CS2302 Operating System Exercises Spring 2022

Exercise 2: Process Synchronization

Due date: Mar. 28, 2022

1. You are designing a data structure for efficient dictionary lookup in a multithreaded application. The design uses a hash table that consists of an array of pointers each corresponding to a hash bin. The array has 1001 elements, and a hash function takes an item to be searched and computes an entry between 0 and 1000. The pointer at the computed entry is either null, in which case the item is not found, or it points to a doubly linked list of items that you would search sequentially to see if any of them matches the item you are searching for. There are three functions defined on the hash table: Insertion (if an item is not there already), Lookup (to see if an item is there), and deletion (to remove an item from the table). Considering the need for synchronization, would you:
   a. Use a mutex over the entire table?
   b. Use a mutex over each hash bin?
   c. Use a mutex over each hash bin and a mutex over each element in the doubly linked list?

   Please justify your answer.
2. You have been hired by LargeConcurrentSystemsRUs, Inc. to review their code. Below is their atomic_swap procedure. It is intended to work as follows:
   a. Atomic_swap should take two queues as arguments, dequeue an item from each, and enqueue each item onto the opposite queue. If either queue is empty, the swap should fail and the queues should be left as they were before the swap was attempted. The swap must appear to occur atomically – an external thread should not be able to observe that an item has been removed from one queue but not pushed onto the other one. In addition, the implementation must be concurrent – it must allow multiple swaps between unrelated queues to happen in parallel. Finally, the system should never deadlock.
   b. Please discuss whether the following implementation is correct. If not, explain why (there may be more than one reason) and rewrite the code, such that it can work correctly. Assume that you have access to enqueue and dequeue operations on queues with the signatures given in the code. You may assume that q1 and q2 never refer to the same queue. You may add additional fields to stack if you document what they are.

```
extern Item *dequeue(Queue *);     // pops an item from a stack
extern void enqueue(Queue *, Item *); // pushes an item onto a stack
void atomic_swap(Queue *q1, Queue *q2) {
   Item *item1;
   Item *item2; // items being transferred
   wait(q1>lock);
   item1 = pop(q1);
   if(item1 != NULL) {
        wait(q2>lock);
        item2 = pop(q2);
        if(item2 != NULL) {
          push(q2, item1);
         push(q1, item2);
         signal(q2>lock);
         signal(q1>lock);
        }
}
    }
```

3. A club has a lounge where the members can sit and chat. Members include both smokers and non-smokers. Smokers can smoke in the lounge when non-smokers are absent. Device a protocol for the lounge. A smoker calls enter Lounge (true) to enter the lounge (the flag true indicates that she is a smoker), then calls smoke(), and finally calls leave Lounge(true) to leave the lounge. Similarly, a non-smoker calls enter Lounge (false) to enter, then sits in the lounge and chats with

others, and finally calls leave Lounge (false) to leave the lounge.

Questions:

a. Declare all the variables/arrays you need for this problem and initialize them.

b. Declare all the semaphores needed for synchronization and mutual exclusion you need for this problem and initialize them.

c. Write pseudo code for the functions noted above so that smoking rules are obeyed.

d. Name and describe three desirable properties that any synchronization algorithm should possess. Explain briefly whether your solution satisfies each property.

① smoker 与 nonsmoker 互斥

② 为了 smoker 和为了 nonsmoker

③ 保护计数

```
Int smoker = nonsmoker = 0
Semaphore smokermutex = nonsmokermutex = mutex = 1

Lounge(true)
{
    wait(smokermutex)
    If(smoker==0)
        wait(mutex)
    smoker++
    signal(smokermutex)
}
    smoke( )
leave(true)
{   wait(smokermutex)
      smoker--
      If(smoker==0)
          signal(mutex)
    signal(smokermutex)
}
```

```
Lounge(false)
{
    the same as smoker.
}
```

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) {};
```

谁可以抽烟？

d.

a) Mutual Exclusion
b) Progress
c) Bounded waiting.

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