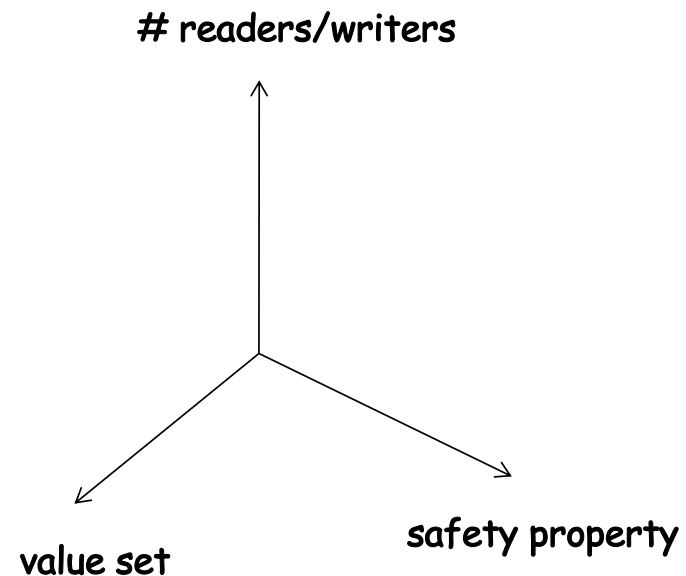# Implementing an atomic bit

SLR206, P1, 2021

The space of registers

- Nb of writers and readers: from 1W1R to NWNR

- Size of the value set: from binary to multi-valued

- Safety properties: safe, regular, atomic

All registers are (computationally) equivalent!

Transformations

From 1W1R binary safe to 1WNR multi-valued atomic

I.   From safe to regular (1W1R)

II.   From one-reader to multiple-reader (regular binary or multi-valued)

III.   From binary to multi-valued (1WNR regular)

IV.   From 1W1R regular to 1W1R atomic (unbounded)

V.   From 1W1R atomic to 1WNR atomic (unbounded)

   ✓ Can be turned into bounded using a bounded (in n) number of signaling registers

This class

- The problem: implement a binary 1W1R atomic register (atomic bit) from binary 1W1R safe ones (safe bits)
  - ✓From a few safe bits only
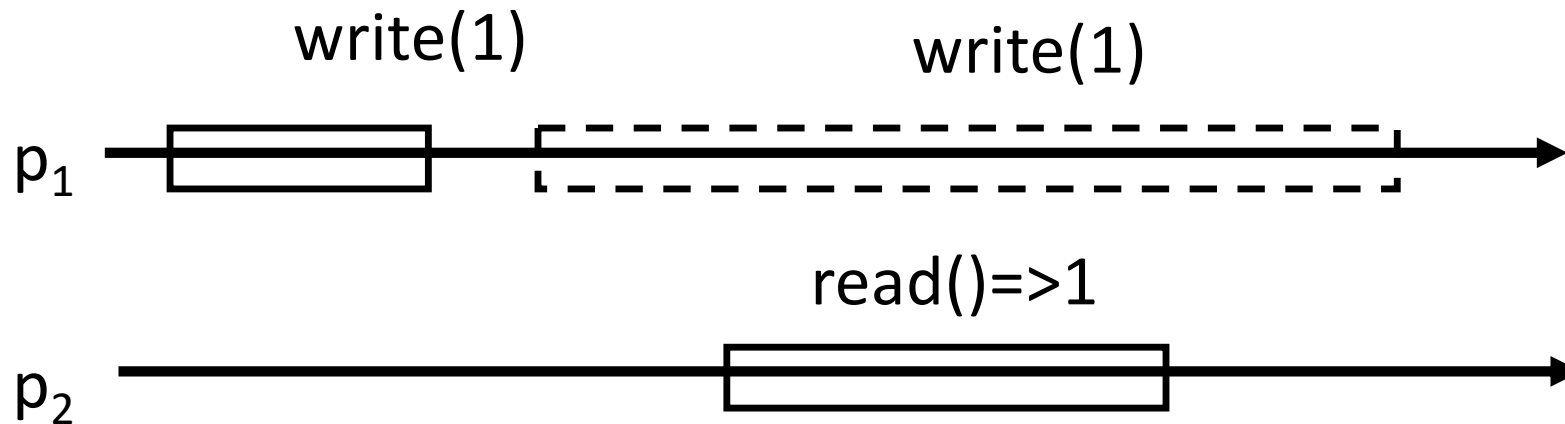 - ✓No unbounded multi-valued registers
  - ✓No ever-growing timestamps

An optimal solution

- No sequence numbers?
- Bounded number of safe bits, O(1)?
- Bounded number of base actions, O(1)?

Can we do it if the reader does not write?

Safe bit to regular bit? Easy

- the writer is allowed only to *change* the value

Can we get an atomic bit this way?

Safe to atomic: Impossible
if the reader does not write (bounded # of regular bits)

Proof sketch (by contradiction):

- Suppose only the writer executes writes on the base (regular) bits (the reader only reads the base objects).

- Every write operation W(1) is a sequence of writes actions $w_1, \ldots w_k$ on base regular bits
  - ✓ Corresponds to the sequence of shared-memory states $s_0, s_1, \ldots, s_k$ (defined for sequential runs)

write(1)   ok

p_1 ──┼──┼──┼───┼──┼──────┼──┼────────┼────→
        s_0 s_1 ... s_k
           w_1 w_2 w_k
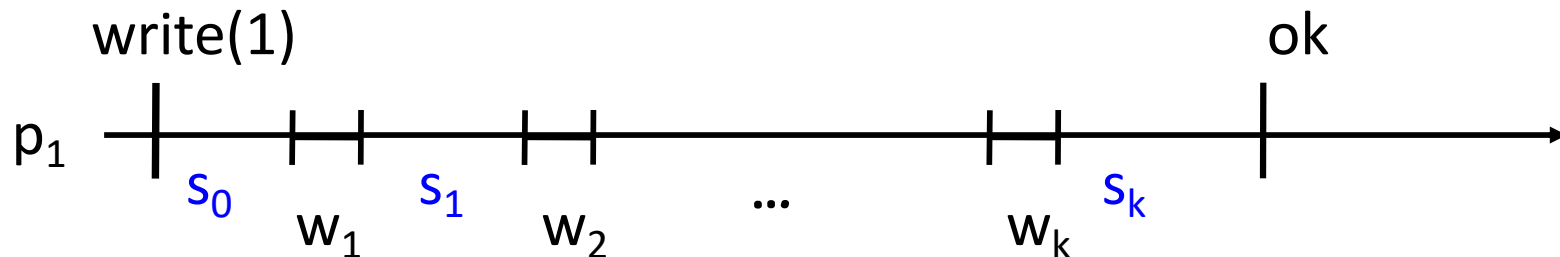
Proof (contd): digests

- There are only finitely many states!

  (bounded # of base registers)

- Each finite (arbitrarily long) sequence $s_0, s_1, \ldots, s_k$ of states defines a bounded digest $d_0, d_1, \ldots, d_m$
  - ✓ $d_0 = s_0$, $d_m = s_k$ (same global state transition)
  - ✓ $d_i = d_j \Rightarrow i = j$ (all digest elements are distinct)
  - ✓ for all (d_i, d_{i+1}), exists (s_j, s_{j+1}) such that $s_j = d_i$ and $s_{j+1} = d_{i+1}$

    7,4,8,4,2,8,3 => 7,4,8,3

- Each write operation "looks" like its digest
- There are only finitely many digests!

Proof (contd.): counter-example

- Consider a run with infinitely many alternating writes:
  $W_1(1)$,$W(0)$,$W_2(1)$,… (no reads)
  - ✓ Writes W_1,W_2,… give an infinite sequence of digests D_1,D_2,…

- At least one digest $D=d_0,d_1,…,d_m$ appears infinitely often in D_1,D_2,…
  - ✓ Why?

- We can amend our run with a sequence of reads R_0,R_1,…,R_m (in that order), each R_i "sees" state d_{m-i}
  - ✓ How?

Quiz 1

- Explain why there can be only finitely many digests

- Explain why in the construction of the proof there is at least one digests that appears infinitely often

- Show how to construct the sequence of reads operations $R_0, R_1, \ldots, R_m$ (in that order) overlapping with $W_1(1), W(0), W_2(1), \ldots$, where each R_i "sees" state d_{m-i}

Proof (contd.): the "switch"

- R_0 "sees" d_m and, thus, returns 1
  - ✓Could have happened right after W(1)

- R_m "sees" d_0 and, thus, returns 0
  - ✓Could have happened right before W(1)

$\Rightarrow$ There exists i such that R_i returns 1 and R_{i+1} returns 0 (by induction on i=0,...,m)

Proof (contd.): contradiction

- The (sequential) execution of R_i and R_{i+1} is indistinguishable (to the reader) from a concurrent one

write(1)   write to a base bit   ok

p_1

... d_{m-i-1} d_{m-i} ...

R_i 1 R_{i+1} 0

p_2

d_{m-i} d_{m-i-1}
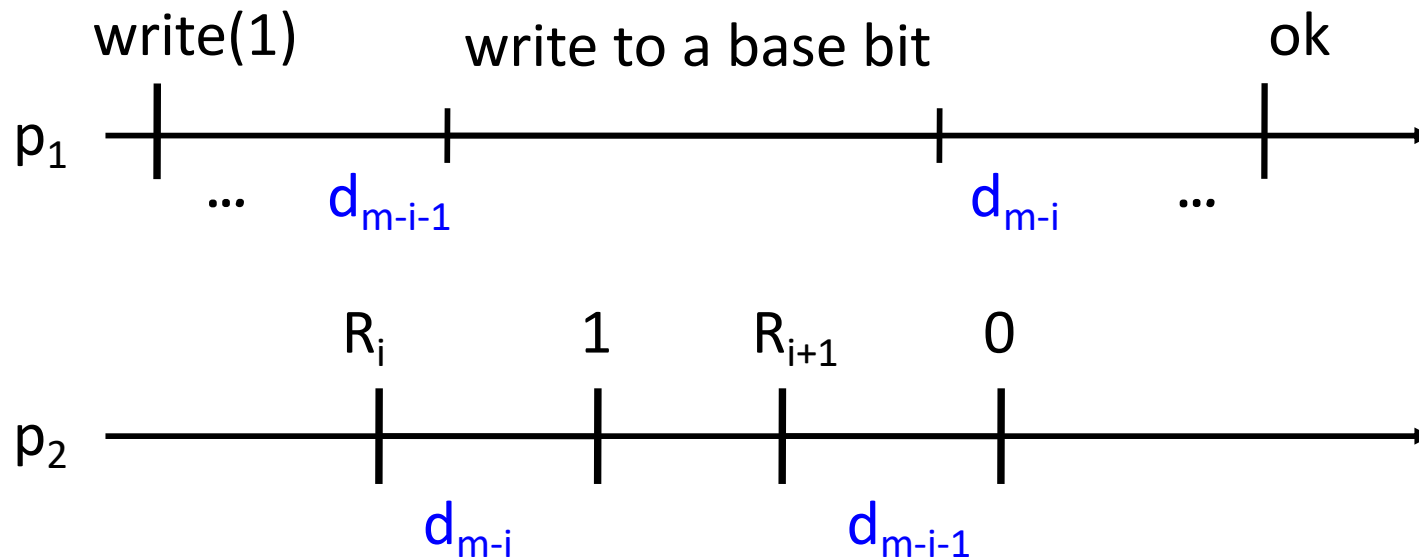
New-old inversion!

The reader must write

- And the writer must read
- But how the writer would tell what it read?
  - ✓ The writer needs at least two bits!
  - ✓ Why?
- Suppose the writer writes to one bit only
  - ✓ there are exactly two digests 0,1 and 1,0
  - ✓ suppose infinitely many W(1) operations export digests 0,1
  - ✓ new-old inversion:

write(1)   change the base bit   ok
                    from 0 to 1

p_1

          read()   1   read()   0

p_2
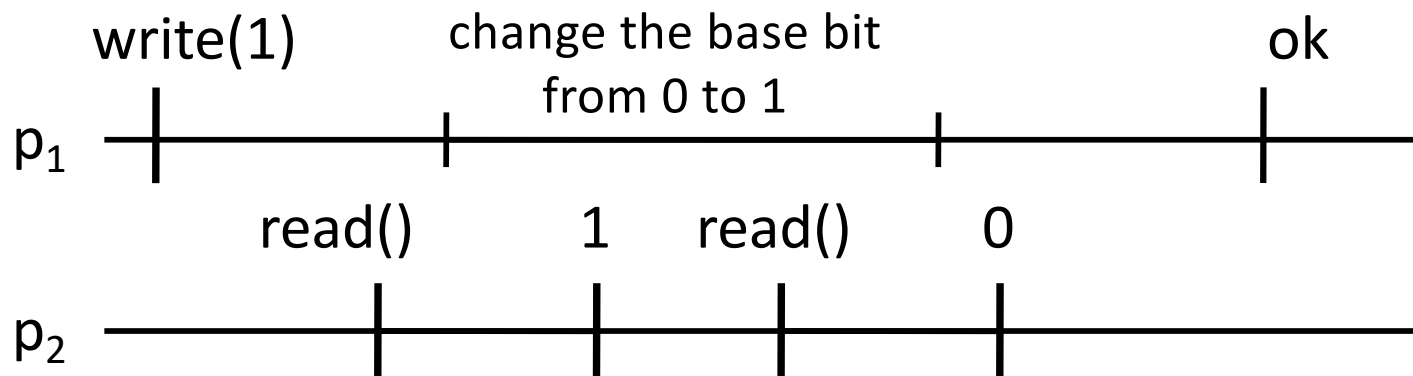
Optimal construction?

- Two bits for the writer
  - ✓ REG: for storing the current value
  - ✓ WR: for signaling to the reader
- One bit for the reader
  - ✓ RR: for signaling to the writer

  Necessary, but is it also sufficient?

Evolutionary approach: Iteration 1

The reader should be able to distinguish the two cases:

- ✓A new value was written: WR≠RR:
- ✓The value is unchanged: WR=RR:

Writer:

change REG
if WR=RR then change WR

Reader:

if WR≠RR then change RR
val:= REG
return val

Does not work: the read value does not depend on RR

Iteration 2

Return the "old" value if nothing changed
(local variable val initialized to the initial value
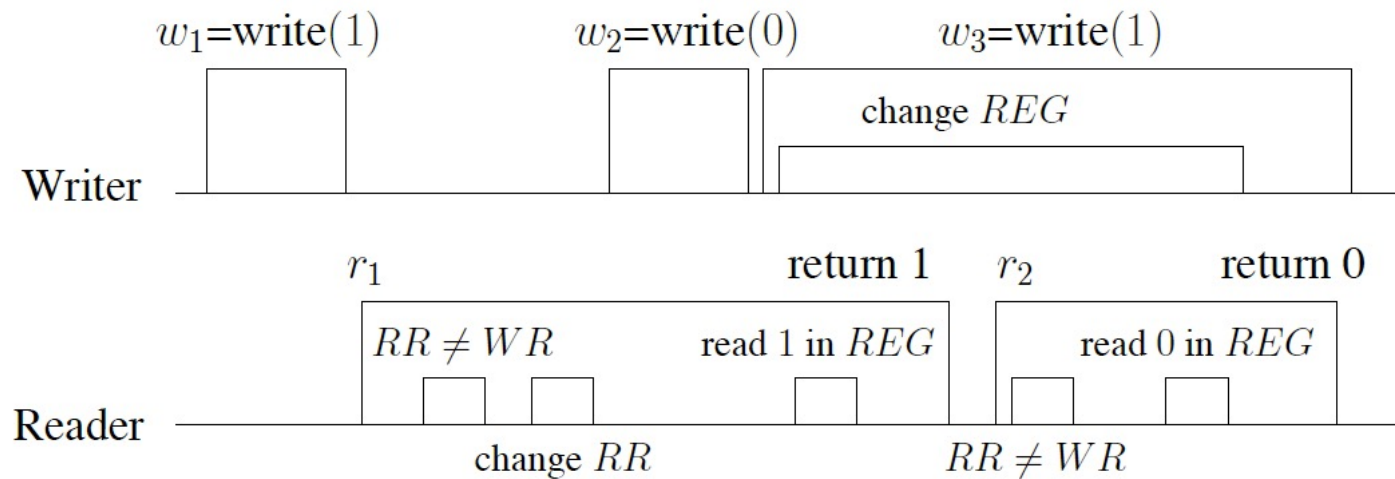of REG)

Writer:

change REG
if WR=RR then change WR

Reader:

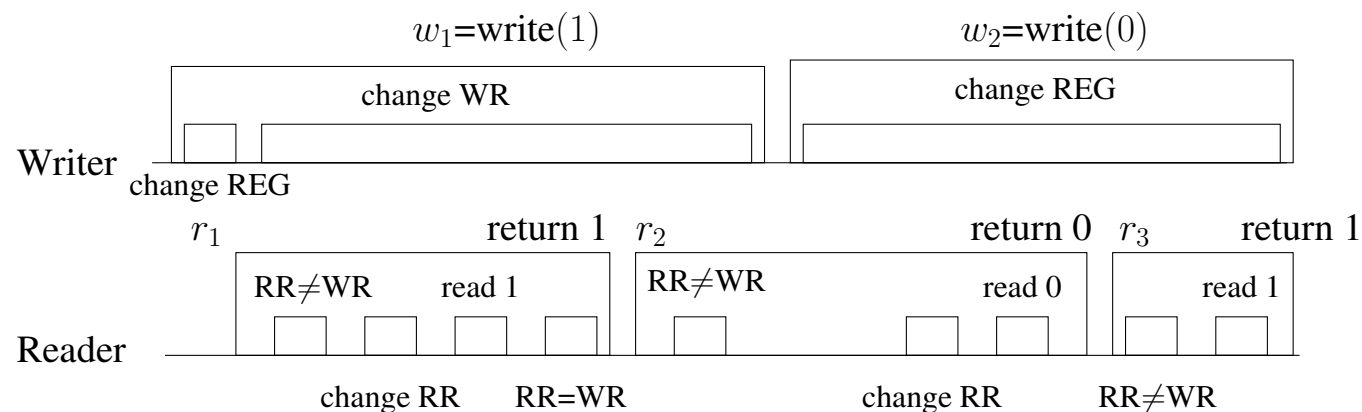if WR=RR then return val
change RR
val:= REG
return val

Counter-example 2?

r_1 reads the new value and r_2 reads the old one?

Is this the case?

© P. Kuznetsov

Counter-example 2, corrected

Does not work: a read finds WR≠RR, a subsequent
read finds WR≠RR and reads an old value in
REG (new-old inversion)

w_1=write(1) w_2=write(0)

change WR   change REG

Writer

change REG

r_1 return 1 r_2 return 0 r_3 return 1

RR≠WR   read 1   RR≠WR   read 0   read 1

Reader

change RR   RR=WR   change RR   RR≠WR

Iteration 3

Only change RR if needed

(read REG before, because otherwise we do not fix the counter-example)

Writer:

change REG

if WR=RR then change WR

Reader:

if WR=RR then return val

val:= REG

if WR≠RR change RR

return val

Construct a counter-example?

Iteration 4

Read WR twice, if WR changed while the read
is executed, return a conservative (old) value

Writer:

change REG
if WR=RR then change WR

Reader:
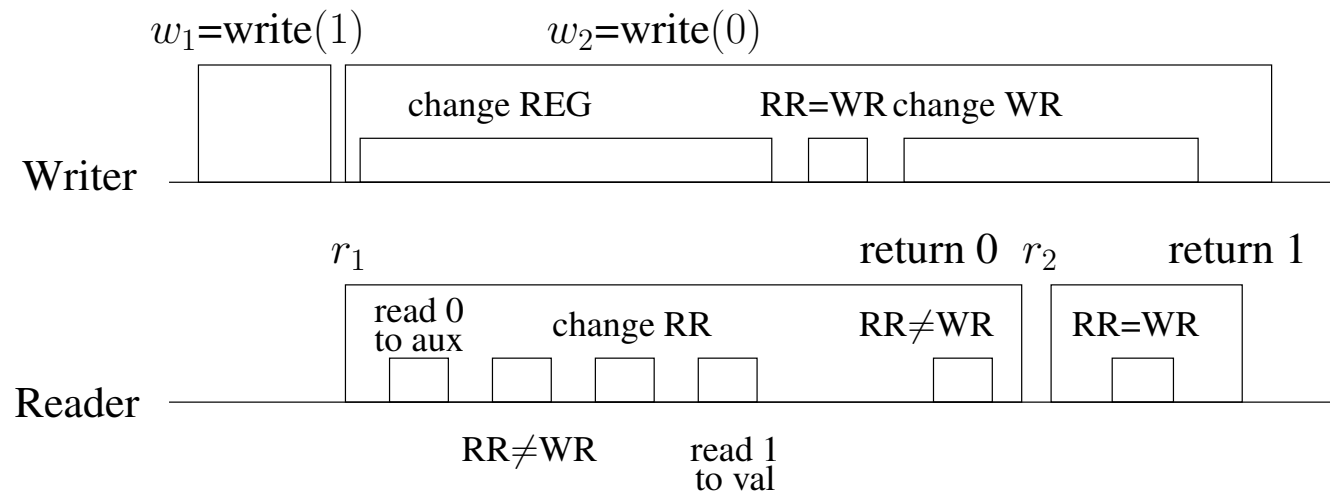
if WR=RR then return val
aux := REG
if WR≠RR change RR
val:= REG
if WR=RR then return val
return aux

© P. Kuznetsov

Counter-example 4

Still a problem: the value stored in val can be too conservative

Solution: evaluate val again

Final solution [Tromp, 1989]

Writer protocol

change REG
if WR=RR then
    change WR

Reader protocol

(1) if WR=RR then return val

(2) aux := REG

(3) if WR≠RR then change RR

(4) val :=REG

(5) if WR=RR then return val

(6) val := REG

(7) return aux

Proof sketch: reading functions

A reading function π: for each complete read operation r (returning v), π(r) is *a* write operation w(v)

Show that for every run of the algorithm, there exists an atomic reading function π :

> (A0) No read r precedes π(r)
>> No read returns a value not yet written
>
> (A1) w precedes r => w= π(r) or w precedes π(r)
>> No read obtains an overwritten value
>
> (A2) r1 precedes r2 => π(r2) does not precede π(r1)
>> No new/old inversion

A run is linearizable iff an atomic reading function exists (Chapter 4.2.4 of the lecture notes)

Proof: constructing π

Let r return a value v

Let ρr be the read of REG that got the value of r :

- If r returns in line 7, ρr is the read action in line 2 of r

- If r returns in line 5, ρr is is the read action in line 4

- If r returns in line 1, ρr is the read in line 4 or 6 of some previous r'
  (depending on how r' returns)

Let ϕr be the last write action on REG that precedes or is concurrent to ρr and writes the value returned by r (and ρr)

Define π(r) as the write operation that contains ϕr

Proof: show that π is atomic

- A0 is easy: by construction of π, π(r) precedes or is concurrent to r

- A1? A2?
  Hint: assume the contrary and come to absurdum

- A complete proof in lecture notes (Chapter 7)

- R. Guerraoui, Vukolic. A Scalable and Oblivious Atomicity Assertion. CONCUR 2008

© P. Kuznetsov

26

Quiz 2

- What if we change the order of lines 2 and 3 in Iteration 2 (Slide 16)?

- Find a mistake in the "counter-example" of Slide 17

- Find a counter-example to the algorithm in Slide 19

- What if the reader returns the value of aux (instead of val) in the first line of Iteration 4 (slide 21)?