MPRI 2.18.2: Solutions for Quiz 2

1 Relaxing atomicity

1.1 Regular registers in Peterson's mutex

The exercise is not well-stated, as we have not defined how a regular register should behave when it can be concurrently updated by multiple writers. But Peterson's algorithm uses a shared variable `turn` that can be written by both processes.

But let us assume that `turn` ensures that every read operation returns:

- A concurrently written value, in case the read operation is concurrent with a write operation,
- The last written value, if the last preceding write operation to terminate before the read operation starts is not concurrent with another write operation,
- *one of the last concurrently written values*, otherwise.

Note that if a few values are written concurrently, a subsequent read operation can choose to return any of them.

Now the following scenario is possible:

- p_0 sets `flag`[0];
- p_1 sets `flag`[1];
- p_0 and p_1 concurrently write to `turn`, 1 and 0, respectively;
- p_0 and p_1 read 0 and 1 in `turn`, respectively;
- p_0 and p_1 concurrently enter their critical sections.

Note, however, that if we keep `turn` to be atomic and let `flag`[0] and `flag`[1] to be only *safe*, the algorithm is still correct (left to the reader to prove).

1.2 Original Bakery

Coming back to it on October 15.

2 Order of cleaning

To see that the resulting algorithm is incorrect consider a run in which *write*(1) completes, then *write*(0) completes (leaving the array in the state $[1, 1, 0, \ldots]$), and suppose that *write*(2) sets the array to $[1, 1, 1, \ldots]$, starts cleaning it "bottom-up" by setting $R[0]$ to 0, and falls asleep (leaving the array in the state $[0, 1, 1, \ldots]$).

A concurrent *read*() will then return 1, violating regularity (the last written value is 0 and the concurrently written value is 2).

3 Cleaning before writing

Suppose that *write*(2) completes, then *write*(0) completes and let (leaving the array in the state $[1, 0, 1, \ldots]$), and suppose that *write*(1) starts cleaning by setting $R[0]$ to 0, and falls asleep (leaving the array in the state $[0, 0, 1, \ldots]$).

A concurrent *read*() will then return 2, violating regularity (the last written value is 0 and the concurrently written value is 1).

4 Multi-valued atomic register with atomic bits?

The register implemented by the algorithm in slide 27 is not going to be atomic even if we use atomic binary registers as base objects.

A possible counter-example for just two processes, the writer and the reader, is constructed as follows. The writer sequentially performs *write*(3), *write*(1), and *write*(2). The concurrent operations of the reader are scheduled as follows.

- The reader starts its read operation just before the beginning of *write*(1) and finds 0 in $R[1]$.
- The writer completes *write*(1), then starts *write*(1) by setting $R[2]$ to 1. Note that at this moment both $R[1]$ and $R[2]$ are set to 1 (the writer has not yet cleaned the array up).
- The reader finishes its read operation by finding 1 in $R[2]$ and returning 2.
- Then the reader starts another read operation, finds 1 in $R[1]$ and returns 1—a new/old inversion.

Interestingly, the algorithm can be fixed by imposing more work on the reader. Imagine that the reader first goes "uphill" until it finds a 1 in some $R[i]$ and then "downhill" from position i down to 1 and returning the *lowest* position of R that stores 1. Note that in the scenario above the first read would return the "conservative" value 1, and not 2.

The reader is invited to check that the algorithm precludes new/old inversions.

5 Unbounded regular register?

Imagine now that the array $R[]$ is unbounded and suppose that the writer can write an arbitrarily large natural number m by setting entry $R[m]$ to 1 and cleaning all entries $R[m-1]$ to $R[0]$ by setting them to 0.

The code of the reader is thesame: it reads he array starting from $R[0]$ until 1 is found.

Though the algorithm is still regular: every read operation returns either a concurrently written value or the last written value, it is not wait-free: if the writer writer larger and larger values, a concurrent read may never terminate: whenever it reaches a position i it is already cleaned by the writer (writing a value larger than i). The algorithm is however *lock-free*: a corect process may forever block, but only if another correct process (the writer in this case) cimpletes infinitely many operations.