UNIVERSITY OF WESTMINSTER

MARKING SCHEME
DO NOT COPY

SCHOOL OF
COMPUTER SCIENCE & ENGINEERING

Module Title: **Concurrent Programming**

Module Code: **6SENG002W**

Exam Period: **January 2021**

Time Allowed: **3.5 Hours**

INSTRUCTIONS FOR CANDIDATES

PLEASE WRITE YOUR STUDENT ID CLEARLY AT THE TOP OF EACH PAGE.

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.

THIS PAPER MUST NOT BE TAKEN OUT OF THE EXAMINATION ROOM

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

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 *Labelled Transition Graph* that is an abstract representation of the program. [**1 mark**]

[**PART Total 4**]

(b) **(i)** Possible `COUNTDOWN` process actions:

- "i" has the value 30:

  Actions: `tick` & `reset`, because the boolean `when` guard for
  "`when ( i == 0 ) soundBuzzwer -> STOP`"
  is false & so this choice's `soundBuzzwer` action is not offered, but the `when` guard for
  "`when ( i > 0 ) tick -> COUNTDOWN[ i - 1 ]`"
  is true, so this choice's `tick` action is offered. The `reset` action is always offered as it has no explicit guard, so is treated as being true. [**5 marks**]
- "i" has the value 0:

  Actions: `soundBuzzwer` & `reset`, because the boolean `when` guard for
  "`when ( i == 0 ) soundBuzzwer -> STOP`"
  is true & so this choice's `soundBuzzwer` action is offered. The `reset` action is always offered. The `when` guard for
  "`when ( i > 0 ) tick -> COUNTDOWN[ i - 1 ]`"
  is false, so this choice's `tick` action is not offered. [**5 marks**]

[**SUBPART Total 10**]

(ii) Just define `HARD_EGG_TIMER` as composite process that calls the `EGG_TIMER` process with the correct parameter value of 500.

```
const FiveMinutes = 300

|| HARD_EGG_TIMER = ( EGG_TIMER( FiveMinutes ) ) .
```

[**SUBPART Total 3**]

(iii) *Alphabet* – the alphabet of an FSP process is the set of (visible) actions it can perform. [**1 mark**]

```
alphabet(EGG_TIMER) = { tick, soundBuzzer, reset }
```

[**1 mark**]
[**SUBPART Total 2**]

(iv) In FSP *deadlock* means that a process ends up in a state that has no out transitions, i.e. it can not perform any actions. In FSP the deadlocked process is represented by the process `STOP`. [**2 marks**]
The deadlock trace for the `EGG_TIMER` process is:

```
<tick, ..., tick, soundBuzzer >
```

that is 240 `ticks` followed by `soundBuzzer`. [**3 marks**]
In the definition of `COUNTDOWN` replace the FSP deadlock process `STOP` with the successfully terminating process `END`. [**1 mark**]
[**SUBPART Total 6**]

[**PART Total 21**]

(c) **(i)**

```
OYSTER = ( topUpOyster -> twentyPounds
           -> toppedUp -> printReceipt -> OYSTER ) .
```

[**3 marks**]

(ii)

```
TICKET
  = (   zone12 -> fourPounds -> printTicket -> TICKET
      | allZones -> tenPounds -> printTicket -> TICKET
    ) .
```

[5 marks]

[**PART Total 8**]

[**QUESTION Total 33**]

Question 2

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

Sets of Train Processes Labels:

```
set Trains = { e, w }
```

[**2 marks**]

Two Train processes:

```
EastTrain
= ( leaveWestStation -> trackA -> lock_trackC -> trackC
    -> unlock_trackC -> trackD -> arriveEastStation -> STOP ) .
```

[**6 marks**]

```
WestTrain
= ( leaveEastStation -> trackE -> lock_trackC -> trackC
    -> unlock_trackC -> trackB-> arriveWestStation -> STOP ) .
```

[**6 marks**]

Shared Track C's mutual exclusion locking & unlocking:

```
SharedTrack = ( lock_trackC -> trackC -> unlock_trackC -> SharedTrack ) .
```

[**4 marks**] .

The complete Railway system: Trains & Shared Tracks:

```
|| TRAINS = ( e:EastTrain || w:WestTrain ) .
```

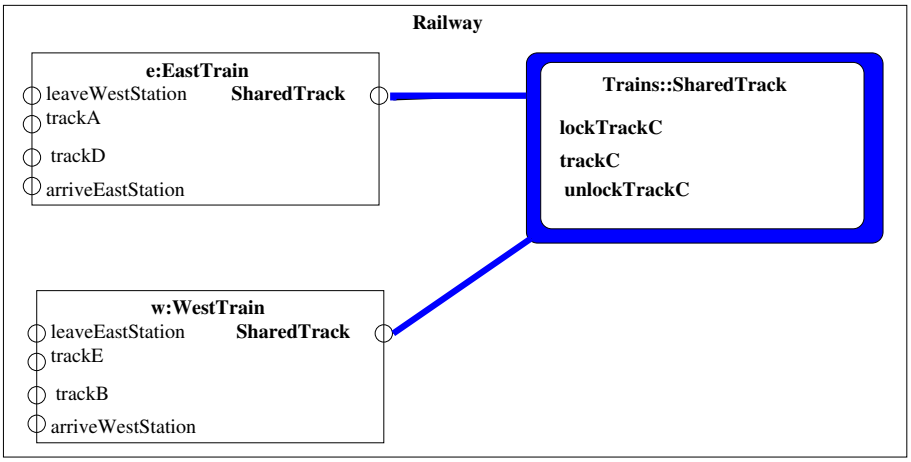
[**3 marks**]

```
|| Railway = ( TRAINS || Trains::SharedTrack ) .
```

[**3 marks**]

[**PART Total 24**]

(b) There are many structure diagrams that would describe the railway system, a possible one is the following.



The main points are that the two trains and the shared track are represented with consistent actions and they share the synchronising locking and unlocking actions. [**5 marks**]

[**PART Total 5**]

(c) Mutual exclusive access to the shared railway track section is achieved by requiring the two train processes to "lock" it before they can travel over it, by performing a synchronised `lockTrackC` action with the shared track process. [**2 marks**] When they have finished with it they must "unlock" it, by performing a synchronised `unlockTrackC` action with the shared track process. [**2 marks**]

[**PART Total 4**]

[**QUESTION Total 33**]

Question 3

(a) Note this is just the main points that students may give, but other relevant information will be marked accordingly.

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. **[2 marks]**

But a thread is not a program, it cannot run on its own, but runs within a program. **[1 mark]**

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

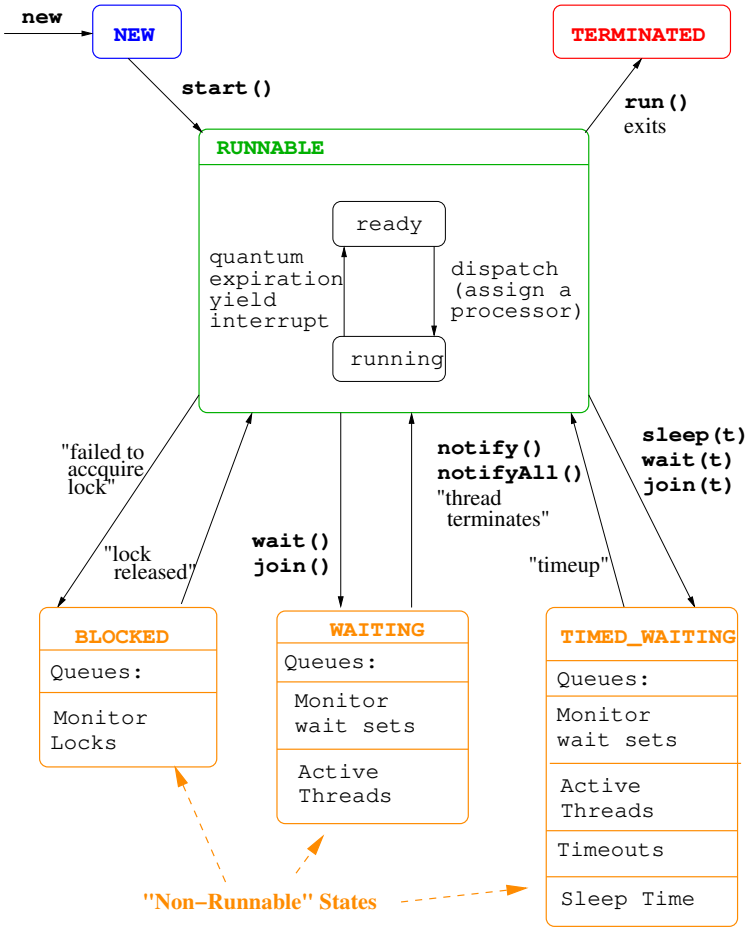
[1 mark]

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 6]

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



[3 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 14]

(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[2]` has 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[1]` have priority 7 & `racer[3]` has priority 2. **[1 mark]**

The *"Runnable"* thread with the highest priority is chosen for execution, i.e. `racer[0]` or `racer[1]`. If there are two threads of the same priority waiting for the CPU, the scheduler chooses them in a round-robin fashion. **[1 mark]**

So one of `racer[0]` or `racer[0]` 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]**

Only when both have stopped or become *"Not Runnable"*, will a lower priority thread start executing. Thus `racer[0]` & `racer[1]` both run to completion before either `racer[2]` or `racer[3]` start. **[1 mark]**

Only after `racer[0]` & `racer[1]` have terminated, the next highest priority threads will run, i.e. just `racer[2]` runs uninterrupted to completion. **[1 mark]**

Then finally the lowest priority thread `racer[3]` runs uninterrupted & to completion, & the program end. **[1 mark]**

all of then

The thread race output is `racer[0]`'s & `racer[1]`'s would be strictly interleaved, followed by all of `racer[2]`'s & finally all of `racer[3]`'s. E.g.

```
Racer[0], i = 10
Racer[1], i = 10
Racer[0], i = 20
Racer[1], i = 20
Racer[0], i = 30
Racer[1], i = 30
Racer[2], i = 10
Racer[2], i = 20
Racer[2], i = 30
Racer[3], i = 10
Racer[3], i = 20
Racer[3], i = 30
```

[**4 marks**]

[**PART Total 13**]

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

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. [**1 mark**]

[**PART Total 6**]

(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)       // [2 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]
```

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

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

[**SUBPART Total 14**]

(ii) Explanation:

All ME processes must share the same binary semaphore. So declare it & pass it to each process via its constructor. [**2 marks**]

The binary semaphore must be initialized to 1 (unlocked), otherwise there would be immediate deadlock. [**1 mark**]

Each process that wants to access the ME resource must first use the shared semaphore's claim/acquire to lock it, & then gains ME access, when finished use, it must use the same shared semaphore's release to unlock it. [**2 marks**]

This leaves the resource available for the other threads to access it again. [**1 mark**]

[**SUBPART Total 6**]

[**PART Total 20**]

[**QUESTION Total 33**]