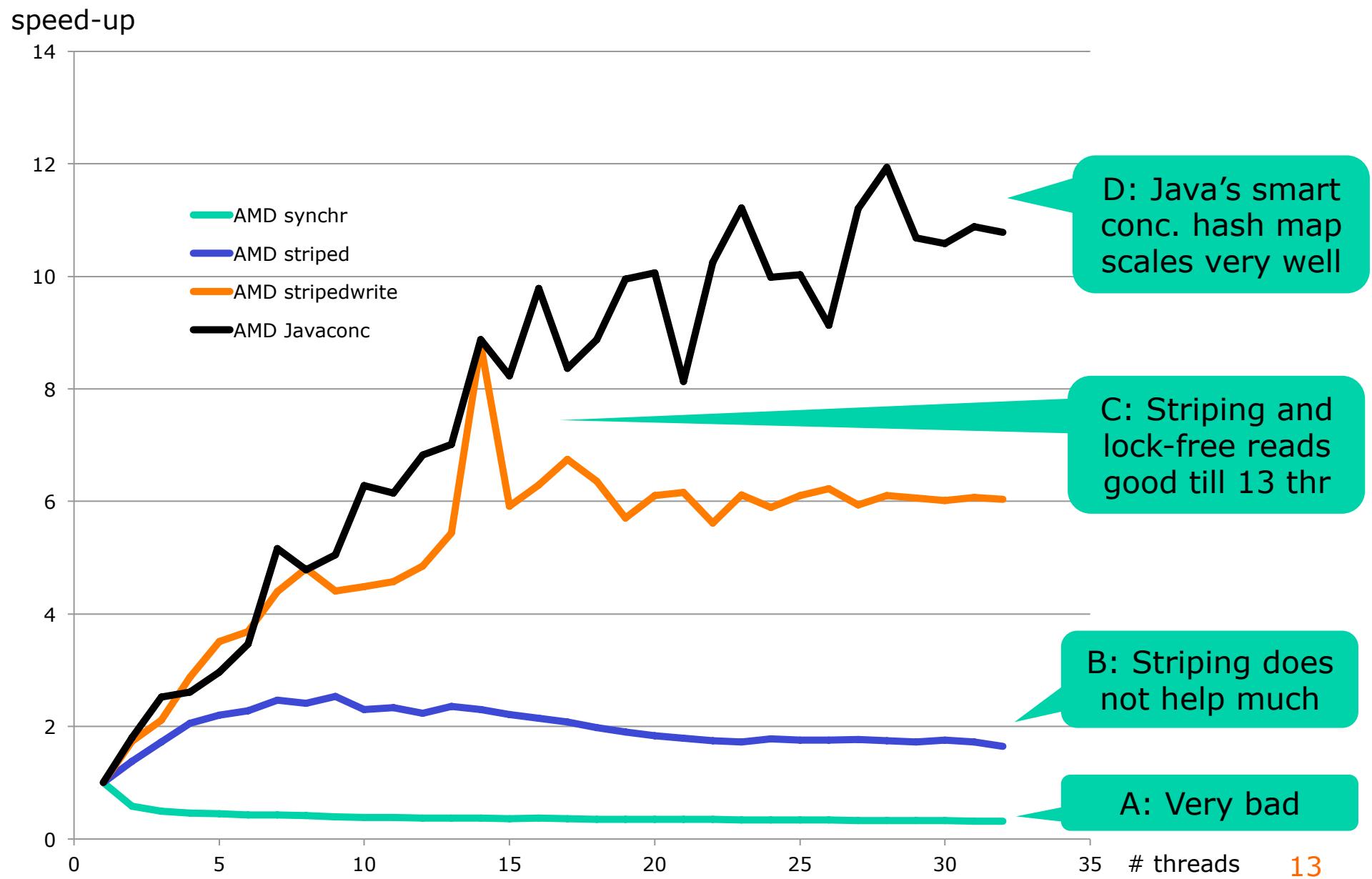
```
put(59014, "Jeff")  
key k is 59014  
k.hashCode() is 14  
bucket 406 % 8 is 6
```

Scalability of hash maps

Intel i7 w 4 cores & hyperthreading



Scalability of hash maps AMD Opteron w 32 cores



Our map interface

- Reduced version of Java's Map<K,V>

```
interface OurMap<K,V> {  
    boolean containsKey(K k);  
    V get(K k);  
    V put(K k, V v);  
    V putIfAbsent(K k, V v);  
    V remove(K k);  
    int size();  
    void forEach(Consumer<K,V> consumer);  
    void reallocateBuckets();  
}
```

```
interface Consumer<K,V> {  
    void accept(K k, V v);  
}
```

```
for (Entry (k,v) : map)  
    System.out.printf(...);
```

```
map.forEach((k, v) ->  
    System.out.printf("%10d maps to %s%n", k, v));
```

TestStripedMap.java

Synchronized map implementation

```
static class ItemNode<K,V> {  
    private final K k;  
    private V v;  
    private ItemNode<K,V> next; }  
    public ItemNode(K k, V v, ItemNode<K,V> next) { ... }  
}
```

Visibility depends
on synchronization

Java monitor
pattern

```
class SynchronizedMap<K,V> implements OurMap<K,V> {  
    private ItemNode<K,V>[] buckets; // guarded by this  
    private int cachedSize; // guarded by this  
    public synchronized V get(K k) { ... }  
    public synchronized boolean containsKey(K k) { ... }  
    public synchronized int size() { return cachedSize; }  
    public synchronized V put(K k, V v) { ... }  
    public synchronized V putIfAbsent(K k, V v) { ... }  
    public synchronized V remove(K k) { ... }  
    public synchronized void forEach(Consumer<K,V> consumer) { ... }  
}
```

Implementing containsKey

```
public synchronized boolean containsKey(K k) {
    final int h = k.hashCode(), hash = h % buckets.length;
    return ItemNode.search(buckets[hash], k) != null;
}
```

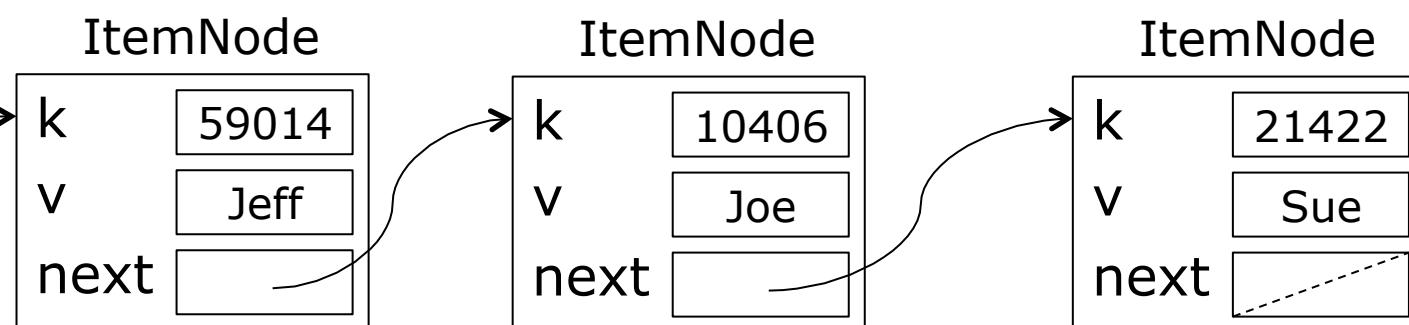
Find bucket

1
2

```
static <K,V> ItemNode<K,V> search(ItemNode<K,V> node, K k) {
    while (node != null && !k.equals(node.k))
        node = node.next;
    return node;
}
```

Search item
node list

3
4
5
6
7



Implementing putIfAbsent

```
public synchronized V putIfAbsent(K k, V v) {
    final int h = k.hashCode(), hash = h % buckets.length;
    ItemNode<K,V> node = ItemNode.search(buckets[hash], k);
    if (node != null)
        return node.v;
    else {
        buckets[hash] = new ItemNode<K,V>(k, v, buckets[hash]);
        cachedSize++;
        return null;
    }
}
```

Search
bucket's
node list

If key exists,
return value

Else add new
item node at
front of list

- All methods are synchronized
 - atomic access to buckets table and item nodes
 - all writes by put, putIfAbsent, remove, reallocateBuckets are visible to containsKey, get, size, forEach

Reallocating buckets

- Hash map efficiency requires short node lists
- When item node lists become too long, then
 - Double buckets array size to newCount
 - For each item node (k,v)
 - Recompute `newHash = k.hashCode() % newCount`
 - Link item node into new list at `newBuckets[newHash]`
- This is a dramatic operation
 - Must lock the entire data structure
 - Can happen at any insertion

ReallocateBuckets implementation

```

public synchronized void reallocateBuckets() {
    final ItemNode<K,V>[] newBuckets = makeBuckets(2 * buckets.length);
    for (int hash=0; hash<buckets.length; hash++) {
        ItemNode<K,V> node = buckets[hash];
        while (node != null) {
            final int newHash = node.k.hashCode() % newBuckets.length;
            ItemNode<K,V> next = node.next;
            node.next = newBuckets[newHash];
            newBuckets[newHash] = node;
            node = next;
        }
    }
    buckets = newBuckets;
}

```

For each item node

Compute new hash

Link into new bucket

- Seems efficient: reuses each ItemNode
 - Links it into a new item node list
 - So destroys the old item node list
 - So read access impossible during reallocation
 - Good 1-core performance, but bad scalability

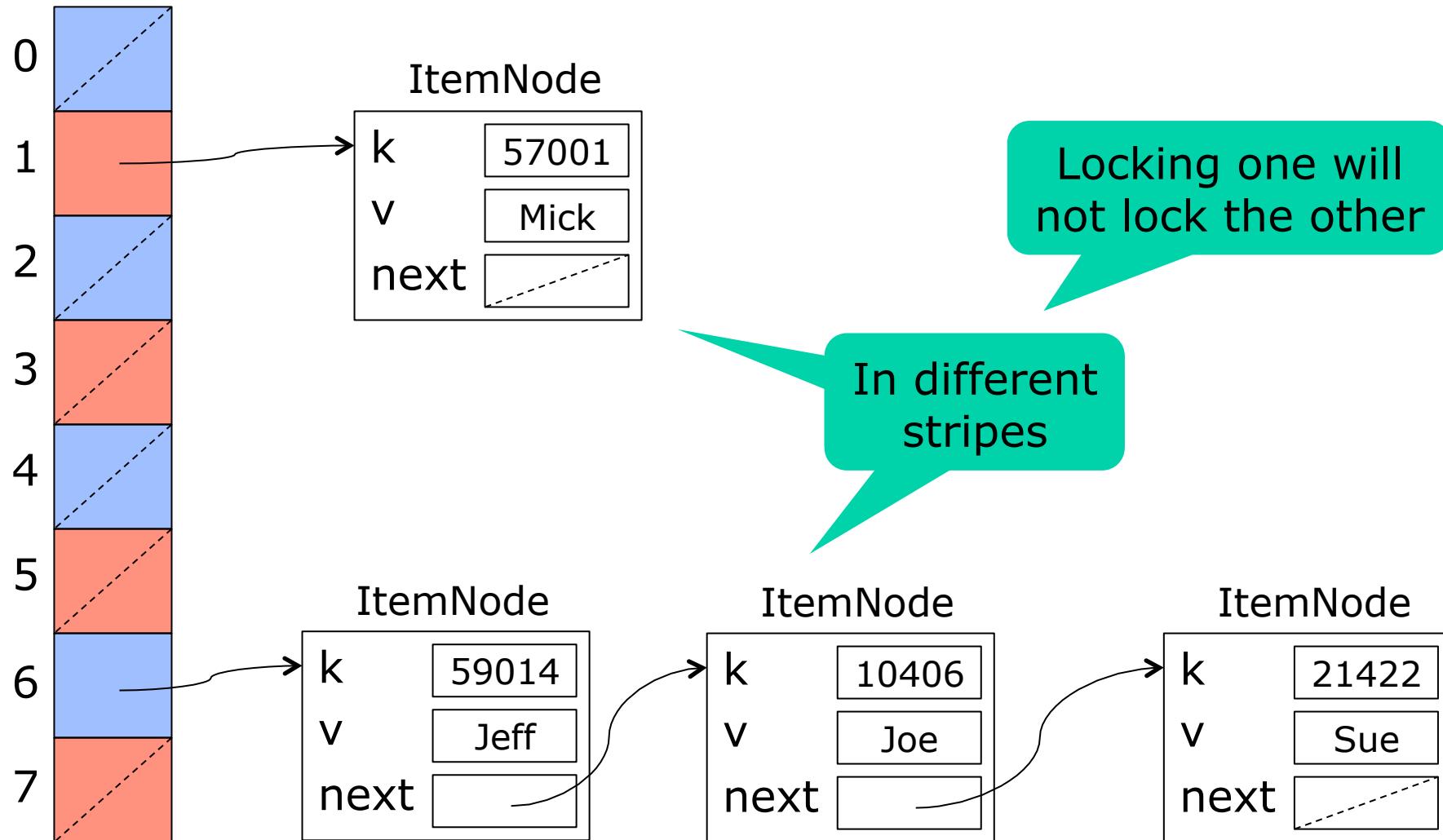
Better scalability: Lock striping

- Guarding the table with a single lock works
 - ... but does not scale well (actually **very** badly)
- Idea: Each bucket could have its own lock
- In practice
 - use a few, maybe 16, locks
 - guard every 16th bucket with the same lock
 - locks[0] guards bucket 0, 16, 32, ...
 - locks[1] guards bucket 1, 17, 33, ...
- With high probability
 - two operations will work on different stripes
 - hence will take different locks
- Less lock contention, better scalability

Lock striping in hash map

Two stripes 0 = blue and 1 = red

buckets



Striped hashmap implementation

NB!

```
class StripedMap<K,V> implements OurMap<K,V> {  
    private volatile ItemNode<K,V>[] buckets;  
    private final int lockCount;  
    private final Object[] locks;  
    private final int[] sizes;  
  
    public boolean containsKey(K k) { ... }  
    public V get(K k) { ... }  
    public int size() { ... }  
    public V put(K k, V v) { ... }  
    public V putIfAbsent(K k, V v) { ... }  
    public V remove(K k) { ... }  
    public void forEach(Consumer<K,V> consumer) { ... }  
}
```

Methods **not**
synchronized

- Synchronization on **lock[stripe]** ensures
 - atomic access within each stripe
 - visibility of writes to readers

Implementation of containsKey

```
public boolean containsKey(K k) {  
    final int h = k.hashCode(), stripe = h % lockCount;  
    synchronized (locks[stripe]) {  
        final int hash = h % buckets.length;  
        return ItemNode.search(buckets[hash], k) != null;  
    }  
}
```

TestStripedMap.java

- Compute key's hash code
 - Lock the relevant stripe
 - Compute hash index, access bucket
 - Search node item list
-
- What if buckets were reallocated between computing "stripe" and locking?

Representing hash map size

- Could use a single AtomicInteger **size**
 - might limit concurrency
- Instead use one **int** per stripe
 - read and write while holding the stripe's lock

```
public int size() {  
    int result = 0;  
    for (int stripe=0; stripe<lockCount; stripe++)  
        synchronized (locks[stripe]) {  
            result += sizes[stripe];  
        }  
    return result;  
}
```

- A stripe might be updated right after we read its size, before we return the sum
 - This is acceptable in concurrent data structures

Striped put(k,v)

```
public V put(K k, V v) {  
    final int h = k.hashCode(), stripe = h % lockCount;  
    synchronized (locks[stripe]) {  
        final int hash = h % buckets.length;  
        final ItemNode<K,V> node = ItemNode.search(buckets[hash], k);  
        if (node != null) {  
            V old = node.v;  
            node.v = v;  
            return old;  
        } else {  
            buckets[hash] = new ItemNode<K,V>(k, v, buckets[hash]);  
            sizes[stripe]++;  
            return null;  
        }  
    }  
}
```

Lock stripe

If k exists, update value to v, return old

And add 1 to stripe size

Else add new item node (k,v)

Reallocating buckets

- Must lock all stripes; how take **nlocks** locks?
 - Use recursion: each call takes one more lock

```
private void lockAllAndThen(Runnable action) {
    lockAllAndThen(0, action);
}

private void lockAllAndThen(int nextStripe, Runnable action) {
    if (nextStripe >= lockCount)
        action.run();
    else
        synchronized (locks[nextStripe]) {
            lockAllAndThen(nextStripe + 1, action);
        }
}
```

TestStripedMap.java

```
synchronized(locks[0]) {
    synchronized(locks[1]) {
        ...
        synchronized(locks[15]) {
            action.run();
        }
    }
}
```

Overall effect of calling
lockAllAndThen(0, action)

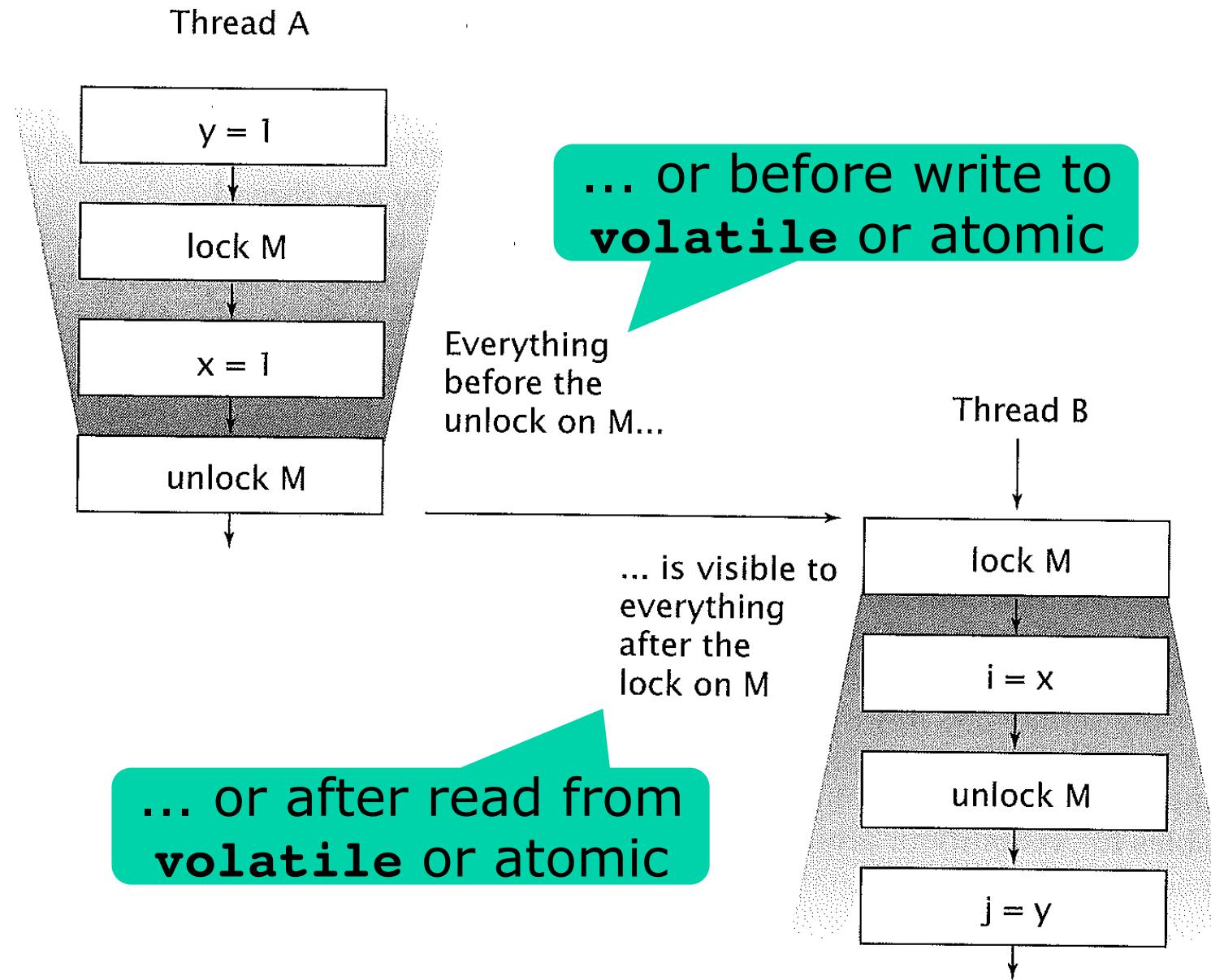
All locks held when
calling **action.run()**

Idea: Immutable item nodes

- We can make read access lock free
- Good if more reads than writes
- A *read* of key *k* consists of
 - Compute `hash = k.hashCode() % buckets.length`
 - Access `buckets[hash]` to get an item node list
 - Search the immutable item node list
- (1) Must make `buckets` access *atomic*
 - Get local reference: `final ImmutableList<ItemNode<K,V>>[] bs = buckets;`
- (2) No lock on reads, how make writes *visible*?
 - Represent stripe sizes using AtomicIntegerArray
 - A hash map write must write to stripe size, **last**
 - A hash map read must read from stripe size, **first**
 - Also, declare `buckets` field **volatile**

Must be atomic

Visibility by lock, volatile, or atomic



Locking the stripes only on write

```
class StripedWriteMap<K,V> implements OurMap<K,V> {
    private volatile ItemNode<K,V>[] buckets;
    private final int lockCount;
    private final Object[] locks;
    private final AtomicIntegerArray sizes;
    ... non-synchronized methods, signatures as in StripedMap<K,V>
}
```

TestStripedMap.java

```
static class ItemNode<K,V> {
    private final K k;
    private final V v;
    private final ItemNode<K,V> next;

    static boolean search(ItemNode<K,V> node, K k, Holder<V> old) ...
    static ItemNode<K,V> delete(ItemNode<K,V> node, K k, Holder<V> old) ...
}
```

Immutable

```
static class Holder<V> { // Not threadsafe
    private V value;
    public V get() { return value; }
    public void set(V value) { this.value = value; }
}
```



To hold "out" parameters

Lock-free ContainsKey

```
public boolean containsKey(K k) {
    final ItemNode<K,V>[] bs = buckets;
    final int h = k.hashCode(), stripe = h % lockCount,
        hash = h % bs.length;
    return sizes.get(stripe) != 0 && ItemNode.search(bs[hash], k, null);
}
```

Read volatile field, once ...

First read sizes, to make previous writes visible

... so that hash and array are consistent

TestStripedMap.java

- In class ItemNode, a plain linked list search:

```
static <K,V> boolean search(ItemNode<K,V> node, K k, Holder<V> old) {
    while (node != null)
        if (k.equals(node.k)) {
            if (old != null)
                old.set(node.v);
            return true;
        } else
            node = node.next;
    return false;
}
```

Item nodes are immutable and so threadsafe

If k found, may return v here

Stripe-locking put(k,v)

```

public V put(K k, V v) {
    final int h = k.hashCode(), stripe = h % lockCount;
    synchronized (locks[stripe]) {
        final ItemNode<K,V>[] bs = buckets;
        final int hash = h % bs.length;
        final Holder<V> old = new Holder<V>();
        final ItemNode<K,V> node = bs[hash],
            newNode = ItemNode.delete(node, k, old);
        bs[hash] = new ItemNode<K,V>(k, v, newNode);
        sizes.getAndAdd(stripe, newNode == node ? 1 : 0);
        return old.get();
    }
}

```

pedMap.java

- To **put(k, v)**
 - Delete existing entry for **k**, if any
 - This may produce a new list of item nodes (immutable!)
 - Add new **(k, v)** entry at head of item node list
 - Update stripe size, *also* for visibility

StripedWriteMap in perspective

- StripedWriteMap design
 - incorporates ideas from Java's ConcurrentHashMap
 - yet is much simpler (Java's uses optimistic concurrency, compare-and-swap, week 11-12)
 - but also less scalable
- Is it correct?
 - I think so ...
 - Various early versions weren't ☹
- Can we test it?
 - We can see if we can break it, week 9
 - Too subtle for ThreadSafe tool (visibility)?

Why is coarse locking so expensive?

- Limited concurrency
 - In SynchMap only 1 thread can work at a time
 - Hence 3, or 31, CPU cores may sit idle
- Increased thread scheduling overhead
 - If lock unavailable, the thread moves to Locking, then to Enabled, then to Running
 - *Context switch* is slow, also causes cache misses
- Atomic operations may be slow on multicore
 - ... and lock taking requires an atomic operation
 - Clearly worse on AMD Opteron than on Intel i7

Goetz p. 229

Plan for today

- Performance and scalability
- Reduce lock duration, use lock splitting
- Hash maps, a scalability case study
 - (A) Hash map à la Java monitor
 - (B) Hash map with lock striping
 - (C) Ditto with lock striping and non-blocking reads
- **An atomic long with “thread striping”**
- Shared mutable state is slow on multicore

A striped thread-safe long

- Use case: more writes (`add`) than reads (`get`)
- Vastly different scalability
 - (a) Java 5's `AtomicLong`
 - (b) Java 8's `LongAdder`
 - (c) Home-made synchronized `LongCounter`
 - (d) Home-made striped long using `AtomicLongArray`
 - (e) Home-made striped long with scattered allocation
- Ideas
 - (d,e) Use thread's `hashCode` to reduce update collisions
 - (e) Scatter `AtomicLongs` to avoid false cache line sharing

TestLongAdders.java

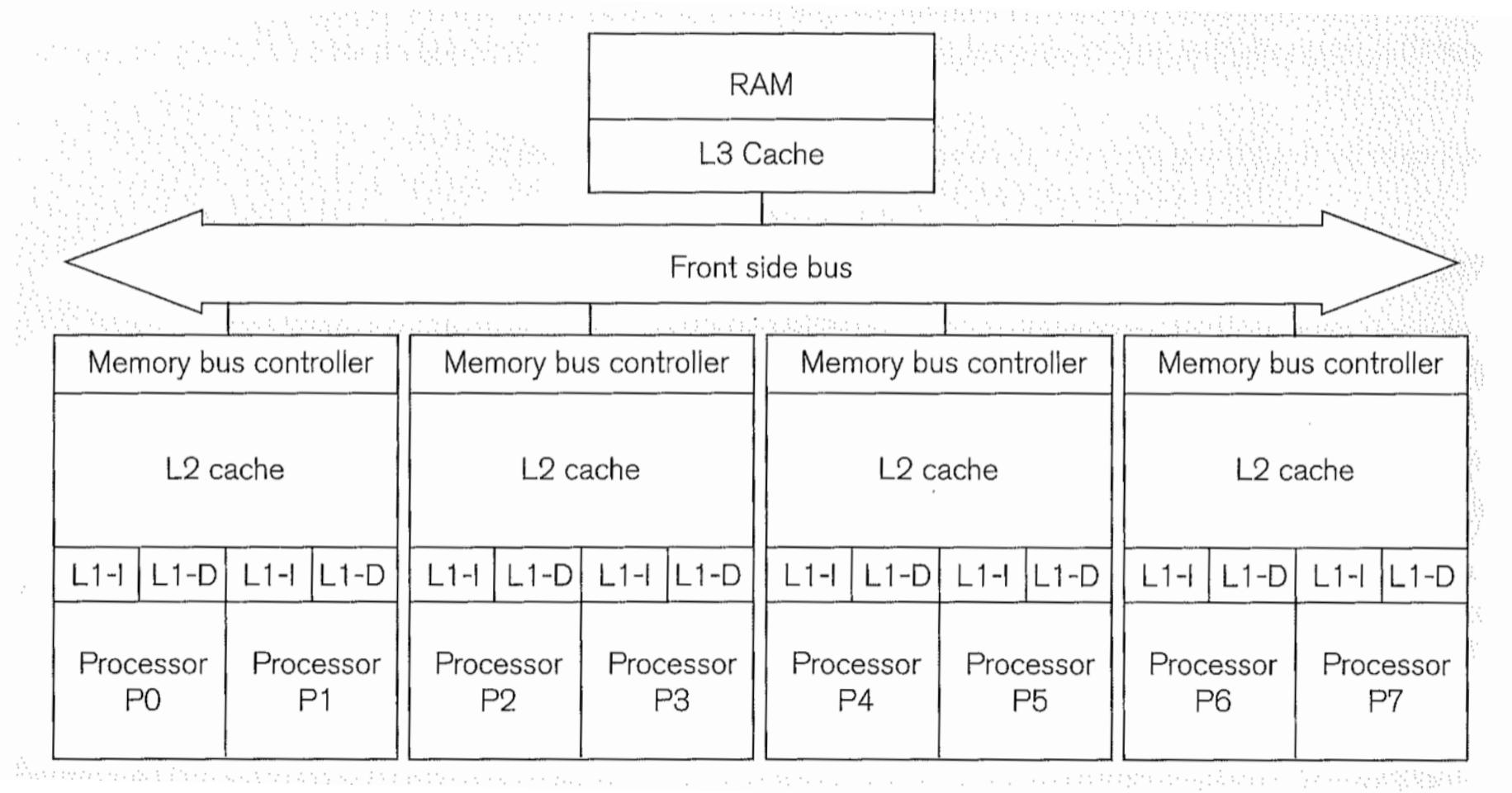
	i7 4c	AMD 32c
(a)	942	3011
(b)	65	54
(c)	1450	14921
(d)	427	1611
(e)	108	922

Wall clock time (ms) for 32 threads making 1 million additions each

Plan for today

- Performance and scalability
- Reduce lock duration, use lock splitting
- Hash maps, a scalability case study
 - (A) Hash map à la Java monitor
 - (B) Hash map with lock striping
 - (C) Ditto with lock striping and non-blocking reads
- An atomic long with “thread striping”
- **Shared mutable state is slow on multicore**

A typical multicore CPU with three levels of cache

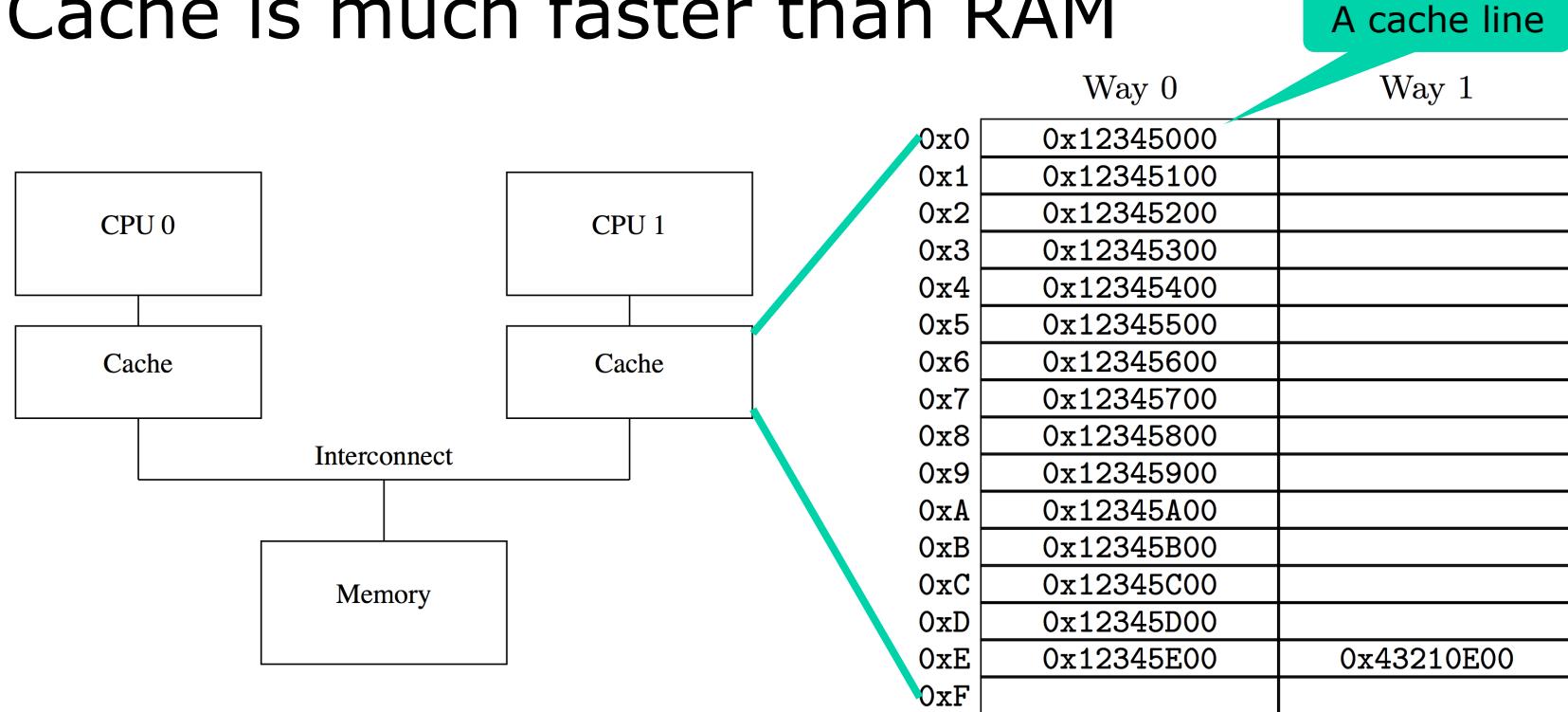


Lin & Snyder 2009, p. 16

- Floating-point add or mul: 0.4 ns
- RAM access: > 100 ns

Fix 1: Each processor core has a cache

- Cache = simple hardware hashtable
- Stores recently accessed values from RAM
- Cache is much faster than RAM



McKenney 2010: Memory barriers

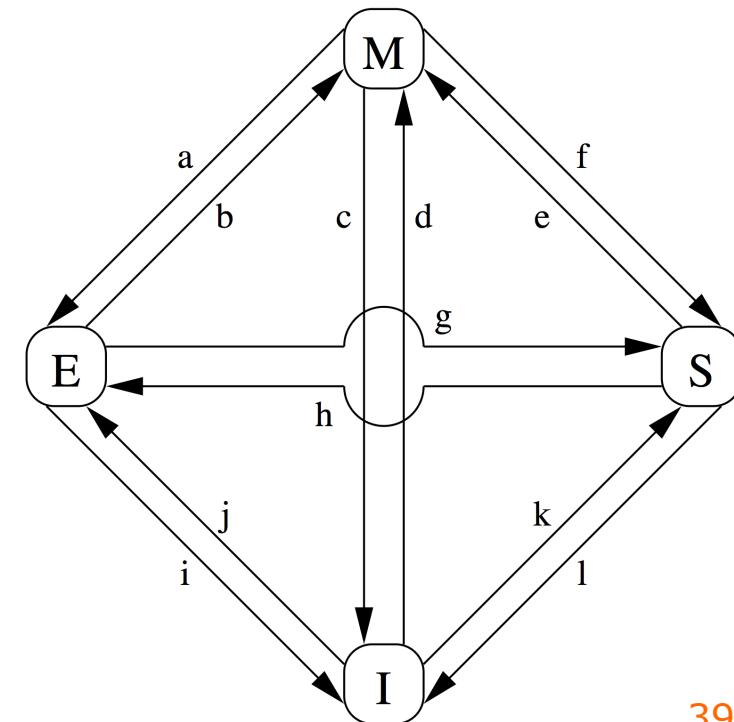
- Two caches may have different values for a given memory address

Fix 2: Get all caches to agree

- Cache coherence; cache line state = M,E,S,I

State	Cache line	Excl	RAM	Read	Write
Modified	Modified by me	Y	not OK	from cache	to cache
Exclusive	Not modified	Y	OK	from cache	to cache -> M
Shared	Others have it	N	OK	from cache	send invalidate
Invalid	Not in use	-	-	elsewhere	send invalidate

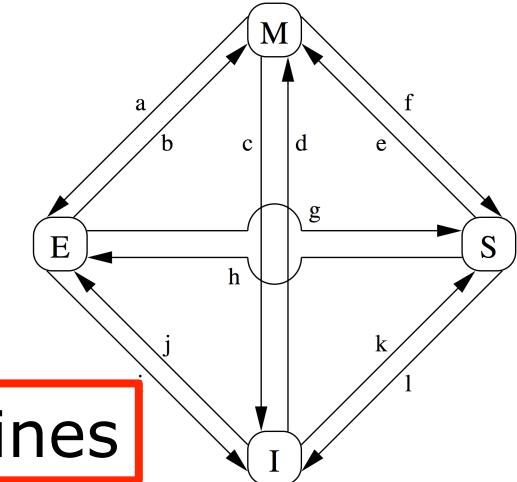
- A cache line
 - has 4 states
 - and 12 transitions a-l
- Cache messages
 - sent by cores to others
 - via memory bus
 - to make caches agree



Transitions and messages

A write in a non-exclusive state requires acknowledge ack* from *all other cores*

Shared mutable state is slow on big machines



		Cause	I send	I receive	My response
M	a	(Send update to RAM)	writeback	-	-
E	b	Write	-	-	-
M	c	Other wants to write	-	read inv	read resp, inv ack
I	d	Atomic read-mod-write	read inv	read resp, inv ack*	-
S	e	Atomic read-mod-write	read inv	inv ack*	-
M	f	Other wants to read	-	read	read resp
E	g	Other wants to read	-	read	read resp
S	h	Will soon write	inv	inv ack*	-
E	i	Other wants atomic rw	-	read inv	read resp, inv ack
I	j	Want to write	read inv	read resp, inv ack*	-
I	k	Want to read	read	read resp	-
S	l	Other wants to write	-	inv	inv ack

One more performance problem: “false sharing” because of cache lines

- A cache line typically is 32 bytes
 - gives better memory bus utilization
 - prefetches data (in array) that may be needed next
- Thus invalidating one (8 byte) long may invalidate the neighboring 3 longs in an array
- Frequently written memory locations should not be on the same cache line

```
for (int stripe=0; stripe<NSTripes; stripe++) {  
    // Believe it or not, this may speed up the code,  
    // presumably because it avoids false sharing:  
    new Object(); new Object(); new Object(); new Object();  
    counters[stripe] = new AtomicLong();  
}
```

This week

- Reading
 - Goetz et al chapter 11 + 13.5
 - Optional: McKenney: *Memory barriers*
- Exercises
 - Make sure you can write well-performing and scalable software using lock striping, immutability, Java atomics, and visibility rules; finish StripedMap and StripedWriteMap classes
- Read before next lecture (24 October)
 - Goetz et al chapter 9

Practical Concurrent and Parallel Programming 8

Peter Sestoft
IT University of Copenhagen

Friday 2014-10-24

Plan for today

- Graphical user interface toolkits, eg Swing
 - not thread-safe, access from event thread only
- Using SwingWorker for long-running work
 - Progress bar
 - Cancellation
 - Display results as they are generated
- A thread-based lift simulator with GUI
- Atomic long with “thread striping” (week 7)
- Shared mutable data on multicore is slow

GUI toolkits are single-threaded

- Java Swing components are **not** thread-safe
 - This is intentional
 - Ditto .NET's System.Windows.Forms and others
- Multithreaded GUI toolkits
 - are difficult to use
 - deadlock-prone, because actions are initiated both
 - *top-down*: from user towards operating system
 - *bottom-up*: from operating system to user interface
 - locking in different orders ... hence deadlock risk
- In Swing, at least two threads:
 - Main Thread – runs `main(String[] args)`
 - Event Thread – runs ActionListeners and so on

From Graham Hamilton's blog post

"Multithreaded toolkits: A failed dream?"

- *"In general, GUI operations start at the top of a stack of library abstractions and go "down". I am operating on an abstract idea in my application that is expressed by some GUI objects, so I start off in my application and call into high-level GUI abstractions, that call into lower level GUI abstractions, that call into the ugly guts of the toolkit, and thence into the OS."*
- *In contrast, input events start off at the OS layer and are progressively dispatched "up" the abstraction layers, until they arrive in my application code.*
- *Now, since we are using abstractions, we will naturally be doing locking separately within each abstraction.*
- *And unfortunately we have the classic lock ordering nightmare: we have two different kinds of activities going on that want to acquire locks in opposite orders. So deadlock is almost inevitable."* (19 October 2004)

https://weblogs.java.net/blog/kgh/archive/2004/10/multithreaded_t.html

Java Swing GUI toolkit dogmas

- Dogma 1: “Time-consuming tasks should **not** be run on the Event Thread”
 - Otherwise the application becomes unresponsive
- Dogma 2: “Swing components should be accessed on the Event Thread only”
 - The components are not thread-safe
- But if another thread does long-running work, how can it show the results on the GUI?
 - Define the work in SwingWorker subclass instance
 - Use `execute()` to run it on a worker thread
 - The Event Thread can pick up the results

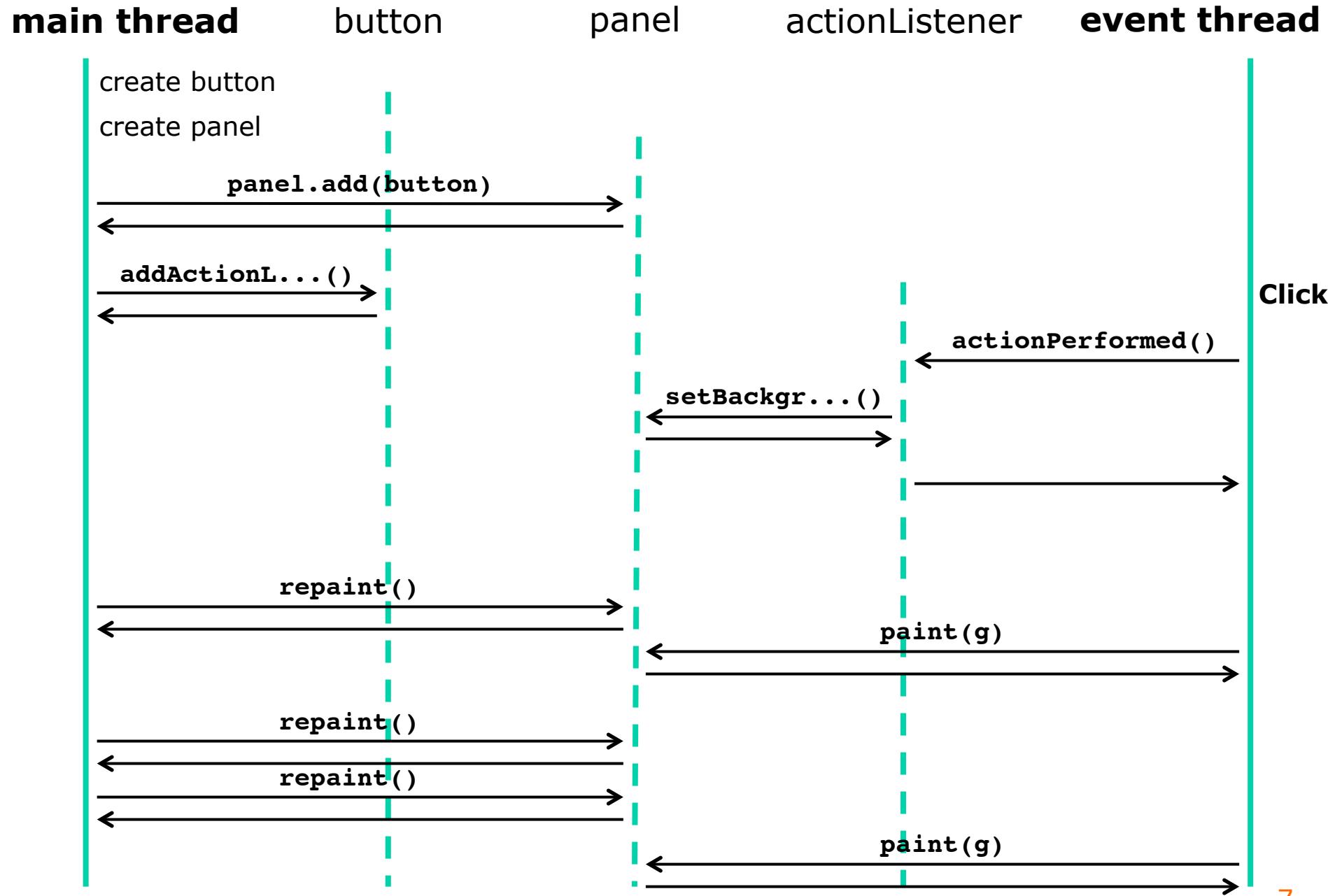
A short computation on the event thread

```
final JFrame frame = new JFrame("TestButtonGui") ;  
final JPanel panel = new JPanel() ;  
final JButton button = new JButton("Press here") ;  
frame.add(panel) ;  
panel.add(button) ;  
button.addActionListener(new ActionListener() {  
    public void actionPerformed(ActionEvent e) {  
        panel.setBackground(new Color(random.nextInt())) ;  
    } }) ;  
frame.pack() ; frame.setVisible(true) ;
```

TestButtonGui.java

- Main thread may create GUI components
 - But should not change eg. background color later
- Event thread calls the ActionListener
 - And can change the background color

Main thread and event thread



Using the main thread for blinking

```
final JPanel panel = new JPanel() {
    public void paint(Graphics g) {
        super.paint(g);
        if (showBar) {
            g.setColor(Color.RED);
            g.fillRect(0, 0, 10, getHeight());
        }
    }
}
final JButton button = ...
frame.pack(); frame.setVisible(true);
while (true) {
    try { Thread.sleep(800); } // milliseconds
    catch (InterruptedException exn) { }
    showBar = !showBar;
    panel.repaint();
}
```

- **repaint()** may be called by any thread
- Causes event thread to call **repaint(g)** later

Fetching webpages on event thread

```
fetchButton.addActionListener(new ActionListener() {  
    public void actionPerformed(ActionEvent e) {  
        for (String url : urls) {  
            System.out.println("Fetching " + url);  
            String page = getPage(url, 200);  
            textArea.append(String.format(..., url, page.length()));  
        }  
    }  
});
```

On event thread

Bad

TestFetchWebGui.java

- Occupies event thread for many seconds
 - The GUI is unresponsive in the meantime
 - Results not shown as they become available
 - GUI gets updated only after all fetches
 - Cancellation would not work
 - Cancel button event processed only after all fetches
 - A progress bar would not work
 - Gets updated only after all fetches

Fetching web with SwingWorker

```

static class DownloadWorker extends SwingWorker<String, String> {
    private final TextArea textArea;
    public String doInBackground() {
        StringBuilder sb = new StringBuilder();
        for (String url : urls) {
            String page = getPage(url, 200),
                result = String.format("%-40s%7d%n", url, page.length());
            sb.append(result);
        }
        return sb.toString();
    }
    public void done() {
        try { textArea.append(get()); } Get result
        catch (InterruptedException exn) { }
        catch (ExecutionException exn) { throw new RuntimeException...; }
    }
}

```

TestFetchWebGui.java

- SwingWorker<T,V> implements Future<T>
- .NET has System.ComponentModel.BackgroundWorker

Fetching web with SwingWorker

```
DownloadWorker downloadTask = new DownloadWorker(textArea);
fetchButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        downloadTask.execute();
    }
});
```

TestFetchWebGui.java

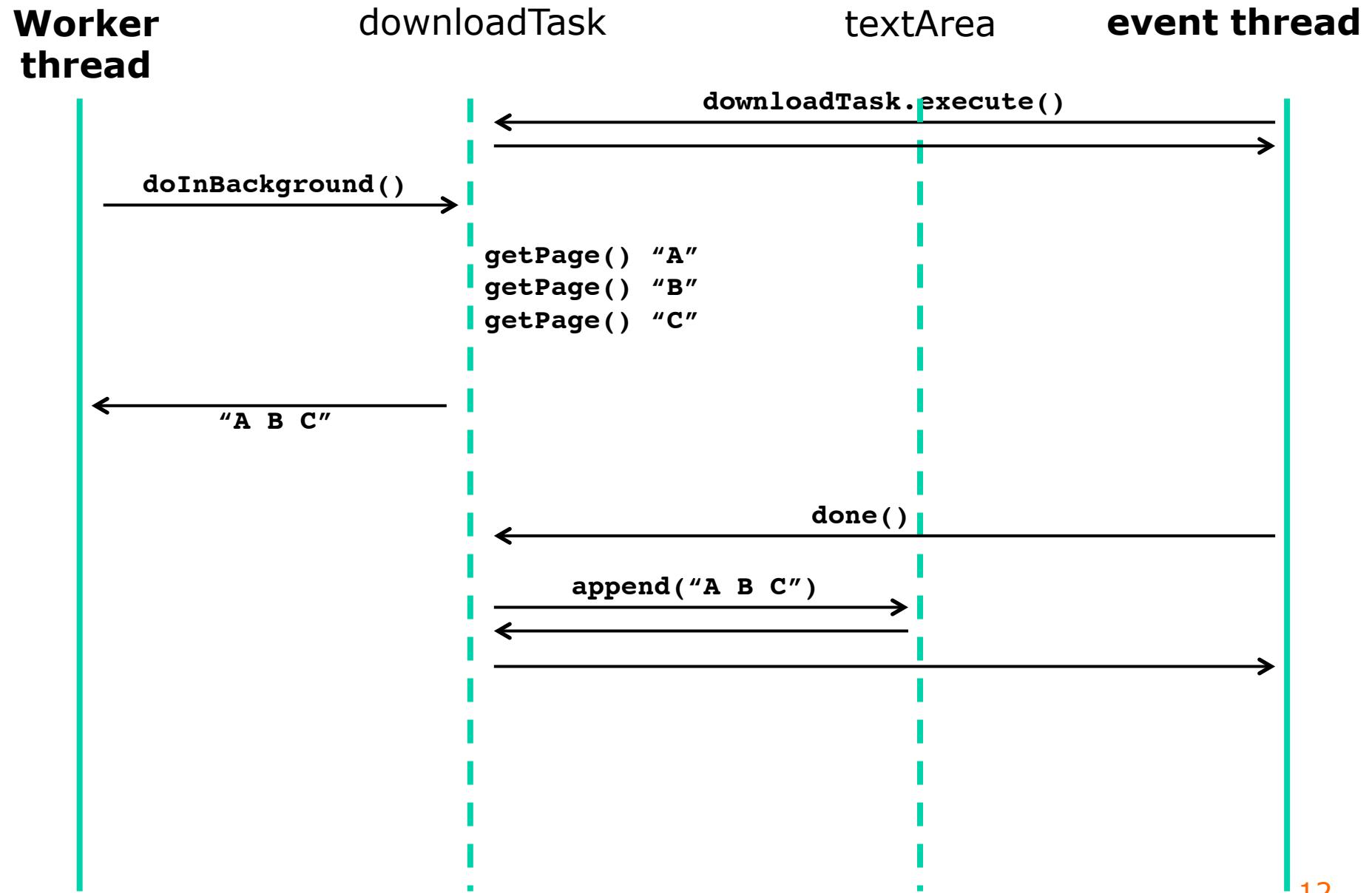
- Event thread runs **execute()**
- Worker thread runs **doInBackground()**
 - which returns the full result when computed
- Event thread runs **done()**
 - obtains the already-computed result with **get()**
 - and writes the result to the **textArea**

Dogma 1

Dogma 2

Worker thread and event thread

W 1



12

Add progress notification

```
static class DownloadWorker extends SwingWorker<String, String> {
    public String doInBackground() {
        int count = 0;
        StringBuilder sb = new StringBuilder();
        for (String url : urls) {
            String page = getPage(url, 200),
                result = String.format("%-40s%7d%n", url, page.length());
            sb.append(result);
            setProgress((100 * ++count) / urls.length());
        }
        return sb.toString();
    }
}
```

On worker
thread

- In the GUI setup, add:

```
downloadTask.addPropertyChangeListener(new PropertyChangeListener() {
    public void propertyChange(PropertyChangeEvent e) {
        if ("progress".equals(e.getPropertyName())) {
            progressBar.setValue((Integer)e.getNewValue());
        }
    }
});
```

On event
thread

Add cancellation

```
static class DownloadWorker extends SwingWorker<String, String> {  
    public String doInBackground() {  
        for (String url : urls) {  
            if (isCancelled())  
                break;  
            ...  
            sb.append(result);  
        }  
        return sb.toString();  
    }  
  
    public void done() {  
        try { textArea.append(get()); }  
        catch (InterruptedException exn) { }  
        catch (ExecutionException exn) { throw new RuntimeException...; }  
        catch (CancellationException exn) { textArea.append("Yrk"); }  
    } }  
}
```

On worker
thread

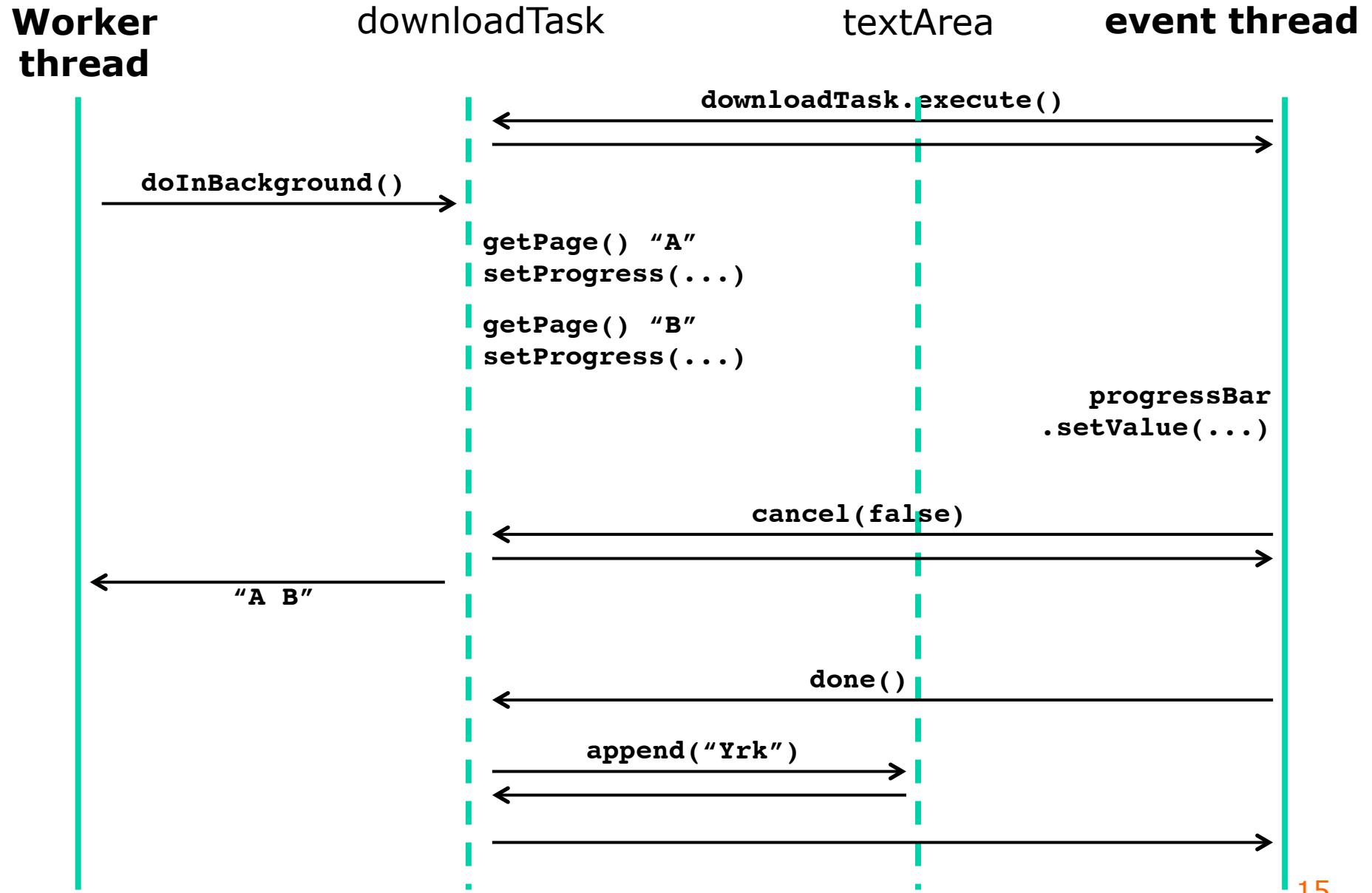
- In the GUI setup, add:

```
cancelButton.addActionListener(new ActionListener() {  
    public void actionPerformed(ActionEvent e) {  
        downloadTask.cancel(false);  
    } });
```

On event
thread

Progress and cancellation

W 1



Show results gradually

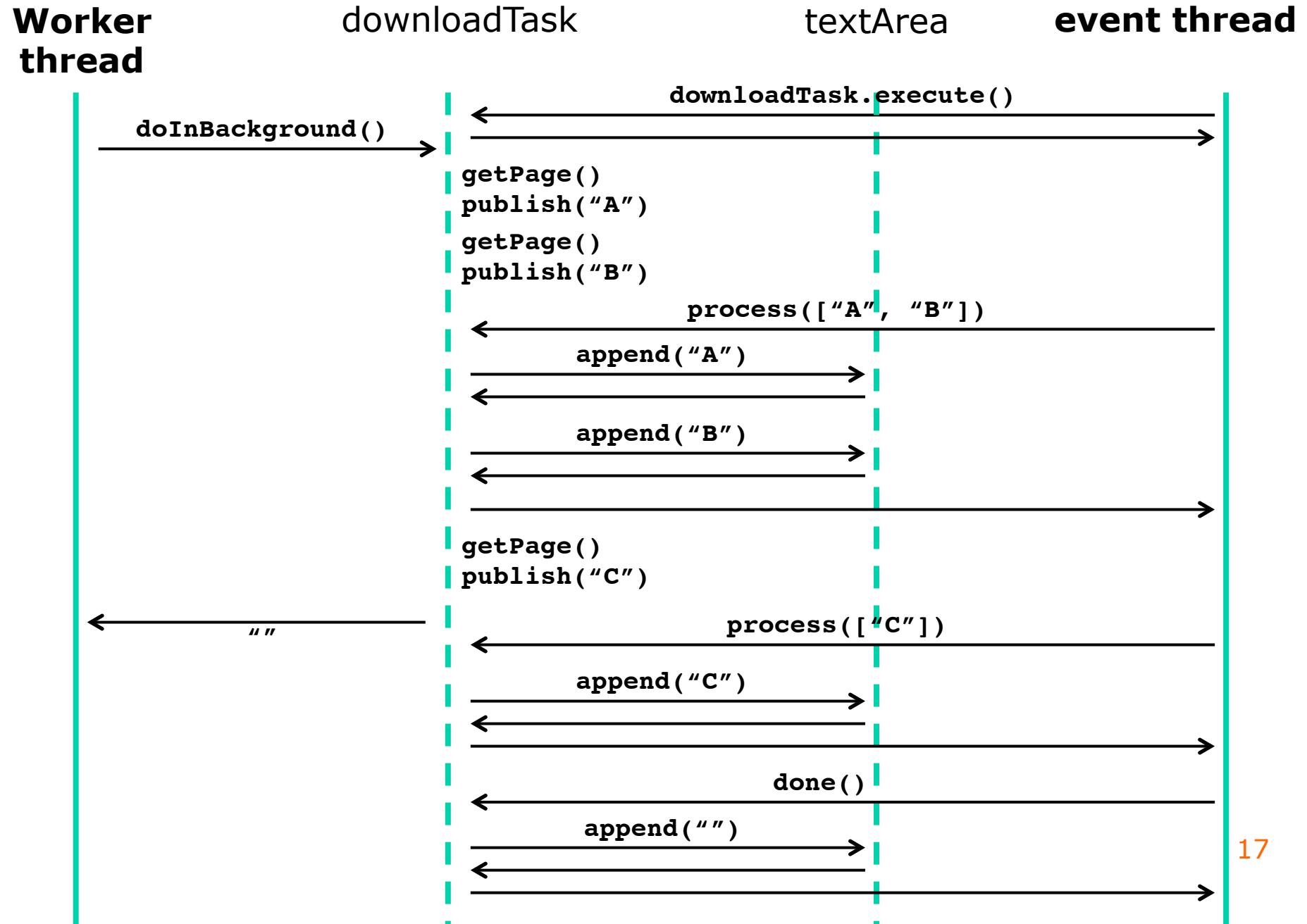
```
static class DownloadWorker extends SwingWorker<String, String> {  
    public String doInBackground() {  
        for (String url : urls) {  
            String page = getPage(url, 200),  
                result = String.format("%-40s%7d%n", url, page.length());  
            publish(result);  
        }  
    }  
  
    public void process(List<String> results) {  
        for (String result : results)  
            textArea.append(result);  
    }  
}
```

On worker
thread

On event
thread

- Worker thread calls **publish(...)** a few times
- Event thread calls **process** with results from calls to **publish** since last call to **process**

Event thread and downloadTask



SwingUtilities static methods

- May be called from any thread:
 - **boolean isEventDispatchThread()**
 - True if executing thread is the Event Thread
 - **void invokeLater(Runnable cmd)**
 - Execute `cmd.run()` asynchronously on the Event Thread
 - **void invokeAndWait(Runnable command)**
 - Execute `cmd.run()` on the Event Thread, wait to complete
- SwingWorker = these + Java executors
 - Goetz Listings 9.2 and 9.7 indicate how
- Other methods that any thread may call:
 - adding and removing listeners on components
 - but the listeners are *called* only on the Event Thread
 - **comp.repaint()** and **comp.revalidate()**

Very proper GUI creation in Swing

as per <http://docs.oracle.com/javase/tutorial/uiswing/concurrency/initial.html>

```
public static void main(String[] args) {  
    SwingUtilities.invokeLater(new Runnable() {  
        public void run() {  
            final Random random = new Random();  
            final JFrame frame = new JFrame("TestButtonGui");  
            final JPanel panel = new JPanel();  
            final JButton button = new JButton("Press here");  
            frame.add(panel);  
            panel.add(button);  
            button.addActionListener(new ActionListener() {  
                public void actionPerformed(ActionEvent e) {  
                    panel.setBackground(new Color(random.nextInt()));  
                }  
            });  
            frame.pack(); frame.setVisible(true);  
        }  
    });  
}
```

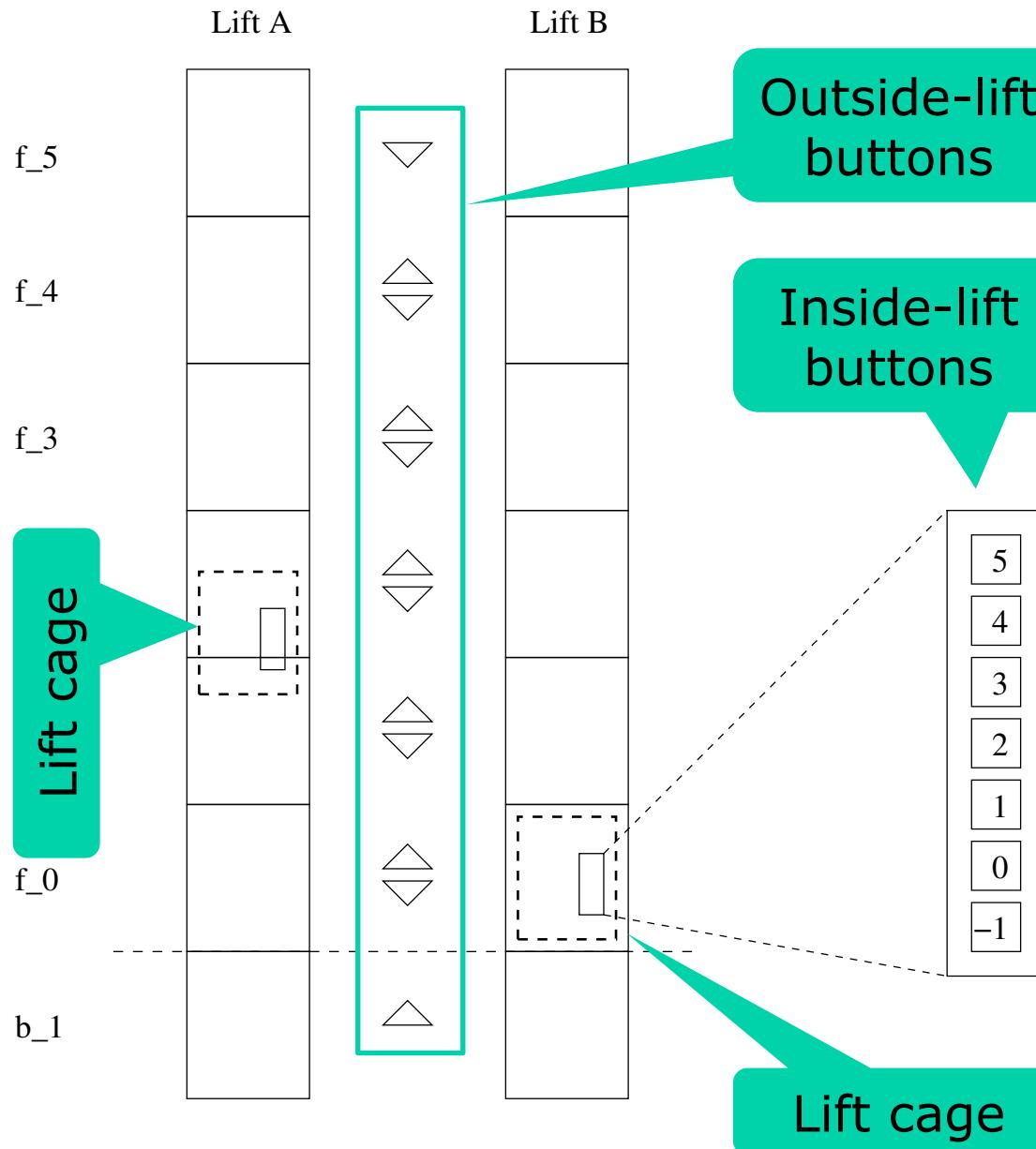
GUI gets built on
the Event Thread

- Avoids interaction with a partially constructed GUI
 - because the Event Thread is busy constructing the GUI

Plan for today

- Graphical user interface toolkits, eg Swing
 - not thread-safe, access from event thread only
- Using SwingWorker for long-running work
 - Progress bar
 - Cancellation
 - Display results as they are generated
- **A thread-based lift simulator with GUI**
- Atomic long with “thread striping” (week 7)
- Shared mutable data on multicore is slow

Example: 2 lifts, 7 floors, 26 buttons



Outside-lift
buttons

Inside-lift
buttons

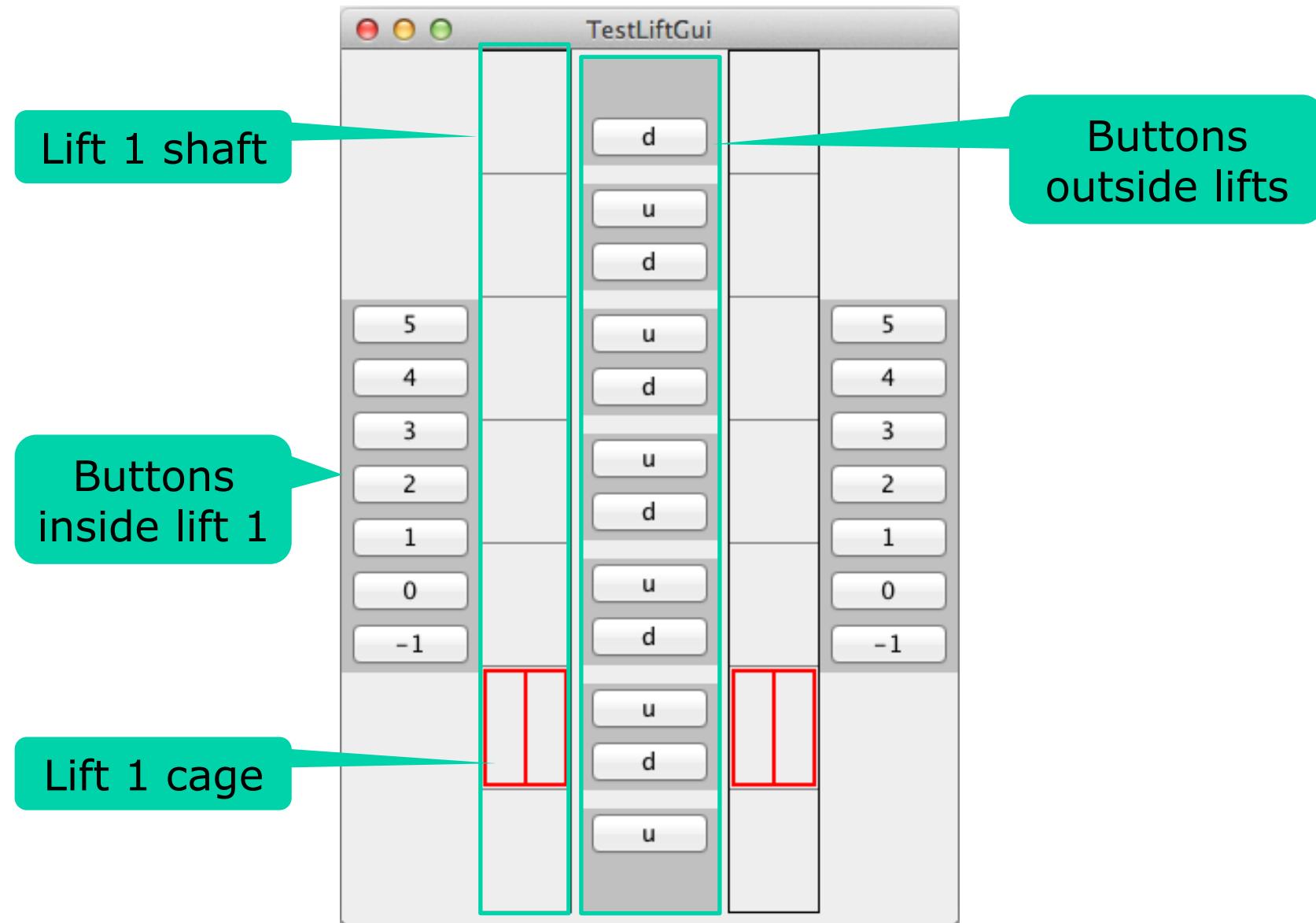
Concurrency:
2 lift cages move
26 buttons pressed

Two lift threads +
the event thread

Modeling and visualizing the lifts

- Use event thread for buttons (obviously)
 - Inside requests: *this lift* must go to floor n
 - Outside requests: *some lift* must go to floor n, and then up (or down)
- An object for each lift
 - to hold current floor, and floors yet to be visited
 - to compute time to serve an outside request
- A thread for each lift
 - to update its state 16 times a second
 - to cause the GUI to display it
- A controller object
 - to decide which lift should serve an outside request

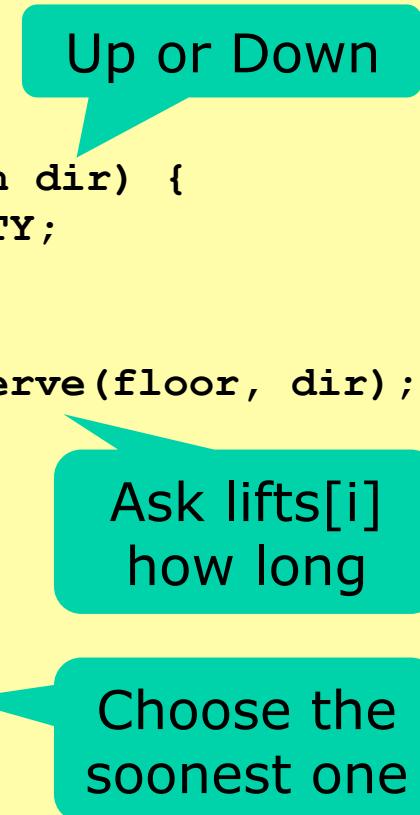
The lift simulator GUI



Lift controller algorithm

- When outside button Up on floor n is pressed
 - Ask each lift how long it would take to get to floor n while continuing up afterwards
 - Then order the fastest lift to serve floor n

```
class LiftController {  
    private final Lift[] lifts;  
    ...  
    public void someLiftTo(int floor, Direction dir) {  
        double bestTime = Double.POSITIVE_INFINITY;  
        int bestLift = -1;  
        for (int i=0; i<lifts.length; i++) {  
            double thisLiftTime = lifts[i].timeToServe(floor, dir);  
            if (thisLiftTime < bestTime) {  
                bestTime = thisLiftTime;  
                bestLift = i;  
            }  
        }  
        lifts[bestLift].customerAt(floor, dir);  
    }  
}
```



The state of a lift

- Current floor and direction (None, Up, Down)
- required stops and directions, **stops[floor]**:
 - **null**: no need to stop at this floor
 - **None**: stop, don't know future direction
 - **Down**: stop, then continue down
 - **Up**: stop, then continue up
 - **Both**: stop, then up (down); stop, then down (up)

```
class Lift implements Runnable {  
    private double floor;  
    private Direction direction; // None or Up or Down  
    // @GuardedBy("this")  
    private final Direction[] stops; // Accessed only on lift thread  
    ...  
    public synchronized void customerAt(int floor, Direction thenDir) {  
        setStop(floor, thenDir.add(getStop(floor)));  
    }  
}
```

Called by controller

TestLiftGui.java

The lift's behavior when going Up

- If at a floor, check whether to stop here
 - If so, open+close doors and clear from **stops** table
- If not yet at highest requested stop
 - move up a bit and refresh display
 - otherwise stop moving

```
switch (direction) {  
case Up:  
    if ((int)floor == floor) { // At a floor, maybe stop here  
        Direction afterStop = getStop((int)floor);  
        if (afterStop != null && (afterStop != Down || (int)floor == highestStop())) {  
            openAndCloseDoors();  
            subtractFromStop((int)floor, direction);  
        }  
    }  
    if (floor < highestStop()) {  
        floor += direction.delta / steps;  
        shaft.moveTo(floor, 0.0);  
    } else  
        direction = Direction.None;  
    break;  
case Down: ... similar to Up ...  
case None: ... if any stops[floor] != null, start moving in that direction ...  
}
```

Executed 16 times/second

TestLiftGui.java

Lift GUI thread safety

- Dogma 1, no long-running on event thread:
 - `sleep()` happens on lift threads, not event thread
- Dogma 2, only event thread works on GUI:
 - Lift thread calls `shaft.moveTo`,
 - which calls `repaint()`,
 - so event thread calls `paint(g)`, OK
- Lift and event threads access `stops[]` array
 - guarded by lift instance `this`
- Only lift thread accesses `floor` and `direction`
 - not guarded

Lift modeling reflection

- Seems reasonable to have a thread per lift
 - because they move concurrently
- Why not a thread for the controller?
 - because activated only by the external buttons
 - but what about supervising the lifts? e.g. if the lift sent to floor 4 going Up gets stuck at floor 3 by some fool with a lot of boxes?
- In Erlang, with message-passing, use
 - a “process” (task) for each lift
 - a “process” (task) for each floor, a “local controller”
 - no central controller
- Also Akka library, week 13-14

Armstrong et al: Concurrent Programming in Erlang (1993) 11.1

Plan for today

- Graphical user interface toolkits, eg Swing
 - not thread-safe, access from event thread only
- Using SwingWorker for long-running work
 - Progress bar
 - Cancellation
 - Display results as they are generated
- A thread-based lift simulator with GUI
- **Atomic long with “thread striping” (wk 7)**
- Shared mutable data on multicore is slow

A “striped” thread-safe long

- Use case: more writes (`add`) than reads (`get`)
- Vastly different scalability
 - (a) Java 5’s `AtomicLong`
 - (b) Java 8’s `LongAdder`
 - (c) Home-made synchronized `LongCounter`
 - (d) Home-made striped long using `AtomicLongArray`
 - (e) Home-made striped long with scattered allocation
- Ideas
 - (d,e) Use thread’s `hashCode` to reduce update collisions
 - (e) Scatter `AtomicLongs` to avoid false cache line sharing

TestLongAdders.java

	i7 4c	AMD 32c
(a)	942	3011
(b)	65	54
(c)	1450	14921
(d)	427	1611
(e)	108	922

Wall clock time (ms) for 32 threads making 1 million additions each

Dividing a long into 32 “stripes”

```
class NewLongAdder {  
    private final static int NSTRIPES = 32;  
    private final AtomicLongArray counters = new AtomicLongArray(NSTRIPES);  
  
    public void add(long delta) {  
        counters.addAndGet(Thread.currentThread().hashCode() % NSTRIPES, delta);  
    }  
  
    public long longValue() {  
        long result = 0;  
        for (int stripe=0; stripe<NSTRIPES; stripe++)  
            result += counters.get(stripe);  
        return result;  
    }  
}
```

TestLongAdders.java

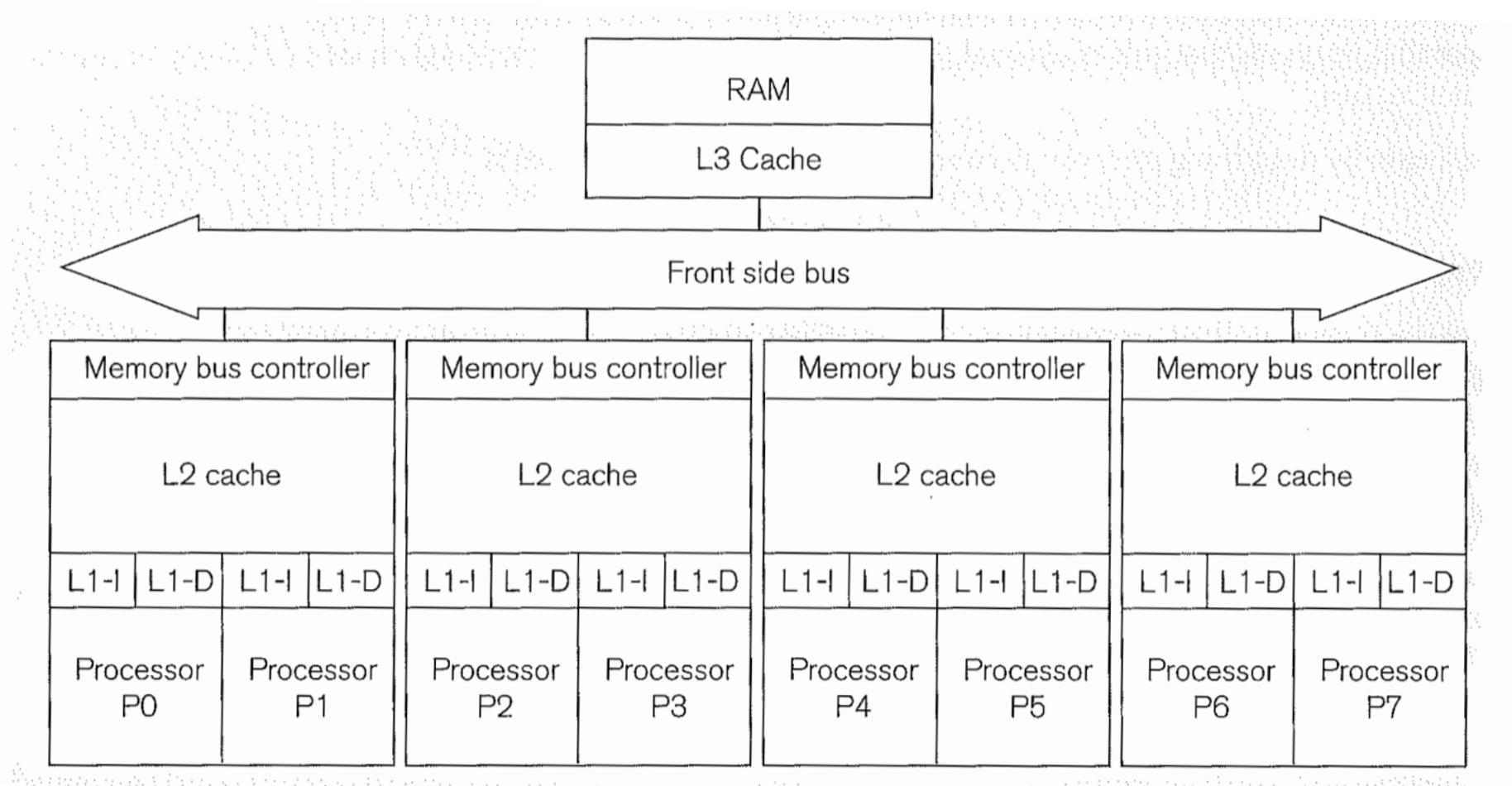
Thread's hashCode
selects stripe

- Two threads unlikely to add to same stripe
- Each stripe has thread-affinity
 - if accessed by thread, likely to be accessed again
- So, fast despite the cost of **hashCode()**

Plan for today

- Graphical user interface toolkits, eg Swing
 - not thread-safe, access from event thread only
- Using SwingWorker for long-running work
 - Progress bar
 - Cancellation
 - Display results as they are generated
- A thread-based lift simulator with GUI
- An atomic long with “thread striping” (week 7)
- **Shared mutable data on multicore is slow**

A typical multicore CPU with three levels of cache

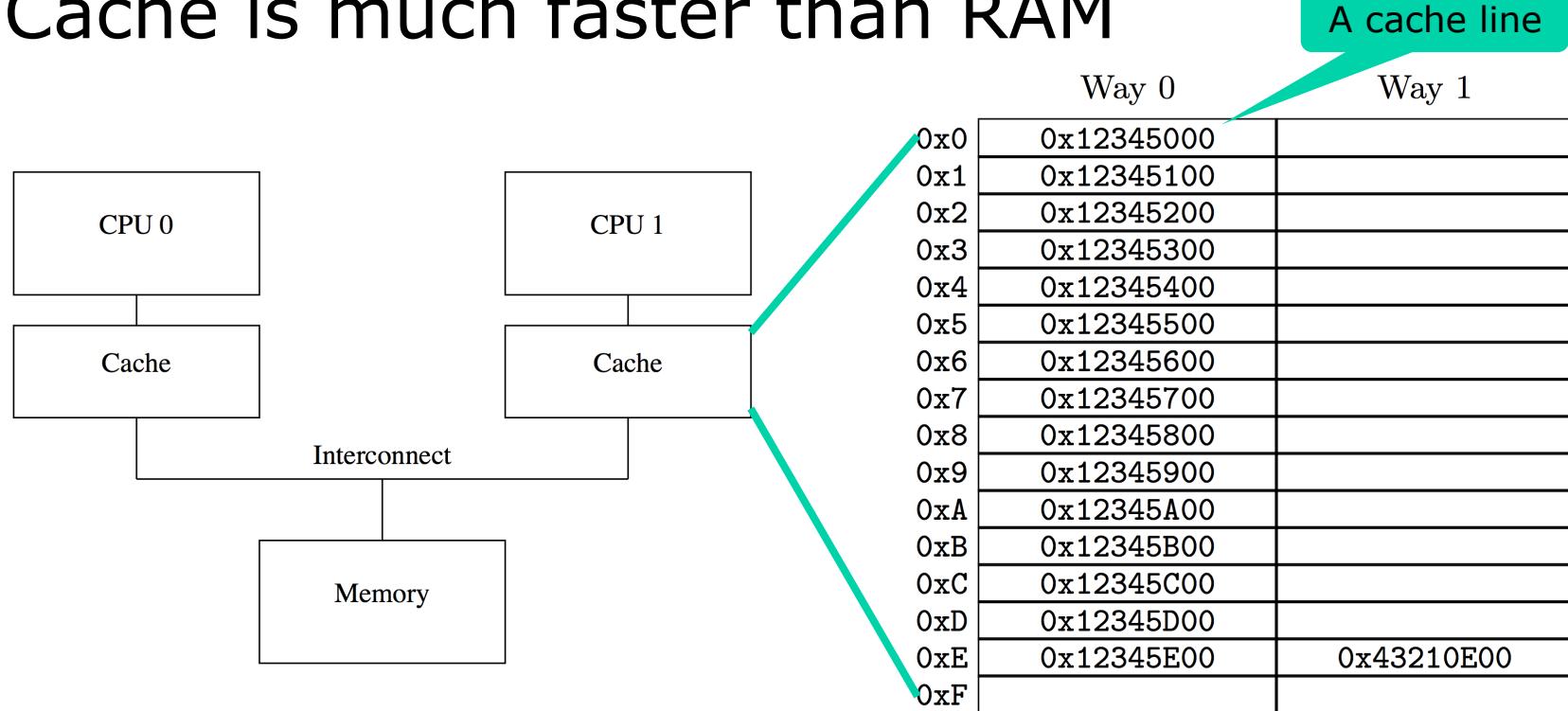


Lin & Snyder 2009, p. 16

- Floating-point register add or mul: 0.4 ns
- RAM access: > 100 ns

Fix 1: Each processor core has a cache

- Cache = simple hardware hashtable
- Stores recently accessed values from RAM
- Cache is much faster than RAM



McKenney 2010: Memory barriers

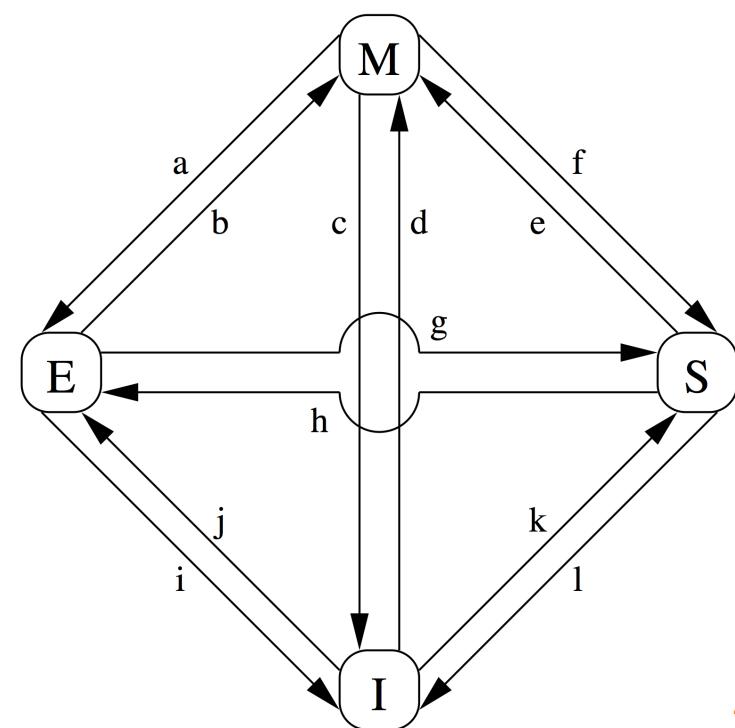
- Two caches may have different values for a given memory address

Fix 2: Get all caches to agree

- Cache coherence; cache line state = M,E,S,I

State	Cache line	Excl	RAM	Read	Write
Modified	Modified by me	Y	not OK	from cache	to cache
Exclusive	Not modified	Y	OK	from cache	to cache -> M
Shared	Others have it too	N	OK	from cache	send invalidate
Invalid	Not in use by me	-	-	elsewhere	send invalidate

- A cache line
 - has 4 states
 - and 12 transitions a-l
- Cache messages
 - sent by cores to others
 - via memory bus
 - to make caches agree



McKenney 2010: Memory barriers

Fast and slow cache cases

- The cache is fast when
 - the local core “owns” the data (state M or E), or
 - data is shared (S) but local core only reads it
- The cache is slow when
 - the data is shared (S) and we want to write it, or
 - the data is not in cache (I)
 - possibly because “owned” by another core

		This core wants to	Messages	Speed
Unshared mutable	M M	Read cache line	0	fast
	M M	Write cache line	0	fast
	E E	Read cache line	0	fast
Shared immutable	E M	Write cache line	0	fast
	S S	Read cache line	0	fast
	I S	Read cache line	1+1	slow
Shared mutable	S M	Write cache line	1+N	very slow
	I M	Write cache line	1+1+N	very slow

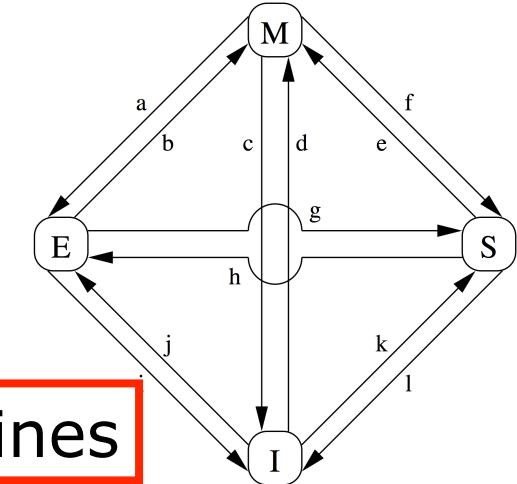
N cores

36

Transitions and messages

A write in a non-exclusive state requires acknowledge ack* from all other cores

Shared mutable state is slow on big machines



		Cause	I send	I receive	My response
M	a	(Send update to RAM)	writeback	-	-
E	b	Write	-	-	-
M	c	Other wants to write	-	read inv	read resp, inv ack
I	d	Atomic read-mod-write	read inv	read resp, inv ack*	-
S	e	Atomic read-mod-write	read inv	inv ack*	-
M	f	Other wants to read	-	read	read resp
E	g	Other wants to read	-	read	read resp
S	h	Will soon write	inv	inv ack*	-
E	i	Other wants atomic rw	-	read inv	read resp, inv ack
I	j	Want to write	read inv	read resp, inv ack*	-
I	k	Want to read	read	read resp	-
S	l	Other wants to write	-	inv	inv ack

One more performance problem: “false sharing” because of cache lines

- A cache line typically is 32 bytes
 - gives better memory bus utilization
 - prefetches data (in array) that may be needed next
- Thus invalidating one (8 byte) long may invalidate the neighboring 3 longs!
- Frequently written memory locations should not be on the same cache line
- Attempts to fix this by “padding”
 - may look very silly (next slide)
 - are not guaranteed to help
 - yet are used in the Java class library code

Scattering the stripes of a long

TestLongAdders.java

```
class NewLongAdderPadded {  
    private final static int NSTRIPES = 32;  
    private final AtomicLong[] counters;  
  
    public NewLongAdderPadded() {  
        this.counters = new AtomicLong[NSTRIPES];  
        for (int stripe=0; stripe<NSTRIPES; stripe++) {  
            // Believe it or not, this sometimes speeds up the code,  
            // presumably because avoids false sharing of cache lines:  
            new Object(); new Object(); new Object(); new Object();  
            counters[stripe] = new AtomicLong();  
        }  
    }  
}
```

Avoid side-by-side
AtomicLong allocation

unless JVM is too clever

- Allocate many AtomicLongs
 - instead of AtomicLongArray
- Scatter the AtomicLongs
 - by allocating some Objects in between

This week

- Reading this week
 - Goetz et al chapter 9
 - Optional: McKenney: *Memory barriers*
- Exercises week 8 = mandatory hand-in 4
 - The week 7 exercises: Write well-performing and scalable software using lock striping, immutability, Java atomics, and visibility rules
 - You can write responsive and correct user interfaces
- Read before next week's lecture
 - Goetz chapter 12

Practical Concurrent and Parallel Programming 9

Peter Sestoft
IT University of Copenhagen

Friday 2014-10-31

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

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 not like 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 buffer using semaphores

```
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;  
    }  
}
```

TestBoundedQueueTest.java

Wait for space

Signal new item

Wait for item

Signal new space

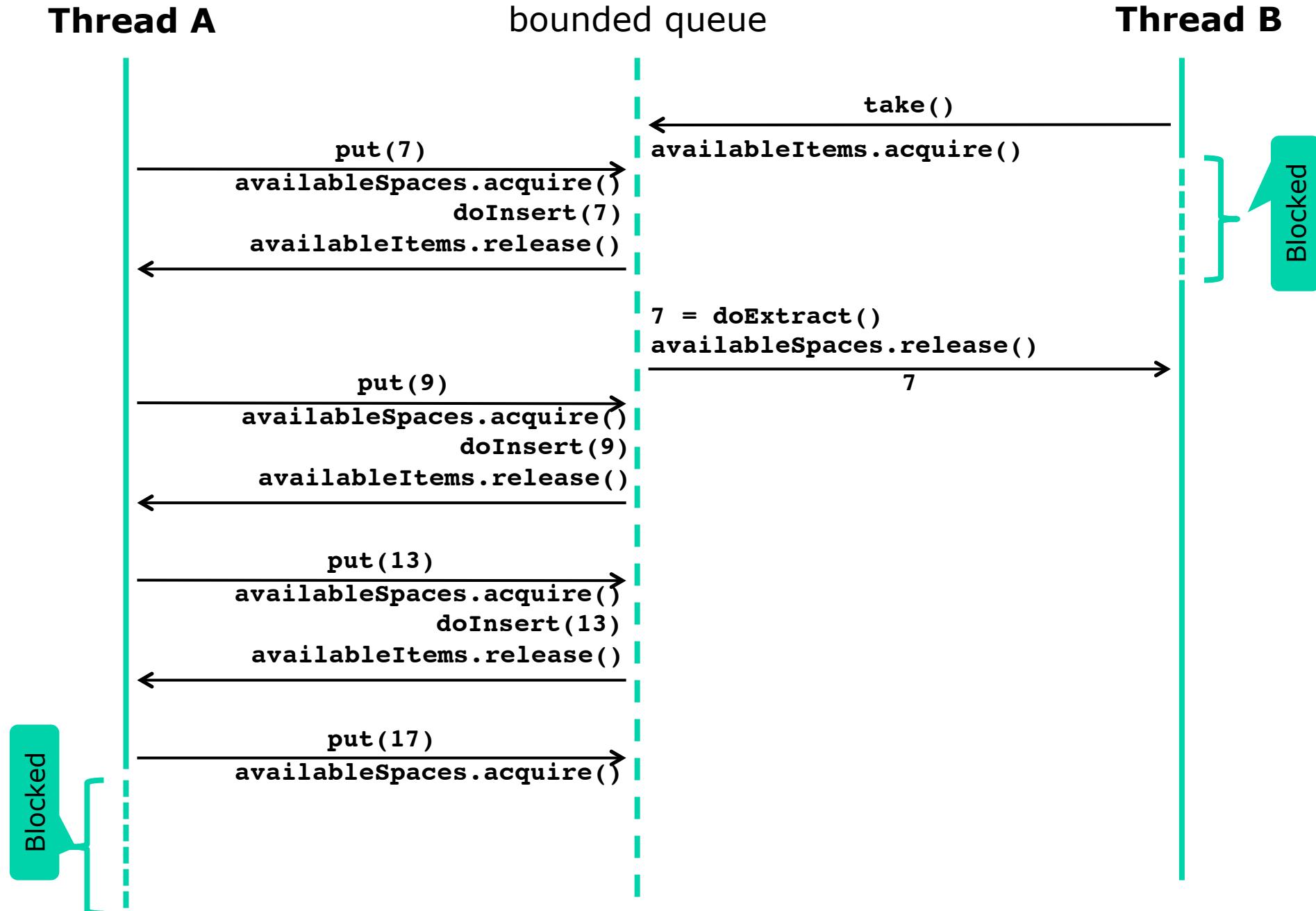
The `doInsert` and `doExtract` methods

```
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



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:

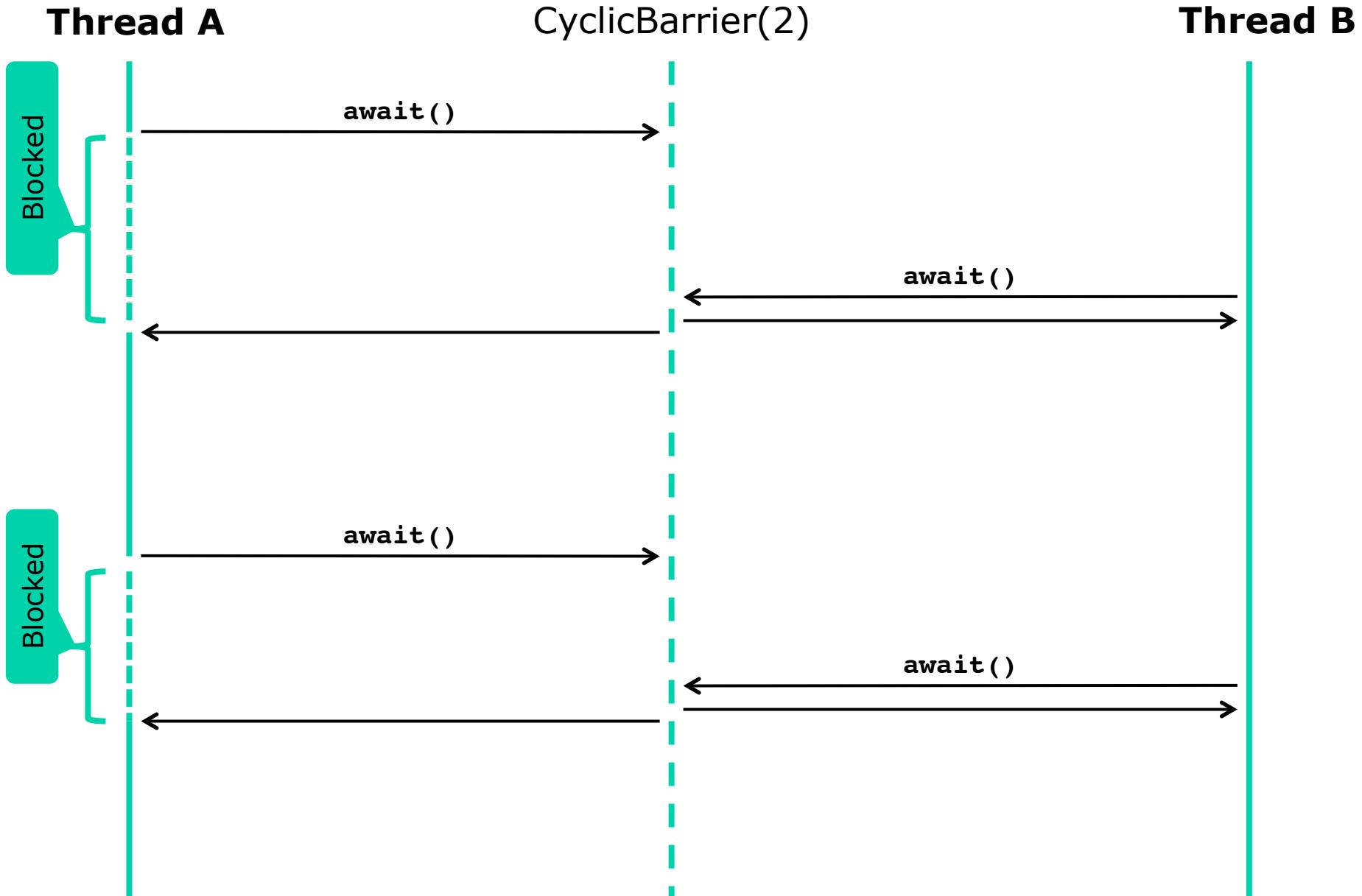
```
assertTrue(bq.isEmpty());
assertTrue(!bq.isFull());
bq.put(7); bq.put(9); bq.put(13);
assertTrue(!bq.isEmpty());
assertTrue(bq.isFull());
assertEquals(bq.take(), 7);
assertEquals(bq.take(), 9);
assertEquals(bq.take(), 13);
assertTrue(bq.isEmpty());
assertTrue(!bq.isFull());
```

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
- A producer inserts 100,000 random numbers
 - Using a *thread-local* random number generator
- A consumer extracts 100,000 numbers
- Afterwards, check that
 - 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; Initialize to 2*npairs+1  
    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++) { Make npairs Producers and npairs Consumers  
                pool.execute(new Producer());  
                pool.execute(new Consumer());  
            }  
            barrier.await(); Main: start, finish threads  
            barrier.await(); Main: start, finish threads  
            assertTrue(bq.isEmpty());  
            assertEquals(putSum.get(), takeSum.get());  
        } catch (Exception e) {  
            throw new RuntimeException(e);  
        }  
    }  
}
```

Goetz p. 255

TestBoundedQueueTest.java

A Producer test thread

```
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'ing 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

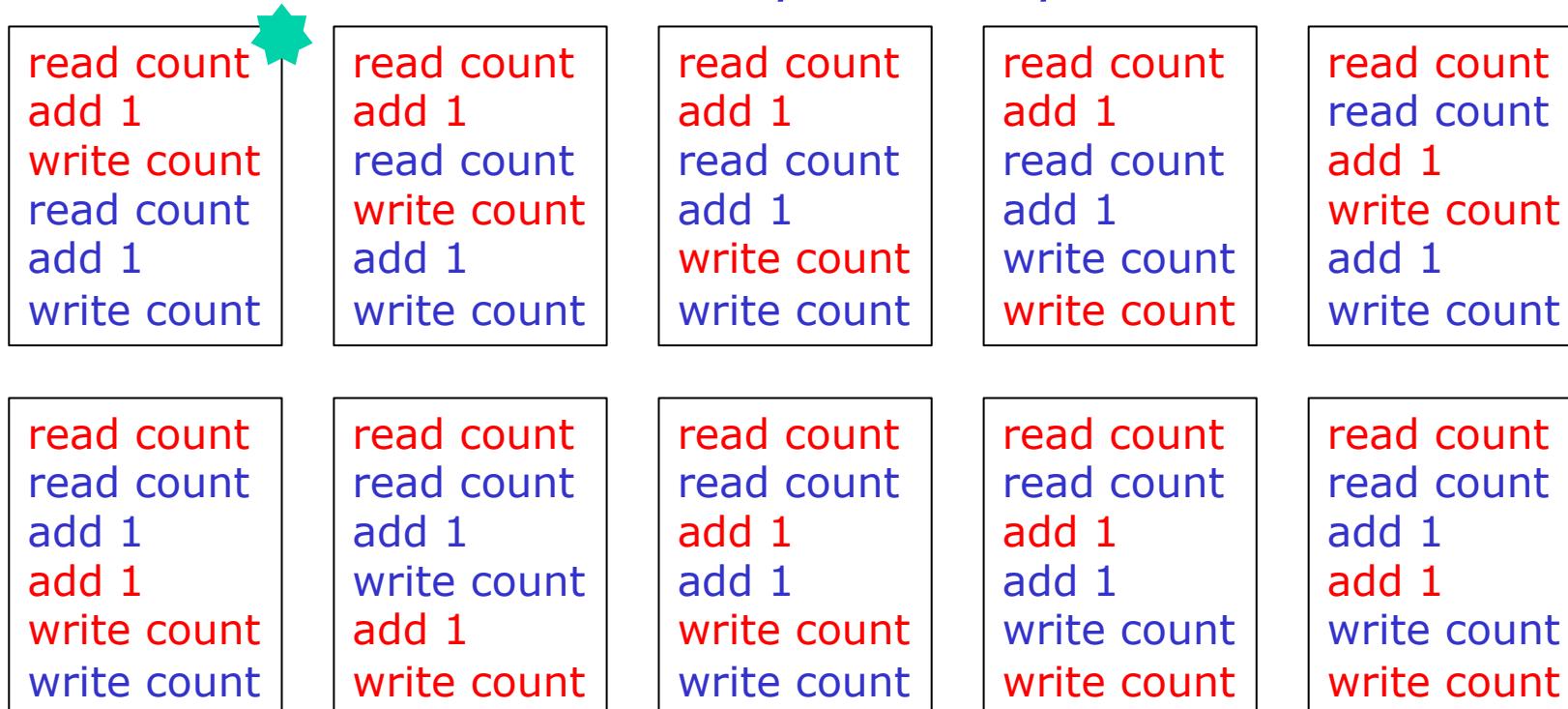
- Sequential
 - *Method coverage*: has each method been called?
 - *Statement coverage*: has each statement been executed?
 - *Branch coverage*: have all branches **if**, **for**, **while**, **do-while**, **switch**, **try-catch** been executed?
 - *Path coverage*: have all paths through the code been 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



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 tests 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?

```
def fac(n: Int): BigInt = if (n==0) 1 else n*fac(n-1)                                Scala
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)
```

```
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
34826943434743360000$
```

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

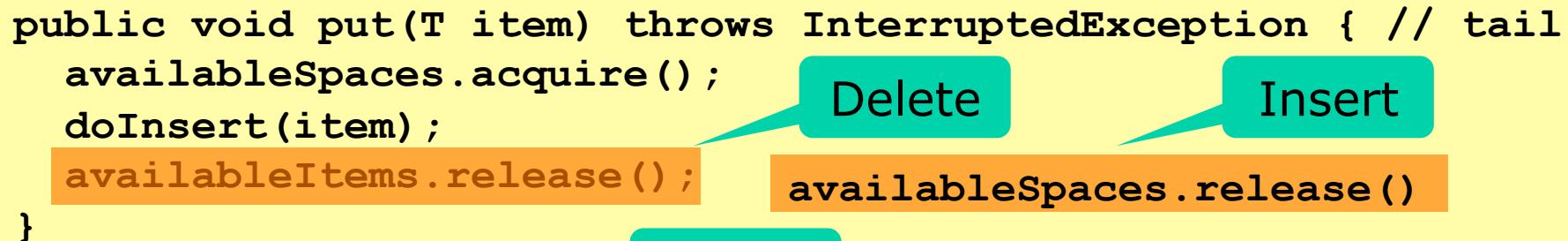
a program P which is correct on test data T is subjected to a series of mutant operators to produce mutant programs which differ from P in very simple ways. The mutants are then executed on T . If all mutants give incorrect results then it is very likely that P is correct (i.e., T is adequate).

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

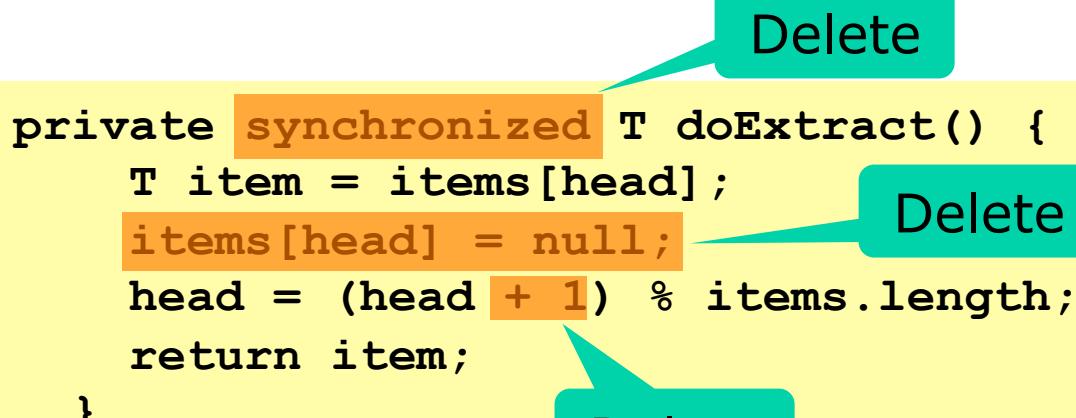
```
public void put(T item) throws InterruptedException { // tail
    availableSpaces.acquire();
    doInsert(item);
    availableItems.release();
}
```



availableItems.release() **Delete**
availableSpaces.release() **Insert**
items[tail] = item; **Delete**

```
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; Delete
    head = (head + 1) % items.length;
    return item;
}
```



items[head] = null; **Delete**
head = (head + 1) % items.length; **Delete**
items[head] = item; **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

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

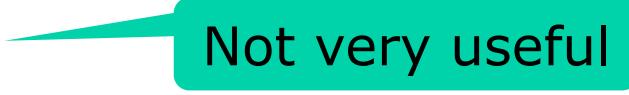
===== system under test
application: TestPhilosophers.java

===== 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*

Non-blocking queue example code

A la Herlihy & Shavit p. 46, 48

TestHSQueues.java

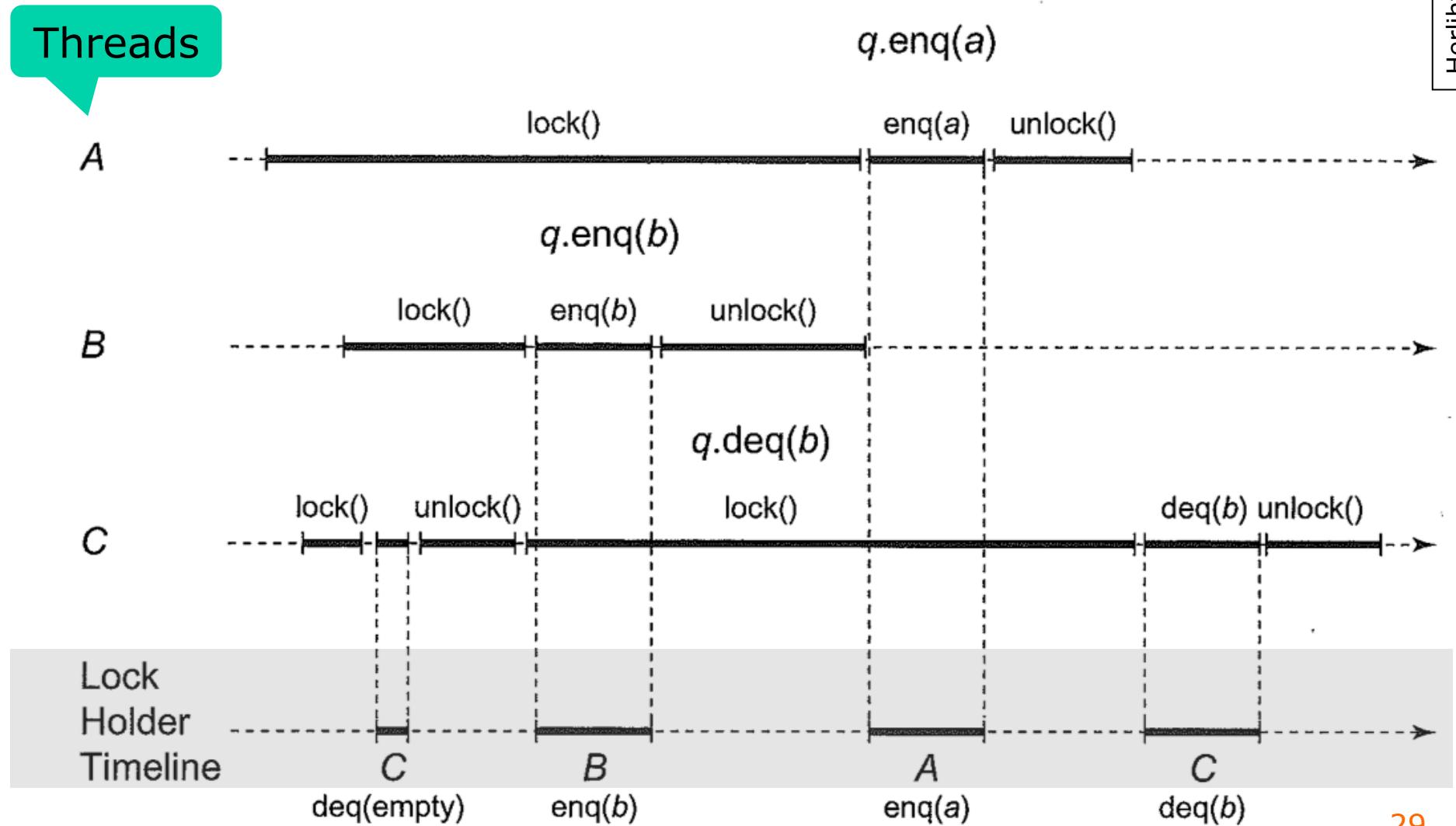
```
class WaitFreeQueue <T> {
    private final T[] items;
    private int tail = 0, head = 0;
    public boolean enq(T item) {
        if (tail - head == items.length)
            return false;
        else {
            items[tail % items.length] = item;
            tail++;
            return true;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

- With only one enqueueuer and one dequeuer, the queue needs no locking!

- With locks, method calls cannot overlap, clear
- Without locks, how understand overlapping calls?
 - One thread calling enq, another calling deq, overlapping

A program run = method calls

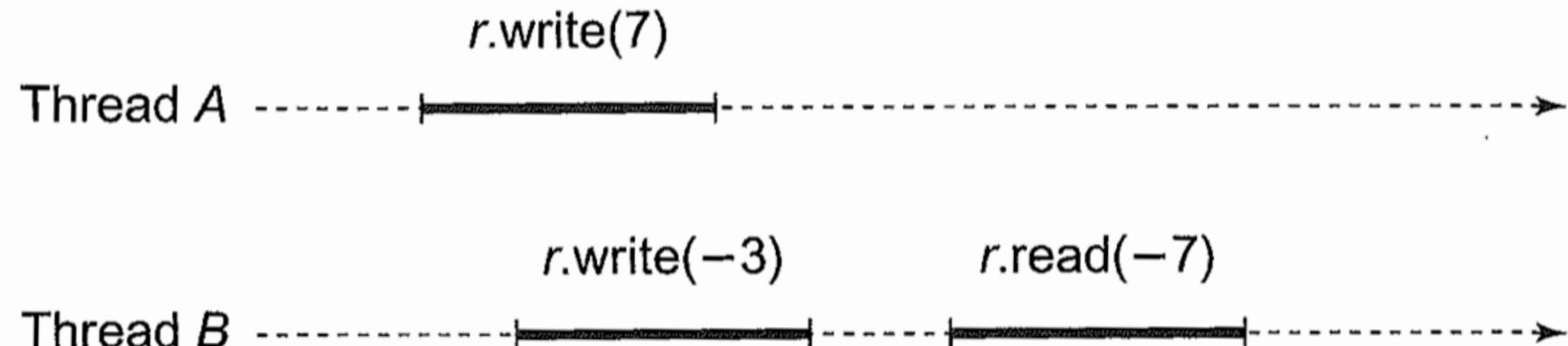
- Method call: invocation, return, and duration
- Method calls may overlap in time



Method call effect seems instantaneous

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

Herlihy & Shavit p. 50



Not acceptable, the method calls' effects are not instantaneous

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.

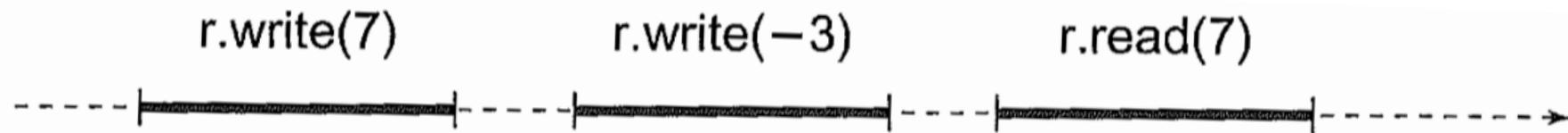
Otherwise the results of these methods reflect transient states that may be adequate for monitoring or estimation purposes, but not for program control.

Class `java.util.concurrent.ConcurrentHashMap` documentation

Sequential consistency and program order

Herlihy & Shavit p. 50

- Principle 3.4.1: *Method calls should appear to take effect in program order*
 - Program order is the *order within a single thread*



Not acceptable

- This scenario is sequentially consistent:

A `q.enq(x)`
`q.enq(y)`
`q.deq(x)`
`q.deq(y)`

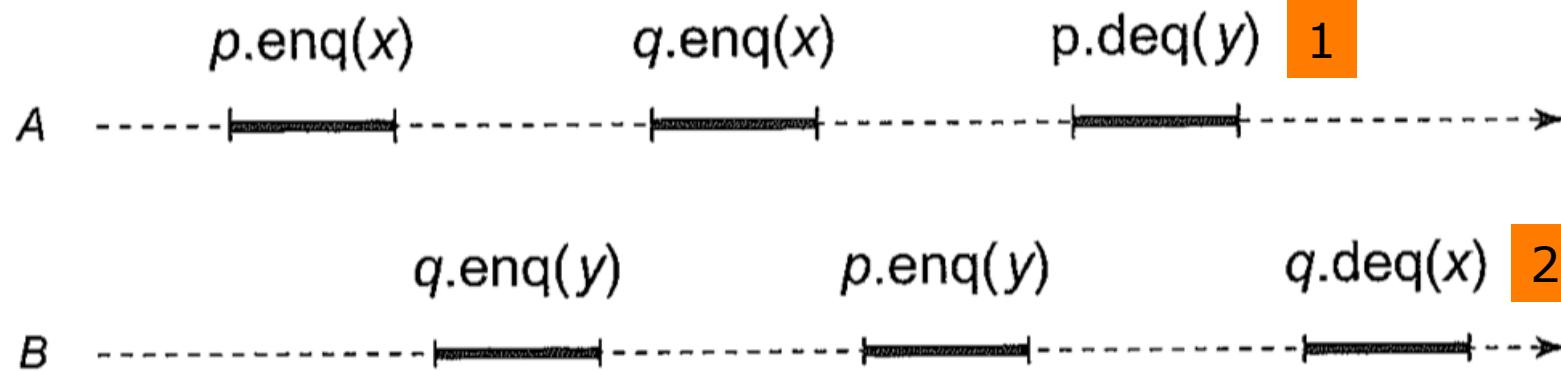
`q.enq(x)` `q.deq(y)`

OR

B `q.enq(y)`
`q.enq(x)`
`q.deq(y)`
`q.deq(x)`

`q.enq(y)` `q.deq(x)`

Seq. consistency is not compositional



Herlihy & Shavit p. 54

- Sequentially consistent for each queue p, q :

B $p.\text{enq}(y)$
A $p.\text{enq}(x)$
A $p.\text{deq}(y)$

AND

B $q.\text{enq}(y)$
A $q.\text{enq}(x)$
B $q.\text{deq}(y)$

- Taken together, they are not seq. consistent:
 - 1 – $p.\text{enc}(y)$ must precede $p.\text{enc}(x)$
 - which precedes $q.\text{enc}(x)$ in thread A program order
 - 2 – $q.\text{enc}(x)$ must precede $q.\text{enc}(y)$
 - which precedes $p.\text{enc}(y)$ in thread B program order

– So $p.\text{enc}(y)$ must precede $p.\text{enc}(y)$, impossible

Reflection on sequential consistency

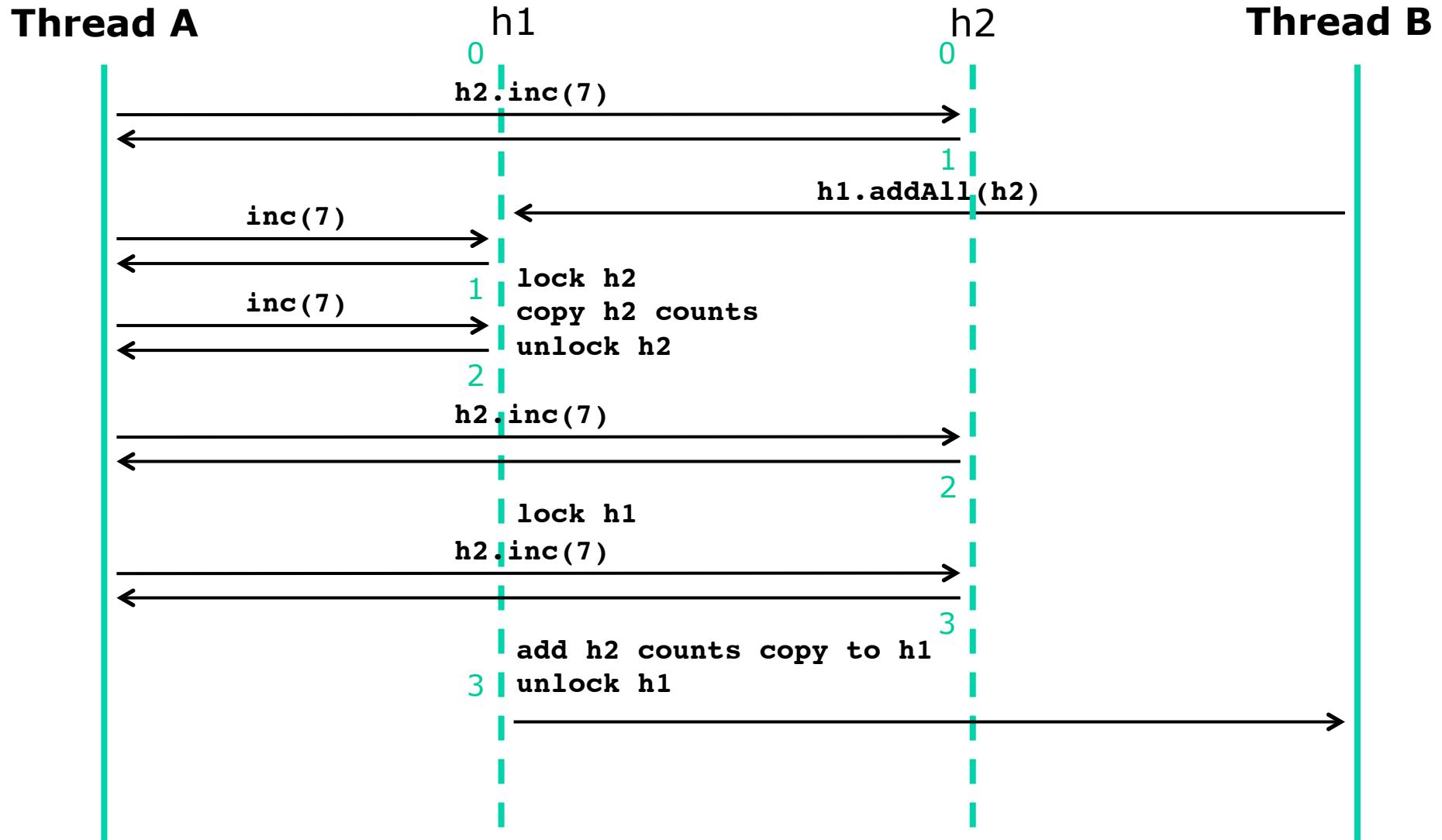
- Seems natural
- It is what synchronization tries to achieve
- If all (unsynchronized) code were to satisfy it, that would preclude optimizations:

Java, or C#, does not guarantee sequential consistency of 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

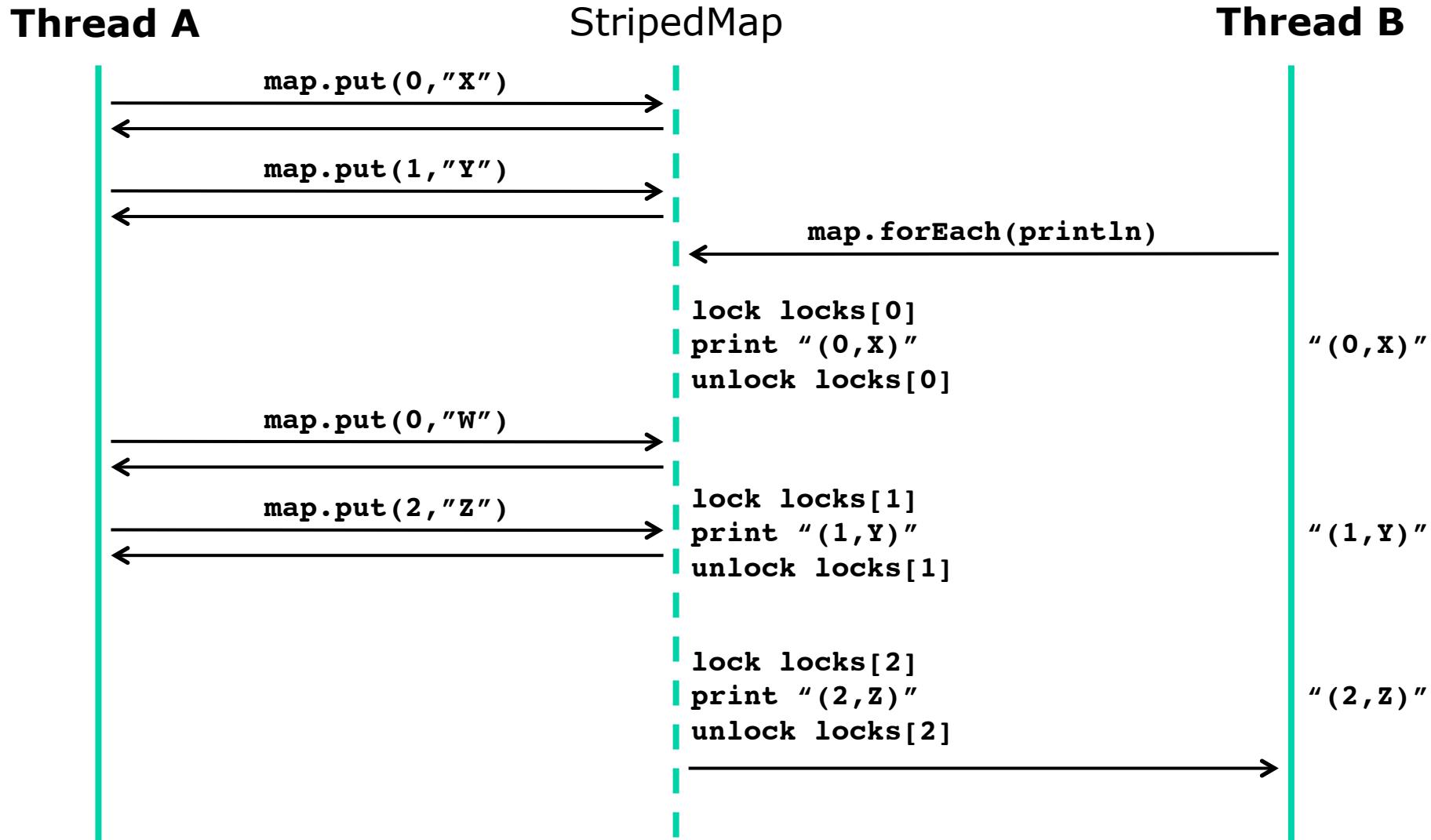
- Principle 3.5.1: *Each method call should appear to take effect instantaneously at some moment between its invocation and response.*
- Usually shown by identifying a *linearization point* for each method.
- In a Java monitor pattern methods, the linearization point is typically at lock release
- In non-locking WaitFreeQueue<T>
 - linearization point of `enc()` is at `tail++` update
 - linearization point of `dec()` is at `head++` update
- Less clear in lock-free methods, week 11-12

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



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) does not skip entries that existed at call time, and (b) does not process an entry twice

This week

- Reading
 - Goetz et al chapter 12
 - Herlihy & Shavit chapter 3
- Exercises
 - Show you can test concurrent software with subtle synchronization mechanisms
- Read before next week's lecture
 - Herlihy and Shavit sections 18.1-18.2
 - Harris et al: *Composable memory transactions*
 - Cascaval et al: *STM, Why is it only a research toy*

Next week's reading: Software transactional memory STM

- Herlihy and Shavit sections 18.1-18.2
 - Brief critique of locking and introduction to STM
- Harris et al: *Composable memory transactions, 2008*
 - Made STM popular again around 2004
 - Using the functional language Haskell
- Cascaval et al: *STM, Why is it only a research toy, 2008*
 - Some people are skeptical, but they use C
 - STM more likely to be useful in mostly-immutable settings than in anarchic imperative/OO settings

Practical Concurrent and Parallel Programming 10

Peter Sestoft
IT University of Copenhagen

Friday 2014-11-07

Plan for today

- What's wrong with lock-based atomicity
- Transactional memory STM, Multiverse library
- A transactional bank account
- Transactional blocking queue
- Composing atomic operations
 - transfer from one queue to another
 - choose first available item from two queues
- Philosophical transactions
- Other languages with transactional memory
- Hardware support for transactional memory

Transactional memory

- Based on transactions, as in databases
- Transactions are composable
 - unlike lock-based concurrency control
- Easy to implement blocking
 - no `wait` and `notifyAll` or semaphore trickery
- Easy to implement blocking choice
 - eg. get first item from any of two blocking queues
- Typically *optimistic*
 - automatically very high read-parallelism
 - unlike *pessimistic* locks
- No deadlocks and usually no livelocks

Transactions

- Know from databases since 1981 (Jim Gray)
- Proposed for programming languages 1986
 - (In a functional programming conference)
- Became popular again around 2004
 - due to Harris, Marlow, Peyton-Jones, Herlihy
 - Haskell, Clojure, Scala, ... and Java Multiverse
- A transaction must be
 - **Atomic**: if one part fails, the entire transaction fails
 - **Consistent**: maps a valid state to a valid state
 - **Isolated**: A transaction does not see the effect of any other transaction while running
 - (But not **Durable**, as in databases)

Difficulties with lock-based atomicity

- Transfer money from account ac1 to ac2
 - No help that each account operation is atomic
 - Can lock both, but then there is deadlock risk
- Transfer an item from queue bq1 to bq2
 - No help that each queue operation is atomic
 - Locking both, nobody can put and take; deadlock
- Get an item from either queue bq1 or bq2
 - Should block if both empty
 - But just calling `b1.take()` may block forever even if there is an available item in `bq2`

A la Herlihy & Shavit §18.2

Transactions makes this trivial

- Transfer amount from account ac1 to ac2:

```
atomic {
    ac1.deposit(-amount);
    ac2.deposit(+amount);
}
```

Pseudo-code

- Transfer one item from queue bq1 to bq2:

```
atomic {
    T item = bq1.take();
    bq2.put(item);
}
```

A la Herlihy & Shavit §18.2

- Take item from queue bq1 if any, else bq2:

```
atomic {
    return bq1.take();
} orElse {
    return bq2.take();
}
```

Transactional account

Acc

```
class Account {  
    private long balance = 0;  
    public void deposit(final long amount) {  
        atomic {  
            balance += amount;  
        }  
    }  
    public long get() {  
        atomic {  
            return balance;  
        }  
    }  
    public void transfer(Account that, final long amount) {  
        final Account thisAccount = this, thatAccount = that;  
        atomic {  
            thisAccount.deposit(-amount);  
            thatAccount.deposit(+amount);  
        }  
    }  
}
```

Pseudo-code

Composite transaction
without deadlock risk

Transactional memory in Java

- Multiverse Java library 0.7 from April 2012
 - Seems comprehensive and well-implemented
 - Little documentation apart from API docs
 - ... and those API docs are quite cryptic
- A transaction must be wrapped in
 - `new Runnable() { ... }` if returning nothing
 - `new Callable<T>() { ... }` if returning a T value
- Runs on unmodified JVM
 - Thus is often slower than locks/volatile/CAS/...
- To compile and run:

```
$ javac -cp ~/lib/multiverse-core-0.7.0.jar TestAccounts.java
$ java -cp ~/lib/multiverse-core-0.7.0.jar:. TestAccounts
```

Transactional account, Multiverse

Acc

```
class Account {  
    private final TxnLong balance = newTxnLong(0);  
    public void deposit(final long amount) {  
        atomic(new Runnable() { public void run() {  
            balance.set(balance.get() + amount);  
        }});  
    }  
    public long get() {  
        return atomic(new Callable<Long>() {public Long call() {  
            return balance.get();  
        }});  
    }  
    public void transfer(Account that, final long amount) {  
        final Account thisAccount = this, thatAccount = that;  
        atomic(new Runnable() { public void run() {  
            thisAccount.deposit(-amount);  
            thatAccount.deposit(+amount);  
        }});  
    } }  
}
```

stm/TestAccounts.java

Callable<Long>
to return a long

Composite transaction
without deadlock risk

Consistent reads may require explicit Read lock

Acc

- Auditor computes balance sum during transfer

```
long sum = atomic(new Callable<Long>() { public Long call() {  
    return account1.get() + account2.get();  
}});  
System.out.println(sum);
```

stm/TestAccounts.java

- Must read both balances in same transaction
 - Does not work to use a transaction for each reading
- Should print the sum only outside transaction
 - After the transaction committed
 - Otherwise risk of printing twice, or inconsistently
- Does not work if **deposit(amount)** uses **balance.increment(amount)** ????

How do transactions work?

- A transaction `txn` typically keeps
 - Read Set: all variables read by the transaction
 - Write Set: *local copy* of variables it has updated
- When trying to commit, check that
 - no variable in Read Set or Write Set has been updated by another transaction
 - if OK, write Write Set to global memory
 - otherwise, discard Write Set and restart `txn` again
- So the Runnable may be called many times!
- How long to wait before trying again?
 - Exponential backoff: wait `rnd.nextInt(2)`,
`rnd.nextInt(4)`, `rnd.nextInt(8)`, ...
 - Should prevent transactions from colliding forever

Nested transactions

- By default, an `atomic` within an `atomic` reuses the outer transaction: So if the inner fails, the outer one fails too
- Several other possibilities, see `org.multiverse.api.PropagationLevel`
 - Default is `PropagationLevel.Requires`: if a there is a transaction already, use that; else create one

Multiverse transactional references

- Only transactional variables are tracked
 - `TxnRef<T>`, a transactional reference to a `T` value
 - `TxnInteger`, a transactional `int`
 - `TxnLong`, a transactional `long`
 - `TxnBoolean`, a transactional `boolean`
 - `TxnDouble`, a transactional `double`
- Methods, uses in a transaction, inside `atomic`
 - `get()`, to read the reference
 - `set(value)`, to write the reference
- Several other methods, eg
 - `getAndLock(lockMode)`, for more pessimism
 - `await(v)`, block until value is `v`

Plan for today

- What's wrong with lock-based atomicity
- Transactional memory STM, Multiverse library
- A transactional bank account
- **Transactional blocking queue**
- **Composing atomic operations**
 - transfer from one queue to another
 - choose first available item from two queues
- Philosophical transactions
- Other languages with transactional memory
- Hardware support for transactional memory

Lock-based bounded queue (wk 9)

```
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 {  
        availableSpaces.acquire();  
        doInsert(item);  
        availableItems.release();  
    }  
  
    private synchronized void doInsert(T item) {  
        items[tail] = item;  
        tail = (tail + 1) % items.length;  
    }  
  
    public T take() throws InterruptedException { ... }  
    ...  
}
```

Use semaphore to block until room for new item

Use lock for atomicity

Transactional blocking queue

```
class StmBoundedQueue<T> implements BoundedQueue<T> {  
    private int availableItems, availableSpaces;  
    private final T[] items;  
    private int head = 0, tail = 0;  
  
    public void put(T item) { // at tail  
        atomic {  
            if (availableSpaces == 0)  
                retry();  
            else {  
                availableSpaces--;  
                items[tail] = item;  
                tail = (tail + 1) % items.length;  
                availableItems++;  
            }  
        }  
    }  
    public T take() {  
        ... availableSpaces++; ...  
    }  
}
```

Atomic action

Use **retry()** to block

Pseudo-code

Real code, using Multiverse library

```

class StmBoundedQueue<T> implements BoundedQueue<T> {
    private final TxnInteger availableItems, availableSpaces;
    private final TxnRef<T>[] items;
    private final TxnInteger head, tail;

    public void put(T item) {      // at tail
        atomic(new Runnable() { public void run() {
            if (availableSpaces.get() == 0)
                retry();
            else {
                availableSpaces.decrement();
                items[tail.get()].set(item);
                tail.set((tail.get() + 1) % items.length);
                availableItems.increment();
            }
        }});
    }

    public T take() {
        ... availableSpaces.increment(); ...
    }
}

```

stm/TestStmQueues.java

Atomic action

Use **retry()** to block

How does blocking work?

- When a transaction executes `retry()` ...
 - The Read Set tells what variables have been read
 - No point in restarting the transaction until one of these variables have been updated by other thread
- Hence NOT a busy-wait loop
 - but automatic version of `wait` and `notifyAll`
 - or automatic version of `acquire` on Semaphore
- Often works out of the box, idiot-proof
- Must distinguish:
 - restart of transaction because could not commit
 - exponential backoff, random sleep before restart
 - an explicit `retry()` request for blocking
 - waits in a queue for Read Set to change

Atomic transfer between queues

```
static <T> void transferFromTo(BoundedQueue<T> from,
                                BoundedQueue<T> to)
{
    atomic(new Runnable() { public void run() {
        T item = from.take();
        to.put(item);
    }});
}
```

stm/TestStmQueues.java

- A direct translation from the pseudo-code
- Can hardly be wrong

Blocking until some item available

```
static <T> T takeOne(BoundedQueue<T> bq1,  
                      BoundedQueue<T> bq2) throws Exception  
{  
    return myOrElse(new Callable<T>() { public T call() {  
        Do this  
        return bq1.take();  
    } },  
    new Callable<T>() { public T call() {  
        or else  
        return bq2.take();  
    } } );  
}
```

stm/TestStmQueues.java

- If **bq1.take()** fails, try instead **bq2.take()**
- Implemented using general **orElse** method
 - taking as arguments two Callables

Implementing method myOrElse

```
static <T> T myOrElse(Callable<T> either, Callable<T> orelse)
    throws Exception
{
    return atomic(new Callable<T>() { public T call() throws ... {
        try {
            return either.call();
        } catch (org.multiverse.api.exceptions.RetryError retry) {
            return orelse.call();
        }
    }});
}
```

stm/TestStmQueues.java

- Exposes Multiverse's internal machinery
- Hand-made implementation
 - Because Multiverse's OrElseBlock seems faulty

Plan for today

- What's wrong with lock-based atomicity
- Transactional memory STM, Multiverse library
- A transactional bank account
- Transactional blocking queue
- Composing atomic operations
 - transfer from one queue to another
 - choose first available item from two queues
- **Philosophical transactions**
- Other languages with transactional memory
- Hardware support for transactional memory

Philosophical Transactions

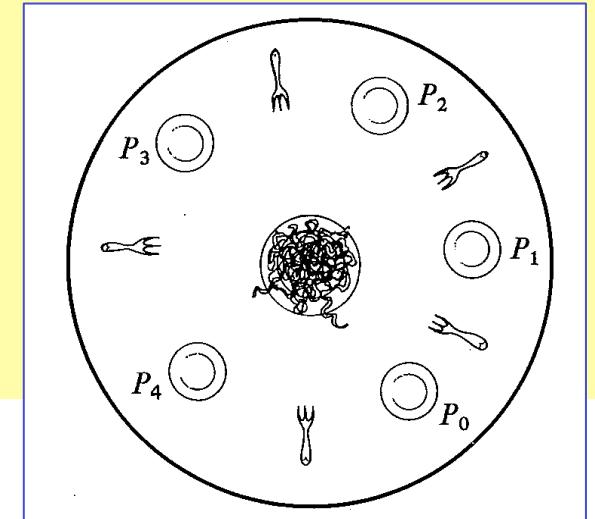
```

class Philosopher implements Runnable {
    private final Fork[] forks;
    private final int place;
    public void run() {
        while (true) {
            int left = place, right = (place+1) % forks.length;
            synchronized (forks[left]) {
                synchronized (forks[right]) {
                    System.out.print(place + " ");
                    // Eat
                }
            }
            try { Thread.sleep(10); } // Think
            catch (InterruptedException exn) { }
        }
    }
}

```

TestPhilosophers.java

Exclusive
use of forks



- Lock-based philosopher (wk 6)
 - Likely to deadlock in this version

TxnBooleans as Forks A

```
class Philosopher implements Runnable {  
    private final TxnBoolean[] forks;  
    private final int place;  
    public void run() {  
        while (true) {  
            final int left = place, right = (place+1) % forks.length;  
            atomic(new Runnable() { public void run() {  
                if (!forks[left].get() && !forks[right].get()) {  
                    forks[left].set(true);  
                    forks[right].set(true);  
                } else  
                    retry();  
            }});  
            System.out.printf("%d ", place); // Eat  
            atomic(new Runnable() { public void run() {  
                forks[left].set(false);  
                forks[right].set(false);  
            }});  
            try { Thread.sleep(10); } // Think  
            catch (InterruptedException exn) { }  
        }  
    }  
}
```

Exclusive
use of forks

Release
forks

TxnBooleans as Forks B

```
class Philosopher implements Runnable {  
    private final TxnBoolean[] forks;  
    private final int place;  
    public void run() {  
        while (true) {  
            final int left = place, right = (place+1) % forks.length;  
            atomic(new Runnable() { public void run() {  
                forks[left].await(false);  
                forks[left].set(true);  
                forks[right].await(false);  
                forks[right].set(true);  
            }});  
            System.out.printf("%d ", place); // Eat  
            atomic(new Runnable() { public void run() {  
                forks[left].set(false);  
                forks[right].set(false);  
            }});  
            try { Thread.sleep(10); } // Think  
            catch (InterruptedException exn) { }  
        }  
    }  
}
```

stm/TestStmPhilosophersB.java

Exclusive
use of forks

Release
forks

Transaction subtleties

- What is wrong with this Philosopher?
 - A variant of B, “eating” inside the transaction

```
public void run() {  
    while (true) {  
        final int left = place, right = (place+1) % forks.length;  
        atomic(new Runnable() { public void run() {  
            forks[left].await(false);  
            forks[left].set(true);  
            forks[right].await(false);  
            forks[right].set(true);  
            System.out.printf("%d ", place); // Eat  
            forks[left].set(false);  
            forks[right].set(false);  
        }});  
        try { Thread.sleep(10); } // This  
        catch (InterruptedException exn) { }  
    }  
}
```

BAD

Transaction has its
own view of the
world until commit

Other transactions
may have taken all
the forks!

Optimism and multiple universes

- A transaction has its own copy of data (forks)
- At commit, it checks that data it used is valid
 - if so, writes the updated data to common memory
 - otherwise throws away the data, and restarts
- Each transaction works in its own “universe”
 - until it successfully commits
- This allows higher concurrency
 - especially when write conflicts are rare
 - but means that a Philosopher cannot know it has exclusive use of a fork until transaction commit
- Transactions + optimism = multiple universes
- No I/O or other side effects in transactions!

Hints and warnings

- Transactions should be short
 - When a long transaction finally tries to commit, it is likely to have been undermined by a short one
 - ... and must abort, and a lot of work is wasted
 - ... and it retries, so this happens again and again
- For example, concurrent hash map
 - short: `put`, `putIfAbsent`, `remove`
 - long: `reallocateBuckets` – not clear it will ever succeed when others `put` at the same time
- Some STM implementations abort the transaction that has already done most work
 - Many design tradeoffs

Some languages with transactions

- Haskell – in GHC implementation
 - TVar T, similar to TxnRef<T>, TxnInteger, ...
- Scala – ScalaSTM, on Java platform
 - Ref[T], similar to TxnRef<T>, TxnInteger, ...
- Clojure – on Java platform
 - (ref x), similar to TxnRef<T>, TxnInteger, ...
- C, C++ – future standards proposals
- Java – via Multiverse library
 - Creator Peter Ventjeer is on ScalaSTM team too
- And probably many more ...

Transactional memory in perspective

- Works best in a mostly immutable context
 - eg functional programming: Haskell, Clojure, Scala
- Mixes badly with side effects, input-output
- Requires transactional (immutable) collection classes and so on
- Some loss of performance in software-only TM
- Still unclear how to best implement it
- Some think it will remain a toy, Cascaval 2008
 - ... but they use C/C++, too much mutable data
- Multicore hardware support would help
 - can be added to cache coherence (MESI) protocols

Hardware support for transactions

- Eg Intel TSX for Haswell CPUs, since 2013
 - New XBEGIN, XEND, XABORT instructions
 - <https://software.intel.com/sites/default/files/m/9/2/3/41604>
- Could be used by future JVMs, .NET/CLI, ...
- Uses core's cache for transaction's updates
- Extend cache coherence protocol (MESI, wk 8)
 - Messages say when another core writes data
 - On commit, write cached updates back to RAM
 - On abort, invalidate cache, do not write to RAM
- Limitations:
 - Limited cache size, ...

This week

- Reading
 - Herlihy and Shavit sections 18.1-18.2
 - Harris et al: *Composable memory transactions*
 - Cascaval et al: *STM, Why is it only a research toy*
- Exercises
 - Show you can use transactional memory to implement histogram and concurrent hashmap
- Read before next week
 - Goetz et al chapters 15 and 16
 - Herlihy & Shavit chapter 11

Practical Concurrent and Parallel Programming 11

Peter Sestoft
IT University of Copenhagen

Friday 2014-11-14*

Plan for today

- Compare and swap (CAS) low-level atomicity
- Examples: AtomicInteger and NumberRange
- How to implement a lock using CAS
- Scalability: pessimistic locks vs optimistic CAS
- Treiber lock-free stack
- The ABA problem
- Progress concepts
 - Lock-free, wait-free, obstruction-free
- Course evaluation feedback

Compare-and-swap (CAS)

- Atomic check-then-set, IBM 1970, Intel 80486 ...
- Java `AtomicReference<T>`
 - `var.compareAndSet(T oldVal, T newVal)`
If `var` holds `oldVal`, set it to `newVal` and return true
- .NET/CLI `System.Threading.Interlocked`
 - `CompareExchange<T>(ref T var, T newVal, T oldVal)`
If `var` holds `oldVal`, set it to `newVal` and return true
- Optimistic concurrency
 - Try to update; if it fails, maybe restart
- Similar to transactional memory (STM, week 10)
 - but only one variable at a time
 - and under programmer control, not automatic
 - low-level machine primitive, where STM is high-level

CAS versus mutual exclusion (locks)

- Optimistic versus pessimistic concurrency
- Pro CAS
 - Almost all modern hardware implements CAS
 - Modern CAS is quite fast, 20-50 cycles
 - CAS is used to implement locks
 - A failed CAS, unlike failed lock acquisition, requires no context switch, see Java Precisely p. 67
 - Therefore fast when contention is low
- Con CAS
 - Restart may fail arbitrarily many times
 - Therefore slow when contention is high
 - CAS slow on some manycore machines (32 c AMD)

Pseudo-implementation of CAS

```
class MyAtomicInteger {  
    private int value;          // Visibility ensured by locking  
    synchronized boolean compareAndSet(int oldValue, int newValue){  
        if (this.value == oldValue) {  
            this.value = newValue;  
            return true;  
        } else  
            return false;  
    }  
  
    public synchronized int get() {  
        return this.value;  
    }  
    ...  
}
```

TestCasAtomicInteger.java

- Only to illustrate CAS semantics
 - In reality **synchronized** is implemented by CAS
 - Not the other way around

AtomicInteger operations via CAS

```
public int addAndGet(int delta) {  
    int oldValue, newValue;  
    do {  
        oldValue = get();  
        newValue = oldValue + delta;  
    } while (!compareAndSet(oldValue, newValue));  
    return newValue;  
}  
public int getAndSet(int newValue) {  
    int oldValue;  
    do {  
        oldValue = get();  
    } while (!compareAndSet(oldValue, newValue));  
    return oldValue;  
}
```

TestCasAtomicInteger.java

- Optimistic concurrency approach
 - read **oldValue** from variable without locking
 - do computation, giving **newValue**
 - update variable if **oldValue** still valid

CAS and multivariable invariants: Unsafe number range [lower,upper]

```
public class NumberRange {  
    // INVARIANT: lower <= upper  
    private final AtomicInteger lower = new AtomicInteger(0);  
    private final AtomicInteger upper = new AtomicInteger(0);  
  
    public void setLower(int i) {  
        if (i > upper.get())  
            throw new IllegalArgumentException("can't set lower");  
        lower.set(i);  
    }  
  
    public void setUpper(int i) {  
        if (i < lower.get())  
            throw new IllegalArgumentException("can't set upper");  
        upper.set(i);  
    }  
}
```

Non-atomic test-then-set, may break *invariant*

Immutable integer pairs

- Use same technique as for factor cache (wk 2)
 - Make *immutable* pair of fields
 - Atomic assignment of reference to immutable pair
- Here, immutable pair of lower & upper bound:

```
private class IntPair {  
    // INVARIANT: lower <= upper  
    final int lower, upper;  
  
    public IntPair(int lower, int upper) {  
        this.lower = lower;  
        this.upper = upper;  
    }  
}
```

Immutable, and
safely publishable

Goetz p. 326

Using CAS to set the pair reference

```
public class CasNumberRange {  
    private final AtomicReference<IntPair> values  
    = new AtomicReference<IntPair>(new IntPair(0, 0));  
  
    public int getLower() { return values.get().lower; }  
  
    public void setLower(int i) {  
        while (true) {  
            IntPair oldv = values.get();  
            if (i > oldv.upper)  
                throw new IllegalArgumentException("Can't set lower");  
            IntPair newv = new IntPair(i, oldv.upper);  
            if (values.compareAndSet(oldv, newv))  
                return;  
        }  
    }  
}
```

Set if nobody
else changed it

- Atomic replacement of one pair by another
 - But may create many pairs before success ...

CAS has visibility effects

- Java's `AtomicReference.compareAndSet` etc have the same visibility effects as `volatile`: "The memory effects for accesses and updates of atomics generally follow the rules for volatiles" (`java.util.concurrent.atomic` package documentation)
- Also in C#/.NET/CLI, Ecma-335, §I.12.6.5: "... atomic operations in the `System.Threading.Interlocked` class ... perform implicit acquire/release operations"

CAS in Java versus .NET

- .NET has static CAS methods in Interlocked
 - One can CAS to any variable or array element, good
 - But can easily forget to use CAS for update, bad
- Java's `AtomicReference<T>` seems safer
 - Because *must* access the field through that class
- But, for efficiency, Java allows standard field access through `AtomicReferenceFieldUpdater`
 - Uses reflection, see next week
 - This is at least as bad as the .NET design
 - And gives poor tool support: IDE, refactoring, ...

Why compare-and-swap?

- *Consensus number* CN of a read-modify-write operation: the maximum number of parallel processes for which it can solve *consensus*, ie. make them agree on the value of a variable
- Atomically read a variable: CN = 1
- Atomically write a variable: CN = 1
- Test-and-set: atomically write a variable and return its old value: CN = 2
- Compare-and-swap: atomically check that variable has value oldVal and if so set to newVal, returning true; else false: CN = ∞

Herlihy: Wait-free synchronization, 1991

Plan for today

- Compare and swap (CAS) low-level atomicity
- Examples: AtomicInteger and NumberRange
- **How to implement a lock using CAS**
- Scalability: pessimistic locks vs optimistic CAS
- Treiber lock-free stack
- The ABA problem
- Progress concepts
 - Lock-free, wait-free, obstruction-free
- Course evaluation feedback

How to implement a lock using CAS

- Let's make a lock class in four steps:
- A: Simple TryLock
 - non-blocking tryLock and unlock, once per thread
- B: Reentrant TryLock
 - non-blocking tryLock and unlock, multiple times
- C: Simple Lock
 - blocking lock and unlock, once per thread
- D: Reentrant Lock = `j.u.c.locks.ReentrantLock`
 - blocking lock and unlock, multiple times per thread

```
class SimpleTryLock {  
    private final AtomicReference<Thread> holder  
        = new AtomicReference<Thread>();  
    public boolean tryLock() {  
        final Thread current = Thread.currentThread();  
        return holder.compareAndSet(null, current);  
    }  
    public void unlock() {  
        final Thread current = Thread.currentThread();  
        if (!holder.compareAndSet(current, null))  
            throw new RuntimeException("Not lock holder");  
    }  
}
```

Try to take
unheld lock

Release, if
holder

- If lock is free, **holder** is **null**
 - Thread can take lock only if **holder** is **null**
- If lock is held, **holder** is the holding thread
 - Only the holding thread can unlock

A philosopher using SimpleTryLock

```
while (true) {  
    int left = place, right = (place+1) % forks.length;  
    if (forks[left].tryLock()) {  
        try {  
            if (forks[right].tryLock()) {  
                try {  
                    System.out.print(place + " "); // Eat  
                } finally { forks[right].unlock(); }  
            }  
        } finally { forks[left].unlock(); }  
    }  
    try { Thread.sleep(10); } // Think  
    catch (InterruptedException exn) { }  
}
```

A fork is a
SimpleTryLock

TestCasLocks.java

- Very similar to Exercise 6.2.5
- Must unlock in **finally**, else an exception may cause the thread to never release lock

Reentrant TryLock, no blocking

```

class ReentrantTryLock {
    private final AtomicReference<Thread> holder = new Atomic...;
    private volatile int holdCount = 0; // valid if holder!=null
    public boolean tryLock() {
        final Thread current = Thread.currentThread();
        if (holder.get() == current) {
            holdCount++;
            return true;
        } else if (holder.compareAndSet(null, current)) {
            holdCount = 1;
            return true;
        }
        return false;
    }
    public void unlock() {
        final Thread current = Thread.currentThread();
        if (holder.get() == current) {
            holdCount--;
            if (holdCount != 0 || holder.compareAndSet(current, null))
                return;
        }
        throw new RuntimeException("Not lock holder");
    }
}

```

The diagram illustrates the execution flow of the `tryLock` and `unlock` methods. It uses callouts to explain the state of the lock holder (represented by a green box) at different points in the code.

- tryLock() Flow:**
 - If the current thread is the holder, the hold count is incremented, and the method returns `true`. Callout: "Already held by current thread".
 - If the current thread is not the holder and the lock is free, the lock is acquired, the hold count is set to 1, and the method returns `true`. Callout: "Unheld and we got it".
 - If the lock is held by another thread, the method returns `false`. Callout: "Held by other".
- unlock() Flow:**
 - If the current thread is the holder, the hold count is decremented. If it reaches 0 or the lock is free, the lock is released and the method returns. Callout: "We hold it, reduce count".
 - If the count is 0 and the lock is released, a `RuntimeException` is thrown. Callout: "If count is 0, release".

Simple Lock, with blocking

```

class SimpleLock {
    private final AtomicReference<Thread> holder = new Atomic...;
    final Queue<Thread> waiters = new ConcurrentLinkedQueue<Thread>();

    public void lock() {
        final Thread current = Thread.currentThread();
        waiters.add(current);
        while (waiters.peek() != current
            || !holder.compareAndSet(null, current))
        {
            LockSupport.park(this);
        }
        waiters.remove();
    }

    public void unlock() {
        final Thread current = Thread.currentThread();
        if (holder.compareAndSet(current, null))
            LockSupport.unpark(waiters.peek());
        else
            throw new RuntimeException("Not lock holder");
    }
}

```

Enter queue
waiting for lock

If first, & lock
free, take it ...

...else park

Got lock,
leave queue

Unpark first
parked thread

Parking a thread

- Static methods in `j.u.c.locks.LockSupport`:
 - `park()`, deschedule current thread until permit becomes available; do nothing if already available
 - `unpark(thread)`, makes permit available for `thread`, allowing it to be scheduled again
- A thread can call `park` to wait for a resource without consuming any resources
- Another thread can `unpark` it when the resource appears to be available again
- Similar to `wait/notifyAll`, but those work only for intrinsic locks

Taking care of thread interrupts

- Parking will *block* the thread
 - may be interrupted by `t.interrupt()` while parked
 - should preserve interrupted status till unparked

```
class SimpleLock {
    ...
    public void lock() {
        final Thread current = Thread.currentThread();
        boolean wasInterrupted = false;
        waiters.add(current);
        while (waiters.peek() != current
            || !holder.compareAndSet(null, current)) {
            LockSupport.park(this);
            if (Thread.interrupted())
                wasInterrupted = true;
        }
        waiters.remove();
        if (wasInterrupted)
            current.interrupt();
    }
}
```

TestCasLocks.java

If interrupted
while parked ...

... note that &
clear interrupt

... & set interrupt
when unparked

Reentrant Lock, with blocking

```

class MyReentrantLock {
    private final AtomicReference<Thread> holder = new AtomicRef...;
    final Queue<Thread> waiters = new ConcurrentLinkedQueue<Thread>();
    private volatile int holdCount = 0; // Valid if holder!=null
    public void lock() {
        final Thread current = Thread.currentThread();
        if (holder.get() == current)
            holdCount++;
        else {
            waiters.add(current);
            while (waiters.peek() != current
                || !holder.compareAndSet(null, current)) {
                LockSupport.park(this);
            }
            holdCount = 1;
            waiters.remove();
        }
    }
    public void unlock() { ... }
}

```

Already held by current thread

Enter queue waiting for lock

If first, & lock free, take it ...

...else park

Got lock, leave queue

- A cross between ReentrantTryLock and SimpleLock: both **holdCount** and **waiters**

Plan for today

- Compare and swap (CAS) low-level atomicity
- Examples: AtomicInteger and NumberRange
- How to implement a lock using CAS
- **Scalability: locks vs optimistic CAS**
- Treiber stack
- The ABA problem
- Progress concepts
 - Lock-free, wait-free, obstruction-free
- Course evaluation feedback

A CAS is machine instruction

- C#
- Bytecode
- x86 code
- Intel x86 Instruction Reference CMPXCHG:

```
Interlocked.CompareExchange(ref x, v, 65)
```

```
ldc.i4.s 0x41
Interlocked::Increment([out] int32&)
```

```
movl %eax, $0x00000041
lock/cmpxchgl (%rcx), %edx
```

InterLocked.cs

Compares the value in the EAX register with the first operand. If the two values are equal, the second operand is loaded into the first operand.

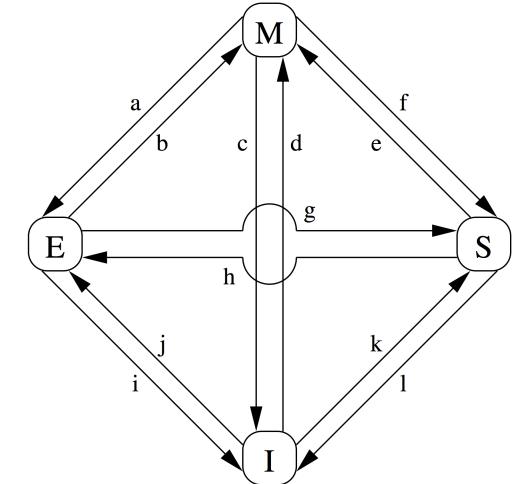
This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically. [...] the first operand receives a write cycle without regard to the result of the comparison. The first operand is written back if the comparison fails; otherwise, the second operand is written into the first one.

So CAS must be very fast?

- YES, it is fast
 - A successful CAS is faster than taking a lock
 - An unsuccessful CAS does not cause thread descheduling
- NO, it is slow
 - If many CPU cores try to CAS the same variable, then memory overhead may be very large
- Performancewise, like transactional memory
 - if mostly reads, then high concurrency
 - if many conflicting writes, then many retries

Week 8 flashback: MESI cache coherence protocol

A write in a non-exclusive state requires acknowledge ack* from *all other cores*



		Cause	I send	I rec	
M	a	(Send update to RAM)	writeback	-	CAS: many messages when other cores write same variable
E	b	Write	-	-	
M	c	Other wants to write	-	read inv	read resp, inv ack
I	d	Atomic read-mod-write	read inv	read resp, inv ack*	-
S	e	Atomic read-mod-write	read inv	inv ack*	-
M	f	Other wants to read	-	read	read resp
E	g	Other wants to read	-	read	read resp
S	h	Will soon write	inv	inv ack*	-
E	i	Other wants atomic rw	-	read inv	read resp, inv ack
I	j	Want to write	read inv	read resp, inv ack*	-
I	k	Want to read	read	read resp	-
S	l	Other wants to write	-	inv	inv ack

Scalability of locks and CAS: Pseudorandom number generation

```
class LockingRandom implements MyRandom {  
    private long seed;  
    public synchronized int nextInt() {  
        seed = (seed * 0x5DEECE66DL + 0xBL) & ((1L << 48) - 1);  
        return (int)(seed >>> 16);  
    }  
}
```

Lock-based

```
class CasRandom implements MyRandom {  
    private final AtomicLong seed;  
    public int nextInt() {  
        long oldSeed, newSeed;  
        do {  
            oldSeed = seed.get();  
            newSeed = (oldSeed * 0x5DEECE66DL + 0xBL) & ((1L << 48)-1);  
        } while (!seed.compareAndSet(oldSeed, newSeed));  
        return (int)(newSeed >>> 16);  
    }  
}
```

TestPseudoRandom.java

A la Goetz p. 327

CAS-based

- (Could one use **volatile** instead?)

Thread-locality is (more) important for scalability

```
class TLLockingRandom implements MyRandom {  
    private final ThreadLocal<MyRandom> myRandomGenerator;  
    public TLLockingRandom(final long seed) {  
        this.myRandomGenerator =  
            new ThreadLocal<MyRandom>() {  
                public MyRandom initialValue() {  
                    return new LockingRandom(seed);  
                } };  
    }  
    public int nextInt() {  
        return myRandomGenerator.get().nextInt();  
    }  
}
```

TestPseudoRandom.java

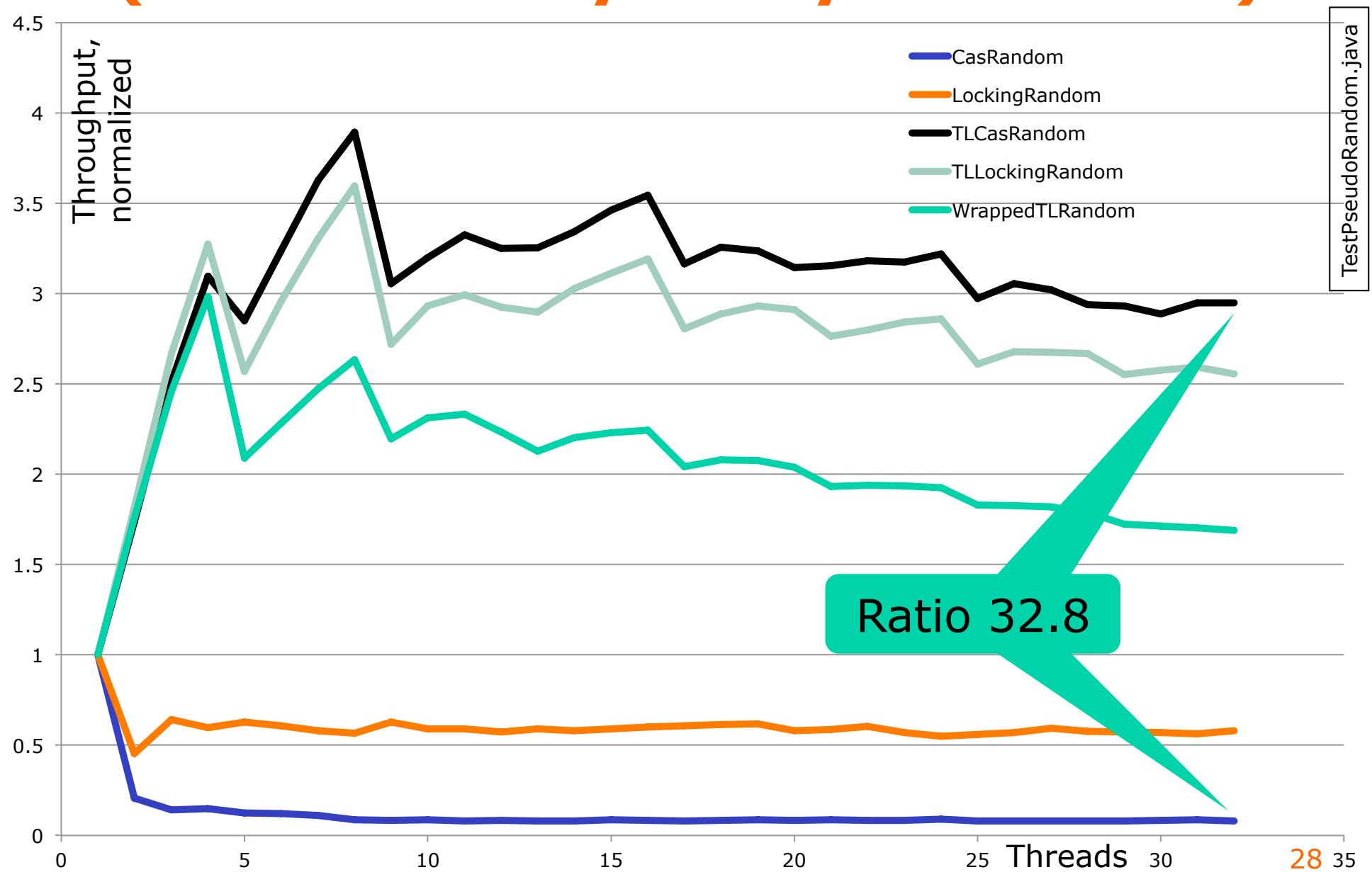
Create this
thread's
generator

Get this
thread's
generator

- A LockingRandom instance for each thread
- A thread's first call to `.get()` causes a call to `initialValue()` to create the instance
- Never access conflicts between threads

Goetz §3.3.3

Random number generator scalability (unrealistically heavy contention)



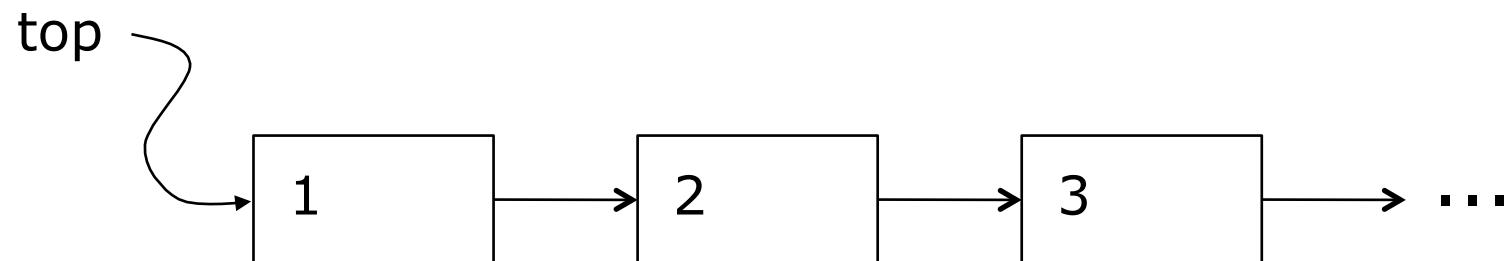
Plan for today

- Compare and swap (CAS) low-level atomicity
- Examples: AtomicInteger and NumberRange
- How to implement a lock using CAS
- Scalability: pessimistic locks vs optimistic CAS
- **Treiber lock-free stack**
- The ABA problem
- Progress concepts
 - Lock-free, wait-free, obstruction-free
- Course evaluation feedback

Treiber's lock-free stack (1986)

```
class ConcurrentStack <E> {  
    private static class Node <E> {  
        public final E item;  
        public Node<E> next;  
  
        public Node(E item) {  
            this.item = item;  
        }  
    }  
  
    AtomicReference<Node<E>> top = new AtomicReference<Node<E>>();  
    ...  
}
```

Goetz Listing 15.6



Treiber's stack operations

```
public void push(E item) {  
    Node<E> newHead = new Node<E>(item);  
    Node<E> oldHead;  
    do {  
        oldHead = top.get();  
        newHead.next = oldHead;  
    } while (!top.compareAndSet(oldHead, newHead));  
}
```

Set top to new
if not changed

```
public E pop() {  
    Node<E> oldHead, newHead;  
    do {  
        oldHead = top.get();  
        if (oldHead == null)  
            return null;  
        newHead = oldHead.next;  
    } while (!top.compareAndSet(oldHead, newHead));  
    return oldHead.item;  
}
```

Set top to next
if not changed

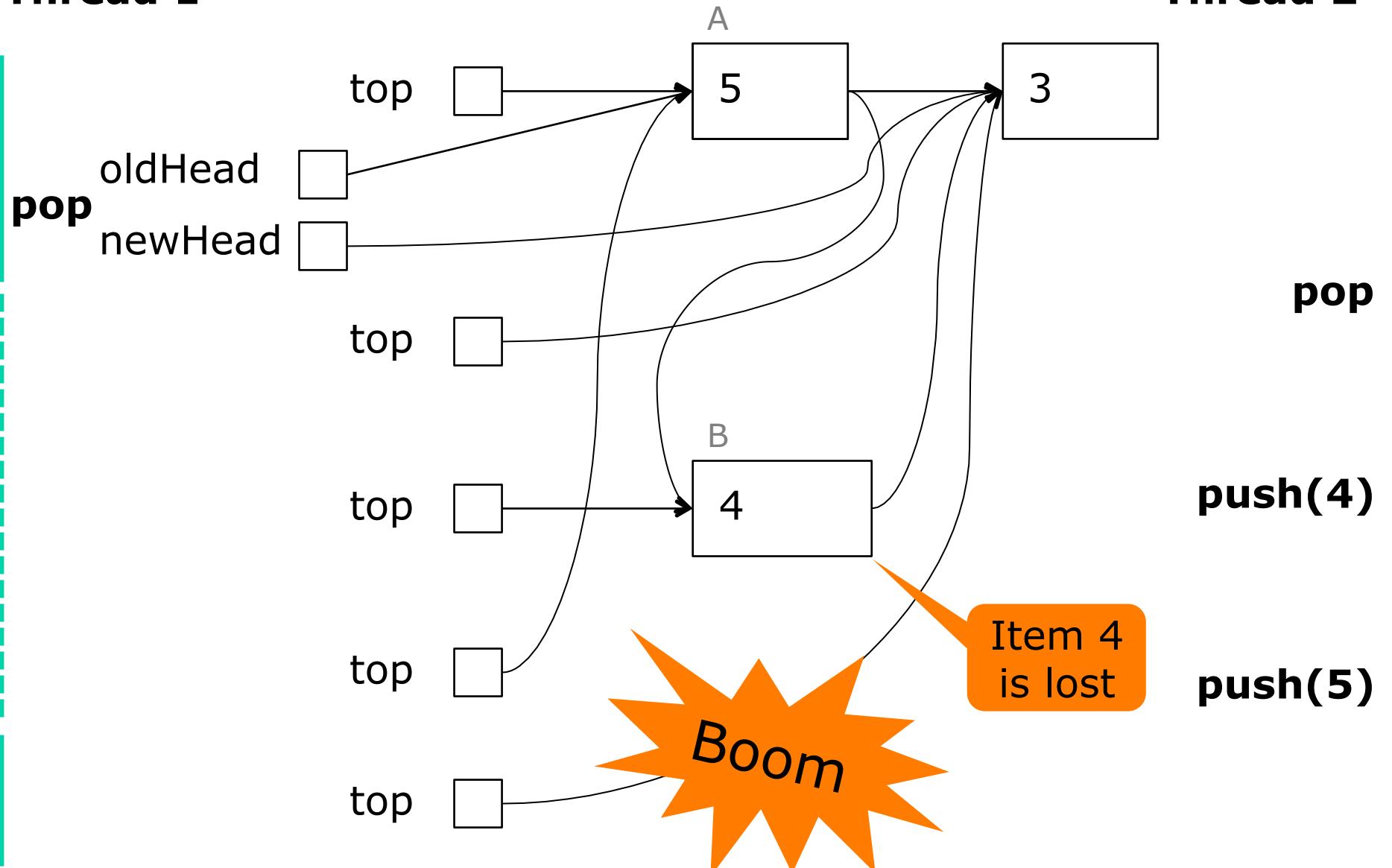
The ABA problem

- CAS variable has value A, then B, then A
 - Hence variable changed, but CAS did not see it
- Eg AtomicInteger was A, then add +b, add -b
 - Not a problem in MyAtomicInteger
- Typically a problem with pointers in C, C++
 - Reference p points at a struct; then free(p); then malloc() returns p, but now a different struct ...
- Standard solution: make pair (p,i) of pointer and integer counter; probabilistically correct
- Rarely an ABA-problem in Java, C#
 - Automatic memory management, garbage collector
 - So objects are not reused while referred to

ABA in Treiber stack à la C

Thread 1

Thread 2



Progress concepts

- *Non-blocking*: A call by thread A cannot prevent a call from thread B from completing
 - Not true for lock-based queue: A holds lock to `put()`, gets descheduled or crashes, while B wants to `take()` but cannot get lock
- *Wait-free*: Every call finishes in finite time
 - True for SimpleTryLock's `tryLock`
 - Not true for AtomicInteger's `getAndAdd`
- *Bounded wait-free*: Every ... in bounded time
- *Lock-free*: Some call finishes in finite time
 - True for AtomicInteger's `getAndAdd`
 - Any wait-free method is also lock-free
 - Lock-free is good enough in practice!

Goetz §15.4 and Herlihy & Shavit §3.7

Obstruction freedom

- *Obstruction-free*: If a method call executes alone, it finishes in finite time
 - Lock-based data structures are not obstruction-free
 - A “lock-free” method is also obstruction-free
 - Obstruction-free sounds rather weak, but in combination with back-off it ensures progress
 - Some people even think it too strong:

... we argue that obstruction-freedom is not an important property for software transactional memory, and demonstrate that, if we are prepared to drop the goal of obstruction-freedom, software transactional memory can be made significantly faster

Ennals 2006: STM should not be obstruction-free

Course evaluation

- General satisfaction with course and teachers
- However, perhaps too much overlap with ITU Software Development BSc program
- Possible actions, fall 2015
 - Compress some of the Threads & Locks stuff
 - Spend more time (> 5 weeks) on
 - transactional memory (week 10)
 - lock-free data structures (week 11-12)
 - message passing and actors (week 13-14)
 - other languages than Java (scattered)

Numerical results (n=32)

Question (6 = agree completely, 1 = disagree completely)	average
Overall: I am happy about this course	5.06
I see a close correlation between the course topics and the exam requirements	5.58
I sense a close correlation between the exam requirements and the exam form	5.61
I think the course is relevant for my future job profile	5.34
My time consumption for this course is too high [...]	3.44
I am satisfied with my effort on this course	4.84

Some PCPP-related thesis projects

- Design, implement and test concurrent versions of C5 collection classes for .NET
 - <http://www.itu.dk/research/c5/>
- The *Popular Parallel Programming (P3)* project
 - Static dataflow partitioning algorithms
 - Dynamic scheduling algorithms on .NET
 - Vector (SSE, AVX) .NET intrinsics for spreadsheets
 - Supercomputing with Excel and .NET
 - <http://www.itu.dk/people/sestoft/p3/>
- Investigate Java Pathfinder for test and coverage analysis of concurrent software
 - <http://babelfish.arc.nasa.gov/trac/jpf>

This week

- Reading
 - Goetz et al section 3.3.3 and chapter 15
 - Herlihy & Shavit chapter 11 and section 3.8
- Exercises
 - Show that you can implement a concurrent Histogram and a ReadWriteLock using CAS
- Read before next week
 - Goetz et al chapters 15 and 16
 - Michael & Scott 1996: Simple, fast, and practical non-blocking and blocking concurrent queue ...

Practical Concurrent and Parallel Programming 12

Peter Sestoft
IT University of Copenhagen

Friday 2014-11-21*

Plan for today

- Michael and Scott unbounded queue
- Perspective: Work-stealing dequeues
- Progress concepts
 - Wait-free, lock-free, obstruction-free
- Java Memory Model
- C#/.NET memory model
- Union-find data structure
- Possible parallel programming projects

Lock-based queue with sentinel

```
class LockingQueue<T> implements UnboundedQueue<T> {
    private Node<T> head, tail;

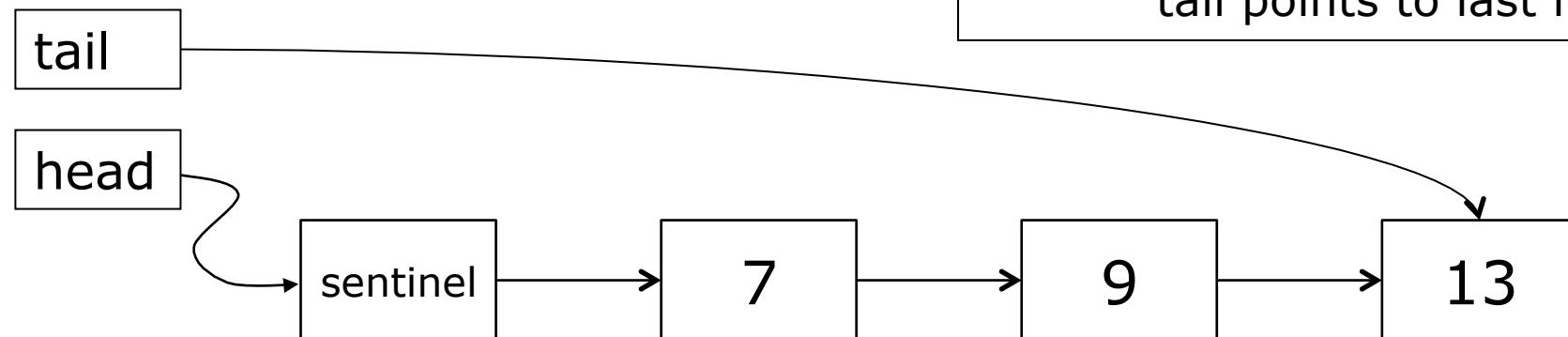
    public LockingQueue() {
        head = tail = new Node<T>(null, null);
    }

    ...
}
```

Make sentinel node

```
private static class Node<T> {
    final T item;
    Node<T> next;
}
```

Invariants:
 $\text{tail.next} = \text{null}$
 If empty, $\text{head} = \text{tail}$
 If non-empty: $\text{head} \neq \text{tail}$,
 head.next is first item,
 tail points to last item



Lock-based queue operations

```
public synchronized void enqueue(T item) {  
    Node<T> node = new Node<T>(item, null);  
    tail.next = node;  
    tail = node;  
}
```

Enqueue
at tail

TestMSqueue.java

```
public synchronized T dequeue() {  
    if (head.next == null)  
        return null;  
    Node<T> first = head;  
    head = first.next;  
    return head.item;  
}
```

Dequeue from
second node,
becomes new
sentinel

- Important property:
 - Enqueue (**put**) updates **tail** but not **head**
 - Dequeue (**take**) updates **head** but not **tail**

Michael-Scott lock-free queue, CAS

```
private static class Node<T> {
    final T item;
    final AtomicReference<Node<T>> next;
}
```

Michael and Scott: Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms, 1996

```
class MSQueue<T> implements UnboundedQueue<T> {
    private final AtomicReference<Node<T>> head, tail;

    public MSQueue() {
        Node<T> dummy = new Node<T>(null, null);
        head = new AtomicReference<Node<T>>(dummy);
        tail = new AtomicReference<Node<T>>(dummy);
    }
}
```

TestMSqueue.java

- If non-empty:
 - **head.next** is first item, **tail** points to last item ("quiescent state") or the second-last item ("intermediate state")

Intermediate state and "help"

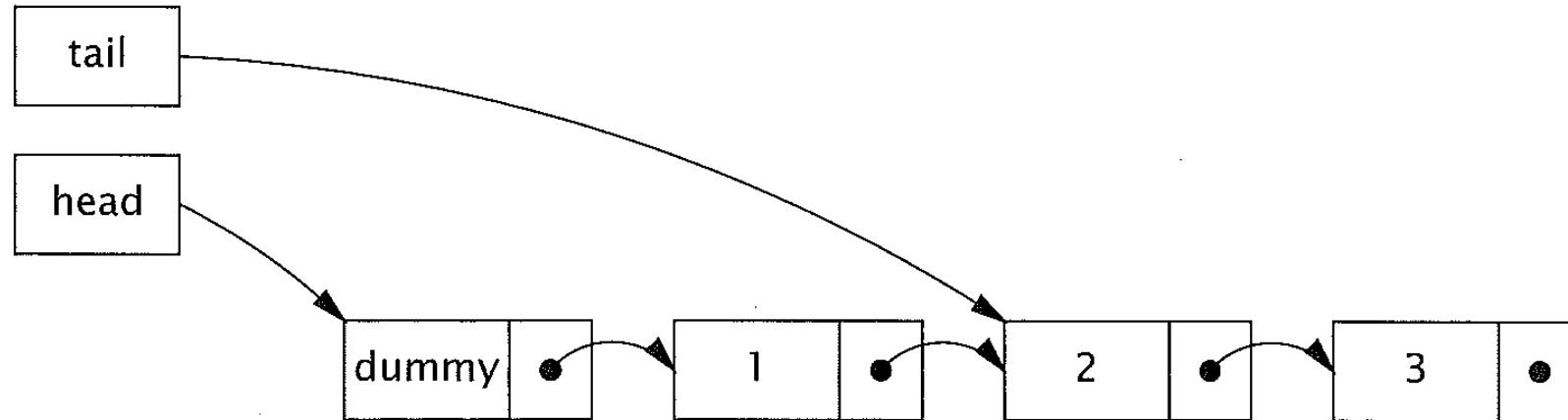


FIGURE 15.4. Queue in intermediate state during insertion.

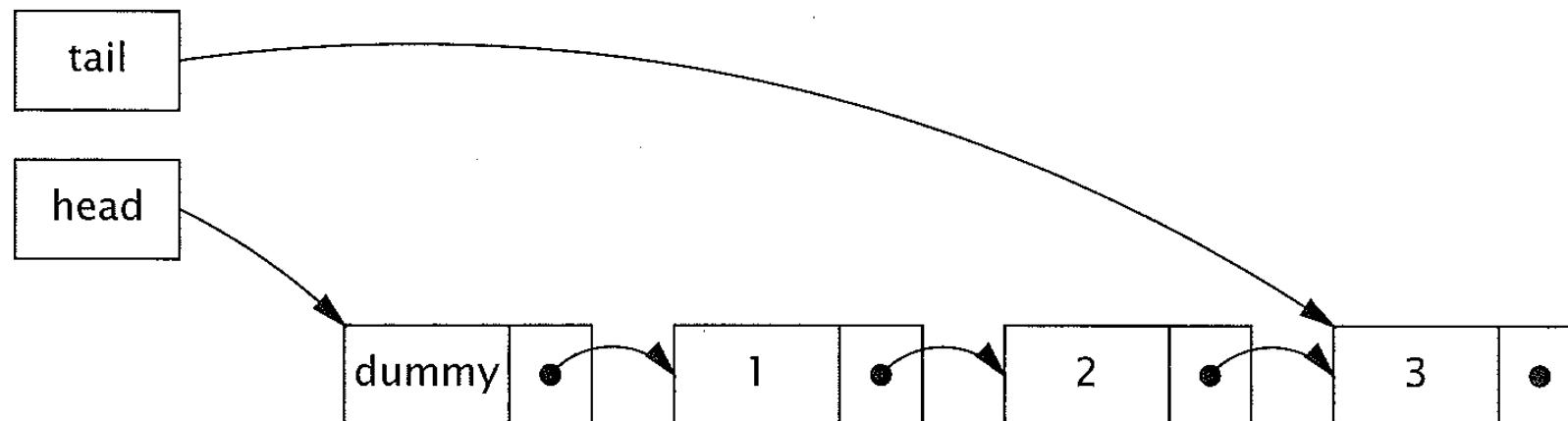
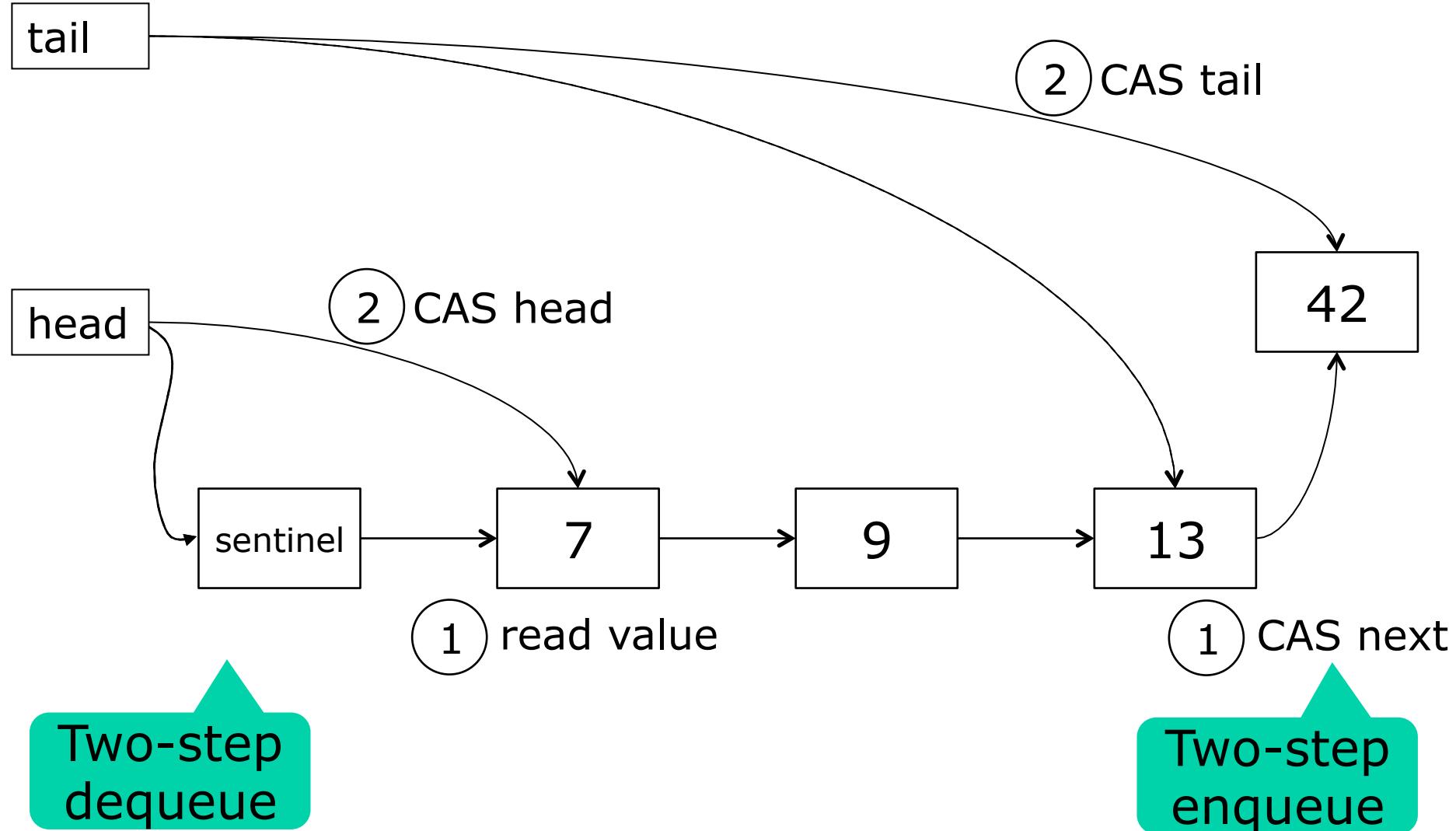


FIGURE 15.5. Queue again in quiescent state after insertion is complete.

Michael & Scott queue operations



Michael-Scott dequeue (take)

```

public T dequeue() {
    while (true) {
        Node<T> first = head.get(),
                last = tail.get(),
                next = first.next.get();
        if (first == head.get()) {
            if (first == last) {
                if (next == null)
                    return null;
                else
                    tail.compareAndSet(last, next);
            } else {
                T result = next.item; ①
                if (head.compareAndSet(first, next)) {
                    return result;
                }
            }
        }
    }
}

```

Needed?

Intermediate,
try move tail (*)

1

2

Try move
head

In Java or C#,
but not C/C++,
(1) can go after (2)

Michael-Scott enqueue (put)

```

public void enqueue(T item) { // at tail
    Node<T> node = new Node<T>(item, null);
    while (true) {
        Node<T> last = tail.get(),
        Needed?           next = last.next.get();
        if (last == tail.get()) {
            if (next == null) {
                if (last.next.compareAndSet(next, node)) {
                    tail.compareAndSet(last, node);
                    return;
                }
            } else {
                tail.compareAndSet(last, next);
            }
        }
    }
}

```

Quiescent, try add

Success, try move tail

1

Intermediate, try move tail

2

"help another enqueueer"

(*) Why must dequeue mess with the tail?

Q 2

Queue is empty,
head==tail

A: enqueue(7)

A: update a.next

B: dequeue()

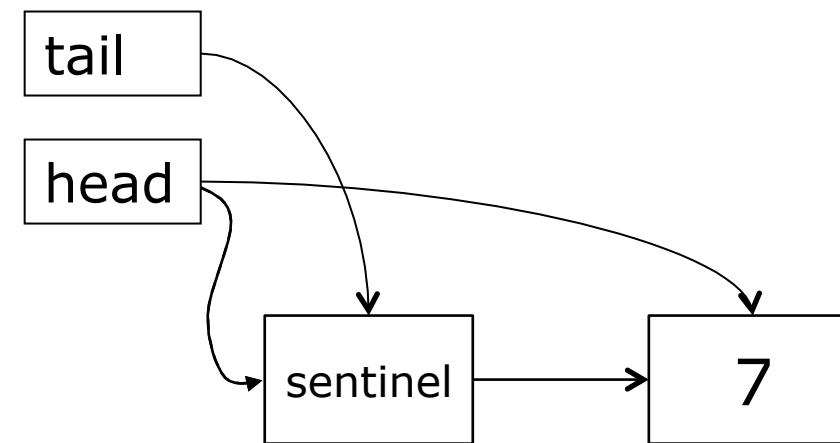
B: update head

Now tail lags behind
head, not good

So next dequeue
should move tail
before moving head

```
while (true) {  
    ...  
    if (first == last) {  
        if (next == null)  
            return null;  
        else  
            tail.compareAndSet(last, next);  
    } else ...  
}
```

Intermediate,
try move tail



TestMSqueue.java

After Herlihy & Shavit p. 233

Understanding Michael-Scott queue

- Linearizable, with linearization points:
 - enqueue: successful CAS at E9
 - dequeue returning null: D3
 - dequeue returning item: successful CAS at D13
- Lineariz'n point = where method takes effect

```
public void enqueue(T item) { // at tail
    Node<T> node = new Node<T>(item, null);
    while (true) {
        Node<T> last = tail.get(),
               next = last.next.get();
        if (last == tail.get()) { // E7
            if (next == null) {
                if (last.next.compareAndSet(next, node)) {
                    tail.compareAndSet(last, node);
                    return;
                }
            } else
                tail.compareAndSet(last, next);
        }
    }
}
```

E9

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

D13

Nice, but ... needs a lot of AtomicReference objects

Q 3

```
private static class Node<T> {  
    final T item;  
    final AtomicReference<Node<T>> next;  
  
    public Node(T item, Node<T> next) {  
        this.item = item;  
        this.next = new AtomicReference<Node<T>>(next);  
    }  
}
```

Must be
CAS'able

One AR
per Node

Q 2

```
private static class Node<T> {  
    final T item;  
    volatile Node<T> next;  
    ...  
}
```

Better, no
AtomicReference
object needed

Instead, make
an "updater"

```
private final AtomicReferenceFieldUpdater<Node<T>, Node<T>> nextUpdater  
= AtomicReferenceFieldUpdater.newUpdater((Class<Node<T>>) (Class<?>) (Node.class),  
                                         (Class<Node<T>>) (Class<?>) (Node.class),  
                                         "next");
```

12

A la Goetz p. 335

Michael-Scott enqueue, using the "updater" for `last.next`

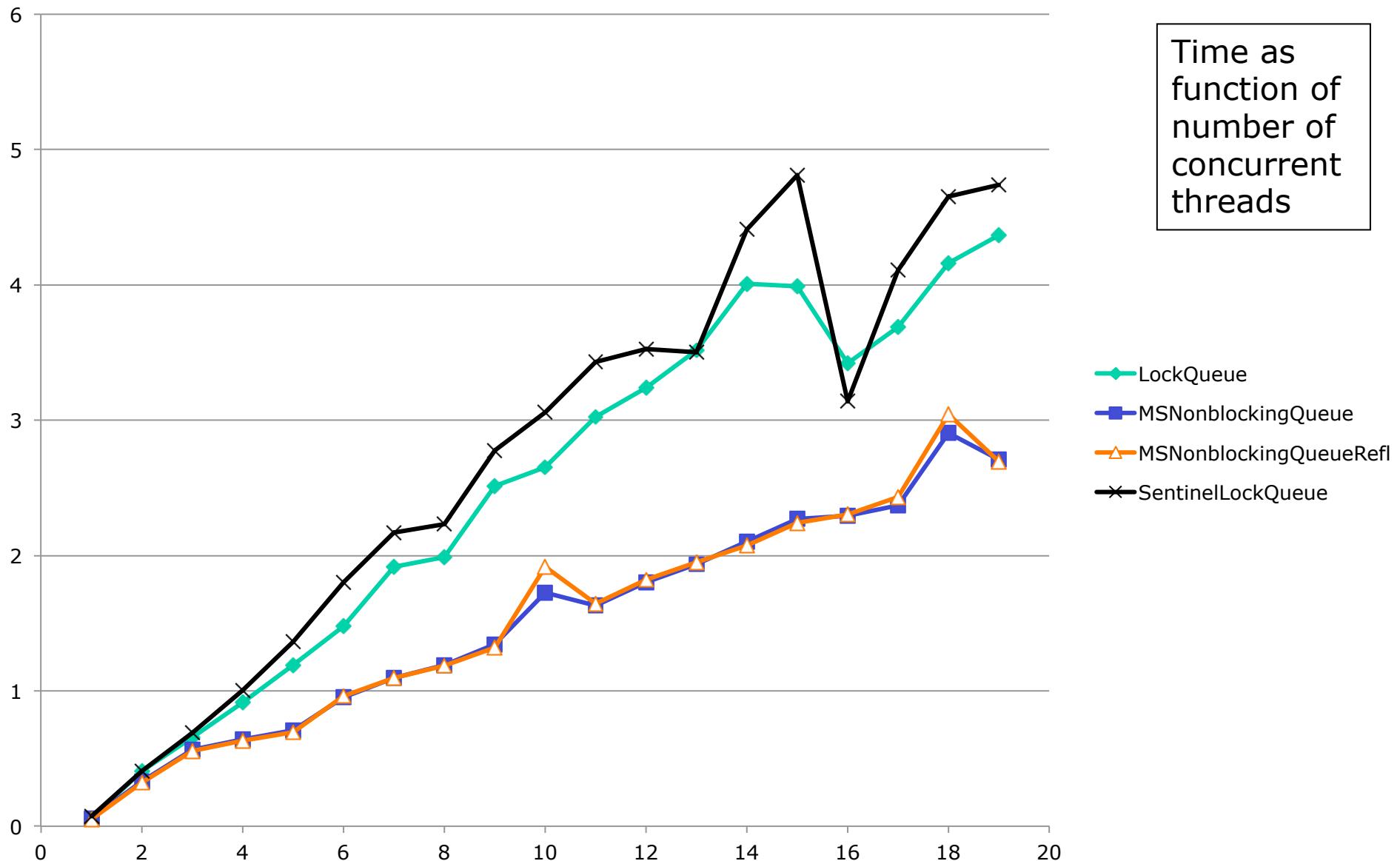
```
public void enqueue(T item) { // at tail
    Node<T> node = new Node<T>(item, null);
    while (true) {
        Node<T> last = tail.get(), next = last.next;
        if (last == tail.get()) {
            if (next == null) {
                if (nextUpdater.compareAndSet(last, next, node)) {
                    tail.compareAndSet(last, node);
                    return;
                }
            } else {
                tail.compareAndSet(last, next);
            }
        }
    }
}
```

If “next” field of
last equals
next, set to **node**

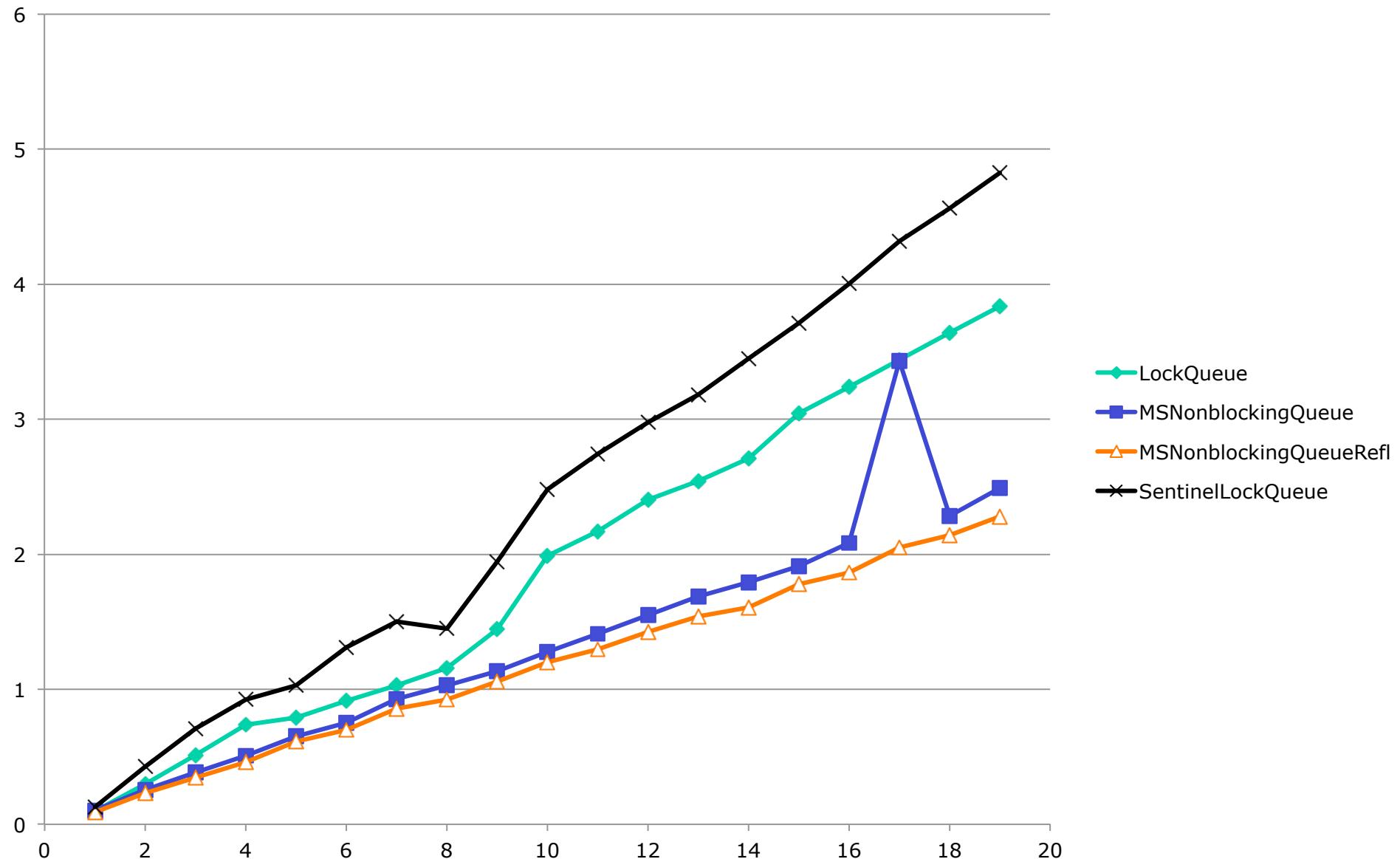
Queue benchmarks

- Queue implementations
 - Lock-based
 - Lock-based, sentinel node
 - Lock-free, sentinel node, `AtomicReference`
 - Lock-free, sentinel node, `AtomicReferenceFieldUpdater`
- Platforms
 - Hotspot 64 bit Java 1.7.0_b147, Windows 7, Xeon W3505, 2.53GHz, 2 cores, 2009Q1
 - Hotspot 64 bit Java 1.6.0_37, MacOS, Core 2 Duo, 2.66GHz, 2 cores, 2008Q1
 - Icedtea Java 1.7.0_b21, Linux, Xeon E5320, 1.86GHz, 4/8 cores, 2006Q4
 - Hotspot 64 bit Java 1.7.0_25-b15, Linux, AMD Opteron 6386 SE, 32 cores, 2012Q4
- Measurements probably flawed: the client threads do no useful work, only en/dequeue
- Nevertheless, **big** differences between machines

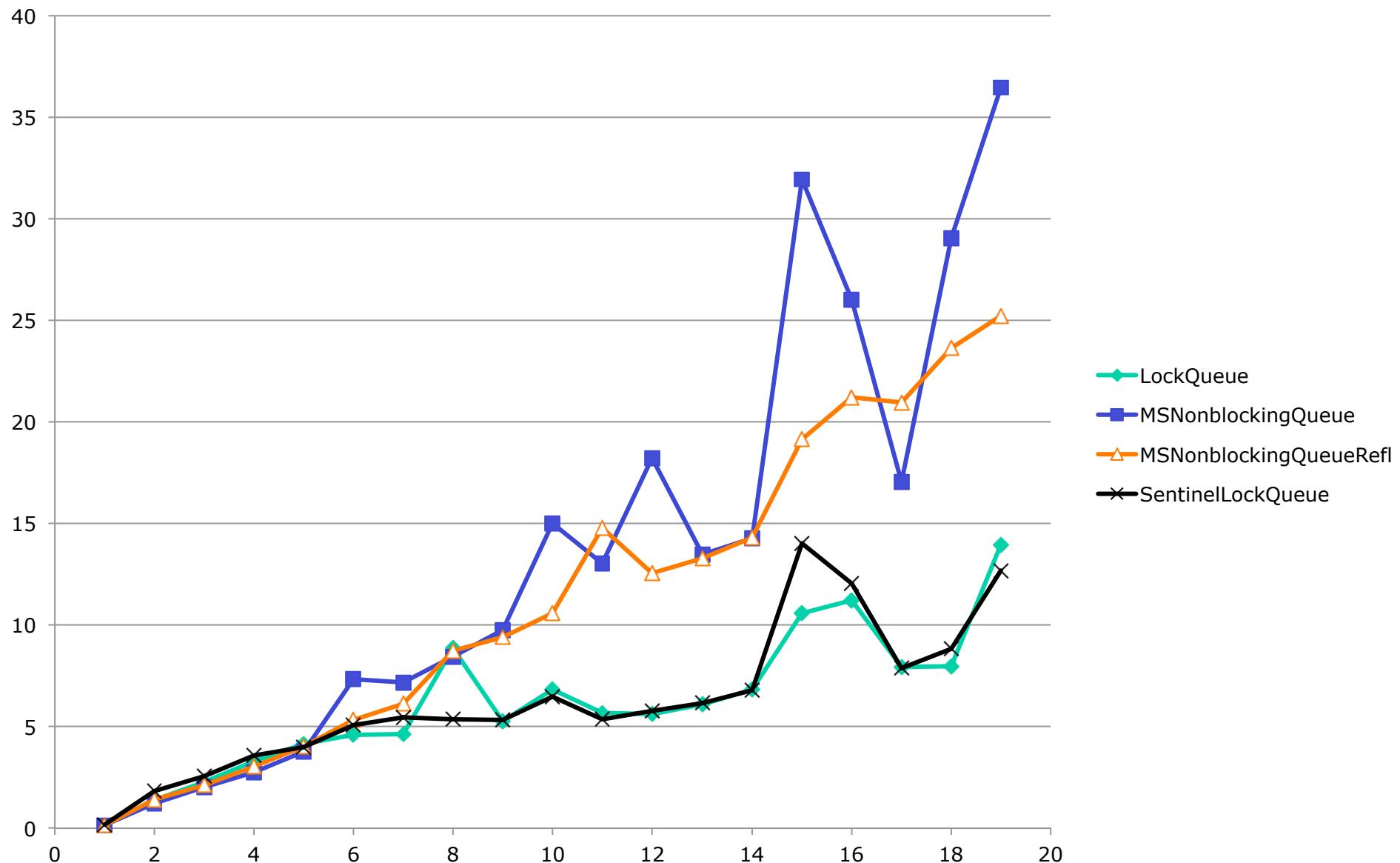
Java 1.7, Xeon W3505, 2 cores



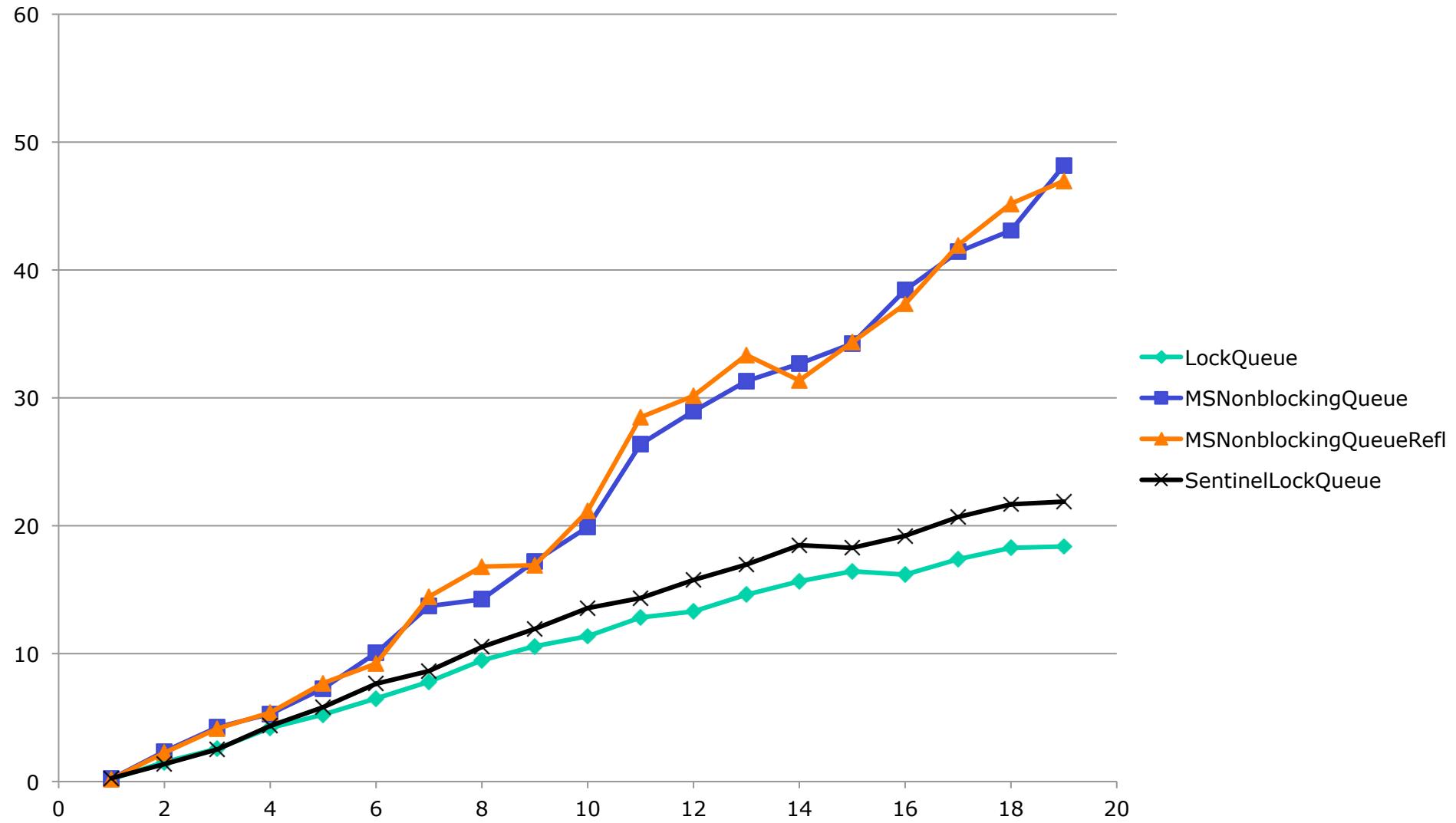
Java 1.6, Core 2 Duo, 2 cores



Java 1.7, Xeon E5320, 4/8 cores



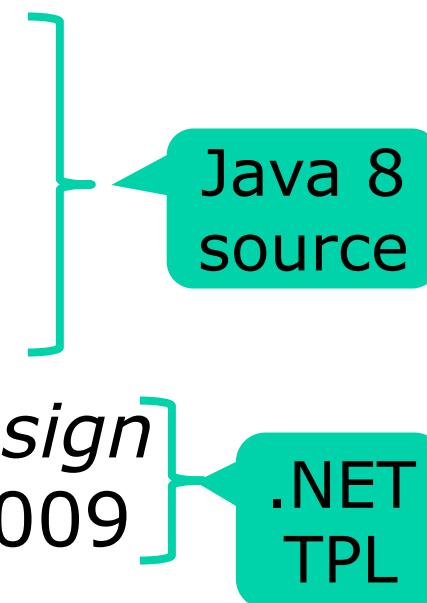
Java 1.7, AMD Opteron, 32 cores



Plan for today

- Michael and Scott unbounded queue
- **Perspective: Work-stealing dequeues**
- Progress concepts
 - Wait-free, lock-free, obstruction-free
- Java Memory Model
- C#/.NET memory model
- Union-find data structure
- Possible parallel programming projects

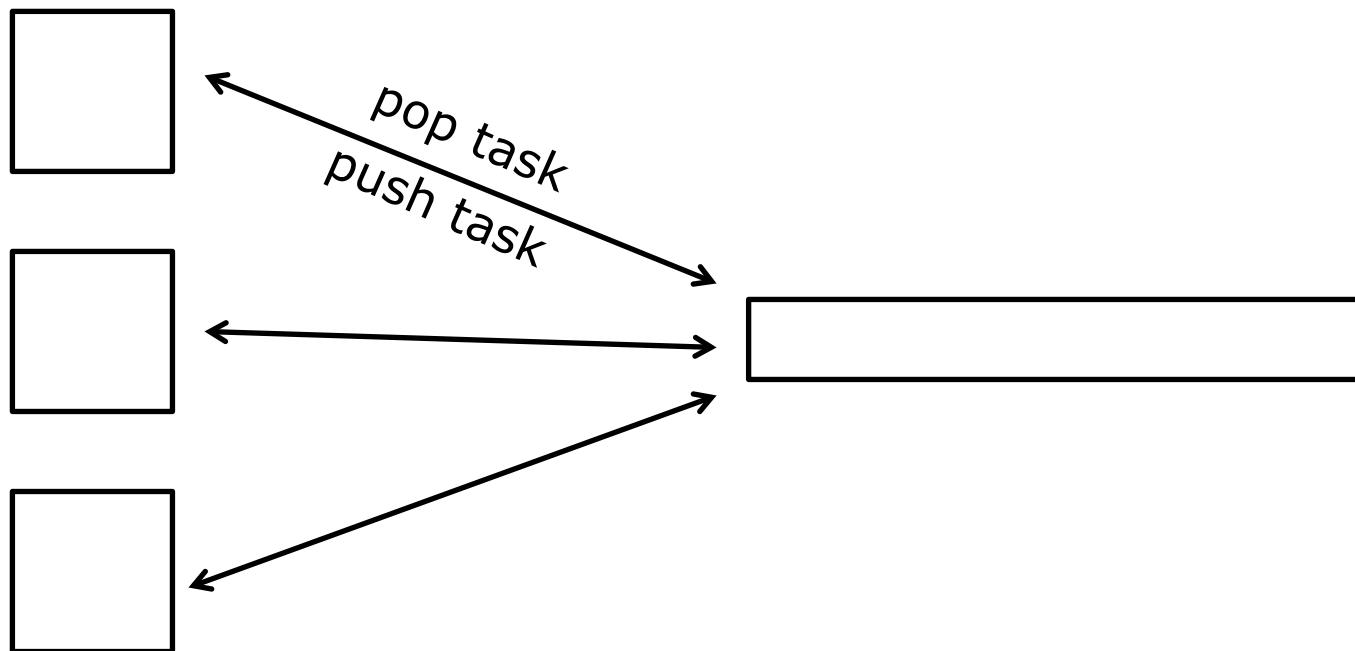
Perspective: Work-stealing dequeues

- Double-ended concurrent queues
 - Used to implement
 - Java 7's Fork-Join framework, and Akka (wk 13-14)
 - Java 8's newWorkStealingPool executor
 - .NET 4.0 Task Parallel Library
 - Chase and Lev: *Dynamic circular work-stealing queue*, SPAA 2005
 - Michael, Vechev, Saraswat: *Idempotent work stealing*, PPoPP 2009
 - Leijen, Schulte, Burckhardt: *The design of a task parallel library*, OOPSLA 2009
- 
- Java 8 source
- .NET TPL

A worker/task framework

Worker
threads

Common task queue

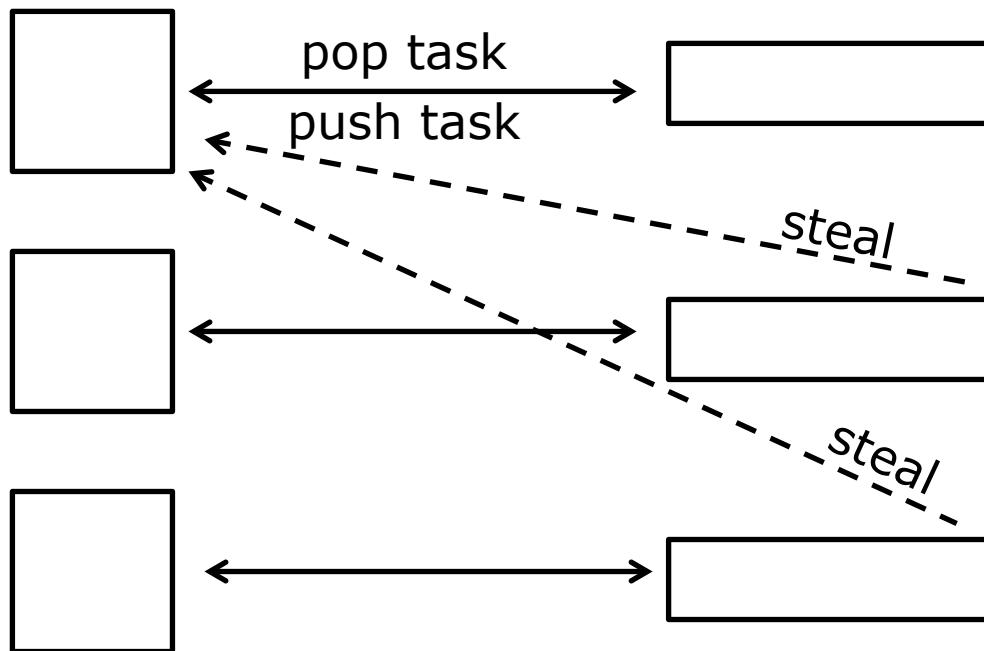


- Worker threads pop and push tasks on queue
- **Not scalable** because single queue is used by many threads

Better worker/task framework

Worker
threads

Thread-local work-
stealing dequeues



```
interface WSDeque<T> {  
    void push(T item);  
    T pop();  
    T steal();  
}
```

- Fewer memory write conflicts:
 - Most queue accesses are from local thread only
 - Pop from bottom, steal from top, conflicts are rare
- **Much better scalability**

Plan for today

- Michael and Scott unbounded queue
- Perspective: Work-stealing dequeues
- **Progress concepts**
 - **Wait-free, lock-free, obstruction-free**
- Java Memory Model
- C#/.NET memory model
- Union-find data structure
- Possible parallel programming projects

Progress concepts

- *Non-blocking*: A call by thread A cannot prevent a call from thread B from completing
 - Not true for lock-based queue: A holds lock to `put()`, gets descheduled or crashes, while B wants to `take()` but cannot get lock
- *Wait-free*: Every call finishes in finite time
 - True for SimpleTryLock's `tryLock`
 - Not true for AtomicInteger's `getAndAdd`
- *Bounded wait-free*: Every ... in bounded time
- *Lock-free*: Some call finishes in finite time
 - True for AtomicInteger's `getAndAdd`
 - Any wait-free method is also lock-free
 - Lock-free is good enough in practice!

Goetz §15.4 and Herlihy & Shavit §3.7

Obstruction freedom

- *Obstruction-free*: If a method call executes alone, it finishes in finite time
 - Lock-based data structures are not obstruction-free
 - A *lock-free* method is also obstruction-free
 - Obstruction-free sounds rather weak, but in combination with back-off it ensures progress
 - Some people even think it too strong:

... we argue that obstruction-freedom is not an important property for software transactional memory, and demonstrate that, if we are prepared to drop the goal of obstruction-freedom, software transactional memory can be made significantly faster

Ennals 2006: STM should not be obstruction-free

Plan for today

- Michael and Scott unbounded queue
- Perspective: Work-stealing dequeues
- Progress concepts
 - Wait-free, lock-free, obstruction-free
- **Java Memory Model**
- **C#/.NET memory model**
- Union-find data structure

- Possible parallel programming projects

Why do I need a memory model?

- Threads in Java and C# and C etc *communicate* via mutable shared *memory*
- We need compiler optimizations for speed
 - Compiler optimizations that are harmless in thread A may seem strange from thread B
 - Disallowing strangeness leads to slow software
- We need CPU caches for speed
 - With caches, write-to-RAM order may seem strange
- So we have to live with some strangeness
- A memory model tells *how much* strangeness
- The Java Memory Model is quite well-defined
 - JLS §17.4, Goetz §16, Herlihy & Shavit §3.8

The happens-before relation in Java

- A *program order* of a thread t is some total order of the thread's actions that is **consistent with the intra-thread semantics** of t
- Action x *synchronizes-with* action y is defined as follows:
 - An unlock action on **monitor** m *synchronizes-with* all subsequent lock actions on m
 - A write to a **volatile variable** v *synchronizes-with* all subsequent reads of v by any thread
 - An action that **starts a thread** *synchronizes-with* the first action in the thread it starts
 - The write of the **default value** (zero, false, or null) to each variable *synchronizes-with* the first action in every thread
 - The **final action in a thread** T_1 *synchronizes-with* any action in another thread T_2 that detects that T_1 has terminated
 - If thread T_1 **interrupts** thread T_2 , the interrupt by T_1 *synchronizes-with* any point where any other thread (including T_2) determines that T_2 has been interrupted
- Action x *happens-before* action y , written $hb(x, y)$, is defined:
 - If x and y are actions of the same thread and x comes before y in *program order*, then $hb(x, y)$
 - There is a *happens-before* edge from the end of a **constructor of an object** to the start of a finalizer for that object
 - If an action x *synchronizes-with* a following action y , then we also have $hb(x, y)$
 - If $hb(x, y)$ and $hb(y, z)$, then $hb(x, z)$ – that is, hb is **transitive**

Strange but legal behavior in Java

- Java Language Specification, sect 17.4:
 - Run these code fragments in two threads
 - Shared fields A, B initially 0; local variables r1, r2

Thread 1

```
r2=A;  
B=1;
```

Thread 2

```
r1=B;  
A=2;
```

- What are the possible results?
 - Strangely, r1==1 and r2==2 is possible
 - An ordering consistent with *happens-before* relation

```
B=1;  
A=2;  
r2=A;  
r1=B;
```

JLS 8 Tables 17.1, 17.5

Why permit such strange behaviors?

- More comprehensible example from JLS 17.4
 - Assume p, q shared, p==q and p.x==0

```
r1 = p;  
r2 = r1.x;  
r3 = q;  
r4 = r3.x;  
r5 = r1.x;
```

Thread 1

```
r6 = p;  
r6.x = 3;
```

Thread 2

- Compiler optimization, common subexpr. elimin.:

```
r1 = p;  
r2 = r1.x;  
r3 = q;  
r4 = r3.x;  
r5 = r2;
```

NB!

```
r6 = p;  
r6.x = 3;
```

(p.x seems to switch from r2=0 to r4=3 and back to r5=0

- Using **volatile** x prevents this strangeness

Cost of volatile (week 4 flashback)

```
class IntArrayVolatile {  
    private volatile int[] array;  
    public IntArray(int length) { array = new int[length]; ... }  
    public boolean isSorted() {  
        for (int i=1; i<array.length; i++)  
            if (array[i-1] > array[i])  
                return false;  
        return true;  
    }  
}
```

TestVolatileCost.java

IntArray	3.4 us	0.01	131072
IntArrayVolatile	17.2 us	0.14	16384

VolatileArray.cs

- In Java, volatile read is 5 x slower in this case
- C#/.NET 4.5, volatile read only 1.2 x slower
 - But still 3.7 x slower than Java non-volatile ...
- Mono .NET performs no optim., so no slower

Volatile prevents JIT optimizations

- For-loop body of `isSorted`, JITted x86 code:

```
0xfffff0: mov    0xc(%rsi),%r8d          ; LOAD %r8d = array field
0xfffff4: mov    %r10d,%r9d
0xfffff7: dec    %r9d
0xfffffa: mov    0xc(%r12,%r8,8),%ecx
0xffffff: cmp    %ecx,%r9d
0xe0002: jae    0xe004b
0xe0004: mov    0xc(%rsi),%ecx
0xe0007: lea    (%r12,%r8,8),%r11
0xe000b: mov    0xc(%r11,%r10,4),%r11d
0xe0010: mov    0xc(%r12,%rcx,8),%r8d
0xe0015: cmp    %r8d,%r10d
0xe0018: jae    0xe006d
0xe001a: lea    (%r12,%rcx,8),%r8
0xe001e: mov    0x10(%r8,%r10,4),%r9d
0xe0023: cmp    %r9d,%r11d
0xe0026: jg     0xe008d
0xe0028: mov    0xc(%rsi),%r8d
0xe002c: inc    %r10d
; i NOW IN %r9d
; i-1 IN %r9d
; LOAD %ecx = array.length
; INDEX CHECK array.length <= i-1
; IF SO, THROW
; LOAD %ecx = array field
; LOAD %r11 = array base address
; LOAD %r11d = arr[i-1]
; LOAD %r8d = array.length
; INDEX CHECK array.length <= i
; IF SO, THROW
; LOAD %r8 = array base address
; LOAD %r9d = array[i]
; IF arr[i] < arr[i-1]
; RETURN FALSE
; LOAD %r8d = array field
; i++
```

array volatile

3 reads of array field

2 index checks

VolatileArray.java

- Non-volatile: read `arr` once, unroll loop, ...:

```
0xcb9: mov    0xc(%rdi,%r11,4),%r8d
0xcbe: mov    0x10(%rdi,%r11,4),%r10d
0xcc3: cmp    %r10d,%r8d
0xcc6: jg     0xd85
; LOAD %rd8d = array[i-1]
; LOAD %rd10d = array[i]
; IF array[i] > array[i-1]
; RETURN FALSE
```

array not volatile

32

C#/.NET memory model?

- Quite similar to Java
 - *C# Language Specification*, Ecma-334 standard
- But weaknesses and unclarities
 - C# **readonly** has no visibility effect unlike **final**
 - C# **volatile** is weaker than in Java
 - Allowed to lift variable read out of loop?
 - “Read introduction” seems downright horrible!
- If you write concurrent C# programs, read:
 - Ostrovsky: The C# Memory Model in Theory and Practice, MSDN Magazine, December 2012
 - Even though optional in this course

- Visibility effect of C#/.NET **readonly** fields not mentioned in C# Language Specification or Ecma-335 CLI Specification (**initonly**)
- In fact, no visibility guarantee is intended...

Right. The CLI doesn't give any special status to **initonly** fields, from a memory ordering/visibility perspective. As with ordinary fields, if they are shared between threads then some sort of fence is needed to ensure consistency. This could be in the form of a volatile write, as Carol suggests, or any of the common synchronization primitives such as releasing a lock, setting an event, etc.

Eric

-----Original Message-----

From: Carol Eidt
Sent: Tuesday, December 4, 2012 10:14 AM
To: Peter Sestoft; Mads Torgersen; Eric Eilebrecht
Cc: Carol Eidt
Subject: RE: Visibility (from other threads) of readonly fields in C#/.NET?

Hi Peter,

I apologize for not responding more quickly to your email. I am adding Eric Eilebrecht to this thread, since he is the CLR's memory ordering expert.

I believe that section I.12.6.4 Optimization addresses this, but one has to read between the lines:

"Conforming implementations of the CLI are free to execute programs using any technology that guarantees, within a single thread of execution, that side-effects and exceptions generated by a thread are visible in the order specified by the CIL. For this purpose only volatile operations (including volatile reads) constitute visible side-effects. (Note that while only volatile operations constitute visible side-effects, volatile operations also affect the visibility of non-volatile references.)"

Where it says " volatile operations also affect the visibility of non-volatile references", this implies (though vaguely) that volatile reads & writes behave as some form of memory fence, though whether it is bi-directional or acquire-release is left unstated. I also believe that the above implies that, in order to achieve the desired visibility of **initonly** fields, one would have to declare a volatile field that would be written last, effectively "publishing" the other fields.

I certainly wouldn't say that the Java memory model make too much fuss over this - it's just fundamentally tricky!

Carol

C#/.NET volatile weaker than Java's

```
class StoreBufferExample {  
    volatile bool A = false;  
    volatile bool B = false;  
    volatile bool A_Won = false;  
    volatile bool B_Won = false;  
    public void ThreadA() {  
        A = true;  
        if (!B)  
            A_Won = true;  
    }  
    public void ThreadB() {  
        B = true;  
        if (!A)  
            B_Won = true;  
    }  
}
```

```
public void ThreadA() {  
    A = true;  
    Thread.MemoryBarrier();  
    if (!B)  
        aWon = 1;  
}
```

```
public void ThreadB() {  
    B = true;  
    Thread.MemoryBarrier();  
    if (!A)  
        B_Won = true;  
}
```

TestVolatile.cs

Ostrovsky 2013

- C#: possible to get **A_Won = B_Won = true !**
 - Not JIT compiler, but CPU store buffer delay on A
 - To fix in C#, add MemoryBarrier call (no Java equ.)

C# volatile vs Java volatile

- A read of a volatile field is called a volatile read. A volatile read has “acquire semantics”; that is, it is guaranteed to occur prior to any references to memory that occur after it in the instruction sequence.
- A write of a volatile field is called a volatile write. A volatile write has “release semantics”; that is, it is guaranteed to happen after any memory references prior to the write instruction in the instruction sequence.

- A C# volatile read may move earlier, a volatile write may move later, hence trouble
- Not in Java:

If a programmer protects all accesses to shared data via locks or declares the fields as volatile, she can forget about the Java Memory Model and assume interleaving semantics, that is, Sequential Consistency.

MemoryBarrier() in C#/.NET

Synchronizes memory access as follows: The processor executing the current thread cannot reorder instructions in such a way that memory accesses prior to the call to MemoryBarrier execute after memory accesses that follow the call to MemoryBarrier.

MemoryBarrier is required only on multiprocessor systems with weak memory ordering (for example, a system employing multiple Intel Itanium processors).

[System.Threading.Thread.MemoryBarrier API docs 4.5](#)

- But seems sometimes to be needed anyway
 - also on x86
- Java does not have such a method, because Java **volatile** gives better guarantees

Plan for today

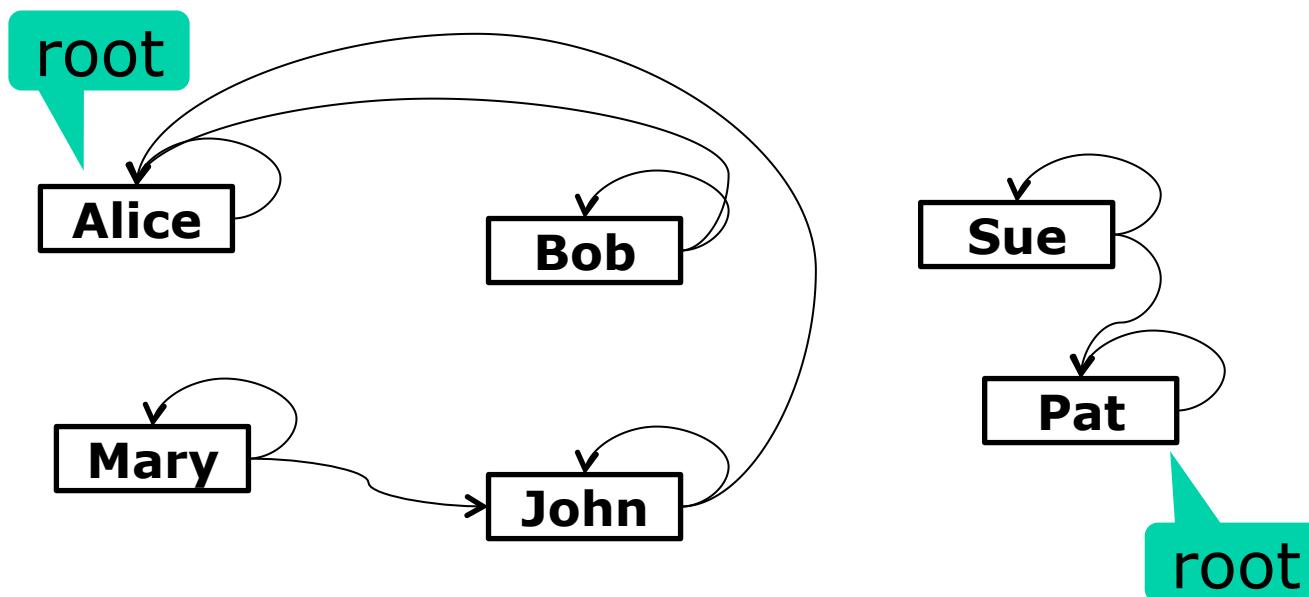
- Michael and Scott unbounded queue
- Perspective: Work-stealing dequeues
- Progress concepts
 - Wait-free, lock-free, obstruction-free
- Java Memory Model
- C#/.NET memory model
- **Union-find data structure**

- Possible parallel programming projects

The union-find data structure

- Efficient way to maintain equivalence classes
- Used in
 - type inference in compilers: F#, Scala, C# ...
 - image segmentation
 - network analysis: chips, WWW, Facebook friends ...
- Example: family relations, who are related?

Tarjan: Data structures and network algorithms, 1983



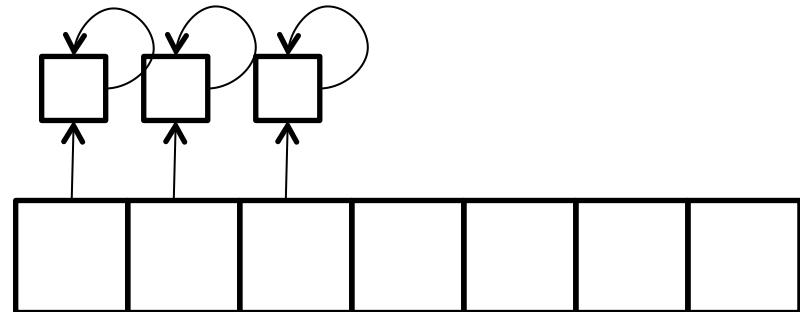
Sue is Pat's sister
Alice is Bob's sister
Mary is John's mother
Mary is Bob's mother

Are Sue and Mary related?

Three union-find implementations

- A: Coarse-locking = Synchronized methods
- B: Fine-locking = Lock on each set partition
- C: Wait-free = Optimistic, CAS-based

```
interface UnionFind {  
    int find(int x);  
    void union(int x, int y);  
    boolean sameSet(int x, int y);  
}
```



```
class Node {  
    volatile int  
    next, rank;  
}
```

```
class CoarseUnionFind implements UnionFind {  
    private final Node[] nodes;  
  
    public CoarseUnionFind(int count) {  
        this.nodes = new Node[count];  
        for (int x=0; x<count; x++)  
            nodes[x] = new Node(x);  
    }  
}
```

TestUnionFind.java

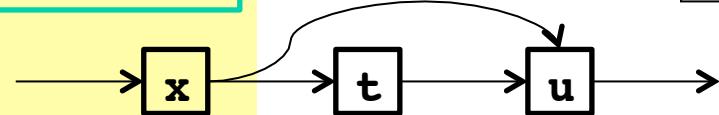
Coarse-locking union-find

```

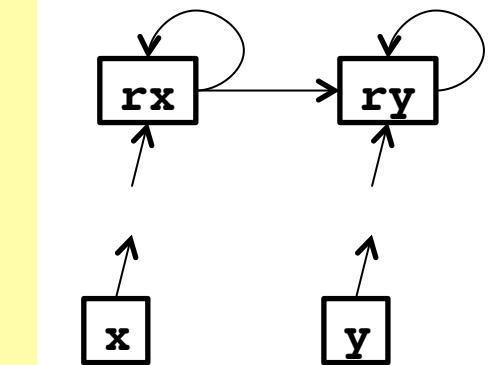
class CoarseUnionFind implements UnionFind {
    private final Node[] nodes;
    public synchronized int find(int x) {
        while (nodes[x].next != x) {
            final int t = nodes[x].next, u = nodes[t].next;
            nodes[x].next = u;
            x = u;
        }
        return x;
    }
    public synchronized void union(int x, int y) {
        int rx = find(x), ry = find(y);
        if (rx == ry)
            return;
        if (nodes[rx].rank > nodes[ry].rank) {
            int tmp = rx; rx = ry; ry = tmp;
        }
        nodes[rx].next = ry;
        if (nodes[rx].rank == nodes[ry].rank)
            nodes[ry].rank++;
    }
}

```

Path halving



Find roots



Union by rank

Fine-locking union-find

- No locking in find
 - Do path compression separately
 - Ensure visibility by **volatile** `next`, `rank` in Node

```
class FineUnionFind implements UnionFind {  
    public int find(int x) {  
        while (nodes[x].next != x)  
            x = nodes[x].next;  
        return x;  
    }  
  
    // Assumes lock is held on nodes[root]  
    private void compress(int x, final int root) {  
        while (nodes[x].next != x) {  
            int next = nodes[x].next;  
            nodes[x].next = root;  
            x = next;  
        }  
    }  
}
```

No path halving

Path compression

Fine-locking union-find

```

public void union(final int x, final int y) {
    while (true) {
        int rx = find(x), ry = find(y);
        if (rx == ry)
            return;
        else if (rx > ry) {
            int tmp = rx; rx = ry; ry = tmp;
        }
        synchronized (nodes[rx]) {
            synchronized (nodes[ry]) {
                if (nodes[rx].next != rx || nodes[ry].next != ry)
                    continue;
                if (nodes[rx].rank > nodes[ry].rank) {
                    int tmp = rx; rx = ry; ry = tmp;
                }
                nodes[rx].next = ry;
                if (nodes[rx].rank == nodes[ry].rank)
                    nodes[ry].rank++;
                compress(x, ry);
                compress(y, ry);
            }
        }
    }
}

```

Consistent lock order

Restart if updated

Union by rank and path compression

Wait-free union-find with CAS

UF C

```
class Node {  
    private final AtomicInteger next;  
    private final int rank;  
}
```

Anderson and Woll: Wait-free parallel algorithms for the union-find problem, 1991

```
public int find(int x) {  
    while (nodes.get(x).next.get() != x) {  
        final int t = nodes.get(x).next.get(),  
                u = nodes.get(t).next.get();  
        nodes.get(x).next.compareAndSet(t, u);  
        x = u;  
    }  
    return x;  
}
```

Path halving with CAS

Atomic update of root
`nodes[x]` to point to
fresh Node(y,newRank)

```
boolean updateRoot(int x, int oldRank, int y, int newRank) {  
    final Node oldNode = nodes.get(x);  
    if (oldNode.next.get() != x || oldNode.rank != oldRank)  
        return false;  
    Node newNode = new Node(y, newRank);  
    return nodes.compareAndSet(x, oldNode, newNode);  
}
```

TestUnionFind.java

Wait-free union-find: union

```
public void union(int x, int y) {  
    int xr, yr;  
    do {  
        x = find(x);  
        y = find(y);  
        if (x == y)  
            return;  
        xr = nodes.get(x).rank;  
        yr = nodes.get(y).rank;  
        if (xr > yr || xr == yr && x > y) {  
            { int tmp = x; x = y; y = tmp; }  
            { int tmp = xr; xr = yr; yr = tmp; }  
        }  
    } while (!updateRoot(x, xr, y, xr));  
    if (xr == yr)  
        updateRoot(y, yr, y, yr+1);  
    setRoot(x);  
}
```

TestUnionFind.java

Union-by-rank,
deterministic

Restart if
updated

Some PCPP-related thesis projects

- Design, implement and test concurrent versions of C5 collection classes for .NET
 - <http://www.itu.dk/research/c5/>
- The *Popular Parallel Programming (P3)* project
 - Static dataflow partitioning algorithms
 - Dynamic scheduling algorithms on .NET
 - Vector (SSE, AVX) .NET intrinsics for spreadsheets
 - Supercomputing with Excel and .NET
 - <http://www.itu.dk/people/sestoft/p3/>
- Investigate Java Pathfinder for test and coverage analysis of concurrent software
 - <http://babelfish.arc.nasa.gov/trac/jpf>

This week

- Reading
 - Michael & Scott 1996: *Simple, fast, and practical non-blocking and blocking concurrent queue ...*
 - Goetz chapter 15 and 16
 - Herlihy & Shavit section 3.8
 - Optional: JLS 8 §17.4
- Exercises
 - Test and experiment with the lock-free Michael & Scott queue
- Read before next week – Claus lectures!
 - Armstrong, Virding, Williams: *Concurrent programming in Erlang*, chapters 1, 2, 5, 11.1

PCPP: PRACTICAL CONCURRENT & PARALLEL PROGRAMMERING

MESSAGE PASSING CONCURRENCY



Claus Brabrand
(((brabrand@itu.dk)))

Associate Professor, Ph.D.
(((Software and Systems)))
 **IT University of Copenhagen**

Introduction

Problems:

- **Sharing && Mutability!**



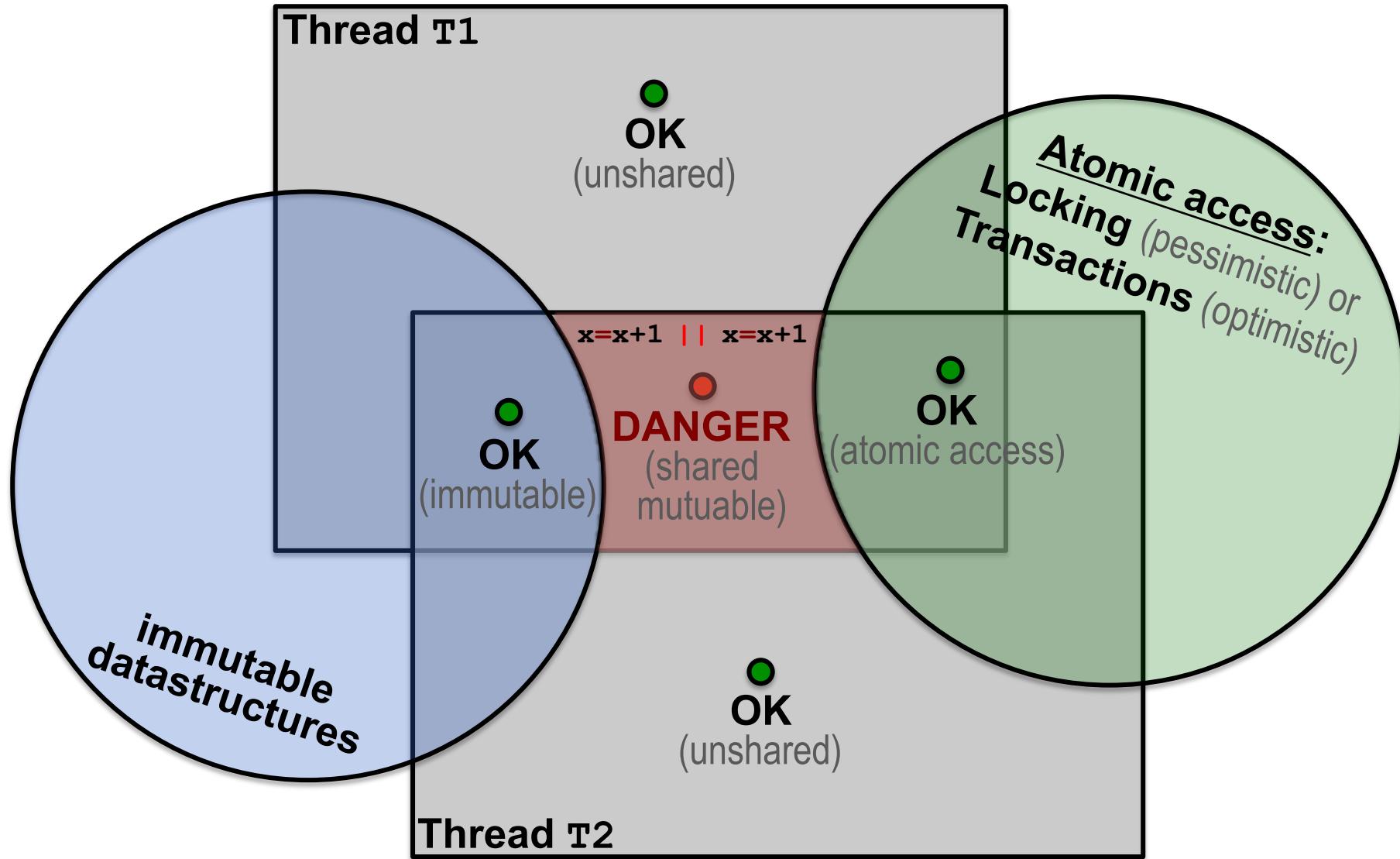
Solutions:

- **1) Atomic access (shared res.):** "synchronized"
 - Locking (pessimistic -concurrency) & Transactions (optimistic-)
 - NB: avoid deadlock!
- **2) Eliminate mutability:** "final"
 - E.g., functional programming
- **3) Eliminate sharing...:** *message passing concurrency*

PROBLEMS: Sharing && Mutability!

SOLUTIONS:

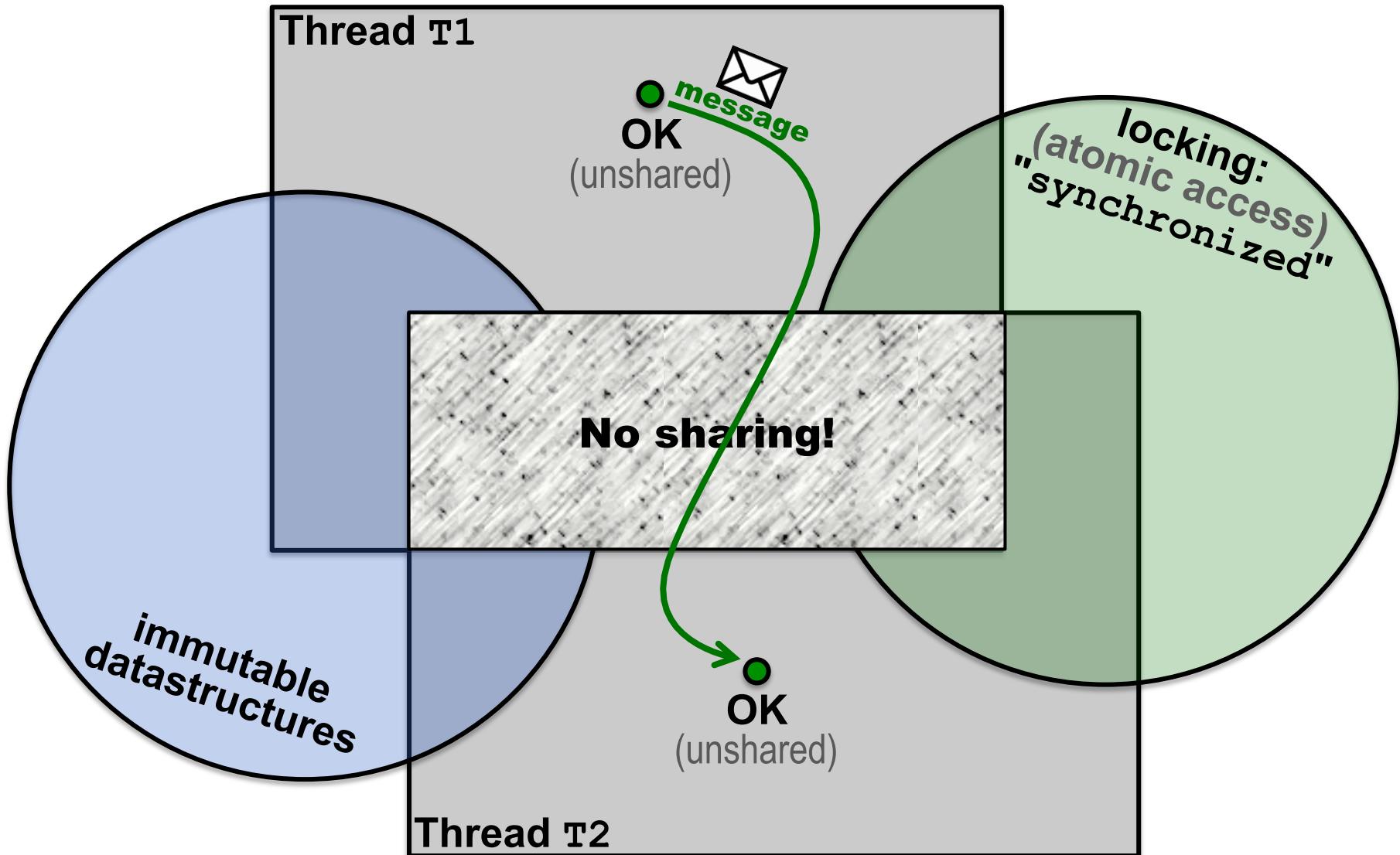
- 1) atomic access!
locking or transactions
NB: avoid deadlock!
- 2) avoid mutability!
- 3) avoid sharing...



PROBLEMS: Sharing && Mutability!

SOLUTIONS:

- 1) atomic access!
locking or transactions
NB: avoid deadlock!
- 2) avoid mutability!
- 3) avoid sharing...



World Wide Web...

In a **distributed setting**,
there's **no shared memory**:

- Communication is achieved
via "message passing"
 - (between concurrently executing servers)



Message Passing Concurrency:

- Same idea (**message passing**)
usable in non-distributed setting:
 - (between processes, inside a server)



Forms of Message Passing

■ Operations:

- **send** and **receive**

■ Symmetry:

- symmetric (send and receive)
- asymmetric (send xor receive)

■ Synchronization:

- synchronous (e.g., phone)
- asynchronous (e.g., email)
- rendez-vous (e.g., barrier)

■ Buffering:

- unbuffered (e.g., blocking)
- buffered (e.g., non-blocking)

■ Multiplicity:

- one-to-one
- one-to-many (or many-to-one)
- many-to-many

■ Addressing:

- direct (naming processes)
- indirect (naming addresses)

■ Reception:

- unconditional (all messages)
- selective (only certain msgs)

■ Anonymity:

- anonymous
- non-anonymous

Synchronous Msg Passing !



Send: `p.send(Value v, Process q);`

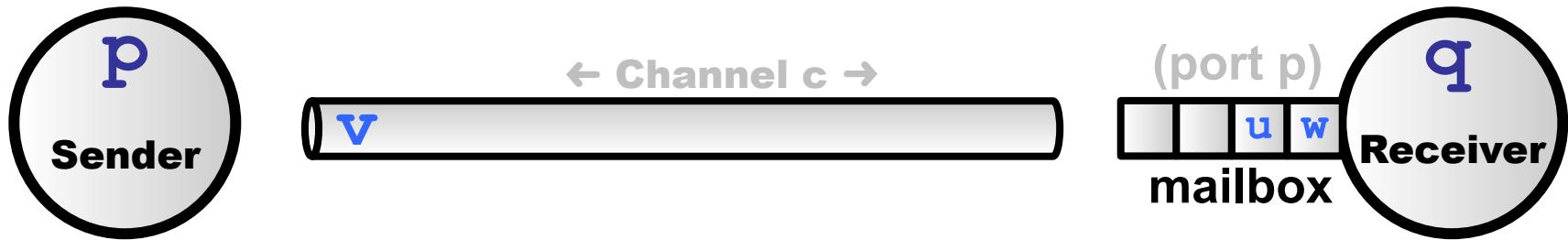
- Sender process **p** **sends** value **v** to receiver process **q**
- Sending process **p** **blocked** until process **q** receives **v**

Receive: `Value receive();`

- Receiver process **q** attempts to **receive** a value **v**
- Receiver process **q** is **blocked** until some value is sent

■ **Synchronous** (i.e., no message buffering)!

Asynchronous Msg Passing !



Send: `void send(Value v, Process q);`

- Sender process **p** **sends** value **v** to process **q**'s mailbox
- Sending process **p** **continues after sending**

Receive: `Value receive();`

- Receiver process **q** attempts to **receive** **v** from its inbox
 - Receiver process **q** is **blocked** until inbox is non-empty
- **Asynchronous** (i.e., messages are buffered)!

Philosophy & Expectations !

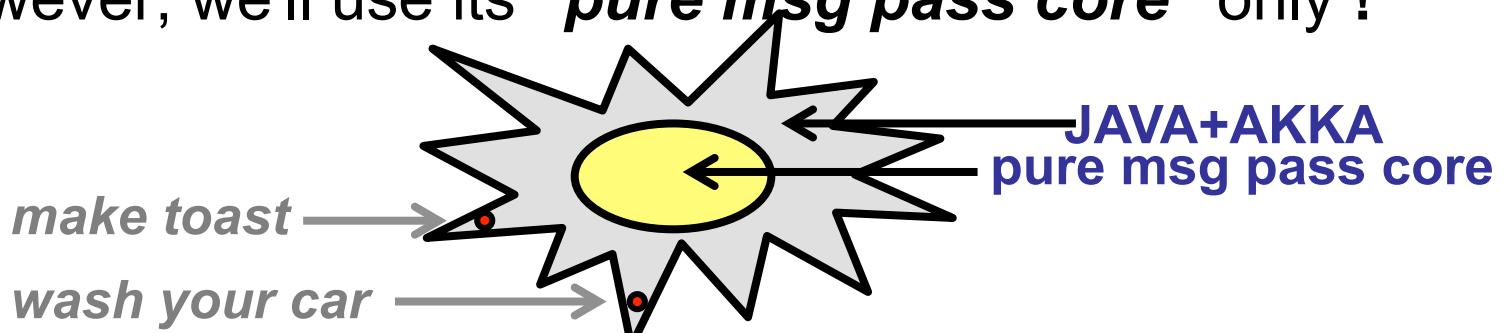
■ ERLANG:

- We'll use as message passing *specification language*
- You have to-be-able-to *read* simple ERLANG programs
 - (i.e., not *write*, nor *modify*)

■ JAVA+AKKA:

- We'll use as msg passing *implementation language*
- You have 2-b-a-2 *read/write/modify* JAVA+AKKA p's
- However, we'll use its "*pure msg pass core*" only !

NB: we're not going to use all of its fantazilions of functions!



An ERLANG Tutorial



"Concurrent Programming in ERLANG"

(Joe Armstrong, Robert Virding, Claes Wikström, Mike Williams)

[Ericsson 1994]

ERLANG

- Named after Danish mathematician
Agner Krarup ERLANG:

...credited for inventing:

- ***traffic engineering***
- ***queueing theory***
- ***telephone networks analysis***



[http://en.wikipedia.org/wiki/Agner_Krarup_Erlang]

- The ERLANG language:

[http://en.wikipedia.org/wiki/Erlang_%28programming_language%29]

- by Ericsson in 1986 (Ericsson Language? :-)

The ERLANG Language (1986)

- Functional language with...:
 - ***message passing concurrency !!!***
 - ***garbage collection***
 - ***eager evaluation***
 - ***single assignment***
 - ***dynamic typing***
- Designed by **Ericsson** to support...:
distributed, fault-tolerant, soft-real-time, non-stop applications
- It supports "***hot swapping***":
 - i.e., code can be changed without stopping a system!

*"Though all concurrency is explicit in ERLANG, processes communicate using **message passing** instead of shared variables, which removes the need for explicit locks."*

-- Wikipedia

Hello World

- Hello World
(in ERLANG)

```
% hello world program:  
-module(helloworld).  
-export([start/0]).  
  
start() ->  
    io:fwrite("Hello world!\n").
```

- Output:

```
Hello world!
```

- Try it out:

[www.tutorialspoint.com/compile_erlang_online.php]

Online ERLANG Compiler

- Online ERLANG Compiler:

- [www.tutorialspoint.com/compile_erlang_online.php]

- Documentation:

- [<http://www.erlang.org/doc/man/io.html>]

- Simple usage:

- One module called: `helloworld`
 - Export one function called: `start/0`
 - Call *your code* from `start()` and `io:write` output

```
-module(helloworld).
-export([start/0]).

yourcode(...) -> ... 

start() -> Val = yourcode(...),    % single assign: unchangable!
          io:write(Val).           % NB: use fwrite for strings!
```

Factorial

- Factorial
(in ERLANG)

```
% factorial program:  
-module(mymath).  
-export([factorial/1]).  
  
factorial(0) -> 1;  
factorial(N) -> N * factorial(N-1).
```

- Usage:

```
> mymath:factorial(6).  
720
```

```
> mymath:factorial(25).  
15511210043330985984000000
```

- Try it out:

[www.tutorialspoint.com/compile_erlang_online.php]

Modularization: Import / Export

- Factorial
(in ERLANG)

```
-module(mymath).  
-export([double/1]).  
  
double(X) -> times(X, 2). % public  
  
times(X, N) -> X * N. % private
```

- Usage:

```
> mymath:double(10).  
20
```

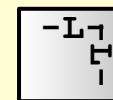
```
> mymath:times(5,2).  
** undef'd fun': mymath:double/2 **
```

- Try it out:

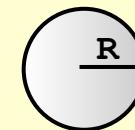
[www.tutorialspoint.com/compile_erlang_online.php]

Pattern Matching

```
-module(mymath).  
-export([area/1]).  
  
area( {square, L} ) ->  
    L * L;  
area( {rectangle, X, Y} ) ->  
    X * Y;  
area( {circle, R} ) ->  
    3.14159 * R * R;  
area( {triangle, A, B, C} ) ->  
    S = (A + B + C)/2,  
    math:sqrt(S*(S-A)*(S-B)*(S-C)).
```



%% patterns in purple!



%% immutable assignment

```
> Thing = {triangle, 6, 7, 8}.  
{triangle,6,7,8}  
> math3:area(Thing).  
20.3332
```

Values (with lists and tuples)

- Numbers: 42, -99, 3.1415, 6.626e-34, ...
- Atoms: abc, 'with space', hello_world, ...
- Tuples: {}, { 1, 2, 3 }, { { x, 1}, { 2, y, 3 } }
- Lists: [], [1, 2, 3], [[x, 1], [2, y, 3]]

```
PCPP =  
  {course, "Practical Concurrent and Parallel Programming",  
   {master, 7.5, { fall, 2014 } }  
   { teachers, [ 'Peter Sestoft', 'Claus Brabrand' ] },  
   { students, [ aaa, bbb, ccc, ... ] }  
 }
```

String (really just
list of characters)

- Recall: *dynamically typed*

Lists: member/2

- **[H | T]** is (standard) **"head-tail constructor"**:
 - **H** is the **head**; i.e., *the first element* (one element)
 - **T** is the **tail**; i.e., *the rest of the list* (zero-or-more)

```
-module(mylists).  
-export([member/2]).  
  
member( X, [] ) -> false;  
member( X, [X|_] ) -> true;  
member( X, [_|T] ) -> member(X, T) .
```

*...for list
construction
de-construction*

```
> mylists:member(3, [1,3,2]).  
true
```

```
> mylists:member(4, [1,3,2]).  
false
```

Lists: append/2

- `[H|T]` is (standard) "**head-tail constructor**":
 - `H` is the **head**; i.e., *the first element* (one element)
 - `T` is the **tail**; i.e., *the rest of the list* (zero-or-more)

```
-module(mylists).                                ...for list
-export([append/2]).                           construction
                                                de-construction
append( [], L ) -> L;
append( [H|L1], L2 ) -> [H|append(L1, L2)].  and re-construction
```

```
> mylists:append([], [a,b])
[a,b]
```

```
> mylists:append([1,2], [3,4])
[1,2,3,4]
```

Actor: Send / Receive / Spawn

■ Send:

- `Pid ! M` // Message M is sent to process Pid
- `Pid ! {some, {complex, structured, [m,s,g]}, 42}`

■ Receive:

- ```
receive
 pattern1 -> ...
 ;
 pattern2 -> ...
end
```
- ```
receive
    {init,N} when N>0 -> ...
    ;
    {init,N} -> ...
end
```

■ Spawn:

- `MyActorId = spawn(mymodule,myactor,[a,r,g,s])`

Order of Receiving Messages

■ Semantics:

```
for (M: message) {  
    for (P: pattern) {  
        M~P (i.e., M matches P)?  
    }  
}
```

This is what happens inside each actor.

■ Example:

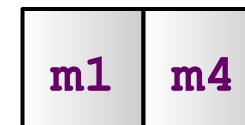
mailbox:



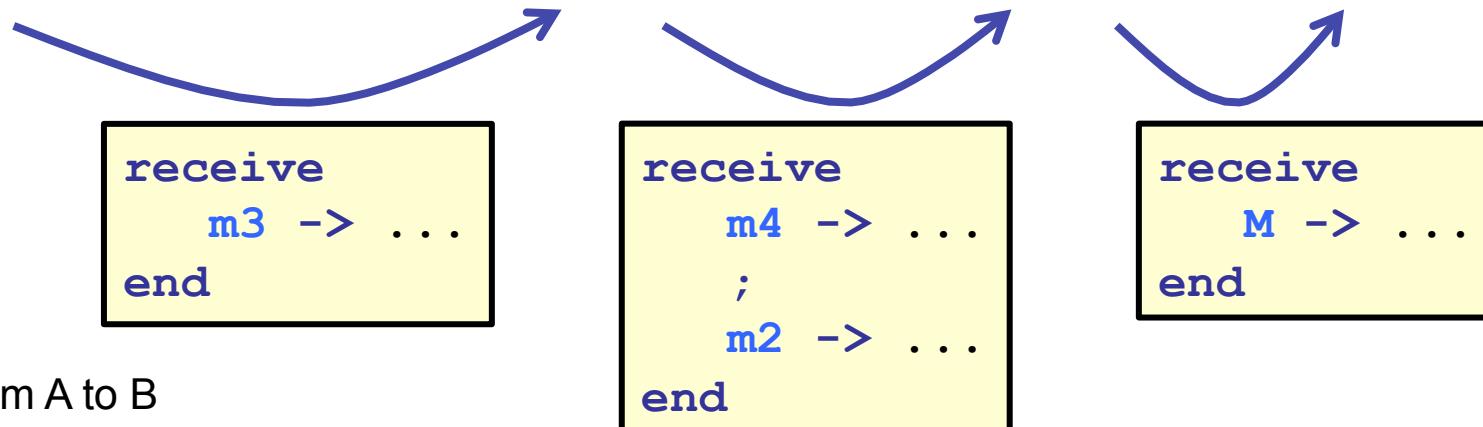
mailbox:



mailbox:



mailbox:



Guarantee:

Msgs sent from A to B
will arrive in order sent

5 Examples (ERLANG & JAVA+AKKA)

1) HelloWorld:

The "Hello World" of message passing; one message is sent to **one actor**.

2) Ecco:

A **person actor** sends a msg to an **ecco actor** that responds with three suffix messages (used for ye olde "hvad drikker møller" kids joke).

3) Broadcast:

Three **person actors** unsubscribe/subscribe to a **broadcast actor** that forwards subsequent incoming msgs to subscribed persons.

4) Primer:

An **actor primer** is created that when initialized with **N=7** creates a **list[]** of that many **slave actors** to factor primes for it. Main bombards the prime actor with msgs ($p \in [2..100]$) that are evenly distributed among the slaves according to **list[p%n]**.

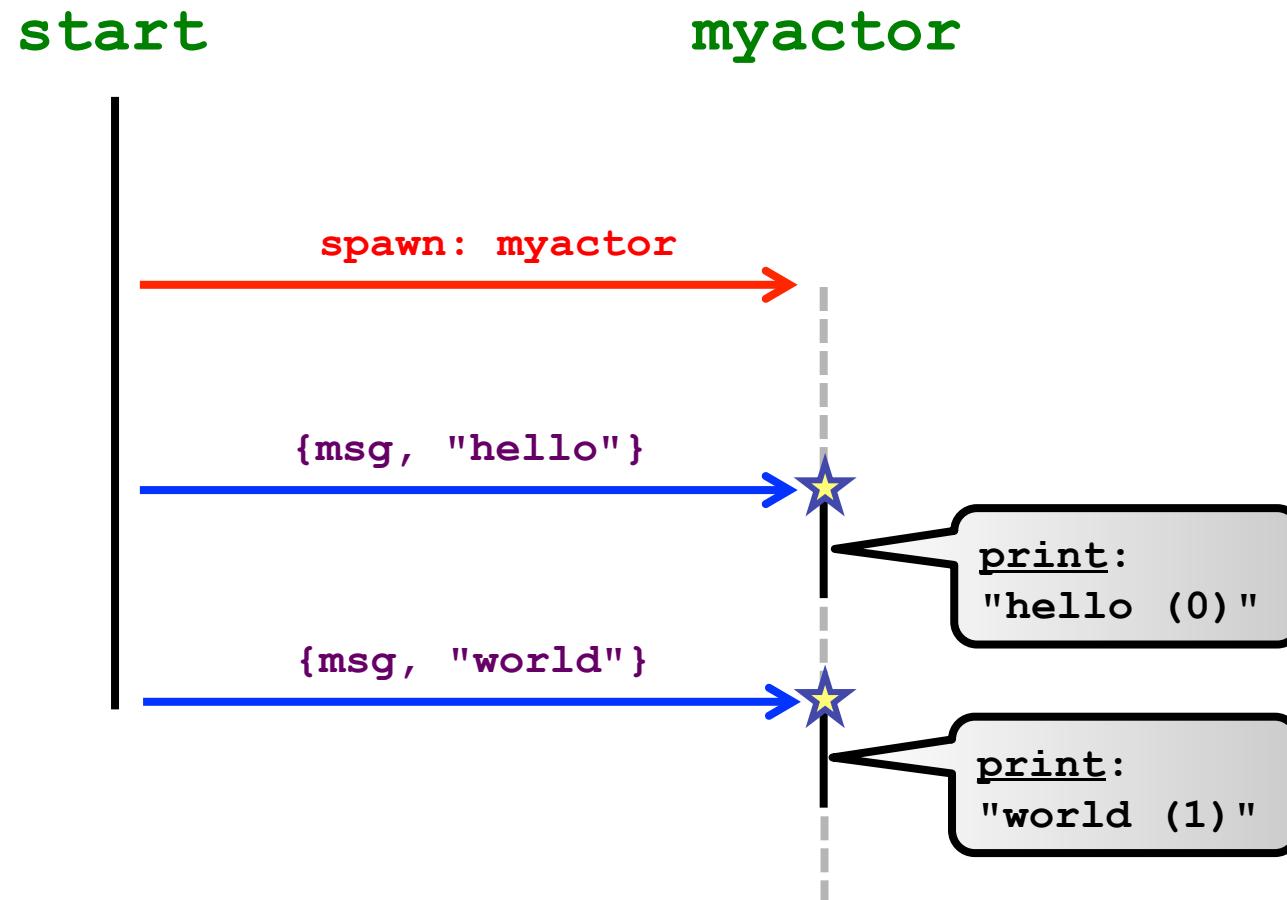
5) ABC:

[lecture-#06]

Two **clerk actors** each bombard a **bank actor** with 100 transfer-random-amount-x-from-an-account-to-other-account msgs. The banks transfer the money by sending **deposit(+x)** to one **account actor** and **deposit(-x)** to the other **account actor**. (The system is called ABC as in Account/Bank/Clerk.)

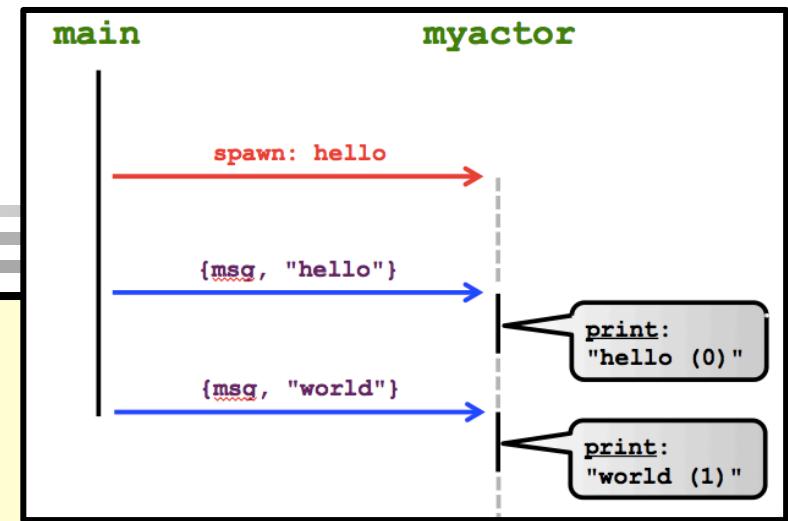
1) HelloWorld

LEGEND:
send, receive, msgs
actors, spawn, rest.



1) HelloWorld.erl

```
-module(helloworld).
-export([start/0,myactor/1]).  
  
myactor(Count) -> %% can have state
    receive
        {msg, Msg} ->
            io:fwrite(Msg ++ " ("),
            io:write(Count),
            io:fwrite("\n"),
            myactor(Count + 1)
    end.  
  
start() ->
    MyActor = spawn(helloworld, myactor, [0]),
    MyActor ! {msg, "hello"},
    MyActor ! {msg, "world"}.
```



hello (1)
world (2)

1) HelloWorld.java

```

import java.io.*;
import akka.actor.*;

// -- MESSAGE
class MyMessage implements Serializable { // must be Serializable:
    public final String s;
    public MyMessage(String s) { this.s = s; }
}

// -- ACTOR ----

class MyActor extends UntypedActor {
    private int count = 0; // can have (local) state

    public void onReceive(Object o) throws Exception { // reacting to message:
        if (o instanceof MyMessage) {
            MyMessage message = (MyMessage) o;
            System.out.println(message.s + " (" + count + ")");
            count++;
        }
    }
}

```

In JAVA+AKKA,
we want to pass
immutable msgs

Otherwise,
we're back to
shared mutable!

```

sequenceDiagram
    participant main
    participant myactor
    main->>myactor: spawn: hello
    activate myactor
    myactor->>myactor: {msg, "hello"}
    activate myactor
    myactor->>myactor: {msg, "world"}
    deactivate myactor
    deactivate myactor
    myactor->>main: print: "hello (0)"
    myactor->>main: print: "world (1)"

```

hello (1)
world (2)

1) HelloWorld.java

```
// -- MAIN -----
public class HelloWorld {
    public static void main(String[] args) {
        final ActorSystem system = ActorSystem.create("HelloWorldSystem");

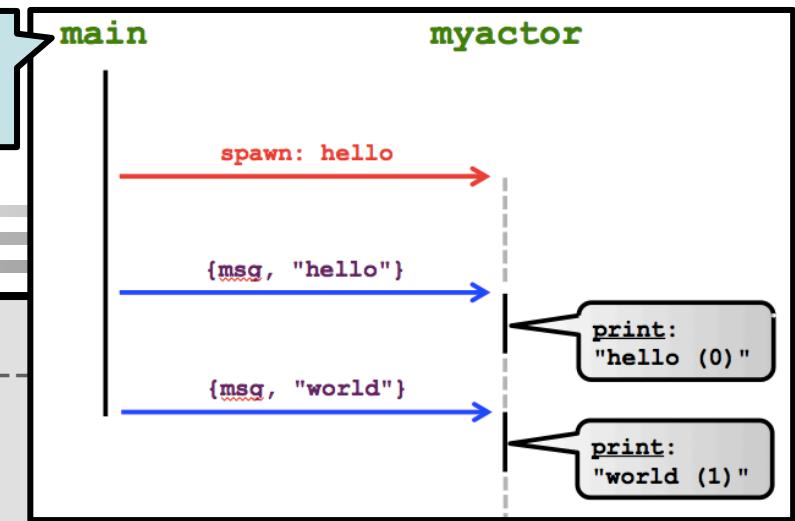
        final ActorRef myactor =
            system.actorOf(Props.create(MyActor.class), "myactor");

        myactor.tell(new MyMessage("hello"), ActorRef.noSender());
        myactor.tell(new MyMessage("world"), ActorRef.noSender());

        try {
            System.out.println("Press return to terminate...");
            System.in.read();
        } catch(IOException e) {
            e.printStackTrace();
        } finally {
            system.shutdown();
        }
    }
}
```

In JAVA+AKKA, the `main()` thread is NOT an actor !

In JAVA+AKKA, the `main()` thread is NOT an actor !



hello (1)
world (2)

1) HelloWorld.java

■ Compile:

```
javac -cp scala.jar:akka-actor.jar HelloWorld.java
```

■ Run:

```
java -cp scala.jar:akka-actor.jar:akka-config.jar:. HelloWorld
```

■ Output:

```
hello (0)  
world (1)
```

2) Ecco



- From Old Danish Kids Joke:

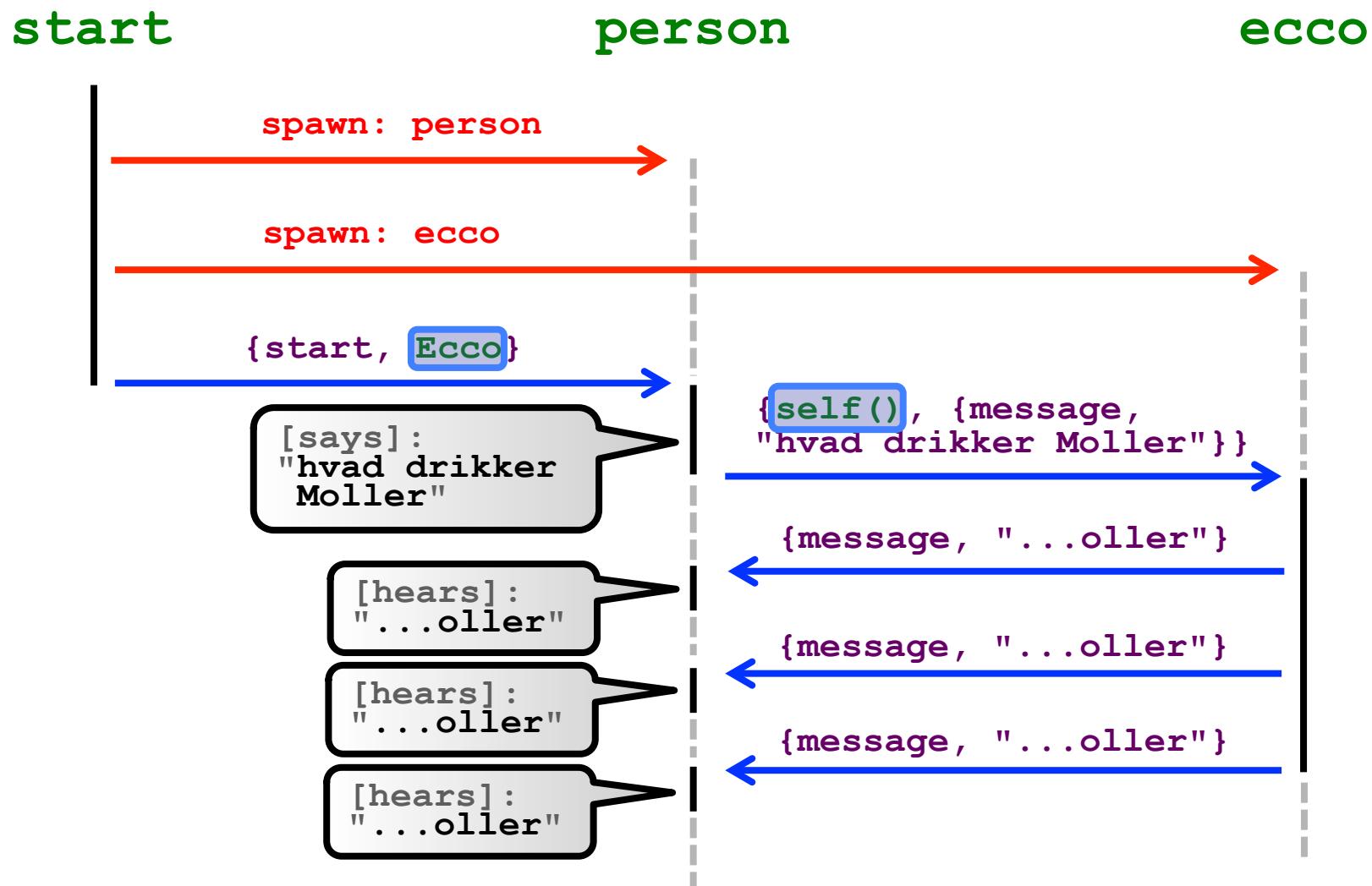
- [<http://www.tordenskjoldssoldater.dk/ekko.html>]

- Huge graffiti in Nordhavnen, Copenhagen:



[<https://www.flickr.com/photos/unacivetta/5745925102/>]

2) Ecco



2) Ecco.erl

```

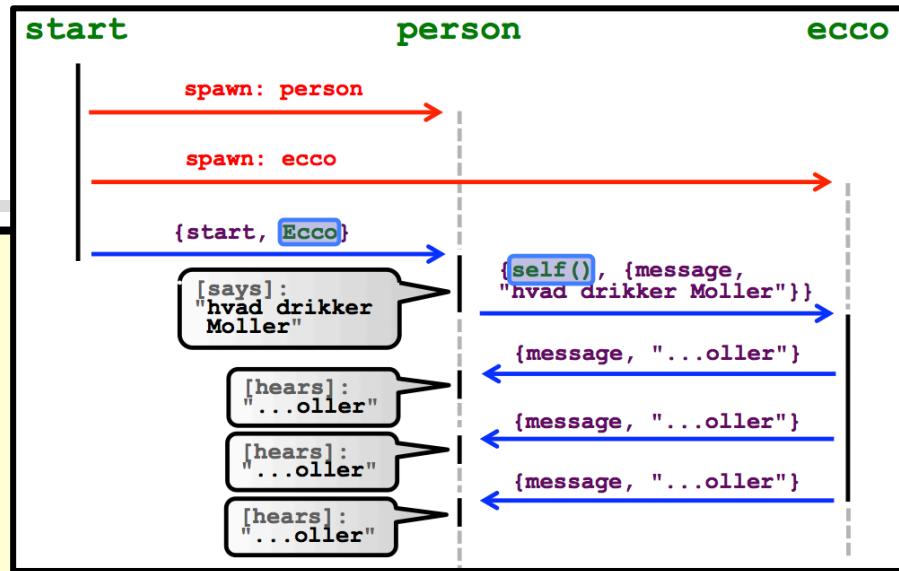
-module(helloworld).
-export([start/0, person/0, ecco/0]).

person() ->
    receive
        {start, Pid} ->
            S = "hvad drikker Moller",
            io:fwrite("[says]: " ++ S ++ "\n"),
            Pid ! {self(), {message, S}} ;
        {message, S} ->
            io:fwrite("[hears]: " ++ S ++ "\n")
    end,
    person().

ecco() ->
    receive
        {Sender, {message, S}} ->
            Sub = substr(S),
            Sender ! {message, Sub},
            Sender ! {message, Sub},
            Sender ! {message, Sub},
            ecco()
    end.

start() ->
    Person = spawn(helloworld, person, []),
    Ecco = spawn(helloworld, ecco, []),
    Person ! {start, Ecco}.

```



```

substr(S) when length(S) < 6 -> "..." ++ S;
substr([_|T]) -> substr(T).

```

```

[says]: hvad drikker Moller
[hears]: ...oller
[hears]: ...oller
[hears]: ...oller

```

2) Ecco.java

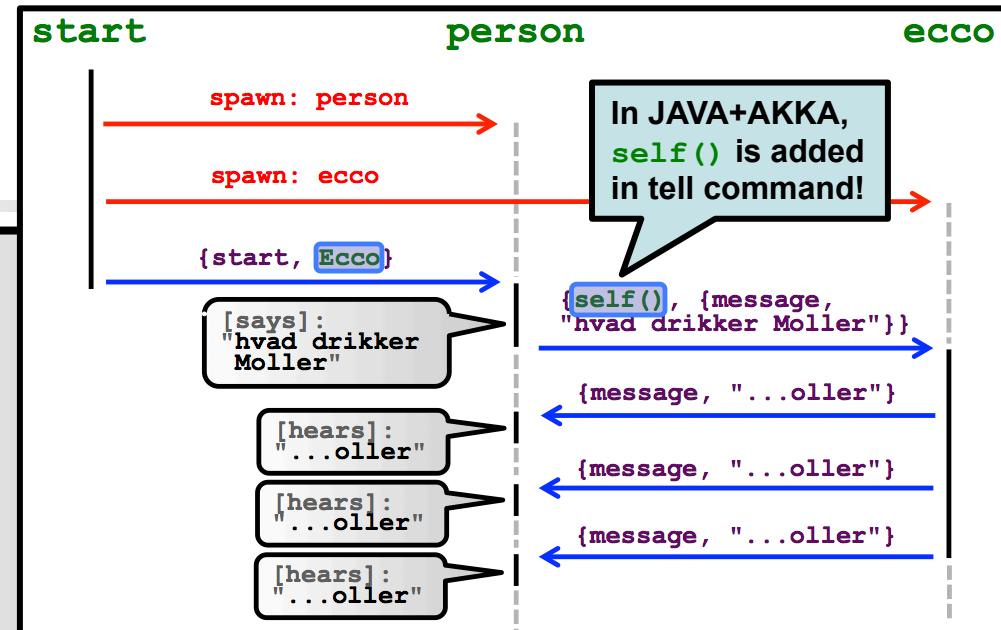
```
import java.io.*;
import akka.actor.*;
```

```
// -- MESSAGES --

class StartMessage implements Serializable {
    public final ActorRef ecco;
    public StartMessage(ActorRef ecco) {
        this.ecco = ecco;
    }
}

class Message implements Serializable {
    public final String s;
    public Message(String s) {
        this.s = s;
    }
}
```

Used for...:
person ← ecco
...and also for:
person → ecco



[says]: hvad drikker Moller
[hears]: ...oller
[hears]: ...oller
[hears]: ...oller

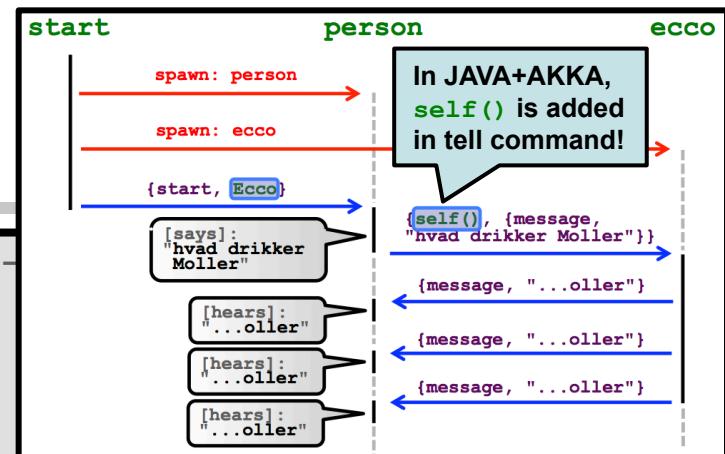
2) Ecco.java

```
// -- ACTORS --

class PersonActor extends UntypedActor {
    public void onReceive(Object o) throws Exception {
        if (o instanceof StartMessage) {
            StartMessage start = (StartMessage) o;
            ActorRef echo = start.echo;
            String s = "hvad drikker moller";
            System.out.println("[says]: " + s);
            echo.tell(new Message(s), getSelf());
        } else if (o instanceof Message) {
            Message m = (Message) o;
            System.out.println("[hears]: " + m.s);
        }
    }
}

class EccoActor extends UntypedActor {
    public void onReceive(Object o) throws Exception {
        if (o instanceof Message) {
            Message m = (Message) o;
            String s = m.s;
            Message reply;
            if (s.length()>5) reply = new Message("..." + s.substring(s.length()-5));
            else reply = new Message("...");
            getSender().tell(reply, getSelf());
            getSender().tell(reply, getSelf());
            getSender().tell(reply, getSelf());
        }
    }
}
```

Here, could also have been:
ActorRef.noSender()



[says]: hvad drikker Moller
 [hears]: ...oller
 [hears]: ...oller
 [hears]: ...oller

2) Ecco.java

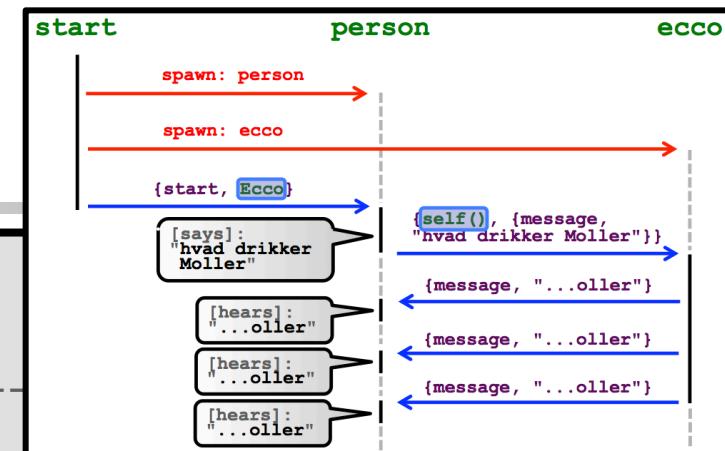
```
// -- MAIN --
public class Ecco {
    public static void main(String[] args) {
        final ActorSystem system = ActorSystem.create("EccoSystem");

        final ActorRef person =
            system.actorOf(Props.create(PersonActor.class), "person");

        final ActorRef ecco =
            system.actorOf(Props.create(EccoActor.class), "ecco");

        person.tell(new StartMessage(ecco), ActorRef.noSender());

        try {
            System.out.println("Press return to terminate...");
            System.in.read();
        } catch(IOException e) {
            e.printStackTrace();
        } finally {
            system.shutdown();
        }
    }
}
```



[says]: hvad drikker Moller
[hears]: ...oller
[hears]: ...oller
[hears]: ...oller

2) Ecco.java

■ Compile:

```
javac -cp scala.jar:akka-actor.jar Ecco.java
```

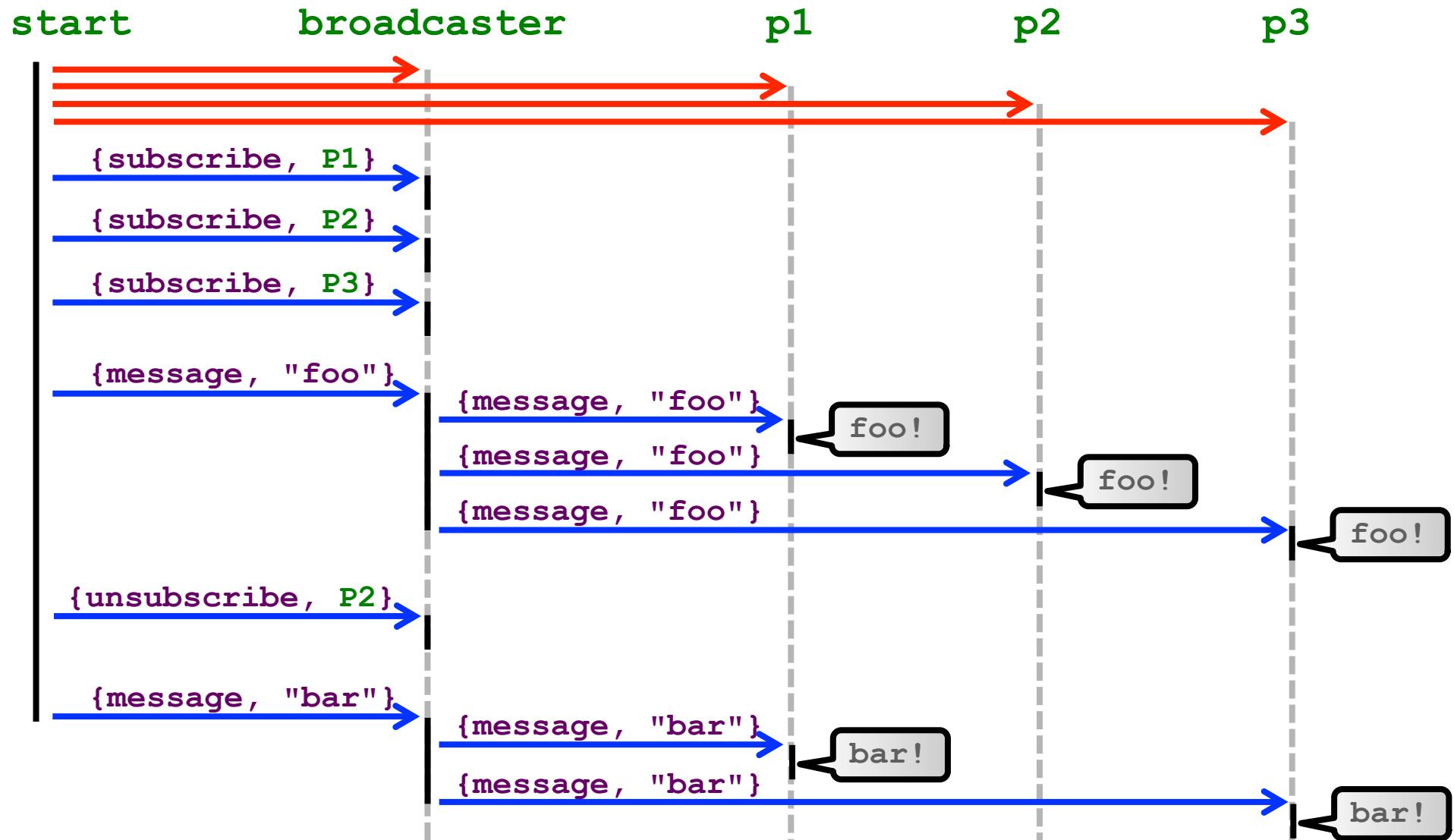
■ Run:

```
java -cp scala.jar:akka-actor.jar:akka-config.jar:. Ecco
```

■ Output:

```
Press return to terminate...
[says]: hvad drikker moller
[hears]: ...oller
[hears]: ...oller
[hears]: ...oller
```

3) Broadcast



3) Broadcast.erl

```

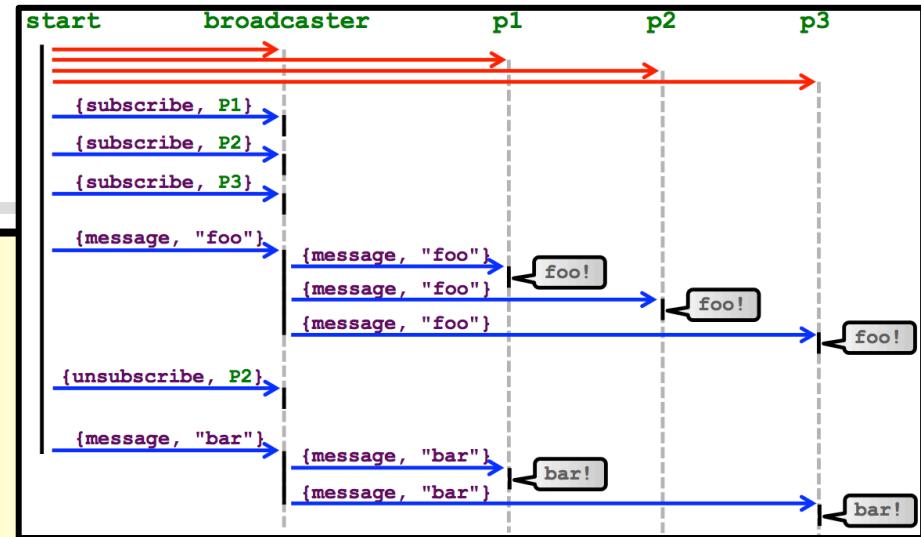
-module(helloworld).
-export([start/0, person/0, broadcaster/1]).

person() ->
    receive
        {message, M} ->
            io:fwrite(M ++ "\n"),
            person()
    end.

broadcast([], _) -> true;
broadcast([Pid|L], M) ->
    Pid ! {message, M},
    broadcast(L, M).

broadcaster(L) ->
    receive
        {subscribe, Pid} ->
            broadcaster([Pid|L]);
        {unsubscribe, Pid} ->
            broadcaster(lists:delete(Pid, L));
        {message, M} ->
            broadcast(L, M),
            broadcaster(L)
    end.

```



```

start() ->
    Broadcaster = spawn(helloworld, broadcaster, []),
    P1 = spawn(helloworld, person, []),
    P2 = spawn(helloworld, person, []),
    P3 = spawn(helloworld, person, []),
    Broadcaster ! {subscribe, P1},
    Broadcaster ! {subscribe, P2},
    Broadcaster ! {subscribe, P3},
    Broadcaster ! {message, "Purses half price!"},
    Broadcaster ! {unsubscribe, P2},
    Broadcaster ! {message, "Shoes half price!!"}.

```

purses half price!
 purses half price!
 purses half price!
 shoes half price!!
 shoes half price!!

3) Broadcast.java

```

import java.util.*;
import java.io.*;
import akka.actor.*;

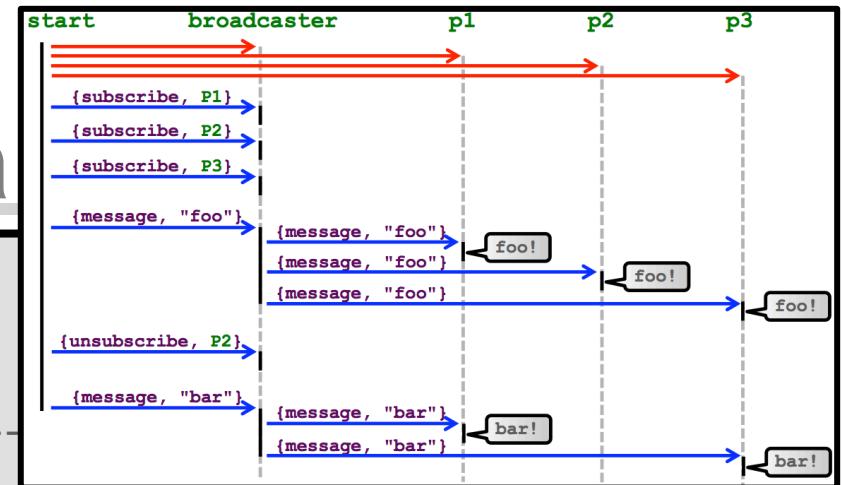
// -- MESSAGES ---

class SubscribeMessage implements Serializable {
    public final ActorRef subscriber;
    public SubscribeMessage(ActorRef subscriber) {
        this.subscriber = subscriber;
    }
}

class UnsubscribeMessage implements Serializable {
    public final ActorRef unsubscribe;
    public UnsubscribeMessage(ActorRef unsubscribe) {
        this.unsubscribe = unsubscribe;
    }
}

class Message implements Serializable {
    public final String s;
    public Message(String s) {
        this.s = s;
    }
}

```



purses half price!
 purses half price!
 purses half price!
 shoes half price!!
 shoes half price!!

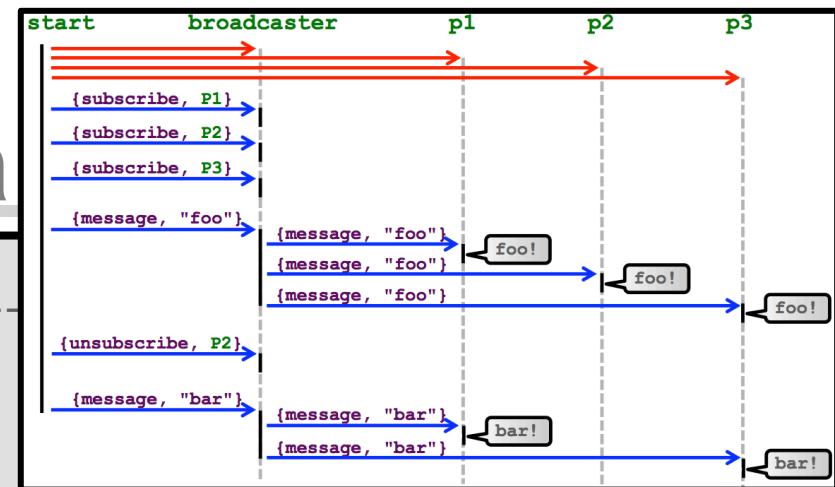
3) Broadcast.java

```
// -- ACTORS ---

class BroadcastActor extends UntypedActor {
    private List<ActorRef> list =
        new ArrayList<ActorRef>();

    public void onReceive(Object o) throws Exception {
        if (o instanceof SubscribeMessage) {
            list.add(((SubscribeMessage) o).subscriber);
        } else if (o instanceof UnsubscribeMessage) {
            list.remove(((UnsubscribeMessage) o).unsubscriber);
        } else if (o instanceof Message) {
            for (ActorRef person : list) {
                person.tell(o, getSelf());
            }
        }
    }
}

class PersonActor extends UntypedActor {
    public void onReceive(Object o) throws Exception {
        if (o instanceof Message) {
            System.out.println((Message) o).s;
        }
    }
}
```

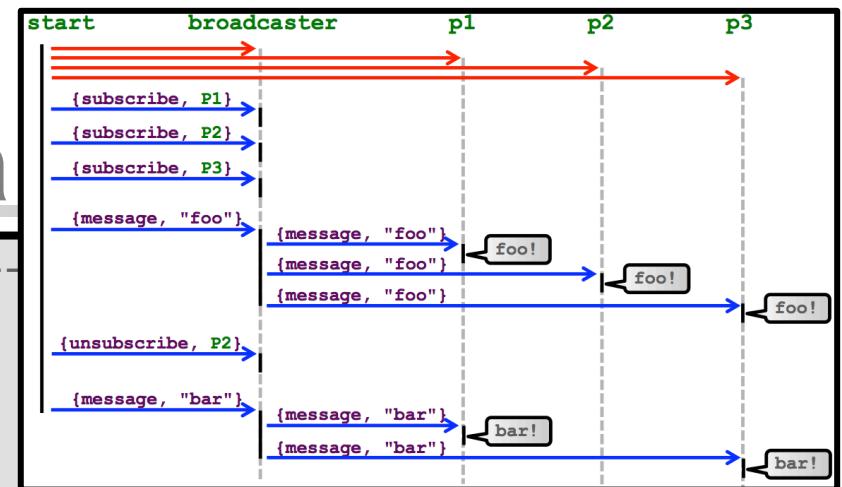


purses half price!
 purses half price!
 purses half price!
 shoes half price!!
 shoes half price!!

3) Broadcast.java

```
// -- MAIN -->

public class Broadcast {
    public static void main(String[] args) {
        final ActorSystem system =
            ActorSystem.create("EccoSystem");
        final ActorRef broadcaster =
            system.actorOf(Props.create(BroadcastActor.class), "broadcaster");
        final ActorRef p1 = system.actorOf(Props.create(PersonActor.class), "p1");
        final ActorRef p2 = system.actorOf(Props.create(PersonActor.class), "p2");
        final ActorRef p3 = system.actorOf(Props.create(PersonActor.class), "p3");
        broadcaster.tell(new SubscribeMessage(p1), ActorRef.noSender());
        broadcaster.tell(new SubscribeMessage(p2), ActorRef.noSender());
        broadcaster.tell(new SubscribeMessage(p3), ActorRef.noSender());
        broadcaster.tell(new Message("purses half price!"), ActorRef.noSender());
        broadcaster.tell(new UnsubscribeMessage(p2), ActorRef.noSender());
        broadcaster.tell(new Message("shoes half price!!"), ActorRef.noSender());
        try {
            System.out.println("Press return to terminate...");
            System.in.read();
        } catch(IOException e) {
            e.printStackTrace();
        } finally {
            system.shutdown();
        }
    }
}
```



purses half price!
 purses half price!
 purses half price!
 shoes half price!!
 shoes half price!!

3) Broadcast.java

■ Compile:

```
javac -cp scala.jar:akka-actor.jar Broadcast.java
```

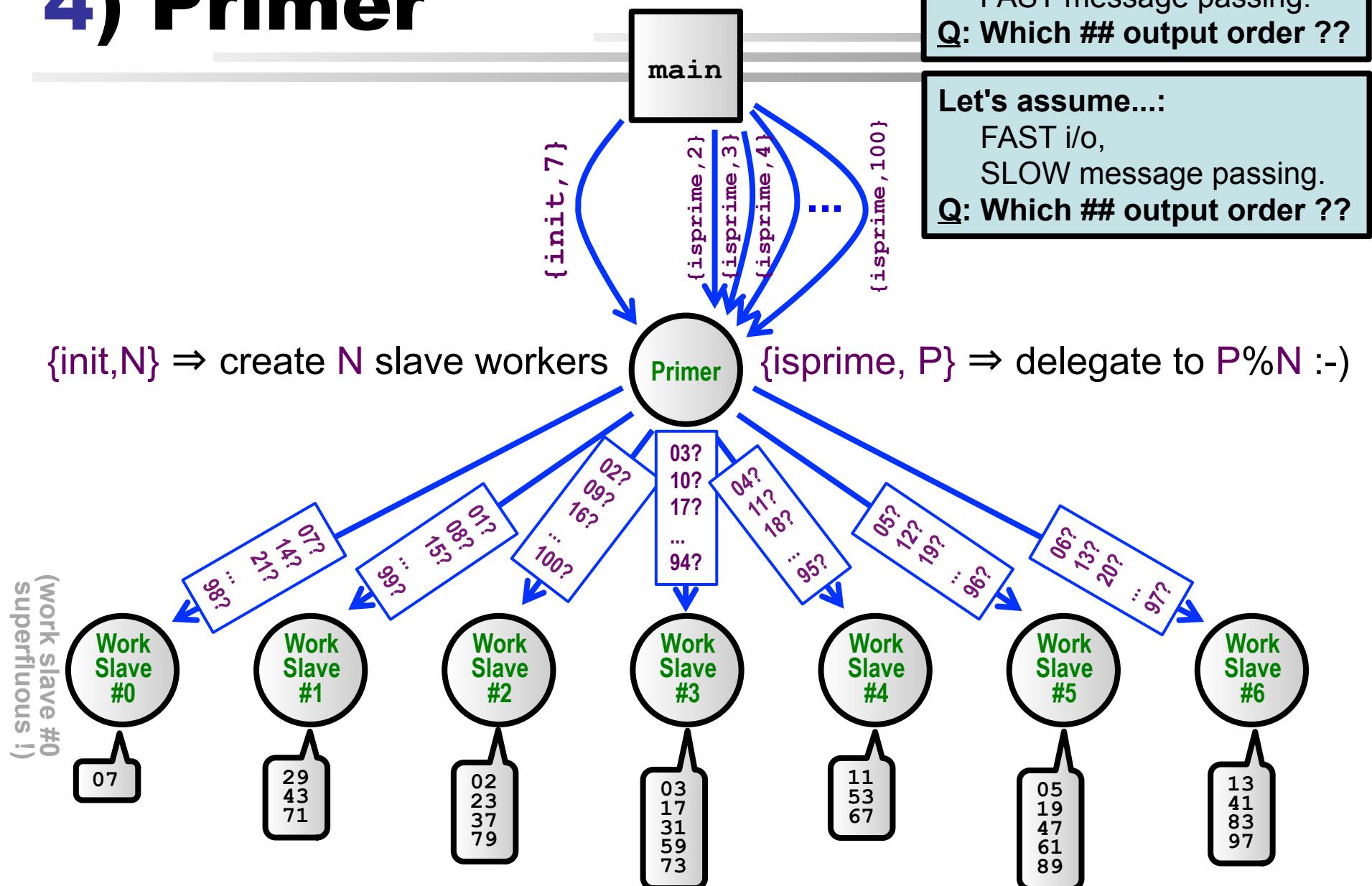
■ Run:

```
java -cp scala.jar:akka-actor.jar:akka-config.jar:. Broadcast
```

■ Output:

```
purses half price!
purses half price!
purses half price!
shoes half price!!
shoes half price!!
```

4) Primer



4) Primer.erl

```
-module(helloworld).  
-export([start/0,slave/1,primer/1]).  
  
is_prime_loop(N,K) ->  
    K2 = K * K, R = N rem K,  
    case (K2 =< N) and (R /= 0) of  
        true -> is_prime_loop(N, K+1);  
        false -> K  
    end.  
  
is_prime(N) ->  
    K = is_prime_loop(N,2),  
    (N >= 2) and (K*K > N).  
  
n2s(N) ->  
    lists:flatten(io_lib:format("~p", [N])).  
  
slave(Id) ->  
    receive  
        {isprime, N} ->  
            case is_prime(N) of  
                true -> io:fwrite("(" ++  
n2s(Id) ++ ") " ++ n2s(N) ++ "\n");  
                false -> []  
            end,  
            slave(Id)  
    end.
```

Slave

```
create_slaves(Max,Max) -> [];  
create_slaves(Id,Max) ->  
    Slave = spawn(helloworld,slave,[Id]),  
    [Slave|create_slaves(Id+1,Max)].  
  
primer(Slaves) ->  
    receive  
        {init, N} when N=<0 ->  
            throw({nonpositive,N}) ;  
        {init, N} ->  
            primer(create_slaves(0,N)) ;  
        {isprime, _} when Slaves == [] ->  
            throw({uninitialized}) ;  
        {isprime, N} when N=<0 ->  
            throw({nonpositive,N}) ;  
        {isprime, N} ->  
            SlaveId = N rem length(Slaves),  
            lists:nth(SlaveId+1, Slaves)  
            ! {isprime,N},  
            primer(Slaves)  
    end.  
  
spam(_, N, Max) when N>=Max -> true;  
spam(Primer, N, Max) ->  
    Primer ! {isprime, N},  
    spam(Primer, N+1, Max).  
  
start() ->  
    Primer =  
        spawn(helloworld, primer, []),  
    Primer ! {init,7},  
    spam(Primer, 2, 100).
```

Primer

4) Primer.java

```
import java.util.*;
import java.io.*;
import akka.actor.*;

// -- MESSAGES ----

class InitializeMessage implements Serializable {
    public final int number_of_slaves;
    public InitializeMessage(int number_of_slaves) {
        this.number_of_slaves = number_of_slaves;
    }
}

class IsPrimeMessage implements Serializable {
    public final int number;
    public IsPrimeMessage(int number) {
        this.number = number;
    }
}
```



4) Primer.java

```
// -- SLAVE ACTOR -----  
  
class SlaveActor extends UntypedActor {  
    private boolean isPrime(int n) {  
        int k = 2;  
        while (k * k <= n && n % k != 0) k++;  
        return n >= 2 && k * k > n;  
    }  
  
    public void onReceive(Object o) throws Exception {  
        if (o instanceof IsPrimeMessage) {  
            int p = ((IsPrimeMessage) o).number;  
            if (isPrime(p)) System.out.println("(" + p%7 + ") " + p); %% HACK: 7 !  
        }  
    }  
}
```

4) Primer.java

```
// -- PRIME ACTOR -----
class PrimeActor extends UntypedActor {
    List<ActorRef> slaves;

    private List<ActorRef> createSlaves(int n) {
        List<ActorRef> slaves = new ArrayList<ActorRef>();
        for (int i=0; i<n; i++) {
            ActorRef slave =
                getContext().actorOf(Props.create(SlaveActor.class), "p" + i);
            slaves.add(slave);
        }
        return slaves;
    }

    public void onReceive(Object o) throws Exception {
        if (o instanceof InitializeMessage) {
            InitializeMessage init = (InitializeMessage) o;
            int n = init.number_of_slaves;
            if (n<=0) throw new RuntimeException("!!! non-positive number!");
            slaves = createSlaves(n);
            System.out.println("initialized (" + n + " slaves ready to work)!");
        } else if (o instanceof IsPrimeMessage) {
            if (slaves==null) throw new RuntimeException("!!! uninitialized!");
            int n = ((IsPrimeMessage) o).number;
            if (n<=0) throw new RuntimeException("!!! non-positive number!");
            int slave_id = n % slaves.size();
            slaves.get(slave_id).tell(o, getSelf());
        }
    }
}
```

4) Primer.java

```
// -- MAIN -----  
  
public class Primer {  
    private static void spam(ActorRef primer, int min, int max) {  
        for (int i=min; i<max; i++) {  
            primer.tell(new IsPrimeMessage(i), ActorRef.noSender());  
        }  
    }  
  
    public static void main(String[] args) {  
        final ActorSystem system = ActorSystem.create("PrimerSystem");  
        final ActorRef primer =  
            system.actorOf(Props.create(PrimeActor.class), "primer");  
        primer.tell(new InitializeMessage(7), ActorRef.noSender());  
        try {  
            System.out.println("Press return to initiate...");  
            System.in.read();  
            spam(primer, 2, 100);  
            System.out.println("Press return to terminate...");  
            System.in.read();  
        } catch(IOException e) {  
            e.printStackTrace();  
        } finally {  
            system.shutdown();  
        }  
    }  
}
```

4) Primer.java

■ Compile:

```
javac -cp scala.jar:akka-actor.jar Primer.java
```

■ Run:

```
java -cp scala.jar:akka-actor.jar:akka-config.jar:. Primer
```

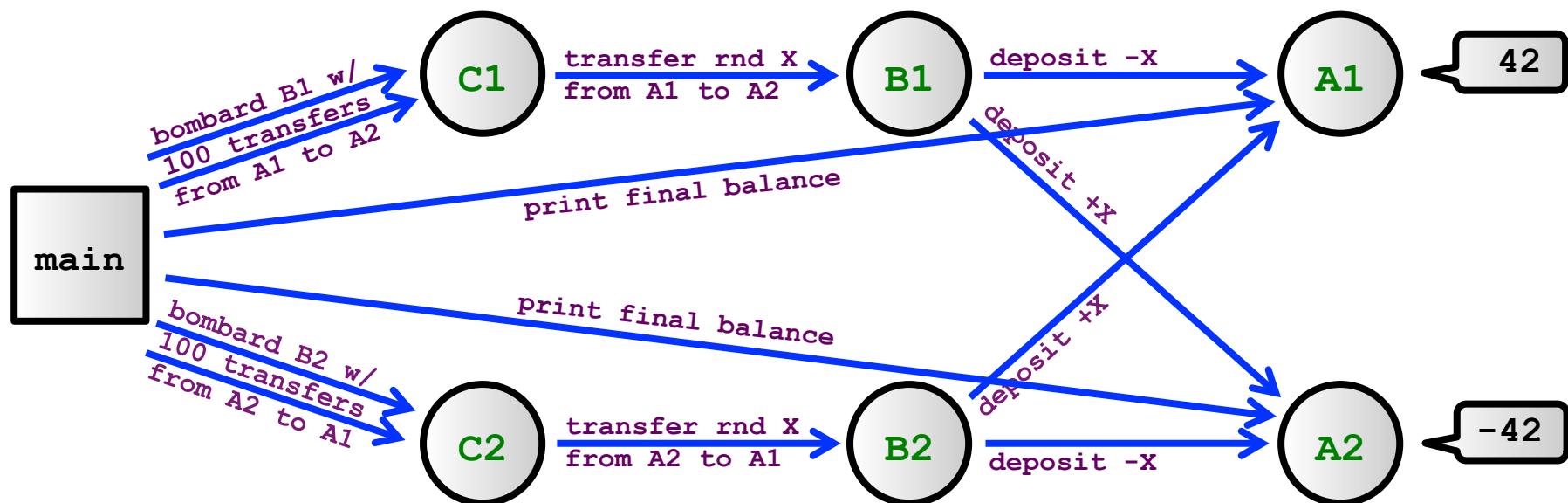
■ Output:

```
press return to initiate...
initialized (7 slaves ready to work)!

(2) 2
(3) 3
Press return to terminate...
(0) 7
(5) 5
(4) 11
(6) 13
(3) 17
(5) 19
(2) 23
(1) 29
(3) 31
```

```
(2) 37
(6) 41
(1) 43
(5) 47
(4) 53
(3) 59
(5) 61
(4) 67
(1) 71
(3) 73
(2) 79
(6) 83
(5) 89
(6) 97
```

5) ABC (Clerk / Bank / Account)



5) ABC.erl

```
-module(helloworld).
-export([start/0,
        account/1,bank/0,clerk/0]).

%% -- BASIC PROCESSING -----
n2s(N) -> lists:flatten( %% int2string
  io_lib:format("~p", [N])). %% HACK!

random(N) -> random:uniform(N) div 10.

%% -- ACTORS -----

account(Balance) ->
  receive
    {deposit,Amount} ->
      account(Balance+Amount) ;
    {printbalance} ->
      io:fwrite(n2s(Balance) ++ "\n")
  end.

bank() ->
  receive
    {transfer,Amount,From,To} ->
      From ! {deposit,-Amount},
      To ! {deposit,+Amount},
      bank()
  end.
```

```
ntransfers(0,_,_,_) -> true;
ntransfers(N,Bank,From,To) ->
  R = random(100),
  Bank ! {transfer,R,From,To},
  ntransfers(N-1,Bank,From,To).

clerk() ->
  receive
    {start,Bank,From,To} ->
      random:seed(now()),
      ntransfers(100,Bank,From,To),
      clerk()
  end.

start() ->
  A1 = spawn(helloworld,account,[0]),
  A2 = spawn(helloworld,account,[0]),
  B1 = spawn(helloworld,bank,[]),
  B2 = spawn(helloworld,bank,[]),
  C1 = spawn(helloworld,clerk,[]),
  C2 = spawn(helloworld,clerk,[]),
  C1 ! {start,B1,A1,A2},
  C2 ! {start,B2,A2,A1},
  timer:sleep(1000),
  A1 ! {printbalance},
  A2 ! {printbalance}.
```

5) ABC.java

(Skeleton)

```

import java.util.Random; import java.io.*; import akka.actor.*;

// -- MESSAGES -----
class StartTransferMessage implements Serializable { /* TODO */ }
class TransferMessage implements Serializable { /* TODO */ }
class DepositMessage implements Serializable { /* TODO */ }
class PrintBalanceMessage implements Serializable { /* TODO */ }

// -- ACTORS -----
class AccountActor extends UntypedActor { /* TODO */ }
class BankActor extends UntypedActor { /* TODO */ }
class ClerkActor extends UntypedActor { /* TODO */ }

// -- MAIN -----
public class ABC { // Demo showing how things work:
    public static void main(String[] args) {
        final ActorSystem system = ActorSystem.create("ABCSystem");
        /* TODO (CREATE ACTORS AND SEND START MESSAGES) */

        try {
            System.out.println("Press return to inspect...");
            System.in.read();

            /* TODO (INSPECT FINAL BALANCES) */

            System.out.println("Press return to terminate...");
            System.in.read();
        } catch(IOException e) {
            e.printStackTrace();
        } finally {
            system.shutdown();
        }
    }
}

```

MANDATORY HAND-IN!

a) Color ABC.erl

(according to color convention):

**send, receive, msgs
actors, spawn, rest.**

(try 2 B as consistent as possible)

b) Implement ABC.java

(as close to ABC.erl as possible)

c) Answer question:

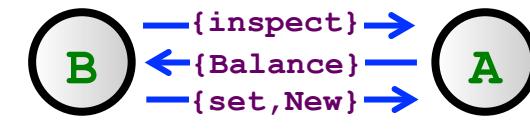
What happens if we replace
{deposit, ±Amount} w/ the msgs?:

*** OUTPUT ***

```

Press return to inspect...
Press return to terminate...
Balance = 42
Balance = -42

```



Thx!



Questions?

PCPP: PRACTICAL CONCURRENT & PARALLEL PROGRAMMERING

MESSAGE PASSING CONCURRENCY II / II



Claus Brabrand
(((brabrand@itu.dk)))

Associate Professor, Ph.D.
(((Software and Systems)))
 **IT University of Copenhagen**

AGENDA



■ 3) Broadcast:

- From ERLANG to JAVA+AKKA
- Communication protocols (one-to-one \Rightarrow one-to-many)

■ AKKA: A proper introduction

- Motivations and benefits of Actors & Message Passing
- Recommendations

■ 4) Primer:

- Hierarchic organization: managers supervise workers
- Performance: MacBook Air -vs- MTLab Server

■ ★ Scatter-Gather:

- Prototypical AKKA Service (dynamic load balancing)
- Extensions...

5) ABC.erl

```
-module(helloworld).
-export([start/0,
        account/1, bank/0, clerk/0]).

%% -- BASIC PROCESSING -----
n2s(N) -> lists:flatten( %% int2string
    io_lib:format("~p", [N])). %% HACK!

random(N) -> random:uniform(N) div 10.

%% -- ACTORS -----

account(Balance) ->
    receive
        {deposit,Amount} ->
            account(Balance+Amount) ;
        {printbalance} ->
            io:fwrite(n2s(Balance) ++ "\n")
    end.

bank() ->
    receive
        {transfer,Amount,From,To} ->
            From ! {deposit,-Amount},
            To ! {deposit,+Amount},
            bank()
    end.
```

MANDATORY HAND-IN!

a) Color ABC.erl

(according to color convention):

**send, receive, msgs
actors, spawn, rest.**

(try 2 B as consistent as possible)

MOTIVATION:

1) Structure of ERLANG:

- Syntax (structure); then
- Semantics (meaning)

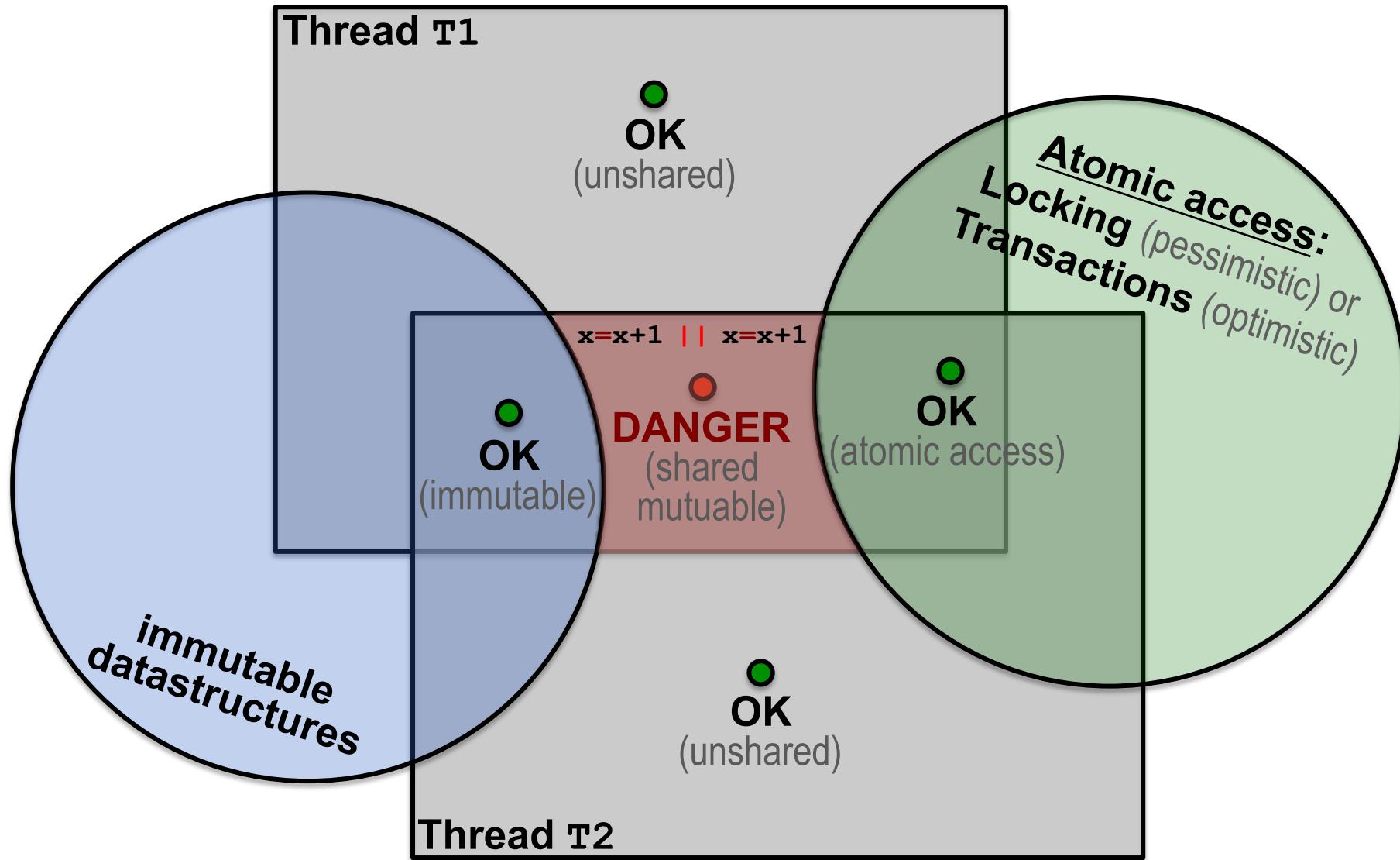
2) Discern linguistic aspects:

**send, receive, msgs
actors, spawn, rest.**

PROBLEMS: Sharing && Mutability!

SOLUTIONS:

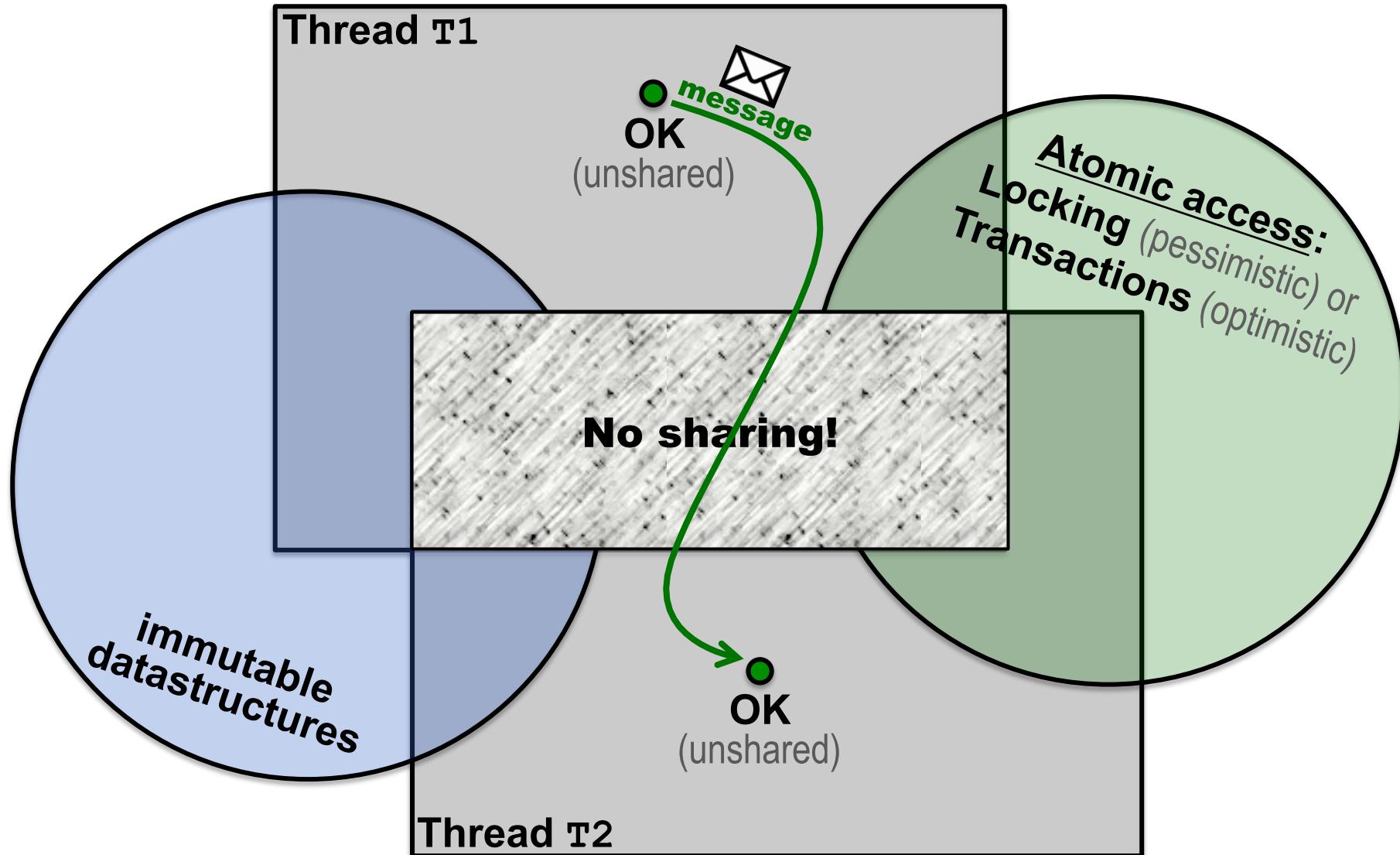
- 1) atomic access!
locking or transactions
(NB: avoid deadlock!)
- 2) avoid mutability!
- 3) avoid sharing...



PROBLEMS: Sharing && Mutability!

SOLUTIONS:

- 1) atomic access!
locking or transactions
(NB: avoid deadlock!)
- 2) avoid mutability!
- 3) avoid sharing...



Philosophy & Expectations!

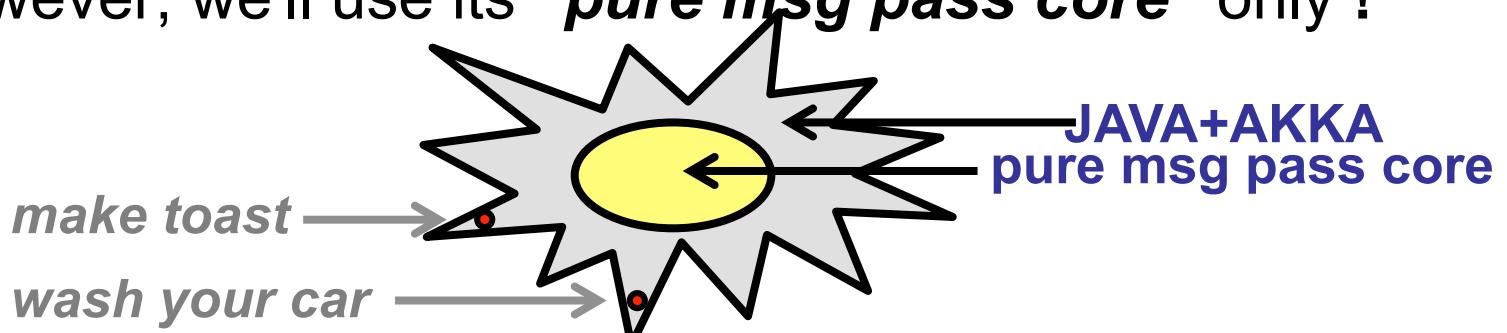
■ ERLANG:

- We'll use as message passing *specification language*
- You have to-be-able-to *read* simple ERLANG programs
 - (i.e., not *write*, nor *modify*)

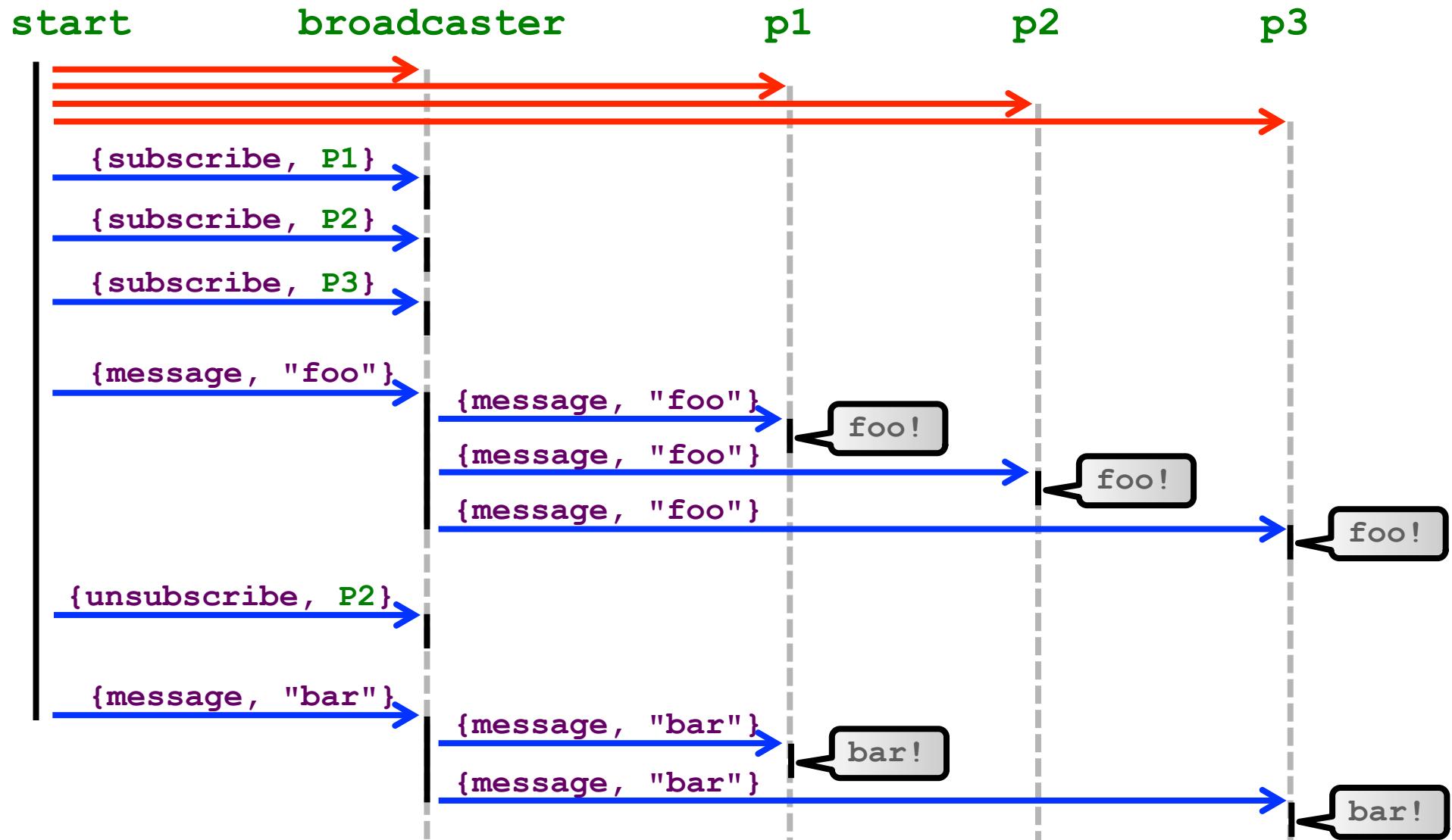
■ JAVA+AKKA:

- We'll use as msg passing *implementation language*
- You have 2-b-a-2 *read/write/modify* JAVA+AKKA p's
- However, we'll use its "*pure msg pass core*" only !

NB: we're not going to use all of its fantazilions of functions!



3) Broadcast



3) Broadcast.erl

```

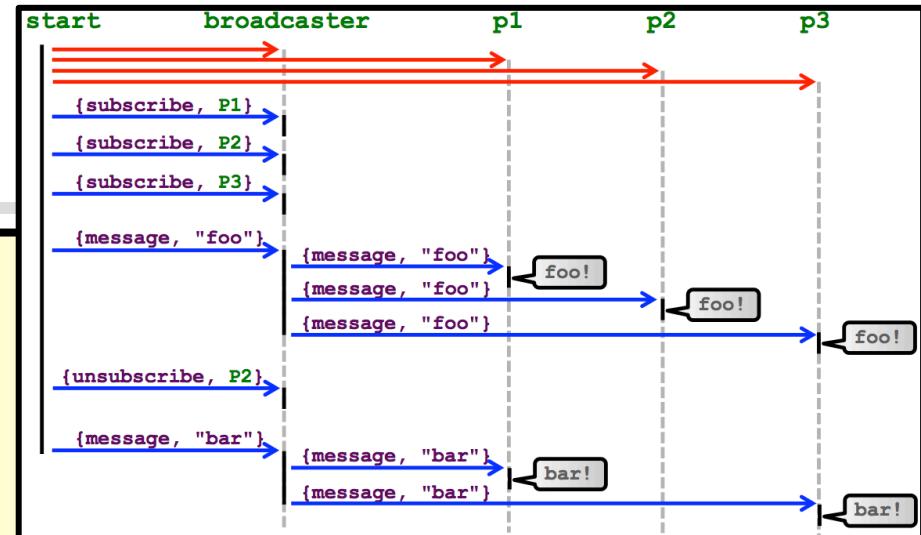
-module(helloworld).
-export([start/0, person/0, broadcaster/1]).

person() ->
    receive
        {message, M} ->
            io:fwrite(M ++ "\n"),
            person()
    end.

broadcast([], _) -> true;
broadcast([Pid|L], M) ->
    Pid ! {message, M},
    broadcast(L, M).

broadcaster(L) ->
    receive
        {subscribe, Pid} ->
            broadcaster([Pid|L]);
        {unsubscribe, Pid} ->
            broadcaster(lists:delete(Pid, L));
        {message, M} ->
            broadcast(L, M),
            broadcaster(L)
    end.

```



```

start() ->
    Broadcaster = spawn(helloworld, broadcaster, []),
    P1 = spawn(helloworld, person, []),
    P2 = spawn(helloworld, person, []),
    P3 = spawn(helloworld, person, []),
    Broadcaster ! {subscribe, P1},
    Broadcaster ! {subscribe, P2},
    Broadcaster ! {subscribe, P3},
    Broadcaster ! {message, "Purses half price!"},
    Broadcaster ! {unsubscribe, P2},
    Broadcaster ! {message, "Shoes half price!!"}.

```

purses half price!
 purses half price!
 purses half price!
 shoes half price!!
 shoes half price!!

3) Broadcast.java

```

import java.util.*;
import java.io.*;
import akka.actor.*;

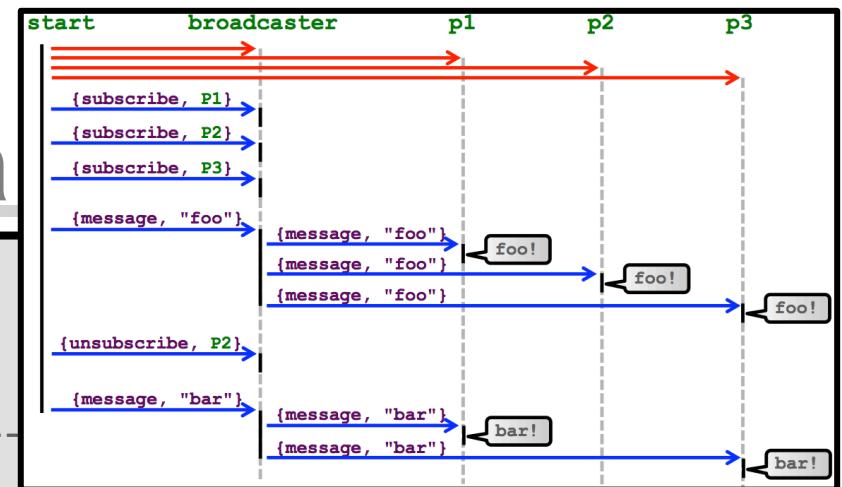
// -- MESSAGES ---

class SubscribeMessage implements Serializable {
    public final ActorRef subscriber;
    public SubscribeMessage(ActorRef subscriber) {
        this.subscriber = subscriber;
    }
}

class UnsubscribeMessage implements Serializable {
    public final ActorRef unsubscribe;
    public UnsubscribeMessage(ActorRef unsubscribe) {
        this.unsubscribe = unsubscribe;
    }
}

class Message implements Serializable {
    public final String s;
    public Message(String s) {
        this.s = s;
    }
}

```

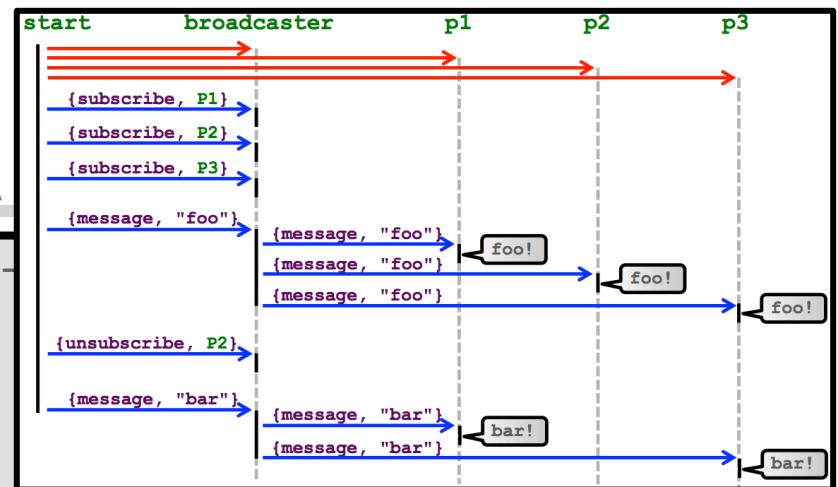


purses half price!
 purses half price!
 purses half price!
 shoes half price!!
 shoes half price!!

3) Broadcast.java

```
// -- MAIN -->

public class Broadcast {
    public static void main(String[] args) {
        final ActorSystem system =
            ActorSystem.create("BroadcastSystem");
        final ActorRef broadcaster =
            system.actorOf(Props.create(BroadcastActor.class), "broadcaster");
        final ActorRef p1 = system.actorOf(Props.create(PersonActor.class), "p1");
        final ActorRef p2 = system.actorOf(Props.create(PersonActor.class), "p2");
        final ActorRef p3 = system.actorOf(Props.create(PersonActor.class), "p3");
        broadcaster.tell(new SubscribeMessage(p1), ActorRef.noSender());
        broadcaster.tell(new SubscribeMessage(p2), ActorRef.noSender());
        broadcaster.tell(new SubscribeMessage(p3), ActorRef.noSender());
        broadcaster.tell(new Message("purses half price!"), ActorRef.noSender());
        broadcaster.tell(new UnsubscribeMessage(p2), ActorRef.noSender());
        broadcaster.tell(new Message("shoes half price!!!"), ActorRef.noSender());
        try {
            System.out.println("Press return");
            System.in.read();
        } catch(IOException e) {
            e.printStackTrace();
        } finally {
            system.shutdown();
        }
    }
}
```



```
start() ->
Broadcaster = spawn(helloworld,broadcaster,[]),
P1 = spawn(helloworld,person,[]),
P2 = spawn(helloworld,person,[]),
P3 = spawn(helloworld,person,[]),
Broadcaster ! {subscribe,P1},
Broadcaster ! {subscribe,P2},
Broadcaster ! {subscribe,P3},
Broadcaster ! {message,"Purses half price!"},
Broadcaster ! {unsubscribe,P2},
Broadcaster ! {message,"Shoes half price!!!"}.
```

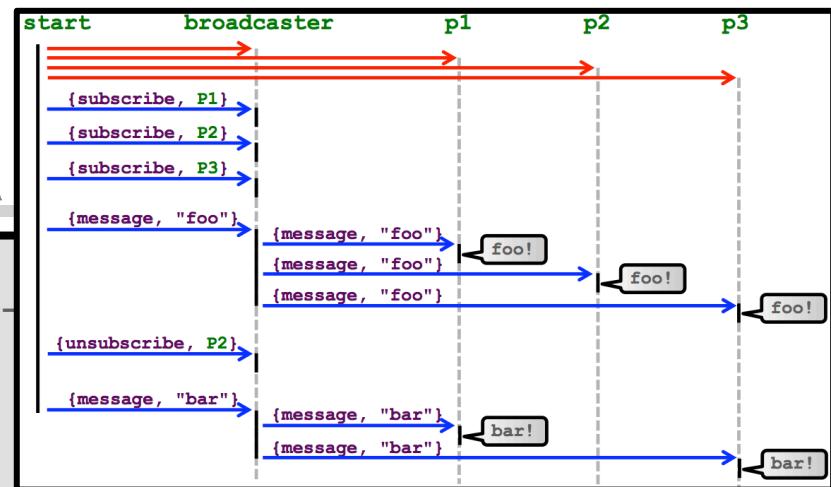
3) Broadcast.java

```
// -- ACTORS ---

class PersonActor extends UntypedActor {
    public void onReceive(Object o) throws Exc' {
        if (o instanceof Message) {
            System.out.println(((Message) o).s);
        }
    }
}

class BroadcastActor extends UntypedActor {
    private List<ActorRef> list =
        new ArrayList<ActorRef>();

    public void onReceive(Object o) throws Exception {
        if (o instanceof SubscribeMessage) {
            list.add(((SubscribeMessage) o).subscriber);
        } else if (o instanceof UnsubscribeMessage) {
            list.remove(((UnsubscribeMessage) o).unsubscriber);
        } else if (o instanceof Message) {
            for (ActorRef person : list) {
                person.tell(o, getSelf());
            }
        }
    }
}
```



```

person() ->
receive
{message,M} ->
io:fwrite(M ++ "\n"),
person()
end.
  
```

```

broadcast([],_) -> true;
broadcast([Pid|L],M) ->
  Pid ! {message,M},
  broadcast(L,M).

broadcaster(L) ->
receive
{subscribe,Pid} ->
broadcaster([Pid|L]);
{unsubscribe,Pid} ->
broadcaster( L \ Pid );
{message,M} ->
broadcast(L,M),
broadcaster(L)
end.
  
```

3) Broadcast.java

■ Compile:

```
javac -cp scala.jar:akka-actor.jar Broadcast.java
```

■ Run:

```
java -cp scala.jar:akka-actor.jar:akka-config.jar:. Broadcast
```

■ Output:

```
purses half price!
purses half price!
purses half price!
shoes half price!!
shoes half price!!
```

AGENDA



■ 3) Broadcast:

- From ERLANG to JAVA+AKKA
- Communication protocols (one-to-one \Rightarrow one-to-many)

■ AKKA: A proper introduction

- Motivations and benefits of Actors & Message Passing
- Recommendations

■ 4) Primer:

- Hierarchic organization: managers supervise workers
- Performance: MacBook Air -vs- MTLab Server

■ ★ Scatter-Gather:

- Prototypical AKKA Service (dynamic load balancing)
- Extensions

AKKA

■ Mountain in Sweden:

- Northern Sweden
- (Close to Norway)



■ Nordic Goddesses: "Àhkkas"

- From Nordic/Arctic/Saami Mythology
- The Àhkkas: daughters of Mother Sun
- Ancient creator goddesses of the past



■ Software runtime middleware:

- For Java Virtual Machine (made in Scala)





Telefonica



SIEMENS

amazon.com®



HSBC



KLOUT

banksimple



Autodesk

CREDIT SUISSE

IGN

Atos

O₂



vmware®



Smart Stream Platform



xerox

novus

DRW TRADING GROUP

navirec

OOYALA

T8 Webware

CARTOMAPIC



BBC

JUNIPER
NETWORKS

ngmoco:)

Maritime Poker

Answers.com®
The world's leading Q&A site

azavea

zeebox
The best thing to happen to TV since TV

Why the Sudden Popularity?!?

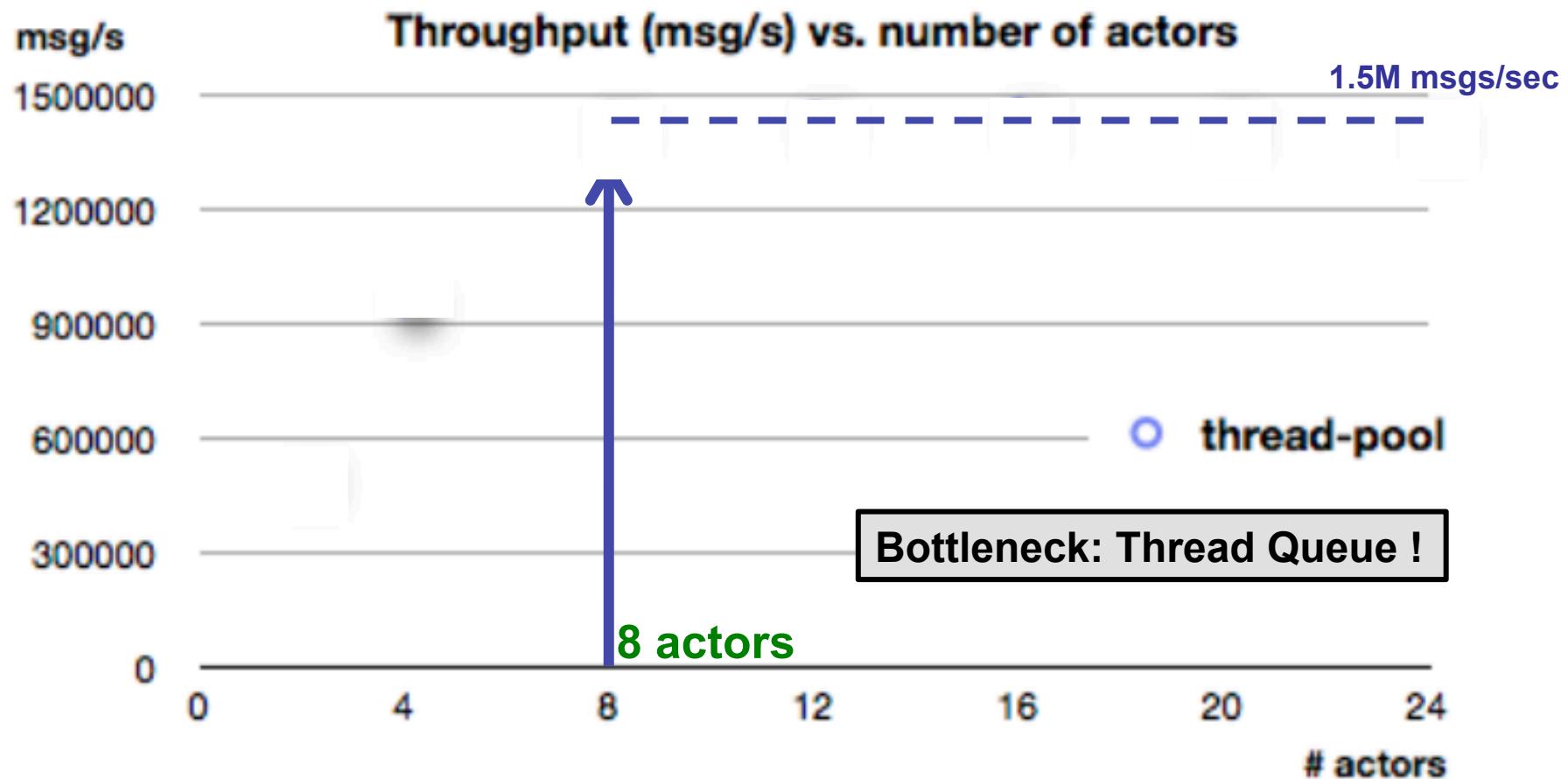
- Recently, processor speed "hit the wall":
 - Speed of light: $c = 3 \cdot 10^8$ m/s (meters per sec)
 - Processor speed: $s = 3 \cdot 10^9$ x/s (instructions per sec)
 -  $c/s = (3 \cdot 10^8 \text{m/s} / 3 \cdot 10^9 \text{x/s}) = 0.1 \text{ m/x}$ (meters per instruction)
 - i.e., "***light travels 10cm per instruction***"
- Before (more speed):
 -  **Moore's Law:** #Transistors-per-CPU **doubles** every 1.8 years
 -  **Dennard Scaling:** Performance-per-Watt **doubled** every 1.57 years
- Now (more speed):
 - We have to ***increase parallelism*** !
 - Recent developments: "***Work Stealing Queue***"

used for:
managing
the actors
effectively

Chase and Lev: "***Dynamic Circular Work-Stealing Queue***", SPAA 2005
Michael, Vechev, Saraswat: "***Idempotent Work Stealing***", PPoPP 2009

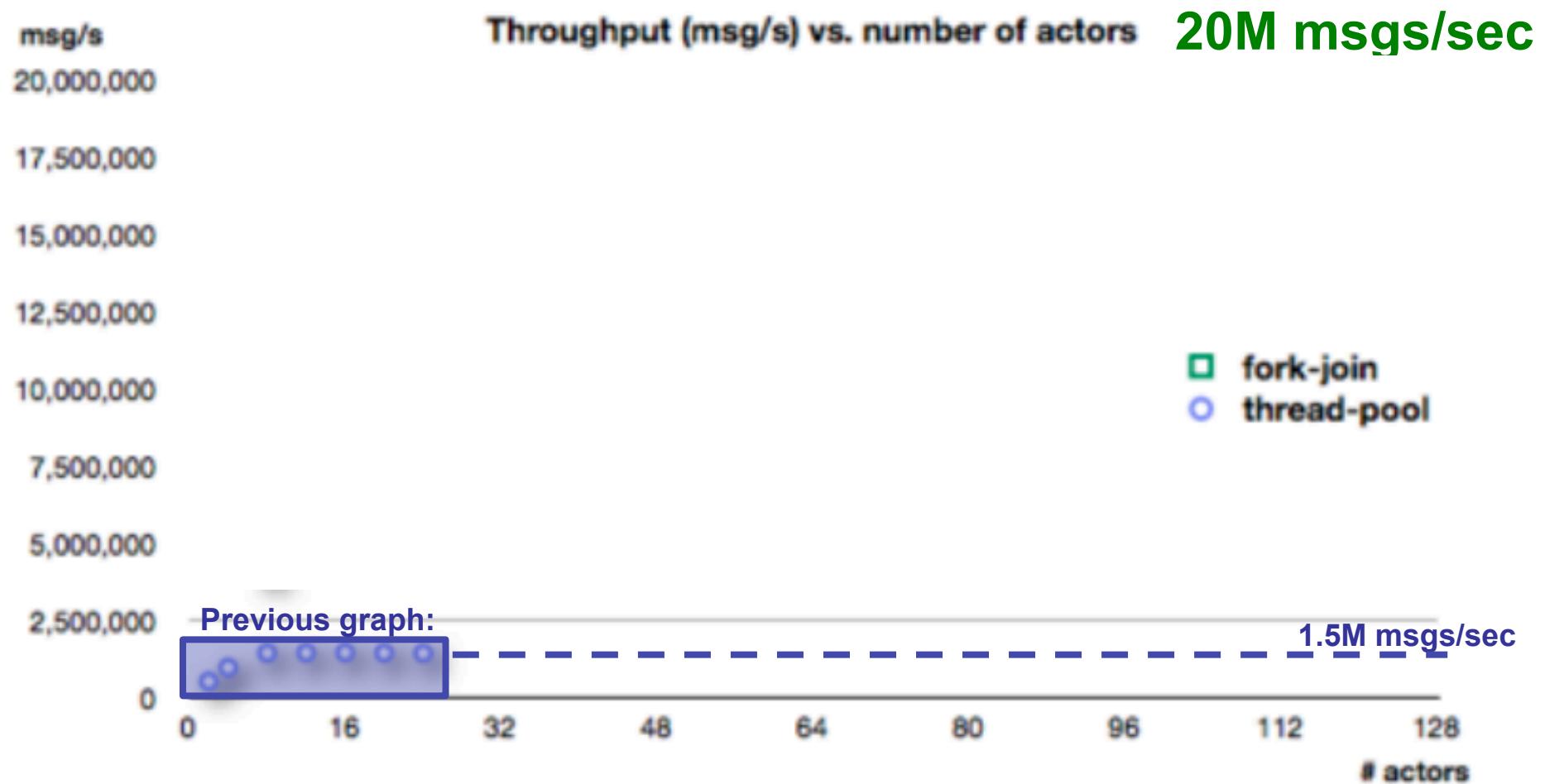
Scalability? :-()

- Using a conventional "Thread Pool Executor":



Scalability!

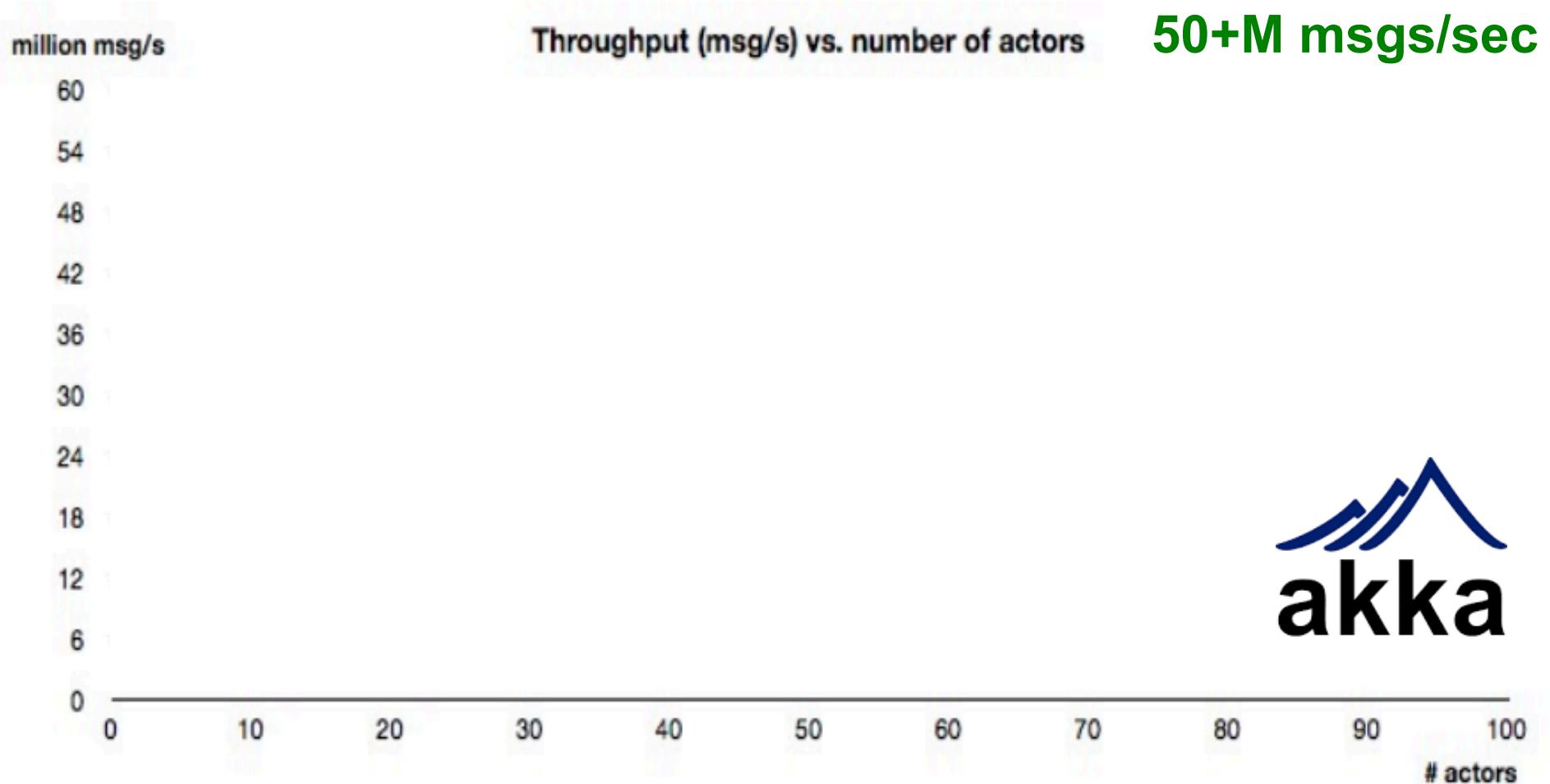
- Java 8's New Fork Join Pool (Work Stealing):



[From "UP UP AND OUT: SCALING SOFTWARE WITH AKKA", Jonas Boner, GOTO Conference, Aarhus, Denmark]

Scalability! :-)

- ...and after optimizing for throughput:



[From "UP UP AND OUT: SCALING SOFTWARE WITH AKKA", Jonas Boner, GOTO Conference, Aarhus, Denmark]

An Actor is....:

- A fundamental *unit of computation* that embodies:
 - **1) Processing**
 - **2) State**
 - **3) Communication**
- In particular: no **shared && mutable state!**
- When an actor receives a message, it can....:
 - **1) perform computation**
 - **2) change state**
 - **3) send messages to actors it knows**
 - **4) spawn new actors**



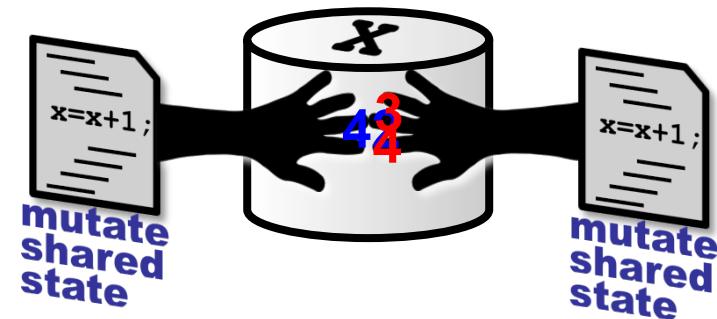
Actors & Msg Passing: Benefits

- **Correct highly concurrent systems:**
 - Higher level abstraction (via message passing)
 - coordination (declarative/what) \neq business logic (imperative/how)
 - No low-level locking (no **shared & mutable** state)
- **Truly scalable systems:**
 - Actors are extremely lightweight entities
 - Actors are location transparent
 - Distributable-by-design
 - Transparently map MP programs onto given hardware:
 - "Scale up" (more processors), "Scale out" (more machines)
- **Self-healing, fault-tolerant systems:**
 - Adaptive load balancing and actor migration
 - "Let it crash" model (deal w/ failure, great success in telecom industry)
 - Manage system overload (graceful service degradation)

The People Metaphor

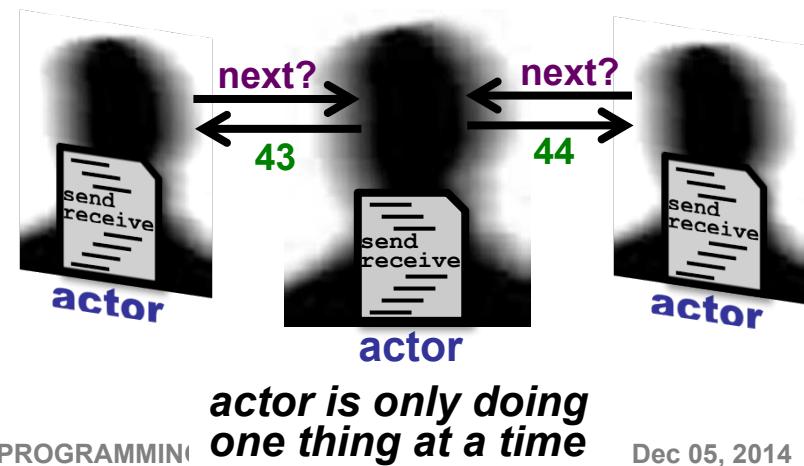
■ Shared Mutable state:

- Concurrency problems!
 - **Shared && mutable !**
- Hard to (later) distribute!



■ Actors (with encapsulated state):

- Can't "look inside head" of a living person (actor) !
- Instead: **ask questions ?**
- ...and **get responses !**
- ...**one at a time !**



The People Metaphor (cont'd)

- **Programming Metaphor:**

- "The People Analogy"



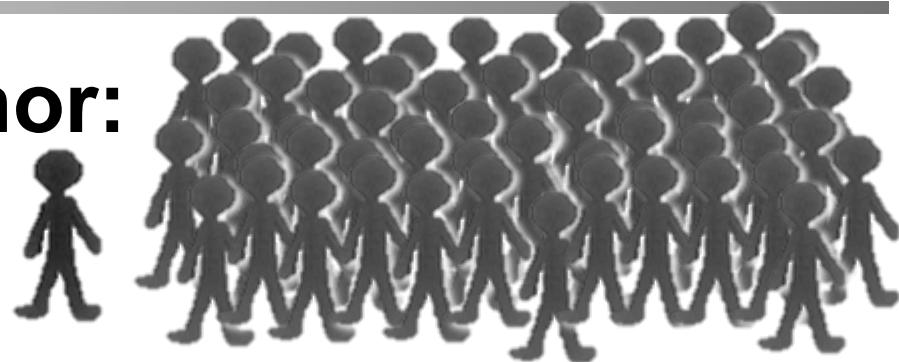
- Think of it as "**coordinating lots of people**":

- Each can do simple tasks
 - Consider workflow (of orders/messages in your system)

The People Metaphor (cont'd)

- **Programming Metaphor:**

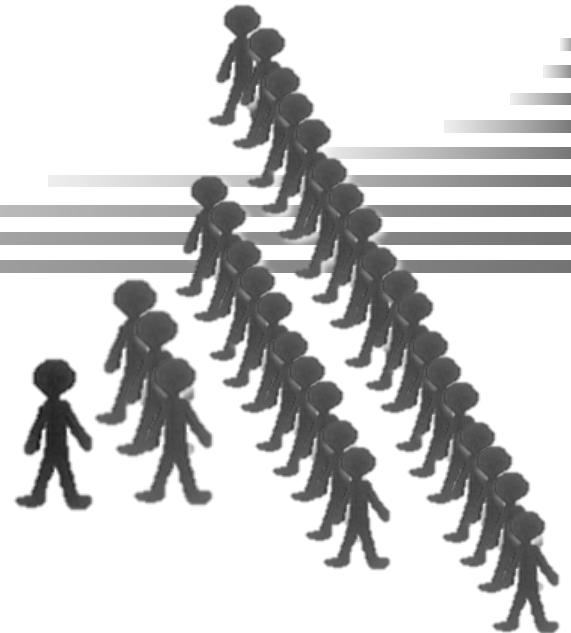
- "The People Analogy"



- Think of it as "**coordinating lots of people**":

- Each can do simple tasks
 - Consider workflow (of orders/messages in your system)
 - Need more work ⇒ hire more people (spawn more actors)

The People Metaphor



- Programming Metaphor:
 - "The People Analogy"
- Think of it as "**coordinating lots of people**":
 - Each can do simple tasks
 - Consider workflow (of orders/messages in your system)
 - Need more work ⇒ hire more people (spawn more actors)
 - Hierarchic organization: managers supervise workers
 - Fault tolerance (expect failures and deal with them)

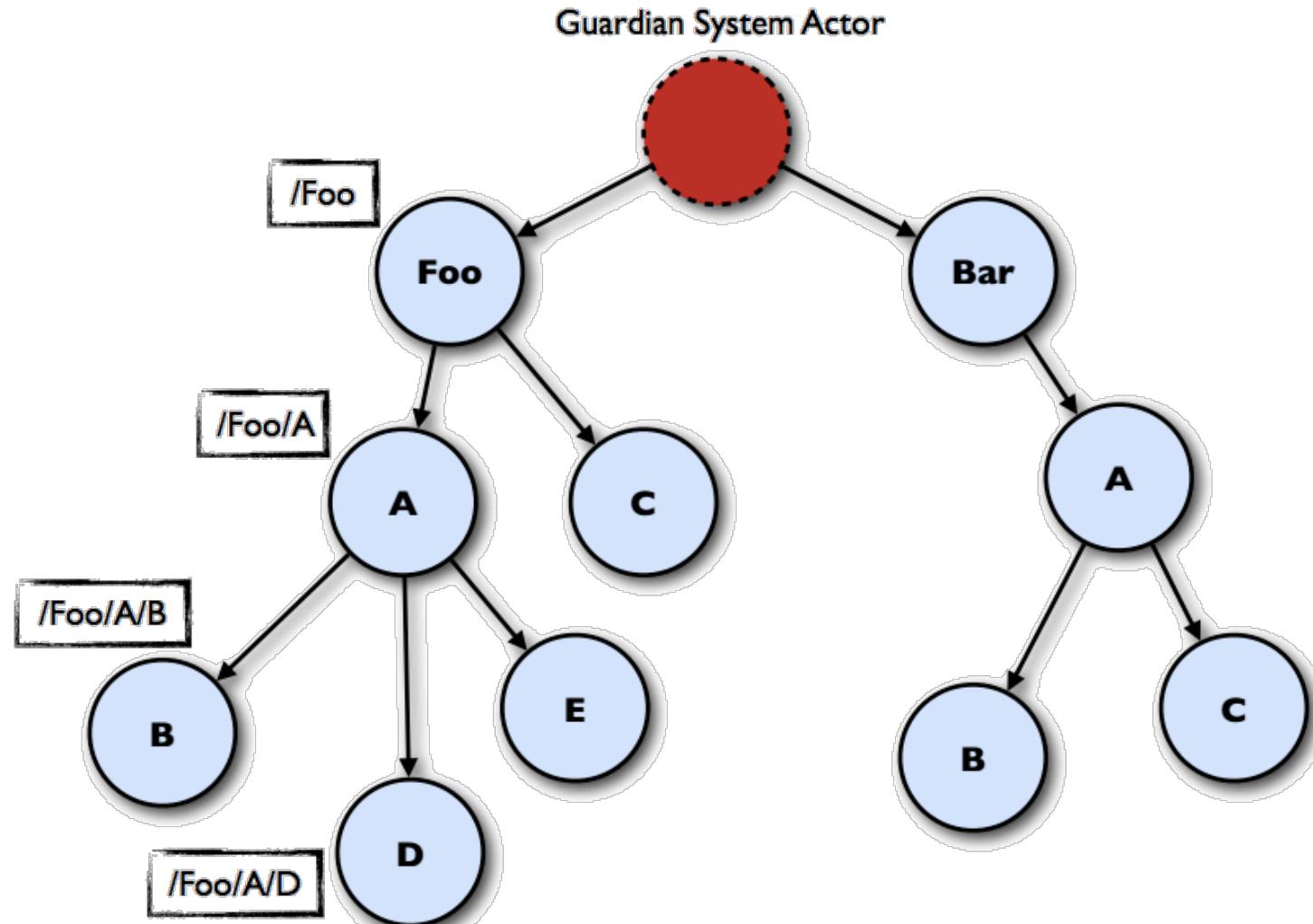
Recommendations

- **1) Actors should be like nice co-workers:**
 - do their job effectively w/o bothering others needlessly
 - should not roll thumbs (idle or blocking operations)
- **2) Actors should not send mutable objects:**
 - O/w we're back to "shared && mutable" ⇒ problems !
- **3) Actors should send data, not programs:**
 - O/w we're back to "shared && mutable" ⇒ problems !
 - (Note: ERLANG does not have higher-order functions)
- **4) Create few top-level actors:**
 - If these crash, your whole system will crash
 - If their workers die, they just hire (spawn) new ones

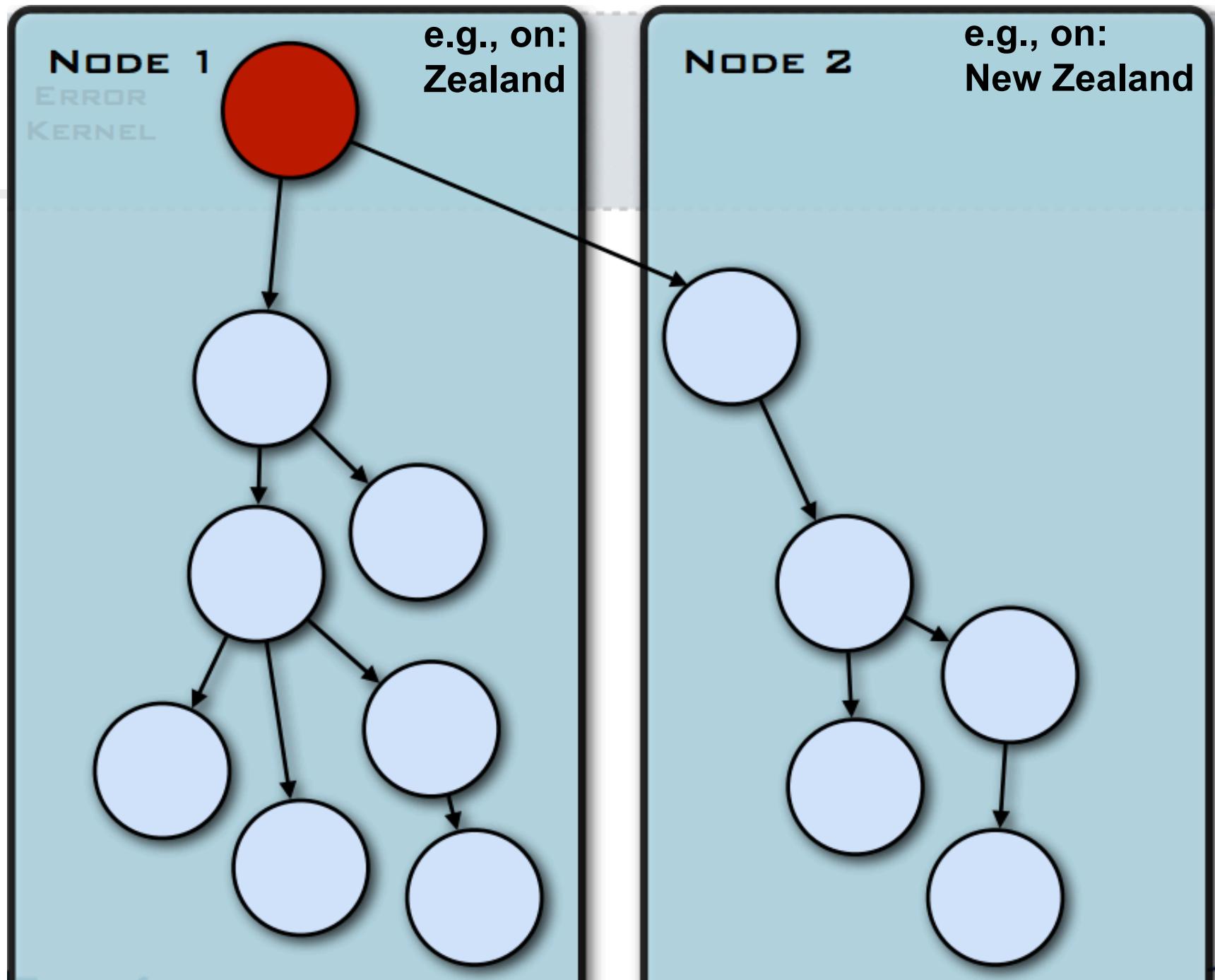
Recommendations

- **5) Managers should supervise Workers:**
 - organization as a hierarchy
 - pass on and schedule tasks for workers
 - "hire" (spawn) more workers by need
 - deal with failure (of your workers)
- **6) Actors should spawn workers for "dangerous operations" (Qatar 2022 ?):**
 - avoid crashing with important data
 - spawn workers for "dangerous operations"
 - deal with failure (of your workers)

A Hierarchy of Actors



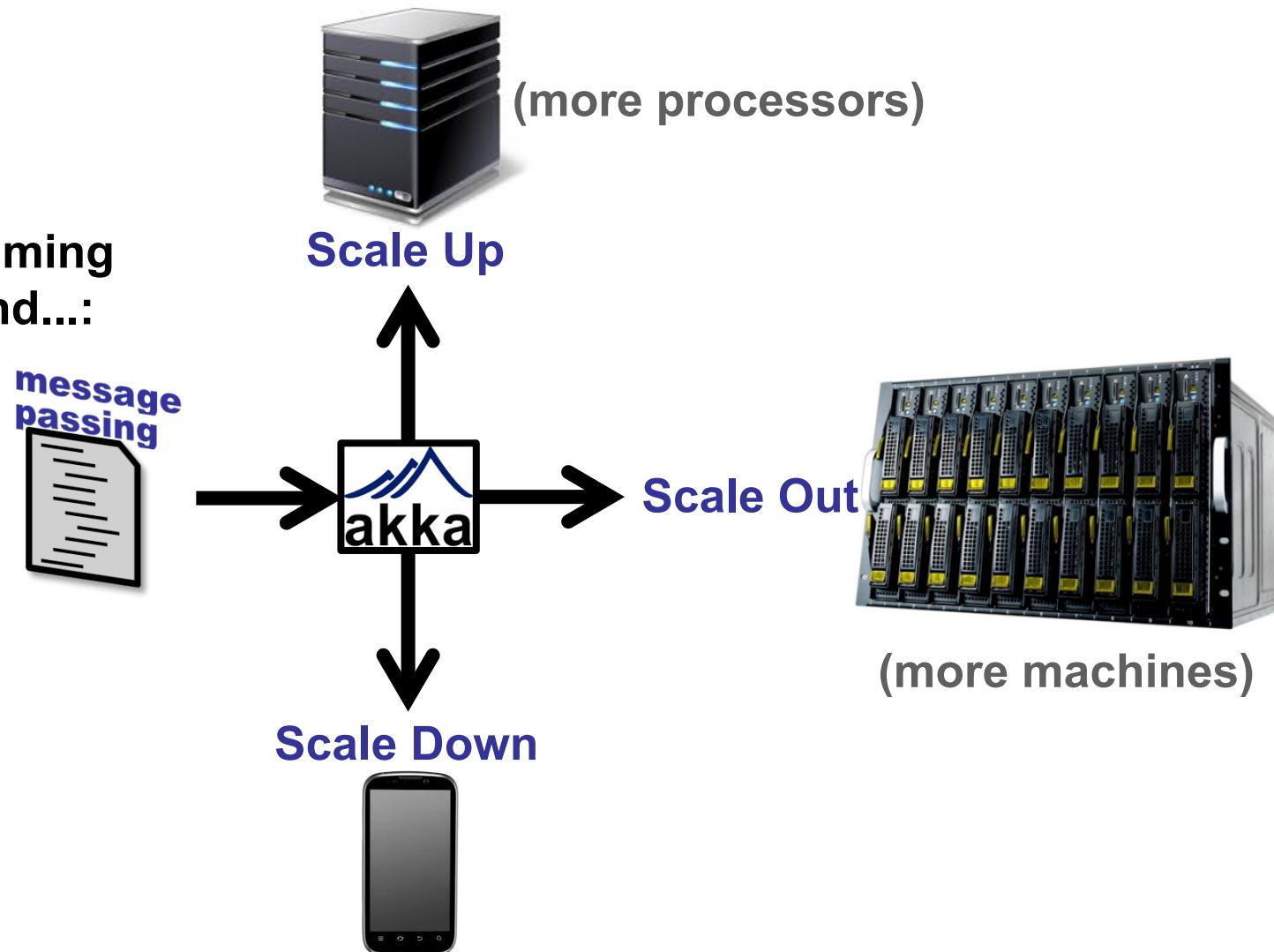
[From "UP UP AND OUT: SCALING SOFTWARE WITH AKKA", Jonas Boner, GOTO Conference, Aarhus, Denmark]



[From "UP UP AND OUT: SCALING SOFTWARE WITH AKKA", Jonas Boner, GOTO Conference, Aarhus, Denmark]

Scale Down, Scale Up, Scale Out

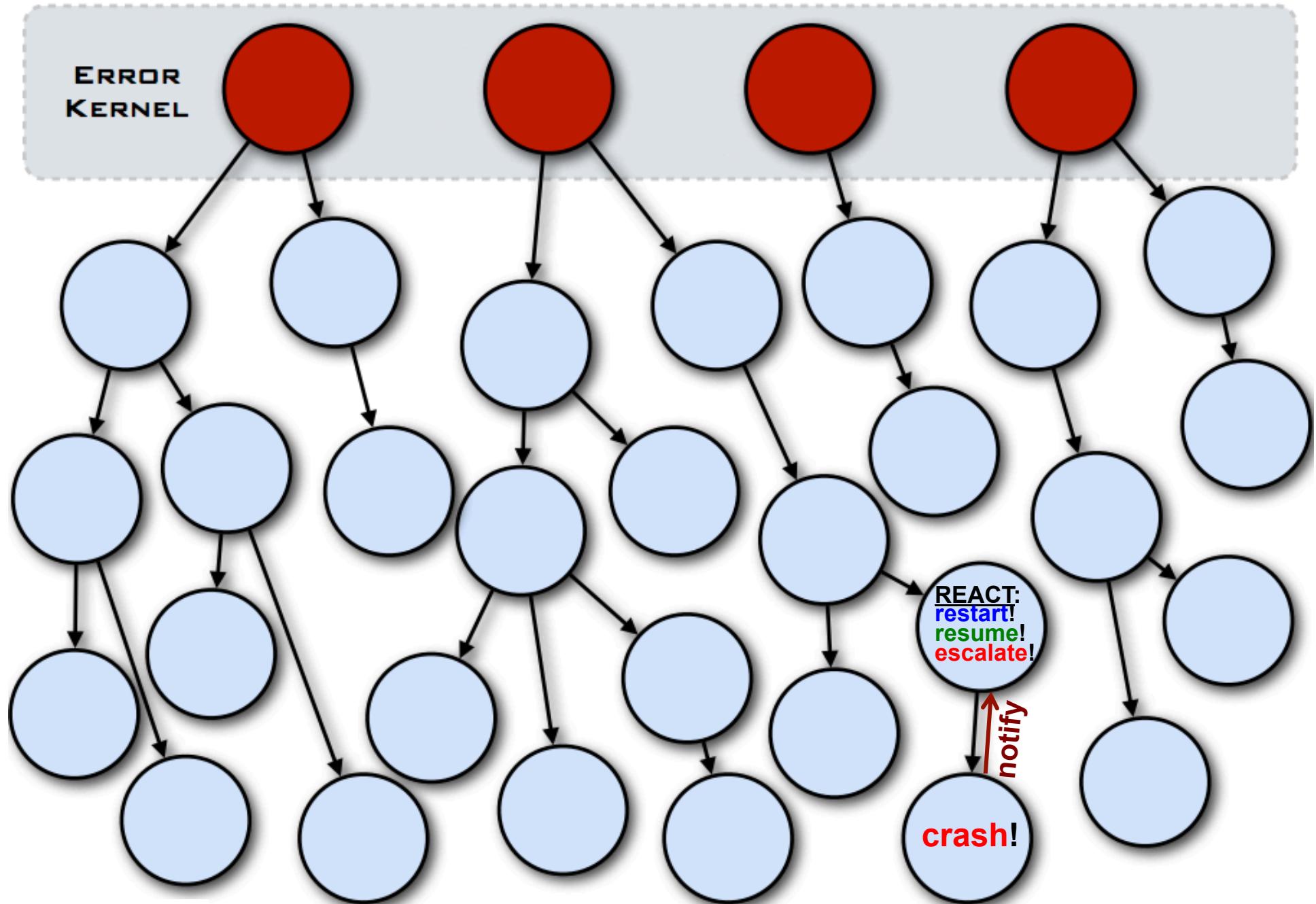
When programming
with **actors** and...:



Fault Tolerance



[From "UP UP AND OUT: SCALING SOFTWARE WITH AKKA", Jonas Boner, GOTO Conference, Aarhus, Denmark]



[From "UP UP AND OUT: SCALING SOFTWARE WITH AKKA", Jonas Boner, GOTO Conference, Aarhus, Denmark]

AGENDA



■ 3) Broadcast:

- From ERLANG to JAVA+AKKA
- Communication protocols (one-to-one \Rightarrow one-to-many)

■ AKKA: A proper introduction

- Motivations and benefits of Actors & Message Passing
- Recommendations

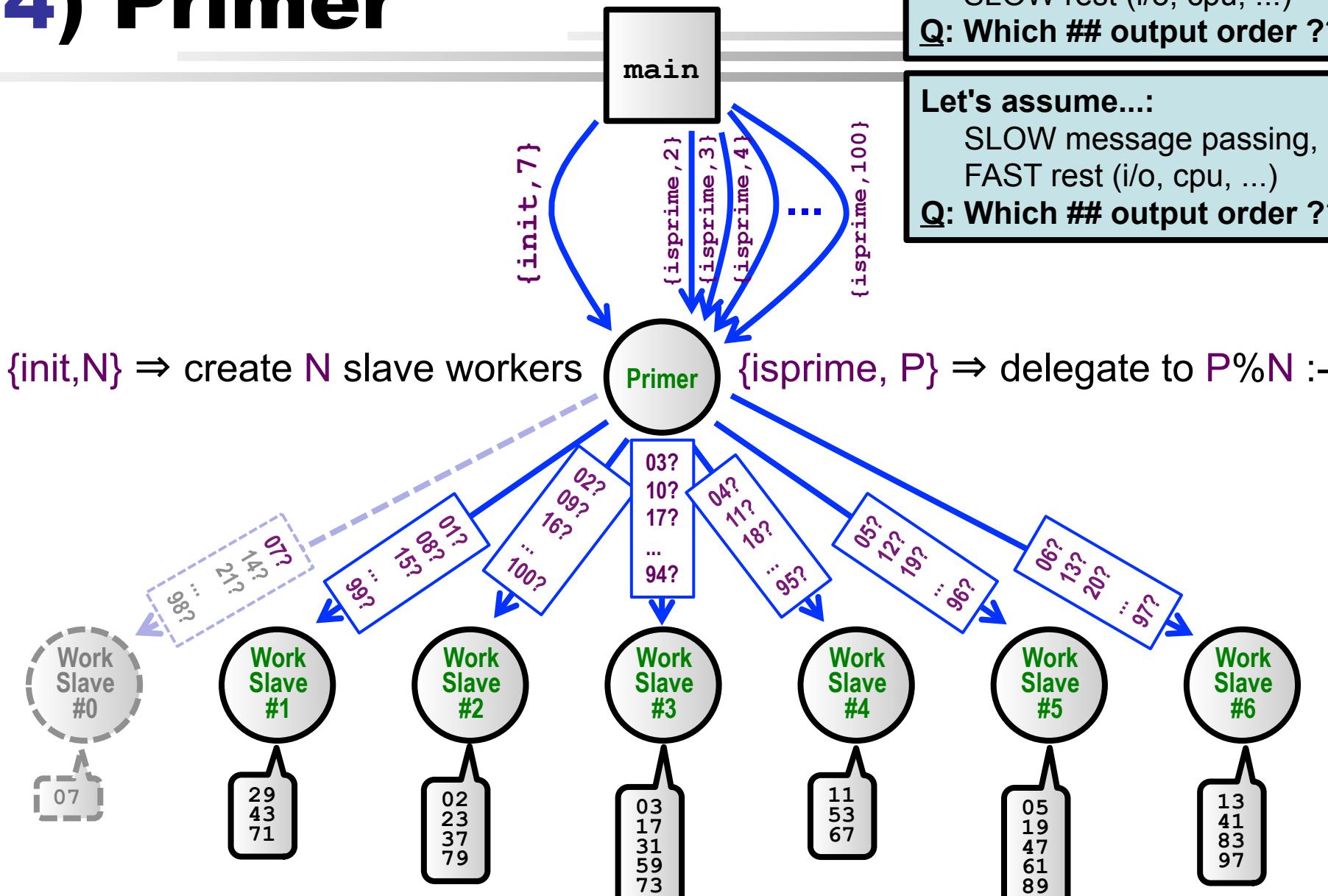
■ 4) Primer:

- Hierarchic organization: managers supervise workers
- Performance: MacBook Air -vs- MTLab Server

■ ★ Scatter-Gather:

- Prototypical AKKA Service (dynamic load balancing)
- Extensions

4) Primer



4) Primer.erl

```
-module(helloworld).  
-export([start/0,slave/1,primer/1]).  
  
is_prime_loop(N,K) ->  
    K2 = K * K, R = N rem K,  
    case (K2 =< N) and (R /= 0) of  
        true -> is_prime_loop(N, K+1);  
        false -> K  
    end.  
  
is_prime(N) ->  
    K = is_prime_loop(N,2),  
    (N >= 2) and (K*K > N).  
  
n2s(N) ->  
    lists:flatten(io_lib:format("~p", [N])).  
  
slave(Id) ->  
    receive  
        {isprime, N} ->  
            case is_prime(N) of  
                true -> io:fwrite("(" ++  
n2s(Id) ++ ")" ++ n2s(N) ++ "\n");  
                false -> []  
            end,  
            slave(Id)  
    end.
```

Slave

```
create_slaves(Max,Max) -> [];  
create_slaves(Id,Max) ->  
    Slave = spawn(helloworld,slave,[Id]),  
    [Slave|create_slaves(Id+1,Max)].  
  
primer(Slaves) ->  
    receive  
        {init, N} when N=<0 ->  
            throw({nonpositive,N}) ;  
        {init, N} ->  
            primer(create_slaves(0,N)) ;  
        {isprime, _} when Slaves == [] ->  
            throw({uninitialized}) ;  
        {isprime, N} when N=<0 ->  
            throw({nonpositive,N}) ;  
        {isprime, N} ->  
            SlaveId = N rem length(Slaves),  
            lists:nth(SlaveId+1, Slaves)  
            ! {isprime,N},  
            primer(Slaves)  
    end.  
  
spam(_, N, Max) when N>=Max -> true;  
spam(Primer, N, Max) ->  
    Primer ! {isprime, N},  
    spam(Primer, N+1, Max).  
  
start() ->  
    Primer =  
        spawn(helloworld, primer, [[]]),  
    Primer ! {init,7},  
    spam(Primer, 2, 100).
```

Primer

4) Primer.java

```
import java.util.*;
import java.io.*;
import akka.actor.*;

// -- MESSAGES ----

class InitializeMessage implements Serializable {
    public final int number_of_slaves;
    public InitializeMessage(int number_of_slaves) {
        this.number_of_slaves = number_of_slaves;
    }
}

class IsPrimeMessage implements Serializable {
    public final int number;
    public IsPrimeMessage(int number) {
        this.number = number;
    }
}
```



4) Primer.java

```
// -- SLAVE ACTOR -----  
  
class SlaveActor extends UntypedActor {  
    private boolean isPrime(int n) {  
        int k = 2;  
        while (k * k <= n && n % k != 0) k++;  
        return n >= 2 && k * k > n;  
    }  
  
    public void onReceive(Object o) throws Exception {  
        if (o instanceof IsPrimeMessage) {  
            int p = ((IsPrimeMessage) o).number;  
            if (isPrime(p)) System.out.println("(" + p % Primer.P + ") " + p); // HACK  
        }  
    }  
}
```

4) Primer.java

```
// -- PRIME ACTOR -----
class PrimeActor extends UntypedActor {
    List<ActorRef> slaves;

    private List<ActorRef> createSlaves(int n) {
        List<ActorRef> slaves = new ArrayList<ActorRef>();
        for (int i=0; i<n; i++) {
            ActorRef slave =
                getContext().actorOf(Props.create(SlaveActor.class), "p" + i);
            slaves.add(slave);
        }
        return slaves;
    }

    public void onReceive(Object o) throws Exception {
        if (o instanceof InitializeMessage) {
            InitializeMessage init = (InitializeMessage) o;
            int n = init.number_of_slaves;
            if (n<=0) throw new RuntimeException("!!! non-positive number!");
            slaves = createSlaves(n);
            System.out.println("initialized (" + n + " slaves ready to work)!");
        } else if (o instanceof IsPrimeMessage) {
            if (slaves==null) throw new RuntimeException("!!! uninitialized!");
            int n = ((IsPrimeMessage) o).number;
            if (n<=0) throw new RuntimeException("!!! non-positive number!");
            int slave_id = n % slaves.size();
            slaves.get(slave_id).tell(o, getSelf());
        }
    }
}
```

4) Primer.java

```
// -- MAIN -----  
  
public class Primer {  
    private static void spam(ActorRef primer, int min, int max) {  
        for (int i=min; i<max; i++) {  
            primer.tell(new IsPrimeMessage(i), ActorRef.noSender());  
        }  
    }  
  
    public static void main(String[] args) {  
        final ActorSystem system = ActorSystem.create("PrimerSystem");  
        final ActorRef primer =  
            system.actorOf(Props.create(PrimeActor.class), "primer");  
        primer.tell(new InitializeMessage(7), ActorRef.noSender());  
        try {  
            System.out.println("Press return to initiate...");  
            System.in.read();  
            spam(primer, 2, 100);  
            System.out.println("Press return to terminate...");  
            System.in.read();  
        } catch(IOException e) {  
            e.printStackTrace();  
        } finally {  
            system.shutdown();  
        }  
    }  
}
```

4) Primer.java

■ Compile:

```
javac -cp scala.jar:akka-actor.jar Primer.java
```

■ Run:

```
java -cp scala.jar:akka-actor.jar:akka-config.jar:. Primer
```

■ Output:

```
press return to initiate...
initialized (7 slaves ready to work)!

(2) 2
(3) 3
Press return to terminate...
(0) 7
(5) 5
(4) 11
(6) 13
(3) 17
(5) 19
(2) 23
(1) 29
(3) 31
```

```
(2) 37
(6) 41
(1) 43
(5) 47
(4) 53
(3) 59
(5) 61
(4) 67
(1) 71
(3) 73
(2) 79
(6) 83
(5) 89
(6) 97
```

ERLANG

-VS-

JAVA+AKKA

■ ERLANG:

SLOW message passing*

=predicted=effect=>

You would get numbers in
slave-worker order:

- 07, 29, 02, 03, 11, 05, ...

[observed in ERLANG]

#0:	07
#1:	29
#2:	02
#3:	03
#4:	11
#5:	05
#6:	13
#1:	43
#2:	23
#3:	17
#4:	53
#5:	19
#6:	41

#2:	02
#3:	03
#5:	05
#0:	07
#4:	11
#6:	13
#3:	17
#5:	19
#2:	23
#1:	29
#3:	31
#2:	37
#6:	41

■ JAVA+AKKA:

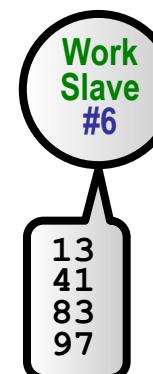
FAST message passing*

=predicted=effect=>

You would get numbers in
numerical order:

- 02, 03, 05, 07, 11, 13, ...

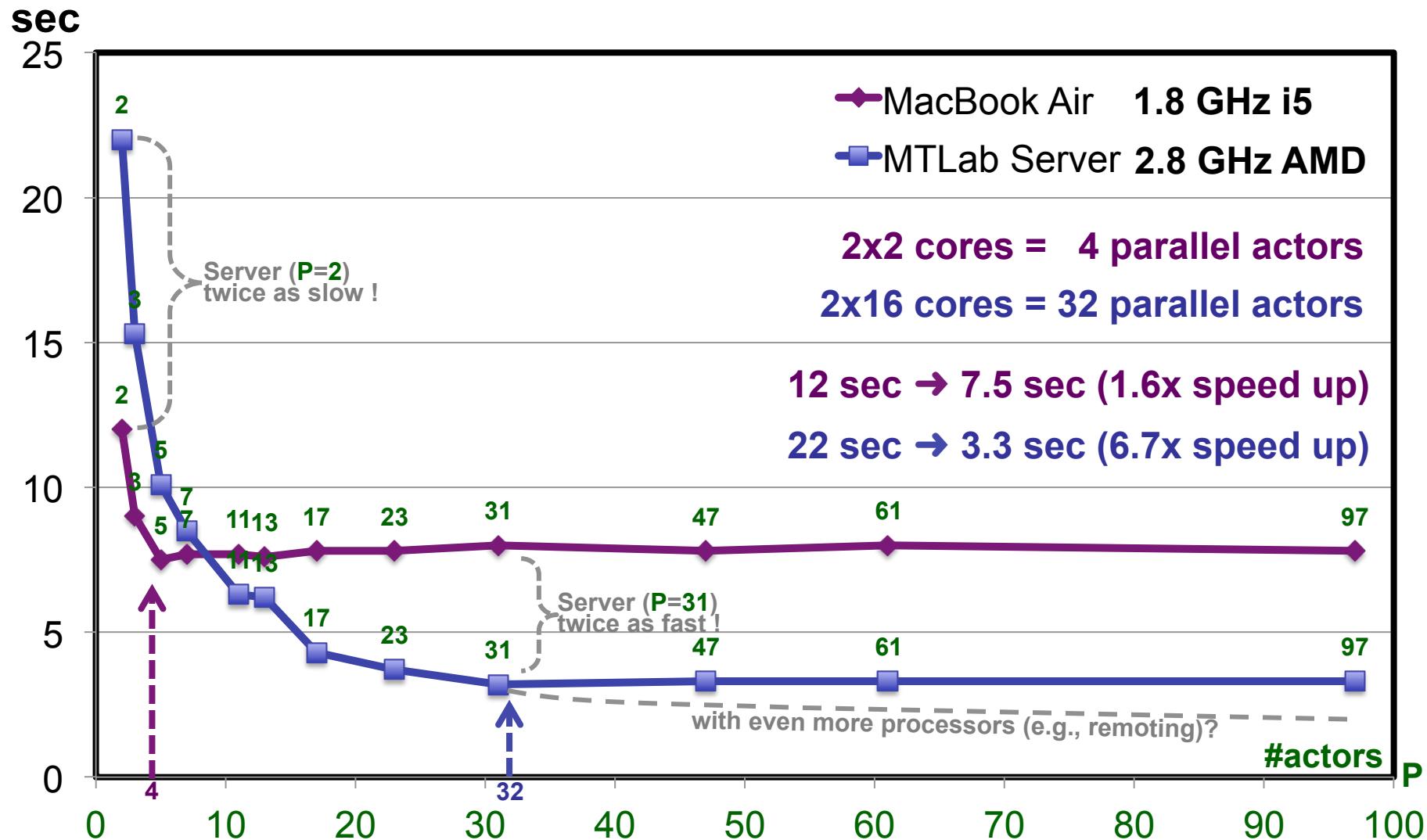
[observed in JAVA+AKKA]



*) relative to computation, I/O, ...

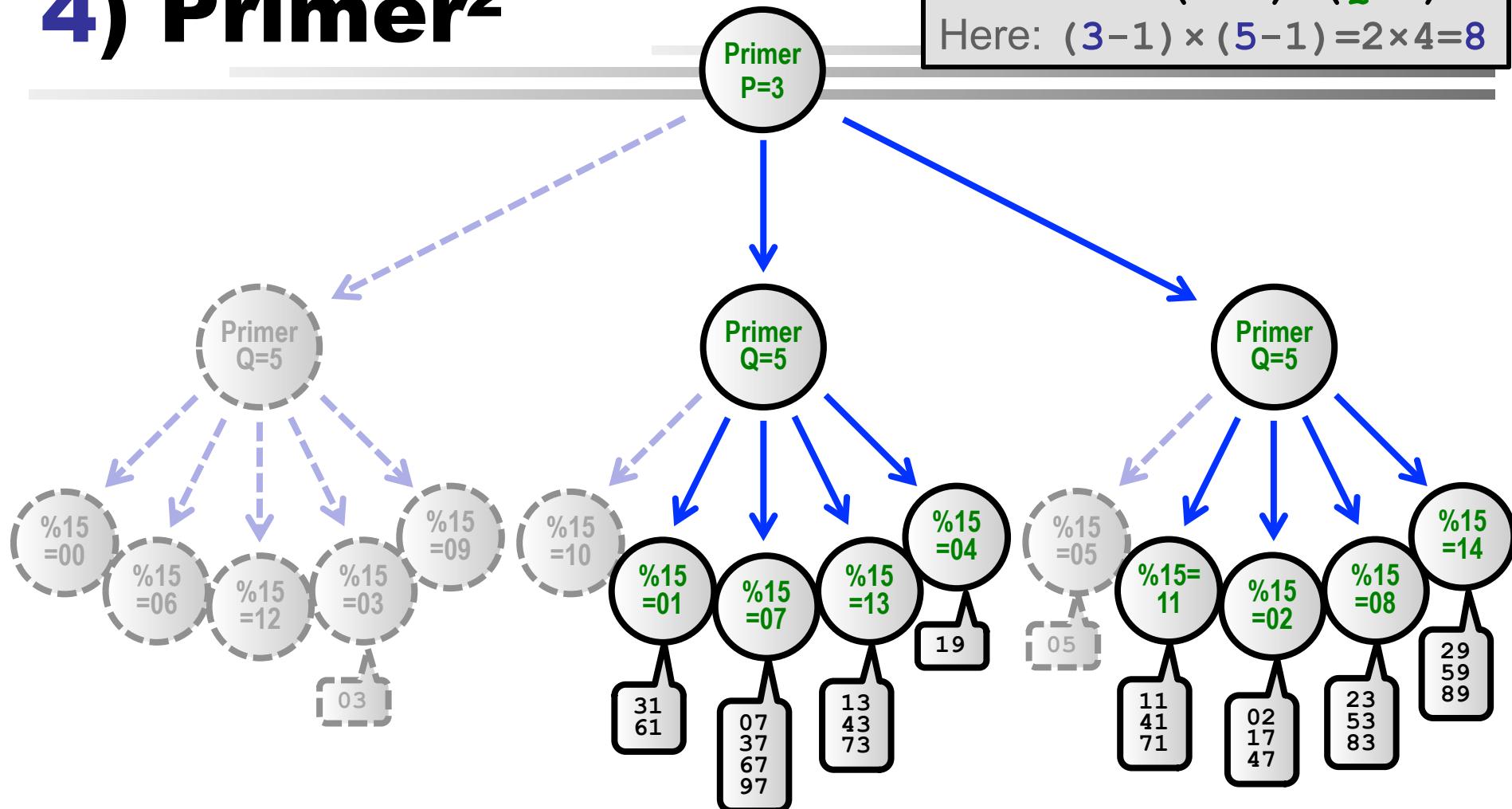
Note: added silly time-consuming computation to every `isPrime()` method call !

Low -vs- High Parallelization !



4) Primer²

#Workers: $P \times Q$
#Effective: $(P-1) \times (Q-1)$
 Here: $(3-1) \times (5-1) = 2 \times 4 = 8$



BTW: This is why you should always use a prime number in a hash function!

AGENDA



■ 3) Broadcast:

- From ERLANG to JAVA+AKKA
- Communication protocols (one-to-one \Rightarrow one-to-many)

■ AKKA: A proper introduction

- Motivations and benefits of Actors & Message Passing
- Recommendations

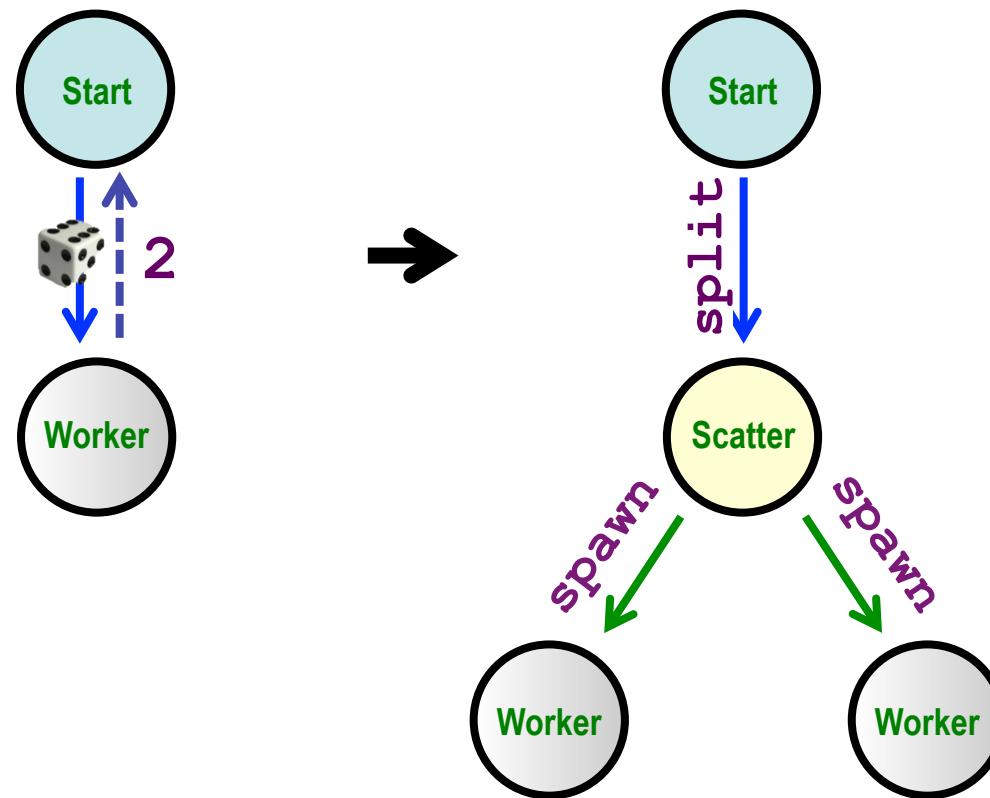
■ 4) Primer:

- Hierarchic organization: managers supervise workers
- Performance: MacBook Air -vs- MTLab Server

■ ★ Scatter-Gather:

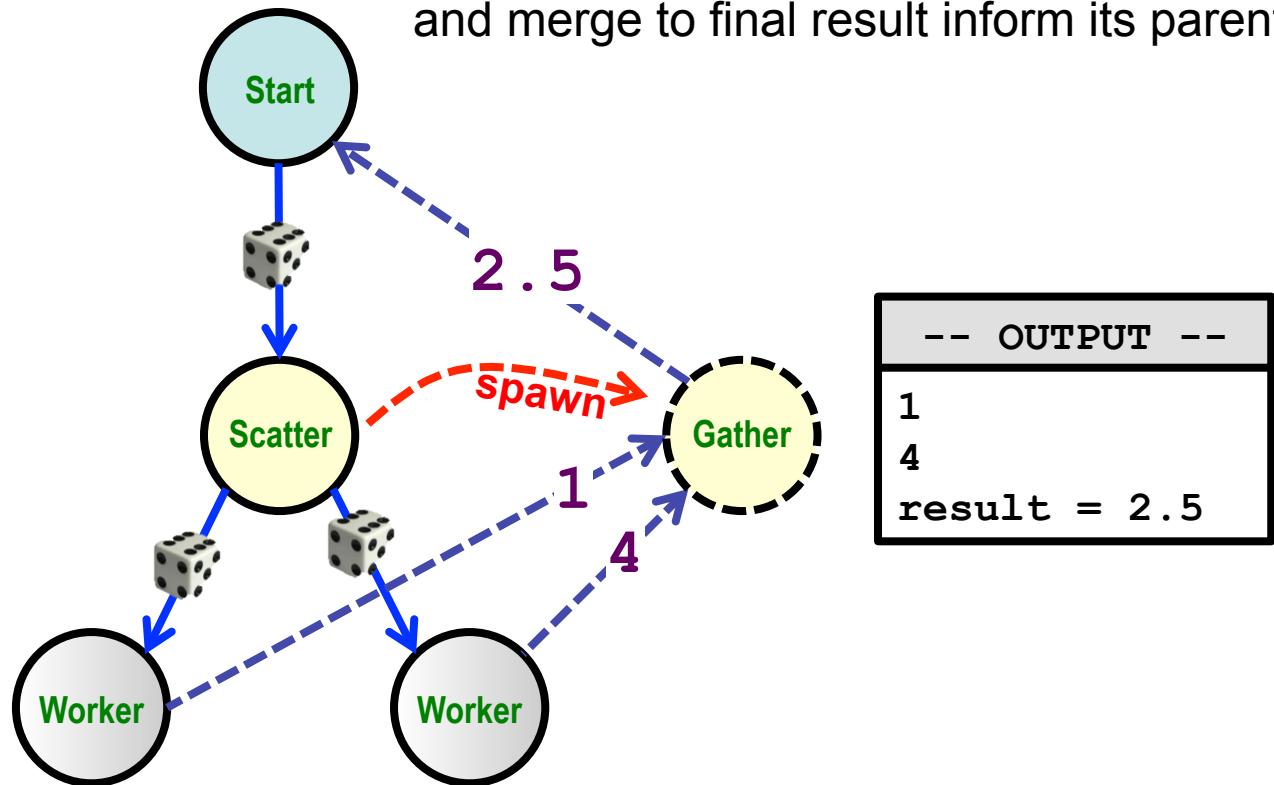
- Prototypical AKKA Service (dynamic load balancing)
- Extensions

6) Scatter-Gather



6) Scatter-Gather

Gatherer:
collect incoming responses
and merge to final result inform its parent



Scatter:

I don't want the result,
send it to my gatherer:

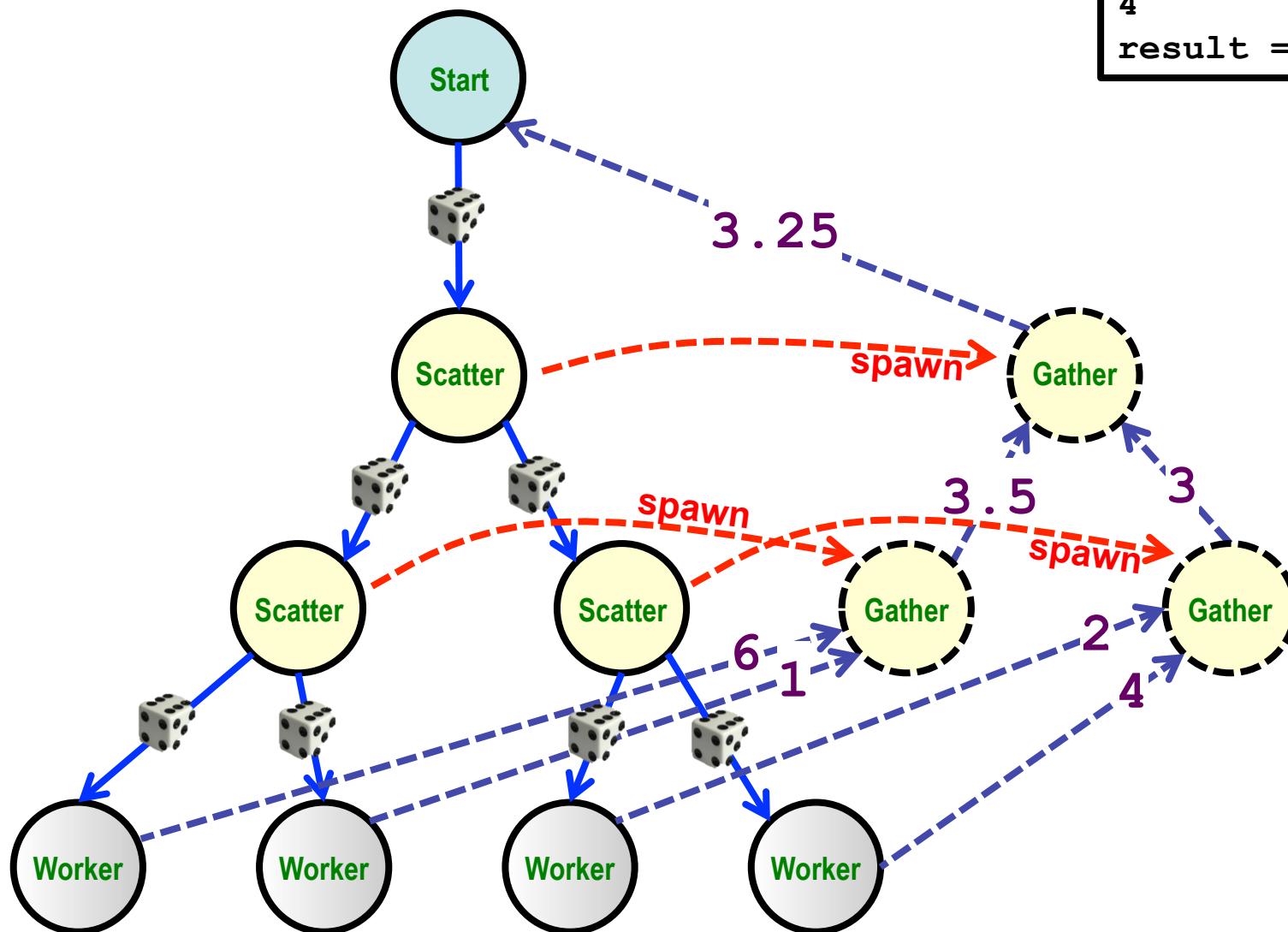
- I'm too busy processing new incoming requests
- besides, it's hard to correlate requests-responses (aka, "the correlation problem")

6) Scatter-Gather

--- OUTPUT ---

200

result = 3.25



6) ScatterGather.erl

```

-module(helloworld).
-export([start/0,worker/0,scatter/2,gather/1]).


%% -- COMPUTE ---

seed() -> {_, A2, A3} = now(), %% Seed wrt Time & Pid !
        random:seed(erlang:phash(node(), 100000),erlang:phash(A2, A3),A3).

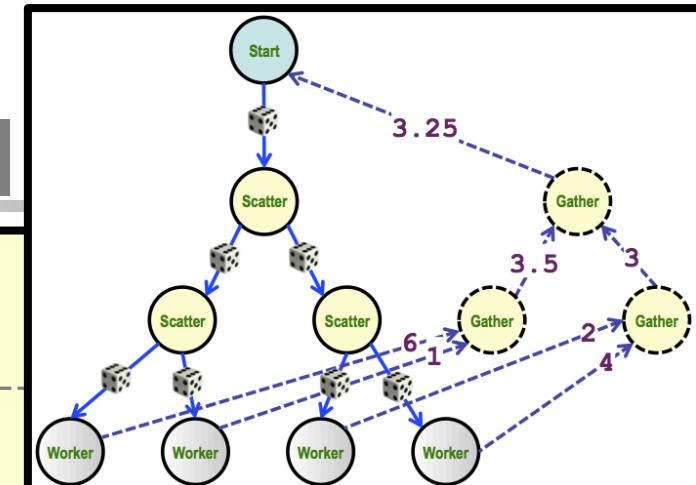
n2s(N) -> lists:flatten(io_lib:format("~p", [N])). %% HACK: num to string conversion!

random(N) -> random:uniform(N).

compute(X) -> random(X).

average(X,Y) -> (X + Y) / 2.

```



```
%% -- START -----
```

```

start() ->
    Worker = spawn(helloworld,worker,[]),
    Worker ! split,
    Worker ! split,
    Worker ! split,
    Worker ! {compute,6,self()},
    receive
        {result,R} ->
            io:fwrite("result = " ++ n2s(R) ++ "\n")
    end.

```

-- OUTPUT --

```

6
1
2
4
result = 3.25

```

6) ScatterGather.erl

```

%% -- WORKER -----
worker() ->
    seed(),
    receive
        split ->
            Left = spawn(helloworld,worker,[]),
            Right = spawn(helloworld,worker,[]),
            scatter(Left, Right) ;
        {compute,X,Caller} ->
            Res = compute(X),
            io:fwrite(n2s(Res) ++ "\n"),
            Caller ! {result,Res},
            worker()
    end.

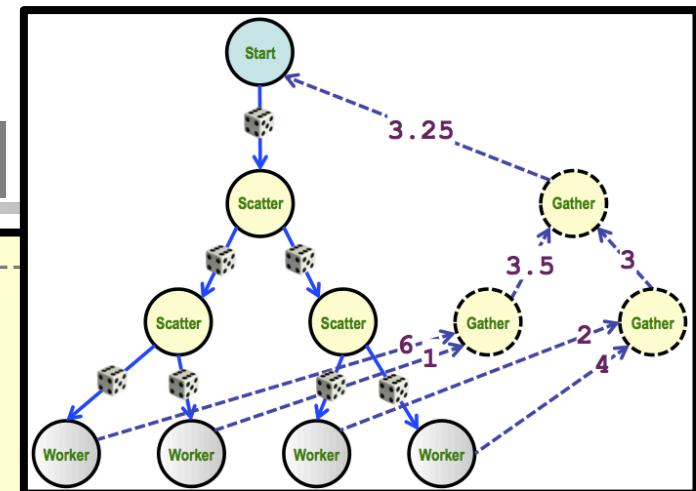
%% -- SCATTER -----
scatter(Left, Right) ->
    receive
        split ->
            Left ! split,
            Right ! split ;
        {compute,X,Caller} ->
            Gather = spawn(helloworld,gather,[Caller]),
            Left ! {compute,X,Gather},
            Right ! {compute,X,Gather}
    end,
    scatter(Left, Right).

```

```

%% -- GATHER -----
gather(Caller) ->
    receive
        {result,Res1} ->
            receive
                {result,Res2} ->
                    Res = average(Res1,Res2),
                    Caller ! {result, Res} % die!
            end
    end.

```



-- OUTPUT --

```

6
1
2
4
result = 3.25

```

6) ScatterGather.java

```
import java.util.Random;      import java.io.*;
import akka.actor.*;

// -- MESSAGES ---

class StartMessage implements Serializable { public StartMessage() { } }

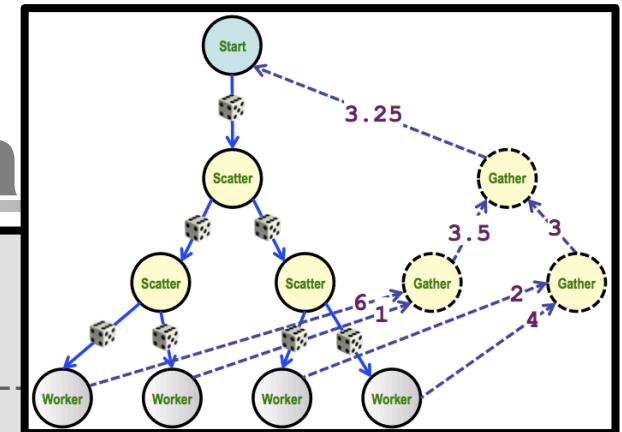
class SplitMessage implements Serializable { public SplitMessage() { } }

class CallerMessage implements Serializable {
    public final ActorRef caller;
    public CallerMessage(ActorRef caller) { this.caller = caller; }
}

class ComputeMessage implements Serializable {
    public final int number;
    public final ActorRef caller;
    public ComputeMessage(int number, ActorRef caller) {
        this.number = number;
        this.caller = caller;
    }
}

class ResultMessage implements Serializable {
    public final double result;
    public ResultMessage(double result) { this.result = result; }
}


```



-- OUTPUT --

```
6  
1  
2  
4  
result = 3.25
```

6) ScatterGather.java

```

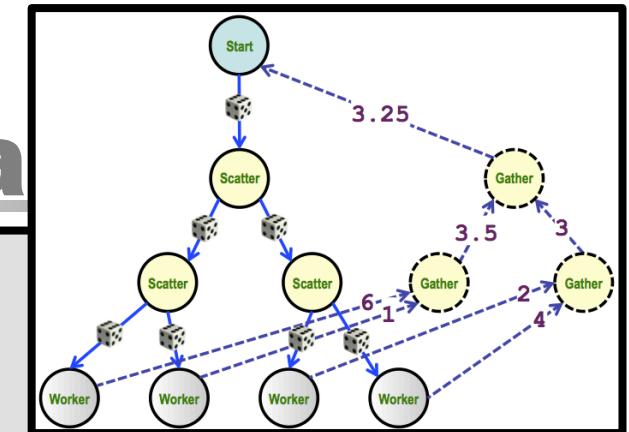
class WorkerScatterActor extends UntypedActor {
    // null => worker , non-null => scatter:
    private ActorRef left, right;
    private final Random rnd = new Random();
    private int random(int n) { return rnd.nextInt(n); }
    private int compute(int n) { return random(n) + 1; }

    private void worker(Object o) throws Exception {
        if (o instanceof SplitMessage) {
            left = getContext().actorOf(Props.create(WorkerScatterActor.class), "left");
            right = getContext().actorOf(Props.create(WorkerScatterActor.class), "right");
        } else if (o instanceof ComputeMessage) {
            ComputeMessage m = (ComputeMessage) o;
            int result = compute(m.number);
            System.out.println(result);
            m.caller.tell(new ResultMessage(result), ActorRef.noSender());
        }
    }

    private void scatter(Object o) throws Exception { [...] }

    public void onReceive(Object o) throws Exception {
        // dispatch according to actor role: 'worker' or 'scatter'
        if (left == null) worker(o);
        else scatter(o);
    }
}

```



-- OUTPUT --

```

6
1
2
4
result = 3.25

```

6) ScatterGather.java

```

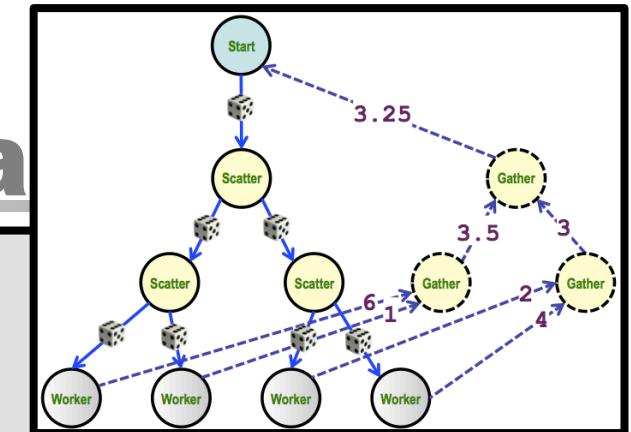
class WorkerScatterActor extends UntypedActor {
    // null => worker , non-null => scatter:
    private ActorRef left, right;
    [...]

    private void worker(Object o) throws Exception { [...] }

    private void scatter(Object o) throws Exception {
        if (o instanceof SplitMessage) {
            left.forward(o, getContext());
            right.forward(o, getContext());
        } else if (o instanceof ComputeMessage) {
            ComputeMessage m = (ComputeMessage) o;
            ActorRef gather = getContext().actorOf(Props.create(GatherActor.class), "g");
            // send message with callter, instead of arguments to gather constructor:
            gather.tell(new CallerMessage(m.caller), ActorRef.noSender());
            left.tell(new ComputeMessage(m.number, gather), ActorRef.noSender());
            right.tell(new ComputeMessage(m.number, gather), ActorRef.noSender());
        }
    }

    public void onReceive(Object o) throws Exception {
        // dispatch according to actor role: 'worker' or 'scatter'
        if (left == null) worker(o);
        else scatter(o);
    }
}

```



-- OUTPUT --

```

6
1
2
4
result = 3.25

```

6) ScatterGather.java

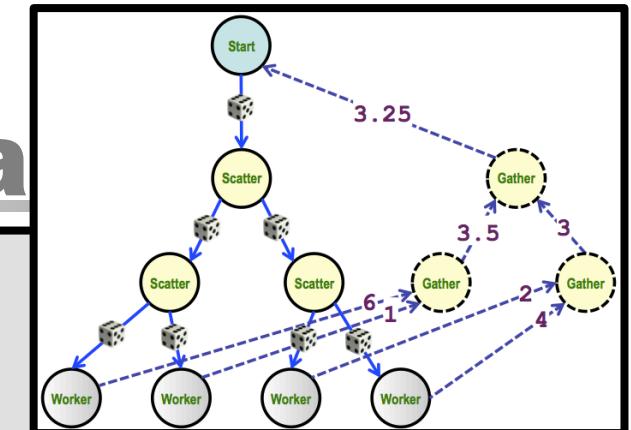
```

class GatherActor extends UntypedActor {
    double res1;
    ActorRef caller;

    private double average(double x, double y) {
        return (x + y) / 2;
    }

    public void onReceive(Object o) throws Exception {
        if (o instanceof CallerMessage) {
            caller = ((CallerMessage) o).caller;
        } else if (o instanceof ResultMessage) {
            if (caller == null) throw new Exception("no caller address!!!");
            if (res1 == 0) {
                res1 = ((ResultMessage) o).result;
            } else {
                double res2 = ((ResultMessage) o).result;
                double res = average(res1, res2);
                caller.tell(new ResultMessage(res), ActorRef.noSender());
                getContext().stop(getSelf()); // die!
            }
        }
    }
}

```



-- OUTPUT --

```

6
1
2
4
result = 3.25

```

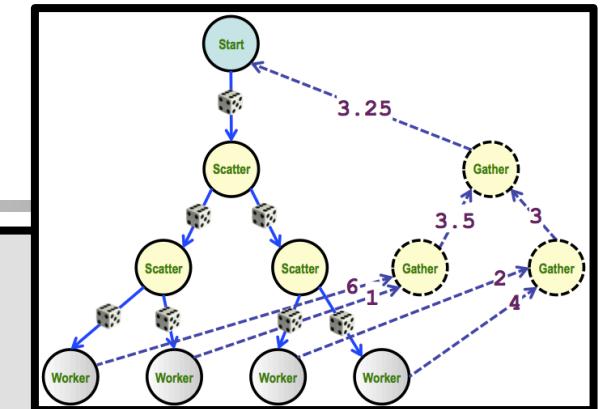
6) ScatterGather.java

```

class StartActor extends UntypedActor {
    public void onReceive(Object o) throws Exception {
        if (o instanceof StartMessage) {
            ActorRef worker =
                getContext().actorOf(Props.create(WorkerScatterActor.class), "worker");
            worker.tell(new SplitMessage(), ActorRef.noSender());
            worker.tell(new SplitMessage(), ActorRef.noSender());
            worker.tell(new ComputeMessage(6, getSelf()), ActorRef.noSender());
        } else if (o instanceof ResultMessage) {
            double result = ((ResultMessage) o).result;
            System.out.println("result = " + result);
        }
    }
}

public class ScatterGather {
    public static void main(String[] args) {
        final ActorSystem system = ActorSystem.create("HelloWorldSystem");
        final ActorRef starter =
            system.actorOf(Props.create(StartActor.class), "starter");
        starter.tell(new StartMessage(), ActorRef.noSender());
        try { System.out.println("Press return to terminate..."); System.in.read();
        } catch(IOException e) { e.printStackTrace();
        } finally { system.shutdown();
        }
    }
}

```



-- OUTPUT --

```

6
1
2
4
result = 3.25

```

Scatter-Gather + ...

■ Adaptive Load balancing:

- Monitor system to extract up-to-date statistics
- Based on statistics, adjust system capacity (cf. our split) or Quality-of-Service (ak²a, "graceful degradation")
 - Note: this may be done on **all** nodes in the hierarchy!

■ Memoization/Caching:

- Often, memoization is used to "cache" already-performed-computations // `Map<Key,Val> cache;`
 - Note: this may be done on **all** nodes in the hierarchy!

■ Fault Tolerance:

- Supervisors react if workers don't respond or crash
- Then: `resume()`, `subtree.restart()`, `parent.escalate()`

More information...

■ ERLANG:

- [<http://www.erlang.org/download/erlang-book-part1.pdf>]

■ AKKA Video Talks:

- [<https://www.youtube.com/watch?v=GBvtE61Wrto>]
- [<https://www.youtube.com/watch?v=t4KxWDqGfcs>] // Slides
 - http://gotocon.com/dl/goto-aar-2012/slides/JonasBonr_UpUpAndOutScalingSoftwareWithAkka.pdf

■ JAVA+AKKA Documentation:

- [<http://doc.akka.io/docs/akka/snapshot/java/untyped-actors.html>]
- [<http://doc.akka.io/docs/akka/2.3.7/AkkaJava.pdf>]

■ JAVA+AKKA API:

- [<http://doc.akka.io/japi/akka/2.3.7/>]

Thx!



Questions?

Reception (ERLANG vs AKKA)

- In ERLANG:
 - Locally nested receives (depending on local state)
- In JAVA+AKKA:
 - You only have one top-level receive:
- Example ⇒ refactored (ready) for JAVA+AKKA:

```
%% -- GATHER -----
gather(Pid) ->
  receive // State #0 ('Res1' not set)
    {result,Res1} ->
      receive // State #1 ('Res1' set)
        {result,Res2} ->
          Res = average(Res1,Res2),
          Pid ! {result, Res} % die.
      end
  end.
```

```
%% -- GATHER' -----
gather(Pid, Res1) ->
  receive
    {result,Res1} when Res1 = undef ->
      gather(Pid, Res1)
    ;
    {result,Res2} ->
      Res = average(Res1, Res2),
      Pid ! {result, Res} % die.
  end.
```

[See also ERLANG Book, program 5.3]