Answer Question 1 and any other two questions

1. Answer the following four parts

   a. Explain in less than 50 words the concept of transactions and then define the four characteristic properties of transactions in less than 20 words each.

   [10 marks]

   Transactions are sequences of operations that are clustered together into a unit. Transactions have ACID properties which means that they are atomic, consistency preserving, isolated and durable. Atomicity means that the transaction is executed completely or not at all. Consistency preserving means that the transaction leads from one consistent state to the other. Isolation means that the effect of the transaction is not visible to concurrent transaction. Durability means that the effect of the transaction persists once it has been completed.

   b. Explain the two-phase locking protocol. Does it achieve serialisability? How can deadlocks occur and how are they dealt with?

   [8 marks]

   The first phase of 2PL acquires all locks that are needed for a transaction and the second phase releases them. Serialisability is achieved if no further locks are acquired after the first lock has been released. Two-phase locking achieves serialisability but is not deadlock free. Databases usually detect deadlocks and resolve them by aborting transactions.

Answers NOT TO BE PRINTED

c. Two transactions are run concurrently against a bank's database. The first transaction transfers funds from one account to a second account. The second transaction sums up the balances of both accounts. Use FSP to model two-phase locking in these transactions and a lock compatibility matrix that includes both Read and Write locks.

[8 marks]

```
const Read=0
const Write=1
range LOCK=Read..Write
range NUMLOCKS=0..9
ACCOUNT = ACCOUNT[0][0],
ACCOUNT[reading:NUMLOCKS][writing:NUMLOCKS] = (
  when (writing==0) lock[Read] -> ACCOUNT[reading+1][writing]
 |when (reading==0 && writing==0)
                       lock[Write]->ACCOUNT[reading][writing+1]
 |when (reading>0) unlock[Read] -> ACCOUNT[reading-1][writing]
 |when (writing>0) unlock[Write] -> ACCOUNT[reading][writing-1]
 |credit -> ACCOUNT[reading][writing]
 |debit -> ACCOUNT[reading][writing]
 |balance -> ACCOUNT[reading][writing]).
||ACCOUNTS = (a:ACCOUNT|| b:ACCOUNT).

TRANSFER = (a.lock[Write]-> b.lock[Write] ->
            a.credit -> b.debit ->
            a.unlock[Write] -> b.unlock[Write]->TRANSFER).

BALANCE = (a.lock[Read]-> b.lock[Read] ->
           a.balance -> b.balance->
           a.unlock[Read] -> b.unlock[Read]->BALANCE).

||BANK = (ACCOUNTS || TRANSFER || BALANCE).
```

d. Extend the above example to specify serializability of the two transactions as an FSP safety property. How would you use the Labelled Transition System Analysis Tool to prove serialisability?

[8 marks]

```
property SERIALIZABILITY=(a.credit->b.debit->SERIALIZABILITY
                         |a.balance->b.balance->SERIALIZABILITY).
```

To prove serializability using LTSA, compose `SERIALIZABILITY` with `BANK` and perform a safety check.

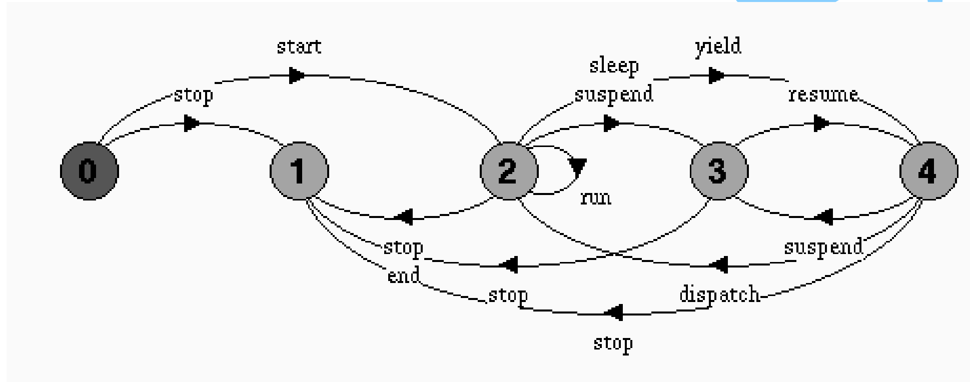
[Total 34 marks]

Answers
NOT TO BE PRINTED

2. Threads in Java – Answer the following three parts

   a. In the C340 course, we have used Java as an example programming language that supports concurrency. Model Java threads as a labelled transition system (LTS). Use that model to explain the semantics of Java threads.

   [16 marks]



A created thread can be in four different states. It can be running (2), runnable (4), not runnable (3) and terminated (1). A Java thread is started by operation start. A running thread may explicitly yield processor time and become runnable. It may implicitly be forced to sleep and become runnable by the Java virtual machine. A runnable thread may be dispatched by the virtual machine and become running again. A running or runnable thread may suspend execution and become not runnable. A not runnable thread can resume execution and then it becomes runnable. A thread in any state terminates through a stop operation.

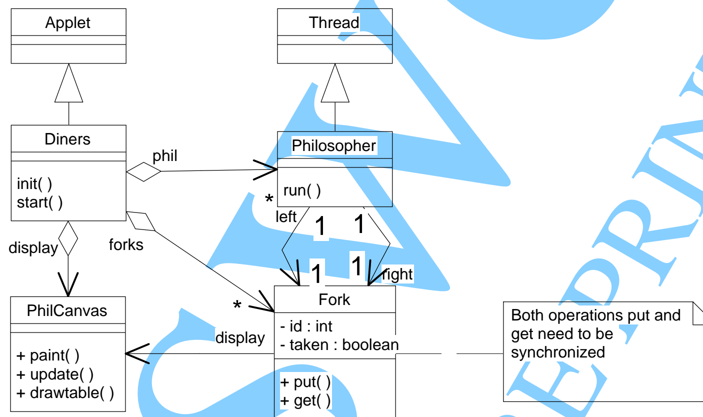
Answers
NOT TO BE PRINTED

b. Consider the following FSP specification of the dining philosophers problem:

```
PHIL=(hungry->left.get->right.get->eating->
      left.put->right.put->thinking->PHIL).
FORK = (left.get-> left.put -> FORK |
        right.get->right.put -> FORK).
||COLLEGE(N=5)= (phil[0..N-1]:PHIL||fork[0..N-1]:FORK)
 /{phil[i:0..N-1].left/fork[i].left,
   phil[i:0..N-1].right/fork[((i-1)+N)%N].right}.
```

Design a UML class diagram for a concurrent implementation of that specification in a Java applet. Define classes, attributes, operations and relationships. Identify those operations that have to be synchronized and argue why this need arises.

[8 marks]



Answers
NOT TO BE REPRINTED

c. The above FSP specification is unsatisfactory because it results in a deadlock if all philosophers pick up the left fork at the same time. If the design you provided in response to the above question is a proper implementation of the model it will have the same flaw. The deadlock can be avoided if one philosopher always picks up the right fork first, but this has the flaw that it is unfair. Find a fair strategy for deadlock avoidance and modify both the FSP specification and your design so that they are both fair and deadlock-free.

[9 marks]

One solution is to make the two actions of picking up left and right forks atomic. The FSP model changes into:

```
PHIL=(hungry->forks.get->eating->
      forks.put->thinking->PHIL).
FORK = (left.get -> left.put -> FORK |
        right.get->right.put -> FORK).
||COLLEGE(N=5)= (phil[0..N-1]:PHIL||fork[0..N-1]:FORK)
 /{phil[i:0..N-1].forks/fork[i].left,
   phil[i:0..N-1].forks/fork[((i-1)+N)%N].right}.
```

In the design this involves changing the `Fork` monitor class into a class `Table` that maintains all chopsticks and can get and put the left and right chopsticks for a philosopher as an atomic action.

[Total 33 marks]

Answers NOT TO BE PRINTED

3. Answer the following four parts

a. Explain the following concepts in less than 30 words each

i. Deadlock
A process is in a deadlock if it is blocked waiting for a condition that will never become true

[2 marks]

ii. Livelock
A process is in a livelock if it is spinning while waiting for a condition that will never become true.

[2 marks]

iii. Liveness Property
A liveness property asserts that something good eventually happens.

[2 marks]

iv. Progress
A progress property asserts that whatever state a system is in, it is always the case that a specified action will eventually be executed.

[2 marks]

v. Fair Scheduling
If a transition from a set is chosen infinitely often and every transition in the set will be executed infinitely often, the scheduling policy is said to be fair.

[2 marks]

[Subtotal 10 marks]

Answers NOT TO BE PRINTED

b. Consider a small airport that has only one runway. The air traffic control tower of the airport needs to coordinate arriving flights. Pilots of incoming aircraft notify the air traffic controller that they are entering the airspace and ask for permission to land. The air traffic controller grants permission to land only to one aircraft at a time in the order they entered the air space. Model this air traffic control problem in FSP. You can assume that the airspace has restricted capacity.

[9 marks]

```
const MaxIF=4
range IF=0..MaxIF

ARRIVE=ARRIVE[0],
ARRIVE[num:IF]=(
        enterarrq[num]->ARRIVE[(num+1)%MaxIF]).

ARRIVALQ=(enterarrq[num:IF]->ARRIVALQ[num]),
ARRIVALQ[n0:IF]=(enterarrq[num:IF]->ARRIVALQ[n0][num]
                |perm_land[n0]->land[n0]->ARRIVALQ),
ARRIVALQ[n0:IF][n1:IF]=(enterarrq[num:IF]->ARRIVALQ[n0][n1][num]
                       |perm_land[n0]->land[n0]->ARRIVALQ[n1]),
ARRIVALQ[n0:IF][n1:IF][n2:IF]=(
         enterarrq[num:IF]->ARRIVALQ[n0][n1][n2][num]
        |perm_land[n0]->land[n0]->ARRIVALQ[n1][n2]),
ARRIVALQ[n0:IF][n1:IF][n2:IF][n3:IF]=(
         perm_land[n0]->land[n0]->ARRIVALQ[n1][n2][n3]).

||AIRPORT=(ARRIVE||ARRIVALQ).
```

Answers
NOT TO BE PRINTED

c. The airport also has aircraft departing from the airport. Arriving flights are given priority over departing flights. Extend the above FSP model in such a way that the air traffic control system manages incoming and outgoing flight. For safety reasons there must only be one aircraft on the runway.

[8 marks]

```
const MaxIF=4
range IF=0..MaxIF

ARRIVE=ARRIVE[0],
ARRIVE[num:IF]=(
        enterarrq[num]->ARRIVE[(num+1)%MaxIF]).

ARRIVALQ=(enterarrq[num:IF]->ARRIVALQ[num]),
ARRIVALQ[n0:IF]=(enterarrq[num:IF]->ARRIVALQ[n0][num]
                |perm_land[n0]->land[n0]->ARRIVALQ),
ARRIVALQ[n0:IF][n1:IF]=(
         enterarrq[num:IF]->ARRIVALQ[n0][n1][num]
        |perm_land[n0]->land[n0]->ARRIVALQ[n1]),
ARRIVALQ[n0:IF][n1:IF][n2:IF]=(
         enterarrq[num:IF]->ARRIVALQ[n0][n1][n2][num]
        |perm_land[n0]->land[n0]->ARRIVALQ[n1][n2]),
ARRIVALQ[n0:IF][n1:IF][n2:IF][n3:IF]=(
         perm_land[n0]->land[n0]->ARRIVALQ[n1][n2][n3]).

DEPART=DEPART[0],
DEPART[num:IF]=(
        enterdepq[num]->DEPART[(num+1)%MaxIF]).

DEPARTUREQ=(enterdepq[num:IF]->DEPARTUREQ[num]),
DEPARTUREQ[n0:IF]=(enterdepq[num:IF]->DEPARTUREQ[n0][num]
          |perm_takeoff[n0]->takeoff[n0]->DEPARTUREQ),
DEPARTUREQ[n0:IF][n1:IF]=(
           enterdepq[num:IF]->DEPARTUREQ[n0][n1][num]
          |takeoff[n0]->DEPARTUREQ[n1]),
DEPARTUREQ[n0:IF][n1:IF][n2:IF]=(
           enterdepq[num:IF]->DEPARTUREQ[n0][n1][n2][num]
          |perm_takeoff[n0]->takeoff[n0]->DEPARTUREQ[n1][n2]),
DEPARTUREQ[n0:IF][n1:IF][n2:IF][n3:IF]=(
           perm_takeoff[n0]->takeoff[n0]->DEPARTUREQ[n1][n2][n3]).

ATC=(perm_land[n:IF]->land[n] ->ATC
    |perm_takeoff[n:IF]->takeoff[n]->ATC).

||AIRPORT=(ARRIVE||ARRIVALQ||DEPART||DEPARTUREQ||ATC)
           <<{perm_land[IF]}.
```

Answers
NOT TO BE PRINTED

d. The air traffic control system of the airport is safety-critical. You therefore have to prove that your above FSP specification is correct. The correctness criteria that need to be checked are that (a) the model does not permit deadlocks and (b) incoming and outgoing aircraft can eventually land and take off, respectively. Modify your specification in such a way that your model can be checked against safety properties (a) and (b) in a fully automatic way by the LTSA.

[6 marks]

Nothing has to be changed for property (a). LTSA can check any model for potential deadlocks. Property (b) can be expressed as

```
progress ARRIVING={perm_land[IF]}
progress DEPARTING={perm_takeoff[IF]}
```

[Total 33 marks]

Answers
NOT TO BE PRINTED

4. Answer the following four parts

a. It is often important that the specification and implementation of concurrent systems guarantee certain properties. In less than a total of 200 words,

i. Define what safety properties are

[2 marks]

Safety properties assert that nothing 'bad' will ever happen during the execution of a concurrent program

ii. Give an example of a safety property

[2 marks]

There is never more than one green traffic light on a four way junction

iii. Indicate the principal ways to specify safety properties in FSP

[2 marks]

Either by safety property that is a process that is composed with the process to be checked, or by implicit transitions to an error state.

iv. Explain how safety properties can be verified using labelled transition system analysis

[3 marks]

An LTS is generated with an error state. Transitions to the error state are added for any action that would violate the property. Then reachability analysis is used to search for a path that would lead to the error state.

[Subtotal 9 marks]

Answers
NOT TO BE PRINTED

b. Imagine a cross-country railroad that only has a single track. It connects village N with village S via village M. Before the station at village M, the track forks and after the station of M the tracks join again, enabling trains to pass each other. You are being asked to write a formal model for the replacement of the railroad's signalling equipment. Use FSP to model the synchronization of rail traffic between S and N in both directions. You may assume that the tracks that lead to M and the station of M have a limited capacity.

[10 marks]

```
const MaxTrains=4
range Trains=0..MaxTrains
NM=NM[0][0],
NM[nb:Trains][mb:Trains]=
  (when (nb==0 && mb<MaxTrains) go_nm->NM[nb][mb+1]
  |when (mb==0 && nb<MaxTrains) go_n->NM[nb+1][mb]
  |when (mb>0) arr_at_m_from_n->NM[nb][mb-1]
  |when (nb>0) arr_at_n->NM[nb-1][mb]
  |when (nb>0&&mb>0) unsafe->ERROR).
SM=SM[0][0],
SM[mb:Trains][sb:Trains]=
  (when (mb==0 && sb<MaxTrains) go_s->SM[mb][sb+1]
  |when (sb==0 && mb<MaxTrains) go_sm->SM[mb+1][sb]
  |when (sb>0) arr_at_s->SM[mb][sb-1]
  |when (mb>0) arr_at_m_from_s->SM[mb-1][sb]
  |when (mb>0&&sb>0) unsafe->ERROR).
N=(go_nm->N
  |arr_at_n->N).
S=(go_sm->S
  |arr_at_s->S).
M=M[0][0],
M[nb:Trains][sb:Trains]=(
  when (sb<MaxTrains) arr_at_m_from_n -> M[nb][sb+1]
 |when (nb<MaxTrains) arr_at_m_from_s -> M[nb+1][sb]
 |when (sb>0) go_s -> M[nb][sb-1]
 |when (nb>0) go_n -> M[nb-1][sb]).
||COUNTRYRR=(NM||SM||N||S||M).
```

Answers NOT TO BE PRINTED

c. We do not want trains to crash on the single tracks. The railtrack company wants to ensure that by demanding there are no trains running in opposite directions on the same segment at any point in time. Extend the above model with either an explicit or implicit FSP safety property such that it can be proven fully automatically.

[5 marks]

Extend `NM` with a further non-deterministic choice

```
|when (nb>0&&mb>0) unsafe->ERROR)
```

extend `SM` with a further non-determistic choice

```
|when (mb>0&&sb>0) unsafe->ERROR
```

d. It is also undesirable for passengers to be stuck on the tracks or at M because of deadlock situations. Is your FSP model deadlock-free? What do you have to modify so that deadlocks do not occur?

[9 marks]

The above specification is not deadlock free. A deadlock can occur if both N and S send trains in the opposite directions. This situation can be resolved, by limiting the number of trains that are going into a particular direction to the number of trains that can be kept at station M. A replacement of processes `N` and `S` with the following resolves the situation.

```
N=N[0],
N[sb:Trains]=(
  when (sb<MaxTrains)go_nm->N[sb+1]
 |go_s -> N[sb-1]
 |arr_at_n->N[sb]).
S=S[0],
S[nb:Trains]=(
  when (nb<MaxTrains)go_sm->S[nb+1]
 |go_n -> S[nb-1]
 |arr_at_s->S[nb]).
```

[Total 33 marks]

Answers
NOT TO BE PRINTED

5. Answer all four parts

   a. Answer in less than 100 words each of the following questions

      i. What are the differences between operating system processes and threads in a programming language, such as Java?

      [5 marks]

      Threads are executed within a process. Threads are more light-weight than processes. Processes have an owner, they are protected against each other by the operating system, the operating system may collect accounting information for each process. The processes of a thread share the same owner, they share the memory allocated by the process.

      ii. Explain the principal ways to implement threads in Java.

      [5 marks]

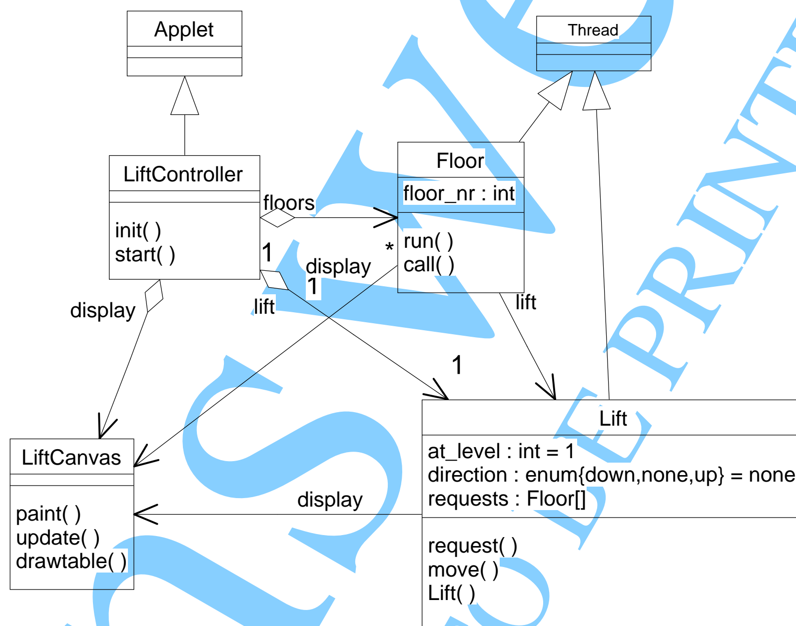
      Threads are implemented by extending class `Thread` or by implementing Interface `Runnable`. In both cases, the threads are Java objects. They start executing when the `start` method is invoked and may be interrupted.

   [Subtotal 10 marks]

Answers
NOT TO BE PRINTED

b. Your company won a contract to build the control software for the lifts of a number of four storey buildings. On each floor there are buttons to request the lift to arrive. These buttons are operated by users of the lift concurrently. The buttons signal to the user when the lift has arrived on the floor. The porters of the buildings where the lifts will be installed require a control display so that they can see how the lift is changing its position. We neglect the interior control panel of the lift. As the user interface shall be portable across platforms, your manager asks you to provide a design for the user interface using Java. Use the Unified Modeling Language to design this concurrent Java program. Include classes, relationships, attributes and operations that are necessary for the lift control applet.

[5 marks]



Answers
NOT TO BE PRINTED

c. Implement those parts of your design, which embed the control algorithms for the lift software. Use Java as a programming language.

[14 marks]

A working implementation is given below. Students would receive full marks if they demonstrate correct implementation of concurrent threads and implement the algorithms of all control operations `run` in `Floor` and `Lift`, an equivalent of `add_request`. properly.

```
public class LiftController extends Applet {
  ...
  public boolean handleEvent(Event event) {
    int i;
    Floor called;
    for (i=1; i<=Lift.MAXFLOORS; i++)
      if (event.target==callbuttons[i]) {
        called=new Floor(i,callbuttons[i],lift);
        called.start();
        return(true);
      }
    return(super.handleEvent(event));
  }
};

class Floor extends Thread {
  public int floor_nr;
  Lift lift;
  Button mybutton;

  public Floor(int nr, Button b, Lift l) {
    floor_nr=nr; lift=l; mybutton=b;
  };

  public void run() {
     lift.add_request(this); // do the request
     System.out.println("Arrived at floor "+floor_nr); // then die
  };
};

class Lift extends Thread {
  static public int MAXFLOORS=4;
  int at_level=1;
  DisplayCounter display;
  int direction=1; // -1 to go down, 1 to go up
  Floor[] requests=new Floor[MAXFLOORS+1];

  public Lift(DisplayCounter d){
    display=d;
  };

  synchronized public void add_request(Floor from) {
    requests[from.floor_nr]=from;
    while (from.floor_nr!=at_level)
      try {
        wait();
      } catch (InterruptedException e){}
    requests[from.floor_nr]=null;
```

Answers
NOT TO BE PRINTED

```
    }

    synchronized private boolean request(int dir) {
      // returns true if we have to go up
      int i;
      i=at_level;
      while (i<=MAXFLOORS && i>0){
        if (requests[i]!=null) {
          return(true);
        }
        i=i+dir;
      }
      return(false);
    }

    synchronized private void move(int dir) {
      at_level+=dir;
      if (dir>0) display.inc(); else display.dec();
      notifyAll();
    }

    public void run() {
      while (true) {
        try {
            sleep(5000);
          } catch (InterruptedException e){}
        if (request(direction))
          move(direction);
        else
          if(request(direction*(-1))){
            direction=direction*(-1);
            move(direction);
          }
      }
    }
  }
```

d. Discuss the following properties of your implementation and how you would prove them?

i. Possibilities of Deadlocks
There are no deadlocks in the above control algorithm. To prove this we could model the concurrent threads in FSP and use reachability analysis

ii. Liveness Properties
The lift eventually will arrive at the requested floor. To prove this use progress properties on the FSP model.

[4 marks]

[Total 33 marks]

END OF PAPER

Answers
NOT TO BE PRINTED