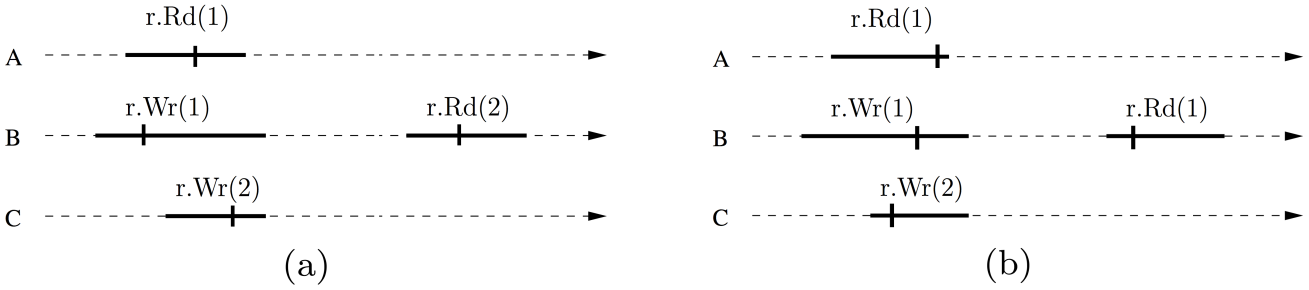
Homework 1

COL380

Problem 1 : Sequential Consistency, Linearizability

1.1

For Fig 1(a) as well as Fig 1(b) we can see that we can choose linearizability points as I have shown, so that the points shown preserve the real time behaviour of method calls and hence the linearizability property is satisfied.
If a history is linearisable its also sequentially consistent, hence both (a) and (b) are linearisable and sequentially consistent.



1.2

- (Sequential Consistency, Linearizability) ∈ W
- (Linearizability, Strict Consistency) ∈ W
- (Sequential Consistency, Strict Consistency) ∈ W

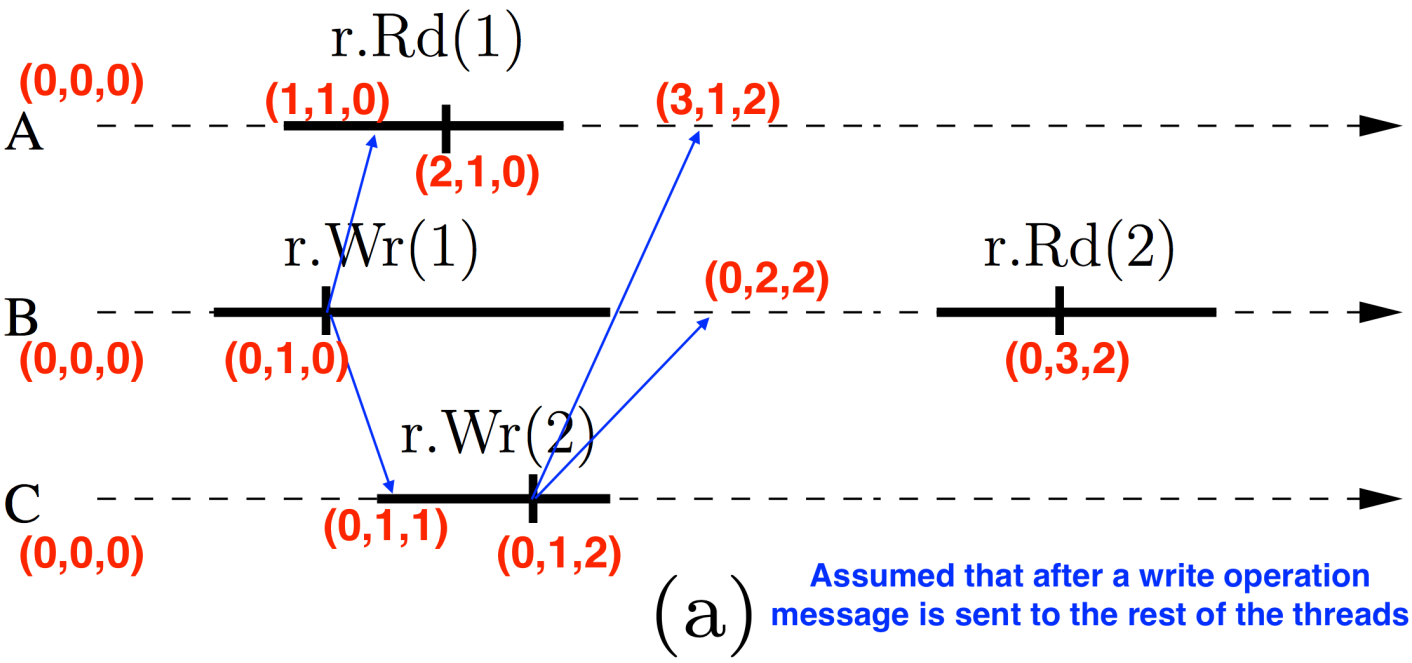
We know that linearizability implies sequential consistency, hence latter is weaker than the first one. In SC we are only concerned about method calls to take effect in program order, whereas real time behaviour is also taken care of in linearizability.

In strict consistency we are given the fixed (default) set of linearisation points in bw the start and end time of each method at which the effect of that method take place, but in linearizability, we can choose the point of effect of methods in bw the start and end times of the method. So we only need one combination/set of points to satisfy the property of linearizability. Hence Linearizability is weaker than strict consistency.

From the first parts we can easily derive the 3rd part by transitivity property of the relation.

1.3

- In Lamport Clocks each process uses a local counter (clock) which is an integer, whereas in Vector Clock each process maintains a vector V_i where nth element of V_i is i's knowledge of latest events of process j.
- Let → represent Happens Before relation.
- In Lamport E1 → E2 ⇒ timestamp(E1) < timestamp(E2), **BUT** timestamp(E1) < timestamp(E2) ⇒ {E1 → E2} or {E1 & E2 are concurrent}. Hence just by the Lamport clock we can't differentiate bw pair of events being concurrent or causal, but we can do so by the Vector Clocks, but these use extra space and time of updating the vector clocks.



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++) {
14             level[me] = i;
15             victim[i] = me;
16             while ((exists k != me) such that (level[k
                   ] >= i && victim[i] == me)) {};
17         }
18     }
19     public void unlock() {
20         int me = ThreadID.get();
21         level[me] = 0;
22     }
23 }
```

Let us consider two threads A and B out of the total n threads. Let A be in its while loop and let both of them be at equal levels. Now consider the situation where B updates victim[i] = me in its section. This renders A to break out from the while loop and execute the next iteration of the parent for loop, which results in incrementation of A's level. Hence A has a higher level than B, in other words A overtakes B in this case. Now such situation can take place between any 2 out of all n threads. Hence it allows threads to overtake others an arbitrary number of times.

Problem 3

3.1 Mutual Exclusion

The given protocol satisfies Mutual Exclusion.
Proof : I will prove using contradiction.
Let us assume the protocol doesn't satisfy mutual exclusion.
$\Rightarrow \exists$ Threads A, B such that CS_a and CS_b are concurrent.

We can infer from the given code that :
1. $Wr_a(turn = A) \rightarrow Rd_a(busy == false) \rightarrow Wr_a(busy = true) \rightarrow Rd_a(turn == A)$
2. $Wr_b(turn = B) \rightarrow Rd_b(busy == false) \rightarrow Wr_b(busy = true) \rightarrow Rd_b(turn == B)$
3. $Rd_a(turn == A) \rightarrow Wr_b(turn = B)$

If A enters CS_a we know that $Rd_a(turn == A)$ will be executed. From 3 we can conclude that

- $Wr_a(busy = true) \rightarrow Rd_a(turn == A) \rightarrow Wr_b(turn = B) \rightarrow Rd_b(busy == false)$

We observe that *busy* is true till the time A executes unlock(). Which implies that B will not be able to read(busy==false), hence B will not enter its critical section before A completes execution of unlock(). Which is a contradiction. Hence proved.

3.2 Deadlock

Consider the situation when thread A executes (busy = true) in lock() post which B executes (turn = me). Now A will not be able to escape from lock() and variable busy will remain true forever which will result in both thread being stuck forever. Which is a deadlock. Hence this protocol is not deadlock free.

Problem 4

Let there be 2 threads A and B. Lets take a situation when a thread A wants to enqueue P in an empty queue. So it will call function enq(), and while executing it it would increment the tail by 1 using the call to CompareAndSet in the while loop. Now let there be a context switch to thread B just after thread A broke from its do-while loop. So right now the tail has been incremented but the data has not been saved at the 1st position (head) of queue. Now B does a successful enqueue at 2nd position. Now B tries to dequeue, in the deq() function, since value at head is NULL, an EmptyException() will be thrown. But the queue is not empty, as enqueue by Thread B precedes the dequeue by B. This is an example of non linearisable history. Hence the queue object is not linearisable.

Problem 5

5.1 Mutual Exclusion

Proof : I will prove using contradiction.
Let us assume the protocol doesn't satisfy mutual exclusion.

$\Rightarrow \exists$ Threads A, B such that CS_a and CS_b are concurrent.

Just before A and B enter their CS, let $label_A$ and $label_B$ be the respective sequences of acquiring new labels. Wlg assume (label[A], A) << (label[B], B). Now when B completed the test successfully in its waiting section, it must have read flag[A] = false OR else (label[B], B) << (label[A], A), which is not possible by our assumption. Now for a thread, its id is fixed and label[] entries are strictly increasing, hence B must have seen flag[A] as false, but the causality relations $label_B \rightarrow Rd_B(flag[A]) \rightarrow Wr_A(flag[A]) \rightarrow label_A$ are a clear contradiction of the assumption (label[A], A) << (label[B], B). Hence we conclude that Bakery Lock algo is mutually exclusive.

5.2 Deadlock

Since the labels are in increasing order there will be some thread A with the least pair (label[A], A). This thread will never wait for another thread and there will not be any thread X, such that (label[X], X) << (label[A], A). So this thread will break out of the while loop and hence enter its critical section. Hence it will never be the case that any thread will get stuck, therefore its deadlock free.