COM1032 Mobile Computing
Lab 5
Android Threads Synchronisation

Purpose:

The purpose of this lab session is to familiarise yourself with the Android and Java Synchronisation programming constructs.

Aim

By the end of the lab you will be able to:

- Differentiate the different ways we can do synchronization in Java
- Practice a number of problems on which synchronization is required and how it affects the logical output and possible errors elimination methods.

Race Condition Example:

Let's start by the ATM example without any synchronisation and see how the shared balance variable will be affected. Create in Android Studio a project named "SyncTests" with empty activity with all default configuration. In the project create a class named "ATM", and implement it as follows:

```
package com.example.synctests;
public class ATM {
        private int balance = 0;
        public void deposit() {
             balance ++;
        }
        public void withdraw() {
              balance --;
        }
        public int getBalance() {
            return balance;
        }
}
```

In the activity_main.xml update the default TextView to the following:

```
<TextView
    android:id="@+id/textview"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Hello World!"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintTop_toTopOf="parent" />
```

In the MainActivity.java import the Runnable Interface, Thread Class, and TextView class as discussed in the lecture. Add the following class members to keep track of the number of times deposit method was called from thread A, and number of times withdrawal method was called from thread B:

```
TextView textView;
ATM atm;
int deposits = 0;
int withdrawals = 0;
Handler handler = new Handler();
```

The Handler is to enable periodic updating of the UI. You will need a Runnable to pass to the handler, and keep sending itself as a message to its run method. This will create the periodic updates of the UI. Implement this Runnable as follows:

```
private final Runnable mUIUpdater = new Runnable(){
   public void run(){
         try {
                        //prepare and send the data here..
                        String msg = "Current Balance: " + String.valueOf(atm.getBalance()) + " after
                               "+ deposits + " deposits and " + withdrawals + " withdrawals";
                        textView.setText(msg);
                        System.out.println(msg);
                        // resend itself after 1000 milliseconds to re-update the UI
                        handler.postDelayed(this, 1000);
        }
        catch (Exception e) {
                        e.printStackTrace();
        }
  }
};
```

In the onCreate method in the MainActivity.java add the following code at the end of the method:

```

atm = new ATM ();

Thread A = new Thread (new Runnable() {
    public void run (){
        while (true) {
            atm.deposit();
            deposits ++;
            try {
                Thread.currentThread().sleep((int)(Math.random() * 1000));
            } catch (InterruptedException e) {}
        }
    }

});

Thread B= new Thread (new Runnable() {
    public void run (){
        while (true) {
            atm.withdraw();
            withdrawals++;
            try {
                Thread.currentThread().sleep((int)(Math.random() * 2000));
            } catch (InterruptedException e) {}
        }
    }

});

A.start();
B.start();

textView = (TextView) findViewById(R.id.textView);//get id of ToThread
handler.post(mUIUpdater);
```

This will create the threads A and B, one calling atm.deposit method in an infinite loop, and the other calling atm.withdraw method infinitely. It will also post the UIUpdater runnable to the first time, and then will be called every one second from inside the run method in the Runnable.

Exercise 1: Run the project and check the UI, and as well the console printing to observe the performance. take a copy of the console output and run again and check if every run will give same output or not.

Semaphores as Mutex:

Now let's try to synchronise the ATM example as follows. Add the following class member in the ATM class:

```
static Semaphore semaphore = new Semaphore(1); // only one thread in the CS
```

Since the balance variable is the shared resource between both threads, updating its value is the critical section that require a mutex to guarantee that only one thread is actively updating the balance. Before every update to the balance value acquire the semaphore, then release it:

This how to acquire the semaphore:

```
try {
    semaphore.acquire();
}
catch (java.lang.InterruptedException e) {}
```

This is how to release it:

```
semaphore.release();
```

Exercise 2: Run the project and observe the output in the UI and the console. Notice that thread A deposits every 1 second, and thread B withdraws every 2 seconds. Run several times and observe. Is there any change in the output between the different runs?

Hint: Balance = withdrawals – deposits

Producer/Consumer Monitor Example:

Create a new project with empty activity named "ProdConsMonitor" using default settings. Add the Producer class to extend Thread and implement as follows:

```
public class Producer extends Thread{
  static final int MAXQUEUE = 5;
  private Vector messages = new Vector();
```

```
 @Override
 public void run (){
       try {
             while (true) {
                    putMessage();
             }
   }catch (InterruptedException e) {}
 }

 private synchronized void putMessage() throws InterruptedException{
      while (messages.size() == MAXQUEUE)
       wait();
   messages.addElement(new java.util.Date().toString());
      System.out.println("put message, vector size = " + messages.size());
   notify();
 }

 public synchronized String getMessage () throws InterruptedException {
      notify();
       while (messages.size() == 0)
            wait();
   String message = (String) messages.firstElement();
       messages.remove(message);
       return message;
  }
}
```

Now create new Consumer class to extend Thread class and implement it as follows:

```
public class Consumer extends Thread {
   private Producer producer;

   public Consumer (Producer p) {
             producer = p;
      }

      @Override
      public void run(){
            try {
                  while (true) {
                          String message = producer.getMessage();
                          System.out.println("Consumer Got Message: " + message);
                   }
     } catch (InterruptedException e) {}
      }
}
```

Back to the onCreate method in the MainActivity.java, create the producer and the consumer threads as follows:

```
Producer p = new Producer();
p.start();
Consumer c = new Consumer(p);
c.start();
```

Notice we used a mutex using synchronised keyword on the method and used the Object class wait/notify messaging.

Exercise 3: Run the project and observe the output in the console. Run several times. Check the messages vector size. Is it the same between the different runs?

Producer/Consumer Locks and Conditions Example:

Now create a new project named “ProdConsLockCond” using empty activity using default settings again. Create a Producer class that extends Thread class and implement it as follows:

```
public class Producer extends Thread {
 ReentrantLock lock;
    Condition con;
    Queue<Integer> queue;
    int size;

    public Producer(ReentrantLock lock, Condition con, Queue<Integer> queue, int size) {
       this.lock = lock; this.con = con; this.queue = queue; this.size = size;
    }
    public void run() {
       for (int i = 0; i < 10; i++) {
            lock.lock();
            while (queue.size() == size) {
                try {
                     con.await();
                } catch (InterruptedException ex) {
                     Logger.getLogger(Producer.class.getName()).log(Level.SEVERE, null, ex);
                }
            }
            queue.add(i);
            System.out.println("Produced : " + i + " queue size: " + queue.size());
            con.signal();
             lock.unlock();
        }
    }
}
```

Now Create the Consumer class to extend the Thread class and implement it as follows:

```
public class Consumer extends Thread {
   ReentrantLock lock;
   Condition con; Queue<Integer> queue;

   public Consumer (ReentrantLock  lock, Condition con, Queue<Integer> queue) {
      this.lock = lock;this.con = con; this.queue = queue;
   }

   @Override
   public void run () {
      for (int i = 0;i<10;i++) {
         lock.lock();
         while (queue.size()<1){
            try {
               con.await();
            } catch (InterruptedException ex) {
               Logger.getLogger(Consumer.class.getName()).log(Level.SEVERE, null, ex);
            }
         }
         System.out.println("Consumed : " + queue.remove());
         con.signal();
         lock.unlock();
      }
   }
}
```

In the onCreate method in the MainActivity.java add the following code at the end of the method:

```
Queue<Integer> queue=new LinkedList<Integer>();
ReentrantLock lock=new ReentrantLock();
Condition con=lock.newCondition();
final int size = 5;
new Producer(lock, con, queue, size).start();
new Consumer(lock, con, queue).start();
```

Exercise 4: Run the project and observe the output in the console. Run several times. Check the messages queue size, and when produced and when consumed. Is it the same between the different runs?

Conclusion

You should now have a working example that demonstrates how to synchronize access to shared variables using various java synchronisation programming constructs. These examples show as well various way to object ownership and how to pass them to different threads. The periodic UI update using Handler messaging is very useful. You should notice missing packages to import to make these code snippets work together. Android Studio suggest these packages for you. You can check the lecture for more information.