

The Power of Registers

Prof R. Guerraoui

Distributed Programming Laboratory

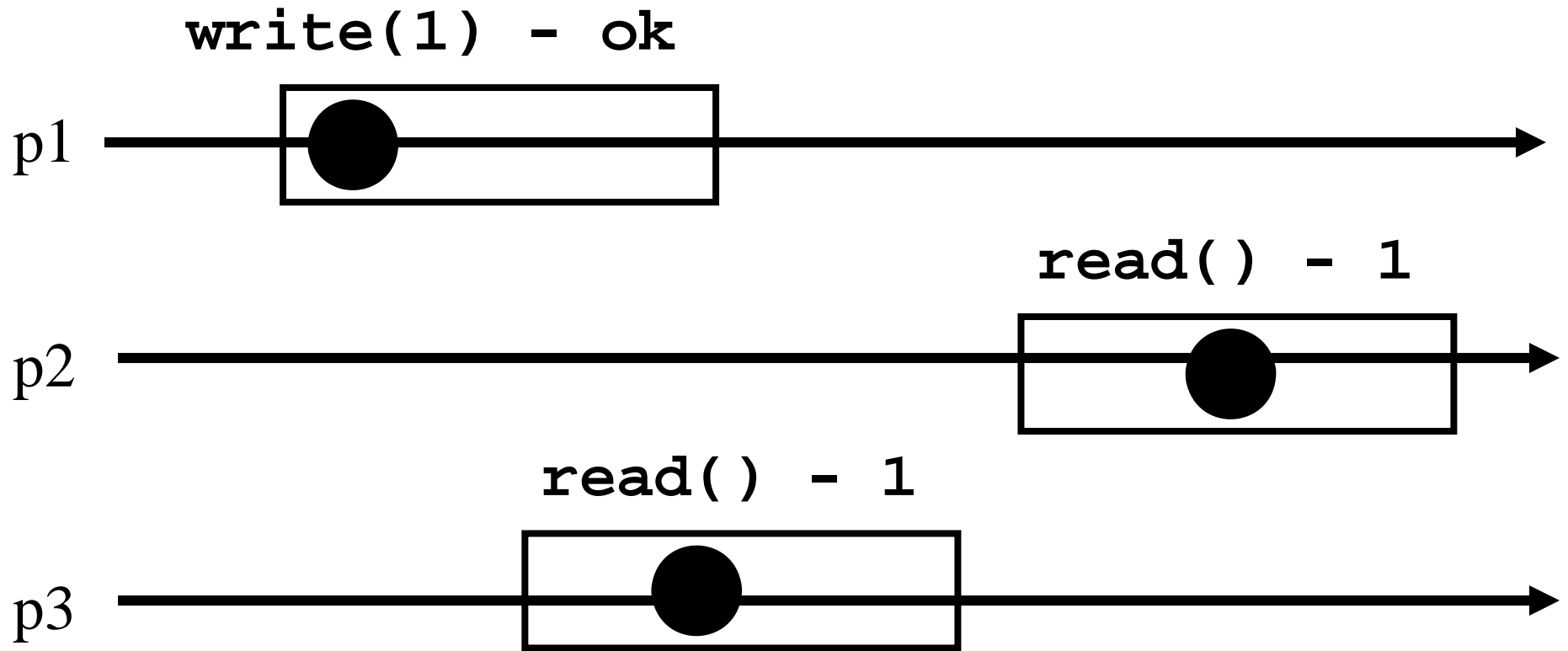


© R. Guerraoui

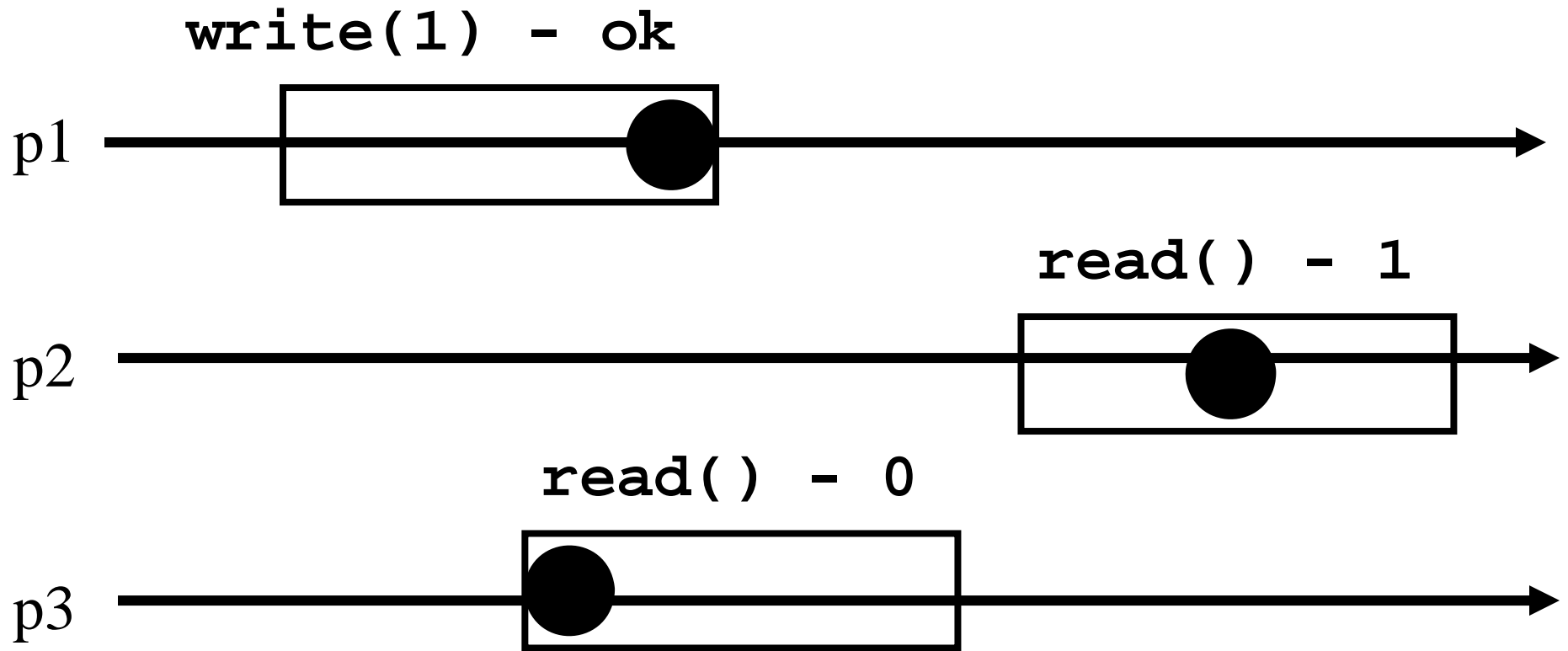
1



Atomic execution



Atomic execution



Registers

- *Question 1:* what objects can we implement with registers?
- Question 2: what objects we cannot implement?

Wait-free implementations of atomic objects

- An *atomic* object is simply defined by its sequential specification; i.e., by how its operations should be implemented when there is no concurrency
- Implementations should be *wait-free*: every process that invokes an operation eventually gets a reply (unless the process crashes)

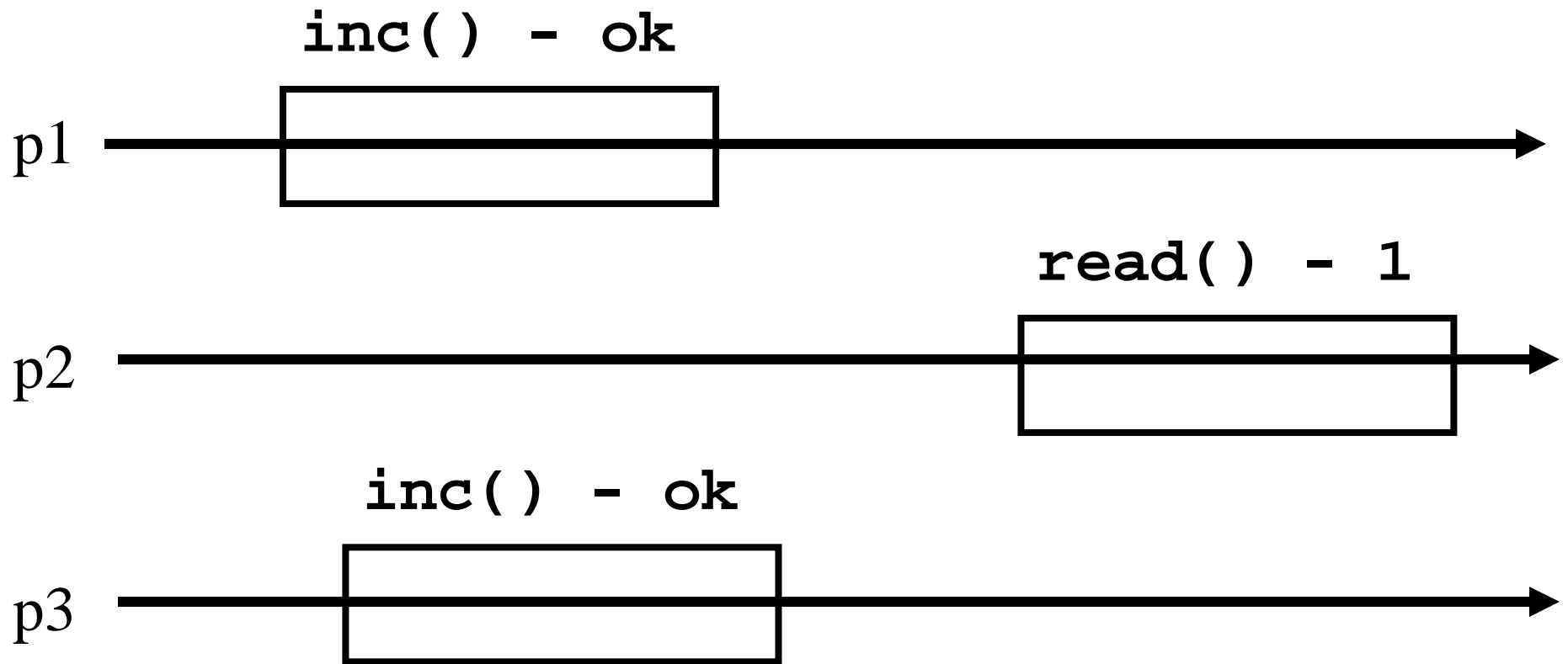
Counter (sequential spec)

- A *counter* has two operations *inc()* and *read()* and maintains an integer *x* *init to 0*
- *read()*:
 - return(x)
- *inc()*:
 - $x := x + 1;$
 - return(ok)

Naive implementation

- The processes share one register Reg
- *read()*:
 - return(Reg.read())
- *inc()*:
 - temp:= Reg.read()+1;
 - Reg.write(temp);
 - return(ok)

Atomic execution?



Atomic implementation

- The processes share an array of registers $\text{Reg}[1, \dots, n]$
- *inc()*:
 - $\text{Reg}[i].\text{write}(\text{Reg}[i].\text{read}() + 1);$
 - $\text{return}(\text{ok})$

Atomic implementation

☛ *read()*:

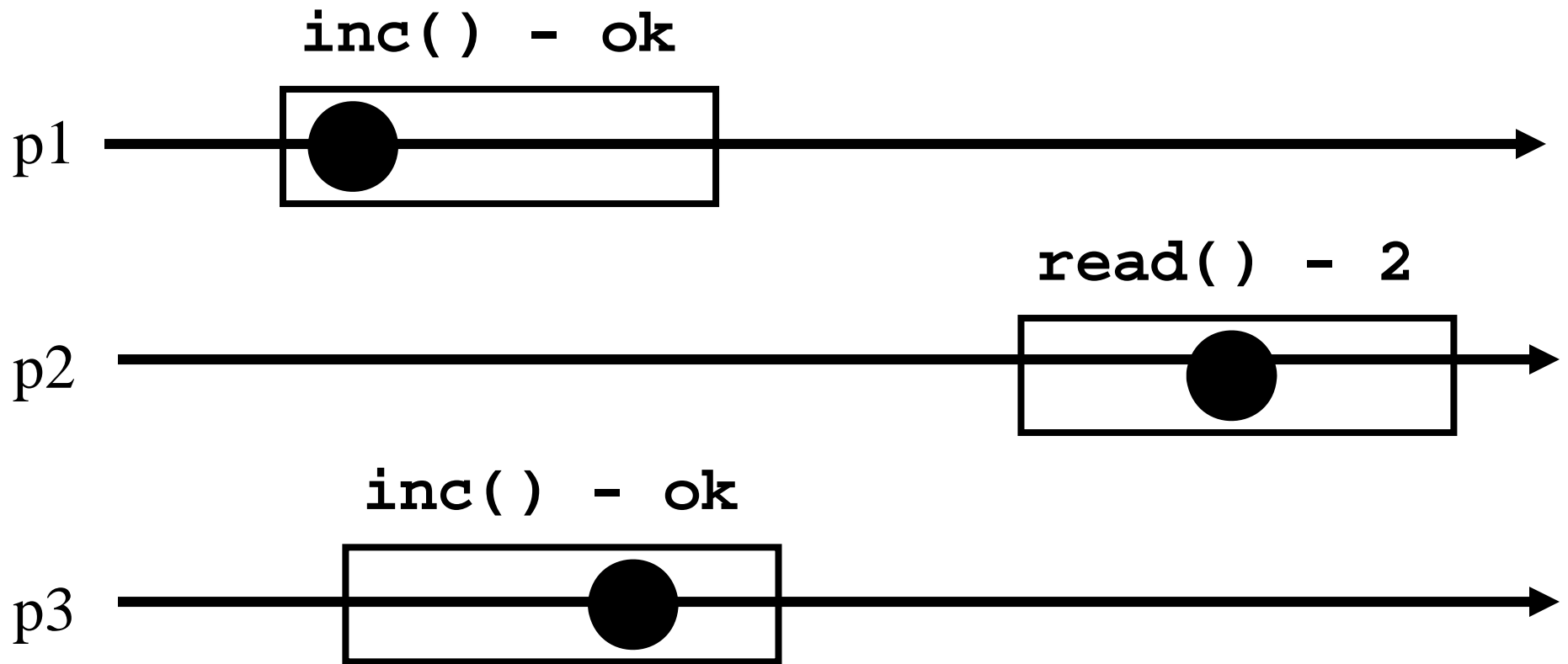
☛ sum := 0;

☛ for j = 1 to n do

☛ sum := sum + Reg[j].read();

☛ return(sum)

Atomic execution?



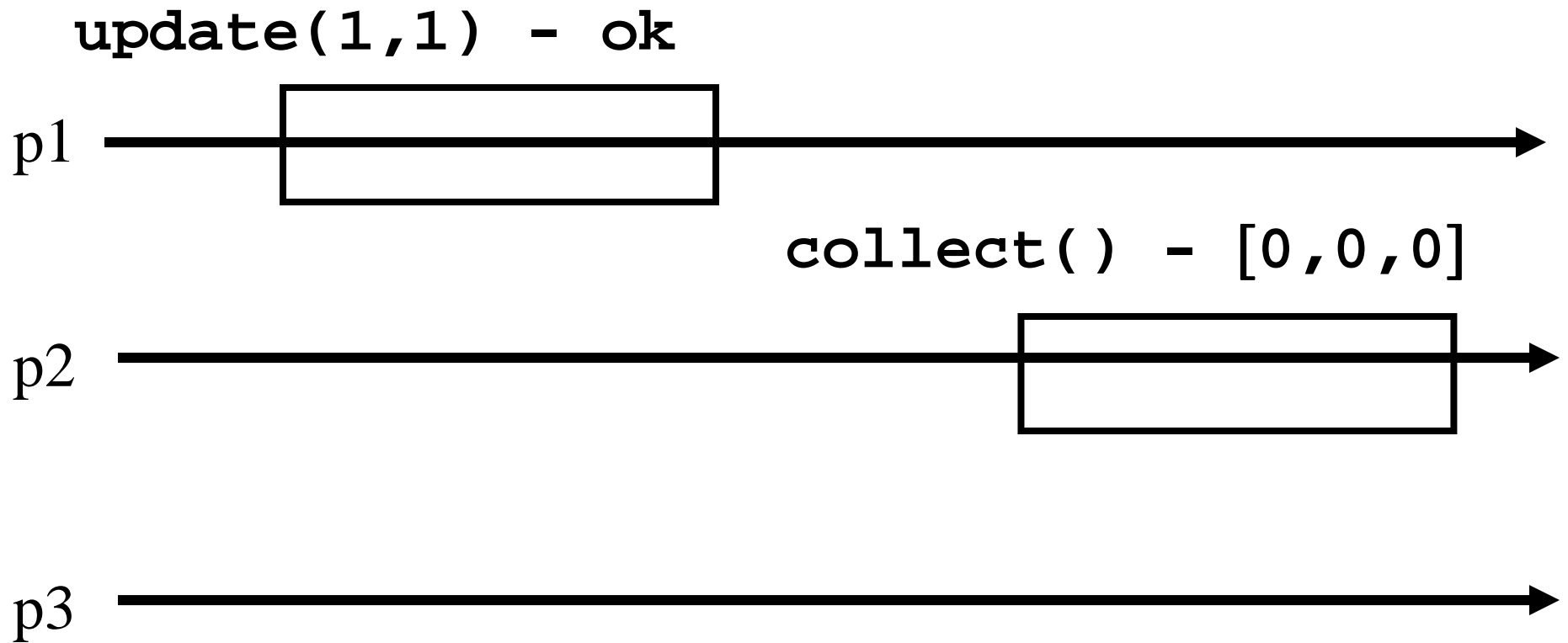
Snapshot (sequential spec)

- A *snapshot* has operations *update()* and *scan()* and maintains an array x of size n
- *scan()*:
 - return(x)
- *update(i, v):*
 - $x[i] := v;$
 - return(ok)

Very naive implementation

- Each process maintains an array of integer variables x init to $[0, \dots, 0]$
- scan()*:
 - return(x)
- update(i, v)*:
 - $x[i] := v;$
 - return(ok)

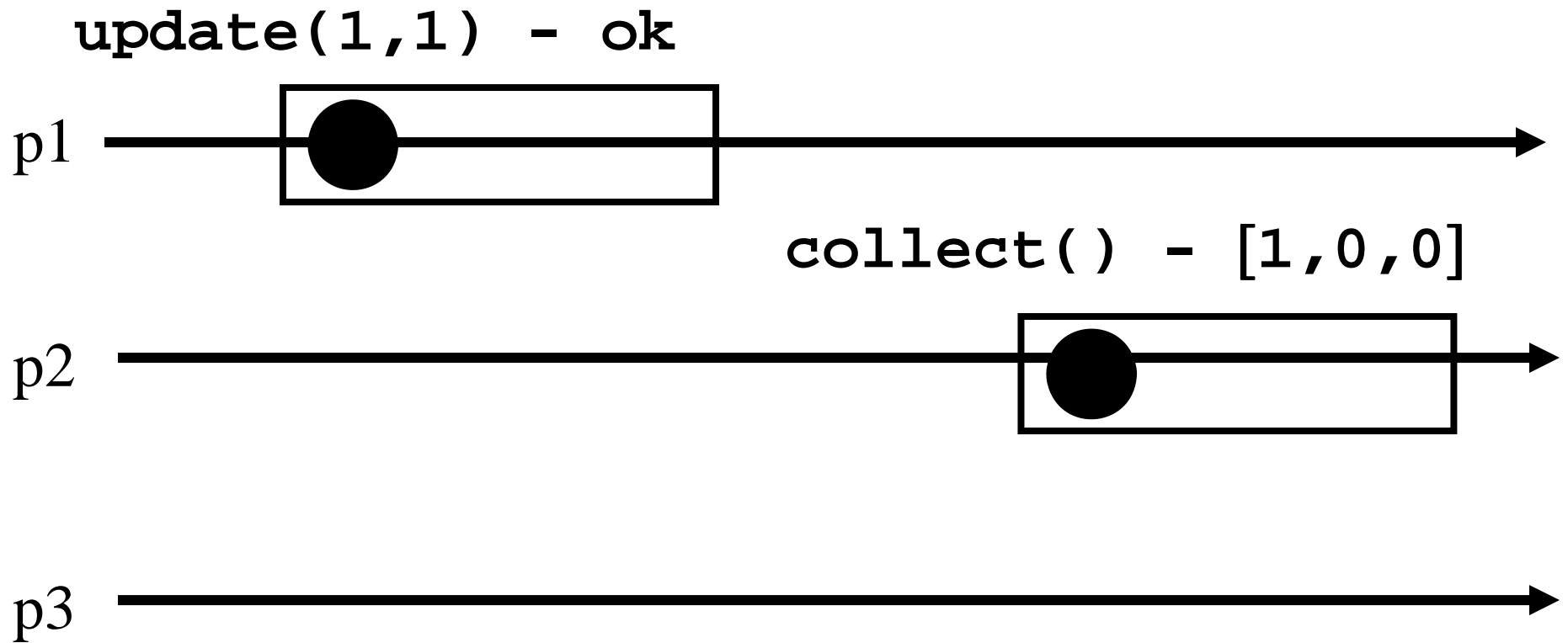
Atomic execution?



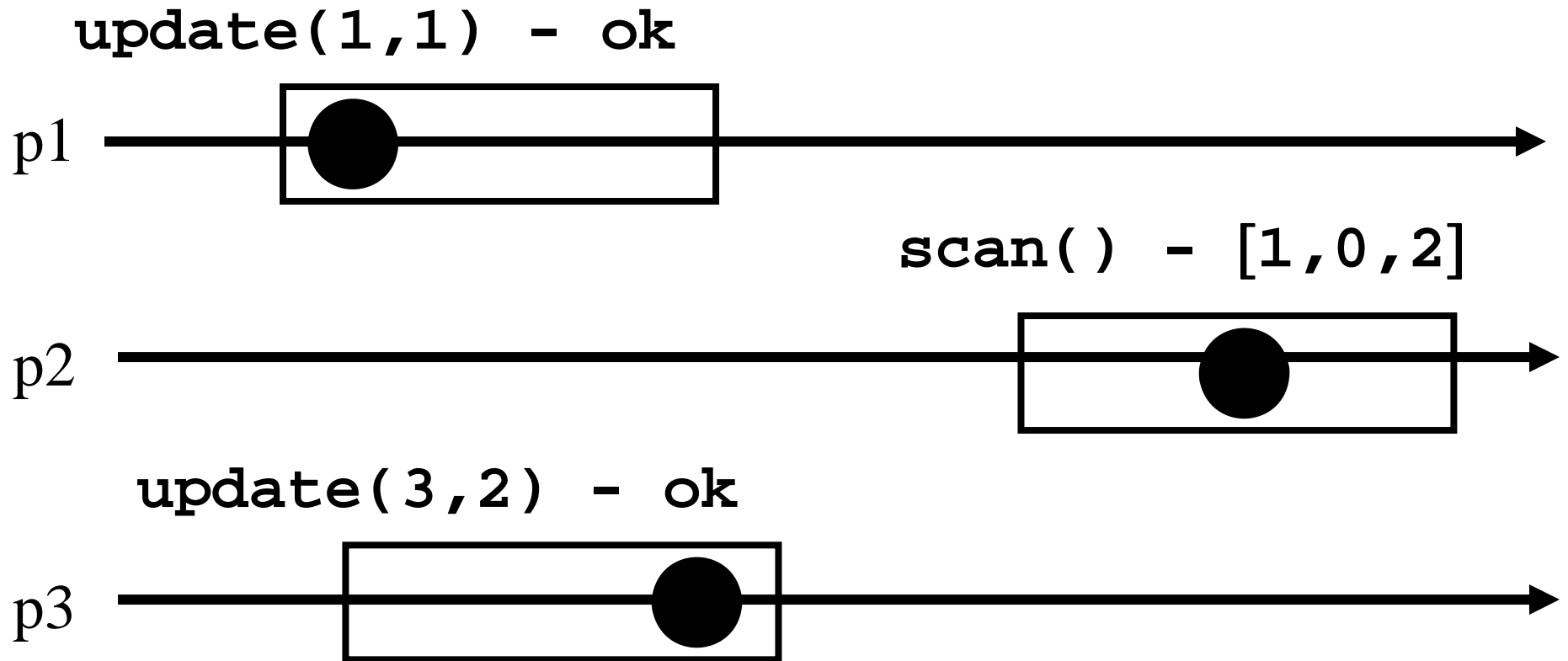
Less naive implementation

- The processes share one array of N registers
Reg[1,..,N]
- *scan()*:
 - for j = 1 to N do
 - x[j] := Reg[j].read();
 - return(x)
- *update(i,v)*:
 - Reg[i].write(v); return(ok)

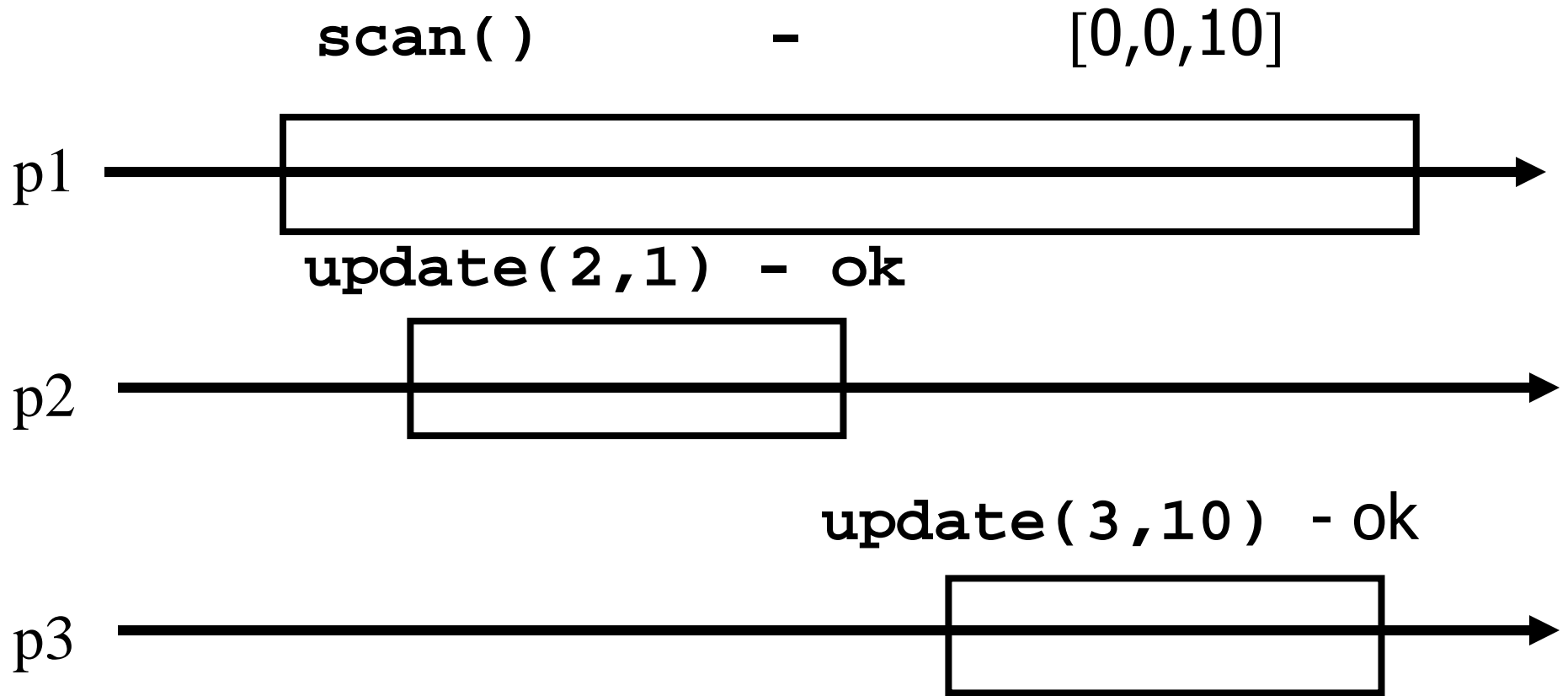
Atomic execution?



Atomic execution?



Atomic execution?



Non-atomic vs atomic snapshot

- What we implement here is some kind of *regular* snapshot:
 - A *scan* returns, for every index of the snapshot, the last written values or the value of any concurrent update
 - We call it *collect*

Key idea for atomicity

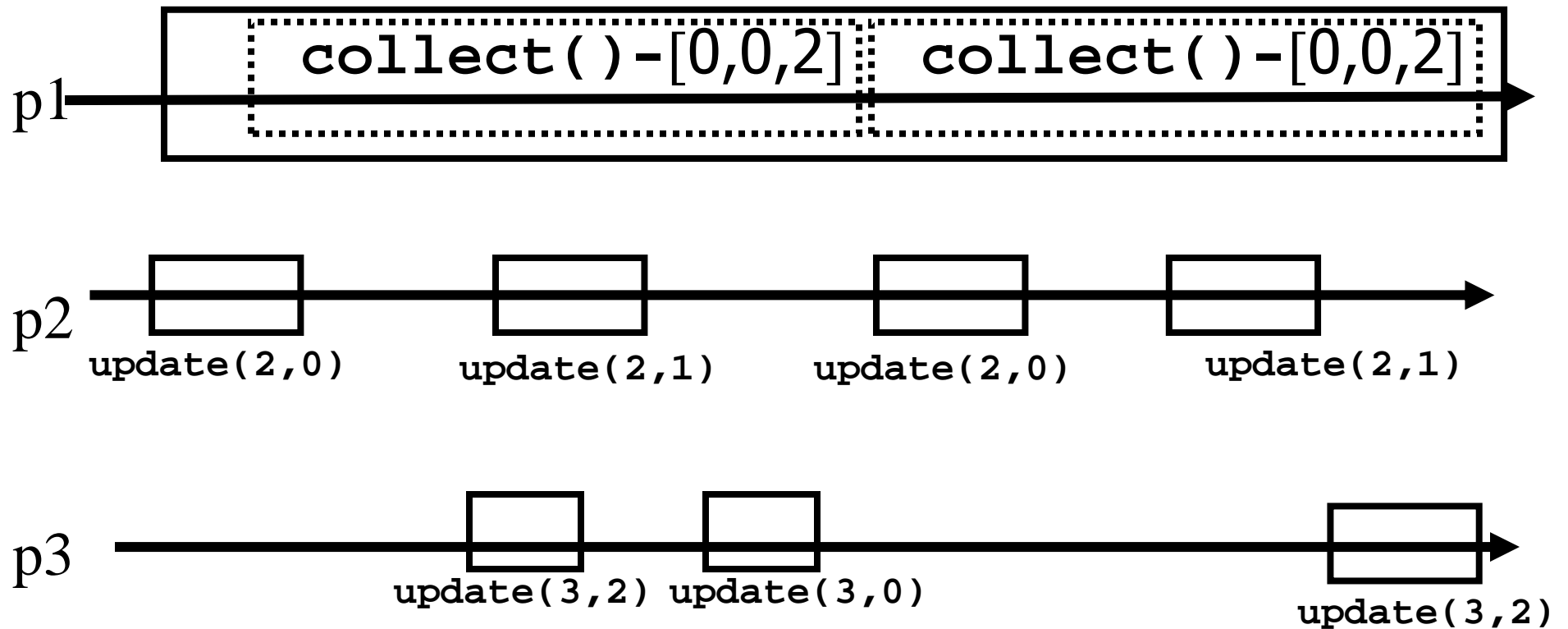
- To *scan*, a process keeps reading the entire snapshot (i.e., it *collect*), until two results are the **same**
- This means that the snapshot did not change, and it is safe to return without violating atomicity

Same value vs. Same timestamp

`scan()`

-

`[0,0,2]`



Enforcing atomicity

- The processes share one array of N registers $\text{Reg}[1, \dots, N]$; each contains a value and a timestamp
- We use the following operation for modularity
- *collect()*:
 - for $j = 1$ to N do
 - $x[j] := \text{Reg}[j].\text{read}()$;
 - return(x)

Enforcing atomicity (cont'd)

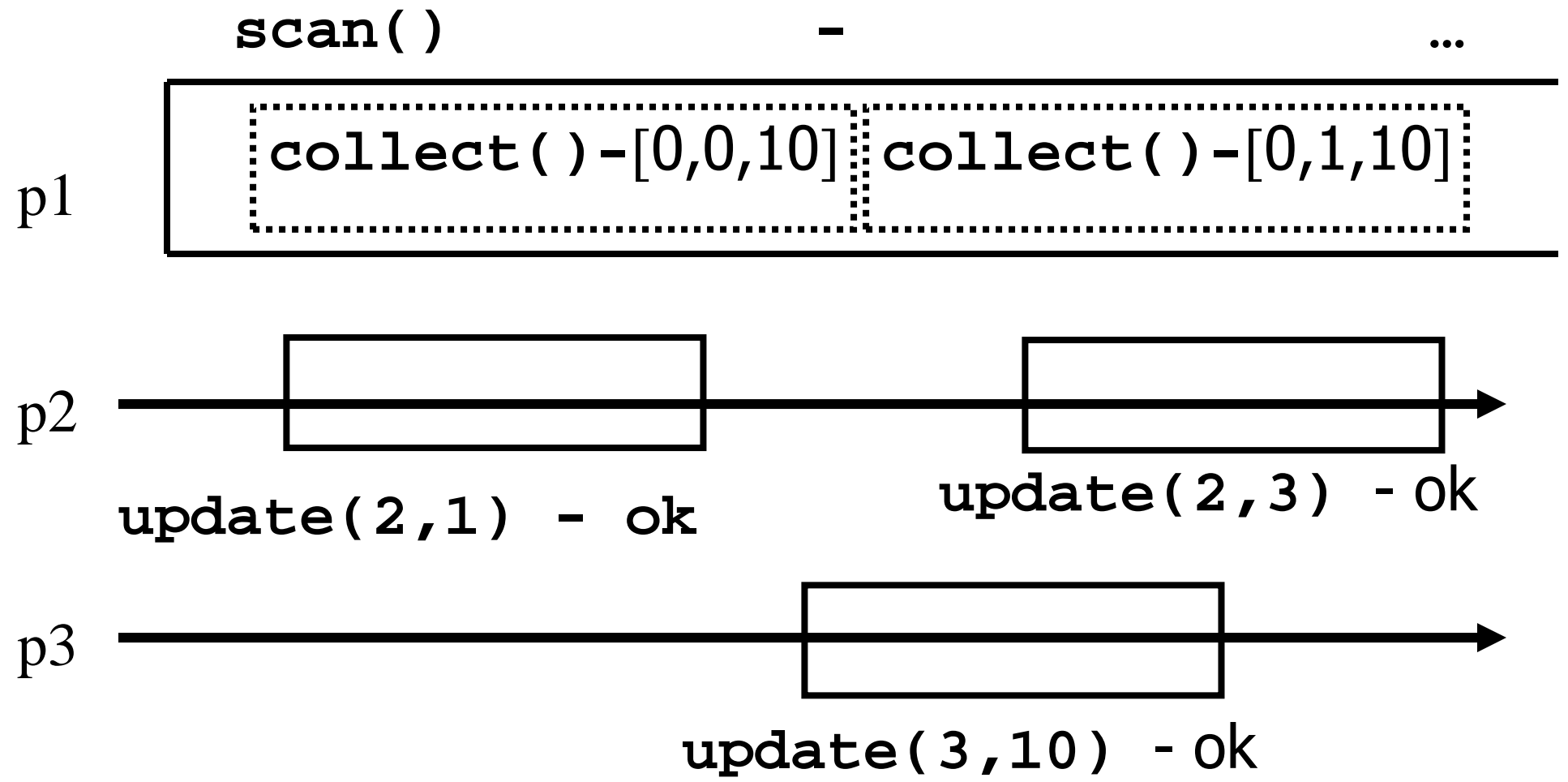
• *scan()*:

- temp1 := self.collect();
- while(true) do
 - temp2 := self.collect();
 - if (temp1 = temp2) then
 - return (temp1.val)
 - temp1 := temp2;

• *update(i,v)*:

- ts := ts + 1;
- Reg[i].write(v,ts);
- return(ok)

Wait-freedom?



Key idea for atomicity & wait-freedom

- The processes share an array of *registers* $\text{Reg}[1,\dots,N]$ that contains each:
 - a value,
 - a timestamp, and
 - a copy of the entire array of values

Key idea for atomicity & wait-freedom (cont'd)

- To *scan*, a process keeps collecting and returns a collect if it did not change, or some collect returned by a concurrent *scan*
 - Timestamps are used to check if the collect changes or if a scan has been taken in the meantime
- To *update*, a process *scans* and writes the value, the new timestamp and the result of the scan

Snapshot implementation

Every process keeps a local timestamp ts

• *update*(i, v):

• $ts := ts + 1;$

• `Reg[i].write(v,ts,self.scan());`

• `return(ok)`

Snapshot implementation

• *scan()*:

- $t1 := \text{self.collect}(); t2 := t1$
- while(true) do
 - $t3 := \text{self.collect}();$
 - if ($t3 = t2$) then return ($t3$);
 - for $j = 1$ to N do
 - if($t3[j,2] \geq t1[j,2]+2$) then
 - return ($t3[j,3]$)
 - $t2 := t3$

Return the first value in each cell in $t3$

Possible execution?

`scan()`

-

`[0,0,3]`

p1



p2



p3



`update(3,1)-ok`

`update(3,2)-ok`

`update(3,3)-ok`