Problem Description:

How to solve deadlock using thread?

Solution:

Following example demonstrates how to solve deadlock using the concept of thread.

```
import java.util.*;
import java.util.concurrent.*;
import java.util.concurrent.locks.*;

public class DeadlockDetectingLock extends ReentrantLock {
   private static List deadlockLocksRegistry
   = new ArrayList();
   private static synchronized void
   registerLock(DeadlockDetectingLock ddl) {
      if (!deadlockLocksRegistry.contains(ddl))
      deadlockLocksRegistry.add(ddl);
   }
   private static synchronized void
   unregisterLock(DeadlockDetectingLock ddl) {
      if (deadlockLocksRegistry.contains(ddl))
      deadlockLocksRegistry.remove(ddl);
   }
   private List hardwaitingThreads = new ArrayList();
   private static synchronized void
   markAsHardwait(List l, Thread t) {
   if (!l.contains(t))
   l.add(t);
}

private static synchronized void
freeIfHardwait(List l, Thread t) {
   if (l.contains(t))
   l.remove(t);
}

private static Iterator getAllLocksOwned(Thread t) {
   DeadlockDetectingLock current;
   ArrayList results = new ArrayList();
   Iterator itr = deadlockLocksRegistry.iterator();
   while (itr.hasNext()) {
      current = (DeadlockDetectingLock) itr.next();
      if    (current.getOwner() == t)
      results.add(current);
   }
   return results.iterator();
}

private static Iterator
getAllThreadsHardwaiting(DeadlockDetectingLock l) {
   return l.hardwaitingThreads.iterator();
}

private static synchronized boolean canThreadWaitOnLock
(Thread t,DeadlockDetectingLock l) {
   Iterator locksOwned = getAllLocksOwned(t);
   while (locksOwned.hasNext()) {
      DeadlockDetectingLock current
      = (DeadlockDetectingLock) locksOwned.next();
      if (current == l)
      return false;
```

```
      Iterator waitingThreads
      = getAllThreadsHardwaiting(current);
      while (waitingThreads.hasNext()) {
         Thread otherthread = (Thread) waitingThreads.next();
         if (!canThreadWaitOnLock(otherthread, l)) {
            return false;
         }
      }
   }
   return true;
}

public DeadlockDetectingLock() {
   this(false, false);
}

public DeadlockDetectingLock(boolean fair) {
   this(fair, false);
}

private boolean debugging;

public DeadlockDetectingLock(boolean fair, boolean debug) {
   super(fair);
   debugging = debug;
   registerLock(this);
}

public void lock() {
   if (isHeldByCurrentThread()) {
      if (debugging)
      System.out.println("Already Own Lock");
      super.lock();
      freeIfHardwait(hardwaitingThreads,
      Thread.currentThread());
      return;
   }
   markAsHardwait(hardwaitingThreads,
   Thread.currentThread());
   if (canThreadWaitOnLock(Thread.currentThread(), this)) {
      if (debugging)
      System.out.println("Waiting For Lock");
      super.lock();
      freeIfHardwait(hardwaitingThreads,
      Thread.currentThread());
      if (debugging)
      System.out.println("Got New Lock");
   }
   else {
      throw new DeadlockDetectedException("DEADLOCK");
   }
}

public void lockInterruptibly() throws InterruptedException {
   lock();
}

locks.
public class DeadlockDetectingCondition implements Condition {
   Condition embedded;
   protected DeadlockDetectingCondition(ReentrantLock lock,
   Condition embedded) {
      this.embedded = embedded;
   }

   Public void await() throws InterruptedException {
      try {
         markAsHardwait(hardwaitingThreads,
         Thread.currentThread());
```

```
            embedded.await();
        }
        finally {
            freeIfHardwait(hardwaitingThreads,
            Thread.currentThread());
        }
    }

    public void awaitUninterruptibly() {
        markAsHardwait(hardwaitingThreads,
        Thread.currentThread());
        embedded.awaitUninterruptibly();
        freeIfHardwait(hardwaitingThreads,
        Thread.currentThread());
    }

    public long awaitNanos(long nanosTimeout)
        throws InterruptedException {
        try {
            markAsHardwait(hardwaitingThreads,
            Thread.currentThread());
            return embedded.awaitNanos(nanosTimeout);
        }
        finally {
            freeIfHardwait(hardwaitingThreads,
            Thread.currentThread());
        }
    }

    public boolean await(long time, TimeUnit unit)
    throws InterruptedException {
        try {
            markAsHardwait(hardwaitingThreads,
            Thread.currentThread());
            return embedded.await(time, unit);
        }
        finally {
            freeIfHardwait(hardwaitingThreads,
            Thread.currentThread());
        }
    }

    public boolean awaitUntil(Date deadline)
    throws InterruptedException {
        try {
            markAsHardwait(hardwaitingThreads,
            Thread.currentThread());
            return embedded.awaitUntil(deadline);
        }
        finally {
            freeIfHardwait(hardwaitingThreads,
            Thread.currentThread());
        }
    }

    public void signal() {
        embedded.signal();
    }

    public void signalAll() {
        embedded.signalAll();
    }
}

public Condition newCondition() {
    return new DeadlockDetectingCondition(this,
    super.newCondition());
}
```

```
private static Lock a = new DeadlockDetectingLock(false, true);
private static Lock b = new DeadlockDetectingLock(false, true);
private static Lock c = new DeadlockDetectingLock(false, true);
private static Condition wa = a.newCondition();
private static Condition wb = b.newCondition();
private static Condition wc = c.newCondition();
private static void delaySeconds(int seconds) {
   try {
      Thread.sleep(seconds * 1000);
   }
   catch (InterruptedException ex) {
   }
}

private static void awaitSeconds(Condition c, int seconds) {
   try {
      c.await(seconds, TimeUnit.SECONDS);
   }
   catch (InterruptedException ex) {
   }
}

private static void testOne() {
   new Thread(new Runnable() {
      public void run() {
         System.out.println("thread one grab a");
         a.lock();
         delaySeconds(2);
         System.out.println("thread one grab b");
         b.lock();
         delaySeconds(2);
         a.unlock();
         b.unlock();
      }
   }).start();
   new Thread(new Runnable() {
      public void run() {
         System.out.println("thread two grab b");
         b.lock();
         delaySeconds(2);
         System.out.println("thread two grab a");
         a.lock();
         delaySeconds(2);
         a.unlock();
         b.unlock();
      }
   }).start();
}

private static void testTwo() {
   new Thread(new Runnable() {
      public void run() {
         System.out.println("thread one grab a");
         a.lock();
         delaySeconds(2) ;
         System.out.println("thread one grab b");
         b.lock();
         delaySeconds(10);
         a.unlock();
         b.unlock();
      }
   }).start();
   new Thread(new Runnable() {
      public void run() {
         System.out.println("thread two grab b");
         b.lock();
         delaySeconds(2);
         System.out.println("thread two grab c");
         c.lock();
```

```
            delaySeconds(10);
            b.unlock();
            c.unlock();
        }
    }).start();
    new Thread(new Runnable() {
        public void run() {
            System.out.println("thread three grab c");
            c.lock();
            delaySeconds(4);
            System.out.println("thread three grab a");
            a.lock();
            delaySeconds(10);
            c.unlock();
            a.unlock();
        }
    }).start();
}

private static void testThree() {
    new Thread(new Runnable() {
        public void run() {
            System.out.println("thread one grab b");
            b.lock();
            System.out.println("thread one grab a");
            a.lock();
            delaySeconds(2);
            System.out.println("thread one waits on b");
            awaitSeconds(wb, 10);
            a.unlock();
            b.unlock();
        }
    }).start();
    new Thread(new Runnable() {
        public void run() {
            delaySeconds(1);
            System.out.println("thread two grab b");
            b.lock();
            System.out.println("thread two grab a");
            a.lock();
            delaySeconds(10);
            b.unlock();
            c.unlock();
        }
    }).start();
}

public static void main(String args[]) {
    int test = 1;
    if (args.length > 0)
    test = Integer.parseInt(args[0]);
    switch (test) {
        case 1:
        testOne();
        break;
        case 2:
        testTwo();
        break;
        case 3:
        testThree();
        break;
        default:
        System.err.println("usage: java
        DeadlockDetectingLock [ test# ]");
    }
    delaySeconds(60);
    System.out.println("--- End Program ---");
    System.exit(0);
    }
```

```
}

class DeadlockDetectedException extends RuntimeException {
   public DeadlockDetectedException(String s) {
      super(s);
   }
}
```

Result:

The above code sample will produce the following result.

```
thread one grab a
Waiting For Lock
Got New Lock
thread two grab b
Waiting For Lock
Got New Lock
thread one grab b
Waiting For Lock
thread two grab a
Exception in thread "Thread-1"
DeadlockDetectedException:DEADLOCK
  at DeadlockDetectingLock.
        lock(DealockDetectingLock.java:152)
  at java.lang.Thread.run(Thread.java:595)
```