CAVENDISH CAMPUS

School of Electronics and Computer Science

Modular Undergraduate Programme
First Semester 2010 – 2011

Module Code: 3SFE605

Module Title: Concurrent Programming

Date: May 2011

Time: 14:00 – 16:00

Instructions to Candidates:

Answer THREE questions.
Each question is worth 33 marks.

Question 1

(a) Describe the two methods by which a programmer can create a thread in a Java program. How would you decide which method to use? Illustrate your answer by means of suitable code fragments. **[8 marks]**

(b) Figure 1 represents the *un-labelled* states (nodes) and transitions (arcs) of the life-cycle of a Java thread.

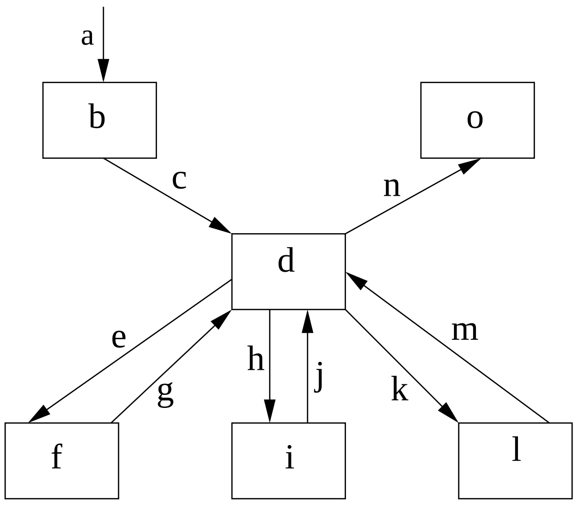

Figure 1: The life-cycle states and transitions of a Java thread.

Identify & describe the labelled (*a* to *o*) states and transitions in Figure 1.

Note you should not re-draw the diagram, but just refer to the labels *a* to *o* in your answer. **[18 marks]**

(c) Within the Java Virtual Machine (JVM) threads use *low-level* actions to interact with the main memory, i.e., transferring values of variables between the *main memory*, the thread's *working copy* and the thread's *execution engine*. Describe these *low-level* actions. **[7 marks]**

Question 2

(a) Briefly describe the main features of the concurrent programming language mechanism known as a *monitor*, as described by C.A.R. Hoare in his 1974 paper. **[5 marks]**

(b) Describe in detail Java's implementation of the *monitor* mechanism. Your answer should be illustrated by fragments of Java code. **[16 marks]**

(c) Describe the main differences between the "classic" *monitor* mechanism as presented by C.A.R. Hoare and Java's version of the monitor mechanism. **[5 marks]**

(d) In Appendix A there is a Java program which provides a simple simulation of sending a *SMS text message* to a mobile phone.

With reference to this program describe in detail the sequence of states of the object `suesphone` and the threads `jim` and `sue` during its execution; assuming that `sue` calls `suesphone`'s `readtext()` method before `jim` calls its `sendtext()` method. **[7 marks]**

Question 3

(a) Explain the concept of a *design pattern*. How do design patterns assist in developing concurrent object-oriented programs? **[7 marks]**

(b) Describe the Java language features *interfaces* and *abstract classes*, and how they assist in the construction of design patterns. **[8 marks]**

(c) Describe the main aims of the *safety preservation* design patterns for concurrent object-oriented programs. Give a brief description of the strategies of three safety design patterns. **[3 marks]**

(d) Describe the overall approach of the *Immutability* design pattern, and the six design steps that are used when applying the *Immutability* design pattern to develop a class. **[7 marks]**

(e) Apply the *Immutability* design pattern to produce a Java class called `OrderedPair`, that can be used to represent an ordered pair of characters, for example (a, A), using the Java type `char`. In addition it should have accessors for each element of the pair called `first` and `second`. **[8 marks]**

Question 4

(a) **(i)** When considering the *correctness* of a concurrent program in terms of desirable properties, we classify these properties as being either a *"safety"* or *"liveness"* property. Explain what is meant by *"safety"* and *"liveness"* properties, and give an example of each. **[3 marks]**

(ii) Describe the ways in which a Java thread can fail to be live. **[5 marks]**

(b) Many design patterns for achieving liveness are based on the technique known as *"splitting synchronization"*. One technique for doing this is known as *lock splitting*, describe this approach. **[7 marks]**

(c) The Java `Rectangle` class is given in Appendix B. This class provides the basis for displaying (i.e., moving and resizing) rectangular shapes, that for example, could be used to implement terminal windows.

You can assume that `moveRectangle()` never deals with its dimensions and `resizeRectangle()` never deals with its location.

The `Rectangle` class is seen as inefficient, i.e., has poor *liveness* characteristics.

(i) Produce a new version with improved performance, by applying the *lock splitting* liveness design pattern.

Note: you only need to give appropriate annotated code fragments which illustrate the main features, **not** a complete program. **[10 marks]**

(ii) Give an annotated diagram which illustrates the structure of your "lock splitting" version of the `Rectangle` class. The diagram should include the `LockSplittingRectangle` class and any new classes and any relationships/links between the `LockSplittingRectangle` class and any new classes. **[8 marks]**

Question 5

(a) Describe the features of the concurrent programming mechanism introduced by E. W. Dijkstra in the late 1960s, known as a *semaphore*. **[5 marks]**

(b) Write a Java class that operates as a general semaphore. The semaphore class' "constructor" should have two arguments:

- the initial value of the semaphore; and
- the upper limit of the semaphore.

[10 marks]

(c) Give a brief description of the Producer/Consumer problem. What types of semaphores are required to solve the Producer/Consumer problem. In addition explain the purpose of each semaphore. **[5 marks]**

(d) Using your semaphore class modify the version of the Producer/Consumer program given in Appendix C, so that it is a safe and live solution of the problem.

Note: you do not have to copy out the whole program, but only those parts that are sufficient to indicate which parts you have modified; you may also make use of the line numbers to help you do this. **[13 marks]**

Appendix A

Program Code for Question 2(d)

The program comprises four classes: `Texter`, `Recipient`, `MobilePhone` and `SMS`.

```
class Texter extends Thread
{
  private final MobilePhone  friendsphone ;

  public Texter( MobilePhone phone )
  {
      friendsphone = phone ;
  }

  public void run()
  {
     String myTxt = new String("Gd Luk. Spk l8r.") ;
     friendsphone.sendtext( myTxt );
  }
}

class Recipient extends Thread
{
   private final MobilePhone myphone ;

   public Recipient( MobilePhone phone )
   {
       myphone = phone ;
   }

   public void run()
   {
     String textmessage = myphone.readtext();
   }
}
```

[Continued Overleaf]

```
class MobilePhone
{
  private String  textmessage = null;
  private boolean got_message = false;

 public synchronized void sendtext (String message)
 {
     while ( got_message ) {
        try {
             wait();
        } catch(InterruptedException e){ }
     }
     textmessage = message ;
     got_message = true ;
     notify();
 }

  public synchronized String readtext()
  {
     while ( !got_message ) {
        try {
             wait();
        } catch(InterruptedException e){ }
     }
     got_message = false ;
     notify() ;
     return textmessage ;
  }
}

class SMS
{
  public static void main(String args[])  {
      MobilePhone  suesphone = new MobilePhone();
      Texter       jim       = new Texter( suesphone );
      Recipient    sue       = new Recipient( suesphone );

      jim.start();
      sue.start();
    }
}
```

Appendix B

Program Code for Question 4 consisting of the Rectangle class.

```
public class Rectangle
{
  protected double x = 0.0;
  protected double y = 0.0;

  protected double width = 0.0;
  protected double height = 0.0;

  public synchronized double x() {  return x;  }

  public synchronized double y() {  return y;  }

  public synchronized double width()  {  return width;   }

  public synchronized double height() {  return height;  }

  public synchronized void adjustLocation()
  {
    x = Calculation1();
    y = Calculation2();
  }

  public synchronized void adjustDimensions()
  {
    width  = Calculation3();
    height = Calculation4();
  }

  protected double Calculation1() {  ...  }
  protected double Calculation2() {  ...  }
  protected double Calculation3() {  ...  }
  protected double Calculation4() {  ...  }

}
```

Appendix C

Program Code for Question 5(d)

Consisting of four classes `Buffer`, `Producer`, `Consumer` and `ProducerConsumer`.

```
class Buffer
{
  public Object[] buffer = null ;
  public final int size ;
  public int in  = 0 ;
  public int out = 0 ;

  public Buffer( int size ) {
    this.size = size ;
    buffer    = new Object[size] ;
  }
}

class Producer extends Thread
{
  private Buffer b = null ;

  public Producer( Buffer buff ) {  b = buff ;  }

  public void run() {
    for (int i = 0; i < 10; i++ ) {
      b.buffer[b.in] = produceItem() ;
      b.in = (b.in + 1) % b.size ;
    }
  }

  private Object produceItem() {  // produce item  }
}
```

[Continued Overleaf]

```
class Consumer extends Thread
{
  private Buffer b    = null ;
  private Object item = null ;

  public Consumer( Buffer buff ) {  b = buff ;  }

  public void run() {
    for (int i = 0; i < 10; i++ ) {
      item = b.buffer[b.out] ;
      b.buffer[b.out] = null ;
      b.out = (b.out + 1) % b.size ;

      consumeItem(item);
    }
  }

  private void consumeItem(Object item) { // consume item  }
}

class ProducerConsumer
{
  private static final int N = 5 ;

  public static void main( String args[] )
  {
    Buffer buffer = new Buffer(N) ;

    Thread p = new Producer(buffer) ;
    Thread c = new Consumer(buffer) ;

    p.start() ;
    c.start() ;
  }
}
```