Concurrent Computing October 1, 2024

Solutions to Exercise 1

Problem 1.
Part 1.a. Regular, not atomic.

Part 1.b. None of the above.

Part 1.c.
Atomic.

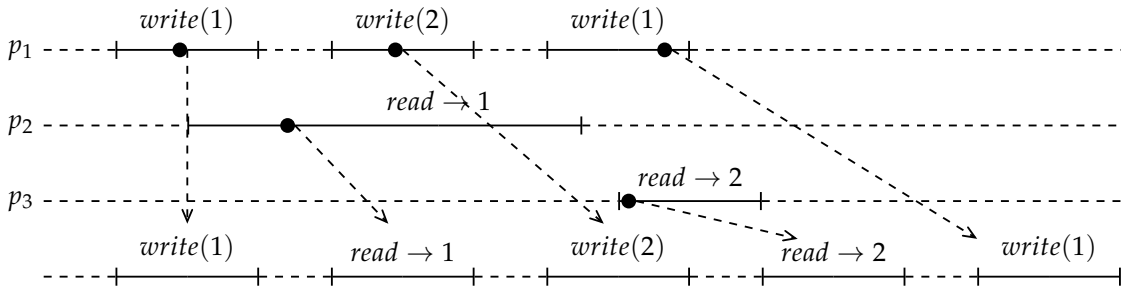

Figure 1: Serialization points and an equivalent sequential execution.

Problem 2. Consider the transformation from (binary) SRSW safe to (binary) MRSW safe registers given in class.

Part 2.a. Prove that the transformation works for multi-valued registers and regular registers.

When a process p_i reads the base regular register $Reg[i]$, p_i gets (a) the value of a concurrent write on $Reg[i]$ (if any) or (b) the last value written to $Reg[i]$ before such concurrent write operations. In case (a), the value v obtained is from a $R.write(v)$ that is concurrent with the *read* of p_i. In case (b), the value v obtained can either be (b.1) from a $R.write(v)$ that is concurrent with the *read* of p_i, or (b.2) from the last value written by a $R.write()$ before the *read* of p_i . Thus, the constructed register is regular.

Part 2.b. Also, prove that the transformation does not work for atomic registers (by providing a counterexample that breaks atomicity).

See execution in Figure 2.

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

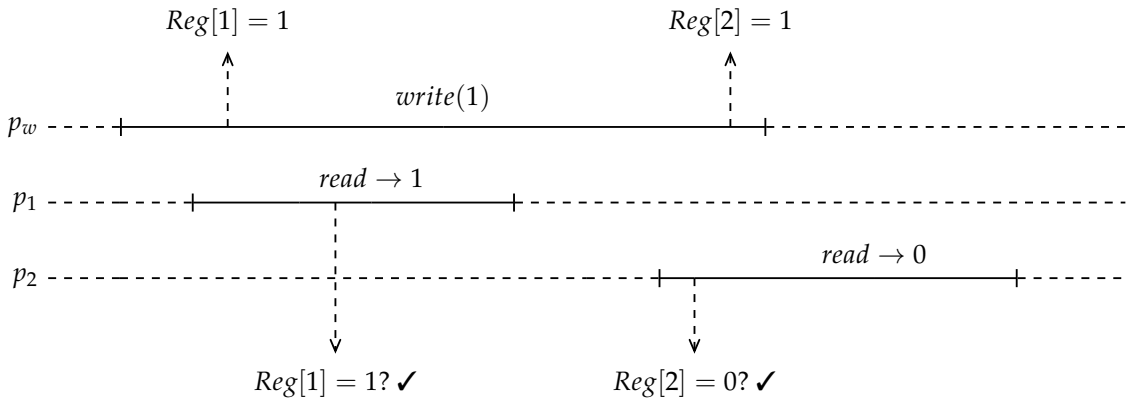

Figure 2: Execution that violates atomicity.

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.

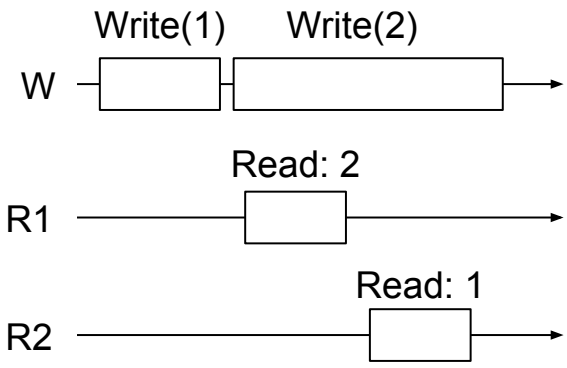

Figure 3: An execution that is possible with a regular register but not with an atomic register. There are three processes: a writer (W) and two readers (R1 and R2). The two reads are concurrent with the second write. The read by R1 completely precedes the read by R2. The execution is not atomic because 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.

Problem 5.

Part 1 Please see Chapter 4 of these lecture notes (first item of Supplementary Material on the website), page 51.

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 4 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.

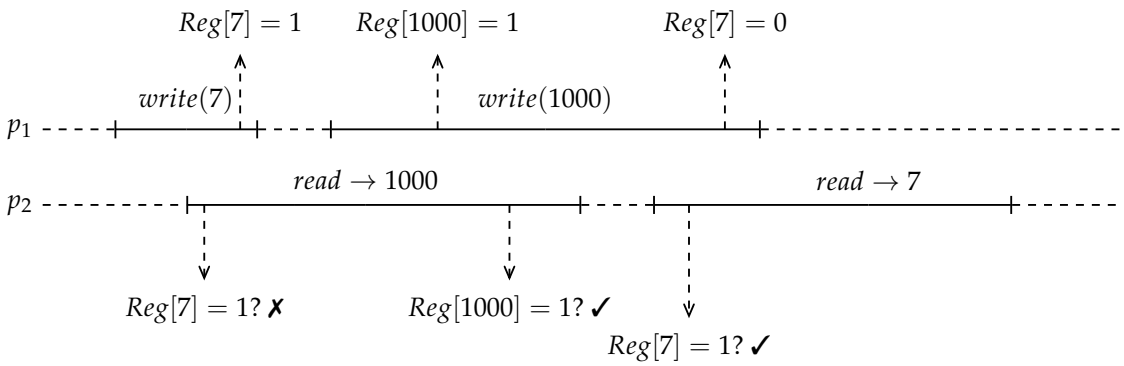

Figure 4: Execution that violates atomicity.