CAVENDISH CAMPUS

**School of Electronics and Computer Science**

Modular Undergraduate Programme
First Semester 2011 – 2012

**Module Code:** ECSE603

**Module Title:** Concurrent Programming

**Date:** Thursday, 10th May 2012

**Time:** 10:00 – 12:00

**Instructions to Candidates:**

Answer THREE questions.
Each question is worth 33 marks.

## Question 1

**(a)** The abstract Finite State Process (FSP) language is used to describe and model concurrent programs. A FSP program is a collection of *abstract* processes. These FSP programs and processes are modelled by Finite State Machines (FSM) and FSMs are "implemented" by Label Transition System Graphs.

**(i)** Describe the FSP abstract model of a process. **[4 marks]**

**(ii)** Describe a Label Transition System Graph (LTS). **[4 marks]**

**(b)** Given the following FSP process:

```
TIMER( N = 2 ) = COUNTDOWN[ N ] ,

COUNTDOWN[ i : 0..N ]
    = (  when( i > 0 )  tick -> COUNTDOWN[i-1]
       | when( i == 0 ) beep -> STOP
       | stop -> STOP
      ) .
```

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

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

- *Alphabet*
- *Trace* **[4 marks]**

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

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

**(i)**
```
TRAFFICLIGHT = ( red -> orange -> green
                     -> orange
                     -> TRAFFICLIGHT ).
```
**[5 marks]**

**(ii)**
```
FAULTY = (   {red, blue, green} -> FAULTY
           | yellow -> kitkat   -> FAULTY )  .
```
**[8 marks]**

**[TOTAL 33]**

## Question 2

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

- A shared integer variable called `VAR`, that can be read and written to. It is initialized to 0 (zero).
- An incrementing process called `INC`, that repeatedly reads the value of the variable and then updates the variable by adding 1 to its value.
- A decrementing process called `DEC`, that repeatedly reads the value of the variable and then updates the variable by subtracting 1 from its value.
- The two processes that share the variable must have **mutually exclusively** access to it.
- The system consists of the two processes and the shared variable.

**(a)** Using the Finite State Process (FSP) language define three processes to model the processes `VAR`, `INC` and `DEC`. **[26 marks]**

**(b)** Using your three processes define a composite process that models the complete system. **[4 marks]**

**(c)** Briefly explain how you have ensured that the two processes `INC` and `DEC` have *mutually exclusive* access to the shared variable `VAR`. **[3 marks]**

**[TOTAL 33]**

## Question 3

**(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]**

**[TOTAL 33]**

## Question 4

**(a)** Describe the concurrent programming feature 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 two threads `w` and `r` from their creation to their termination, when `r` calls the `read` method **before** `w` calls the `assign` method. **[13 marks]**

**[TOTAL 33]**

## Question 5

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

**(b)** What are the advantages and disadvantages of using semaphores? Explain why *monitors* are generally considered to be "better" than semaphores? **[7 marks]**

**(c)** **(i)** Assuming that you have available a Java `Semaphore` class, that implements a semaphore.

Give suitable Java code fragments to illustrate how semaphores can be used to achieve *mutual exclusion* of a critical section by two Java threads. **[14 marks]**

**(ii)** With reference to your code given in answer to part **(i)**; explain how you have used the semaphore mechanism to achieve mutual exclusion of the critical section. **[3 marks]**

**[TOTAL 33]**

## Appendix A

**Program for Question 3:** consisting of two classes Racer and RaceStarter.

```
class Racer extends Thread
{
  Racer(int id)
  {
    super( "Racer[" + id + "]" ) ;
  }

  public void run()
  {
    for ( int i = 1; i < 40; i++ )  {
      if ( i % 10 == 0 )
      {
        System.out.println( getName() + ", i = " + i ) ;
        yield();
      }
    }
  }
}

class RaceStarter
{
  public static void main( String args[] )
  {
    Racer[] racer = new Racer[4] ;

    for ( int i = 0; i < 4; i++ )  {
        racer[i] = new Racer(i) ;
    }

    racer[0].setPriority(8);
    racer[3].setPriority(3);

    for ( int i = 0; i < 4; i++ )  {
        racer[i].start() ;
    }
  }
}
```

## Appendix B

**Program for Question 4:** comprises four classes: `SharedVar`, `Writer`, `Reader` and `System`.

```
class SharedVar
{
  private int contents;
  private boolean new_value = false;

  public synchronized int read()
  {
     while ( !new_value )
     {
        try {  wait();  }
        catch(InterruptedException e){ }
     }
     new_value = false;
     notifyAll();
     return contents;
  }

  public synchronized void assign(int value)
  {
     while ( new_value )
     {
        try {  wait();  }
        catch(InterruptedException e){ }
     }
     contents = value;
     new_value = true;
     notifyAll();
  }
}
```

**[Continued Overleaf]**

```
class Writer extends Thread
{
  private SharedVar sharedvar;

  public Writer(SharedVar sv) {  sharedvar = sv;  }

  public void run() {  sharedvar.assign(1);  }
}


class Reader extends Thread
{
   private SharedVar sharedvar;
   private int value;

   public Reader(SharedVar sv) {  sharedvar = sv;  }

   public void run() {  value = sharedvar.read();  }
}


class System
{
  public static void main(String args[])
  {
      SharedVar sv = new SharedVar();
      Thread w = new Writer(sv);
      Thread r = new Reader(sv);

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