/**
 * lock
 * multiple readers, single writer lock
 * implemented from first principles with java primitives: synchronized,
 * wait() and notifyAll().
 * waiting readers and writers all have equal priority to acquire locks.
 * no optimizations to give earlier lock requests access.
 * @AUTHOR Arthur Goldberg
 * 4/30/03
 */
package IMAP;
class lock {
    // number of readers
    private int readers;
    // writer is 'true' if a writer holds the lock
    private boolean writer;
    // all access to 'readers' and 'writer' take place when the calling thread
    // owns synchronizeObject's monitor, so they variables stay consistent
    private Object synchronizeObject;
    
    lock() {
        readers = 0;
        writer = false;
        synchronizeObject = new Object();
    }
    /**
     * get_read_lock
     * Get a read lock. Block, perhaps forever, until the write lock is released.
     *
     */
    public void get_read_lock() throws InterruptedException {
        while (true) {
            synchronized (synchronizeObject) {
                if (!writer) // no writer
                    { // one more reader
                        readers++; // release ownership of the synchronizeObject's monitor, and return
                        break;
                    }
                // wait until a lock is released
                synchronizeObject.wait();
            }
            return;
        }
    }
    /**
     * release_read_lock
     * Release a read lock. Assumes the thread has a read lock.
     *
     */
    public void release_read_lock() {
        synchronized (synchronizeObject) {
            // one fewer reader
            readers--;
            if (readers == 0) {
                // if readers > 0 then there's no point in notifying any threads
                // because there cannot be any threads waiting to read, and any
                // thread waiting to write still will not be able to acquire
                // the lock
                // must notify all waiting threads, so that last lock release
                // gives all threads a chance to gain lock
                synchronizeObject.notifyAll();
            }
        }
    }
}
* get_write_lock
* Get the write lock. Blocks, perhaps forever, until all other locks are
* released.
* /
public void get_write_lock() throws InterruptedException
{
    while( true )
    {
        synchronized (synchronizeObject)
        {
            if( !writer && (readers == 0 )) // no writer or readers
            {
// make this thread the writer
                writer = true;
// release ownership of the synchronizeObject's monitor, and return
                break;
            }
// wait until a lock is released
            synchronizeObject.wait();
        }
        return;
    }
}/**
* release_write_lock
* Release the write lock. Assumes the thread has the write lock.
*/
void release_write_lock()
{
    synchronized (synchronizeObject)
    {
// now no writer
        writer = false;
// must notify all waiting threads
        synchronizeObject.notifyAll();
    }
}
}/**
* lock_use_example
* @AUTHOR Arthur Goldberg
* 4/30/03
* /
class mailbox {
    lock mailboxLock;
    mailbox()
    {
        mailboxLock = new lock();
        // etc.
    }
    // all the other mailbox code
}
class IMAP_commands {
    // read commands: SELECT, FETCH
    void FETCH_command()
    {
        mailbox the_mailbox;
// decide which mailbox to use, suppose it is 'the_mailbox'
        try
        {
// lock it
            the_mailbox.mailboxLock.get_read_lock();
// do all the mailbox reading
        }
catch {foo f}()
        {}finally
        {
// then release the lock
the_mailbox.mailboxLock.release_read_lock();
}

// write commands: APPEND, CLOSE, STORE
void APPEND_command()
{
    mailbox the_mailbox;
    // decide which mailbox to use, suppose it is 'the_mailbox'
    // lock it
    the_mailbox.mailboxLock.get_write_lock();
    // do all the mailbox writing
    // then release the lock
    the_mailbox.mailboxLock.release_write_lock();
}

/**
 * test the lock code
 * use asserts
 */
class testLockCode {
    static int readers = 0;
    static int writers = 0;
    static Object S;
    public static void main (String[] args) {
        S = new Object();
        testThread[] testThreads = new testThread[1000];
        for ( int i; i<100; i++)
            testThreads[i] = new testThread(30, 1000, 1000);
        for ( int i; i<100; i++)
            new testThreads[i].start();
    }
    static private void test()
    {
        synchronized (O)
        {
            assertTrue( ((writers == 0) && (readers >= 0)) ||
            ((writers == 1) && (readers == 0)));
        }
    }
    static public void increment_writer()
    {
        synchronized (O)
        {
            writers++;
        }
    }
    static public void decrement_writer()
    {
        synchronized (O)
        {
            writers--;
        }
    }
    static public void increment_reader()
    {
        synchronized (O)
        {
            readers++;
        }
    }
    static public void decrement_reader()
    {
        synchronized (O)
        {
            readers--;
        }
    }
}
class testThread implements Runnable {
  int iterations;
  int idleSleepTime;
  int holdSleepTime;

  testThread ( int i, int s1, int s2)
  {
    iterations = i;
    idleSleepTime = s1;
    holdSleepTime = s2;
  }

  public void run()
  {
    Random r = new Random();
    for( int i=0; i<iterations; i++)
    {
      sleep( (long) idleSleepTime );
      if( r.nextBoolean() )
      {
        get_write_lock();
        increment_writer();
        test();
        System.out.println(this.hashCode() + " get_write_lock");
        sleep( (long) holdSleepTime );
        test();
        decrement_writer();
        System.out.println(this.hashCode() + " release_write_lock");
        release_write_lock();
      }
      else
      {
        get_read_lock();
        increment_reader();
        test();
        System.out.println(this.hashCode() + " get_read_lock");
        test();
        decrement_reader();
        System.out.println(this.hashCode() + " release_read_lock");
        release_read_lock();
      }
    }
  }
}