# DISTRIBUTED REGISTERS

## DISTRIBUTED SYSTEMS

## Master of Science in Cyber Security

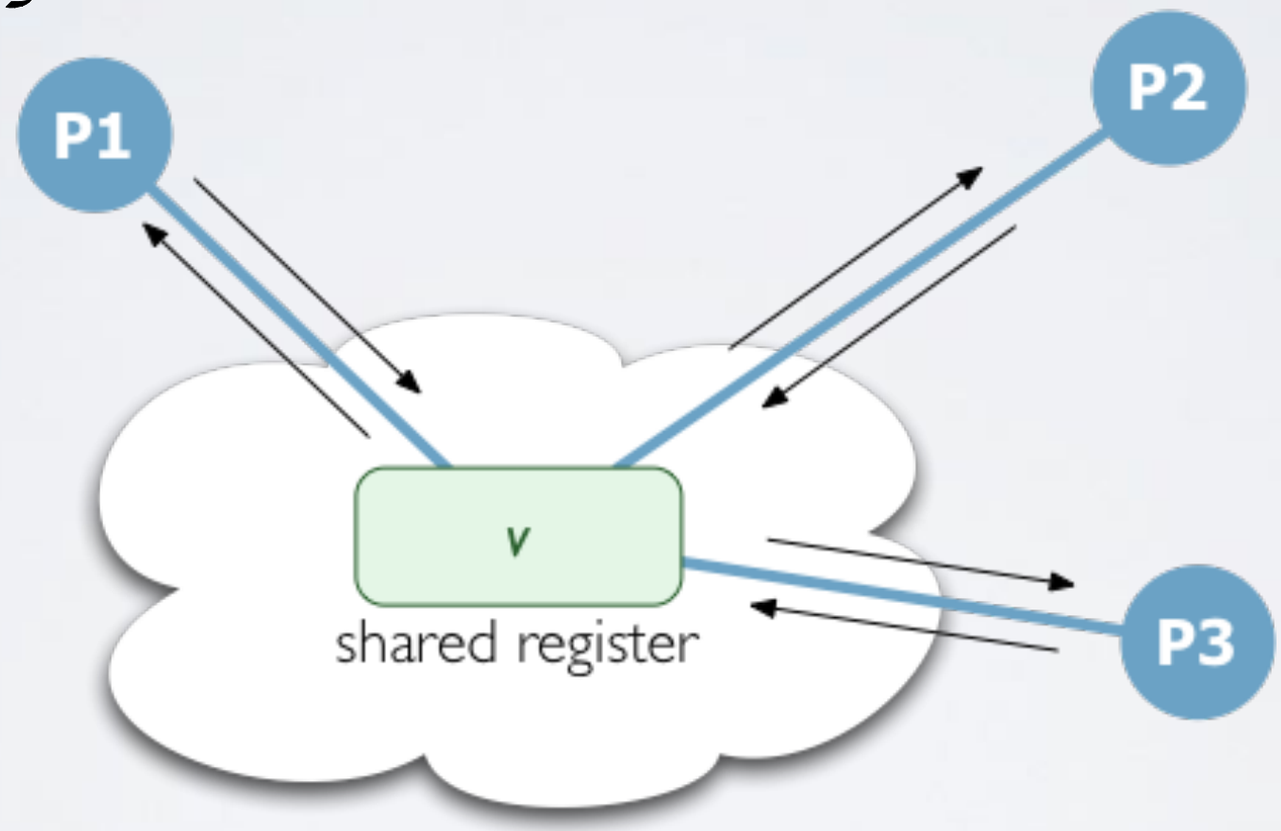

SAPIENZA
UNIVERSITÀ DI ROMA

CIS SAPIENZA
CYBER INTELLIGENCE AND INFORMATION SECURITY

# REGISTER: DEFINITION

A register is a shared variable accessed by processes through read and write operations

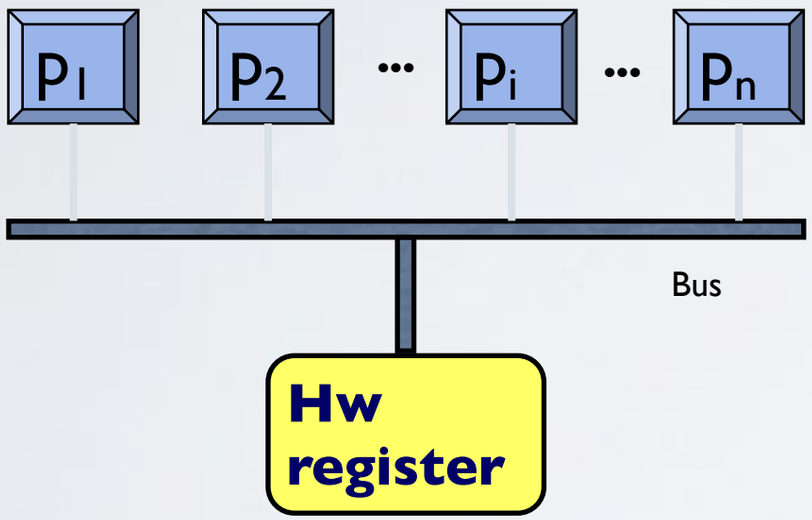
2 operations



# REGISTER ABSTRACTION

## Multiprocessor machine

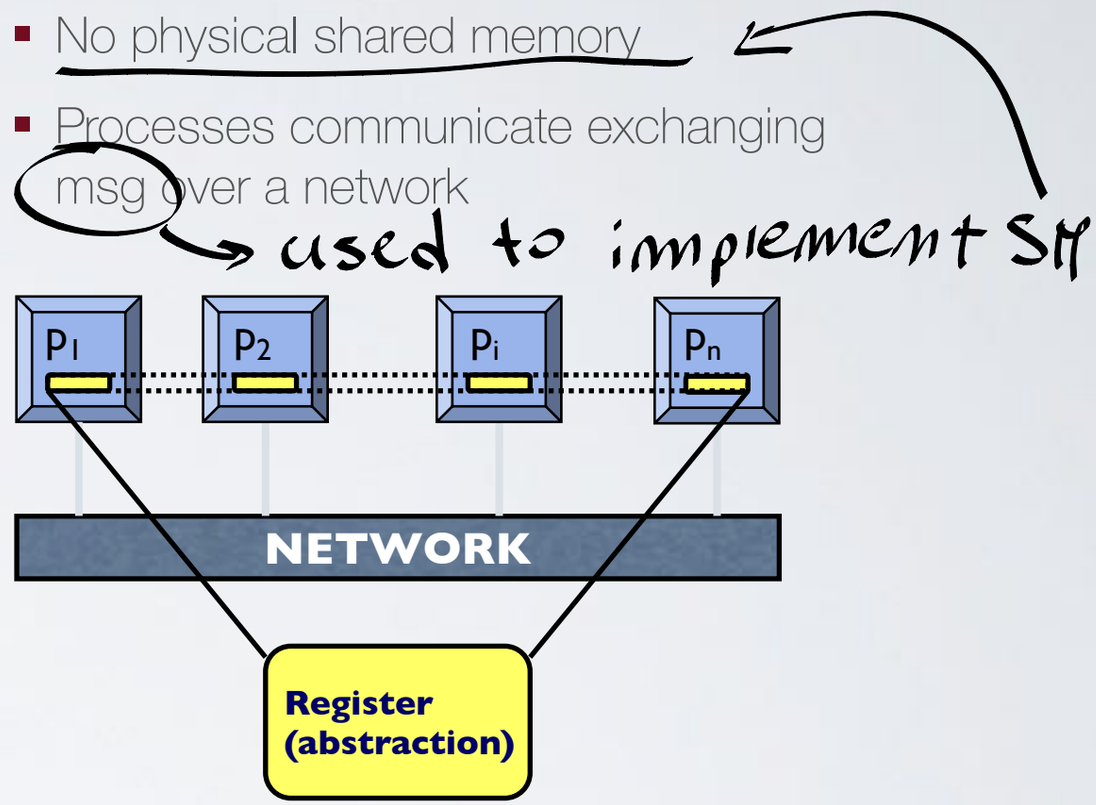
- Processes typically communicate through registers at hardware level




The set of these registers constitute the physical memory

## Distributed message passing system

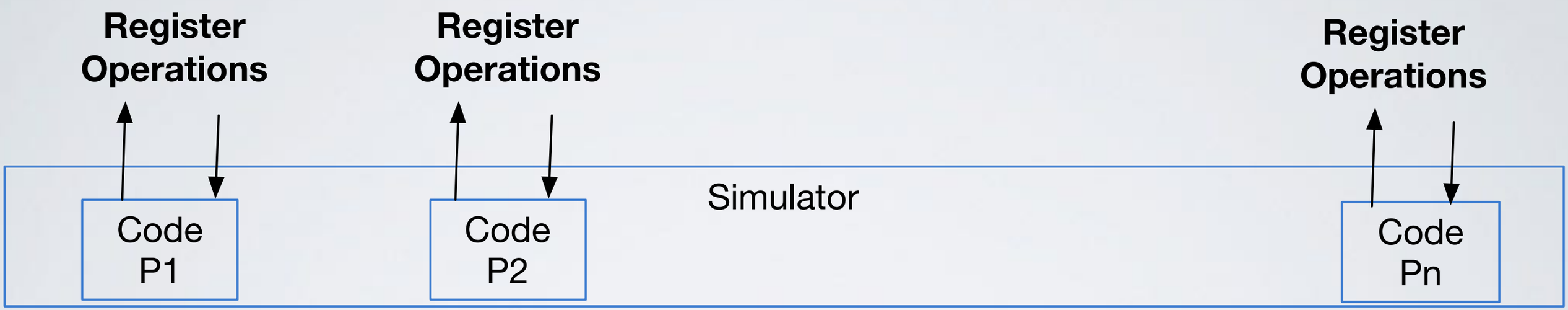
- No physical shared memory
- Processes communicate exchanging msg over a network

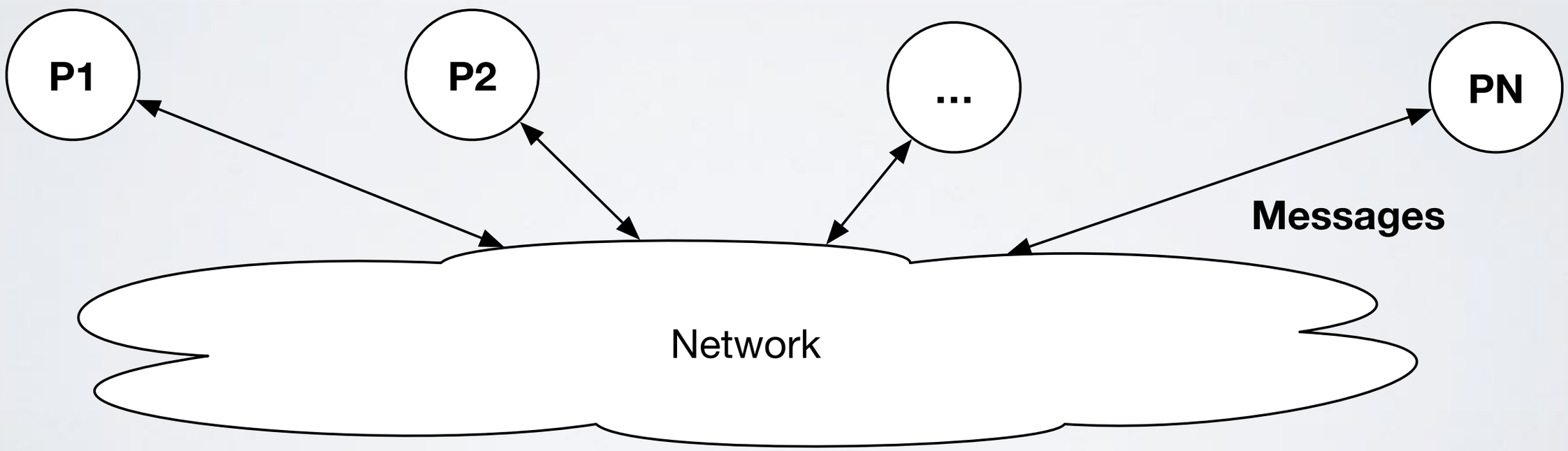



Register abstraction support the design of distributed solution, by hiding the complexity of the underlying message passing system and the distribution of the data

# REGISTER SIMULATION

at the end memory is synchronized

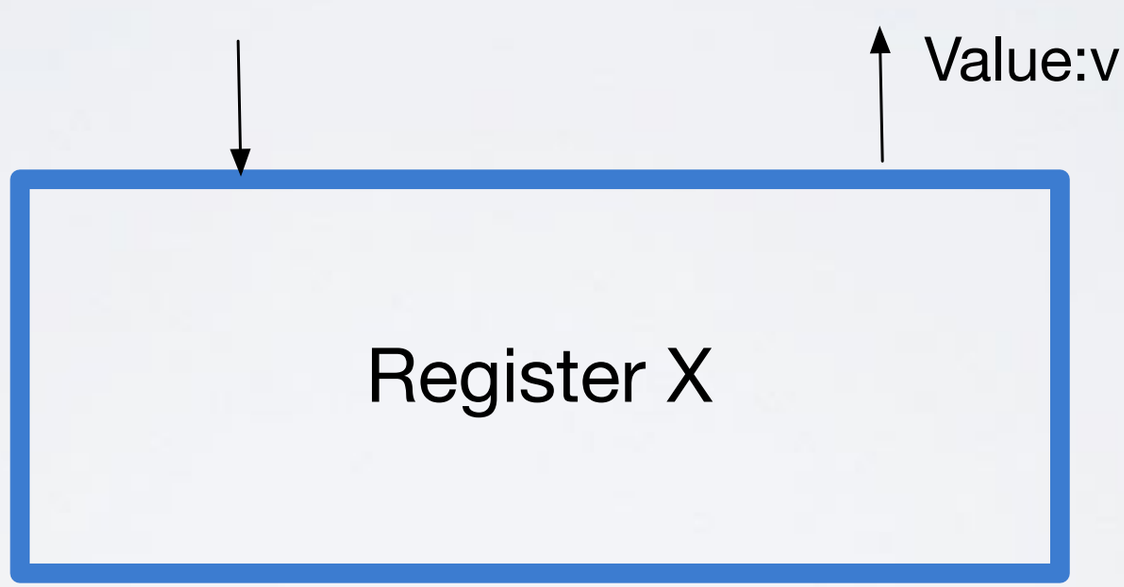


# REGISTER OPERATIONS

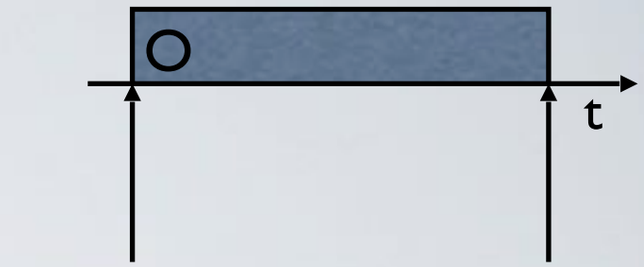
A process accesses a register through:

- **Read operation**, read()→v: it returns the "current" value *v* of the register; this operation does not modify the content of the register;
- **Write operation**, write(v): it writes the value *v* in the register and returns *true* at the end of the operation



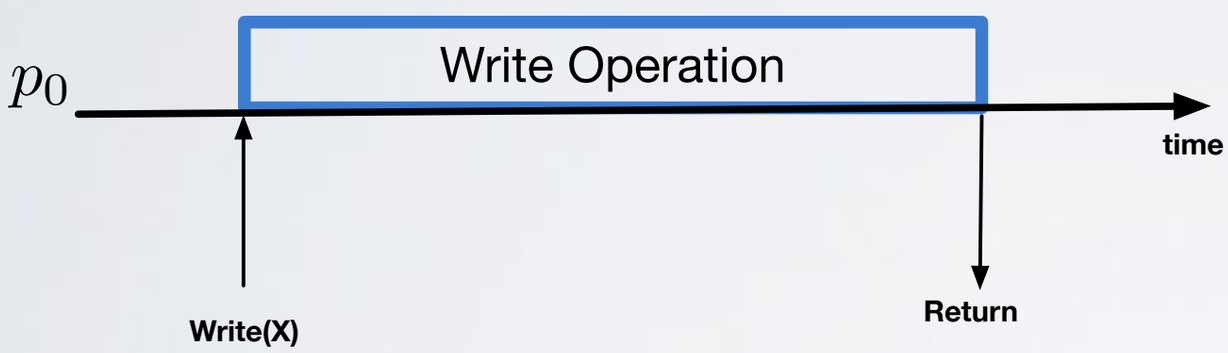

Each operation starts with an invocation and terminates when the corresponding response is received

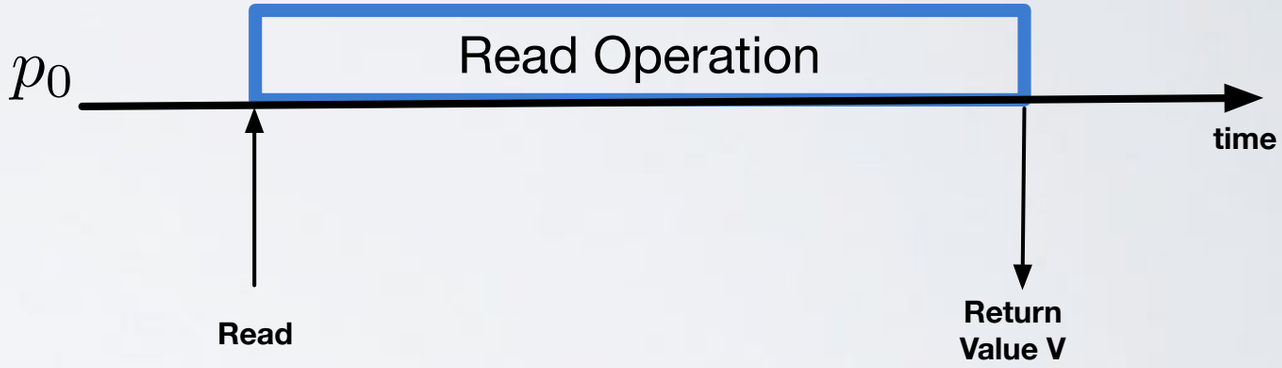
# OPERATIONS




Every operation is characterized by two events:

- Invocation → not atomic → take time ⇒ you have to wait until the return
- Return
  - Confirmation for the write operation
  - A value for the read operation



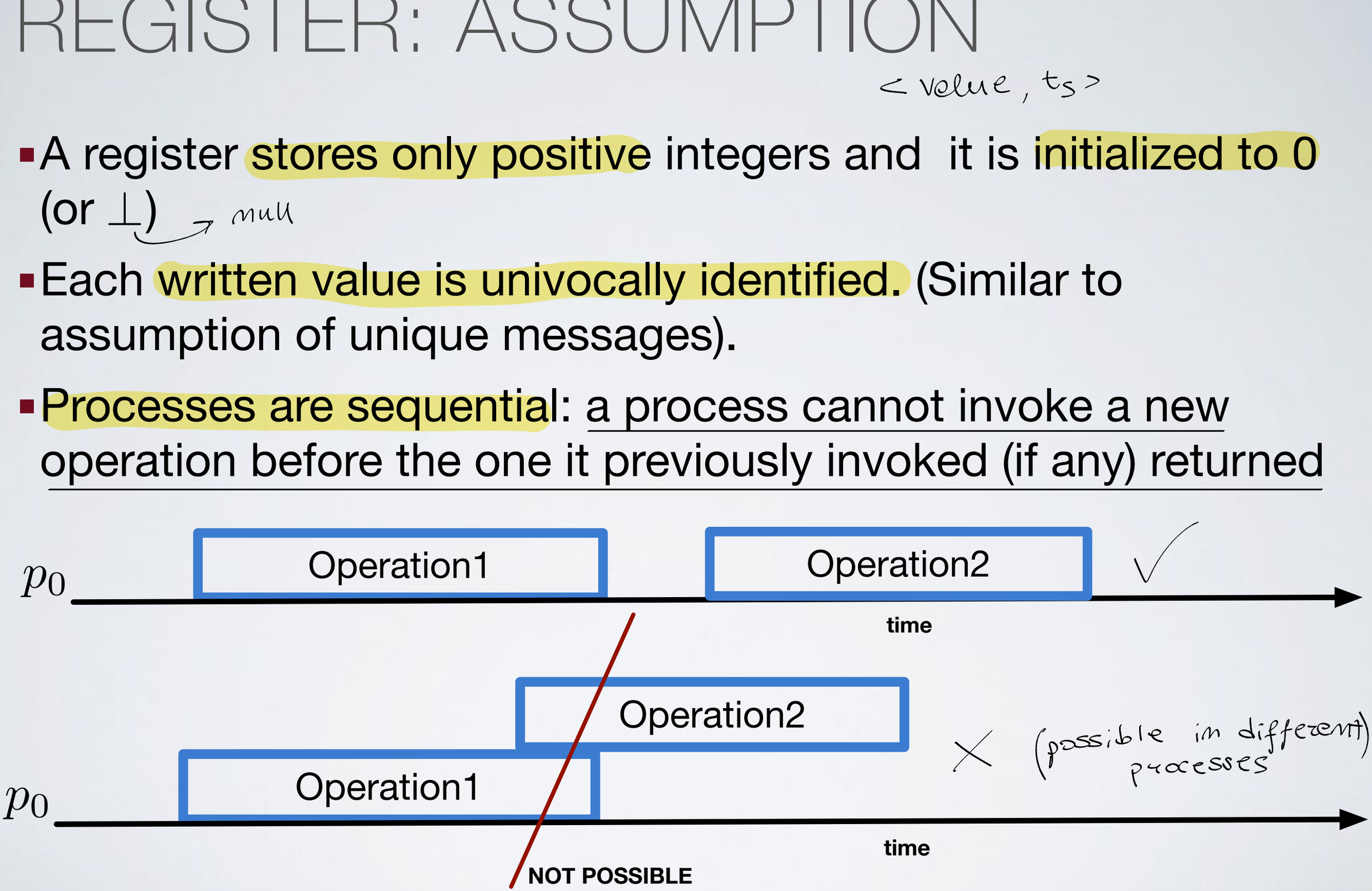

Each of these events occur at a single indivisible point of time

An operation is complete if both the invocation and the return events occurred.

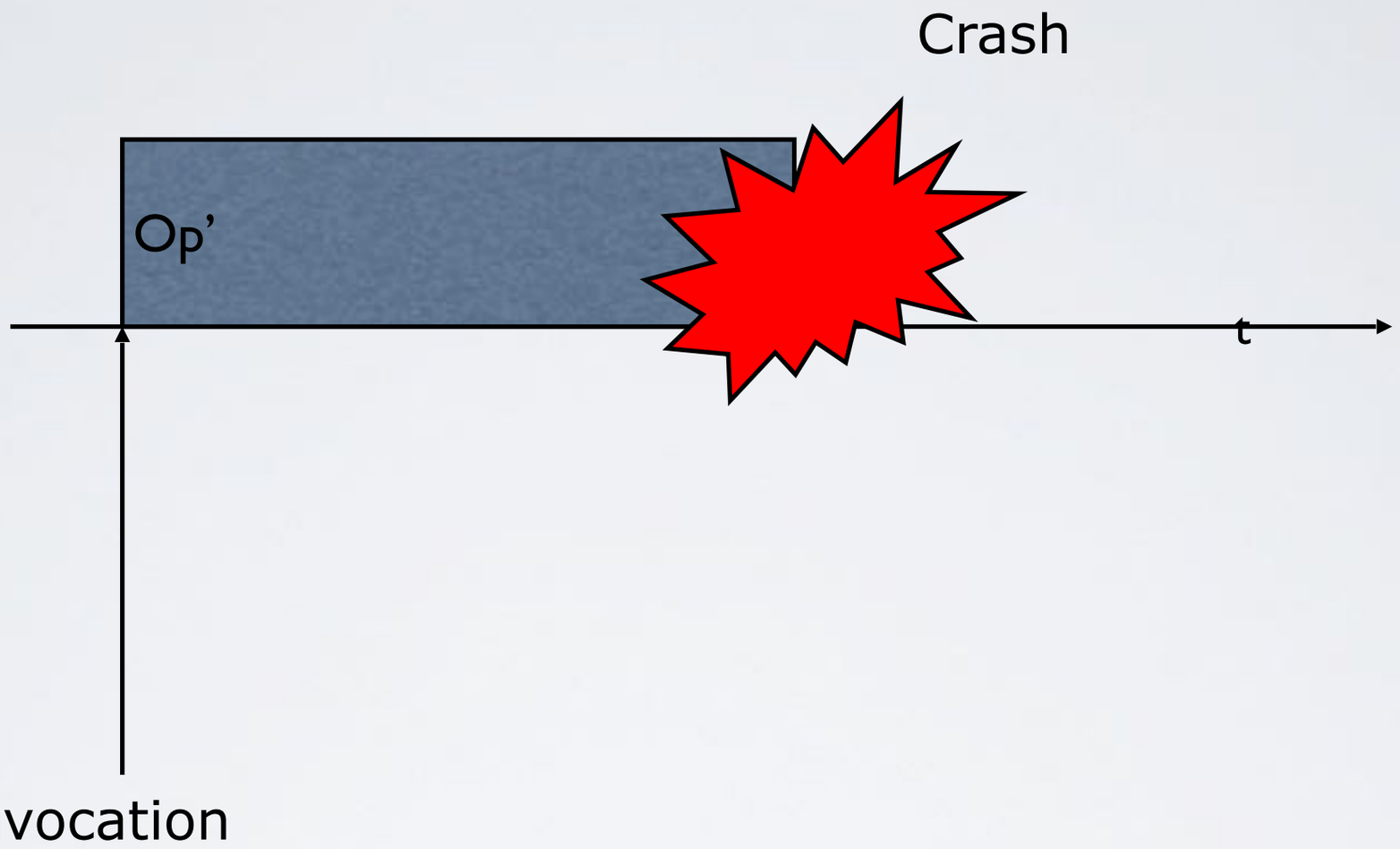
# REGISTER: ASSUMPTION

<value, ts>

- A register stores only positive integers and it is initialized to 0 (or $\perp$) → null
- Each written value is univocally identified. (Similar to assumption of unique messages).
- Processes are sequential: a process cannot invoke a new operation before the one it previously invoked (if any) returned



# FAILED OPERATIONS




A Failed operation is an operation invoked by some process $p_i$ that crashes before obtaining a return.

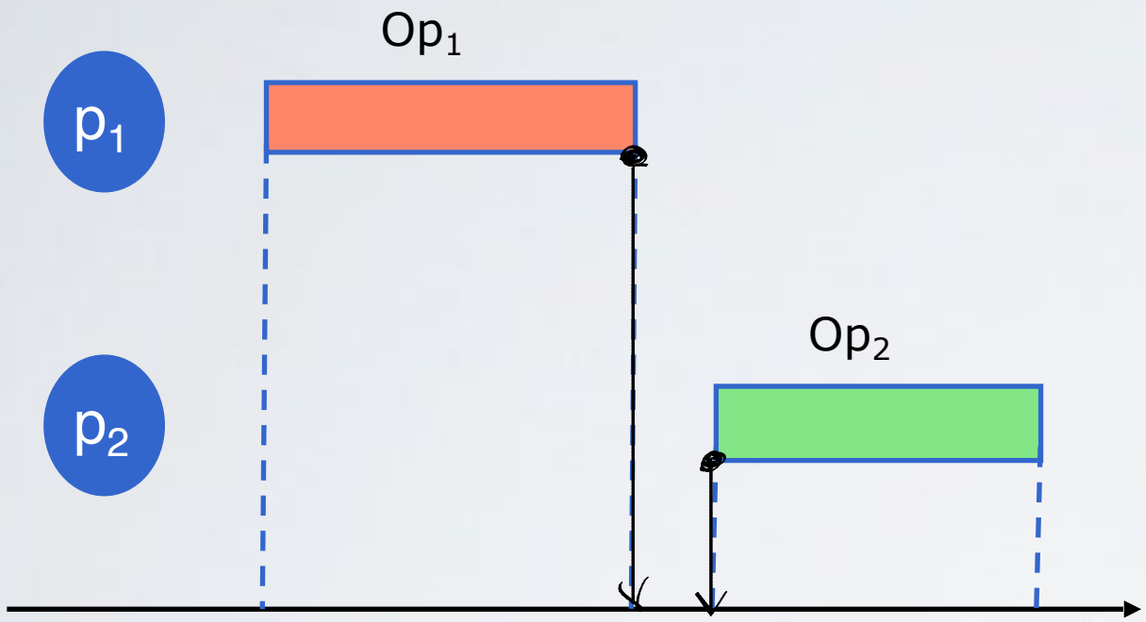
# OPERATION PRECEDENCE

$t_{return} - t_{invocation}$

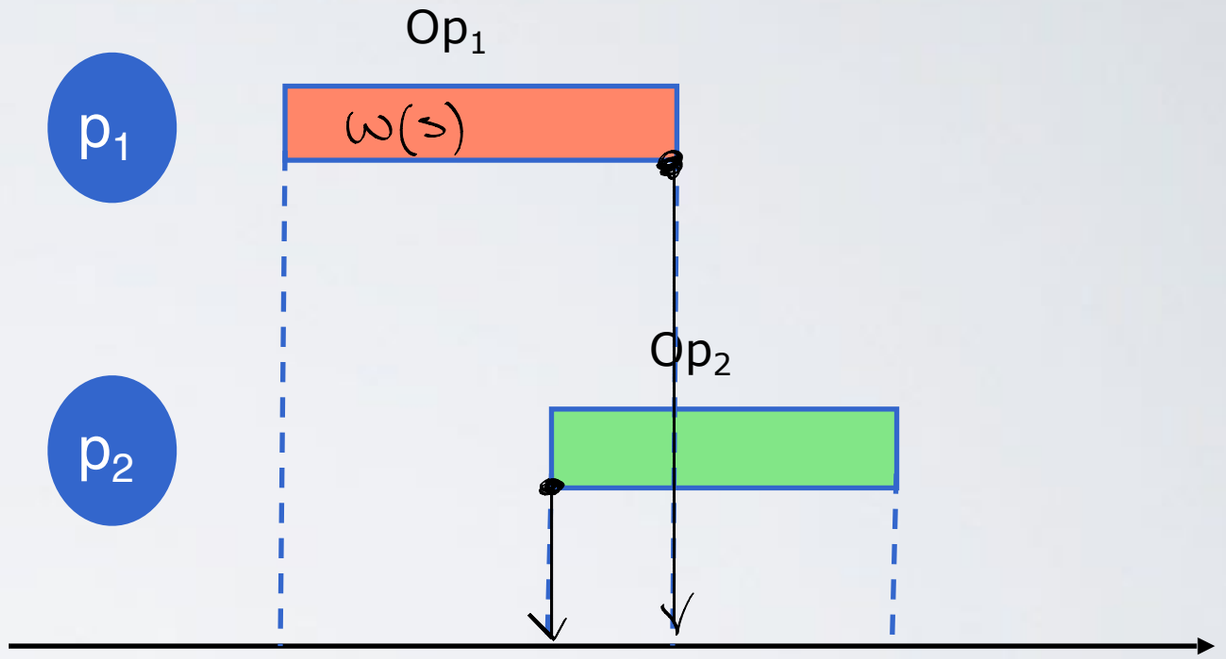
//

- The execution of an operation invoked by a process *p*, is the time interval defined by the invocation event and the return event.
- Given two operations *o* e *o'*, ***o* precedes *o'*** if the return event of *o* precedes the invocation event of *o'*.
- An operation *o* invoked by a process *p* may precede an operation *o'* invoked by *p'* only if *o* completes.
- If no precedence relation between two operations can be defined, they are said to be **concurrent.**

# EXAMPLE

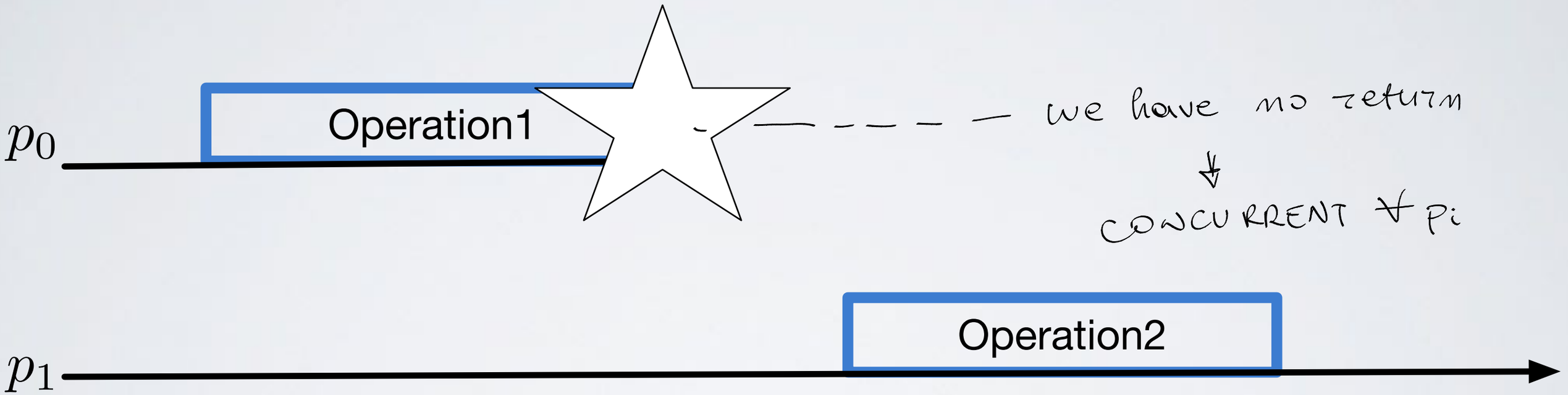



**$Op_1$ precedes $Op_2$**

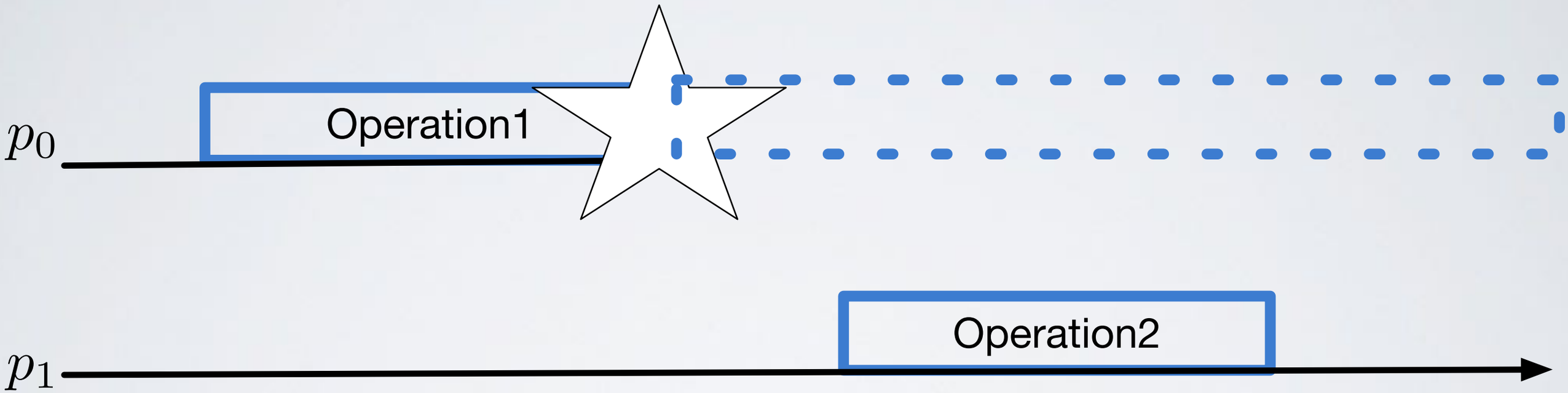



**$Op_1$ is concurrent with $Op_2$**

# EXAMPLE: FAILURES



# EXAMPLE: FAILURES



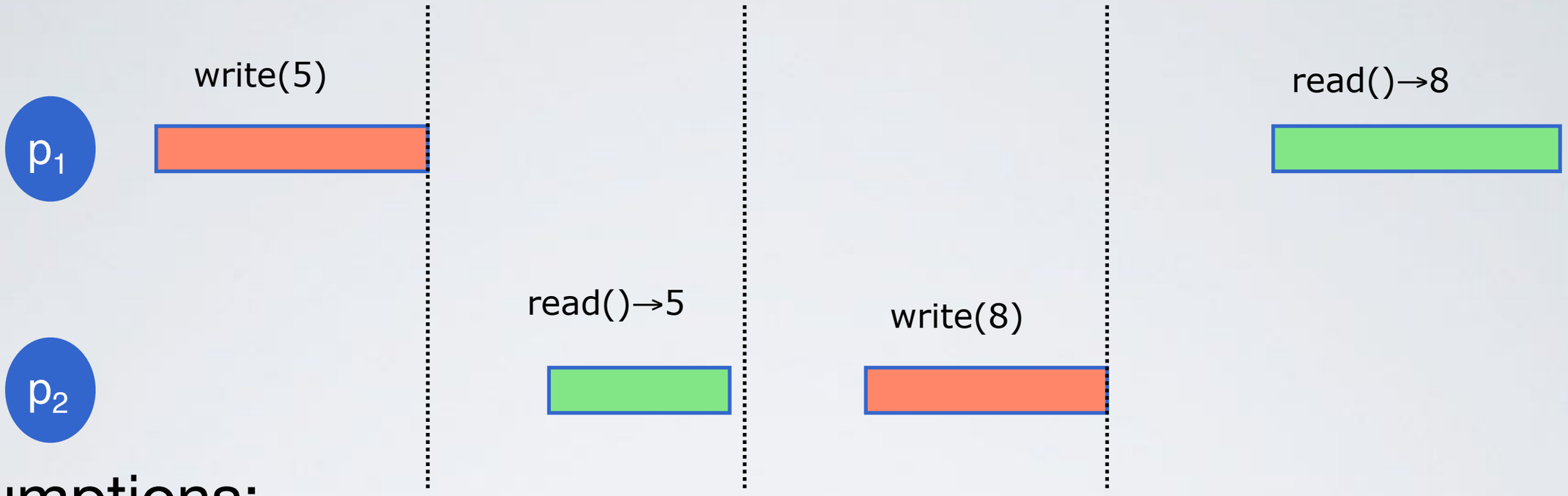

**$Op_1$ never returns.**

**Therefore it is concurrent with $Op_2$.**

# REGISTERS SEMANTIC

read() → problem. when it can release?

# STRICTLY SERIALISED SEMANTIC




## Assumptions:

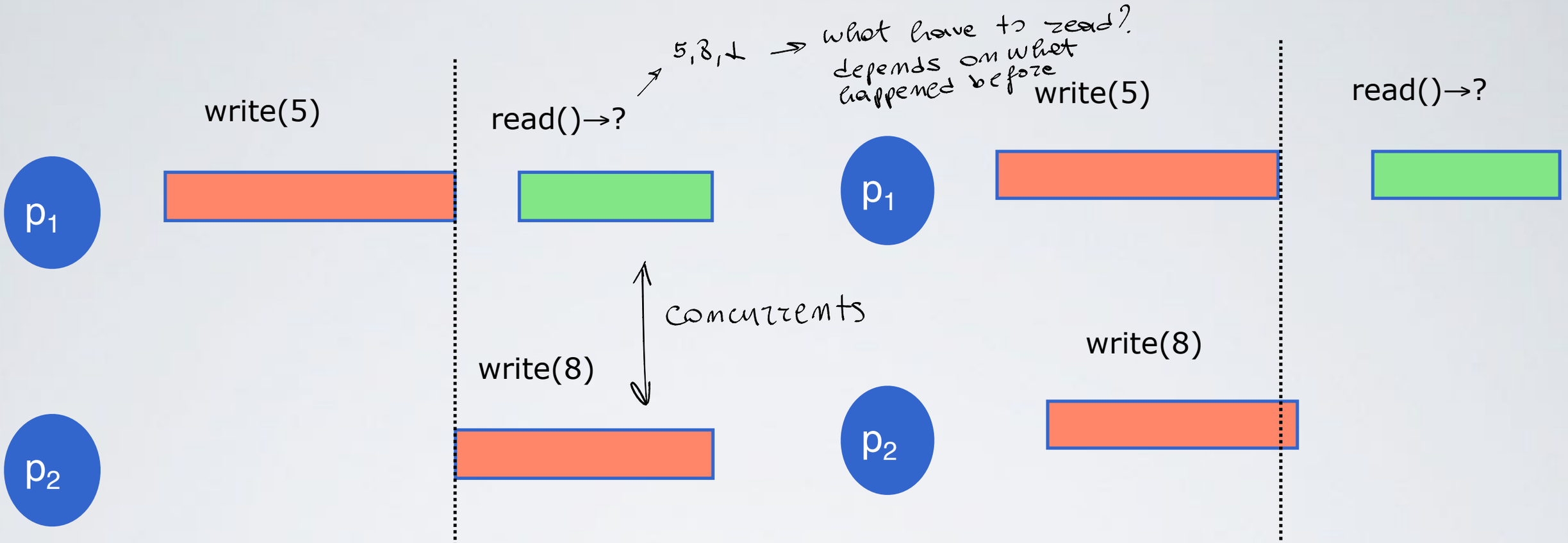
- serial access: a process does not invoke an operation on the register if there is another process that previously invoked an operation on it and this latter did not complete yet
- no failures

## Specification:

- **Liveness**. Each operation eventually terminates
- **Safety**. Each read operation returns the last value written

complex

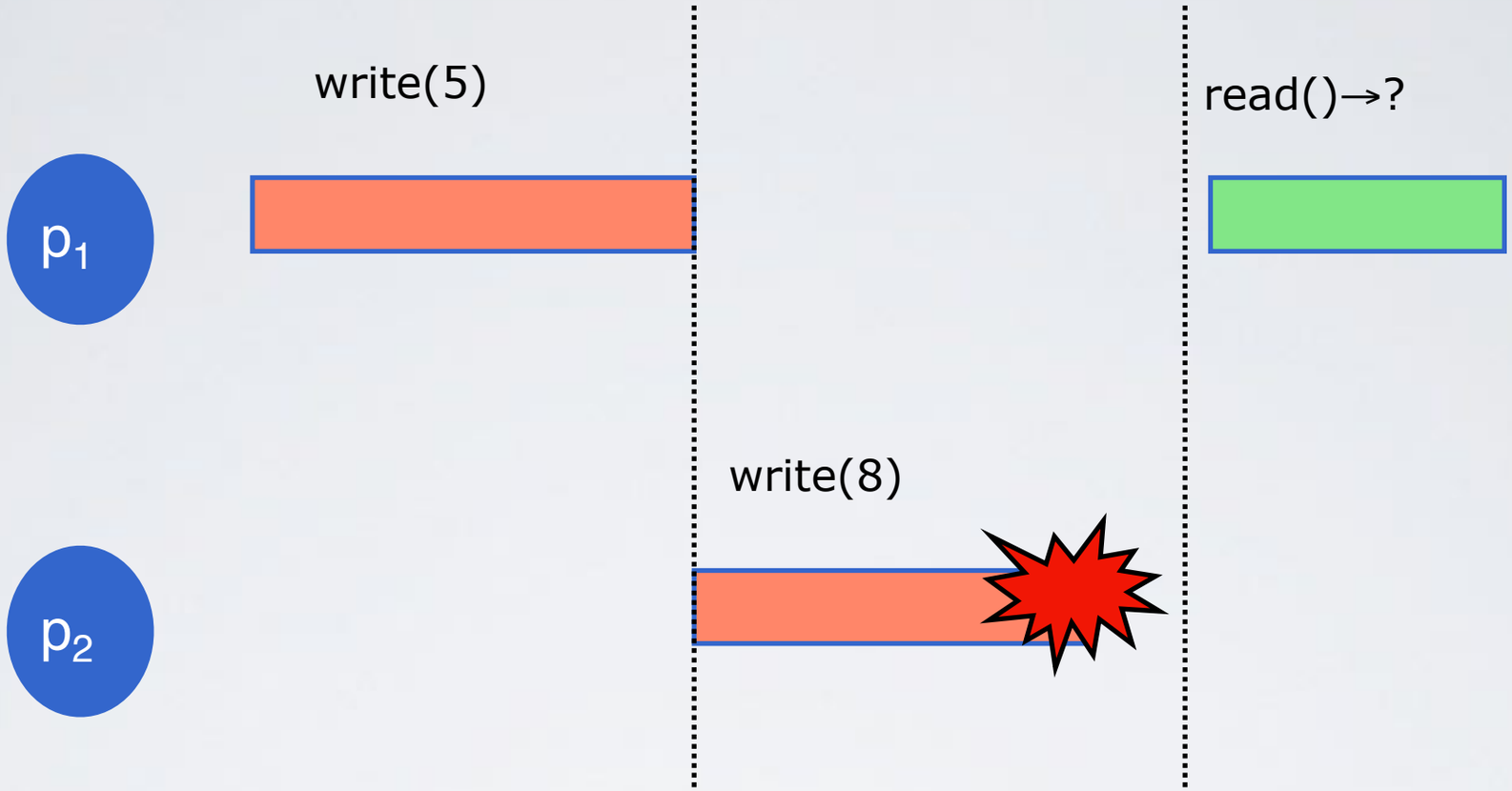
# REGISTER SEMANTICS: CONCURRENCY




Assumptions:

- several processes can access the register
- concurrent access
- no failures

# REGISTER SEMANTICS: FAILURES



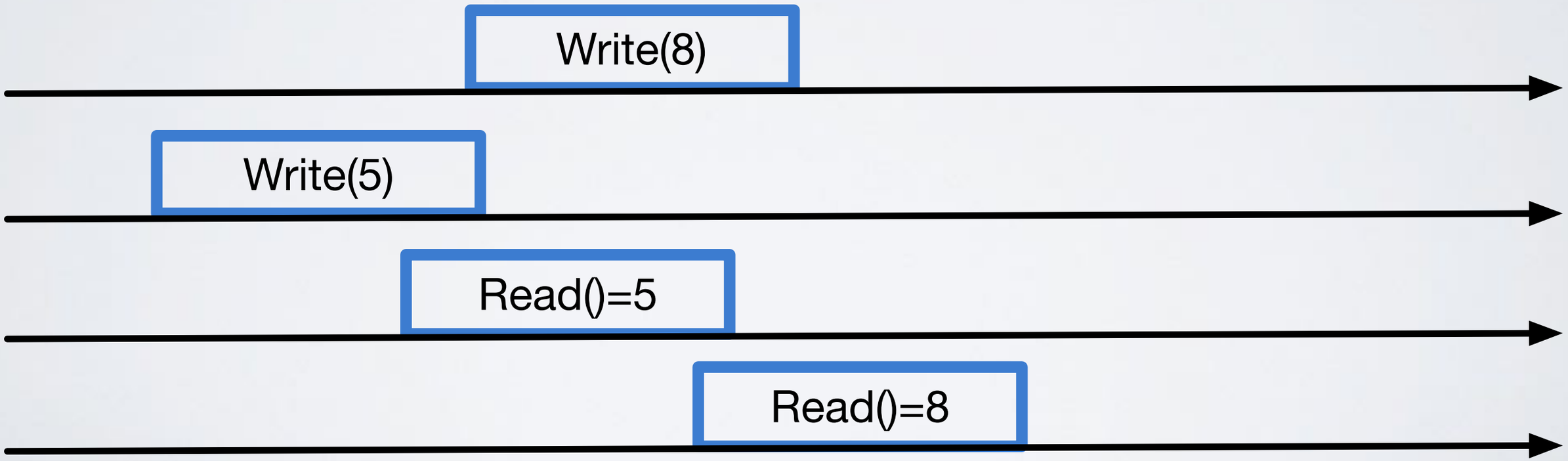

Assumptions:

- several processes can access the register
- serial access
- processes can fail by crashing, i.e. after some point in time they stop to run their algorithm forever

# GOAL

DEFINE A SET OF USEFUL SEMANTICS TO COPE WITH CONCURRENCY.

THE SEMANTIC RESTRICTS THE SET OF ALLOWED EXECUTIONS.



# SEMANTICS (CALL SYSTEM)

**REGULAR CONSISTENCY**

**SEQUENTIAL CONSISTENCY**

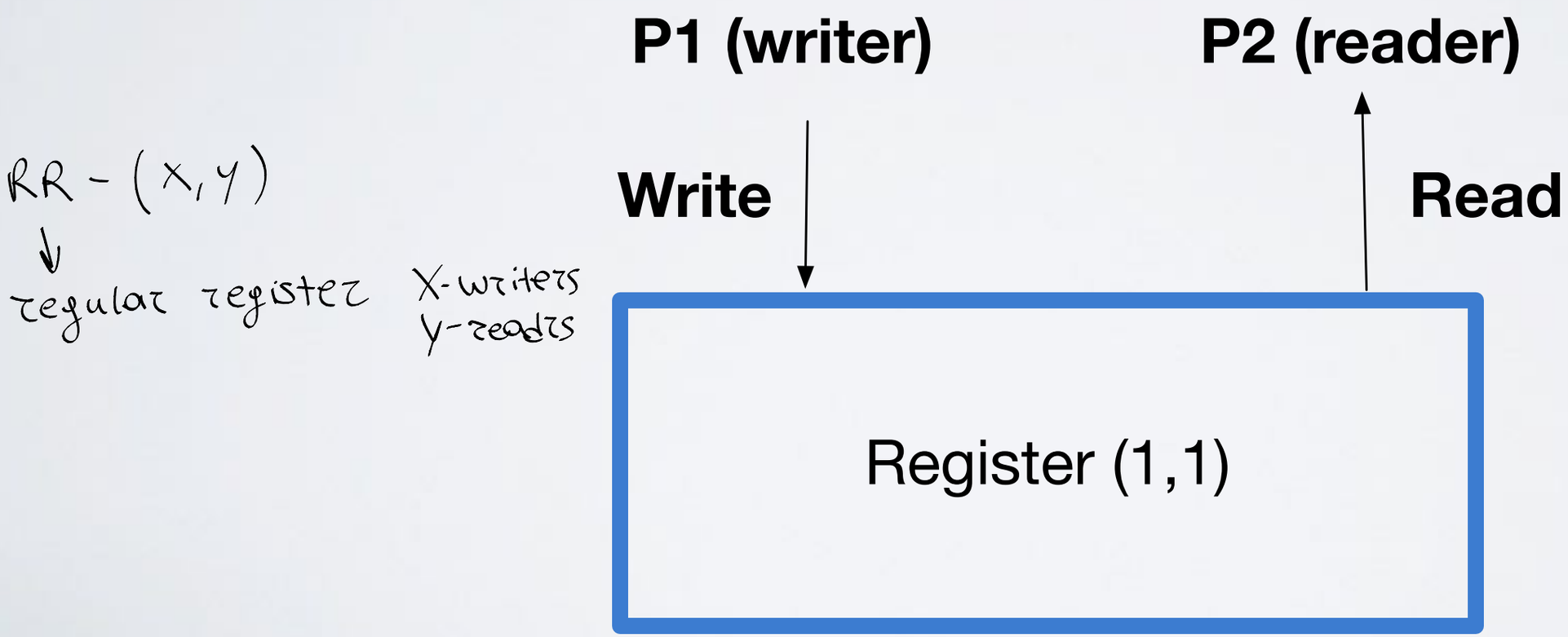
# LINEARIZABILITY/ ATOMIC CONSISTENCY

# REGULAR REGISTERS: FAIL-STOP, FAIL-SILENT

# REGISTER: NOTATION

(X,Y) denotes a register where X processes can write and Y processes can read

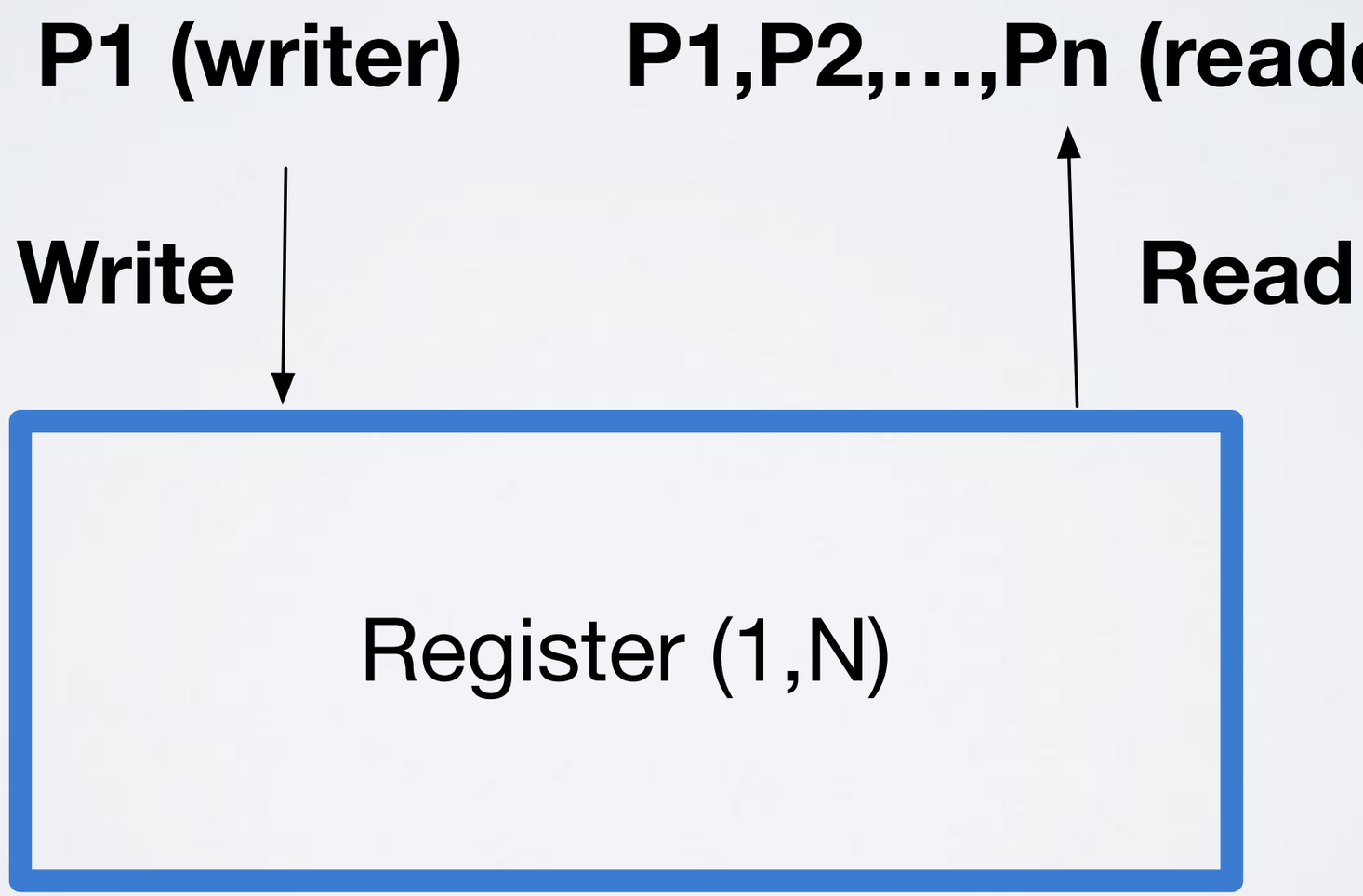
- (1,1) denotes a register where only a process can write and only a process can read.
  - It is a priori known which process can write and which can read



# REGISTER: NOTATION

(X,Y) denotes a register where X processes can write and Y processes can read

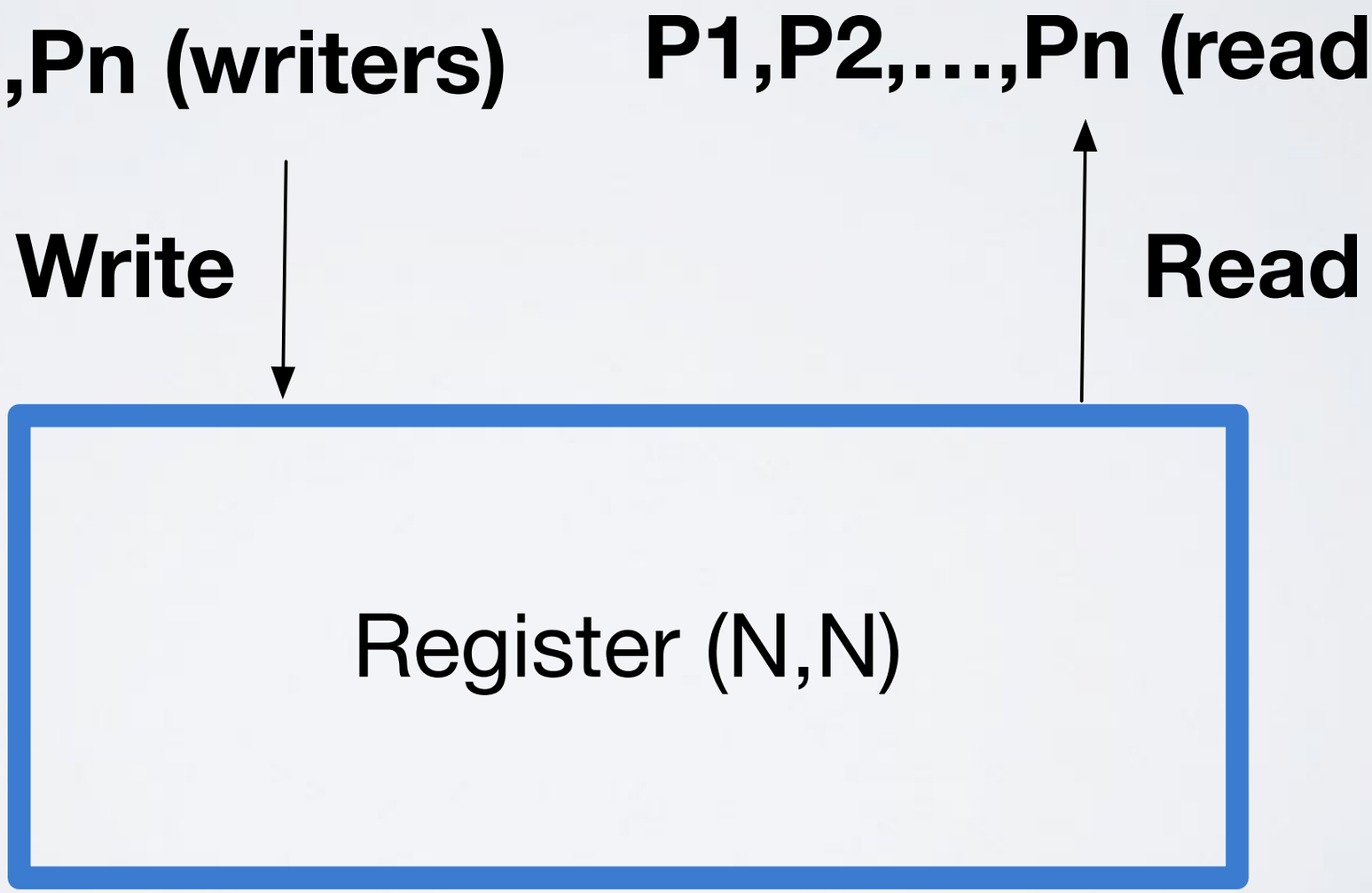
- (1,N) denotes a register where a single process, a priori known, can write, and N processes can read



# REGISTER: NOTATION

(X,Y) denotes a register where X processes can write and Y processes can read

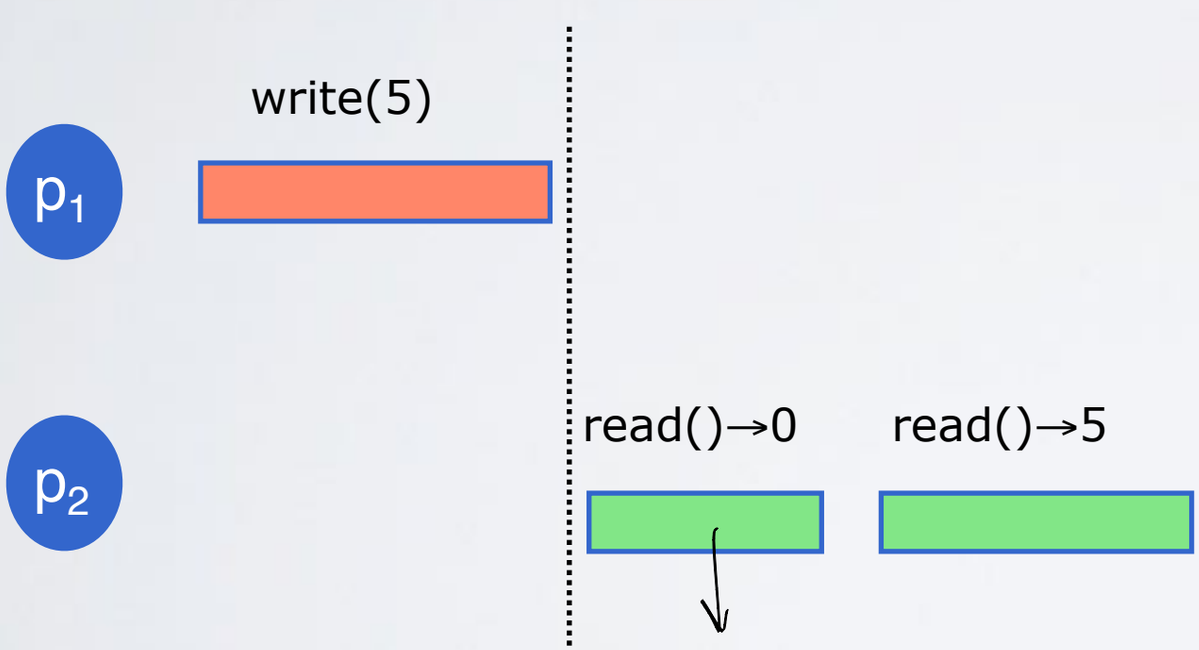
- (N,N) denotes a register where everyone can read/write.

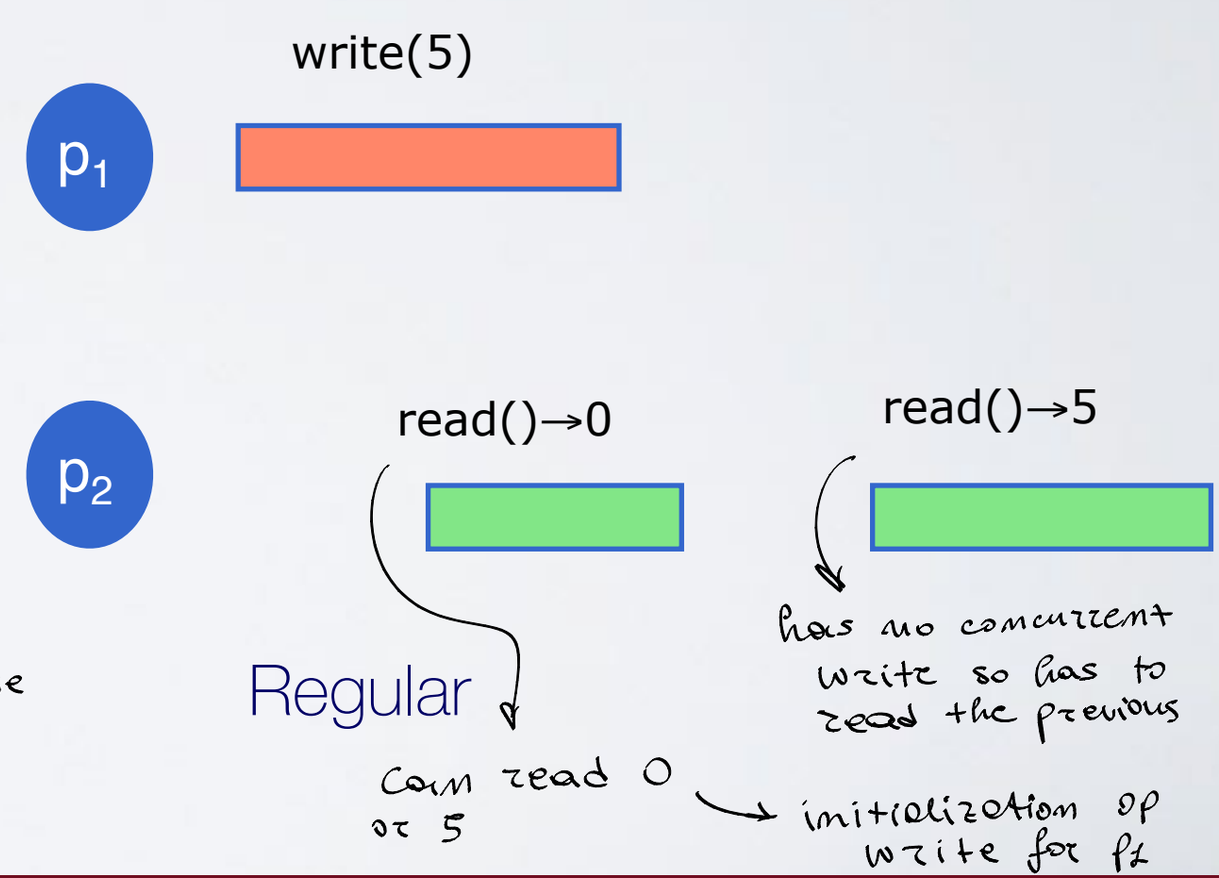


# (1,N) REGULAR REGISTER: SPECIFICATION

**Termination**. If a correct process invokes an operation, then the operation eventually receives the corresponding confirmation.

**Validity**. A read operation returns the last value written or the value concurrently written.

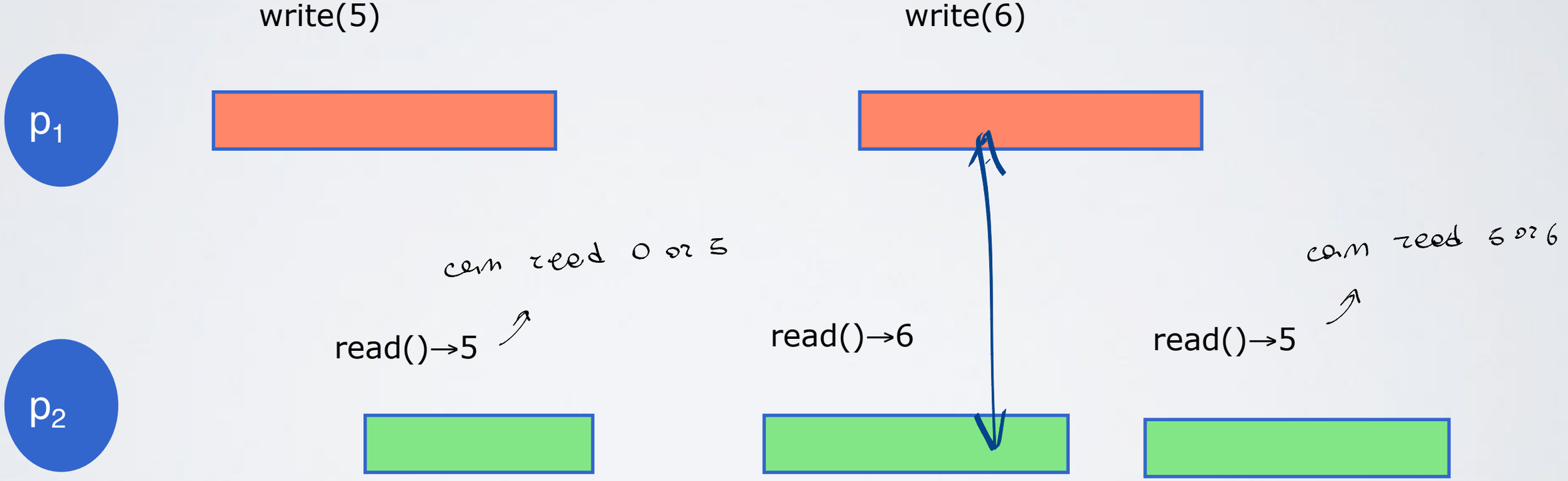


# (1,N) REGULAR REGISTER: SCENARIO

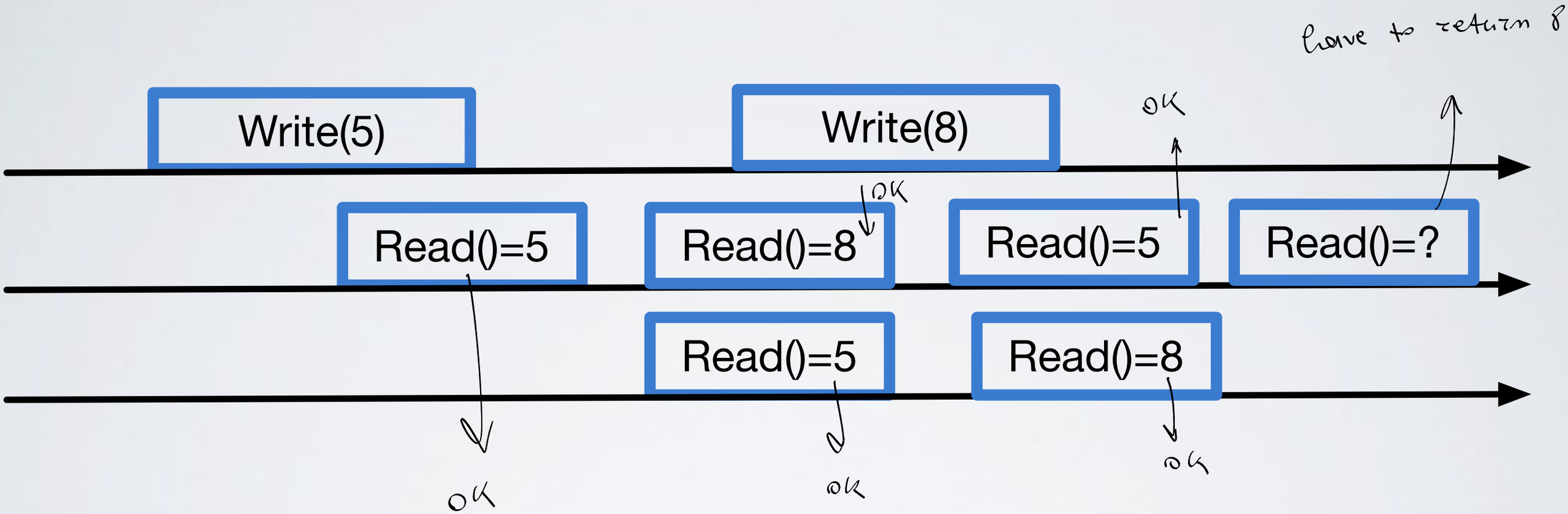
NOTE: In a regular register, a process can read a value v and then a value v', even if the writer has written v' and then v, as long as the write and the read operations are concurrent

it's regular because • are concurrent even if write(6) is after read()

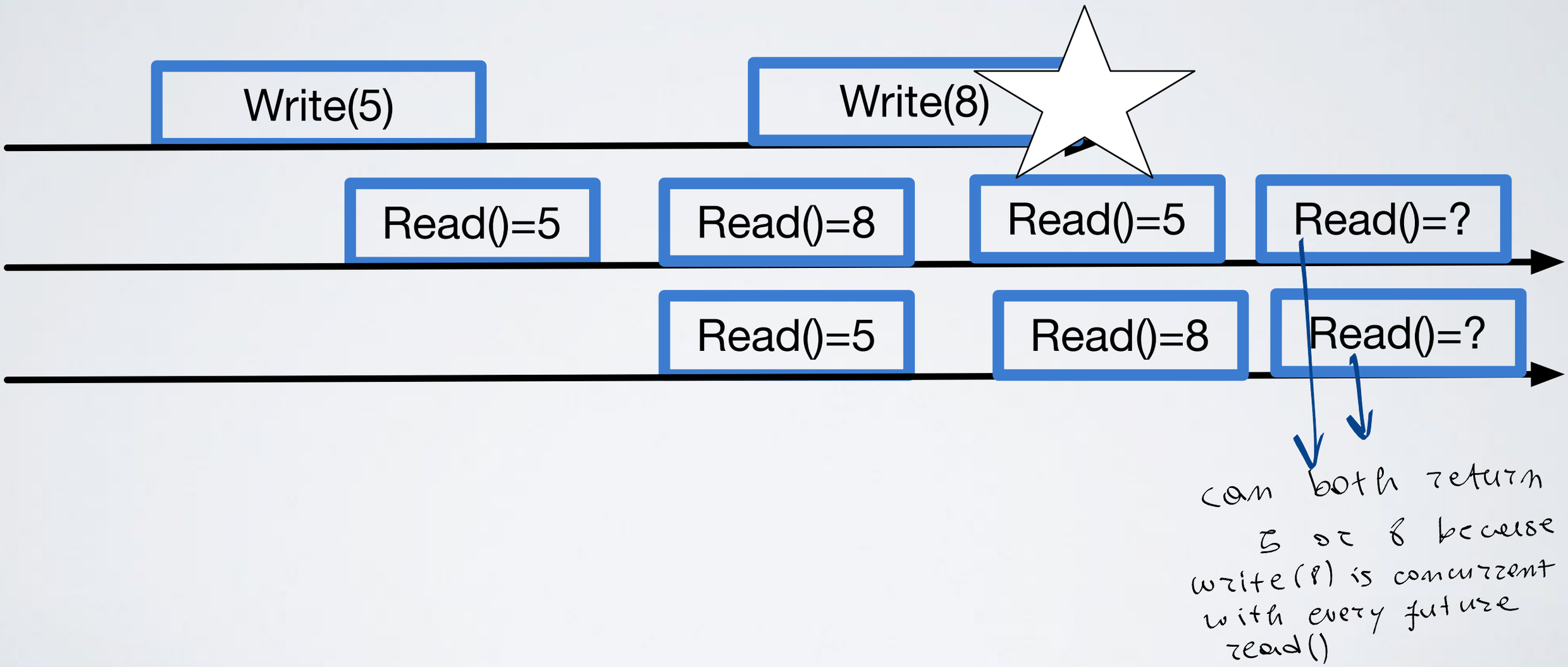


# (1,N) REGULAR REGISTER: SCENARIO

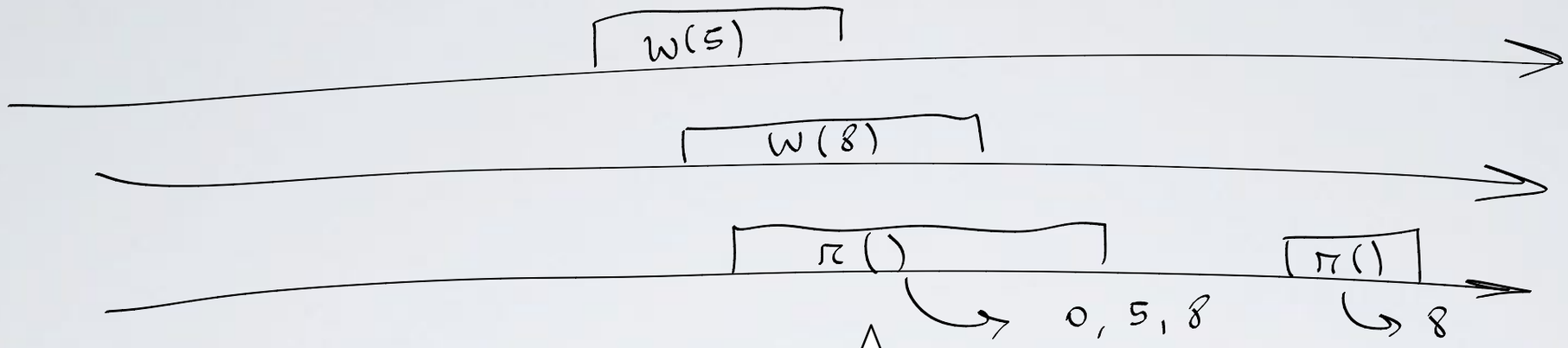
example

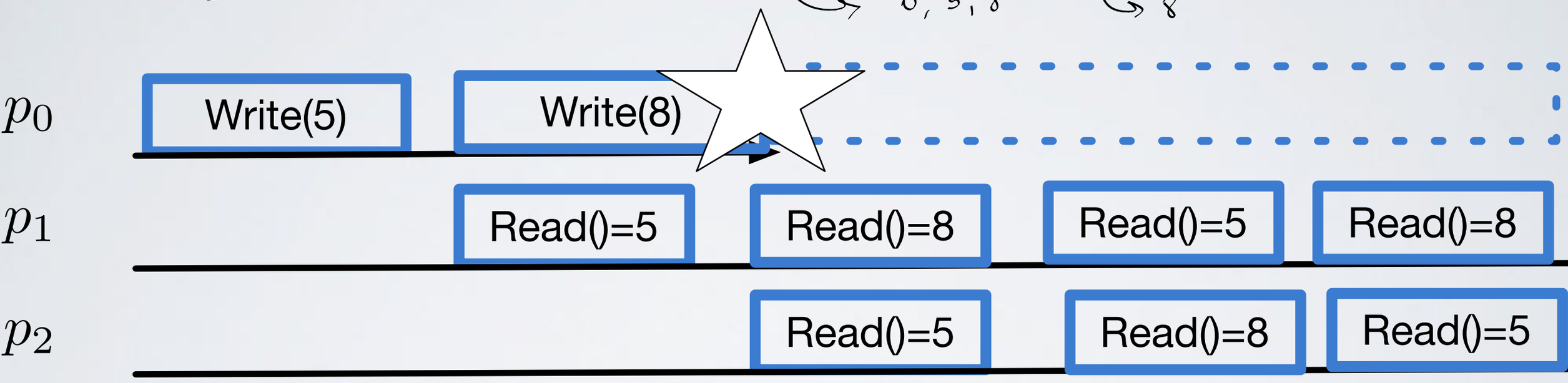


# (1,N) REGULAR REGISTER: SCENARIO



# (1,N) REGULAR REGISTER: SCENARIO




Write(8) is concurrent with any future read

It is possible that p1 reads always 5, and p2 8, or that alternate between values

# (1,N) REGULAR REGISTER: INTERFACE

**Module 4.1:** Interface and properties of a $(1, N)$ regular register

**Module:**

**Name:** $(1, N)$-RegularRegister, **instance** *onrr*.

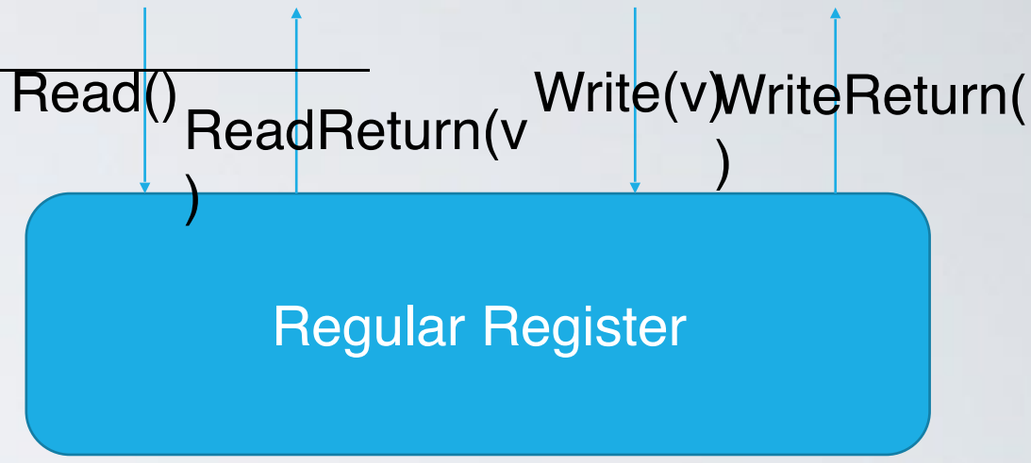
**Events:**

**Request:** $\langle$ *onrr*, *Read* $\rangle$: Invokes a read operation on the register.

**Request:** $\langle$ *onrr*, *Write* $\mid v \rangle$: Invokes a write operation with value $v$ on the register.

**Indication:** $\langle$ *onrr*, *ReadReturn* $\mid v \rangle$: Completes a read operation on the register with return value $v$.

**Indication:** $\langle$ *onrr*, *WriteReturn* $\rangle$: Completes a write operation on the register.



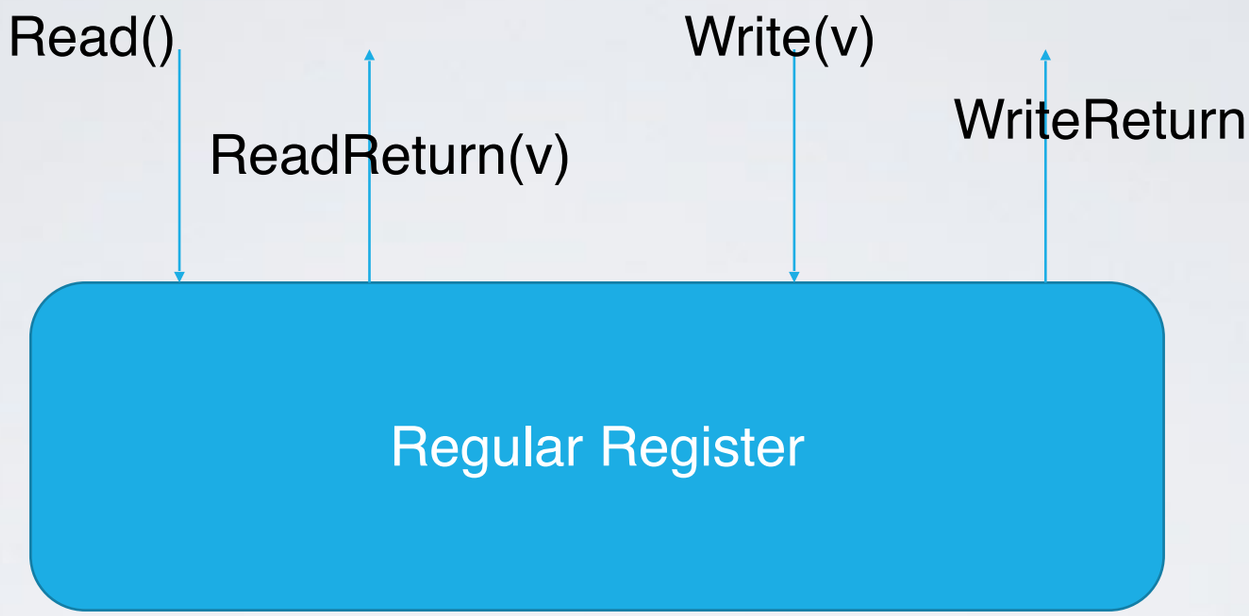

We first Assume P.

**Properties:**

**ONRR1:** *Termination:* If a correct process invokes an operation, then the operation eventually completes.

**ONRR2:** *Validity:* A read that is not concurrent with a write returns the last value written; a read that is concurrent with a write returns the last value written or the value concurrently written.

# (1,N) REGULAR REGISTER - ALGORITHM CREATION

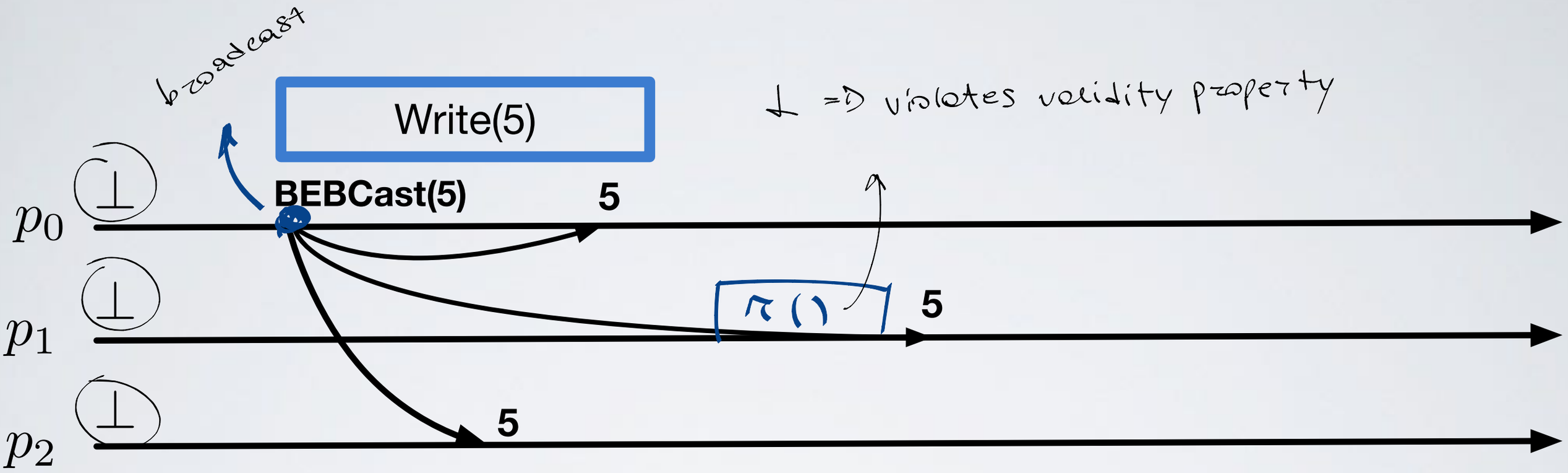



MAIN IDEA: Each process keeps a local copy of the register.
(an integer initially set to $\perp$).

The single writer has to ensure the consistencies of copies.

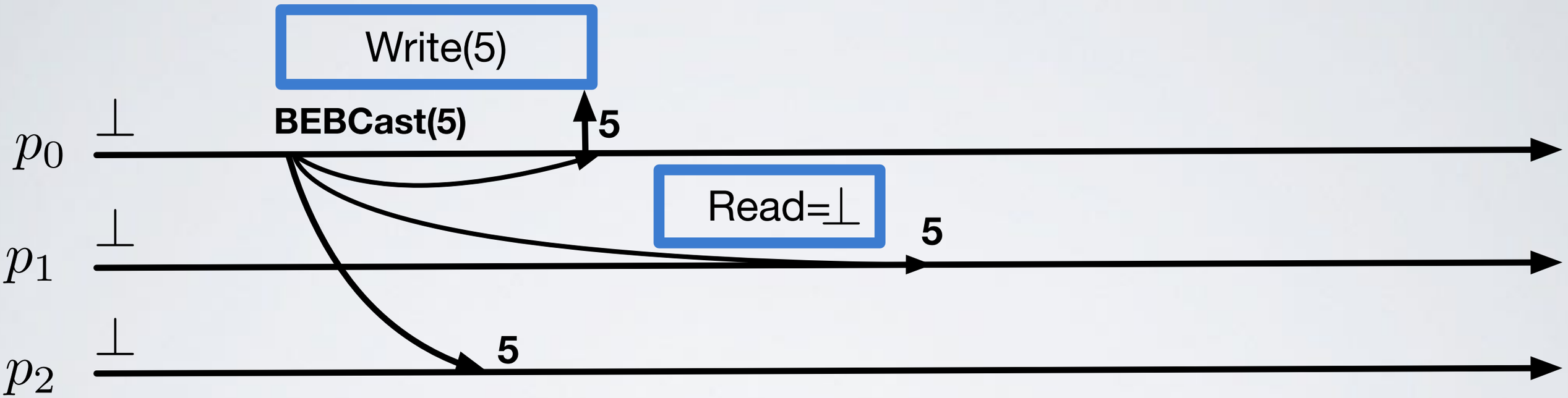
When I read I read from my local copy

# (1,N) REGULAR REGISTER - ALGORITHM CREATION




When a write of value v is trigged by the only write p0, then p0 BEBCast v to all.
When I receive a new value, I update my local copy.

# (1,N) REGULAR REGISTER - ALGORITHM CREATION

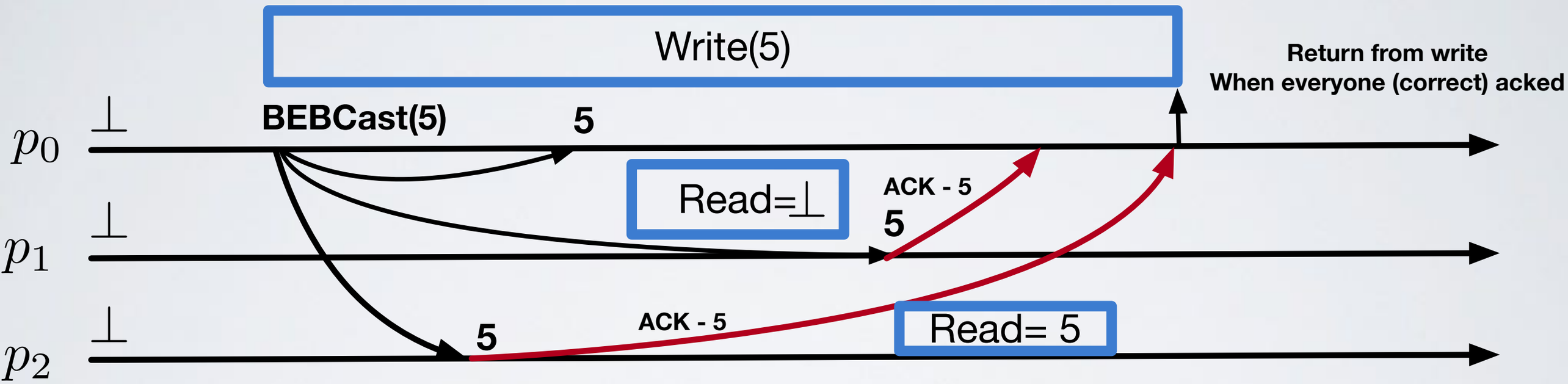



Problem: When the write returns? If it returns After the BEBCast, or after the local delivery on p0 It does not work.

ASSUME FD P.

# (1,N) REGULAR REGISTER - ALGORITHM CREATION

adding ACKs



# READ-ONE-WRITE-ALL REGULAR REGISTER

**Fail-Stop Algorithm**: processes can crash but the crashes can be reliably detected by all the other processes

Uses:

- Perfect failure detector
- Perfect point-to-point link
- Best effort broadcast

# READ-ONE-WRITE-ALL REGULAR REGISTER

Algorithm Idea:

- Each process stores a local copy of the register
- Read-One: each read operation returns the value stored in the local copy of the register
- Write-All: each write operation updates the value locally stored at each process the writer consider to have not crashed
- A write completes when the writer receives an ack from each process that has not crashed

use of perfect failure detector

# READ-ONE-WRITE-ALL REGULAR REGISTER

**upon event** $\langle$ *onrr*, *Init* $\rangle$ **do**
    $val := \bot$;
    *correct* := $\Pi$;
    *writeset* := $\emptyset$;

**upon event** $\langle$ $\mathcal{P}$, *Crash* | $p$ $\rangle$ **do**
    *correct* := *correct* $\setminus$ $\{p\}$;

**upon event** $\langle$ *onrr*, *Read* $\rangle$ **do**
    **trigger** $\langle$ *onrr*, *ReadReturn* | $val$ $\rangle$;

**upon event** $\langle$ *onrr*, *Write* | $v$ $\rangle$ **do**
    **trigger** $\langle$ *beb*, *Broadcast* | [WRITE, $v$] $\rangle$;

**upon event** $\langle$ *beb*, *Deliver* | $q$, [WRITE, $v$] $\rangle$ **do**
    $val := v$;
    **trigger** $\langle$ *pl*, *Send* | $q$, ACK $\rangle$;

**upon event** $\langle$ *pl*, *Deliver* | $p$, ACK $\rangle$ **do**
    *writeset* := *writeset* $\cup$ $\{p\}$;

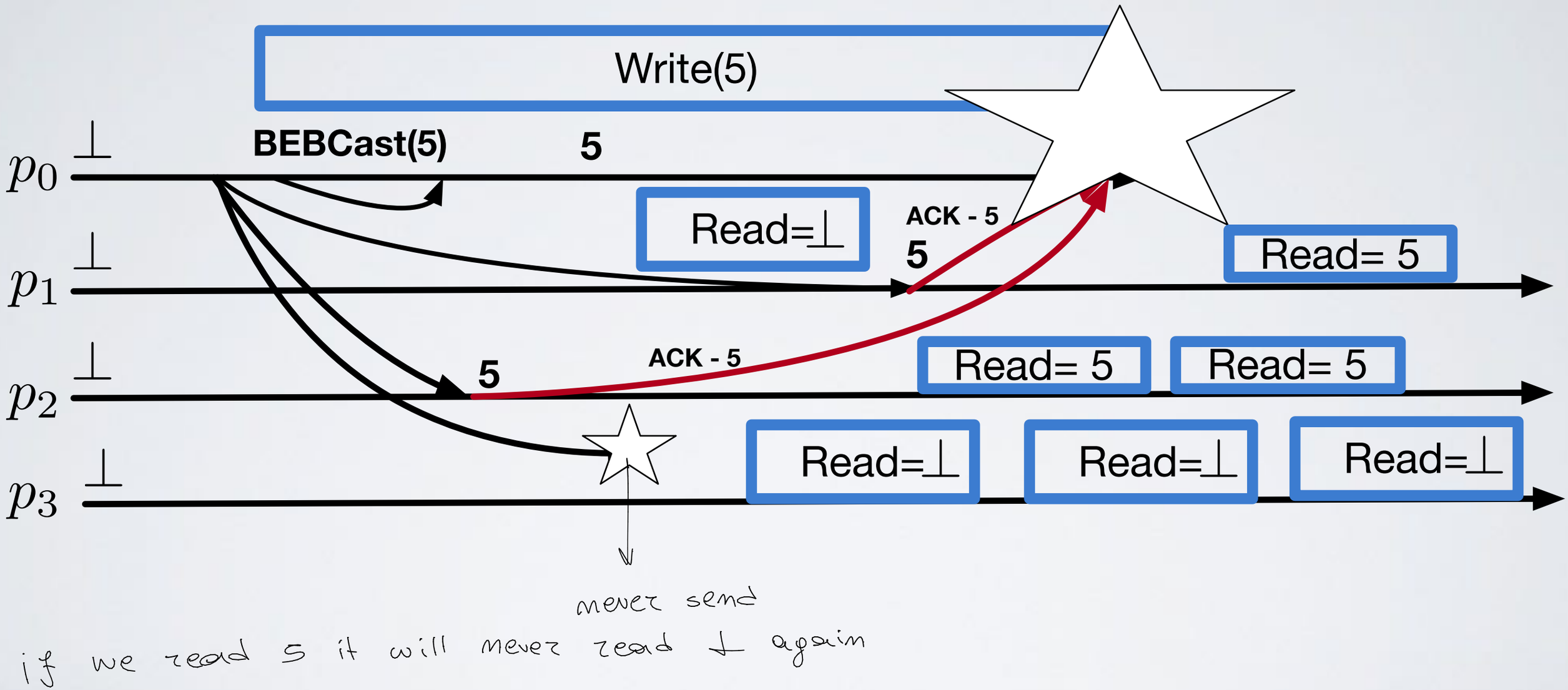
**upon** *correct* $\subseteq$ *writeset* **do**
    *writeset* := $\emptyset$;
    **trigger** $\langle$ *onrr*, *WriteReturn* $\rangle$;

# (1,N) REGULAR REGISTER

**Correctness**:


- Termination -
  - read: trivial, it is local.
  - write: from the properties of the communication primitives and from the completeness property of the perfect failure detector.
- Validity – Because of the strong accuracy property of the perfect failure detector, each write operation can complete only after all processes that have not crashed updated their local copy of the register. So, the two following cases can hold:
  - The read operation is not concurrent with the last write that has been invoked, the process will read the last value written
  - The read operation is concurrent with the last write. For the no creation property of the channels, the value returned is either the last value written or the one being written. This latter is concurrent with the read operation

# (1,N) REGULAR REGISTER: FAILED WRITE RUN



# (1,N) REGULAR REGISTER: PERFORMANCES

**MESSAGE COMPLEXITY?**

**COMMUNICATION STEPS (OR MESSAGE DELAYS)?**

# (1,N) REGULAR REGISTER: PERFORMANCES

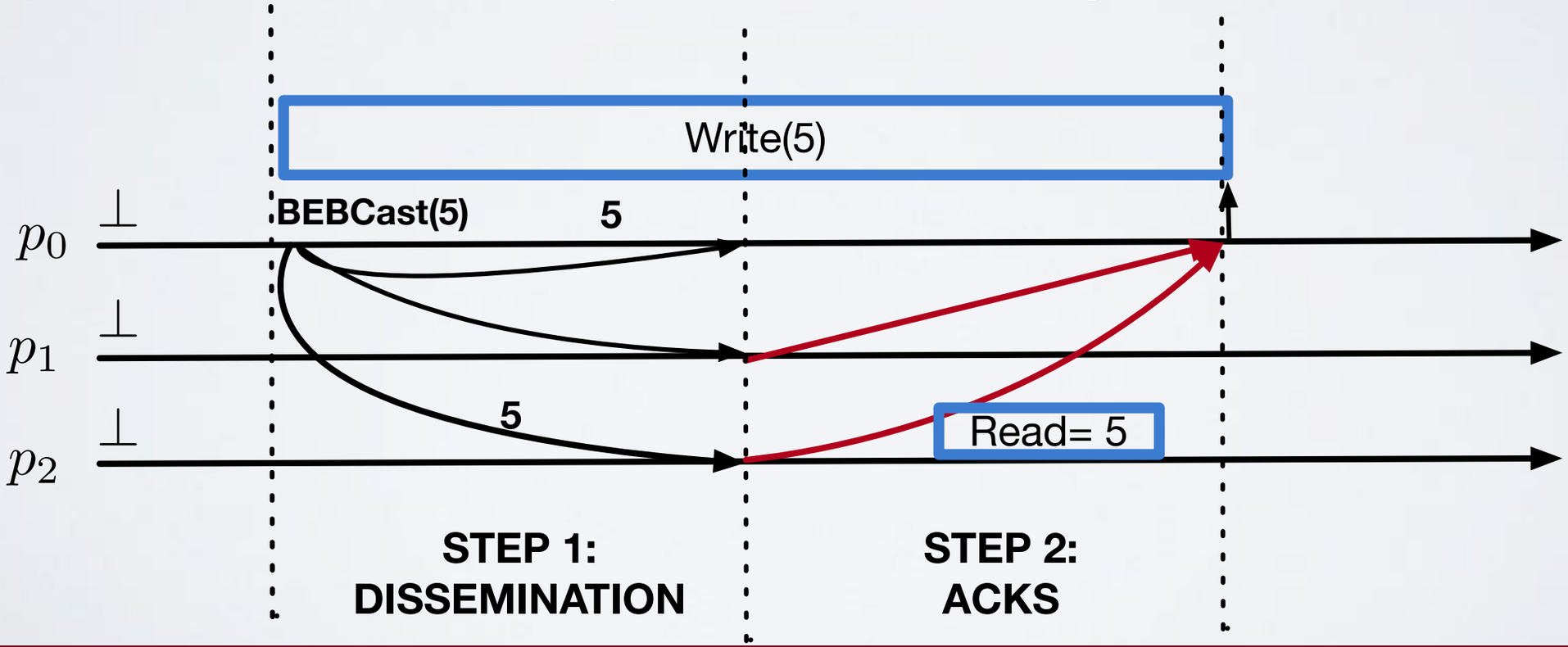
**MESSAGE COMPLEXITY?**

-O(N) for each write (1 Beb, and at most N acks).

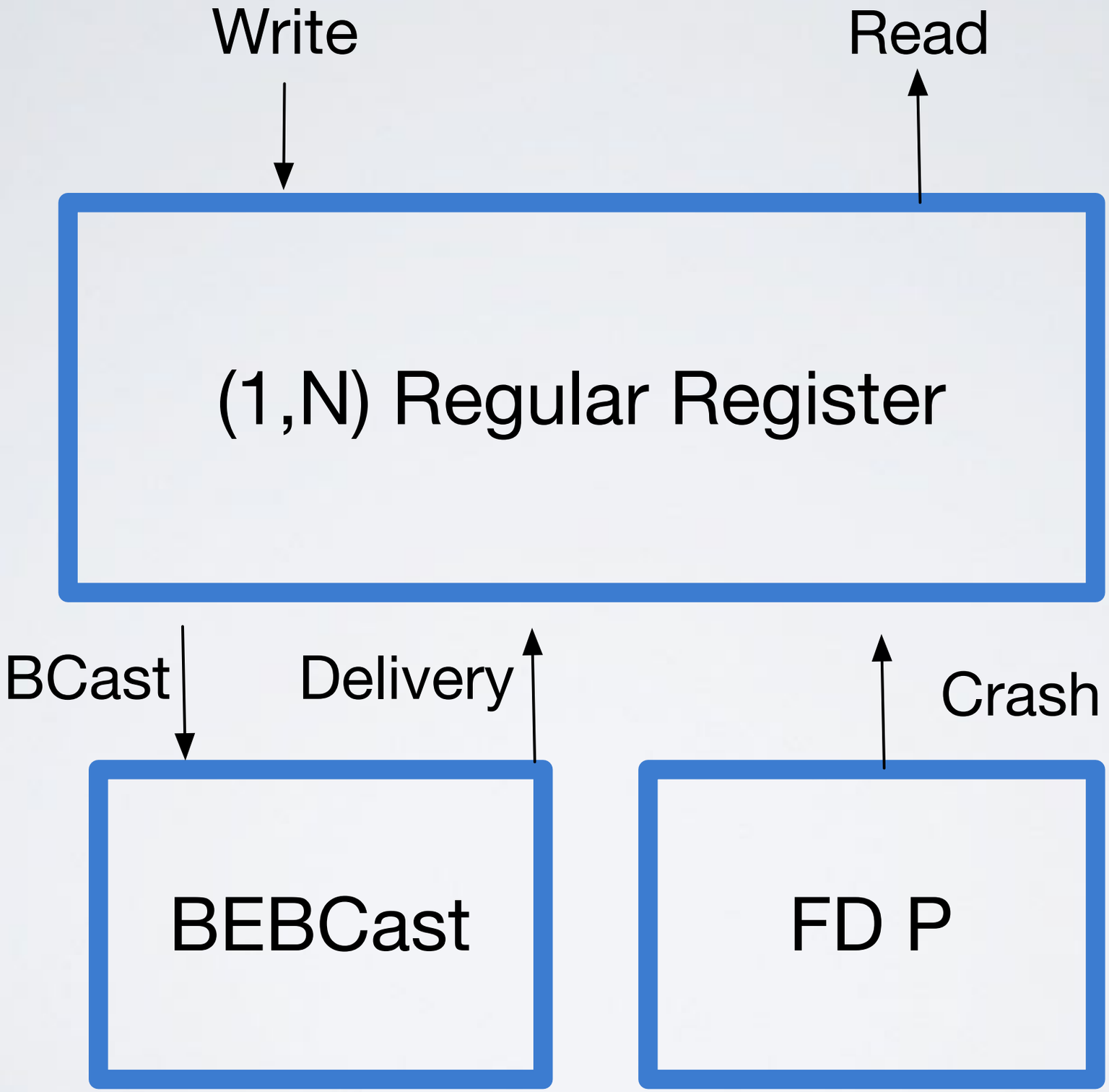
-O(1) for each read: local no message exchanged.

**COMMUNICATION STEPS (OR MESSAGE DELAYS)?**

- 2 steps: 1 to reach everyone, and 1 to get acks back.

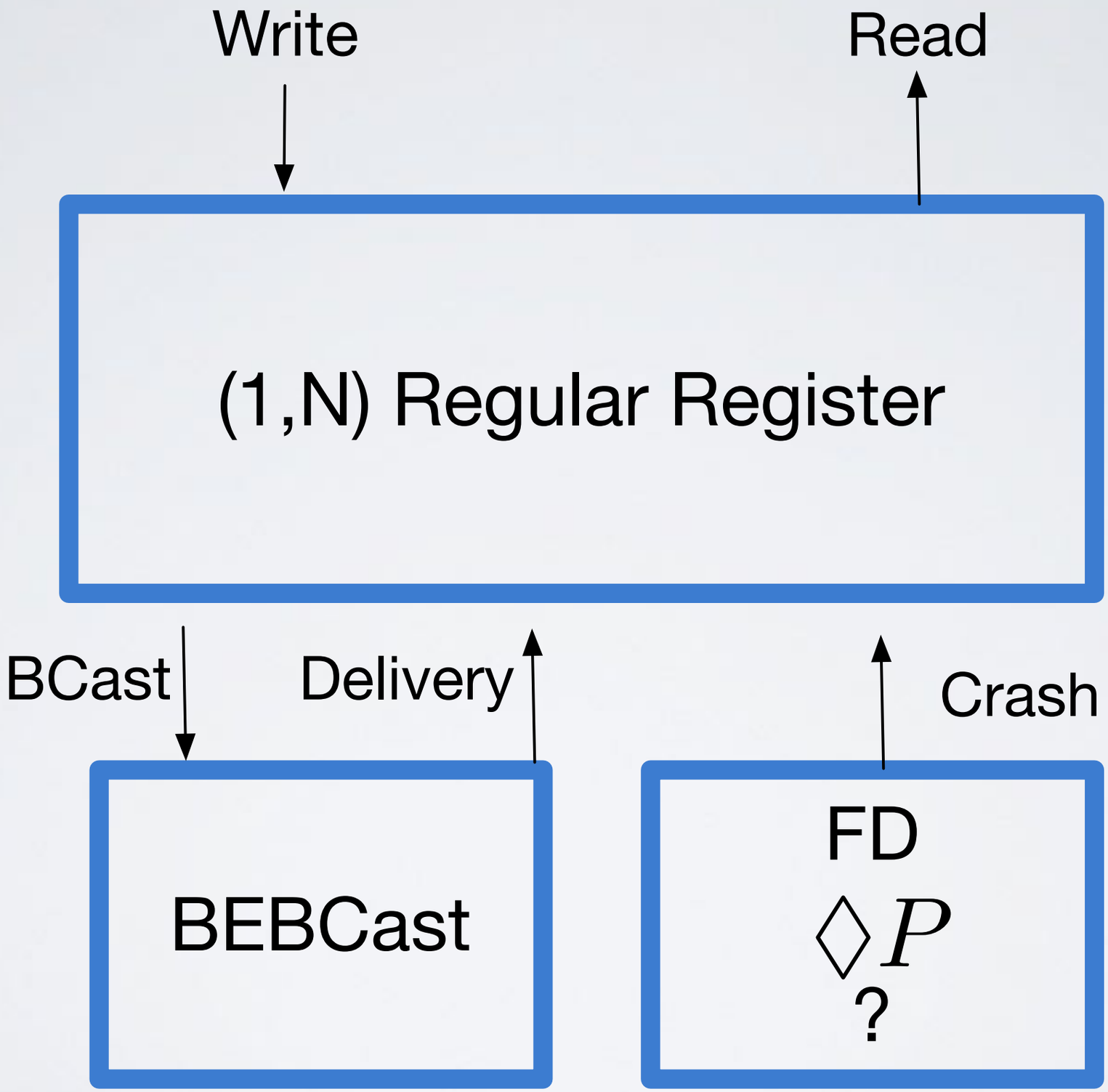


# (1,N) REGULAR REGISTER: DISCUSSION



# (1,N) REGULAR REGISTER: DISCUSSION

What Happens If we use Eventual. P?



# (1,N) REGULAR REGISTER: DISCUSSION

Eventual Accuracy of Ev. P:

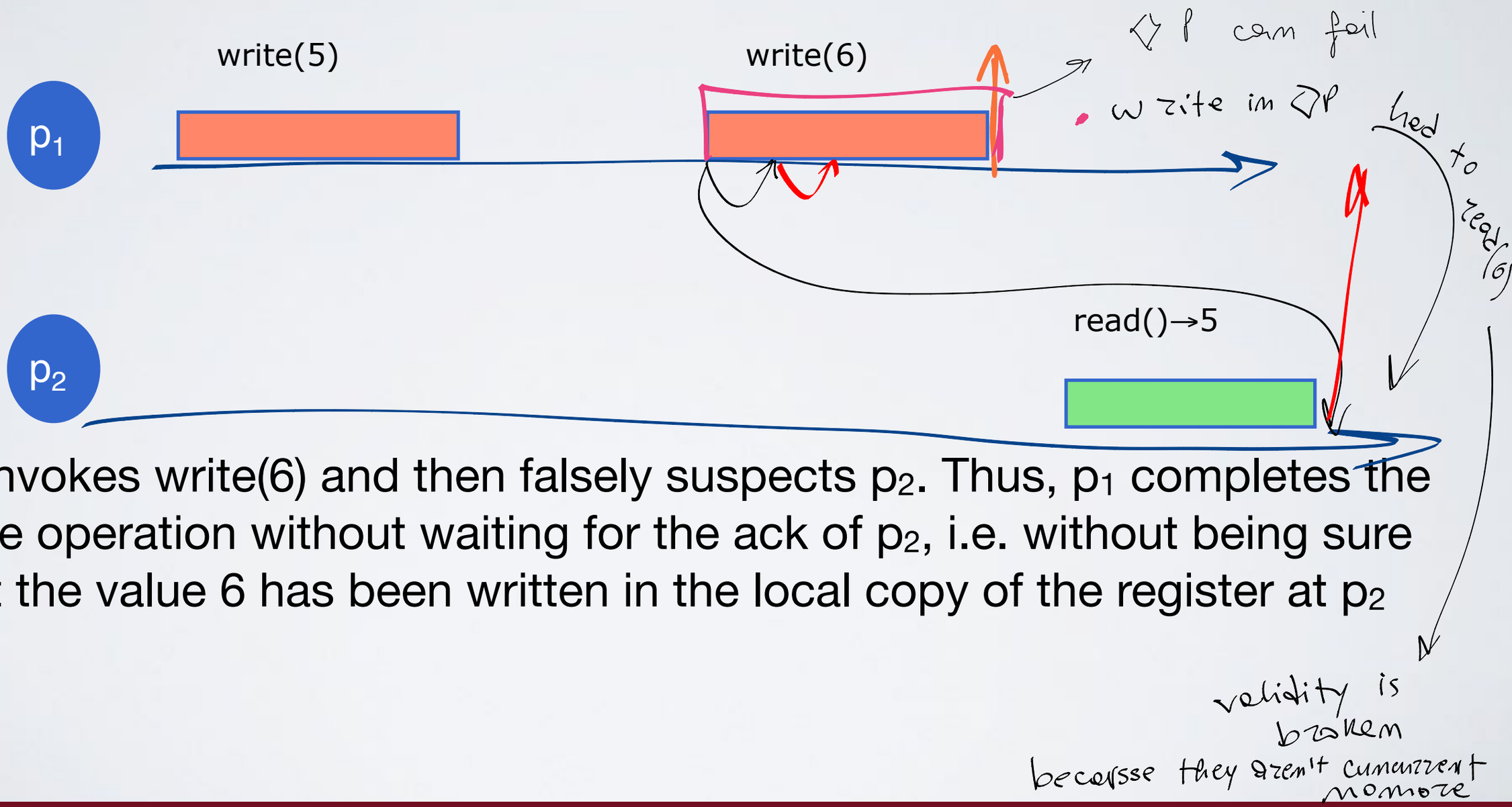
**upon event** $\langle\ pl, Deliver \mid p, \text{ACK}\ \rangle$ **do**
$writeset := writeset \cup \{p\};$

**upon** $correct \subseteq writeset$ **do**
$writeset := \emptyset;$
**trigger** $\langle\ onrr, WriteReturn\ \rangle;$

# READ-ONE-WRITE-ALL RR: PROBLEM

1st property with failure detector is ok

The algorithm does not guarantee validity if the failure detector is not perfect. The following scenario could happen:




- $p_1$ invokes write(6) and then falsely suspects $p_2$. Thus, $p_1$ completes the write operation without waiting for the ack of $p_2$, i.e. without being sure that the value 6 has been written in the local copy of the register at $p_2$

# IS IT POSSIBLE TO ADAPT THE ALGORITHM FOR FAIL-SILENT?

# MAJORITY VOTING REGULAR REGISTER

**Fail-silent algorithm**:``process crashes can never be reliably detected''

- Failure model: crash
- No failure detector

Assumptions:

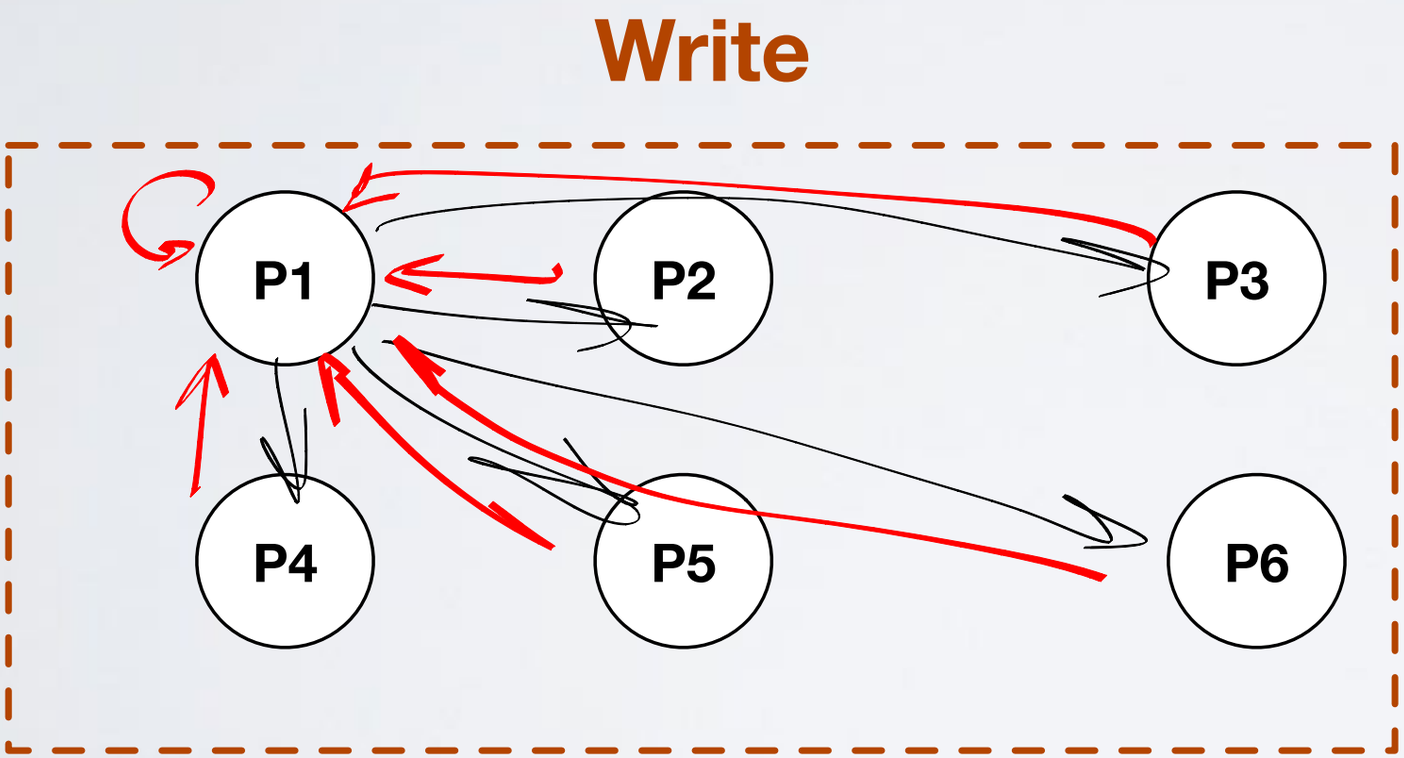
- N processes: 1 writer and N readers
- **A majority of correct processes**

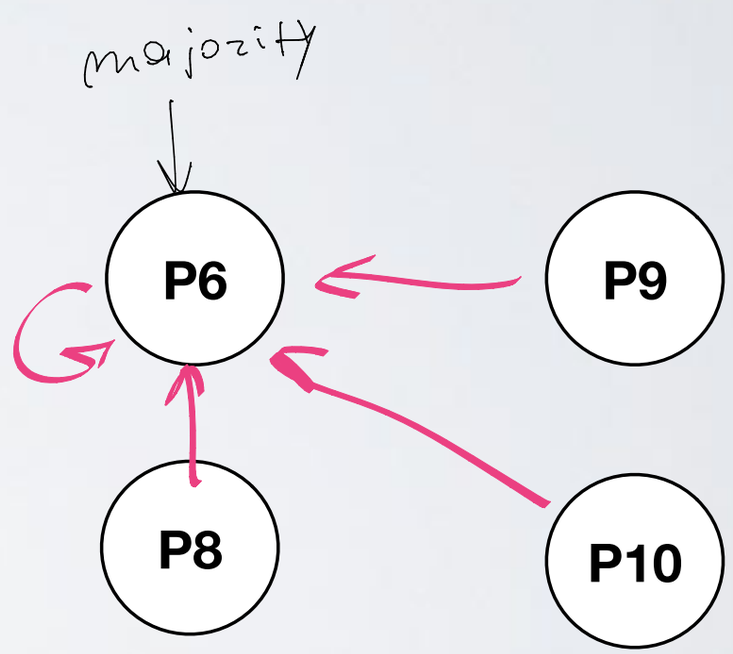
Communication Primitives:

- Perfect point-to-point link
- Best-effort broadcast

# MAJORITY VOTING REGULAR REGISTER

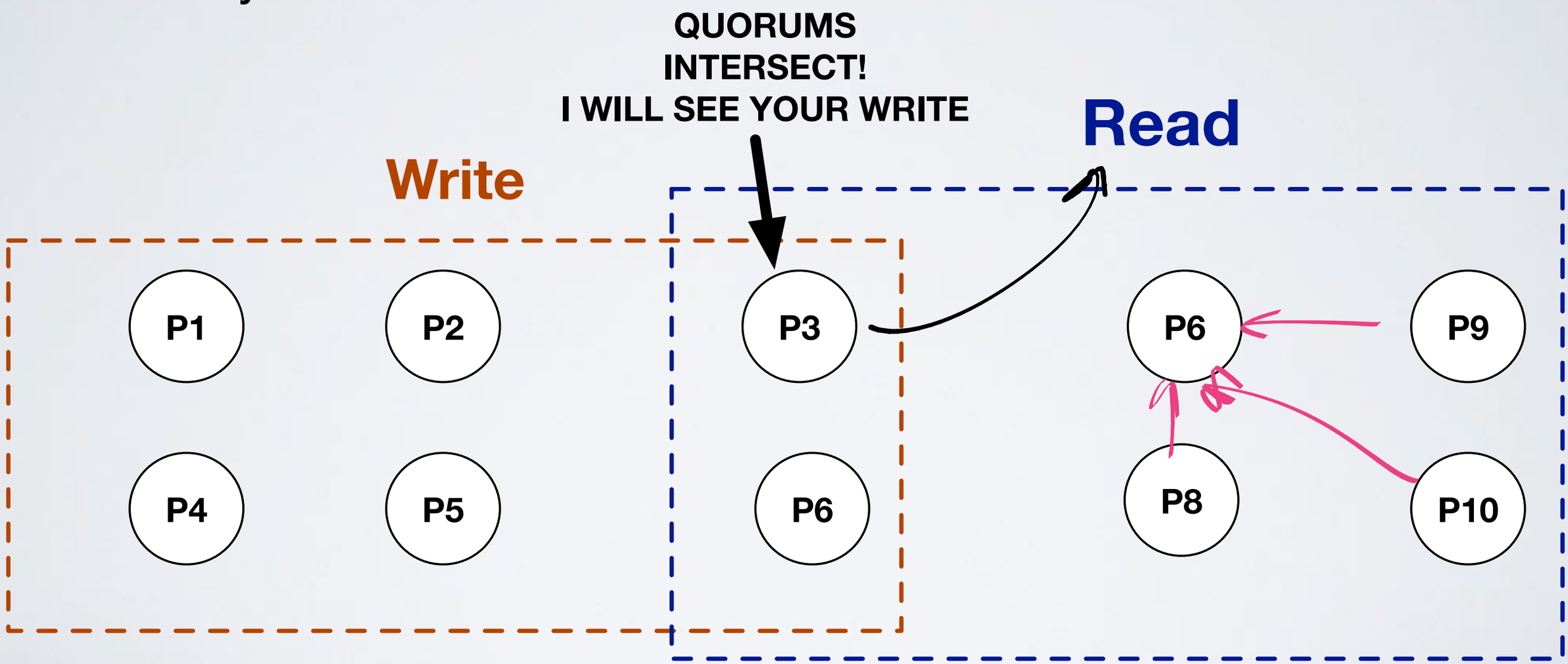
Write: BEBCast, and then I wait for acks from n/2+1 processes (a quorum of processes). Assuming majority of corrects this eventually terminates.



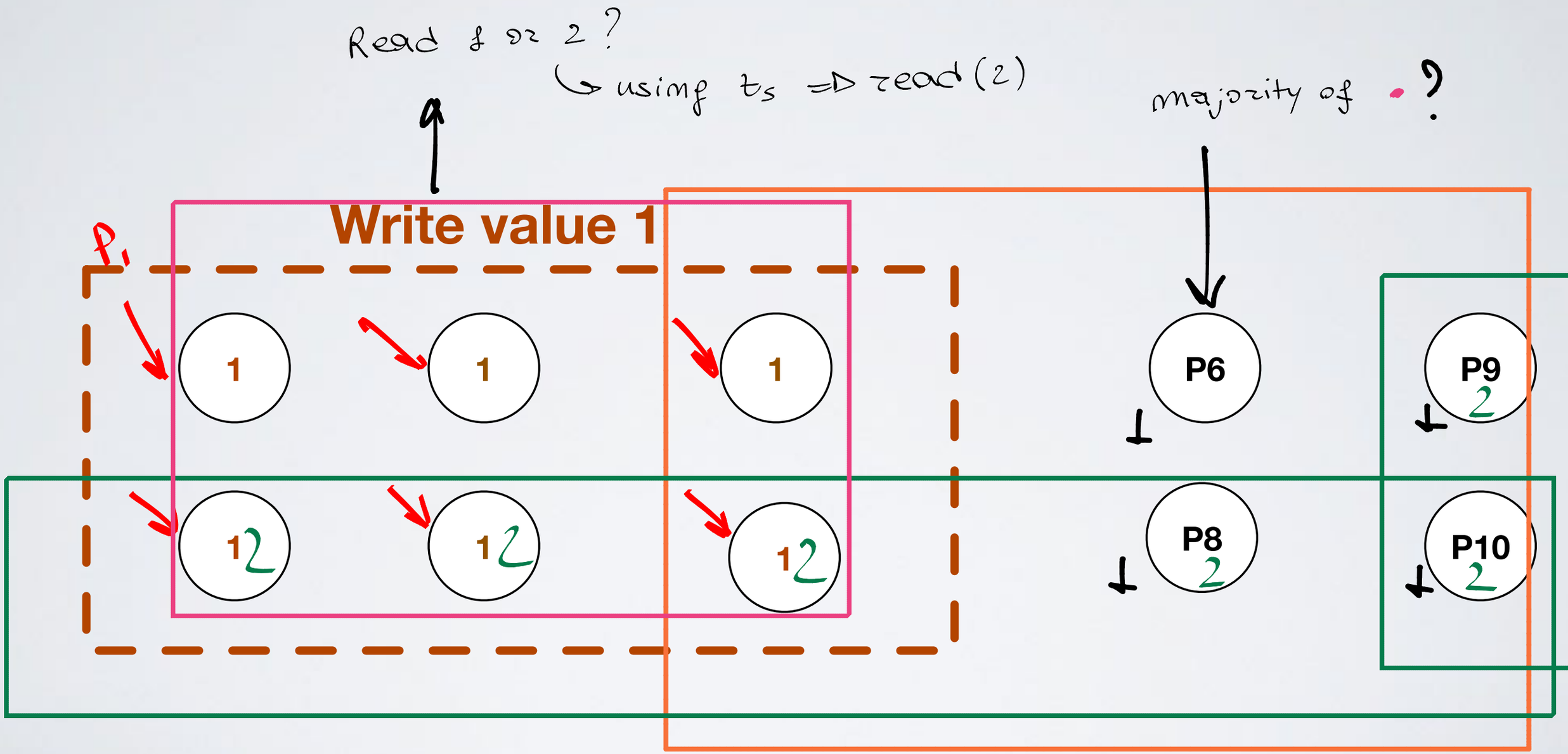
# MAJORITY VOTING REGULAR REGISTER

Read: BEBCast, and wait for register values from n/2+1 processes (another quorum). Assuming majority of corrects this eventually terminates.



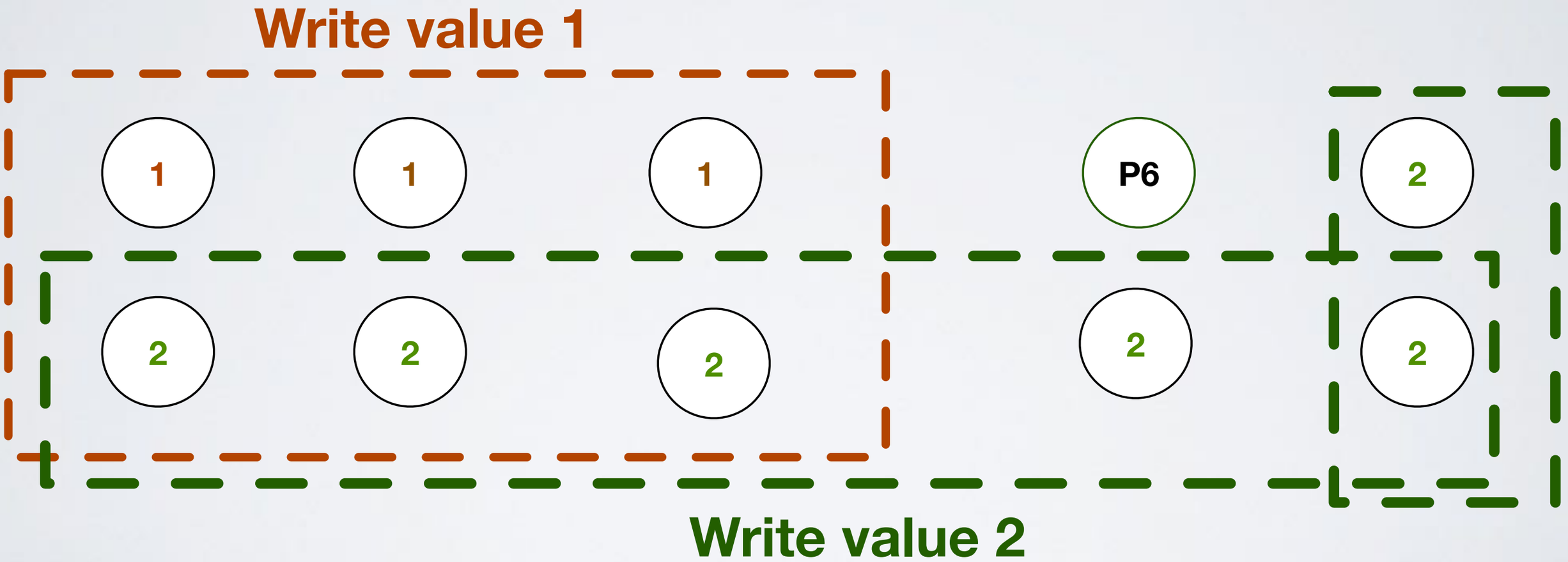
# MAJORITY VOTING REGULAR REGISTER

A small problem has to be fixed. You write 1.



# MAJORITY VOTING REGULAR REGISTER

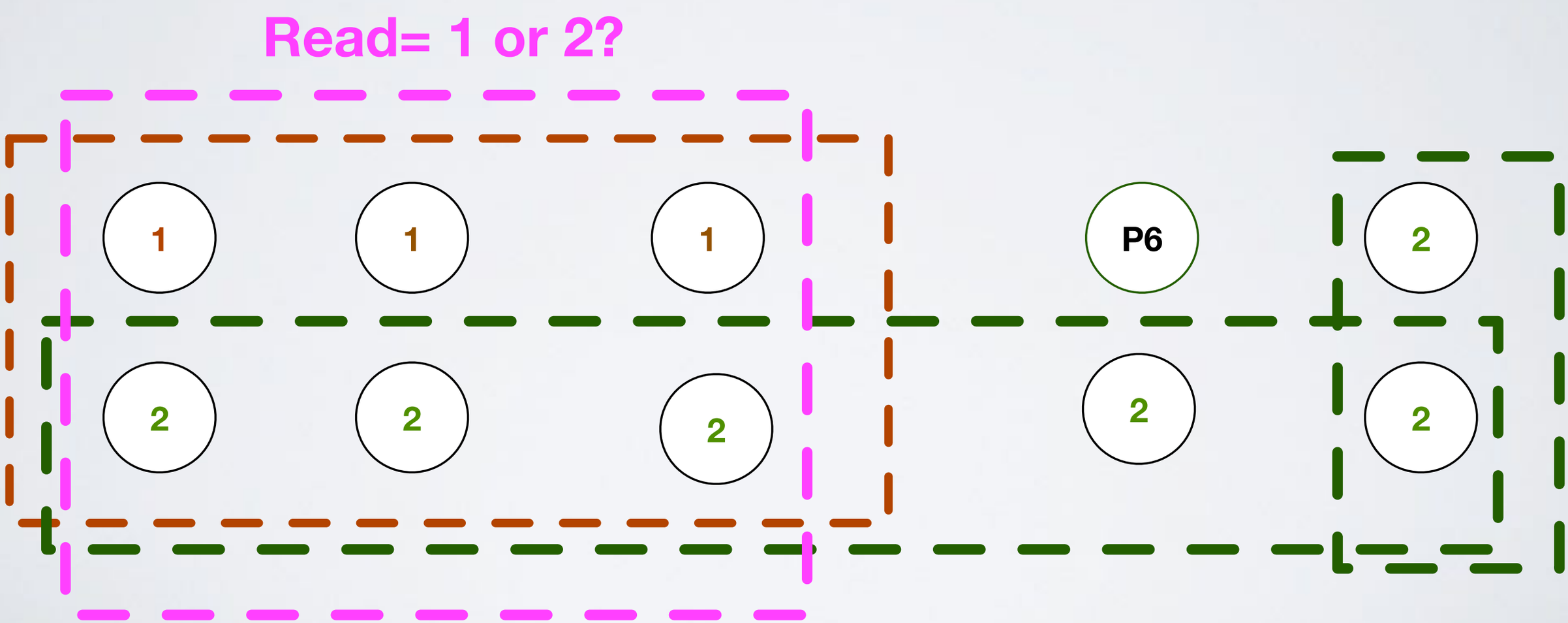
A small problem has to be fixed. You write 1, then you update to 2.



# MAJORITY VOTING REGULAR REGISTER

A small problem has to be fixed. You write 1, then you update to 2.

Then someone reads and sees both values. How to pick the



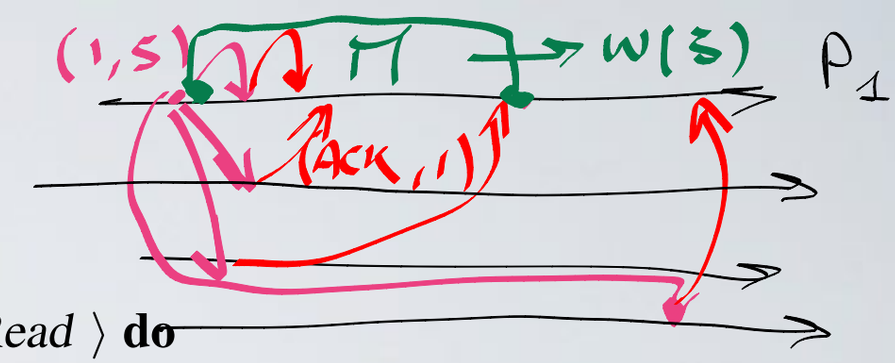
# MAJORITY VOTING REGULAR REGISTER

Quorum changes according to the ACKS

IDEA:

- Each process locally stores a copy of the current value of the register
- Each written value is univocally associated to a **timestamp**
- The writer and the reader processes use a set of witness processes, to track the last value written
- Quorum: the intersection of any two sets of witness processes is not empty
- "Majority Voting": each set is constituted by a majority of processes
- The reader takes the value with the greatest timestamps among the processes in quorum.

# MAJORITY VOTING




**upon event** ⟨ *onrr*, *Init* ⟩ **do**
  $(ts, val) := (0, \perp)$;
  $wts := 0$; → time stamp of write (only 1 can modify it because we have only 1 writer)
  $acks := 0$;
  $rid := 0$;
  $readlist := [\perp]^N$;

**upon event** ⟨ *onrr*, *Write* | $v$ ⟩ **do**
  $wts := wts + 1$;
  $acks := 0$;
  **trigger** ⟨ *beb*, *Broadcast* | [WRITE, $wts, v$] ⟩;

**upon event** ⟨ *beb*, *Deliver* | $p$, [WRITE, $ts', v'$] ⟩ **do**
  **if** $ts' > ts$ **then**
    $(ts, val) := (ts', v')$;
  **trigger** ⟨ *pl*, *Send* | $p$, [ACK, $ts'$] ⟩;
  updating only if I recive a new one

**upon event** ⟨ *pl*, *Deliver* | $q$, [ACK, $ts'$] ⟩ **such that** $ts' = wts$ **do**
  $acks := acks + 1$;
  **if** $acks > N/2$ **then** → when is a majority
    $acks := 0$;
    **trigger** ⟨ *onrr*, *WriteReturn* ⟩; → termination is not a problem

Validity is a problem

**upon event** ⟨ *onrr*, *Read* ⟩ **do**
  $rid := rid + 1$;
  $readlist := [\perp]^N$;
  **trigger** ⟨ *beb*, *Broadcast* | [READ, $rid$] ⟩; Π = majority

**upon event** ⟨ *beb*, *Deliver* | $p$, [READ, $r$] ⟩ **do**
  **trigger** ⟨ *pl*, *Send* | $p$, [VALUE, $r, ts, val$] ⟩;

some problem for write

**upon event** ⟨ *pl*, *Deliver* | $q$, [VALUE, $r, ts', v'$] ⟩ **such that** $r = rid$ **do**
  $readlist[q] := (ts', v')$;
  **if** $\#(readlist) > N/2$ **then**
    $v := highestval(readlist)$;
  $readlist := [\perp]^N$;
  **trigger** ⟨ *onrr*, *ReadReturn* | $v$ ⟩;

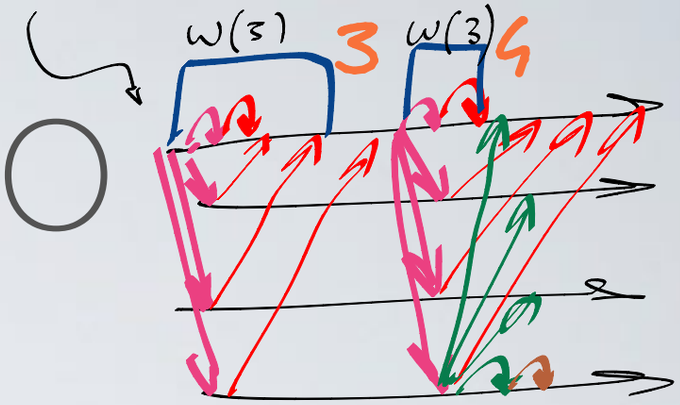
Message complexity W: 2N

// // read: 2N

delay write: 2

delay read: 2

# FUNCTIONING SCENARIO



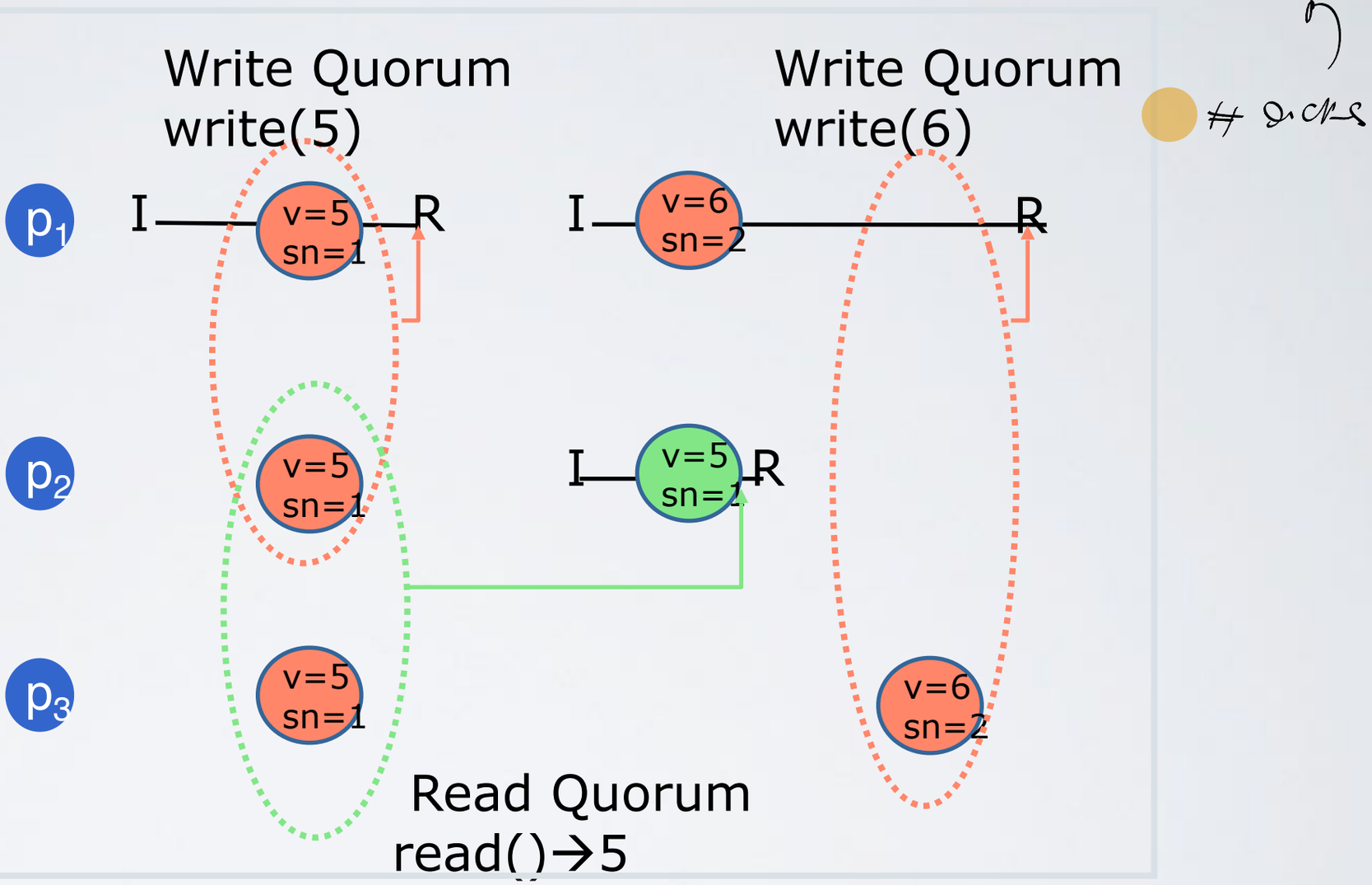

(1,3) regular register

$\Pi=\{p_1,p_2,p_3\}$

$p_1$ is the writer

I=invokation

R=response



# MAJORITY VOTING

Correctness:

- Termination – from the properties of the communication primitives and the assumption of a majority of correct processes
- Validity – from the intersection property of the quorums

Performance - Message Complexity:

- Write –at most 2N messages
- Read - at most 2N messages

Performance - Communication Steps:

- Write -2 steps
- Read - 2 steps

Resiliency? (How many faults do you tollerate?)

# CONSISTENCY: SEQUENTIAL AND ATOMIC

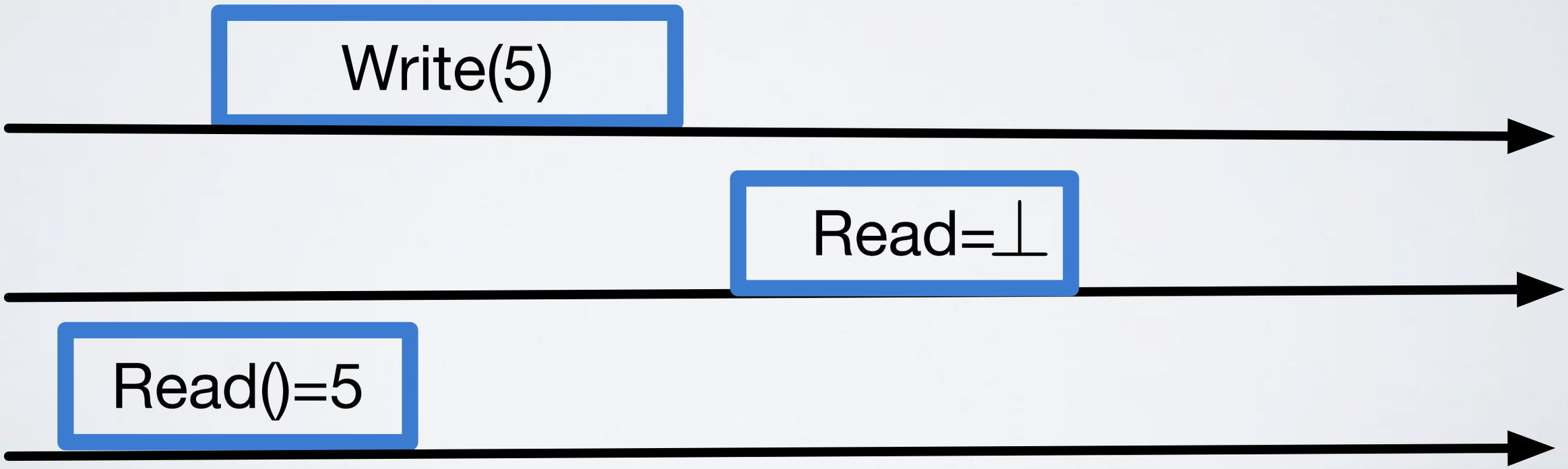
# CONSISTENCY PROPERTIES

We will see three kind of consistency:

- Regular (the one you just seen)
- Sequential
- Linearizability/Atomicity

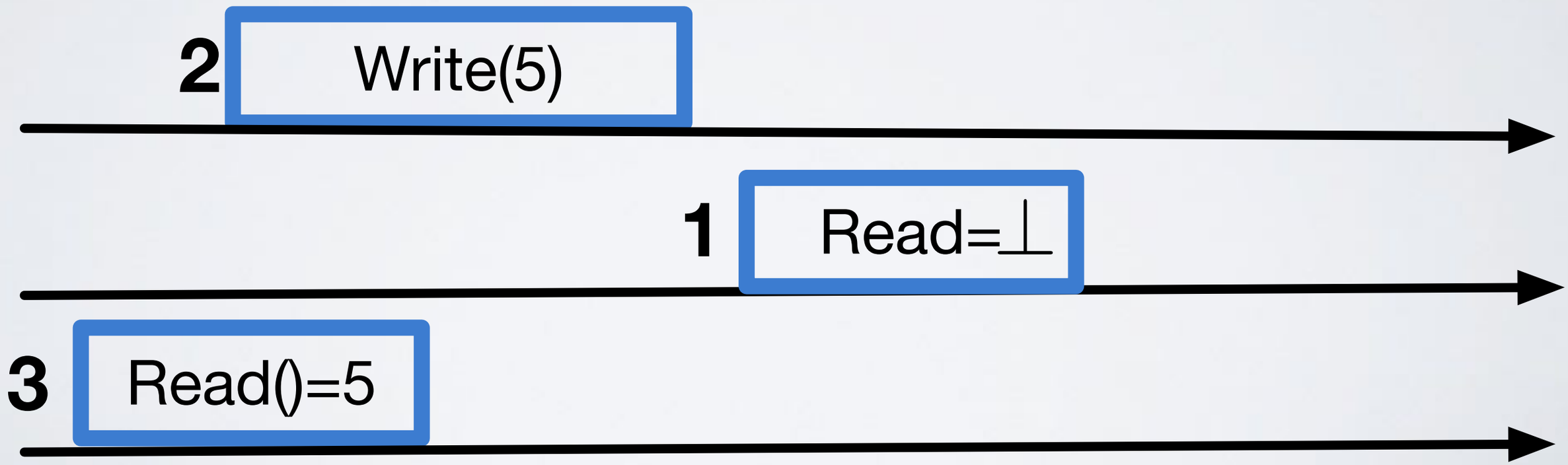
# SEQUENTIAL CONSISTENCY

*"The result of any execution is the same as if the (read and write)* ***operations by all processes*** *on the data store were executed in* ***some sequential order*** *and the* ***operations of each individual process appear in*** *this sequence* ***in the order specified by its program***.*"* Lamport 1979.



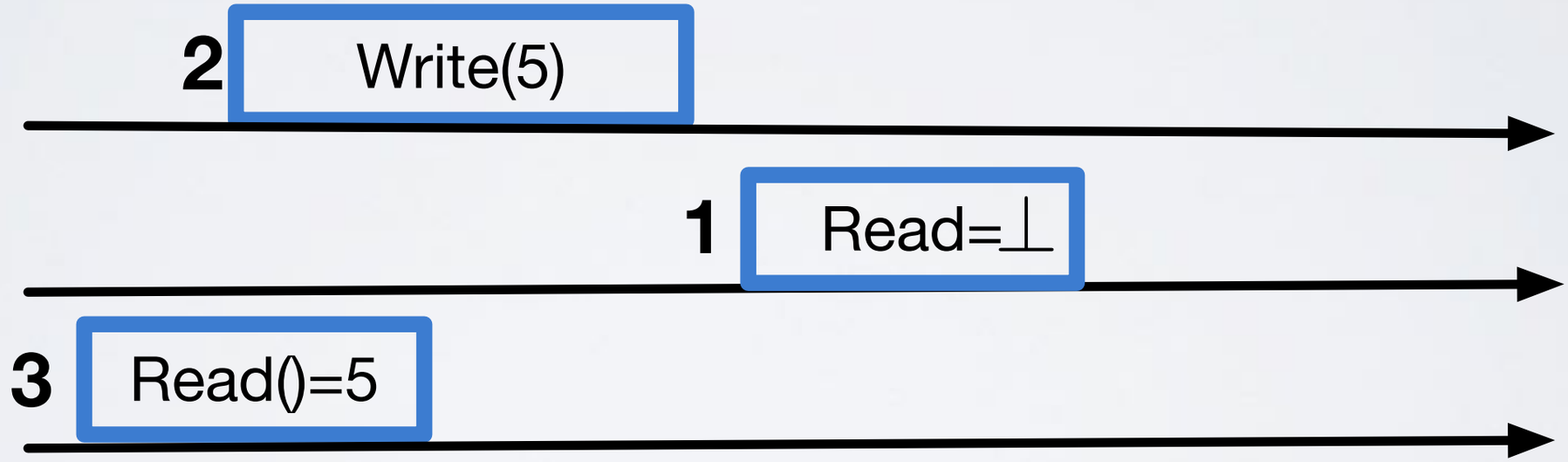
# SEQUENTIAL CONSISTENCY

*"The result of any execution is the same as if the (read and write)* ***operations by all processes*** *on the data store were executed in* ***some sequential order*** *and the* ***operations of each individual process appear in*** *this sequence* ***in the order specified by its program***.*"* Lamport 1979.



# SEQUENTIAL CONSISTENCY

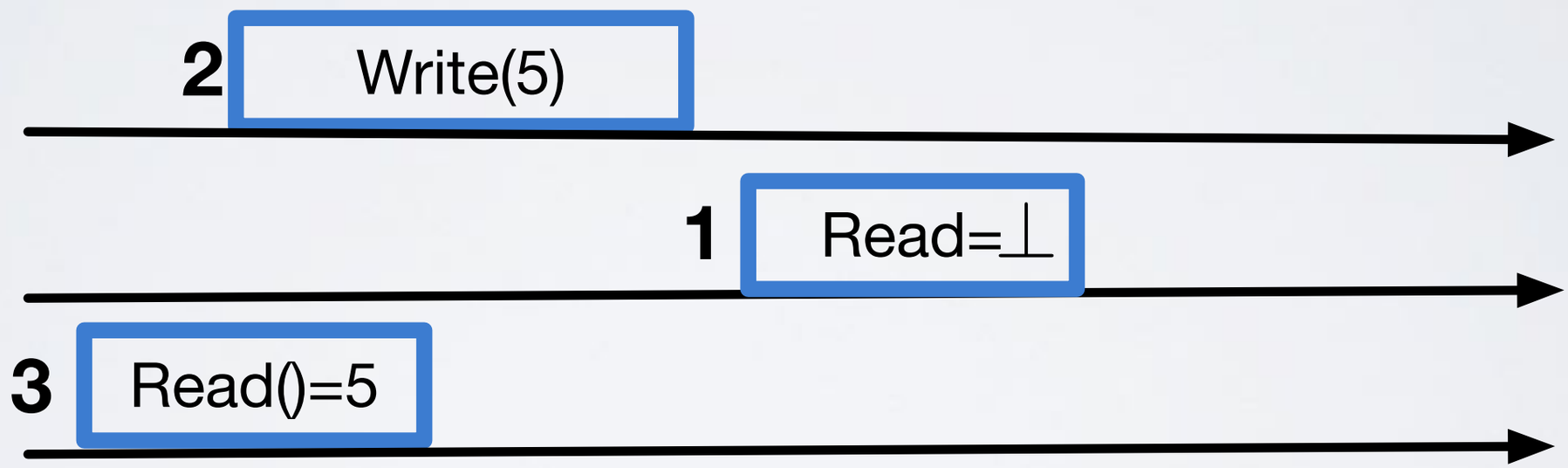
*"The result of any execution is the same as if the (read and write)* ***operations by all processes*** *on the data store were executed in* ***some sequential order*** *and the* ***operations of each individual process appear in*** *this sequence* ***in the order specified by its program****."* Lamport 1979.




The sequential order is p2 reads \bot, p1 writes 5 and then p3 reads 5

# SEQUENTIAL CONSISTENCY

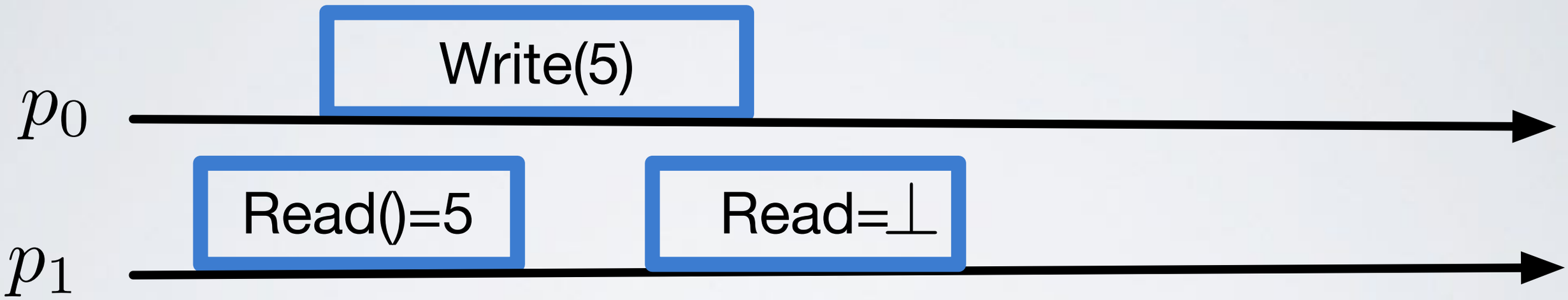
*There exists a global ordering, that respects the local ordering seen by each process.*




The sequential order is p2 reads \bot, p1 writes 5 and then p3 reads 5

# SEQUENTIAL CONSISTENCY

*There exists a global ordering, that respects the local ordering seen by each process.*



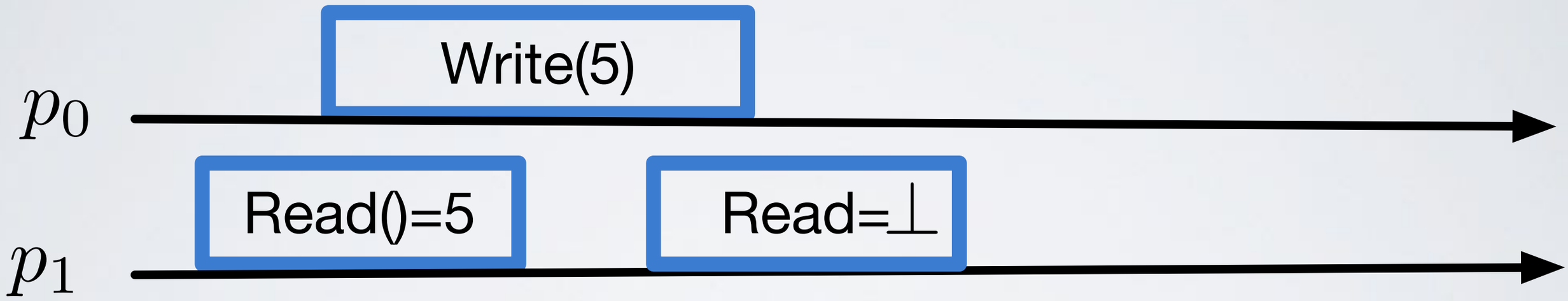

**SHUFFLE1: P0W, P1R, P1R**

**SHUFFLE2: P1R,P0W,P1R**

**SHUFFLE3: P1R,P1R,P0W**

# SEQUENTIAL CONSISTENCY

*There exists a global ordering, that respects the local ordering seen by each process.*



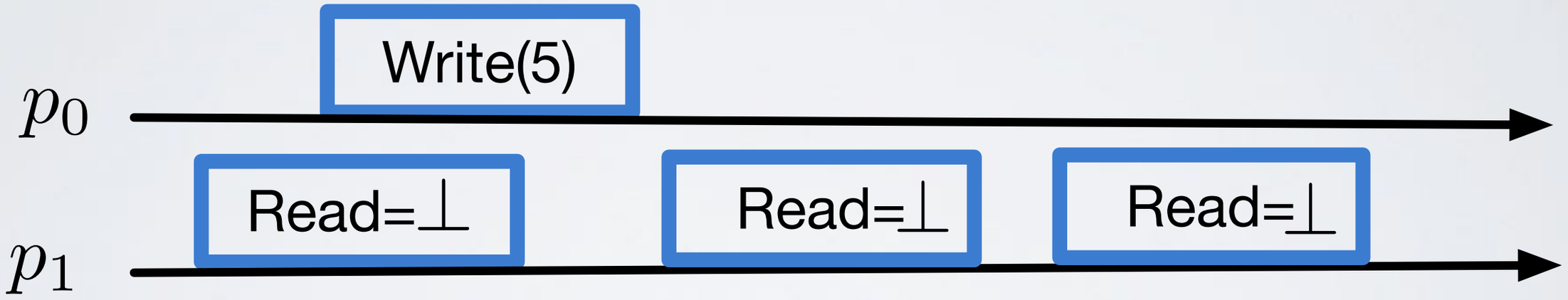

**SHUFFLE1: P0W, P1R, P1R -> P0W(5),P1R=5,P1R=5**

**P1R,P0W,P1R-> P1R=\BOT, P0W(5), P1R=5**

**P1R,P1R,P0W -> P1R=\BOT, P1R=\BOT, POW(5)**

# SEQUENTIAL CONSISTENCY

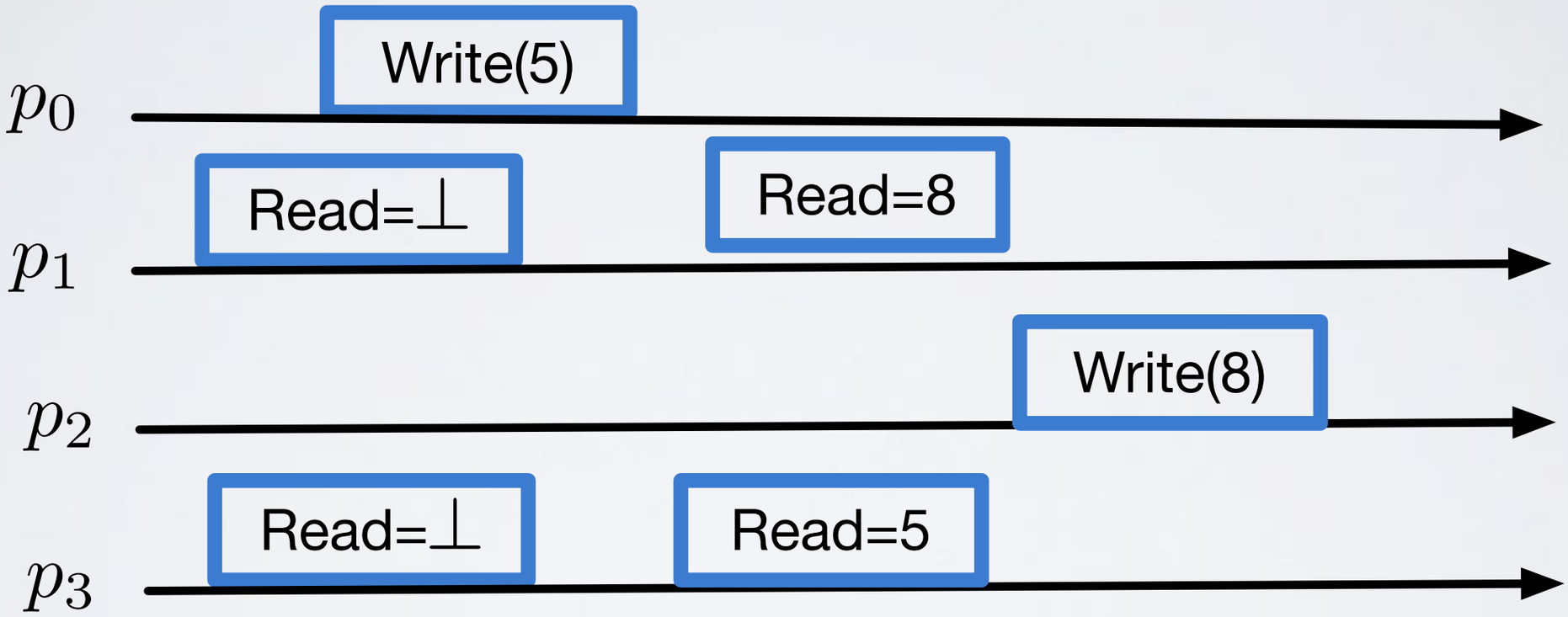
*Sequential consistency gives to each process the illusion of using a single storage. Even if the results does not respect the global constraint that happens in the real execution.*




**SeqEx: P1R=\bot, P1R=\bot, P1R=\bot, P0W(5)**

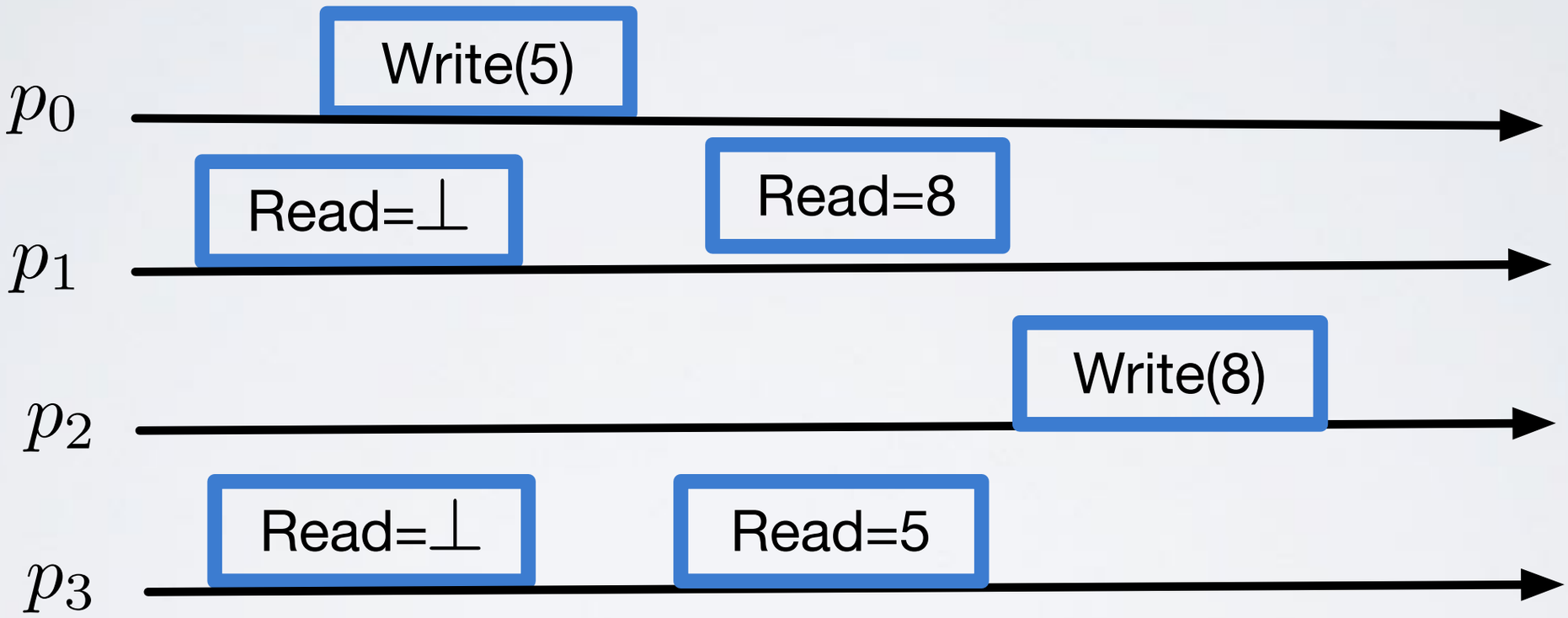
# SEQUENTIAL CONSISTENCY

*Sequential consistency gives to each process the illusion of using a single storage. Even if the results does not respect the global constraint that happens in the real execution.*



# SEQUENTIAL CONSISTENCY

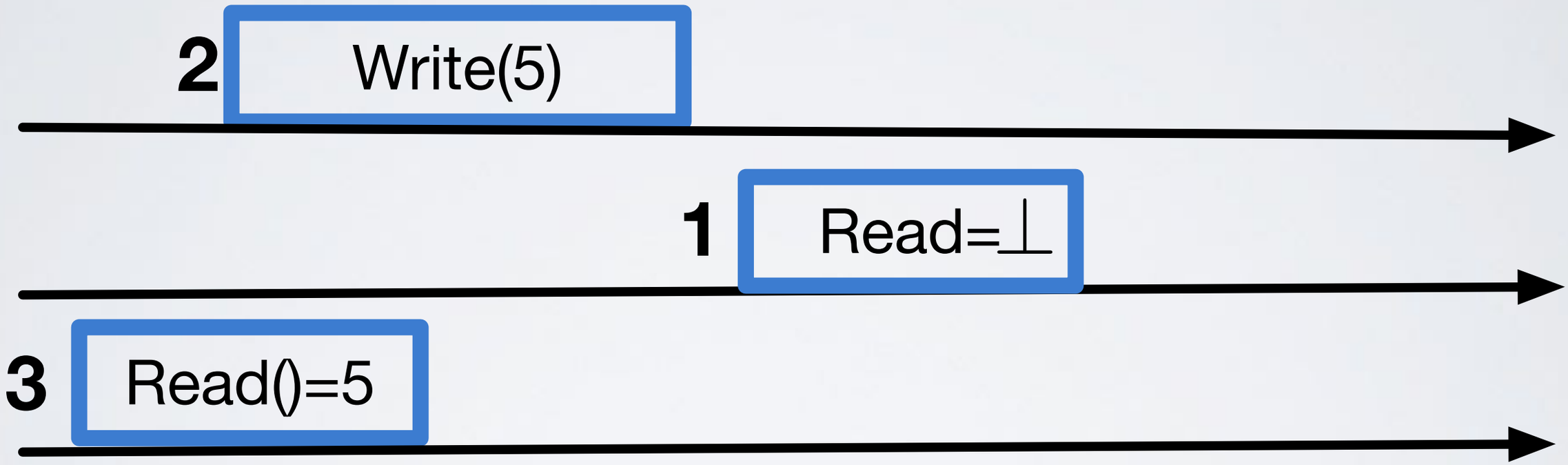
*Sequential consistency gives to each process the illusion of using a single storage. Even if the results does not respect the global constraint that happens in the real execution.*




**SeqEx.: P1R,P3R, P0W(5),P3R,P2W(8),P1R**

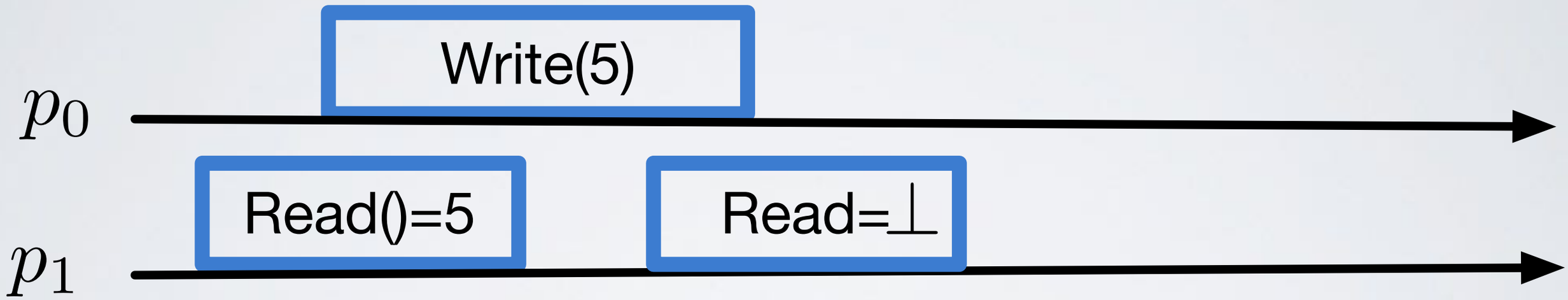
# SEQUENTIAL CONSISTENCY

This run is sequential consistent but **not REGULAR**



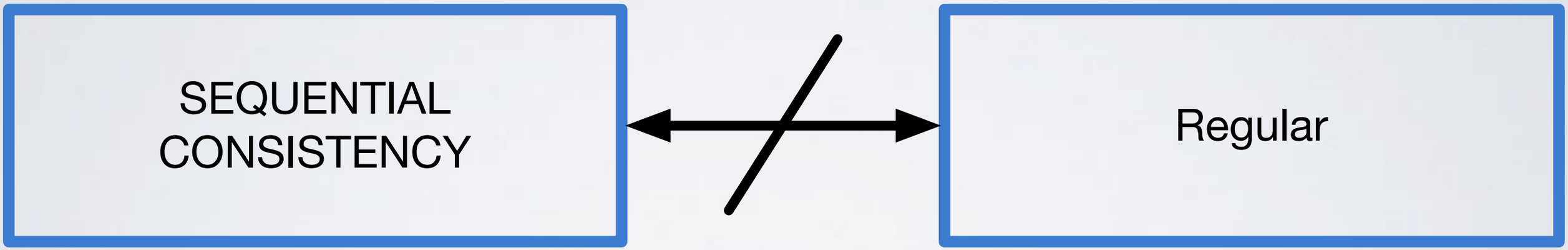
# SEQUENTIAL CONSISTENCY

This run is regular  but not **sequential consistent**



# SEQUENTIAL CONSISTENCY VS REGULAR

Sequential consistency and regular consistency are orthogonal.



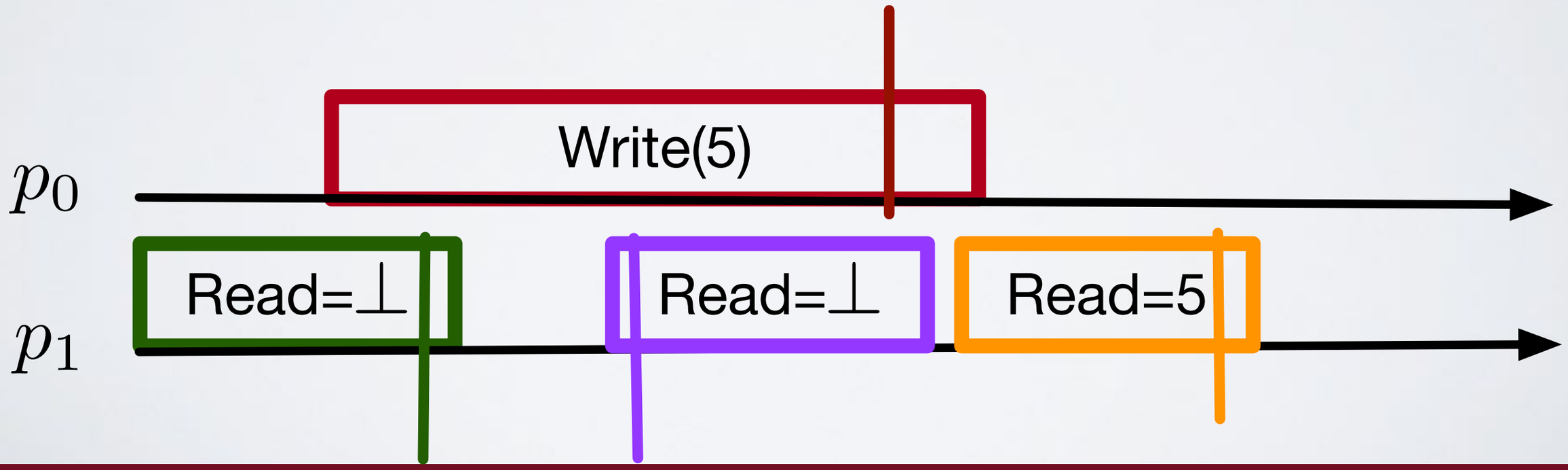
# ATOMICITY

Atomicity (or linearizzability):

*"Each operation should appear to take effect instantaneously at some moment between its start and completion."*

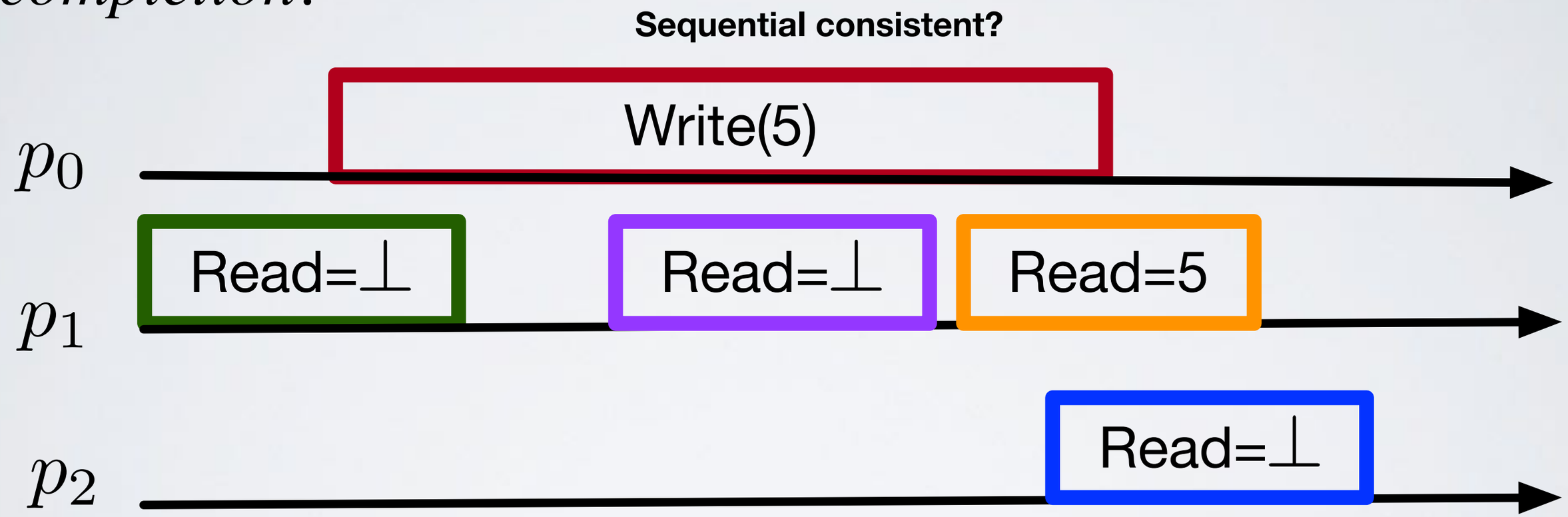
The ``moment'' is the linearisation point of the operation

**The horizontal line is the linearisation point of each operation.**



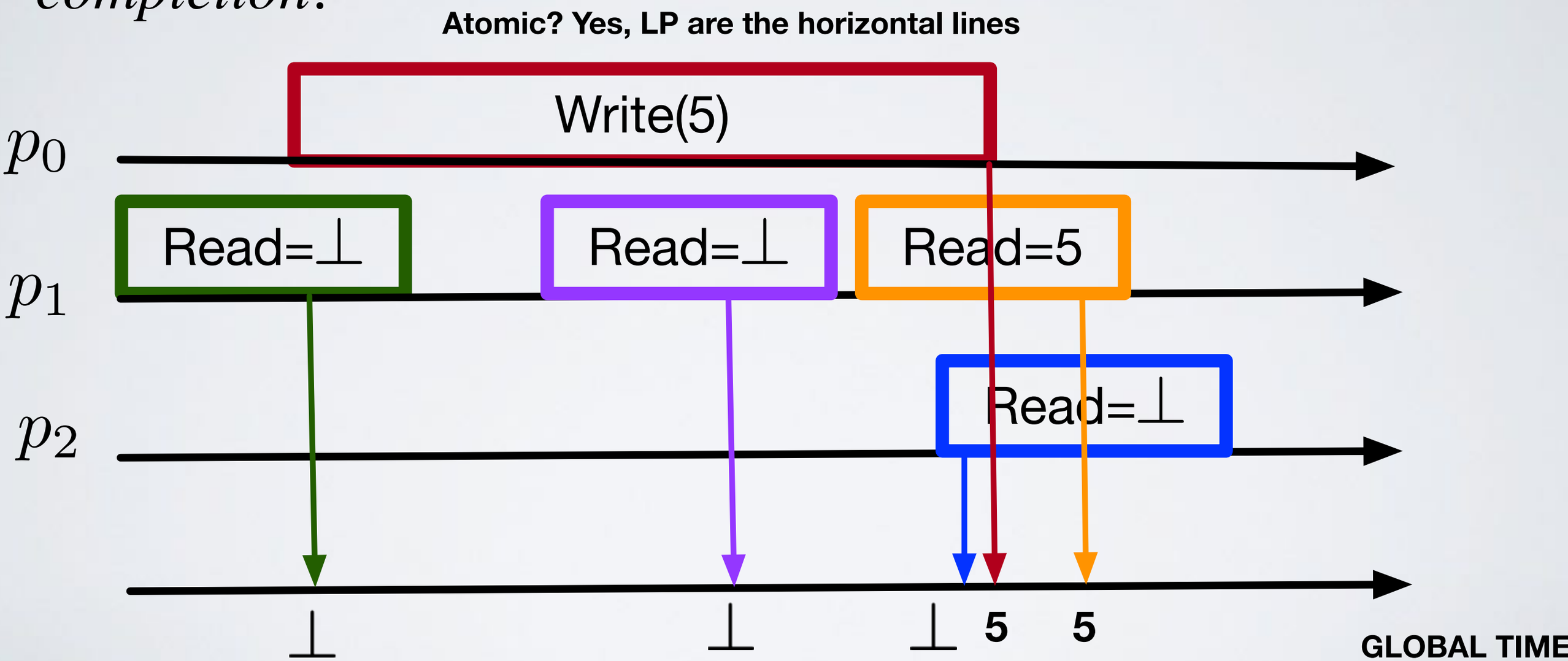
# ATOMICITY

*"Each operation should appear to take effect instantaneously at some moment between its start and completion."*



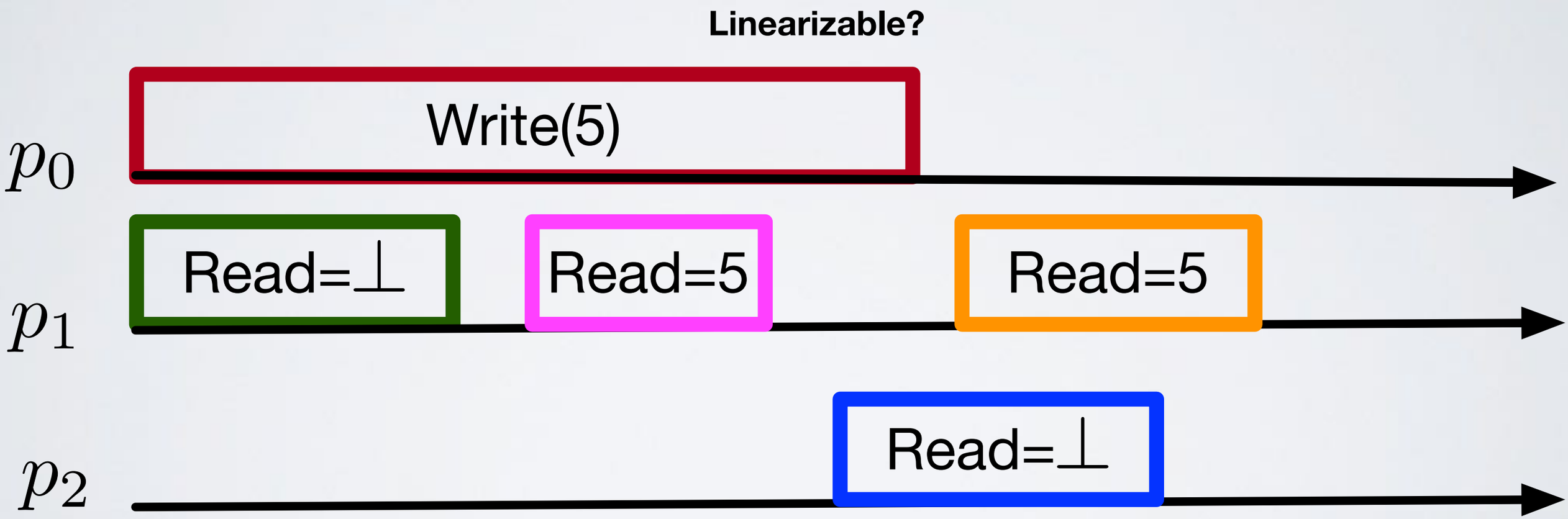
# ATOMICITY

*"Each operation should appear to take effect instantaneously at some moment between its start and completion."*



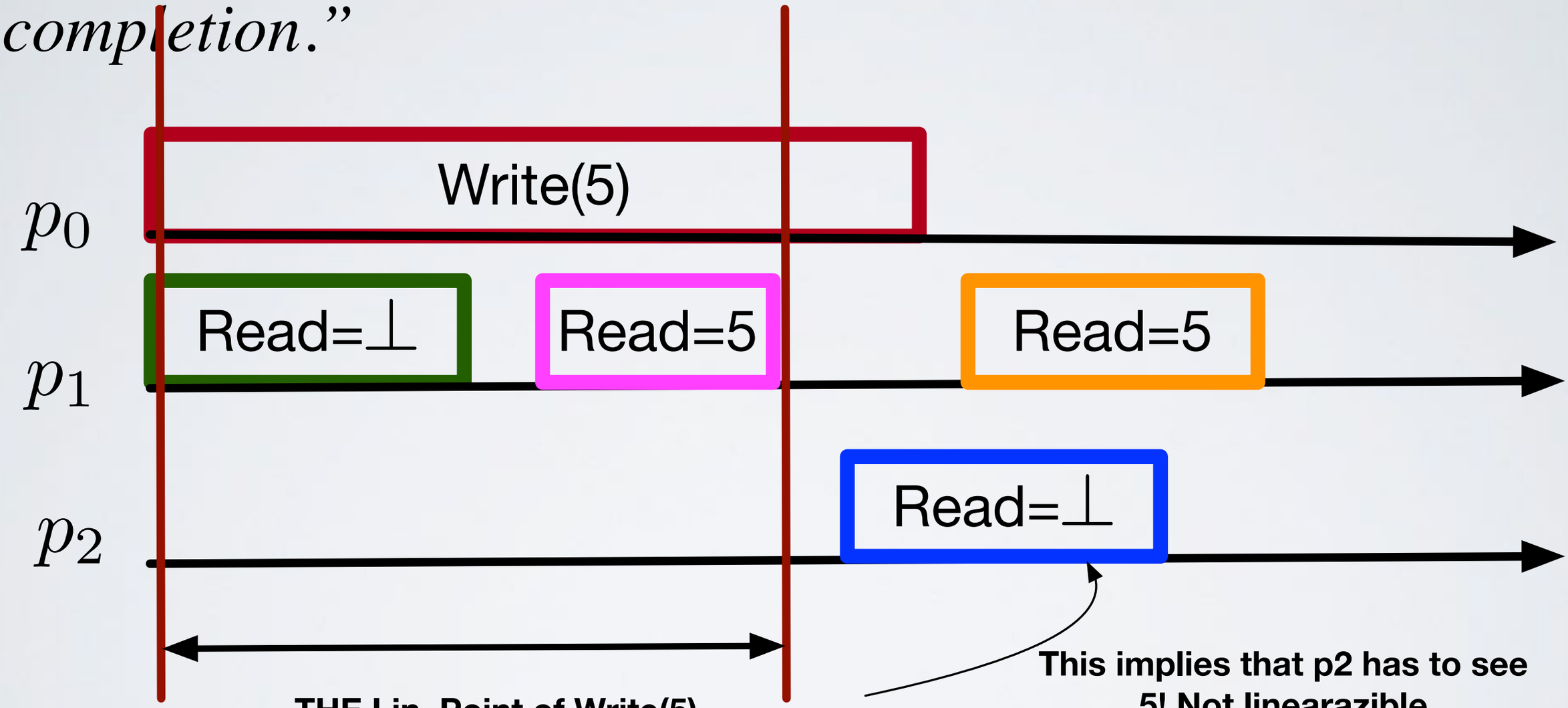
# ATOMICITY

*"Each operation should appear to take effect instantaneously at some moment between its start and completion."*



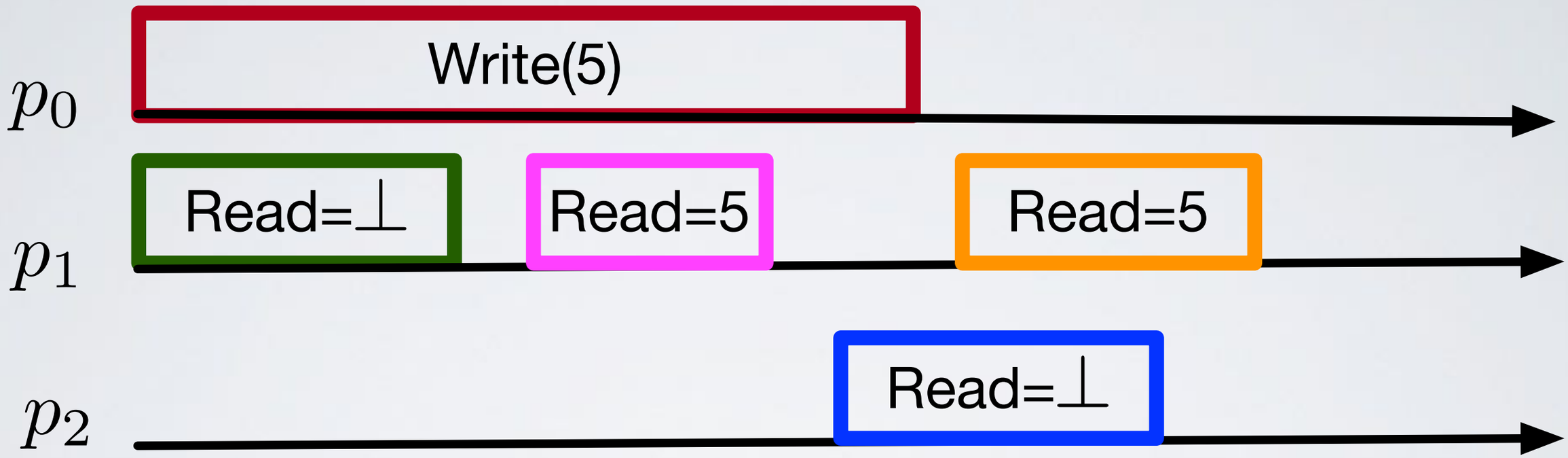
# ATOMICITY

*"Each operation should appear to take effect instantaneously at some moment between its start and completion."*



# ATOMICITY

This run is bot regular and sequential cons. but not atomic

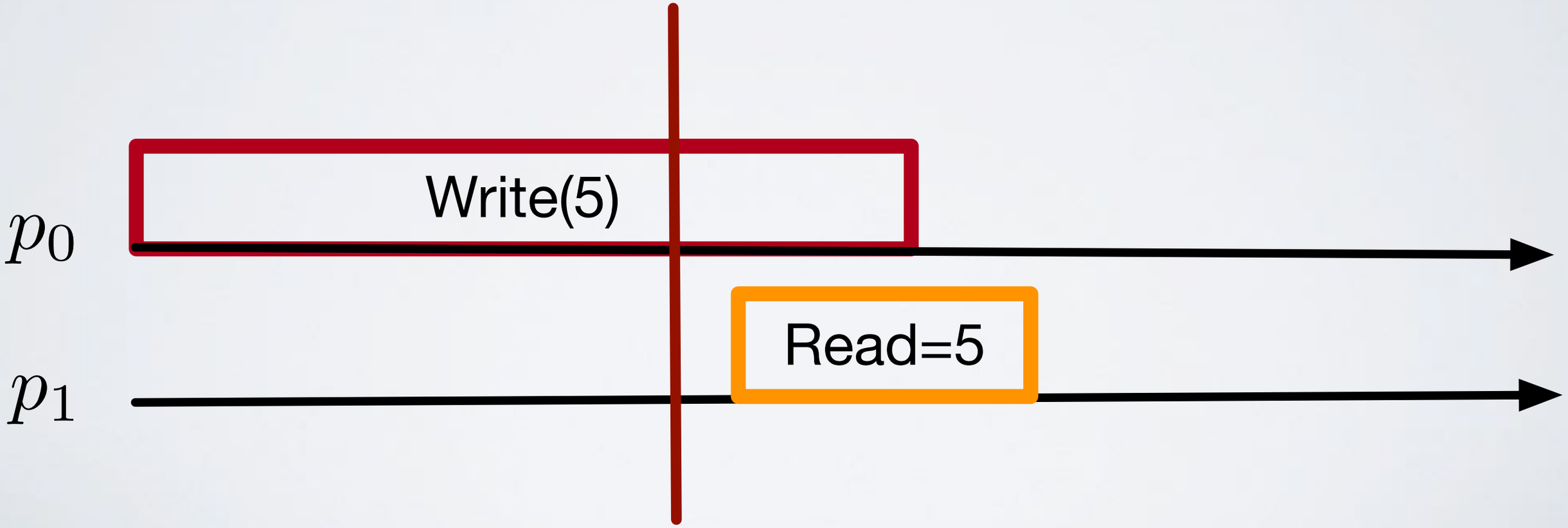



Seq Ex. P2R, P1R, P0W, P1R, P1R

# ATOMICITY

Any linearisable run is also regular! Linearizability implies regular.

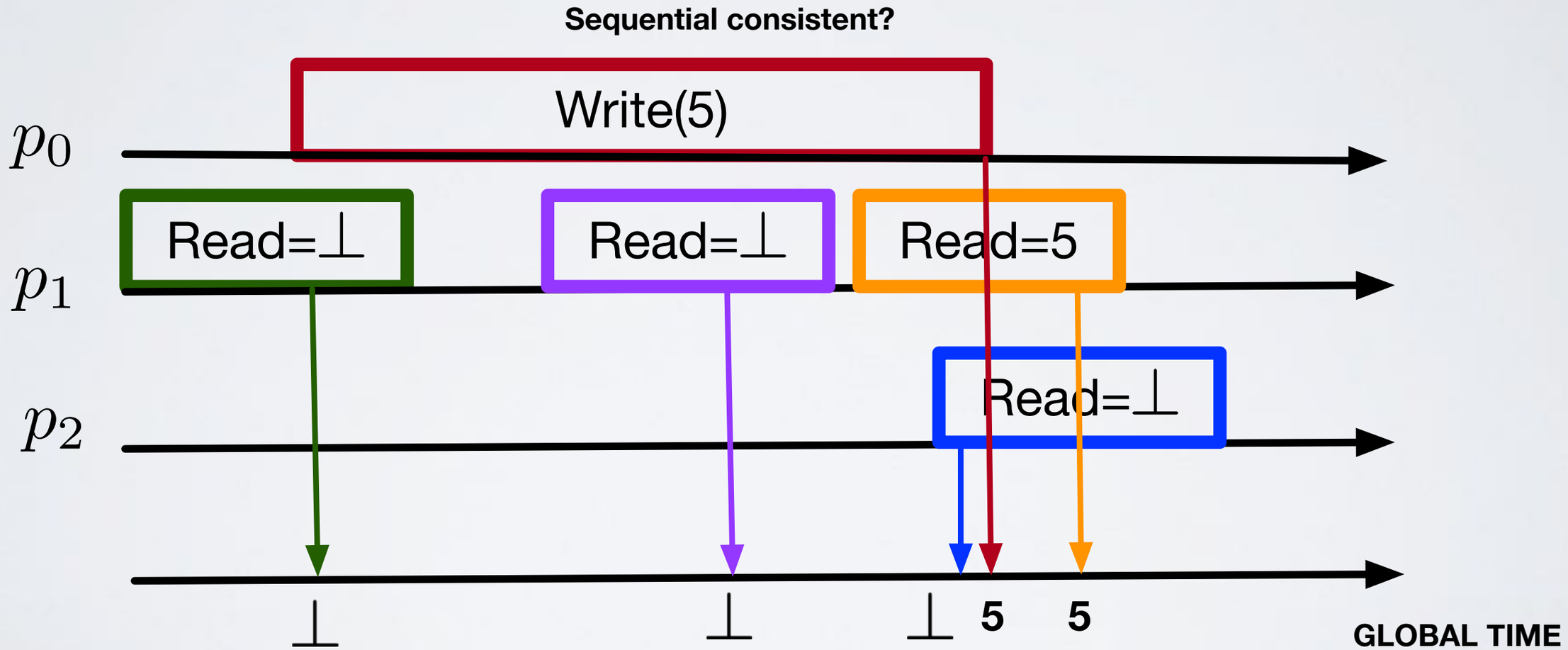
Intuition: a write as a linearisation point (LP), after the LP anyone sees the write. The LP happens inside the write (by definition), thus each successive read has to read the value.



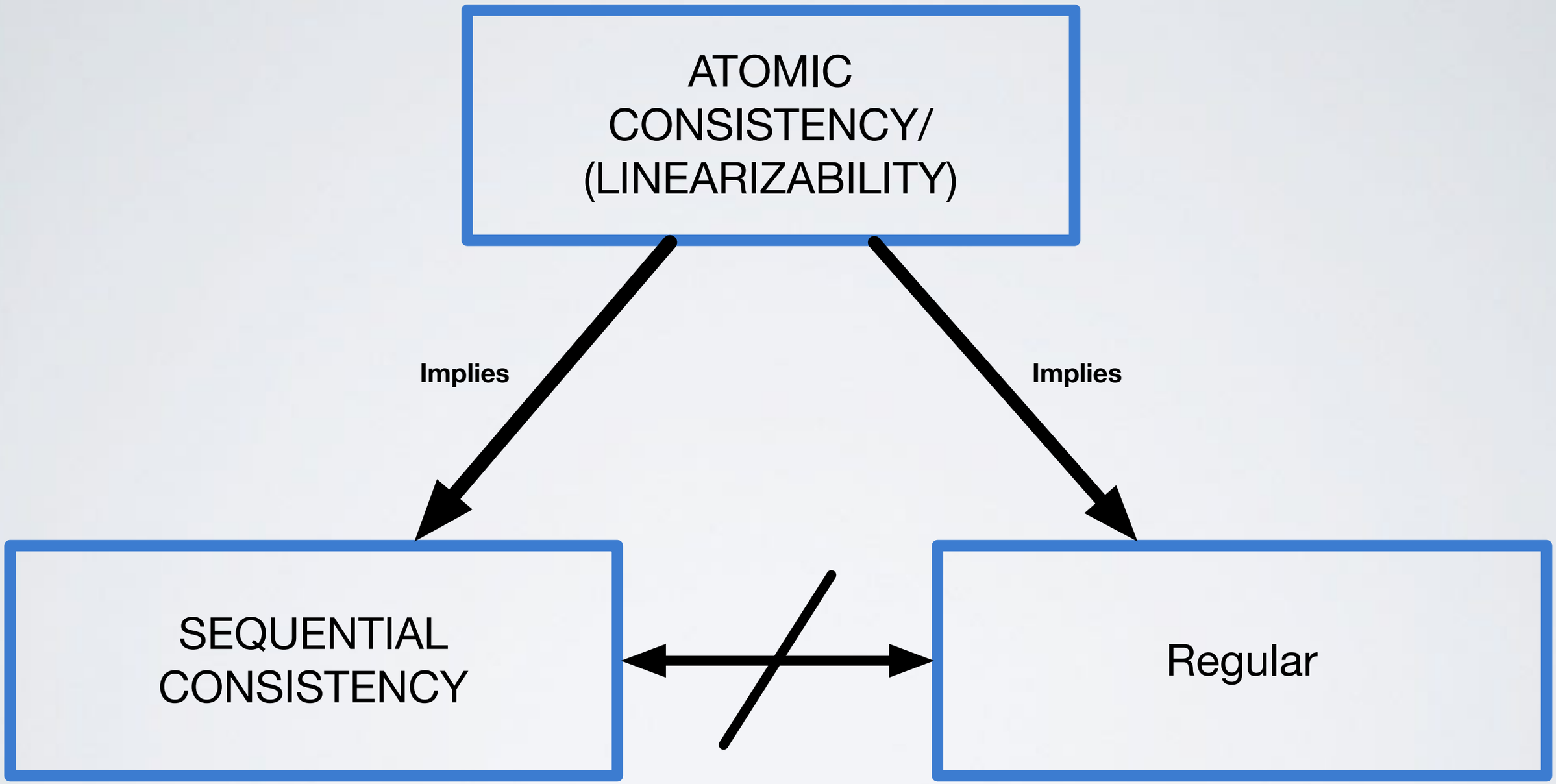
# ATOMICITY

Any linearisable run is also sequential cons. Linearizability implies sequential consistency

Intuition: take the LPs of all operations. This points in time define a sequential execution that respects the local order!



# RELATIONSHIP BETWEEN CONS.



# COMPOSITIONALITY OF CONSISTENCY CONDITIONS

# COMPOSITIONALITY

Given a set of registers, such that each one of them independently respects a consistency condition, we would like that any execution on this set of registers respects the same consistency conditions.

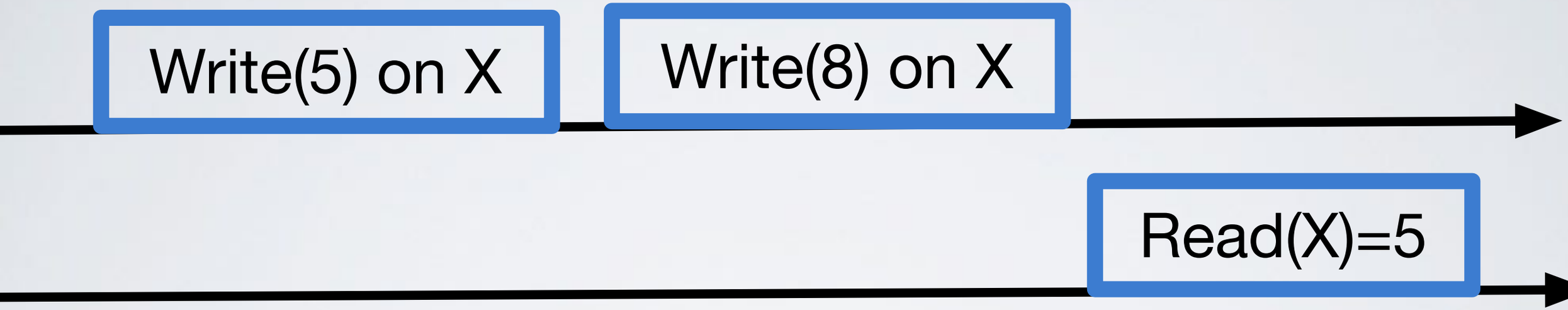
Stated more formally:

- Let an execution E be the sequence of write and read operations on a set of registers R.
- E|r_i be the sequence of operations in E that operate on register r_i.

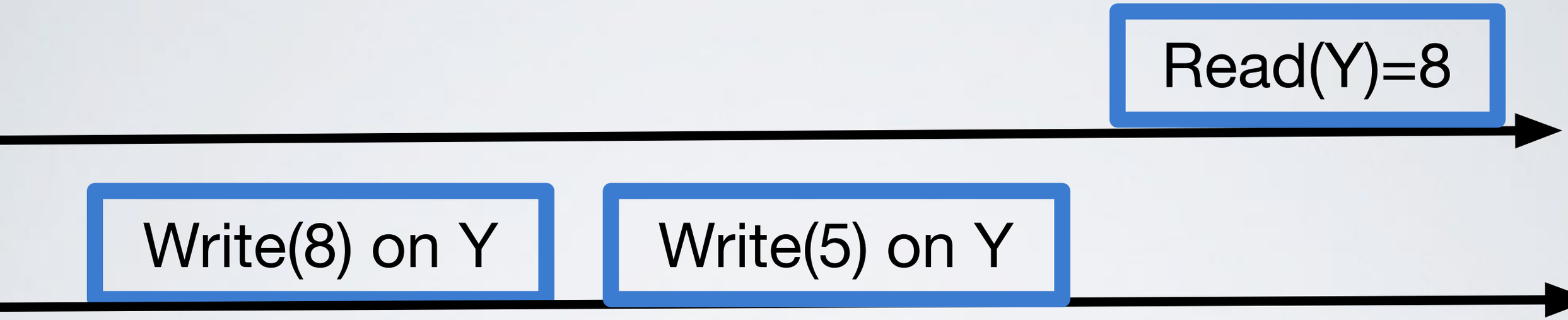
We say that a consistency property C is compositional iff:

$$\forall r_i \in R | C(E|r_i) = True \iff C(E) = True$$

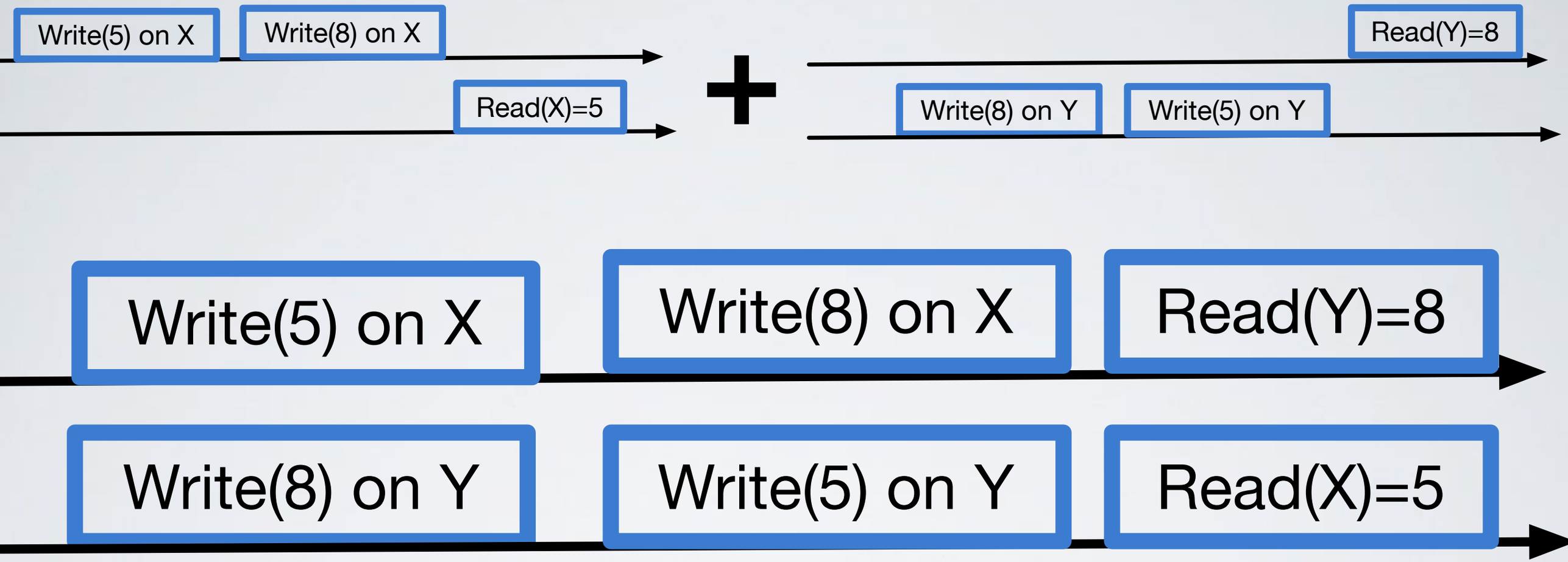
# SEQUENTIAL CONSISTENT



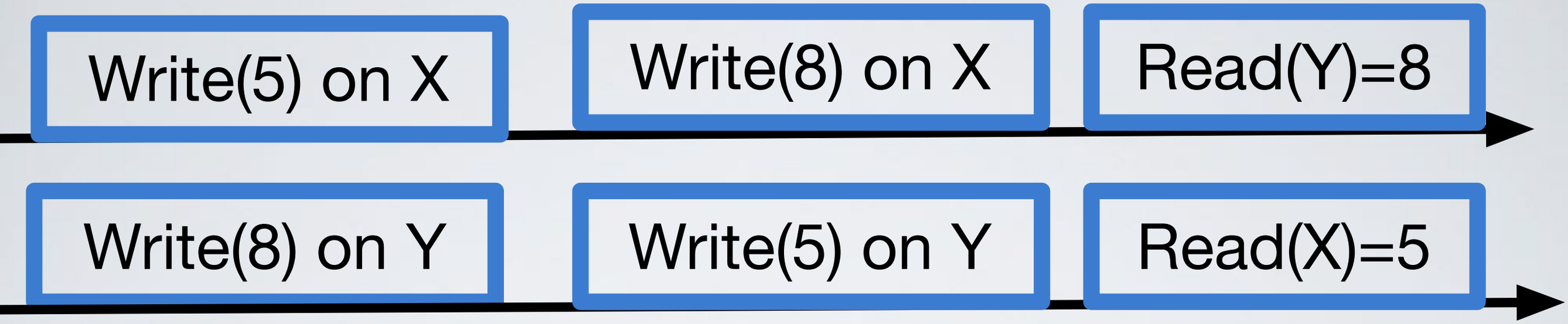
# SEQUENTIAL CONSISTENT



# SEQUENTIAL CONSISTENT?



# SEQUENTIAL CONSISTENT?




**The only possible order has to have P0R(Y) before P1W(5,Y) and after P1W(8,Y).**

**P0W(5,X),P1W(8,Y),P0W(8,X),P0R(1)=8, P1W(5,Y), P1R(X)=8**
**But this violates the value seen on P1R(X)!**

**SEQUENTIAL CONSISTENCY IS NOT COMPOSITIONAL**

# ATOMIC AND REGULAR

Atomic and regular consistency are compositional:

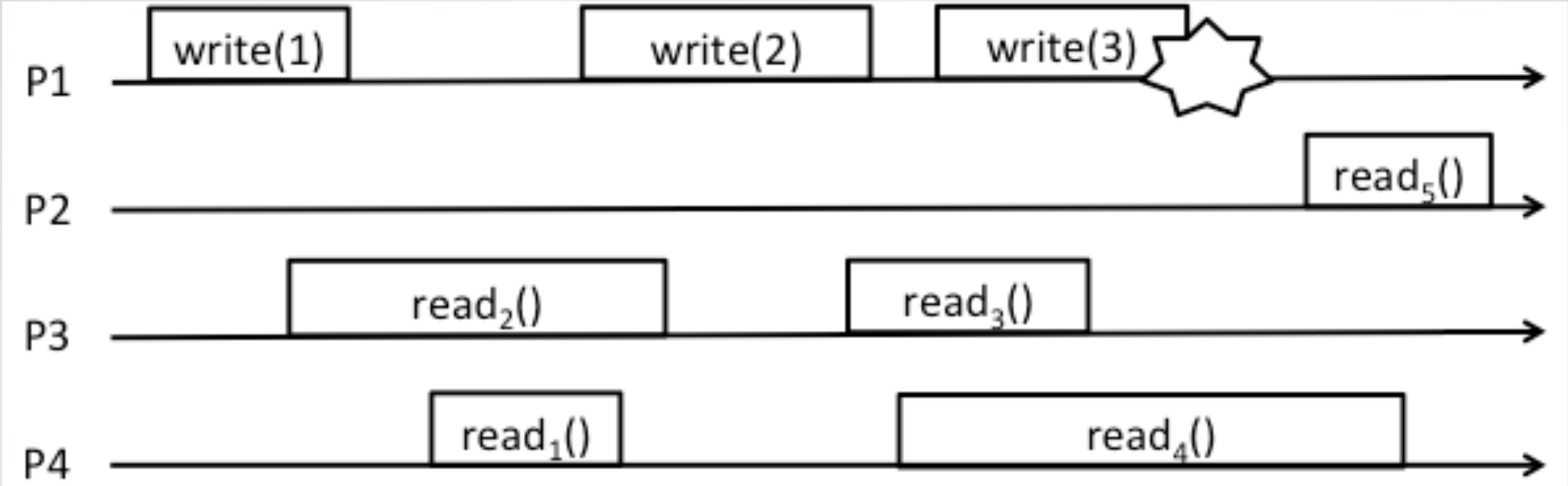
- You can use a set of atomic registers independently implemented, and the resulting executions is linearisable
- You can use a set of regular registers  independently implemented, and the resulting executions is regular

Sequential consistency is not compositional:

- If you use a set of sequentially consistent registers that are independently implemented, the resulting run is not always sequentially consistent.

# EXERCISE

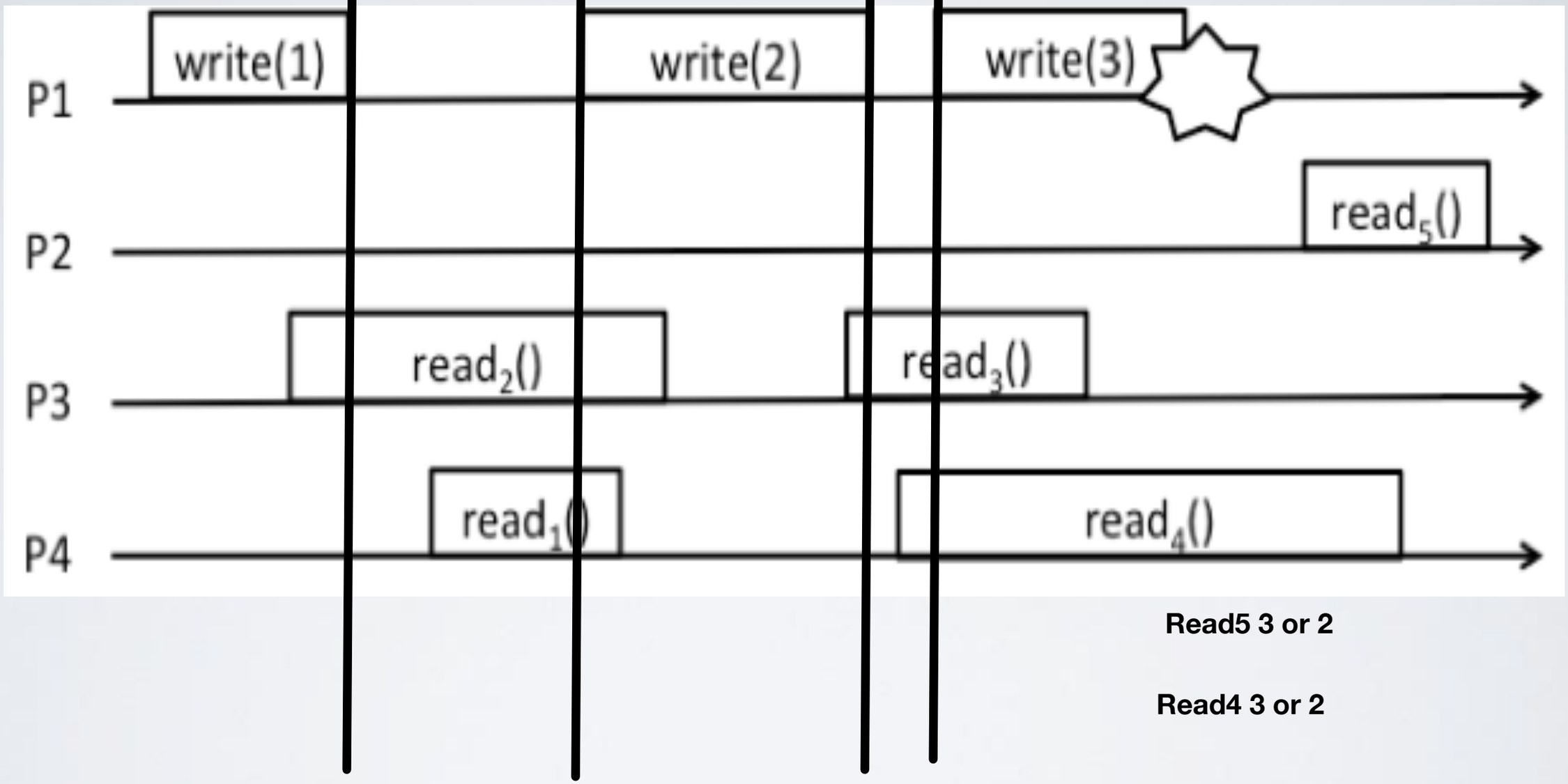
Consider the execution depicted in the following figure and answer the questions




1. Define ALL the values that can be returned by read operations (Rx) assuming the run refers to a regular register.

2. Define ALL the values that can be returned by read operations (Rx) assuming the run refers to an atomic register.

# EXERCISE - REGULAR SOLUTION




Read5 3 or 2

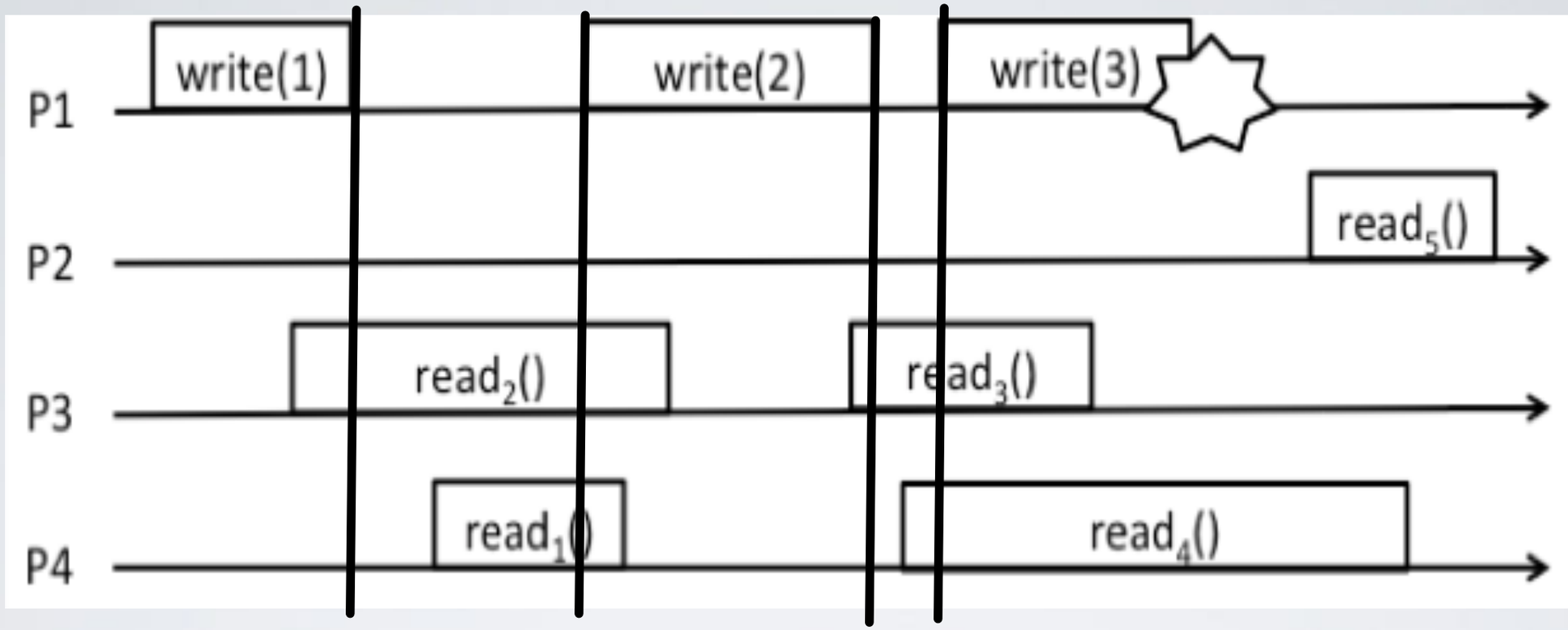
Read4 3 or 2

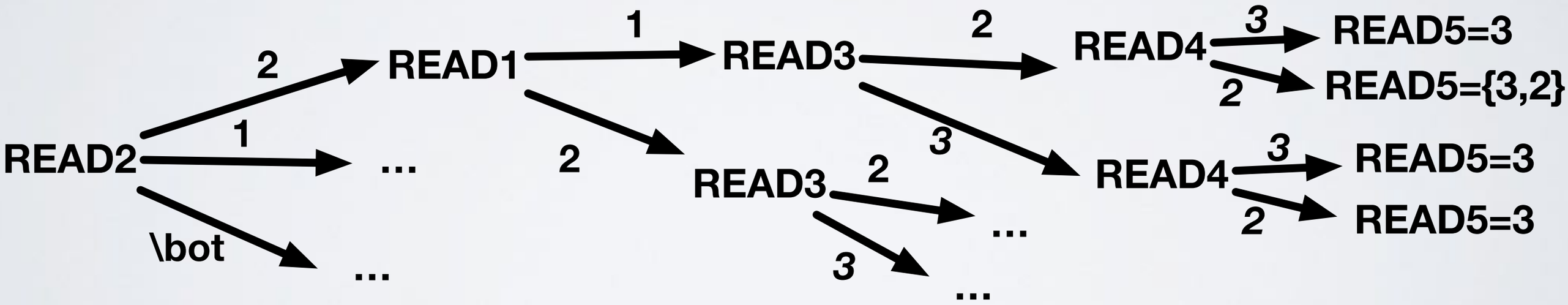
Read2 Intersects two writes, it can return \bot,1,2

Read3 intersects two writes, read 1,2, 3

Read1 inters. one write, so ti can return 1,2

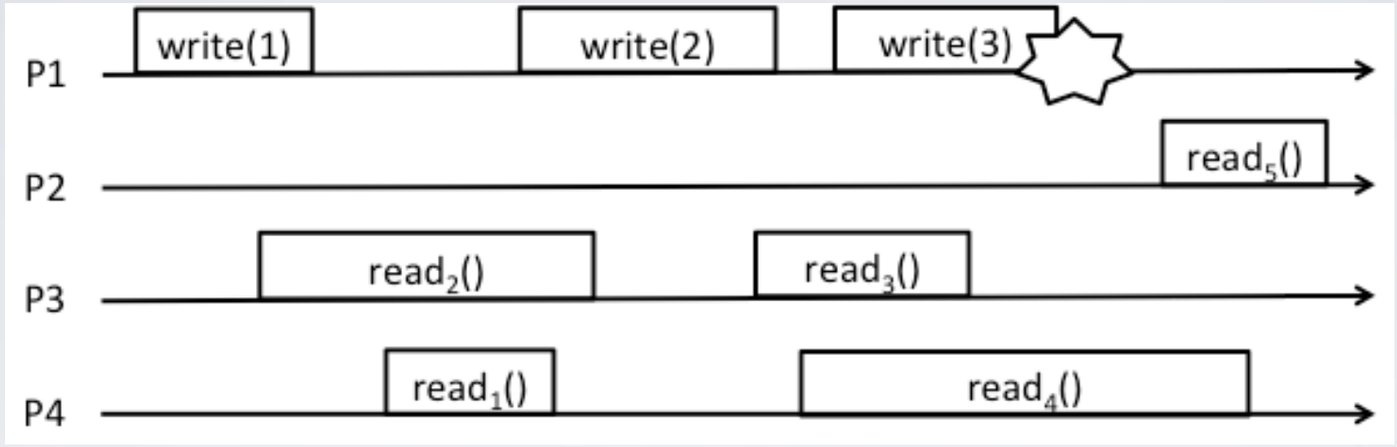
# EXERCISE - ATOMIC SOLUTION

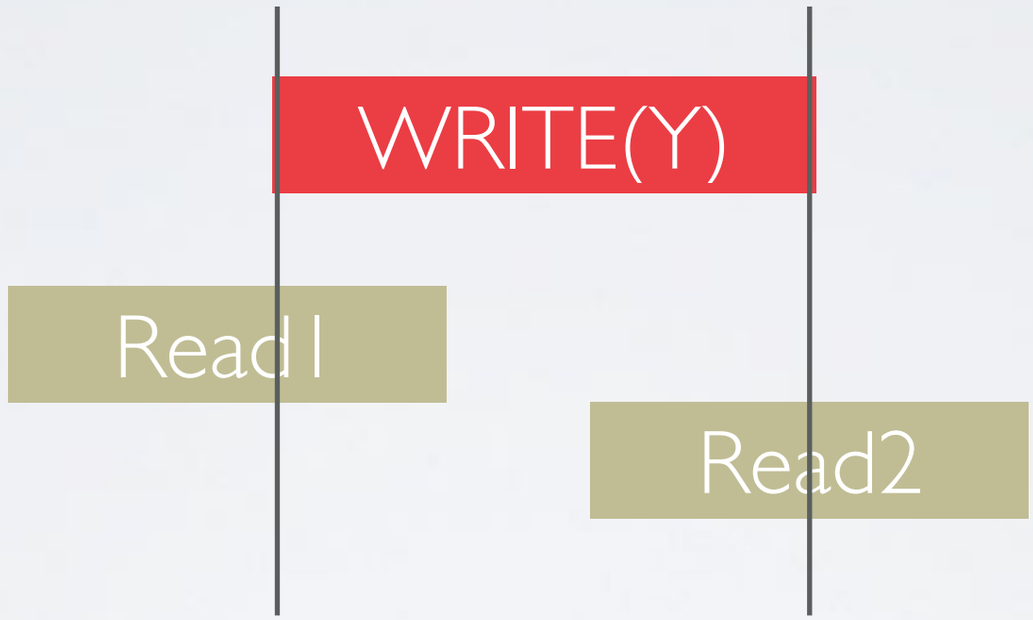



A path leaf-root is a possible atomic run

# EXERCISE - ATOMIC SOLUTION

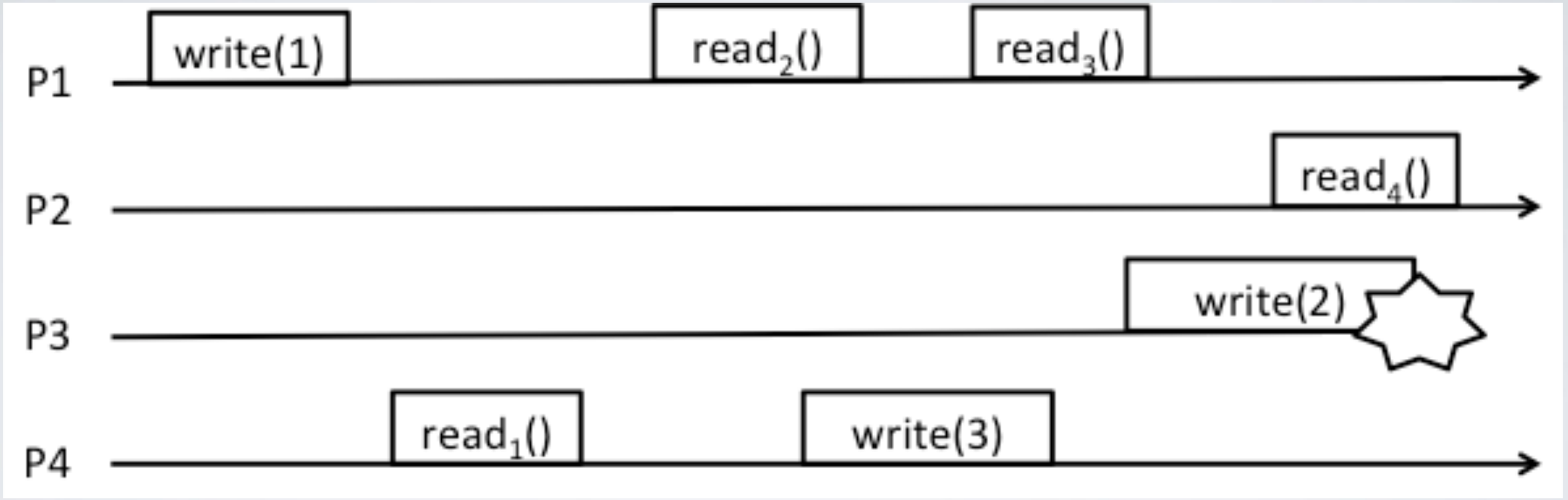



When you have this pattern
Read 1 influences Read 2.
If read1=Y then read2 Y or greater

# EXERCISE

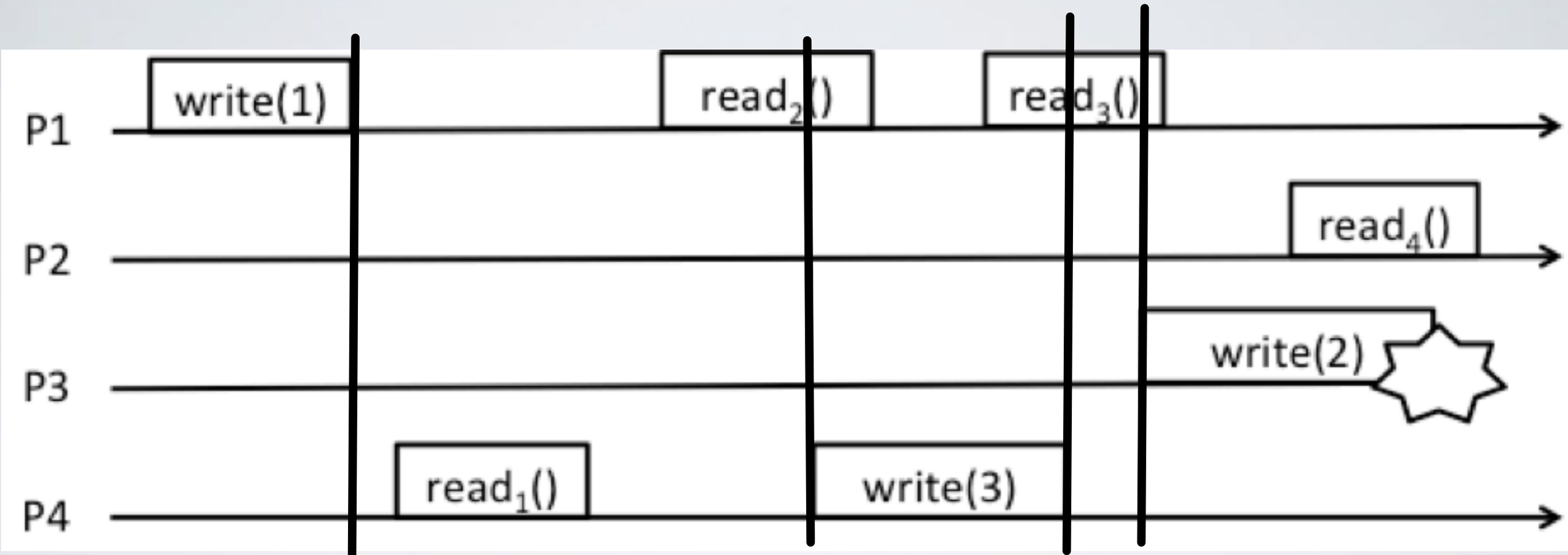
Consider the execution depicted in the following figure and answer the questions




1. Define <u>ALL the values that can be returned by read operations (Rx) assuming the run refers to a regular register.</u>

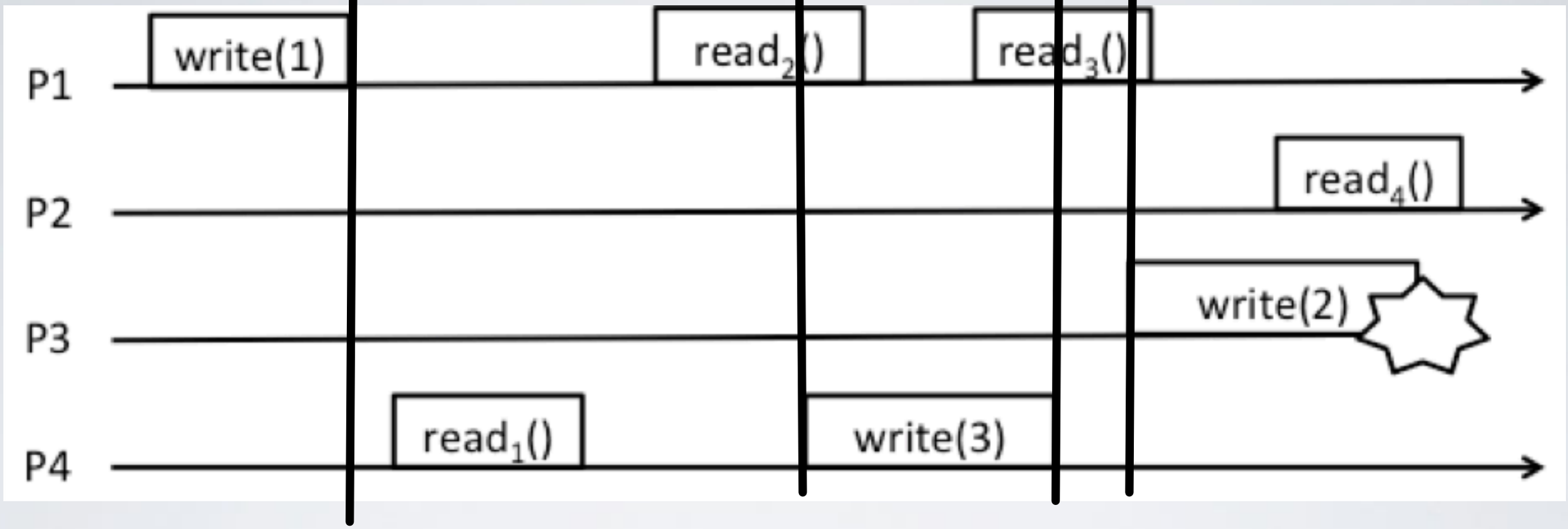
2. Define <u>ALL the values that can be returned by read operations (Rx) assuming the run refers to an atomic register.</u>

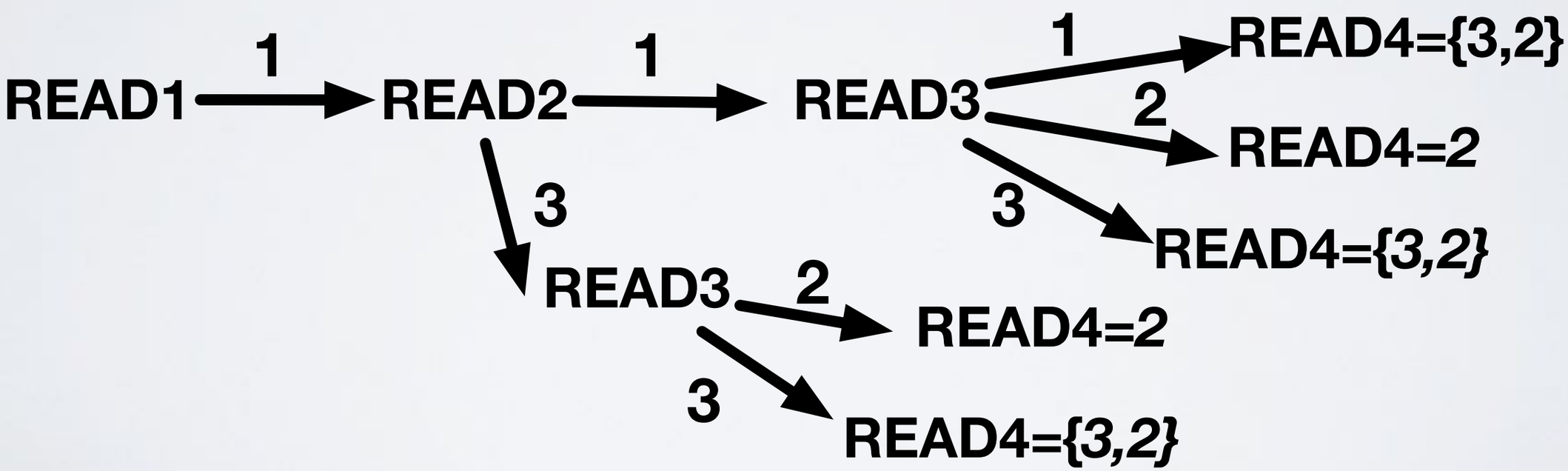
# EXERCISE- REGULAR SOLUTION




**READ1={1}** **READ2={1,3}** **READ3={1,3,2}** **READ4={3,2}**

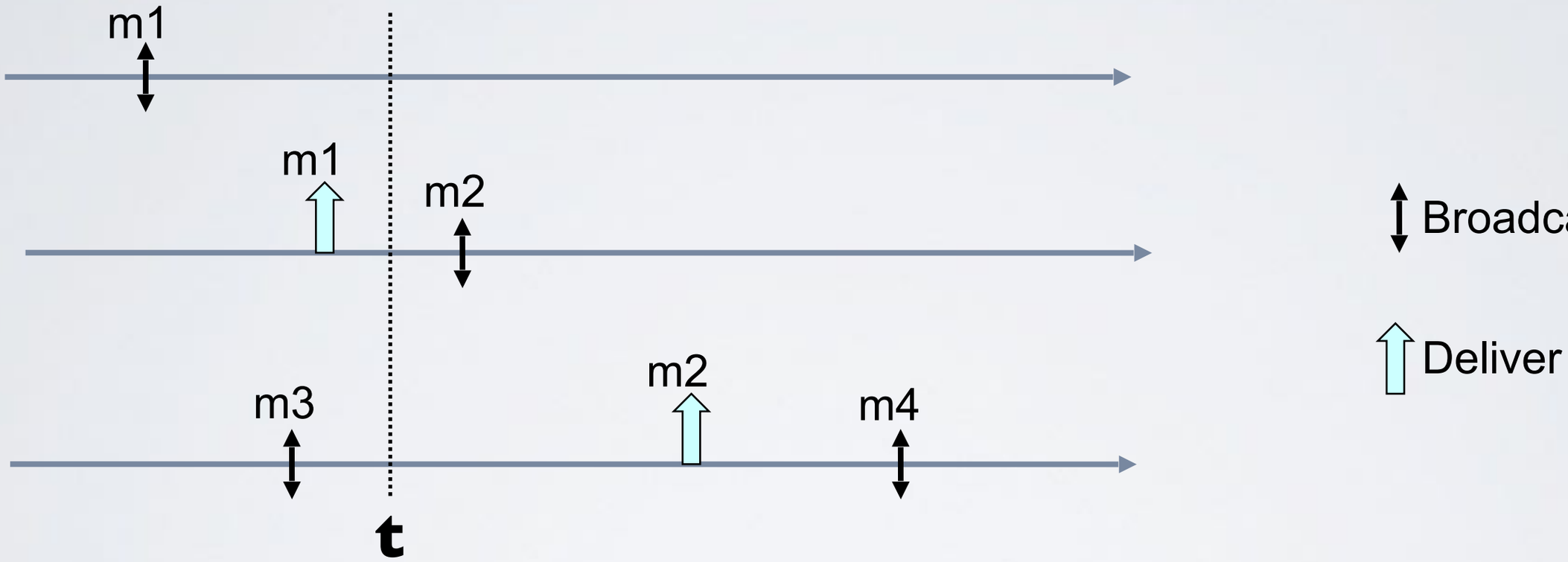
# EXERCISE- ATOMIC SOLUTION



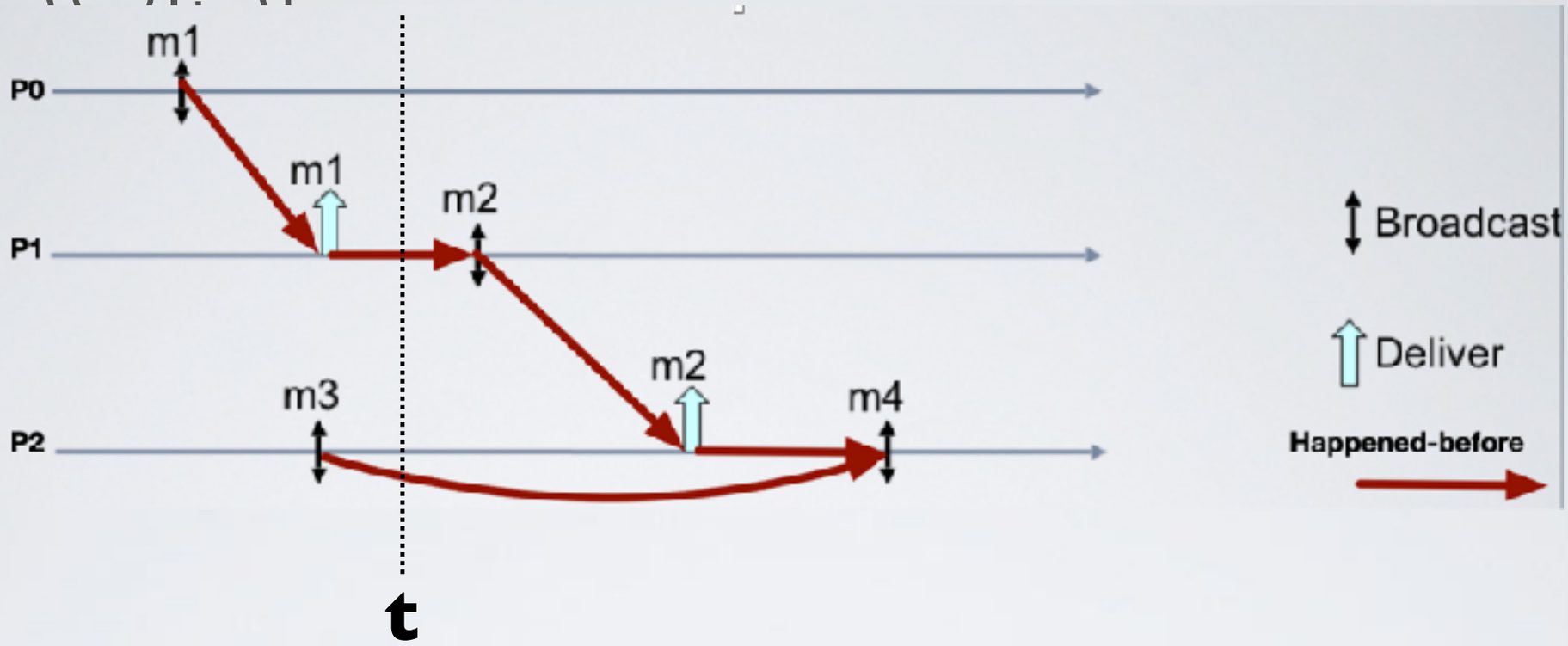



# EXERCISE

Provide all the delivery sequences after instant t such that causal order is satisfied.



# EXERCISE




M1 -> M2 -> M4 and M3 -> M4  We have M3 || M1 and M3 || M2

If P1 delivers M3 before Bcast(M2) then M3 -> M2.  Sequences, supposing M3 delivered after Bcast(M2) on P1:

P0:  M1, M2, M3, M4  or M3, M1, M2, M4 or M1, M3, M2, M4

P1:  M1, M2, M3,  M4 or M1, M3, M2, M4

P2:  M1, M2, M3, M4  or M3, M1, M2, M4 or M1, M3, M2, M4

If P1 delivers M3 before Bcast(M2) then:

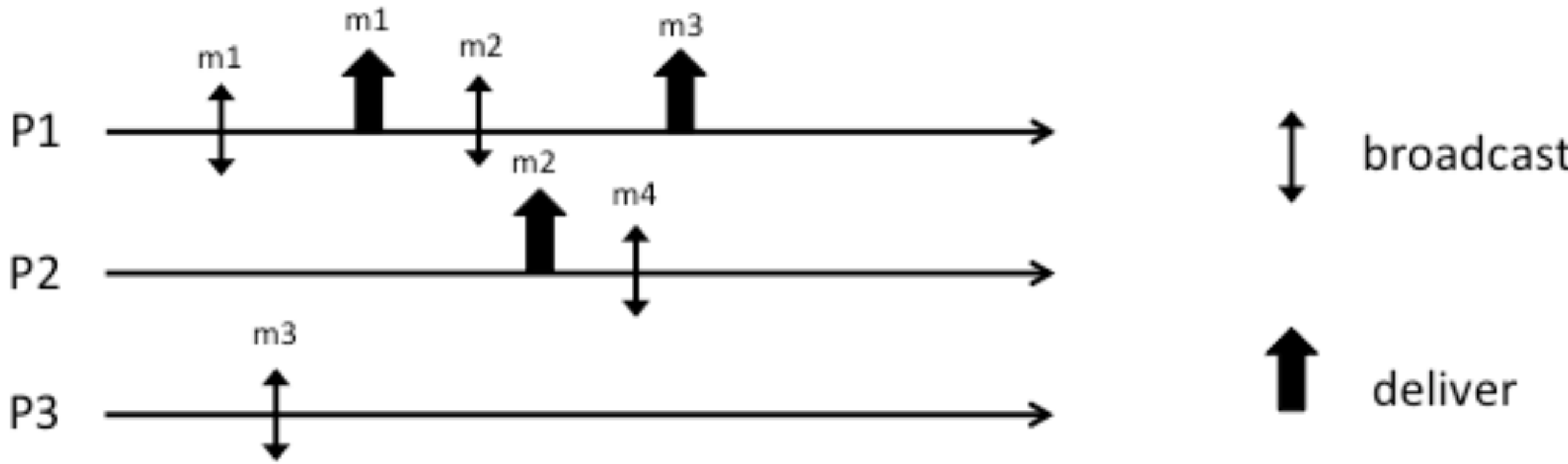
P0:  M3, M1, M2, M4 or M1, M3, M2, M4

P1: M1, M3, M2, M4

P2: M3, M1, M2, M4 or M1, M3, M2, M4

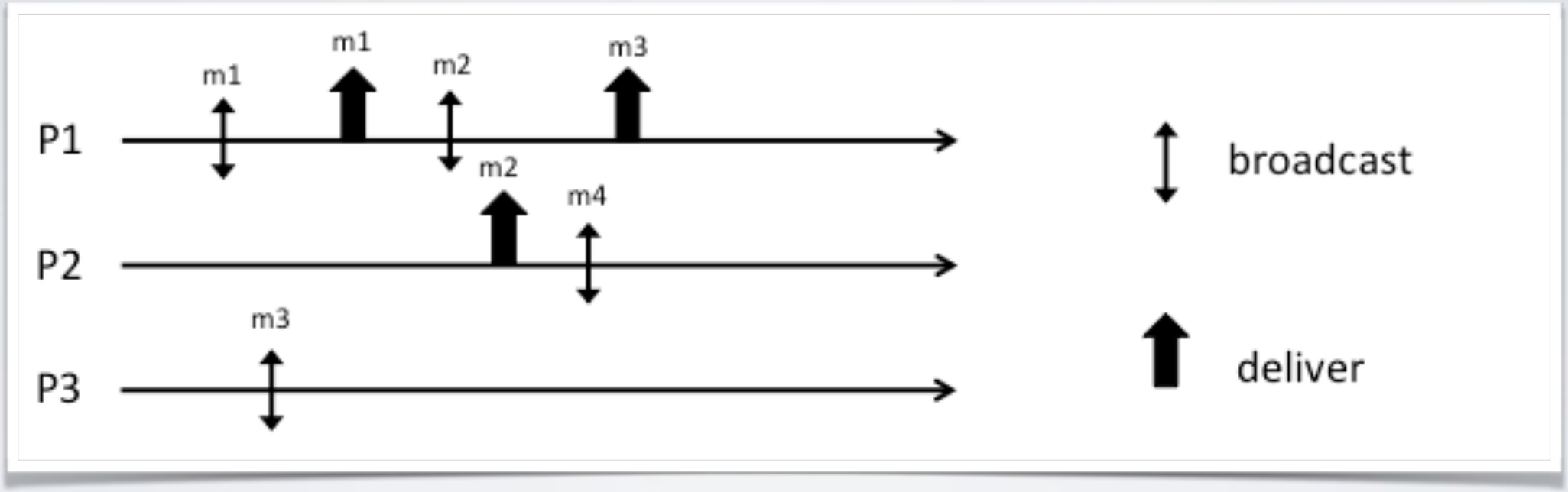
# EXERCISE

1. Provide all the possible sequences satisfying Causal Order
2. Complete the execution in order to have a run satisfying FIFO order but not causal order



# EXERCISE

1. Provide all the possible sequences satisfying Causal Order




M1 -> M2 -> M4 concurrent M3 || M1 , M3 || M2 and M3 || M4

If P2 delivers M3 before Bcast(M4) then M3 -> M4. Sequences, supposing M3 delivered **after** Bcast(M4) on P2:

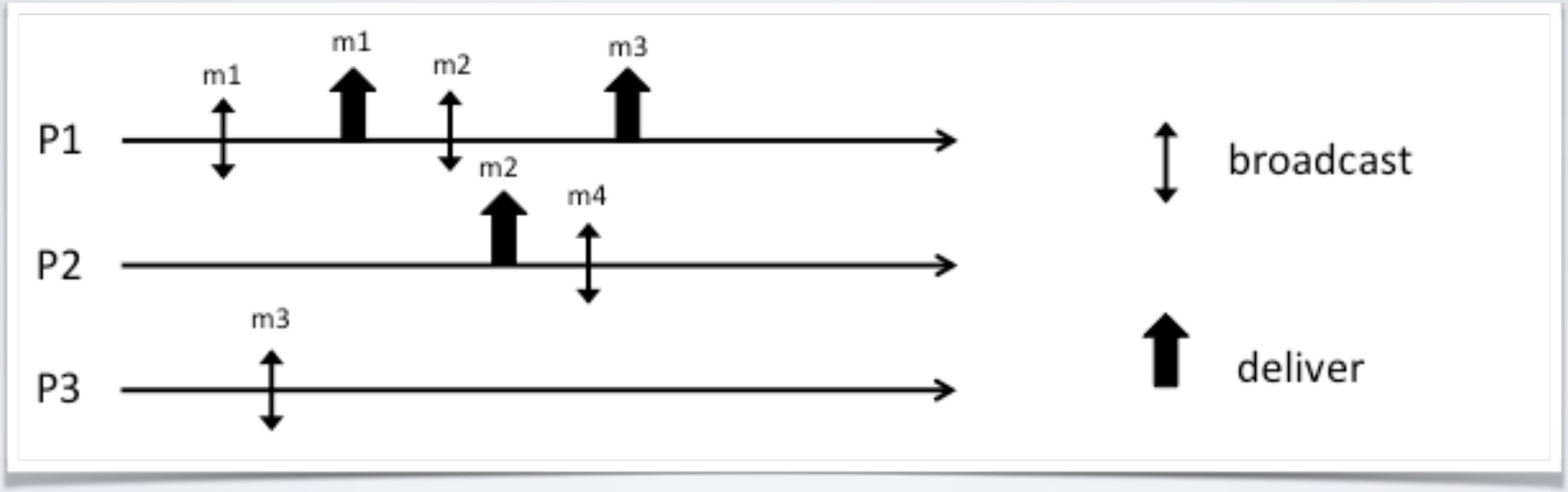
P1: M1, M2 , M3, M4 or M1, M2, M4, M3 or M1, M3, M2, M4

P2 : M1, M2, M3, M4 or M1, M2, M4, M3

P3: M1, M2 , M3, M4 or M1, M2, M4, M3 or M1, M3, M2, M4 or M3, M1, M2, M4

# EXERCISE

1. Provide all the possible sequences satisfying Causal Order




M1 -> M2 -> M4 concurrent M3 || M1 , M3 || M2 and M3 || M4

If P2 delivers M3 before Bcast(M4) then M3 -> M4. Sequences, supposing M3 delivered **before** Bcast(M4) on P2:

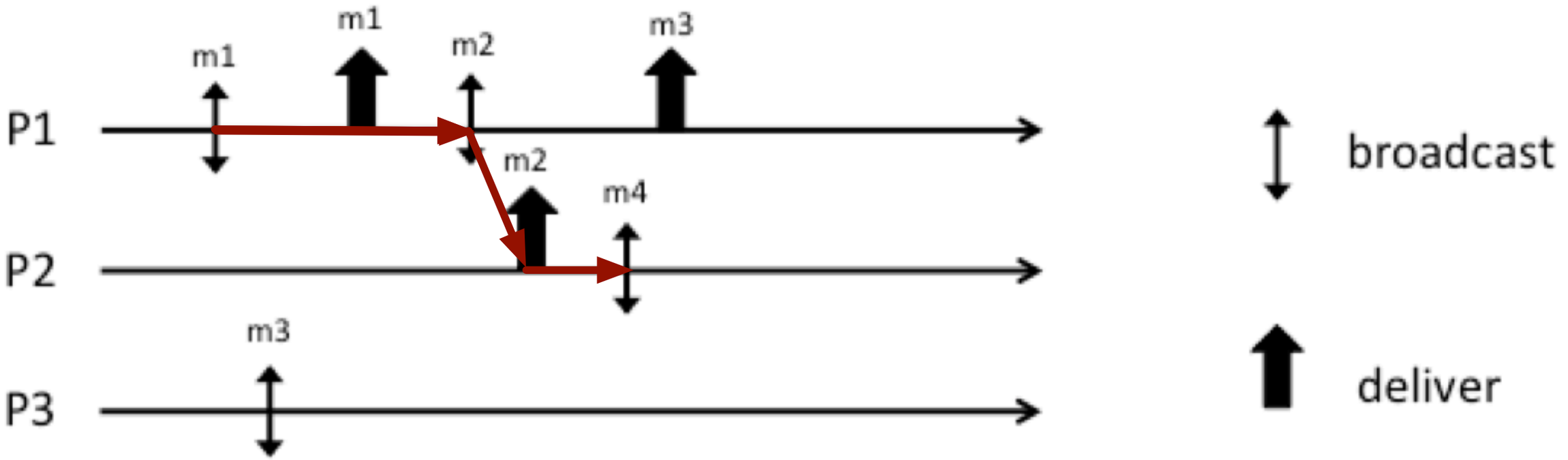
P1: M1, M2 , M3, M4 or M1, M3, M2, M4,

P2 : M1, M2, M3, M4 or M1, M2, M3, M4

P3: M1, M2 , M3, M4 or M1, M3, M2, M4 or M3, M1, M2, M4

# EXERCISE

1. Complete the execution in order to have a run satisfying FIFO order but not causal order




**FIFO: M1,M2**

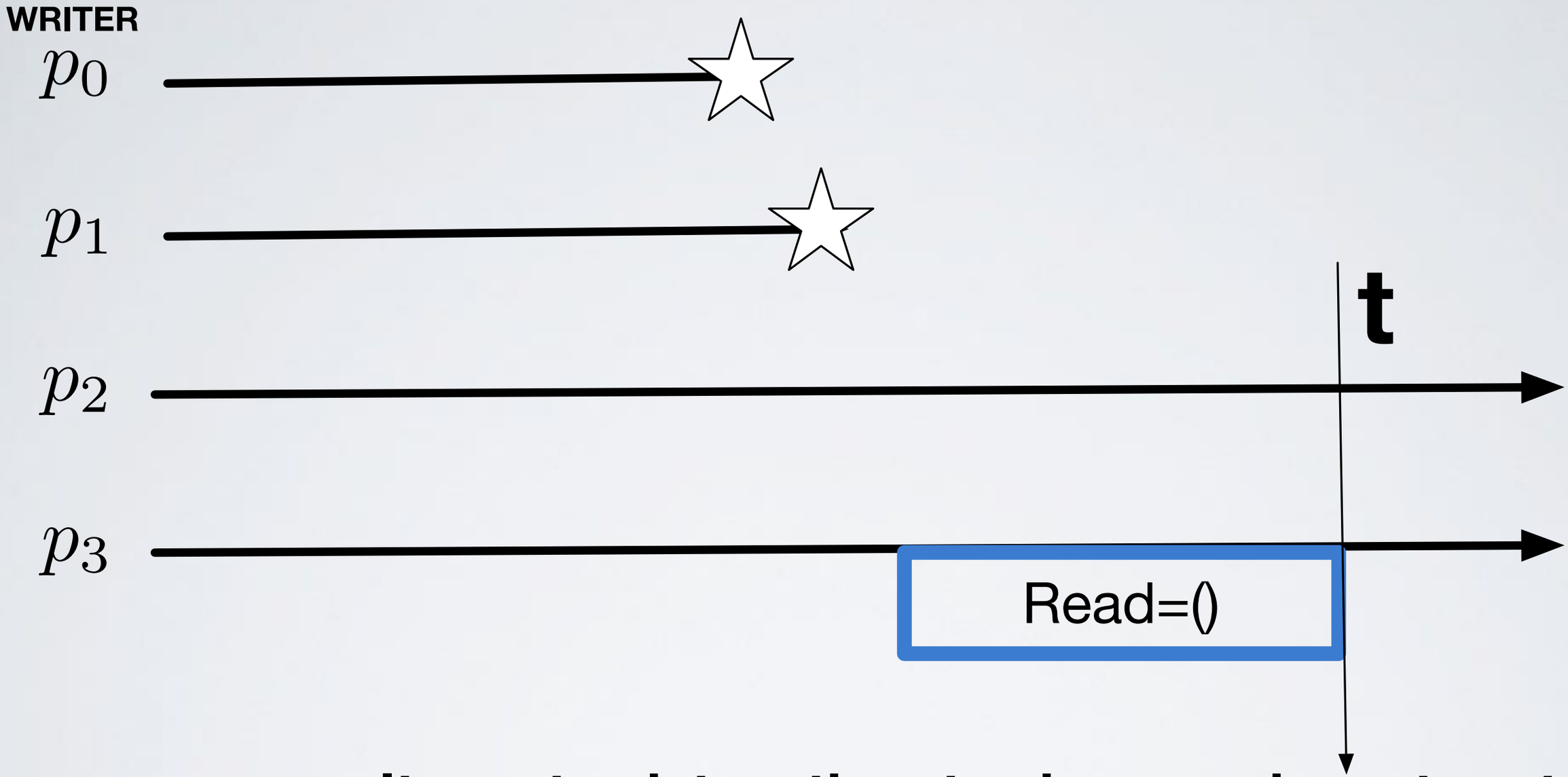
**P1: M1,M2,M4,M3 P2: M1,M2,M3,M4 P3: M4,M1,M2,M3**

# EXERCISE

**

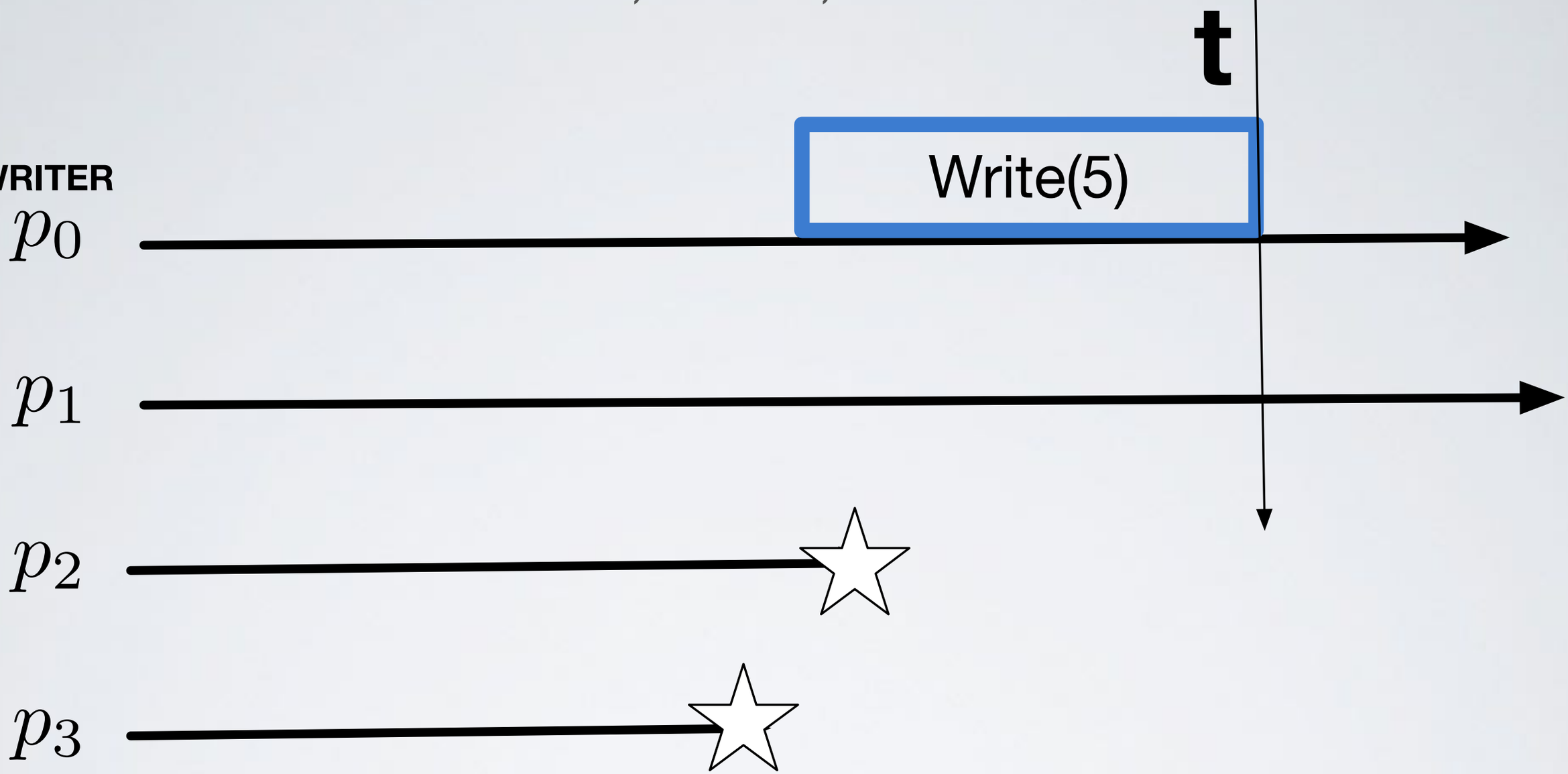
**Exercise 4.8:** *Does any implementation of a regular register require a majority of the correct processes in a fail-silent model with no failure detector? What if an eventually perfect failure detector (Module 2.8) is available?*

# EXECUTION 1, P0,P1 FAULTY



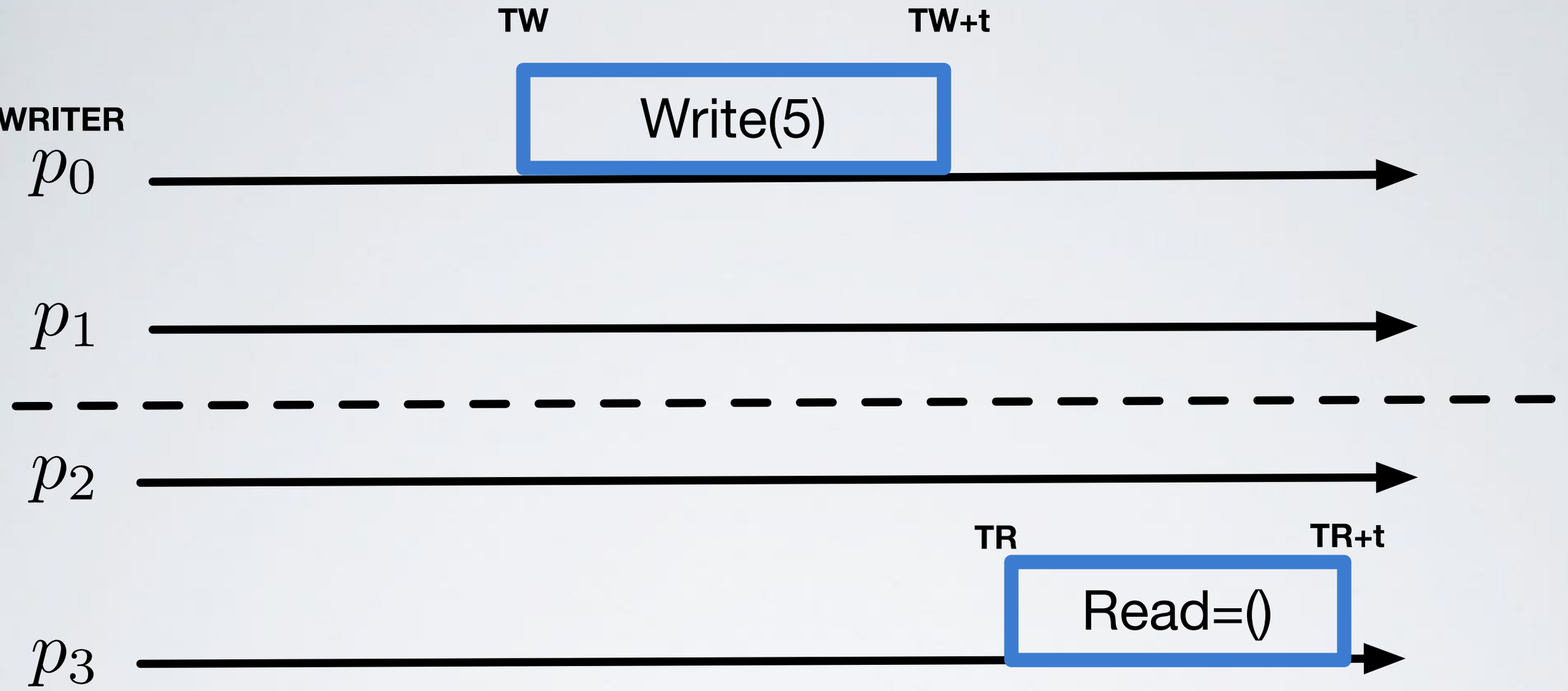

**It must exists a time t, when you have to return**

# EXECUTION 2, P2,P3 FAULTY




**It must exists a time t, when you have to return**

# EXECUTION 3, PARTITION




**PARTITION ARGUMENT**

**We SLOW Down messages from {p0,p1} to {p2,p3} and viceversa**

**Until time TR+t. For {p0,p1} this is eq. to EX2.**

**For {p2,p3} this is eq. to EX1.**