CAVENDISH CAMPUS

School of Informatics

Modular Undergraduate Programme
First Semester 2007 – 2008

Module Code: 3SFE605

Module Title: Concurrent Programming

Date: 21st May 2008

Time: 14:00 – 16:00

Instructions to Candidates:

Answer THREE questions.
Each question is worth 33 marks.

Question 1

(a) In Java concurrent programming is achieved by using *threads*. Describe the concept of a *thread*. **[5 marks]**

(b) A Java thread throughout its existence is always in one of several *thread states*. Describe these *thread states* for a Java JDK 1.5 (or later version) thread. In addition, explain how a Java thread changes state and give Java code fragments that produce the state changes. Your answer should include a diagram of these states and the transitions between them. **[20 marks]**

(c) Briefly describe the Java scheduling algorithm, and what rôle thread priority plays in this. Further, explain how the Java scheduler would schedule the three `racer` threads defined in the program given in Appendix A. Finally, give an example of the output that the program could produce. **[8 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 B 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) 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? **[7 marks]**

(b) The Java programming language provides the "`interface`" facility. Briefly, explain what this is and how it facilitates the develop of software using *design patterns*. **[7 marks]**

(c) What is the general aim of the *safety preservation* design patterns for ensuring a safe concurrent object-oriented program. Explain how the *Full Synchronization* safety design pattern attempts to achieve this aim, by describing its four design steps. **[9 marks]**

(d) Apply the *Full Synchronization* design pattern, with *"balking"*, to produce a Java `BalkingStack` class which represents a stack of `Objects`. Its requirements are:

- A stack of objects, the `maxsize` being specified as a parameter of the constructor.
- A `push(item)` method which pushes an object onto the stack, if the stack is full then the method "balks".
- A `pop()` method which removes the object at the top of the stack and returns it, if the stack is empty then the method "balks".

[10 marks]

Question 4

(a) What type of systems are *flow* design patterns applicable to? Give two examples of these types of systems. **[5 marks]**

(b) Describe the following types of *components* that are used in flow design patterns:

(i) Representational Components **[8 marks]**

(ii) Transformational Components **[5 marks]**

(iii) Coordination Components **[3 marks]**

(c) Assuming that the following classes have already been defined:

- `ProducerStage` – the class of the producer.
- `ConsumerStage` – the class of the consumer, which has a `put(item)` method, used to push data to the consumer.
- `Item` – the class of items placed into the buffers.
- `Buffer` – the class which provides the underlying buffer data structure, and has public methods `put` and `take`.

(i) Using the above classes, construct a *Put-Only Buffer* class that uses a thread to *push* data to a *consumer* stage. **[8 marks]**

(ii) Describe the sequence of method calls involved in an *item* of data flowing from a Producer stage via the *Put-Only Buffer* to a Consumer stage. **[4 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 1(c)

The program comprises 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 racer1 = new Racer(1);
    Racer racer2 = new Racer(2);
    Racer racer3 = new Racer(3);

    racer1.setPriority(7);

    racer1.start();
    racer2.start();
    racer3.start();
  }
}
```

Appendix B

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()
  {
     friendsphone.sendtext( new String("Spk l8r.") );
  }
}

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 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() ;
  }
}
```