Name: Shivank Goel
Entry Number: 2014CS10565

Problem 1:

1)

a) B(r.Wr(1)) => A(r.Rd(1)) => C(r.Wr(2)) => B(r.Rd(2)) gives us a sequentially consistent execution of methods , hence **it is sequentially consistent**. Also this series of execution is linearized and hence **it is linearizable** by choosing linearization point suitably for first 3 method calls which all are overlapping in real time.

b) C(r.Wr(2)) => B(r.Wr(1)) => A(r.Rd(1)) => B(r.Rd(1)) gives us a sequentially consistent and linearizable order of method calls. Hence it is also both **sequentially consistent and linearizable**.

2) (Sequential Consistency , Linearizability)
(Linearizability , Strict Consistency)

i.e
(Weakest) Sequential < Linearizable < Strict (Strongest)

Sequential consistency ensures arbitrary ordering of different threads as long as they are following program order. Linearizability assumes instantaneous execution of each method at some moment in real time. Thus program order is also conserved automatically and it enforces a stronger condition on interleaving of two threads i.e by incorporating real time and avoiding arbitrary interleaving of different processes. Only overlapping methods can be arbitrarily scheduled according to linearization points.

Strict Consistency incorporates both linearizability and strict consistency. For eg. in sequential consistency thread A may read from x and then thread B write to x during overlapping time durations it's valid according to strict consistency, also valid according to linearizability since it will assume write happened before read by choosing suitable linearization point (but actually it may not had happened that write occurred before read) but strict consistency enforces this last step that write occurs before read.

3)
Lamport Timestamps

- If E1 occurs before E2 then less timestamp(E1) < timestamp(E2) **BUT** if timestamp(E1) < timestamp(E2) then either E1 occurs before E2 or E1,E2 were concurrent. So we can't distinguish in this case. This problem is solved in Vector Clocks.
- Each Process use only 1 local integer clock

Vector Clocks

- We can identify concurrent events.
- Each Process use a vector of integer clocks.

Figure 1a) Vector Clocks

- **ThreadA** (0,0,0) Receive Message From B (1,2,0) ; Read value (2,2,0)
- **ThreadB** (0,0,0) ; Writes Value 1 (0,1,0) ; Send Message to A (0,2,0) ; Receive Message From C (0,3,2) ; Read value (0,4,2)
- **ThreadC** (0,0,0) ; Writes value 2 (0,0,1) ; Send Message to B (0,0,2)

Problem 2:

```
1 class Filter implements Lock {
2       int[] level;
3       int[] victim;
4       public Filter(int n) {
5               level = new int[n];
6               victim = new int[n]; // use 1..n-1
7               for (int i = 0; i < n; i++) {
8                       level[i] = 0;
9               }
10      }
11      public void lock() {
12              int me = ThreadID.get();
13              for (int i = 1; i < n; i++) { //attempt level 1
14                      level[me] = i;
15                      victim[i] = me;
16                      // spin while conflicts exist
17                      while ((∃k != me) (level[k] >= i && victim[i] == me)) {};
18              }
19      }
20      public void unlock() {
21              int me = ThreadID.get();
22              level[me] = 0;
23      }
24 }
```

Yes, Filter Lock can allow some thread to arbitrarily overcome another thread. There exists one victim at each level and that is that is the last thread to reach line#15 i.e. victim[i] = me. And thus there is always 1 less number of threads reaching next level unless all threads in above levels get emptied.

Let A and B both want to go to critical section and starts from level 0 , it so happens that A gets stuck in one of the while loop of some level L and B acquires the CS. Now B goes to level 0 and wait till it reaches next level and in meantime it is possible that A also acquires CS and again starts from level 0. Since in level 0 B called lock() earlier than A it is quite possible (though not necessary) that A reaches line#15 later than B and thus again B is no longer victim since now A is and B can move to next level and in this way can keep overcoming A in each level arbitrary number of times, till there comes a stage where B becomes victim later than A.

Problem 3:

```
int turn;
bool busy = false;
1       void lock() {
2               int me = tid.get();
3               do {
4                       do {
5                               turn = me;
6                       } while(busy);
7                       busy = true;
8               } while (turn != me);
9       }

1       void unlock(){
2               busy = false;
3       }
```

a)

Yes the algorithm satisfies mutual exclusion i.e both threads can't go to CS at same time.
Proof:
Let there are two threads. For contradiction
CS(A,i) ->CS(B,j) and CS(B,j) -> CS(A,i) for some iteration i and j of the two threads over critical section.
Now,

From the code we know that
i) writeA(turn=A) -> readA(busy = false) -> writeA(busy = true) -> readA(turn = A) -> CSA
ii) writeB(turn=B) -> readB(busy = false) -> writeB(busy = true) -> readB(turn = B) -> CSB

Let first A reached CSA and then B also reached CSB when A was already in CSA . Since A has acquired the lock after reading turn = A at some point and then B also acquired lock by writing that turn=B again
iii) readA(turn = A) -> writeB(turn = B)

From i , ii and iii
writeA(busy = true) -> readA(turn = A) -> writeB(turn=B) -> readB(busy = false)
We arrive at a contradiction that B reads busy to be false where we nowhere made it false since both A and B are still in their critical section. Hence it follows mutual exclusion.

b)

No it is **not deadlock free**. Consider two threads A and B. Consider following execution steps:
-> A sets turn = A
-> A sets busy = true
-> B sets turn = B
Now since turn is not equal to A , A will not go in critical section and busy will remain true forever.
B is already stuck in the inner while loop i.e while(busy)
-> A sets turn = A
After this update A also gets stucked in inner while loop i.e. while(busy)

Problem 4:

```
class MyQ<T> {
        Atomic Int head , t a i l ;
        T items [ MAX INT SIZE ] ;
        void enq (T x ){
                int s l o t ;
                do {
                        s l o t = t a i l . get ( ) ;
                } while ( ! t a i l . CompareAndSet ( s l o t , s l o t +1));
                items [ s l o t ] = x ;
        }

        T deq (T x ){
                T value ; int s l o t ;
                do{
                        s l o t = head . get ( ) ;
                        value = items [ s l o t ] ;
                        i f ( value == NULL)
                                throw EmptyException ( ) ;

                } while ( ! head . CompareAndSet ( s l o t , s l o t +1));
        return value ;
        }
} ;
```

The FIFO queue implementation is not linearizable because in linearization we assume instantaneous execution of each method at some moment in real time. Thus all methods must be all atomic i.e taken place simultaneously i.e it should not be possible that say thread A remain stuck at some line of method and B completed the method even though B called that method after A.

In the above case we can consider { tail.CompareAndSet(slot,slot+1) , items[slot] = x } as non-atomic step in enqueue method.

Consider threads A,B,C running in parallel
Let initially slot was 0 .
Let thread A set tail.slot = 1 and let A has not yet written items[1]
Let thread B set tail.slot = 2 and write items[2] = somevalue
Let thread C tries to dequeue now, since head is at index/slot 1 which is not yet written by A it will return exception , but ideally it should not be. It happened because we considered enqueue operation to be atomic which actually wasn't.

Another scenario could be

Thread A
        Slot = 0
        //Tail.slot = compare_update(0,1) (It was to be run but not yet tun , Not atomic step)
Thread B
        Slot = 0
        Tail.slot = compare_update(0,1)
        Items[1] = some value
Thread A

Tail.slot = compare_update(1,1)
Items[1] = some other value

Hence values can also be overwritten. Thus due to non-atomicity of combination of various atomic steps algorithm is not linearizable.

Problem 5:

```
1 class Bakery implements Lock {
2          boolean[] flag;
3          Label[] label;
4          public Bakery (int n) {
5                    flag = new boolean[n];
6                    label = new Label[n];
7                    for (int i = 0; i < n; i++) {
8                              flag[i] = false; label[i] = 0;
9 }
10                   }
11                    public void lock() {
12                             int i = ThreadID.get();
13                             flag[i] = true;
14                             label[i] = max(label[0], ...,label[n-1]) + 1;
15                             while ((∃ k != i)(flag[k] && (label[k],k) << (label[i],i))) {};
16                   }
17                    public void unlock() {
18                             flag[ThreadID.get()] = false;
19                   }
20         }
```

Deadlock Free

Lock can only occur when two or more threads get stuck in while{} loop due to variables that none of them are being able to update to get out of that loop.

Here any thread only waits till some other flag of lower entry number has it's flag true. So for contradiction let's say their is deadlock. Now consider out of all the threads in the deadlock the thread with lowest pair value (label[j],j) for some value of j. Now, since there doesn't exist any other thread lower pair value than this which means while loop will become false and jth thread can access Critical Section. Hence it contradicts that j was in deadlock.

Mutual Exclusion

Suppose for contradiction let A and B be two threads which are both in Critical Section. Let label i and j be respectively last labels for threads A and B before entering their critical section.

Without loss of generality let's assume (label[i],i) < (label[j],j) . Which means A must had entered critical section before B. Now before B entered critical section it must had found flag[i] = false else, it couldn't enter critical section. (Because (label[i],i) > (label[j],j) is not possible by our assumption) .

So flag of A must had been set after B read it and then labelling of A was given after setting its flag i.e.

labelB -> readB(flag[A]) -> writeA(flag[A]) -> labelA

But it contradicts our assumption (label[i],i) < (label[j],j) , and hence Bakery Lock algorithm satisfies mutual exclusion.