Concurrent Computing September 16, 2025

Solutions to Exercise 1

Problem 1. Regular registers allow readers to read either the value of the latest finished write (latest write that happened before the read) or any write concurrent to the read. Note: regular (and safe) registers are only defined in single writers contexts.

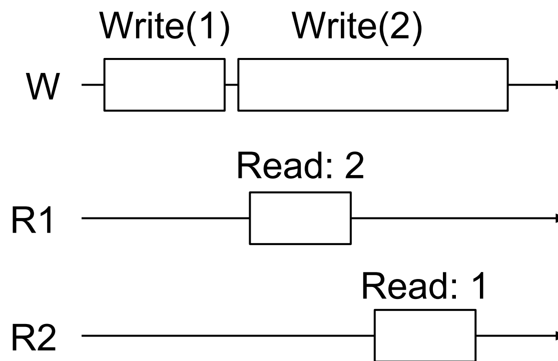

Figure 1: An execution that is possible with a regular register but not with an atomic register. There are three processes: a writer (W) and two readers (R1 and R2). The two reads are concurrent with the second write. The read by R1 completely precedes the read by R2. The execution is not atomic because it is impossible to assign linearization points to all operations: if the linearization point of *Write(2)* is before that of the read by R1, then the read by R2 cannot have a linearization point; if the linearization point of *Write(2)* is after that of the read by R1, then that read cannot have a linearization point. Execution (a) of problem 2 is another example of execution allowed by a regular register but not by an atomic one.

Problem 2.

Part 2.a.

Regular, not atomic. This is because of the second read from p3 reading an older value than the first read. This would not be allowed in atomic execution, but is allowed in regular execution as 1 was the latest value written before the write concurrent to the 2nd read. The other reads would be allowed in any execution.

Part 2.b.

None of the above. As the read from p4 is not concurrent to any write, it should read the latest value even with a safe register. The other reads would be allowed in any execution.

Part 2.c.
Atomic.

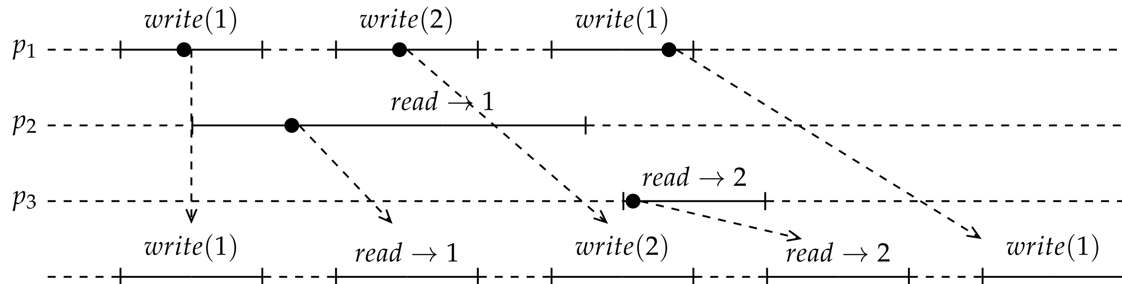

Figure 2: Serialization points and an equivalent sequential execution.

Problem 3. Consider the transformation from binary MRSW safe registers to binary MRSW regular registers, given in class.

Part 3.a. Prove that the transformation does **not** generate **multi-valued** MRSW **regular** registers (from **multi-valued** MRSW **safe** base registers) by providing a counterexample that breaks regularity.

If the registers are multi-valued, then two consecutive reads on the safe register *Reg* may return arbitrary values, breaking regularity of the register implementation. Since the safe register is binary in the correct implementation (and thus limited to two values), this does not occur in the transformation given in class.

Part 3.b. Also, prove that the resulting registers (in the original transformation) are not binary **atomic** (just regular) by providing a counterexample that breaks atomicity.

The counterexample can be easily built by scheduling two distinct reads during a write(1) operation on the register. Since the underlying register is safe, we can ensure that the first operation returns 1, while the second (non-overlapping) operation returns 0, contradicting atomicity.

Problem 4.

Part 1 Please see Chapter 4 of these lecture notes (available on the website) for the proof, page 51.

Here is a shorter, less formal argument: Consider the execution of a reader. Let the original target position t be the location of the 1 written by the most recent completed write w_t:

1. If the reader encounters a 1 at position $i < t$, then this 1 must have been written by some later write w_i, since w_t must have previously reset register i to 0. Thus w_i is concurrent with the read and i is a valid return value.
2. If the reader reads a 0 at position t, it means that some later write w_i (concurrent with the read) has written a 1 at position $i > t$ and has already reset to 0 all intermediate registers j with $t \leq j < i$. In this case, we change the target to $w_t := w_i$ and $t := i$, and recursively apply the same reasoning (cases 1–3) with the new target.

3. If the reader encounters a 1 at position t, then t is a valid return value because w_t is either the latest completed write or one concurrent with the read.

This process must eventually terminate, since each time we advance the target t to a strictly larger index.

Part 2 For this, notice that if the writer first clears the array by writing 0's, it is possible for the value of the array to be all 0's, which is not a valid state.

Part 3 Figure 3 presents an example that violates atomicity. Such an execution can occur if the first *read* operation of p_2 gets 0 while retrieving $Reg[7]$ and gets 1 while retrieving $Reg[1000]$. This can occur since both $write(7)$ and $write(1000)$ are concurrent with the *read* operation. Afterwards, the second *read* operation of p_2 will return 7 since $write(1000)$ has not yet set $Reg[7]$ to zero.

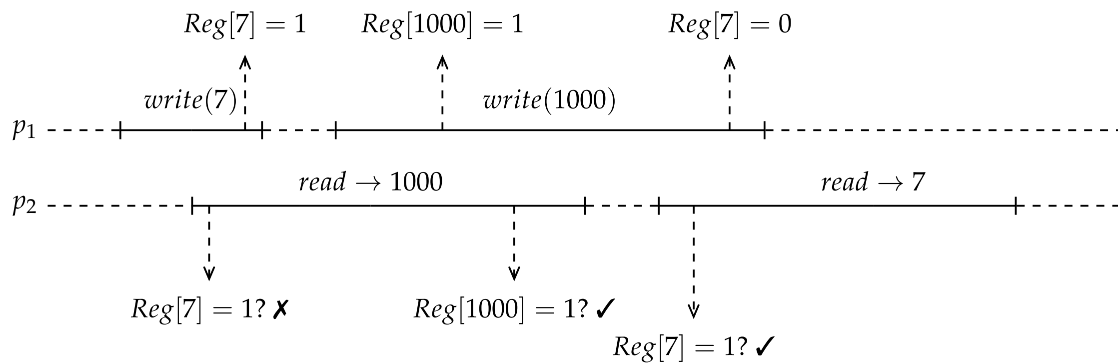

Figure 3: Execution that violates atomicity.