Assignment 5 SAQs

Justin Cohler

SAQ 1

Exercise 101: Explain why the fine-grained locking algorithm is not subject to deadlock.

The main mechanism by which fine-grained locks avoid deadlocks is in the hand-over-hand method of obtaining locks. On page 202 of the text, “To guarantee progress, it is important that all methods acquire locks in the same order, starting at the *head* and following *next* references toward the tail.” If the different calls to the same element of the list acquired the locks in different orders, thread *A* could obtain the *head* component of a pair, while thread *B* could simultaneously obtain the *tail* component of the pair, which would set the two in a deadlock. Order of obtaining *pred* and *curr* ensures deadlock-freedom.

SAQ 2

Exercise 103: Explain why the optimistic and lazy locking algorithms are not subject to deadlock.

Optimistic locking ensures that a lock will obtain candidate *pred* and *curr* locks each attempt. However, in the time between locking the *pred* and *curr* locks, a new node may be inserted between the two. A validation step ensures once two locks are created that one still follows the other. If not, the process restarts. Each thread may remove nodes partway through the process, but because we restart if both locks a.) exist, and b.) follow one another, we are ensured there can be no deadlocking between two threads, since a linked-list ensures two nodes cannot both follow one another (cycle).

Lazy locking operates by keeping a boolean per node indicating it is set. Separate threads may traverse nodes that have been removed from the list by another thread. However, since the value of next of a removed node is still legitimate, the traverser of the outdated node will find the list again. Here we avoid deadlocking by locking down the target nodes and then ensuring that they are still reachable from the *head* node. No matter what, a given node that comes “offline” from the active linked list can always follow pointers back to the linked list (in the worst case to the tail).

SAQ 3

Exercise 104. Show a scenario in the optimistic algorithm where a thread is forever attempting to delete a node.

Hint: since we assume that all the individual node locks are starvation-free, the livelock is not on any individual lock, and a bad execution must repeatedly add and remove nodes from the list.

Let's take an example where we have nodes *A* and *B*, where *A* points to *B.* Thread 1 attempts to delete node B. The optimistic lock would traverse the list until it reached node B, at which point would find the predecessor A. It would then obtain locks on each and validate that A still pointed to B. However, let's assume another thread, Thread 2 inserted a node between A and B. The validation would return false, and the entire handshake would occur again.

If each time, a node is inserted between the *pred* and *curr* nodes before locks are obtained on each of them, the *curr* node will never be deleted, and the delete will constantly retry.

SAQ 4

Exercise 121. Design a bounded lock-based queue implementation using an array instead of a linked list.

1. Allow parallelism by using two separate locks for head and tail .

Below, the BoundedTwoLockQueue is described, with code found in stack/bounded_twolock.go. The main differences are that now we essentially have one condition associated with the headLock, and another condition associated with a tailLock. We can add to the queue until we reach capacity with one lock, and can pop from the front with the other, simply broadcasting each time when additional work needs to be done to shrink the array, or to wait for space to become available to add to the array:

2. Try to transform your algorithm to be lock-free. Where do you run into difficulty?

Attempting to make the algorithm lock-free is challenging, because with a slice in Go, there is no ability to compare and swap pointers to specific areas of the list, as one can with a linked list. With a slice/array, the allocated space of that array is inherent in the design of the data structure, and without a lock, several threads will copy different states of the array into memory to alter the array and write back to memory. This is the classic example where a lock is needed, because each thread will have potentially out-of-date copies of the array and then overwrite one another back to memory, causing inconsistent queue states.

SAQ 5

Exercise 122. Consider the unbounded lock-based queue *deq()* method. Is it necessary to hold the lock when checking that the queue is not empty? Explain.

I would argue that it is necessary to hold the lock when checking that the queue is not empty. Let's consider not locking when checking. Consider a queue with one element remaining. Two threads, A and B, check whether the queue is not empty. Because both can read without the lock, both will see that the queue is not empty before either pops the last value. Next, each will attempt to obtain the lock. Let's say Thread B obtains the lock and pops the last element. Once it releases the lock, Thread A obtains the lock, having already checked for emptiness, and attempts to pop the head of the queue. However, there is no element there, and an exception is raised. In order to ensure state is correct, a lock is needed during the state check of the queue.

SAQ 6

Design an unbounded lock-based stack implementation based on a linked list in Go.

The methods are found in hw5/src/stack/unbounded.go, in the associated repository. The IntStack and IntNode are found in int_stack.go. Additionally, tests are found in the hw5/src/test/test.go file.

SAQ 7

Design a bounded lock-based stack using an array in Go. It must use a single lock and a bounded array.

The methods are found in hw5/src/stack/bounded.go, in the associated repository. The IntStack and IntNode are found in int_stack.go. Additionally, tests are found in the hw5/src/test/test.go file.

SAQ 7.1

Try to make your algorithm lock-free. Where do you run into difficulty?

The methods for a go-based lock-free unbounded queue are found in hw5/src/stack/unbounded_lockfree.go. Additionally, tests are found in the hw5/src/test/test.go file.

However, for an array-based bounded lock-free queue, we have the same issue we ran into in problem #4. While we can attempt to wait until we have a valid size & state for an array, there is no guarantee that my thread's alteration to the array will not overwrite another thread, which has written back to memory in between the time my thread read from memory and when my thread writes back.