State whether the following are TRUE or FALSE. Give a brief explanation for your answer. One mark for correct answer, and one mark for the explanation.

1. False.
   A deadlock is guaranteed by the presence of a cycle in the resource-allocation graph, only in the case where there is a single instance of each resource type. In the multiple-instance case, the cycle does not imply a deadlock because one process in the cycle might be holding one instance of a resource while its predecessor process on the cycle is making a claim. That claim is however possible servicable by another instance which is free, or being held by someother process not waiting for any resources.

2. False.
   The medium-term scheduler’s job is to swap some processes already present in main memory out for sometime, in order to be scheduled later. It does this because the memory is overloaded or some resources are blocked and the concerned process is stalling or not functioning efficiently. It may also need to create vacancy if a high priority process needs to be scheduled by the long-term scheduler. In this case the medium-term scheduler needs to decide which process to send out of main memory.
3. **True.**
   A system which remains in a safe state can never enter a deadlock. Since the process moves from an unsafe state and then back again to a safe state and subsequently never leaves the safe state, it can never be involved in a deadlock.

4. **False.**
   In SRTF, a process can never be preempted in favour of another process which was present in the ready queue, at the time when the process was allotted the CPU. It could get preempted by processes entering the ready queue during the course of its current burst on the CPU.

5. **False.**
   During a context-switch all variables associated with a process are copied from the registers and CPU data area to the main memory. There is no exchange of values of multiple variables.

6. **False.**
   The purpose of implementing a small block of code as an atomic instruction is to ensure data integrity. It is used for that part of a code where the process is manipulating shared data, and the modification of such data should not be left incomplete.

7. **True.**
   In a priority scheduling environment, each process comes with an integer in a fixed range indicating its priority. Either preemptively or non-preemptively, the next process is selected on the basis of priority values. In a constant stream of higher priority processes, a low priority process may never get executed. Thus, the aging policy increases the priority of each process with the passage of time, so that even the lowest priority processes eventually get executed when their priority reaches the maximum value.

8. **False.**
   The number of processes executing at a time is bounded by the number of processors. This number has nothing to do with the number of register sets available.

9. **True.**
   This is achieved by the following code, where the semaphore variable $mutex$ is initialised to 0.

   \[ P_1 \]
$I_1$
signal(mutex)
$P_2$
wait(mutex)
$I_2$

10. **True.**
The dispatch latency (delay caused by context-switches) can be reduced if a system has multiple register-sets rather than just a single set of registers, if the number of processes in memory is at most the number of register-sets.

11. **False.**
If a process terminates either at or before the end of its time-slice, then it is moved out of the memory, its resources are released, and the next process in the ready queue is given the CPU. Processes other than the terminating one are not sent out of the main memory temporarily.

12. **True.**
A priority inversion protocol protects low priority processes from pre-emption in favour of higher priority processes, at the time when they are modifying data structures which are shared with processes at that higher priority level, because if that is not done, then the shared data could be in an inconsistent state.

13. **True.**
When a process is created, the instructions are at the level of machine language code, and not at a high level or assembly level. The instructions are thus specific to the processor type it is compiled for. If one attempts to run the process on a different processor, then either the instructions wont be understood by that processor, or it will misunderstand the instructions. In either case, the resulting execution will not conform to what was designed.

14. **False.**
Hold-and-wait is a necessary condition for deadlocks to arise. Since the process is not currently holding any resources, the hold-and-wait condition does not hold currently. Thus there can be no deadlock currently.
15. **True.**
It is not possible to determine by any direct calculation, the length of
the next CPU-burst of a process. Thus, this is estimated statistically
on the basis of the burst-lengths in its history (previous runs on the
CPU).

**Section B**

(4 X 5=20 marks)

16. Describe the life cycle of a process in a *multiprogramming*
environment, including a description of its various states.

**Solution.**
The various states of a process are *new, ready, running, waiting* and
*terminated*.

A new process is one which has just been loaded into the main memory.
It has its variables, PCB data initialised and its resources allocated.

A ready process is one which is waiting to be allocated time on the
CPU to execute. A process here either comes from the new state after
all the initialisation work is done, or from the running state due to
an interrupt/preemption or from the waiting state after completion of
the service routine for which it was waiting.

A running process is one which is currently executing on a CPU. A pro-
cess moves to this state from the ready state when the CPU schedul-
er selects it.

A process enters the waiting state, when it has either invoked a sub-
routine and is waiting for its return before resuming its own execution
or it has made a request for an I/O operation and can resume exe-
cution after the completion of that operation. Processes move to this
state from the running state.

A process is in the terminated state when it either finishes execution
normally, or has been terminated abnormally due to some error, over
use of resources or completion of its useful functioning. Processes enter
this state from the running state in case of normal termination and
from any of the other states in case of exceptional termination.

17. Describe briefly the roles of the short-term (CPU) scheduler, the long-
term (job) scheduler and the medium-term scheduler. Specify the
circumstances under which the medium-term scheduler is invoked.
Solution.
In a multiprogramming environment, the CPU-scheduler (or short-term scheduler) decides the next process to be run on the CPU, from among those waiting for their turn in the ready queue. This set of processes is obviously a subset of the processes loaded into main memory. The CPU-scheduler also decides if and when to confiscate the CPU from a currently running process in order to allot it to another process. The CPU-scheduler needs to make these decisions very quickly in view of the very short lengths of typical CPU-bursts. If it is not quick, then the fraction of time spent on context-switch overhead will approach the fraction spent doing computational work, which leads to low system utilisation. Thus the algorithms used to check the criteria for the selection of the next process need to be quick and efficient even at the price of some loss of accuracy.

The job (long-term) scheduler selects processes to be loaded from secondary storage to main memory in order to get a chance to execute. This scheduler is needed because all programs cannot normally be present in main memory at the same time, due to limits on the available memory. This scheduler controls the degree of multiprogramming, and also selects the process to be loaded so as to maintain a good balance of CPU-intense and I/O-intense processes. It can use less efficient algorithms to accurately select the next process on the basis of the criteria specified. This is because it is invoked relatively infrequently. Except at system startup it is normally invoked only when a process is terminated and moved out of main memory or the medium-term scheduler moves some process out of main memory temporarily. If there is a preemptive priority protocol in place, then it might come into play when a high priority process enters the job pool.

The medium-term scheduler acts as a swapper, and its role is to move a process out of main memory, and the execution of that process is resumed later when the long-term scheduler decides to send it to main memory again. The medium-term scheduler is invoked when either there is an unexpected rise in memory usage or drain of other system resources leading to reduced performance. It might also be invoked when there is a poor balance of CPU-intense and I/O-intense processes, which is regulated at the loading time by the job-scheduler. The medium-term scheduler is responsible for deciding which process needs to be sent out, when the need arises. It makes its decision on the basis of the length of the process, the fraction of its execution
completed and its priority among other factors.

18. Describe the general structure of any program with a critical section. In particular, list the significant units in its code and briefly describe the role of each unit.

**Solution.**

**Code for process a process with a critical section**

```
repeat
    ENTRY SECTION
    CRITICAL SECTION
    EXIT SECTION
    REMAINDER SECTION
until false
```

The **looping** reflects the fact that a process with a critical section could execute its critical section multiple times interleaved with its remainder section.

The **entry section** determines which process will enter its critical section, since multiple processes are forbidden from entering their critical sections at the same time. In particular, it determines when the current process enters its critical section.

The **critical section** is the portion of code where a process modifies data shared with other processes. In the interests of data consistency, only one process should be permitted to access to its critical section at a time.

The **exit section** is the portion of code where a process signals that it has finished its current stint in its critical section, and thus the next waiting process can enter its critical section in a mutually exclusive manner (since this process has finished its current critical section phase).

The **remainder section** is the portion of code where a process does not modify or use data shared with other processes. In this phase, the process is like an independent process.

19. List the FOUR approaches to deadlocks and describe them briefly in a few lines. State the advantage and disadvantage of each of them.

**Solution.**

The four approaches to handling the deadlock problem are:
• Deadlock Prevention.
In this approach, the system policy on resource allocation is designed in such a way that at least one of the four necessary conditions for deadlocks (mutual exclusion, hold-and-wait, no preemption, circular wait) does not hold. An advantage of this approach (apart from the obvious one that deadlocks do not arise in the system) is that the allocation policy is simple and not time consuming. A disadvantage is underutilisation of resources.

• Deadlock Avoidance.
In this approach, every process is required to declare its maximum needs of each resource type at the time of its creation. The deadlock avoidance algorithm always maintains the system in a ”safe state” (a certain type of state in which deadlocks never arise). If a request for resources is made by a process, the system checks if the request is legal (within the stated limits of the process), feasible (sufficient resources are available) and also for safety (if the making the allocation still maintains the system in a safe state). The request is granted only if all three conditions are met. The advantage is that deadlocks never arise, but the drawbacks are that each process is required to know its future requirements in advance and the overhead of the safety state checking algorithms and associated bookkeeping is quite high.

• Deadlock Detection and Recovery.
In this approach, no algorithm is in place to ensure deadlocks do not arise. Instead the system runs deadlock detection algorithms, either periodically, or when system performance deteriorates, to see if a deadlock has arisen. If a deadlock is detected, then the system recovers from the deadlock either by resource preemption or process termination.

If resources are preempted then the processes from which they were reclaimed can be partially rolled back to their last safe state. If processes are terminated, then all the computation associated with them needs to be reperformed. The advantage of this approach is that the frequency of invocation of the algorithm is far lower than in deadlock avoidance. The disadvantage is that if a deadlock is detected then computation gets wasted whether there is rollback or termination. Also the detection algorithm is expensive. The system also needs to decide the process or set of processes to terminate, and the resources to be preempted. These
decision algorithms are expensive.

- Deadlock Ignorance.

In this approach, the system does nothing to ensure deadlocks do not arise nor to take any concrete action if one does arise. The effect is that when the number of blocked processes and resources rise beyond a point the system eventually crashes. The advantage is that there is no overhead nor underallocation of resources. This approach works on the assumption, that the frequency of deadlocks is so low that it does not warrant the cost incurred in detection or prevention/avoidance algorithms. The disadvantage is that when a deadlock occasionally arises there is loss of computation.

**Section C**  

(2 X 10=20 marks)

20. Consider a system consisting of $m$ resources of the same type, being shared by $n$ processes. Resources can be requested and released by processes only one at a time. Show that the system is deadlock free if the following conditions hold:

(a) The maximum needs of each process is between 1 and $m$.

(b) The sum of all maximum needs is less than $m + n$.

**Solution.**

In order to prove that a system is deadlock free, it is sufficient to prove that none of the processes in the system is involved in a deadlock. Clearly, we need only consider processes holding resources, since hold-and-wait is a necessary condition.

Let us assume that all the $m$ resources are currently being held among a set of $k$ processes. The number of processes not holding any resource is thus $n - k$. The total of maximum requirements of that set of processes is at least $n - k$. Thus the total of the maximum resources of the processes currently holding at least one resource each is at most $m + n - 1 - (n - k) = m + k - 1$. Thus, the total deficit (difference between maximum need and current holding) among these processes is at most $m + k - 1 - m = k - 1$. Since total deficit is at most $k - 1$ and number of processes is $k$, there is at least one process which is saturated (deficit=0). Such a process cannot be waiting for any resources, and hence the hold-and-wait condition fails. We conclude that the system can never be involved in a deadlock.
21. What are the three requirements of a critical section protocol? Describe them. Prove that the following implementation meets all the requirements in a two process critical section problem. The processes are \( P_i \) and \( P_j \).

**Solution.**

The three requirements of a critical section protocol are:

- **Mutual Exclusion**
  At most one process can be executing its critical section at a time. Other processes need to wait.

- **Progress.**
  If a set of processes are waiting to enter their critical sections, then the decision of which process enters the critical section next cannot be determined by any process outside this set. Also this decision cannot be postponed indefinitely.

- **Bounded Waiting**
  After a process \( P \) makes a request to enter its critical section, at most \( k \) other processes can enter their critical sections, before \( P \) is allowed to enter its critical section. Here \( k \) is a fixed positive integer.

In the given code, a process can enter its critical section only if either it is the only process waiting to enter its critical section (and the other process is in its remainder section) or the turn variable is its number. Thus at most one process can enter its critical section at a time. Note that if one process is already in its critical section then the turn variable is pointing to it. If the other process has to enter then the turn variable needs to be changed to it. That is only possible by the process in the critical section and this change of the turn variable happens only when that process leaves the critical section and enters the exit section. Thus, mutual exclusion is satisfied.

If both processes are in the entry section, then they are both entitled to decide the next process to enter. This decision is made immediately on the basis of the turn variable and the corresponding process enters the critical section. If there is exactly one process in the entry section and the other process is in the remainder section, then the flag values ensure that the process in the entry section immediately enters the critical section, irrespective of the value of turn. Thus it is the only process making the decision and the progress condition is satisfied.
If a process $P_i$ makes a request to enter the critical section, then if the other process $P_j$ is granted entry, it is because of the turn variable’s value. This value is changed at the exit section after $P_j$ finishes its critical section, and subsequently $P_j$ can enter its critical section only after $P_i$’s request has been granted. Thus each process needs to wait for at most one other process’s entry into critical section, before its own request is granted. Thus bounded waiting requirement is satisfied.