Writing while reading registers

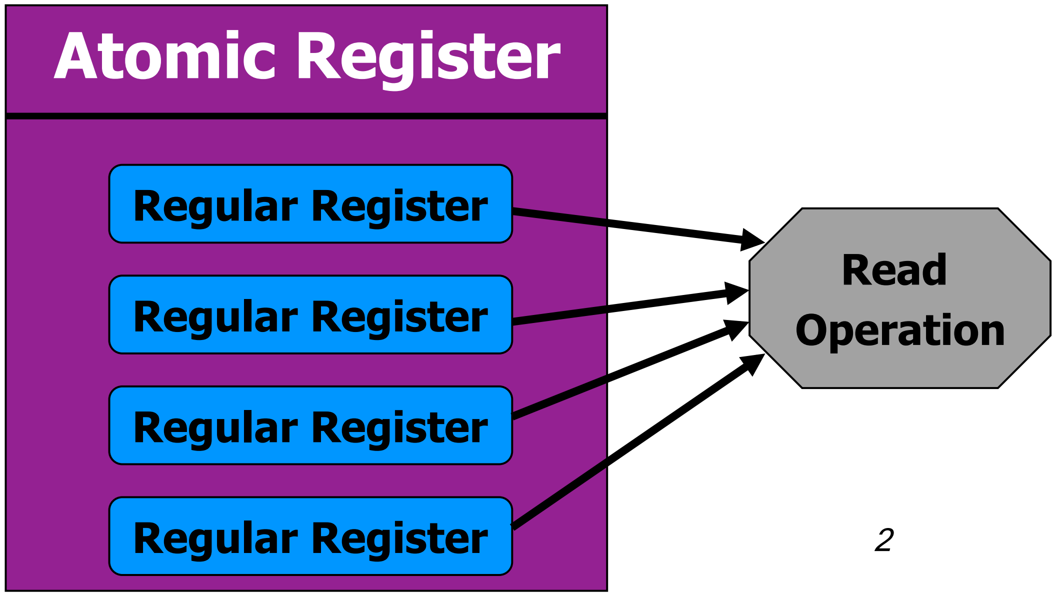
R. Guerraoui

Distributed Programming Laboratory

When readers need to write?

Register Implementation (readers don't write):

```
Read()
1: x := read(…)
2: y := read(…)
3: return(x)
```



SRSW *regular* ⇒ SRSW *atomic*

- *Reg* : SRSW ***register***
- *t, x* : local variables

Read()

```
1. (t′, x′) = Reg.read()
2. if (t′ > t) then t:=t′ ; x:=x′
3. return(x)
```

Write(v)

```
1. t := t+1
2. Reg.write(v,t);
```

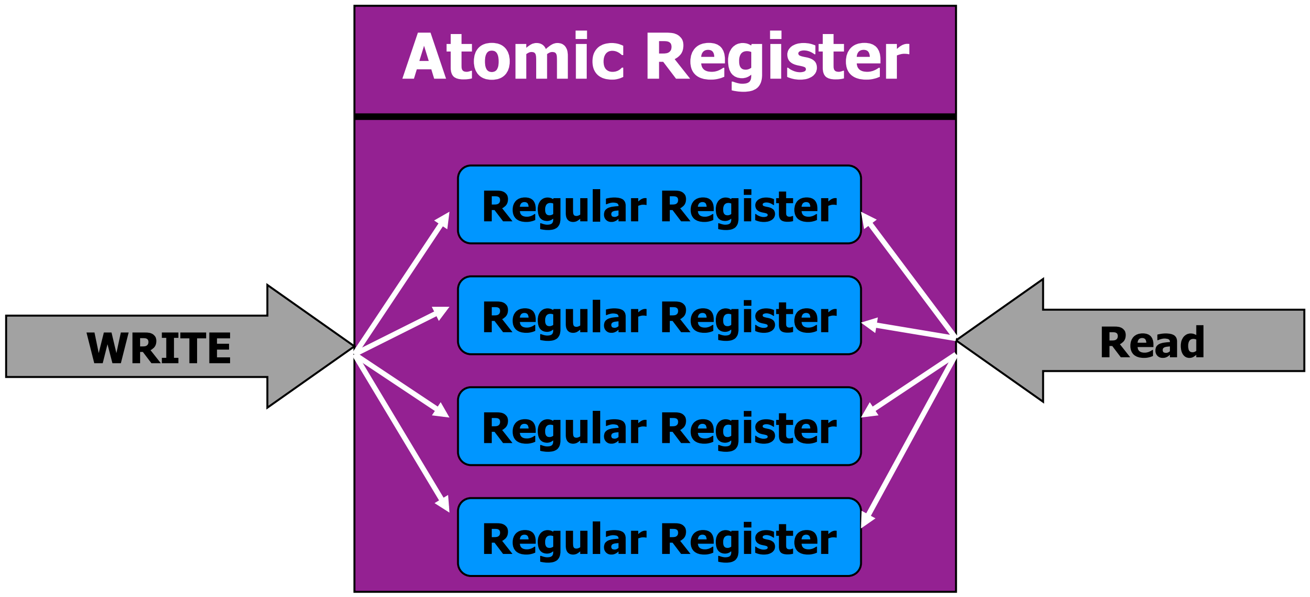
SRSW *regular* ⇒ SRSW *atomic*

- Not for multiple readers…
- Not without timestamps…
  - variable t representing logical time
- ***What is behind these limitations?***

Bound on SWSR atomic register implementations

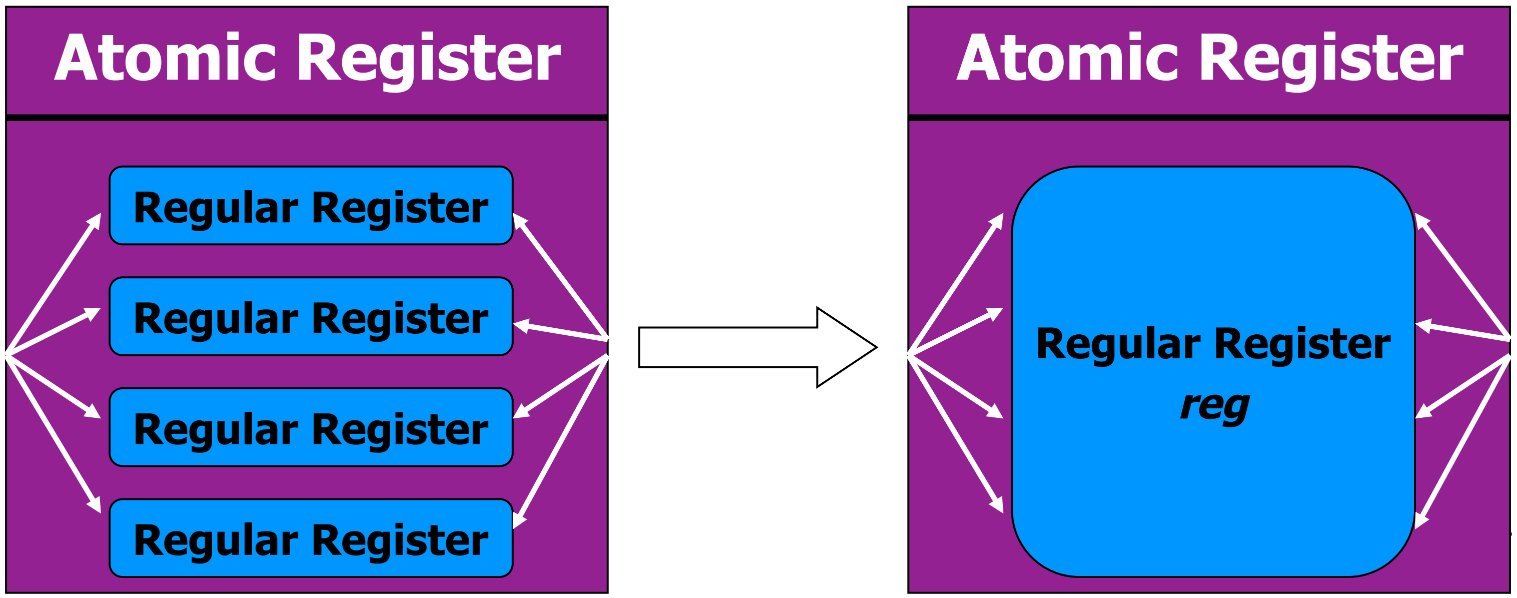
- Theorem 1:

  There is no *wait-free* algorithm that:
  - Implements a <u>SWSR atomic</u> register.
  - Uses a *finite* number of SWSR *regular* registers.
  - The registers can be written only by the writer (of the atomic register).



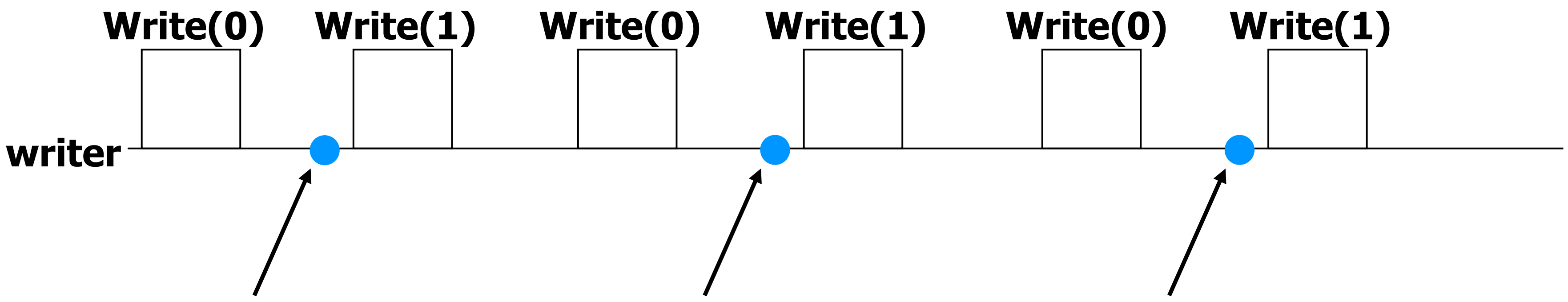
The proof

- Assume an algorithm… show contradiction
- Replace any number of **SW**SR regular registers with a single one (w.l.o.g) - ***reg***



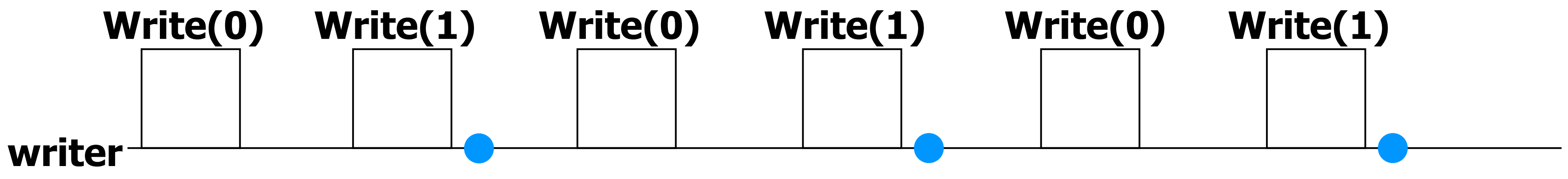
The Proof (cont'd)

- Consider an execution in which the writer alternates writing 0 and 1 infinitely many times.
  - *reg* can assume **finite** number of values.
  - There is a value v0 that appears infinitely many times in *reg* after a **Write(0)**.



The Proof (cont'd)

- Consider the subset of **Write(1)** ops starting when *reg* is in state v0.
    - *reg* can assume **finite** number of values after **Write(1)**.
    - There is a value vn that appears infinitely many times in *reg* after a **Write(1)**.



- The state of *reg* changes infinitely many times from v0 to vn when **Write(1)** occurs.

The Proof (cont'd)

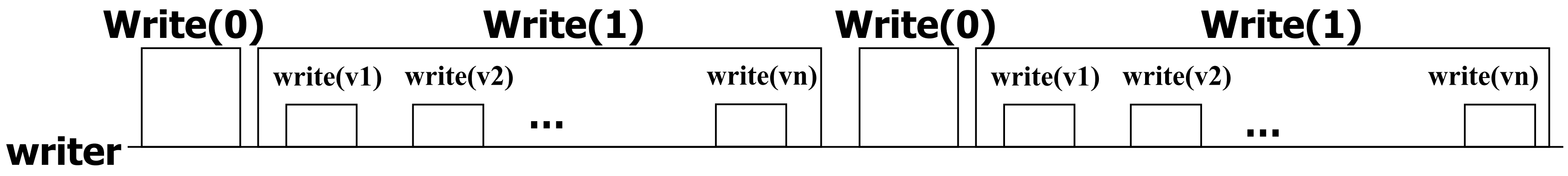
Similarily (generalization):

There must exist values v_0, v_1, … v_n, such that

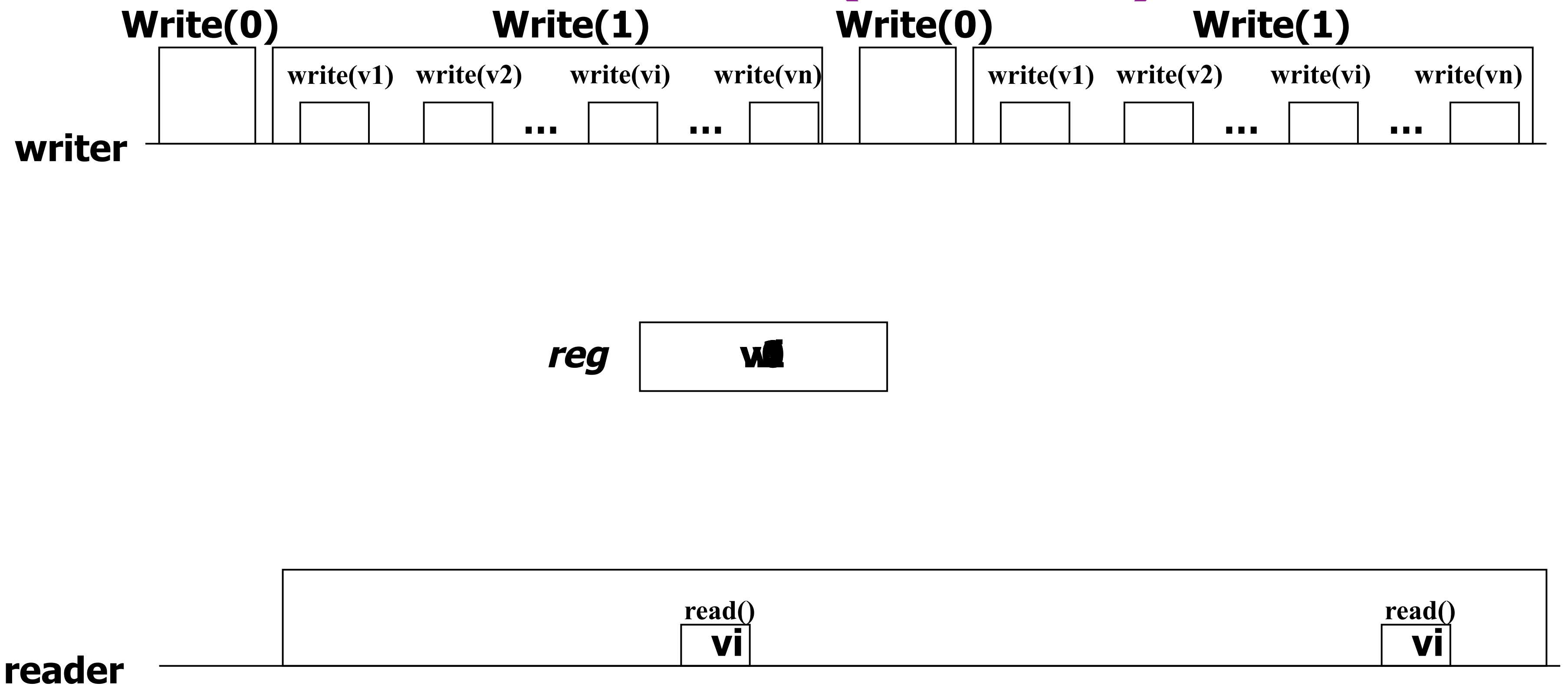
a) v_0 is the value of *reg* before infinite **Write(1)** ops.

b) v_n is the value of *reg* after infinite **Write(1)** ops.

c) $\forall i<n$: *reg* changes infinitely many times from v_i to v_{i+1} during infinite **Write(1)** ops.

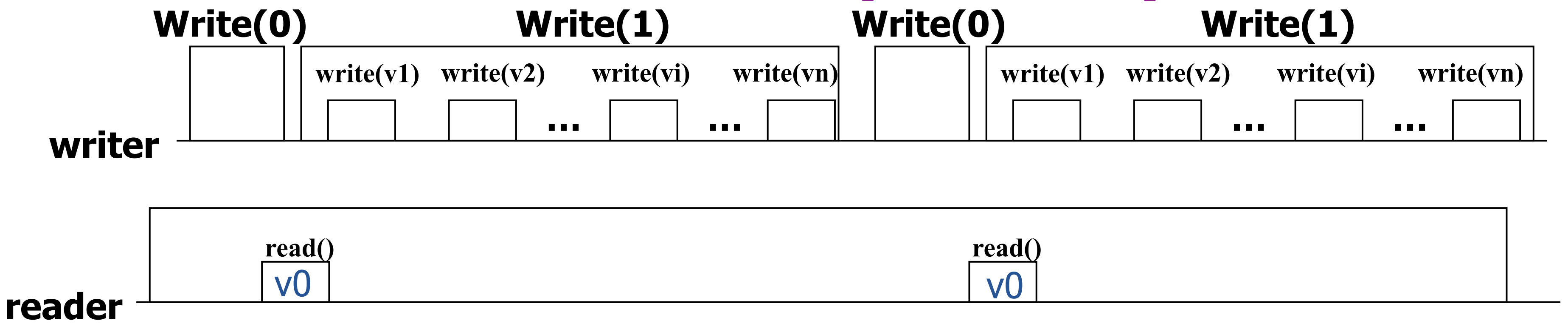


The Proof (cont'd)

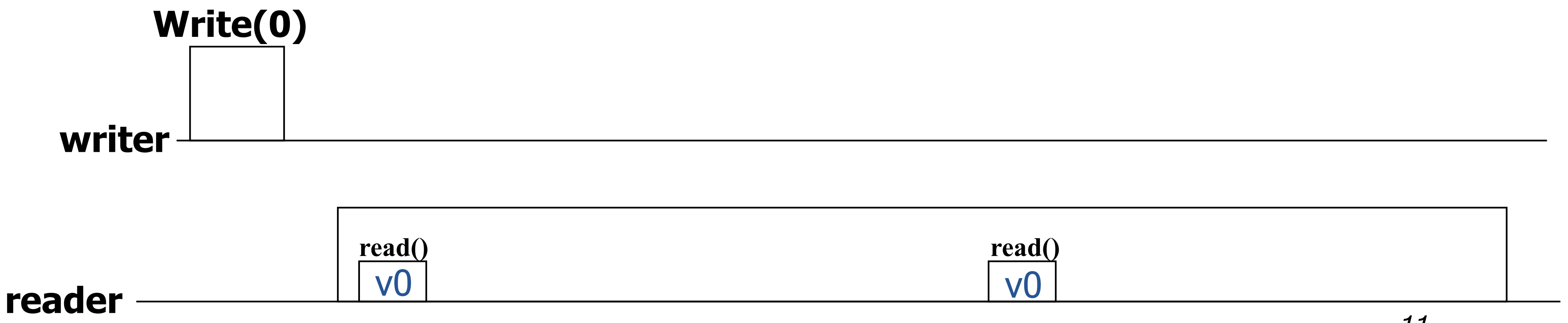


Execution 1

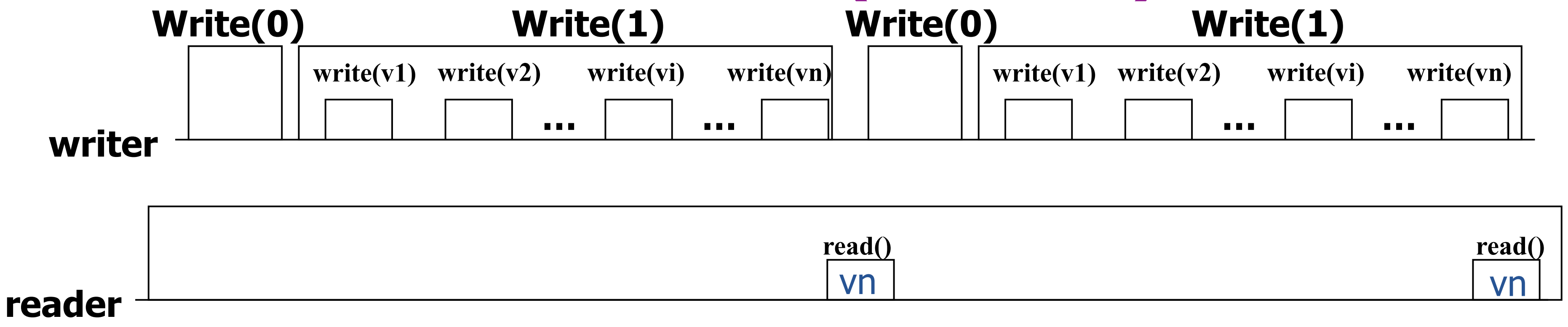
The Proof (cont'd)



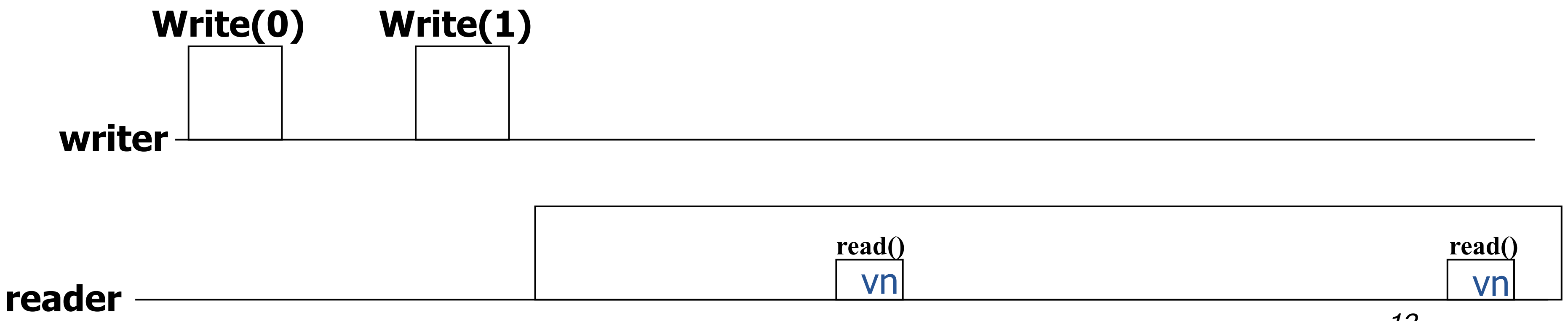
Read() returns 0



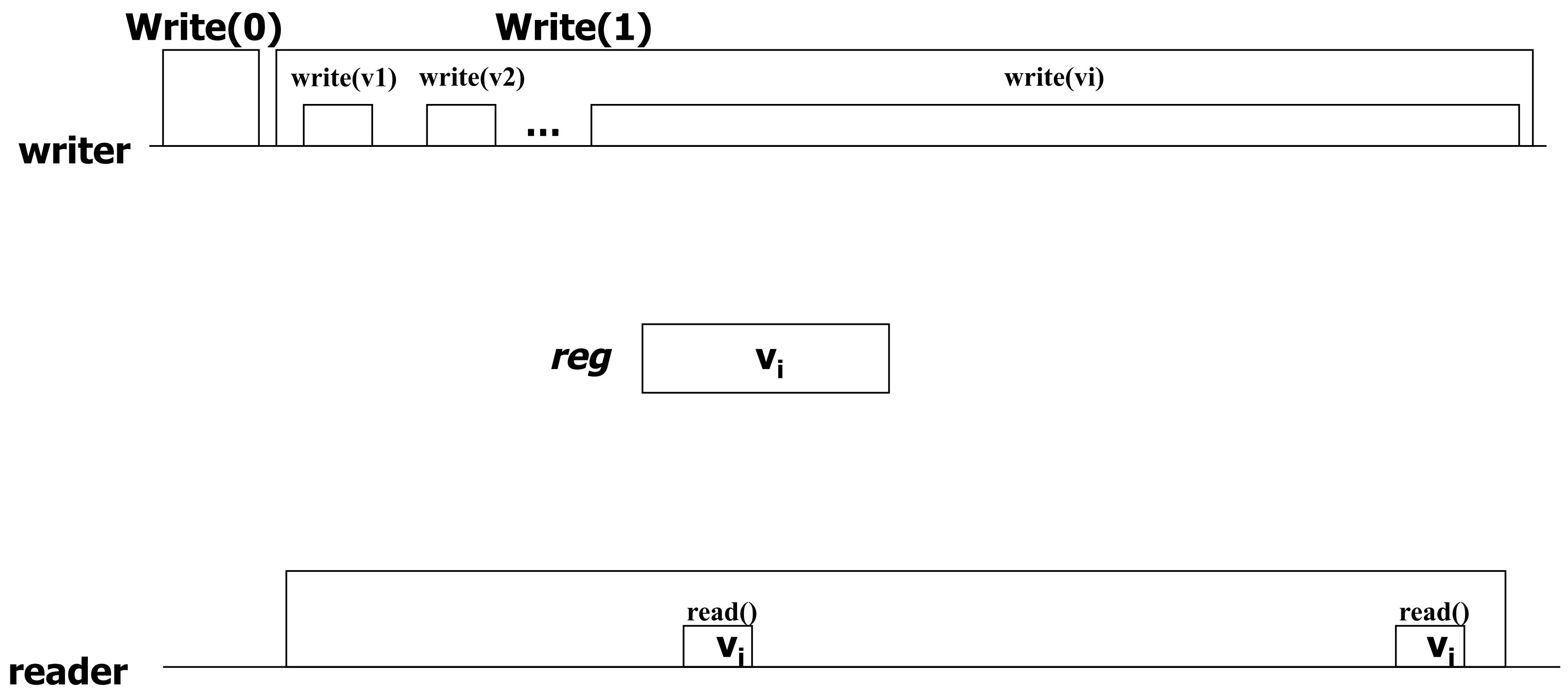
The Proof (cont'd)



Read() returns 1



The Proof (cont'd)



Execution 2

The Proof (cont'd)

- There is a minimum i ($0<i<=n$) such that:

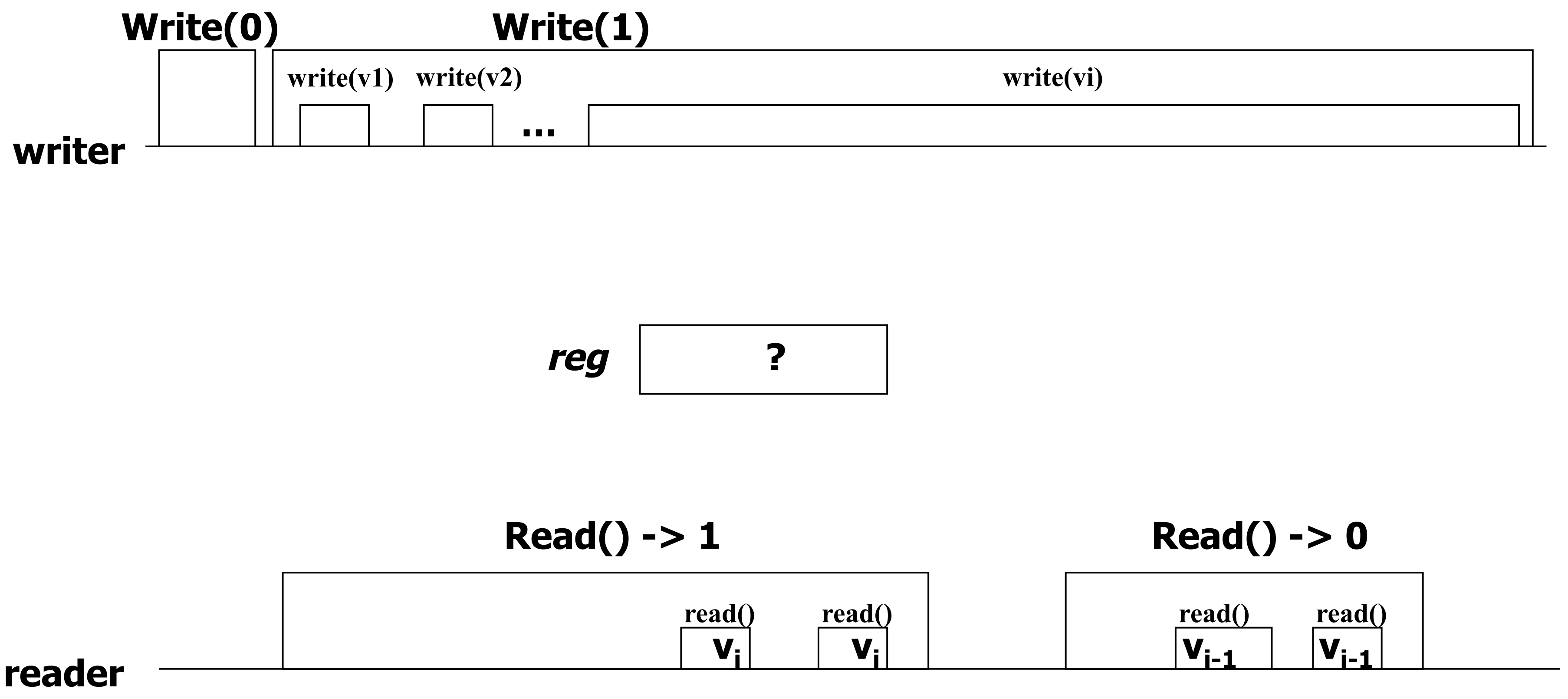
  If the reader always reads v_i, then:

  - The reader returns 1.

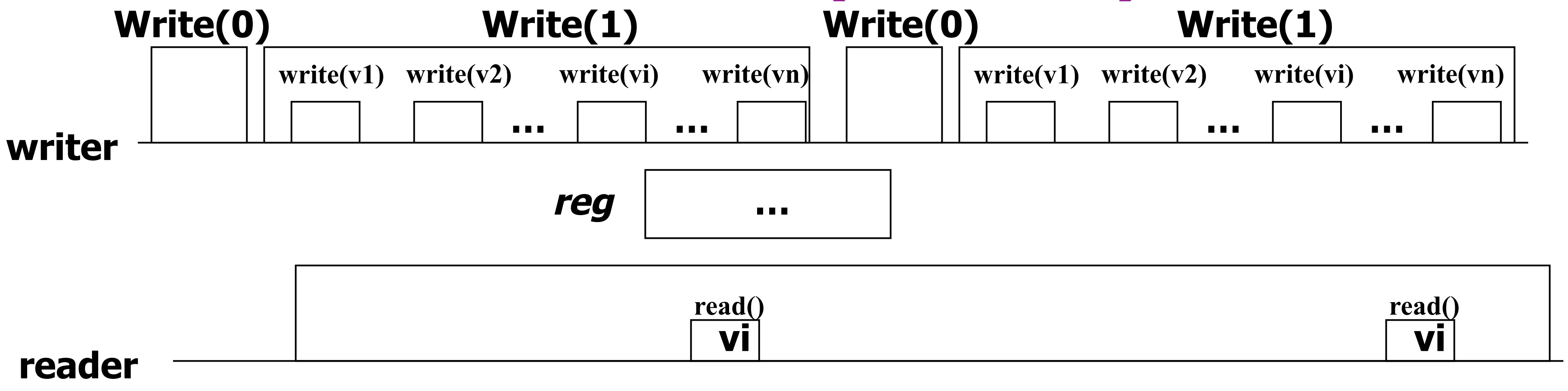
  If the reader always reads v_{i-1}, then:

  - The reader returns 0.

The Proof (end)



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.

Summary

- The reader needs to write in order to reduce the ***space complexity:***
    - Reduce space from *unbounded* to *bounded*.
    - Key requirement: reader–writer communication

- The (bounded) algorithm will come a bit later

Single to *Multi* Reader:
SRSW atomic to MRSW atomic

Write(v)

```
1. t1 := t1+1
2. for j = 1 to N
3.       WReg.write(v,t1)
```

Single to *Multi* Reader:
SRSW atomic to MRSW atomic

Read()

```
1. for j = 1 to N do
2.         (t[j], x[j]) := RReg[i, j].read()
3. (t[0], x[0]) = WReg[i].read()
4. (t, x) := highest(t[..], x[..])
5. for j = 1 to N do
6.         RReg[j, i].write(t, x)
7. return(x)
```

Single to *Multi* Reader:
SRSW atomic to MRSW atomic

- 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)

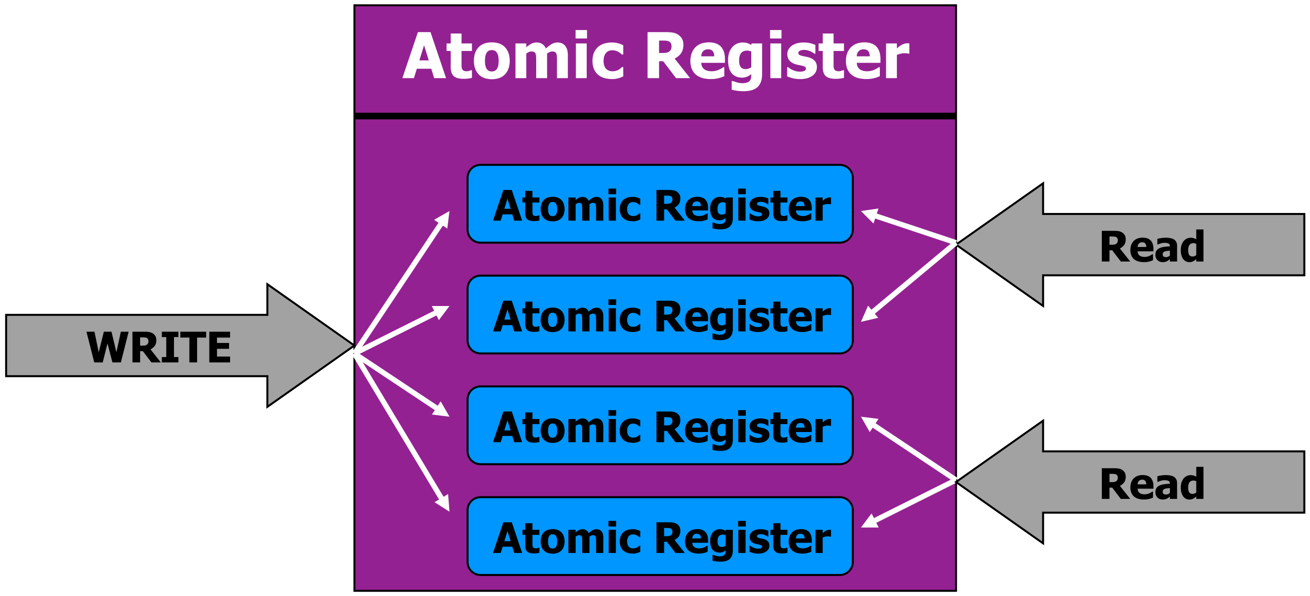
Bound on SWMR atomic register implementations

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

Bound on SWMR atomic register implementations

- Theorem 2:

  There is no *wait-free* algorithm that:
  - Implements a SWMR atomic register.
  - Uses *any* number of SWSR *atomic* registers.
  - The registers can be written only by the writer (of the atomic register).



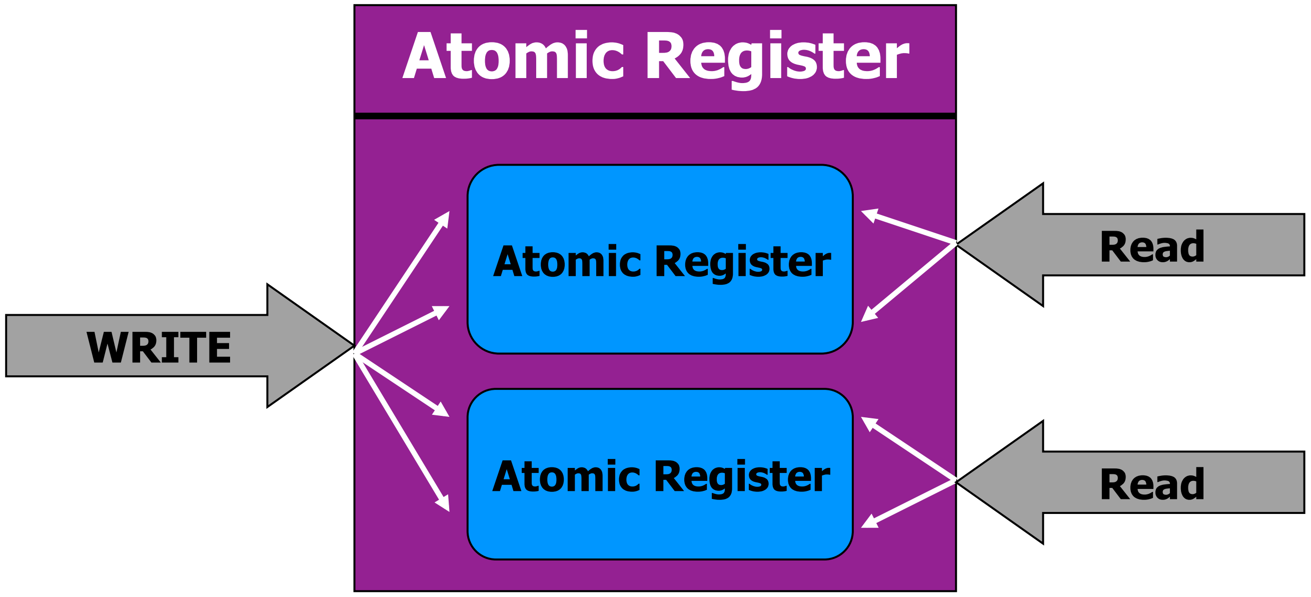
The proof

- 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.

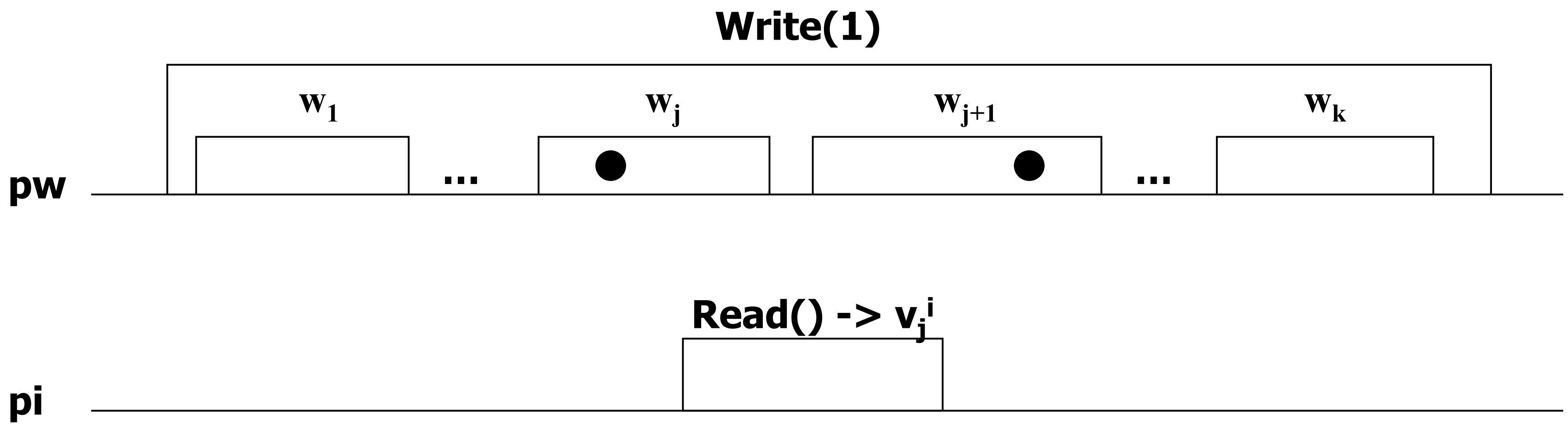
The proof

- We replace all atomic registers read by p1 by a single one – *reg1*.
- We replace all atomic registers read by p2 by a single one – *reg2*



The proof (cont'd)

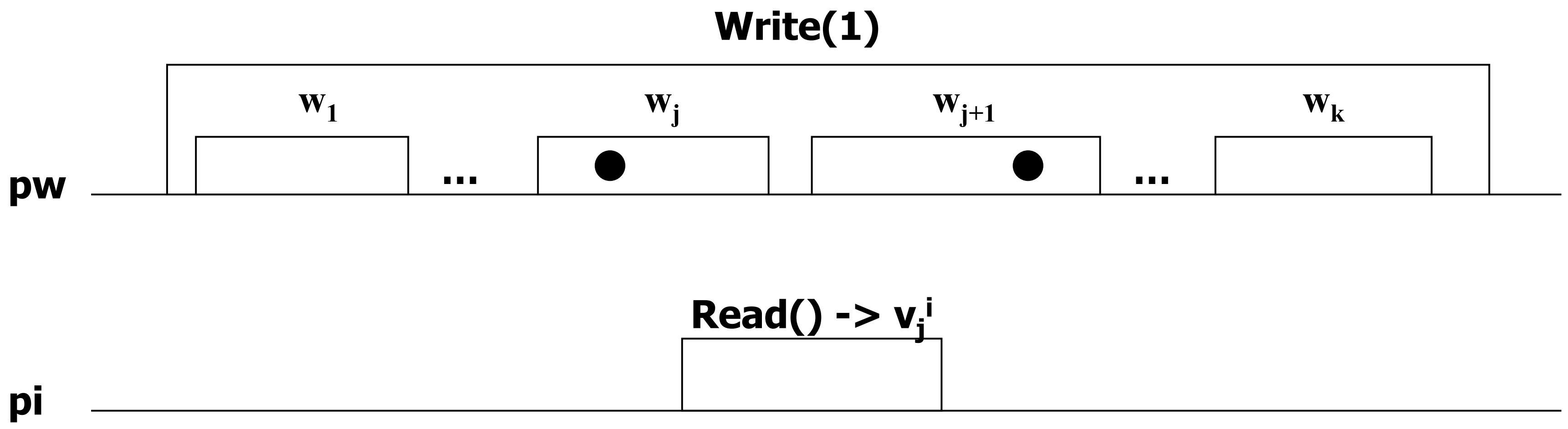
- Consider the first write of 1 into *reg**
- This consists of a number of low-level writes w1 to wk into reg1/reg2



The proof (cont'd)

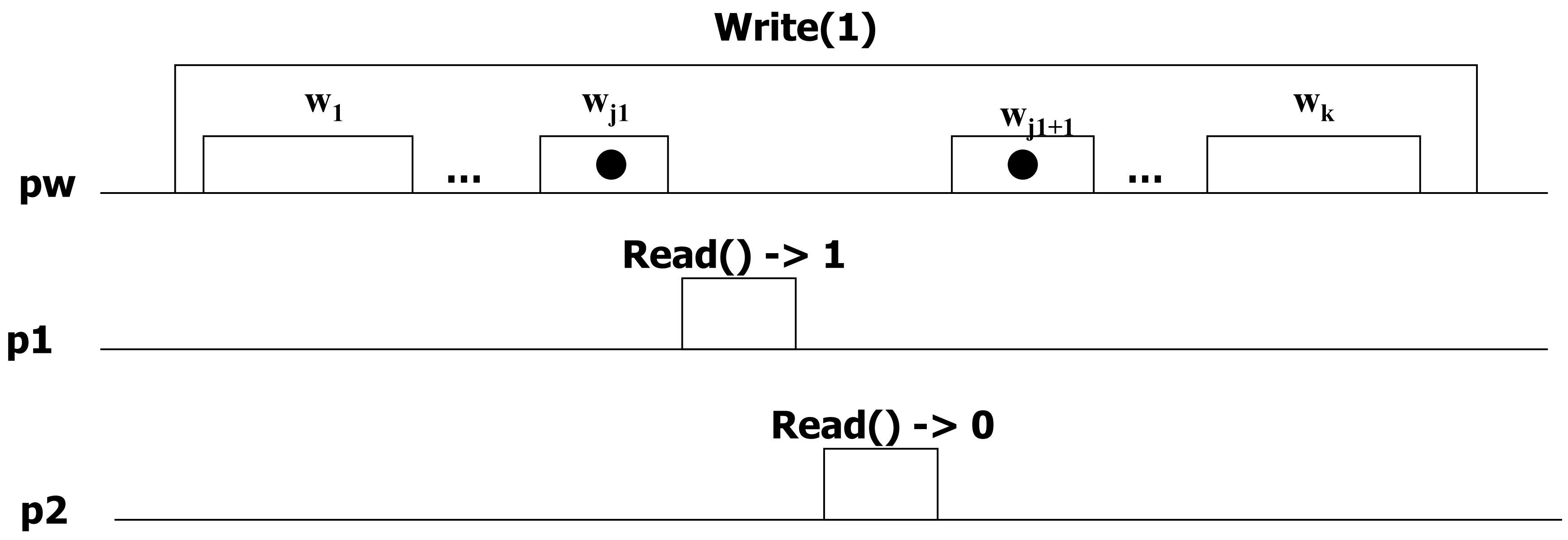
- $\forall i \in \{1,2\}, \exists\, j_i$: $1 \le j_i \le k$:

  $\forall j < j_i$: $v_j^i = 0$ 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*



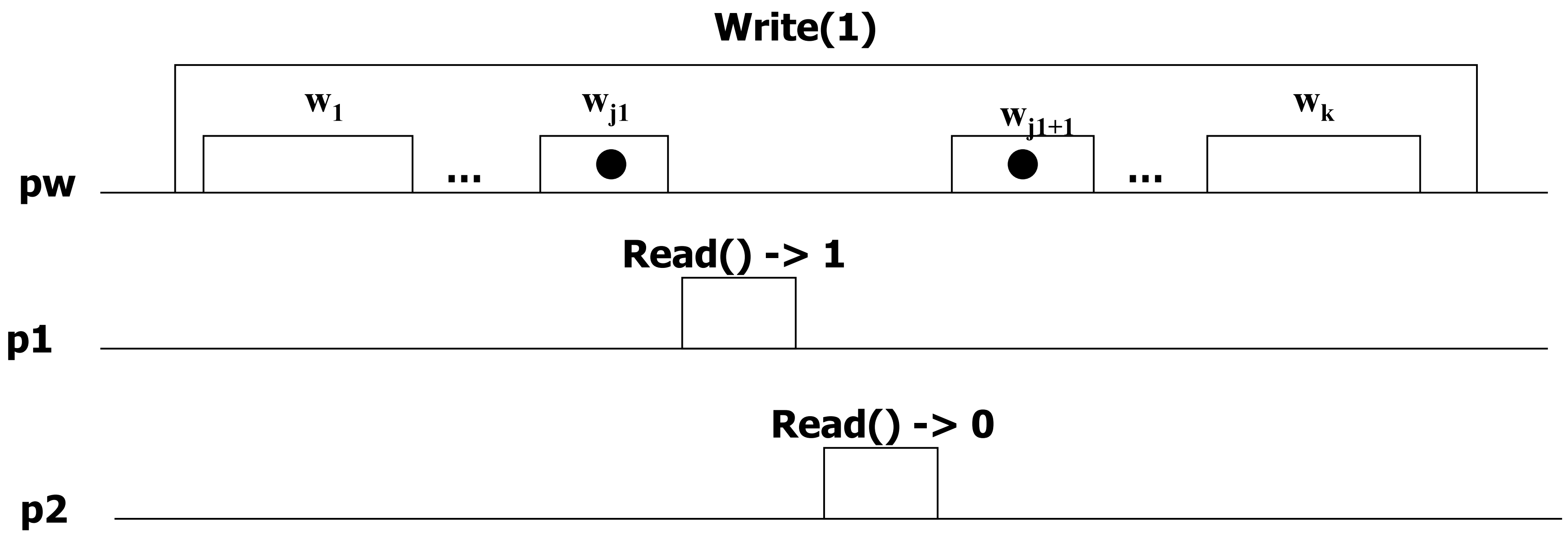
The proof (end)

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



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).

Summary

- The readers *need* to write in implementations of:
  - *multi-reader*
  - *wait-free*
  - *atomic*

  (out of weaker base objects)
- Even when the available space is unbounded
- Same idea:
  - Implementing SWMR atomic from SWMR regular
- We can implement SWMR regular from SWSR atomic

From safe to atomic: one bit

Wait-free implementation one *SWSR atomic bit*

- Brute force (the reader does not write):
  1. SWSR *safe* to SWSR *regular* bit
     - Simple
  2. SWSR regular *bit* to SWMR *multivalued*
     - O(N) in space and time
  3. SWMR *regular* to SWSR *atomic*
     - Timestamps (unbounded space)

From safe to atomic: one bit

Wait-free implementation of one *SWSR atomic bit*

- Something different:

  The reader should write!

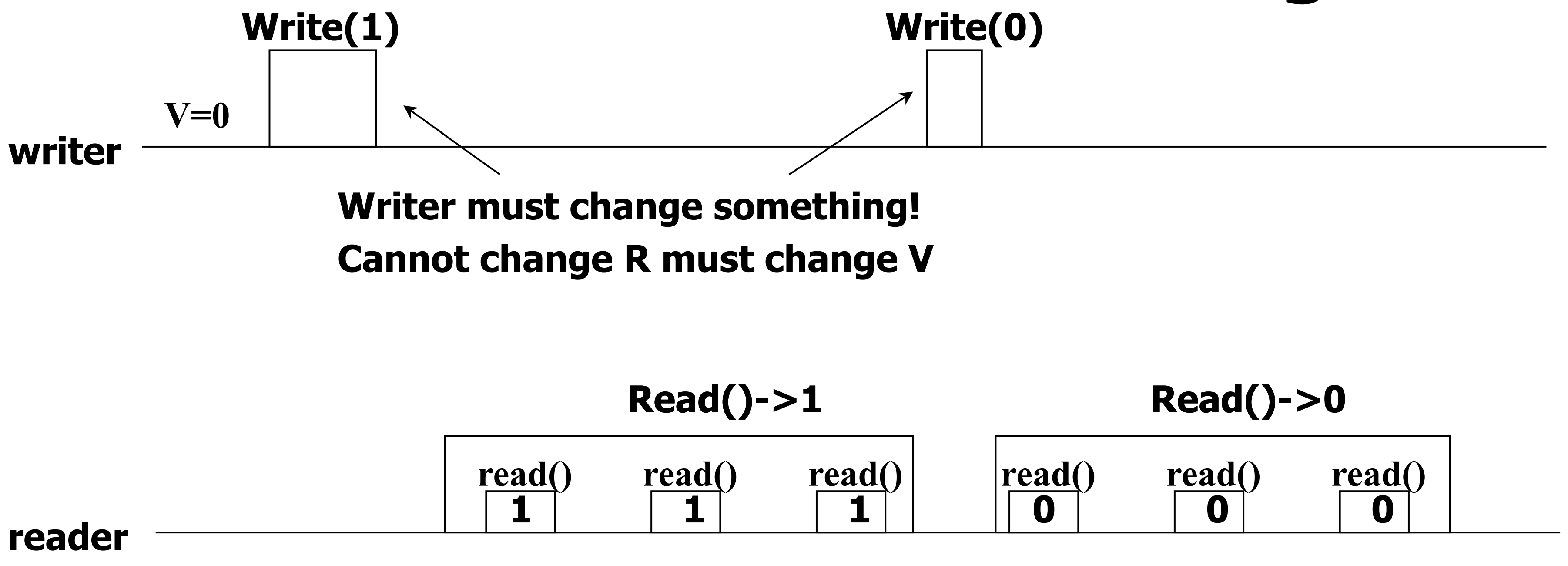
- Aim for O(1) complexity in space and in time

How many safe bits?

- A single one will not be enough (Theorem 1)
  - We need at least:
    - one for *writer* to write value
    - one for reader will write

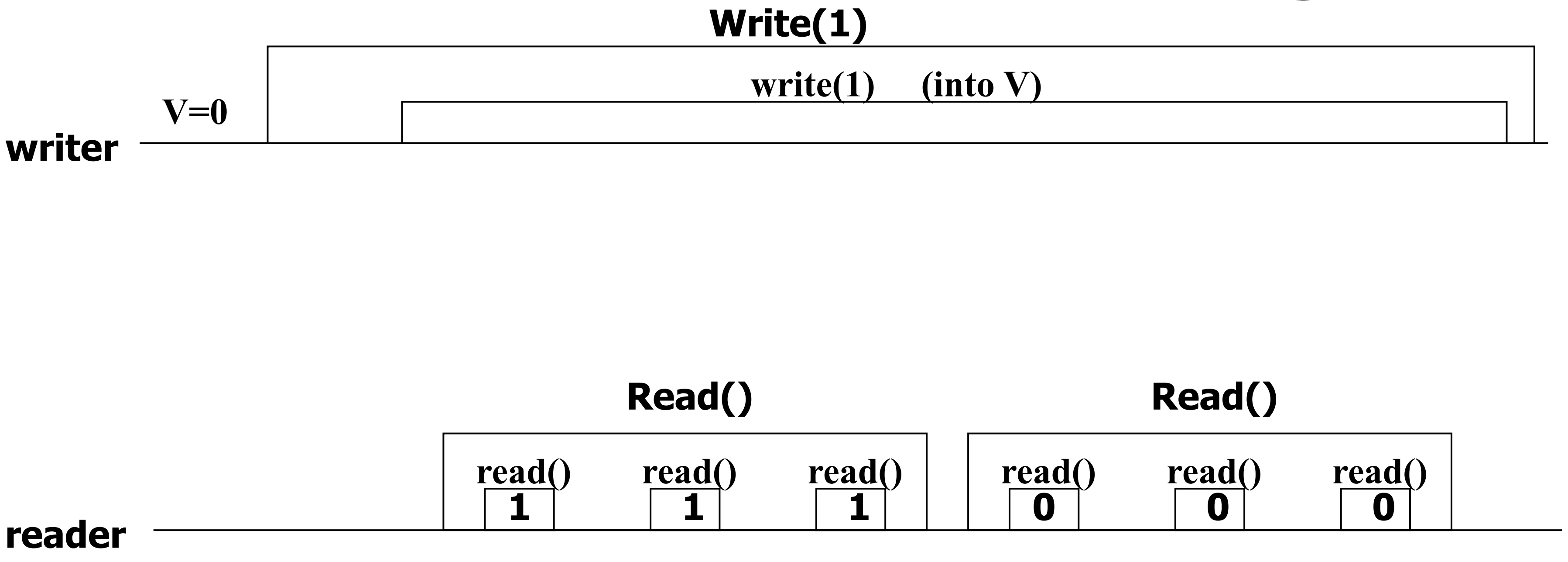
- Can we do it with only 2 SWSR safe bits?
  - No…
- Assume two bits
  - V, written by the writer and read by the reader
  - R, written by the reader and read by the writer

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 executes:
  - <u>if</u> W=R <u>then</u> { … }
- We mean:

  1) r :=read(R)

  2) <u>if</u> (W=r) <u>then</u> …
- r is a local variable
- A copy of W is stored localy

Tromp's algorithm

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

Tromp’s algorithm

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

Tromp's algorithm

```
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

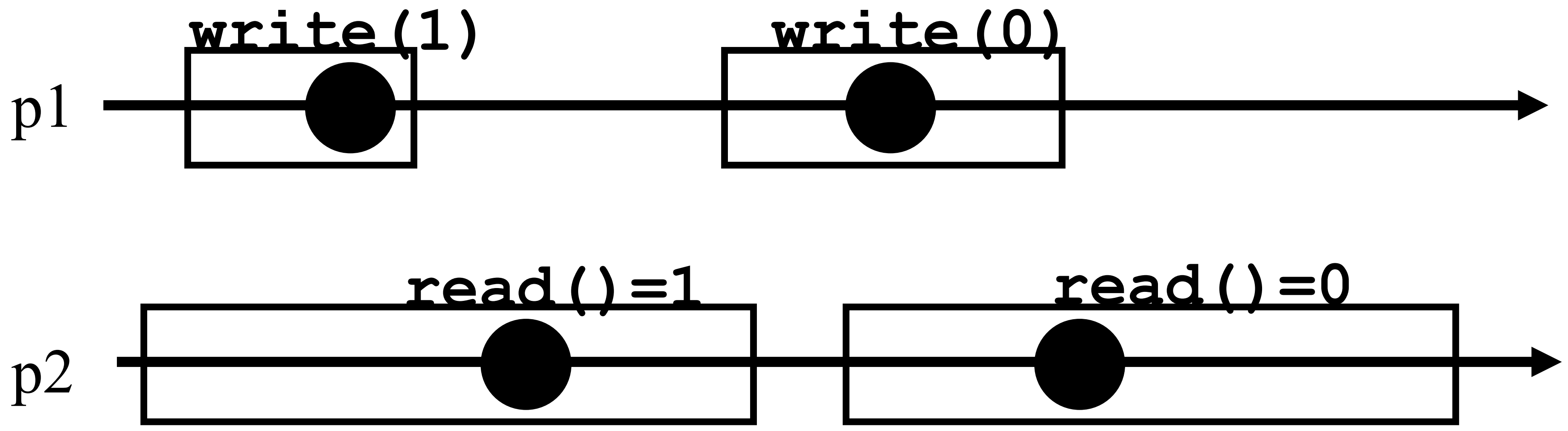
Correctness

- Liveness – straigthforward
- Safety – a bit more difficult

Atomicity (review)

For every execution:

- We can assign a *serialization point* for each operation.
- Each operation takes place instantaneously at its serialization point.



Atomicity (conditions)

For every execution:

There exists a *partial order* of operations such that:

1. All **Write** operations are ordered.
2. Each **Read** operation is ordered with respect to all **write** ops.
3. Each **Read** operation returns the value of the immediately preceding **Write** operation.
4. If op1 precedes op2, then **not**(op2 < op1) in the ordering.

Atomicity (conditions)

For every execution:

There exists a *partial order* of operations such that:

1. All **Write** operations are ordered.
2. Each **Read** operation is ordered with respect to all **write** ops.
3. Each **Read** operation returns the value of the immediately preceding **Write** operation.
4. If op1 precedes op2, then **not**(op2 < op1) in the ordering.

Define ordering:

1. Writes are ordered as they are issued.
2. Reads:
   - Find last “Read(V)” that precedes return for **Read**.
   - Find “Write(V)” that wrote that value.
   - **Write** that contains “Write(V)” ordered before **Read**.

Atomicity (conditions)

For every execution:

There exists a *partial order* of operations such that:

1. All **Write** operations are ordered.
2. Each **Read** operation is ordered with respect to all **Write** ops.
3. Each **Read** operation returns the value of the immediately preceding **Write** operation.
4. If op1 precedes op2, then **not**(op2 < op1) in the ordering.

Define ordering:

1. Writes are (trivially) ordered.
2. Reads:
   - Find last "Read(V)" that precedes return for **Read**.
   - Find "Write(V)" that wrote that value.
   - **Write** that contains "Write(V)" ordered before **Read**.

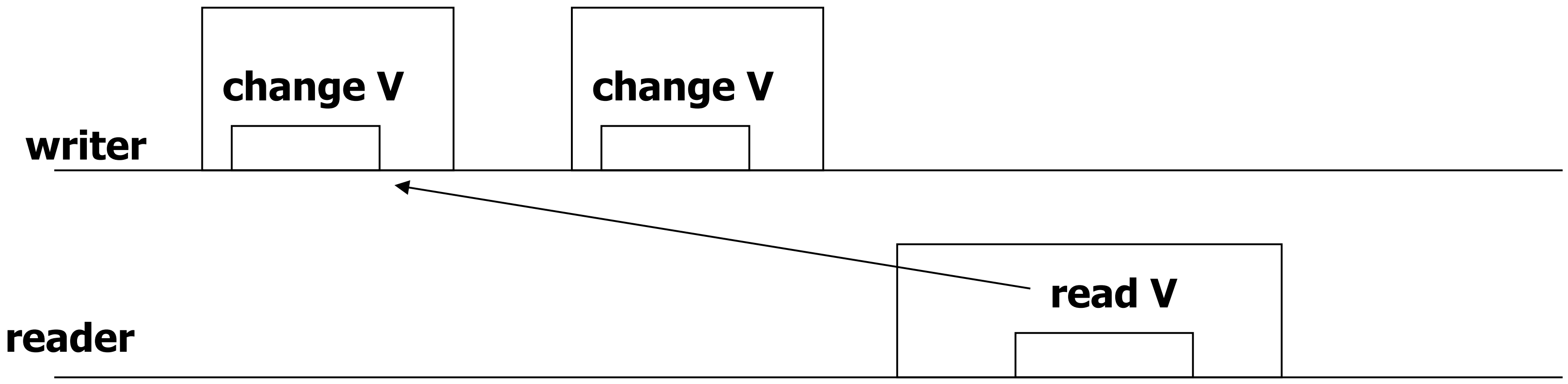
Correctness 1 (Safety)

- Each **Read** operation returns the value of the immediately preceding **Write** operation.

Correctness 1

- Each **Read** operation returns the value of the immediately preceding **Write** operation.

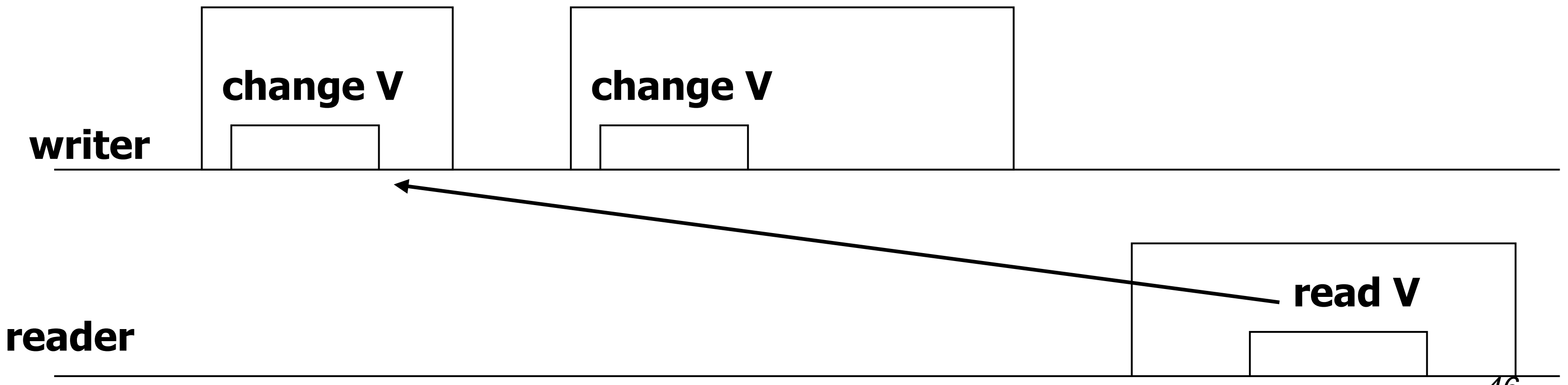
  - Assume for the sake of contradiction…



Correctness 1

- Case 1: **Read** op returns on line 5 or 7
  - Returns v or x read *during* **Read** op.
  - V *acts like* a regular register.
  - read(V) *can not* return old value.

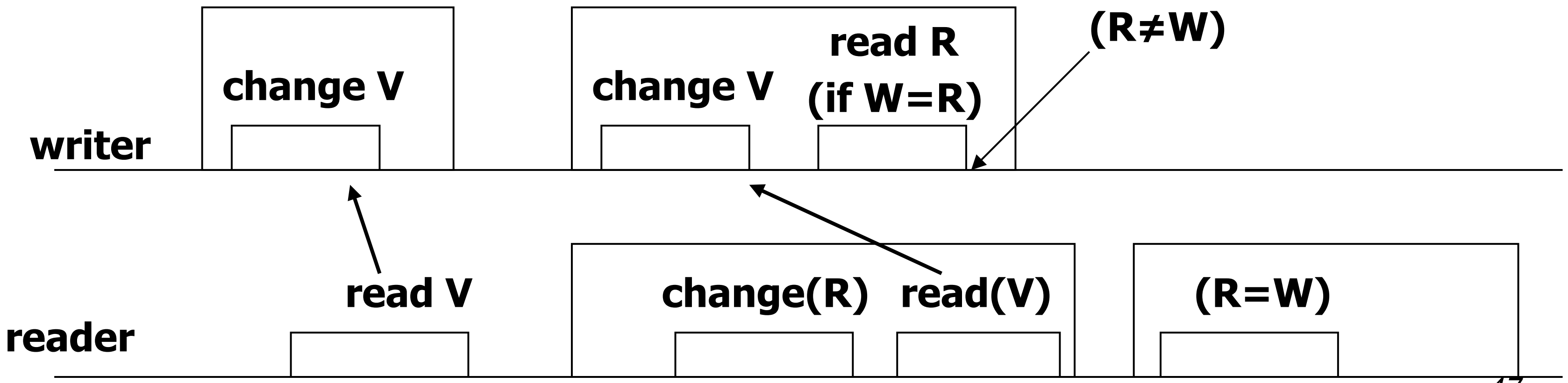
  Contradiction…



Correctness 1

- Case 2: **Read** op returns on line 1.
  - Returns v from previous **Read** op: (R=W)
  - But, after write operation, (R≠W).
  - So there must have been a previous **Read**.
  - And that Read must have "Read(V)"

Contradiction…



Correctness 2 (Regularity)

- A **Read** returns the value of the concurrent **Write** or a previous **Write**.

- The writer is only allowed to access the shared memory to change the value of the implemented register. If a read operation is concurrent with a write that changes the value, it is allowed to return both 0 and 1

Correctness 3 (Atomicity)

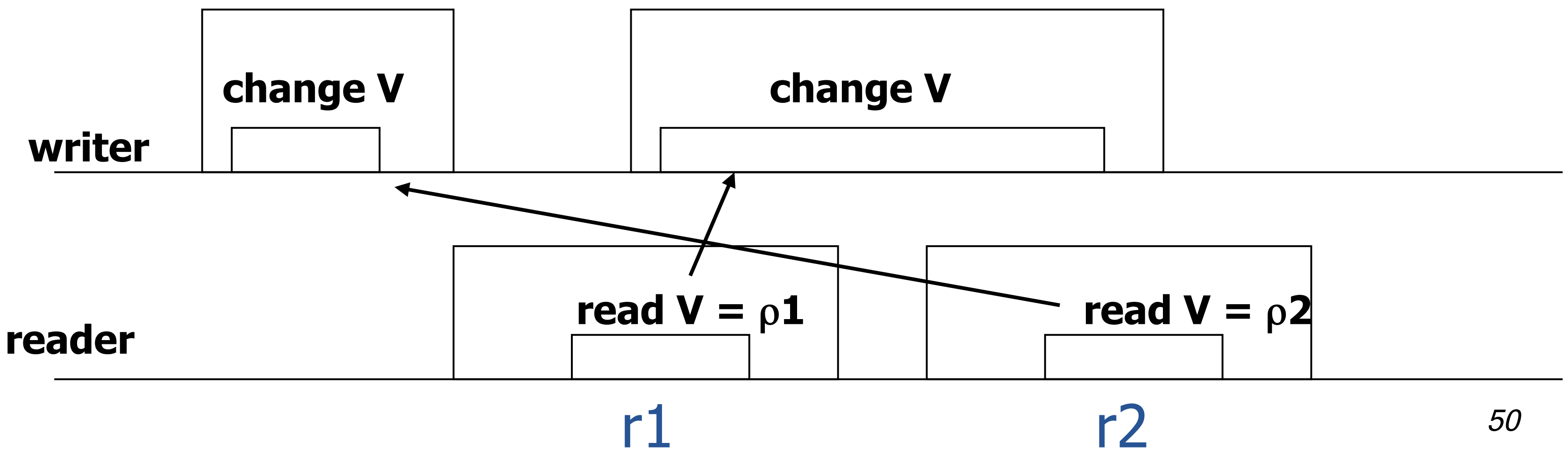
- **Lemma:** If Read r1 precedes r2 and ri returns the value written by the Write vi (i=1..2), then

  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

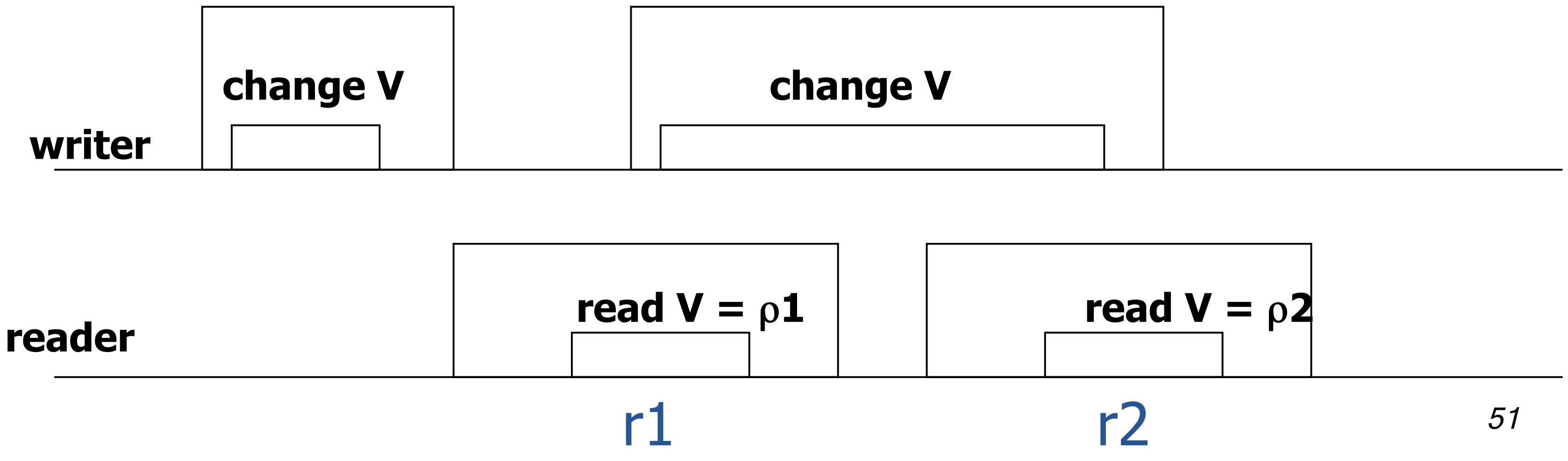
Correctness 3

- If **Read** r1 precedes **Read** r2, then not(r2< r1).
  - Assume for the sake of contradiction…



Correctness 3

- Let ρi be the read(V) returned by ri (i=1..2).
- **Claim 1:** ρ1 precedes ρ2
  - ρ1 ∈ r1 or some **Read** that precedes r1.
  - If ρ2∈r2, then **Claim 1** is trivial (since r1→r2).

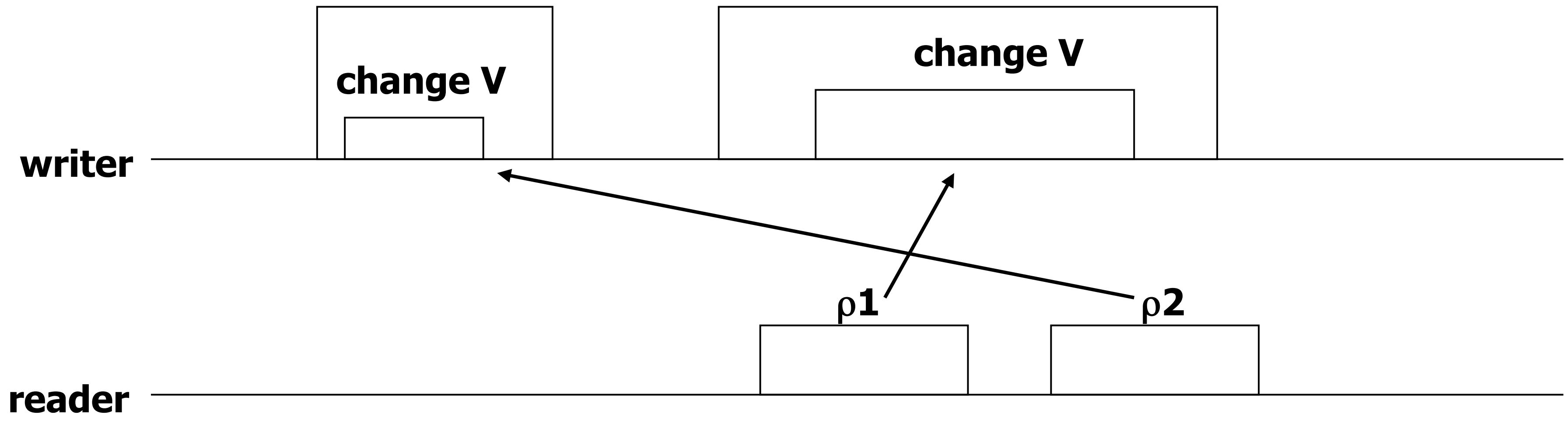


Correctness 3

- Let ρi be the read(V) returned by ri (i=1..2).
- **Claim 1:** $\rho 1$ precedes $\rho 2$
  - $\rho 1 \in r1$ or some **Read** that precedes r1.
  - If $\rho 2 \notin r2$, then r2 returns in line 1:
    - Observe that $\rho 1 \neq \rho 2$.
    - If $\rho 2 \rightarrow r1$ then r1 does not change v
      - r1 returns in line 1 and $\rho 1 = \rho 2$
    - If $\rho 2 \in r1$ then:
      - $\rho 1$ is a read(V) in line 2 or 4 of r1 or earlier.
      - $\rho 2$ is a read(V) in line 4 or 6 of r1 or later.

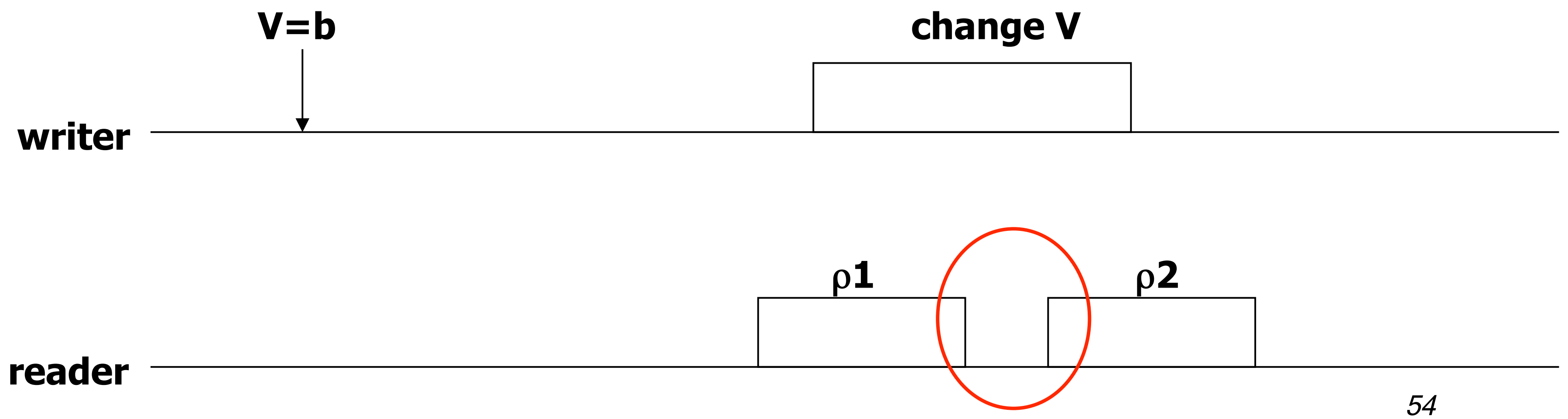
Correctness 3

- **Claim 2:** There is a change(V) operation by writer that started before ρ1 finished and finished after ρ2 started



Correctness 3

- **Claim 3:** Every "Read(W)" operation by the reader between ρ1 and ρ2 returns the same value.
- **Proof:** The writer is busy changing V (Claim 2).



Correctness 3

- 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 (**))!

Correctness 3

- 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≠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, (*)

Correctness 3

- There are 3 exhaustive cases
- (iii) $\rho1$ 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 $\rho2$ (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) $\Rightarrow$ $\rho1=\rho2$ – a contradiction w. Claim 1, (*)

Tromp's algorithm

```
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

$W \neq R \Leftrightarrow$ there is a new value

$W = R \Leftrightarrow$ no new values

Tromp's algorithm

```
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

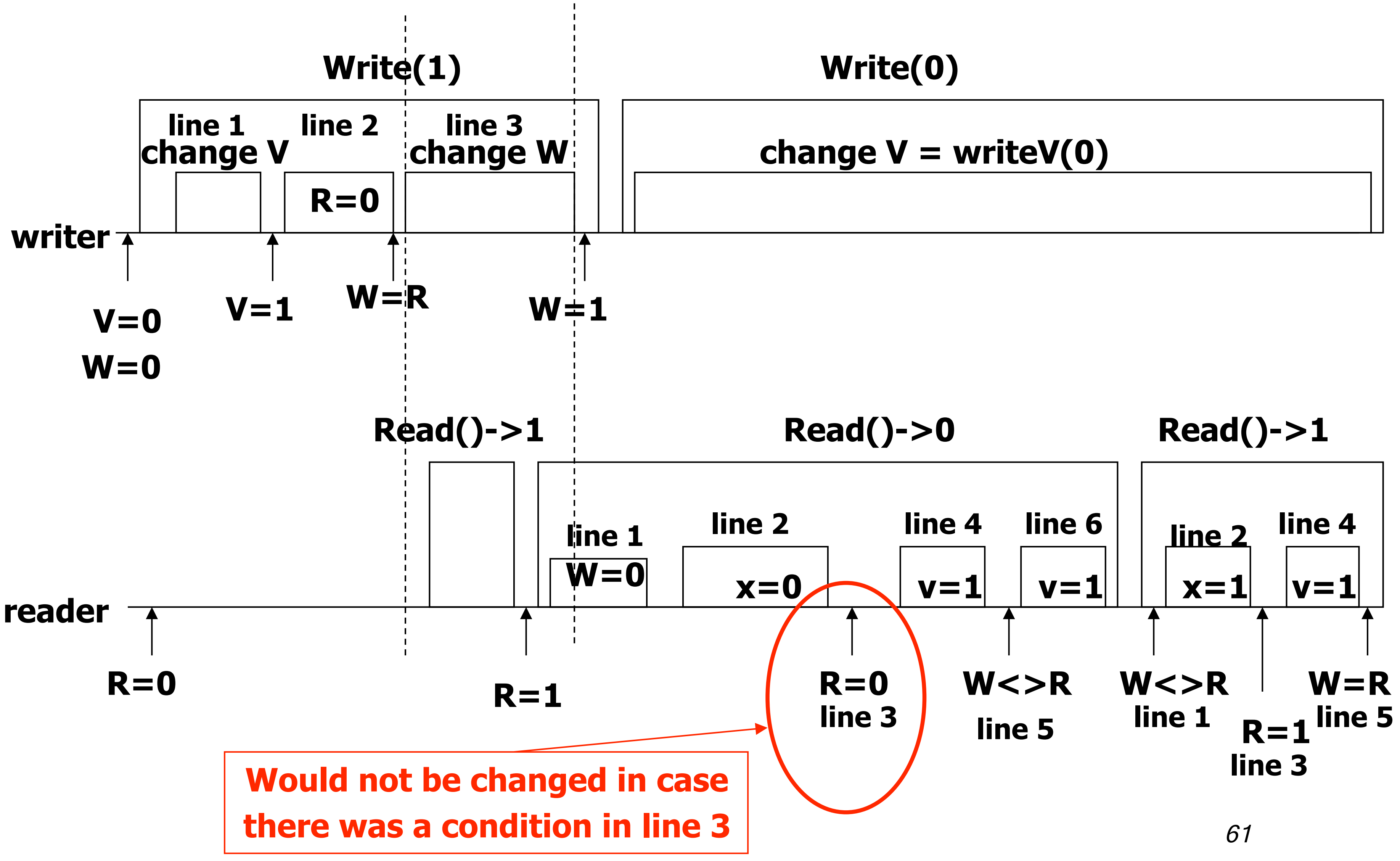
W≠R ⇔ there is a new value

W=R ⇔ no new values

Condition in line 3?

- 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 (**))!

Condition in line 3?



Tromp's algorithm

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

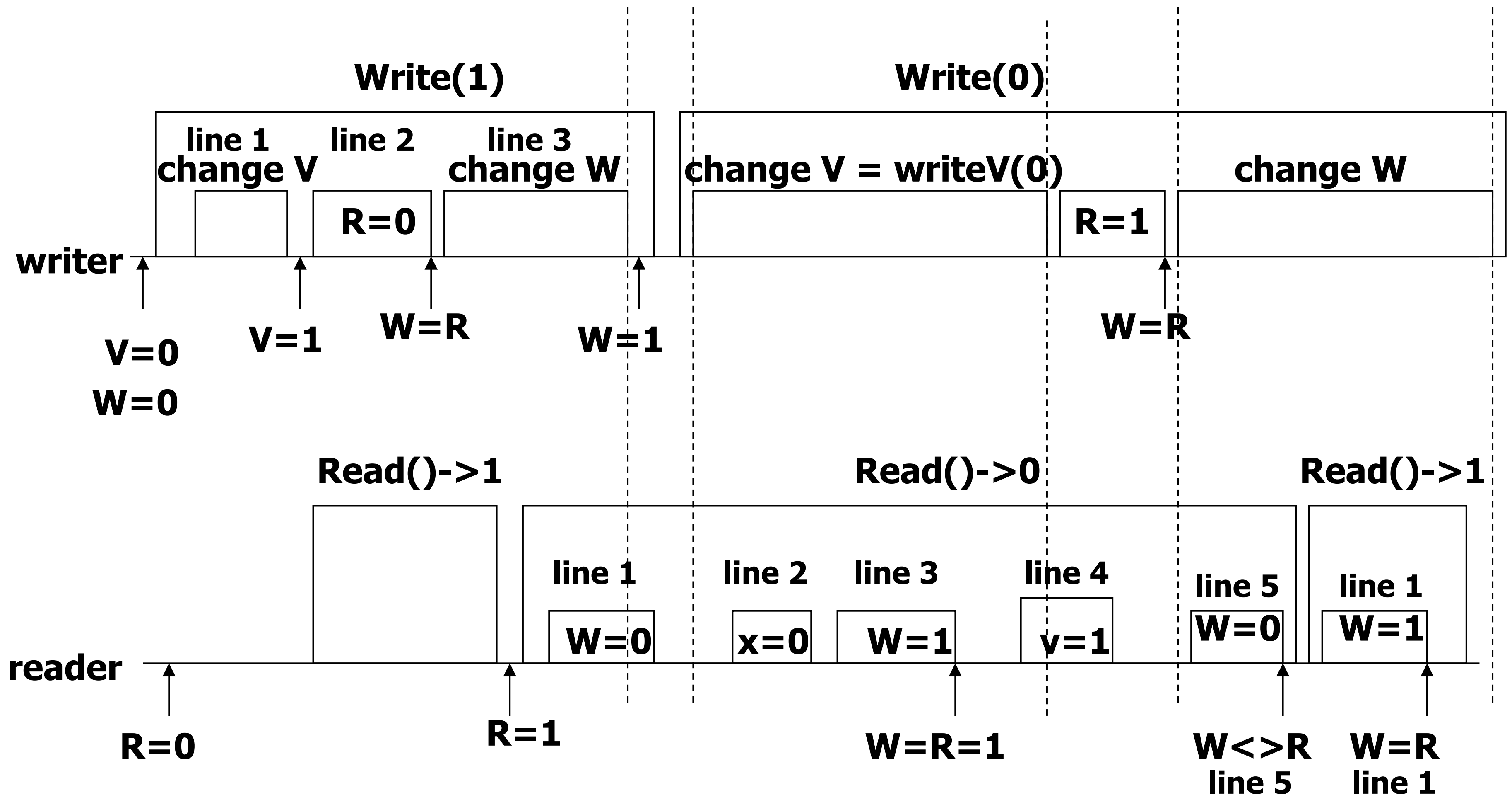
```
Read()
1: if (W=R) then return(v)
2: x := read(V)

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

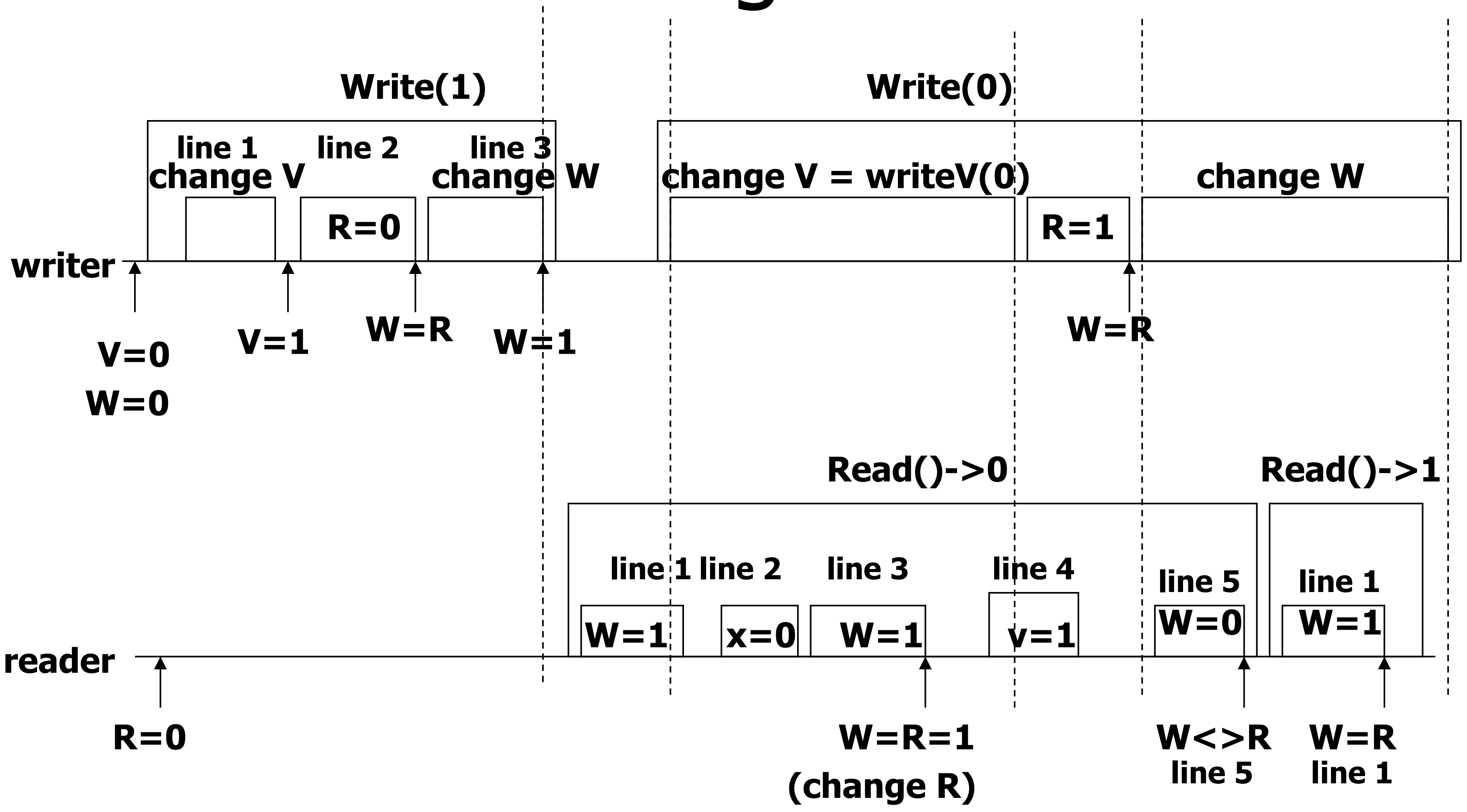
$W \neq R \Leftrightarrow$ there is a new value

$W = R \Leftrightarrow$ no new values

Removing line 6?



Removing line 6?


Write(1)
Write(0)
line 1
change V
line 2
R=0
line 3
change W
change V = writeV(0)
R=1
change W
writer
V=0
W=0
V=1
W=R
W=1
W=R
Read()->0
Read()->1
line 1
W=1
line 2
x=0
line 3
W=1
line 4
v=1
line 5
W=0
line 1
W=1
reader
R=0
W=R=1
(change R)
W<>R
line 5
W=R
line 1