Concurrent Algorithms October 26, 2021

Exercise 3 Solution

Problem 1. The transformation does not work for multiple readers (the result is not an atomic register). The non-atomic execution in Figure 1 is possible in this case. Since the register is regular, the read by R1 may read the value 2 being concurrently written by W. Since this is R1's first read operation, the timestamp it obtains for value 2 is higher than its local timestamp. Later, the read by R2 (also concurrent with *Write(2)*) may read the previous value of the register (1). Since this is R2's first read operation, the timestamp it obtains for value 1 is also higher than its local timestamp.

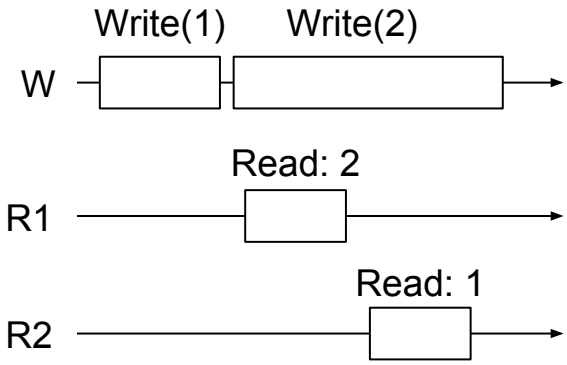

Figure 1: An example execution that violates atomicity.

Problem 2. The transformation does not work for multiple writers because each writer has a local timestamp, i.e., the different writers do not share the same time. In fact, consider two writers W_1 and W_2. W_1 is a more active writer than W_2. Assume W_1 has already written 10 times to the register ($t_1 = 10$). After that, W_2 writes for the first time to the register, its local timestamp will then be less than 10 ($t_2 = 1$). Therefore, any subsequent read after W_2's write will miss W_2's newly written value, and return the last value written by W_1. This violates the sequential property of registers, i.e., a sequential read after a write operation should always return the last written value.

Problem 3. **The notion of regular registers is not well defined in the case of multiple writers. Thus, this exercise is not in the scope of the exam, and is only left as practice.**

Yes, the transformation works. The proof follows:

Let Reg[N] denote the vector of binary MWMR registers that would compose the N-valued MWMR regular register R. To prove that the same algorithm yields an N-valued MWMR regular register, we must prove that any read operation always returns a valid value from R. As regularity is only well-defined for *single writer* registers, we provide now a generalization. When considering the read operation, a valid value can be either:

- any value v written by a concurrent write operation ($v \in concurrent$).

- any value v written by the latest starting and completed write operation[1], or any operations concurrent with it ($v \in recent$).

For simplicity and without loss of generality, we also restrict *recent* to exclude any operation also in *concurrent*, so that $recent \cap concurrent = \emptyset$.

For any single reader, we prove the following[2]:

1. $\exists i : Reg[i] = 1$
2. $Reg[i] = 1 \Rightarrow i \in recent \vee i \in concurrent$

The first case means that the reader always terminates and returns a value while the second case ensures that this value is valid (for a regular register). This concludes our proof after we finish proving each of the two Lemmas.

Lemma 1 $\exists i : Reg[i] = 1$

Proof $Reg[0]$ is initially 1. If it is read 0, it follows that some $R.write(j) : j > 0$ has completed or is concurrently writing 0 to $Reg[0]$. As the writing is done right to left, it must then have finished writing 1 in $Reg[j]$. If $Reg[j] = 0$, the same logic applies for some $k > j$ recursively up to $Reg[N]$, which is never erased ($Reg[N].write(0)$ is never called) by any writer. □

Lemma 2 $Reg[i] = 1 \Rightarrow i \in recent \vee i \in concurrent$

Proof Take the first (and only) $i : Reg[i] = 1$ reached by the reader (i.e., the value returned by the read). Suppose for the sake of contradiction that $i \notin concurrent \wedge i \notin recent$. This is equivalent to *both*:

1. There is no concurrent write $R.write(i)$ ($\iff i \notin concurrent$). Let us denote the last completed (with earliest starting point) $R.write(i)$ by any writer by W_i.
2. $\exists W_j = R.write(j) : j \neq i \wedge j \in recent \wedge W_j$ started after W_i completed ($\iff i \notin recent$)

Looking at the second statement, there are only two possible cases:

- a) If $j > i$, then the read of $Reg[i] = 0$ since Reg[i] is regular, $R.write[j]$ should have finished writing $Reg[i].write(0)$ and there are no writes of $Reg[i].write(1)$ concurrent with $R.write(j)$ (2.) or after (1. and $j \in recent$). But $Reg[i] = 1$. *Contradiction*
- b) Else $j < i$, we have that $Reg[j] = 0$ (because $Reg[i] = 1 \Rightarrow Reg[j] = 0, \forall j < i$, ATTA[3]). This means that $Reg[j]$ must have been erased by a later or concurrent write $W_k = R.write(k) : k > j$ ($\Rightarrow k \in recent \vee k \in concurrent$)
  - b.1) If $k > i$ then W_k should have already finished $Reg[i].write(0)$ since it is either writing of has finished writing $Reg[j].write(0)$ and it writes Reg[i].write(0) first ATTA ($i > j$), which was last set to 1 before W_j started (by supposition). But $Reg[i] = 1$. *Contradiction*
  - b.2) If $k < i$ then we have case b) again (recursively). Since there are a finite number of values between j and i, eventually we end up in case b.3)

[1] By this, we mean the write operation with the latest starting point that has returned before the read in question started.
[2] $Reg[i] = v$ means that the reader in question has read v from $Reg[i]$ (we only ever talk about a single reader).
[3] According to the algorithm

- b.3) If $k = i$, in which case $W_k = R.write(i)$ which erased $Reg[j]$ must have been concurrent with W_j, contradicting 2, or happened after W_j and is concurrent with this read (since $j \in recent$), contradicting 1. *Contradiction*

This concludes the contradiction, so we have: $\overline{i \notin concurrent \wedge i \notin recent} \iff i \in recent \vee i \in concurrent$. Thus i is a valid return.

□