**J2 How does a fair Read-Write lock prevent many active readers from starving a writer? Can more than one writer acquire the lock at the same time?**

In a fair mode, threads contend for entry using approximately arrival-order policy. In fair read-write lock if the write thread waits for a long period of time, then it will prevent readers from accessing read operation until the write operation this will prevent many active readers who arrived first than the writer while accessing read operation. A thread that tries to acquire a fair read lock will block if either the write lock is held, or there is a waiting writer thread. The thread will not acquire the read lock until after the oldest currently waiting writer thread has acquired and released the write lock.

No, we cannot be able to perform more than one writes operation to acquire the lock at the same time. If one task is doing a write operation and another task wants to do an operation 9reads or write), it’s block until the write operation finishes.

**J3 In the slide “Potential Deadlock,” describe a series of event that could lead to a Deadlock.**

There is a deadlock in our concurrent application when there are two or more tasks waiting for a shared resource that must be free from another thread that is waiting for another shared resource that must be free by one of the first ones. In the slide there are two thread A and B. A is a linked List and B is a stack data structure. Thread A remove item from Linked list and push onto the stack whereas Thread B Pop item from the stack and insert into the Linked list. Thread A lock the linked list and stack thread and do the necessary operation and unlock it after it done. Same as Thread B lock the stack and linked list thread and do the necessary operation and unlock it after completion. Like in our definition of deadlock when two or more try to work on shared data then it led to deadlock. Both thread lock and unlock each thread to do necessary operation.

**J4 In the slide “Potential Deadlock,” why do we need two locks? Would the code still work if we only used one lock?**

We need two because we are performing operation on both data type that why we need to lock on both threads. When we are working on Thread A we remove from linked list and added that removed data to the stack same as when in thread b removed data from stack is inserted on the linked list. The code will be work if we use only one lock, but it won’t give us an appropriate or exact required data.

**J5 A Parallel Stack can be implemented using an array to store the values, and a Top pointer to indicate the index of the current top element. The following three methods are needed: Push(x): put item x on top of stack Pop(y): remove top of stack and return in y Clear: initialize stack to empty Implement a Parallel Stack class in Java and test it using many parallel threads.**

package lab;  
  
import java.util.Date;  
  
public class Main {  
  
 public static void main(String[] args) throws InterruptedException {  
  
 Date start, end;  
  
 System.*out*.println("Start");  
 start = new Date();  
  
 Thread threadPush = new Thread(new TaskThread("a", "Push"));  
 Thread threadPop = new Thread(new TaskThread("Pop"));  
 Thread threadClear = new Thread(new TaskThread("Clear"));  
  
 threadPush.start();  
 threadPop.start();  
 threadClear.start();  
  
 threadPush.join();  
 threadPop.join();  
 threadClear.join();  
  
 end = new Date();  
 System.*out*.printf("Thread Time Elapsed: %d%n", end.getTime() - start.getTime());  
 }  
}

package lab;  
  
import java.util.Stack;  
import java.util.concurrent.locks.Lock;  
import java.util.concurrent.locks.ReentrantReadWriteLock;  
  
public class TaskThread implements Runnable {  
  
 private final Stack<String> stringStack = new Stack<>();  
  
 String st;  
 String operation;  
  
 private final ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();  
 private final Lock r = rwl.readLock();  
 private final Lock w = rwl.writeLock();  
  
 public TaskThread(String st, String operation) {  
 this.st = st;  
 this.operation = operation;  
 }  
  
 public TaskThread(String operation) {  
 this.operation = operation;  
 }  
  
 @Override  
 public void run() {  
  
 stringStack.push("aa");  
 stringStack.push("bb");  
  
 if (operation.equalsIgnoreCase("Push")) {  
 String a = stringStack.push(st);  
 System.*out*.println("Push "+a);  
 } else if (operation.equalsIgnoreCase("Pop")) {  
 if (!stringStack.isEmpty()) {  
 String b = stringStack.pop();  
 System.*out*.println("Pop : "+b);  
 }  
 } else if (operation.equalsIgnoreCase("clear")) {  
 stringStack.clear();  
 System.*out*.println("Stack Clear");  
 }  
 }  
  
 public String pop(String key) { // Read Operation  
 r.lock();  
 try {  
 return stringStack.pop();  
 } finally {  
 r.unlock();  
 }  
 }  
  
 public String push(String st) { // Write  
 w.lock();  
 try {  
 return stringStack.push(st);  
 } finally {  
 w.unlock();  
 }  
 }  
  
 public void clear() {  
 w.lock();  
 try {  
 stringStack.clear();  
 } finally {  
 w.unlock();  
 }  
 }  
  
}