Practical Concurrent and Parallel Programming 6

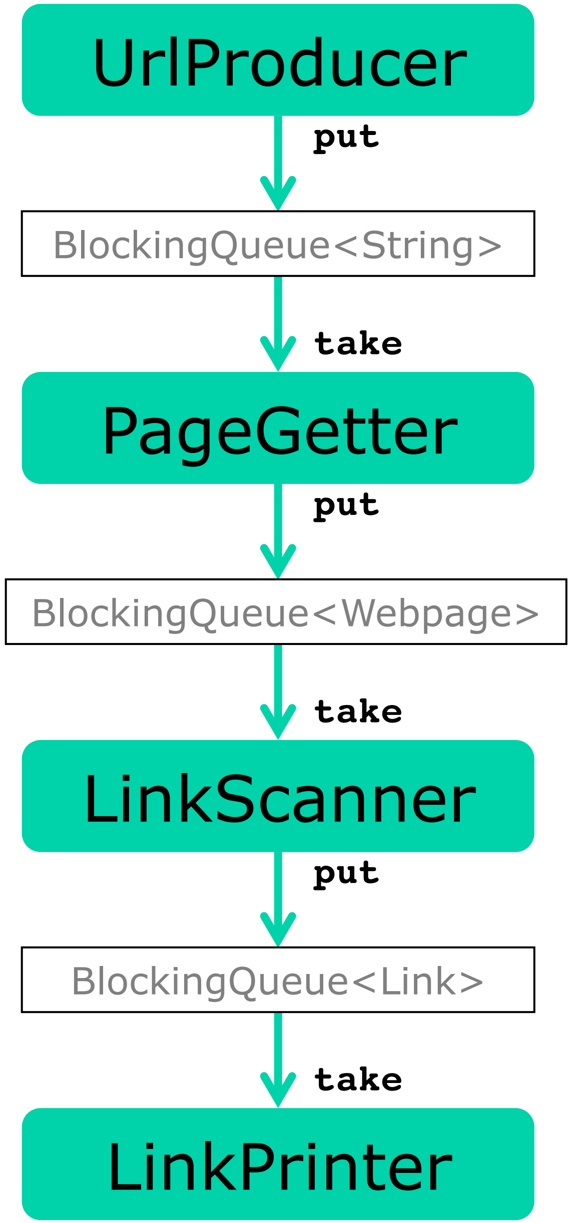
Peter Sestoft
IT University of Copenhagen

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

TestPipeline.java

```
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();
```

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.

Java 8 class library documentation

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	4885
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

Acc A

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

Acc A

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

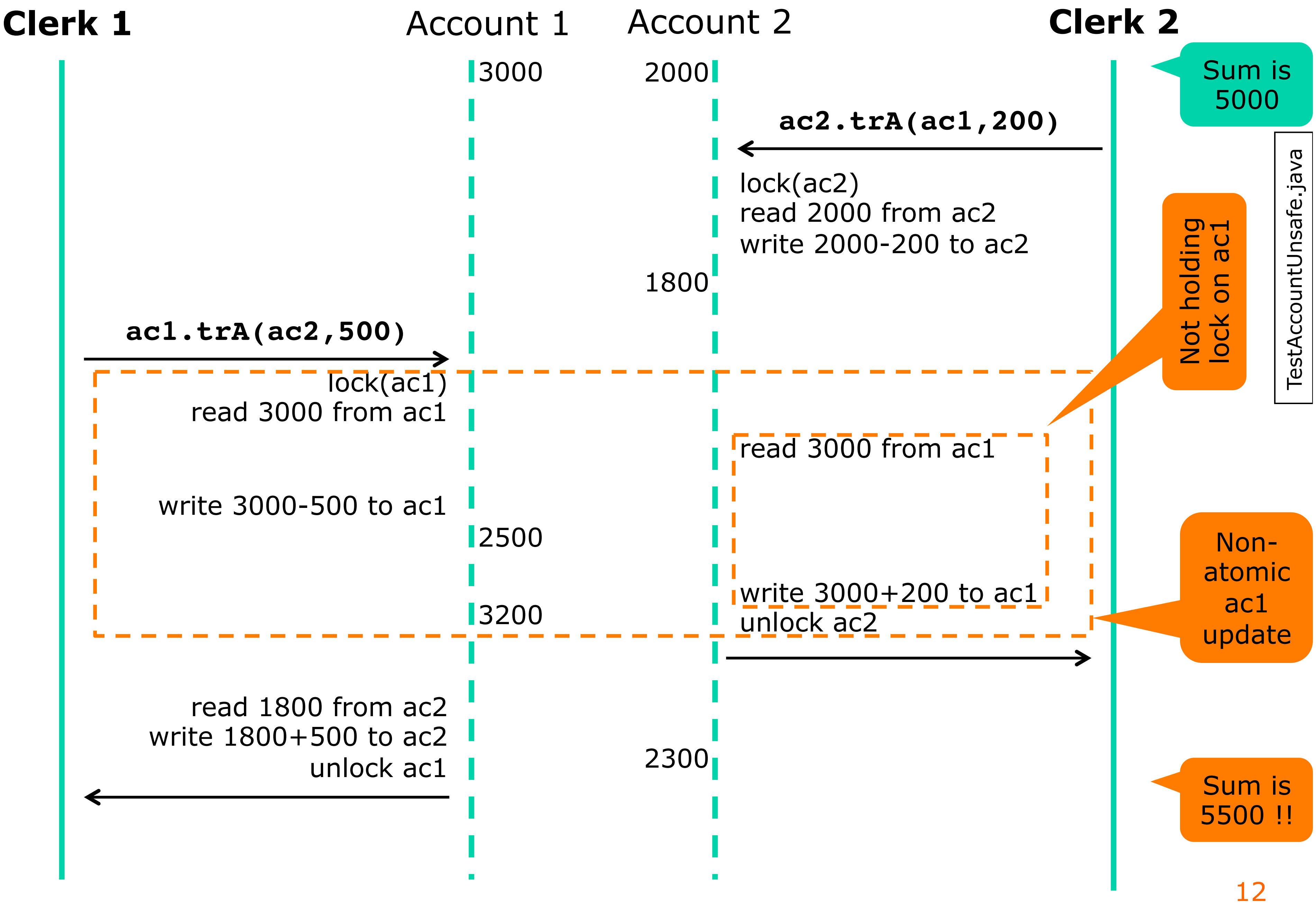
- Main thread occasionally prints balance sum:

```
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



Acc B

TestAccounts version B

- TransferA was bad: Only one thread locks ac1
  - This does not achieve atomic update
- Attempt at atomic update of each account:

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

- But a *transfer* is still not atomic
  - so wrong, non-5000, account sums are observed:

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

TestAccountUnsafe.java

Acc C

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;
  } }
}
```

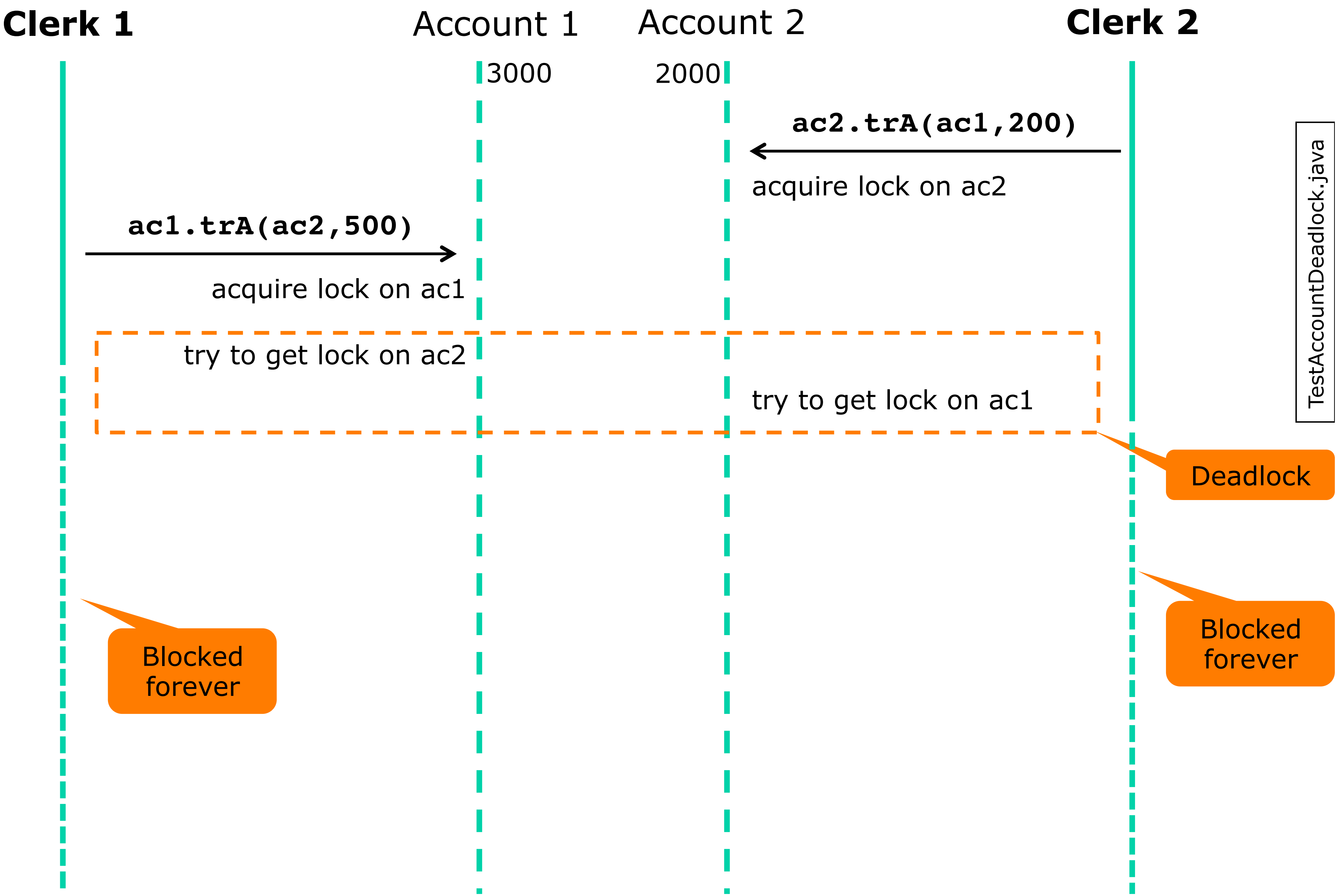
Bad

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



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;
    } }
}
```

Almost unbad

TestAccountLockOrder.java

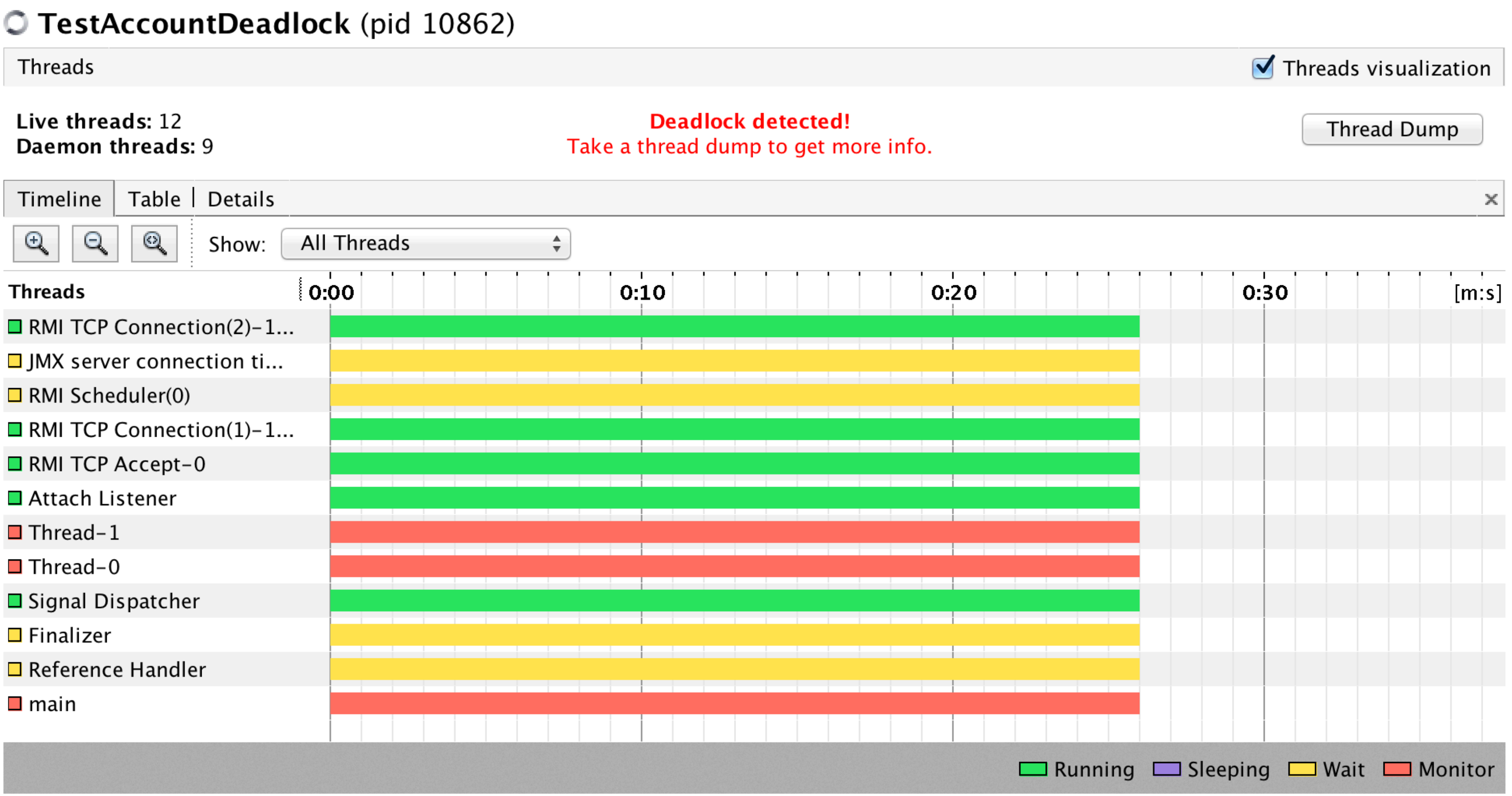
- Small risk of equal hashcodes and so deadlock
- See Goetz 10.1.2 + exercise how to eliminate

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



Thread dump points to deadlock scenario

```
Found one Java-level deadlock:
==============================
"Thread-1":
  waiting to lock monitor 0x00007fc43a010b48 (object 0x0000000740088b40, a Account),
  which is held by "Thread-0"
"Thread-0":
  waiting to lock monitor 0x00007fc43a010d58 (object 0x0000000740088b28, 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 <0x0000000740088b40> (a Account)
        - locked <0x0000000740088b28> (a Account)
        at TestAccountDeadlock$2.run(TestAccountDeadlock.java:29)
        at java.lang.Thread.run(Thread.java:745)
"Thread-0":
        at Account.transferC(TestAccountDeadlock.java:61)
        - waiting to lock <0x0000000740088b28> (a Account)
        - locked <0x0000000740088b40> (a Account)
        at TestAccountDeadlock$1.run(TestAccountDeadlock.java:23)
        at java.lang.Thread.run(Thread.java:745)
```

transferC method is involved

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
  - Or on a WorkStealingPool when only 2 cores
- 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

Taxi A

Deadlocks may be hard to spot

Bad

Goetz p. 212

```
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;
  }
}
```

Lock taxi

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

Deadlock risk!

Lock dispatcher

Call **getLocation**, locks taxi

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 try-able) locks

Acc G

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

```
class Account {
  private final Lock lock = new ReentrantLock();

  public void deposit(long amount) {
    lock.lock();
    try {
      balance += amount;
    } finally {
      lock.unlock();
    }
  }

  public long get() {
    lock.lock();
    try {
      return balance;
    } finally {
      lock.unlock();
    }
  }
}
```

Acquire lock

Always release it

Acquire lock

Always release it

TestAccountTryLock.java

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

Acc G

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 retry

In any case, release acquired locks

Sleep 0-500 ns before retry to save CPU time

Like Goetz p. 280

TestAccountTryLock.java

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

Bloch p. 276

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

Lecture 5
blocking queue

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 (may be 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

file:///Users/sestoft/java/pcpp/ts/guardedby/threadsafe-html/index.html#by-ty...

Google Maps | Wikipedia | Java 8 API | Multiverse API | PCPP | LearnIT PCPP

Summary **Findings** Packages

Group by: Type

GuardedBy annotation violated (1)

- @GuardedBy annotation on field 'count' violated

TestGuardedBy.java

- 45 Problem location
- 48 Synchronized read
- 48 Synchronized write
- 51 Unsynchronized read
- 51 Unsynchronized write
- 54 Synchronized read

Accesses

```
14  // (see lecture 6) in ~/lib/ts/ and run it AFTER compiling as abov
15  //    java -jar ~/lib/ts/threadsafe.jar
16  // Then read ThreadSafe's report in a browser:
17  //    open threadsafe-html/index.html
18
19  // Or do the whole thing in Eclipse, where it works more smoothly.
20
21  // From JSR 305 jar file jsr305-3.0.0.jar:
22  import javax.annotation.concurrent.GuardedBy;
23
24  import java.io.IOException;
25
26  public class TestGuardedBy {
27    public static void main(String[] args) throws IOException {
28      final LongCounter lc = new LongCounter();
29      Thread t = new Thread(new Runnable() {
30      public void run() {
31        while (true)           // Forever call increment
32          lc.increment();
33      }
34        });
35      t.start();
36      System.out.println("Press Enter to get the current value:");
37      while (true) {
38        System.in.read();          // Wait for enter key
39        System.out.println(lc.get());
40      }
41    }
42  }
43
44  class LongCounter {
45    @GuardedBy("this")
46    private long count = 0;
47    public synchronized void increment() {
48      count++;
49    }
50    public void decrement() {
51      count++;
52    }
53    public synchronized long get() {
54      return count;
55    }
56  }
```

Violation

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 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