

Specifiche dei Registri

Corso di Sistemi distribuiti

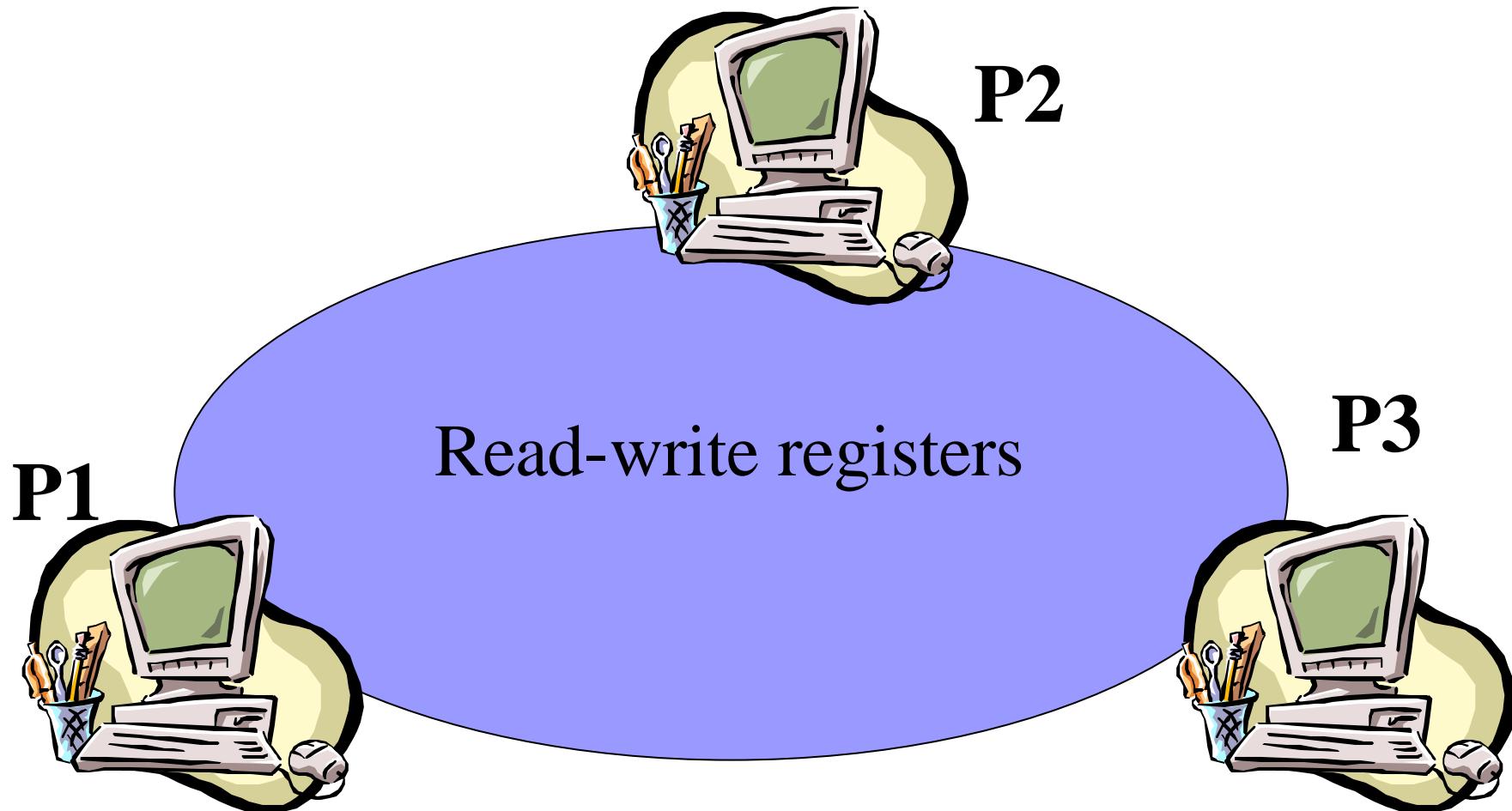
Laurea Specialistica in Ingegneria Informatica

Courtesy by Rachid Guearraoui (EPFL)

Overview of this lecture

- (1) ***Overview of a register***
- (2) ***Register specifications: safe, regular, and atomic***
- (3) ***Register executions: safe, regular, and atomic***
- (4) ***Register implementation***

The application model



Register: operations

- 1. Read(): returns the value of the register and does not modify the value of the register
 - 2. Write(x): updates the value of the register and returns a constant value *ok*
-
- *E.G. We consider here a register of integers*
 - *NB. Every written value is uniquely identified*

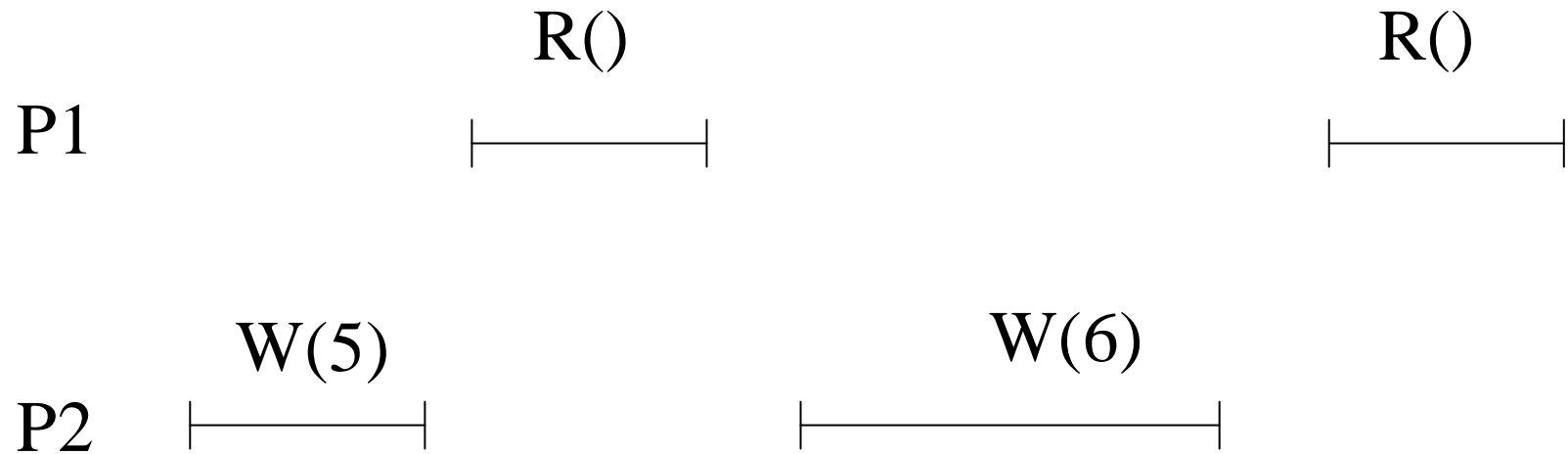
Register

- ☛ A register is a typical abstraction provided at the hardware level on top of a multiprocessor
- ☛ We aim at providing that abstraction at the **software** level on top of a network of geographically distant processes
- ☛ This abstraction is useful for various kinds of applications, e.g., collaborative work and distributed file sharing

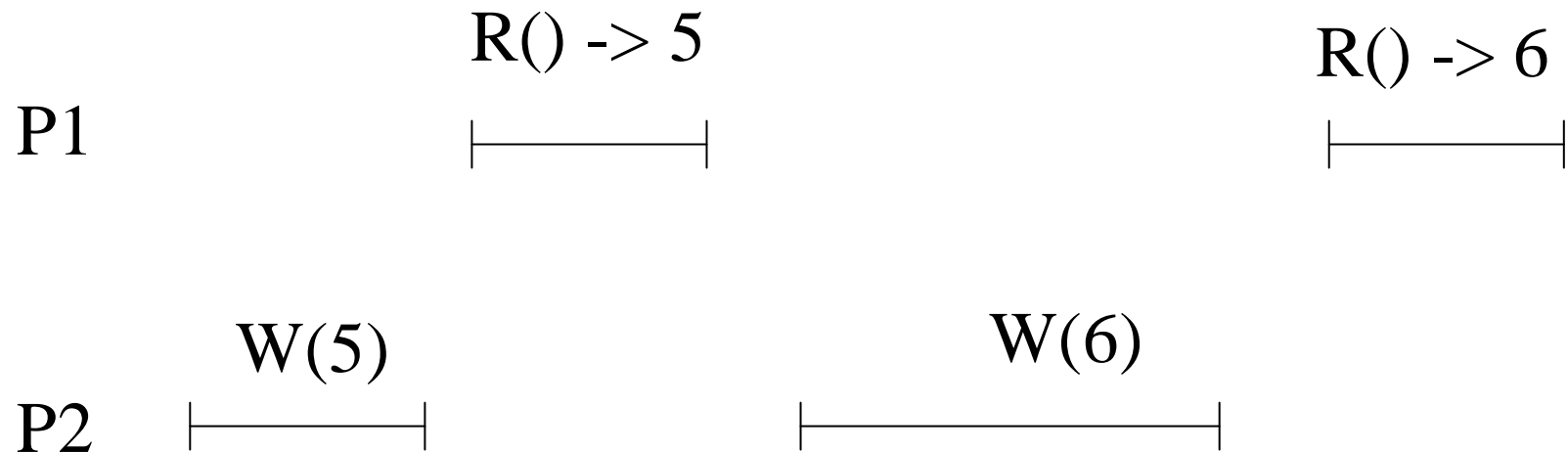
Register: specification

- The sequential specification of a register describes how the register is supposed to behave in the absence of concurrency and failures
- More precisely, the sequential specification of a register is the set of sequential histories involving the register such that the value returned by a *Read()* is the argument of the last *Write()*

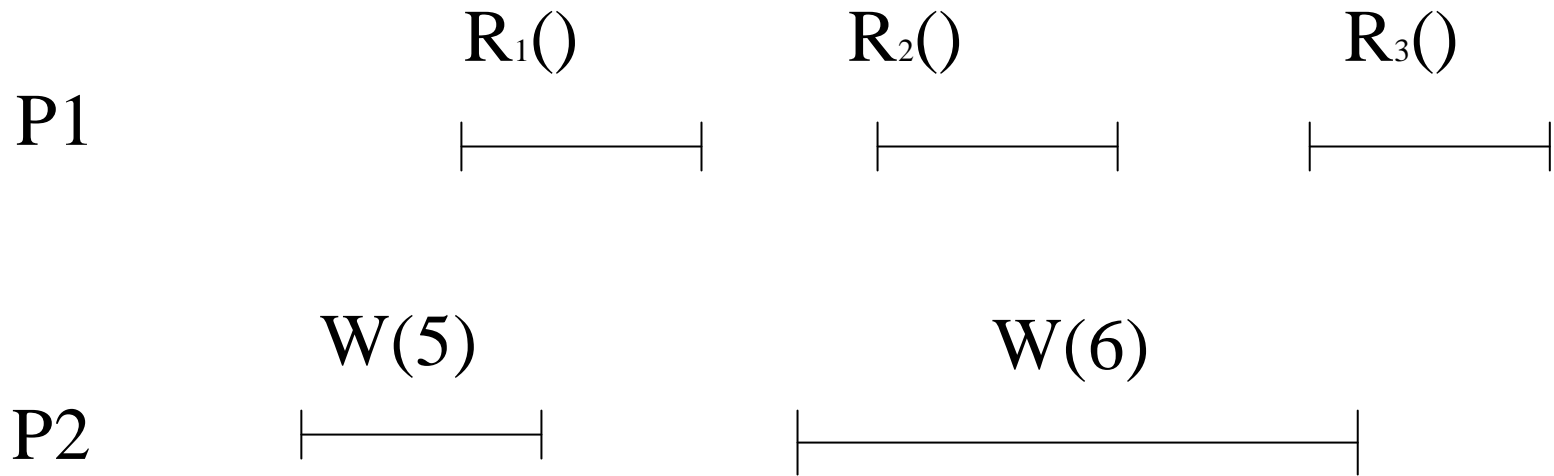
Sequential execution



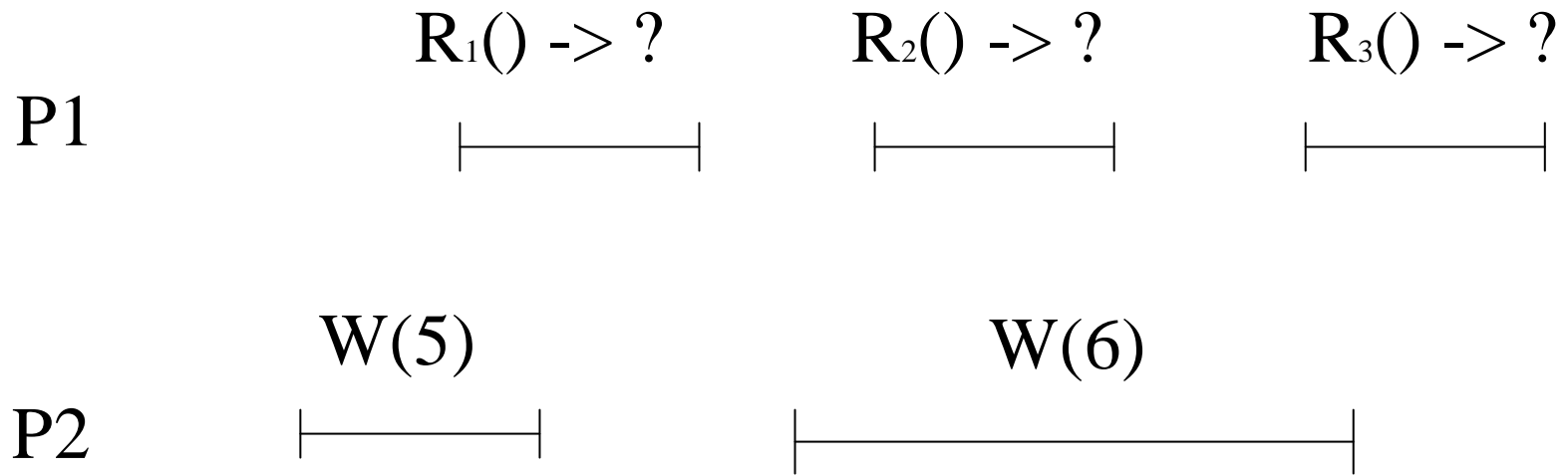
Sequential execution



Concurrent execution



Concurrent execution



Correctness

- ***Liveness***: something good must happen
- ***Safety***: nothing bad must happen

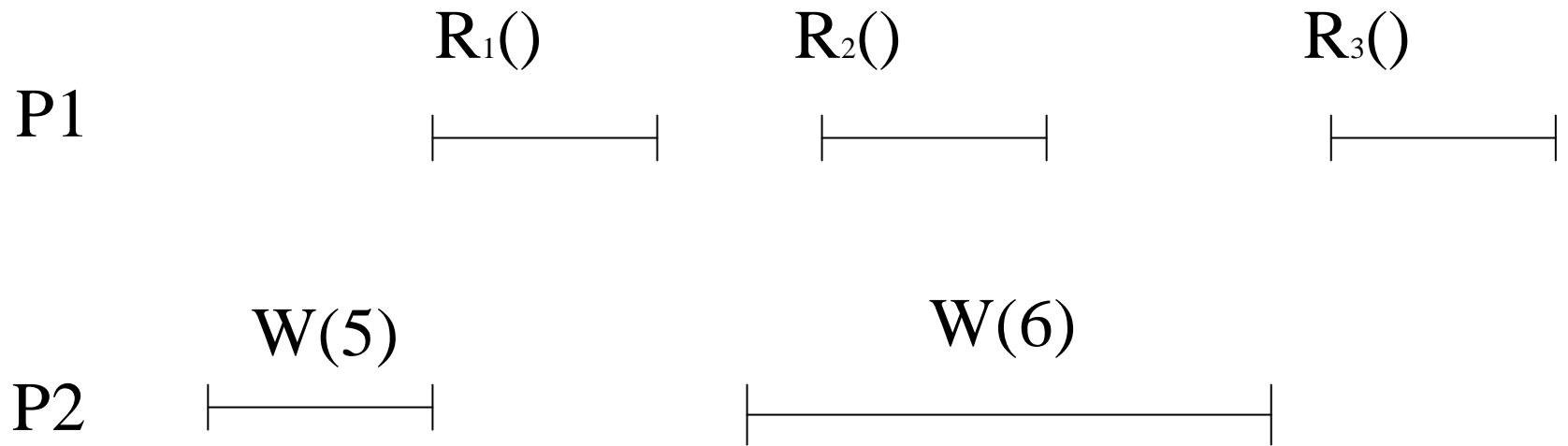
Liveness: wait-free

- If a process p_i requests an operation (Read() or Write()) , and p_i does not crash, then the operation eventually terminates, i.e., returns a reply
- I.E., the liveness of a process does not depend on other processes
- Any solution based on locking is hence excluded

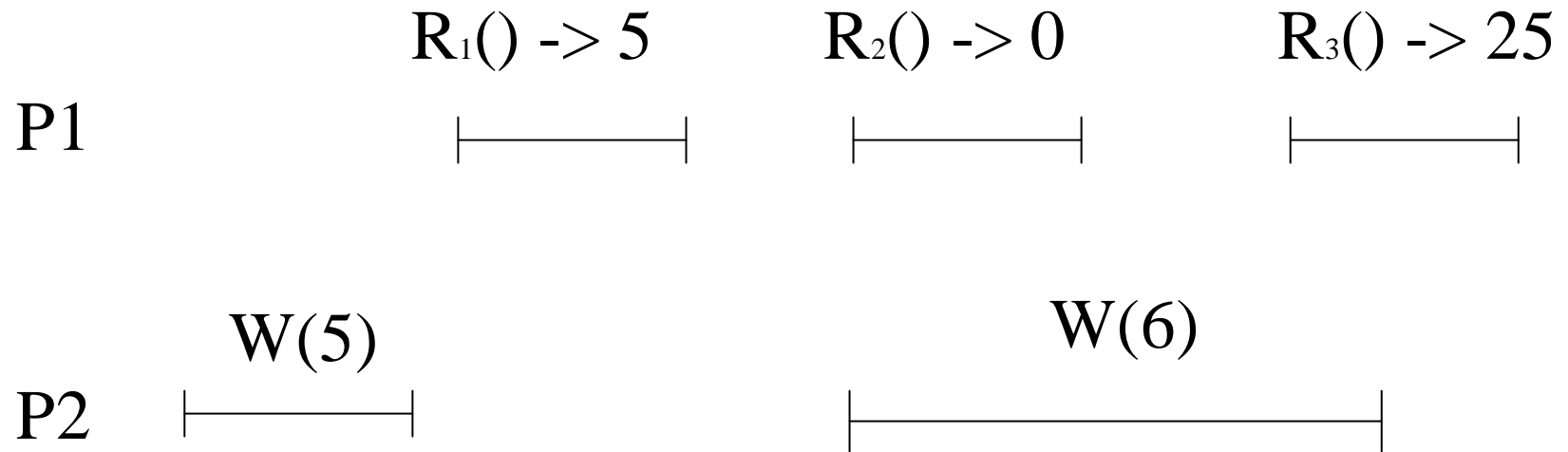
Safety: three levels

- Safe register: Read() returns the last value written if there is no concurrency or failure
- Regular register: safe register where Read() should return some value written
- Atomic register: regular register that provides the illusion of a sequential execution

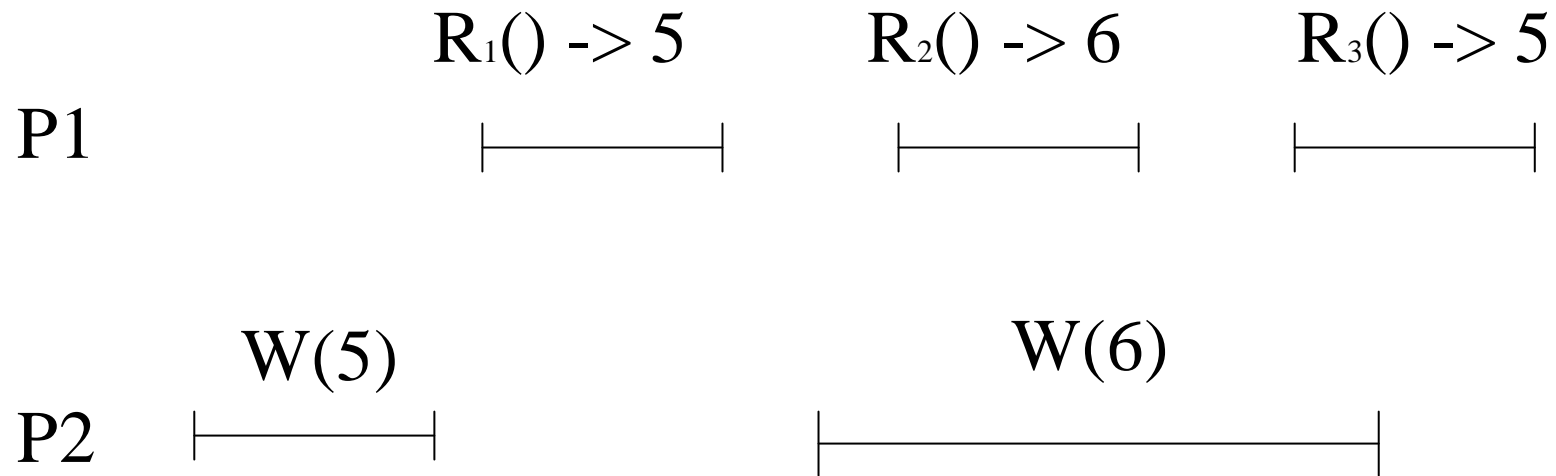
Execution



Results 1



Results 2



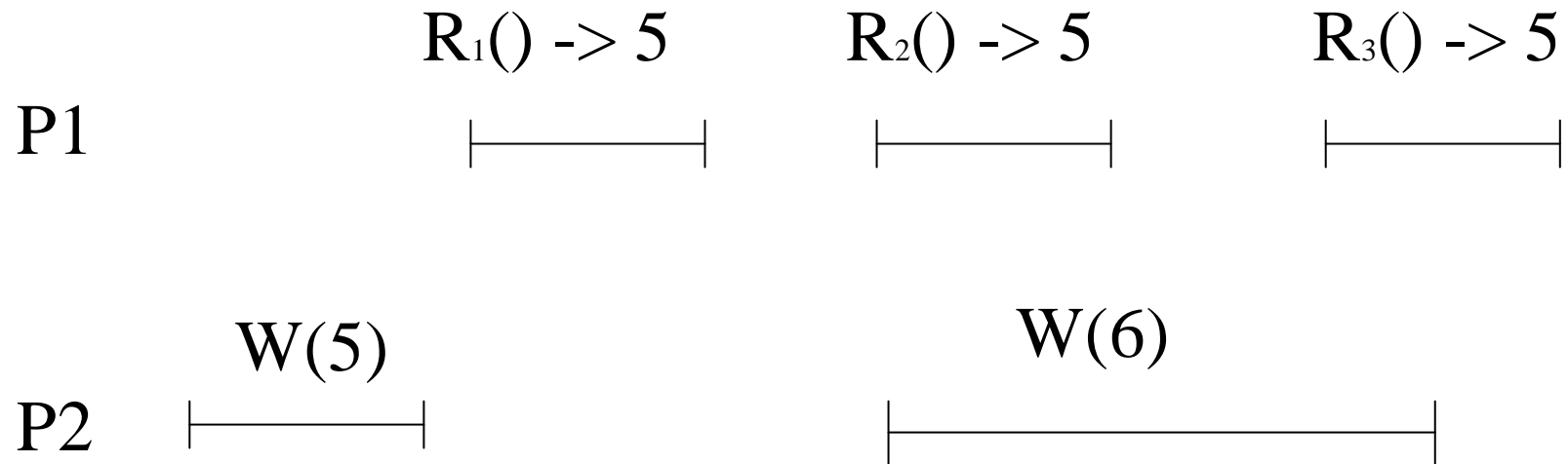
Results 3



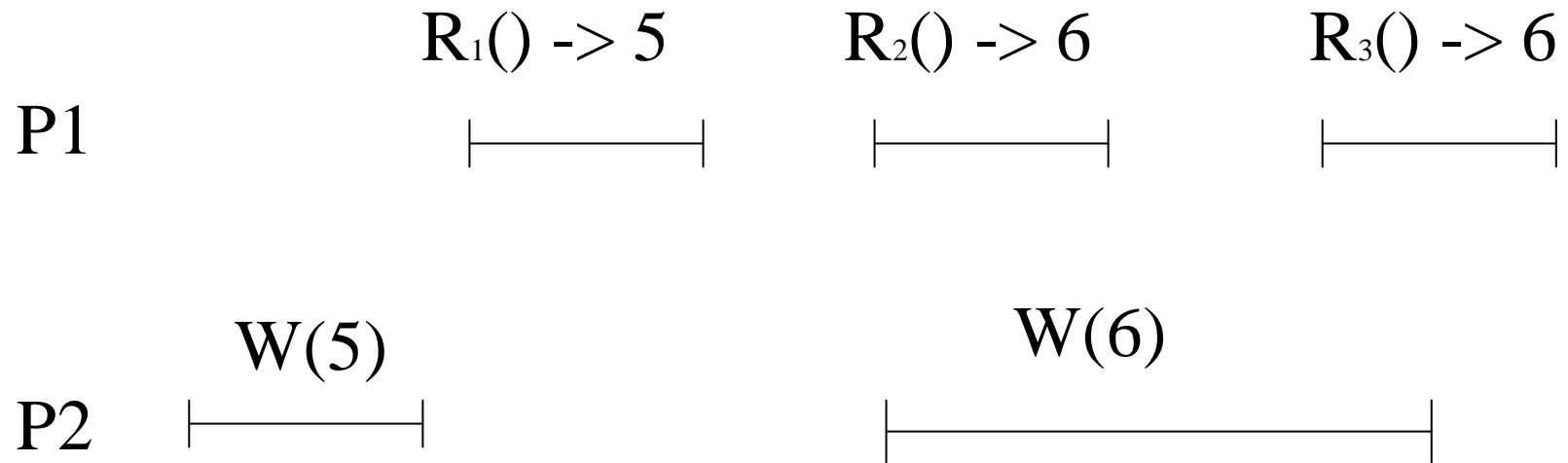
Results 4



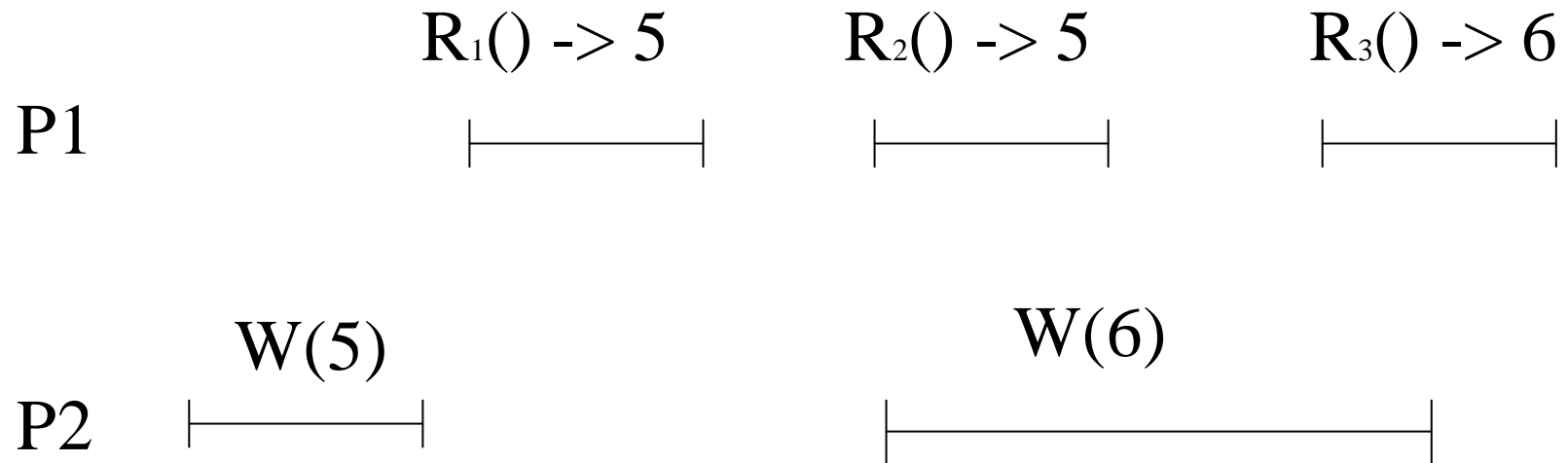
Results 5



Results 6



Results 7



Correctness

- Results 1: safe register
- Results 2; 3; 4: regular register
- Results 5; 6 and 7: atomic register

Safe register

- A safe register is like a sequential register: put in a concurrent context, it might behave however in an arbitrary manner, e.g., it can output any integer
- To show that a register is safe, we need to show that, if there is no concurrency or failure, a `Read()` returns the last `Write()`

Regular register

- A regular register is a safe register that behaves better in the face of concurrency and failure
- To show that a register is regular, we need to show that (a) it is safe, and (b) any value returned by a `Read()` must be an input parameter of some `Write()`

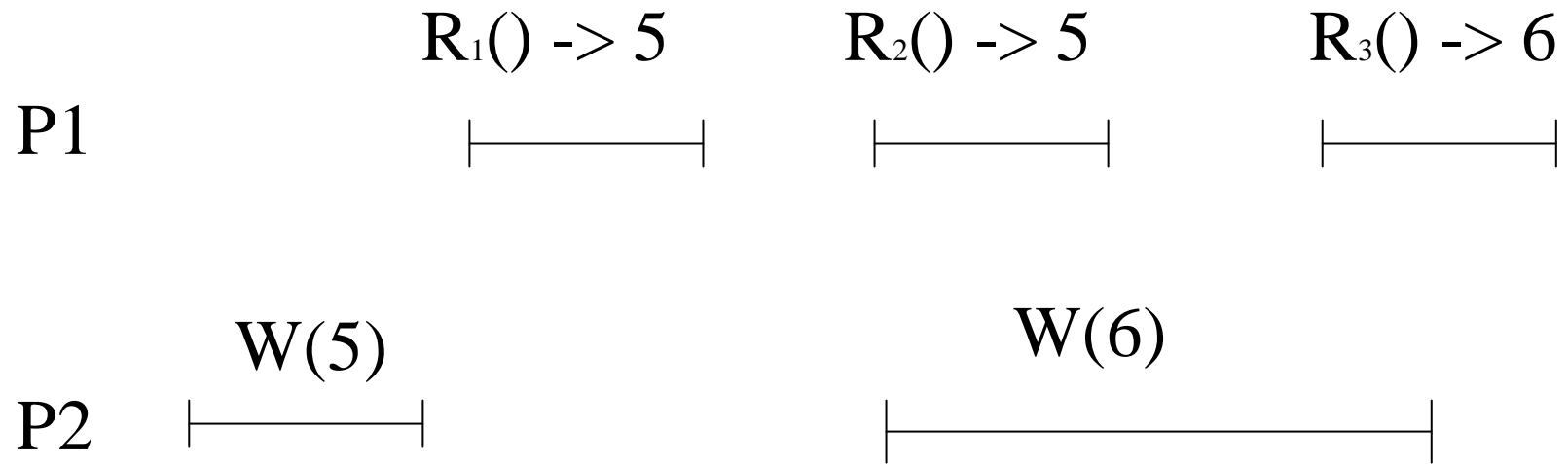
Atomic register

- To check the atomicity of a register, we typically check that it is regular, and the following (ordering) proposition holds
 - If a `Read()` returns a value written by a `Write()` w_1 , and a `Read()` invoked later returns a value written by a `Write()` w_2 , then w_2 does not precede w_1

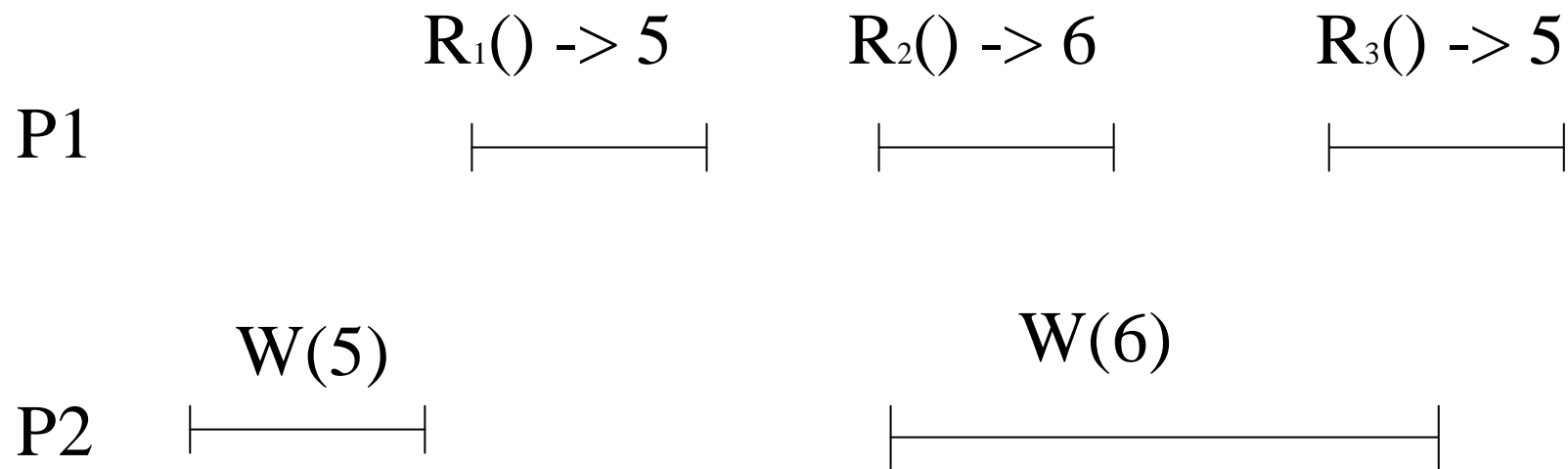
Regular vs Atomic

- A regular register is atomic when two successive Read() do not overlap a Write()
- The regular register might in this case allow the first Read() to obtain the new value and the second Read() to obtain the old value

Regular vs Atomic



A regular non-atomic register



Implementing a register

- From message passing to shared memory
- Implementing the register comes down to implementing `Read()` and `Write()` operations at every process

Implementing a register

- Before returning a Read() value, the process must communicate with other processes
- Before performing a Write(), i.e., returning the corresponding ok, the process must communicate with other processes

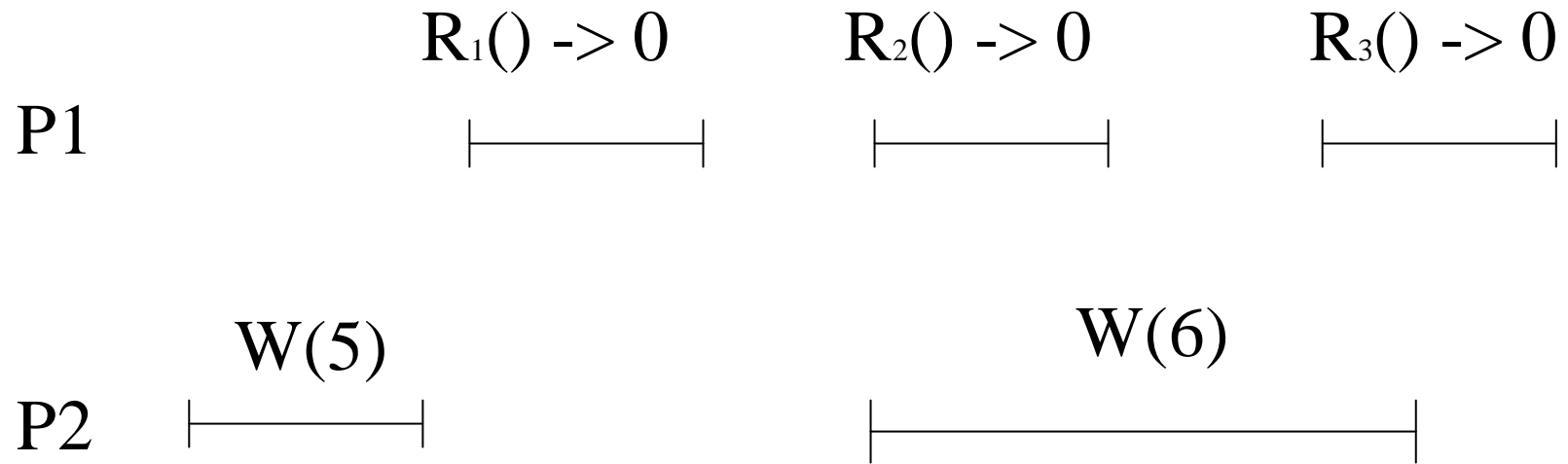
A bogus algorithm

- We assume that no process fails
- We assume that channels are perfect
- Every process holds a copy of the register value v

A bogus algorithm

- Read() at p_i
 - Return v_i
- Write(v) at p_i
 - $v_i := v$
 - Return ok
- The resulting register is live but not safe:
 - Even in a sequential execution, a Read() by p_j might not return the last written value, say by p_i

No safety



A simplistic algorithm

- We assume that no process fails
- We assume that channels are perfect
- Intuition: one process, say p_1 , holds the value of the register

A simplistic algorithm

- Read() at p_i

- send $[R,i]$ to p_1
- when receive $[v]$
- Return v

- Write(v) at p_i

- send $[W,v,i]$ to p_1
- when receive $[ok]$
- Return ok

- At p_1 :

T1:

when receive $[R,i]$
Send $[v_1]$ to p_i

T2:

when receive $[W,v,i]$
 $v_1 := v$
send $[ok]$ to p_i

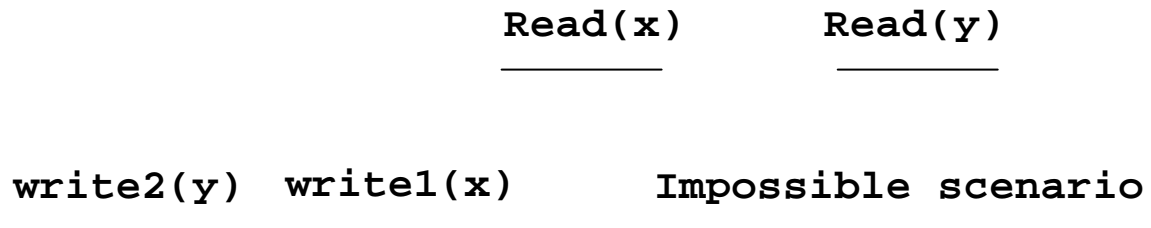
Correctness (liveness)

- By the assumption that
 - (a) no process fails,
 - (b) channels are reliable

no wait statement blocks forever, and hence every invocation eventually terminates

Correctness (safety)

- (a) If there is no concurrency or failure, a Read() returns the last value written
- (b) A Read() must return some value written
- (c) If a Read() returns a value written by a given Write1(), and another Read() that starts later returns a value written by a different Write2(), then Write1() cannot start after Write2() terminates



Correctness (safety – 1)

- (a) If there is no concurrency or failure, a `Read()` returns the last value written
 - Assume a `Write(x)` terminates and no other `Write()` is invoked. The value of the register is hence x at p_1 . Any subsequent `Read()` invocation by some process p_j returns the value of p_1 , i.e., x , which is the last written value

Correctness (safety – 2)

- (b) A Read() returns some value written
 - Let x be the value returned by a Read(): by the properties of the channels, x is the value of the register at $p1$. This value does necessarily come from a Write().

Correctness (safety – 3)

- (c) If a Read() returns a value x written by a given Write1(), and another Read() that starts later returns a value y written by a different Write2(), then the Write1() cannot start after Write2() terminates.
 - Assume Write(x) starts after Write(y) terminates. Since the value is stored solely at $p1$, no Read() can return y after a Read() returned x

What if?

- Processes might crash?
- If p1 crashes, then the register is not live (wait-free)
- If p1 is always up, then the register is atomic and wait-free

A simple algorithm

- ☛ We assume that
 - ☛ any number of processes can fail by crashing (no recovery)
 - ☛ channels are perfect
 - ☛ failure detection is ***perfect***

A simple algorithm

- We implement a **regular** [N,N] register
 - every process has a local copy of the register value
 - every process reads **locally** and writes **globally**, i.e., at all (non-crashed) processes

A simple algorithm

Write(v) at p_i

- send $[W, v]$ to all
- for every p_j , wait until either:
 - received $[ack]$ or
 - suspected $[p_j]$
- Return ok

At p_i :

when receive $[W, v]$
from p_j

$v_i := v$

send $[ack]$ to p_j

Read() at p_i

- Return v_i

Correctness (liveness)

- ✓ A read() is local and eventually returns
- ✓ A write() eventually returns by the
 - (a) the strong completeness property of the failure detector, and
 - (b) the reliability of the channels

Correctness (safety – 1)

- (a) In the absence of concurrency, a `Read()` returns the last value written
 - Assume a `Write(x)` terminates and no other `Write()` is invoked. By the accuracy property of the failure detector, the value of the register at all processes that did not crash is x . Any subsequent `Read()` invocation by some process p_j returns the value of p_j , i.e., x , which is the last written value

Correctness (safety – 2)

- (b) A Read() returns some value written
 - Let x be the value returned by a Read(): by the properties of the channels, x is the value of the register at some process. This value does necessarily come from a Write().

What about ordering? (i.e., atomicity)

