# Writing while reading registers

*Marko Vukolic*
*Distributed Programming Laboratory*





## When readers need to write?

- To improve complexity
  - Reader-writer communication
- To facilitate multiple readers (atomic regs)
  - Reader-reader communication



## From SRSW regular to SRSW atomic

- We use one SRSW ***register*** Reg and two local variables t and x
- **Read()**
  - (t',x') = Reg.read();
  - if t' > t then t:=t'; x:=x';
  - return(x)
- **Write(v)**
  - t := t+1;
  - Reg.write(v,t);



## From SRSW regular to SRSW atomic

- The transformation would not work for multiple readers
- The transformation would not work without timestamps (variable t representing logical time)
- ***What is behind these limitations?***



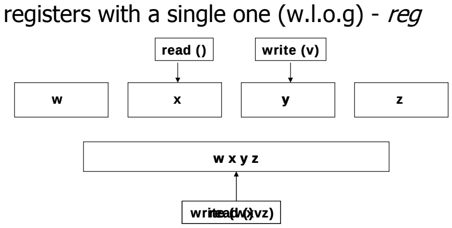
## Bound on SWSR atomic register implementations

- Theorem 1:
  - There is no wait-free algorithm that implements an (SWSR) atomic register using a finite number of (SWSR) regular register that can be written by the writer (of the atomic register).
- I.e., there is no "simple" solution w/o timestamps – readers need to write!



## The proof

- We assume such an algorithm and show contradiction
- We replace any number of **SW**SR regular registers with a single one (w.l.o.g) - *reg*



# The Proof (cont'd)

- Consider an execution in which the writer changes the value of the atomic register (*reg**) from 0 to 1 infinitely many times
- Let zeros[i] denote the state of *reg* after i-th write of 0 in *reg** (before its change to 1)
- *reg* can assume **finite** number of values ⇒ ⇒ there is a value v0 that appears infinitely many times in zeros[]

7

# The Proof (cont'd)

- Consider the changes of *reg** from 0 to 1, strating from the state v0 of *reg*
- *reg* can assume **finite** number of values ⇒ ⇒ there is a value vn that appears infinitely many times in *reg* upon changing *reg** from 0 to 1
- ⇒ the state of *reg* changes infinitely many times from v0 to vn (when *reg** is changed from 0 to 1)

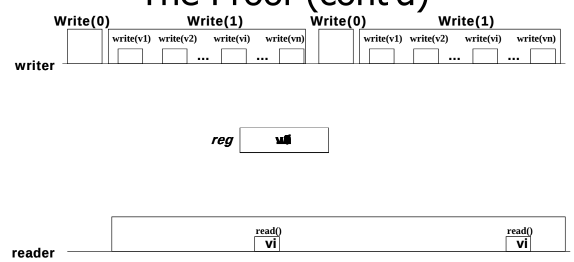
8

# The Proof (cont'd)

- Similarily (generalization): There must exist values $v_0$, $v_1$, … $v_n$, s.t.
  - (i) $v_0$ is the final value of *reg* after each of an infinite number of writes of 0 to *reg**
  - (ii) $v_n$ is the final value of *reg* after each of an infinite number of writes of 1 to *reg**
  - (iii) ∀i<n: *reg* is changed infinitely many times from $v_i$ to $v_{i+1}$ during infinely many changes of *reg** from 0 to 1

9

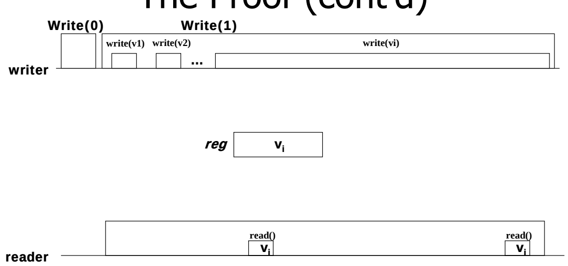
# The Proof (cont'd)




Execution 1

10

# The Proof (cont'd)




Execution 2

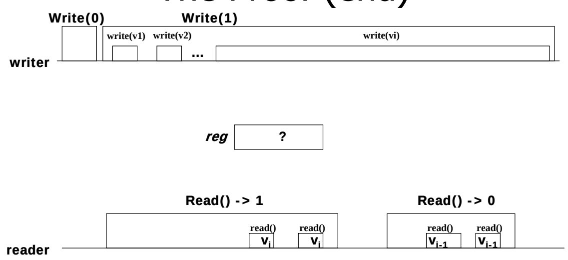
11

# The Proof (cont'd)

- There is a minimum i (0<i<=n) such that:
  - if the reader keeps reading $v_i$, the reader returns 1
  - if the reader keeps reading $v_{i-1}$, the reader returns 0

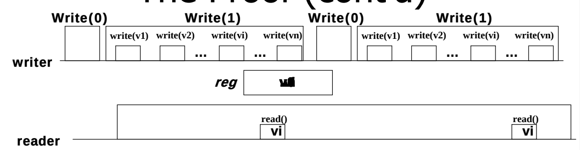
12

# The Proof (end)




13

# The Proof (cont'd)




**If readers write (and writers read), executions 1 and 2 do not have to be indistinguishable to the reader. Execution 1 (shown in this slide) has an infinite no. of writes. We could imagine the algorithm in which the reader writes something (say a bit) before the first low-level read. This is read by writer at the end of Write(1). The reader does not change this bit before next Read.**

**Then, the writer simply writes some aditional bit at the begining of the next change from 0 to 1. Hence, reader reads this in the second low-level read along with vi. This makes the reader distinguish execution 1 from execution 2.**

14

# Summary

- The reader needs to write to reduce the ***complexity***
  - From unbounded space complexity to a bounded one
  - Reader – Writer communication
- The (bounded) algorithm will come a bit later

15

# From SRSW atomic to MRSW atomic (cont'd)

- **Write(v)**
  - t1 := t1+1;
  - for j = 1 to N
    - WReg.write(v,t1);

16

# From SRSW atomic to MRSW atomic (cont'd)

- **Read()**
  - for j = 1 to N do
    - (t[j],x[j]) = RReg[i,j].read();
  - (t[0],x[0]) = WReg[i].read();
  - (t,x) := highest(t[..],x[..]);
  - for j = 1 to N do
    - RReg[j,i].write(t,x);
  - return(x)

17

# From SRSW atomic to MRSW atomic (cont'd)

- The transformation would not work for multiple writers
- The transformation would not work if the readers do not communicate (i.e., if a reader does not write)

18

## Bound on SWMR atomic register implementations

- Theorem 2:
  - There is no *wait-free* algorithm that implements a (SWMR) atomic register using *any* number of (SWSR) atomic registers that can be written by the writer (of the SWMR atomic register).

19

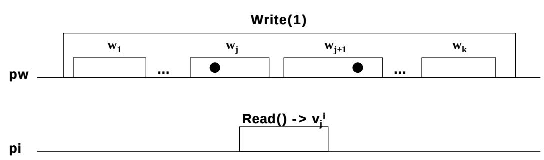
## The proof

- We assume such an algorithm and show contradiction
  - Denote the SWMR register by *reg**
- We assume 2 readers (p1 and p2)
  - The writer is pw
- We replace all atomic registers read by p1 (resp., p2) by a single one – *reg1* (resp., *reg2*)
  - As in the proof of Theorem 1

20

## The proof (cont'd)

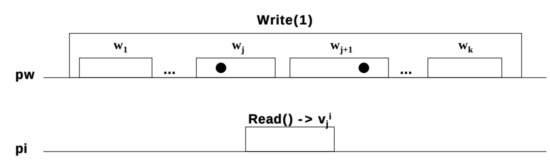
- Consider the first write of 1 into *reg**
- This consists of number of low-level writes w1 to wk into reg1/reg2




21

## The proof (cont'd)

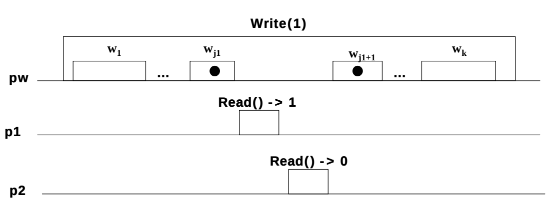
- $\forall i \in \{1,2\}$, $\exists\, j_i$: $1 \le j_i \le k$:
  $\forall j < j_i$: $v_j^i = 0$ and $\forall j \ge j_i$: $v_j^i = 1$
- Observe that $j_1$ does not equal $j_2$
  - $w_{ji}$ must write to *regi*




22

## The proof (end)

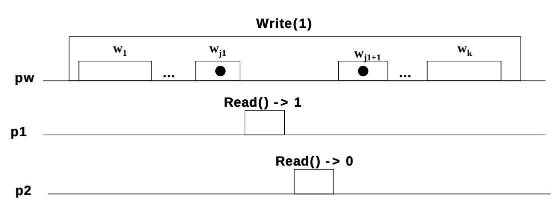
- w.l.o.g. assume $j_1 < j_2$




23

## The proof (end)

- w.l.o.g. assume $j_1 < j_2$




**If readers write, the proof is simple to break. Assume that the writer writes a timestamp along the value. The reader p1 would simply writeback the timestamp/value pair to a dedicated SWSR atomic register read by p2 (as in the transformation seen in the class).**

24

## Summary

- The readers need to write in SWMR wait-free atomic implementations (out of weaker base objects)
  - Applies to implementing SWMR atomic from any number SWMR regular
    - We can implement SWMR regular from SWSR atomic
  - Even when the available space is unbounded
  - Reader – Reader communication



## From safe bits to an atomic one

- We focus on (wait-free) implementing SWSR atomic bit
- Brute force (the reader does not write):
- SWSR safe bit to SWSR regular bit
  - Simple
- SWSR regular bit to SWMR regular multivalued
  - O(N) in space and time
- SWMR regular to SWSR atomic
  - Timestamps (unbounded space)



## From safe bits to an atomic one

- Or try something different
  - The reader should write!
- Aim for O(1) complexity in space and in time

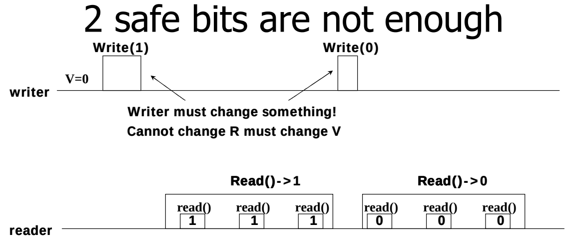


## How many safe bits?

- A single one will not be enough (Theorem 1)
  - We need at least one in which the reader will write
- Can we do it with only 2 SWSR safe bits?
  - No…
- Assume two bits
  - V, written by the writer and read by the reader
  - R, written by the reader and read by the writer



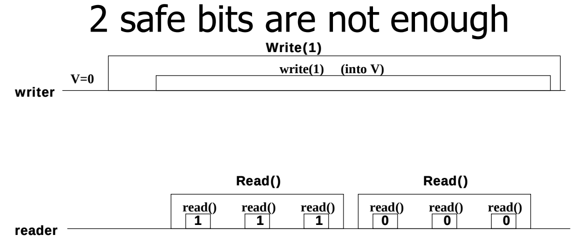
## 2 safe bits are not enough




- After Write(1) V must equal 1
  - Assuming that the initial value is 0
  - Dual if the initial value is 1
- After Write(0) V must equal 0



## 2 safe bits are not enough




- The proof holds regardless of the number of bits in which the reader writes
- The writer needs (at least) 2 bits for himself

## 3 bits are enough (Tromp’s algorithm)

- 2 bits owned (written) by the writer
  - V (for a value) and W (control flag)
- 1 bit owned by the reader (R – control flag)
- When the writer (resp., reader) executes:
  - If W=R then { … }
- We mean:
  - 1) r:=read R (resp., w:=read W)
  - 2) if (W=r) then (resp., if w=R then)
- r (resp., w) is a local variable
- A copy of W (resp., R) is also stored localy

31

## Tromp’s algorithm

- **Write(v)**

0: if old ≠ v then
1: change V;
2: if W=R then
3: change W;
4: old:=v

32

## Tromp’s algorithm

- **Write(v)**

(0: if old ≠ v then)
1: change V;
2: if W=R then
3: change W;
(4: old:=v)

33

## Tromp’s algorithm

- **Write(v)**

1: change V;
2: if W=R then
3: change W;

**- Handshaking**

W≠R ⇔ there is a new value
W=R ⇔ no new values

- **Read()**

1: if W=R then return v
2: x := read V
3: if W≠R then change R
4: v := read V
5: if W=R then return v
6: v := read V
7: return x

34

## Correctness

- Liveness – straigthforward
- Safety – a bit more difficult
- We first prove regularity
  - Read-Write linearizability
- Then we show that a later Read never returns an older value than some preceding Read
  - Read-Read linearizability

35

## Correctness - Regularity

- 2 cases:
  - A Read is concurrent with some Write
    - Simple: left as an exercise
  - A Read is not concurrent with any Write
    - Proved here

36

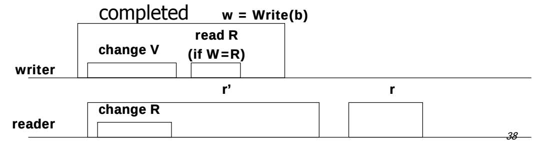
## Correctness - Regularity (cont'd)

- Assume the Read r is not concurrent with any Write
- Let w be the last complete Write preceding the Read writing the bit *b*
  - If r returns in line 5 or 7 then the returned value has been read during r
    - By safety of V, r returns *b*

37

## Correctness - Regularity (cont'd)

- If r returns in line 1
  - then the reader saw W=R (in line 1)
  - Before completing w, the writer made W≠R
  - There was a Read r' (s.t. r' precedes r) that completed change R ***after*** the read of R in w started
    - Otherwise, the writer would again make W≠R
    - i.e., r' completed change R after change V in w completed




38

## Correctness - Regularity (cont'd)

- If r returns in line 1
  - then the reader saw W=R (in line 1)
  - Before completing w, the writer made W≠R
  - There was a Read r' (s.t. r' precedes r) that changed R ***after*** the read of R in w started
    - Otherwise, the writer would again make W≠R
    - i.e., r' changed R after change V in w completed
    - Let r' be the first such Read
  - In line 4 of r' reader reads v:=*b* (by safety of V)
  - All subsequent Reads read v=*b* (including r) until another Write is invoked

39

## Read-read linearizability

- **Lemma:** 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
- **Proof:** Suppose v2 precedes v1 (*)
- r1 does not return the initial value (no Write precedes the initial Write)
- r2 returns some value read by some low-level read from V
  - Otherwise r2 returns the same value as r1 (the initial value)
    - See line 1 of reader's code

40

## Read-read linearizability (cont'd)

- Let ρi be the read from V returned by ri (i=1..2)
- **Claim 1:** ρ1 precedes ρ2 (ρ1→ρ2)
  - ∀i ∈ {1,2}: ρi∈ri or ρi is belong to some read that precedes ri
  - If ρ2∈r2 Claim 1 is trivial (since r1→r2)
  - If ρ2∉r2, r2 returns in line 1 and ρ2 is the latest v := read V (in line 4 or 6) that precedes r2

41

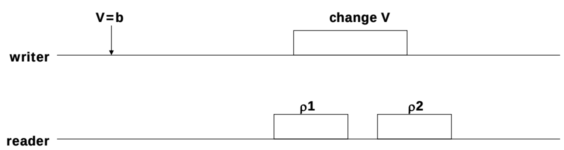
## Read-read linearizability (cont'd)

- **Claim 1 (cont'd):** ρ1 precedes ρ2 (ρ1→ρ2)
  - It is not possible that ρ2→ρ1
    - Observe that ρ1≠ρ2 by (*)
    - If ρ2→r1 then r1 does not change v
      - r1 returns in line 1 and ρ1=ρ2
    - If ρ2∈r1
      - ρ1 is a read V in line 2 or 4 of r1, or some earlier read, while
      - ρ2 is a read V in line 4 or 6 of r1
  - Hence (by (*)) ρ1→ρ2!

42

## Read-read linearizability (cont'd)

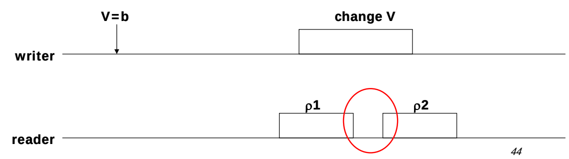
- **Claim 2:** there is a change V operation by writer that started before $\rho 1$ finished and finished after $\rho 2$ started




43

## Read-read linearizability (cont'd)

- **Claim 3:** Every read W operation (lines 1,3,5) by the reader between $\rho 1$ and $\rho 2$ returns the same value
- **Proof:** the writer is busy changing V (Claim 2)




44

## Read-read linearizability (cont'd)

- There are 3 exhaustive cases
- (i) $\rho 1$ is x := read V (line 2)
  - $\rho 1 \in$ r1 and r1 returns in line 7 (**)
  - 2 subcases:
    - (a) $\rho 2$ is the read in line 4 of r1
      - Then r1 does not execute line 6
      - r1 returns in line 5 (contradicts (**))!
    - (b) $\rho 2$ is some later read
      - By Claim 3, W=R in line 5 of r1
      - r1 returns in line 5 (contradicts (**))!

45

## Read-read linearizability (cont'd)

- There are 3 exhaustive cases
- (ii) $\rho 1$ is v := read V (line 4)
  - r1 must return in line 5
    - After finding W=R
  - By Claim 3, W is not changed before $\rho 2$ (i.e., some read V) is invoked
  - But 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 $\Rightarrow$ $\rho 1 = \rho 2$ – a contradiction w. Claim 1, (*)

46

## Read-read linearizability (end)

- There are 3 exhaustive cases
- (iii) $\rho 1$ is v := read V (line 6)
  - r1 is a subsequent read that returns in line 1
    - Otherwise v is overwritten in line 4
    - r1 finds W=R in line 1
  - By Claim 3, W is not changed before $\rho 2$ (i.e., some read V) is invoked
  - But there is no subsequent read of V, (nor change of R), before W$\neq$R (line 1)
    - i.e., as in case (ii) $\Rightarrow$ $\rho 1 = \rho 2$ – a contradiction w. Claim 1, (*)

47

## Tromp's algorithm

- **Write(v)**

1: change V;
2: if W=R then
3: change W;

**- Handshaking**

W$\neq$R $\Leftrightarrow$ there is a new value
W=R $\Leftrightarrow$ no new values

- **Read()**

1: if W=R then return v
2: x := read V
3: ~~if W≠R then~~ change R
4: v := read V
5: if W=R then return v
6: v := read V
7: return x

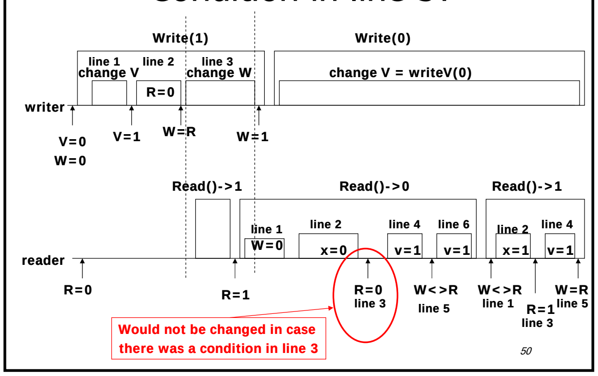
48

# Condition in line 3?

- There are 3 exhaustive cases
- (i) ρ1 is x := read V (line 2)
  - ρ1∈r1 and r1 returns in line 7 (**)
  - 2 subcases:
    - (a) ρ2 is the read in line 4 of r1
      - Then r1 does not execute line 6
      - r1 returns in line 5 (contradicts (**))!
    - (b) ρ2 is some later read
      - By Claim 3, W=R in line 5 of r1
      - r1 returns in line 5 (contradicts (**))!



# Condition in line 3?






# Exercise

- **Write(v)**

1: change V;
2: if W=R then
3: change W;

- **Read()**

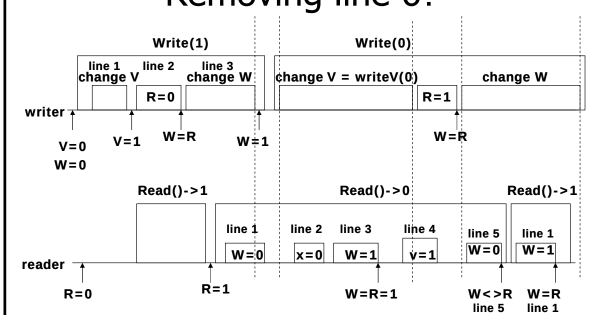
1: if W=R then return v
2: x := read V
3: if W≠R then change R
4: v := read V
5: if W=R then return v
~~6: v := read V~~
7: return x

**- Handshaking**

W≠R ⇔ there is a new value
W=R ⇔ no new values



# Removing line 6?






# Removing line 6?