UNIVERSITY OF WESTMINSTER

MARKING SCHEME
DO NOT COPY

SCHOOL OF
COMPUTER SCIENCE & ENGINEERING

Module Title: **Concurrent Programming**

Module Code: **6SENG006W**

In-Class Test: **Version 1, 14th December, 2022**

Start Time: **09:00**

Submission Deadline: **10:30**

RAF Submission Deadline: **TBC**

INSTRUCTIONS FOR CANDIDATES

Version 1 of In-Class Test. There will be 3 slightly different versions, 1 for each of 3 Tutorial Groups. Tests will be done on Blackboard.

There are EIGHT questions in the test.

Answer ALL EIGHT questions.

Questions 1 - 4 are worth 10 marks each.

Questions 5 - 8 are worth 15 marks each.

YOU MUST SUBMIT YOUR ANSWERS BEFORE THE SUBMISSION DEADLINE.

Question 1

Explain concurrency concepts:

(a) *Process:* A *process* has a self-contained execution environment. A process generally has a complete, private set of basic run-time resources; in particular, each process has its own memory space. Processes are often seen as synonymous with programs or applications. **[2 marks]**

(b) *Asynchronous Action:* occurs when a process performs an action independently of any other process, i.e. it does not require the participation in or synchronisation on the action by another process. So a process can perform an asynchronous action whenever it wants to, as it cannot be blocked from occurring by another process. **[2 marks]**

(c) *Nondeterminism:* means that the occurrence of an event or action cannot be predicted in advance, or that the order in which a collection of events or actions cannot be predicted in advance. E.g. tossing a coin, or rolling a dice several times in a row, or a process waiting for input from several other processes with unknown execution times. (Random, unpredictable.) **[2 marks]**

(d) *Interference:* occurs when concurrently executing processes share an object, e.g. a file, bank account balance, etc, & concurrently update its state without having mutually exclusive access to it, resulting in the state/data being inconsistent. **[2 marks]**

(e) *Individual starvation:* occurs when one (or more) processes in a concurrent system is blocked from accessing the shared resource(s) it needs to make progress, e.g. shared data or device. E.g. one philosopher being kept form picking up 2 forks, thus cannot eat. The result is that it cannot make progress, so can be thought of as partial system deadlock, as opposed to a complete system deadlock. It is a Liveness property to avoid individual starvation no process is continuously blocked, by other processes.

[2 marks]

[QUESTION Total 10]

Question 2

(a)
```
Chocolates  = (   { mars, flake, crunchie } -> runout -> Chocolates
                | twirl -> deliverTwirl -> Chocolates
              ) .
```

[**5 marks**]

(b)
```
Animals = (   dog -> ( woof -> Animals | meow -> ERROR )
            | duck -> quack -> END ) .
```

[**5 marks**]

[**QUESTION Total 10**]

Question 3

(a) A Java thread can be defined by subclassing the `Thread` class and override the `run()` method. [**2 marks**]

The new thread class should have a constructor, and in that it should always call one of the `Thread` class's constructors, e.g. `super(...)`, this ensures the thread is constructed properly. [**1 mark**]

The thread's code to be executed is placed inside the `run()` method. This is executed once the thread has been started. [**1 mark**]

Example code:

```
class SimpleThread extends Thread
{
   public SimpleThread(){ super(); ...  } // constructor

   public void run(){ // code to execute }
}
```

[**2 marks**]

[**PART Total 6**]

(b) This code does not work because using `manager.run()` does not create a thread to execute the `manager`'s `run()` method, but simply runs it in the

current thread. The result is that the 3 methods are executed sequentially not concurrently. [**2 marks**]

The correct way is to call the `start()` method:

```
manager.start()
worker_1.start()
worker_2.start()
```

[**2 marks**]

[**PART Total 4**]

[**QUESTION Total 10**]

Question 4

(a) They all act as sycnhronisation mechanisms & have the same operations: initialise the semaphores value, claim & release the semaphore. A *mutex* is just another name for a *binary semaphore*. A *binary semaphore* can only have the values 0 or 1. A *general semaphore* can have the values 0 up to N, where N ≥ 1.

[**PART Total 3**]

(b) *initialise(s, v)*: sets the initial value of a semaphore s to a specific value v, e.g. for a binary semaphore 0 would mean its locked (a claim fails), 1 means that it is unlocked (a claim succeeds). [**1 mark**]

claim(s): this is used to decrement the value of a semaphore. If the semaphore's value is greater than 0 then it is decremented & the process contiues to execute. If it is 0 then the process is blocked (placed in a queue) until the value of a semaphore is at least 1 & can then be decrementing & the process can then continue to execute. claim(s) is an atomic uninterruptible action which means that if several claims are attempted simultaneously on a single semaphore then only one of them would succeed & the others would be blocked. [**3 marks**]

release(s): this is used to increment the value of a semaphore. If the semaphore's value is greater than 0 then it is incremented & the process contiues to execute. If it is 0 then it is incremented & the process checks if any processes are blocked, if so it wakes one up (removed from the queue)

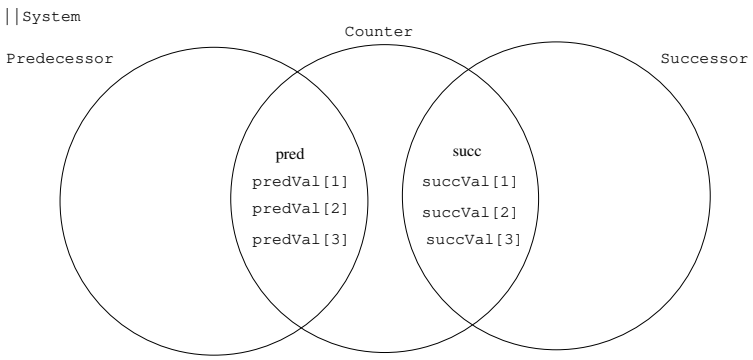
& the process can then continue to execute. Similarly to claim(s), relase(s) is an atomic uninterruptible action. [**3 marks**]

[**PART Total 7**]

[**QUESTION Total 10**]

Question 5

(a) SYSTEM *Alphabet* diagram.



[**5 marks**]

(b)

Action	Type	Processes
None	Asynchronous	Predecessor
pred, predVal[1], predVal[2], predVal[3]	Synchronous	Counter, Predecessor
None	Asynchronous	Counter
succ, succVal[1], succVal[2], predVal[3]	Synchronous	Counter, Successor
None	Asynchronous	Successor

[**8 marks**]

(c) When the Counters value is 3 the SYSTEM can perform the actions: `pred, succVal[3], predVal[3]` [**2 marks**]

[**QUESTION Total 15**]

Question 6

(a) Description of Java thread state.

NEW State: A new thread is created & put in this state before it is started. **[1 mark]**

RUNNABLE State: A thread is in this after the `start()` method has been called & it is ready to be executed in this state, the only state a thread can execute in. **[1 mark]**

BLOCKED State: A thread attempted to acquire a synchronisation lock, but failed, so is placed in this state until the lock is released. **[1 mark]**

WAITING State A thread in this state is waiting (`wait()`) for another thread to perform a particular action or to terminate (`join()`). Exits when notified (`notify()` or `notifyAll()`) action done, or other thread terminates. **[1 mark]**

TIMED_WAITING State A thread is waiting as for the WAITING state, but with time limits. In addition it can call `sleep(t)`. It exits as for the WAITING state or the time limits expire. **[1 mark]**

TERMINATED State: A thread dies when its `run()` method exits normally. **[1 mark]**

[PART Total 6]

(b) The thread is executing so it is in the RUNNABLE state, if it *fails to acquire a synchronisation lock*, it then is moved to the BLOCKED state. In this state it is blocked & cannot execute its run method. **[2 marks]**

It is moved from the BLOCKED state back to the RUNNABLE state when the synchronisation lock is release by some other thread. **[1 mark]**

Once back in the RUNNABLE state is available to be re-shceduled & can then continue to execute its run method again. **[1 mark]**

[PART Total 4]

(c) At A the thread is executing the monitor (`synchronized`) `method1`, so it is in the RUNNABLE state & holds the monitor's synchronisation lock. **[1 mark]**

When it calls `wait()`, it releases the monitor's lock, is added to the monitor's *wait-set*, moved to WAITING state, and cannot execute its run method. **[1 mark]**

It stays in the WAITING state until it is notified via `notify()` or `notifyAll()` that the state has been modified. It is then moved out of the wait-set & back to the RUNNABLE thread state. **[1 mark]**

When it is scheduled to execute it can attempt to acquire the monitor's lock, but if it fails it will be moved to the BLOCKED state until it is released & moved back to the RUNNABLE state & tries again. (See part (b).) **[1 mark]**

If it eventually acquires the monitor's lock it re-enters the monitor at B and continues. **[1 mark]**

[PART Total 5]

[QUESTION Total 15]

Question 7

(a) Modifications to convert `Variable` class into a secure and correctly functioning *monitor*:

```
1    class Variable
2    {
3       private int     variable = 0 ;         // added "private"
4       private boolean updated  = false ;     // added "private"
5
6       public synchronized int value()        // added "synchronized"
7       {
8          while ( !updated )
9          {
10           try {
11                 wait() ;                    // replaced by "wait()"
12            } catch(InterruptedException e){ }
13         }
14         updated = false ;
15         notifyAll() ;                       // added "notifyAll()"
16         return variable ;
17      }
18
19      public synchronized void assign( int newValue ) // added "synchronized"
```

```
20      {
21         while ( updated )
22         {
23            try {
24                 wait() ;                    // replaced by "wait()"
25            } catch(InterruptedException e){ }
26         }
27         variable = newValue ;
28         updated = true ;
29         notifyAll() ;                       // added "notifyAll()"
30      }
31   }
```

[1 mark per modification.]

[**PART Total 8**]

(b) Added `private` (`protected` is acceptible) ensure secure data encapsulation of all data members. [**1 mark**]

Added `synchronized` to both `value` and `assign` methods as a result turning `Variable` into a "monitor". This ensures mutually exclusive execution of `Variable`'s 2 `synchronized` methods and thus ensures safe & secure sharing of its data members. [**2 marks**]

Replaced `Thread.sleep(1000)` by `wait()`, this causes the calling thread to be added to `Variable`'s *wait-set* and moves it to the `WAITING` thread state & it releases `Variable`'s synchronization lock. This reduces the possibility of deadlock & is more efficient than repeatedly sleeping. [**2 marks**]

Added `notifyAll()` to both `value` and `assign` methods. This is used to signal to all threads in `Variable`'s *wait-set* that the state has been modified & that the waiting threads may be able to complete their monitor method. All waiting threads moved back to the `RUNNABLE` thread state. [**2 marks**]

[**PART Total 7**]

[**QUESTION Total 15**]

Question 8

(a) The Readers & Writers Problem is a generalisation of the mutual exclusion problem. The problem is that a number of processes share some piece of data, usually some kind of data base. A number of N "Writers" processes that only "write" to the data base. M "Readers" processes that only "read" from the data base. To maintain the consistency of the data base reading & writing must be mutually exclusive, as must writing by different Writers. But for the sake of efficiency multiple Readers are allowed.

The mutual exclusion requirements are related to the reading & writing to the data base, either:
K ($\leq$ M) Readers can have exclusive access to the data base or 1 Writer.

[5 marks]

(b) In the Readers/Writers problem two binary semaphores are used:

`mutex`: is a binary semaphore, it is used to ensure mutually exclusive access to the variable that records the count of Readers currently reading the database. `mutex` is shared & used by all the Readers processes. When the Readers attempt to begin reading they have to increment the counter, so this must be done mutually exclusively otherwise there will be interference. Similarly when they stop reading & increment the counter.
[4 marks]

`writing`: is a binary semaphore, controls when writing to the database is allowed. When it is 0 then writing is not allowed either because at least one Reader is reading or one Writer is writing. When it is 1 then writing is allowed because no reading or writing is occurring. It is shared & used by all the Readers & Writers processes.

The first Reader that starts reading claims it to block the Writers from writing, & the last Reader releases it when it finishes reading, thus allowing writing.

The Writers write to the database from within their critical section. So the Writers use `writing` as the lock for the critical section to ensure mutually exclusive access to it.
[6 marks]

[QUESTION Total 15]