Array-Based Queuing Lock

Array-Based Queuing Lock (ABQL) is a variation of the Ticket Lock algorithm with a bounded number of concurrent threads and improved scalability due to better cache behaviour.

We assume that there are `N` threads and we allocate a shared Boolean array `pass[]` of length `N`. We also allocate a shared integer value `next`. In practice, `next` is an unsigned bounded integer that wraps to 0 on overflow, and we assume that the maximal value of `next` is of the form $k\texttt{N}-1$. Finally, we assume at our disposal an atomic `fetch_and_add` instruction, such that `fetch_and_add(next,1)` increments the value of `next` by 1 and returns the original value of `next`.

The elements of `pass[]` are spinlocks, assigned individually to each thread in the waiting queue. Initially, each element of `pass[]` is set to `false`, except `pass[0]` which is set to `true`, allowing the first coming thread to acquire the lock. Variable `next` contains the number of the first available place in the waiting queue and is initialized to 0.

Here is an implementation of the locking algorithm in pseudocode:

```
procedure abql_init()
    for i = 1 to N - 1 do
        pass[i] := false
    end-for
    pass[0] := true
    next := 0
end-procedure

function abql_acquire()
    var my_ticket := fetch_and_add(next,1) mod N
    while not pass[my_ticket] do
    end-while
    return my_ticket
end-function

procedure abql_release(my_ticket)
    pass[my_ticket] := false
    pass[(my_ticket + 1) mod N] := true
end-procedure
```

Each thread that acquires the lock must eventually release it by calling `abql_release(my_ticket)`, where `my_ticket` is the return value of the earlier call of `abql_acquire()`. We assume that no thread tries to re-acquire the lock while already holding it, neither it attempts to release the lock which it does not possess.

Notice that the first assignment in `abql_release()` can be moved at the end of `abql_acquire()`.

Verification task 1. Verify the safety of ABQL under the given assumptions. Specifically, you should prove that no two threads can hold the lock at any given time.

Verification task 2. Verify the fairness, namely that the threads acquire the lock in order of request.

Verification task 3. Verify the liveness under a fair scheduler, namely that each thread requesting the lock will eventually acquire it.

You have liberty of adapting the implementation and specification of the concurrent setting as best suited for your verification tool. In particular, solutions with a fixed value of `N` are acceptable. We expect, however, that the general idea of the algorithm and the non-deterministic behaviour of the scheduler shall be preserved.