Computing with anonymous processes

Prof R. Guerraoui

Distributed Programming Laboratory

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)

Counter (atomic implementation)

- The processes share an array of SWMR registers Reg[1,..,n] ; the writer of register Reg[i] is pi

- *inc():*
  - temp := Reg[i].read() + 1;
  - Reg[i].write(temp);
  - return(ok)

Counter (atomic implementation)
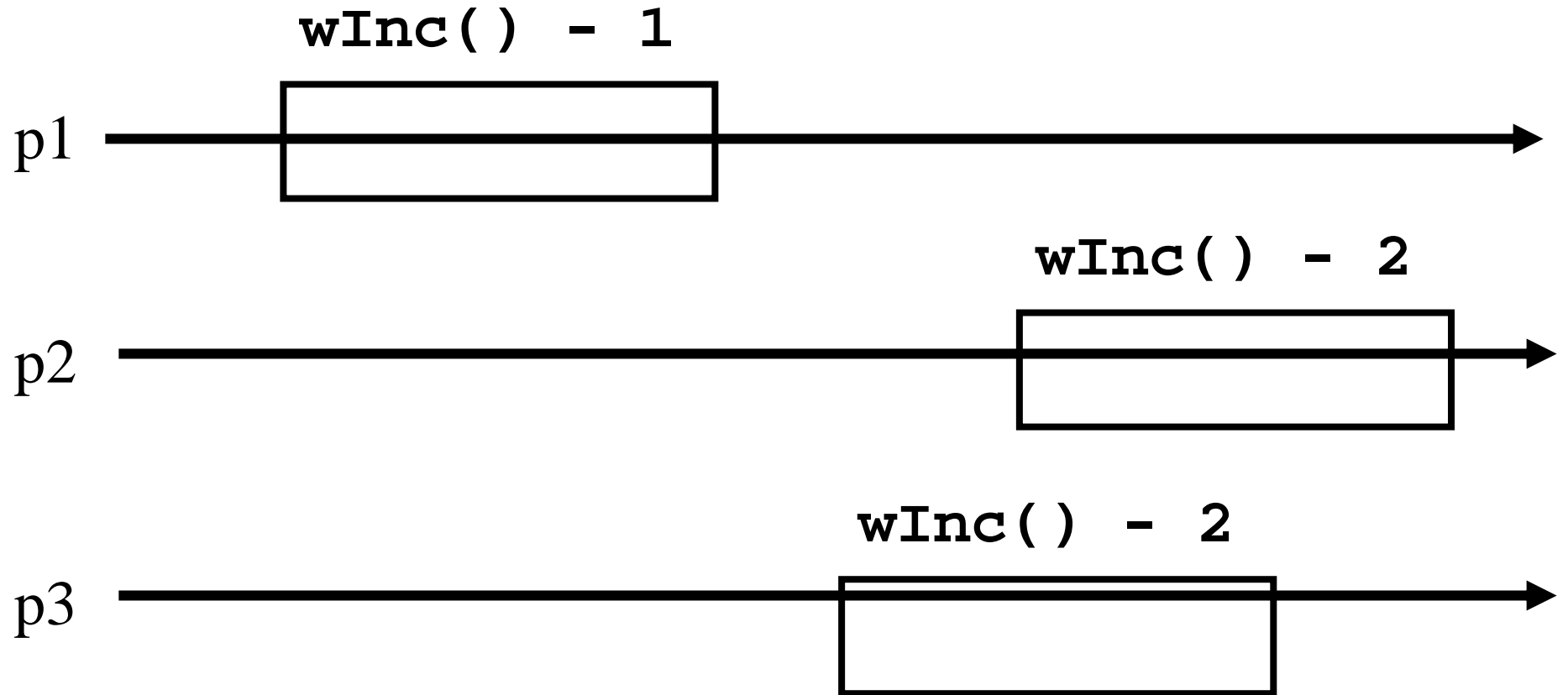
- *read():*
  - sum := 0;
  - for j = 1 to n do
    - sum := sum + Reg[j].read();
  - return(sum)

Weak Counter

- A *weak counter* has one operation *wInc()*
- *wInc():*
  - x := x + 1;
  - return(x)
- correctness: if an operation precedes another, then the second returns a value that is larger than the first one (regularity vs atomicity)
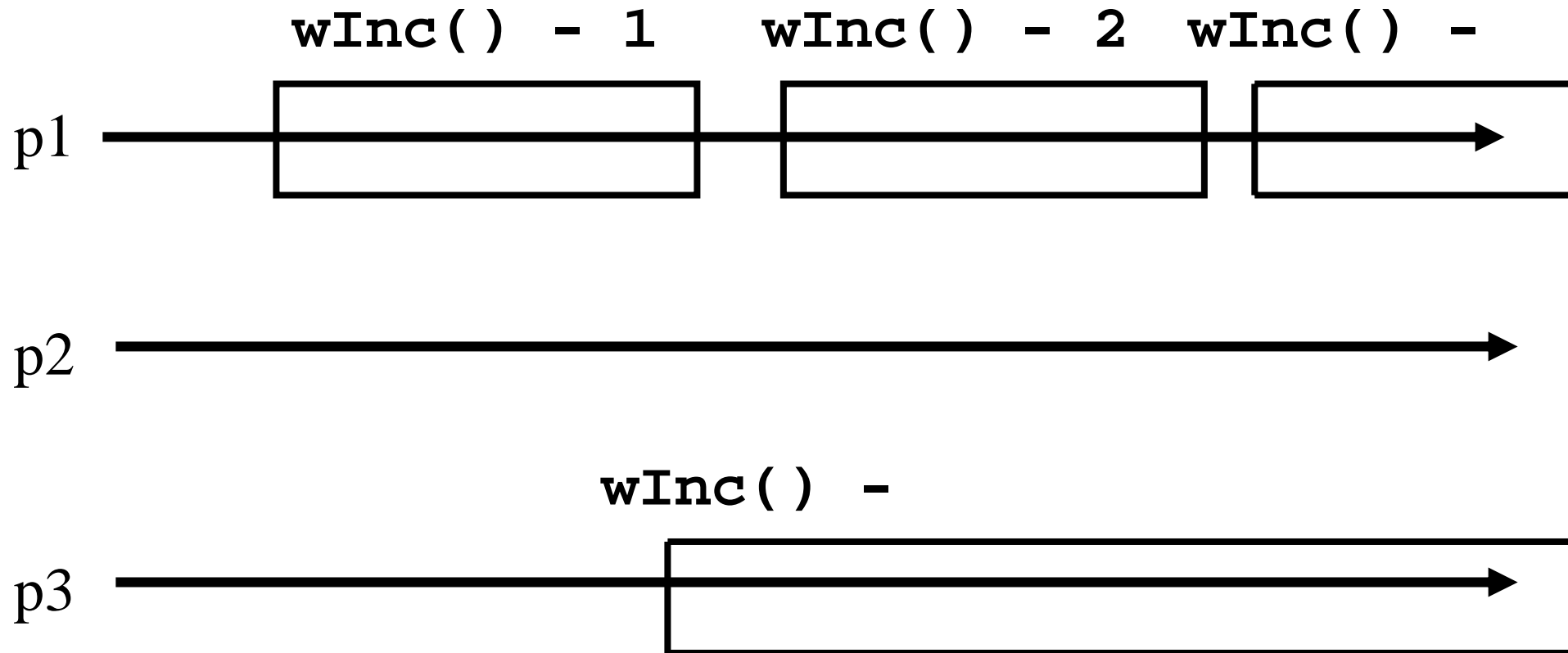
Weak counter execution

wInc() - 1

p1

wInc() - 2

p2

wInc() - 2

p3

Weak Counter
(lock-free implementation)

- The processes share an (infinite) array of MWMR registers Reg[1,..,n,..,], init to 0

- *wInc():*

  - i := 0;

  - while (Reg[i].read() ≠ 0) do

    - i := i + 1;

  - Reg[i].write(1);

  - return(i);

Weak counter execution

```
wInc() - 1    wInc() - 2  wInc() -
```

p1

```
wInc() -
```

p2

p3

Weak Counter
(wait-free implementation)

- The processes also use a MWMR register L
- *wInc():*
    - i : = 0;
    - while (Reg[i].read() ≠ 0) do
        - if L has been updated n times then
            - return the largest value seen in L
        - i := i + 1;
    - L.write(i);
    - Reg[i].write(1);
    - return(i);

Weak Counter
(wait-free implementation)

- *wInc():*
  - t:= l := L.read(); i := 0;
    - while (Reg[i].read() ≠ 0) do
    - if L.read() ≠ l then
      - l := L.read(); t := max(t,l); k := k+1
      - if k = n then return(t)
    - i := i + 1;
  - L.write(i);
  - Reg[i].write(1);
  - return(i);

Snapshot (sequential spec)

- A *snapshot* has operations *update()* and *scan()* and maintains an array x of size n
- *scan():*
  - return(x)
- NB. No component is devoted to a process
- *update(i,v):*
  - x[i] := v;
  - return(ok)

Key idea for atomicity & wait-freedom

- The processes share a *Weak Counter*: Wcounter, init to 0;
- The processes share an array of *registers* Reg[1,..,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 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 := Wcounter.wInc();
- Reg[i].write(v,ts,self.scan());
- return(ok)

Snapshot implementation

- *scan():*
  - ts := Wcounter.wInc();
  - while(true) do
    - If some Reg[j] contains a collect with a higher timestamp than ts, then return that collect
    - If n+1 sets of reads return identical results then return that one

Consensus (obstruction-free)

- We consider binary consensus

- The processes share two infinite arrays of registers: $Reg_0[i]$ and $Reg_1[i]$

- Every process holds an integer i init to 1

- Idea: to impose a value v, a process needs to be fast enough to fill in registers $Reg_v[i]$

Consensus (obstruction-free)

- *propose(v):*
  - while(true) do
    - If $Reg_{1-v}[i] = 0$ then
      - $Reg_v[i] := 1;$
      - if $i > 1$ and $Reg_{1-v}[i-1] = 0$ then return(v);
    - else $v := 1-v;$
    - $i := i+1;$
    - end

Consensus (solo process)

q(1)

```
Reg0(1)=0
```

```
Reg1(1):=1
```

```
Reg0(2)=0
```

```
Reg1(2):=1
```

```
Reg0(1)=0
```

Consensus (lock-step)

q(1)   p(0)

Reg0(1)=0   Reg1(1)=0

Reg1(1):=1   Reg0(1):=1

Reg0(2)=0   Reg1(2)=0

Reg1(2):=1   Reg0(2):=1

Reg0(1)=1   Reg0(1)=1

Consensus (binary)

- **propose(v):**
  - while(true) do
    - If $Reg_{1-v}[i] = 0$ then
    - $Reg_v[i] := 1$;
    - if $i > 1$ and $Reg_{1-v}[i-1] = 0$ then return(v);
    - else if $Reg_v[i] = 0$ then v:= 1-v;
    - if $v = 1$ then wait(2i)
    - i := i+1;
    - end