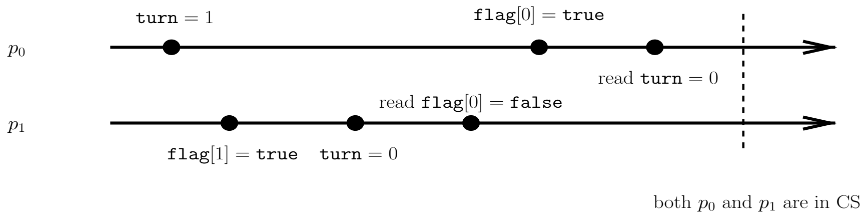
EFREI M1: Distributed Algoruthms 2019: Solutions for Quiz 1

1 2-process Peterson's algorithm

Suppose that p_0 executes the first two lines of its algorithm in the reverse order:

1. `turn` $= 1$;
2. `flag`$[0] =$ `true`;

Then the following execution scenario is possible:



(Note that we do not care about the order in which the first two lines are executed by p_1.)
Here p_0 sets `turn` to 1, then p_1 sets `turn` to 0, `flag`[0] to `true` (the order in which these two operations are performed does not matter) reads `false` in `flag`[0] and proceeds to the critical section. Then p_0 reads 0 in `turn` and also proceeds to the critical section—a contradiction.

2 N-process Peterson's algorithm

Algorithm 1 N-process Peterson's algorithm

```
Shared variables:
  level[0, ..., N − 1] = {−1}
  waiting[0, ..., N − 2] = {−1}

Trying section: code for process p_i:
  for m from 0 to N − 2 do
    level[i] = m;
    waiting[m] = i;
    while(waiting[m] == i && (∃ k ≠ i : level[k] >= m));
Critical section:
  ...
Exit section:
  level[i] = −1;
```

Mutual exclusion. To prove that Algorithm 1 ensures the property of mutual exclusion, suppose, by contradiction, that it has an execution in which two processes are in their critical sections at some time t.

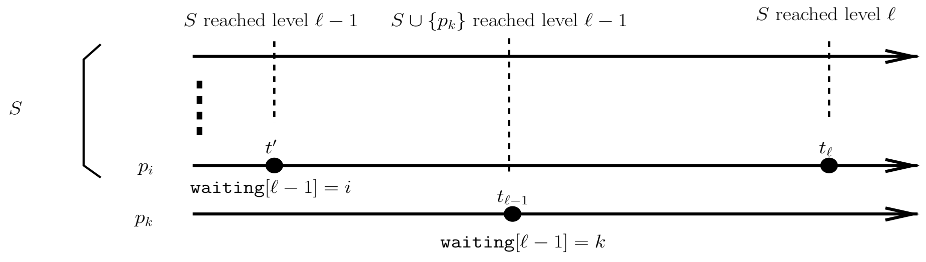
We say that a process p_i *reached level* ℓ ($\ell = 0, \dots, N-1$) if it is in the critical section or `level`$[i]$ stores ℓ or a *higher* value. Thus, by our assumption, two processes reached level $N-1$ at the same time.

Intuitively, a process that reached level ℓ is in the critical section or in the waiting phase ℓ or higher. By the algorithm, a process p_i executing its ℓ-th waiting phase should wait for every process that reached level ℓ to complete their critical sections, unless there is another process that wrote to *waiting*$[\ell]$ after p_i.

Suppose, inductively, that for some $\ell = N-1$ down to 1, a set S of $N-\ell+1$ processes reached level ℓ or higher at some time t_ℓ. (In the base case, $\ell = N-1$ and we have a set of 2 such processes.)

By the algorithm, before time t_ℓ, every process $p_i \in S$ sets `level`$[i]$ to $\ell-1$ and writes i in *waiting*$[\ell-1]$. Without loss of generality, assume that p_i is the last process in S to update *waiting*$[\ell-1]$ before t_ℓ, and let t' be the time when this happens. Hence, at time t', for every other process in $p_j \in S$, `level`$[j]$ stores $\ell-1$ or a higher value. Indeed, if at time t', for some process $p_j \in S$, `level`$[j]$ stores a value less than $\ell-1$, then to reach level ℓ by time t_ℓ, p_j must write j to `waiting`$[\ell-1]$ at some time between t' and t_ℓ, contradicting the assumption that p_i is the last process in S to write to `waiting`$[\ell-1]$ before t_ℓ.

Since $|S| = N-\ell+1$ and $\ell \leq N-1$, there is at least one process in S besides p_i. Thus, to reach level ℓ, between t' and t_ℓ, p_i must have read a value other than i in *waiting*$[\ell-1]$: otherwise, p_i would have to wait until all other processes in S complete their critical sections and set their `level` variables to -1. Thus, at some time $t_{\ell-1}$ between t' and t_ℓ, a process $p_k \notin S$ has written k in *waiting*$[\ell-1]$. Thus, at time $t_{\ell-1}$, at least $|S|+1 = N-\ell+2$ processes reached level $\ell-1$.



By induction, we derive that at some time t_0, at least $N+1$ process must have reached level 0, contradicting the fact that we have exactly N processes.

Starvation-freedom. Now we prove that Algorithm 1 ensures the property of starvation-freedom, i.e., assuming that no process fails in the trying, critical, or exit sections, every process in the trying section eventually enters its critical section. By the algorithm, the only possiblity for a process in the trying section not to enter its critical section is to *block* in line 8 at some level $\ell = 0, \dots, N-2$. A process p_i blocks at level ℓ if, after setting `level`$[i]$ to ℓ and `waiting`$[\ell]$ to i, it keeps reading `waiting`$[\ell]$ and `level`$[0, \dots, N-1]$ to always find `waiting`$[\ell]$ $==$ 1 and `level`$[j] \geq \ell$ for some $j \neq i$. Since, prior to this, every process p_i writes i in `waiting`$[\ell]$, at most one process can be blocked at any given level.

Suppose, by contradiction that there exists a non-empty set B of blocked processes, and let p_I be the process that is blocked at the highest level ℓ. Let t be the time when p_i writes i to `waiting`$[\ell]$ for the last time. Thus, any process p_j that reaches level ℓ must have written j to `waiting`$[\ell]$ before t: otherwise, p_i would eventually read a value other than i and "unblock". Moreover, any such process that p_j must eventually complete level ℓ and proceed to the critical section: otherwise, it would block at a level higher than ℓ, violating our choice of p_i.

Thus, eventually, p_i would find out that no other process has reached level ℓ and proceed to level $\ell + 1$ or its critical section if $\ell = N - 2$—a contradiction.

3 Safety

Safety of an implementation

The set of runs of an implementation I is trivially *prefix-closed*: every prefix of a run of I is also a run if I.

Suppose that all *finite* runs of I are *safe* (with respect to some safety property P). We want to show that even *infinite* runs of I are also in P.

Let σ be any infinite run of I. Let σ_1, …, σ_k, … be prefixes of σ, where σ_i, $i = 1, 2, \ldots$, has length i. By our assumption, every σ_i is in P. Since P is limit-closed, $\sigma = \lim i \to \infty \sigma_i$ is also in P.

Checking safety

We want to argue that to check that a safety property P is violated, we can look for a *finite* run.

Indeed, consider a run $\sigma \notin P$. If σ is finite we are done: for every extension σ' of σ, we have $\sigma' \notin P$ (otherwise, P is not prefix-closed).

Let σ be infinite. Suppose, by contradiction, that σ has no unsafe prefixes, Then, by limit-closedness of P, we get that σ (as the infinite limit of these safe prefixes) is safe—a contradiction.

Determining safety

Given a property P, we want to construct S, a safety property, and L, a liveness property, such that $P = S \cap L$.

S can be constructed as a *prefix-* and *limit-closure* of P, defined as P plus all prefixes and limits of runs in P:

$$\begin{aligned} S = \quad & \{\sigma : \exists \sigma' \in P,\ \sigma \textit{ is a prefix of } \sigma'\} \cup \\ & \{\sigma : \exists \sigma_1, \sigma_2, \ldots \in P,\ \forall i, \sigma_i \textit{ is a prefix of } \sigma_{i+1},\ \sigma = \lim_{i \to \infty} \sigma_i\} \end{aligned}$$

By construction, S is prefix- *and* limit-closed.

We define L as the *largest possible set* that gives P under intersection with S:

$$L = P \cup \neg S$$

Recall that a liveness property must contain extensions of *all* possible runs: something good should always be able to happen eventually. In this sense, it is better to make L as large as possible.

By construction, $S \cap L = P$.

It remains to show that L is indeed a liveness property, i.e., for every finite σ, there exists $\sigma' \in L$, an extension of σ.

Consider any $\sigma \notin L$. By the definiton of L, $\sigma \in S - P$, and, by the definition of S, σ is either a finite prefix of a run in P or an infinite limit of a sequence of runs in P. Since, σ is finite, we derive that an extension of σ is in P.

4 Liveness

First of all, we observe that wait-freedom (WF) is a subset of every other property in the table, i.e., WF is the strongest liveness property in the set.

Consider obstruction-freedom (OF) and lock-freedom (LF) and take any run $\sigma \in LF$. LF is an independent property, so it guarantees progress to some process in all runs, while OF only guarantees progress if some process runs in isolation (for sufficiently long). Therefore, every run in LF is also in OF. Further, any run in which no process ever runs in isolation, e.g., in which processes run one-by-one in the round-robin order, but no process makes progress is, *trivially*, in OF, not in LF. Thus, $LF \subsetneq OF$.

Here we use standard logical reasoning. Consider a set of runs defined as follows:

$$P = \{\sigma : A\sigma \Rightarrow B\sigma\},$$

i.e., P consists of all runs σ, such that if σ satisfies A, then it satisfies B. Then any run that *does not* satisfy A is *trivially* in P.

For example, consider the property: "I like all fruits, but if it is an apple, then I only like red ones." Then if you give me an orange, I should like it.

Similarly, when deadlock-freedom says: "if every process is correct, then some process makes progress", a run in which not every process is correct, is trivially deadlock-free.

The remaining relations can be established analogously.

5 Queue implementations

The case of one enqueuer and one dequeuer

In the sequential impemenation presented in Slide 40 of the first lecture (only the `deq` operation is descrbed, the complementing `enq` operation is naturally deined), only the enqueuer is updating *tail* and only the dequeuer can update *head*. Thus, the only possible race conditions that can be observed here are of the *read-write* type.

Formally, assuming that both *head* and *tail* are atomic shared variables, we linearize *deq* at the moment it writes to *head*, and *enq*—at the moment when it writes to *tail*. Showing that the resulting history is legal is left as an exercise.

Two-lock algorithm

The algorithm discussed above can be easily generalized to the multi-enqueuer multi-dequeueur case by using two locks: one for *head* and one for *tail*. An enqueuer must hold the lock on *tail* before add an element and a dequeuer must hold the lock on *head* to fetch an element.

As all *enq* (resp., *deq*) can only be executed sequentially, any execution of this algorithm can be seen as an execution of the one-enqueuer one-dequeuer algorithm above. The liveness property of the algorithm is determined by the type of locks we use: e.g., by using starvation-free locks, we obtain a starvation-free queue implementation.