# Exercise 2 Solution

1. A mutex over the entire table is undesirable since it would unnecessarily restrict concurrency. Such a design would only permit a single insert, lookup or delete operation to be outstanding at any given time, even if they are to different hash bins. A mutex over each element in the doubly linked list would permit the greatest concurrency, but a correct, deadlock-free implementation has to ensure that all elements involved in a delete or insert operation, namely, up to three elements for a delete, or two elements and the hash bin for inserts/some deletes, are acquired in a well-defined order. A mutex over each hash bin is a compromise between these two solutions – it permits more concurrency than solution 1, and is easier to implement correctly than solution 2.
2. The code has three problems: it can deadlock, it fails to restore q1, and it has unmatched wait() and signal().

```
    void atomic_swap(Queue *q1, Queue *q2) {
            Item *item1;
            Item *item2; // items being transferred
            if(q1->id > q2->id) {
                    // impose ordering on P operations
                    Tmp = q1;
                    q1 = q2;
                    q2 = tmp;
            }
            wait(q1->lock);
            wait(q2->lock);
            item1 = dequeue(q1);
            if(item1 !=    NULL) {
                    item2 = dequeue(q2);
                    if(item2 != NULL) {
                            enqueue(q2, item1);
                            enqueue (q1, item2);
                     } else {
                            enqueue (q1, item1);
            }
            signal(q2->lock);
            signal(q1->lock);
        }
```

3.

Smokers can enter the lounge at any time. If smokers in the lounge want to smoke, they must make sure that non-smokers are absent. If there are some non-smokers who are waiting to enter the lounge, the smokers who are not smoking cannot smoke any more. Non-smokers cannot enter the lounge if some smokers are smoking in the lounge.

a. Variable:

int smokingCount=nonSmoCount=0;//the number of smokers and nonsmokers in the lounge

b. Semaphore:

Semaphore smoking=mutexSmoker=mutexNonSmoker=enter=1;

c.

***nonSmoker:***

```
enterlounge(false)
{
   wait(enter);
   wait(mutexNonSmoker);
   if(nonSmoCount==0)
      wait(smoking);
   nonSmoCount++;
   signal(mutexNonSmoker);
   signal(enter);
}
//chat()
leaveLounge(false)
{
   wait(mutexNonSmoker);
   nonSmoCount--;
   if(nonSmoCount==0)
      signal(smoking);
   signal(mutexNonSmoker);
}
```

***Smoker:***

```
enterlounge(true);
{
   wait(enter);
   signal(enter);
}
//chat()
smoke()
{
   wait(mutexSmoker)
   if(smokingCount==0)
     wait(smoking);
   smokingCount++;
   signal(mutexSmoker);
   //smoke
   wait(mutexSmoker);
   smokingCount--;
   If(smokingCount==0)
   signal(smoking);
   signal(mutexSmoker);
}
leaveLounge(true) {};
```

d.

a) Mutual Exclusion

b) Progress

c) Bounded waiting.

We can show that the above solution satisfies the three desirable properties.