SFDV3006
Concurrent Programming

Lecture 2 - Synchronization

Introduction

- Last week we have seem how easy it is to create threads in Java
- In the example from Lab1 (Q3) we have seen parallel execution of the even thread and odd thread which gave us some interesting output of the even and odd numbers mixed
- This week we look at the problems that will be encountered when when concurrent programs **share data** and how synchronization can be used to solve this.

Concurrent execution

- Concurrent execution can be either of the two types:
  - Concurrency
  - Parallelism
- Concurrency: logically simultaneous processing. Does not imply multiple PEs processing elements (I.e, CPUs) . Requires interleave execution on a single PE.
- Parallelism: Physically simultaneous processing . Multiple CPUs or cores
- Both concurrency and parallelism will access and need shared resources (such as memory, files etc)

Interleaving and interference

- Instructions from two or more threads will execute in parallel with no order between them – this is called interleaving
- Interleaving is not a problem in itself however if this happens with threads that are sharing the same data and modifying it then we will have *interference.*
- Destructive update, caused by the arbitrary *interleaving of read and write actions*, is called ***interference***.

Interference - example

- Assume that a shared object of the following class is shared among more than one thread which will call the *getNextInt()*

```
public class Sequence {
   private int next = 0;
   public int getNextInt() {
      next = next + 1;
      return next;
   }
}
```

  - Let say at some point during the execution the value of the *next* is 3 and two threads execute *getNextInt()* in parallel – what value will this method return for both?
  - What will be value of *next* after both of them complete executing the method?

Interference – example (cont'd)

- We know that the correct value of *next* after both the threads complete should be 5.
- It may happen that sometimes it will be 5 which is correct however sometimes it will be 4 which is wrong!
- To understand how this happens we need to **trace** the execution of the two threads – like the example of t1 and t2 when the value of next is 3 shown below

t1	t2
Read 3	
Increment next	Read 3
	Increment next
Write next – 4 !!	Write next – 4!!

Correct value of next after t1 and t2 execute should be 5 However we get only 4 since we lose an update from one of threads!

Interference – cont'd

- Why did this happen?
  - The increment operation was not atomic – it is divided into steps which are read → increment → write
  - Atomic operations finish completely and cannot be interrupted by another thread
- Problems with interference:
  - Violates the *safety property* – it is producing incorrect results.
  - Difficult to detect and test. This problem is difficult to debug but can have devastating effects (such as Therac 25 in which cancer patients actually died)
- Solution to prevent interference:
  - Make the shared object *thread safe*
  - Make sure that only thread can execute the method at any one time
  - Make sure that the method is executed atomically and the results of execution are available to all other threads

Solution to interference - synchronization

- The solution for interference is ***mutual exclusion*** - make sure only one thread or process is inside the ***critical section***
- *Critical section* is that part of the code which is accessing shared variables/objects that should not be accessed by more than one thread concurrently
- Our problem was the *getNextInt()* method which allows more than one thread to change the value of the *next* – this was the critical section
- Solution is to use the Java ***synchronized*** keyword to make sure that there is only one thread executing in the critical section at any time – which in our example is the *getNextInt()* method

```
public synchronized int getNextInt() {
  next = next + 1;
  return next;
}
```

Now only one thread can execute this method at any one time.

Traces after synchronization

- After synchronization the traces for the shared *Sequence* object will be either of these two

The threads are now serialized – they runone after another – this means no concurrency or parallelism!!

t1	t2
Read 3	
Increment next	
Write next – 4	
	Read 4
	Increment next
	Write next – 5

When t1 runs first

t1	t2
	Read 3
	Increment next
	Write next – 4
Read 4	
Increment next	
Write 5	

When t2 runs first

Java synchronization

- Every object in Java has **lock** associated with it, normally this lock is not used.
- When a method is declared as *synchronized* then this lock becomes active and a thread needs to acquire the lock first before it can call methods on the shared object.
- If another thread wants to call a method on the object the JVM will check if the lock for that object in available – if it is available the thread acquires the lock and executes the method
- If the lock is not available (which means another thread is executing in the shared object) than the calling thread has to wait till the lock is released by the first thread

Java synchronization - issues

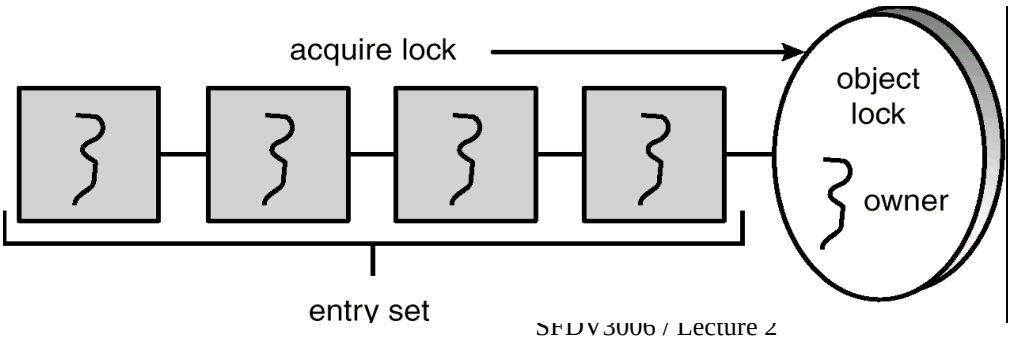
•When a lock is not contended – there is only one thread, the thread acquires the lock

•When a lock is (highly) contended – many threads are trying to acquire to the lock – only one thread acquires the lock and the other thread have to wait in the ***entry set*** (see diagram below)

•This type of locking effectively ***serializes*** access to the object lock and to the object itself – no concurrency or parallel processing!!

•Affects of serializing access to object lock on performance?

- Java locking with synchronized also causes a thread to ***block*** (until the lock becomes available)
- There is no way for a thread to back-off if the lock is already acquired.



Implementing Shared Objects and threads in Java - 1

- Assume that in our program we have two threads both of which access the *Sequence* object concurrently.
- A instance of the Sequence class will need to be passed to the both the threads usually in the constructor – this is done as shown below.

```
//create an instance of Sequence – the shared obj
Sequence sequence = new Sequence();
SequenceThread s1 = new SequenceThread(sequence);
SequenceThread s2 = new SequenceThread(sequence);
s1.start(); s2.start();
```

- The above code will be part of the main thread
- See the next page for *SequenceThread* class

Implementing Shared Objects and threads in Java - 2

- The SequenceThread class will be as shown below.

```
public class SequenceThread extends Thread {
   private Sequence sequence = null;
   public SequenceThread(Sequence _sequence) {
      sequence = _sequence;
   }

   public void run() {
      for (int i=0; i<=10; i++)
         System.out.println(sequence.getNextInt());
   }
}
```


What to synchronize?

- When a class has more than one methods which read or write shared variables than all those methods should be synchronized
- Consider the following slightly modified code example from the *Java Tutorial* in which every method is synchronized

```
public class SynchronizedCounter {
    private int c = 0;
    public synchronized void increment() {
        c++;
    }
    public synchronized void decrement() {
        c--;
    }
    public synchronized int getValue() {
        return c;
    }
}
```

Threads do not need a lock for methods that are not synchornized.

What happens if the getValue() method was not synchronized?

Lock scope

- One way of increasing the performance with locking is to decrease the *scope of the lock* (the block) to the minimum.

```
class X{
   public synchronized void x1(){
       //many statements
       //…..
       //<start critical section>
       counter++;
       //<end critical section>
       //some more statements
       //…….
   }
}
```

Scope is for the whole method
Lock is held for more time by a thread

```
class X{
   public void x1(){
       //many statements
       //…..
       synchronized (this){
       //<start critical section>
              counter++;
       //<end critical section>
       }
       //some more statements
       //…….
   }
}
```

Scope is reduced only to the critical section. Lock is held for a shorter time.
Note the use of this – which refers to the current object.

Books in the LRC

- *The Art of MultiProcessor Programming* – Maurice Herlihy and Nir Shavit, Morgan Kaufmann
- *The Java Language Specification*, 4e – Chapter on threads – also available online
- More later as I find whats there

Resources and references

- Concurrent Programming in Java, 2e, Doug Lea, Addison Wesley
- Concurrency: State Models & Java Programs, 2e, Jeff Magee & Jeff Kramer, Wiley
- Java Synchronization - *http://bit.ly/javasync*

  From the Java Tutorial – read Thread Interference and Synchronized Methods for this lecture
- Java Concurrency tutorial – is the official Java tutorial for using the Thread API and concurrency in Java - *http://bit.ly/jconctut*
- *Course Website – http://sfdv3006.wikidot.com*