Problem 1: Bakery with Safe Registers

Show that the original Lamport’s bakery algorithm (slide 21 in class01-intro.pdf) is correct even when all the registers it uses are only safe.

Solution. 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 \( \ell_i \) and \( p_j \) with ticket number \( \ell_j \) 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 \( flag[j] \) 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 concurrently with \( r_f \).
  
  In this case, every read of \( label[i] \) performed by \( p_j \) after reading \( flag[i] \) and before attending 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 \( \ell_i \).
  
  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 \( \ell_j \). By the algorithm, \( \ell_i \geq \ell_j + 1 \) and, thus, \( (\ell_j, j) < (\ell_i, i) \)—a contradiction.

To prove starvation-freedom, assume, by contradiction again, that a process \( p_i \) is blocked forever in its trying section, even though every process is correct.

Without loss of generality, assume that \( p_i \)’s ticket number \( \ell_i \) is such that \( (\ell_i, i) < (\ell_j, j) \) for every other process \( p_j \) that is blocked forever in its trying section. By the algorithm, there is a time after which for all such blocked processes \( p_j \), \( flag[j] = false \).

Now the two following cases are possible:

- All processes are blocked in their trying sections. Hence, eventually, \( p_i \) will find out that \( (\ell_i, i) < (\ell_j, j) \) for all \( j \neq i \) and enter its critical section—a contradiction.

- Some process \( p_k \) is not blocked in its trying section. Eventually, \( p_k \) will exit its trying section and set \( label[k] \) to a value higher than \( \ell_i \) and \( flag[k] \) to \( false \). Again, eventually, \( p_i \) will find out that \( (\ell_i, i) < (\ell_j, j) \) for all \( j \neq i \) and enter its critical section—a contradiction.
Problem 2: Safety and Liveness

A property is a set of histories. Here we consider histories in which processes propose values in \{0, 1\} and then output values in \{commit, abort\}. We assume that in a history, a process proposes a value at most once, outputs a value at most once, and only if it previously proposed a value.

Classify the following properties into safety/liveness. If a property is an intersection of the two, specify the corresponding safety and liveness properties. Justify your answers.

- Every process eventually outputs a value.
  Every finite execution can be extended to contain a commit or abort event for every process. Thus, this is a liveness property.

- If every process proposes 1 and no process crashes (stops taking steps), then no process can output abort.
  Any finite execution in which every process proposes 1 and some process outputs abort can be extended to an infinite one in which some process is faulty (takes only finitely many steps). Thus, this is a liveness property.

- Eventually, all processes output the same value.
  As we assumed that a process outputs a value at most once, any finite execution in which two different values are output violates the property.
  An infinite execution in which some process never outputs a value also violates the property.
  Thus, the property is a mixture of safety and liveness. We can represent it as the intersection of the liveness property:
    - Every process eventually outputs,
  and the safety property:
    - No two processes output different values.

Problem 3: Progress Conditions

We say that a property \( P \) is stronger than a property \( P' \) if \( P \subseteq P' \). What is the relation between starvation-freedom (SF) and lock-freedom (LF)? Explain why.

Solution. The two properties are incomparable: LF \( \not\subseteq \) SF and SF \( \not\subseteq \) LF.

Indeed, an execution in which every process is correct but only one process makes progress (which can, e.g., happen in our the lock-free atomic snapshot algorithm discussed in the lecture) is in LF but not in SF.

Further, any execution in which some process is faulty and no process makes progress is in SF (trivially, as the condition on the scheduler imposed by SF) but not in LF.

Problem 4: Atomic Registers

Consider the implementation of a one-writer \( N \)-reader (1WNR) atomic register (Transformation V in the slides).

In the read() operation, the process writes the value it just read back to RR[[]]. Is it possible to find an implementation in which the reader does not write? Justify your answer.
Solution. Suppose, by contradiction that such an implementation is possible.

Let the writer change the value of the implemented register from 0 to 1. Let the corresponding write operation modify a sequence of registers \( R_1, \ldots, R_k \) and let \( \omega_1, \ldots, \omega_k \) be the corresponding (atomic) write operations.

Let \( v_{i,\ell}, \ell = 1, \ldots, k \), denote the value that a read operation performed by \( p_i \) and applied right after \( \omega_\ell \) must return. Let \( v_{i,0} \) denote the value that \( p_i \) will return just before \( \omega_1 \).

Claim 1. \( v_{i,0} = 0, v_{i,k} = 1, \) and for all \( \ell = 1, \ldots, k-1, v_{i,\ell} \in \{0,1\} \).

Immediate from the fact that the implemented register is atomic.

Claim 2. For all \( \ell = 1, \ldots, k \), \( i \) and \( j \), \( v_{i,\ell} = v_{j,\ell} \).

Indeed, suppose by contradiction, that for some \( \ell \in \{1, \ldots, k-1\} \), we have \( v_{i,\ell} \neq v_{j,\ell} \). By Claim 1, we can suppose, without loss of generality, that \( v_{i,\ell} = 1 \) and \( v_{j,\ell} = 0 \).

Now we schedule, just after \( \omega_\ell \), a read operation of \( p_i \) followed by a read operation by \( p_j \). By the definition of \( v_{j,\ell} \), the read operation of \( p_i \) must return 1. Further, as implemented read operations do not modify the memory, the read operation of \( p_i \) must return \( v_{j,\ell} = 0 \).

Thus, we constructed a new-old inversion, establishing a contradiction.

Claim 3. There exists \( \ell \in \{1, \ldots, k\} \), such that for all \( i \), \( v_{i,\ell-1} = 0 \) and \( v_{i,\ell} = 1 \).

Immediate from Claim 1 and Claim 2.

Consider \( \ell \) established in Claim 3. Recall that \( \omega_\ell \) is an atomic write operation on a 1W1R register \( R_\ell \). Let \( R_\ell \) be read by a distinct process \( p_i \). As \( R_\ell \) can only be read by \( p_i \), no process \( p_j \), \( j \neq i \), can distinguish its read operation executed just before \( \omega_\ell \) from its read operation executed just after \( \omega_\ell \).

Thus, for all \( j \neq i \), we have \( v_{j,\ell-1} = v_{j,\ell} \), contradicting Claim 2.

Problem 5: ABA in Atomic Snapshots

Show that the atomic snapshot is subject to the ABA problem (affecting correctness) in case the written values are not unique.

Solution. Figure 1 gives an example of a run in which \( p_1 \) and \( p_2 \) update the memory concurrently with a snapshot taken by \( p_2 \). In the first scan, \( p_2 \) sees the old value of \( p_1 \) (1) and the new value of \( p_3 \) (2), then \( p_3 \) and \( p_1 \) write back their “old” values (in this order), and then we repeat this scenario with the second scan of \( p_2 \).

The resulting execution is not linearizable: there is no place between the updates where we can linearize the snapshot operation by \( p_2 \).

Figure 1: ABA in atomic snapshots: \( p_2 \) gets two identical scans, but the scan outcome (in red) does not belong to the set of allowed snapshots (in blue).