BGN: 2191A

P5

Q4

a.
   i. Whether write requests have higher priority than read requests, and so readers might have to wait for the write to finish even if the read was requested first, or whether read requests have higher priority, and so the writer might have to wait for an arbitrary number of reads to complete, as more reads be requested even while the writer is waiting.

      Alternatively, all requests could have the same priority and be served FIFO.

      Perhaps different readers might have different priorities.

   ii. If there are two simple locks, then this could ensure that a write cannot occur when a read is being performed, and vice versa, while allowing for concurrent reads. (Technically this could also allow concurrent writes but the question states that the resource is only updated by one writer at a time).

      The same effect could be achieved with a specialised lock primitive, but such a primitive could also include information about the priority of the lock-holder. For example, if a low-priority read is taking place, and then a high-priority write is requested, then the priority of the reader could be boosted to high, so that a middle-priority reader could not starve the high-priority writer (priority inheritance to prevent priority inversion).

      The specialised locking primitive could also keep track of how long a lock has been held for, timing out reads/writes which are taking an unexpectedly long time.

      Furthermore, when a write occurs, the primitive could store a copy of the pre-write resource to serve to any readers while the write lock is being held.

   iii. In general, only a partial order is required. Any two locks which might be held by the same agent at the same time must be ordered, but two locks which can never be held at the same time by the same agent need not be comparable. In this particular example, it is unlikely that the same client will perform a read and a write concurrently.

   iv. The timeout transition (wherein a lock is automatically released if it is held for a certain amount of time) should allow somebody requesting a greater lock to do so, instead of passing the lock directly to the next requester of the same lock. This would prevent deadlock.

For example: consider two agents each wanted to acquire locks A and B. Agent 1 intends to act in the following order (A<B<C):

1. Agent 1 acquires lock A
2. Agent 1 acquires lock B
3. Agent 1 performs some long-running process
4. Agent 1 releases lock A
5. Agent 1 acquires lock A
6. Agent 1 releases both locks

If, between steps 3 and 4, lock A times out and is released, Agent 2 would be able to acquire lock A. Agent 1 would block on step 5 and deadlock would occur. To resolve this, lock B must temporarily be lower in the lock ordering than lock A in the moments following the timeout, allowing agent 2 to block on lock B until agent 1 has finished.

b.

i. In the recursive call, the program will try to reacquire the lock it is already holding and cannot release it until it has done so. This causes deadlock.

ii. This is possible but impractical and leads to the potential for hard-to-spot bugs. This would indeed solve the problem and prevent deadlock, as the lock the program tries to acquire is already held by itself, and so does not block. However, presumably the lock is released somewhere else in the function block. Suppose the lock is released during the recursive call, and then the recursive call returns to the main call. The function might look something like this:

```
Func F(x) {
        Lock.acquire();
        …
        F(y);
        …                               // Point A
        Lock.release();
```

The programmer might assume that at point A, the lock is held, as it is between the acquisition and release lines. However, in this scenario the lock has been released by point A.

A modification to this approach which would solve this problem would be for the lock to store a count of how many times the same program has acquired it, and then when the program tries to release it, decrements the counter. When the counter reached 0, the lock is released.

c.

i. This pseudocode assumes both setValue and getValue are atomic.

```
// postMessage(queue, message)
// receieveMessage(queue)
// isEmpty(queue)

func setValue(varname, value) {
          postMessage(varname, value);     // Post the new value as a message

          // The message queue is determined by the variable name
}

func getValue(varname) {
          x = null;  // If the variable has never been "set", return null
          while (!isEmpty(varname)) {      // Get the most recent message on that variable's queue
                    x = receieveMessage(varname);
          }
          setValue(varname, x); // Push the value back on the queue so the next reader can access it
          return x;
}
```

ii. Emulation of a semaphore would be possible but not worth it. The message queue could contain all of the programs which want to acquire the semaphore, and there could be some central agent (perhaps elected using the Bully algorithm) who counts the number of incoming requests and sends each requester either a “ACCEPTED” or “REJECTED” message depending on whether the number of programs currently holding the semaphore has exceeded the limit.

This, however, is impractical. This is because you would have to handle the case of dropped messages or dropped replies, polling, request idempotency, etc. It is likely that the messaging protocol itself relies on some locking primitives (such as simple locks) which would be much better suited to implement a semaphore or other locking primitives.