CS307 Operating System Exercises Spring 2020

Exercise 2: Process Synchronization

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.

Answer:

I would use the scheme c .

Reason:

When the Deletion and Insertion is taking, we can't take the Lookup operation. For the fact that the hash bin will be changed in this two operation, so the result of Lookup is inconsistent if we

take the Lookup operation at the same time. Thus, we need a Mutex over each hash bin to separate Lookup operation. Deletion and Insertion can be taken on the same hash bin, because Insertion and deletion just manipulate one item in the hash bin. The hash bin has a data structure of doubly linked list so the operation one the specific element will not affect the others in the same hash bin. However, the Deletion and Insertion can't be taken on the same element. For example, if the newly added item's next element is the one being deleted, then the doubly linked list will break after the deletion. So we need a Mutex over each element in each doubly linked list to separate the Deletion and Insertion.

2. You have been hired by Large−Concurrent−Systems−R−Us, 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);
		}
	}
}
```

Reason:

1. The operation: signal(q1->lock) is placed a wrong position. In that case, if : item1 == NULL, then the signal(q1->lock) will never be called, and the lock will exist forever, which causes deadlock.
2. The operation: signal(q2->lock) has the same mistake with signal(q1->lock). If the item1==NULL and it will cause deadlock.
3. q2->lock is inside the q1->lock, and this may cause deadlock. If two queue: queue A and queue B is used in two different threads' atomic_swap call. In one call A is q1 and B is q2, while in the other call A is q2 and B is q1, and this will cause deadlock. Two threads are waiting for each other to exit.

4. If either queue is empty, the queues can't be left as they were before the swap was attempted.

Because it doesn't call push() operation when the swap failed.

Correct Answer:

```
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);
        signal(q1->lock);
        wait(q2->lock);
        item2 = pop(q2);
        signal(q2->lock);
        if(item1 != NULL) {
                if(item2 != NULL) {
                        wait(q1->lock);
                        push(q1, item2);
                `       signal(q1->lock);
                        wait(q2->lock);
                        push(q2, item1);
                `       signal(q2->lock);
                }
        }
        else{
                wait(q1->lock);
                push(q1, item1);
                signal(q1->lock);
                wait(q2->lock);
                push(q2, item2);
                signal(q2->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.

Answer:

a.

```
static int SmokerCount = 0;
static int NSmokerCount = 0;
```

b.

```
semaphore Mutex;
Mutex->value = 1;
semaphore SmokerMutex, NSmokerMutex;
SmokerMutex->value = 1;
NSmokerMutex->value = 1;
semaphore Smoker, NSmoker;
Smoker->value = 1;
NSmoker->value = 1;
```

c.

```
void enter_Lounge(bool flag) {
    if (flag) {
        wait(Mutex);
        wait(SmokerMutex);
        SmokerCount++;
        if (SmokerCount == 1) {
            wait(NSmoker);
        }
        signal(SmokerMutex);
        signal(Mutex);

        // do smoking and chats in the lounge
        smoke();
```

```
            wait(SmokerMutex);
            leave_Lounge(flag);
            SmokerCount--;
            if (SmokerCount == 0) {
                signal(Smoker);
            }
            signal(SmokerMutex);

        }
        else {
            wait(Mutex);
            wait(NSmokerMutex);
            NSmokerCount++;
            if (NSmokerCount == 1) {
                wait(Smoker);
            }
            signal(NSmokerMutex);
            signal(Mutex);

            // do chats in the lounge

            wait(NSmokerMutex);
            leave_Lounge(flag);
            NSmokerCount--;
            if (NSmokerCount == 0) {
                signal(NSmoker);
            }
            signal(NSmokerMutex);

        }
}
```

d.

Three desirable properties: 1.Mutual exclusion 2.Progress 3. Bounded Waiting

Mutual exclusion: `wait(NSmoker)` in the `enter_Lounge(true)` and `wait(Smoker)` in the `enter_Lounge(false)` ensure mutual exclusion between smokers and non-smokers. Binary semaphore `SmokerMutex` and `NSmokerMutex` ensure mutual exclusion between club members who want to leave the lounge.

Progress: When the `NSmokerCount == 0` or the `SmokerCount == 0`, the program will call: `signal(NSmoker)` and `signal(Smoker)`. This ensures the waiting threads (waiting smokers or

waiting non-smokers) can release the lock at the beginning (`if (SmokerCount == 1) in enter_Lounge(true)` && `if (NSmokerCount == 1) in enter_Lounge(false)`), then enter their critical sections. So it satisfies the Progress property.

Bounded waiting: This waiting scheme will not cause starvation because I add a binary semaphore (Mutex) as mutex lock at the beginning. This semaphore plays a significant role which ensures the new coming threads are queuing in FCFS order. So it will not cause starvation, which means all the threads only need to wait for the critical section to be used for an upper limit number of times before the threads can enter the critical section.