Challenge 3

Nonblocking Concurrent Queue using LL/SC Synchronization

Background

Certain multicore architectures provide a concurrency mechanism in the form of a pair of **atomic** operations:

- load-linked (LL)
- store-conditional (SC).

The semantics of these operations are specified as follows. To each scalar memory location `v`, there is associated a set of thread IDs, `valid_v`. If `tid` is the ID of the executing thread, then the two operations behave as follows:

```
LL(v):
  valid_v ← valid_v U {tid};
  return v;

SC(v,x):
  if (tid in valid_v) {
    valid_v ← emptyset;
    v ← x;
    return true;
  } else
    return false;
```

`LL(v)` returns the value stored in `v` as usual, but also adds `tid` to the valid set of `v`. `SC(v,x)` checks to see if the executing thread is in the valid set of `v`; if it is, then it clears `valid_v`, stores the value `x` in `v`, and returns *true*; otherwise it returns *false* without modifying `v` or `valid_v`.

A thread typically uses these operations as follows:

```
t ← LL(v)      // reads v
   …
if (SC(v,x)) … // succeeds only if no thread modified v after the read above
```

In 2008, Claude Evéquoz proposed a concurrent FIFO queue based on this mechanism. To keep things simple, we assume the element type is `int`, `-1` indicates a "null" entry, and nonnegative values represent non-null entries. The shared data structures and enqueue/dequeue operations are shown below:

```C/C++
int Q[LEN];  // array with indexes in 0..LEN-1
unsigned int Head, Tail;

bool enqueue(int val) {
  unsigned int t, tailSlot;
  int slot;
  while (true) {
```

```
    t = Tail;
    if (t == Head + LEN)
      return false; // queue is full
    tailSlot = t % LEN;
    slot = LL(&Q[tailSlot]);
    if (t == Tail) {
      if (slot != null) {
        if (LL(&Tail) == t)
          SC(&Tail, t+1);
      } else if (SC(&Q[tailSlot], val)) {
        if (LL(&Tail) == t)
          SC(&Tail, t+1);
        return true; // success
      }
    }
  }
}

int dequeue() {
  unsigned int h, headSlot;
  int slot;
  while (1) {
    h = Head;
    if (h == Tail)
      return null; // empty
    headSlot = h % LEN;
    slot = LL(&Q[headSlot]);
    if (h == Head) {
      if (slot == null) {
        if (LL(&Head) == h)
          SC(&Head, h+1);
      } else if (SC(&Q[headSlot], null)) {
        if (LL(&Head) == h)
          SC(&Head, h+1);
        return slot; // success
      }
    }
  }
}
```

Notes:

- The implementation uses array `Q` as a cyclic bounded buffer. Initially, `Head` = `Tail` = 0 and `Q[i]` = `null` (-1) for 0 ≤ i < `LEN`.
- `Head` and `Tail` increase monotonically; for this challenge, you may assume the "`unsigned int`" type is unbounded.

- The number of elements stored in the queue is `Tail - Head`, and these elements are located at positions `Head%LEN`, `(Head+1)%LEN`, ..., `(Tail-1)%LEN` of `Q`.
- We assume a sequentially consistent memory model, i.e., an execution is an interleaved sequence of atomic actions from the different threads, and the value read from a memory location is the last value written to that location.

Tasks

These can be done in any order. Simplify or add assumptions as needed. **The first set of tasks use only the enqueue operation**:

1. In your favorite language, write a program **P** incorporating Evéquoz's FIFO queue (only the `enqueue` operation is needed). The queue is initially empty. **P** generates `NT` threads (`NT ≥ 1`), with IDs `0`, ..., `NT-1`. Each thread calls enqueue on its thread ID, then terminates.
2. Show that all executions of **P** terminate (i.e., all threads terminate).
3. Show that no out-of-bound array indexes occur on any execution of **P**.
4. Assuming `NT ≤ LEN`, show that in any execution of **P**,
    a. all calls to enqueue return *true* (success);
    b. at the final state, the size of the queue (`Tail-Head`) is `NT`;
    c. at the final state, the contents of the queue are some permutation of the integers `0`, ..., `NT-1`.
5. Assuming `NT > LEN`, show that in any execution of **P**, at the final state,
    a. the queue is full (`Tail-Head=LEN`)
    b. the data in the queue is some permutation of a subset of size LEN of `0`, ..., `NT-1`.

If you have time, add the dequeue function...

Let program **P'** be like **P**, except that each thread first enqueues its ID, then dequeues an entry, storing the result in some variable.

6. Show that all executions of **P'** terminate.
7. Show that no out-of-bound array indexes occur on any execution of **P'**.
8. Assuming `NT ≤ LEN`, show that in any execution of **P'**,
    a. at the final state, the queue is empty: `Tail=Head` and all entries are `null`
    b. each call to `enqueue` returns *true*
    c. the set of values returned by dequeue is exactly { `0`, ..., `NT-1` }.