Readers and Writers Problem

- **solution details**

Readers and Writers Problem

This problem occurs when many threads of execution try to

access the same shared resources at a time. There are N readers

to read data and K Writers to write data to shared resources

The readers-writers problem is a classical problem of process synchronization, it relates to a data set such as a file that is shared between more than one process at a time. Among these various processes, some are Readers - which can only read the data set; they do not perform any updates, some are Writers - can both read and write in the data sets.

The readers-writers problem is used for managing synchronization among various reader and writer process so that there are no problems with the data sets, i.e. no inconsistency is generated.

Let's understand with an example - If two or more than two readers want to access the file at the same point in time there will be no problem. However, in other situations like when two writers or one reader and one writer wants to access the file at the same point of time, there may occur some problems, hence the task is to design the code in such a manner that if one reader is reading then no writer is allowed to update at the same point of time, similarly, if one writer is writing no reader is allowed to read the file at that point of time and if one writer is updating

a file other writers should not be allowed to update the file at the same point of time. However, multiple readers can access the object at the same time.

1- semaphore mutex, wrt; // semaphore mutex is used to ensure mutual exclusion when readcnt is updated i.e. when any reader enters or exit from the critical section and semaphore wrt is used by both readers and writers

2- int readcnt; // readcnt tells the number of processes performing read in the critical section, initially 0

Functions for semaphore :

– wait() : decrements the semaphore value.

– signal() : increments the semaphore value.

Writer process:

1- Writer requests the entry to critical section.

2- If allowed i.e. wait() gives a true value, it enters and performs the write. If not allowed, it keeps on waiting

3- It exits the critical section.

```
do {
   // writer requests for critical section
   wait(wrt);

   // performs the write

   // leaves the critical section
   signal(wrt);

} while(true);
```

Reader process:

Reader requests the entry to critical section.

If allowed:

it increments the count of number of readers inside the critical section. If this reader is the first reader entering, it locks the wrt semaphore to restrict the entry of writers if any reader is inside.

It then, signals mutex as any other reader is allowed to enter while others are already reading.

After performing reading, it exits the critical section. When exiting, it checks if no more reader is inside, it signals the semaphore “wrt” as now, writer can enter the critical section.

If not allowed, it keeps on waiting.

```
do {

   // Reader wants to enter the critical section
   wait(mutex);

   // The number of readers has now increased by 1
   readcnt++;

   // there is atleast one reader in the critical section
   // this ensure no writer can enter if there is even one reader
   // thus we give preference to readers here
   if (readcnt==1)
      wait(wrt);

   // other readers can enter while this current reader is inside
   // the critical section
   signal(mutex);
// current reader performs reading here
   wait(mutex);   // a reader wants to leave

   readcnt--;

   // that is, no reader is left in the critical section,
   if (readcnt == 0)
```

signal(wrt); // writers can enter

signal(mutex); // reader leaves

} while(true);

- **A deadlock**

A deadlock occurs if each process or thread in a group is holding a resource that is needed by another member of the group, in a circular-waiting pattern. In the dining philosophers problem If all philosophers happen to get hungry at the same time, and each one starts to hold his right/left chopstick, this will, eventually leads to a deadlock because every one will be waiting for his neighbor forever.

Example :

```
Semaphore fork[N];
right = getID();
left = (right+1) % N;
while (true) {
   /* think for a while */
   fork[right].acquire();
   fork[left].acquire();
   /* eat for a while */
   fork[right].release();
   fork[left].release();
}
```

Examples with it's solution : Example 1:

```
class Account {
  double balance;

  void withdraw(double amount){
     balance -= amount;
  }

  void deposit(double amount){
     balance += amount;
  }

   void transfer(Account from, Account to, double amount){
        sync(from);
        sync(to);

        from.withdraw(amount);
        to.deposit(amount);

        release(to);
        release(from);
    }

}
```

Solution

in this case you sort the accounts on something unique to them (such as an account number, or their primary key in the DB). The actual ordering doesn't matter as long as it is a stable ordering across all participants

```
 public static void transfer(Account from, Account to, double amount){
      Account first = from;
      Account second = to;
      if (first.compareTo(second) < 0) {
          // Swap them
          first = to;
          second = from;
      }
      synchronized(first){
           synchronized(second){
                from.withdraw(amount);
                to.deposit(amount);
           }
      }
 }
```

Example 2 :

```
public class TestDeadlockExample1 {
  public static void main(String[] args) {
    final String resource1 = "ratan jaiswal";
    final String resource2 = "vimal jaiswal";
    // t1 tries to lock resource1 then resource2
    Thread t1 = new Thread() {
      public void run() {
        synchronized (resource1) {
         System.out.println("Thread 1: locked resource 1");

         try { Thread.sleep(100);} catch (Exception e) {}

         synchronized (resource2) {
          System.out.println("Thread 1: locked resource 2");
         }
        }
      }
    };

    // t2 tries to lock resource2 then resource1
    Thread t2 = new Thread() {
      public void run() {
        synchronized (resource2) {
          System.out.println("Thread 2: locked resource 2");

          try { Thread.sleep(100);} catch (Exception e) {}

          synchronized (resource1) {
            System.out.println("Thread 2: locked resource 1");
          }
        }
      }
    };


    t1.start();
    t2.start();
  }
}
```

solution

for a problem is found at its roots. In deadlock it is the pattern of accessing the resources A and B, is the main issue. To solve the issue we will have to simply re-order the statements where the code is accessing shared **resources**.

```
// Thread-2
Runnable b2 = new Runnable() {
    public void run() {
        synchronized (b) {
            // Thread-2 have resource2 but need resource1 also
            synchronized (a) {
                System.out.println("In block 2");
            }
        }
    }
};


        new Thread(b1).start();
        new Thread(b2).start();
    }

    // resource1
    private class resource1 {
        private int i = 10;

        public int getI() {
            return i;
        }

        public void setI(int i) {
            this.i = i;
        }
    }

    // resource2
    private class resource2 {
        private int i = 20;

        public int getI() {
            return i;
        }

        public void setI(int i) {
            this.i = i;
        }
    }
```

```
public class DeadlockSolved {

   public static void main(String ar[]) {
      DeadlockSolved test = new DeadlockSolved();

      final resource1 a = test.new resource1();
      final resource2 b = test.new resource2();

  // Thread-1
Runnable b1 = new Runnable() {
   public void run() {
      synchronized (b) {
         try {
            /* Adding delay so that both threads can start trying to lock resources */
            Thread.sleep(100);
         } catch (InterruptedException e) {
            e.printStackTrace();
         }
         // Thread-1 have resource1 but need resource2 also
         synchronized (a) {
            System.out.println("In block 1");
         }
      }
   }
};
```

Example 3:

```
public class DeadLockProgram {
   // Creating Object Locks
   static Object ObjectLock1 = new Object();
   static Object ObjectLock2 = new Object();

   private static class ThreadName1 extends Thread {
     public void run() {
       synchronized (ObjectLock1) {
         System.out.println("Thread 1: Has  ObjectLock1");
         /* Adding sleep() method so that
            Thread 2 can lock ObjectLock2 */
         try {
            Thread.sleep(100);
         }
         catch (InterruptedException e) {
            e.printStackTrace();
         }
         System.out.println("Thread 1: Waiting for ObjectLock 2");
         /*Thread 1 has ObjectLock1
           but waiting for ObjectLock2*/
         synchronized (ObjectLock2) {
           System.out.println("Thread 1: No DeadLock");
         }
       }
     }
   }
   private static class ThreadName2 extends Thread {
     public void run() {
       synchronized (ObjectLock2) {
         System.out.println("Thread 2: Has  ObjectLock2");
         /* Adding sleep() method so that
            Thread 1 can lock ObjectLock1 */
         try {
            Thread.sleep(100);
         }
         catch (InterruptedException e) {
            e.printStackTrace();
         }
         System.out.println("Thread 2: Waiting for ObjectLock 1");
         /*Thread 2 has ObjectLock2
           but waiting for ObjectLock1*/
         synchronized (ObjectLock1) {
           System.out.println("Thread 2: No DeadLock");
         }
       }
     }
   }

   public static void main(String args[]) {
     ThreadName1 thread1 = new ThreadName1();
     ThreadName2 thread2 = new ThreadName2();
     thread1.start();
     thread2.start();
   }
}
```

Solution

avoid Deadlock is to reshuffle the pattern in which we are accessing the object locks ObjectLock1 and ObjectLock2. In other words, just reverse the order in which we are passing the ObjectLock1 and ObjectLock2 either in ThreadName1 class or ThreadName2 class but not in both. Let's try reshuffling in the above code and see whether we still get the Deadlock or not.

```
public class DeadLockProgram {
   // Creating Object Locks
   static Object ObjectLock1 = new Object();
   static Object ObjectLock2 = new Object();

   private static class ThreadName1 extends Thread {
     public void run() {
       synchronized (ObjectLock2) {
          System.out.println("Thread 1: Has  ObjectLock1");
          /* Adding sleep() method so that
             Thread 2 can lock ObjectLock2 */
          try {
             Thread.sleep(100);
          }
          catch (InterruptedException e) {
             e.printStackTrace();
          }
          System.out.println("Thread 1: Waiting for ObjectLock 2");
          /*Thread 1 has ObjectLock1
            but waiting for ObjectLock2*/
          synchronized (ObjectLock1) {
            System.out.println("Thread 1: No DeadLock");
          }
       }
     }
   }
   private static class ThreadName2 extends Thread {
     public void run() {
       synchronized (ObjectLock2) {
          System.out.println("Thread 2: Has  ObjectLock2");
          /* Adding sleep() method so that
             Thread 1 can lock ObjectLock1 */
          try {
             Thread.sleep(100);
          }
          catch (InterruptedException e) {
             e.printStackTrace();
          }
          System.out.println("Thread 2: Waiting for ObjectLock 1");
          /*Thread 2 has ObjectLock2
            but waiting for ObjectLock1*/
          synchronized (ObjectLock1) {
            System.out.println("Thread 2: No DeadLock");
          }
       }
     }
   }

   public static void main(String args[]) {
     ThreadName1 thread1 = new ThreadName1();
     ThreadName2 thread2 = new ThreadName2();
     thread1.start();
     thread2.start();
   }
}
```

• A Starvation

• much less common a problem than deadlock

• situation where a thread is unable to gain regular access to shared resources and is unable to make progress.

– most commonly starved resource is CPU cycles

Trying to avoid deadlock, we may pick the right fork, at first, and check if the left fork is not available, put the right fork down again. If all of them got hungry at the same time, they will all hold their right fork and check their neighbors, which would be asking for the chopstick that leads to putting all right forks down, this will end up in a livelock causing Starvation

Example :

```
/* same variables as before */
while (true) {
  /* think for a while */
  fork[right].acquire();
  do {
    if (fork[left] is in use) fork[right].release();
    else fork[left].acquire();
  } while (I have less then 2 forks);
  /* eat for a while */
  fork[right].release();
  fork[left].release();
}
```

• How did we solve deadlock & starvation:

• happens when shared resources are made unavailable for long periods by "greedy" threads.

As we mentioned, the deadlock problem can result from picking up all left/right chopsticks at the same time. Our solution avoided getting in this mistake by checking ,before picking , if both of the chopsticks is available at the same time, then he can pick them together. Since picking both of them is not an atomic, can be interrupted, operation , we used the asymmetric solution in which, the even-numbered philosophers will pick their right chopstick first then left and vice versa. In view of the fact that the Starvation results from the proposed deadlock solution which leads to livelock, our solution will not fell in livelock and consequently it is starvation free solution.

Example 1 :

```
synchronized public void acquireRead(){
while(writing || waitingWriters > 0 && !readersTurn)
{
                ..wait();..
        }
        ++readers;
}
synchronized public void releaseRead(){
        --readers; readersTurn = false;
        if(readers == 0) notifyAll();
```

```
synchronized public void acquireWrite(){
while(readers > 0 || writing){
                ++waitingWriters;
                try { ..wait(); }
                }
        --waitingWriters; writing = true;
}
synchronized public void releaseWrite(){
        writing = false; readersTurn = true;
        notifyAll();        } }
```

Here we check on a boolen variable called turn it make the process start after the last one finishes and we release the process to notify the other to starts

Example 2 :

```
// Java program to illustrate Deadlock situation
class DeadlockDemo extends Thread {
    static Thread mainThread;
    public void run()
    {
        System.out.println("Child Thread waiting for" +
                           " main thread completion");
        try {

            // Child thread waiting for completion
            // of main thread
            mainThread.join();
        }
        catch (InterruptedException e) {
            System.out.println("Child thread execution" +
                                          " completes");
        }
    }
    public static void main(String[] args)
                   throws InterruptedException
    {
        DeadlockDemo.mainThread = Thread.currentThread();
        DeadlockDemo thread = new DeadlockDemo();

        thread.start();
        System.out.println("Main thread waiting for " +
                           "Child thread completion");

        // main thread is waiting for the completion
        // of Child thread
        thread.join();

        System.out.println("Main thread execution" +
                                       " completes");
    }
}
```

Solution

threads are also waiting for each other. But here waiting time is not infinite after some interval of time, waiting thread always gets the resources whatever is required to execute thread run() method.

```java
// Java program to illustrate Starvation concept
class StarvationDemo extends Thread {
    static int threadcount = 1;
    public void run()
    {
        System.out.println(threadcount + "st Child" +
                           " Thread execution starts");
        System.out.println("Child thread execution completes");
        threadcount++;
    }
    public static void main(String[] args)
                throws InterruptedException
    {
        System.out.println("Main thread execution starts");

        // Thread priorities are set in a way that thread5
        // gets least priority.
        StarvationDemo thread1 = new StarvationDemo();
        thread1.setPriority(10);
        StarvationDemo thread2 = new StarvationDemo();
        thread2.setPriority(9);
        StarvationDemo thread3 = new StarvationDemo();
        thread3.setPriority(8);
        StarvationDemo thread4 = new StarvationDemo();
        thread4.setPriority(7);
        StarvationDemo thread5 = new StarvationDemo();
        thread5.setPriority(6);

        thread1.start();
        thread2.start();
        thread3.start();
        thread4.start();

        // Here thread5 have to wait because of the
        // other thread. But after waiting for some
        // interval, thread5 will get the chance of
        // execution. It is known as Starvation
        thread5.start();

        System.out.println("Main thread execution completes");
    }
}
```