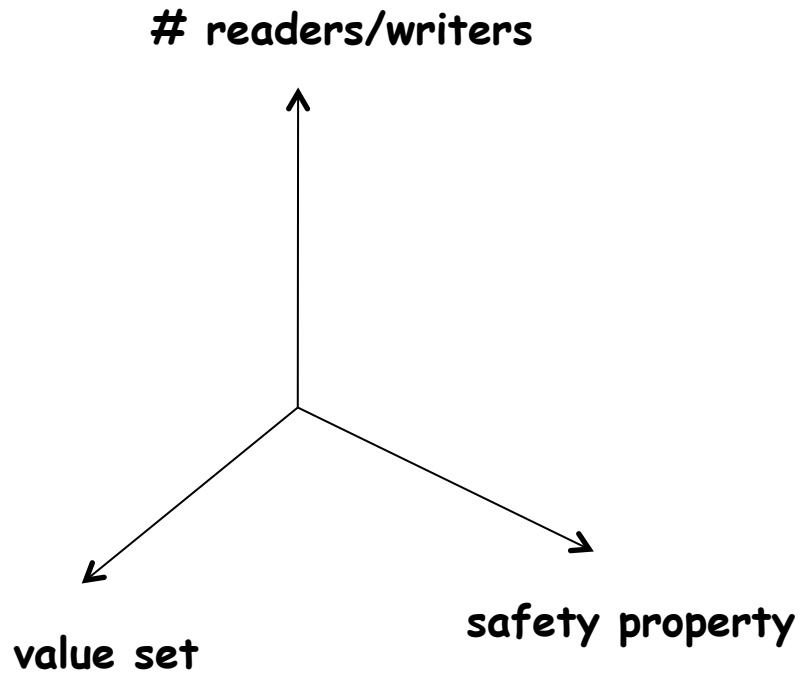
Atomic snapshots

INF346, 2015

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 regular to atomic (1W1R)
V. From 1W1R to 1WNR (multi-valued atomic)
VI. From 1WNR to NWNR (multi-valued atomic)
VII. From safe bit to atomic bit (optimal, coming later)

This class

- Atomic snapshot: reading multiple locations atomically
  - ✓Write to one, read *all*

Atomic snapshot: sequential specification

- Each process p_i is provided with operations:
  - ✓$update_i(v)$, returns ok
  - ✓$snapshot_i()$, returns $[v_1,\ldots,v_N]$

- In a sequential execution:

  For each $[v_1,\ldots,v_N]$ returned by $snapshot_i()$, v_j (j=1, …,N) is the argument of the last $update_j(.)$

  (or the initial value if no such update)

Snapshot for free?

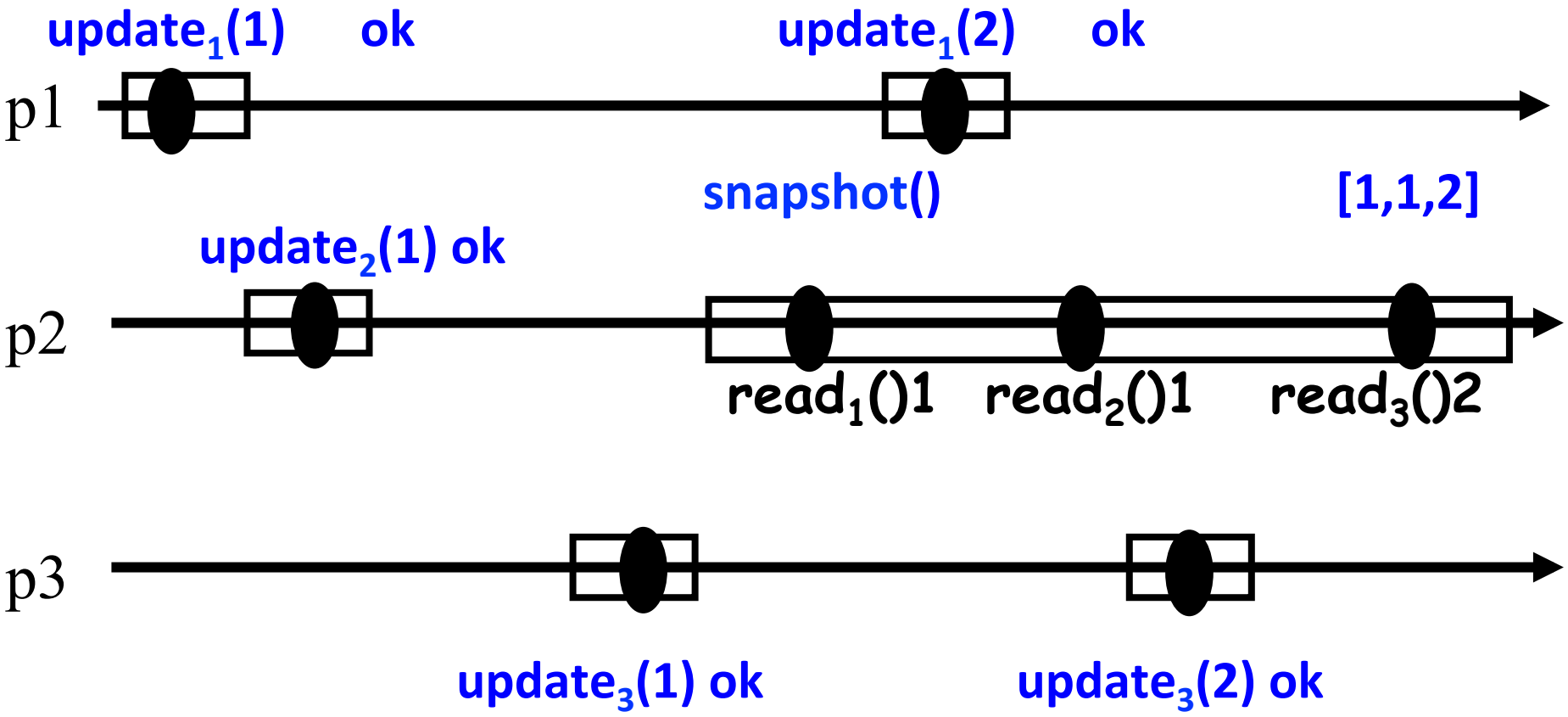
Code for process p_i:

initially:
shared 1W1R *atomic* register $R_i := 0$

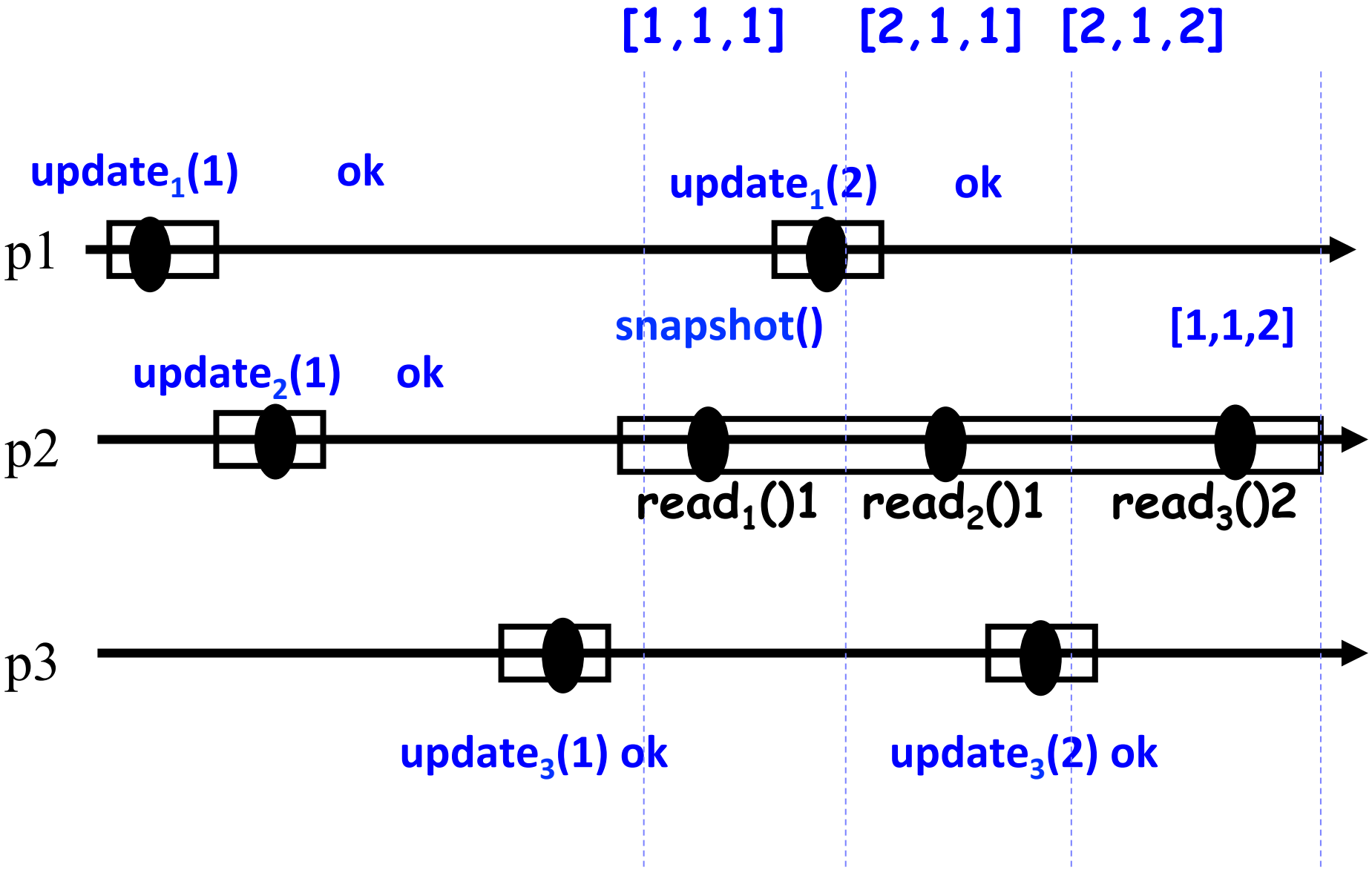
upon snapshot()
$[x_1,\ldots,x_N] := \text{scan}(R_1,\ldots,R_N)$ /*read $R_1,\ldots R_N$*/
return $[x_1,\ldots,x_N]$

upon update$_i$(v)
R_i.write(v)

Snapshot for free?



Snapshot for free?



- What about 2 processes?

- What about lock-free snapshots?
  - ✓At least one correct process makes progress (completes infinitely many operations)

Lock-free snapshot

Code for process p_i (all written value are unique, e.g., equipped with a sequence number)

Initially:

shared 1W1R atomic register $R_i := 0$

upon snapshot()

$[x_1,\ldots,x_N]$:= scan($R_1,\ldots,R_N$)

repeat

$[y_1,\ldots,y_N] := [x_1,\ldots,x_N]$

$[x_1,\ldots,x_N]$:= scan($R_1,\ldots,R_N$)

until $[y_1,\ldots,y_N] = [x_1,\ldots,x_N]$

return $[x_1,\ldots,x_N]$

upon update$_i$(v)

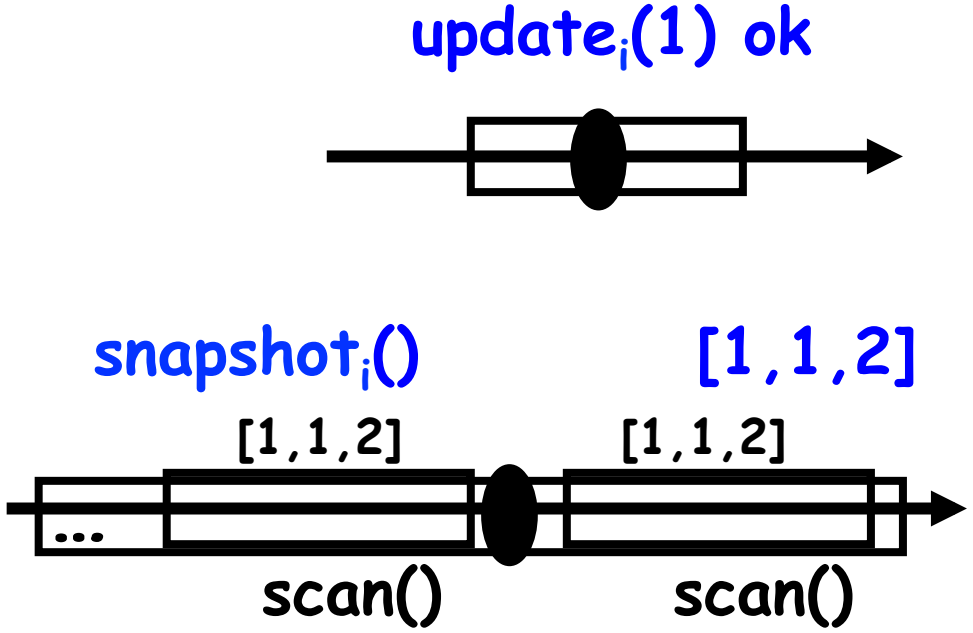
R_i.write(v)

Linearization

Assign a linearization point to each operation

- $update_i(v)$
  - ✓R_i.write(v) if present
  - ✓Otherwise remove the op
- $snapshot_i()$
  - ✓if complete – any point between identical scans
  - ✓Otherwise remove the op

Build a sequential history S in the order of linearization points

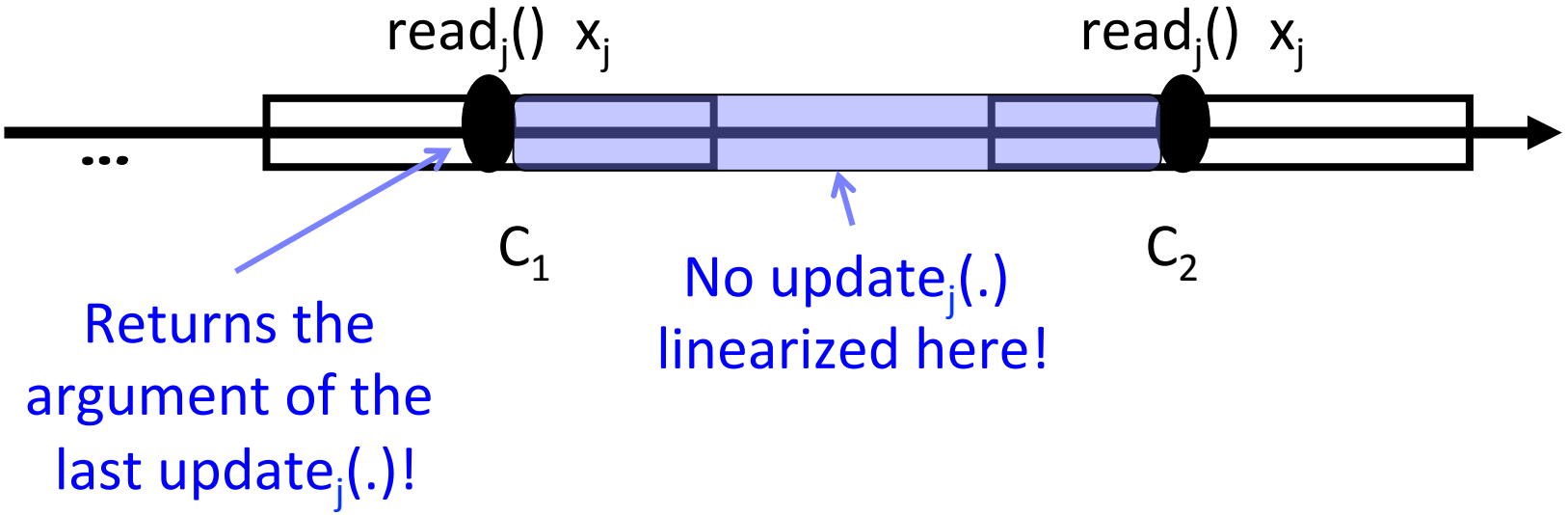


Correctness: linearizability

S is legal: every $snapshot_i()$ returns the last written value for every p_j

Suppose not: $snapshot_i()$ returns $[x_1,\ldots,x_N]$ and some x_j is not the the argument of the last $update_j(v)$ in S preceding $snapshot_i()$

Let C_1 and C_2 be two scans that returned $[x_1,\ldots,x_N]$



Correctness: lock-freedom

An $update_i()$ operation is wait-free (returns in a finite number of steps)

Suppose process p_i executing $snapshot_i()$ eventually runs in isolation (no process takes steps concurrently)

- All scans received by p_i are distinct
- At least one process performs an update between
- There are only finitely many processes => at least one process executes infinitely many updates

What if base registers are regular?

General case: helping?

What if an update interferes with a snapshot?

- Make the update do the work!

upon snapshot()

$[x_1,\ldots,x_N]$:= scan($R_1,\ldots, R_N$)
$[y_1,\ldots,y_N]$:= scan($R_1,\ldots,R_N$)
if $[y_1,\ldots,y_N] = [x_1,\ldots,x_N]$ then
    return $[x_1,\ldots,x_N]$
else
    let j be such that
        $x_j \neq y_j$ and $x_j=(u,U)$
    return U

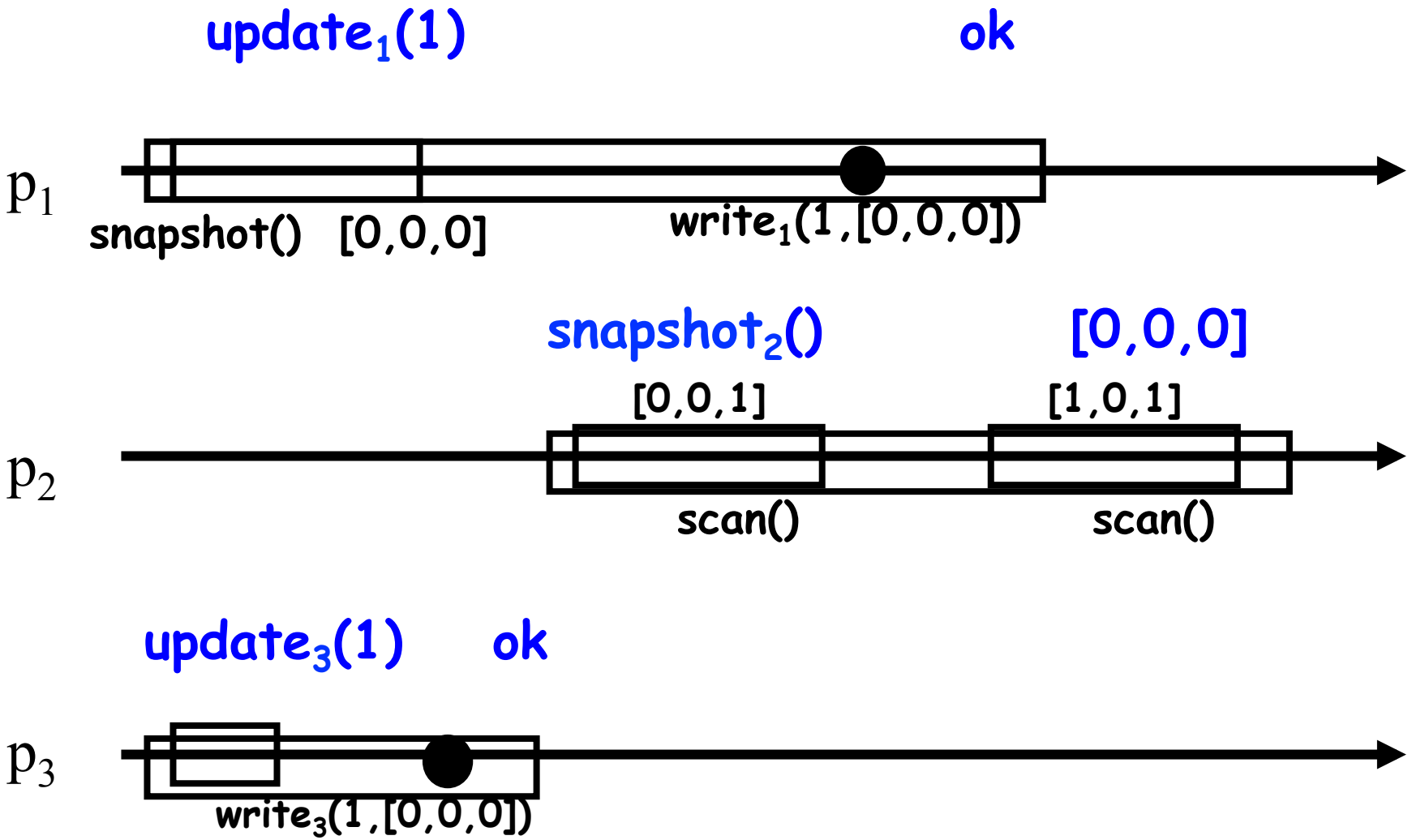
upon update$_i$(v)

S := snapshot()
R_i.write(v,S)

If two scans differ – some update succeeded to snapshot!

Would this work?

Not that easy!



General case: wait-free atomic snapshot

upon snapshot()

$[x_1,\ldots,x_N]$:= scan($R_1,\ldots,R_N$)

while true do

  $[y_1,\ldots,y_N]$:= $[x_1,\ldots,x_N]$

  $[x_1,\ldots,x_N]$:= scan($R_1,\ldots,R_N$)

  if $[y_1,\ldots,y_N] = [x_1,\ldots,x_N]$ then

    return $[x_1,\ldots,x_N]$

  else if $moved_j$ and $x_j \neq y_j$ then

    let $x_j = (u,U)$

    return U

  for each j: $moved_j := moved_j \vee x_j \neq y_j$

upon update$_i$(v)

  S := snapshot()

  R_i.write(v,S)

If a process moved twice: its last snapshot is valid!

Correctness: wait-freedom

Claim 1 Every operation (update or snapshot) returns in $O(N^2)$ steps (bounded wait-freedom)

snapshot: does not return after a scan if a concurrent process moved and no process moved twice

- At most N-1 concurrent processes, thus (pigeonhole), after N scans:
  - ✓Either at least two consecutive identical scans
  - ✓Or some process moved twice!

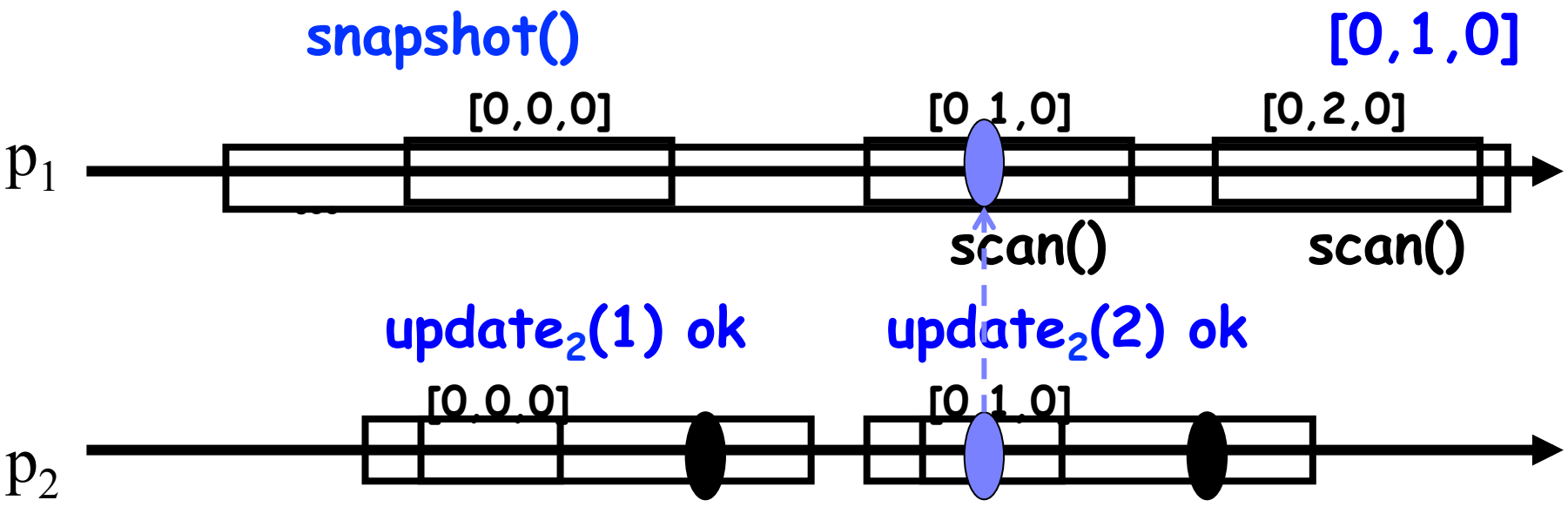
update: snapshot() + one more step

Correctness: linearization points

update$_i$(v): linearize at the R_i.write(v,S)

complete **snapshot()**

- If two identical scans: between the scans
- Otherwise, if returned U of p_j: at the linearization point of p_j's snapshot



The linearization is:

- Legal: every snapshot operation returns the most recent value for each process
- Consistent with the real-time order: each linearization point is within the operation's interval
- Equivalent to the run (locally indistinguishable)

(Full proof in the lecture notes, Chapter 6)

One-shot atomic snapshot (AS)

Each process p_i:

update$_i$(v_i)

S_i := snapshot()

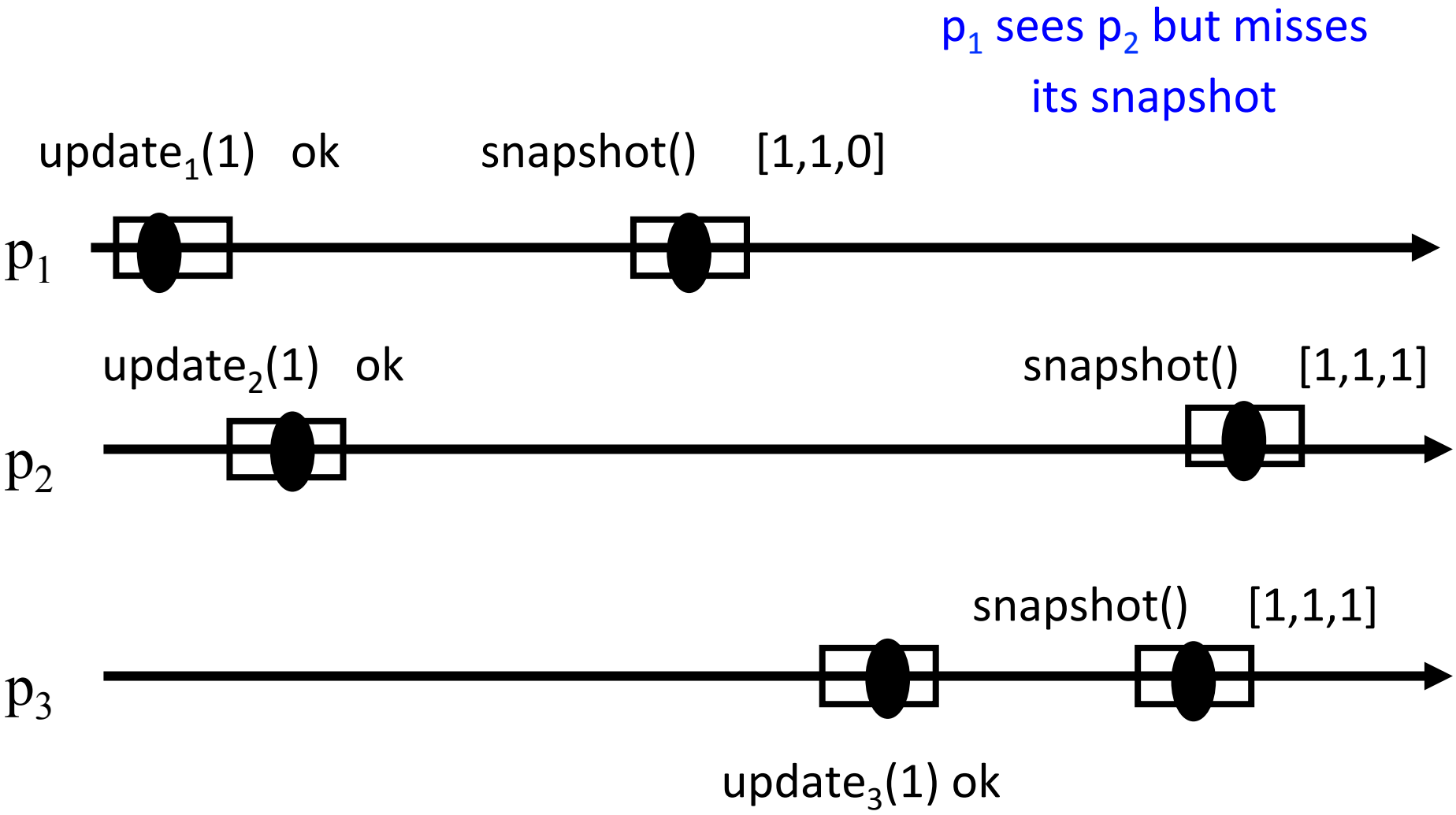
$S_i = S_i[1],\ldots,S_i[N]$

(one position per process)

Vectors S_i satisfy:

- Self-inclusion: for all i: v_i is in S_i
- Containment: for all i and j: S_i is subset of S_j or S_j is subset of S_i

“Unbalanced” snapshots



Enumerating possible runs: two processes

Each process p_i (i=1,2):

$update_i(v_i)$

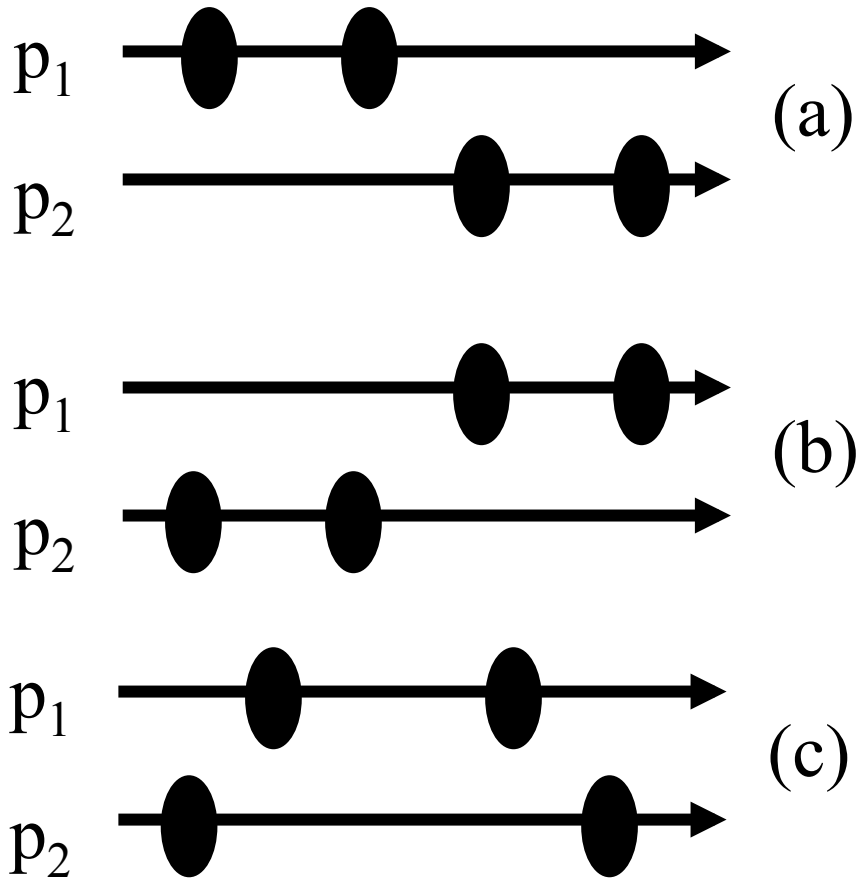
S_i := snapshot()

Three cases to consider:

(a) p_1 reads before p_2 writes

(b) p_2 reads before p_1 writes

(c) p_1 and p_2 go "lock-step": first both write, then both read



Quiz: atomic snapshots

Prove that one-shot atomic snapshot satisfies self-inclusion and containment:

- Self-inclusion: for all i: v_i is in S_i
- Containment: for all i and j: S_i is subset of S_j or S_j is subset of S_i

Bibliographic project

- 15 mins presentation of a research paper + 5 mins discussion
  - ✓What is the problem? What is its motivation?
  - ✓What is the idea of the solution?
  - ✓What is new and what is interesting here?
    - Technical details: less necessary

- Final grade = 1/3 for the presentation (April 22) + 2/3 exam (April 24)

- The list of papers (with pdfs) and the link to a form to submit your choice:
  - ✓http://perso.telecom-paristech.fr/~kuznetso/INF346-2015/
  - ✓By March, 2015

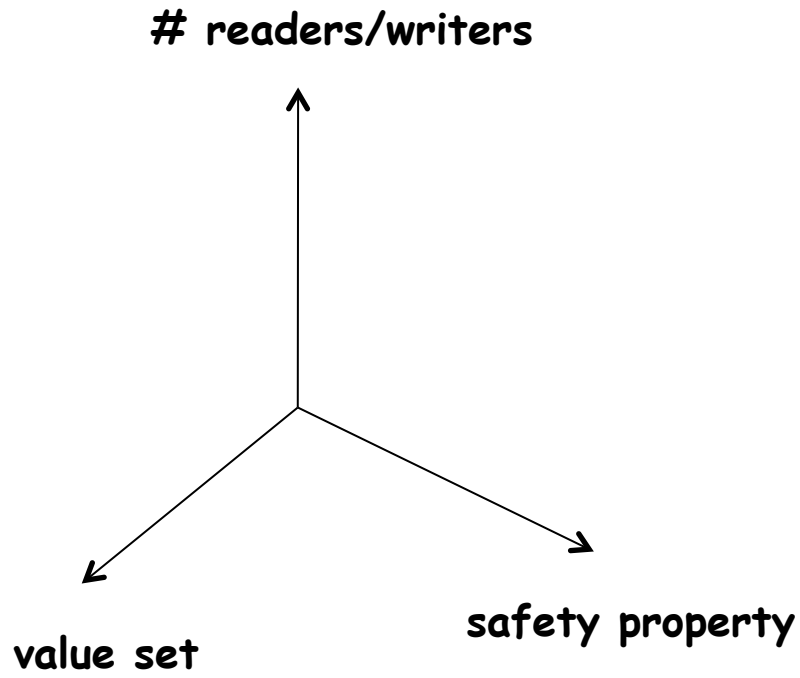
Algorithms for Concurrent Systems

Implementing an atomic bit

MPRI, period 1, 2015

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 bounded (in n) sequence numbers

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

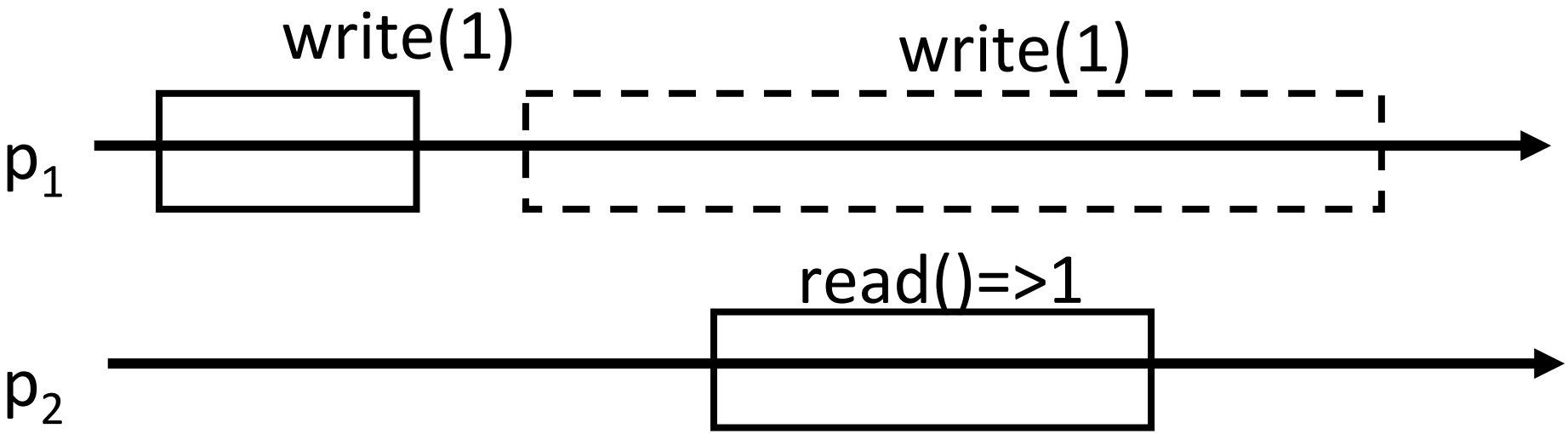
An optimal solution

- 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

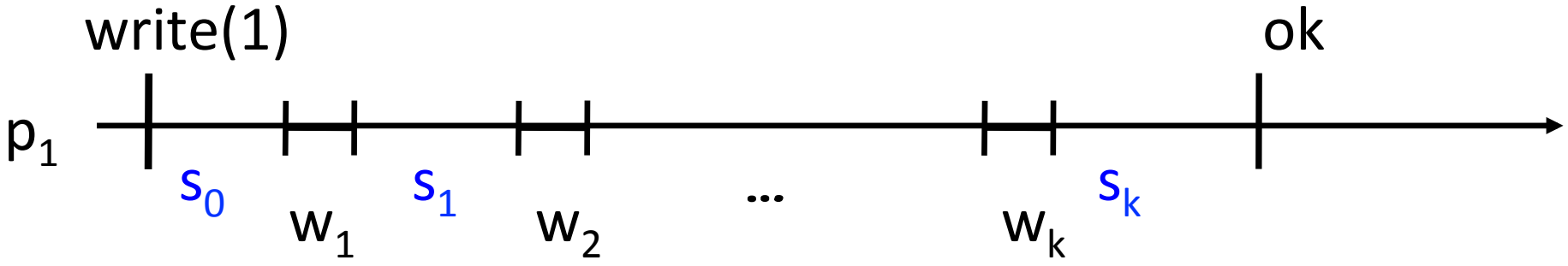


Can we get an atomic bit this way?

Impossible if the reader does not write for bounded # of regular bits!

Proof sketch (by contradiction):

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



Proof (contd): digests

- There are only finitely many states!
  (bounded # of base registers)
- Each sequence $s_0,s_1,\ldots,s_k$ of states (though possibly unbounded) defines a bounded digest $d_0,d_1,\ldots,d_m$
  - ✓ $d_0=s_0$, $d_m=s_k$ (same global state transition)
  - ✓ $d_0=s_0 \Rightarrow i=j$ (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 $W_1,W_2,\ldots$ give an infinite sequence of digests $D_1,D_2,\ldots$
- At least one digest $D=d_0,d_1,\ldots,d_m$ appears infinitely often in $D_1,D_2,\ldots$
  - ✓Why?
- We can amend our run with a sequence of reads $R_0,R_1,\ldots,R_m$ (in that order), each R_i "sees" state d_{m-i}
  - ✓How?

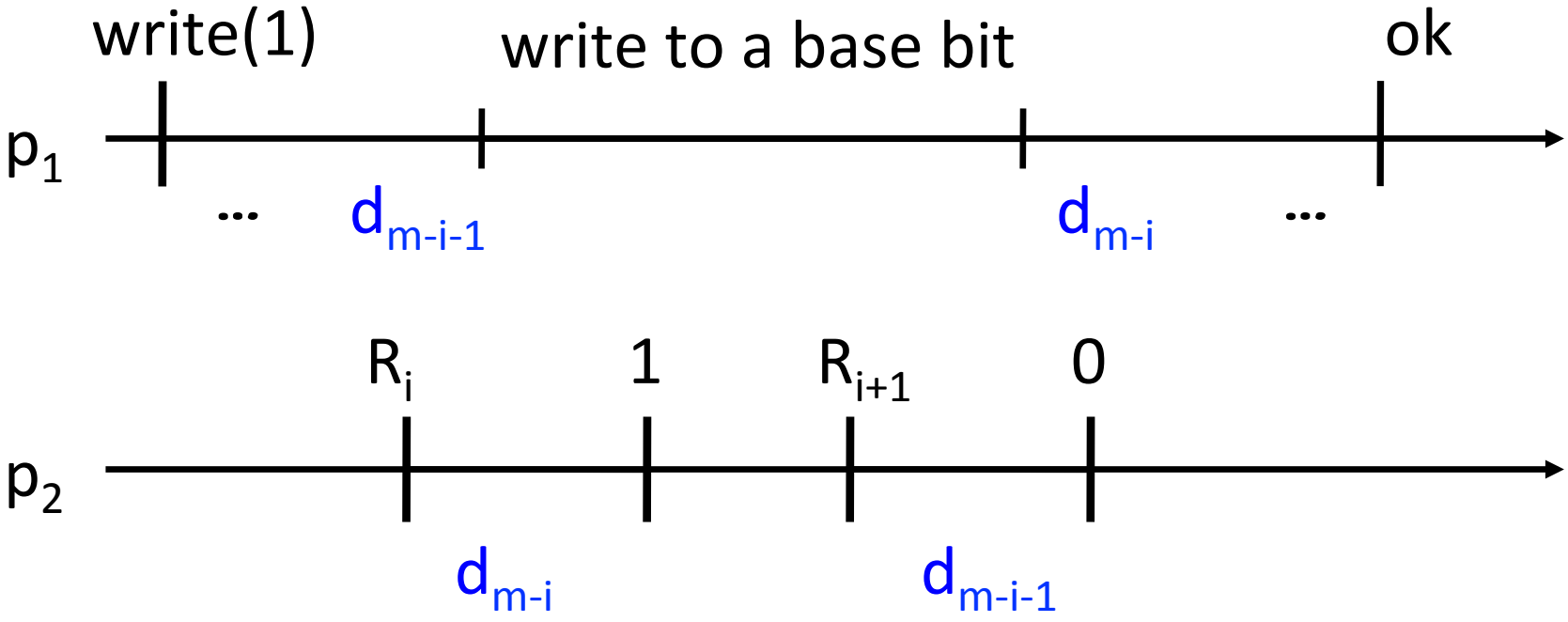
Proof (contd.): the "switch"

- 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

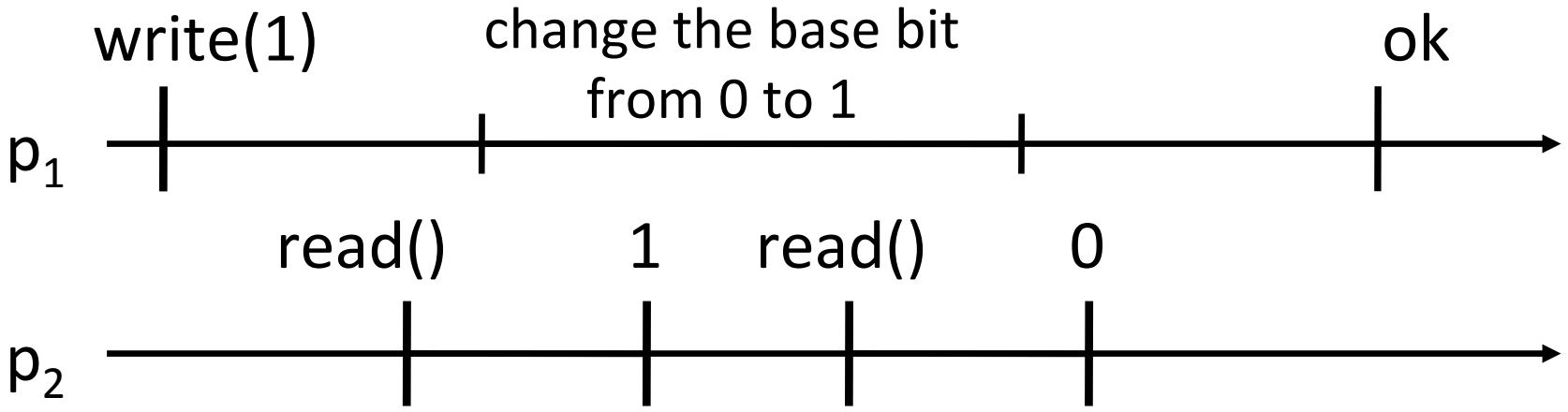
- The (sequential) execution of R_i and R_{i+1} is indistinguishable (to the reader) from a concurrent one



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:



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:

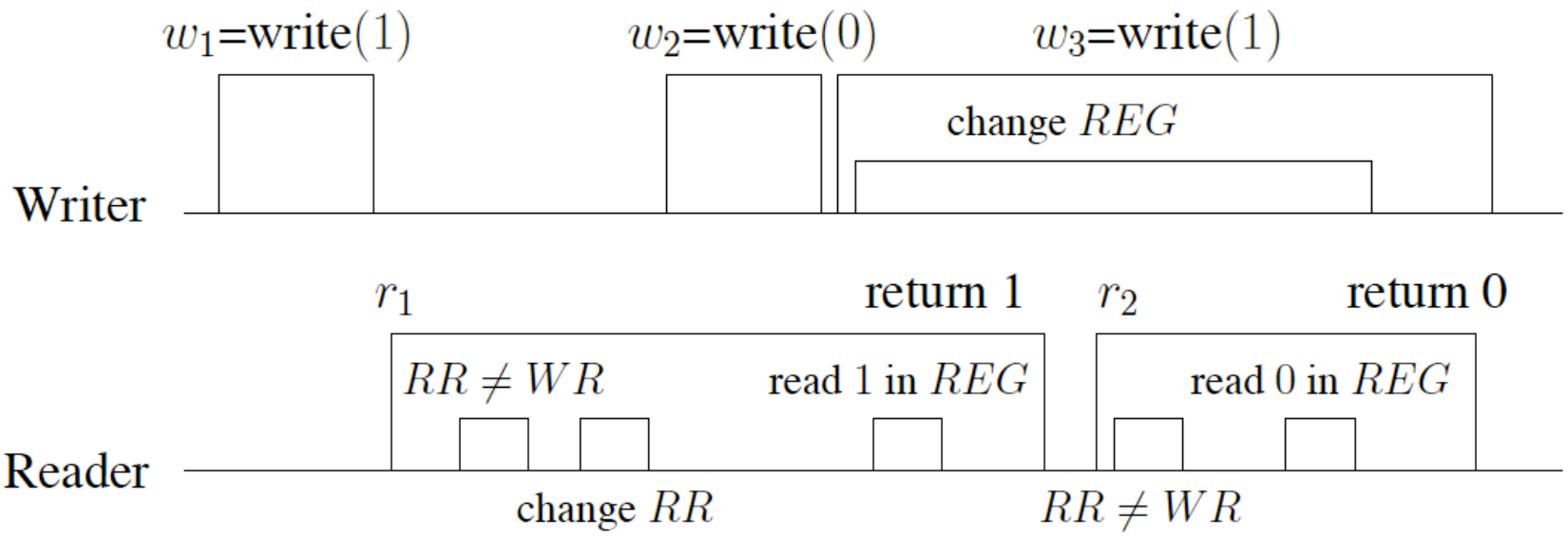
change REG
if WR=RR then change WR

Reader:

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

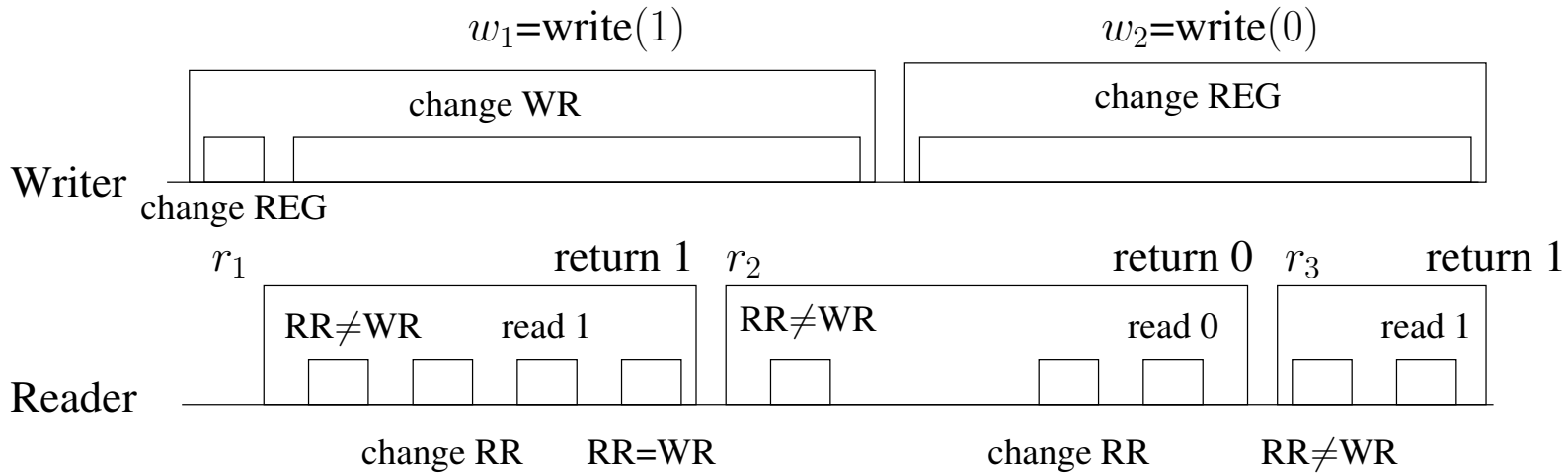
Counter-example 2

Does not work? r_1 reads the new value and r_2 reads the old one



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)



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

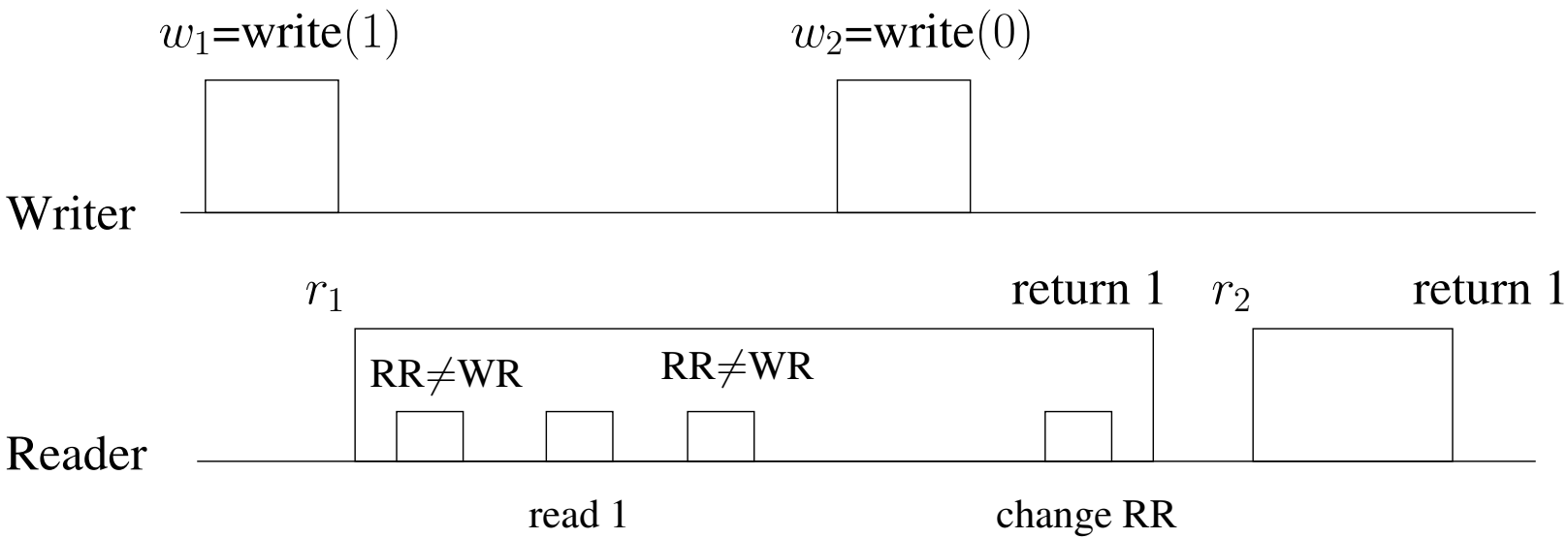
if WR≠RR change RR

return val

Construct a counter-example?

Counter-example 3

Does not work: a read sets RR=WR while the value in val has been overwritten



Solution: check WR again before returning the value

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

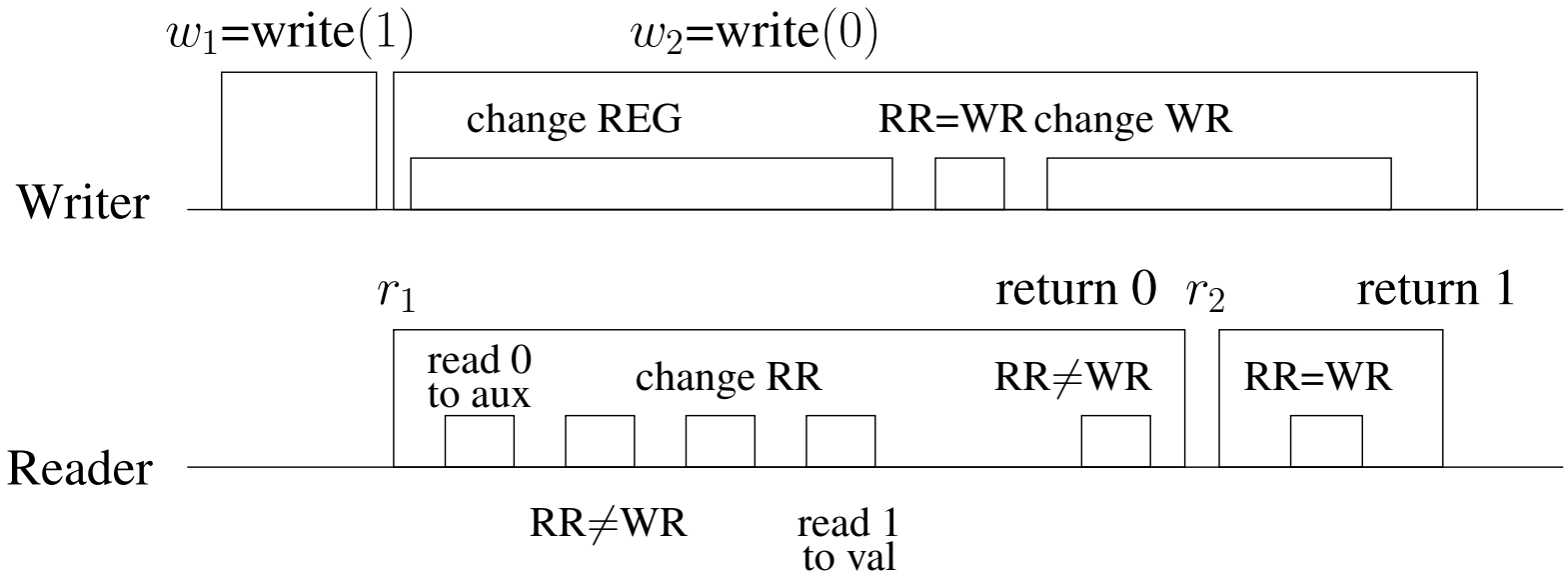
val:= REG

if WR=RR then return val

return aux

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) r_1 precedes r_2 => π(r_2) does not precede π(r_1)
  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 ρ_r be the read of REG that got the value of r
  - ✓ If r returns in line 7, ρ_r is the read action in line 2 of r
  - ✓ If r returns in line 5, ρ_r is is the read action in line 4
  - ✓ If r returns in line 1, ρ_r is is the read in line 4 or 6 of some previous r' (depending on how r' returns)
- Let φ_r be the last write action on REG that precedes or is concurrent to ρ_r and writes the value returned by r (and ρ_r)
- Define π(r) as the write operation that contains φ_r

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