UNIVERSITY OF WESTMINSTER

MARKING SCHEME
DO NOT COPY

FACULTY OF
SCIENCE & TECHNOLOGY
Department of Computer Science

Module:	**Concurrent Programming**
Module Code:	**6SENG002W, 6SENG004C**
Module Leader:	P. Howells
Date:	11th January 2018
Start:	10:00
Time allowed:	2 Hours

Instructions for Candidates:
You are advised (but not required) to spend the first ten minutes of the examination reading the questions and planning how you will answer those you have selected.
Answer THREE questions.
Each question is worth 33 marks.

DO NOT TURN OVER THIS PAGE
UNTIL THE INVIGILATOR INSTRUCTS YOU TO DO SO.

Module: Concurrent Programming **[MARKING SCHEME]**
Module Code: 6SENG002W, 6SENG004C
Date: 11th January 2018

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**]

[**PART Total 4**]

(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 < N )` 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**]
- `COUNT[i+1]` is a an indexed process.
  For example, `COUNT[i+1]` allows local processes to be indexed, and thus their behaviour to be dependant on the index value. [**2 marks**]

[**SUBPART Total 8**]

(ii)

- *Alphabet* – the alphabet of a process is the set of actions in which it can engage. [**1 mark**]

```
alphabet( COUNT ) = { inc, dec }
```

[1 mark]

- *Transition* – actions cause *transitions* from one state to another. [1 mark]
  One of the following transitions:

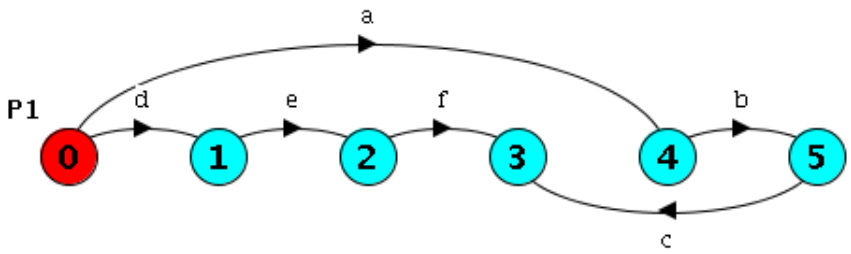
```
COUNT = Q0,
Q0 = (inc -> Q1),
Q1 = (dec -> Q0 | inc -> Q2),
Q2 = (dec -> Q1).
```

[1 mark]

[SUBPART Total 4]

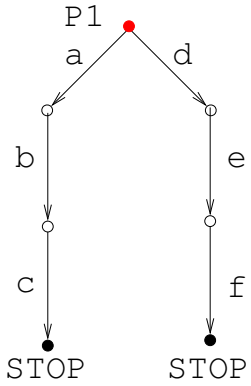
[PART Total 12]

(c) (i) P1 state machine:



[5 marks]

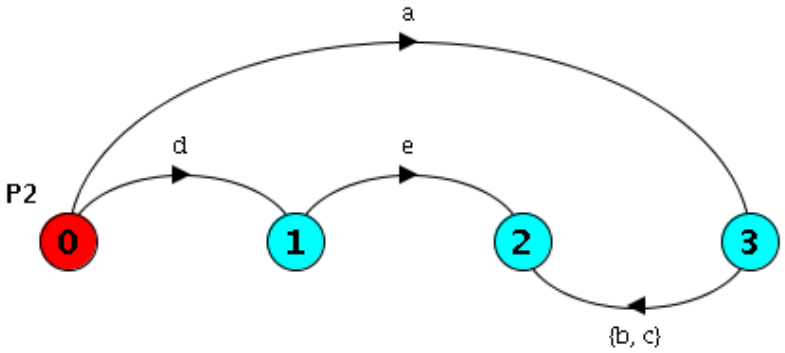
P1 trace tree.



[4 marks]

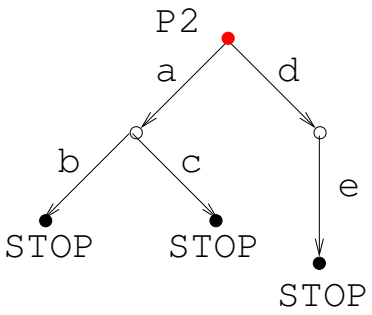
[SUBPART Total 9]

(ii) P2 state machine.



[**4 marks**]

P2 trace tree.



[**4 marks**]

[**SUBPART Total 8**]

[**PART Total 17**]

[**QUESTION Total 33**]

Question 2

(a) The FSP processes.

The required declarations.

```
const MAX_ID = 3
range DOC_ID = 0 .. MAX_ID
```

[**1 mark**]

Sets of Actions & Process Labels.

```
set PRINT_Actions = { print[DOC_ID], acquire, release, serviceprinter }

set All_Users = { ad, u1, u2 }
```

[**4 marks**]

The shared variable VAR.

```
PRINTER = PRINTER_FREE ,

PRINTER_FREE = ( acquire -> PRINTER_INUSE ) ,

PRINTER_INUSE
    = (   print[ doc : DOC_ID ] -> release -> PRINTER_FREE
        | serviceprinter -> release -> PRINTER_FREE
      ) .
```

PRINTER [**1 mark**] , PRINTER_FREE [**3 marks**] , PRINTER_INUSE [**7 marks**] .

The two processes.

```
User( DOC = 1 )
   = ( acquire -> print[ DOC ] -> release -> User ) +PRINT_Actions .
```

[**5 marks**]

```
Admin = ( acquire -> serviceprinter -> release -> Admin ) +PRINT_Actions .
```

[**4 marks**]

[**PART Total 25**]

(b) The complete systems

```
|| MutualExclusion_PRINTER
      = (    All_Users :: PRINTER
           || u1 : User ( 1 ) || u2 : User ( 2 )
           || ad : Admin
        ) .
```

[**5 marks**]

[**PART Total 5**]

(c) Mutual exclusive access to the `PRINTER` is achieved by requiring the three processes to "lock" it before they can use 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 sub-classing the `Thread` class or implementing `Runnable` interface. [**1 mark**]

How these work:

- 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 ;
```

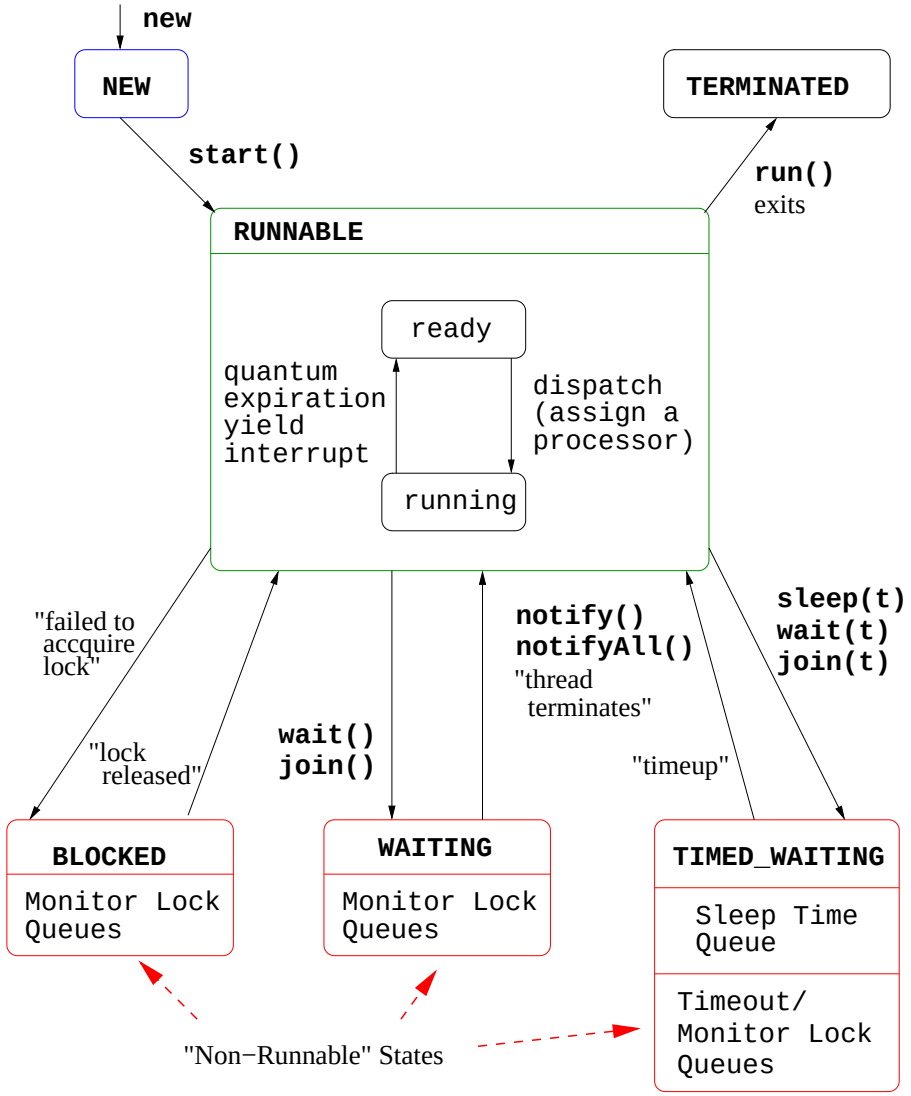
```
    public SubClass()
    {
      t = new Thread(this) ;
      t.start();
    }

    // ‘‘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 9**]

(b) The diagram illustrates a JDK 1.5 Java thread's life-cycle/states & which method calls cause a transition to another state.



[**7 marks**]

NEW State: A new thread is created but not started, thereby leaving it in this state by executing:

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

In this state a thread is an empty `Thread` object, no system resources have been allocated for it yet. In this state calling any other method besides `start()` causes an `IllegalThreadStateException`. [**1 mark**]

RUNNABLE State: A thread is in this after executing the `start()` method:

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

[**1 mark**]

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

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. [**1 mark**]

BLOCKED State Thread state 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**]

WAITING State A thread in the waiting state is waiting for another thread to perform a particular action.

A thread is in the waiting state due to calling one of the following methods, with no timeout parameter.

- A thread called `wait()` on an object & is waiting for another thread to call `notify()` or `notifyAll()` on that object. [**1 mark**]
- A thread called `otherthread.join()` & is waiting for the specified thread `otherthread`to terminate. [**1 mark**]

TIMED_WAITING State Thread state for a waiting thread with a specified waiting time.

A thread is in the timed waiting state due to calling one of the following methods with a specified positive waiting time:

- A thread calls `sleep(t)` & is waiting for the timeout `t`.
- A thread calls `wait(t)` on an object & is waiting for the timeout `t` or another thread to call `notify()` or `notifyAll()` on that object.
- A thread calls `Thread.join(t)` & is waiting for the timeout `t` or is waiting for a specified thread to terminate.

[**2 marks**]

TERMINATED State: A thread dies from "natural causes", i.e., a thread dies naturally when its `run()` method exits normally. [**1 mark**]

[**PART Total 18**]

(c) (The following points are relevant, but other correct answers will be marked accordingly.)

Every Java thread is a member of a *thread group*. When you create a thread, you can either: allow the runtime system to put the new thread in some reasonable default group, or you can explicitly set the new thread's group when you create the thread. [**1 mark**]

The thread is a permanent member of whatever thread group it joins upon its creation. A thread can not be moved to a new group after it has been created. [**1 mark**]

The Java runtime system creates the `ThreadGroup` "`main`", & unless specified otherwise, all new threads are members of `main`. Because the runtime system automatically places the new thread in the same group as the thread that created it. [**1 mark**]

The following code creates a `ThreadGroup` named `Group_A` and places a thread called `Thread_A_1` in it:

```
ThreadGroup myThreadGroup = new ThreadGroup("Group_A");
Thread myThread = new Thread(myThreadGroup, "Thread_A_1");
```

[**1 mark**]

Missing features on a group of threads: starting and terminating. Since all threads must be started and terminated. [**2 marks**]

[**PART Total 6**]

[**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 that behave like critical sections, i.e., mutually exclusive execution. **[2 marks]**

The main components are:

- "Permanent" variables, used to represent the state of the resource. Plus one or more "condition variables" which are used to control access to the resource. **[1 mark]**
- Procedures & functions that implement operations on the resources by manipulating the monitor variables. Two types, those that are visible outside of the monitor, i.e. its interface & those that are invisible, i.e. helpers. **[1 mark]**
- A monitor body, statements which are executed only once when the monitor is started. It initializes the state of the monitor i.e. the resource. **[1 mark]**

[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. [**2 marks**]

`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 17**]

(c) **(i)** Execution of `MessageSystem` program:

1. Initially `mb` is unlocked & its wait-set is empty. `p` & `r` are created & started, i.e., made "Runnable". **[1 mark]**
2. First `p` calls `post()` & acquires `mb`'s lock. Since `message_posted` is `false`, it can post the message & releases `mb`'s lock. **[1 mark]**
3. If `r` calls `retrieve()` while `p` is executing `post()`, `r` is moved to the `BLOCKED` state. **[1 mark]**
4. Once `p` releases the lock `r` move from the `BLOCKED` state to the `RUNNABLE` state. **[1 mark]**
5. `r` can now acquire `mb`'s lock, since `message_posted` is `true` it can complete `retrieve()` & release the lock. **[1 mark]**
6. Both `r` & `p` terminate. **[1 mark]**

[SUBPART Total 6]

(ii) When deadlock has occurred all of the `Poster` and `Retriever` threads would be stuck in the `MessageBoard` monitor's *wait-set*. **[1 mark]**

This would have happened because they would have attempted to do a `post()` or `retrieve()` and been blocked & then called `wait()`, since only `notify()` is used it is possible that a thread that could "unblock" the situation is never notified. **[2 marks]**

The simplest change that could be made to the two `MessageBoard` methods `post()` and `retrieve()` is to use `notifyAll()` instead of `notify()`, then all the threads would be notified & one of them would be able to "unblock" the system. **[2 marks]**

[SUBPART Total 5]

[PART Total 11]

[QUESTION Total 33]

Question 5

(a) Semaphores are concurrent programming language mechanism used to achieve mutual exclusion & synchronization. **[1 mark]**

A semaphore `s` is an integer variable which can take only non–negative values, ($s \geq 0$). **[1 mark]**

There are two types of semaphores: binary semaphores (0 or 1) & general semaphores ($n \geq s \geq 0$). **[2 marks]**

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. **[2 marks]**

[PART Total 9]

(b) The Dining Philosophers problem (for 5 Philosophers) illustrates the dangers of deadlock when sharing resources amongst competing concurrent processes. The problem is stated as follows: 5 philosophers spend their lives thinking and eating. They share a common dining room where there is a table with a plate for each philosopher. In the centre there is a bowl of spaghetti, and the table is laid with five forks. When a philosophers has finished thinking he feels hungry and enters the dining room, goes to his plate and picks up the fork on his left, but since the spaghetti is so tangled he needs to pick up and use the fork on his right as well. When he has finished eating he puts down both forks and leaves the room. Plus table diagram. **[4 marks]**

Deadlock can occur when all 5 philosophers have sat down & picked up their own fork, then since there is no free fork they all starve. Deadlock can be avoided by introducing a Butler who allows at most 4 Philosophers into the dining room. Then there will always be a free fork which can be used by one of the philosophers. **[2 marks]**

[PART Total 6]

(c) The following code fragments are expected:

```
class Philosopher extends Thread
{
  private Semaphore butler;                                  // [1 mark]
  private Semaphore right_fork, left_fork;                   // [2 mark]

   public Philosopher(int id, Semaphore but,                 //
                      Semaphore right, Semaphore left ){     // [2 mark]
      super("Philosopher #" + id);
      right_fork = right;
      left_fork  = left;
      butler  = but;
```

```
    }

    public void run() {
      while ( true ) {
          // think
          butler.claim();             // [1 mark]
          right_fork.claim();         // [1 mark]
          left_fork.claim();          //
          // eat
          right_fork.release();       // [1 mark]
          left_fork.release();        //
          butler.release();           // [1 mark]
      }
    }
  }

  class DiningPhilosophers{
    private static final int NumPhils = 5 ;

    public static void main( String args[] ) {
      Semaphore   butler = new Semaphore(NumPhils - 1,    //
                                         NumPhils - 1 ) ; // [2 mark]

      Semaphore[] forks = new Semaphore[NumPhils] ;       // [2 mark]
      Thread[]    phils = new Thread[NumPhils] ;          //

      for (int i = 0 ; i < NumPhils ; i++ ) {
         forks[i] = new Semaphore(1, 1) ;                 // [2 mark]
      }

      for (int i = 0 ; i < NumPhils ; i++ ) {
        phils[i] = new Philosopher( i, butler,            // [3 mark]
                                    fork[i],
                                    fork[(i + 1) % NumPhils]) ;
        phils[i].start() ;
      }
    }
  }
```

[**PART Total 18**]

[**QUESTION Total 33**]