Answer question 1 and any other two questions.

1)

a) What is a condition variable and what operations can be performed on it? If a particular operation has more than one possible form of semantics identify each one. [10]

b) Using any of the primitives you identified above, write code for a monitor which provides shortest-job next allocation for a single resource. This monitor should have two operations `request(time : int)` and release(). When a process calls `request(time)`, it delays until the resource can be allocated to it. After acquiring the resource and using it, a process calls `release()`. The `request` operation will always allocate the resource to the requesting process that has specified the lowest value of `time` parameter. If there are no pending requests the resource is freed. [7]

c) A World-Wide-Web (WWW) navigation session is defined as the activity of browsing through a sequence of WWW pages by the process of following links. Such a sequence of WWW pages is called a *trail*. Let us assume that you have available a function called, *useful*, which given a set of keywords and a trail, returns a number between zero and one indicating how "good" the trail is.

Give the pseudo-code of a non-deterministic algorithm that would automatically find a trail whose usefulness is greater than a half. The input to your algorithm is a starting node for the navigation, a set of keywords and the required length of the trail to be found. Briefly explain the main idea behind your algorithm. [9]

d) Give the pseudo-code of a *probabilistic* solution to the dining philosophers problem. In what sense is this solution preferable to the solution using semaphores? [7]

[TURN OVER]

2)

a) What is meant by the term *common memory arbiter*? [4]

b) Describe Peterson's algorithm for mutual exclusion among many processes and explain its operation. [11]

c) The fetch and add instruction is an instance of an atomic machine instruction which performs two actions. You may regard it as having the following behaviour:

```
int FA(int &v) { return v++; }
```

Write a simple n-process mutual exclusion algorithm based on the fetch and add instruction. You need not worry about starvation. State any assumptions you make. [8]

d) A student's attempt at implementing a solution to the bounded buffer problem using semaphores is as follows:

```
process producer
  var i : item_type

  do true ->
    i := produce()
    P(mutex)
    add_into_buffer(i)
    V(mutex)
    V(not_empty)
  od
end
```

```
process consumer
  var i : item_type

  do true ->
    P(mutex)
    P(not_empty)
    i = remove_from_buffer()
    V(mutex)
    consume(i)
  od
end
```

Sadly, the student has committed several errors, and has completely omitted any initialisation information. Correct the student's code and add initialisation. You do not need to write the routines `produce`, `consume`, `add_into_buffer`, or `remove_from_buffer`. [10]

3) In a fairground there is a roller coaster. Suppose there are n passenger processes and one car process. The passengers repeatedly wait to take rides in the car which can hold C passengers ($C < n$). The car can only go around the tracks when it is full.

a) Develop code for the actions of the passenger and car processes using whatever synchronisation mechanism you choose. [16]

b) Generalise your answer to employ m car processes ($m > 1$). Since there is only one track, cars cannot pass each other, and must finish going around the track in the order they started. Again, a car can only go around the track when it is full. [17]

[CONTINUED]

4) The following shared-memory model for concurrent execution of programs has been suggested, where the shared memory is called the *pool*. The program behaviour is modelled as a game between *programmer* and *computer* as follows, where an *atom* (also referred to as a *program statement*) is a simple instruction or a test condition, and a *packet* is a set of atoms. The programmer makes the first move and then play alternates between the computer and the programmer. The programmer in their turn submits a packet of atoms to the pool for eventual execution and the computer in their turn selects a single atom from the pool and then executes it. The programmer may at any stage submit the empty packet, denoted by *skip*, and may also submit *end* to denote that no more packets will be submitted. The computer may decide at any stage to *wait* instead of selecting an atom for execution.

Answer the following three parts relating to above model.

a) Show how sequential and concurrent execution of program statements can be enforced by the programmer. [11]

b) What would you consider to be *fair* play by the computer? [11]

c) Consider the following pseudo-code in a concurrent programming language, where **cobegin** (S_1) // ... // (S_n) **coend** specifies the concurrent execution of a sequence S_1, ..., S_n of programs:

```
1.  x := true; y := 1;

2.  cobegin (while x do y := y + 1) end while // (x := false) coend;
```

Outline how the programmer should play to model the above program so as to ensure its eventual termination, given that the computer is playing *fairly*. [11]

[TURN OVER]

5) Answer the following three parts.

a) Define the notion of serialisability using an example of two transactions that are serialisable and another two transactions that are *not* serialisable. [11]

b)

i) Can deadlock always be avoided in the two-phase locking protocol? Give an example to illustrate your answer. [5]

ii) Show how deadlock can be detected in the two-phase locking protocol. [6]

c) In a client-server model, communication between a client and a server can be realised via the *remote procedure call* (RPC) model. In the synchronous RPC model the client executes a statement that calls a procedure, and then, prior to continuing execution, waits for the called procedure to be executed by the server and return control back to it.

Suggest some problems with this model, in the event of client or server failure, and how queues may be used to solve these problems. [11]

[END OF PAPER]