UNIVERSITÉ DE NEUCHÂTEL

Institut d'informatique
Université de Neuchâtel

CONCURRENCY: MULTI-CORE PROGRAMMING & DATA PROCESSING

Lab02 - Explicit locking in Java

RECAP

- Creating a Thread in Java
    - Implement Runnable interface
    - Extend Thread class
- Implicit locks
    - Synchronized methods
    - Synchronized statements
- Interacting with a Thread

IMPLICIT LOCKS: SYNCHRONIZED

Are there any disadvantages?

- Can not interrupt a thread while waiting to acquire a lock
- Might need to wait forever to acquire a lock
- Must release a lock within same stack frame where acquired
- Lock interface provides more extensive locking operations

EXPLICIT LOCKS

- Package: java.util.concurrent.locks
- Interfaces: Condition, **Lock, ReadWriteLock**
- Classes:
  - AbstractOwnableSynchronizer
  - AbstractQueuedLong
  - SynchronizerAbstractQueuedSynchronizer
  - LockSupport
  - **ReentrantLock**
  - **ReentrantReadWriteLock**
  - **ReentrantReadWriteLock.ReadLock**
  - **ReentrantReadWriteLock.WriteLock**

LOCK INTERFACE

- Methods:
  - Acquire lock

    `lock()`
  - Acquire lock unless thread is interrupted

    `lockInterruptibly()`
  - Aquire the lock only if it is free at invocation time

    `tryLock()`
  - Try to acquire lock for a given period of time

    `tryLock(timeout, unit)`
  - Release lock

    `unlock()`

REENTRANT LOCK

- Same behavior as the implicit monitor lock + some more
- Supports Fairness Policy: `public ReentrantLock(boolean fair)`
- Allows for `Condition` to be associated with this lock
- Provides Additional Methods for:
    - Queries:
        - Number of holds on this lock by the current threads
        - Whether current thread is waiting to acquire this lock
        - Whether any thread is waiting for the given condition associated with this lock
        - Whether lock is held by this thread
    - Returns a Collection of threads, the number of threads waiting for this lock (with or without the given Condition)

REENTRANT LOCK

```
class Counter {
    ReentrantLock lock = new ReentrantLock();
    int count;

    @Override
    public void increment() {
        lock.lock();
        count++;
        lock.unlock();
   }
}
```

What could happen if an exception is thrown?

REENTRANT LOCK

```
class Counter {
    ReentrantLock lock = new ReentrantLock();
    int count;

    @Override
    public void increment() {
        lock.lock();
        try {
            count++;
        } finally {
            lock.unlock();
        }
    }
}
```

EXERCISES

1. Write a Java program that spawns multiple threads. All threads access a shared counter (initialized to 0) in a loop (of *i* iterations). All threads read the counter to a local variable (on the stack), modify the variable (as follows) and then store it back to the counter. The threads are split in two groups with respect to the modification of the variable: threads in first group will increment the variable, threads in second group will decrement it. We define n, number of threads in first group, and m, number of threads in the second group. n, m and i will be given as runtime arguments to the program. When all threads finish, the program prints the value of the shared counter and the duration of the execution. (Check the value for equal number of incrementing and decrementing threads, i.e. n = m, and for i = 100000 iterations). In the first version of your program, reuse the Counter classes from Lab01.
2. Write a new Counter version of your counter using ReentrantLock and update the first version of your program.
3. Submit the Java program from 1) + the new Counter.