Lock framework vs Thread synchronization in Java

Difficulty Level : Medium Last Updated : 24 Sep, 2021

Thread synchronization mechanism can be achieved using Lock framework, which is present in **java.util.concurrent** package. Lock framework works like synchronized blocks except locks can be more sophisticated than Java's synchronized blocks. Locks allow more flexible structuring of synchronized code. This new approach was introduced in Java 5 to tackle the below-mentioned problem of synchronization.

Let's look at an Vector class, which has many synchronized methods. When there are 100 synchronized methods in a class, only one thread can be executed of these 100 methods at any given point in time. Only one thread is allowed to access only one method at any given point of time using a synchronized block. This is a very expensive operation. Locks avoid this by allowing the configuration of various locks for different purpose. One can have couple of methods synchronized under one lock and other methods under a different lock. This allows more concurrency and also increases overall performance.

Example:

```
Lock lock = new ReentrantLock();
lock.lock();

// Critical section
lock.unlock();
```

A lock is acquired via the lock() method and released via the unlock() method. Invoking an unlock() without lock() will throw an exception. As already mentioned the Lock interface is present in java.util.concurrent.locks package and the ReentrantLock implements the Lock interface.

Note: The number of lock() calls should always be equal to the number of unlock() calls.

In the below code, the user has created one resource named "TestResource" which has two methods and two different locks for each respectively. There are two jobs named "DisplayJob" and "ReadJob". The LockTest class creates 5 threads to accomplish 'DisplayJob' and 5 threads to accomplish 'ReadJob'. All the 10 threads share single resource "TestResource".

```java
import java.util.Date;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

// Test class to test the lock example
// 5 threads are created with DisplayJob
// and 5 thread are created with ReadJob.
// Both these jobs use single TestResource named "test".
public class LockTest
{
    public static void main(String[] args)
    {
        TestResource test = new TestResource();
        Thread thread[] = new Thread[10];
          for (int i = 0; i < 5; i++)
          {
             thread[i] = new Thread(new DisplayJob(test),
             "Thread " + i);
          }
          for (int i = 5; i < 10; i++)
          {
             thread[i] = new Thread(new ReadJob(test),
             "Thread " + i);
          }
          for (int i = 0; i < 10; i++)
          {
             thread[i].start();
          }
    }

}
// DisplayJob class implementing Runnable interface.
// This uses TestResource object passed in the constructor.
// run method invokes displayRecord method on TestResource.
class DisplayJob implements Runnable
{

    private TestResource test;
    DisplayJob(TestResource tr)
    {
        test = tr;
    }
    @Override
    public void run()
    {
        System.out.println("display job");
        test.displayRecord(new Object());
    }
}
// ReadJob class implementing Runnable interface.
// which uses TestResource object passed in the constructor.
// run method invokes readRecord method on TestResource.
class ReadJob implements Runnable
{

    private TestResource test;

    ReadJob(TestResource tr)
    {
        test = tr;
    }
    @Override
    public void run()
    {
        System.out.println("read job");
        test.readRecord(new Object());
    }
}
// Class which has two locks and two methods.

class TestResource
{

    // displayQueueLock is created to make
    // displayQueueLock thread safe.
    // When T1 acquires lock on testresource(o1)
    // object displayRecord method
    // T2 has to wait for lock to be released
    // by T1 on testresource(o1) object
    // displayRecord method.  But T3, can execute
    // readRecord method with out waiting for lock
    // to be released by t1 as
    // readRecord method uses readQueueLock not
    // displayQueueLock.
    private final Lock
    displayQueueLock = new ReentrantLock();
    private final Lock
    readQueueLock = new ReentrantLock();

    // displayRecord uses displayQueueLock to
    // achieve thread safety.
    public void displayRecord(Object document)
    {
        final Lock displayLock = this.displayQueueLock;
        displayLock.lock();
        try
          {
             Long duration =
                         (long) (Math.random() * 10000);
             System.out.println(Thread.currentThread().
             getName() + ": TestResource: display a Job"+
             " during " + (duration / 1000) + " seconds ::"+
             " Time - " + new Date());
             Thread.sleep(duration);
          }
          catch (InterruptedException e)
```

```
            {
               e.printStackTrace();
            }
            finally
            {
               System.out.printf("%s: The document has been"+
               " dispalyed\n", Thread.currentThread().getName());
               displayLock.unlock();
            }
      }

      // readRecord uses readQueueLock to achieve thread safety.
      public void readRecord(Object document)
      {
          final Lock readQueueLock = this.readQueueLock;
          readQueueLock.lock();
          try
            {
               Long duration =
                          (long) (Math.random() * 10000);
               System.out.println
               (Thread.currentThread().getName()
               + ": TestResource: reading a Job during " +
               (duration / 1000) + " seconds :: Time - " +
               new Date());
               Thread.sleep(duration);
            }
            catch (InterruptedException e)
            {
               e.printStackTrace();
            }
            finally
            {
               System.out.printf("%s: The document has"+
               " been read\n", Thread.currentThread().getName());
               readQueueLock.unlock();
            }
      }
  }
```

Output:

display job

display job

display job

display job

display job

read job

```
read job
read job
read job
read job
Thread 5: TestResource: reading a Job during 4 seconds :: Time – Wed Feb 27 15:49:53 UT
C 2019
Thread 0: TestResource: display a Job during 6 seconds :: Time – Wed Feb 27 15:49:53 UT
C 2019
Thread 5: The document has been read
Thread 6: TestResource: reading a Job during 4 seconds :: Time – Wed Feb 27 15:49:58 UT
C 2019
```

In the above example, DisplayJob not required to wait for ReadJob threads to complete the task as ReadJob and Display job uses two different locks. This can not be possible with “synchronized”.

Differences are as follows:

Parameters	Lock Framework	Synchronized
Across Methods	Yes, Locks can be implemented across the methods, you can invoke lock() in method1 and invoke unlock() in method2.	Not possible
try to acquire lock	yes, trylock(timeout) method is supported by Lock framework, which will get the lock on the resource if it is available, else it returns false and Thread wont get blocked.	Not possible with synchronized
Fair lock management	Yes, Fair lock management is available in case of lock framework. It hands over the lock to long waiting thread. Even in fairness mode set to true, if trylock is coded, it is served first.	Not possible with synchronized
List of waiting threads	Yes, List of waiting threads can be seen using Lock framework	Not possible with synchronized

Parameters	Lock Framework	Synchronized
Release of lock in exceptions	`Lock.lock(); myMethod();Lock.unlock();` unlock() cant be executed in this code if any exception being thrown from myMethod().	Synchronized works clearly in this case. It releases the lock