UNIVERSITY OF WESTMINSTER

MARKING SCHEME
DO NOT COPY

protected

SCHOOL OF COMPUTER SCIENCE & ENGINEERING

Module Title:	**Concurrent Programming**
Module Code:	**6SENG002W**
In-Class Test:	**EXAMPLE ICT**
Start Time:	**11:00**
Submission Deadline:	**13:15**
RAF Submission Deadline:	**13:50**

INSTRUCTIONS FOR CANDIDATES

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 what each of the following concurrency concepts mean:

(a) 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) A synchronous action is one that is performed by at least 2 processes at exactly the same time. So a synchronised action must be executed at the same time by all the processes that participate in it. Therefore, if a synchronised action can not be executed by at least one of the processes that share it, then it can not be executed. [**2 marks**]

(c) 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**]

(d) Mutual exclusion protocol: used to control access to a concurrently executing process's critical sections, involves acquiring a synchronisation lock, i.e. locking & once finished releasing the lock, i.e. unlocking.

```
waiting -->
LOCK --> execute critical section --> UNLOCK -->
continue
```

[**2 marks**]

(e) Livelock: occurs when concurrently executing processes are active & appear to be making "useful" progress, but in fact are stuck in an endless cycle of activities that in fact do not result in real progress of the process. E.g. philosophers endlessly picking up/putting down a fork, but never getting to eat. [**2 marks**]

[TOTAL 10]

Question 2

The FSP process definitions for the *Labelled Transition System (LTS)* graphs:

(a)
```
PROC1 = (   act1 -> act2 -> PROC1
          | act1 -> act3 -> PROC1 ).
```
[**2 marks**]

(b)
```
PROC2 = (   act1 -> (   act2 -> PROC2
                      | act3 -> PROC2 ) ).
```
[**3 marks**]

(c)
```
|| TWO_P1s = ( PROC1 || PROC1 ).
```
[**5 marks**]

[TOTAL 10]

Question 3

(a) Threads are sometimes called lightweight processes because:

- Threads exist & run within the context of a process/program, it cannot exist on its own.
- It takes advantage of & uses the resources allocated for that process/program, & accesses the its environment.
- But it requires its own resources within a running program, e.g. its own execution stack & program counter.
- Creating a new thread requires fewer resources than creating a new process.

[**4 marks**]

(b)
```
// Code 1:
        myThread1.start() ;
        myThread2.start() ;
```
In this case because the `start()` method is called on the 2 thread objects, 2 new threads are created, one for each of the threads. Consequently their `run()` methods will execute concurrently & thus their outputs will be randomly interleaved. E.g.

```
myThread1
myThread2
myThread2
myThread1
myThread1
myThread2
```

[**3 marks**]

```
// Code 2:
        myThread1.run() ;
        myThread2.run() ;
```

In this case because the `run()` method is called on the 2 thread objects, & not `start()`, so no new threads are created, for the 2 threads. Consequently their `run()` methods will be execute sequentially in order they are called by the calling thread, e.g. the `main` thread, so not concurrently & thus their outputs will be in order:

```
myThread1
myThread1
myThread1
myThread2
myThread2
myThread2
```

[**3 marks**]

[TOTAL 10]

Question 4

(a) 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. They all have the same operations: initialise the semaphores value, claim & release the semaphore. [**4 marks**]

(b) *Mutex*: used as the lock to ensure mutually exclusive access to concurrently executing processes critical sections of code. E.g. philosophers picking up forks. [**2 marks**]

Binary semaphore: to achieve the ordering of the action of a collection of concurrently executing processes, by requiring processes to claim a semaphore before performing an action. **[2 marks]**

General semaphore: to manage resource allocation, by its value indicating the number of resources available to be claimed/used by a collection of concurrently executing processes. E.g. items in a buffer to be retrieved. **[2 marks]**

[TOTAL 10]

Question 5

(a) Draw the alphabet diagram for `MI6`. **[6 marks]**

MI6

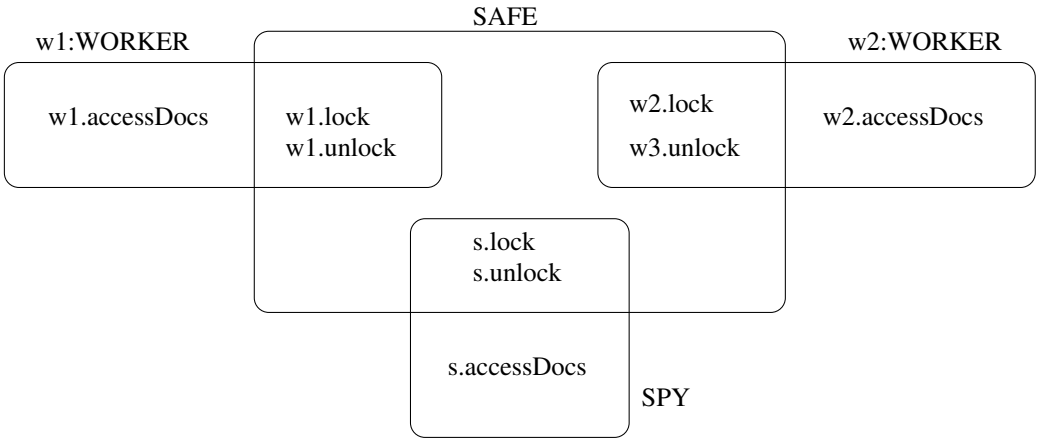

Figure 1: Alphabet diagram for `MI6`.

(b) For each action in `MI6`:

ACTION	Async/Sync	PROCESSES
s.accessDocs,	Async,	SPY
s.lock,	Sync,	SAFE, SPY
s.unlock,	Sync,	SAFE, SPY
w1.accessDocs,	Async,	w1 : WORKER

```
w1.lock,          Sync,        SAFE, w1 : WORKER
w1.unlock,        Sync,        SAFE, w1 : WORKER
w2.accessDocs,    Async,       w2 : WORKER
w2.lock,          Sync,        SAFE, w2 : WORKER
w2.unlock         Sync,        SAFE, w2 : WORKER
```

[**9 marks**]

Question 6

In relation to a multi-threaded Java program, its threads and thread life-cycle states, answer the following questions.

(a) After a thread has been created using `new`, it is in the NEW Thread state & it is an empty Thread object. No system resources have been allocated to it yet. When a thread is in this state, you can only start the thread by calling the start() method. Calling any other method besides start() raises an IllegalThreadStateException. [**2 marks**]

(b) Can Execute code just in: RUNNABLE.
Not able to Execute code in: NEW, BLOCKED, WAITING, TIMED_WAITING & TERMINATED. [**2 marks**]

(c) If a thread fails to acquire a synchronisation lock then it must have been executing code, i.e. in the RUNNABLE state. So its state transition is from the RUNNABLE state to the BLOCKED state. So it moves to a non-RUNNABLE state, it is not available to be scheduled & has to remain there until the lock it tried to acquire has been released. [**3 marks**]

(d) All threads have a priority between 1 & 10. When there are several threads ready to be executed, the JVM picks one of the threads in the RUNNABLE state with the highest priority. If there is a group of threads with the same priority waiting to execute, the scheduler chooses them in a "round-robin" fashion. [**3 marks**]

(e) When a thread is in the `TIMED_WAITING` state then one of the following must happen before it returns to RUNNABLE.

- A thread called `Thread.sleep(t)` & is waiting for the timeout `t`.

- A thread called `Object.wait(t)` on an object & is waiting for either the timeout t to have elapsed, or another thread to call `Object.notify()` or `Object.notifyAll()` on that object.
- A thread called `Thread.join(t)` & is waiting for either the timeout t to have elapsed, or for the specified thread to terminate.
- Any of the above & another thread calls `Thread.interrupt()` on the thread.

[**5 marks**]

[TOTAL 15]

Question 7

(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**]

[TOTAL 15]

Question 8

See solution of the Dining Philosophers problem in the appendix of the ICT.

(a) The integer `Fork` array, the value in `Fork[i]` is the number of forks currently available for philosopher i to pick up, this can be 0, 1 or 2. [**1 mark**]

(b) If a philosopher does not have sufficient forks available to eat, i.e. < 2, then it "waits" until it has 2 available. So if philosopher i has < 2 forks it calls `wait( OK_to_eat[i] )`, thus releasing the fork monitor & waiting on the condition variable `OK_to_eat[i]`. [**2 marks**]

(c) The left & right neighbour philosophers to philosopher 2 update the number of forks available to it, i.e. philosophers 1 & philosopher 3. [**2 marks**]

(d) When a philosopher (j say) puts down its 2 forks it: increments the number of forks available to its left (j+1) & right (j - 1) neighbour philosophers, it checks if the left (j+1) neighbour philosopher now has 2 forks available, if it does then it "signals" this to it by calling `signal( OK_to_eat[j+1] )`, similarly for its right (j - 1) neighbour philosopher. [**3 marks**]

(e) `Line 6:  Condition[] OK_to_eat ;`

This is declaring an array of "condition variables", these are queues of processes that are waiting on some "condition". The intention is that the condition must be true before a process can complete the called monitor method.

The Java equivalent is "wait-set" associated with every monitor & synchronisation lock. [**4 marks**]

(f) The "`signal(condVar)`" method is used to signal that the "condition" associated with the condition variable `condVar` is now true. The result is that just the process at the front of the `condVar` queue is signalled, i.e. woken-up & removed from the queue.

The Java equivalent is `Object.notify()` which only "wakes up" one thread that is waiting in the monitor's "wait-set". It is not `Object.notifyAll()`, as this wakes up all the threads in the monitor's "wait-set".

[**3 marks**]

[TOTAL 15]

END OF THE IN-CLASS TEST PAPER