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

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

NB!

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

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

• 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

```java
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;
}
```

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

```java
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();
```

- 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

```java
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();
```

- 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

Thread “main” (active)

Object lc (passive)

Thread t1 (active)

get()

increment()

increment()

increment()

increment()

get()

get()

get()

increment()
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;
    }
}

• 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

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

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

WARNING: Useless

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

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

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

Loop while zero
Visibility by synchronization

Thread A

\[ y = 1 \]
\[ \text{lock } M \]
\[ x = 1 \]
\[ \text{unlock } M \]

\text{unlock} = \text{release}

\text{exit synchronized}

Everything before the unlock on M...

Thread B

\[ \text{lock } M \]
\[ i = x \]
\[ \text{unlock } M \]
\[ j = y \]

\text{enter synchronized}

\text{lock} = \text{acquire}

... is visible to everything after the lock on M
Communication through mutable shared state fails if no visibility

Thread “main” (active)  Object mi (passive)  Thread t (active)

set(42)  

get()  

0  

get()  

0  

get()  

0  

get()  

0  

get()  

0  

get()  

0

BAD: does not see the write
The volatile field modifier

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

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

- 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.

- 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

• Because a webserver is naturally concurrent
  – So servlets should be thread-safe

• We use similar, simpler examples:

```java
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

```java
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

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

• Call the server from multiple threads:

```java
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; }
}

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

Like Goetz p. 18
Bad attempt to count calls

```java
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; }
}
```

• 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(); }
}

• 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

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

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

Like Goetz p. 26

Without synchronized the two fields could be written by different threads.
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

Thread t1 (active)

Object lc (passive)

Thread t2 (active)

increment() not atomic

increment()
Atomic update without excess locking

```java
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;
    }
}
```

- Correct but subtle

Like Goetz p. 31
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.

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

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

- Immutable, so automatically thread-safe

Like Goetz p. 49

**Q:** Why?

Nothing can change between test and return

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

- 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

```java
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;
    }
}
```

Like Goetz p. 50

NB!
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 acceptable for immutable classes.

3. **Make classes immutable unless there’s a very good reason to make them mutable.** Immutable classes provide many advantages, and their only disadvantage is the need to be immutable in 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-
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?

- 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 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