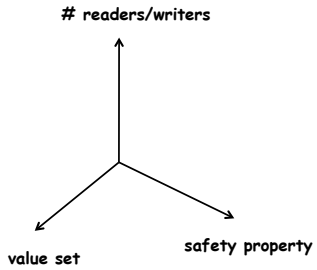
FDS 2012:
Atomic and immediate snapshots

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)

This class

- Atomic snapshot: reading multiple locations atomically
  - ✓Write to one, read *all*
- Immediate snapshot
  - ✓Write-read in groups
- Topological representation of shared-memory computing

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?

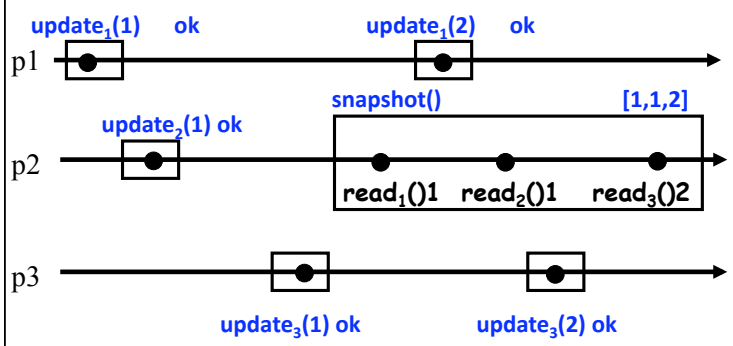
Code for process p_i:

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

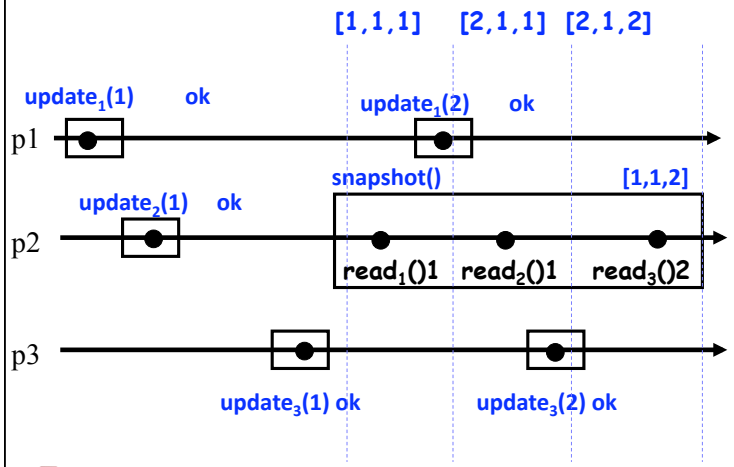
upon snapshot()
$[x_1,\ldots,x_N]$:= scan($R_1,\ldots,R_N$) /*read $R_1,\ldots R_N$*/
return $[x_1,\ldots,x_N]$

upon $update_i(v)$
R_i.write(v)

Snapshot for free?



Snapshot for free?



- What about 2 processes?
- What about non-blocking snapshot?
  - ✓At least one correct process makes progress (completes infinitely many)

Non-blocking 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)

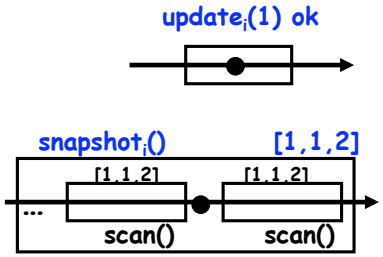
R_i.write(v)

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

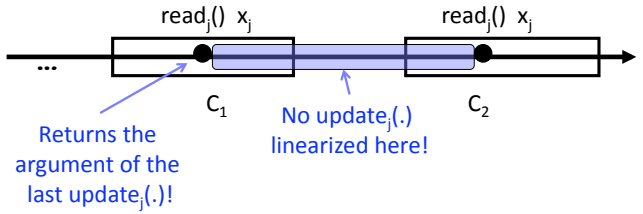


Correctness: linearizability

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: non-blockingness

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

General case: helping?

What if an update interferes with a snapshot?

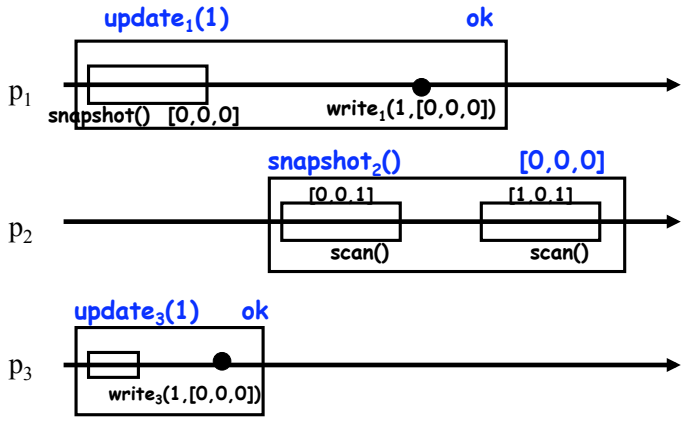
- Make the update do the work!

upon snapshot()
$[x_1,\dots,x_N]$:= scan($R_1,\dots, R_N$)
$[y_1,\dots,y_N]$:= scan($R_1,\dots,R_N$)
if $[y_1,\dots,y_N] = [x_1,\dots,x_N]$ then
return $[x_1,\dots,x_N]$
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()
$[x_1,\dots,x_N]$:= scan($R_1,\dots,R_N$)
while true do
$[y_1,\dots,y_N]$:= $[x_1,\dots,x_N]$
$[x_1,\dots,x_N]$:= scan($R_1,\dots,R_N$)
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 := 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!

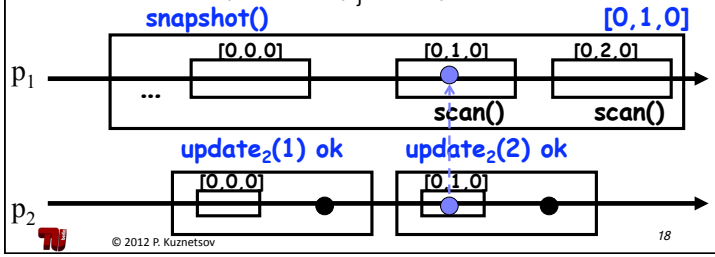
update: snapshot() + one more step

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 return 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)

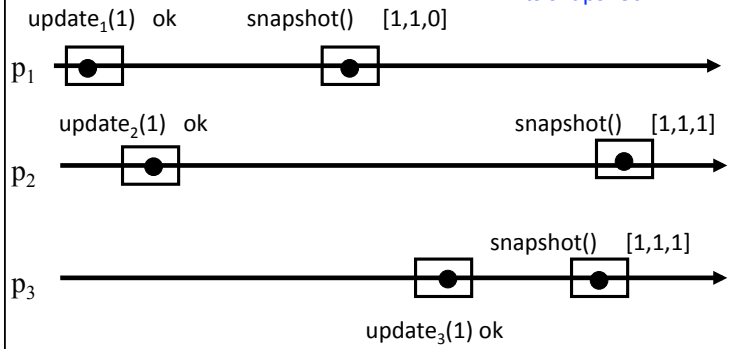
Vectors S_i satisfy:

- 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

Homework: prove the two properties are met by AS

"Unbalanced" snapshots

p_1 sees p_2 but misses its snapshot



Enumerating possible runs: two processes

Each process p_i (i=1,2):

$update_i(v_i)$

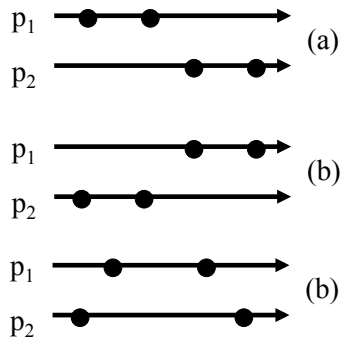
S_i := snapshot()

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

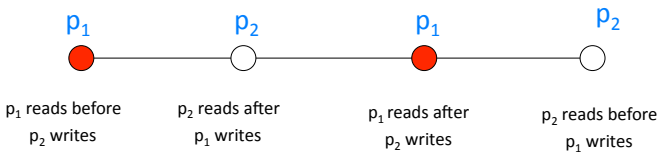


Topological representation of concurrency: two processes

Vertex: local state

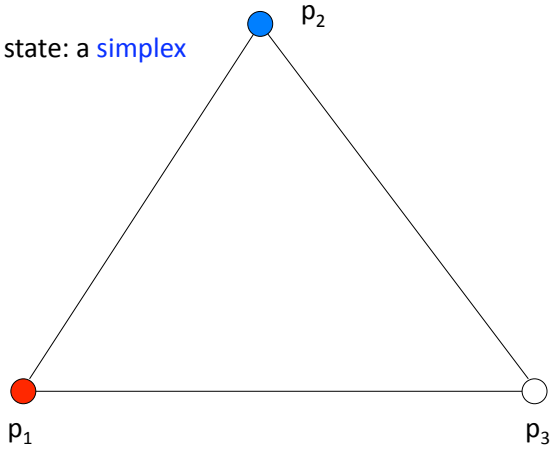
Simplex (set of vertexes): global state

Simplicial complex: set of reachable global states

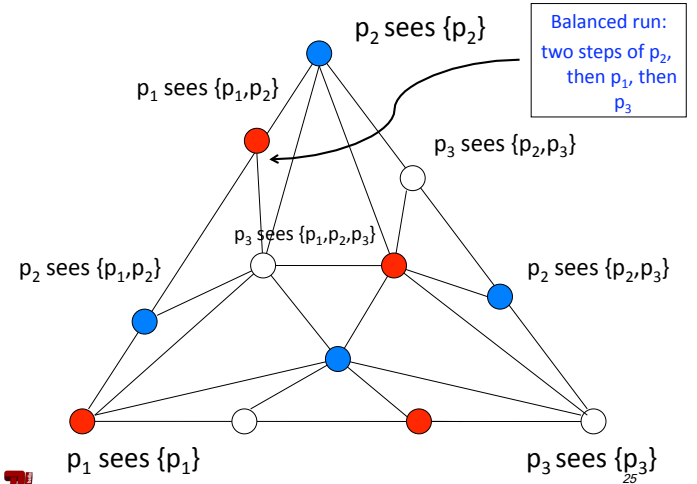


Topological representation: three processes

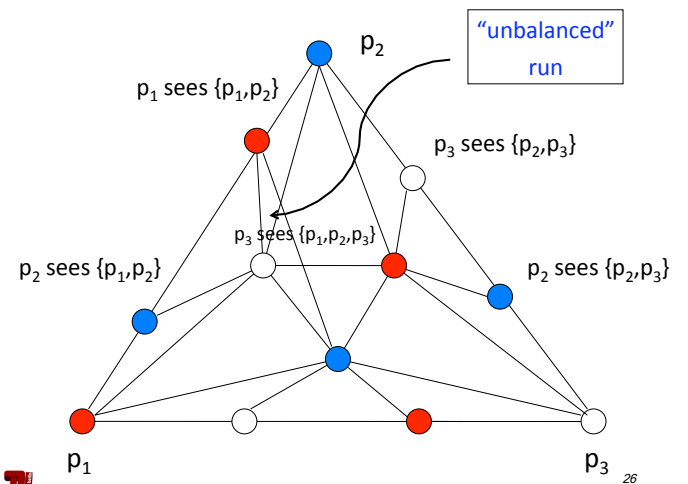
Initial state: a simplex



Topological representation: one-shot AS



Topological representation: one-shot AS



One-shot *immediate* snapshot (IS)

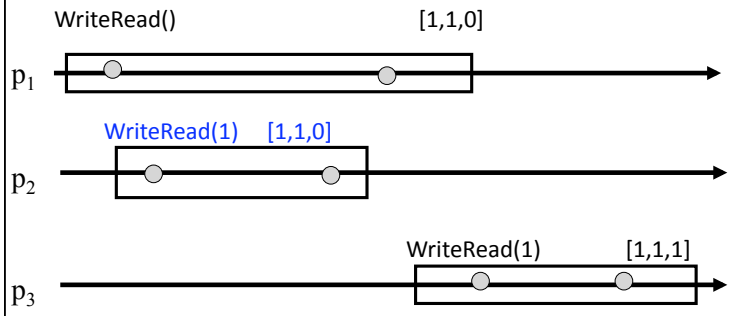
One operation:
ReadWrite(v)

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

"Unbalanced" run eliminated



Every IS run is "balanced"

The runs of IS match the block runs of one-shot AS that can be written as:

$$B_1\ B_2\ \dots\ B_k$$

where

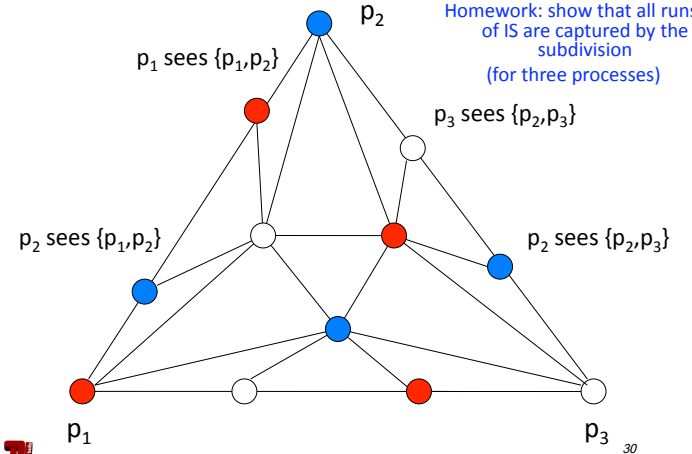
- each B_i = $update_{\Pi i}()$ $snapshot_{\Pi i}()$
  - ✓All processes in Π_i update and then all processes in Pi take snapshots
- $\Pi_1,\dots,\Pi_k$ is a partition of $\{p_1,\dots,p_N\}$
  - ✓E.g., $\Pi_1=\{p_2\}$; $\Pi_2=\{p_1\}$; $\Pi_3=\{p_3\}$ in the balanced run above

Homework: prove the equivalence of IS and block runs of AS

Topological representation: one-shot IS

A *subdivision*!

Homework: show that all runs of IS are captured by the subdivision (for three processes)



IS is equivalent to AS (one-shot)

- IS is a restriction of one-shot AS => IS is stronger than one-shot AS
  - ✓Every run if 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

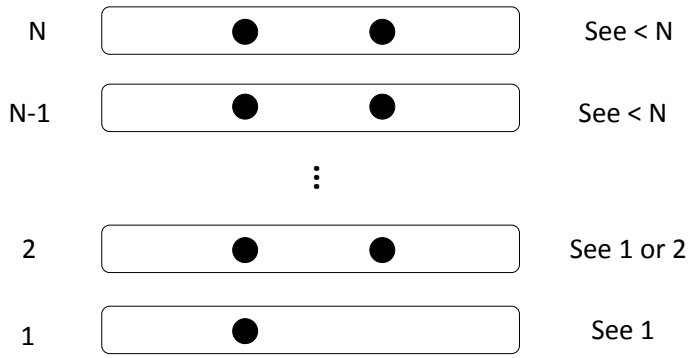
shared variables:

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

upon ReadWrite$_i$(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



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 and 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:

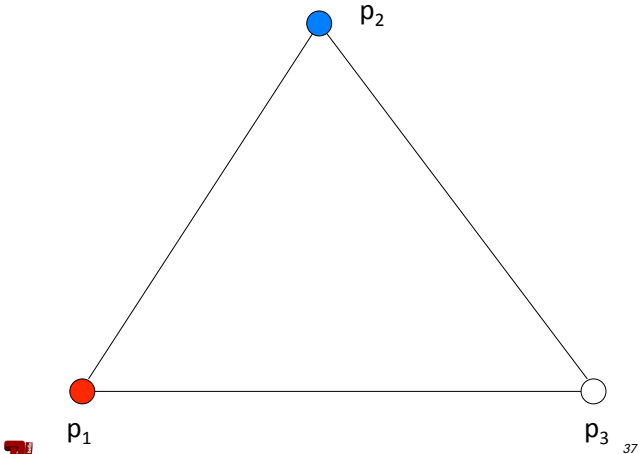
r := 0

while true do

r := r+1

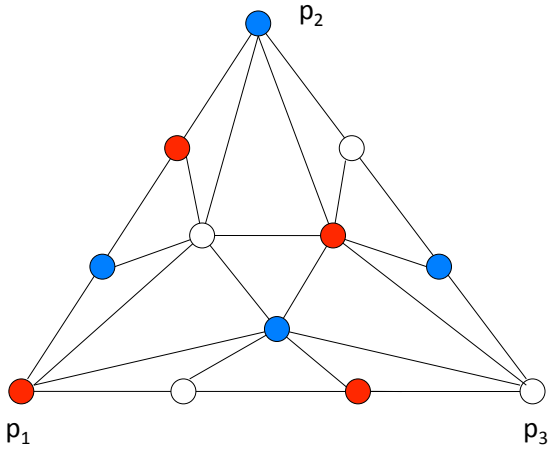
v_i := IS_r.WriteRead$_i$(v_i)

Iterated standard chromatic subdivision



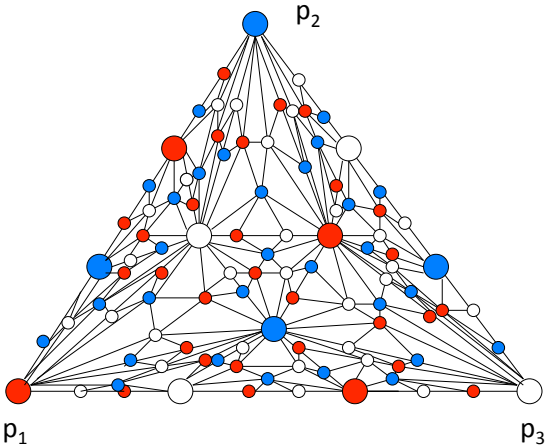
37

ISDS: one round



38

ISDS: two rounds



39

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 AS in the non-blocking manner:
  - ✓At least one correct process performs infinitely many read or write operations
  - ✓Good enough for protocols solving distributed tasks!

40