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

```java
//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

```java
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));
```
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));

• 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:

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

• Sequential Java 8 stream:

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

• Parallel Java 8 for loop

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

Classical efficient imperative loop

Pure functional programming ...

... and thus parallelizable and thread-safe
Performance results (!!)

• Counting the primes in 0 ... 99,999

<table>
<thead>
<tr>
<th>Method</th>
<th>Intel i7 (us)</th>
<th>AMD Opteron (us)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential for-loop</td>
<td>9962</td>
<td>40548</td>
</tr>
<tr>
<td>Sequential stream</td>
<td>9933</td>
<td>40772</td>
</tr>
<tr>
<td>Parallel stream</td>
<td>2752</td>
<td>1673</td>
</tr>
<tr>
<td>Best thread-parallel</td>
<td>2969</td>
<td>4885</td>
</tr>
<tr>
<td>Best task-parallel</td>
<td>2631</td>
<td>1874</td>
</tr>
</tbody>
</table>

• 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:

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

• Naively add method for transfers:

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

Bad
Two clerks working concurrently

```java
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();
```

- Main thread occasionally prints balance sum:
  ```java
  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

Clerk 1

<table>
<thead>
<tr>
<th>Account 1</th>
<th>Account 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>3000</td>
<td>2000</td>
</tr>
</tbody>
</table>

*ac1.trA(ac2, 500)*

- lock(ac1)
- read 3000 from ac1
- write 3000-500 to ac1
- read 1800 from ac2
- write 1800+500 to ac2
- unlock ac1

Clerk 2

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ac2.trA(ac1, 200)</td>
<td></td>
</tr>
<tr>
<td>lock(ac2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>read 2000 from ac2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>write 2000-200 to ac2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Sum is 5000

Non-atomic ac1 update

Not holding lock on ac1

TestAccountUnsafe.java

Sum is 5500 !!
TestAccounts version B

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

```java
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
```
Must lock both accounts

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

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

- 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

Clerk 1

Account 1

3000

ac1.trA(ac2,500)

acquire lock on ac1

try to get lock on ac2

Blocked forever

Clerk 2

Account 2

2000

ac2.trA(ac1,200)

acquire lock on ac2

try to get lock on ac1

try to get lock on ac2

Deadlock

Blocked forever
Avoiding deadlock, serial no.

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

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

– Take locks in serial number order:

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

Atomic and deadlock free
Avoiding deadlock, lock order

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

```java
class TestAccountLockOrder {  
  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;  
      }  
    }  
  }  
}  

class Account {  
  int serial;  
  long balance;  
}  
```  

• Cumbersome, we may encapsulate lock-taking

```java
class TestAccountLockOrder {  
  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

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

```java
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;
        } }
}
```

• 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

Deadlock detected!
Take a thread dump to get more info.
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 <0x0000000740088b40> (a Account)
  - locked <0x0000000740088b28> (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
  - 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
Deadlocks may be hard to spot

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;
    }
}
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;
    }
}
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

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

```java
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();
        }
    }
}
```
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
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();
                }
            } finally {
                ac2.lock.unlock();
            }
        }
        try { Thread.sleep(0, (int)(500 * Math.random())); } 
        catch (InterruptedException exn) { } 
    }
}
Livelo\textsc{k}: nobody makes progress

- The \texttt{transferG} method never deadlocks
- In principle it can \textit{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

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

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

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

• Compile like this:

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

• 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=.
binary=.
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
```
Add method, forget synchronized

Violation
Analysing unsafe account transfer

• Problem found, but message is subtle:

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

<table>
<thead>
<tr>
<th>Accesses</th>
<th>Guards for access to field</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Account.balance:</td>
</tr>
<tr>
<td></td>
<td>Account.this &lt;unknown&gt;</td>
</tr>
<tr>
<td></td>
<td>UnsafeAccount.java: 18</td>
</tr>
<tr>
<td></td>
<td>UnsafeAccount.java: 18</td>
</tr>
<tr>
<td></td>
<td>UnsafeAccount.java: 22</td>
</tr>
<tr>
<td></td>
<td>UnsafeAccount.java: 26</td>
</tr>
<tr>
<td></td>
<td>UnsafeAccount.java: 27</td>
</tr>
<tr>
<td></td>
<td>UnsafeAccount.java: 27</td>
</tr>
</tbody>
</table>
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:

```java
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(); } }
}
```
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**

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