Question 1.  [7 marks]
Each of the following statements is false. In one sentence, explain why. (If you disagree that the statement is false, provide your reasoning.)

Part (a)  [1 mark] Malloc is a system call.

Malloc sometimes calls sbrk or mmap which are system calls, but often malloc operates only as a function call. The data structures it manages are at user-level, and it only needs to make a system call if it runs out of memory to allocate.

Part (b)  [1 mark] An advantage of threads is that all threads can access the entire address space.

Threads cannot access each other’s stack.

Part (c)  [1 mark] Locks and semaphores are identical structures.

Semaphores have a variable associated with them. More than one thread can pass through a sem_wait call if the value of the semaphore is greater than 1. (They need to say something that shows that they know what semaphores are.)

Part (d)  [1 mark] A system which uses a preemptive scheduling algorithm is just as efficient (ratio of useful work to operating system overhead) as a system which uses non-preemptive scheduling.

The overhead of context switches is greater in a preemptive system.

Part (e)  [1 mark] Round-robin scheduling is a good policy for minimizing turnaround time.

Round-robin is good for minimizing response time, but because it cycles through all the processes, it extends their overall time to completion.

Part (f)  [1 mark] Limited direct execution means that once the OS has prepared a process for execution, the process has full control of the CPU.

The process still cannot execute privileged instructions.

Part (g)  [1 mark] The producer/consumer problem is a special case of the readers/writers problem with one reader and one writer.

The consumer is modifying the shared data, whereas the reader is only observing. The producer is adding to the shared data while the writer could be modifying any part of the shared data. This leads to a different synchronization strategy.
Question 2.  [8 marks]

Consider adding synchronization to the following working implementation of linked list `lookup()` and `insert()` functions using a single lock for the whole list.

```c
1 struct node *create_node(int value) {
2     struct node *n;
3     n = malloc(sizeof(struct node));
4     n->val = value;
5     n->next = NULL;
6     return n;
7 }
```

```c
8 int lookup(struct list *L, int value) {
9     struct node *cur = L->head;
10    while(cur != NULL) {
11        if(cur->val == value) {
12            return 1;
13        }
14        cur = cur->next;
15    }
16    return 0;
17 }
```

```c
18 void insert(struct list *L, int value) {
19    struct node *newnode = create_node(value);
20    struct node *cur = L->head;
21    LOCK()
22    if(L->head == NULL || L->head->val > value) {
23        newnode->next = L->head;
24        L->head = newnode;
25        UNLOCK
26        return;
27    }
28    while(cur->next && cur->next->val <= value) {
29        cur = cur->next;
30    }
31    newnode->next = cur->next;
32    cur->next = newnode;
33    UNLOCK()
34    return;
35 }
```

Part (a)  [2 marks] Is it necessary to lock `create_node`? Explain.

No. The data in `create_node` is not shared until it is added to the list.

Part (b)  [2 marks] Add lock and unlock calls to `insert` so that the lock is held for the minimum time. -1 if `LOCK` is in the wrong place, -0.5 for each `UNLOCK`

Part (c)  [2 marks]

If there are multiple threads calling `insert` and `lookup` concurrently, is locking `lookup` required? Explain carefully, referring to the code.

No. A thread calling `lookup` will always be able to correctly traverse the list. It just may not see the newly inserted node. What could go wrong? If thread A holds the lock it could be about to change `L->head`. If it has not yet changed `L->head` then the lookup function will see the old `L->head`, and that node’s next pointer won’t change, so it can traverse the list fine. If thread A has changed `L->head` then it has run line 24, so `L->head->next` will be the old head, and the connections are set up properly. A similar argument holds if we are changing something in the middle of the list.

Because line 33 is run before line 34, and line 23 before 24, the list is always properly connected.
Part (d)  [2 marks]

Suppose we now add a delete function. Is locking lookup required, when threads can be calling insert, delete, and lookup concurrently? Explain.

No. If the lookup function is currently at the node being deleted when it is interrupted, then it might discover that it cannot follow the pointer to the next node in the list, or the memory might be freed and reused before the lookup function can continue its traversal. It might also be about to follow the pointer to the deleted node when the node is deleted.
Question 3.  [6 marks]

Consider the following multi-level queue algorithm.

- Processes in queue 0 are scheduled using round robin and a time quantum of 2 time units.
- Processes in queue 1 are scheduling using round robin and a time quantum of 4 time units.
- Processes that use their full time quantum move to (or stay in) queue 1.
- Processes that do not use their full time quantum move to (or stay in) queue 0.
- The scheduler only chooses processes from queue 1 when queue 0 is empty.
- Every 20 time steps, the kernel swaps the two queues. Queue 1 becomes queue 0 and queue 0 becomes queue 1.

Part (a)  [2 marks] When processes arrive, should they begin in queue 0 or queue 1? Explain your answer with an example.

They should begin in queue 0. If a process with a short CPU burst begins in queue 1 it might have a long time to wait before being scheduled.

Part (b)  [2 marks] Does this solve the starvation problem? Explain your answer with an example.

Yes. Every 20 time steps processes will get at least a little CPU time even if they immediately use up their time quantum and go back to queue 1.

Part (c)  [2 marks] Does this algorithm give priority to interactive processes? Explain.

Yes. If processes don’t use up their quantum they go to queue 0 which is given priority.