Concurrent Algorithms
Lecture Notes
Writing while reading registers

R. Guerraoui
Distributed Programming Laboratory

October 22, 2009

Lecture notes written by Laurent Bindschaedler (SIN).

Contents

1. Foreword

In the previous lessons, we saw how to implement Multi-Reader-Multi-Writer (MRMW) atomic registers from Single-Reader-Single-Writer(SRSW) safe registers. Some of these algorithms made use of many lower-level registers to implement a single higher-level register. This lesson we will show that we can't do better than these algorithms.

We will challenge two assumptions:

- Readers need to write[1]
- We have an infinite number of registers[2]

When we say that readers write, we mean that the *Read* operation on the higher-level register needs to use *write* (not only *read*) operations on the lower-level register(s).

2. Bound on SWSR atomic register implementations

Theorem 1. *There is no* ***wait-free*** *algorithm that:*

- *Implements a SRSW atomic register,*
- *Uses a* ***finite*** *number of* ***bounded*** *SRSW* ***regular*** *registers,*
- *And where the base registers can only be written by the writer[3].*

In other words, this theorem states that it is **impossible** to have a SRSW atomic register unless you allow infinite memory or readers to write. These requirements are tight: there is no way to do without them. This is called an "impossibility result". This theorem carries very strong implications since it tells you that there is no point trying to implement a register such as the one defined above since no such algorithm can possibly exist.

We now prove this theorem. Since this is the first proof, we will go slowly and try to explain things in as much detail as possible.

Proof of theorem 1. We proceed by contradiction, i.e. we assume that we have an algorithm that implements a SRSW atomic register using a **finite** number of **bounded** SRSW **regular** registers and where the base registers can only be written by the writer. Note that we have no idea what that algorithm could look like and in fact we don't care. We will prove that no matter what it does,

[1] This is due to the problem we ran into when implementing Multi-Reader (MR) registers from Single-Reader (SR) registers.

[2] This follows from the use of timestamps which we needed to introduce to solve the regular to atomic transformation. Indeed, timestamps imply infinite memory because we need the ability to keep incrementing (even after a very long period of time).

[3]By writer, we mean the writer of the atomic register.

it can't work.

The proof idea is as follows: we will build a counterexample[4] where the above-assumed algorithm fails to provide atomicity. If we succeed in building that counterexample, we prove the claim. In building that counterexample, we can carefully craft whatever scenario we want. This is called an "adversary model". In this case, we are the adversary and our goal is to make the algorithm fail.

Let us begin with setting up the context:

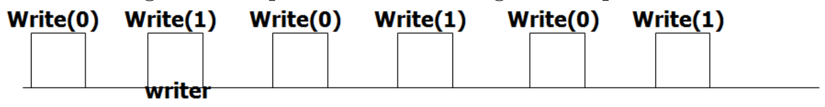
- We will assume that the higher-level register implemented by the algorithm is a *binary* register. Let us denote that register by ***Reg***. This is somewhat of a simplification, but it is without loss of generality (WLOG) since if the algorithm doesn't work for a binary register, it will necessarily not work for a *multi-valued* (M-valued) register.
- We will also assume that we have a single SWSR regular (possibly *M-valued*) register at our disposal. Let us call this register ***reg***. This simplification is also WLOG for the following reason: we can replace n registers of m bits of memory with a single register of $n * m$ bits of memory, encoding each of the n registers as an element of an array contained in the single big register.

Now, we describe a scenario where the algorithm fails.

The writer keeps writing 0s and 1s alternatively and infinitely many times to ***Reg***, i.e. the sequence of operations looks like (see figure 1):

Write(0), Write(1), Write(0), Write(1), Write(0), ...

Figure 1: Sequence of alternating Write operations.



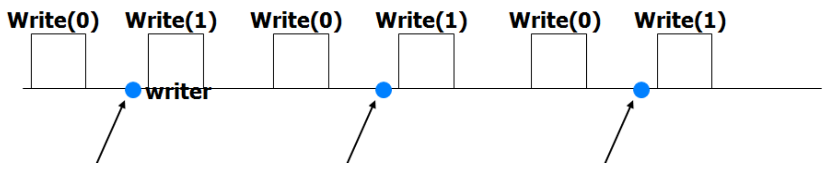
During these operations, ***reg*** may go through several different states[5], e.g. s_0, s_1, s_2, etc. and not necessarily in a deterministic order. However, we know that ***reg*** is bounded, i.e. the number of different internal states is limited (it may be large, but it is finite). This means that, since we're writing infinitely many times to ***Reg***, some state (call it v_0) of ***reg*** must appear infinitely many

[4] This is some kind of a prowess. Again, because we're building a counterexample for an algorithm we don't even know.

[5] Remember that, while the user of ***Reg*** sees 0 or 1, the internal implementation may be much more complicated (it could use several bits to encode the value). Note that this assumption, while potentially counter-intuitive and seemingly not very useful is *necessary* since we want to prove that *any* algorithm will not work.

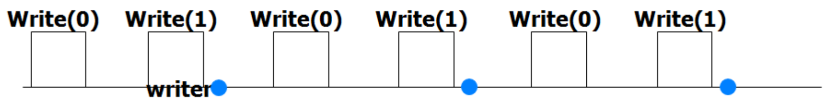
times[6] after a Write(0)[7]. See figure 2.

Figure 2: ***reg*** is in state v_0 after an infinite sequence of Write(0).



We now consider the subset of Write(1) operations starting when ***reg*** is in state v_0. We again use the same argument to state that ***reg*** can only go though a finite number of states (i.e. it can only assume a finite number of values) after a Write(1). This means that there is a value (or a state) v_n which appears infinitely many times in ***reg*** after a Write(1)[8]. See figure 3.

Figure 3: ***reg*** is in state v_n following v_0 after an infinite sequence of Write(1).



Does v_n necessarily follow v_0 immediately? Remember that we need to keep the algorithm general[9]. We therefore allow intermediate states between v_0 and v_n. These occur infinitely many times. We state the following:

There must exist values v_0, v_1, ... v_n[10], such that (see figure 4):

- v_0 is the value of ***reg*** before infinite Write(1) operations,
- v_n is the value of ***reg*** after infinite Write(1) operations,
- $\forall i < n$: ***reg*** changes infinitely many times from v_i to v_{i+1} during infinite Write(1) operations.[11]

[6] Note that we can't say that several states appear infinitely many times. All we can say with absolute certainty that at least one such state does.

[7] We can't guarantee that v_0 appears after every Write(0), but we know that it occurs at least in an infinite subsequence and that is what we are interested in.

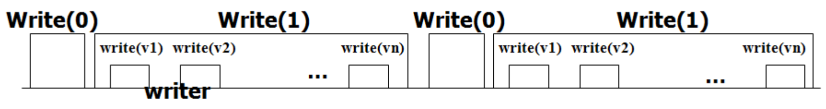
[8] As before, we can't guarantee that v_0 is followed by v_n after every single occurrence. However, we consider subsequences of those occurrences where it is the case. These sequences occur infinitely many times.

[9] Making such an assumption here would reduce the number of possible algorithms and thus an algorithm that would solve the problem could no longer fall under this proof - rendering it useless.

[10] We can at least guarantee v_0 and v_n, which corresponds to the simplest transition possible. However, there could be other intermediate transitions.

[11] Once again, we can't guarantee that the same sequence will occur between every v_0 and v_n, but we are considering the subsequence of those that do. Again, this subsequence is infinitely long.

Figure 4: The sequence of states v_1s $\forall i \in \{0, 1, ...n\}$.

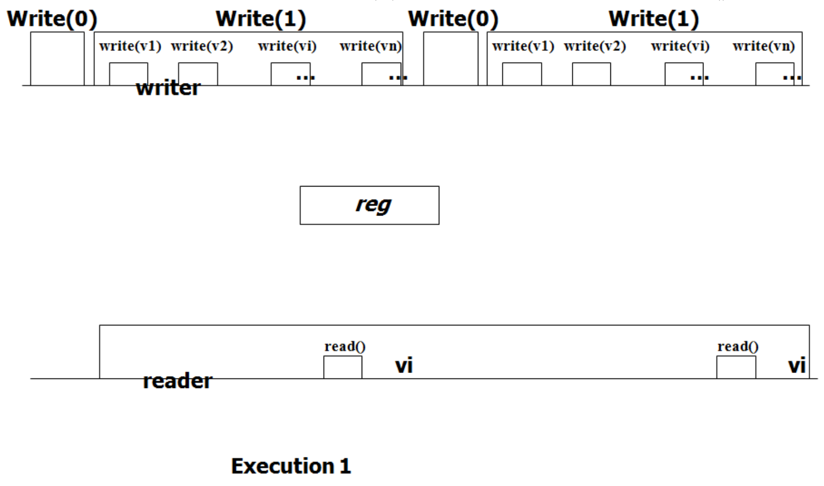


We now insert the reader in the scenario. Remember that we also control what the reader does since we control the full scenario. The reader, just as the writer, does not necessarily perform just one read operation on ***reg***, which means that we must also consider that ***reg*** can go through many different states (v_i) during a Read. We thus consider a sequence of reads.

Let us describe two different executions:

- Execution 1: Consider two Write(1) operations and one Read() operations on ***Reg***. Furthermore, assume that the reader reads ***reg*** when it is in state v_i, concurrently with the first Write(1), and decides to re-read it later, i.e. it reads ***reg*** a second time, concurrently with the second Write(1), and reg is in state v_i again. See figure 5.

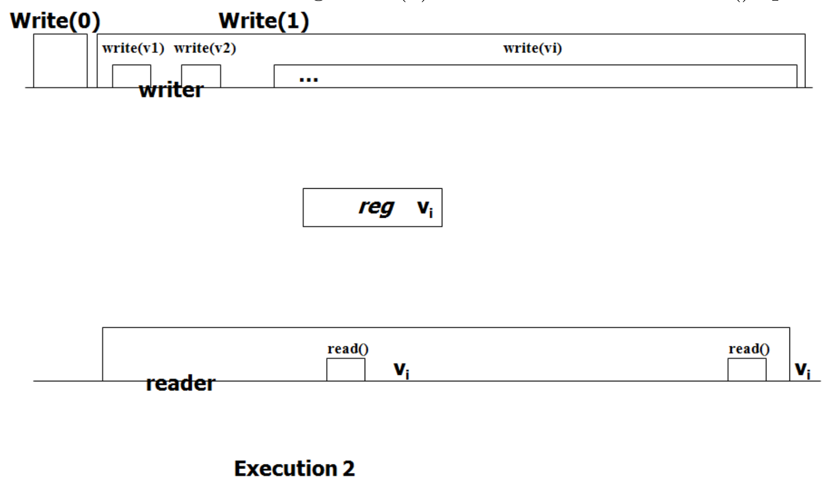
Figure 5: Execution 1 - two Write(1) concurrent with one Read() operation.



- Execution 2: Consider one single and long Write(1) operation and one Read() operation on ***Reg***. Assume, as in case 1 that the reader reads ***reg*** when it is in state v_i, concurrently with the Write(1), and decides to re-read it later, i.e. it reads ***reg*** a second time, concurrently with the same Write(1) - which took a long time to complete -, and reg is still in state v_i. See figure 6.

Remember that the reader is wait-free, that is, it cannot keep reading forever and that it must return a deterministic value. What are the possible values for v_i? If $i = 0$, ***reg*** is in state v_0 and thus the Read() returns value 0. Similarly,

Figure 6: Execution 2 - one long Write(1) concurrent with one Read() operation.



if $i = n$, the Read() returns value 1. What happens when $i \neq 0, n$? We claim there is a minimum i with $0 < i \leq n$ such that:

- If the reader always reads (through read()) v_i, then
  - The reader returns (through Read()) 1.
- If the reader always reads (through read()) v_{i-1}, then
  - The reader returns (through Read()) 0.

Note that in this situation, because the underlying register ***reg*** is regular (this is where we use the last part of the hypothesis), it may be that ***reg*** returns (through read()) v_{i-1} after returning v_i, which will cause the reader to return (through Read()) 1 and then 0. This clearly violates the assumption that ***Reg*** was atomic: executions 1 and 2 appear indistinguisable to the reader, even though they should.

Therefore, since ***Reg*** is not atomic, we have proven our claim that an algorithm algorithm that implements a SRSW atomic register using a **finite** number of **bounded** SRSW **regular** registers and where the base registers can only be written by the writer can't exist. □

Let us now look back on the proof. Consider the simplest candidate for the above algorithm, which would simply forward the 0 or 1 to the underlying register. We know that this algorithm doesn't work (because of read inversions - cf. lesson 2). We assumed (in the above proof) a more complicated candidate algorithm and brought it down to the same problem with 0 and 1 using v_i and v_{i+1}. This somehow conveys philosphically the idea that no matter what we do, we need to deal with this intrisic property of regular registers differently (e.g. using infinite memory or readers writing) - there is no escaping it.

Let us also briefly describe why the above proof wouldn't have worked having assumed that readers could write (and writers read). If that were the case, we could make it so that executions 1 and 2 be distinguishable to the reader. Consider the algorithm in which the reader writes a bit before it calls its first read() operation[12] and have the writer read this bit at the end of its Write(1) operation. All the writer needs to do is write another additional bit at the beginning of the next transition from Write(0) to Write(1). Since the reader now reads this additional bit in the second read() operations, along with v_i, it can distinguish between the two executions.

3. Bound on SWMR atomic register implementations

Theorem 2. *There is no **wait-free** algorithm that:*

- *Implements a <u>SWMR atomic</u> register,*
- *Uses **any** number of SRSW **atomic** registers,*
- *And where the base registers can only be written by the writer*[13].

As for theorem 1, theorem 2 is an *impossibility result.* In short, it states that there is no simple solution to implement Single-Writer-Multiple-Readers (SWMR) registers: we have no choice, readers need to write.

Proof of theorem 1. We proceed again by contradiction, i.e. we assume that we have an algorithm that implements a <u>SWMR atomic</u> register using **any** number of SRSW **atomic** registers and where the base registers can only be written by the writer.

We set up the context:

- We will denote the higher-level register implemented by the algorithm by ***reg****.
- We will assume one writer ***pw*** and two readers ***p1*** and ***p2***. This assumption of only two readers is a simplification, but it is WLOG since if it doesn't work with two readers, it can't work with more than two.
- We replace all atomic underlying registers read by ***p1*** by a single register called ***reg1*** and similarly for ***p2***, we denote the single register by ***reg2***.

Let us now consider the first write of value 1 (Write(1)) in the high-level register ***reg****. This Write operation is implemented as a series of low-level write operations into registers ***reg1*** and ***reg2***: $w_1, w_2, ...w_j, w_{j+1}, ...w_k$.[14]

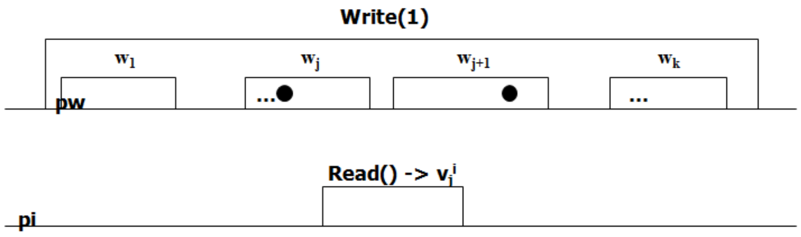
We now consider the Read() operation which access the state of either ***reg1*** or ***reg2*** at state v_j^i (i.e. when the value is read in ***regi*** and when the register has been written w_j - see figure 7). We know that for either ***p1*** or ***p2***, there has to be an index j_i such that for smaller indices the value read is 0 and for

[12] The reader does not change this bit again, until the next Read() operation.
[13] By writer, we mean the writer of the atomic register.
[14] Note that this sequence can write to ***reg1*** or ***reg2*** in any order - this is WLOG.

larger indices it is 1. Furthermore, j_1 cannot be equal to j_2 since the writer cannot write to the two underlying registers simultaneously. We have formally:

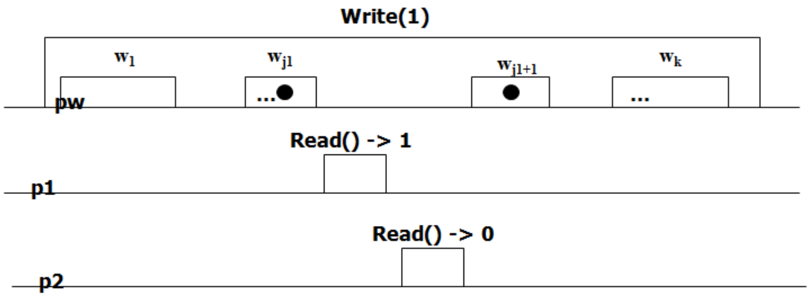
$$\forall i \in \{1,2\}, \exists j_i : 1 \leq j_i \leq k : \forall j < j_i : v_j^i = 0 \text{ and } \forall j \geq j_i : v_j^i = 1$$

Figure 7: Sequence of low-level writes and Read() finding the register in state v_j^i.



Assume WLOG that $j_1 < j_2$ and now consider a Read() operation by ***p1*** followed by a Read() operation by ***p2*** before the writer gets the chance to move on to w_{j+1} (which was going to update ***reg2***'s state so its value would be 1). As a result, it is easy to see that the first Read() by ***p1*** will return 1 whereas the second Read() by ***p2*** will return 0, thus violating the atomicity property (see figure 8).

Figure 8: Proof of theorem 2: the execution that violates atomicity.



□

Let us finish our analysis by considering what would have happened if readers could write. This proof would no longer work since ***w1*** would simply need to write a timestamp along with the value and then ***p1*** could easily write the timestamp and value to ***p2*** before completing the Read().

In conclusion, we have shown with theorems 1 and 2 that readers **must** write in implementations of *multi-reader wait-free atomic* registers[15]. This is true even when the available space is unbounded.

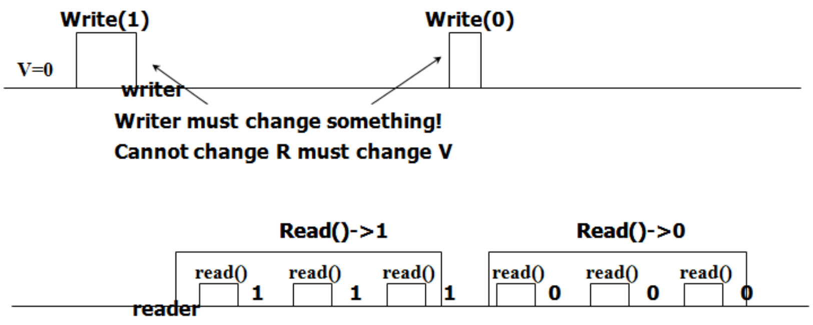
[15] Considering of course that the registers are implemented out of weaker base objects

4. Tromp's algorithm

In this section, we consider the last missing piece: an implementation of a SRSW atomic binary register using SRSW safe binary registers. We aim for time and space complexity of $O(1)$ and we allow readers to write.

The first question to ask is how many registers we need. From now on, we shall use the words bit and register interchangeably (in this context, registers are binary). Will one bit be enough? Certainly not since theorem 1 proves that readers need to write. Can two bits be enough then? No, consider the following simple example (see figure 9) where the writer invoques Write(1) on its high-level register. This Write(1) is repercuted in a write(1) in the base register. However, that base register is *at best* regular and thus the reader could see 0 after seeing 1.

Figure 9: Two bits are not enough.



The last example hints at a fundamental fact: the writer needs (at least) 2 registers so it can indicate that a change was made. Are three bits enough? The short answer is *yes*. After decades of research, a Dutch Ph.D student came up with the solution. This algorithm, which has now become famous, bears his name: Tromp's algorithm.

Let us define the setting:

- The writer has two bits: V, holding the value, and W, the control flag.
- The reader has one bit: R, the control flag.
- Notation: if (W = R) then { ... } = { r := read(R); if (W = r) then ... }

Code listing 1 shows the implementation of Write(v) and code listing 2 shows the implementation of Read(v).

Listing 1: Tromp's algorithm: Write(v)

```
Write(v):
    change(V)
    if ( W = R ) then
        change(W)
```

Listing 2: Tromp's algorithm: Read(v)

```
Read():
    if ( W = R ) then return v
    x := read(V)
    if ( W != R ) then change(R)
    v := read(V)
    if ( W = R ) then return v
    v := read(V)
    return x
```

Behind these seemingly simple 9 lines of code lies probably one of the most difficult algorithms to understand.

Notice that reader and writer handshake through registers W and R:

- $W = R \Leftrightarrow$ there is no new value, i.e. the reader knows the latest value.
- $W \neq R \Leftrightarrow$ there is a new value, i.e. the reader is not up-to-date.

We now give a sketch of the proof. We prove *correctness*, which means *liveness* and *safety*. Proving *liveness* is straightforward - there are no locks or loops. Proving *safety* is going to be a little more difficult. Recall that atomicity means that for every execution, each operation can be considered as taking place instantaneously at a serialization point. More precisely, we have:

- For every execution:
  - There exists a *partial order* of operations such that:
    - All **Write** operations are ordered,
    - Each **Read** operation is ordered with respect to all **Write** operations,
    - Each **Read** operation returns the value of the immediately preceding **Write** operation,
    - If ***op1*** precedes ***op2***, then **not**(***op2*** $<$ ***op1***) is in the ordering.

We define the above-mentioned ordering:

- **Write**s are ordered as they are issued.
- **Read**s are ordered as follows:
  - Find the last Read(V) that precedes the return command (in the **Read** operation).
  - Find the Write(V) that wrote the value returned.
  - The **Write** operation during which the Write(V) was issued is ordered before the **Read** operation.

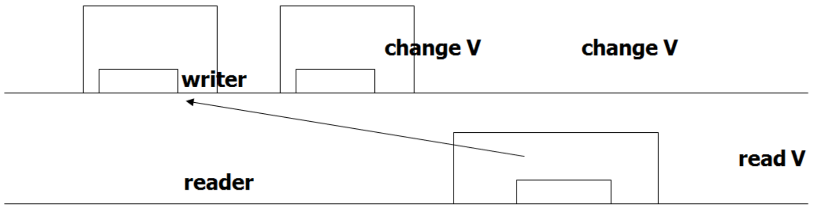
We start by proving that each **Read** operation returns the value of the immediately preceding **Write** operation (*safety* property). Assume for the sake of contradiction that a read returns a written value that was written before the immediately preceding written value (see figure 10). If the Read is concurrent with the second Write operation, there is no problem - we return a valid value. If not, we distinguish two cases:

- The Read returns on line 5 or 7 (these distinguish from a Read that returns on line 1 because at least one read occured before the return). In this case, we return a value read *during* the Read operation. Since V acts like a regular register (given the writes are completed), there is no way that the read returns an old value.

- The Read returns on line 1 (W = R). This means we return the value of variable v that was read during a previous Read operation. However, after a Write operation, we have that W $\neq$ R, which means there had to be a previous Read operation and that Read operation must have read V (as in case 1 - i.e. return on line 5 or 7) and therefore it can't return an old value.

Since both cases end up in a contradiction, we proved the statement.

Figure 10: A read returns a written value that was written before the immediately preceding written value.



We then prove that a **Read** returns the value of the concurrent **Write** or a previous **Write** (*regularity* property). This is pretty easy to see since the writer can only access the shared memory to change the value of the implemented register. If a read is concurrent with a write that changes the value, it is allowed to return both 0 and 1.

Before proving the *atomicity* property, we give and prove the following lemma.

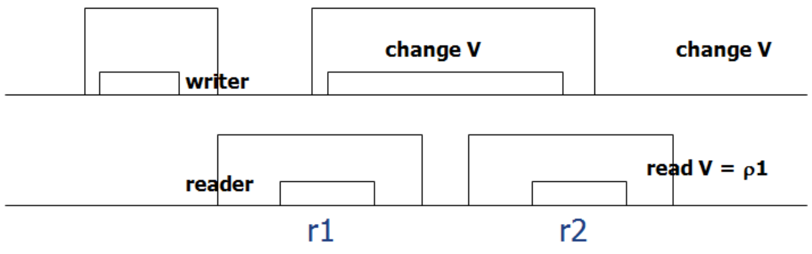
Lemma 3. *If Read* **r1** *precedes* **r2** *and* **ri** *returns the value written by the Write* **vi** $(i = 1, 2)$*, then* **v1** = **v2** *or* **v1** *precedes* **v2***, i.e. there is no read inversion - a Read returns the value of the concurrent Write or a previous Write.*

Proof of lemma 3. Proceed by contradiction and suppose that ***v2*** precedes ***v1***. If that were the case, then ***r1*** would have to return something which is not the initial value and ***r2*** would have to return some value coming from a low-level read of V or the same value as ***r1*** (cf. reader's code). This is clearly a contradiction since ***r2*** cannot return the value written by ***v2***.

□

We now prove that if **Read** ***r1*** precedes **Read** ***r2***, then **not**(***r2*** < ***r1***) (*atomicity* property). Assume for the sake of contradiction that ***r2*** returns a value which was written before the value read by ***r1*** (see figure 11). Let us denote the value returned during Read ***ri*** by ρ_i $(i = 1, 2)$. We will need three claims

Figure 11: ***r2*** returns a value which was written before the value read by ***r1***.



in order to prove the above statement.

Claim 1: ρ_1 precedes ρ_2. We know that $\boldsymbol{\rho_1} \in \boldsymbol{r1}$ (i.e. $\boldsymbol{\rho_1}$ was read during ***r1***) or $\boldsymbol{\rho_1}$ was read during some Read that preceeded ***r1***. We then have:

- $\boldsymbol{\rho_2} \in \boldsymbol{r2}$: The claim trivially holds since ***r1*** is before ***r2***.
- $\boldsymbol{\rho_2} \notin \boldsymbol{r2}$: Then ***r2*** returns in line 1. Notice that $\rho_1 \neq \boldsymbol{\rho_2}$. If $\boldsymbol{\rho_2}$ preceeded ***r1***, then ***r1*** did not change local variable $\boldsymbol{v}$, i.e. it returned in line 1 and we have $\boldsymbol{\rho_1} = \boldsymbol{\rho_2}$. Otherwise, $\boldsymbol{\rho_2} \in \boldsymbol{r1}$ ($\boldsymbol{\rho_2}$ was read during ***r1***), then $\boldsymbol{\rho_1}$ comes from a read in line 2 or 4 of Read ***r1*** or earlier and ρ_2 comes from a read in line 4 or 6 of ***r1*** or later.

In both cases, $\boldsymbol{\rho_1}$ precedes $\boldsymbol{\rho_2}$.

Claim 2: There is a *change(V)* operation by the writer that started before $\boldsymbol{\rho_1}$ finished and which finished after $\boldsymbol{\rho_2}$ started. It is impossible that the *change(V)* be before or during $\boldsymbol{\rho_1}$ and that $\boldsymbol{\rho_2}$ comes after. Hence $\boldsymbol{\rho_2}$ can't return an old value.

Claim 3: Every Read(W) operation by the reader which occurs between $\boldsymbol{\rho_1}$ and $\boldsymbol{\rho_2}$ returns the same value. This follows trivially from Claim 2: since the writer is busy changing V, it can't change W.

We can now prove the main statement: if **Read *r1*** precedes **Read *r2***, then **not**(***r2*** < ***r1***) (*atomicity* property). We distinguish three cases:

- $\boldsymbol{\rho_1}$ is *x := read(V)* (line 2 of the algorithm). It follows that $\boldsymbol{\rho_1} \in \boldsymbol{r1}$ (i.e. $\boldsymbol{\rho_1}$ was read during ***r1***) and ***r1*** returns in line 7 (holds by Claim 2). We then distinguish 2 subcases:
  - $\boldsymbol{\rho_2}$ is the read in line 4 of ***r1***. In this case, ***r1*** does not execute line 6 and ***r1*** returns in line 5 (which contradicts Claim 2).
  - $\boldsymbol{\rho_2}$ is some later read. By Claim 3, W=R in line 5 of ***r1*** and ***r1*** returns in line 5 (which contradicts Claim 2).
- $\boldsymbol{\rho_1}$ is *v := read(V)* (line 4 of the algorithm). It follows that ***r1*** must return in line 5 after finding W=R. By Claim 3, W is not changed before $\boldsymbol{\rho_2}$ (i.e. some read(V)) is invoked. However, there is no subsequent read of V (nor change of R) before W $\neq$ R (line 1), i.e. there is no new read of v before W is changed, and thus that $\boldsymbol{\rho_1} = \boldsymbol{\rho_2}$, contradicting Claim 1.

- ρ_1 is *v := read(V)* (line 6 of the algorithm). In this case, ***r1*** is a subsequent read that returns in line 1, since otherwise v is overwritten in line 4. Thus, ***r1*** finds W = R in line 1. By claim 3, W is not changed before ρ_2 (i.e. before some read V) is invoked. However, there is no subsequent read of V (nor change of R) before W $\neq$ R (line 1), i.e. $\boldsymbol{\rho_1} = \boldsymbol{\rho_2}$, contradicting Claim 1.

Since every case ends up in a contradiction, we have proven the statement.

Taking the two and a half pages of proof as a witness and despite its small size, this algorithm is in fact one of the most complicated algorithms. Even Tromp's Ph.D supervisor admitted that he still did not fully understand why the algorithm worked. Understanding why it works is very difficult: it just does. A good way of practising one's understanding is to try to remove one line and analyze what fails in the modified algorithm.

5. Conclusion

We showed that a new concept arises in the implementation of wait-free atomic multi-reader registers: the fact that readers need to write. This was done methodologically, using proofs by contradiction to obtain what is called an "impossibility result". This fact raises one important question: how efficient are wait-free implementations of registers knowing that writing is on average ten times slower than reading? However, this tradeoff seems worth it considering the benefits of using wait-free algorithms (e.g. liveness, safety).

We then presented Tromp's algorithm implementing a wait-free atomic binary register using three safe binary registers. This was the missing piece of our analysis of wait-free register implementations and is a major result in this field. During the process, we noticed how complex the topic is and how difficult it is to prove (and to convince ourselves) that a wait-free algorithm (even a short one) really works.