# **Question 1**

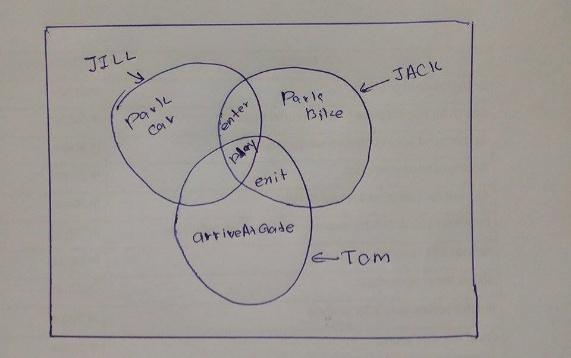
1. What is concurrent programming?

*The* ***subprograms*** *can be executed concurrently and* ***can overlap in time****, with* ***no ﬁxed order****. (lecture 1, page 1.21)*

*The execution of a program is termed a process and the execution of a concurrent program thus consists of multiple processes. (lecture 2, page 2.7)*

# **Question 2**

## Venn diagram



From outside the playground they can see only parking area

3 people Jack, Jill, Tom -> primitive process

Park car, park bike, arrive at gate are actions

1. Write a FSP for each primitive process

*// IMPORTANT: JILL should be upper case and the function names should be camel case and definitely start with a lowercase letter*

*JILL = ( parkCar -> enter -> play -> STOP ) .*

*TOM = ( parkBike -> enter -> play -> exit -> STOP ) .*

*JACK = ( arriveAtGate -> play -> exit -> STOP ) .*

1. Write a FSP for each composite process

*||PLAYGROUND = ( JILL || TOM || JACK ) .*

1. Write process alphabet for all the actions of the composite process in the diagram

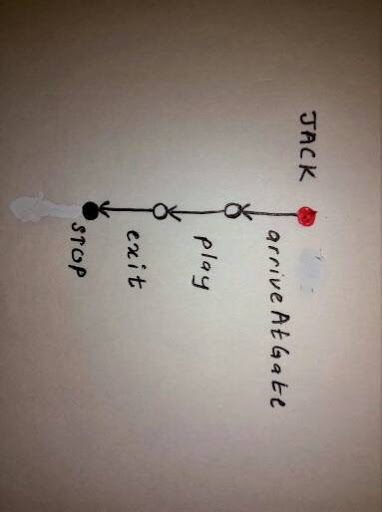
*alphabet(JILL) = {enter, parkCar, play}*

*alphabet(TOM) = {enter, exit, parkBike, play}*

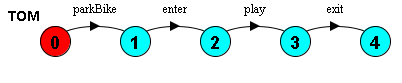
*alphabet(JACK) = {arriveAtGate, exit, play}*

*alphabet(PLAYGROUND) = {arriveAtGate, enter, exit, parkBike, parkCar, play}*

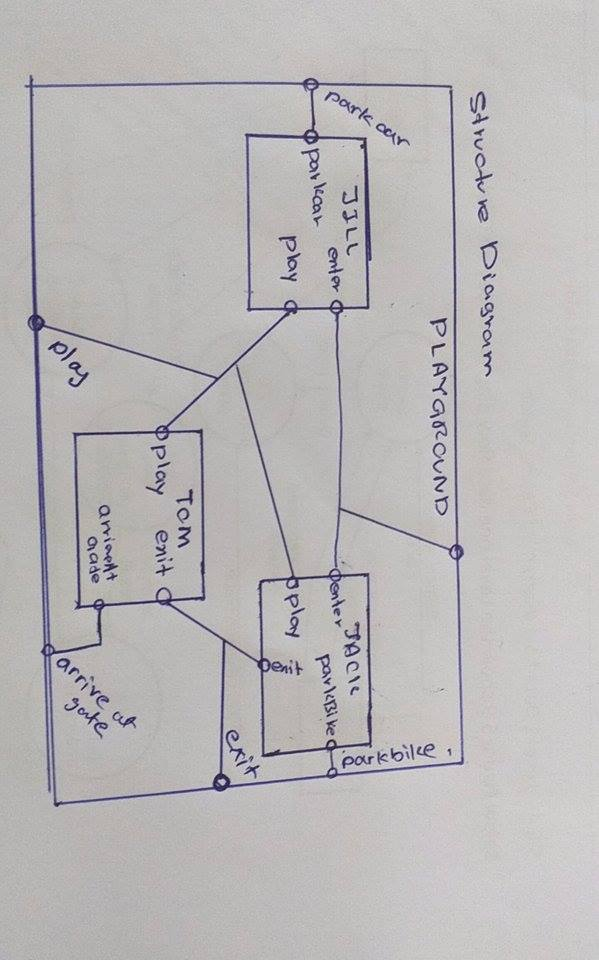
1. Draw the trace tree for Jack



1. Draw LTSA graph for Tom



1. Draw a structure diagram for the composite process.



1. Briefly explain the following constructs used in FSP
   1. Single pipe (|)

*This is the Choice operator. If x and y are actions; P & Q are processes, then (x -> P | y-> Q) describes a process which initially engages in either of the actions x or y. After the first action has occurred, the subsequent behavior is described by P if the first action was x and Q if the first action was y. (2013, question 1)*

* 1. STOP

*STOP is a special predefined process that engages in no further actions, it is used to represent the “deadlocked” process. (2013, question 1)*

* 1. When (Boolean expression)

*This is a guard for a guarded action. The choice (when (B) x -> P | y -> Q) means that when the boolean guard B is true the actions x and y are both eligible to be chosen; false the action x cannot be chosen. (2013, question 1)*

* 1. -> explain the symbol (transition)

*-> is action prefix. If x is an action and P a process then the action prefix (x -> P) describes a process that initially engages in the action x and then behaves P. (2013, question 1)*

(Marks 33)

# **Question 3**

1. Describe the two methods by which a programmer can create a thread in a Java program. Illustrate your answer by means of suitable code fragments.
2. - 3 - a, 2013 - 3 - a, answers were taken from 2013 marking scheme)
   * *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. 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]*

* 1. How would you decide which method to use?

*Decision: create a thread using the Runnable interface if your class needs to subclass some other class, e.g. Applet; otherwise use sub-classing.*

* 1. What is concurrent programming?

*Refer answer of question 1*

*[1 mark]*

* 1. How join works?

*It means that calling t.join() makes the caller thread wait for the thread t to finish its execution.*

*[1 mark]*

*(Marks 8)*

1. Write a concurrent java program which consists of 2 threads, the 1st thread prints numbers 1-100 and the 2nd thread the letters a-z. Start both threads. Explain what is happening.

(Marks 12)

class ThreadOne extends Thread {

  public void run(){

    for (int i = 0; i < 100; i++){

        System.out.println(i+1);

    }

  }

}

class ThreadTwo extends Thread {

  public void run(){

    for (char ch = 'a'; ch <= 'z'; ch++) {

   System.*out*.println(ch);

}

  }

}

public class TestClass{

  public static void main(String[] args){

    Thread threadOne = new ThreadOne();

    Thread threadTwo = new ThreadTwo();

       threadOne.start();

       threadTwo.start();

  }

}

*Both the threads will be at the ready state. One of the threads will be given the chance to execute (running state) while the other thread waits (ready state) then the other thread will be given the chance to execute and the currently running thread will be put to the ready state. Therefore, both the threads will be executed* ***according to the scheduling scheme in java runtime*** *depending on the priority of the thread.*

*Output: Unpredictable.*

1. Modify the code you have written in previous question. So that the 2nd thread waits for the 1st thread to terminate before it begins the output. Predict the output.

(Marks 6)

public class TestClass{

  public static void main(String[] args){

    Thread threadOne = new ThreadOne();

    Thread threadTwo = new ThreadTwo();

threadOne.start();

    try {

   threadOne.join();

} catch (InterruptedException e) {

   e.printStackTrace();

}

threadTwo.start();

  }

}

*Output: prints 1- 100 and then prints a -z*

1. Modify the subsection (3), so that each thread sleeps for 1 second after printing a number or a letter. Explain the output.

Ex: Output cannot be determined.

class ThreadOne extends Thread {

  public void run(){

    for (int i = 0; i < 100; i++){

        System.out.println(i+1);

         try {

            Thread.sleep(1000);

         } catch (InterruptedException e) {

            e.printStackTrace();

         }

    }

  }

}

class ThreadTwo extends Thread {

  public void run(){

    for (char ch = 'a'; ch <= 'z'; ch++) {

     System.*out*.println(ch);

  try {

      Thread.*sleep*(1000);

  } catch (InterruptedException e) {

      e.printStackTrace();

  }

}

  }

}

public class TestClass{

  public static void main(String[] args){

    Thread threadOne = new ThreadOne();

    Thread threadTwo = new ThreadTwo();

       threadOne.start();

       try {

   threadOne.join();

  } catch (InterruptedException e) {

   e.printStackTrace();

  }

       threadTwo.start();

  }

}

(Marks 7)

(Marks 33)

# **Question 5 - Semaphore (Lecture 9)**

1. What do you understand by the term semaphore? (Operations: claim(), acquire(), release())

*A semaphore is a programming language mechanism that is used in concurrent programming to achieve the following:*

* ***mutual exclusion***
* ***processes synchronization***

*Together these enable concurrent processes to safely share resources and coordinate their activities.*

* ***claim(s)—if s > 0 then s := s - 1 & the process can continue, otherwise the process is suspended on s.***

***while s = 0 do skip ; // do nothing***

***s := s - 1 ;***

* ***release(s) — if some process P has been suspended by a previous claim(s) then wake up P, otherwise***

***s := s + 1.***

***If a number of processes have been suspended on s then select one of them to be woken up one, the others remain suspended.***

***For binary semaphores the definition of release(s) is changed to s := 1.***

* ***initialize(s, v) — initialize the semaphore s with the value v, s := v.***

1. What are the different types of semaphores?

●        ***General semaphore***

●        ***Binary semaphore***

1. What are the differences between 2 types of semaphores?

|  |  |
| --- | --- |
| ***General semaphore*** | ***Binary semaphore*** |
| ***general semaphores can take on any value greater than 0. (However, general semaphores are (almost) always defined to have a maximum value, that they must not exceed.)*** | ***binary semaphores can only take on the values 0 or 1.*** |

1. Write the code for semaphore using the producer consumer problem

***References:*** [***http://www.javamadesoeasy.com/2015/03/semaphore-used-for-implementing.html***](http://www.javamadesoeasy.com/2015/03/semaphore-used-for-implementing.html)

import java.util.concurrent.Semaphore;

public class ConsumerProducer{

  public static void main(String[] args) {

         Semaphore semaphoreProducer=new Semaphore(1);

         Semaphore semaphoreConsumer=new Semaphore(0);

         System.*out*.println("semaphoreProducer permit=1 | semaphoreConsumer permit=0");

     Producer producer=new Producer(semaphoreProducer,semaphoreConsumer);

     Consumer consumer=new Consumer(semaphoreConsumer,semaphoreProducer);

      Thread producerThread = new Thread(producer, "ProducerThread");

      Thread consumerThread = new Thread(consumer, "ConsumerThread");

      producerThread.start();

      consumerThread.start();

  }

}

/\*\*

\* Producer Class.

\*/

class Producer implements Runnable{

  Semaphore semaphoreProducer;

  Semaphore semaphoreConsumer;

  public Producer(Semaphore semaphoreProducer,Semaphore semaphoreConsumer) {

         this.semaphoreProducer=semaphoreProducer;

         this.semaphoreConsumer=semaphoreConsumer;

  }

  public void run() {

         for(int i=1;i<=5;i++){

             try {

                 semaphoreProducer.acquire();

                 System.*out*.println("Produced : "+i);

                 semaphoreConsumer.release();

             } catch (InterruptedException e) {

                   e.printStackTrace();

             }

         }

  }

}

/\*\*

\* Consumer Class.

\*/

class Consumer implements Runnable{

  Semaphore semaphoreConsumer;

  Semaphore semaphoreProducer;

  public Consumer(Semaphore semaphoreConsumer,Semaphore semaphoreProducer) {

         this.semaphoreConsumer=semaphoreConsumer;

         this.semaphoreProducer=semaphoreProducer;

  }

  public void run() {

         for(int i=1;i<=5;i++){

             try {

                 semaphoreConsumer.acquire();

                 System.*out*.println("Consumed : "+i);

                 semaphoreProducer.release();

             } catch (InterruptedException e) {

                   e.printStackTrace();

             }

         }

  }

}

(Marks 20)

(Marks 33)