Reentrant Lock in Java

Difficulty Level : Medium Last Updated : 04 Feb, 2021

Background

The traditional way to achieve thread synchronization in Java is by the use of synchronized keyword. While it provides a certain basic synchronization, the synchronized keyword is quite rigid in its use. For example, a thread can take a lock only once. Synchronized blocks don't offer any mechanism of a waiting queue and after the exit of one thread, any thread can take the lock. This could lead to starvation of resources for some other thread for a very long period of time.
Reentrant Locks are provided in Java to provide synchronization with greater flexibility.

What are Reentrant Locks?

The ReentrantLock class implements the Lock interface and provides synchronization to methods while accessing shared resources. The code which manipulates the shared resource is surrounded by calls to lock and unlock method. This gives a lock to the current working thread and blocks all other threads which are trying to take a lock on the shared resource.

As the name says, ReentrantLock allows threads to enter into the lock on a resource more than once. When the thread first enters into the lock, a hold count is set to one. Before unlocking the thread can re-enter into lock again and every time hold count is incremented by one. For every unlocks request, hold count is decremented by one and when hold count is 0, the resource is unlocked.

Reentrant Locks also offer a fairness parameter, by which the lock would abide by the order of the lock request i.e. after a thread unlocks the resource, the lock would go to the thread which has been waiting for the longest time. This fairness mode is set up by passing true to the constructor of the lock.
These locks are used in the following way:

```
public void some_method()
{
        reentrantlock.lock();
```

```
            try
            {
                //Do some work
            }
            catch(Exception e)
            {
                e.printStackTrace();
            }
            finally
            {
                reentrantlock.unlock();
            }

    }
```

Unlock statement is always called in the finally block to ensure that the lock is released even if an exception is thrown in the method body(try block).

ReentrantLock() Methods

- **lock():** Call to the lock() method increments the hold count by 1 and gives the lock to the thread if the shared resource is initially free.
- **unlock():** Call to the unlock() method decrements the hold count by 1. When this count reaches zero, the resource is released.
- **tryLock():** If the resource is not held by any other thread, then call to tryLock() returns true and the hold count is incremented by one. If the resource is not free, then the method returns false, and the thread is not blocked, but exits.
- **tryLock(long timeout, TimeUnit unit):** As per the method, the thread waits for a certain time period as defined by arguments of the method to acquire the lock on the resource before exiting.
- **lockInterruptibly():** This method acquires the lock if the resource is free while allowing for the thread to be interrupted by some other thread while acquiring the resource. It means that if the current thread is waiting for the lock but some other thread requests the lock, then the current thread will be interrupted and return immediately without acquiring the lock.
- **getHoldCount():** This method returns the count of the number of locks held on the resource.
- **isHeldByCurrentThread():** This method returns true if the lock on the resource is held by the current thread.

ReentrantLock() Example

In the following tutorial, we will look at a basic example of Reentrant Locks.

Steps to be followed

1. Create an object of ReentrantLock
2. Create a worker(Runnable Object) to execute and pass the lock to the object
3. Use the lock() method to acquire the lock on shared resource
4. After completing work, call unlock() method to release the lock

Below is the implementation of problem statement:

```
// Java code to illustrate Reentrant Locks
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.locks.ReentrantLock;

class worker implements Runnable
{
  String name;
  ReentrantLock re;
  public worker(ReentrantLock rl, String n)
  {
    re = rl;
    name = n;
  }
  public void run()
  {
    boolean done = false;
    while (!done)
    {
      //Getting Outer Lock
      boolean ans = re.tryLock();

      // Returns True if lock is free
      if(ans)
      {
        try
        {
          Date d = new Date();
          SimpleDateFormat ft = new SimpleDateFormat("hh:mm:ss");
          System.out.println("task name - "+ name
                     + " outer lock acquired at "
```

```
                    + ft.format(d)
                    + " Doing outer work");
            Thread.sleep(1500);

            // Getting Inner Lock
            re.lock();
            try
            {
              d = new Date();
              ft = new SimpleDateFormat("hh:mm:ss");
              System.out.println("task name - "+ name
                          + " inner lock acquired at "
                          + ft.format(d)
                          + " Doing inner work");
              System.out.println("Lock Hold Count - "+ re.getHoldCount());
              Thread.sleep(1500);
            }
            catch(InterruptedException e)
            {
              e.printStackTrace();
            }
            finally
            {
              //Inner lock release
              System.out.println("task name - " + name +
                          " releasing inner lock");

              re.unlock();
            }
            System.out.println("Lock Hold Count - " + re.getHoldCount());
            System.out.println("task name - " + name + " work done");

            done = true;
          }
          catch(InterruptedException e)
          {
            e.printStackTrace();
          }
          finally
          {
            //Outer lock release
            System.out.println("task name - " + name +
                        " releasing outer lock");

            re.unlock();
            System.out.println("Lock Hold Count - " +
                        re.getHoldCount());
          }
        }
        else
        {
          System.out.println("task name - " + name +
                       " waiting for lock");
          try
          {
            Thread.sleep(1000);
          }
```

```
            catch(InterruptedException e)
            {
              e.printStackTrace();
            }
          }
        }
      }
    }

    public class test
    {
      static final int MAX_T = 2;
      public static void main(String[] args)
      {
        ReentrantLock rel = new ReentrantLock();
        ExecutorService pool = Executors.newFixedThreadPool(MAX_T);
        Runnable w1 = new worker(rel, "Job1");
        Runnable w2 = new worker(rel, "Job2");
        Runnable w3 = new worker(rel, "Job3");
        Runnable w4 = new worker(rel, "Job4");
        pool.execute(w1);
        pool.execute(w2);
        pool.execute(w3);
        pool.execute(w4);
        pool.shutdown();
      }
    }
```

Sample Execution

```
Output:
task name - Job2 waiting for lock
task name - Job1 outer lock acquired at 09:49:42 Doing outer work
task name - Job2 waiting for lock
task name - Job1 inner lock acquired at 09:49:44 Doing inner work
Lock Hold Count - 2
task name - Job2 waiting for lock
task name - Job2 waiting for lock
task name - Job1 releasing inner lock
Lock Hold Count - 1
task name - Job1 work done
task name - Job1 releasing outer lock
Lock Hold Count - 0
task name - Job3 outer lock acquired at 09:49:45 Doing outer work
```

```
task name - Job2 waiting for lock
task name - Job3 inner lock acquired at 09:49:47 Doing inner work
Lock Hold Count - 2
task name - Job2 waiting for lock
task name - Job2 waiting for lock
task name - Job3 releasing inner lock
Lock Hold Count - 1
task name - Job3 work done
task name - Job3 releasing outer lock
Lock Hold Count - 0
task name - Job4 outer lock acquired at 09:49:48 Doing outer work
task name - Job2 waiting for lock
task name - Job4 inner lock acquired at 09:49:50 Doing inner work
Lock Hold Count - 2
task name - Job2 waiting for lock
task name - Job2 waiting for lock
task name - Job4 releasing inner lock
Lock Hold Count - 1
task name - Job4 work done
task name - Job4 releasing outer lock
Lock Hold Count - 0
task name - Job2 outer lock acquired at 09:49:52 Doing outer work
task name - Job2 inner lock acquired at 09:49:53 Doing inner work
Lock Hold Count - 2
task name - Job2 releasing inner lock
Lock Hold Count - 1
task name - Job2 work done
task name - Job2 releasing outer lock
Lock Hold Count - 0
```

Important Points

1. One can forget to call the unlock() method in the finally block leading to bugs in the program. Ensure that the lock is released before the thread exits.
2. The fairness parameter used to construct the lock object decreases the throughput of the program.

The ReentrantLock is a better replacement for synchronization, which offers many features not provided by synchronized. However, the existence of these obvious benefits

are not a good enough reason to always prefer ReentrantLock to synchronize. Instead, make the decision on the basis of whether you need the flexibility offered by a ReentrantLock.

This article is contributed by **Abhishek**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.
Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.