Name: Abdur Rahman

Question 1:

Solution:

The lemma "Every (non-crashed) Pop operation O returns the first key that was in PHYS at the time O was linearized" **does not hold anymore in this case.**

Further on, in the cases when the pop operation O returns EMPTY, and O is linearized at return statement, the PHYS at time of operation O, can have the top pointer pointing to a node as there is nothing stopping PHYS to be modified by another thread between the time top pointer is read to the return statement.

Thus Pop operation O can return EMPTY, while the PHYS contains a NON-EMPTY top pointer. **Thus, in this case, although PHYS might be equal to ABS, but return value will not be correct.**

Name: Abdur Rahman

Question 2.2:

Solution:

The source of slowdown was the faulty padding in the padded_subcounter struct. The padding was introduced after the atomic count variable. The way structs are mapped in memory is by the order in which elements in struct are defined. Thus the first atomic count variable "v" had no padding between itself and the atomic done variable which was read by all the threads. We can safely assume that in faulty executions v and done variable could have been in same cache line, making the cache line to be shared among all the threads. **This caused false sharing between first count variable "v" and the "done" variable**

A tiny performance improvement was achieved by moving the done variable after the counters array. This performance gain might be attributed to memory management/page allocation of the C++ program.

Name: Abdur Rahman

Question 3.1:

Solution:

For this part of question, I introduced the variable turn and wants_to_enter as non-volatile and non-atomic variables.

Code:

```
class mutex_t {
private:
    bool wants_to_enter[2];
    int turn;
public:
    mutex_t(): turn(0) {
        wants_to_enter[0] = false;
        wants_to_enter[1] = false;
    }
    void lock() {
        printf("Executing lock: %d\n", tid);
        wants_to_enter[tid] = true;
        while(wants_to_enter[1 - tid]) {
            if(turn != tid){
                wants_to_enter[tid] = false;
                while(turn != tid){}
                wants_to_enter[tid] = true;
            }
        }
        printf("Executed lock: %d\n", tid);
    }
    void unlock() {
        printf("Executing Unlock: %d\n", tid);
        turn = 1 - tid;
        wants_to_enter[tid] = false;
        printf("Executed Unlock: %d\n", tid);
    }
};
```

Output:

```
Executing lock: 0
Executed lock: 0
Executing Unlock: 0
Executed Unlock: 0
Executing lock: 0
Executed lock: 0
Executing lock: 1
Executing Unlock: 0
Executed Unlock: 0
Executing lock: 0
```

After some debugging, I reached to a conclusion that without atomics and volatile, the algorithm is reaching deadlock in certain iterations, as shown above.

In the iterations it reaches deadlock, is when one of the threads has acquired the lock, and before the thread could release the lock another thread tries to acquire the lock. The deadlock is due to shared variables between the threads, both the turn variable and the want to enter variable.

Compiler could be reading the want_to_enter variable from local register, since it has no indication that this variable is meant to be shared with another thread, which can change it to another value. It can also be caused by the turn variable, as the other thread can cause the turn variable to change, and the current thread might be reading it from the saved register.