CS5300: PARALLEL AND CONCURRENT PROGRAMMING, Fall 2020

QUIZ 2

SAI HARSHA KOTTAPALLI

CS17BTECH11036

1. Let us say we have two threads A and B trying to acquire lock and capacity is set to 2. Lets us assume we have swapped the lines and A maps to 0 and B maps to 1.
   Now flag[0] is true so A can enter CS while B has flag[slot] set to false, waiting on its flag[slot] value to be set to true. Now after A completes and sets "flag[(slot + 1) % size] = true;" and then gets swapped out.
   Now B upon finding that flag[1] is true enters the CS does its work quickly and unlocks() quickly. Essentially setting flag[0] to be true and flag[1] to be false.
   Now thread A upon waking up sets flag[0] to be false unaware that B has already completed its work.
   Now Both flag[0] and flag[1] are set to false so any thread trying to acquire lock will spin indefinitely, causing deadlock.
2. If we change line number 30 to myNode.set(qnode), this might cause deadlock. This is because in CLH locks, previously, the predecessor nodes which aren't in use anymore are used as the node for the next lock acquisition(future lock accesses) as the old node for the thread (qnode) might still be referenced by the tail and the thread's successor.
   An example for deadlock, Lets say we have threads A, B, C, D competing to lock in the order given where A is in CS. Now after completion of A it would set myNode to its qNode, Now assume B, C, D are slow and stuck at line number 25 while, A again enters lock, it would try to attain the lock while keeping D as its predecessor. Now as for B, it would check A's node and see that locked is

true so it would keep spinning. A would also not progress because its waiting on D's locked value which is set to true. Now A, B, C, D all of them are waiting in a cyclic order and a deadlock has occurred.

3. a. Directly checking the code it seems like Line number 20 is the linearization point but actually line number 19 can also be the linearization point. Since both curr and pred are locked nobody else can modify curr and pred anymore, that is add in between or remove them. As this is guaranteed, Execution of Line number 19 and 20 is as if the thread is executing all alone with any modifications from others. Since attaining the locks itself is the linearization point here, (Line7 or 13, depending on the execution) until unlock() any of these can be considered as linearization points fo the case of returning true.
   b. Similar to the above argument, it seems line number 45 is the linearization point as this is where the node is unlinked, but since the two nodes curr and pred are locked by line 42(Line 36 or 42 depending on execution) nobody else can insert any nodes in between or try to delete pred or curr. Since running in isolation is guaranteed because of usage of locks and line number 45 is before unlock() it can be considered as the linearization point.
4. The contains() method implementation for LazyList does not require the usage of locks hence it is wait-free as there aren't any blocking calls made in the implementation. As for correctness, with the help of marking bit, it returns true iff the node it was searching is present and unmarked.

   The linearization points for it would be line number 6 as discussed in class.

5. Now let us assume the current set looks like:

   -INF -> 100 -> 200 -> 300 -> INF

   where -INF and +INF are sentinel nodes, rest being regular nodes.

Now lets use say a thread A wants to insert 150. A searches the list and sets pred to be node with key 100 and curr to be node with 200. Now before it acquires lock on these nodes, it got swapped out and some other thread say B comes and

quickly inserts 175 between 100 and 200. Now A comes back, acquires lock on pred and curr and continues running validate() to find that pred is no longer pointing to curr. Hence, validate() returns false and thread A starts again. This time it searches the list and sets pred to be 100 and curr to be 175, again before it acquires lock it got swapped out and a new thread say C comes and tries to remove 175. Since there are no other contenders it proceeds and finally sets node with key 100 next value to node with key value 200. Now thread A wakes up and attaining the locks and executes validate to find that node with key 175 is no longer reachable from the head/ pred doesn't point to curr it had. So it starts again trying to insert 150.
Similar to the above scenario threads might keep adding 175 and removing it indefinitely not giving any chance for thread A to insert 150 and hence runs forever.