Parallel Programming
Exercise Session 7

Spring 2020

Feedback: Exercise 6

Longest Sequence

Given a sequence of numbers:

[1, 9, 4, 3, 3, 8, 7, 7, 7, 0]

find the longest sequence of the same consecutive number

Longest Sequence

```
public class LongestSequenceMulti extends RecursiveTask<Sequence> {

  protected Sequence compute() {
    if (// work is small)
      // do the work directly

    else {
      // split work into pieces

      // invoke the pieces and wait for the results



      // return the longest result
    }
  }
}
```

Outline almost as before, except:

Longest Sequence

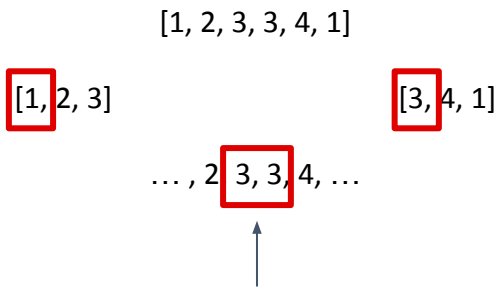
```
public class LongestSequenceMulti extends RecursiveTask<Sequence> {

  protected Sequence compute() {
    if (// work is small)
      // do the work directly

    else {
      // split work into pieces

      // invoke the pieces and wait for the results

      // check that result is not in between the pieces


      // return the longest result
    }
  }
}
```

Outline almost as before, except:

Longest Sequence

```
public class LongestSequenceMulti extends RecursiveTask<Sequence> {

  protected Sequence compute() {
    if (// work is small)
      // do the work directly

    else {
      // split work into pieces

      // invoke the pieces and wait for the results

      // check that result is not in between the pieces


      // return the longest result
    }
  }
}
```

Outline almost as before, except:



Longest Sequence

Discuss solution

Lecture Recap

Locks

Used in combination with Threads

➔ Are they needed in single-threaded program?

Locks

Used in combination with Threads

➔ Are they needed in single-threaded program?

➔ Are they needed on a single processor machine?

Thread Safe Counter

```
public class Counter {
  private int value;
  // returns a unique value

  public int getNext() {
    return value++;
  }
}
```

How to implement a thread safe Counter?

Thread Safe Counter

```
public class SyncCounter {
  private int value;

  public synchronized int getNext() {
    return value++;
  }
}
```

```
public class AtomicCounter {
  private AtomicInteger value;

  public int getNext() {
    return value.incrementAndGet();
  }
}
```

```
public class LockCounter {
  private int value;
  private Lock = new ReentrantLock();

  public int getNext() {
    lock.lock();
    try {
      return value++;
    } finally {
      lock.unlock()
    }
  }
}
```

How to implement a thread safe Counter?

Thread Safe Counter

```
public class SyncCounter {
  private int value;

  public synchronized int getNext() {
    return value++;
  }
}
```

```
public class AtomicCounter {
  private AtomicInteger value;

  public int getNext() {
    return value.incrementAndGet();
  }
}
```

```
public class LockCounter {
  private int value;
  private Lock = new ReentrantLock();

  public int getNext() {
    lock.lock();
    try {
      return value++;
    } finally {
      lock.unlock()
    }
  }
}
```

What is the difference between synchronized and a Lock?

Java: The **synchronized** keyword

Synchronization is built around an internal entity
known as the intrinsic lock or monitor lock

Every intrinsic lock has an object (or class) associated with it

A thread that needs exclusive access to an object's field has to acquire
the object's intrinsic lock before accessing them

java.util.concurrent.Lock Interface

More low-level primitive than synchronized.

Clients need to implement:

- lock(): Acquires the lock, blocks until it is acquired
- trylock(): Acquire lock only if it is lock is free when function is called
- unlock(): Release the lock

Allows more flexible structuring than synchronized blocks

What does it mean to be more flexible?
Why is this useful?

Lock Flexibility

Synchronized forces all lock acquisition and release to occur in a block-structured way

```
synchronized (A) {
  synchronized (B) {
  }
}
```

```
A.lock();
B.lock();
B.unlock();
A.unlock();
```

The following lock order cannot be expressed using synchronized blocks

```
A.lock();
B.lock();
A.unlock();
B.unlock();
```

As we will see later in the course, such order is useful for implementing concurred data structures and referred to as “hand-over-hand” locking (or “chain-locking”)

Lock Flexibility

Consider a list of locks that you should be acquired

```
public int getNext(List<Lock> locks) {

      // acquire all locks

      // critical section

      // release all locks

}
```

Can this be achieved using synchronized?

Lock Flexibility

Is the Lock acquired?

```
lock.isLocked()
```

Is the Lock acquired by current thread?

```
lock.isHeldByCurrentThread()
```

Try acquire the Lock without blocking

```
lock.tryLock()
```

https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/ReentrantLock.html

Implementing Classes of java.util.concurrent.Lock

ReentrantLock
ReentrantReadWriteLock.ReadLock
ReentrantReadWriteLock.WriteLock

Readers/Writers Lock will be covered in detail in 3 weeks

https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/Lock.html

Basic Synchronization Rule

Access to **shared** and **mutable state** needs to be **always protected**!

Synchronization Issues

Data Race: ??

Synchronization Issues

Data Race: A program has a data race if, during any possible execution, a memory location could be written from one thread, while concurrently being read or written from another thread.

Synchronization Issues

Data Race: A program has a data race if, during any possible execution, a memory location could be written from one thread, while concurrently being read or written from another thread.

Deadlock: ??

Synchronization Issues

Data Race: A program has a data race if, during any possible execution, a memory location could be written from one thread, while concurrently being read or written from another thread.

Deadlock: Circular waiting/blocking (no instructions are executed/CPU time is used) between threads, so that the system (union of all threads) cannot make any progress anymore.

Quiz: What is wrong with this code?

```
void exchangeSecret(Person a, Person b) {
    a.getLock().lock();
    b.getLock().lock();
    Secret s = a.getSecret();
    b.setSecret(s);
    a.getLock().unlock();
    b.getLock().unlock()
}
```

```
public class Person {
  private ReentrantLock mLock = new ReentrantLock();
  private String mName;

  public ReentrantLock getLock() {
    return mLock;
  }

  ...
}
```

Quiz: What is wrong with this code?

```
void exchangeSecret(Person a, Person b) {
    a.getLock().lock();
    b.getLock().lock();
    Secret s = a.getSecret();
    b.setSecret(s);
    a.getLock().unlock();
    b.getLock().unlock()
}
```

```
public class Person {
  private ReentrantLock mLock = new ReentrantLock();
  private String mName;

  public ReentrantLock getLock() {
    return mLock;
  }

  ...
}
```

Thread 1:
exchangeSecret(p1, p2)

Deadlock

Thread 2:
exchangeSecret(p2, p1)

Possible solution

```
void exchangeSecret(Person a, Person b) {
      ReentrantLock first, second;
      if (a.GetName().compareTo(b.GetName()) < 0) {
            first = a.getLock(); second = b.getLock();
      } else if (a.GetName().compareTo(b.GetName()) > 0) {
            first = b.getLock(); second = a.getLock();
      } else { throw new UnsupportedOperationException(); }
      first.lock();
      second.lock();
      Secret s = a.getSecret();
      b.setSecret(s);
      first.unlock();
      second.unlock();
}
```

Always acquire and release the Locks is the same order

Deadlocks and Race conditions

Not easy to spot

Hard to debug

➔ Might happen only very rarely

➔ Testing usually not good enough

Reasoning about code is required

Lesson learned: Need to be careful when programming with locks

Exercise 7

Exercise 7

Banking System

- Multi-Threaded Implementation
- Coding exercise: Use **synchronized** and/or **Locks**
  - Might have to make additions to existing classes
- Reason about Performance
- Reason about Deadlocks
- Run Tests

Multi-threaded Implementation

Task 1 – Problem Identification:

The methods of the classes **Account** and **BankingSystem** must be thread-safe.

You should understand why the current implementation does not work for more than one thread.

Thread-Safe – transferMoney()

Task 2 – Synchronized:

A simple solution to make the *transferMoney()* thread-safe is to use the **synchronized** keyword:
public synchronized boolean transferMoney(…)

Even though the code works as expected, the performance is poor.

The performance of the multi-threaded implementation is worse than the single-threaded. Why does this happen?

Performance of transferMoney()

Task 3 – Locking:

Since the solution with the synchronized keyword does not perform well, you should find a better strategy to achieve the thread-safe implementation.

- *Does your proposed solution work if a transaction happens from and to the same account?*
- *How do you know that your proposed solution does not suffer from deadlocks?*

ThreadSafe - sumAccounts()

Task 4 – Summing Up

With a fine-grained synchronization on the transfer method, the method sumAccounts() may return incorrect results when a transaction takes place at the same time.

- Explain why the current implementation of the sumAccounts() method is not thread-safe any more.
- You should provide a thread-safe implementation.
- Is there any way to parallelize this method?

Testing

You should run the provided tests for your implementation.

If the test succeeds, your code is not necessarily correct.

It is hard to reproduce a bad interleaving.