SCHOOL OF ELECTRONICS AND COMPUTER SCIENCE

2009-2010

Code: 3SFE605 **Level:** 6

Title: Concurrent Programming

Date: 13 May 2010

Time: 14:00

Duration: 2 Hours

INSTRUCTIONS TO CANDIDATES

Answer THREE questions

Each question is worth 33 Marks

Question 1

(a) The Java programming language uses *threads* to provide concurrency. Give the definition of a *thread.* **[5 marks]**

(b) During the life-cycle of a Java (JDK 1.6 or above) thread, it can be in any one of several states. Describe these possible states and what causes a Java thread to change state. Your answer should be illustrated by a diagram and suitable fragments of Java code. **[18 marks]**

(c) With reference to the program given in Appendix A, briefly describe the Java scheduling algorithm, and what rôle thread priority plays in this. Explain how the Java scheduler would schedule the `Racer` threads, and give an example of the output that could be produced. **[10 marks]**

Question 2

(a) Describe the concurrent programming concept known as a *monitor.*

In addition, describe in detail how the Java programming language has implemented the monitor concept. Your answer should be illustrated by fragments of Java code.

Finally, comment on how well you think Java has implemented the monitor concept. **[20 marks]**

(b) With reference to the Java program given in Appendix B. Describe in detail the sequence of states of the object `sv` and the threads `w` and `r` during the following scenarios.

(i) From their creation to their termination, when `w` calls `sv`'s `assign` method before `r` calls its `read` method. **[8 marks]**

(ii) Assume the object and threads have been created and started as in part **(i)**, that `r` has just started executing `sv`'s `read` method and that `w` is blocked trying to execute the `assign` method.

Then describe in detail the sequence of states from this point to the thread's termination. **[5 marks]**

Question 3

(a) Give a brief definition of the concept of a *design pattern*. How do design patterns improve code reuse and address the problem of "hacking" in the development of concurrent object-oriented programs? **[8 marks]**

(b) Give reasons for why the Java "`interface`" feature is so useful when developing software using the *design patterns* approach. **[7 marks]**

(c) Give a brief description of the *Full Synchronization* design pattern, and its four design steps. **[9 marks]**

(d) The Java code for a basic `Stack` class is given in Appendix C.

Apply the *Full Synchronization* design pattern, with *"balking"*, to this class to produce a `BalkingStack` class.

Its requirements are:

- The `push(item)` method pushes an object onto the stack, if the stack is full then the method "balks".
- The `pop()` method removes the object at the top of the stack and returns it, if the stack is empty then the method "balks".

Note: you do not need to give the complete code for your `BalkingStack` class, it is acceptable to give just the modifications to the `Stack` class and any new classes if required. **[9 marks]**

Question 4

(a) Explain why there is a need to "balance *safety* and *liveness* requirements" when applying design patterns to concurrent object-oriented programs. Discuss the role that synchronization plays in achieving this balance. **[8 marks]**

(b) Many design patterns for achieving liveness are based on the technique known as *splitting synchronization*. In practice this technique is achieved by applying the technique know as *splitting classes*; describe this approach. **[9 marks]**

(c) **(i)** The `Cuboid` class given in Appendix D, provides the basis for representing simple *cuboids* (i.e., rectangular boxes or cubes) for a graphics application. (Assume that `moveCube(offset)` never deals with its dimensions and `resizeCube(amout)` never deals with its location.)

The `Cuboid` class is seen as inefficient, i.e., has poor *liveness* characteristics. With the aim of improving performance, produce a new version `ClassSplittingCuboid` by applying the *splitting classes* 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 "class splitting" version of the `Cuboid` class. The diagram should include the `ClassSplittingCuboid` class and any new classes and any relationships/links between the `Cuboid` class and any new classes. **[6 marks]**

Question 5

(a) Describe the features of the *semaphore* concurrent programming mechanism. **[7 marks]**

(b) What are the advantages and disadvantages of using semaphores? Explain why *monitors* were considered to be "better" than semaphores? **[6 marks]**

(c) Given the following Java `interface` for a semaphore:

```
public interface Semaphore
{
     public void claim() ;
     public void release() ;
}
```

Write a Java class called `BinarySemaphore`, that implements this semaphore interface and operates as a *binary* semaphore. **[12 marks]**

(d) Using your Java binary semaphore class `BinarySemaphore` (from part **(c)**) show, by giving suitable code fragments, how it could be used to achieve the *mutual exclusion* of a critical section by two threads. **[8 marks]**

Appendix A

Program for Question 1: consisting of two classes Racer and RaceStarter.

```
1   class Racer extends Thread
2   {
3     Racer(int id)
4     {
5       super( "Racer[" + id + "]" ) ;
6     }
7
8     public void run()
9     {
10      for ( int i = 1; i < 40; i++ )  {
11        if ( i % 10 == 0 )
12        {
13          System.out.println( getName() + ", i = " + i ) ;
14          yield();
15        }
16      }
17    }
18  }
19
20  class RaceStarter
21  {
22    public static void main( String args[] )
23    {
24      Racer[] racer = new Racer[4] ;
25
26      for ( int i = 0; i < 4; i++ )  {
27          racer[i] = new Racer(i) ;
28      }
29
30      racer[0].setPriority(8);
31      racer[3].setPriority(3);
32
33      for ( int i = 0; i < 4; i++ )  {
34          racer[i].start() ;
35      }
36    }
37  }
```

Appendix B

Program for Question 2: comprises four classes: `SharedVar`, `Writer`, `Reader` and `System`.

```
1    class SharedVar
2    {
3      private int contents;
4      private boolean new_value = false;
5
6      public synchronized int read()
7      {
8         while ( !new_value )
9         {
10           try {  wait();  }
11           catch(InterruptedException e){ }
12        }
13        new_value = false;
14        notifyAll();
15        return contents;
16     }
17
18     public synchronized void assign(int value)
19     {
20        while ( new_value )
21        {
22           try {  wait();  }
23           catch(InterruptedException e){ }
24        }
25        contents = value;
26        new_value = true;
27        notifyAll();
28     }
29   }
```

[Continued Overleaf]

```
30  class Writer extends Thread
31  {
32    private SharedVar sharedvar;
33
34    public Writer(SharedVar sv) {  sharedvar = sv;  }
35
36    public void run() {  sharedvar.assign(1);  }
37  }
38
39
40  class Reader extends Thread
41  {
42     private SharedVar sharedvar;
43     private int value;
44
45     public Reader(SharedVar sv) {  sharedvar = sv;  }
46
47     public void run() {  value = sharedvar.read();  }
48  }
49
50
51  class System
52  {
53    public static void main(String args[])
54    {
55        SharedVar sv = new SharedVar();
56        Thread w = new Writer(sv);
57        Thread r = new Reader(sv);
58
59        w.start();
60        r.start();
61    }
62  }
```

Appendix C

Program for Question 3: comprises one class Stack.

```
1   public class Stack
2   {
3     public final Object[] stack   ;
4     public final int      maxsize ;
5     public       int      top     ;
6
7     public BalkingStack( int maxsize )
8     {
9           stack = new Object[maxsize] ;
10          this.maxsize = maxsize ;
11          top = -1;
12    }
13
14    public void push( Object item )
15    {
16          if ( top == maxsize - 1 )
17              System.out.println( "Error: stack full" ) ;
18          else
19              stack[++top] = item ;
20    }
21
22    public Object pop()
23    {
24      if ( top == -1 )
25         System.out.println( "Error: stack empty" ) ;
26      else
27          return stack[top--] ;
28    }
29  }
```

Appendix D

Program Code for Question 4 comprises the Cuboid class.

```
1   public class Cuboid
2   {
3     protected double x = 0.0;
4     protected double y = 0.0;
5     protected double z = 0.0;
6
7     protected double width  = 0.0;
8     protected double height = 0.0;
9     protected double depth  = 0.0;
10
11    public synchronized double x() { return x; }
12    public synchronized double y() { return y; }
13    public synchronized double z() { return z; }
14
15    public synchronized double width()  { return width; }
16    public synchronized double height() { return height; }
17    public synchronized double depth()  { return depth; }
18
19    public synchronized void moveCuboid(int offset)
20    {
21      x = calcX(offset);
22      y = calcY(offset);
23      z = calcZ(offset);
24    }
25
26    public synchronized void resizeCuboid(int amount)
27    {
28      width  = calcWidth(amount);
29      height = calcHeight(amount);
30      depth  = calcDepth(amount);
31    }
32
33    protected double calcX(int offset) {  // ...  }
34    // etc
35  }
```