CAVENDISH CAMPUS

School of Electronics and Computer Science

Modular Undergraduate Programme
First Semester 2012 – 2013

Referred/Deferred EXAM

Module Code: ECSE603

Module Title: Concurrent Programming

Date: July 2013

Time: 10:00 – 12:00

Instructions to Candidates:

Answer THREE questions.
Each question is worth 33 marks.

© University of Westminster 2013

Question 1

(a) When modelling concurrent systems and processes using the abstract Finite State Process (FSP) language, it is necessary to take an *abstract* view of a processes. Describe this FSP abstract view of a process. **[4 marks]**

(b) Given the following FSP process:

```
COUNT ( N = 2 )  =  COUNT[0],

COUNT[ i : 0..N ]
    =  (   when( i < N )  inc -> COUNT[i+1]
         | when( i > 0 )  dec -> COUNT[i-1]
       ).
```

(i) Explain the meaning of the following FSP language features used in this process: "->", "|", "when (i < N)" and "COUNT[i+1]". **[8 marks]**

(ii) With reference to the above process explain the meaning of the following terms:

- *Alphabet*
- *Transition*

[4 marks]

(c) For each of the following FSP processes give the corresponding:

- *Labelled Transition System Graph*
- *Trace Tree.*

(i)

```
P1  = (   a -> b -> c -> STOP
        | d -> e -> f -> STOP ) .
```

[9 marks]

(ii)

```
P2 = (   a -> ( b -> STOP | c -> STOP )
       | d -> e -> STOP ) .
```

[8 marks]

[TOTAL 33]

© University of Westminster 2013

Question 2

The following is a specification of a system consisting of two processes sharing a printer.

- A shared printer called `PRINTER`, that can be used to print a document. It also allows users to cancel the printing of a document.
- A user process called `User`, that repeatedly prints a document. Note that a document is denoted by a `DOC_ID`, which is a number.
- An administrator process called `Admin`, that only services the printer, and does not print.
- All the user and administrator processes that share the printer must obviously have **mutually exclusive** access to it.
- The system consists of the **two** user processes, that print different documents, **one** administrator process and **one** shared printer.

(a) Define three Finite State Process (FSP) language processes to model the `PRINTER`, `User` and `Admin`. **[25 marks]**

(b) Using your three types of processes define a composite process that models the complete system. **[5 marks]**

(c) Briefly explain how you have ensured that the two user processes `User` and the `Admin` process have *mutually exclusive* access to the shared printer `PRINTER`. **[3 marks]**

[TOTAL 33]

© University of Westminster 2013

Question 3

(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. **[9 marks]**

(b) Describe and explain the life-cycle states of a JDK 1.5 Java thread. Your answer should include fragments of Java code which affect the life-cycle and a diagram illustrating the relationships between the states. **[18 marks]**

(c) The Java API provides the **ThreadGroup** class for managing groups of threads. Give a brief explanation of the relationship between threads and thread groups. What "missing" features do you think should be added to the `ThreadGroup` class and why? **[6 marks]**

[TOTAL 33]

Question 4

(a) Briefly describe the concurrent programming concept known as a *monitor*, as proposed by C.A.R. Hoare. **[5 marks]**

(b) Describe in detail how the Java language supports the concept of a monitor. Your answer should be illustrated by fragments of Java code. **[17 marks]**

(c) With reference to the Java program given in Appendix A.

(i) Describe the sequence of states of the object `mb` and the threads `p` and `r` during their execution; assuming that `p` calls the `post()` method **before** `r` calls the `retrieve()` method. **[6 marks]**

(ii) If the `MessageSystem` class created several `Poster` threads and several `Retriever` threads rather than just one of each, deadlock could occur.

Assuming deadlock has occurred, explain what has happened to the `Poster` and `Retriever` threads. What is the simplest change that could be made to the two `MessageBoard` methods `post()` and `retrieve()` that would stop this happening. **[5 marks]**

[TOTAL 33]

© University of Westminster 2013

Question 5

(a) Describe the features of the *semaphore* concurrent programming mechanism. **[9 marks]**

(b) What is the *Dining Philosophers* problem (for 5 Philosophers)? Explain how *deadlock* can occur and how it can be avoided by the use of a *Butler*.

[6 marks]

(c) Assume that you have available a Java class that implements a `Semaphore`; where the constructor has the following form:

```
Semaphore( int max_value, int initial_value )
```

Use this semaphore class to write a Java program for the Dining Philosophers problem which avoids deadlock by using a Butler.

You must provide a `Philosopher` class, which represents the behaviour of a philosopher and a `DiningPhilosopher` class which creates the philosophers, forks and butler.

Note: assume that all threads will execute forever. **[18 marks]**

[TOTAL 33]

© University of Westminster 2013

Appendix A

Program for Question 4: comprises four classes `Poster`, `Retriever`, `MessageBoard` and `MessageSystem`.

```
class Poster extends Thread
{
  private final MessageBoard messageboard;

  public Poster(MessageBoard mb)
  {
      messageboard = mb;
  }

  public void run()
  {
     messageboard.post( new String("Hello mate.") );
     messageboard.post( new String("Good Luck.") );
  }

}


class Retriever extends Thread
{
   private final MessageBoard messageboard;

   public Retriever(MessageBoard mb)
   {
       messageboard = mb;
   }

   public void run()
   {
      Object message = null ;

      message = messageboard.retrieve();
      message = messageboard.retrieve();
   }
}
```

[Continued Overleaf]

© University of Westminster 2013

```
class MessageBoard
{
  private Object  message = null;
  private boolean message_posted = false;

  public synchronized Object retrieve()
  {
     while ( !message_posted ) {
        try {
             wait();
        } catch(InterruptedException e){ }
     }
     message_posted = false;
     notify();
     return message ;
  }

  public synchronized void post(Object new_message)
  {
     while ( message_posted ) {
        try {
             wait();
        } catch(InterruptedException e){ }
     }
     message = new_message;
     message_posted = true;
     notify();
  }
}

class MessageSystem
{
  public static void main(String args[]) {
      MessageBoard mb = new MessageBoard() ;
      Poster        p = new Poster(mb) ;
      Retriever     r = new Retriever(mb) ;

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

© University of Westminster 2013