Practical Concurrent and Parallel Programming

Kasper Østerbye

IT University of Copenhagen

Friday 2020-09-04

Plan for today

- A few notes on yesterdays feedback
- CPU caches and parallelism
- Visibility (of memory writes)
- Instance creation and synchronization

First feedback session

Reentrant

```
class Reentrant <T> {
  private List<T> myElements;

  public synchronized void add(T element){
    myElements.add( element ); }
  public synchronized void addAll(Collection<> col){
    for (T elem: col) {
      add( elem ); }
  }
}
```

Which of the following descriptions of `synchronized` is the correct?

- When a thread try to enter a synchronized method, it will try to obtain the lock. If the lock is already taken, it will wait.
- When a thread try to enter a synchronized method, it will try to obtain the lock. If the lock is already taken by an other thread, it will wait

Reentrant

```
class Reentrant <T> {
  private List<T> myElements;

  public synchronized void add(T element){
    myElements.add( element ); }
  public synchronized void addAll(Collection<> col){
    for (T elem: col) {
      add( elem ); }
  }
}
```

Which of the following descriptions of `synchronized` is the correct?

- When a thread try to enter a synchronized method, it will try to obtain the lock. If the lock is already taken, it will wait.
- When a thread try to enter a synchronized method, it will try to obtain the lock. If the lock is already taken by an other thread, it will wait

Reentrancy is implemented by associating with each lock an acquisition count and an owning thread.

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;
}} // TestLongCounter.java
```

Why needed? 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 { WARNING: Useless
  private int value = 0;
  public void set(int value) { this.value = value; }
  public int get() { return value; }
}
```

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

```
final MutableInteger mi = new MutableInteger();
Thread t = new Thread(() -> {
    while (mi.get() == 0) { } //Loop while zero
});
t.start();
mi.set(42);
```

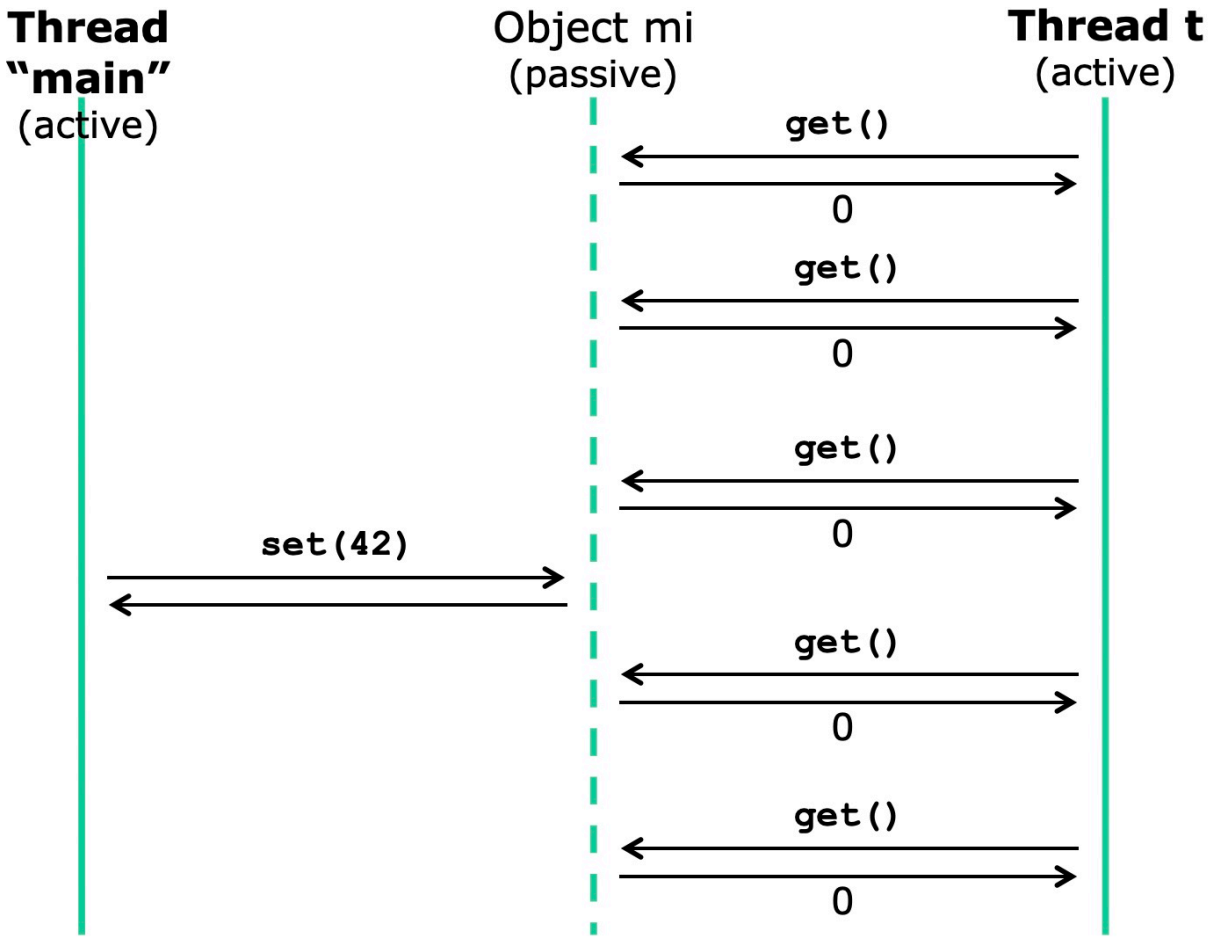
This write by thread ”main” may be forever invisible to thread t

Why can the other thread not see the write of 42?

It is because of the caching done inside a modern CPU.

Lets just spend 5 min on this video.

42 is just written to main's cache



Reordering of instructions

```
public class PossibleReordering {
    static int x = 0, y = 0;
    static int a = 0, b = 0;
    public static void main(String[] args) throws InterruptedException {
        Thread one = new Thread(() -> {
            a = 1; x = b; });
        Thread other = new Thread(() {
            b = 1; y = a; });
        one.start(); other.start();
        one.join(); other.join();
        System.out.println("( " + x + "," + y + ")");
    }
}
```

We can get (0,1), (1,0), and (1,1)

Reordering of instructions

```
public class PossibleReordering {
    static int x = 0, y = 0;
    static int a = 0, b = 0;
    public static void main(String[] args) throws InterruptedException {
        Thread one = new Thread(() -> {
            a = 1; x = b; });
        Thread other = new Thread(() {
            b = 1; y = a; });
        one.start(); other.start();
        one.join(); other.join();
        System.out.println("( " + x + "," + y + ")");
    }
}
```

We can get (0,1), (1,0), and (1,1)

But also (0,0)

Reordering of instructions

```
public class PossibleReordering {
    static int x = 0, y = 0;
    static int a = 0, b = 0;
    public static void main(String[] args) throws InterruptedException {
        Thread one = new Thread(() -> {
            a = 1; x = b; });
        Thread other = new Thread(() {
            b = 1; y = a; });
        one.start(); other.start();
        one.join(); other.join();
        System.out.println("( " + x + "," + y + ")");
    }
}
```

We can get (0,1), (1,0), and (1,1)

But also (0,0)

- The Java compiler can *reorder* instructions with no data-dependencies
- Memory caching can make assignments to field-variables appear out of order.

Aside - why would java reorder statements???

```
a = ...
b = ...
//stuff that change a
//stuff that change b
a++;
b++;
```

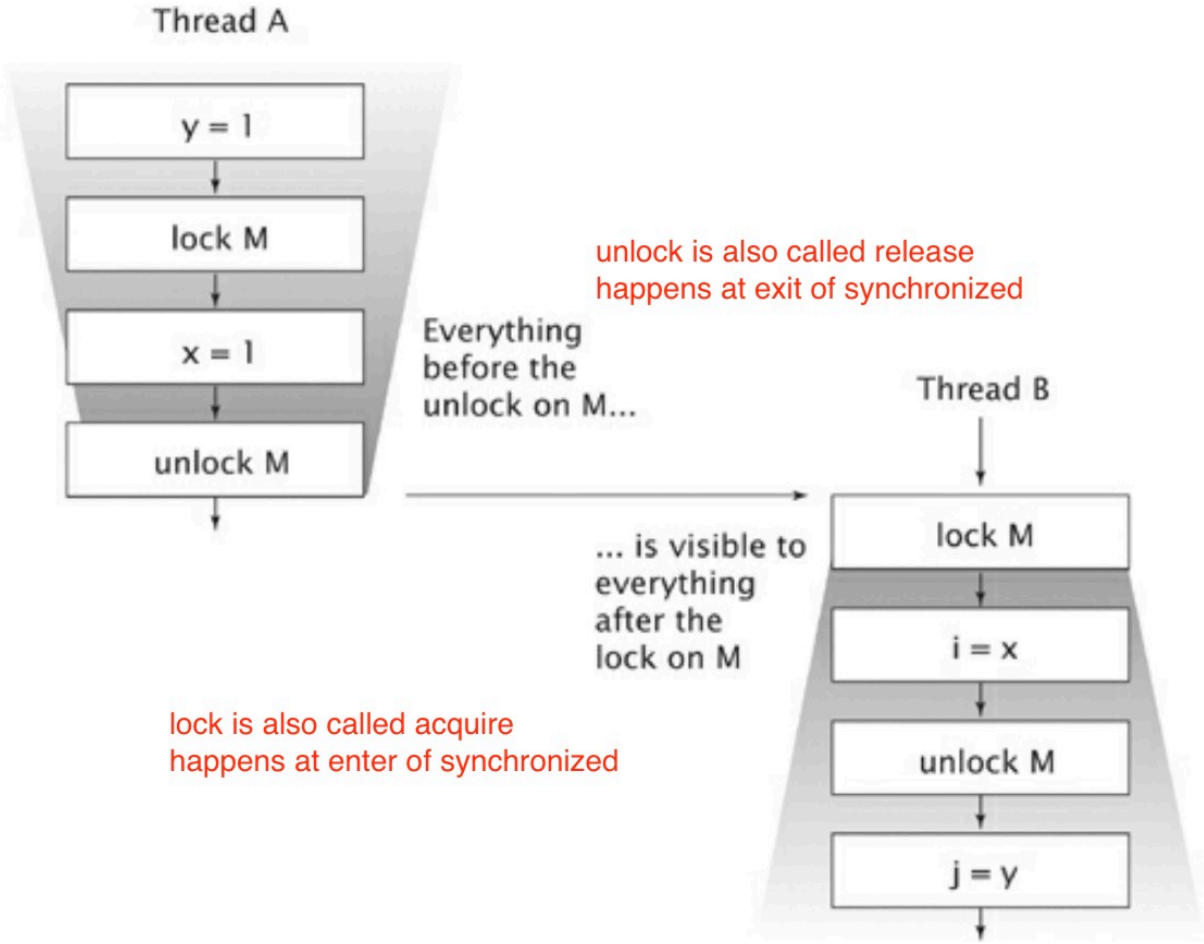
Sometimes it can yield better performance to let the JVM JIT compiler allow to change this to:

```
a = ...
//stuff that change a
a++;
b = ...
//stuff that change b
b++;
```

Because this can allow variable `a` to remain in one of the cpu registers, and not be written to memory until the end

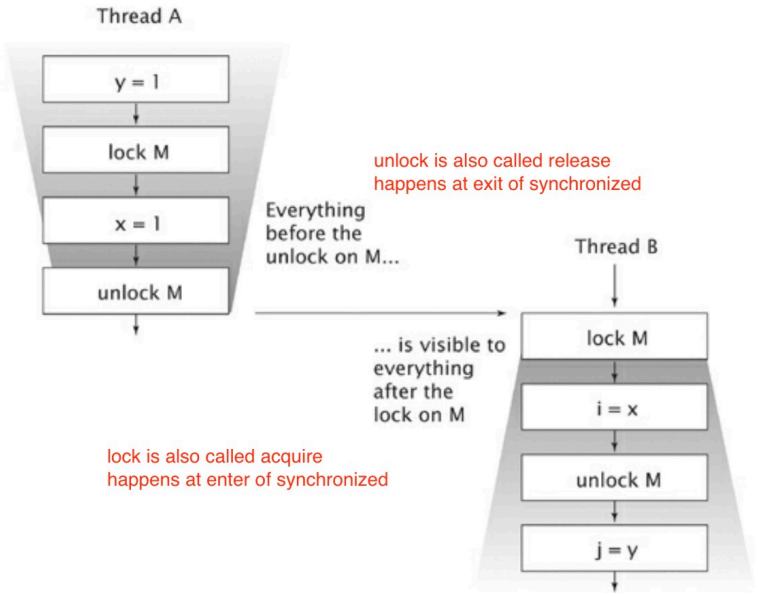
Visibility by synchronization

One solution is to use `synchronized`



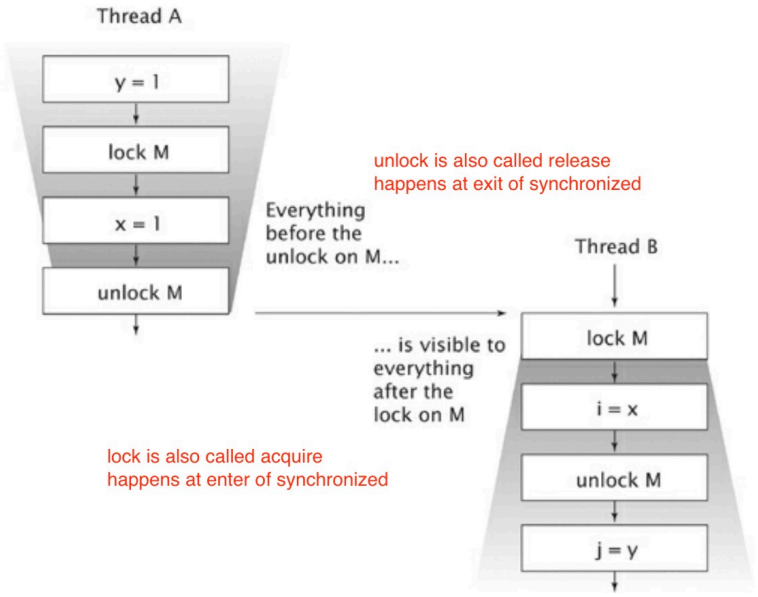
MutableInteger - synchronization

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



MutableInteger - synchronization

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



Why is the synchronized keyword needed for `get()`?

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; }
}
```

- Volatile variable rule: A write to a volatile field *happens-before* every subsequent read of that same field.

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 locks (synchronized)
- Rule 2: If circumstances are right, and you are an expert, maybe use volatile instead
- Rule 3: There are few experts

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 collection – 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)

Break

Publishing an object

- What is the problem of making an object, and telling other threads about the object
- Some examples that do not work
- Text book has examples of how it can go wrong

Unsafe publication

Listing 16.3. Unsafe Lazy Initialization. ***Don't do this.***

```
@NotThreadSafe
public class UnsafeLazyInitialization {
    private static Resource resource;

    public static Resource getInstance() {
        if (resource == null)
            resource = new Resource(); // unsafe publication
        return resource;
    }
}
```

Problems:

- two threads might get into a *race-condition* from the test to the assignment
- Due to caching, you can a Resource which is not initialized (because the fields were not yet written)

Unsafe publication

Listing 16.3. Unsafe Lazy Initialization. ***Don't do this.***

```
@NotThreadSafe
public class UnsafeLazyInitialization {
    private static Resource resource;

    public static Resource getInstance() {
        if (resource == null)
            resource = new Resource(); // unsafe publication
        return resource;
    }
}
```

Problems:

- two threads might get into a *race-condition* from the test to the assignment
- Due to caching, you can a Resource which is not initialized (because the fields were not yet written)

Would it help to make `resource` volatile?

Safe publication

Initialization must *happen-before* publication

If you just want to make it work, use 'synchronized' for `getInstance()`.

Safe publication

Initialization must *happen-before* publication

If you just want to make it work, use 'synchronized' for `getInstance()`.

Other ways to publish thread-safely:

- Initializing an object reference from a static initializer;
- Storing a reference to it into a volatile field or AtomicReference;
- Storing a reference to it into a final field of a properly constructed object; or
- Storing a reference to it into a field that is properly guarded by a lock.
- Inserting a reference into a `ThreadSafe` datastructure.

Immutable objects

- What are they?
- And how can it go wrong in publishing them too

Immutable objects

- What are they?
- And how can it go wrong in publishing them too

An object is immutable if:

- Its state cannot be modifled after construction;
- All its flelds are final and
- It is properly constructed.

Safe ways to publish objects

We need to assure that initialization *happens-before* publication.

Safe ways to publish objects

We need to assure that initialization *happens-before* publication.

That is, somewhere the code should be `synchronized`.