100 points total.

1 Problems (50 points)

1.1 ABA problem [10 points]

Many processors implement special versions of load and store instructions called, `linked-load (LL)` and `store-conditional (SC)` respectively. The LL instruction reads from memory location `p`. A subsequent SC instruction to `p` attempts to store a new value at `p`, but succeeds only if `p` is untouched since that thread issued LL to `p`.

How do you think these instructions can help in solving the ABA problem? Give a clear example. How do you think `LL/SC` are implemented in hardware?

1.2 Stack (1) [10 points]

A standard stack data structure has the following operations: `push()` and `pop()`. Are each of the executions presented in 1 (for two threads), linearizable and/or sequentially consistent? Provide an explanation.

1.3 Stack (2) [10 points]

The implementation of lock-free unbounded stack is in listings 2 and 3. The ABA problem could arise in `pop()` method with improper garbage collection. How can this happen? Give a scenario. Also, explain how the code can be fixed to avoid the ABA problem.

1.4 Queue (1) [10 points]

An implementation of an unbounded queue has been provided in listing 4. The `deq()` is blocking in the sense that it spins until there is an item to dequeue. Assume that the `bucket` array is too large to be filled and `last` is the index of the next unused position in `bucket`.

- Explain whether `enq()` and `deq()` methods are wait-free or lock-free.
- What are the linearization points for `enq()` and `deq()`? Are they execution dependent or independent?

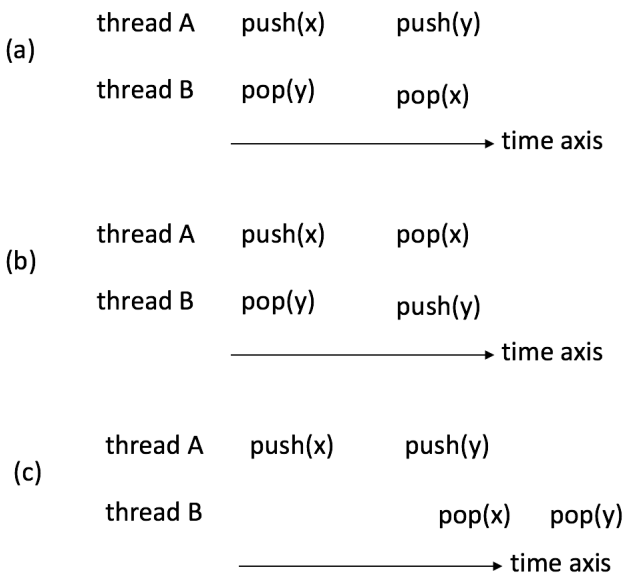


Figure 1: Executions on stack

1.5 Queue (2) [10 points]

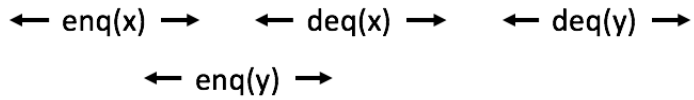
Listing 5 shows the `deq()` method of unbounded lock-based queue. Explain why holding the lock is necessary to check whether the queue is empty.

2 Programming (50 points)

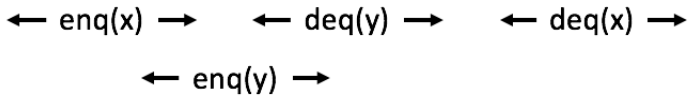
Linearizability is a useful and intuitive consistency correctness condition that is widely used to reason and prove correctness of concurrent implementations of common data structures. Though linearizability is non-blocking, it may impose a synchronization requirement that is stronger than what is needed 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 $\overline{T}$ that has been enqueued k places "before" T has not yet been dequeued.

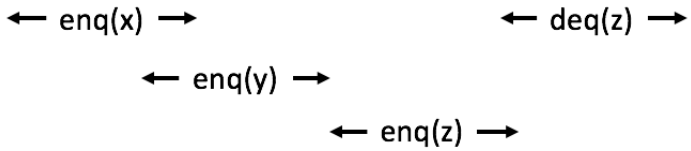
In this assignment, you will construct a n-semi-linearizable queue. To understand semi-linearizable queue, consider the following concurrent executions of a linearizable queue:



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 any linearizable queue (e.g., you can use java.util.concurrent.ConcurrentLinkedQueue from JDK). Plot your throughput as a function of the number of threads. (at least 8 different thread counts, threads varying from 4 to 40. Use Rlogin.)

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.

Your program should be invoked as:

```
java QueueTest <qname> <threads> <duration> [<n>]
```

where `qname` is either LQueue for linearizable queue or SLQueue for semi-linearizable queue, `threads > 0` is the number of test threads, `duration > 0` is the duration of the test in seconds and `n` is the value of n for the semi-linearizable queue (`n` is not required when LQueue is supplied as the `qname`). Each thread should perform both `enqueue` and `dequeue` repeatedly with equal probability.

As output, your program should (from a single-thread) emit a line containing three numbers separated by spaces: the first value should be the total number of enqueues done by all threads, and the second should be the total number of successful dequeues done by all threads, and the third the total number of number of nodes remaining in the queue. A second line of output should emit the throughput. No other output is acceptable.

Your code and writeup (with filename `hw6_<myPID>.pdf`) should be included in a zip file named `hw6_<myPID>.zip` to be uploaded to Canvas by the announced due date.

```
1  public class LockFreeStack<T> {
2      AtomicReference<Node> top = new AtomicReference<
         Node>(null);
3      static final int MIN_DELAY = 10;
4      static final int MAX_DELAY = 100;
5      Backoff backoff = new Backoff(MIN_DELAY,
         MAX_DELAY);
6
7      protected boolean tryPush(Node node) {
8          Node oldTop = top.get();
9          node.next = oldTop;
10         return (top.compareAndSet(oldTop, node));
11     }
12
13     public void push(T value) {
14         Node node = new Node(value);
15         while(true) {
16             if(tryPush(node)) {
17                 return;
18             } else {
19                 backoff.backoff();
20             }
21         }
22     }
```

Figure 2: Lock-free Unbounded Stack

```
1  protected Node tryPop() throws EmptyException {
2          Node oldTop = top.get();
3          if (oldTop == null) {
4              throw EmptyException();
5          }
6          Node newTop = oldTop.next;
7          if (top.compareAndSet(oldTop, newTop)) {
8              return oldTop;
9          } else {
10             return null;
11         }
12     }
13
14     public T pop() throws EmptyException {
15         while(true) {
16             Node returnNode = tryPop();
17             if (returnNode != null) {
18                 return returnNode.value;
19             } else {
20                 backoff.backoff();
21             }
22         }
23     }
24
25     public class Node {
26         public T value;
27         public Node next;
28         public Node(T value) {
29             this.value = value;
30             this.next = null;
31         }
32     }
```

Figure 3: Lock-free Unbounded Stack

```
1  public class YetAnotherQueue<T> {
2      AtomicReference<T>[] bucket;
3      AtomicInteger last;
4
5      public void enq(T item) {
6          int idx = last.getAndIncrement();
7          bucket[idx].set(item);
8      }
9
10     public T deq() {
11         while (true) {
12             int end = last.get();
13             for (int idx = 0; idx < end; idx++) {
14                 T value = bucket[idx].getAndSet(null
                       );
15                 if (value != null) {
16                     return value;
17                 }
18             }
19         }
20     }
21
22 }
```

Figure 4: YetAnotherQueue - Unbounded

```
1  public T deq() throws EmptyException {
2      T result;
3      deqLock.lock();
4      try {
5            if (head.next == null) {
6              throw new EmptyException();
7            }
8            result = head.next.value;
9            head = head.next;
10       } finally {
11           deqLock.unlock();
12     }
13     return result;
14 }
```

Figure 5: UnboundedQueue - deq() method