Software Development (CS2500)

Lecture 57: Special Topics in Concurrency

M. R. C. van Dongen

5 March, 2014

Contents

1 Outline **1**

2 Shared Mutable Data **2**

3 Excessive Synchronisation **4**

4 The Executor Framework **7**

5 Concurrency Utilities **7**
5.1 Concurrent Collections 7
5.2 Synchronizers 8

6 Lazy Initialisation **10**
6.1 Class Attributes 10
6.2 Instance Attributes 10

7 For Friday **11**

8 Acknowledgements **11**

1 Outline

This lecture continues studying concurrency-related issues. We start by looking at problems that may arise when you access shared mutable data in monitors and how to avoid these problems. We continue by looking at techniques that prevent excessive synchronisation. This is followed by a brief visit to the *executor framework*, which is a framework for executing task. Next we study some recent alternatives to `Java`'s built-in locking mechanism. We conclude this lecture by studying concurrent *lazy evaluation*. This lecture is based on [Bloch 2008].

2 Shared Mutable Data

The `synchronized` keyword guarantees that only one thread is allowed access to an object's monitor. The keyword is best viewed an a mechanism for locking and unlocking the monitor. If used properly, you can use the exclusive access to deal with shared data.

The `Java` language guarantees that all read and write operations are *atomic,* unless the variable's type is `double` or `long`. Here an atomic operation is an operation that works immediately, without interfering with concurrent threads. Some programmers exploit this to implement `synchronized`-free "locking." In short, this *doesn't work.*

The following exemplifies the previous statements. The class `StopThread` uses a class variable `stopRequested` for inter-thread communication. The idea is that assigning `true` to the variable stops the background thread `thread`.

Don't Try this at Home

```
import java.util.concurrent.TimeUnit;

public class StopThread {
    private static boolean stopRequested;

    public static void main( String[] args )
                throws InterruptedException {
        final Thread thread = new Thread( new Runnable( ) {
            @Override
            public void run( ) {
                int i = 0;
                while (!stopRequested) {
                    i++;
                }
            }
        } );
        thread.start( );

        TimeUnit.SECONDS.sleep( 1 );
        stopRequested = true;
    }
}
```

Here the method `TimeUnit.SECONDS.sleep( 1 )` lets the current thread sleep for 1 second. The method throws an exception if the call is interrupted, which explains why the `main( )` declares the exception `InterruptedException`.

The programmer was hoping this would let the main thread sleep for a second and then set `stopRequested` to `true`, which should let the background thread exit the `while` loop.

Unfortunately, this is not how `Java` works. The problem is that whereas `Java` guarantees that writing `true` is atomic, it doesn't guarantee when the background thread can see the change, unless there is proper synchronisation.

Because of this restriction `Java` compilers and JVMs may use a technique called *hoisting* that transforms the code

Java

```
while (!stopRequested) {
    i++;
}
```

into

```Java
if (!stopRequested) {
    while (true) {
        i++;
    }
}
```

The resulting program won't terminate.

We can only fix the problem by adding synchronisation. The following demonstrates the idea.

```Java
private static boolean stopRequested;

private static synchronized void requestStop( ) {
    stopRequested = true;
}
private static synchronized boolean stopRequested( ) {
    return stopRequested;
}
public static void main( String[] args )
              throws InterruptedException {
    final Thread thread = new Thread( new Runnable( ) {
        @Override
        public void run( ) {
            int i = 0;
            while (!stopRequested( )) {
                i++;
            }
        }
    } );
    thread.start( );
    TimeUnit.SECONDS.sleep( 1 );
    requestStop( );
}
```

Note that if we want to implement synchronisation, we must `synchronize` both "setter" and "getter" methods.

The following is an alternative solution, which uses the keyword `volatile`. Basically, the value returned by reading a `volatile` variable is the value of the most recent write.

```Java
private static volatile boolean stopRequested;

public static void main( String[] args )
              throws InterruptedException {
    final Thread thread = new Thread( new Runnable( ) {
        @Override
        public void run( ) {
            int i = 0;
            while (!stopRequested) {
                i++;
            }
        }
    } );
    thread.start( );
    TimeUnit.SECONDS.sleep( 1 );
    stopRequested = true;
}
```

The `volatile` keyword seems useful but it doesn't guarantee mutual exclusion. This is why it is only safe to use it in the present of atomic read and write operations.

To see why this is important, consider the following code fragment.

Don't Try this at Home

```
private static volatile int nextSerialNumber = 0;

public static int generateSerialNumber( ) {
    return nextSerialNumber++;
}
```

With this implementation you can easily create repeated serial numbers. To understand this, notice that the post-increment operation is really a shorthand for the following.

Don't Try this at Home

```
public static int generateSerialNumber( ) {
    final int current = nextSerialNumber;
    nextSerialNumber = current + 1;
    return current;
}
```

The first two statements in the mthod do *not* behave as a single atomic operation. Since there is no synchronisation, a thread may be "interrupted" by another thread calling the method. Should that happen, both threads assign the same value to `current`....

The best solution to dealing with shared data is confining it to a single thread. Making one thread responsible for updating is a possible solution to the problems laid out in this section. Values are shared using object references. The responsibe thread prepares the data. The responsible thread can share the data with another thread if it wants to. Synchronisation is only needed when the thread shares an object reference. Client threads use getters without synchronisation. Such objects are called *effectively immutable.* Sharing effectively immutable objects is called *safe publication.* The following is a non-exhaustive list of techniques for safely publishing an object reference:

- Store the reference as a class attribute that's initialised at class contruction time.
- Store it in a `volatile` attribute.
- Store it in a `final` variable.
- Store it in a variable with internally locked getters and setters.
- Store it in a concurrent collection.[1]

3 Excessive Synchronisation

Let's face it, synchronisation takes time. This is why special care should be taken when sharing `synchronized` variables. A delay caused by a slow, broken, or malicious client results in a delay for all threads that depend on the attribute. This is why, in general, it is a bad idea to call client methods inside a `synchronized` method or block.

A method is called *alien* if it is designed to be overridden. There are several reasons why calling an alien method inside `synchronized` methods and blocks should be avoided.

The following is the first example that uses the `Observer` pattern to inform observers about the arrival of new members in a set.

[1] More about that later.

Don't Try this at Home

```
public ObservableSet<E> extends ForwardingSet<E> {
    public ObservableSet( Set<E> set ) { super( set ); }

    private final List<SetObserver<E>> observers
        = new ArrayList<SetObserver<E>>( );

    public void addObserver( SetObserver<E> observer ) {
        synchronized(observers) { observers.add( observer ); }
    }

    public boolean removeObserver( SetObserver<E> observer ) {
        synchronized(observers) { return observers.remove( observer ); }
    }

    private void notifyElementAdded( E element ) {
        synchronized(observers) {
            for( SetObserver<E> observer : observers ) {
                observer.update( this, element );
            }
        }
    }

    …
}
```

Here `update( )` is a callback method that informs the observer about the addition of the new element in the set.

Calling alien callback method is dangerous in general. The following example shows why.

Don't Try this at Home

```
final ObservableSet<Integer> set = new ObservableSet<>( );

set.addObserver( new SetObserver<Integer>( ) {
    @Override
    public void update( ObservableSet<Integer> s, Integer e ) {
        …
        if (⟨condition⟩) {
            s.removeObserver( this );
        }
        …
    }
} );
```

Allowing alien callbacks like this will result in an exception if the callback is made during `Iterator`-based iterations. The `ArrayList` class simply doesn't allow modifications at iteration time.

Let's get back to the main thread. In the following example, we're going to create a deadlock situation.

Don't Try this at Home

```
set.addObserver( new SetObserver<Integer>( ) {
    @Override
    public void update( ObservableSet<Integer> set, Integer e ) {
        final SetObserver<Integer> observer = this;
        final Thread thread = new Thread( new Runnable( ) {
            @Override public void run( ) {
                ...
                set.remove( observer );
                ...
            }
        } );
        thread.start( );
    } } );

synchronized(set) {
    for (SetObserver<Integer> observer : set) {
        set.add( 666 );
    }
}
```

Here the call `thread.start( )` creates a new thread. The thread is started and it calls `set.remove( )`. If the method `update( )` is called in a `synchronized` block on the `ObservableSet`, the call will never terminate because `remove( )` requires locking the monitor of `set` and `set` is locked by a *different* thread.

The example is presented to show the nub of the problem. However, similar scenarios may also occur in real applications. For example, when calling alien methods in a `synchronized` block.

Solving the problem is easy. We create a copy of the set and iterate over the copy.

Java

```
private void notifyElementAdded( E element ) {
      final List<SetObserver<E>> copy = null;
      synchronized(observers) {
          copy = new ArrayList<SetObserver<E>>( observers );
      }
      for (SetObserver<E> observer : copy) {
          observer.update( this, element );
      }
}
```

The following is an alternative solution that removes the need for locking altogether. The class `CopyOnWriteArrayList`, which is provided by the package `java.util.concurrent`, provides a *thread-safe* solution for iterating, adding, and removing members to a list.

Java

```
private final List<SetObserver<E>> observers
    = new CopyOnWriteArrayList<SetObserver<E>>( );

public void addObserver( SetObserver<E> observer ) {
    observers.add( observer );
}

public boolean removeObserver( SetObserver<E> observer ) {
    observers.remove( observer );
}

public void notifyElementAdded( E element ) {
    for (SetObserver<E> observer : observers) {
        observers.add( observer );
    }
}
```

An alien method call outside a `synchronized` block is an *open call.*Open calls avoid (certain)

synchronisation errors. Furthermore, they increase concurrency because they remove the need for locking. As a rule, you should minimise the time spent in `synchronized` blocks.

4 The Executor Framework

As we've noticed in our last thread-based lecture, using `wait` and `notify` is difficult. An *executor* is an object that creates, manages, and terminates tasks. An executor also provides synchronisation primitives. The `ExecutorService` class provides tools for creating executors.

The following shows how to create, start, and terminate a single-threaded executor with the `ExecutorService`.

Creation You create the executor as follows.

```Java
final ExecutorService executor
    = ExecutorService.singleThreadExecutor( );
```

Execution To start a worker thread, you offer the executor a task.

```Java
executor.execute( runnable );
```

Joining It needed, you can wait for the executor's tasks' graceful termination by calling the instance method `awaitTermination( )`. This is the equivalent of joining the executor's thread.

```Java
executor.awaitTermination( );
```

Termination You can shut down the executor.

```Java
executor.shutdown( );
```

Another executor service is a *thread pool.* This is an executor service that manages multiple threads. Offering tasks for execution works as usual. However, the service is now multi-threaded. you can control the minimum number of threads,you can control the maximum number of threads, and you can request a fixed number of threads.

5 Concurrency Utilities

In this section we shall study some concurrency utilities. These concurrency utilities are high-level objects that remove the need for `wait` and `notify`.

5.1 Concurrent Collections

Before the arrival of the concurrent collections, multi-threaded applications had to lock collections for iteration, querying, and modification. As we've noticed before, this poses serious problems, including deadlock. The *concurrent collections* provides high-level concurrent implementations of collections interfaces such as `List`, `Queue`, and `Map` implementations. The concurrent collections

implementations manage synchronisation internally: there is no need for external locking. In fact, externally locking concurrent collection objects usually results in performance degradation. The concurrent collections are in the package `java.util.concurrent`.

5.2 Synchronizers

A *synchronizer* is an object that provides thread synchronisation. Synchronizers synchronize with the instance methods `await( )` and `countDown( )`. The instance method `countDown( )` decrements the counter of the synchroniser. The instance method `await( )`, blocks until the counter is zero. The following are some common synchronizers.

`CountDownLatch` A `CountdownLatch` is a use-once synchronizer. The method `await( )` blocks until it has been called by a given, required number of threads, which is determined at construction time. We shall use it in the next example.

`Semaphore` The second synchronizer is the `Semaphore`, which we studied in the last lecture about threads.

`CyclicBarrier` The `CyclicBarrier` is a cyclic version of `CountDownLatch`.

The following example, would have required a lot of programming if it had to be implemented with `wait` and `notify` only.

- We use an executor to create 3 tasks.
- Each task carries out a computation.
- The tasks start by reporting they're ready. This is done by notifying the main thread.
- A task may start its computation when all tasks are ready.
- The main thread waits until all tasks are done.

The following is the first part of the implementation. You must import `java.util.concurrent.*` for this.

```Java
final ExecutorService executor = Executors.newCachedThreadPool( );
final int nthreads = 3;
final CountDownLatch ready = new CountDownLatch( nthreads );
final CountDownLatch start = new CountDownLatch( 1 );
final CountDownLatch done = new CountDownLatch( nthreads );
```

The `ExecutorService` class method `newCachedThreadPool( )` returns an *unbounded thread pool* that will create new threads when they're needed. Used threads are recycled to avoid thread creation.

The `CountDownLatch` variable `start` is used by the main thread to inform the tasks they may start. If is decremented once all tasks have started running.

The `CountDownLatch` variable `ready` is decremented by the tasks to inform the main thread they've started executing. The `CountDownLatch` variable `done` is decremented by the tasks to inform the main thread they've finished executing.

The following shows the task that is carried out by the worker threads.

```Java
final Runnable task = new Runnable( ) {
    @Override public void run( ) {
        ready.countDown( ); // report presence
        try {
            start.await( ); // wait till all tasks have reported presence
            computation( ); // carry out main computation
        } catch (InterruptedException e) {
            // Omitted
        } finally {
            done.countDown( ); // report task completed
        }
    }
};
```

The first thing each task will do is calling `countDown( )` on the `CountDownLatch` `ready`. This has the effect of decrementing the counter of `ready`. Next they will wait until the counter of `start` is zero.

Further on the main thread is waiting until the counter of `ready` is zero (using `ready.await( )`). When that happens, it will decrement the counter of `start`, which will notify the tasks they can start their computation.

Each worker thread decrements the `CountDownLatch` variable `done` when they it's done with its task. Further on, the main thread will be waiting until all threads have finished (by calling `done.await( )`.

The following, which shows the code that executes the worker threads, and waits until they're done, is the last part of the example.

```Java
for (int number = 0; number != nthreads; number++) {
    executor.execute( task );
}
try {
    ready.await( );     // wait till all tasks are ready
    start.countDown( ); // let tasks start conmputation
    done.await( );      // wait till all tasks are done
} catch (InterruptedException e) {
    // Omitted
} finally {
    executor.shutdownNow( );
}
```

This is where the action starts.

- The main thread starts by creating the tasks in the `for` loop.
- The main thread waits until all threads have started executing their task. This is done by the call `ready.await( )`, the counter of which is decremented by each worker thread.
- Next the main thread notifies the worker threads they may start working. This is done by calling `start.countDown( )`.
- The call `done.await( )` blocks until each worker thread has called `done.countDown( )`, which it does when it has finished working.
- Finally, the call `exceutor.shutdownNow( )` shuts down the executor immediately. This is needed if you don't need the executor any firther. Leaving out the call will hang your program because the executor thread and its worker threads will still be alive.

6 Lazy Initialisation

An expression is *strict* when it's evaluated when it's defined. The *Lazy evaluation* programming paradigm lets you write expressions that are evaluated when needed, as opposed to when defined. This lets you implement "infinite data structures." There are dedicated lazy languages, including the functional language `Haskell`. Lazy evaluation is useful when computations are expensive or not always needed.

Lazy evaluation can be simulated. You simply postpone the computation until it's needed. Postponing computations can be done by encapsulating them with the `Command` pattern. You trigger the initialisation at a later stage. It is also possible to trigger just one initialisation.

6.1 Class Attributes

The following example initialises a shared, read-only class attribute.

```Java
private static class WrapperClass {
    private static final ⟨type⟩ value = computation( );
}

private static ⟨type⟩ getValue( ) {
    return WrapperClass.value;
}
```

This works because the `WrapperClass` class is created only when `WrapperClass.value` is needed. Creating the class triggers the computation of the value and since The value is `final`, the computation is carried out only once.

What is really nice is that modern JVMs will synchronise attribute access only to initialise the value. Next the JVM will optimise the byte code for efficient reads of the attribute. When doing this, the JVM will remove the need for testing/synchronisation.

6.2 Instance Attributes

The following example that uses lazy evaluation to initialise a shared, read-only instance attribute.

```Java
private volatile ⟨type⟩ attribute = null;

public ⟨type⟩ getValue( ) {
    ⟨type⟩ value = attribute;
    if (value == null) {
        synchronized(this) {
            value = attribute;
            if (value == null) {
                attribute = value = computation( );
            }
        }
    }
    return value;
}
```

The following are the key observations. Here it is assumed that `computation( )` returns a proper object reference (not `null`).

- First of all, the attribute is `volatile`. Without this, one thread may not be able to see the write of a different thread in time.

- The variable `value` is used to reduce the read operations on the attribute. This is useful the attribute is `volatile` and reading it directly would need more time.
- We claim the attribute is initialised only once—we shall prove that next. If the first assignment to `value` is not `null` returning `value` is correct.
- Only entering the outer `if` triggers possible initialisations. Assume we enter the outer `if` statement, then need exclusive access for the initialisation. This is why the initialisation is in a `synchronized` block. When we enter the `synchronized` block there is a (small) possibility a different thread has already initialised. This is why we only assign a value to the attribute if it isn't equal to `null`.

  Finally, notice that you cannot synchronise on the attribute because the attribute doesn't have a monitor when it's equal to `null`.

7 For Friday

Study the lecture notes.

8 Acknowledgements

This lecture is based on [Bloch 2008, Items 66, 67, 69, and 71].

References

Bloch, Joshua [2008]. *Effective Java*. Addison–Wesley. ISBN: 978-0-321-35668-0.