Software Development (cs2500)

Lecture 57: Special Topics in Concurrency

M. R. C. van Dongen

5 March, 2014

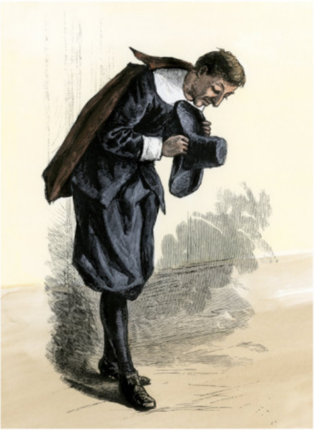
Outline

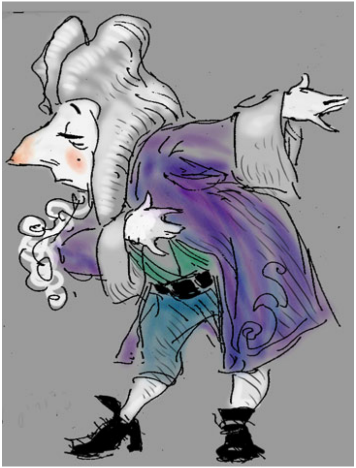
- Revisit *deadlock*.
- Accessing shared synchronised data.
- Avoiding excessive synchronisation.
- The *Executor Framework*.
- Concurrency utilities.
- Lazy initialisation.
- `Deadlock` based on on line `Java` documentation.
- The rest is based on [Bloch 2008].

Deadlock

- Deadlock occurs when:
  - Two or more threads each own a shared resource;
  - Each thread requires a resource from some other thread;
  - No thread is willing to release their resource.
- When deadlock occurs, no thread can proceed.
  - The program cannot terminate.
- A `synchronized` block/method is a shared object (resource).
- Therefore, executing them may also cause deadlock.

Enter Alphonse and Gaston





Example (Continued)

Java

```
public class Friend {
    private final String name;

    public Friend( final String name ) {
        this.name = name;
    }

    public synchronized void bow( final Friend bower ) {
        System.out.println( bower.name
                            + " has bowed to "
                            + this.name );
        pause( );
        bower.bowBack( this );
    }

    public synchronized void bowBack( final Friend bower ) {
        System.out.println( this.name
                            + " has bowed back to "
                            + bower.name );
    }
}
```

Example (Continued)

Java

```
final Friend alphonse = new Friend( "Alphonse" );
final Friend gaston = new Friend( "Gaston" );

new Thread( new Runnable( ) {
    @Override public void run( ) {
        alphonse.bow( gaston );
    } } ).start( );

new Thread( new Runnable( ) {
    @Override public void run( ) {
        gaston.bow( alphonse );
    } } ).start( );
```

Accessing Shared Mutable Data

- `synchronized` guarantees exclusive monitor access.
- Best view it as a lock.
- Lets you deal with shared data.

Atomic Reading and Writing

- `Java` guarantees that all read & write operations are *atomic*,
  - Except for `double` or `long`.
- Sometimes used to implement `synchronized`-free “locking.”
- In short, this *doesn’t work*.

How Long before this Terminates?

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;
    }
}
```

Hoisting

Java

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

Java

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

Fixing the Problem

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( );
}
```

The volatile Keyword

Makes sure Reads Read the Last 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;
}
```

No Mutual Exclusion;

Don't Try This at Home

```
private static volatile int nextSerialNumber = 0;

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

No Mutual Exclusion; No Atomicity

Don't Try This at Home

```
private static volatile int nextSerialNumber = 0;

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

Don't Try This at Home

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

Sharing Mutable Data

- Confine mutable data to a single thread.
- Make one thread responsible for updating the values.
  1. Values are shared using object references.
  2. The thread prepares the data.
  3. Thread synchronises only when it shares object references.
  4. Client threads use getters without synchronisation.
- Such objects are called *effectively immutable*.
- Sharing effectively immutable objects is called *safe publication*.
- Implementation techniques:
  - Store the reference as class attribute at class construction 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.

Synchronisation Takes Time

- Synchronisation takes time.
- Be careful when sharing `synchronized` variables.
- Delay caused by weak/malicious client delays all threads.
- Do not call client methods in `synchronized` blocks/methods.

Alien Methods

- A method is called *alien* if it is designed to be overridden.
- Avoid alien method calls in `synchronized` blocks.

Calling an Alien Method

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 );
            }
        }
    }

    ...
}
```

Calling an Alien Method : Where is It?

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 );
            }
        }
    }

    ...
}
```

Calling an Alien Method

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 );
            }
        }
    }

    ...
}
```

Calling an Alien Method in a synchronized Block

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 );
            }
        }
    }

    ...
}
```

Ouch

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 );
        }
        ...
    }
} );
```

Deadlock

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 );
    }
}
```

Solving the Problem

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 );
      }
}
```

java.util.concurrent to the Rescue

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 );
    }
}
```

Final Thoughts

- An alien method call outside a `synchronized` block is an *open call*.
  - Open calls avoid (certain) synchronisation errors.
  - They increase concurrency.
- As a rule, minimise the time spent in synchronised blocks.

Executors

- Using `wait` and `notify` is difficult.
- *An executor* creates, manages, and terminates tasks.
- An executor also provides synchronisation primitives.
- `ExecutorService` provides tools for creating executors.

Using Executors

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 You can join with all tasks.

Java

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

Termination You can shut down the executor.

Java

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

Thread Pools

- 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;
  - You can request a fixed number of threads.

Concurrent Collections

- Using normal collections requires locking.
- Poses serious problems: deadlock.
- *Concurrent collections* provides solution with concurrent
  - `List`,
  - `Queue`, and
  - `Map` implementations.
- Concurrent collections lock themselves:
  - External locking? No thanks.
  - External locking slows down.
- Concurrent collections are in `java.util.concurrent`.

Synchronizers

- A *synchronizer* coordinates thread synchronisation.
- They synchronise with `await( )` and `countDown( )`.

`CountDownLatch`	Non-reusable synchronizer.
`Semaphore`	A semaphore object.
`CyclicBarrier`	Cyclic version of `CountDownLatch`.

Example

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

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 );
```

Example (Continued)

Worker Thread

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
        }
    }
};
```

Example (Continued)

Main Thread

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( );
}
```

Introduction

- An expression is *strict* when it's evaluated when it's defined.
- It is *lazy* when it's evaluated when it's needed.
- Lets you implement "infinite data structures."
- Lazy language: `Haskell`.
- Useful when computations are expensive or not always needed.

Simulating Lazy Evaluation

- Lazy evaluation can be simulated.
- Simply postpone the computation until it's needed.
  - Use the `Command` pattern.
- Trigger the initialisation at a later stage.
- Also possible to trigger just one initialisation.

Initialising Read-Only Class Attributes

WrapperClass only Created when value is Needed

Java

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

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

Initialising Read-Only Class Attributes

JVM will Optimise Byte Code after Initialisation

Java

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

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

Initialising Read-Only Instance Attributes

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;
}
```

For Friday

- Study the lecture notes.

Acknowledgements

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

Bibliography

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

About this Document

- This document was created with `pdflatex`.
- The LaTeX document class is `beamer`.