Java - Lesson 2.1

Multithreading: Synchronization

Author: Kirill Volkov
https://github.com/vulko/JL2.Multithreading
vulkovk@gmail.com

Synchronization

- Threads share memory and can access same objects simultaneously
- This is an efficient way of multitasking, however it may require synchronized access to shared data

- Synchronization itself prevents issues arising while accessing shared data by multiple threads, however it might introduce other issues like:
  - Thread starvation *(when other threads don't let target thread access shared data)*
  - Livelock *(same situation when 2 persons try to let each other pass the door, so they are stuck letting each other pass but none actually going through)*

Thread interference

- Several threads can access and change state of a single object at the same time
- Possible memory coexistency issues

```
class Counter {
    private int c = 0;

    public void increment() {
        c++;
    }

    public void decrement() {
        c--;
    }

    public int value() {
        return c;
    }
}
```

Memory coexistence

Possible scenario:

- Counter value is 0
- Thread 1: increment value
- Thread 2: decrement value
- Thread 1: print Counter.value() to console

Thread 1 is expecting value to be 1, but when printing out to console, value was modified by another thread and equals 0.

Synchronization primitives

- ***synchronized*** modifier makes sure that when a method of an object is invoked, only 1 thread can execute it, other threads that will try to execute a method on the same object, they will be suspended until the thread that locked this object first will release it

```
public class SynchronizedCounter {
    private int c = 0;

    public synchronized void increment() {
        c++;
    }

    public synchronized void decrement() {
        c--;
    }

    public synchronized int value() {
        return c;
    }
}
```

Synchronization primitives

- ***synchronized*** block is a general approach that allows a thread that is executing such block to capture a lock on the given object, so other threads will be suspended if try to capture lock on the same object.
- Lock is released when exiting such a block

```
    public void increment() {
        synchronized (this) {
            c++;
        }
    }
```

=

```
    public synchronized void increment() {
            c++;
    }
```

```
public class SomeClass {

    public void someMethod() {
        synchronized (SomeClass.class) {

           // this code will be synchronized for all objects of this class

        }
    }

}
```

Atomic operations

Atomic operation is an operation that happens all-in-one.

“It either happens completely, or doesn’t happen at all”

- value++ is executed with 2 assembler commands
- ++value is executed with 1 assembler command

Atomic operations never lead to thread interference.

Most of operations are NOT atomic!

Atomic operations

Java, as many other execution environments cache values, for faster access, so there is no need to access RAM every time a value is read.

To avoid such behavior, ***volatile*** modifier is used.

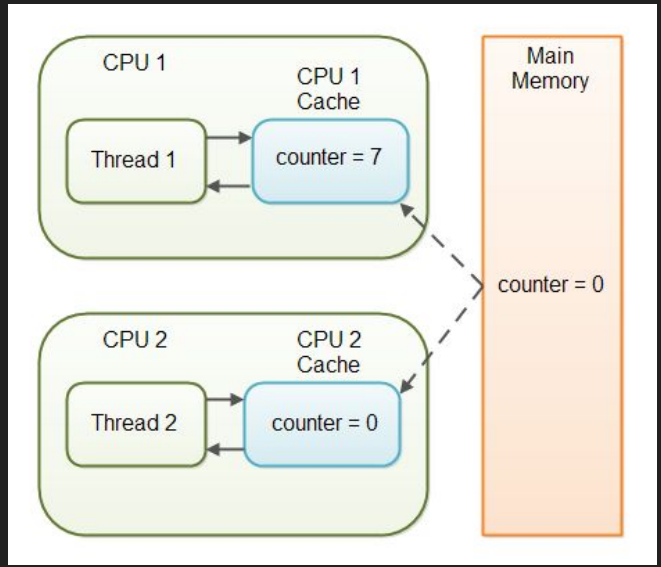
```
public class SynchronizedCounter {

    private volatile int c = 0;

    public synchronized void increment() {
        c++;
    }

    public synchronized void decrement() {
        c--;
    }

    public synchronized int value() {
        return c;
    }
}
```



Synchronization: best practices

```
public class Color {
    private int mARGBValue = 0xffffffff;
    private String mName = ”NONE”;

    public Color(int r, int g, int b, int a, String name) {
        setColor(r, g, b, a);
        mName = name;
    }

    public synchronized void setColor(int red, int green, int blue, int alpha) {
        mARGBValue = ((alpha << 32) (red << 16) | (green << 8) | blue);
    }

    public synchronized int getARGB() {
        return mARGBValue;
    }

    public synchronized String getName() {
        return mName;
    }
}
```

```
        Color color = new Color (255, 0, 0, 0, "Black");
…
        // one thread executes:
        int myColorInt = color.getRGB();      // Statement 1
        String myColorName = color.getName(); // Statement 2
…
        // another thread calls after “Statement 1” but before “Statement 2” was executed
        color.setColor(1, 2, 3, 4);
```

Synchronization: best practices

To avoid such issues, extra synchronization is required

```
        synchronized (color) {
                int myColorInt = color.getRGB();
                String myColorName = color.getName();
        }
…
        synchronized (color) {
                color.setColor(1, 2, 3, 4);
        }
```

Synchronization creates overhead!

Another approach is to use Immutable objects, so the state can’t be changed

- class has to be ***final***
- all fields ***private*** and ***final***, no references to mutable objects
- no setters, only public getters

Synchronization: best practices

```
public final class ImmutableColor {

    private final int mARGBValue;
    private final String mName;

    public Color(int r, int g, int b, int a, String name) {
        mARGBValue = ((alpha << 32) (red << 16) | (green << 8) | blue);
        mName = name;
    }

    public int getARGB() {
        return mARGBValue;
    }

    public String getName() {
        return mName;
    }
}
```

Now any thread can access this object without any potential interference or memory coexistence issues.

No need for synchronization!

High level concurrency

Java provides several packages of high level concurrency:

Executors, thread pools, blocking queues etc

- **java.util.concurrent**

Different types of locks

- **java.util.concurrent.locks**

Atomics for lock-free thread safe concurrency

- **java.util.concurrent.atomic**

High level concurrency: Locks

Condition	Condition factors out the Object monitor methods (wait, notify and notifyAll) into distinct objects to give the effect of having multiple wait-sets per object, by combining them with the use of arbitrary Lock implementations.
Lock	Lock implementations provide more extensive locking operations than can be obtained using synchronized methods and statements.
Semaphore	A counting semaphore.
ReadWriteLock	A ReadWriteLock maintains a pair of associated locks, one for read-only operations and one for writing.
ReentrantLock	A reentrant mutual exclusion Lock with the same basic behavior and semantics as the implicit monitor lock accessed using synchronized methods and statements, but with extended capabilities.
ReentrantReadWriteLock	An implementation of ReadWriteLock supporting similar semantics to ReentrantLock.
StampedLock	A capability-based lock with three modes for controlling read/write access.

Locks

```
Lock l = ...;
l.lock();
try {
    // access the resource protected by this lock
} finally {
    l.unlock();
}
```

Same as:

```
Object lock = new Object();
synchronized (lock) {
    // access the resource protected by this lock
}
```

Locks: Condition

```
class BoundedBuffer {
   final Lock lock = new ReentrantLock();
   final Condition notFull  = lock.newCondition();
   final Condition notEmpty = lock.newCondition();

   final Object[] items = new Object[100];
   int putptr, takeptr, count;

   public void put(Object x) throws
                              InterruptedException {
      lock.lock();
      try {
         while (count == items.length)
            notFull.await();

         items[putptr] = x;
         if (++putptr == items.length)
            putptr = 0;

         ++count;
         notEmpty.signal();
      } finally {
         lock.unlock();
      }
   }
```

```
   public Object take() throws InterruptedException {
      lock.lock();
      try {
         while (count == 0)
            notEmpty.await();

         Object x = items[takeptr];
         if (++takeptr == items.length)
            takeptr = 0;

         --count;
         notFull.signal();

         return x;
      } finally {
         lock.unlock();
      }
   }

}
```

Locks: StampedLock

```
class Point {
    private double x, y;
    private final StampedLock sl = new StampedLock();

    void move(double deltaX, double deltaY ) {
                                  // an exclusively locked method
        long stamp = sl.writeLock();
        try {
            x += deltaX;
            y += deltaY;
        } finally {
            sl.unlockWrite(stamp);
        }
    }

    double distanceFromOrigin() {
                                        // A read-only method
        long stamp = sl.tryOptimisticRead();
        double currentX = x,
               currentY = y;

        if (!sl.validate(stamp)) {
            stamp = sl.readLock();
            try {
                currentX = x;
                currentY = y;
            } finally {
                sl.unlockRead(stamp);
            }
        }

        return Math.sqrt(currentX * currentX
                         + currentY * currentY);
    }
```

```
    void moveIfAtOrigin(double newX, double newY ) {
                                        // upgrade
                                        // Could instead start with
                                        // optimistic, not read mode
        long stamp = sl.readLock();
        try {
            while (x == 0.0 && y == 0.0) {
                long ws = sl.tryConvertToWriteLock(stamp);
                if (ws != 0L) {
                    stamp = ws;
                    x = newX;
                    y = newY;
                    break;
                } else {
                    sl.unlockRead(stamp);
                    stamp = sl.writeLock();
                }
            }
        } finally {
            sl.unlock(stamp);
        }
    }

}
```

Locks: DeadLock

```
final Object obj1 = new Object();
final Object obj2 = new Object();

new Thread(new Runnable() {
        public void run() {
            synchronized (obj1) {
                System.out.println("Thread 1 captured LOCK 1");
                Utils.waitFor(100, false);                                  // sleep 100 ms
                synchronized (obj2) {                                       // this thread will be locked forever!
                    System.out.println("Thread 1 captured LOCK 1 and LOCK 2");
                }
            }
            System.out.println("You will never see this message!");
        }
}).start();

new Thread(new Runnable() {
        public void run() {
            synchronized (obj2) {
                System.out.println("Thread 2 captured LOCK 2");
                Utils.waitFor(100, false);                                  // sleep 100 ms
                synchronized (obj1) {                                       // this thread will be locked forever!
                    System.out.println("Thread 2 captured LOCK 1 and LOCK 2");
                }
            }
            System.out.println("You will never see this message!");
        }
}).start();
```

High level concurrency: Atomics

AtomicBoolean	A boolean value that may be updated atomically.
AtomicInteger	An int value that may be updated atomically.
AtomicIntegerArray	An int array in which elements may be updated atomically.
AtomicLong	A long value that may be updated atomically.
AtomicLongArray	A long array in which elements may be updated atomically.
AtomicReference<V>	An object reference that may be updated atomically.
AtomicReferenceArray<E>	An array of object references in which elements may be updated atomically.
DoubleAccumulator	One or more variables that together maintain a running double value updated using a function.
DoubleAdder	One or more variables that together maintain an initially zero double sum.
LongAccumulator	One or more variables that together maintain a running long value updated using a function.
LongAdder	One or more variables that together maintain an initially zero long sum.

High level concurrency:

BlockingDeque<E> BlockingQueue<E>	A Deque/Queue that additionally supports blocking operations that wait for the deque to become non-empty when retrieving an element, and wait for space to become available in the deque when storing an element.
Callable<V>	A task that returns a result and may throw an exception.
ConcurrentMap<K,V>	A Map providing thread safety and atomicity guarantees.
Executor	An object that executes submitted Runnable tasks.
ExecutorService	An Executor that provides methods to manage termination and methods that can produce a Future for tracking progress of one or more asynchronous tasks.
Future<V>	A Future represents the result of an asynchronous computation.
RunnableFuture<V>	A Future that is Runnable.
RunnableScheduledFuture<V>	A ScheduledFuture that is Runnable.
ScheduledExecutorService	An ExecutorService that can schedule commands to run after a given delay, or to execute periodically.
ScheduledFuture<V>	A delayed result-bearing action that can be cancelled.

BlockingQueue

```
private static final  BlockingQueue<Apple> mQueue =
                            new  ArrayBlockingQueue<Apple>(10);

private static  class Apple {

    private static int mAppleID = 0;

    public String  toString() {
        return "apple #" + (++mAppleID);
    }
}

private static  class Tree implements Runnable {

    public void  run() {
        try {
            while (true) {
                mQueue.put(produce());
            }
        } catch (InterruptedException ex) {
        }
    }

    Apple  produce() {
        return new Apple();
    }
}
```

```
private static  class AppleLover implements Runnable {

    private String mName;
    private int mApplesConsumed = 0;

    AppleLover(String name) {
        mName = name;
    }

    public void  run() {
        try {
            while (true) {
                Utils.waitFor(500, false);
                consume( mQueue.take());
            }
        } catch (InterruptedException ex) {
        }
    }

    void  consume(Apple apple) {
        System.out.println(mName + " eaten an " +
apple.toString());
    }
}

    public static void  execute() {
        Tree appleTree = new  Tree();
        AppleLover adam = new  AppleLover("Adam");
        AppleLover eve = new  AppleLover("Eve");

        new  Thread(appleTree).start();
        new  Thread(adam).start();
        new  Thread(eve).start();
    }
```

ThreadPoolExecutor

ThreadPoolExecutor(**int** corePoolSize, **int** maximumPoolSize,
long keepAliveTime, **TimeUnit** unit, **BlockingQueue**<**Runnable**> workQueue,
optional ***ThreadFactory*** *threadFactory,* ***RejectedExecutionHandler*** *handler*)

- Core and maximum pool sizes
- On-demand construction
- Keep-alive times
- Queuing
- Hook methods

```
// Start thread pool executor
ExecutorService executor = Executors.newFixedThreadPool(MAX_CORES);
for (int i = 0; i < NUM_ITERATIONS; i++) {
    executor.execute(new Runnable() {

        @Override
        public void run () {
            // to some threaded job
        }

    });
}

// Wait till all threads finish
executor.shutdown();
while (!executor.isTerminated()) {}

System.out.println("All tasks complete!");
```

Future, Callable, FutureTask

```
    interface ArchiveSearcher {
        String  search(String target);
    }

    class App {
        ExecutorService executor = ...
        ArchiveSearcher searcher = ...

        void  showSearch(final String target) throws InterruptedException {
            Future<String> future = executor.submit(new Callable<String>() {

                public String  call() {
                    return searcher.search(target);
                }});

            // some other code here

            try {
                displayText( future.get());                              // blocks current thread until result is available, if not yet
            } catch (ExecutionException ex) {
                cleanup();
                return;
            }
        }
    }

// OR USE a FutureTask: a class implementation
    FutureTask<String> futureTask = new FutureTask<String>(new Callable<String>() {
        public String  call() {
            return searcher.search(target);
        }
    });
    executor.execute( futureTask);
```

High level concurrency:

CopyOnWriteArrayList<E> CopyOnWriteArraySet<E>	A thread-safe variant of ArrayList in which all mutative operations (add, set, and so on) are implemented by making a fresh copy of the underlying array. *Set uses COWA.*
CountDownLatch	A synchronization aid that allows one or more threads to wait until a set of operations being performed in other threads completes.
CyclicBarrier	A synchronization aid that allows a set of threads to all wait for each other to reach a common barrier point.
Phaser	A reusable synchronization barrier, similar in functionality to CyclicBarrier and CountDownLatch but supporting more flexible usage.
Executors ***[static]***	Factory and utility methods for Executor, ExecutorService, ScheduledExecutorService, ThreadFactory, and Callable classes defined in this package.
ForkJoinPool ForkJoinTask<V> ForkJoinWorkerThread	An ExecutorService for running ForkJoinTasks. Abstract base class for tasks that run within a ForkJoinPool. A thread managed by a ForkJoinPool, which executes ForkJoinTasks.
ThreadLocalRandom	A random number generator isolated to the current thread.
ThreadPoolExecutor	An ExecutorService that executes each submitted task using one of possibly several pooled threads, normally configured using Executors factory methods.