Practical Concurrent and Parallel Programming 9

Peter Sestoft
IT University of Copenhagen

Friday 2015-10-30

Plan for today

- **Locking on multiple objects**
- **Deadlock and locking order**
- **Tool: jvisualvm, a JVM runtime visualizer**
- Explicit locks, **`lock.tryLock()`**
- Liveness
- Concurrent correctness: safety + liveness
- The Java memory model

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

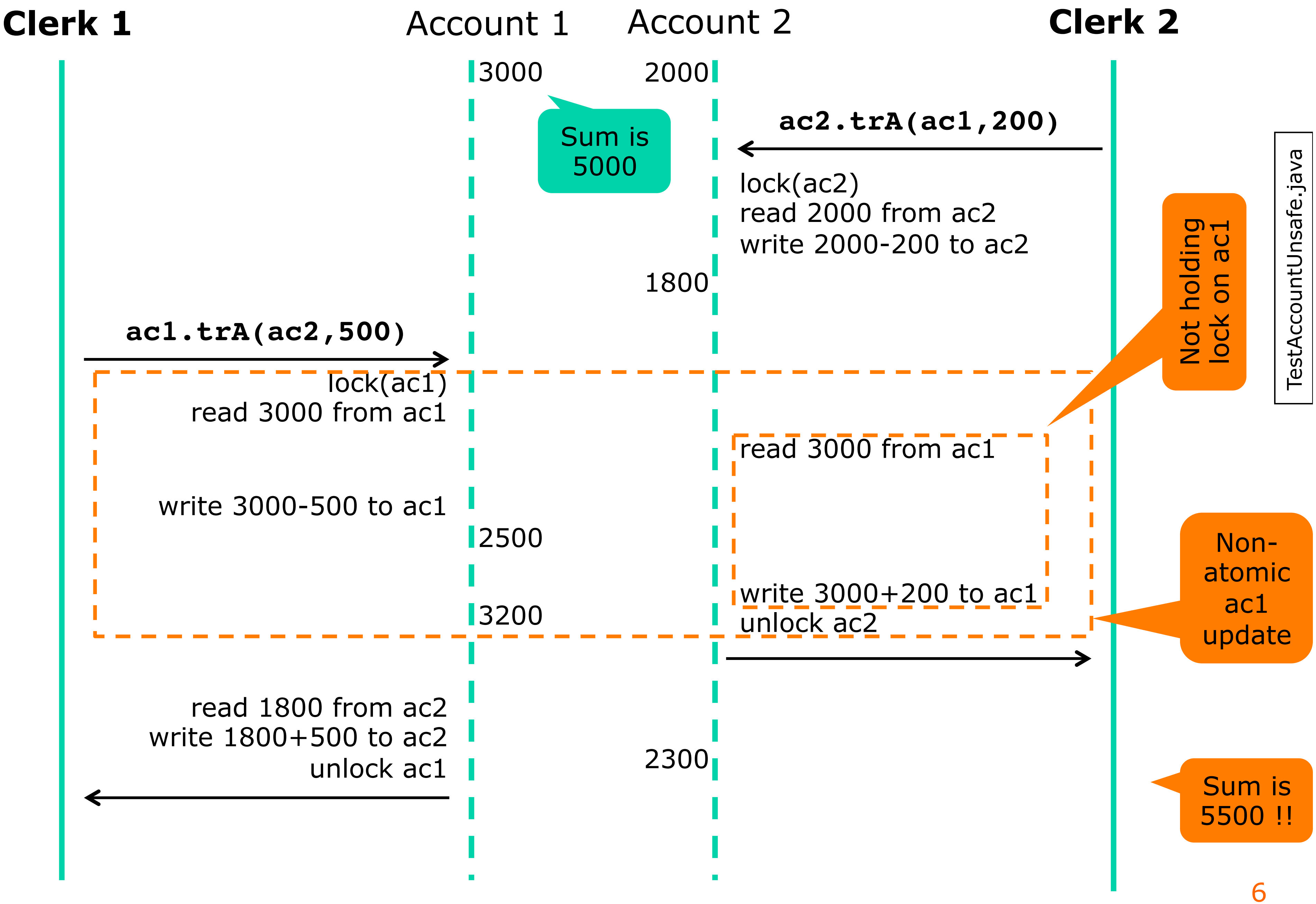
- Main thread occasionally prints balance sum:

```
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?

Acc A

Losing updates with transferA



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);
}
```

TestAccountUnsafe.java

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

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

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;
  } }
}
```

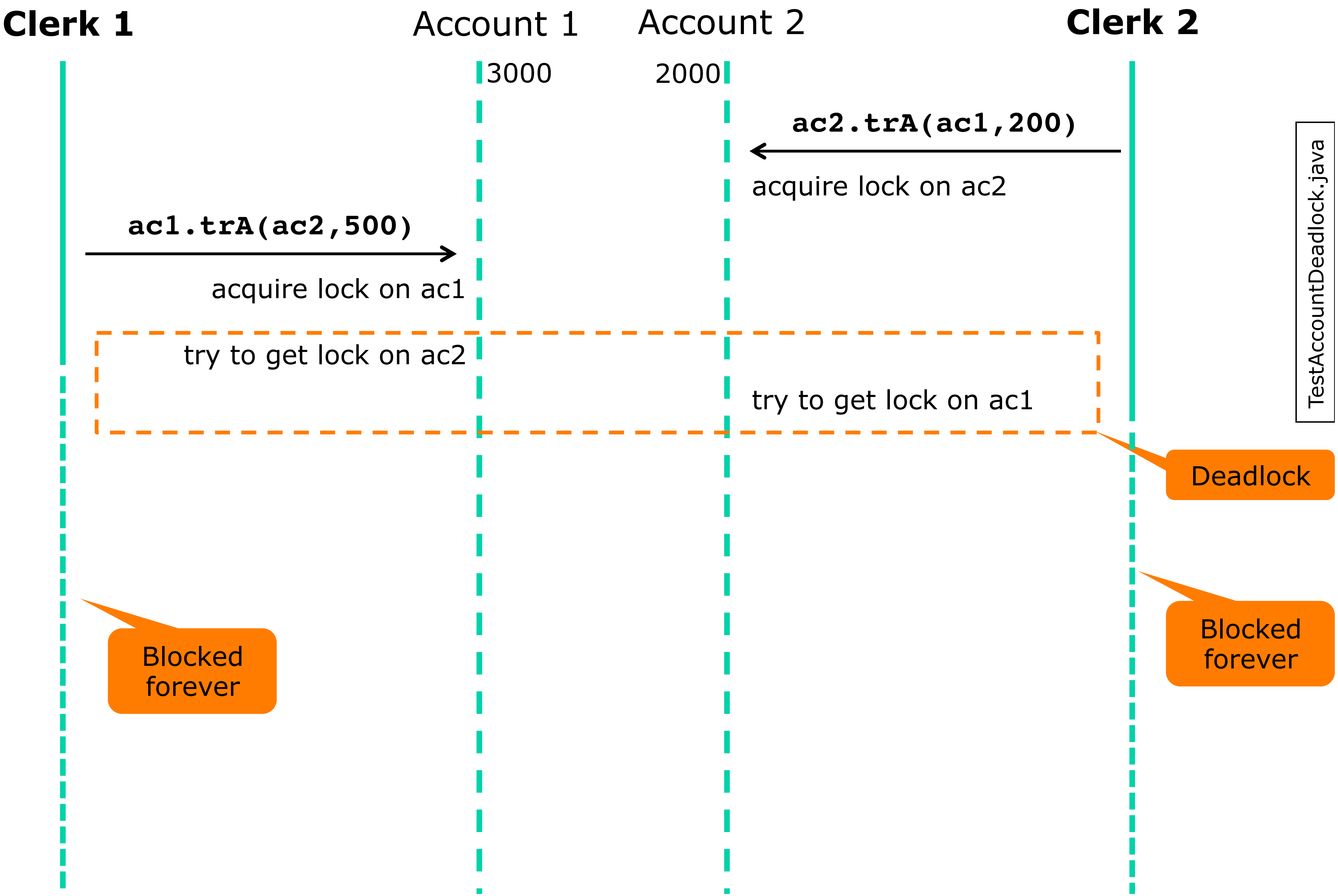
Bad

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

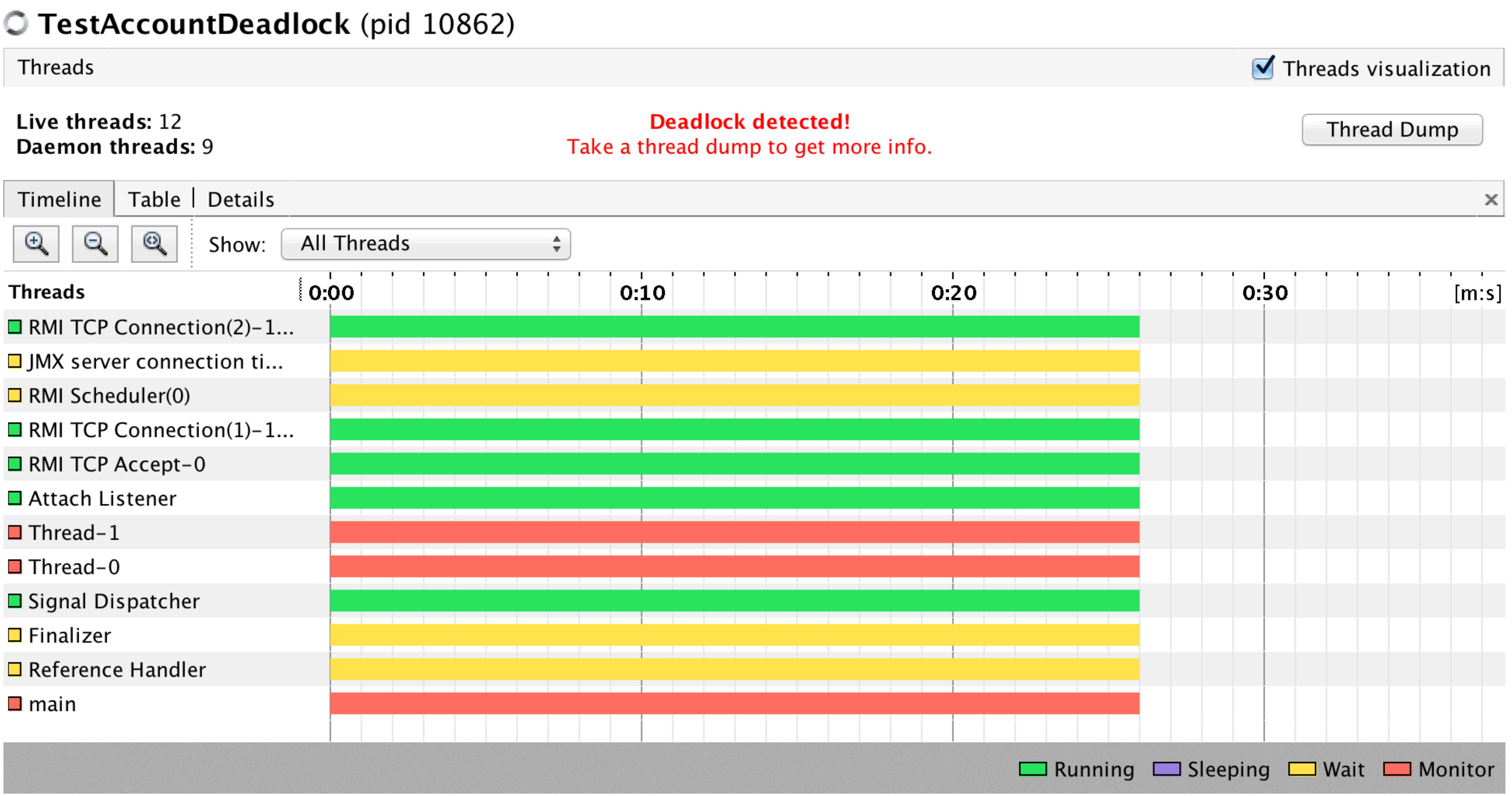
- Small risk of equal hashcodes and so deadlock
- See Goetz 10.1.2 + exercise how to eliminate

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 the 4-stage pipeline from week 5 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
  - Maybe the reason C# has **`lock`** only on statements, 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

Bad

```
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

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, move, test, release

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, next week

Plan for today

- Locking on multiple objects
- Deadlock and locking order
- Tool: jvisualvm, a JVM runtime visualizer
- **Explicit locks, `lock.tryLock()`**
- **Liveness**
- Concurrent correctness: safety + liveness
- The Java memory model

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, next week

Taking two locks, using tryLock()

Acc G

```
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

In any case, release acquired locks first

Else retry

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 random sleep periods to be same always
  - requires the operation interleaving to be the same

Correctness = Safety + Liveness

Goetz p. 8

- Safety: nothing bad ever happens
  - Invariants are preserved, no updates lost, etc
- Liveness: something good eventually 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

Thread scheduler, priorities, ...

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

Bloch p. 286

Item 72: Don't depend on the thread scheduler

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

Plan for today

- Locking on multiple objects
- Deadlock and locking order
- Tool: jvisualvm, a JVM runtime visualizer
- Explicit locks, `lock.tryLock()`
- Liveness
- Concurrent correctness: safety + liveness
- **The Java memory model**

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

Surprising results

```
class StoreBufferExample {
  volatile boolean A = false,
                   B = false;
  int A_Won = 0, B_Won = 0;
  public void ThreadA() {
    A = true;
    if (!B)
      A_Won = 1;
  }
  public void ThreadB() {
    B = true;
    if (!A)
      B_Won = 1;
  }
}
```

Executed on one thread

Executed on another thread

TestStoreBuffer.java

Ostrovsky 2013

- Without **`volatile`**, can get **`A_won = B_won = 1`**
  - Not JIT compiler, but CPU store buffer delay on A
  - Memory updates are *not sequentially consistent*
- With **`volatile`**, this is impossible (in Java)

Interleavings assuming sequentially consistent memory model

Initially: **`A = B = false`** and **`A_Won = B_Won = 0`**

A=true
if (!B)
A_Won=1
B=true
if (!A)
	A=true
	if (!B)
	B=true
	A_Won=1
	if (!A)
		A=true
		if (!B)
		B=true
		if (!A)
		A_Won=1

A won

A=true
B=true
if (!B)
if (!A)
	A=true
	B=true
	if (!A)
	if (!B)

Nobody won

B=true
if (!A)
B_Won=1
A=true
if (!B)
	B=true
	if (!A)
	A=true
	B_Won=1
	if (!B)
		B=true
		if (!A)
		A=true
		if (!B)
		B_Won=1

B won

B=true
A=true
if (!A)
if (!B)
	B=true
	A=true
	if (!B)
	if (!A)

Nobody won

Experiments on 4-core Intel i7

- Java, without volatile and with volatile:

	A loses	A wins
B loses	0	436649
B wins	550463	12888

	A loses	A wins
B loses	2668	438518
B wins	558814	0

TestStoreBuffer.java

```
if (!B)
B=true
if (!A)
B_Won=1
A=true
A_Won=1
```

Some weird executions??

Not sequentially consistent: seen from thread A, the if (!B) moved before A=true

- On 1-core ARM, double-wins seem impossible

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 T1 *synchronizes-with* any action in another thread T2 that detects that T1 has terminated
  - If thread T1 interrupts thread T2, the interrupt by T1 *synchronizes-with* any point where any other thread (including T2) determines that T2 has been interrupted

- Action x *happens-before* action y, written hb(x,y), is defined like this:
  - 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

Java Language Specification §17.4

Goetz §16.3.1

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

JLS 8 Tables 17.1, 17.5

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

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
  - But makes code slower, see lecture 4: VolatileArray.java

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 [illegible]oo much fuss over this - it's just fundamentally tricky!

Carol

Works in Java, dubious in C#

C#/.NET volatile weaker than Java's

```
class StoreBufferExample {
  volatile bool A = false,
                B = false;
  int A_Won = 0, B_Won = 0;
  public void ThreadA() {
    A = true;
    if (!B)
      A_Won = 1;
  }
  public void ThreadB() {
    B = true;
    if (!A)
      B_Won = 1;
  }
}
```

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

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

TestStoreBuffer.cs

Ostrovsky 2013

- C#: possible to get **`A_Won = B_Won = 1 !!!`**
  - Even with **`volatile`**
  - To fix in C#, add MemoryBarrier call

Experiments on 4-core Intel i7

- C#/.NET 4.6, without and with volatile:

	A loses	A wins
B loses	600	874916
B wins	108249	16235

	A loses	A wins
B loses	522	912084
B wins	72290	15102

TestStoreBuffer.cs

C# volatile
has no effect!!

- Volatile in C# **not** the same as in Java
- Volatile keyword in C, C++, Java and C# has four different meanings...

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.

C# Language Spec 2006, §17.4.3

- 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.

Lochbihler: Making the Java memory model safe, ACM TOPLAS, December 2013

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 sometimes is needed anyway
  - also on x86, contradicting the API docs ...
- Java does not have MemoryBarrier, because Java **`volatile`** gives good guarantees

This week

- Reading
  - Goetz et al chapter 10 + 13.1 + 16
  - Java Language Specification §17.4
  - Bloch item 67
- Exercises week 9
  - Show that you can write non-deadlocking code, and that you can use tools such as jvisualvm
  - Show that you can use locks correctly
- 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
  - Scanned PDF on LearnIT
- 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