CS5510: Multiprocessor Programming : Homework VI

Sergio Sainz

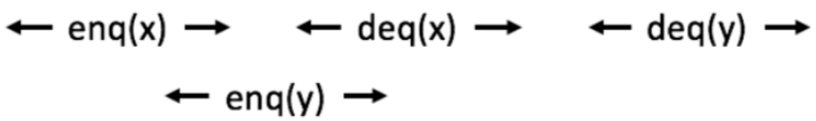
November 2019

1 Part II: Programming (35 points)

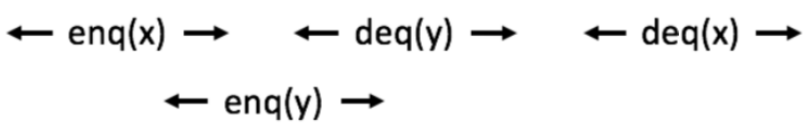
A template will not be provided to you this time. It is your responsibility to provide your solution in a Gradle project. It may be easier to clone the Homework-4/5's template and modify it to your needs.

1.1 Queue [35 points]

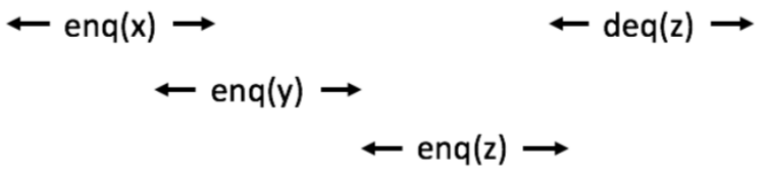
Linearizability is a useful and intuitive consistency correctness condition that is widely used to reason and prove correctness of concurrent implementations of common data structures. Though linearizability is non-blocking, it may impose a synchronization requirement that is stronger than what is needed for some applications, limiting scalability. For example, in highly concurrent servers, a set of threads (called a "thread pool") is often used to execute concurrent entities of the application, which are called "tasks". A thread pool typically uses a queue, where tasks are stored. Threads dequeue tasks from this task queue for execution. Such a queue need not have strict FIFO behavior. A queue that does not allow one task to be bypassed by another task "too much" is often sufficient. For example, it is acceptable to dequeue a task T from the queue even if a task T' that has been enqueued k places "before" T has not yet been dequeued.
In this assignment, you will construct a n-semi-linearizable queue. To un- derstand semi-linearizable queue, consider the following concurrent executions of a linearizable queue:



For a 2 semi-linearizable queue, the following execution is possible:



The following execution is not possible for a 2 semi-linearizable queue. However it is acceptable for 3 semi-linearizable queue.



Implement an n-semi-linearizable queue and measure its throughput (number of operations enqueue/dequeue per seconds) and compare it against other linearizable queues (e.g., java.util.concurrent.ConcurrentLinkedQueue, Unbounded Lock-Free Queue, and Bounded Lock-Free Queue (in Q1.5)). Plot your throughput as a function of the number of threads.
Hint: A possible implementation for an n semi-linearizable queue is by using a queue, where each of its elements is an n-element array. Thus, enqueue will insert multiple nodes in the same queue element, and they will be considered equally by dequeue (so you can randomly select any of them). Another possible implementation is by using a normal queue. For dequeue, you can pick any of the last n nodes and return. Note that the last n elements are treated equally in an n semi-linearizable queue.
Evaluate your implementation on RLogin and plot a throughput vs. thread-count graph containing each of the four queues. Perform enqueues and dequeues with equal probability.
Answer

Decided to design the semi-n-linearizable queue by using half of bounded-lock-free queue from exercise 1.5 and also the bag data structure developed in programming exercise from homework 5.

Idea is to create an array three times the number of threads to be used, then in each array slot, create a LockFreeBag.

During enq method, a writer atomic integer counter is getAndIncremented and then it is divided by n. The result is then modulo array size to find the array's slot. Thread pushes the item to the slot's bag.

During deq method, a reader atomic integer counter is getAndIncremented and then it is divided by n. The result is then modulo array size to find the array's slot. Thread pops the item from the slot's bag and returns to caller.

Because bags are unordered, then, the result is n-linearizable.

We can see the result is that for most cases the semi-n-linearizable queue is the fastest one among them all. This is expected as both the bag data structure and the bounded lock-free queue (array based) are by design reducing contention.

Notice the disadvantage is that the semi-n-linearizable queue is blocking when there is not enough items to remove.