COM1032 Operating Systems
Lab 5
Threads Synchronisation

Purpose:

The purpose of this lab session is to familiarise yourself with the 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 a class named "ATM", and implement it as follows:

```
public class ATM {
         private static int balance = 0;
         private static int deposits = 0;
         private static int withdrawals = 0;
         public void deposit() {
                  balance ++;
         }
         public void withdraw() {
                  balance --;
         }
         public int getBalance() {
                  return balance;
         }
         public static void main(String[] args) {
                  ATM 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) {}
```

```
                                                System.out.println ("Thread A have made " +
deposits + " deposits and see the balance as " + balance);
                                        }
                                }
                        });
                        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) {}
                                                System.out.println ("Thread B have made " +
withdrawals + " withdrawals and see the balance as " + balance);
                                        }
                                }
                        });
                        A.start();
                        B.start();
        }
}
```

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.

Exercise 1: a) Run the project and check the console messages to observe the performance. Since these threads run forever, you will need to interrupt the process. Take a copy of the console output and run again and check if every run will give same output or not.

b) Change the waiting time, to make thread B faster than Thread A.

c) If you cancel the sleeping or make it of equal time, accidently the order of execution by the scheduler might make the thread that withdraws be faster than that that deposits, and balance become negative. Check that total deposits and total withdrawals will create the correct balance every time you run without sleeping or equal sleeping.

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); // the parameter 1 means only one thread in the CS
```

Import the required class:

```
import java.util.concurrent.Semaphore;
```

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 is how to acquire the semaphore:

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

This is how to release it:

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

https://docs.oracle.com/en/java/javase/13/docs/api/java.base/java/util/concurrent/Semaphore.html

Exercise 2: Run the ATM file again, after updating the balance updating methods (atm.deposit and atm.withdraw) to apply a semaphore acquiring and releasing to guarantee that only one thread can access the balance variable at the same time. Observe the output in 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:

Lets resume the Producer/Consumer example from previous labs. Update the Producer class to extend Thread and implement the run method as follows:

```
import java.util.Vector;
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) {}
      }
}
```

Now create the test class, which was the Factory class in last week lab. In the main method, create the producer and the consumer threads as follows:

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

Notice we created a critical section by using synchronised keyword on the methods `putMessage` and `getMessage. We also` used the Object class wait/notify messaging to make sure the queue has something before the consumer reads, and notify the waiting consumer when adding to the queue by the producer.

https://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html

https://docs.oracle.com/en/java/javase/13/docs/api/java.base/java/lang/Object.html#wait()

https://docs.oracle.com/en/java/javase/13/docs/api/java.base/java/lang/Object.html#notify()

https://docs.oracle.com/en/java/javase/13/docs/api/java.base/java/lang/Object.html#notifyAll()

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:

Update the Producer class as follows:

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

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;
    }
```

```
  @Override
  public void run() {
     for (int i = 0; i < 10; i++) {
            lock.lock();
            while (queue.size() == size) {
                   try {
                          con.await();
                   } catch (InterruptedException ex) {
                          ex.printStackTrace();
                   }
            }
            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:

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

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) {
                ex.printStackTrace();
            }
         }
         System.out.println("Consumed : " + queue.remove());
         con.signal();
         lock.unlock();
      }
   }
}
```

Now update the main method in the Factory class as follows:

```
import java.util.LinkedList;
import java.util.Queue;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.ReentrantLock;

public class Factory {
            public static void main(String[] args) {
                        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();


        }
}
```

https://docs.oracle.com/en/java/javase/13/docs/api/java.base/java/util/concurrent/locks/Condition.html

https://docs.oracle.com/en/java/javase/13/docs/api/java.base/java/util/concurrent/locks/ReentrantLock.html

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, i.e. the main method in Factory created the Lock object and owns it, and passed it to the threads that want to synchronise using it. We used Semaphores, Synchronised Keyword, Conditions, and Locks. You should be able to understand the theoretical notion of a Monitor (a mechanism to control concurrent access to an object) that will be discussed in the lecture and can be implemented as follows (passing an object to the synchronised critical section):

Thread 1:

```
public void a() {
    synchronized(someObject) {
        // do something (1)
    }
}
```

Thread 2:

```
public void b() {
    synchronized(someObject) {
        // do something else (2)
    }
}
```

This prevents the two Threads 1 and 2 that have access to the (sameObject) accessing the monitored (synchronized) section at the same time when the same object is sent to the method. One will start, and monitor will prevent the other from accessing the region before the first one finishes. Other threads with other objects will be able to call the method concurrently among those who share the (sameObject) instance.

This synchronization mechanism placed at class hierarchy root: `java.lang.Object`.
The `wait` and `notify` methods use object's monitor to communication among different threads.

Exercise 5: Implement a process scheduler that creates the following threads:

1. Process Creation Thread: One thread that receives the new process execution order, by identifying a file format for the basic commands to be executed by this thread. This thread should create a PCB (process control block) as a data structures implemented as a Java class if you choose Java with the text (code) file included, and maintain a ready queue of these PCB objects.
2. Dispatcher Thread: A second thread that checks the ready queue and dispatches one process according to the scheduling algorithm implemented in this thread. The various scheduling algorithms are implemented in Java in this file: https://github.com/MMayla/Process-Scheduling-Simulator/blob/master/app_pack/ProcessScheduler.java

3. CPU Thread: A third thread should simulate the CPU. Once assigned a PCB by the dispatcher, it should be loading the code file, and start executing the instructions one by one. You choose how the code file is formatted, and how it is read (decoded) and executed. If one instruction line is asking for I/O, the PCB with the current values of variables and Program Counter should be added to I/O queue, and notify the dispatcher thread that the CPU now wants another process to execute. If a process terminates, its PCB state should be updated and archived for accounting and the dispatcher thread should be notified to send another process PCB.
4. I/O thread: A fourth thread is the I/O manager. It is responsible for the I/O queue, and follows a scheduling algorithm on the I/O Queue. Once a PCB object is added to this queue, it waits for its turn to be serviced. Once its turn comes, it is executed, by reading an input of the user if required and updating the required variable. Then added back to the ready queue.

<u>Synchronisation Example Requirements:</u>

Synchronisation need to be maintained on the ready queue (accessed by all threads), and I/O queue (accessed by CPU and I/O threads only). The ready queue is inserted into by the process creation thread, accessed by all other threads: the dispatcher removes one PCK object to the CPU to execute, the CPU thread might added it the I/O queue if required, or to the archived queue (accessed by the CPU thread only) when terminated. The I/O thread process the I/O requests of the processes added to the I/O queue, then adds them back to the ready queue.

<u>Process Code file format can be as the following examples:</u>

```
Priority = 99
Code:
int x = 5;
int y = 10;
int z = x + y;
exit;
```

The above example is a compute only process. For example, if you add a “print z;” at the end of the code section before the “exit;” instruction, then there is I/O, and at this line execution, this process should be pre-emptied from the CPU and added to the I/O Queue. You can also read from the user a variable by including a read instruction, for example: “read x;” instead of hard coding its value as in the above example. You can also limit your instruction set to only integer data types, and hence you do not need to define a data type for your variable, such as in python: “x = 5;”. To parse this in Java you read line by line, and tokenise, and create the required variables and values. However, to make your life easier, you can use the ScriptEngine class and evaluate it as a Javascript string.

```
ScriptEngineManager manager = new ScriptEngineManager();
ScriptEngine engine = manager.getEngineByName("js");
Object result = engine.eval("4*5");
```

You can maintain some instructions that your CPU supports and make that as limited as you wish, by providing a 2 processes at least that uses the Instruction set that you support. In the coursework you will need to include a section in the report about your supported Instruction Set, and code file format.

This exercise contributes to your final coursework, and as you elaborate on it, you can achieve more of the coursework requirements. You can also use the SimpleShell class that you created in Lab2 to be your first thread code. Since this will contribute to your coursework, there will be no model answer for it next week.