UNIVERSITY OF WESTMINSTER

MARKING SCHEME
DO NOT COPY

SCHOOL OF
COMPUTER SCIENCE & ENGINEERING

Module Title: **Concurrent Programming**

Module Code: **6SENG002W, 6SENG004C**

Exam Period: **January 2020**

Time Allowed: **2 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) "+IN" is the FSP operator alphabet extension "P+Actions", where Actions is a set of actions that are added to the alphabet of the process P. **[2 marks]**

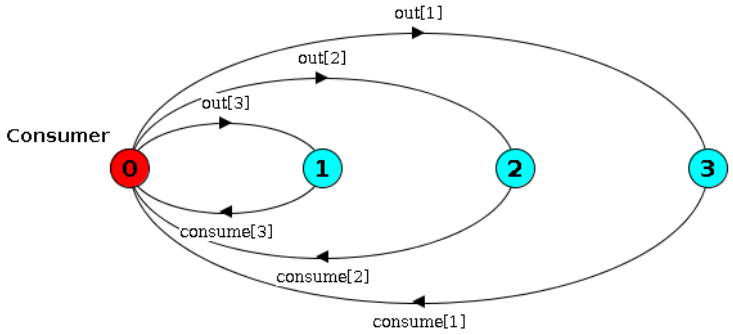
The effect of "+IN" on Producer_2, is to add the actions: in[1], in[2], in[3] to the alphabet of Producer_2. But only in[1], in[3] are additions as in[2] is already in its alphabet. **[3 marks]**

[PART Total 5]

(b)
```
Consumer = (   out[ 1 ] -> consume[1] -> Consumer
             | out[ 2 ] -> consume[2] -> Consumer
             | out[ 3 ] -> consume[3] -> Consumer
           ) .
```

[3 marks]

The LTS for the Consumer process.



[4 marks]

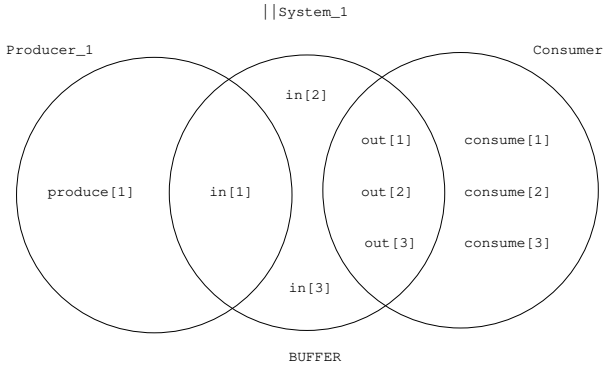
[PART Total 7]

(c) Note in[DATA] is equal to in[1], in[2], in[3].

```
alphabet( Producer_1 ) = { in[1], produce[1] }                  //[1]

alphabet( Producer_2 ) = { in[1], in[2], in[3], produce[2] }    //[2]
                       = { in[DATA], produce[2] }

alphabet( BUFFER )     = { in[1..3], out[1..3] }                //[2]

alphabet( Consumer )   = { out[1..3] , consume[1..3] }          //[2]
```

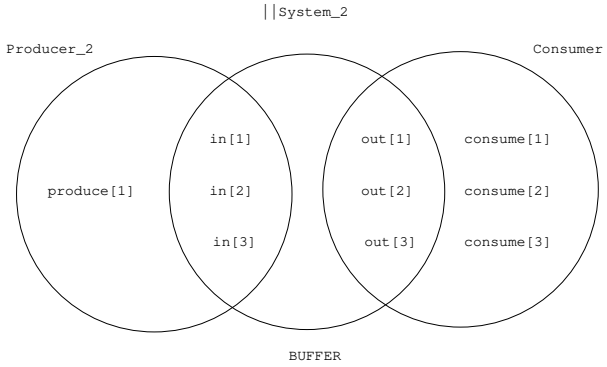
[**PART Total 7**]

(d) **(i)** SYSTEM_1 *Alphabet* diagram.



[**5 marks**]

(ii) SYSTEM_2 *Alphabet* diagram.



[**5 marks**]

(iii) In SYSTEM_1 BUFFER's actions are performed as follows:

Action	Type	Processes
in[1]	Synchronous	Producer_1
in[2], in[3]	Asynchronous	
out[1], out[2], out[3]	Synchronous	Consumer

In SYSTEM_1 BUFFER can perform all input & output actions, but only in[1] is synchronised with the Producer_1, the other are done independently of Producer_1. [**2 marks**]

In SYSTEM_2 BUFFER's actions are performed as follows:

Action	Type	Processes
in[2]	Synchronous	Producer_1
in[1], in[3]	Blocked	Producer_1
out[1], out[2], out[3]	Synchronous	Consumer

In SYSTEM_2 the only input action BUFFER can perform is `in[2]` & consequently the only output action is `out[2]`, as it never input 2 or 3. [**2 marks**]

[**PART Total 14**]

[**QUESTION Total 33**]

Question 2

(a) There are obviously many FSP processes that could be used to model this shared ATM, two customers and a cashier. However, the main requirement is to ensure the mutually exclusive use of the shared ATM. This has to be done by using a synchronised locking/unlocking ATM or Lock process that they all use; & using alphabet extension on the users to force the ATM to either sycnhronise or block its actions. The following is a *sample solution*. marks will be awarded for similarity to the key aspects of this solution.

Declarations.

```
const MAX_ATM_CASH = 3
range ATM_CASH     = 0 .. MAX_ATM_CASH
range AMOUNT       = 1 .. 5
```

[**2 marks**]

Sets of Actions & Process Labels.

```
set ATM_Actions = { start, finish,
                    withdraw[AMOUNT], dispensed[ AMOUNT ],
                    refillATM
                  }

set ATM_Users = { nigel, jack, jill }
```

[**4 marks**]

Two approaches for the shared ATM:

1. Use 2 separate process: one deals with the cash withdrawals, e.g. `withdraw[ amt : AMOUNT ]` & `dispensed[ amt ]` actions. The other is just a `start` & `finish` locking process to ensure ME. This is simpler than (2).

```
ATM_LOCK = ( start -> finish -> ATM_LOCK ) .

ATM_MACHINE = ATM_MONEY[ MAX_ATM_CASH ] ,

ATM_MONEY[ cash : ATM_CASH ]
  = (   withdraw[ amt : AMOUNT ] -> dispense[ amt ]
                                  -> ATM_MONEY[ cash - amt ]
      |
        refillATM -> ATM_MONEY[ MAX_ATM_CASH ]
    ) .

|| ATM = ( AMT_LOCK || ATM_MACHINE ) .
```

[6 marks]

2. Use one process that integrates the ATM actions & the start/finish actions into one process using a collection of mutually recursive local processes. This is more sophisticated than (1).

```
ATM = ATM_FREE[ MAX_ATM_CASH ] ,

ATM_FREE[ cash : ATM_CASH ] = (   start -> ATM_INUSE[ cash ]
                                | shutdown -> END ) ,

ATM_INUSE[ cash : ATM_CASH ]
  = (   withdraw[ amt : AMOUNT ]
                    -> // dispense cash & finish
      |
        refillATM -> finish -> ATM_FREE[ MAX_ATM_CASH ]
    ) .
```

[9 marks]

A `Customer`'s behaviour: acquire ME control of ATM, withdraw cash, release ME contorl of ATM. The `Cashier`'s behaviour is similar.

```
Customer( AMT = 1 ) = ( start ->
                          withdraw[ AMT ] -> // get cash, etc
                        finish -> END
                      ) +ATM_Actions .
```

[5 marks]

```
Cashier = (   start -> refillATM -> finish -> Cashier
            | finishwork -> END ) +ATM_Actions .
```

[**3 marks**]

[**PART Total 23**]

(b) The complete ATM systems:

```
|| Customers = ( jack : Customer( 2 )  ||  jill : Customer( 3 ) ) .

|| BANK_ATM = (     ATM_Users :: ATM
                 || Customers
                 || nigel : Cashier
       ) .
```

[**4 marks**]

[**PART Total 4**]

(c) Mutual exclusive access to the shared ATM is achieved by requiring the two customers & cashier processes to "lock" it before they use it, by performing a synchronised `start` action with the ATM process. [**2 marks**] When they have finished using it they must "unlock" it, by performing a synchronised `finish` action with the ATM process. [**2 marks**]

In addition, the `start` & `finish` actions must be labelled by the user to ensure individual user/ATM access. Plus alphabet extension of the user's alphabtes to block the ATM from performing actions asynchronously. [**2 marks**]

[**PART Total 6**]

[**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:

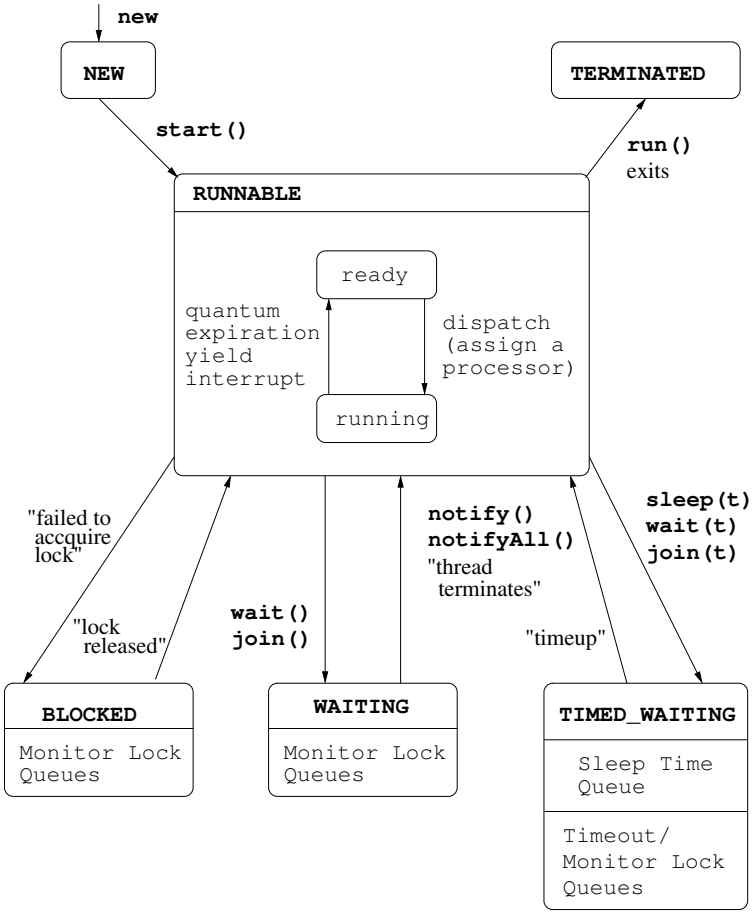
```
class SubClass extends SuperClass implements Runnable
{
   private Thread t ;

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

   // ‘‘run()’’ method used by t
   public void run(){ ... }
}
```

  [2 marks]

Two Thread Creation Methods: there are two ways to create a thread: `extend Thread` class & `implement Runnable` interface, because if your class needs to subclass (`extend`) some other class, e.g. `Applet`, then you cannot use the first method, therefore you need to have another method, i.e. the second one. **[1 mark]**

[PART Total 9]

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

The purpose of the `ThreadGroup` class is to help programmers organise & manage collectuions of threads. Every Java thread is a member of a *thread group*. When you create a thread, you can either: allow the run time 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 run time system creates the `ThreadGroup` "`main`", & unless specified otherwise, all new threads are members of `main`. Because the run time 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**]

Additional features that could be added to the `ThreadGroup` class are: methods for starting, returning their thread state, and appropriately terminating them. Since all threads must be started, have a state 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/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; *public methods* that implement operations on the resource & define its interface; 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) (i) Execution of `MessageSystem` program:

1. Initially `mb` is unlocked & its wait-set is empty. `p` & `r` are created & are in the `NEW` state. **[1 mark]**

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

[SUBPART Total 7]

(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 12]

[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) Something similar to the following is expected, but any code that has the correct elements is acceptable: creates, intialises & uses 2 binary semaphore as required, creates T1 & T2 threads & has the correct use of "acquires" & "releases". Students may fail to initialise the Java semaphores correctly, e.g. miss out the "`T2_semaphore.drainPermits()`". Also if they use "claim" instead of Java's "acquire" is acceptable, & will ignore minor syntax erros etc.

```
import java.util.concurrent.Semaphore ;

class T1 extends Thread                          // [1]
{
  private final Semaphore T1_ProgressSema ;      // [1]
  private final Semaphore T2_ProgressSema ;      //

  T1( Semaphore t1sema, Semaphore t2sema )       // [1]
  {
     super("T1") ;                               //
     this.T1_ProgressSema = t1sema ;             // [1]
     this.T2_ProgressSema = t2sema ;             //
  }

  public void run()
  {
     try{
           T1_ProgressSema.acquire() ;           // [1]
           System.out.println( "T1: Action 1") ;
           T2_ProgressSema.release() ;           // [1]

           T1_ProgressSema.acquire() ;           // [1]
           System.out.println( "T1: Action 3") ;
```

```
            T2_ProgressSema.release() ;          // [1]
         } catch ( InterruptedException ie ){} ;
  } // run
} // T1
```

[**8 marks**]

```
class T2 extends Thread
{
  private final Semaphore T1_ProgressSema ;
  private final Semaphore T2_ProgressSema ;

  T2( Semaphore t1sema, Semaphore t2sema )
  {
     super("T2") ;
     this.T1_ProgressSema = t1sema ;
     this.T2_ProgressSema = t2sema ;             // [1]
  }

  public void run()
  {
     try{
           T2_ProgressSema.acquire() ;           // [1]
           System.out.println( "T2: Action 2" ) ;
           T1_ProgressSema.release() ;           // [1]

           T2_ProgressSema.acquire() ;
           System.out.println( "T2: Action 4" ) ;
           T1_ProgressSema.release() ;           // [1]

        } catch ( InterruptedException ie ){} ;
  }
} // T2
```

A lot of duplication in T2, as its very similar to T1. [**4 marks**]

```
class OrderActions_T1_T2
{
  private static final int NumberOfPermits = 1 ; // Binary Semaphore
```

```
  public static void main (String args[])
  {
    // create two binary semaphores to control progress of T1 & T2
    Semaphore T1_semaphore = new Semaphore( NumberOfPermits ) ;  // [1]
    Semaphore T2_semaphore = new Semaphore( NumberOfPermits ) ;  // [1]

    // T1_semaphore = 1, initial acquire (claim) succeeds
    // T2_semaphore = 0, initial acquire (claim) block
    int numpermits = T2_semaphore.drainPermits() ;               // [1]

    // create & start the 2 Threads
    Thread T1 = new T1( T1_semaphore, T2_semaphore ) ;  // [1]
    Thread T2 = new T2( T1_semaphore, T2_semaphore ) ;  // [1]

    T1.start() ;                                        // [1]
    T2.start() ;                                        //
  }
} // OrderActions_T1_T2
```

[**6 marks**]

[**PART Total 18**]

(c) Ordering the actions of T1 & T2:

Use 2 binary semaphores, T1_semaphore & T2_semaphore initialised to 1 & 0 respectively. [**1 mark**]

Each semaphore is used to control the progress of one thread (T1), by requiring it to acquire (claim) the semaphore (T1_semaphore) before doing its actions & its only released by the other thread (T2).

	T1_semaphore	T2_semaphore
T1	acquires	releases
T2	releases	acquires

[**2 marks**]

- T1's progress is blocked by having to acquire T1_semaphore
- T1 allows T2 to progress by releasing T2_semaphore
- T2's progress is blocked by having to acquire T2_semaphore
- T2 allows T1 to progress by releasing T1_semaphore

[2 marks]

Since `T1_semaphore` is initialised to 1, `T1`'s initial acquire suceeds so it can proceed, but since `T2_semaphore` is initialised to 0, `T2`'s initial acquire fails & is has to wait. **[1 mark]**

[PART Total 6]

[QUESTION Total 33]