Distributed Algorithms

Shared memory basics

EFREI, 2019

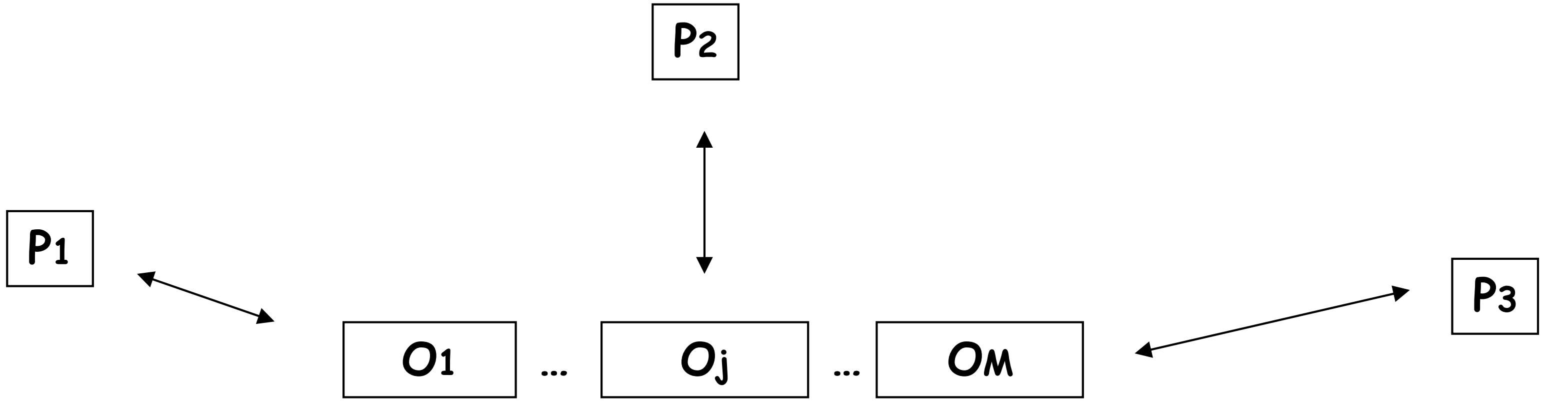
M2 Big Data

Project: the first meetings 23-25/10

- Design a simple distributed system
  - ✓Atomic key-value store (read-write register)
- AKKA: actor-based programming framework
  - ✓Check https://doc.akka.io/docs/akka/current/guide/introduction.html?language=java
  - ✓Make sure you have what is necessary (jdk, maven)
  - ✓Play with the “getting started example”
  - ✓Do the homework (bootstraping a system)

Shared memory model

- Processes communicate by applying operations on and receiving responses from *shared objects*
- A shared object is a state machine
  - ✓States
  - ✓Operations/Responses
  - ✓Sequential specification
- Examples: read-write registers, TAS,CAS,LLSC,…



Read-write register

- Stores *values* (in a *value set* V)
- Exports two operations: read and write
  - ✓Write takes an argument in V and returns ok
  - ✓Read takes no arguments and returns a value in V

Shared memory guarantees

Processes invoke operations on the shared objects and:

- Liveness: the operations eventually return *something*
- Safety: the operations never return *anything incorrect*

Liveness

- An operation is *complete* if its invocation is followed by a matching response
  - ✓write(v) -> ok
  - ✓read() -> a value in V
- A process invoking an operation may fail (stop taking steps) before receiving a response
- A process is correct (in a given run) if it never fails

Under which condition a correct process makes progress?

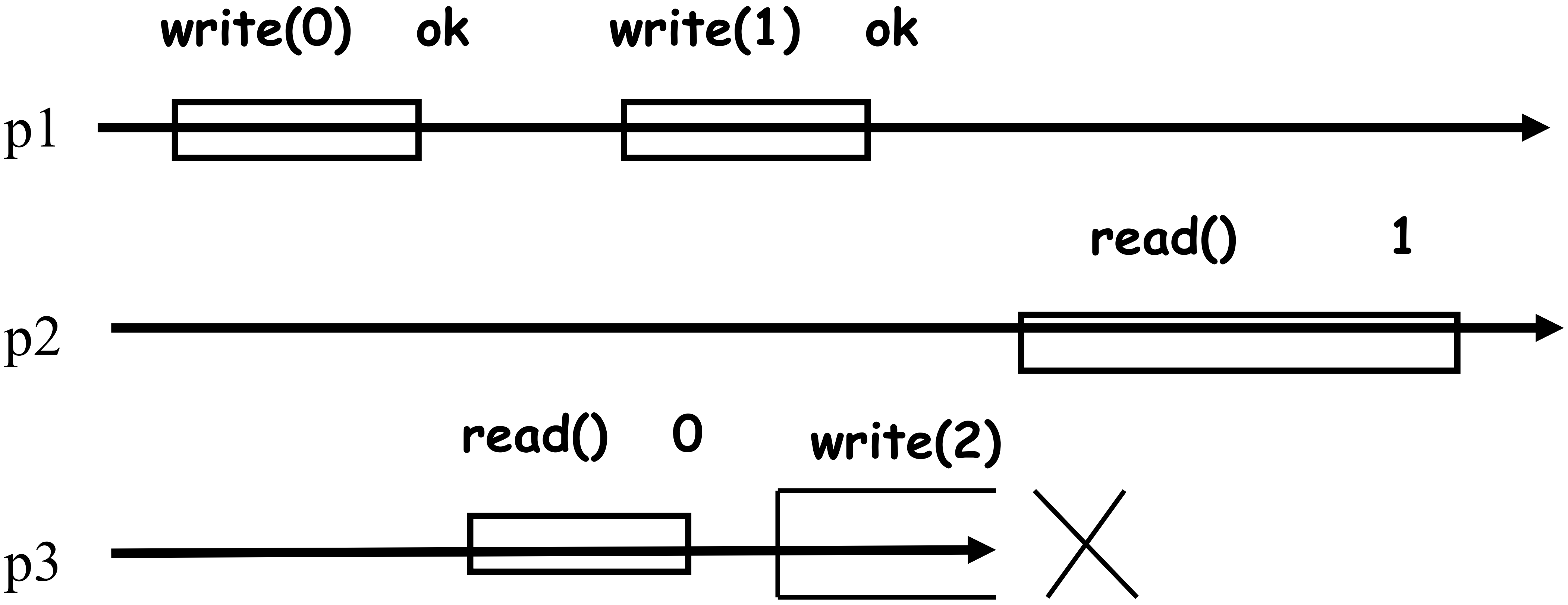
Wait-freedom: unconditional progress

Every operation invoked by a correct process eventually completes

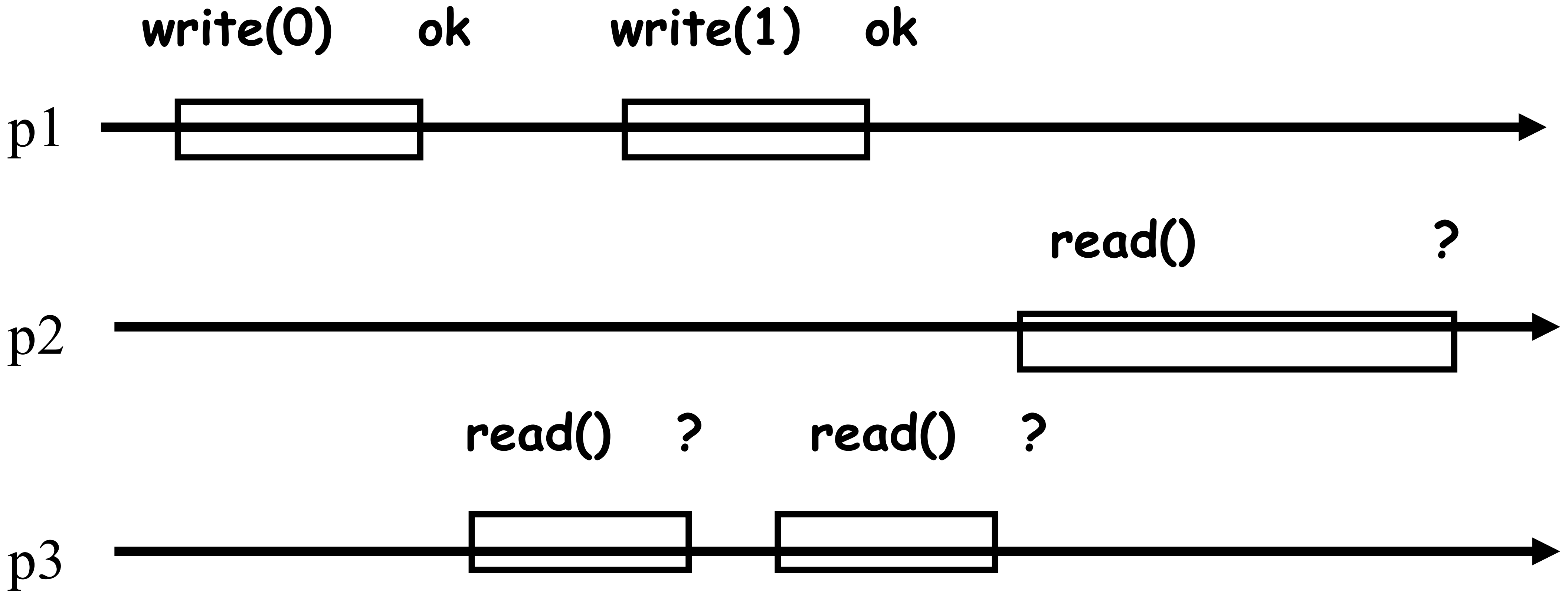
All objects considered in this class are wait-free

We consider well-formed runs: a process never invokes an operation before returning from the previous invocation

A shared memory run



A shared memory run

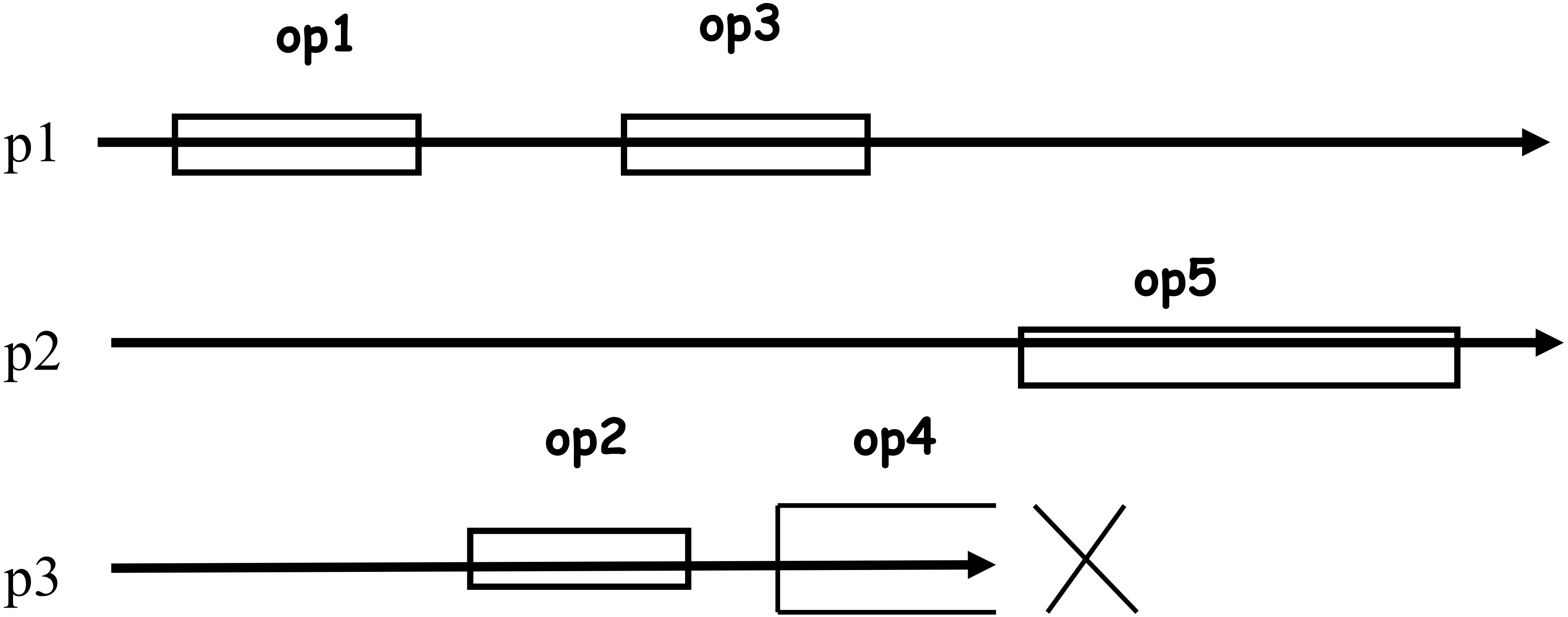


Operation precedence

- Operation op1 precedes operation op2 in a run R if the response of op1 precedes (in global time) the invocation of op2 in R

- If neither op1 precedes op2 nor op2 precedes op1 than op1 and op2 are concurrent

Operation precedence



Safety (registers)

Informally, every read operation returns the “last” written value (the argument of the “last” write operation)

- ✓What does the “last” mean?
- ✓What if operations overlap?

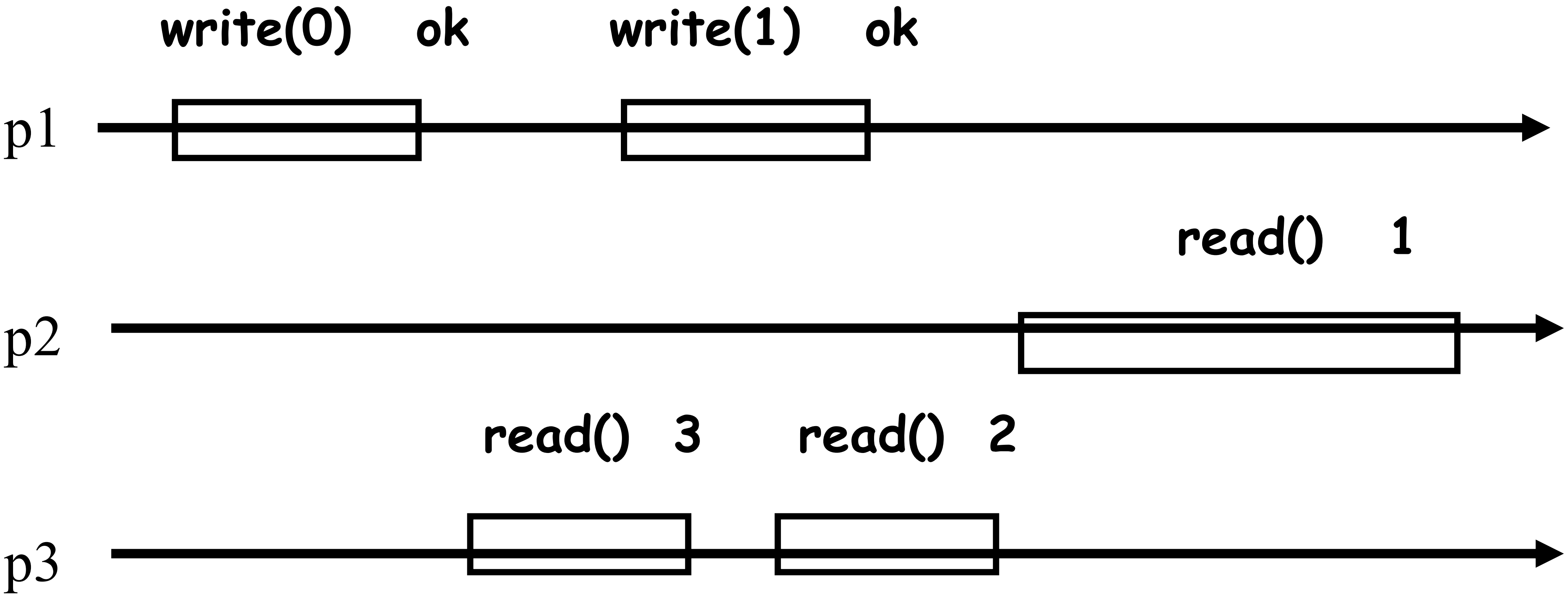
Safety criteria

- Safe registers: every read that does not overlap with a write returns the last written value
- Regular registers: every read returns the last written value, or a concurrently written value

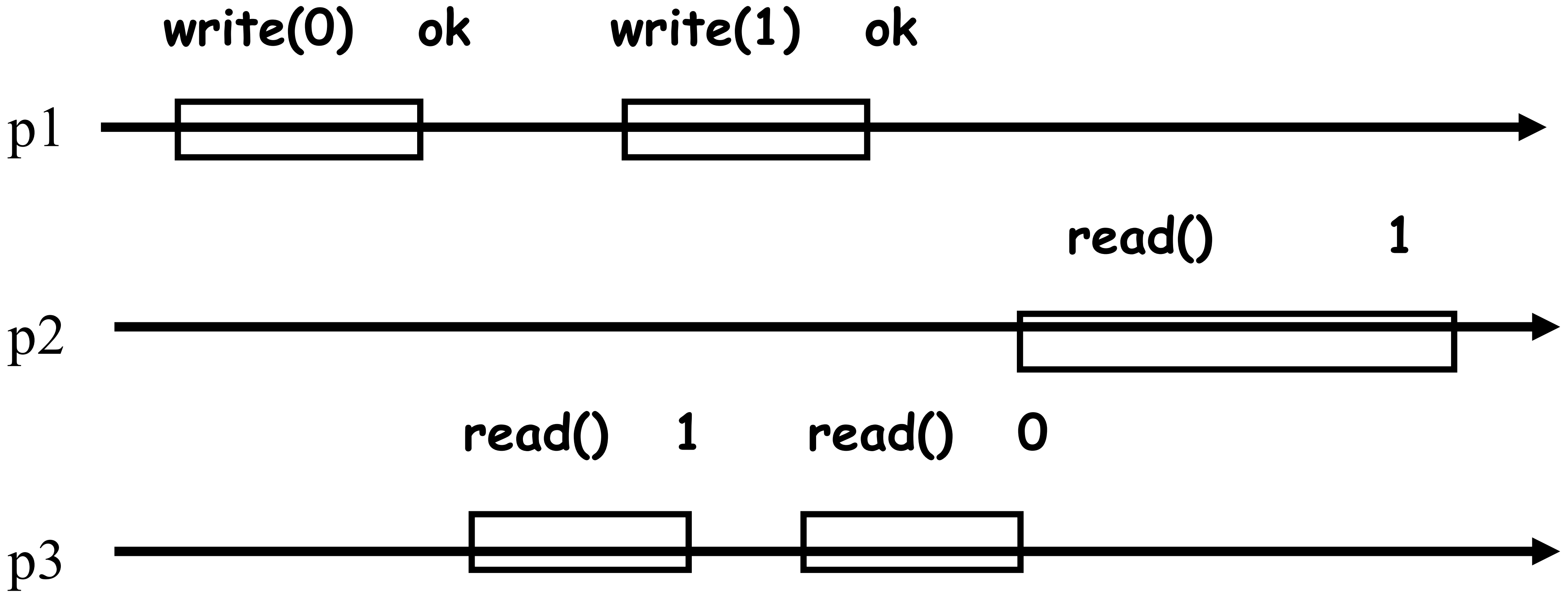
  (assuming one writer)

- Atomic registers: the operations can be totally ordered, preserving legality and precedence (linearizability)
  - ✓≈ if read1 returns v, read2 returns v', and read1 precedes read2, then write(v') cannot precede write(v)

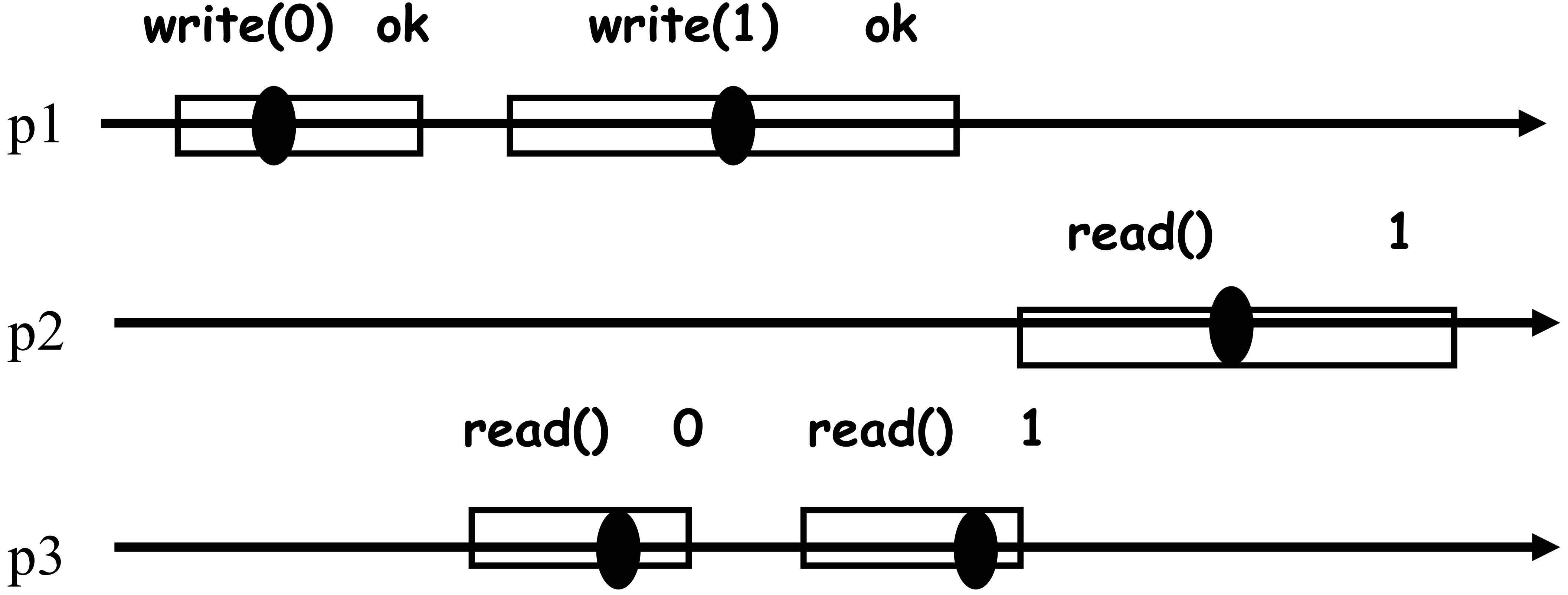
Safe register



Regular register



Atomic register



Quiz 1: relaxing atomicity?

- Show that the original Lamport's Bakery algorithm works even when all base registers are safe?

Bakery [Lamport’74,original]

```
// initialization
flag: array [1..N] of bool = {false};
label: array [1..N] of integer = {0}; //assume no bound

// code for process i that wishes to enter CS
flag[i] = true; //enter the doorway
label[i] = 1 + max(label[1], ..., label[N]); //pick a ticket
flag[i] = false; //exit the doorway
for j=1 to N do {
          while (flag[j]); //wait until j is not in the doorway
          while (label[j]≠0 and (label[j],j)<<(label[i],i));
          // wait until j is not “ahead”
}
…
// critical section
…
label[i] = 0; // exit section
```

Space of registers

- Values: from binary (V={0,1}) to multi-valued
- Number of readers and writers: from 1-writer 1-reader (1W1R) to multi-writer multi-reader (NWNR)
- Safety criteria: from safe to atomic

1W1R binary safe registers can be used to implement
an NWNR multi-valued atomic registers!

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), unbounded

V. From 1W1R to 1WNR (multi-valued atomic), unbounded

1WNR binary safe -> 1WNR binary regular

Let p1 be the only writer and 0 be the initial value

Code for process p1:

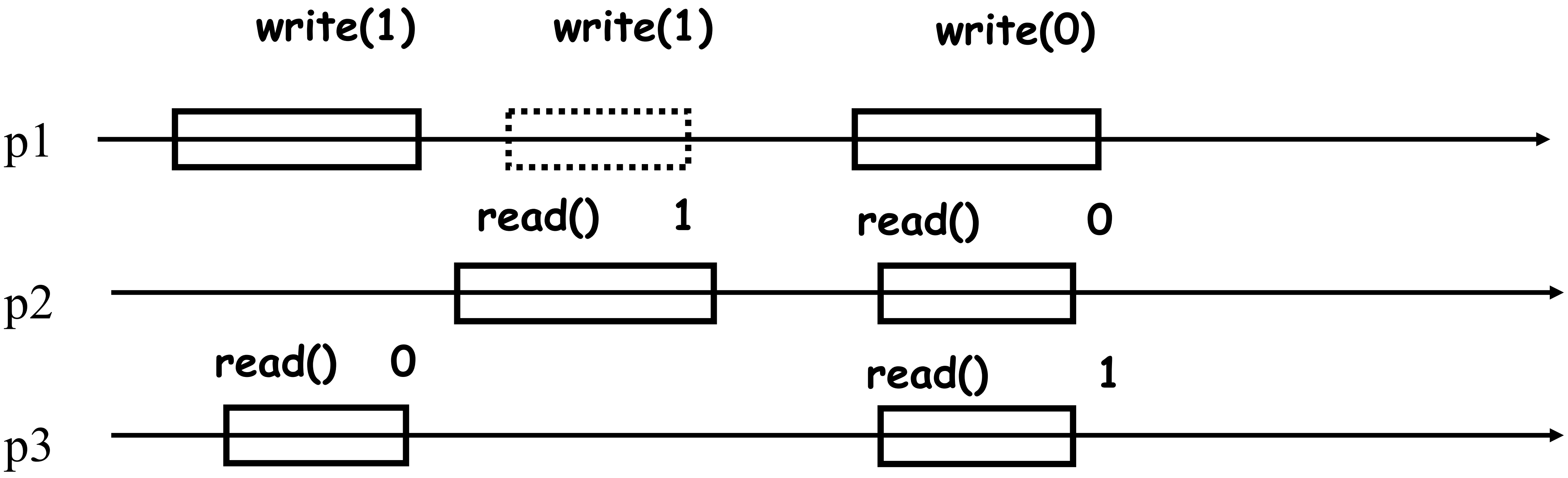
```
initially:
     shared 1WNR safe register R := 0
     lv := 0      \\ last written value

upon write(v)
     if v ≠ lv then
        lv := v
        R.write(v)
     return ok

upon read()
     return R.read()
```

1WNR binary safe -> 1WNR binary regular

- Correctness:
  - ✓R is touched only to change its value
  - ✓ both 0 and 1 are legal values in case of concurrency!



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)

1W1R (binary regular) -> 1WNR (binary regular)

Let p1 be the only writer and 0 be the initial value

Code for process pi:

```
initially:
  shared R[1..N] (1W1R binary regular registers) := 0^N
        // R[i] is written by p1 and read by pi

upon read()
  return R[i].read()

upon write(v)  // if i=1
  for all j do R[j].write(v)
  return ok
```

1W1R (binary regular) -> 1WNR (binary regular)

- Correctness:
  - ✓enough to consider a read that does not overlap with any write
  - ✓the last written value cannot be missed
- Works also for multi-valued and safe registers

What if 1W1R registers are atomic?

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)

Binary -> M-valued (1WNR regular)

Code for process pi:

```
initially:
   shared array R[0,..M-1] of 1WNR registers := [1,0,…,0]

upon read()
   for j = 0 to M-1 do
            if R[j].read() = 1 then return j

upon write(v)  // if i=1
   R[v].write(1)
   for j=v-1 down to 0 do R[j].write(0)
   return ok
```

Binary -> M-valued (1WNR regular)

- Correctness:
  - ✓only the last or concurrently written value can be returned
  - ✓every operation returns in O(M) steps

Quiz 2: what if?

```
Code for process pi:
    initially:
                shared array R[0,..M-1] of 1WNR registers := [1,0,…,0]

    upon read()
                for j = 0 to M-1 do
                     if R[j].read() = 1 then return j

    upon write(v)  // if i=1
                R[v].write(1)
    for j=0 to v-1 do R[j].write(0)
                return ok
```

Quiz 3: what if?

```
Code for process pi:
   initially:
            shared array R[0,..M-1] of 1WNR registers := [1,0,…,0]

   upon read()
            for j = 0 to M-1 do
                if R[j].read() = 1 then return j

  upon write(v)  // if i=1
            for j=v-1 down to 0 do R[j].write(0)
            R[v].write(1)
            return ok
```

© P. Kuznetsov

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)

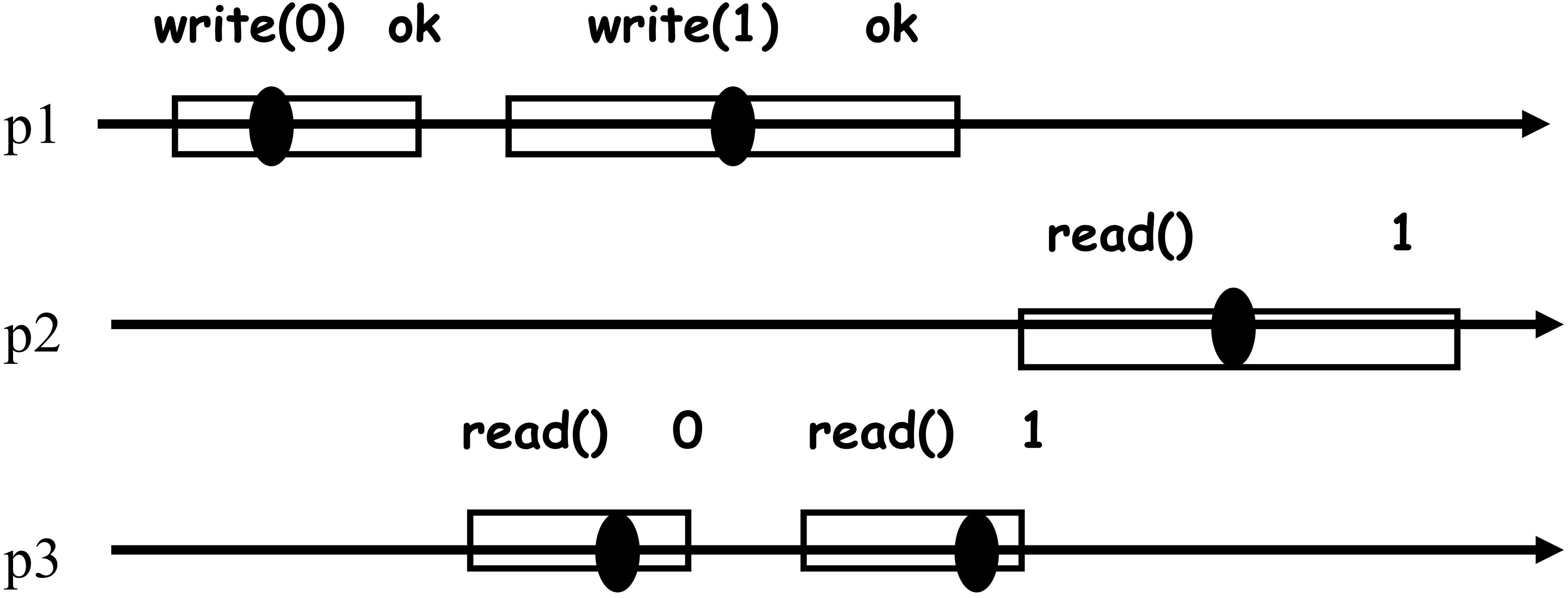
Atomic registers

A register is *atomic* if every history it produces is linearizable

Informally, the complete operations (and some incomplete operations) are seen as taking effect instantaneously at some time between their invocations and responses

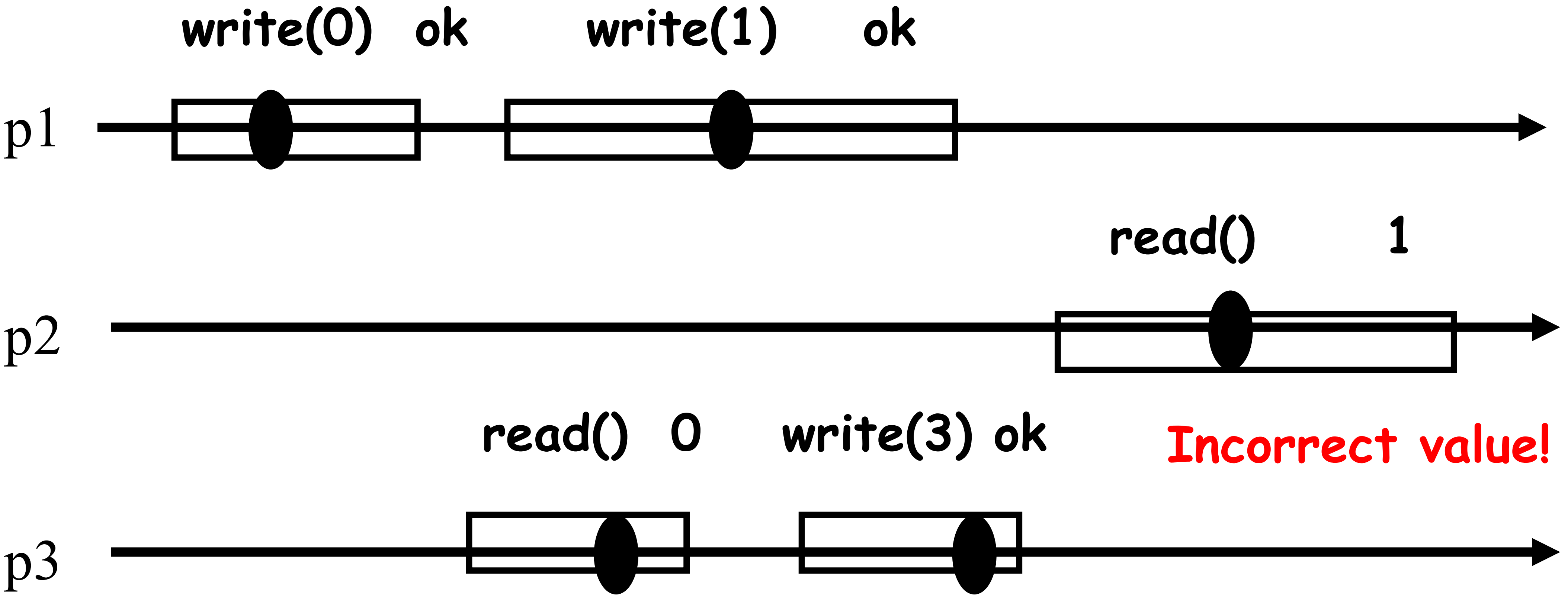
(The operations are *atomic*)

Atomic?



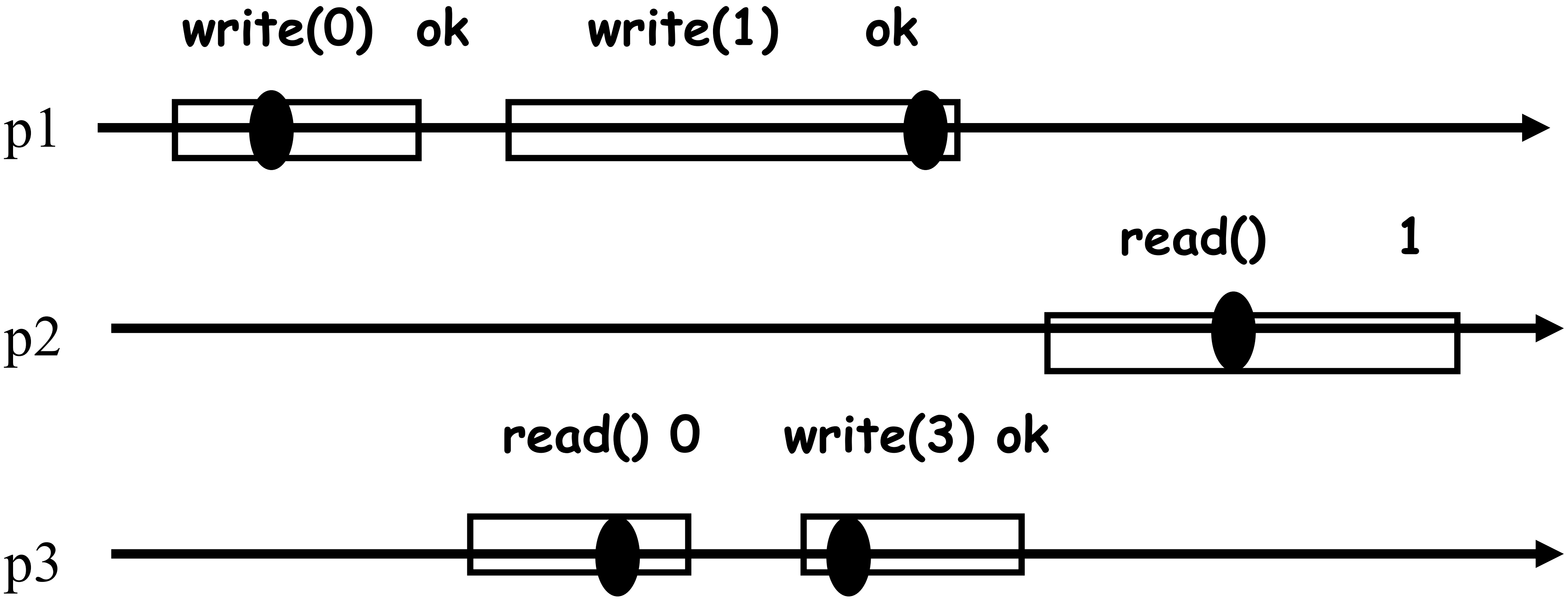
© P. Kuznetsov

Atomic?

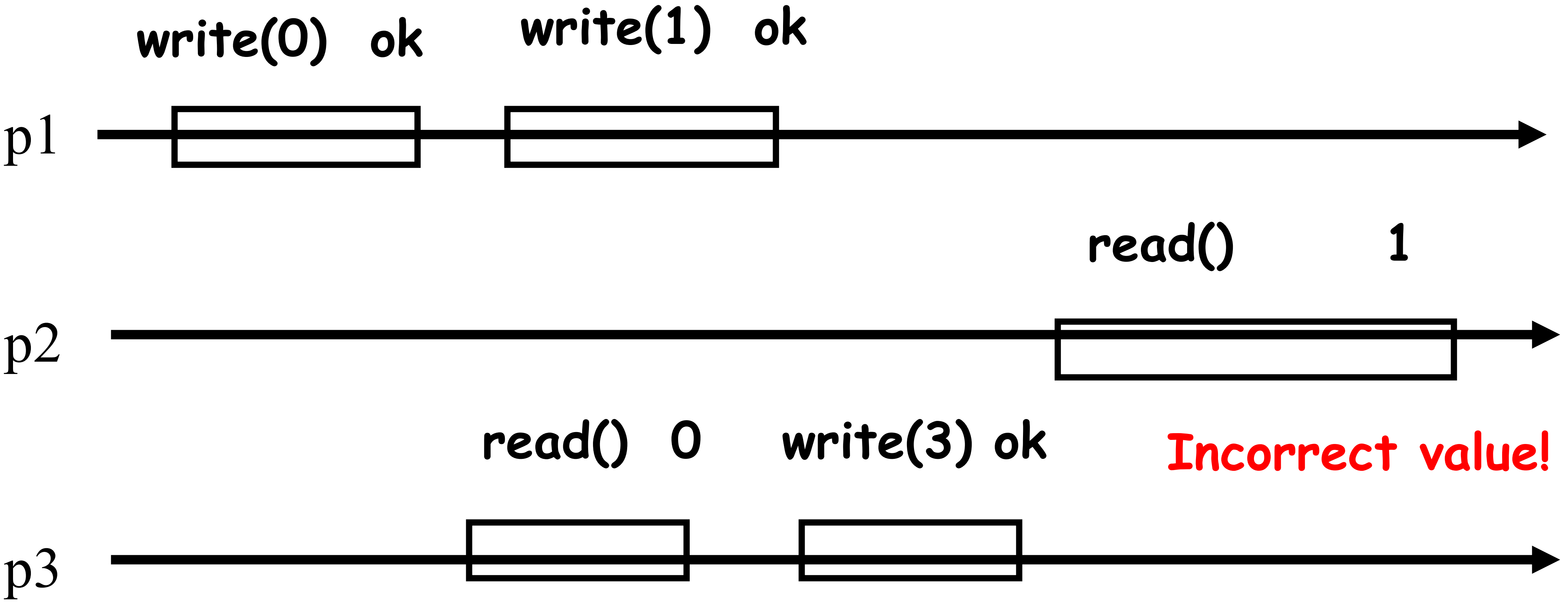


Incorrect value!

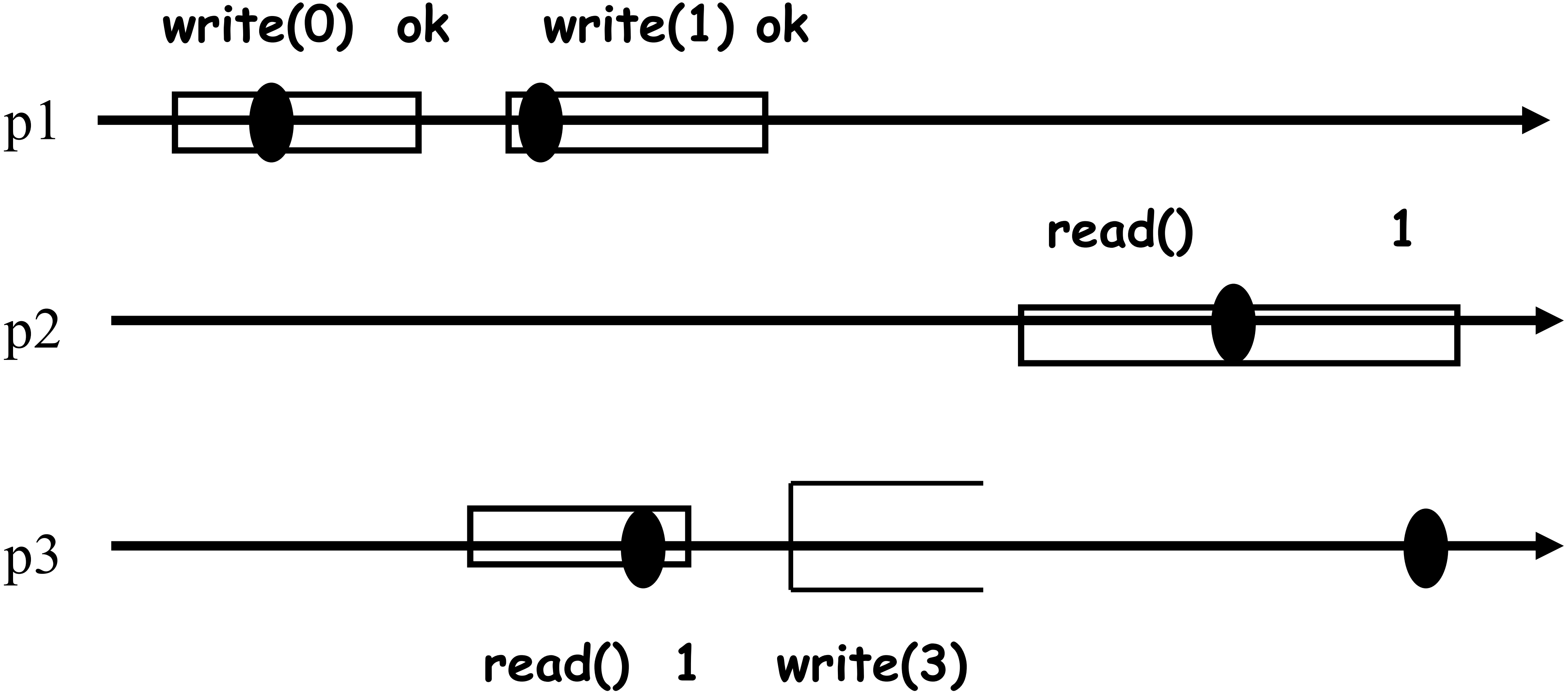
Atomic?



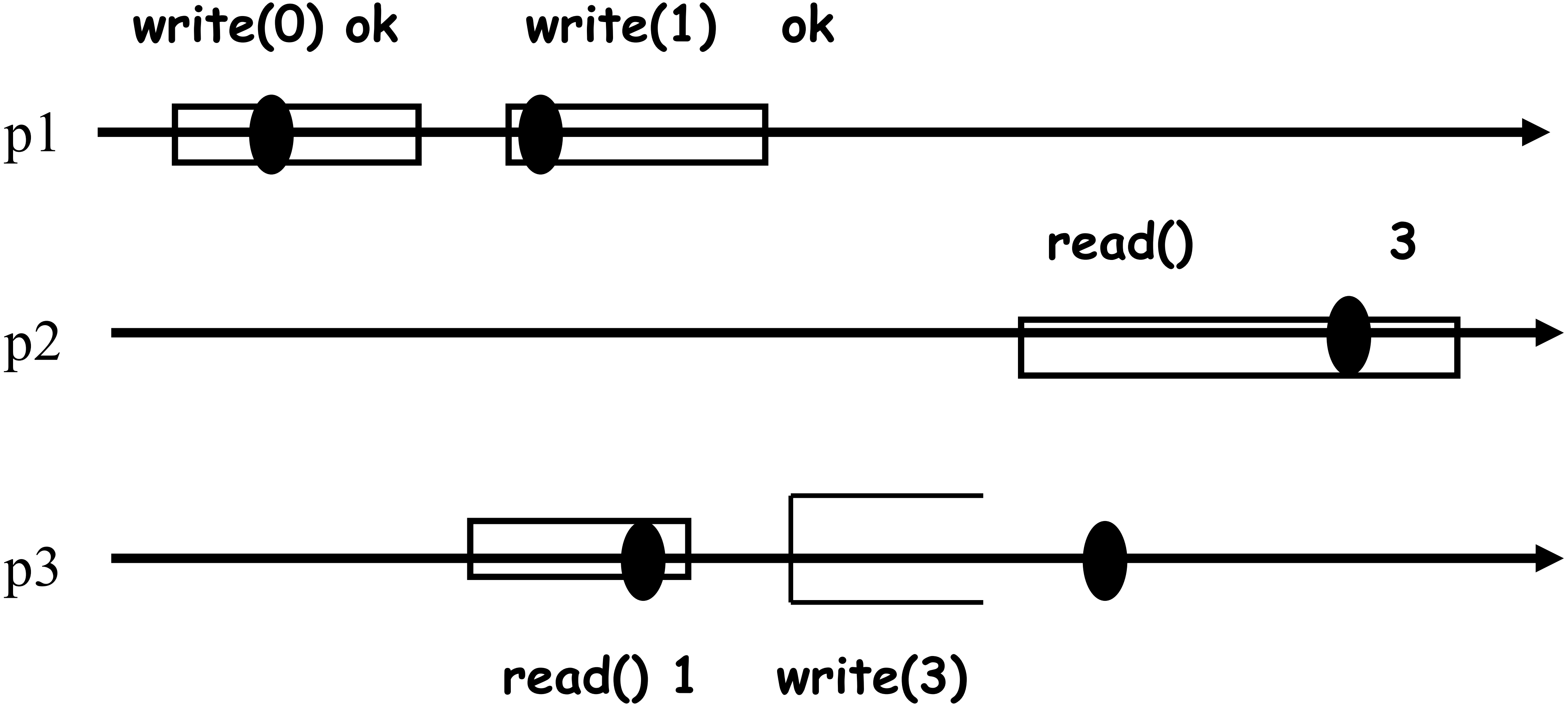
Atomic?


write(0) ok
write(1) ok
p1
read() 1
p2
read() 0
write(3) ok
Incorrect value!
p3

Atomic?

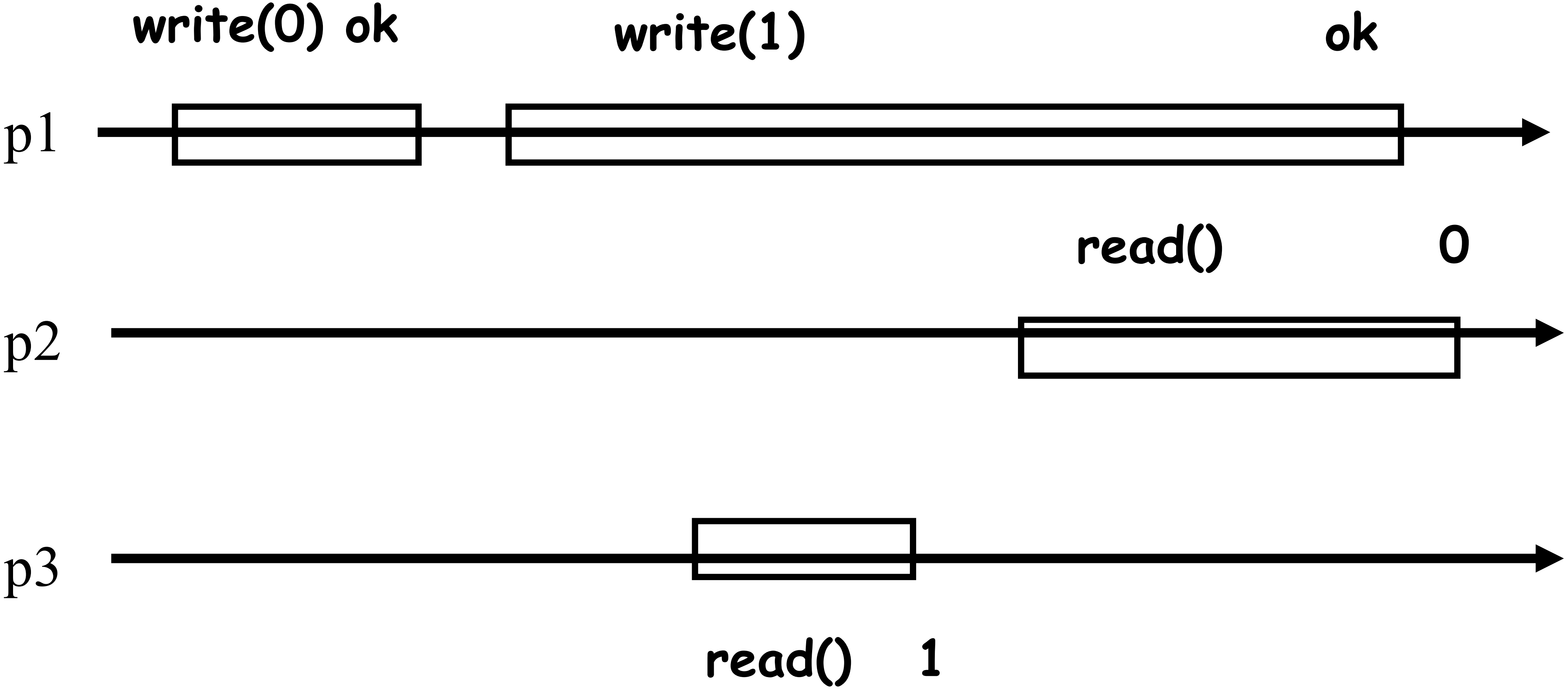


Atomic?

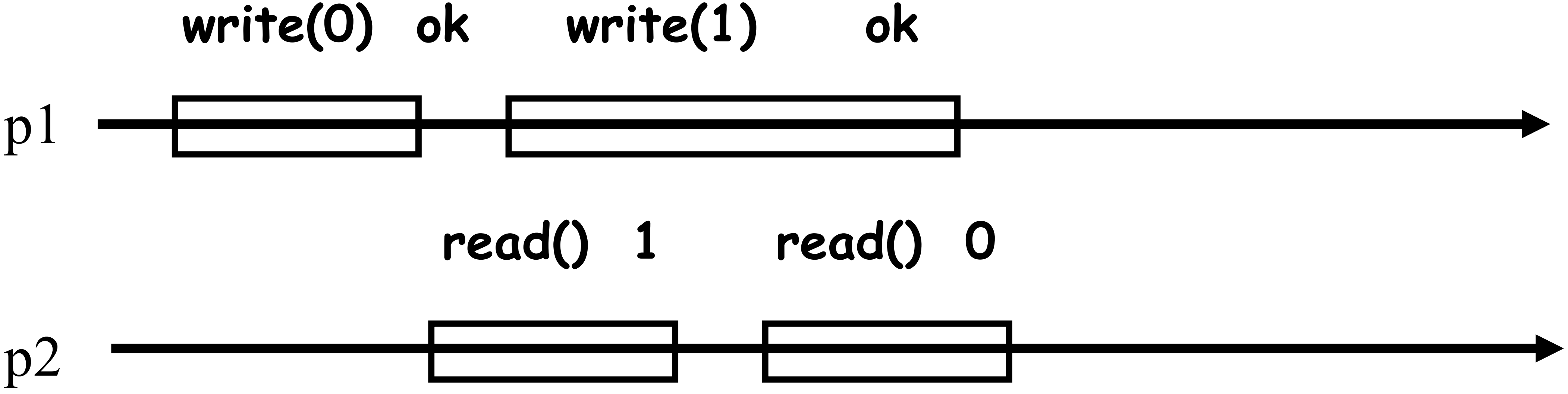


© P. Kuznetsov

Atomic?



From 1W1R regular to 1W1R atomic



Write a timestamp?

1W1R regular -> 1W1R atomic

Code for process pi:

```
initially:
     shared 1W1R regular register R := 0
     local variables t := 0, x := 0

upon read()
     (t’,x’) := R.read()
     if t’ > t  then t:=t’; x:=x’;
     return(x)

upon write(v)   // if i=1
     t:=t+1
     R.write(t,v)
```

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)

Transformations-I

From safe to regular (binary 1W1R)

- Writer touches shared memory only to change
- A concurrent read is allowed to return any value (0 or 1)

Transformations-II

From one-reader to multiple-reader (regular binary or multi-valued)

- Every reader is assigned a dedicated register to read
- Writer writes in all
- Reader reads its own register

Transformations-III

From binary to M-valued (1WNR regular)

- Every *value* in {0,…,M-1} is assigned a dedicated 1WNR register
- Write(v) sets R[v] to 1 and sets R[v-1] … R[0] to 0
- Read returns the smallest v such that R[v]=1

Transformation IV (unbounded)

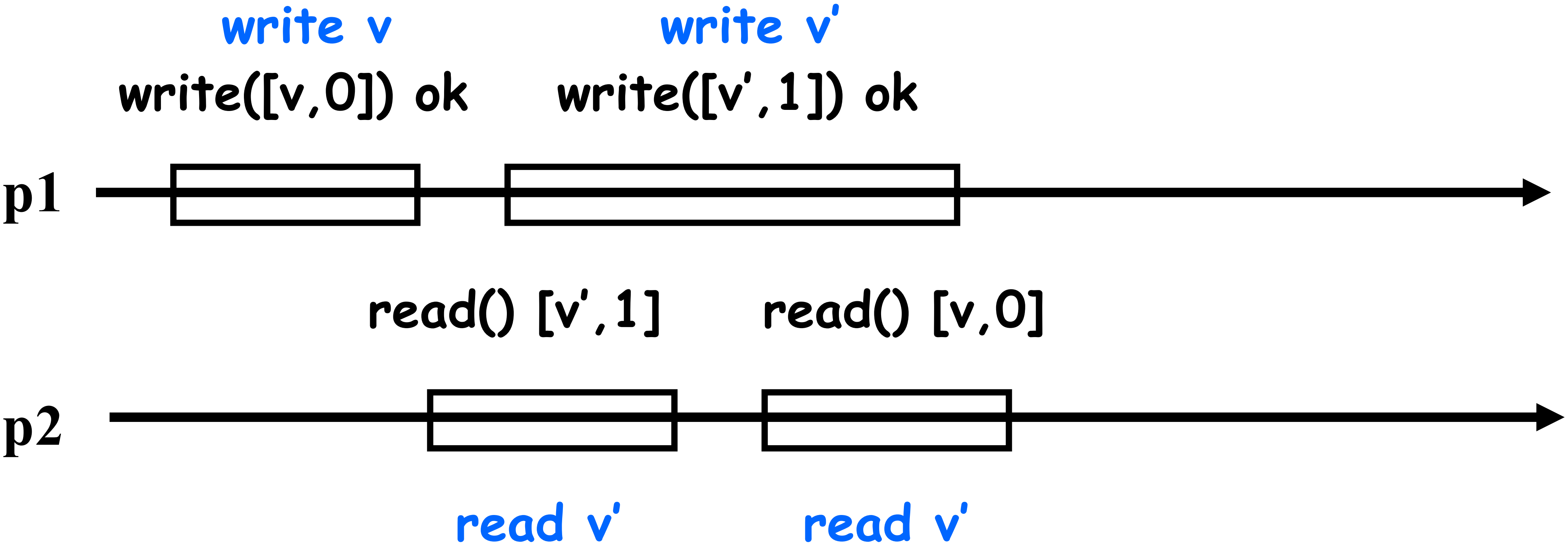
From regular to atomic (1W1R multi-valued)

- Write a timestamp with a value
- The reader returns the latest value and ignores the old one

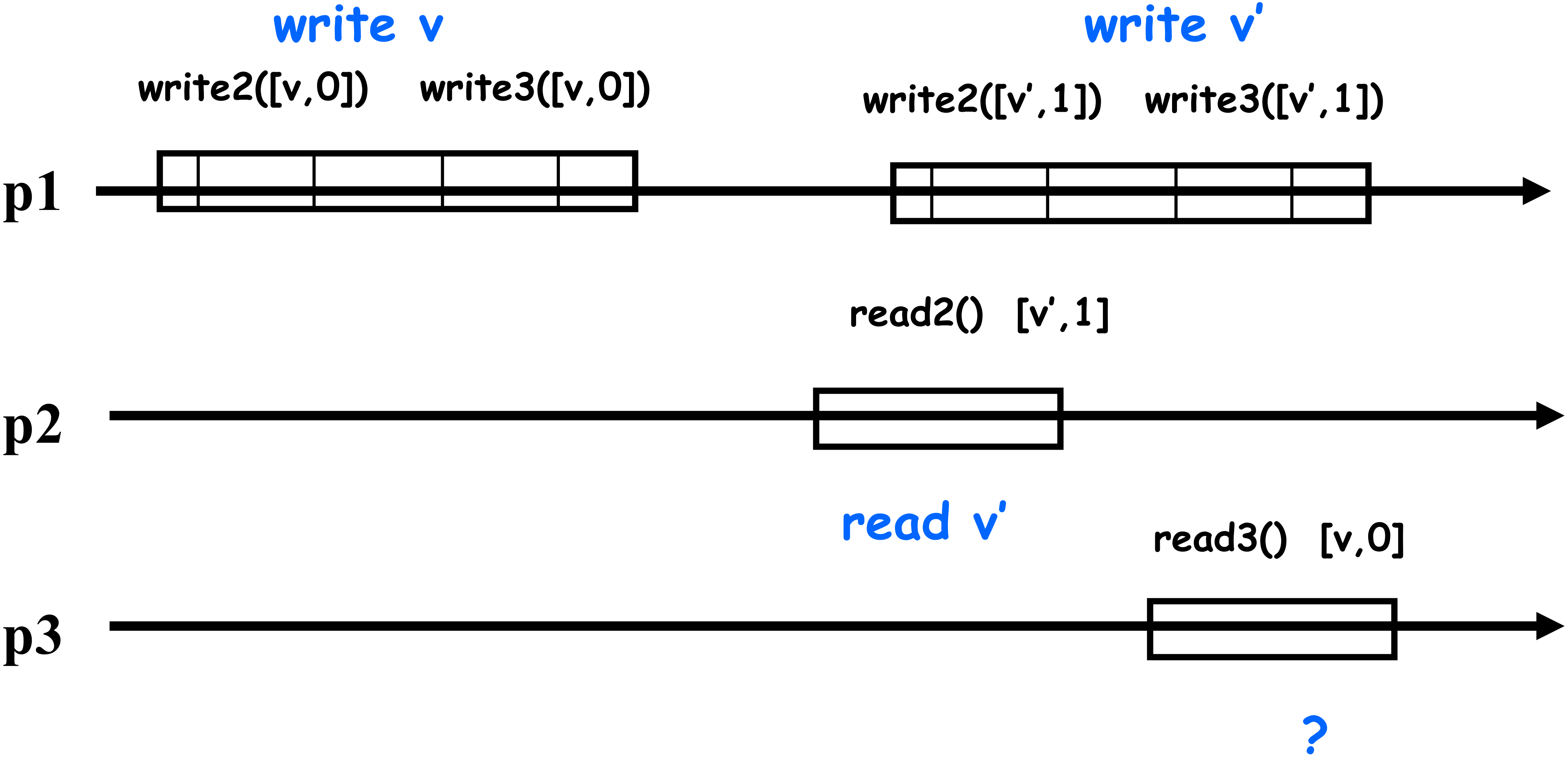
Transformation IV

From regular to atomic (1W1R multi-valued)

- Write a timestamp with a value
- The reader returns the latest value and ignores the old one

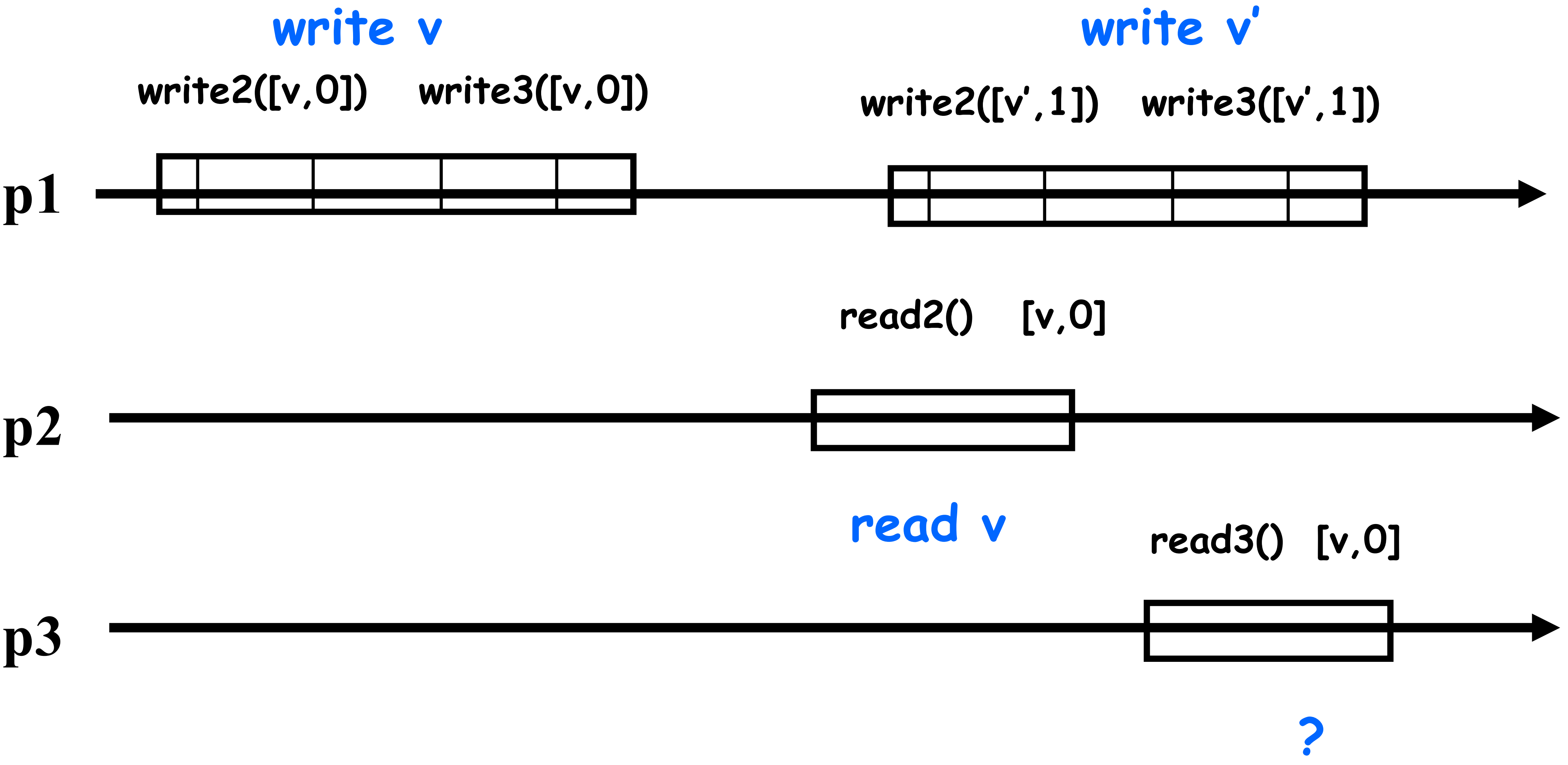


Multiple readers?



Multiple readers?

Does not work either!



Transformation V (unbounded)

```
shared:
   matrix RR[1..N][1..N] of 1W1R atomic registers := 0^NxN
   // for all i,j, RR[i][j] is read by pi and written by pj

   array WR[1..N] of 1W1R atomic registers := 0^N
   // for all i WR[i] is written by p1 and read by pi




upon write(v)   // code for p1
   ts:=ts+1
   for all j do WR[j].write([v,ts])
   return ok
```

A bounded construction coming soon…

Transformation V

```
upon read() // code for pi
   for all j=1,…,N  do (t[j],x[j]) := RR[i][j].read()
   (t[0],x[0]) := WR[i].read()
    (tmax,xmax) := highest(t,x)
    for all j do RR[j][i].write([tmax,xmax]);
    return(xmax)
```

(Here highest(t,x) computes the value x[j] written with the highest timestamp t[j])

Transformation V: correctness

If read1 returns v and read1 precedes read2 then read2 cannot return a value that is older than v – sufficient for proving that a one-writer regular register is linearizable

- What if the reader does not write?
- What about multiple writers?

Appendix: “Quasi-bounded” construction: 1WNR regular + atomic bit->1WNR atomic

```
initially:
  shared 1WNR atomic bit WFLAG := false
  shared 1WNR regular registers R1,R2:=initial

upon write(v)
  WFLAG:=true
  R1.write(v)
  WFLAG:=false
  R2.write(v)

upon read()
  val:=R1
  wf:=WFLAG
  if !wf then return(val)
  val:=R2
  return(val)
```

© P. Kuznetsov

Appendix: bounded construction: 1WNR regular->1WNR atomic

Replace

- WFLAG with a ternary LEVEL registers (0,1,2)
- arrays WC[0..N] and RC[0..N] to exchange “signals” between the processes

```
upon write(v)
   LEVEL:=1  // incomplete write
   R1.write(v)
   LEVEL:=2
   LEVEL:=0  // “safe” to return the value read in R1
   R2.write(v)
   for j=1,...,N do
   lr := RC[j];
   WC[j] := !lr; // a new value for p_j is written
```

Appendix: bounded construction: 1WNR regular->1WNR atomic

```
Upon read() // for reader p_i
   val := R1
   lw:=WC[i]
   if lw != RC[i] then
      FC[i]:=false;
      RC[i] := lw; // the new value is read
   case LEVEL:
      0: return(val)
      2: FC[i]:=true; return(val)
      1: for j=1,...,N do
   lr := RC[j];
   lf := FC[j];
   lw := WC[j];
            if (lr = lw) && lf then
      FC[i] := true
            return (val)
          val := R2;
          return(val);
```

Chapter 8 of lecture notes for the code of Read and the proofs.
Bug detection is welcome!
More on signaling in the next lecture…