UNIVERSITY OF WESTMINSTER

MARKING SCHEME
DO NOT COPY

School of Computer Science and Engineering

Module:	**Concurrent Programming**
Module Code:	**6SENG002W, 6SENG004C**
Module Leader:	P. Howells
Date:	16th January 2019
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.

Only the THREE questions with the HIGHEST MARKS will count towards the FINAL MARK for the EXAM.

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

Question 1

(a) (i)

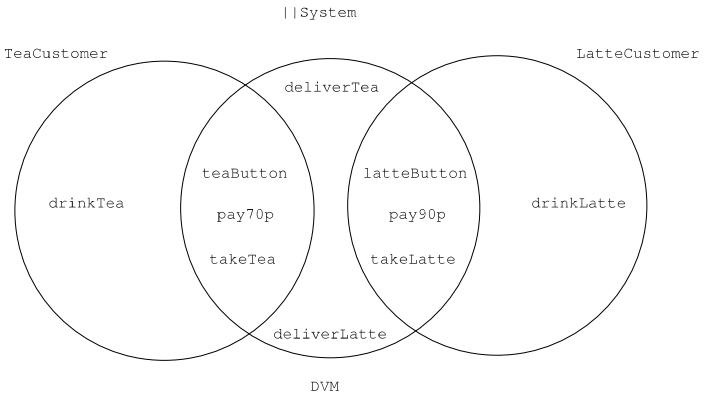
```
alphabet(DVM) = { deliverLatte, deliverTea, latteButton,
                  pay70p, pay90p, takeLatte, takeTea,
                  teaButton }

alphabet(TeaCustomer) = { drinkTea, pay70p, takeTea,
                          teaButton }

alphabet(LatteCustomer) = { drinkLatte, latteButton,
                            pay90p, takeLatte }
```

[**SUBPART Total 3**]

(ii) Alphabet diagram for the `System`.



[**SUBPART Total 7**]

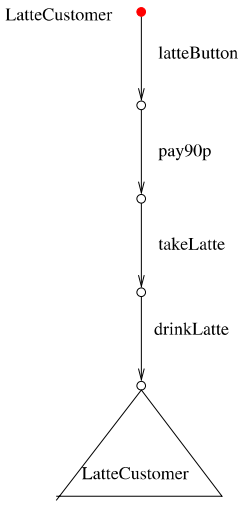
(iii) Type of `System`'s actions.

Action	Type	Processes
teaButton	Synchronous	DVM, TeaCustomer
latteButton	Synchronous	DVM, LatteCustomer
pay70p	Synchronous	DVM, TeaCustomer
pay90p	Synchronous	DVM, LatteCustomer
deliverTea	Asynchronous	DVM
deliverLatte	Asynchronous	DVM
takeTea	Synchronous	DVM, TeaCustomer
takeLatte	Synchronous	DVM, LatteCustomer
drinkTea	Asynchronous	TeaCustomer
drinkLatte	Asynchronous	LatteCustomer

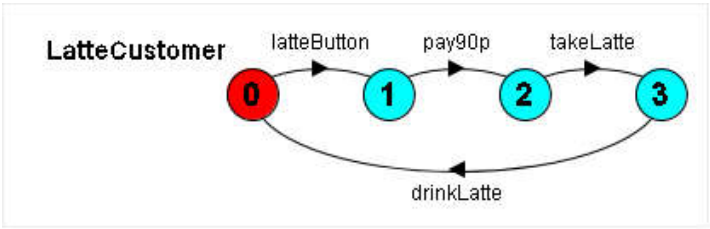
[SUBPART Total 5]

[PART Total 15]

(b) **(i)** `LatteCustomer` trace tree & LTS graph:



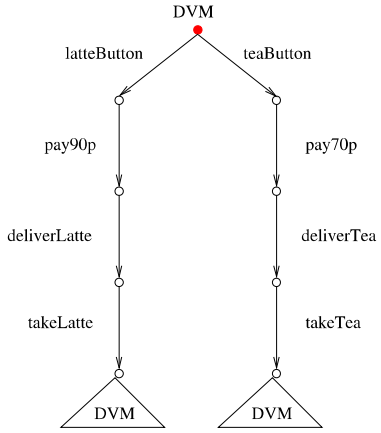
[2 marks]



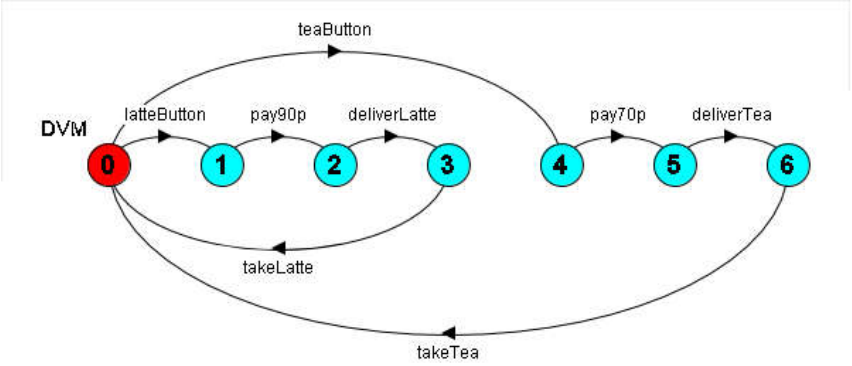
[4 marks]

[SUBPART Total 6]

(ii) `DVM` trace tree & LTS graph:



[3 marks]



[5 marks]
[SUBPART Total 8]

[PART Total 14]

(c) *Deadlock* means that all the processes of the program are "stuck", i.e. blocked from making progress, usually when they form a cycle of dependency between them that cannot be broken. E.g. the 5 dining philosophers each with one fork. [2 marks]

Deadlock is represnted in FSP language by the pre-defined process `STOP`. [1 mark]

A deadlocked state in a Labelled Transition System (LTS) graph is one that has no "out transitions". [1 mark]

[PART Total 4]

[**QUESTION Total 33**]

Question 2

(a) There are obviously many FSP processes that could be used to model this shared counter and two doorman system. However, the main requirement is to ensure the mutually exclusive use of the shared counter. This has to be done by using a locking/unlocking process that the 2 doormen must synchronise with. The following is a *solution outline* of what is expected. Marks will be awarded for similarity to the key aspects of this solution.

Declarations.

```
const MAXCAPACITY = 5

range INT = 0 .. MAXCAPACITY
```

[**2 marks**]

Sets of Actions & Process Labels.

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

set DOORMEN = { enter, exit }
```

[**4 marks**]

Two approaches for the shared `Counter`:

1. Use 2 separate process: one deals with the incrementing & decrementing of the counter, via `read[v]` & `write[nv : INT]` actions. The other is just a acquire/release process to ensure ME.
2. Use one process that integrates the counter actions & the acquire/release actions into one process using a collection of mutually recursive local processes.

So (1) is simpler so upto [**6 marks**] ; (2) is more sophisticated so upto [**8 marks**] .

`EntryDoorman` & `ExitDoorman` processes, their behaviour is virtually identical: acquire ME control of counter, incrementing or decrementing the counter, release ME control of counter. [**7 marks**]

[**PART Total 25**]

(b) The complete systems:

```
|| DoorMen = ( enter : EntryDoorman || exit : ExitDoorman ) .

|| Concert_Hall = ( DOORMEN :: Counter || DoorMen ) .
```

[**4 marks**]

[**PART Total 4**]

(c) Mutual exclusive access to the shared counter is achieved by requiring the two doormen processes to "lock" it before they can read from & write to it, by performing a synchronised `acquire` action with the shared counter process. [**2 marks**] When they have finished with it they must "unlock" it, by performing a synchronised `release` action with the shared counter process. [**2 marks**]

[**PART Total 4**]

[**QUESTION Total 33**]

Question 3

(a) Single *thread* similar to sequential program: it has a beginning, an end, a sequence, and at any given time during the run time of the program there is a single point of execution. But a thread is not a program, it cannot run on its own, but runs within a program. [**2 marks**]

Definition: A *thread* is a single sequential flow of control within a process.

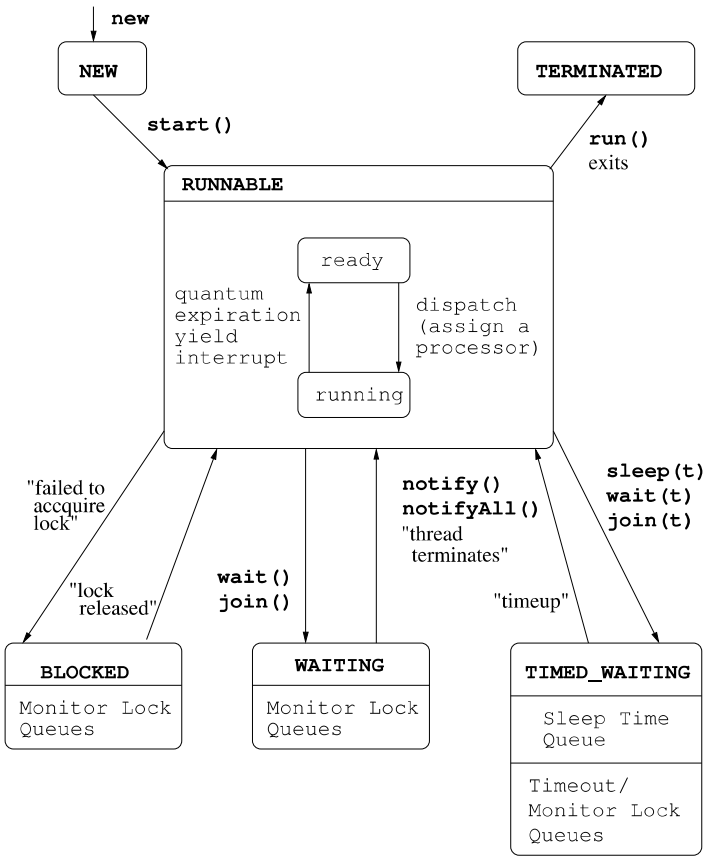
[**1 mark**]

Thread similar to real process, i.e. both are single sequential flows of control, but a thread is "lightweight" because it runs within the context of a full-blown program and takes advantage of the resources allocated for that program and the program's environment. [**1 mark**]

A thread must have its own resources within a running program, e.g. execution stack and program counter. The code running in the thread works only within that context. [**1 mark**]

[**PART Total 5**]

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



[**7 marks**]

Student's will either give invented code fragments or may make reference to parts of the program code given in Appendix B, either is acceptable.

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) **Java scheduling:** the following (or similar) points should be covered.

Java uses *fixed priority preemptive* scheduling – this schedules threads based on their *priority* relative to other *"Runnable"* threads. [**1 mark**]

Thread priorities range between `MIN_PRIORITY` (= 1) and `MAX_PRIORITY` (= 10), normally a threads priority is set to `NORM_PRIORITY` (= 5). [**1 mark**]

When a thread is created, it inherits its priority from the thread that created it; thus `racer[1]` & `racer[2]` have priority 5. [**1 mark**]

A thread's priority can be modified at any time after its creation using the `setPriority()` method; thus `racer[0]` & `racer[3]` have priorities 7 & 2. [**1 mark**]

The *"Runnable"* thread with the highest priority is chosen for execution, i.e. `racer[0]`. Only when it stops, yields, or becomes *"Not Runnable"*, will a lower priority thread start executing. Thus `racer[0]` runs to completion before either `racer[1]`, `racer[2]` or `racer[3]` start. [**1 mark**]

If there are two threads of the same priority waiting for the CPU, the scheduler chooses them in a round-robin fashion. So one of `racer[1]` or `racer[2]` start to execute, after doing one `println` it `yields` causing the other thread to be run. The other thread behaves in a similar fashion, thus resulting in a strict interleaving of the two thread's outputs. [**1 mark**]

Then after `racer[1]` & `racer[2]` have terminated, the lowest priority thread `racer[3]` runs uninterrupted & to completion. [**1 mark**]

The thread race output is all of `racer[0]`'s then `racer[1]`'s & `racer[2]`'s would be strictly interleaved & starting with either one & finally `racer[3]`'s, e.g.

```
Racer[0], i = 10
Racer[0], i = 20
Racer[0], i = 30
```

```
Racer[1], i = 10
Racer[2], i = 10
Racer[1], i = 20
Racer[2], i = 20
Racer[1], i = 30
Racer[2], i = 30
Racer[3], i = 10
Racer[3], i = 20
Racer[3], i = 30
```

[**3 marks**]

[**PART Total 10**]

[**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) Students may include code similar to that given below &/or refer to the code in Appendix B, to illustrate basic structure & features of a Java monitor.

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 run time 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 `jillsphone` is unlocked & its wait-set is empty. `jill` & `steve` are created in the NEW state, & then started, i.e., made RUNNABLE. [**1 mark**]
2. Then `jill` calls `readtext()` & acquires `jillsphone`'s lock & as a result `steve` would enter the BLOCKED state. [**1 mark**]
3. Then `jill` finds that `got_message` is `false`, then `wait` is called, this causes `jill` to be placed into `jillsphone`'s wait-set, it enters the WAITING state & releases `jillsphone`'s lock. [**1 mark**]

4. Once `jill` releases the lock `steve` leaves the BLOCKED state & re-enters the RUNNABLE state & can acquire `jillsphone`'s lock. [**1 mark**]
5. `steve` can complete the `sendtext` call, at the end it calls `notifyAll()` & finally releases `jillsphone`'s lock, finishes & enters the TERMINATED state. [**1 mark**]
6. `jill` is woken up by `steve`'s `notifyAll()`, so `jill` is removed from the wait-set & moves from the WAITING to the RUNNABLE state. [**1 mark**]
7. When `jill` re-acquires the `jillsphone`'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) 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)` & `initialise(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 following is expected, but different relevant points will also be accepted.

Advantages: very simple, solves simple problems easily. [**1 mark**]

Disadvantages:

1. Unstructured and hence leads to errors. Examples of the types of errors.
2. Used for mutual exclusion, process ordering & conditional synchronization
3. Can only test one semaphore at a time.
4. Committed to testing a semaphore, i.e. can not withdraw.

[**4 marks**]

The main reason that monitors are seen as "better" than semaphores is that they provide a structured approached as opposed to the unstructured approached of semaphores. [**2 marks**]

[**PART Total 7**]

(c) **(i)** Something similar to the following is expected, but any code that has the correct elements is acceptable: ensures ME using a binary semaphore, two threads & has the right "claims" & "releases". Students may use Java's "acquire" instead of "claim", this is acceptable.

```
class MEProcess extends Thread                          // [1 mark]
{
  private int pid ;
  private BinarySemaphore semaphore;                    // [1 mark]

  public MEProcess (int id, BinarySemaphore sema)       // [1 mark]
  {
     pid = id ;
     semaphore = sema ;
  }

  public void run()
  {
     semaphore.claim() ;                                // [2 mark]
     System.out.println("Process #" + pid
                        + ": in Critical Section");
     semaphore.release() ;                              // [2 mark]
  }
}

class Mutex
```

```
{
  public static void main(String args[])
  {
    BinarySemaphore binsema = new BinarySemaphore(1); // [2 mark]

    Thread p1 = new MEProcess(1, binsema);            // [2 mark]
    Thread p2 = new MEProcess(2, binsema);            // [2 mark]

    p1.start();
    p2.start();
  }
}
```

[**SUBPART Total 13**]

(ii) Explanation: Using a binary semaphore initialized to 1. [**2 marks**] Use of claim/release of the same semaphore, around each threads critical section. [**2 marks**]

[**SUBPART Total 4**]

[**PART Total 17**]

[**QUESTION Total 33**]