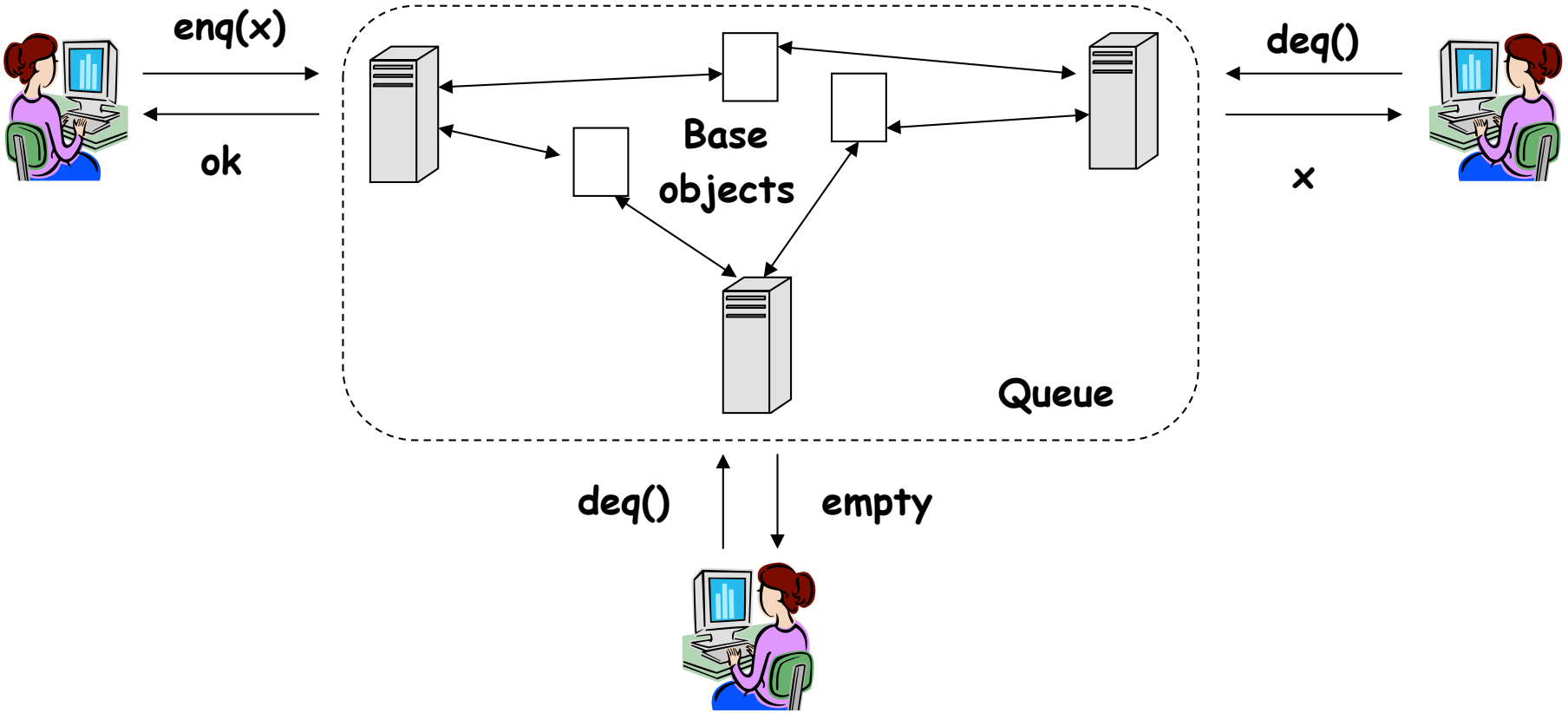
TDC1:
Atomic snapshot and atomic bits

Implementing a concurrent object

Using *base* objects, an implementation of an object O creates an illusion that an *atomic* object O is available



Implementation: correctness

A (wait-free) implementation A is correct if in every well-formed run of A

- **Wait-freedom:** every operation run by p_i returns in a finite number of steps of p^i
- **Strict linearizability** ≈ operations "appear" instantaneous (the corresponding *history* is *strictly linearizable*)
  - ✓The formal definition follows

Histories

A history of a given run of A is a sequence of invocation and responses on the implemented object in that run

E.g., p1->Q.enqueue(x), p2->Q.dequeue(),p1->ok, p2->x,…

A history is sequential if every invocation is followed by a corresponding response

E.g., p1->Q.enqueue(x), p1->ok, p2->Q.dequeue(), p2->x,…

(A sequential history has no concurrent operations)

Legal histories

A sequential history is *legal* if it satisfies the sequential specification of the implemented object

Examples:

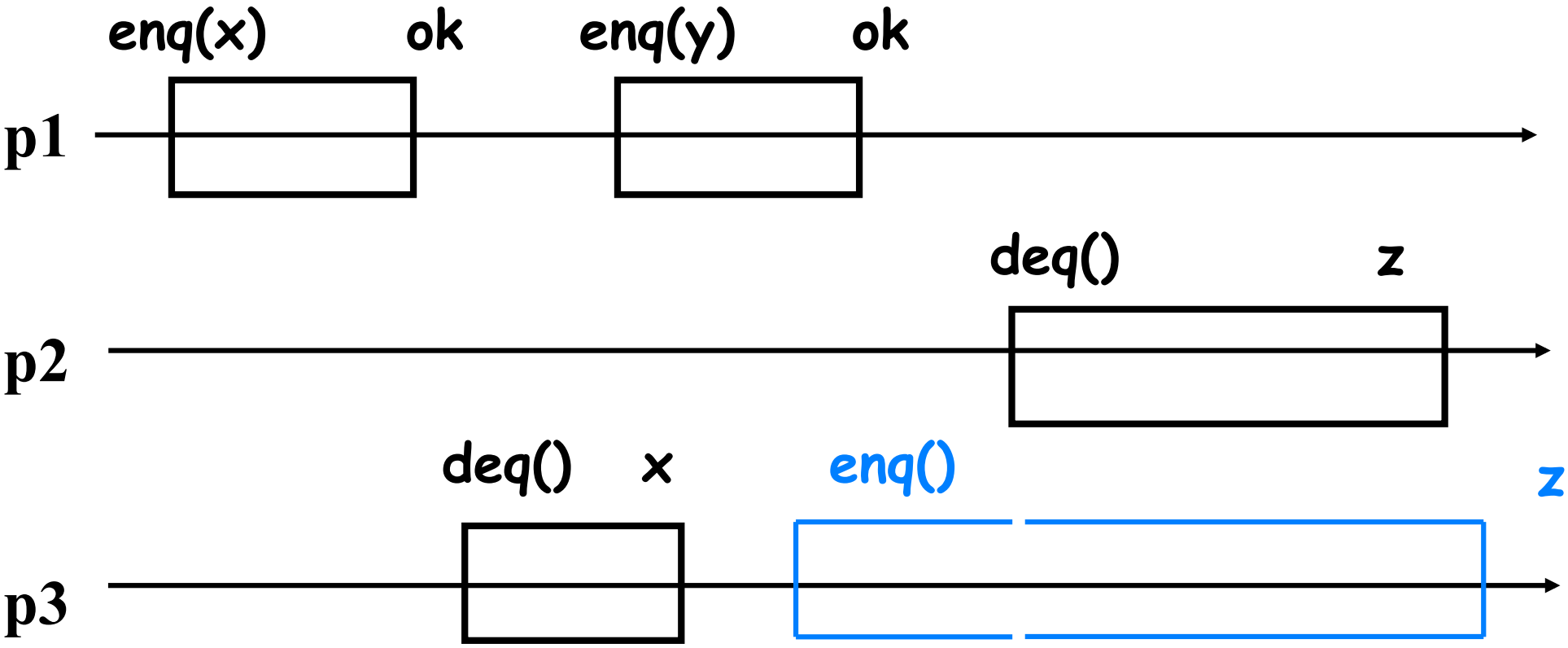
- Read-write registers: every read returns the argument of the last write
- LIFO queues: every dequeue() returns the argument of the last enqueued element

Complete operations and completions

An operation op is *complete* in H if H contains both the invocation and the response of op

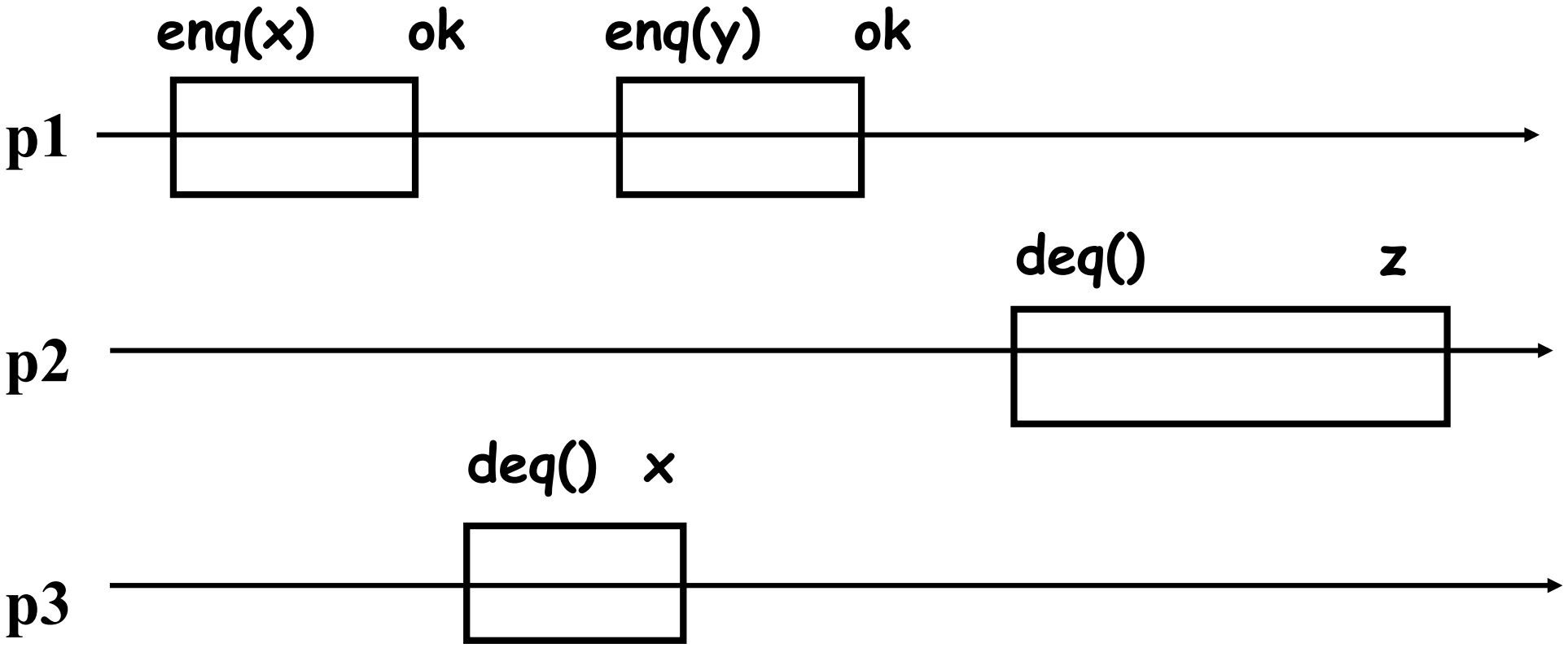
A *completion* of H is a complete history H' that contains all complete operations of H and a *subset* of incomplete operations of H completed with *matching* responses

Complete operations and completions



p1-enq(x); p1-ok; p3-deq(); p1-enq(y); p3-x;
p3-enq(); p1 -ok; p2-dequeue(); p2-z; p3->z

Complete operations and completions



p1-enq(x); p1-ok; p3-deq(); p1-enq(y); p3-x; p1 -ok;
p2-dequeue(); p2-z

Equivalence

Histories H and H’ are *equivalent* if for all pi

$H|p_i = H'|p_i$

E.g.:

$H=p_1$-write(0); p_3-read(); p_1-ok; p_3-3

$H'=p_1$-write(0); p_1-ok; p3-read(); p_3-3

Strict linearizability

A history H is *strictly linearizable* if there exists a legal sequential history S such that:

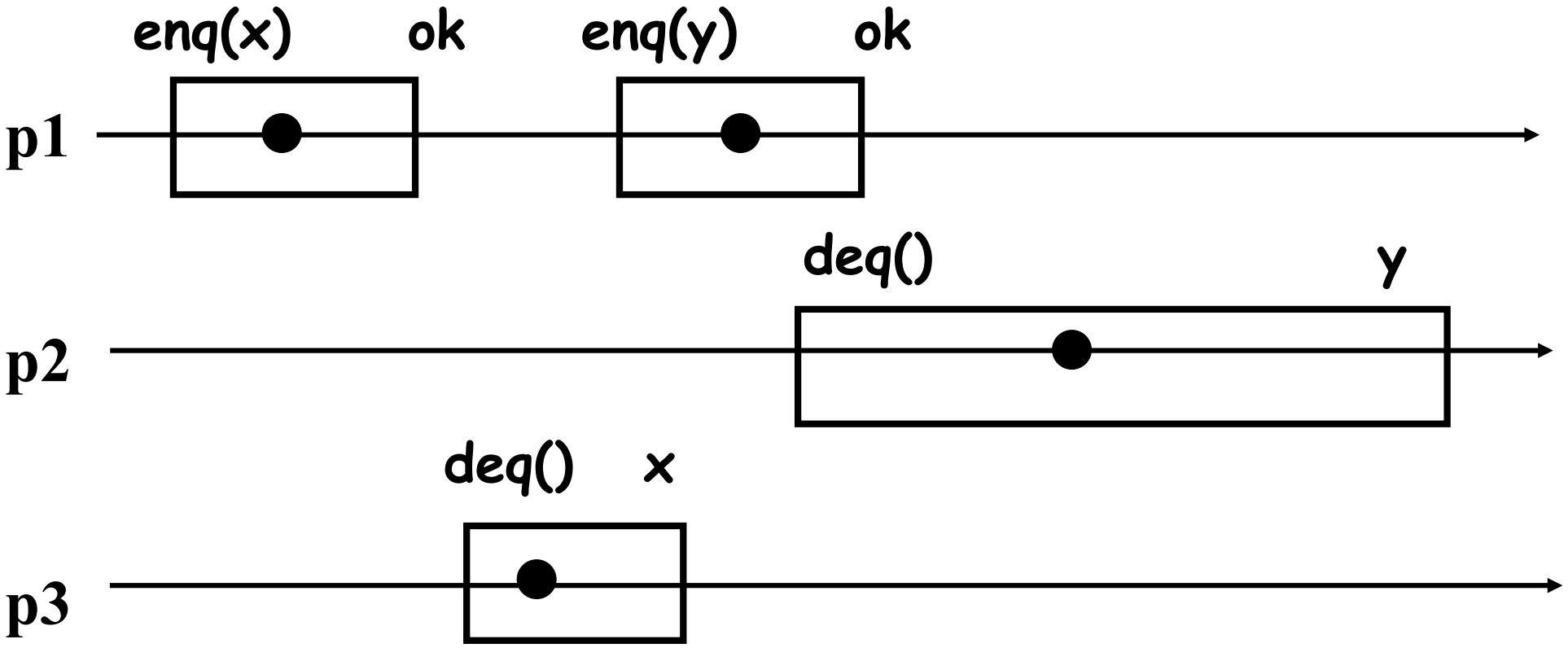
- S is equivalent to some completion of H
- S preserves the precedence relation of H:

op1 precedes op2 in H => op1 precedes op2 in S

All complete operations and a subset of incomplete operations can be *linearized* (totally ordered preserving precedence)

An implementation is strictly linearizable if every finite run of it produces a strictly linearizable history

Linearization

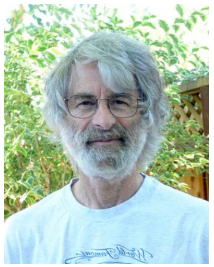


p1-enq(x); p1-ok; p3-deq(); p1-enq(y); p3-x;
p1 -ok; p2-dequeue(); p2-y

This class

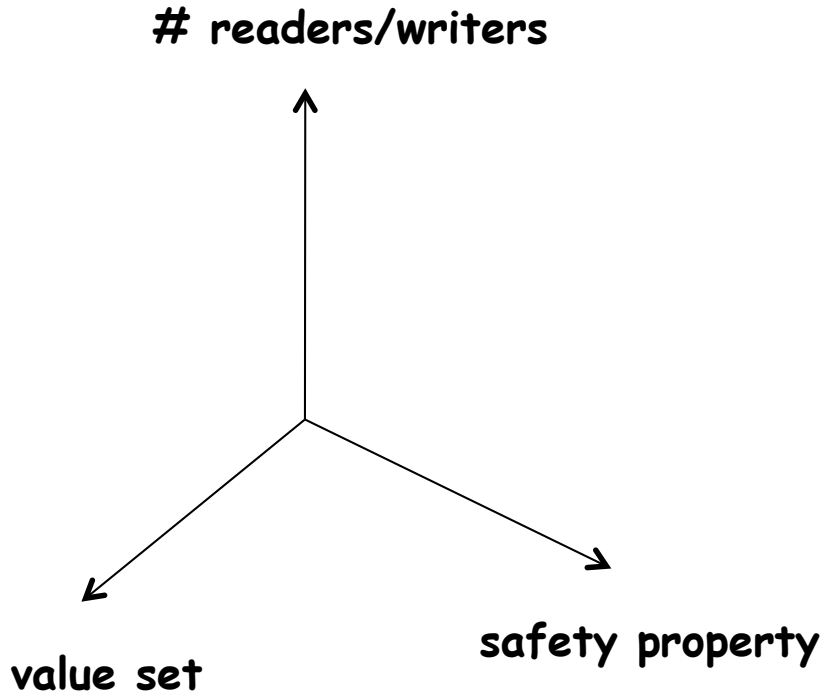
- Atomic snapshot: reading multiple locations atomically
- Atomic bit construction: from safe to atomic

The space of registers



L. Lamport

- 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 atomic registers to N-Atomic Snapshot!
- ✓ Write to one, read all *atomically*

Atomic snapshot: specification

- Each process pi is provided with
  - ✓ $update_i(v)$, returns ok
  - ✓ $scan_i()$, returns $[v_1,\ldots,v_N]$

- In a sequential execution:

  For each $[v_1,\ldots,v_N]$ returned by $scan_i()$, 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 scan()

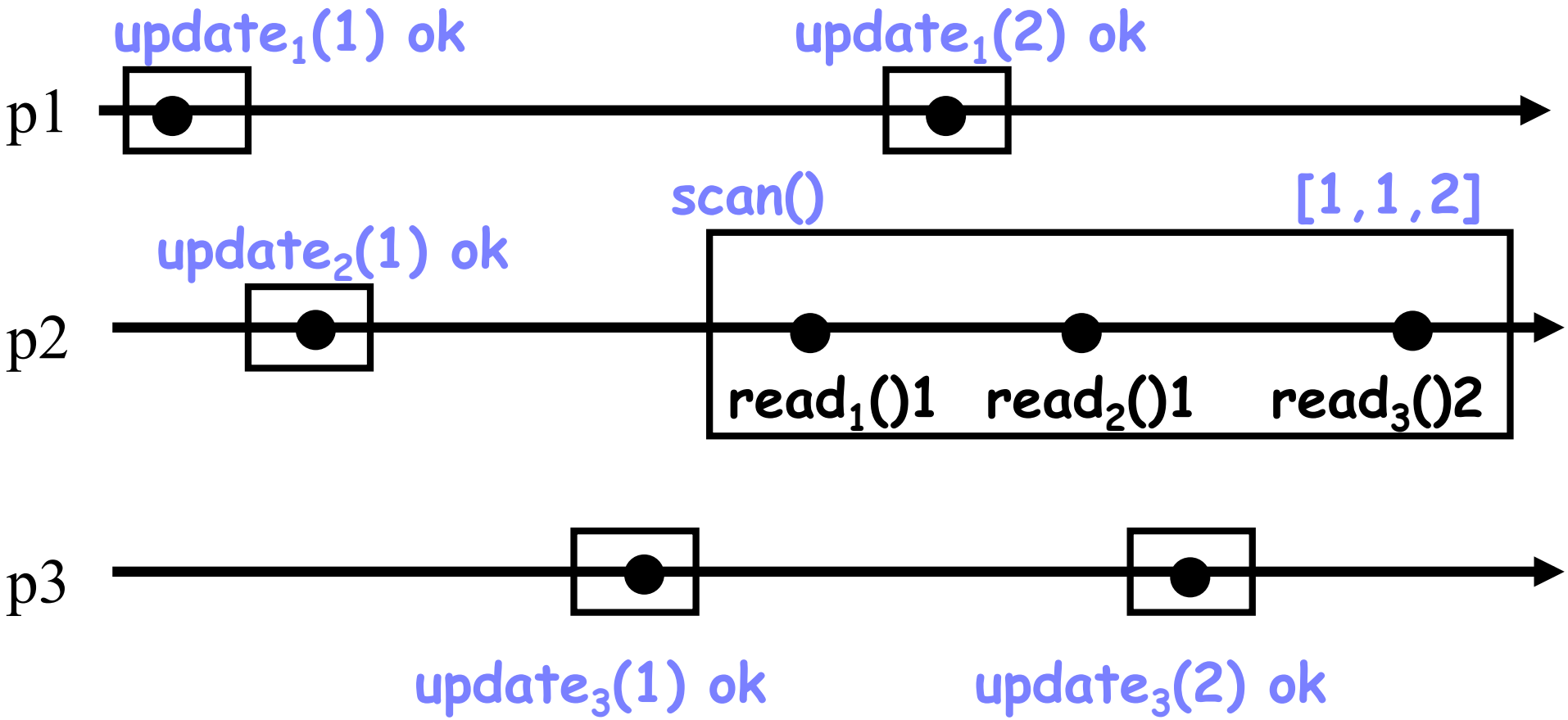
for all j do x_j := collect(R_1,…,R_N) /*read R_1,…R_N*/

return [x_1,…,x_N]

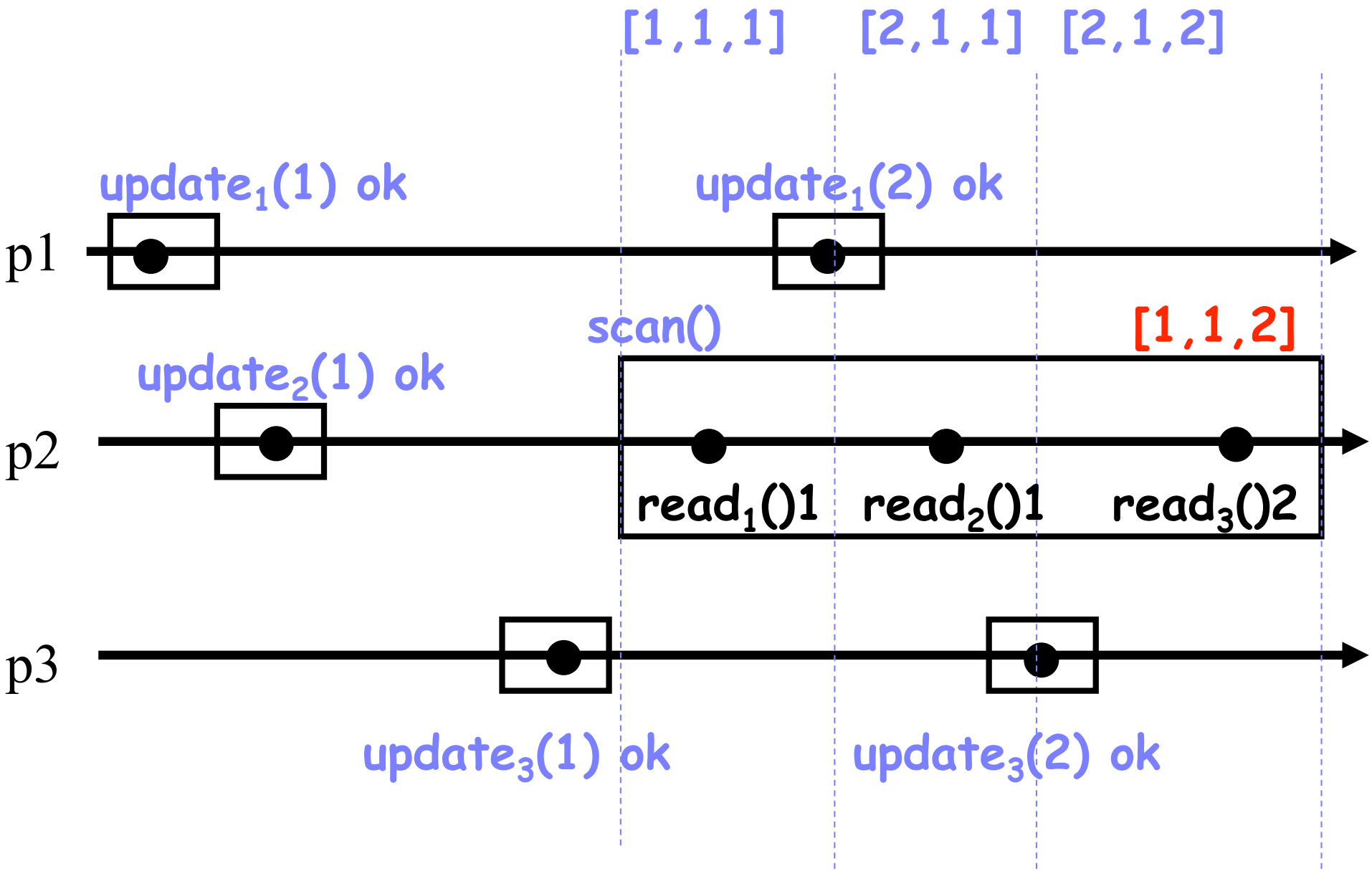
upon update$_i$(v)

R_i.write(v)

Snapshot for free?



Snapshot for free?



- What about 2 processes? Can we get a snapshot for free?

- What about obstruction-free snapshot?
  - ✓Every operation that eventually runs in isolation eventually returns

Obstruction-free snapshot

Code for process p_i (all written value are unique):

Initially:

shared 1W1R atomic register $R_i := 0$

upon scan()

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

repeat

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

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

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

return $[x_1,\ldots,x_N]$

upon update$_i$(v)

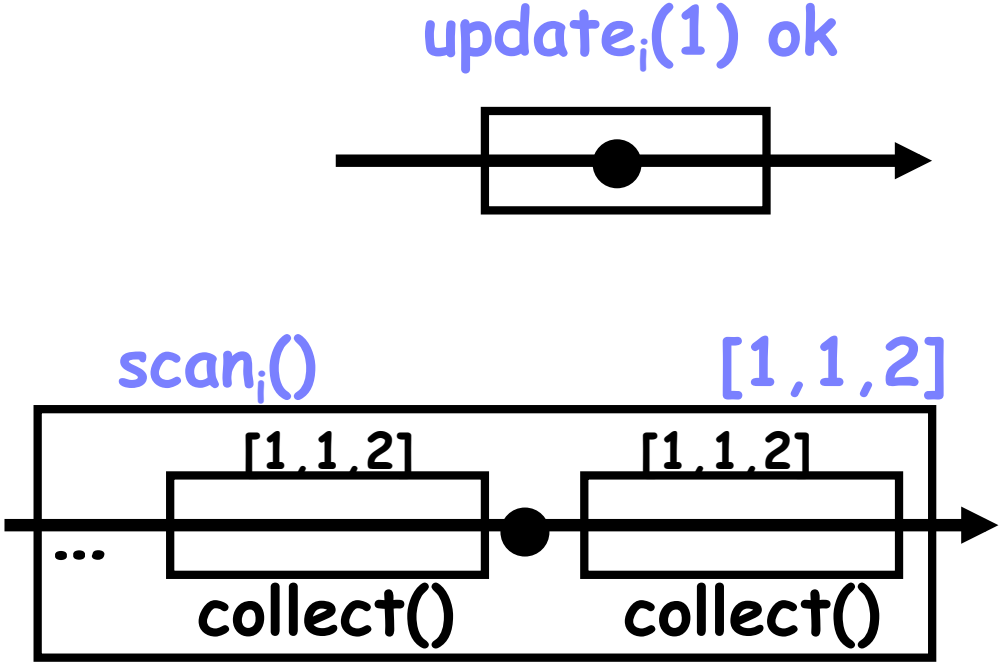
R_i.write(v)

Linearization

Assign a linearization point to each operation

- $update_i(v)$
  - ✓ R_i.write(v) if present
  - ✓ Otherwise remove
- $scan_i()$
  - ✓ if complete – any point between identical collects
  - ✓ Otherwise remove

Build a sequential history S in the order of linearization points

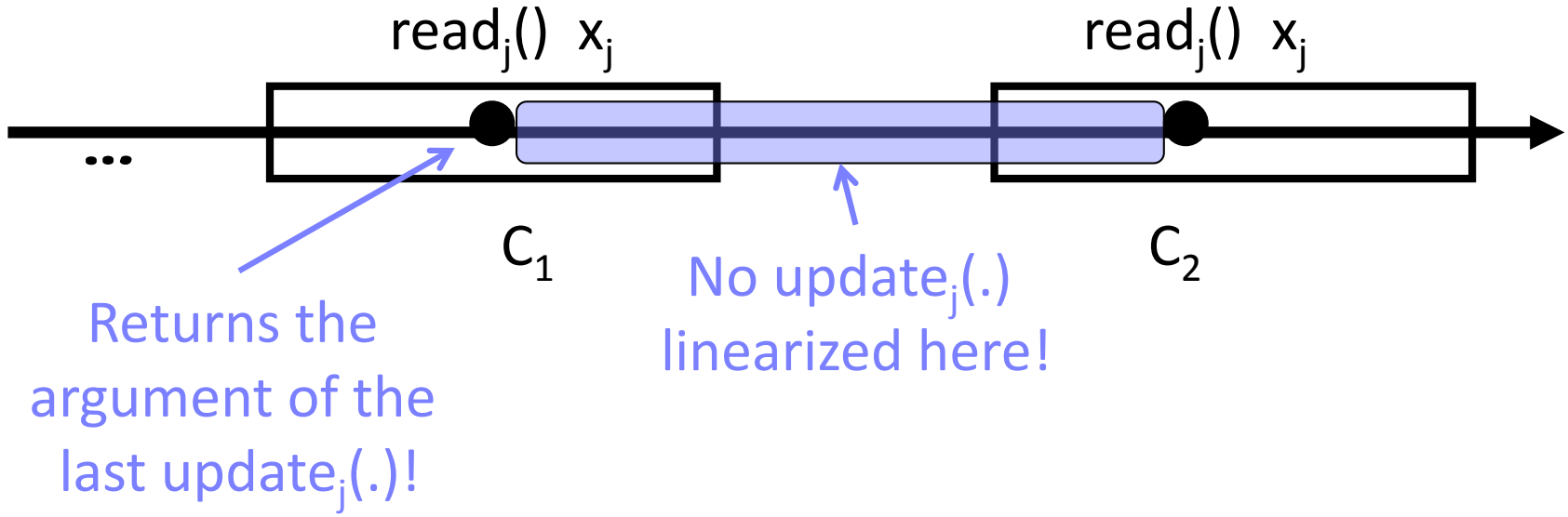


Correctness: linearizability

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

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

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



Correctness: obstruction-freedom

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

- There is a point after which no concurrent update can bring disagreement
- Eventually, two consecutive collects agree --- $scan_i()$ returns

Every operation that eventually encounters no contention terminates

General case: helping?

What if an update interferes with a scan?

- Make the update do the work!

upon scan()

$[x_1,\ldots,x_N]$:= collect($R_1,\ldots, R_N$)
$[y_1,\ldots,y_N]$:= collect($R_1,\ldots,R_N$)
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)

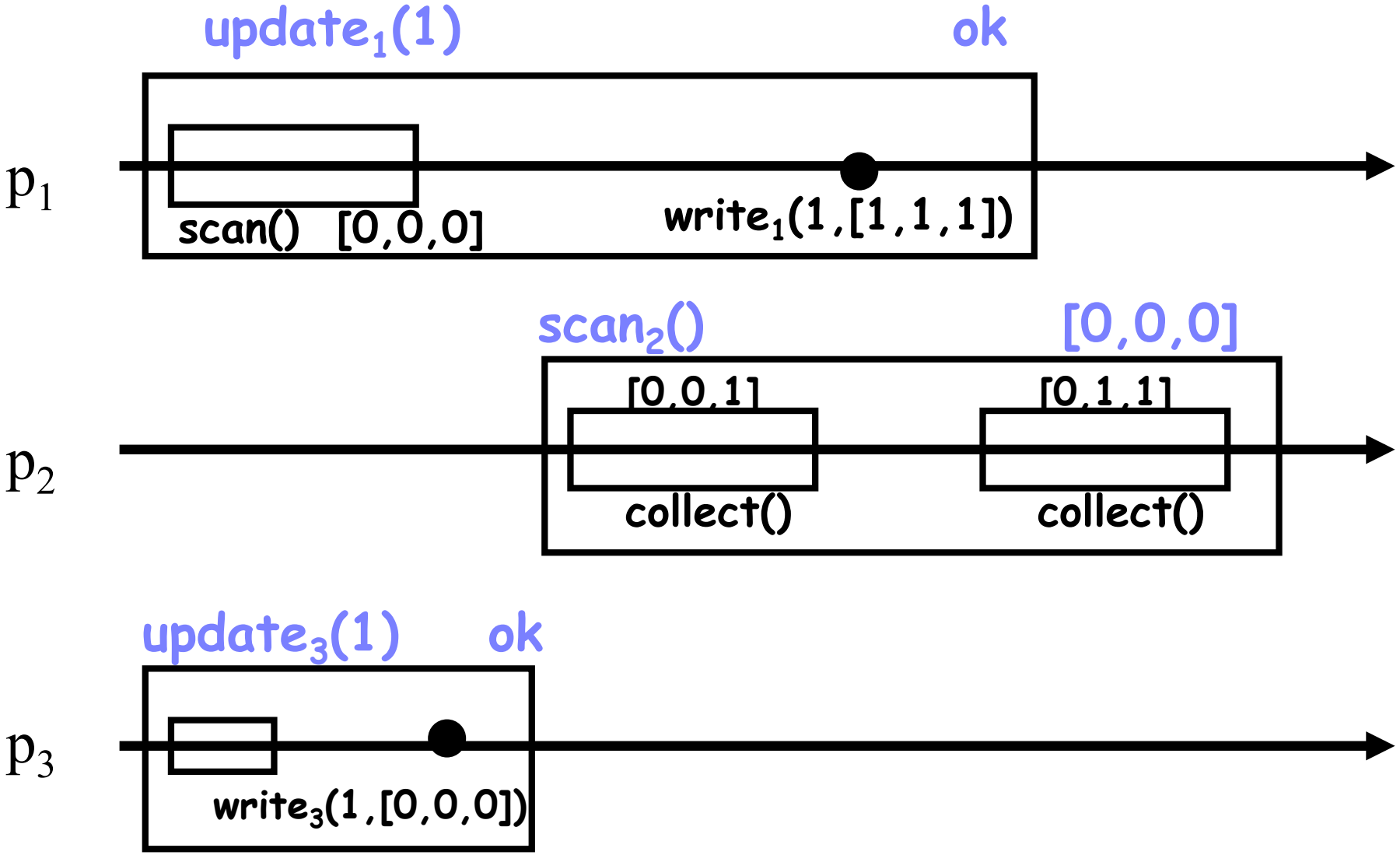
S := scan()

R_i.write(v,S)

If two collects differ – some update succeeded to scan!

Would this work?

Not that easy!



General case: wait-free atomic snapshot

upon scan()

$[x_1,\dots,x_N]$:= collect($R_1,\dots,R_N$)

while true do

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

  $[x_1,\dots,x_N]$:= collect($R_1,\dots,R_N$)

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

    return $[x_1,\dots,x_N]$

  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 := scan()

  R_i.write(v,S)

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

Correctness: wait-freedom

Claim 1 Every operation (update or scan) returns in $O(N^2)$ steps

Scan: does not return after a collect if a concurrent process moved and no process moved twice

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

Update: scan() + one more step

Correctness: wait-freedom

Claim 1 Every operation (update or scan) returns in $O(N^2)$ steps

Scan:

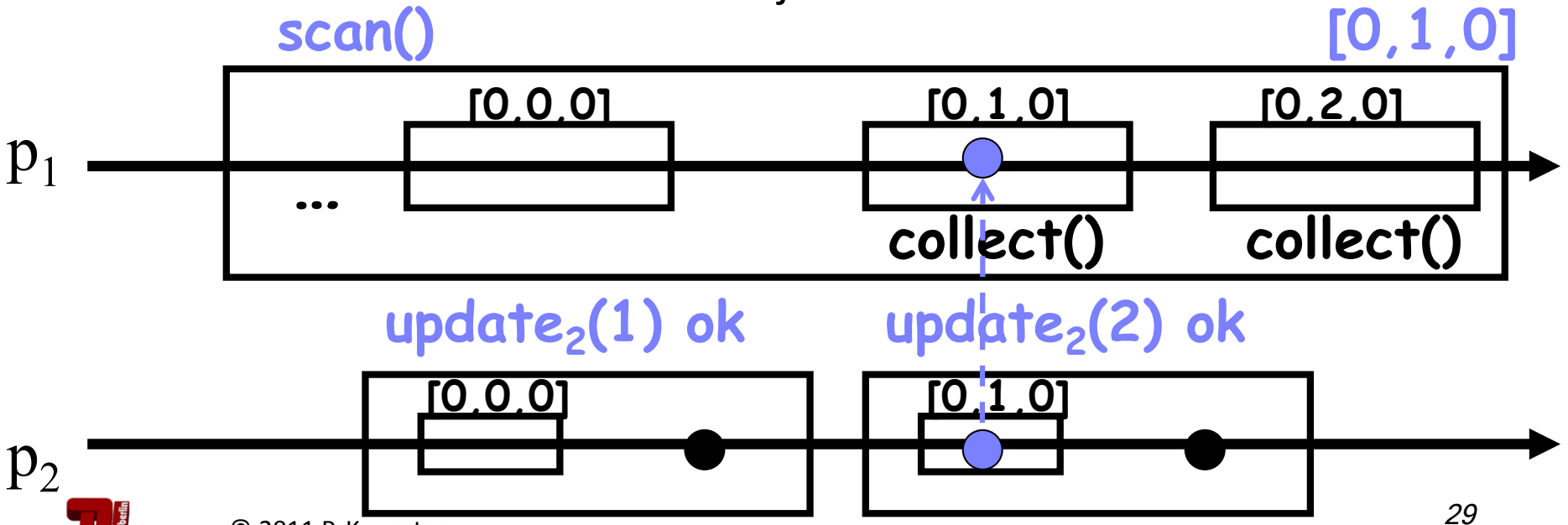
- Scan not return if a concurrent process moves and no process moves for the second time
- At most N-1 concurrent processes, after N+1 collects:
  - ✓Either at least two consecutive identical collects
  - ✓Or some process moved twice!

Correctness: linearization points

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

complete **scan()**

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



This class

- Atomic snapshot: reading multiple locations atomically
- Implementing an atomic bit

Binary transformations

How to implement an atomic bit?

The problem: implement a binary 1W1R atomic register (atomic bit) from binary 1W1R safe ones (safe bits).

A brute-force approach

- Binary 1W1R safe => binary 1W1R regular
- Binary 1W1R regular => multi-valued 1W1R regular
- Multi-valued 1W1R regular => multi-valued 1W1R atomic

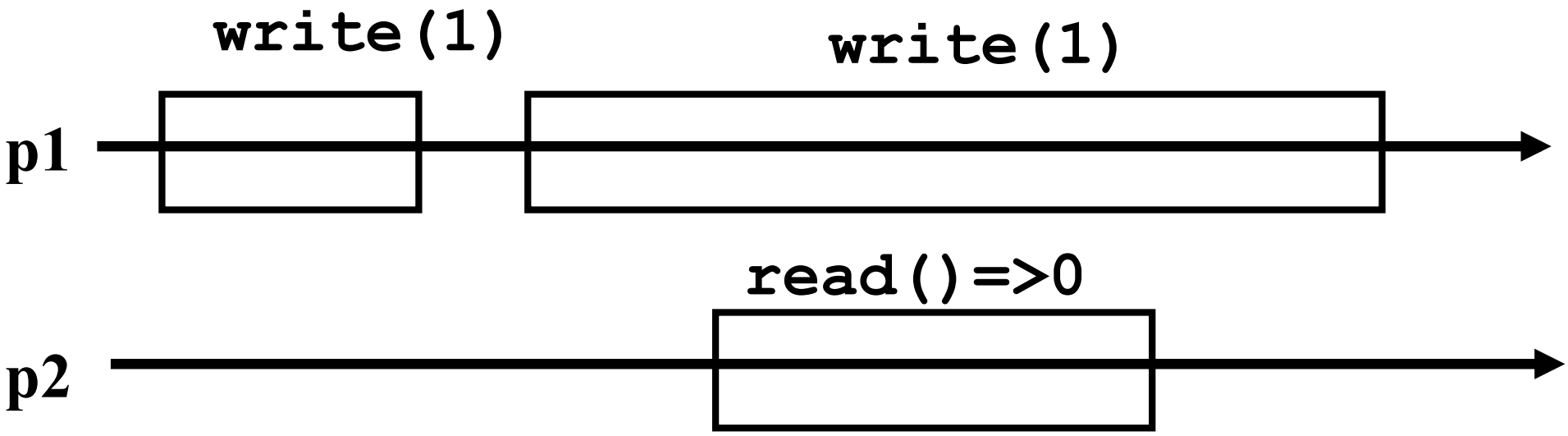
☹ Timestamps, $\Omega(N)$ time and $\Omega(N^2)$ space complexity

An optimal solution

- No timestamps
- Minimal number of bits, O(1)
- Minimal number of operations, O(1)

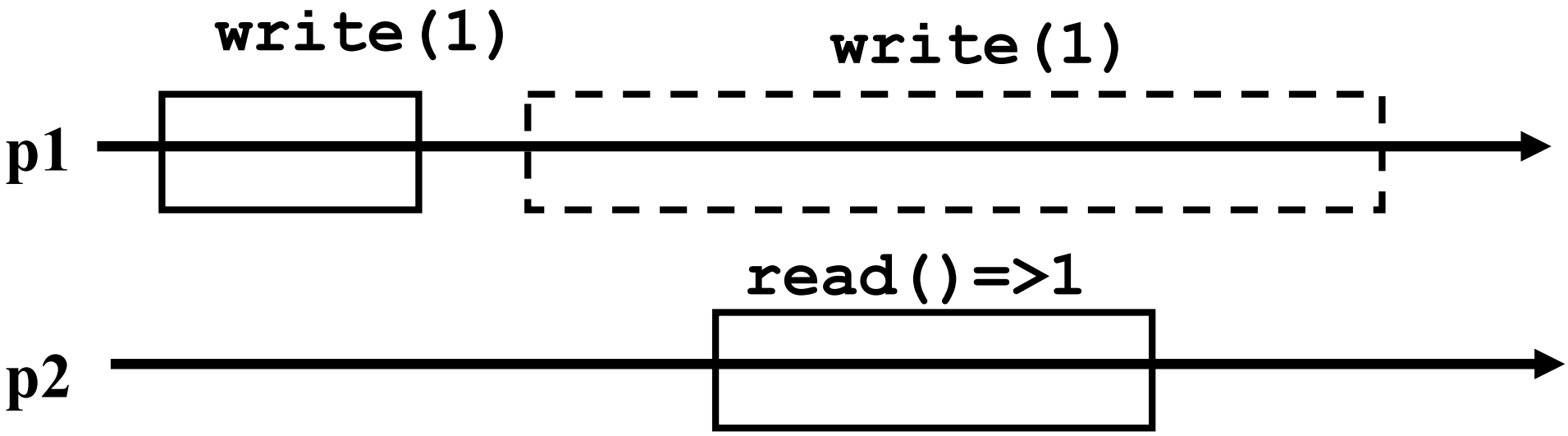
One bit solution? Safe bits

- Safe bit can return a bogus value in presence of concurrency



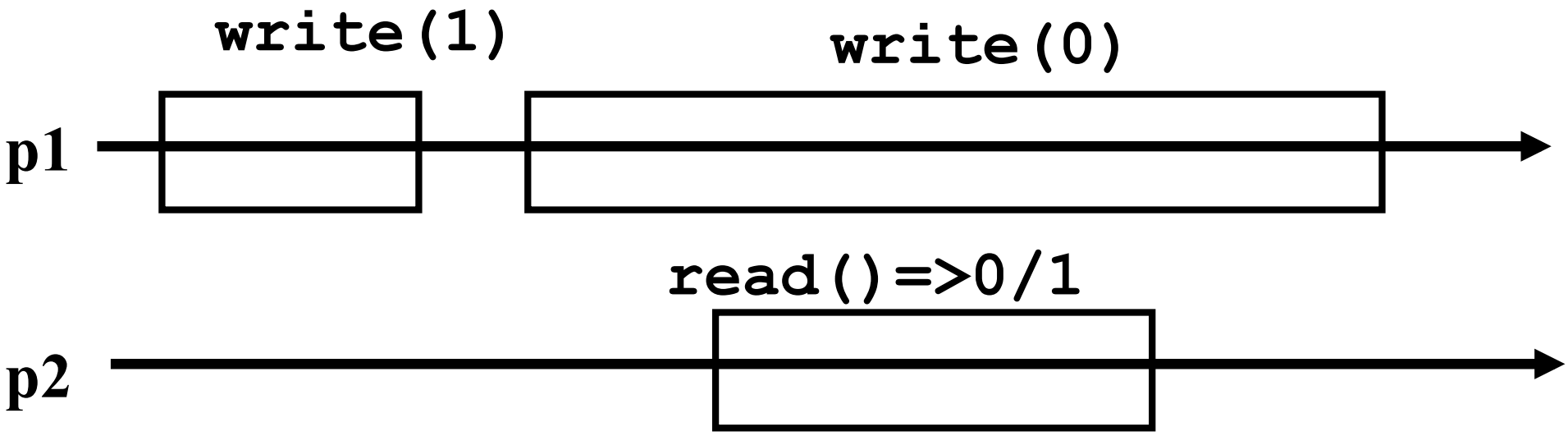
One bit solution? Regular bits

- Safe bit => regular bit: the writer is allowed only to *change* the value



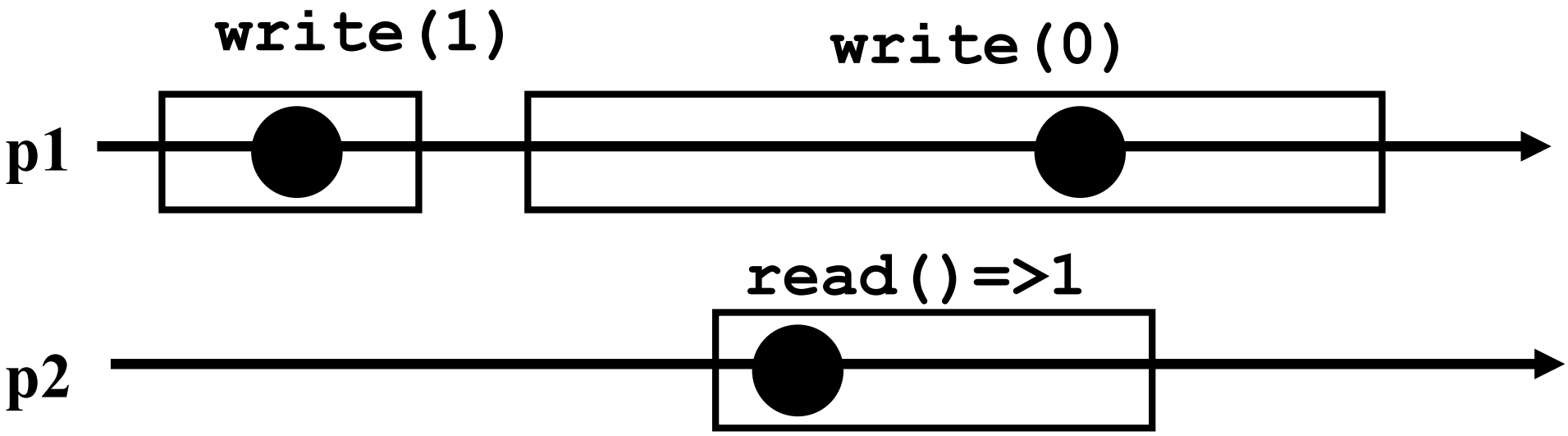
One bit solution? Regular bits

- Safe bit => regular bit: the writer is allowed only to *change* the value



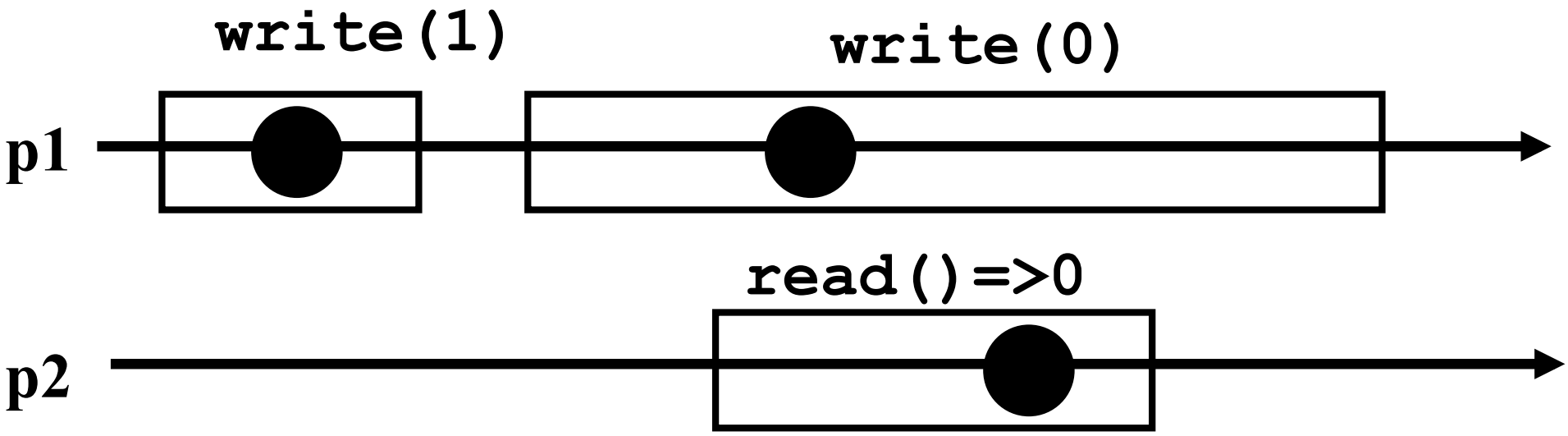
One bit solution? Regular bits

- Safe bit => regular bit: the writer is allowed only to *change* the value



One bit solution? Regular bits

- Safe bit => regular bit: the writer is allowed only to *change* the value



One bit solution? Regular bits

Writer:

change V

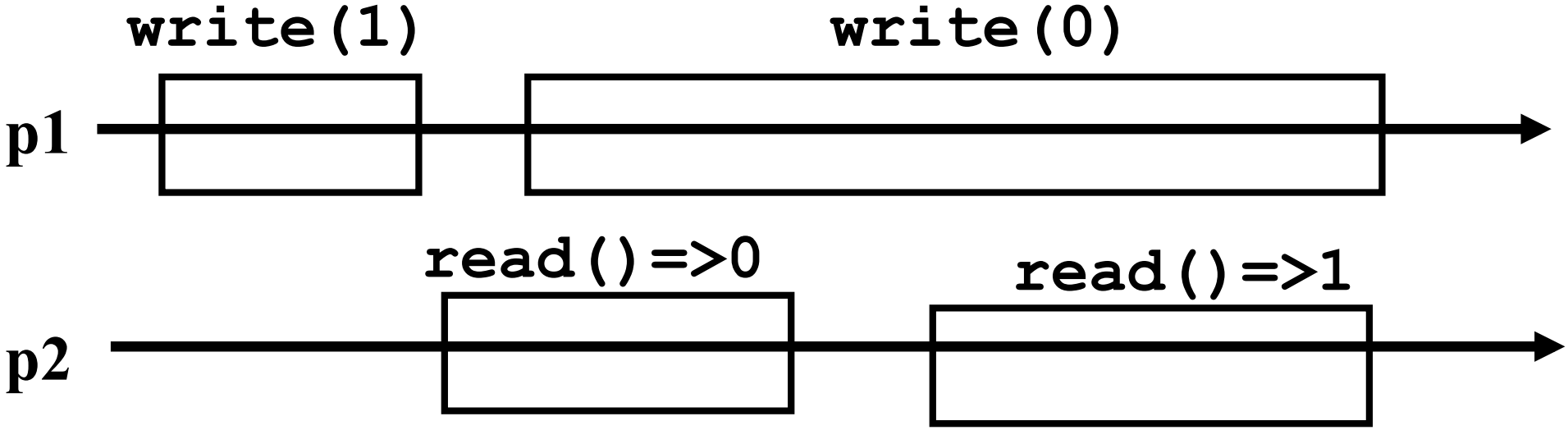
Reader:

v := V

return v

One bit solution? Atomic?

New/old inversion: the reader is not able to distinguish a new value from an old one.



Two bit solution?

Writer protocol

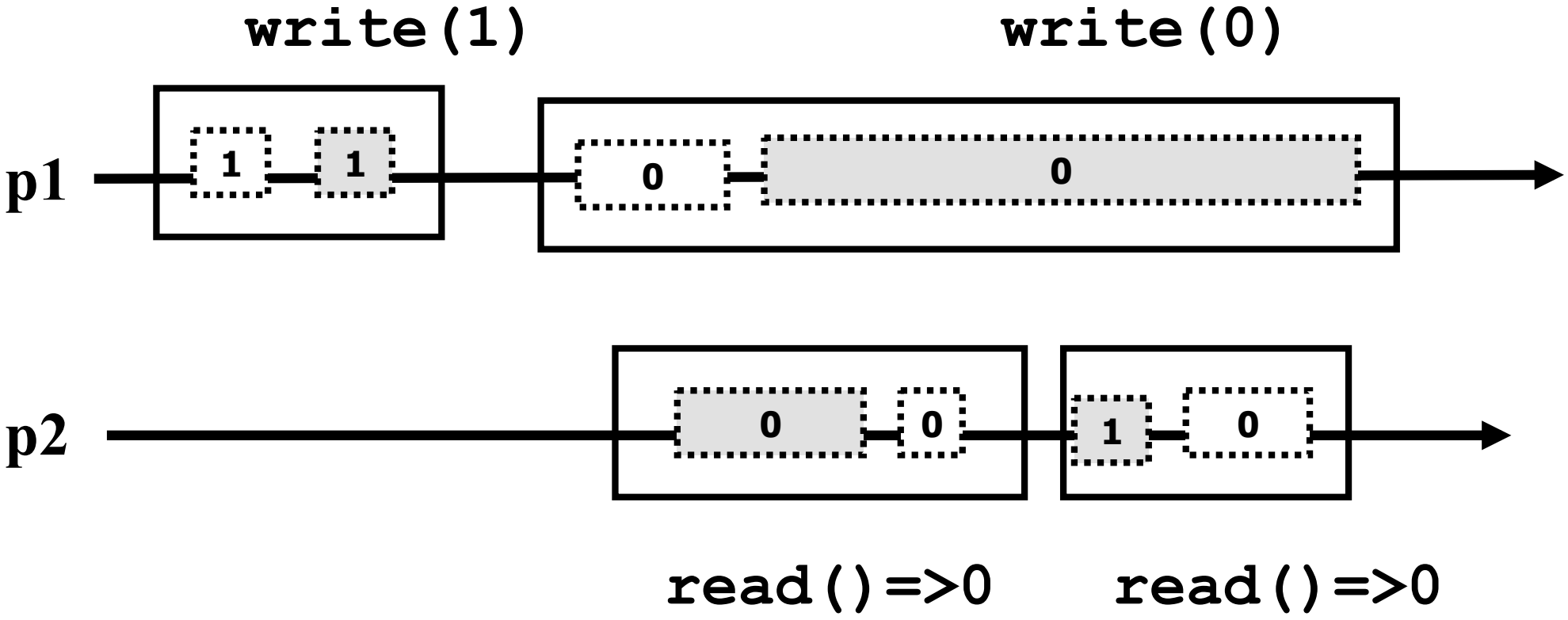
change V1

change V2

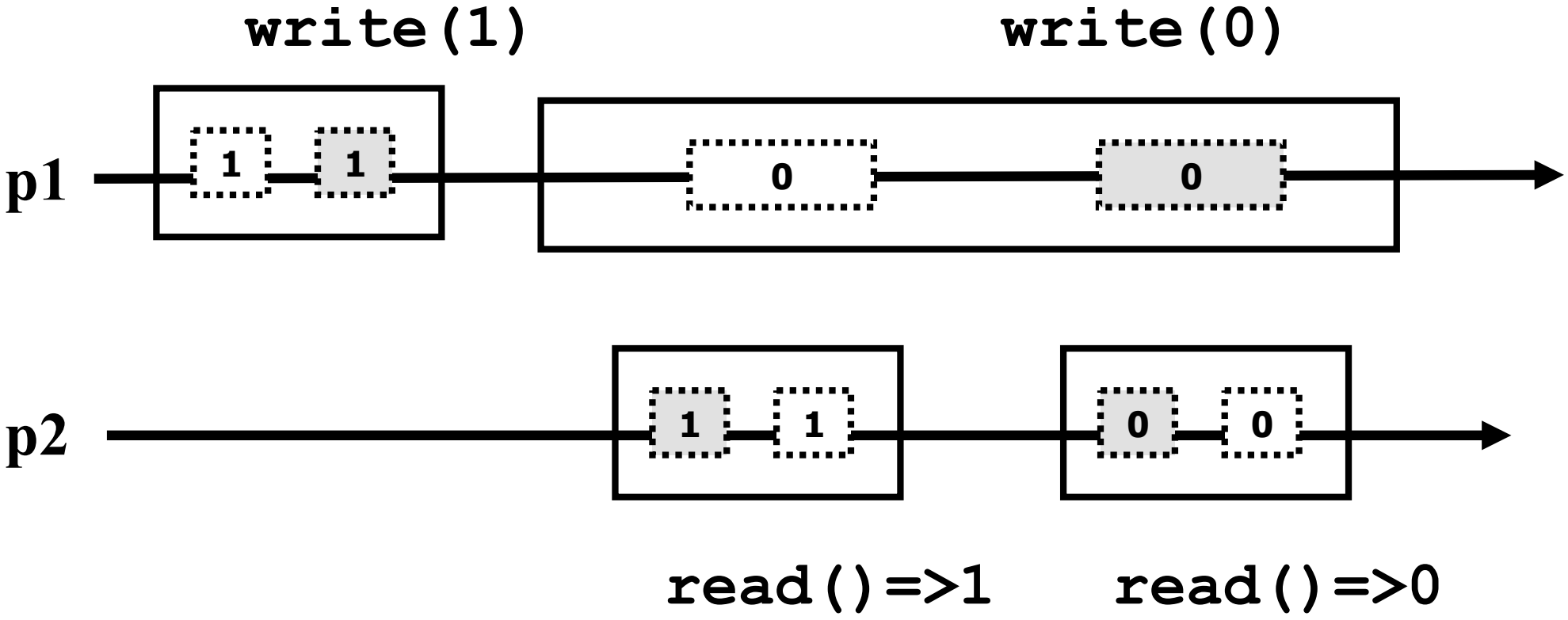
Reader protocol

```
x := V2
y := V1
If x=y then
    v:=x
    return v
else
    return v
```

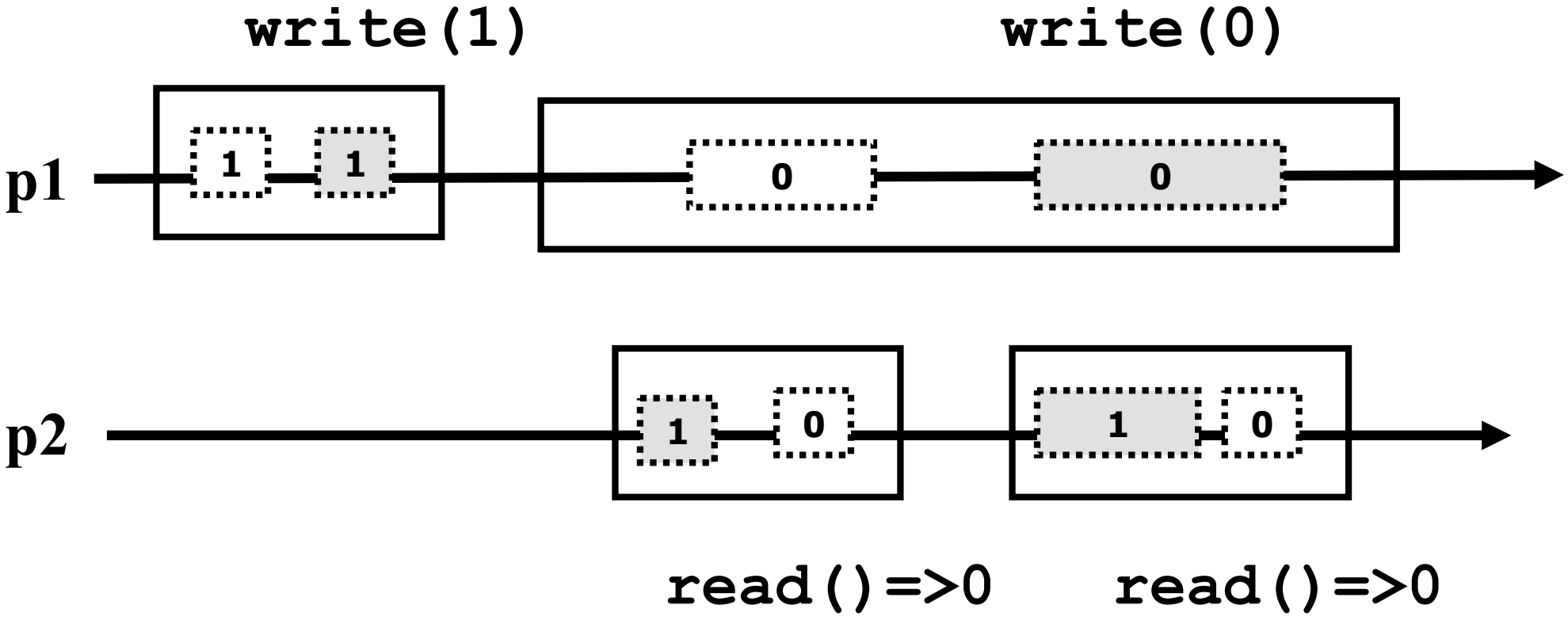
Two bit solution?



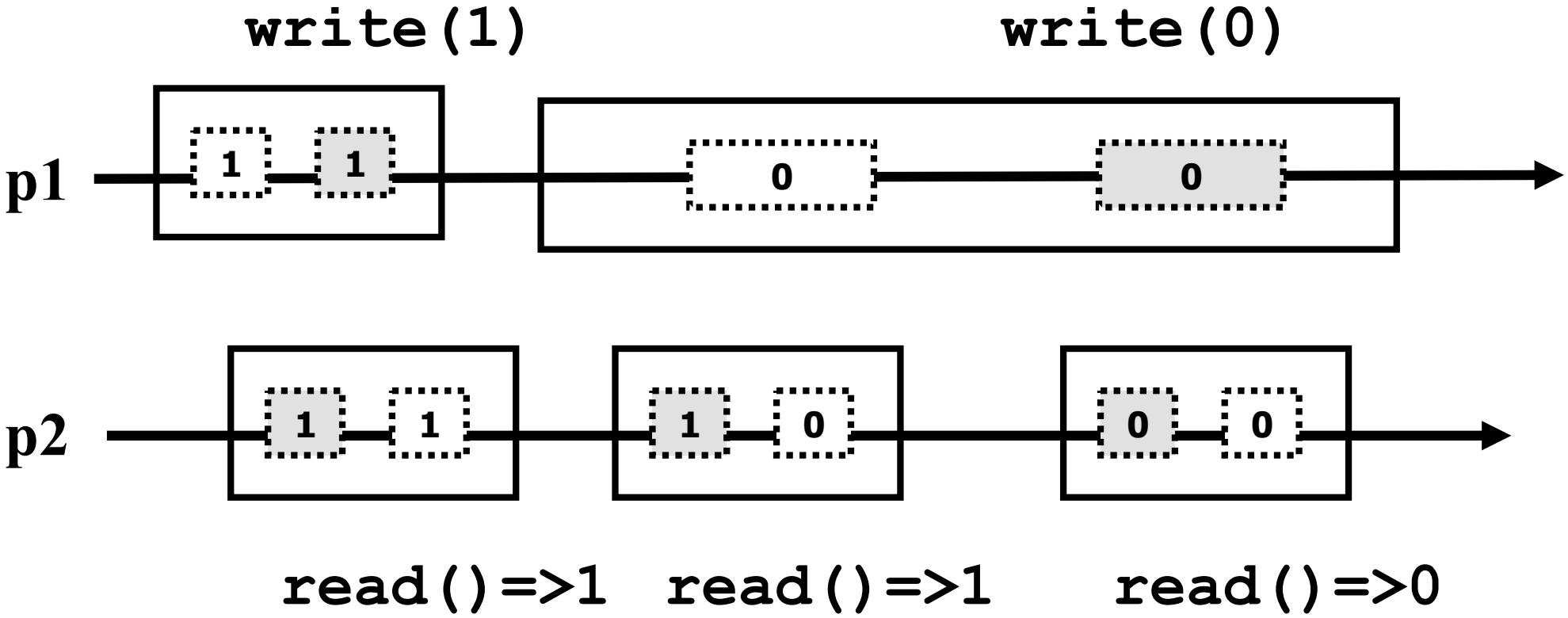
Two bit solution?



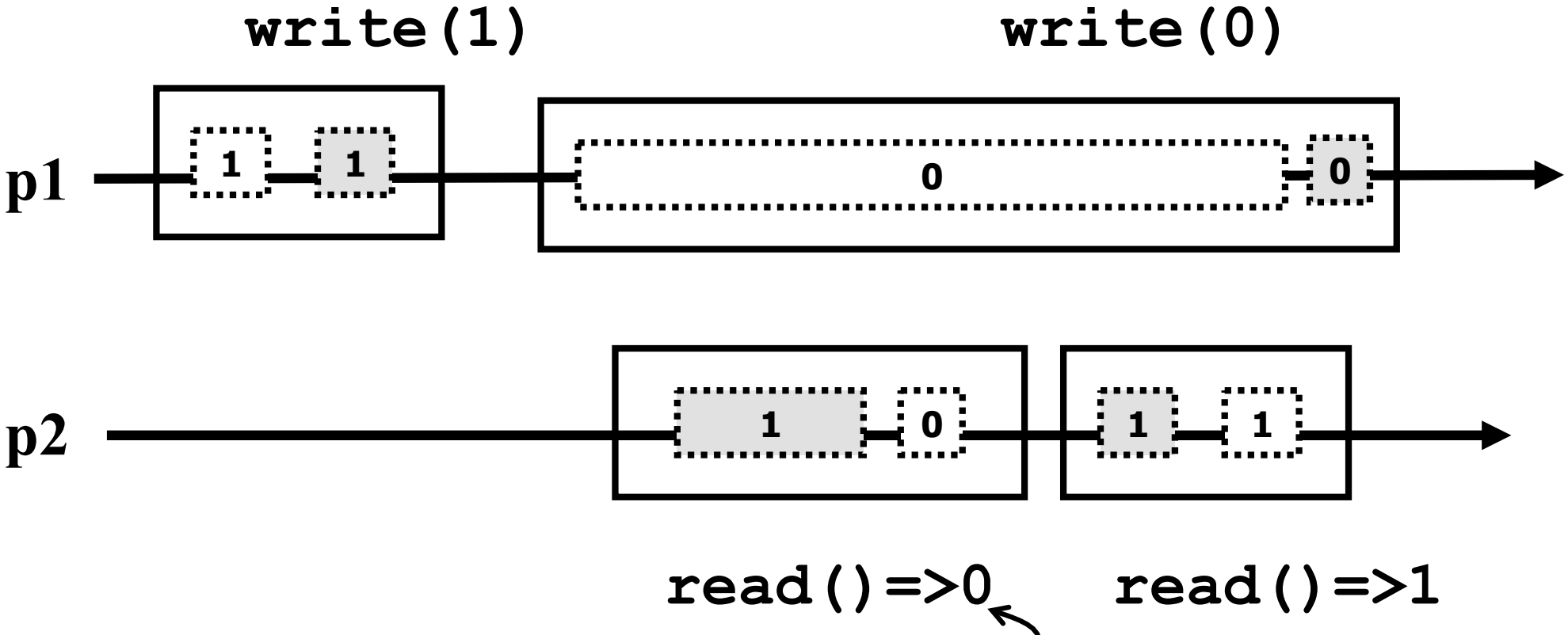
Two bit solution?



One more read?

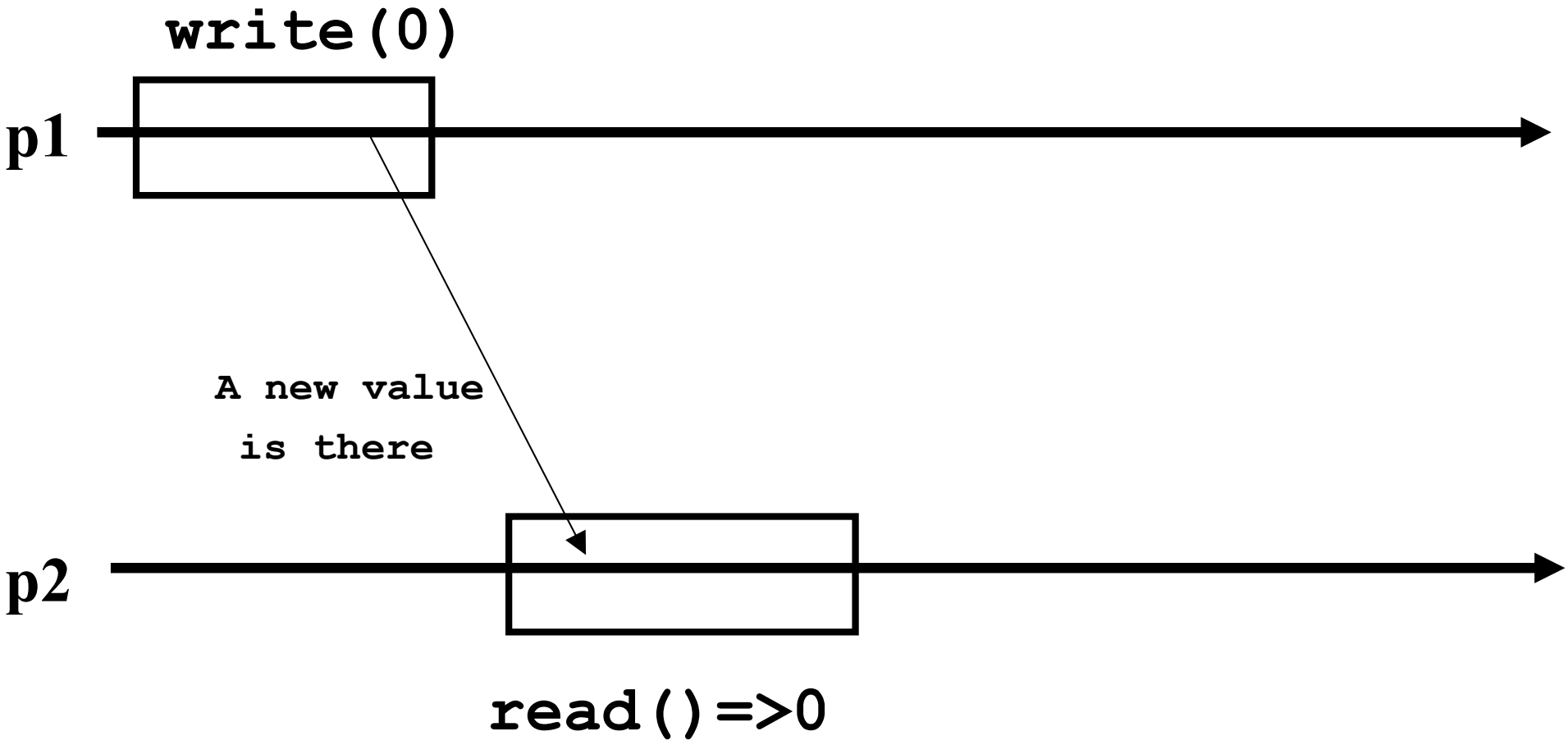


Two bit solution?



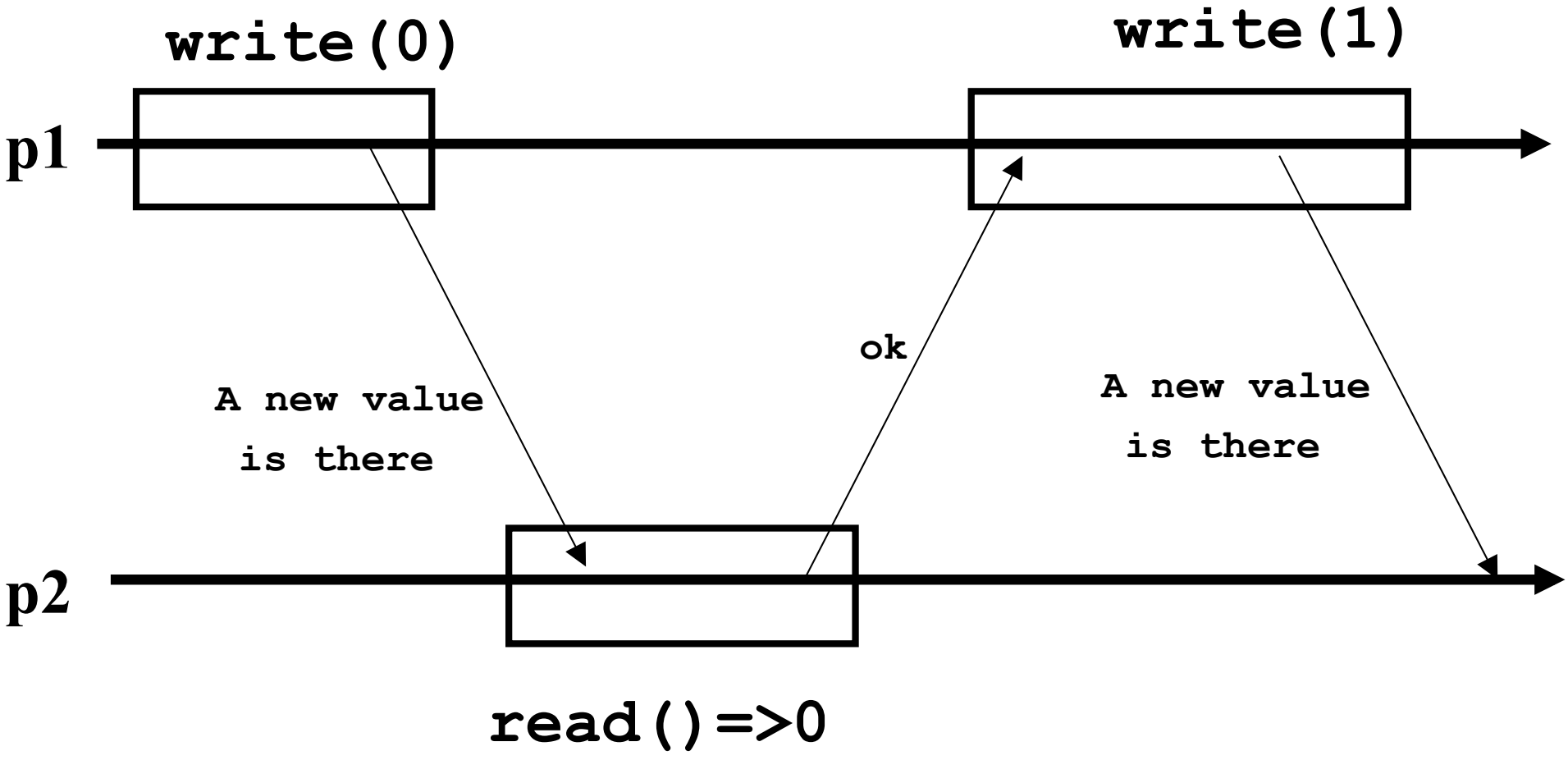
The reader must write!

(And the writer must read)



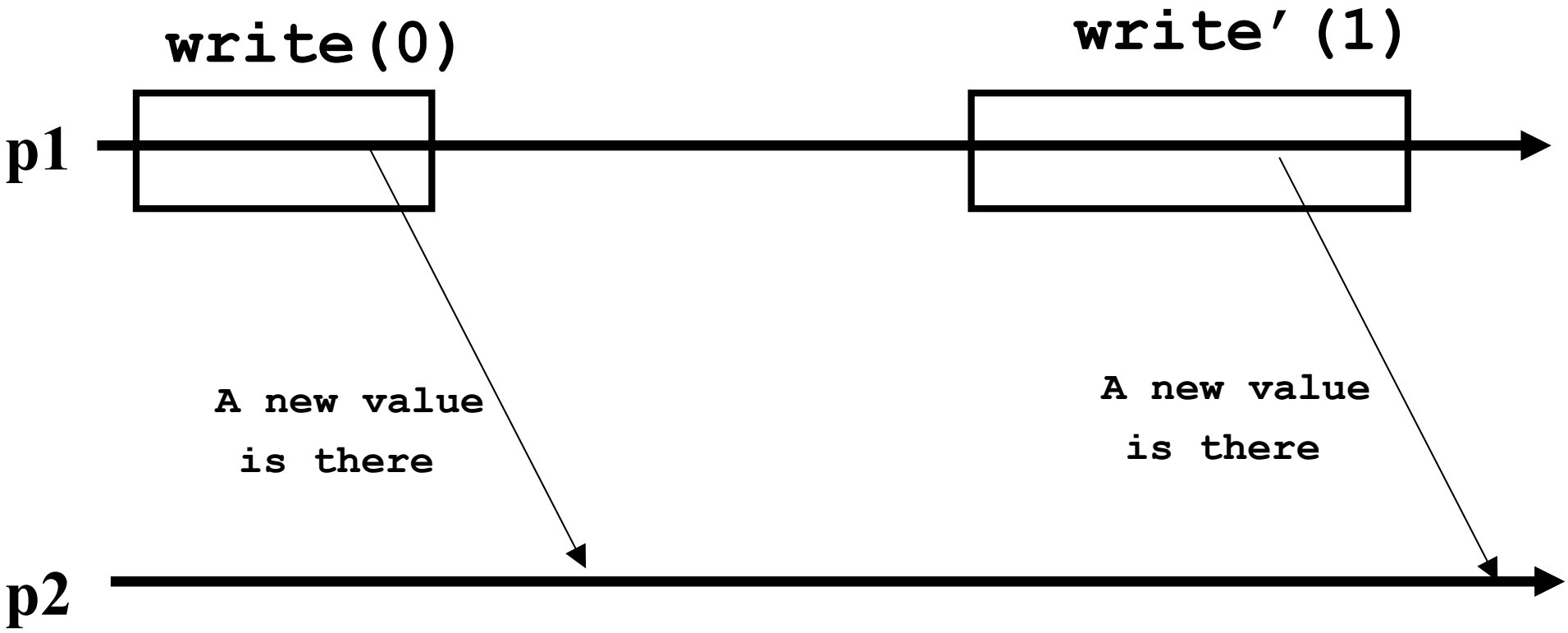
The reader must write!

(And the writer must read)



The reader must write!

(And the writer must read)



The write procedure should differ if the previous write has not been ack'ed

Handshake?

Writer protocol

change V

if W=R then change W

Reader protocol

if W=R then return v

v:= V

if W≠R then change R

return v

W≠R: the new value is there

W=R: the reader got it

Too simple!

Counter-example: a read operation R1 and a write operation W1

- R1 sees W≠R (a new value is there)
- R1 reads 0 in V
- W1 changes V from 0 to 1 (even a newer value)
- W1 sees R≠W (the new value is still not acknowledged) and returns
- R1 sees R≠W, changes R and returns 0

Any subsequent read sees R=W and returns 0 (the old value!)

Final solution [Tromp, 1989]

Writer protocol

change V

if W=R then
  change W

Reader protocol

(1) if W=R then return v

(2) x := V

(3) if W≠R then change R

(4) v :=V

(5) if W=R then return v

(6) v := V

(7) return x

- Next class: homework set 1!

- Class of May 10: shift to Friday May 13 (Auditorium 1, 12pm)?