TITLE

Laboratory 8. Locks

PREREQUISITES

- OOP basic knowledge
- Java programming language basic knowledge
- Algorithms

RESOURCES

- Course slides
- [Java Tutorial]
- [Java Concurrency]

LABORATORY INSTRUCTIONS

A ***lock*** is a tool that provides exclusive access to a shared resource by multiple threads. Instead of using implicit locking via the ***synchronized*** keyword, the Java Concurrency API supports various explicit locks specified by the ***Lock*** interface.

Differences between Locks and Synchronized Block:

- In contrast to the ***synchronized block*** that is fully contained within a method, the ***Lock*** has *lock()* and *unlock()* operations in separate methods.
- A *synchronized block* doesn't support the fairness. Any thread can acquire the lock once released without having the option to specify preferences.
- A "waiting" thread that is acquiring the acces to the *synchronized block* cannot be intrerruped while th *Lock* API provides the ***lockInterruptibly()*** that can be used to interrupt the thread when it is waiting for the lock.

```
public interface Lock {
   void lock();
   void lockInterruptibly() throws InterruptedException;
   boolean tryLock();
   boolean tryLock(long time, TimeUnit unit);
   Condition newCondition();
   void unlock();
}
```

Fig. 1 The Lock interface

Where a Lock replaces the use of synchronized methods and statements, a ***Condition*** replaces the use of the Object monitor methods.

Conditions (also known as condition queues or condition variables) provide a means for one thread to suspend execution (to "wait") until notified by another thread that some state condition may now be true.
The *Condition* class provides the ability for a thread to wait for some condition to occur while executing the critical section.

```
public interface Condition {
   void await() throws InterruptedException;
   boolean await(long time, TimeUnit unit)
           throws InterruptedException;
   boolean awaitUntil(Date deadline) throws InterruptedException;
   long awaitNanos(long nanosTimeout) throws InterruptedException;
   void awaitUninterruptibly();
   void signal(); // wake up one waiting thread
   void signalAll(); // wake up all waiting threads
}
```

Fig. 2 The Condition Interface

One of the various Lock implementations is the ReentrantLock:

```
public class ReentrantLock implements Lock {
   ReentrantLock(boolean fair);
   public int getHoldCount();
   public boolean isLocked();
   public boolean isHeldByCurrentThread();
   public int getQueueLength();
}
```

Fig. 3 ReentrantLock Class

Simple Lock usages:

```
Lock lock = ...;
lock.lock();
try {
   // critical section
} finally {
   lock.unlock();
}
```

Fig. 4 Simple Lock usage

```
public class CriticalSection {
   //...
   ReentrantLock lock = new ReentrantLock();
   int counter = 0;

   public void perform() {
       lock.lock();
```

```java
        try {
            // Critical section here
            counter++;
        } finally {
            lock.unlock();
        }
    }
    //...
}
```

Fig. 5 Simple ReentrantLock usage

ReentrantLock with Condition example:

```java
import java.util.Stack;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.ReentrantLock;

public class ReentrantLockWIthCondition {

    Stack<String> stack = new Stack<>();
    int size = 5;

    ReentrantLock lock = new ReentrantLock();
    Condition stackEmptyCondition = lock.newCondition();
    Condition stackFullCondition = lock.newCondition();

    public void push(String item){
        try {
            lock.lock();
            while(stack.size() == size) {
                stackFullCondition.await();
            }
            stack.push(item);
            stackEmptyCondition.signalAll();
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }
    }

    public String pop() {
        try {
            lock.lock();
            while(stack.size() == 0) {
                stackEmptyCondition.await();
            }
            return stack.pop();
        } catch (InterruptedException e) {
```

```
            e.printStackTrace();
        } finally {
            stackFullCondition.signalAll();
            lock.unlock();
        }
        return null;
    }
}
```

LABORATORY TASKS

1. Implement in Java the Producer-Consumer example in Chapter 5 slides using locks.
2. Implement the “Dining philosophers problem” (see Chapter 4 slides) using locks.