Modern Operating System Exercise 2

UNIkeEN

March 9, 2023

Problem 1

I will choose **(b).Use a mutex over each hash bin**.

On the one hand, a mutex over the entire table can reduce the efficiency of this data structure. At any single time, at most one operation (insertion, lookup and deletion) can occur. On the other hand, as for a mutex over each hash bin and each element in the doubly linked list, although it provides good concurrency, it requires more space to store more lock or semaphore variables. Moreover, this algorithm design is more difficult to implement to correctly prevent deadlock, since one insert or delete operation can involve up to three elements in the same hash bin.

A mutex over each hash bin not only provide better concurrency, but also reduce space complexity and programming complexity. So I will choose **option (b)**.

Problem 2

The given implementation is incorrect. Reasons are as follows.

- It may cause deadlock. (For example, P_1 swap from Q_1 to Q_2, P_2 swap from Q_2 to Q_3 and P_3 swap from Q_3 to Q_1, it can cause cycle waiting and then cause deadlock.)
- Not all 'wait's have corresponding 'signal's. (if q1 is empty, it's lock will not be released.)
- It fails to restore q1 when q2 is empty.
- Inconsistent function names may cause compilation errors. (I will change all the 'pop' and 'push' in the original code to 'dequeue' and 'enqueue'.)

To prevent cycle waiting, I will add additional index field to queue. Here are my implementation.

```
extern Item *dequeue(Queue *);
extern void enqueue(Queue *, Item *);
void atomic_swap(Queue *q1, Queue *q2) {
    Item *item1;
    Item *item2; // items being transferred
    Queue *tmp;

    if (q1->index > q2->index){ // to avoid cycle waiting
        tmp = q1; q1 = q2; q2 = tmp;
    }
```

```
    wait(q1->lock);
    item1=dequeue(q1);
    if (item1 != NULL){
        wait(q2->lock);
        item2 = dequeue(q2);
        if (item2 != NULL){
            enqueue(q1, item2);
            enqueue(q2, item1);
        }
        else{
            enqueue(q1, item1); //restore q1 if q2 is empty
        }
        signal(q2->lock);
    }
    signal(q1->lock);
}
```

Problem 3

(a) variables:

```
int num_smoker=num_nonsmoker = 0;
```

(b) semaphores:

```
Semaphore mutex_enter = 1; // enter 1 member every single time
Semaphore mutex_smoker = mutex_nonsmoker = 1;
Semaphore smoke = 1; // smoke == 1 means that it can change smoking status
```

(c). The pseudo code is as follows.
For smokers:

```
enter_Lounge(true){
    wait(mutex_enter);
    //enter
    signal(mutex_enter);
}

smoke(){
    wait(mutex_smoker);
    if (num_smoker == 0)
        wait(smoke);
    num_smoker++;
    signal(mutex_smoker);
    //smoke
    wait(mutex_smoker);
```

```
    num_smoker--;
    if (num_smoker == 0)
        signal(smoke);
    signal(mutex_smoker);
}

leave_Lounge(true){
    //leave
}
```

For non-smokers:

```
enter_Lounge(false){
    wait(mutex_enter);
    wait(mutex_nonsmoker);
    if (num_nonsmoker == 0)
        wait(smoke);
    num_nonsmoker++;
    //enter
    signal(mutex_nonsmoker);
    signal(mutex_enter);
}

leave_Lounge(false){
    wait(mutex_nonsmoker);
    num_nonsmoker--;
    //leave
    if (num_nonsmoker == 0)
        signal(smoke);
    signal(mutex_nonsmoker);
}
```

If there are some nonsmokers in the lounge, smokers can enter but can't smoke until all nonsmokers are leave.

Problem 4

(a) variables:

```
int num_s_i = 0; //number of inserter and searcher
```

(b) semaphores:

```
Semaphore mutex=1;
Semaphore nonexclusive=exclusive=1;
Semaphore inserter=1; //Make sure that there is only one working inserter.
```

(c). The pseudo code is as follows. For searchers:

```
wait(nonexclusive);
wait(mutex);
prev = num_s_i;
num_s_i++;
signal(mutex);
if (prev==0)
    wait(exclusive);
signal(nonexclusive);
//search
wait(mutex);
num_s_i--;
if (num_s_i==0);
    signal(exclusive);
signal(mutex);
```

For inserters:

```
wait(inserter);
wait(nonexclusive);
wait(mutex);
prev = num_s_i;
num_s_i++;
signal(mutex);
if (prev==0)
    wait(exclusive);
signal(nonexclusive);
//insert
wait(mutex);
num_s_i--;
if (num_s_i==0);
    signal(exclusive);
signal(mutex);
signal(inserter);
```

For deleters:

```
wait(nonexclusive);
wait(exclusive);
//selete
signal(exclusive);
signal(nonexclusive);
```

It's similar with the no-starvation solution of reader-writer problem. The searcher is the same as reader, the inserter is a special reader that can only exist one at the same time (so we add an independent semaphore outside the normal reader model), and the deletor is the same as writer.