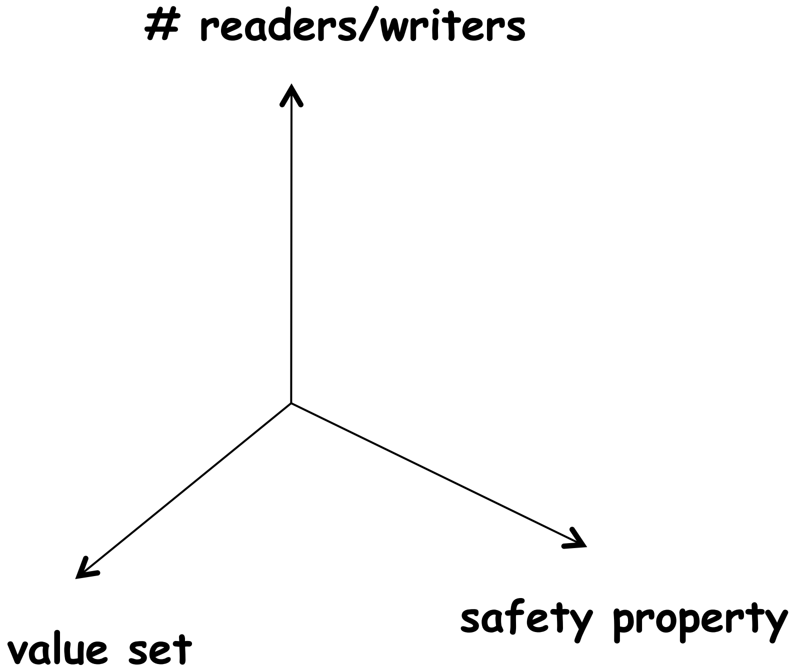
Atomic and immediate snapshots

MPRI, P1, 2018

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 $(j=1,\ldots,N)$ 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 1WNR *atomic* register R_i := 0

upon snapshot()

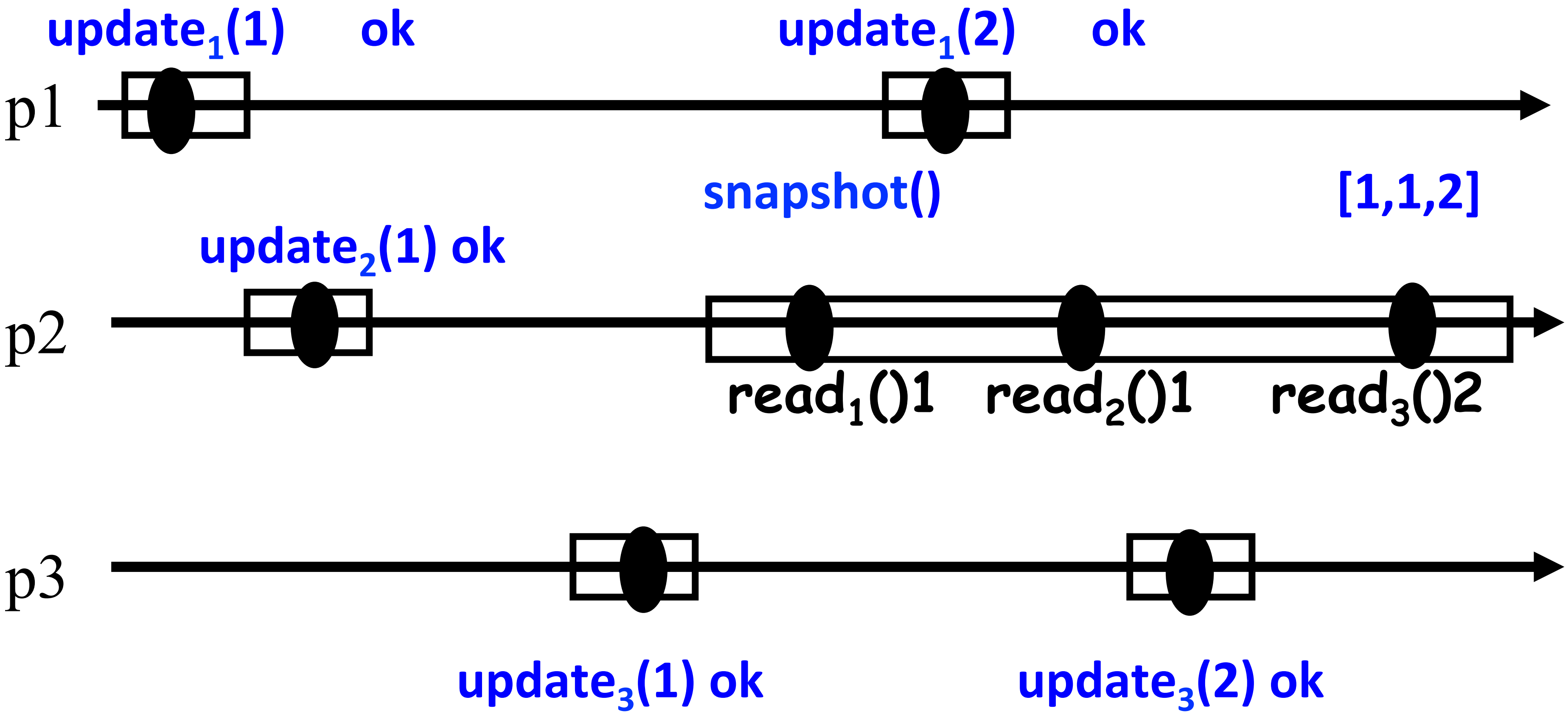
$[x_1,\ldots,x_N]$:= scan$(R_1,\ldots,R_N)$ /*read $R_1,\ldots R_N$*/

return $[x_1,\ldots,x_N]$

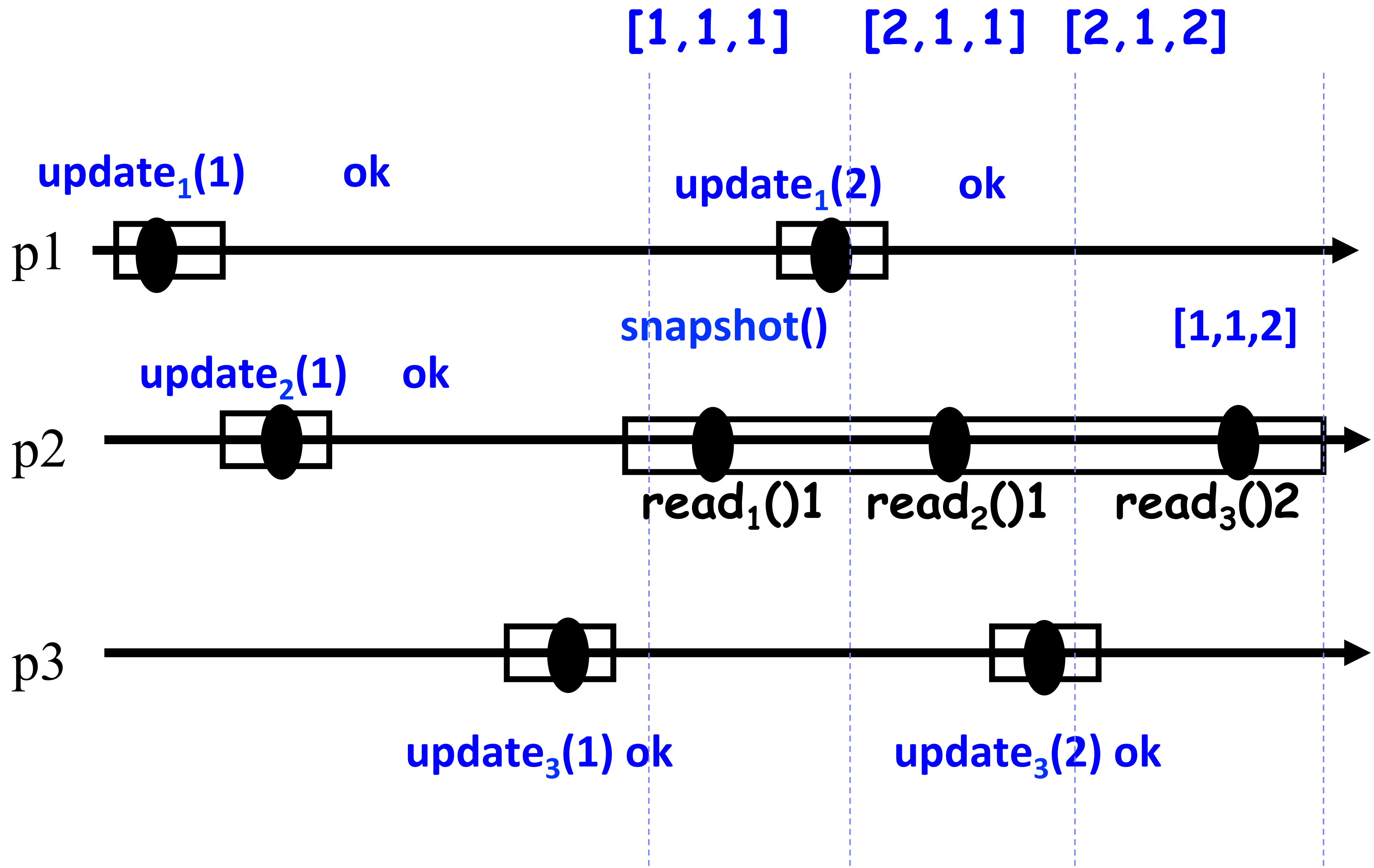
upon update$_i$(v)

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 values, including the initial one, are unique, e.g., equipped with a sequence number)

Initially:

shared 1W1R atomic register $R_i := 0$

upon snapshot()

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

repeat

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

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

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

return $[x_1,\ldots,x_N]$

upon update$_i$(v)

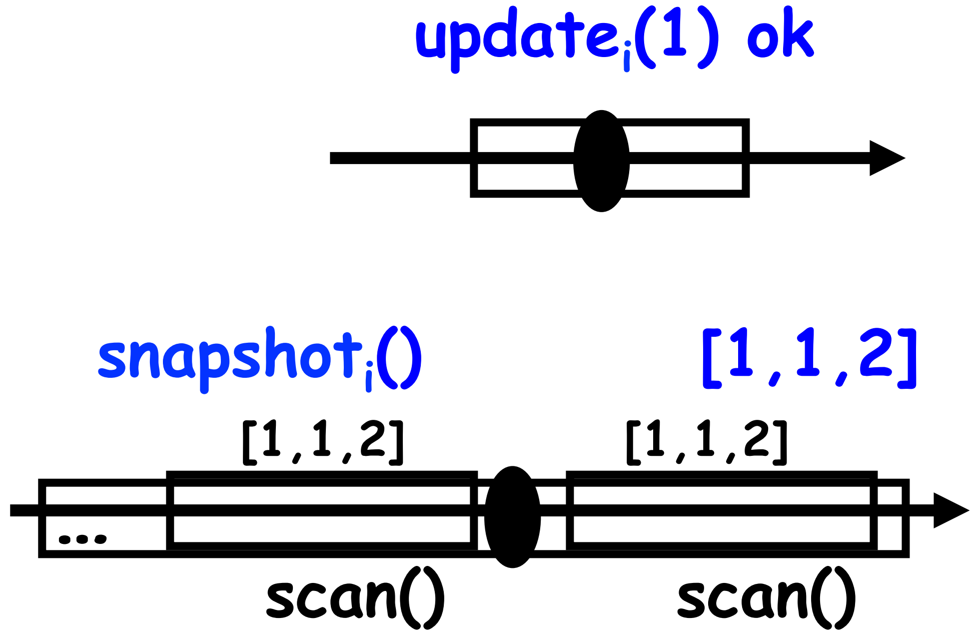
R_i.write(v)

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

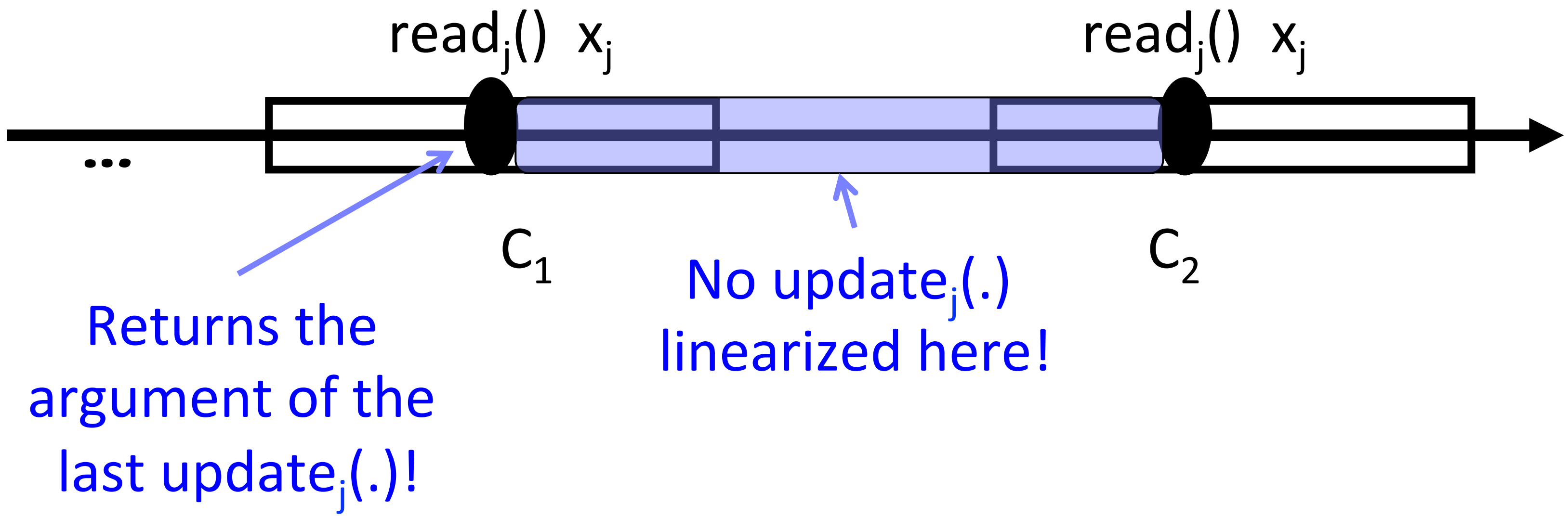


Correctness: linearizability

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()

$[x_1,\ldots,x_N] := \text{scan}(R_1,\ldots, R_N)$
$[y_1,\ldots,y_N] := \text{scan}(R_1,\ldots,R_N)$
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

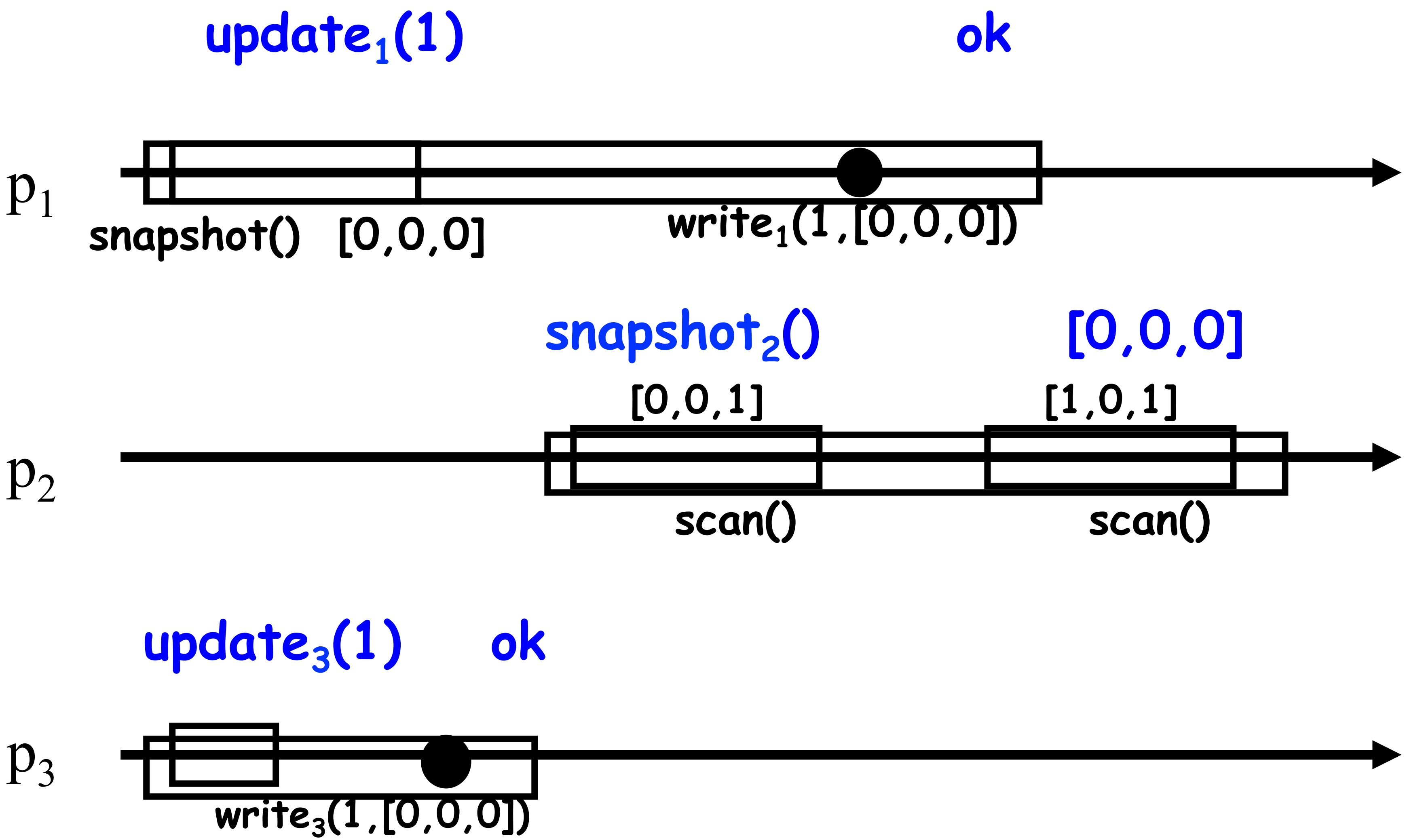
upon update$_i$(v)

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

If two scans differ – some update succeeded!

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!

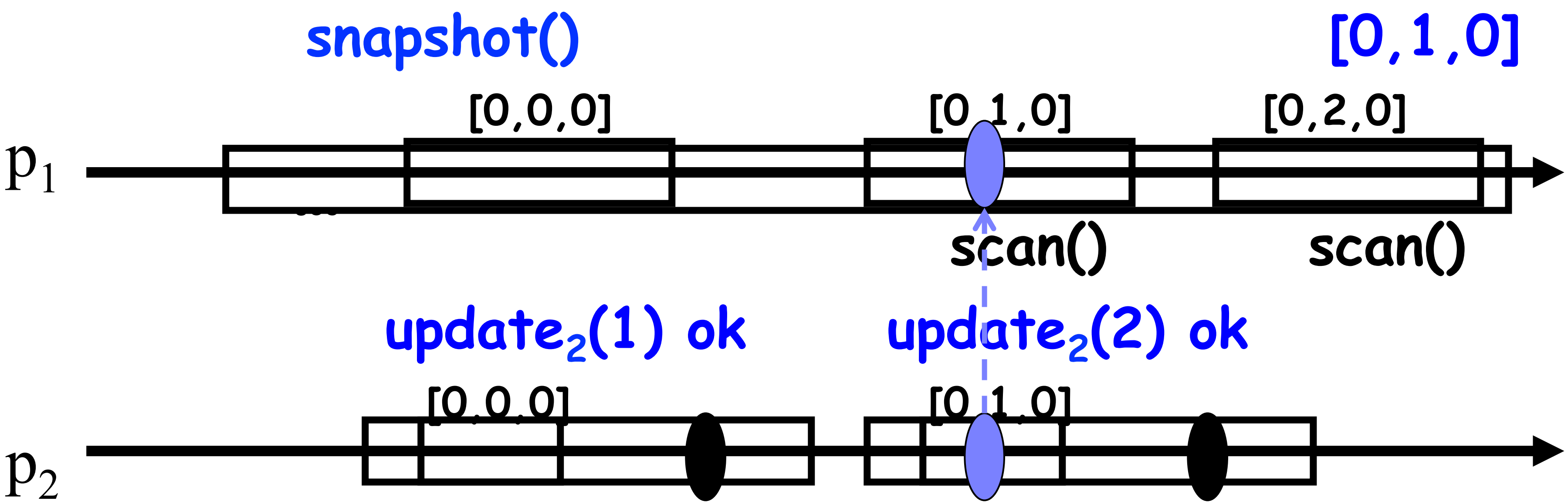
update: snapshot() + one more step

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()

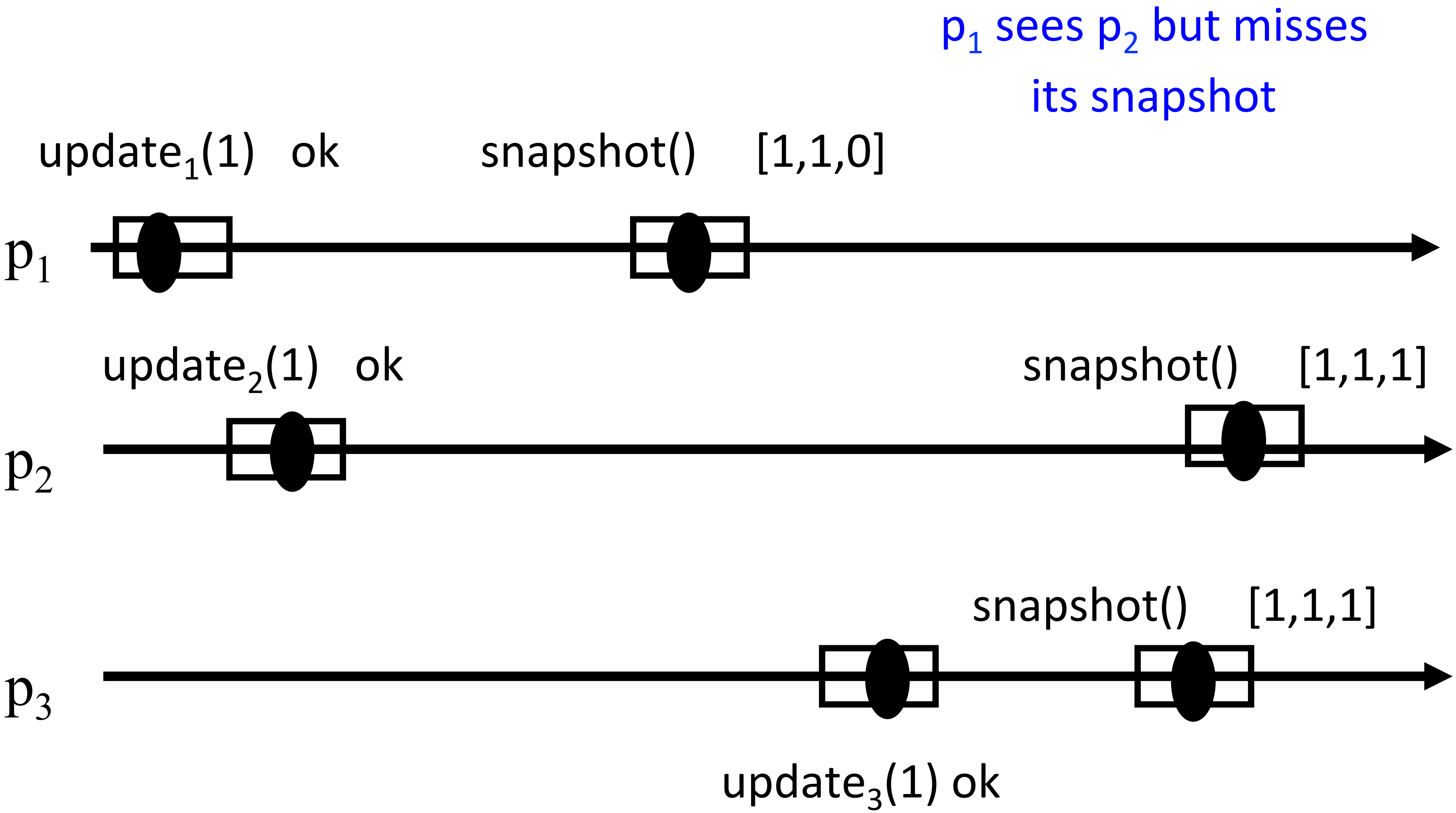
$S_i = S_i[1],\ldots,S_i[N]$

(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):

  $update_i(v_i)$

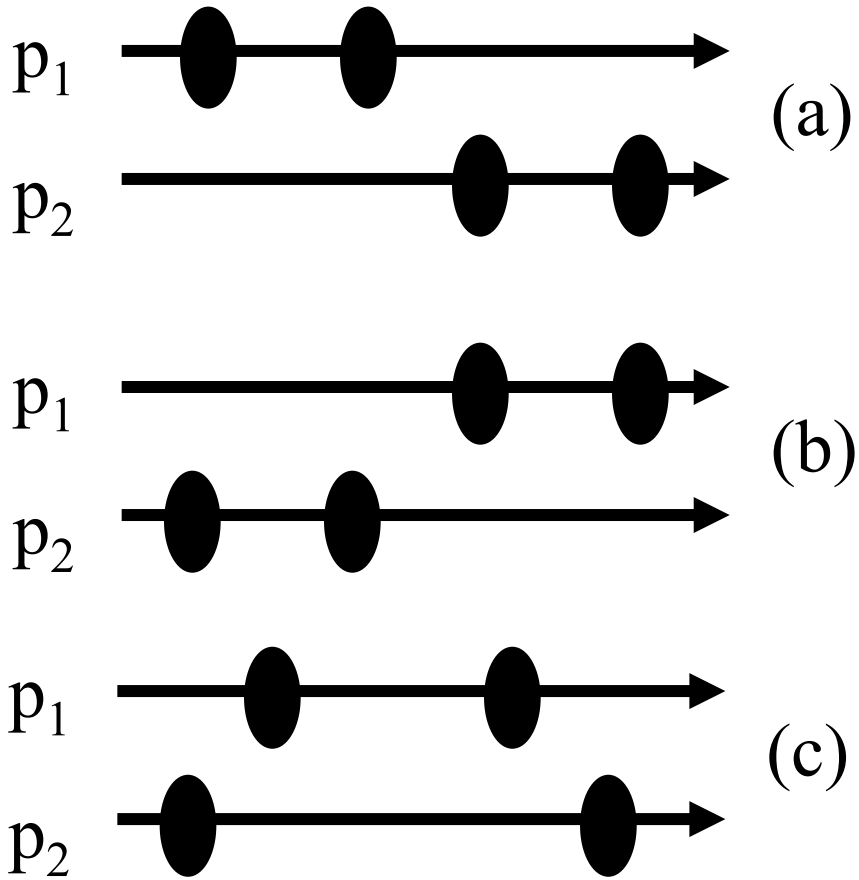
  S_i := snapshot()

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 1: atomic snapshots

1. 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
2. Show that the atomic snapshot is subject to the ABA problem (affecting correctness) in case the written values are not unique

One-shot atomic snapshot (AS)

Each process p_i:

  $update_i(v_i)$

  S_i := snapshot()

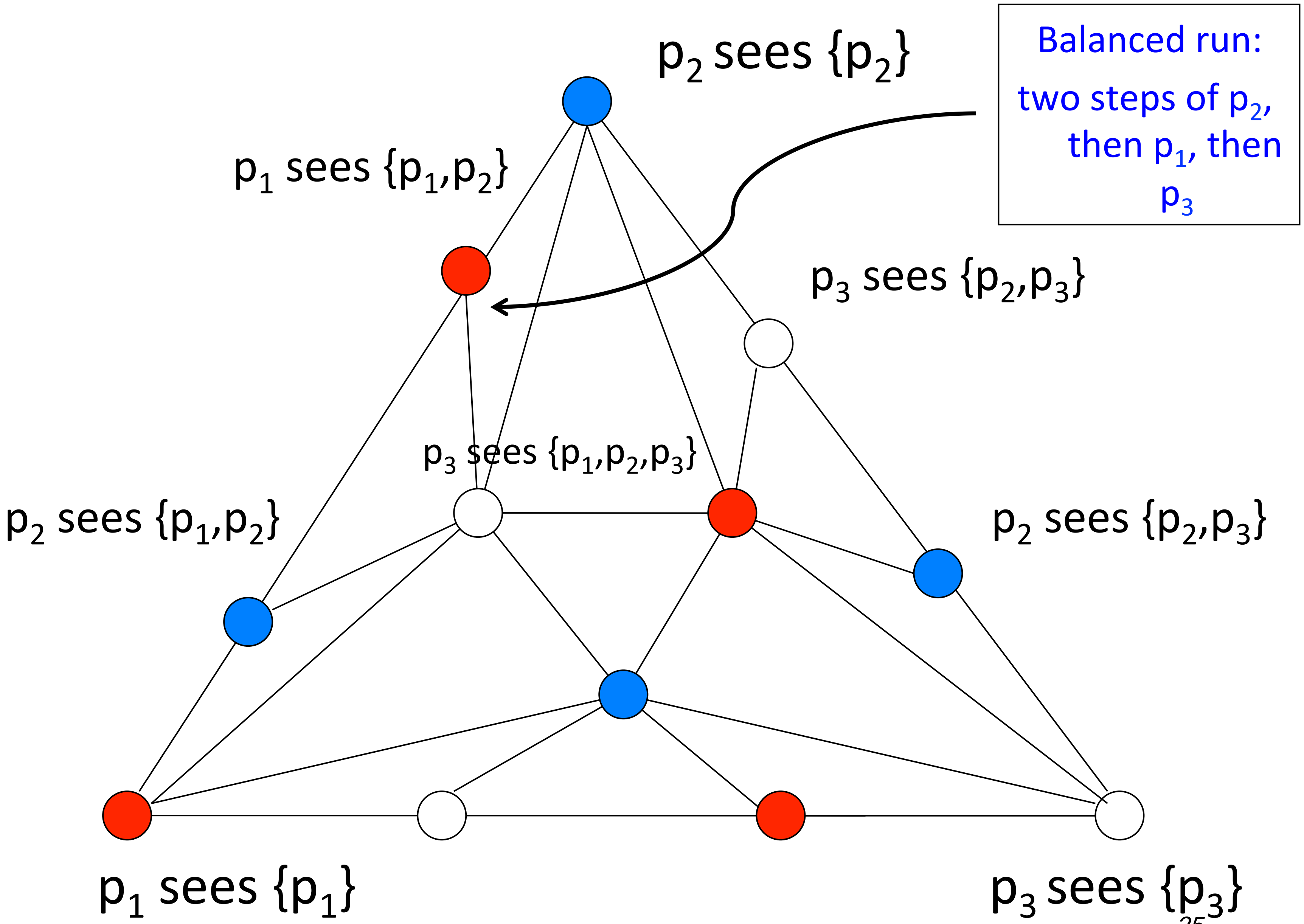
$S_i = S_i[1],\ldots,S_i[N]$

(one position per process)

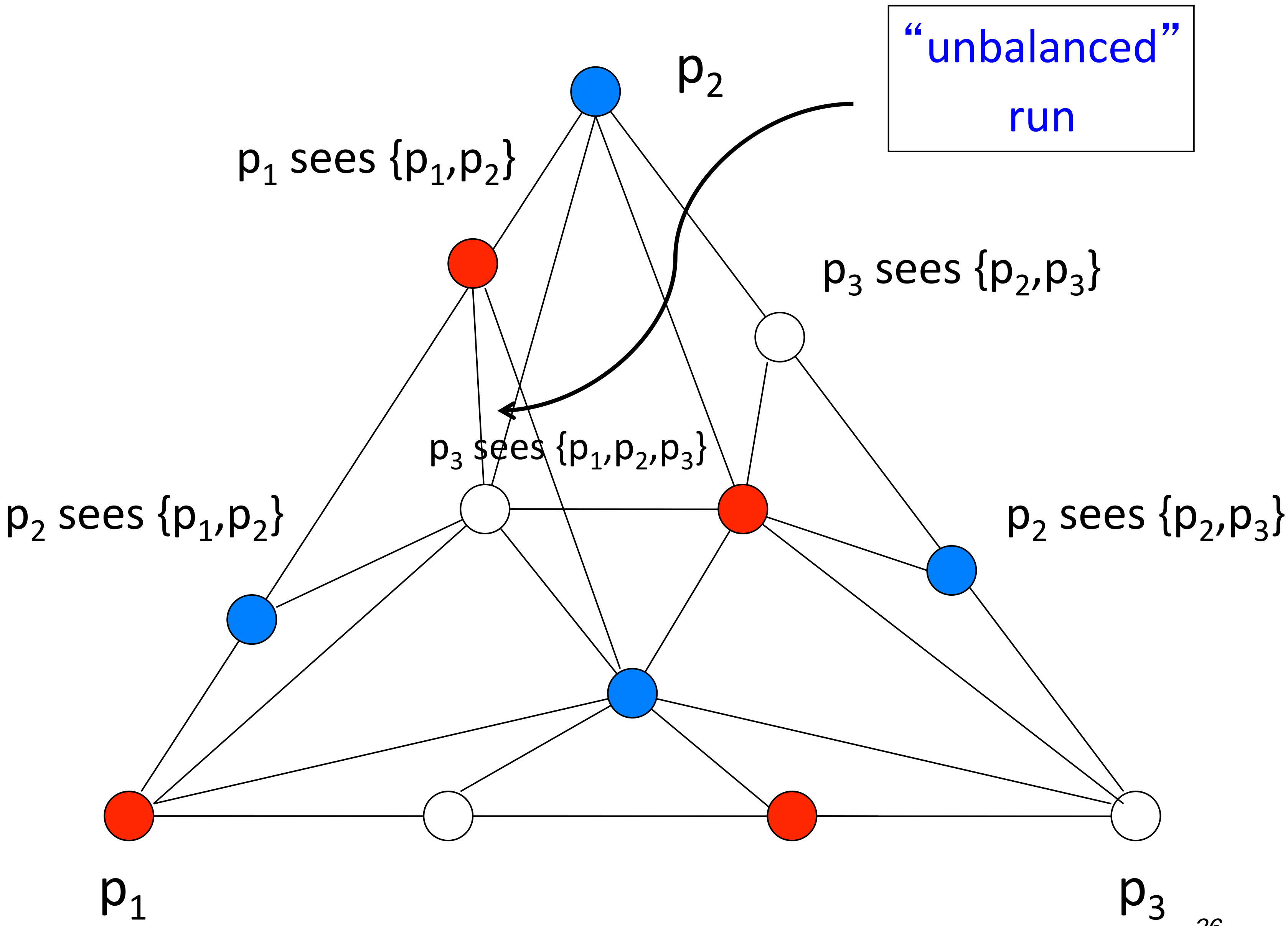
Vectors S_i satisfy:

- 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

Topological representation: one-shot AS



Topological representation: one-shot AS



One-shot *immediate* snapshot (IS)

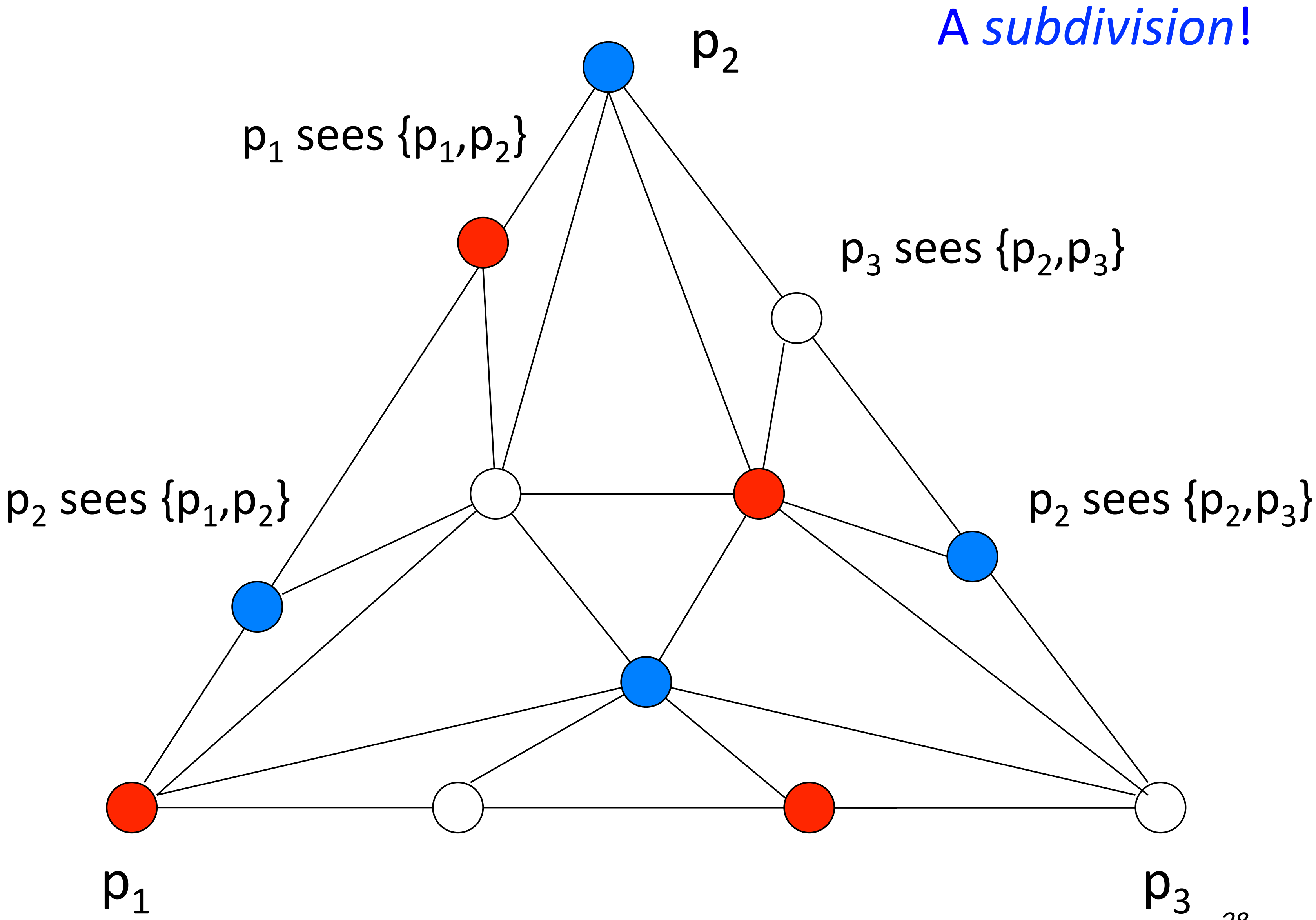
One operation:
WriteRead(v)

Each process p_i:
S_i := $WriteRead_i(v_i)$

Vectors $S_1,\ldots,S_N$ 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
- Immediacy: for all i and j: if v_i is in S_j, then is S_i is a subset of S_j

Topological representation: one-shot IS



IS is equivalent to AS (one-shot)

- IS is a restriction of one-shot AS => IS is stronger than one-shot AS
  - ✓Every run of IS is a run of one-shot AS
- Show that a few (one-shot) AS objects can be used to implements IS
  - ✓One-shot ReadWrite() can be implemented using a series of update and snapshot operations

IS from AS

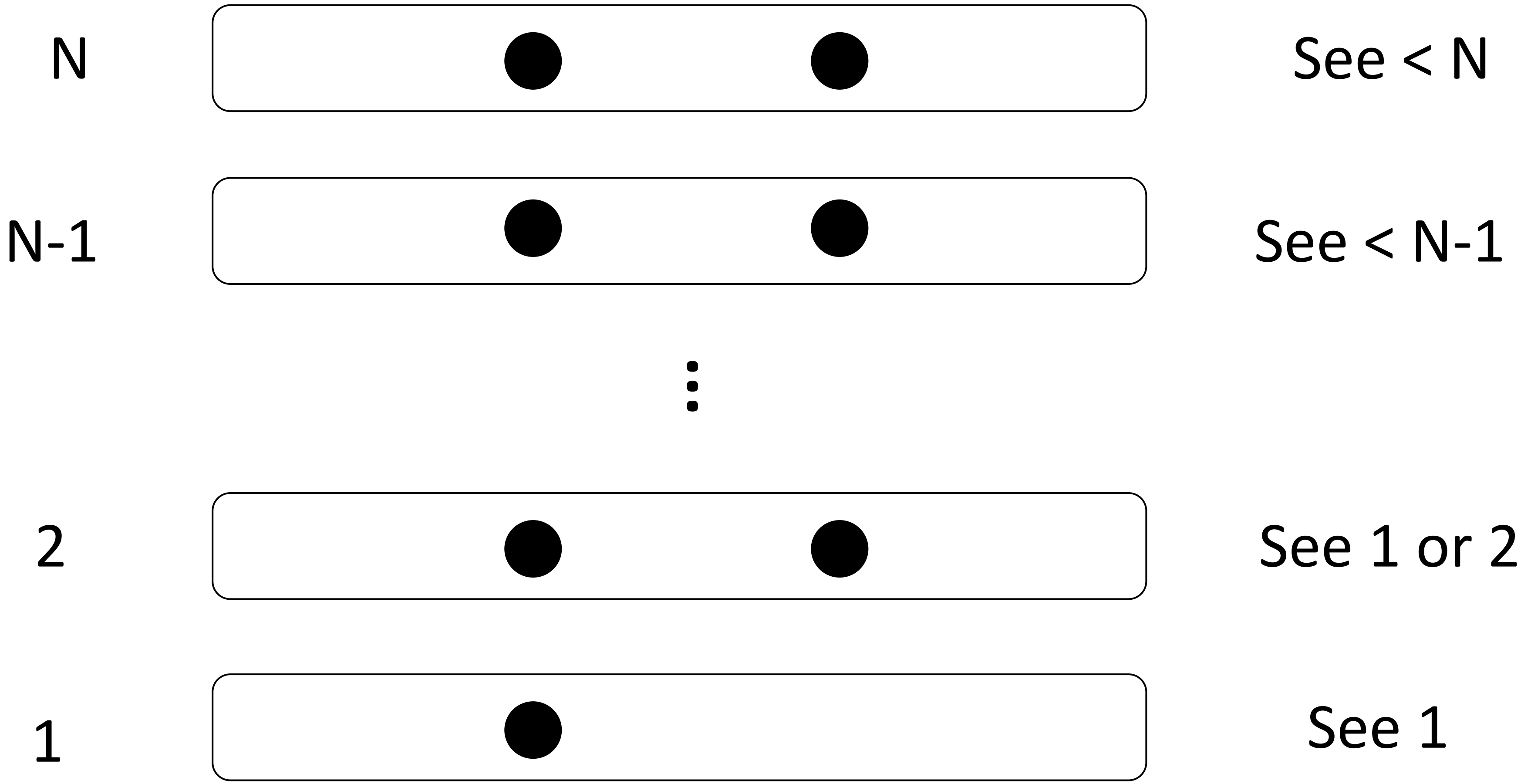
shared variables:

$A_1,\dots,A_N$ – atomic snapshot objects, initially [T,…,T]

Upon WriteRead$_i$(v_i)

```
r := N+1
while true do
        r := r-1            // drop to the lower level
        A_r.update_i(v_i)
        S :=  A_r.snapshot()
        if |S|=r then       // |S| is the number of non-T values in S
            return S
```

Drop levels: two processes, N>3



Correctness

The outcome of the algorithm satisfies Self-Inclusion, Snapshot, and Immediacy

- By induction on N: for all N>1, if the algorithm is correct for N-1, then it is correct for N

- Base case N=1: trivial

Correctness, contd.

- Suppose the algorithm is correct for N-1 processes
- N processes come to level N
  - ✓At most N-1 go to level N-1 or lower
  - ✓(At least one process returns in level N)
  - ✓Why?
- Self-inclusion, Containment and Immediacy hold for all processes that return in levels N-1 or lower
- The processes returning at level N return all N values
  - ✓The properties hold for all N processes! Why?

Iterated Immediate Snapshot (IIS)

Shared variables:

IS_1, IS_2, IS_3,…   // a series of one-shot IS

Each process p_i with input v_i:

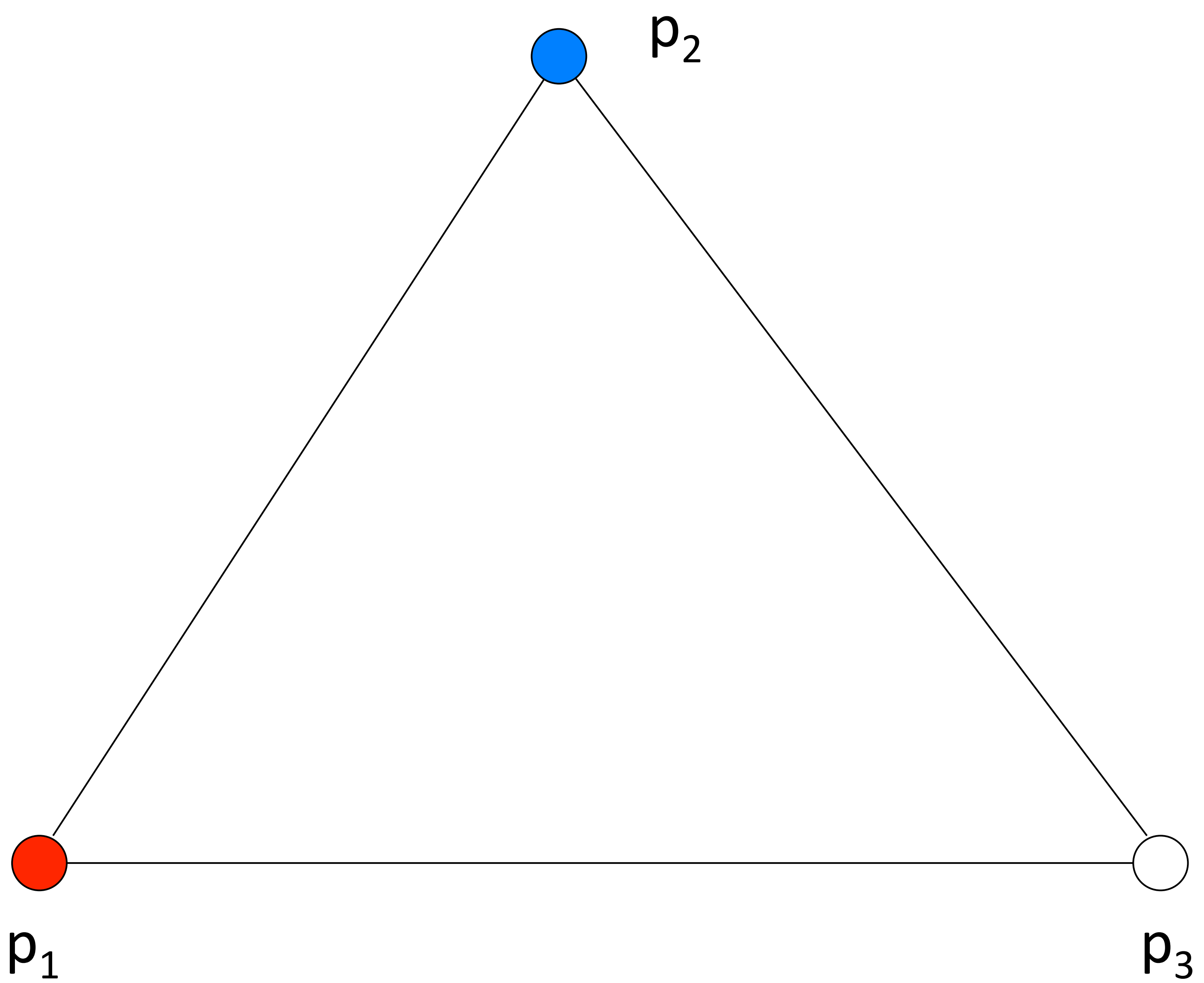
r := 0

while true do

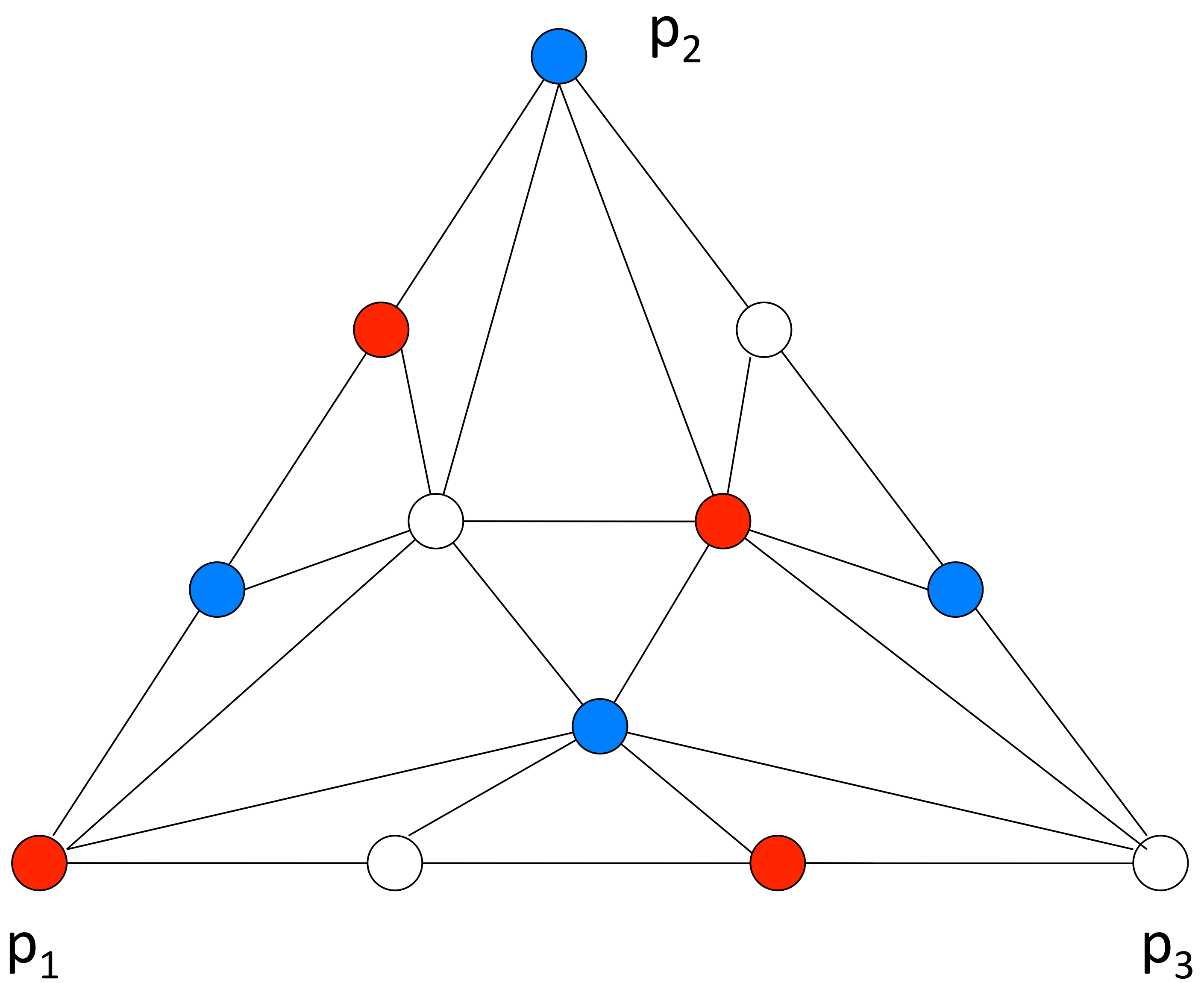
  r := r+1

  v_i := IS_r.WriteRead$_i$(v_i)

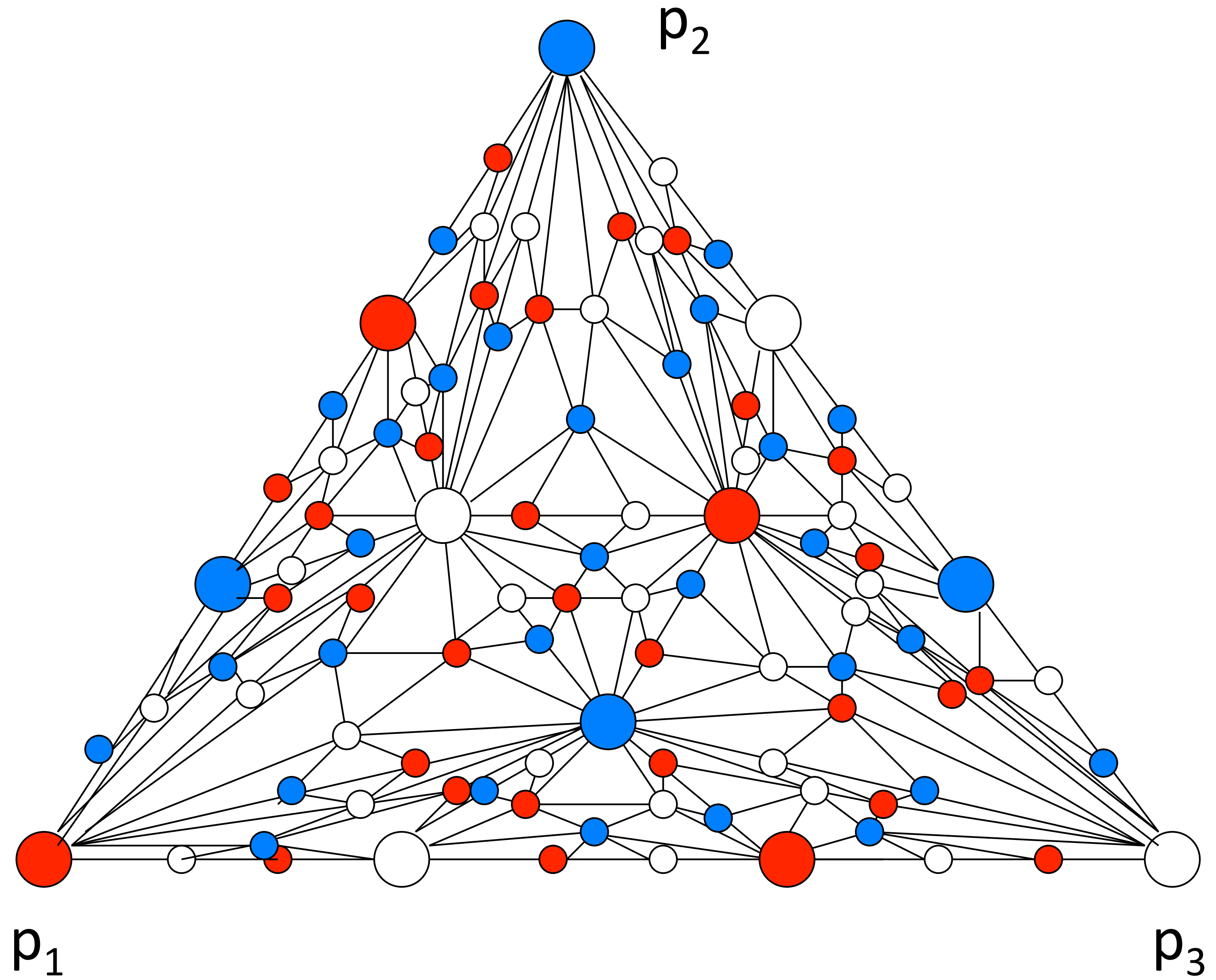
Iterated standard chromatic subdivision (ISDS)



ISDS: one round of IIS



ISDS: two rounds of IIS



IIS is equivalent to (multi-shot) AS

- AS can be used to implement IIS (wait-free)
  - ✓Multiple instances of the construction above (one per iteration)

- IIS can be used to implement multi-shot AS in the lock-free manner:
  - ✓At least one correct process performs infinitely many read or write operations
  - ✓Good enough for protocols solving distributed tasks!

From IIS to AS

We simulate an execution of full-information protocol (FIP) in the AS model, i.e., each process p_i runs:

state := input value of p_i
repeat
    update$_i$(state)
    state := snapshot()
until undecided(state)

Recursively, vector of vectors

(the input value and the decision procedure depend on the problem being solved)

If a problem is solvable in AS, it is solvable with FIP

For simplicity, assume that the k-th written value = k
("without loss of generality" – every written value is unique)

From IIS to AS: non-blocking simulation

Shared: $IS_1, IS_2, \ldots$ // an infinite sequence of one-shot IS memories

Local: at each process, c[1,…,N]=[(0,T),…,(0,T)]

Code for process p_i:

```
r:=0; c[i].clock:=1; // p_i's initial value
repeat forever
        r:=r+1
        view := IS_r.WriteRead(c)  // get the view in IS_r
        c := top(view)  // get the top clock values
        if |c|=r then // the current snapshot completed
            if undecided(ctop) then
                    c[i].val:=ctop;
                    c[i].clock:=c[i].clock+1 // update the clock
            else
                    return decision(ctop) // return the decision
```

From IIS to AS

Each process p_i maintains a vector clock c[1,…,N]

- Each c[j] has two components:
  - ✓ c[j].clock: the number of updates of p_j "witnessed" by p_i (c.clock - the corresponding vector)
  - ✓ c[j].val: the most recent value of p_j's vector clock "witnessed" by p_i (c.val – the corresponding vector)
- To perform an update: increment c[i].clock and set c[i].val to be the "most recent" vector clock
- To take a snapshot: go through iterated memories until $|c| = \Sigma_j c[j].clock$ is "large enough",
  - ✓ i.e. $|c| = r$ (the current round number)
  - ✓ As we'll see, $|c| \geq r$: every process p_i begins with c[i]=1

- We say that $c \geq c'$ iff for all j, c[j].clock $\geq$ c' [j].clock (c observes a more recent state than c)
  - ✓ Not always the case with c and c' of different processes

- $|c| = \Sigma_j$ c[j].clock (sum of clock values of the last seen values)

- For c = c[1],…c[N] (vector of vectors c[j]), top(c) is the vector of most recent seen values:

c[1]	= [1	3	2]
c[2]	= [4	2	1]
c[3]	= [2	1	5]
top(c)	= [4	3	5]

From IIS to AS: correctness

Let c_r denote the vector evaluated by an undecided process p_i in round r (after computing the top function)

Lemma 1 $|c_r| \geq r$

Proof sketch

$c_{r+1} \geq c_r$ (by the definition of top)

Initially $|c_1| \geq 1$ (each process writes c[1].clock=1 in IS_1)

Inductively, suppose $|c_r| \geq r$, for some round r:

- If $|c_r|=r$, then c', such that $|c'|=r+1$, is written in IS_{r+1}
- If $|c_r|>r$, then c', such that $c' \geq c_r$ (and thus $|c'| \geq |c_r|$) is written in in IS_{r+1}

In both cases, $c_{r+1} \geq r+1$

From IIS to AS: correctness

Lemma 2 Let c_r and c_r' be the clock vectors evaluated by processes p_i and p_j, resp., in round r. Then $|c_r| \leq |c_r'|$ implies $c_r \leq c_r'$

Proof sketch

Let S_i and S_j be the outcomes of IS_r received by p_i and p_j

$$c_r = top(S_i) \text{ and } c_r' = top(S_j)$$

Either S_i is a subset of S_j or S_j is a subset of S_i (the Containment property of IS)

Suppose S_i is a subset of S_j, then each clock value seen by p_i is also seen by p_j Why?

=> $|c_r| \leq |c_r'|$ and $c_r \leq c_r'$ Why?

From IIS to AS: correctness

Corollary 1 (to Lemma 2) All processes that complete a snapshot operation in round r, get the same clock vector c, |c|=r

Corollary 2 (to Lemmas 1 and 2) If a process completes a snapshot operation in round r with clock vector c, then for each clock vector c' evaluated in round r'≥r, we have $c \leq c'$

From IIS to AS: linearization

Lemma 3 Every execution's history is linearizable (with respect to the AS spec.)

Proof sketch

Linearization

- Order snapshots based on the rounds in which they complete
- Put each update(c) just before the first snapshot that contains c (if no such snapshot – remove)

By Corollaries 1 and 2, snapshots and updates put in this order respect the specification of AS – legality

The linearization points take place "within the interval" of k-th update and k-th snapshot of p_i - between the k-th and the (k+1)-th updates of c[i].val – precedence

From IIS to AS: liveness

Lemma 4 Some correct undecided process completes infinitely many snapshot operations (or every process decides).

Proof sketch

By Lemma 1, a correct process p_i does not complete its snapshot in round r only if $|c_r|>r$

Suppose p_i never completes its snapshot

=> c_r keeps grows without bound and

=> some process p_j keeps updating its c[j]

=> some process p_j completes infinitely many snapshots

(Chapter 9 in lecture notes)

IIS=AS for wait-free task solutions

- Suppose we simulate a wait-free protocol for solving a task:
  - ✓Every process starts with an input
  - ✓Every process taking sufficiently many steps (of the full-information protocol) eventually decides (and thus stops writing new values, but keeps writing the last one)
  - ✓Outputs match inputs (we'll see later how it is defined)
- If a task can be solved in AS, then it can be solved in IIS
  - ✓We consider IIS from this point on

Quiz 2

1. Would the (one-shot) IS algorithm be correct if we replace A_r.update$_i$(v_i) with U_r[i].write(v_i) and A_r.snapshot() with scan(U_r[1],…,U_r[N])?
2. Would it be possible to use only one array of N registers?
3. Complete the proofs of Lemma 2 and Corollaries 1 and 2