CAVENDISH CAMPUS

School of Electronics and Computer Science

Modular Undergraduate Programme
Second Semester 2012 – 2013

Module Code: ECSE603

Module Title: Concurrent Programming [MARKING SCHEME]

Date: Tuesday, 7th May 2013

Time: 10:00 – 12:00

Instructions to Candidates:

Answer THREE questions.
Each question is worth 33 marks.

Question 1

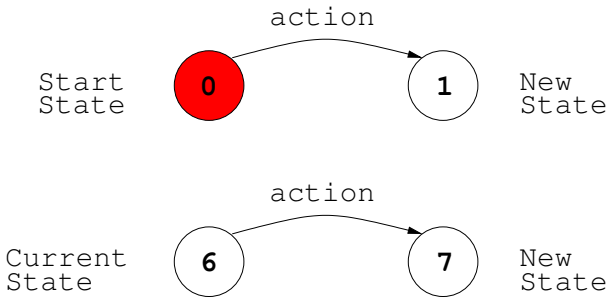
(a) The FSP view of an *abstract process*:

- Ignores the details of *state representation* and *program/machine instructions*. [**1 mark**]
- Simply considers a process as having a *state* modified by indivisible or atomic *actions*. [**1 mark**]
- Each action causes a *transition* from the current state to the next state. [**1 mark**]
- The order in which actions are allowed to occur is determined by a *FSM/LTG* that is an abstract representation of the program. [**1 mark**]

A Label Transition System Graph (LTS) use the following diagrammatic conventions:

- *Initial state* is always numbered 0.
- *Transitions* are always drawn in a clockwise direction.

[**1 mark**]



[**2 marks**]

This form of state machine description is known as a *Labelled Transition System (LTS)*, since transitions are labelled with action names. [**1 mark**]

[**PART Total 8**]

(b) **(i)** Meaning of FSP language features.

- `->` is action prefix.
  If `x` is an action and `P` a process then the action prefix `(x -> P)` describes a process that initially engages in the action `x` and then behaves as `P`. **[2 marks]**
- `|` is the choice operator.
  If `x` and `y` are actions; `P` & `Q` are processes then `(x -> P | y -> Q)` describes a process which initially engages in either of the actions `x` or `y`. After the first action has occurred, the subsequent behaviour is described by `P` if the first action was `x` and `Q` if the first action was `y`. **[2 marks]**
- `when( i == 0 )` is a guard for a guarded action.
  The choice `(when (B) x -> P | y -> Q)` means that when the *boolean guard* `B` is *true* the actions `x` and `y` are both eligible to be chosen; *false* the action `x` cannot be chosen. **[2 marks]**
- `STOP` is a process.
  `STOP` is a special predefined process that engages in no further actions, it is used to represent the *"deadlocked"* process. **[2 marks]**

[SUBPART Total 8]

(ii)

- *Transition* – a transition occurs when a process moves from one state to another state by means of performing an action. **[1 mark]**
  For example, in the `MINUTE_ALARM` process, transition occur when it performs either of its actions and moves to the next state, e.g. preforming `sound.alarm` moves it to the final `STOP` state. **[1 mark]**
- *Trace* – the sequence of actions produced by the execution of a process (or set of processes). **[1 mark]**
  Some of the traces of the `MINUTE_ALARM` process are:

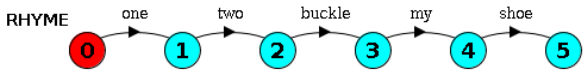
```
traces( MINUTE_ALARM )
    = { <>, <tick>, <tick,tick>, <tick,tick,tick>,
         <tick, ..., tick,sound.alarm> }
```

  [1 mark]

[SUBPART Total 4]

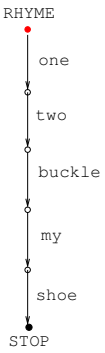
[PART Total 12]

(c) **(i)** RHYME state machine:



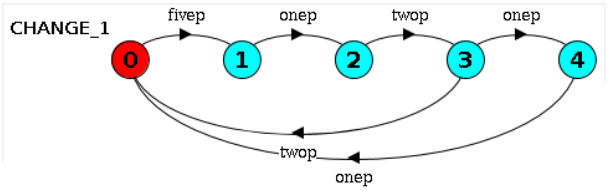
[**2 marks**]

RHYME trace tree.



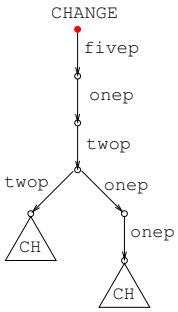
[**2 marks**]

(ii) CHANGE state machine.



[**5 marks**]

CHANGE trace tree.



[**4 marks**]

[**PART Total 13**]

[**QUESTION Total 33**]

Question 2

(a) The FSP processes.

The required declarations.

```
const MAXINT = 2

range INT = 0 .. MAXINT
```

[**1 mark**]

Sets of Actions & Process Labels.

```
set VAR_Actions = { read[INT], write[INT], acquire, release }

set PROCS = { inc, dec }
```

[**4 marks**]

The shared variable VAR.

```
VAR = Unlocked_VAR[ 0 ] ,

Unlocked_VAR[ v : INT ] =  ( acquire -> Locked_VAR[ v ] ) ,

Locked_VAR[v : INT ]
      = (   read[ v ]          -> Locked_VAR[ v ]
          | write[ nv : INT ] -> Locked_VAR[ nv ]
          | release            -> Unlocked_VAR[ v ] ) .
```

`VAR` [**1 mark**] , `Unlocked_VAR` [**3 marks**] , `Locked_VAR` [**7 marks**]
.

The two processes.

```
INC = ( acquire ->
           read[ v : INT ] -> write[ v + 1 ] ->
        release -> INC ) +VAR_Actions .
```

[**5 marks**]

```
DEC = ( acquire ->
  read[ v : INT ] -> write[ v - 1 ] ->
release -> DEC ) +VAR_Actions .
```

[**5 marks**]

[**PART Total 26**]

(b) The complete systems

```
|| Shared_VAR = ( PROCS :: VAR || inc : INC || dec : DEC ) .
```

[**4 marks**]

[**PART Total 4**]

(c) Mutual exclusive access to the shared variable `VAR` is achieved by requiring the two processes "lock" it before they can access it, by performing an `acquire` action & when they have finished they must "unlock" it, by performing a `release` action. [**3 marks**]

[**PART Total 3**]

[**QUESTION Total 33**]

Question 3

(a) The two methods are:

- Subclass the `Thread` class and override the `run()` method, [**1 mark**]

  Code Example:

```
class SimpleThread extends Thread
{
   // ‘‘Thread( String )’’ constructor
   public SimpleThread(String str){  super(str);  }

   public void run(){  // ‘‘body’’ of the thread  }
}
```

[**2 marks**]

- Provide a class that implements the `Runnable` interface by defining its own `run()` method. [**1 mark**]

  Then create the thread using a `Thread` constructor that takes a reference to an instance of a `Runnable` object. [**1 mark**]

  Code example:

```
class SubClass extends SuperClass implements Runnable
{
   private Thread t ;

   public SubClass()
   {
     t = new Thread(this) ;
     t.start();
   }

   // ‘‘run()’’ method used by t
   public void run(){ ... }
}
```

  [**2 marks**]

Decision: create a thread using the `Runnable` interface if your class needs to subclass some other class, e.g. `Applet`; otherwise use sub-classing. [**1 mark**]

[**PART Total 8**]

(b) Label and describe the states and transitions labelled in Figure 1. (Note that the order in which the three non-runnable states blocked, waiting & timed_waiting is irrelevant, as long as the entry & exits are labelled correctly.)

(a) A call of `new` creates a new thread, but it is not started, i.e.,

```
Thread myThread = new MyThreadClass();
```

[**1 mark**]

(b) `NEW` state, in this state a thread is an empty `Thread` object, no system resources have been allocated for it yet. [**1 mark**]

(c) A call of `Thread.start()`, creates the system resources necessary to run the thread, schedules it to run, and calls its `run()` method. [**1 mark**]

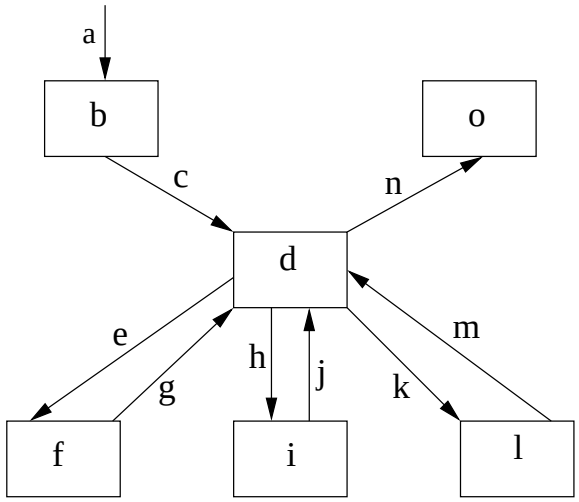


Figure 1: The life-cycle states and transitions of a Java thread.

(d) `RUNNABLE` state. This state is called *"Runnable"* rather than *"Running"* because the thread may not be running when in this state, since there may be only a single processor. When a thread is running it's *"Runnable"* and is the current thread & its `run()` method is executing sequentially. **[2 marks]**

(e) *"failed to acquire lock"*. A thread has attempted & failed to acquire a synchronization (monitor) lock. **[1 mark]**

(f) `BLOCKED` state. This state is for a thread blocked waiting for a monitor lock, this means that another thread currently has *"acquired"* or *"holds"* the lock & therefore the thread can not proceed & is blocked. **[1 mark]**

A thread in the blocked state is waiting for a monitor lock so that it can either: **enter** a `synchronized` block/method; or **reenter** a `synchronized` block/method after calling `Object.wait`. **[1 mark]**

(g) *"lock released"*. The thread that was holding the synchronization (monitor) lock has released it. **[1 mark]**

(h) "`wait(), join()`". A call of either (with no timeout parameter), moves the thread to the `WAITING` state. **[1 mark]**

(i) `WAITING` state. A thread in this state is waiting for another thread to perform a particular action to allow it to return to the `RUNNABLE` state. **[1 mark]**

(j) "`notify()` or `notifyAll()` or `otherthread` to terminate". A thread that called `wait()` on an object & is waiting for another thread to call `notify()` or `notifyAll()` on that object. A thread that called `otherthread.join()` is waiting for the thread `otherthread` to terminate. **[2 marks]**

(k) "`wait(t)` or `join(t)` or `sleep(t)`". A thread that called any of these would move from the `RUNNABLE` state to the `TIMED_WAITING` state. **[1 mark]**

(l) `TIMED_WAITING` state. Threads in this state are waiting either for another thread to perform the appropriate actions, i.e., `notify()` or `notifyAll()` or terminate, or for the specified time `t` to elapse. **[1 mark]**

(m) "`notify()` or `notifyAll()` or `otherthread` terminates or time is up". A timed version of (j), i.e., a call of `wait(t)` is waiting for a call of `notify()` or `notifyAll()` or for time `t` to have elapsed. Similarly, a call of `otherthread.join(t)` for either `otherthread` terminates or time has elapse. For a call to `sleep(t)` time `t` to elapse. **[1 mark]**

(n) "`run()` method exits". When this happens the thread moves from the `RUNNABLE` state to the `TERMINATED` state. **[1 mark]**

(o) `TERMINATED` state. When a thread dies from "natural causes", i.e., a thread dies naturally when its `run()` method exits normally, it moves into this state. **[1 mark]**

[PART Total 18]

(c) The interaction of threads with the Java Virtual Machine (JVM) main memory, and thus with each other, is in terms of the following low-level *atomic (indivisible)* actions:

Thread actions:

- `use` transfers the contents of the thread's working copy of a variable to the thread's execution engine. **[1 mark]**
- `assign` transfers a value from the thread's execution engine into the thread's working copy of a variable. **[1 mark]**
- `load` puts a value transmitted from main memory by a `read` action into the thread's working copy of a variable. **[1 mark]**
- `store` transmits the contents of the thread's working copy of a variable to main memory for use by a later `write` operation. **[1 mark]**

Main Memory actions

- `read` transmits the contents of the master copy of a variable to a thread's working memory for use by a later `load` operation. [**1 mark**]
- `write` puts a value transmitted from the thread's working memory by a `store` action into the master copy of a variable in main memory. [**1 mark**]

Thread & Main Memory "Tightly Synchronized" actions: `lock` causes a thread to acquire one claim on a lock, & `unlock` causes a thread to release one claim on a lock. [**1 mark**]

[**PART Total 7**]

[**QUESTION Total 33**]

Question 4

(a) A monitor is a structuring device, which encapsulates a resource only allowing access via a controlled interface of synchronized procedures/methods that behave like critical sections, i.e., mutually exclusive execution. **[2 marks]**

The main components are: *permanent variables* that represent the state of the resource; *methods* that implement operations on the resource; and a *monitor body* that is executed when the monitor is started. **[3 marks]**

[PART Total 5]

(b) Java's monitors are built into the definition of the `Object` class.

```
class ObjMonitor{
  private Object data;
  private boolean condition;
  public void ObjMonitor(){
     data = ... ;
     condition = ... ;
  }
  public synchronized object operation1() {
     while (!condition) {
        try { wait(); } catch(InterruptedException e){ }
     }
     // do operation1
     condition = false;
     notifyAll();
     return data;
  }
  public synchronized void operation2() {
     ...
  }
}
```

[4 marks]

Monitor Data: a monitor usually has two kinds of private variables, e.g., `ObjMonitor` – `data` which is the encapsulated data/object, & `condition` which indicates the correct state of `data` to perform the operations. **[2 marks]**

Monitor Body: the monitor body in Java is provided by the object's constructors, e.g. `ObjMonitor()`. [**1 mark**]

Monitor methods: identified by `synchronized`, e.g. `operation1()`. When control enters a `synchronized` method, the calling thread acquires the monitor, e.g., `ObjMonitor`, preventing other threads from calling any of `ObjMonitor`'s methods. When `operation1()` returns, the thread releases the monitor thereby unlocking `ObjMonitor`. [**2 marks**]

The acquisition and release of a monitor is done *automatically* and *atomically* by the Java runtime system. [**1 mark**]

`notifyAll()`**:** wakes up *all* the threads waiting on the monitor held by the current thread. Awakened threads compete for the monitor, one thread gets the monitor and the others go back to waiting. `notify()` just wakes up 1 of the waiting threads. [**1 mark**]

`wait()`**:** is used with `notifyAll()` & `notify()` to coordinate the activities of multiple threads using the same monitor. `wait(long ms)` & `wait(long ms, int ns)` wait for notification or until the timeout period has elapsed, milliseconds & nanoseconds. [**1 mark**]

`wait(ms)` causes the current thread to be placed in the *wait set*, for this object and then relinquishes the monitor lock. [**1 mark**]

The thread is blocked (not scheduled) until: another thread calls `notify()` or `notifyAll()` for this object and the thread is chosen as the thread to be awakened; or the specified amount of real time has elapsed. [**1 mark**]

The thread is then removed from the wait set for this object and re-enabled for thread scheduling. [**1 mark**]

Once it has gained control of the object, all its synchronization claims on the object are restored to the situation before the `wait`, & it then returns from the invocation of the `wait` method. [**1 mark**]

[**PART Total 16**]

(c)

- Hoare monitors not set in an OO context, just a new language construct, unlike Java's which are built into the definition of the `Object` class. Java monitors are used by combining built in class methods & `synchronised` method modifier. [**1 mark**]
- Hoare monitors could have several condition variables associated with each monitor, Java uses the lock associated with every object. Thus Hoare monitors can allow for more fine grained control of access to

the monitor. This can only be achieved in Java by the programmer introducing extra objects to act as "condition variable" locks & use the `synchronised(object){...}` statement. [**2 marks**]

- Hoare's `wait` & `signal` constructs were parameterized by a condition variable, e.g., `wait(condvar)` (See above). Java's `wait` & `notifyAll` are not since they only apply to the monitor object's lock. [**1 mark**]
- Hoare monitor's procedure visibility & initialization are provided by Java's `public`, `private` method modifiers & the class constructor respectively. [**1 mark**]

[**PART Total 5**]

(d)

1. Initially the monitor `billsPhone` is unlocked & its wait-set is empty. `bill` & `jim` are created in the NEW state, & then started, i.e., made RUNNABLE. [**1 mark**]
2. Then `bill` calls `readtext()` & acquires `billsPhone`'s lock & as a result `jim` would enter the BLOCKED state. [**1 mark**]
3. Then `bill` finds that `got_message` is `false`, then `wait` is called, this causes `bill` to be placed into `billsPhone`'s wait-set, it enters the WAITING state & releases `billsPhone`'s lock. [**1 mark**]
4. Once `bill` releases the lock `jim` leaves the BLOCKED state & re-enters the RUNNABLE state & can acquire `billsPhone`'s lock. [**1 mark**]
5. `jim` can complete the `sendtext` call, at the end it calls `notifyAll()` & finally releases `billsPhone`'s lock, finishes & enters the TERMINATED state. [**1 mark**]
6. `bill` is woken up by `jim`'s `notifyAll()`, so `bill` is removed from the wait-set & moves from the WAITING to the RUNNABLE state. [**1 mark**]
7. When `bill` re-acquires the `billsPhone`'s lock, since `got_message` is `true` it can complete its `read` call & finally releases the lock, finishes & enters the TERMINATED state. [**1 mark**]

[**PART Total 7**]

[**QUESTION Total 33**]

Question 5

(a) A semaphore s is an integer variable which can only take non–negative values, ($s \geq 0$). There are two types of semaphores: binary semaphores (0 or 1) and general semaphores ($n \geq s \geq 0$). **[1 mark]**

Description of semaphore operations: `claim(s)`, `release(s)` & `initialize(s, v)` **[3 marks]**

The operations `claim(s)` & `release(s)` are primitive, i.e. non–overlapping & atomic . Thus they can not be executed simultaneously, only serially. **[1 mark]**

[PART Total 5]

(b) Something similar to the following is expected, but alternatives will be accepted.

```
public class Semaphore
{
  private       int semaphore = 0 ;
  private final int MAX_VALUE ;

  public Semaphore( int max_value, int initial_value ) { // [1 marks]
    MAX_VALUE = max_value ;                              // [1 marks]
    semaphore = initial_value ;                          //
  }

  public synchronized void claim() {    // [1 mark]
    while ( semaphore == 0 ) {          // [1 mark]
      try {
            wait();                     // [1 mark]
        } catch (Exception e) {}
      }
    semaphore-- ;                       // [1 mark]
    notifyAll() ;                       //
  }

  public synchronized void release() {  // [1 marks]
    while ( semaphore == MAX_VALUE ) {  // [1 marks]
      try {
            wait();                     // [1 mark]
        } catch (Exception e) {}
```

```
      }
    semaphore++ ;                          //  [1 mark]
    notifyAll();                           //
  }
}
```

[**PART Total 10**]

(c) Producer/Consumer problem arises when there is a difference in the relative speeds of execution of a producer of data & the consumer of the data, i.e., producer faster than consumer. [**1 mark**]

Then the producer of data must have somewhere to store it until the consumer is ready to consume it. Solved by introducing a shared buffer. Producer inserts data into it & the consumer removes data from it. [**1 mark**]

Semaphores:

`mutex`**:** binary semaphore, used to ensure mutual exclusion for access to the buffer. [**1 mark**]

`free_space`**:** general semaphore, indicates how many free slots there are in the buffer. [**1 mark**]

`num_item`**:** general semaphore, indicates the number of items which are currently in the buffer. [**1 mark**]

[**PART Total 5**]

(d) Uses classes: `Semaphore` & `Buffer`.

```
class Producer extends Thread
{
  private Buffer b = null ;
  private final Semaphore mutex, num_items, free_space ; // vars

  public Producer(Buffer    buff,      // Modified constructor
                  Semaphore me,        // [2 mark]
                  Semaphore ni,
                  Semaphore fs   ) {
    b = buff ;  mutex = me ;  num_items  = ni ;  free_space = fs ;
  }
```

```
  public void run() {
    for (int i = 0; i < 10; i++ ) {
      free_space.claim() ;              // [1 mark]
      mutex.claim() ;                   // [0.5 mark]
      // as before
      mutex.release();                  // [0.5 mark]
      num_items.release();              // [1 mark]
    }
  }
  // etc.
}

class Consumer extends Thread {
  private Buffer b = null ;
  private final Semaphore mutex, num_items, free_space ;  // vars
  private Object item = null ;

  public Consumer(Buffer    buff,          // Modified constructor
                  Semaphore me,            // [1 mark]
                  Semaphore ni,
                  Semaphore fs    )
  {
    b = buff ;   mutex = me ;  num_items  = ni ;  free_space = fs ;
  }

  public void run() {
    for (int i = 0; i < 10; i++ ) {
      num_items.claim() ;               // [1 mark]
      mutex.claim() ;                   // [0.5 mark]
      // as before
      mutex.release() ;                 // [0.5 mark]
      free_space.release();             // [1 mark]
      consumeItem(item);
    }
  }
  // etc.
}

class ProducerConsumer {
  private static final int N = 5 ;
```

```
  public static void main( String args[] ) {
    Buffer buffer        = new Buffer(N) ;
    Semaphore mutex      = new Semaphore(1, 1) ;  // [1 mark]
    Semaphore num_items  = new Semaphore(N, 0) ;  // [1 mark]
    Semaphore free_space = new Semaphore(N, N) ;  // [1 mark]

    Thread p = new Producer(buffer, mutex, num_items, free_space) ;
    Thread c = new Consumer(buffer, mutex, num_items, free_space) ;
                                                          // [1 mark]
    p.start() ;
    c.start() ;
  }
}
```

[**PART Total 13**]

[**QUESTION Total 33**]