EFREI M1: Distributed Algoruthms 2019: Solutions for Quiz 3

1 Original Bakery

We prove first *mutual exclusion*: no two processes are in their critical sections at the same time.

Assume the contrary: p_i with ticket number ℓ_i and p_j with ticket number ℓ_j are at the critical section at a given time t_c. Assume that $(\ell_i, i) << (\ell_j, j)$.

Notice that the *binary* registers *flag*[i] and *flag*[i] are only updated in order to *change* their values (setting it from *true* to *false* or vice versa). Thus, as we have seen in the class, the registers behave like *regular* ones: only the last written or a concurrently written values can be read in them.

Thus, when p_j passes the first waiting phase (waiting until p_i is not in the doorway), it reads *false* in *flag*[i] written by a concurrent or a preceding write by p_i.

Let w_f be the last write on *flag*[i] that p_i performs before t_c. By the algorithm p_i writes *false* in w_f. Let r_f be the last read of *flag*[i] that p_j performs before t_c. By the algorithm r_f returns *false*.

Two cases are possible:

- w_f is performed *before or concurrenty* with r_f.

  In this case, every read of *label*[i] performed by p_j after reading *flag*[i] and before enetering its critical section at time t_c is *not* concurrent with any write on *label*[i] by p_i and, by the definition of a safe register, every such read must return ℓ_i.

  Since, by our assumption, $(\ell_i, i) << (\ell_j, j)$, p_j cannot be in its critical section at time t_c—a contradiction.

- w_f is performed *after* r_f.

  Thus, for r_f to return *false*, the preceding write w'_f of *true* to *flag*[i] must be performed by p_i *after or concurrently with* r_f. Thus, when read of *label*[j] performed by p_i after w'_f is not overlapping with a write on *label*[j] and must return ℓ_j. By the algorithm, $\ell_i \geq \ell_j + 1$ and, thus, $(\ell_j, j) << (\ell_i, i)$—a contradiction.

Proving starvation-freedom is left as an exercise.

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).