BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE, PILANI
TEST-1, FIRST SEMESTER, 2019-2020
ADVANCED OPERATING SYSTEMS (CS G623)

DATE: 14th Sep 2020 **MAX MARKS: 10** **WEIGHTAGE: 10%** **TIME: 30 Min**

Q.1. Let us assume that in a distributed system there are *R* shared resources (each resource has a single instance) and *N* processes (*R*<*N*). Each of the *N* processes require all or the sub-set of *R* resources to complete their task. There are unique locks corresponding to each of the *R* resources which should be acquired in order to use the resources. We have to design a mutual exlusion algorithm wherein there is a centralized process a.k.a *manager process* that maintains the location of *current lock owner*. The lock stays with the *current owner* (or at the location of the *current owner*) even after the lock is released, but has not been requested by some other process. Each lock is initially placed/located at the *manager process*, but it is not held by the *manager process*. The algorithm is as follows:

Assumptions:

(a) The channel is reliable
(b) No failures of processes or links
(c) Messages latency is bounded, but the bounds are not known.
(d) Each process only needs the resource for a limited amount of time.
(e) Size of the critical section is fixed.
(f) The number of lock reacquires in a loop is limitted on each process.

Algorithm to Acquire the Lock:

(a) If some process requires a lock, it needs to send acquire() message to the *manager process*.
(b) The location or the *current owner ID* of the lock is returned by the *manager process*.
(c) The requesting node contacts the owner directly.
(d) The lock owner creates a local FIFO queue of process ID's with acquire requests it has received.

Algorithm to Release the Lock (executed by lock owner):

(a) Dequeue the FIFO queue to read the process ID of the process who is next in line to become lock owner (*NextOwnerID*).
(b) Send *NextOwnerID* to the *manager process*.
(c) Pass the lock and the remaining queue to the process with the ID equal to NextOwnerID.

Q. Is this algorithm correct? Argue that the algorithm is correct or give a counter-example. [2 M]

Answer: The algorithm suffers from race condition: A node gets the current owner from the manager, but, its request to the current owner arrives after the lock and the queue has already been passed to a new owner. The algorithm does not specify what to do in this case, so, the assumption is that the late requesters will keep waiting for the lock forever (bounded waiting condition of the process execution is violated). It should be noted that the old owner cannot simply forward such delayed requests, because the lock may have changed owners. **[Marking Scheme: 2 M for race-condition, 1 M for any other relevant limitation, 0 otherwise]**

Q.2. Suppose there is a network of processes wherein every process knows about itself and all of its immediate neighbors. Provide example how these processes can exchange information to discover the global topology of the network. **[2 M]**

Answer: It is a relatively simple question. Here, each process broadcasts its own identifier and those of its immediate neighbors to every other process in the network. The algorithm for broadcast is not important. Processes can use the collected information to determine the global topology of the network. **[Marking Scheme: 2 M for correctly disseminating topological information in the network, 0 otherwise]**

Q.3. Several distributed computing systems have a primary objective of resource sharing. Give one example of distributed system where shared resource is a disk, a network bandwidth, and a processor. **[2 M]**

Answer: File server, Ethernet LAN, processes in a multiprogrammed OS **[Marking Scheme: 2 M for all correct examples, 1 M for two correct examples, 0 otherwise]**

Q.4. A semaphore is an object with two atomic methods **Wait** and **Signal**, a private integer counter and a private queue. The semantics of a semaphore is very simple. Suppose ***S*** is a semaphore whose private counter has been initialized to a non-negative integer.

- When **Wait** is executed by a process, we have two possibilities:
  The counter of *S* is positive. In this case, the counter is decreased by 1 and the process resumes its execution.

The counter of *S* is zero. In this case, the process is suspended and put into the private queue of ***S***.

- When **Signal** is executed by a process, we also have two possibilities:
  The queue of *S* has no waiting thread. The counter of ***S*** is increased by one and the thread resumes its execution.
  The queue of *S* has waiting threads. In this case, the counter of ***S*** must be zero. One of the waiting processes will be allowed to leave the queue and resume its execution. The thread that executes **Signal** also continues.

Suppose we have three processes P, Q, and R. The three processes are allowed to execute their critical section in the following order: Process P executes first and then process Q and process R can execute in any order and then process P executes again. This cycle continues indefinitely. **Write pseudo (code) to synchronize these three processes.** **[4 M]**

Answer:

Let us define three semaphore variables:

Semaphore s1, s2, s3;

s1 = 2, s2 = 0, s3 = 0

Process P Code

```
while(true){
        wait(s1);
        wait(s1);
        <critical section>
        signal(s2);
        signal(s3);
}
```

Process Q Code

```
while(true){
        wait(s2);
        <critical section>
        signal(s1);
}
```

Process R Code

```
while(true){
        wait(s3);
        <critical section>
        signal(s1);
}
```

[Marking Scheme: 2 M for correct code of process P, 1 M each for the correct code of process Q and R, no binary marking]