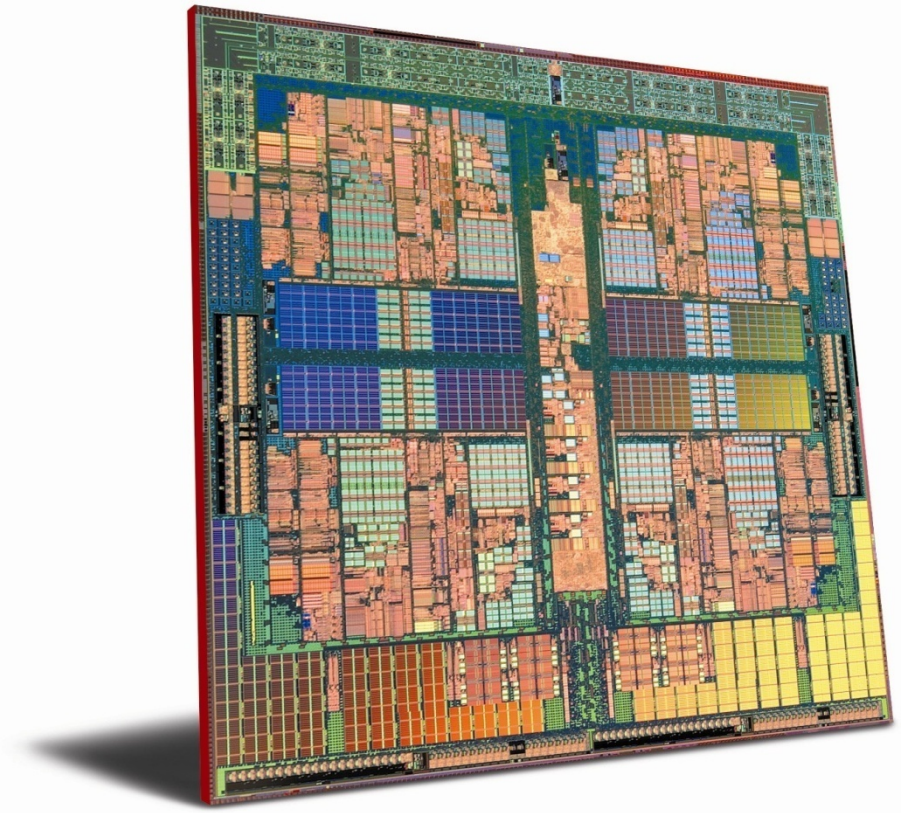
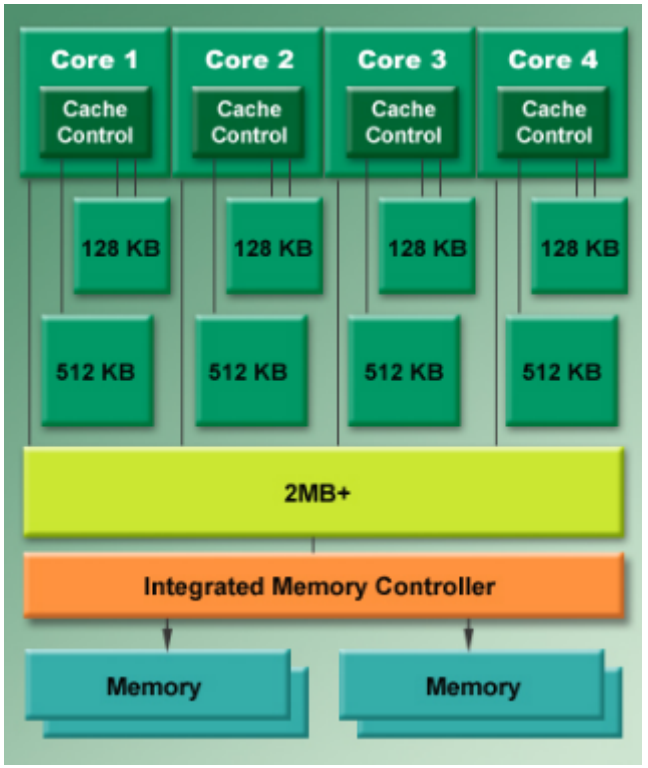
Wait-free Computing

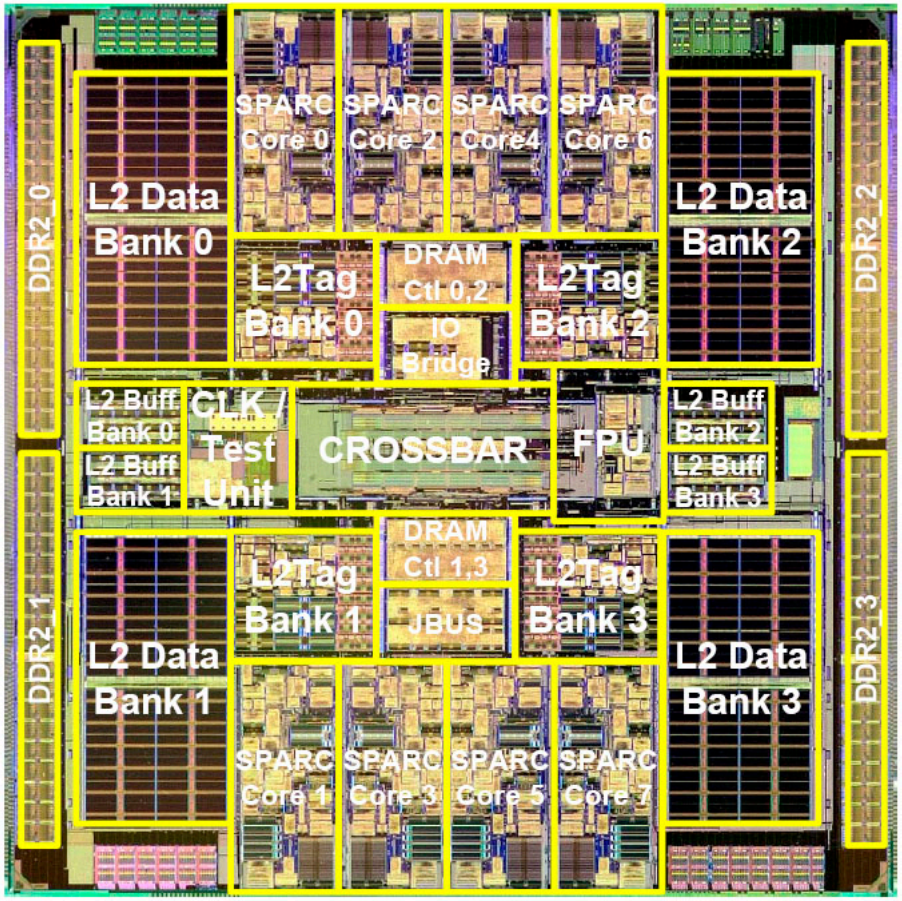
Prof R. Guerraoui
Distributed Programming Laboratory

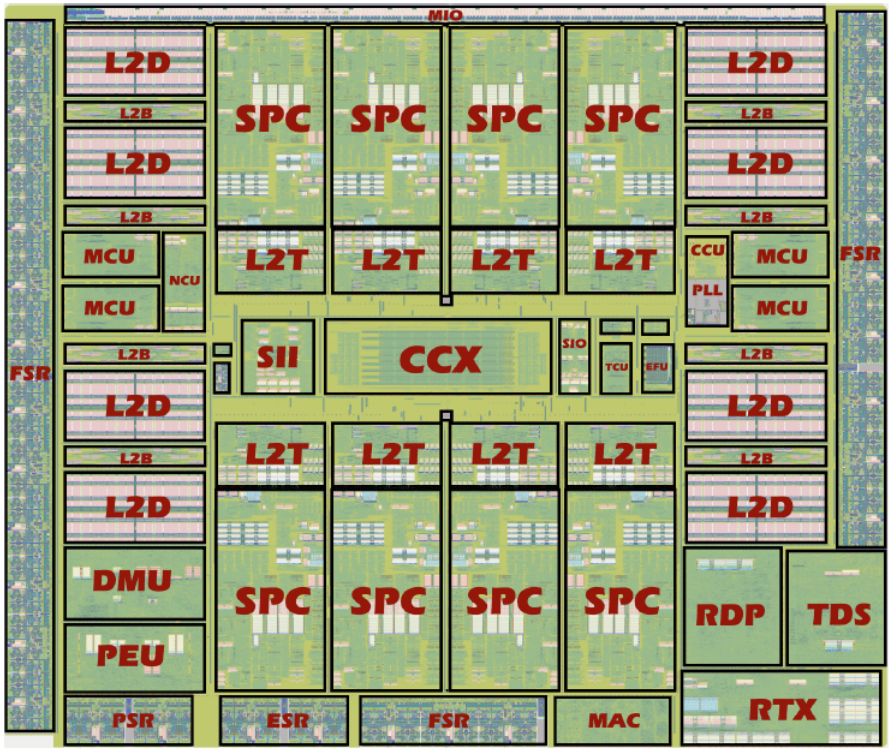
AMD Opteron (4 cores)





SUN's Niagara CPU2 (8 cores)



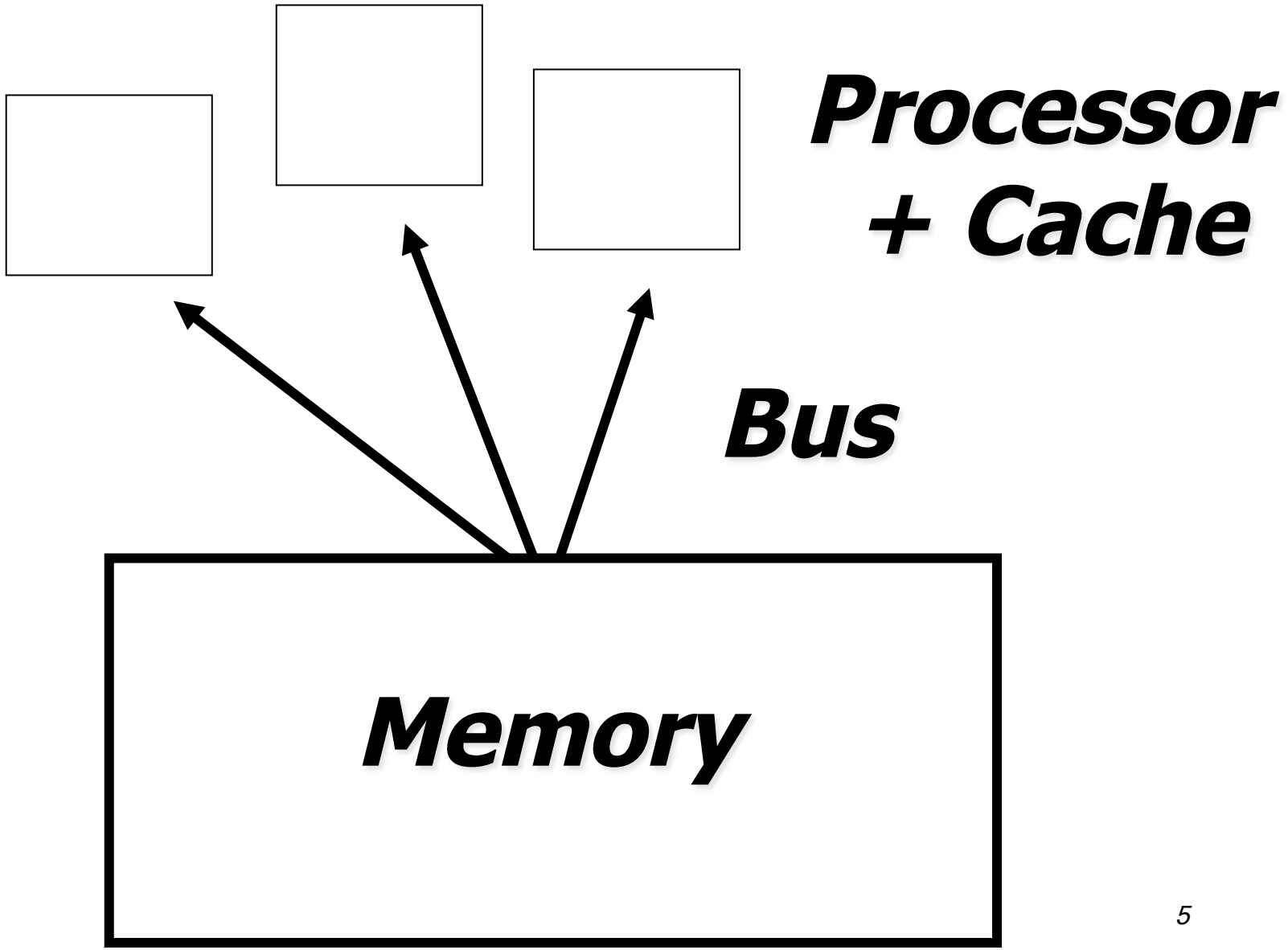


CCX – Crosssbar
CCU – Clock control
DMU/PEU – PCI Express
EFU – Efuse for redundancy
ESR – Ethernet SERDES
FSR – FBD SERDES
L2B – L2 write-back buffers
L2D – L2 data arrays
L2T – L2 tag arrays
MCU – Memory controller
MIO – Miscellaneous I/O
PSR – PCI Express SERDES
RDP/TDS/RTX/MAC – Ethernet
SII/SIO – I/O data path to and from memory
SPC – SPARC cores
TCU – Test and control unit

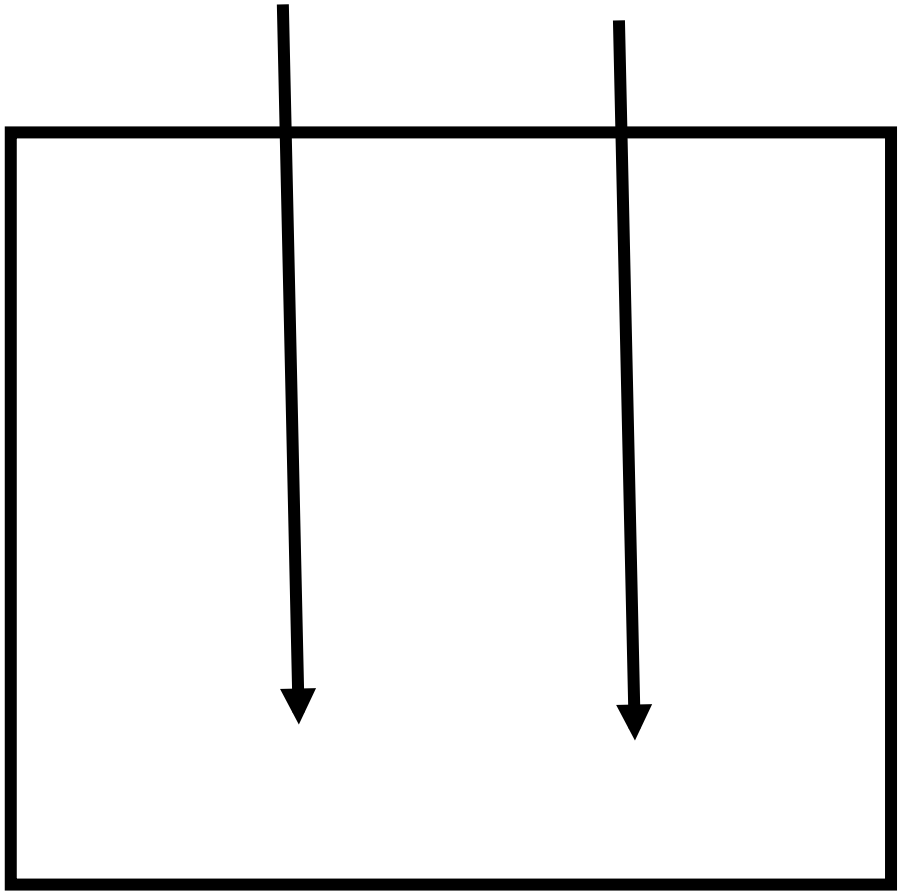
Principles of an architecture

- Two fundamental components that ***fall apart***: ***processors*** and ***memory***

Simple view



Concurrent processes



Shared object

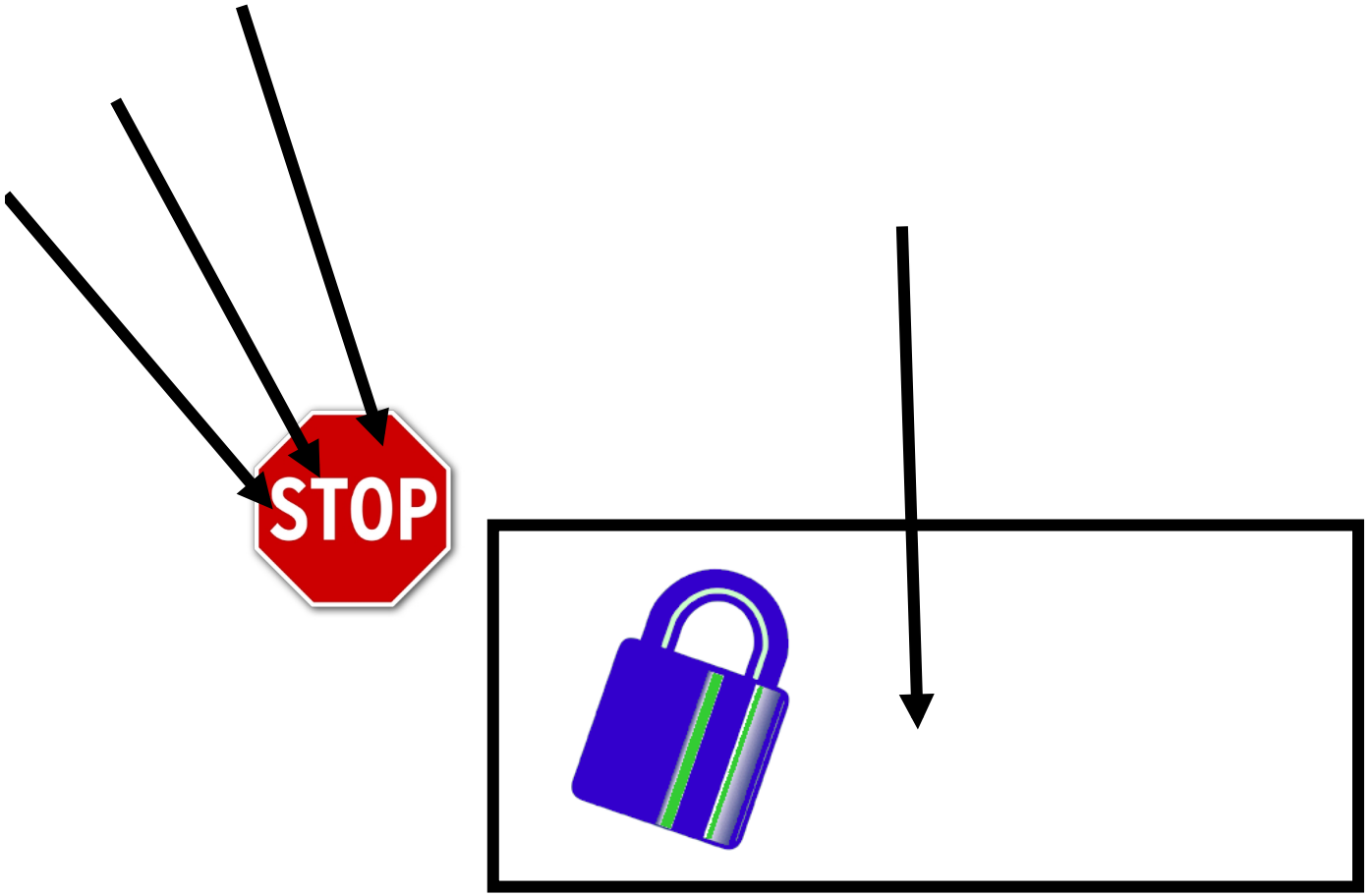
Counter

```
public class Counter

private int c = 0;

public long getAndIncrement()
{
return c++;
}
```

Locking (mutual exclusion)



Implicit use of a lock

```
public class SynchronizedCounter {
    private int c = 0;
    public synchronized void increment() {
        c++;
    }
    public synchronized void getAndincrement()
{
        return c++;
    }
    public synchronized int value() {
        return c;
    }
}
```

Locking with compare&swap()

- A ***Compare&Swap*** object maintains a value x, init to ⊥, and y;

- It provides one operation: ***c&s(old,new);***

  ✓Sequential spec:

  - c&s(old,new)

    {y := x; if x = old then x := new; return(y)}

Locking with compare&swap()

```
lock() {
repeat until
unlocked = this.c&s(unlocked,locked)
}

unlock() {
          this.c&s(locked,unlocked)
     }
```

Locking with test&set()

- A ***Test&Set*** object maintains binary values x, init to 0, and y;

- It provides one operation: ***t&s()***

  ✓Sequential spec:

  ✓ t&s() {y := x; x: = 1; return(y);}

Locking with test&set()

```
lock() {
repeat until (0 = this.t&s());
}

unlock() {
          this.setState(0);
     }
```

Locking with test&set()

```
lock() {
while (true)
 {
 repeat until (0 = this.getState());
 if 0 = (this.t&s()) return(true);
 }
}

unlock() {
          this.setState(0);
     }
```

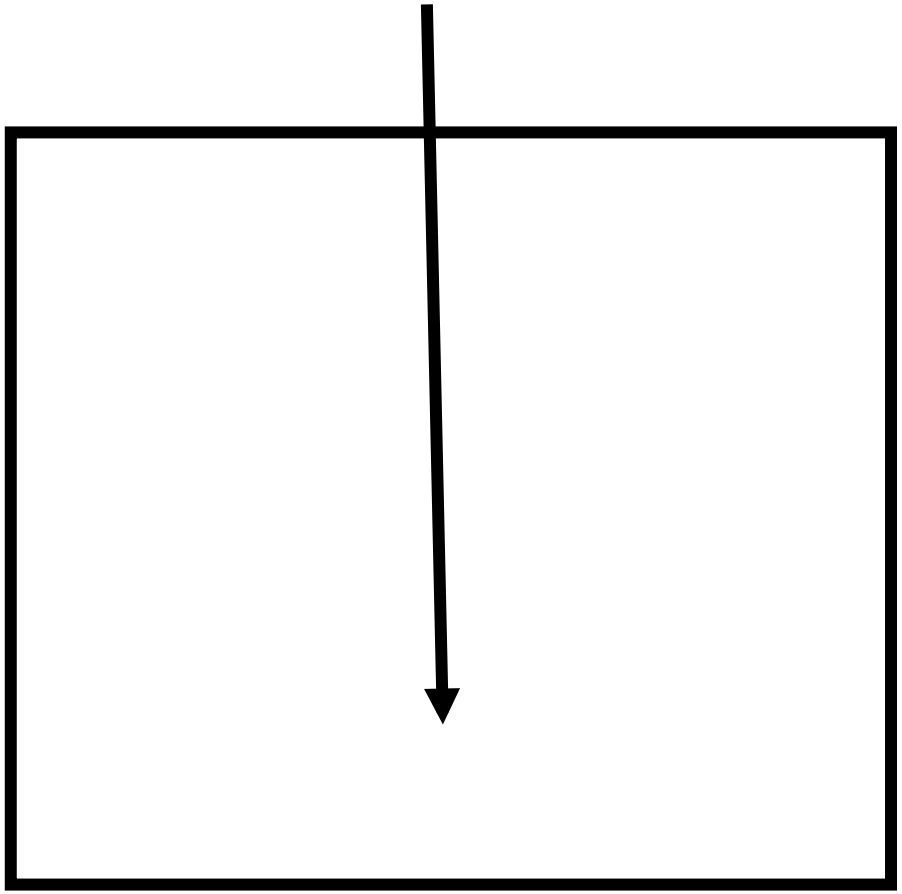
Explicit use of a lock

```
Lock l = ...;
     l.lock();
     try {
// access the resource protected by this lock
     } finally {
         l.unlock();
     }
```

Locking (mutual exclusion)

- ***Difficult:*** 50% of the bugs reported in Java come from the mis-use of « synchronized »
- ***Fragile:*** a process holding a lock prevents all others from progressing
- ***Slow:*** the act of locking itself impacts performance

Locked object



One process at a time

Processes are asynchronous

- *Page faults*
- *Pre-emptions*
- *Failures*
- *Cache misses, …*

Processes are asynchronous

- A cache miss can delay a process by ten instructions
- A page fault by few millions
- An os preemption by hundreds of millions…

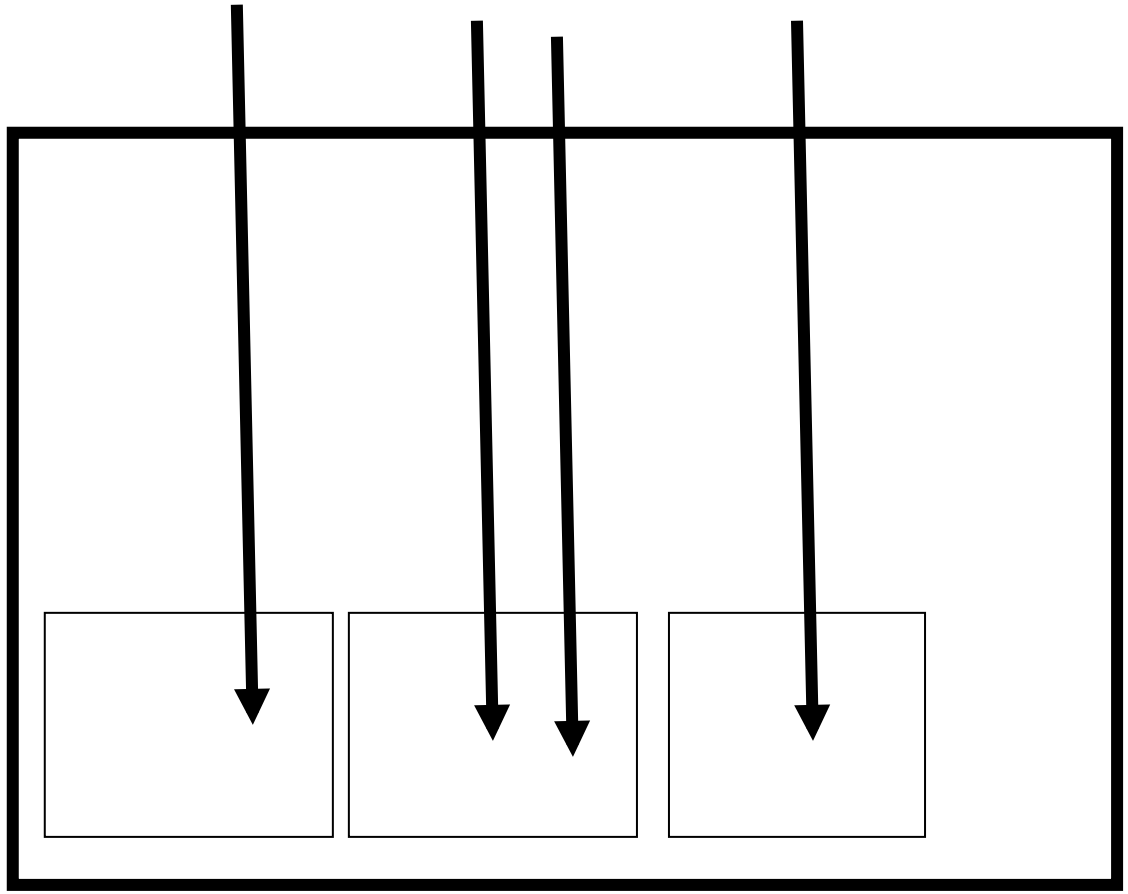
Coarse grained locks => slow

Fine grained locks => errors

Processes are asynchronous

- ***Page faults, pre-emptions, failures, cache misses, …***
- A process can be delayed by millions of instructions …

Alternative to locking?



Wait-free atomic objects

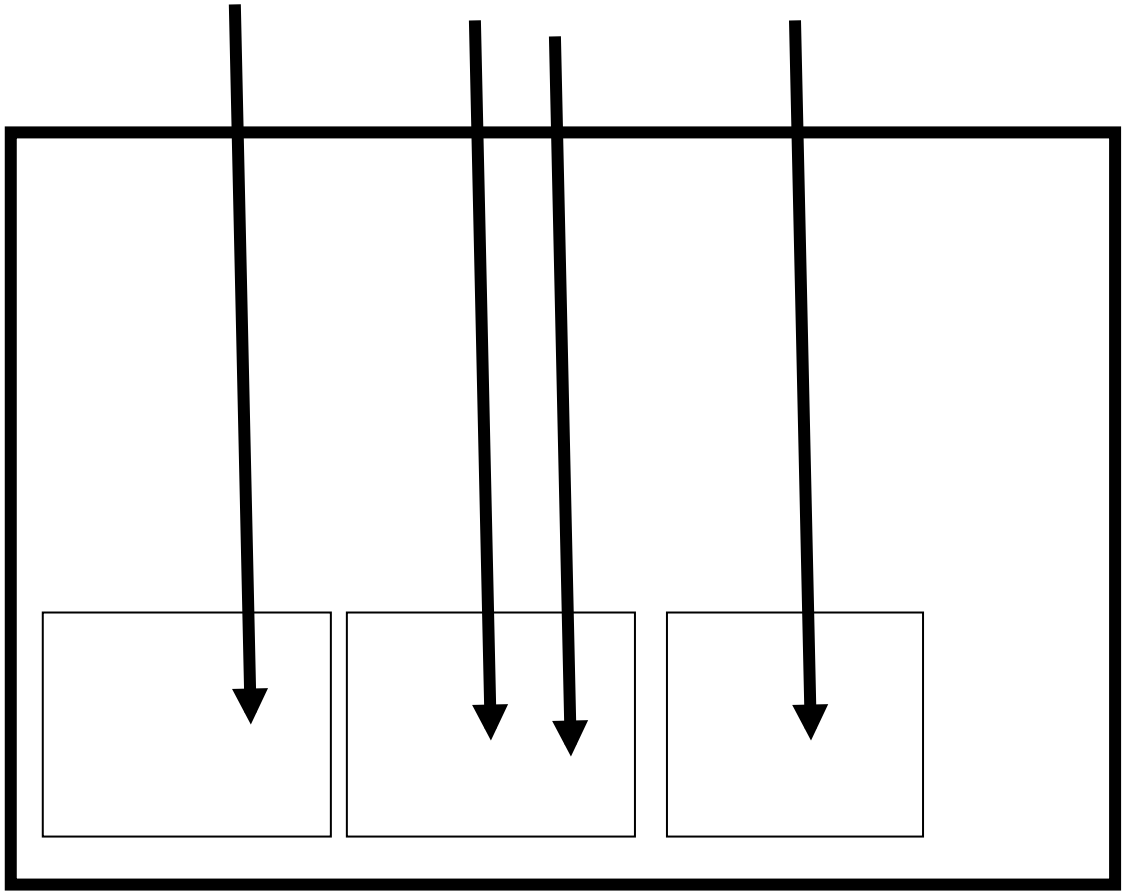
- ***Wait-freedom:*** every process that invokes an operation eventually returns from the invocation (robust … unlike locking)

- ***Atomicity:*** every operation appears to execute instantaneously (as if the object was locked…)

In short

The fundamental question is how to
wait-free implement high-level
atomic objects out of primitive base objects

Concurrent processes



Shared object

Processes

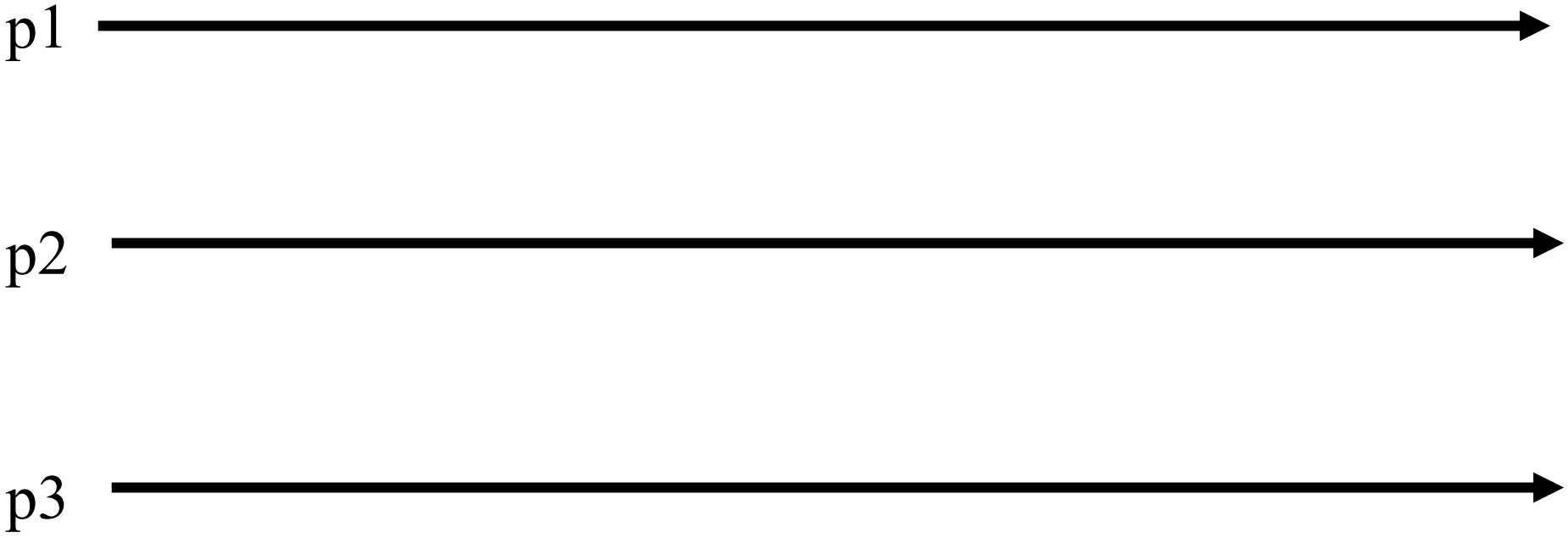
- We assume a finite set of processes
- Processes are denoted by p1,..pN or p, q, r
- Processes have unique identities and know each other (unless explicitly stated otherwise)

Processes

- Processes are ***sequential*** units of computations

- Unless explicitly stated otherwise, we make no assumption on process (relative) speeds

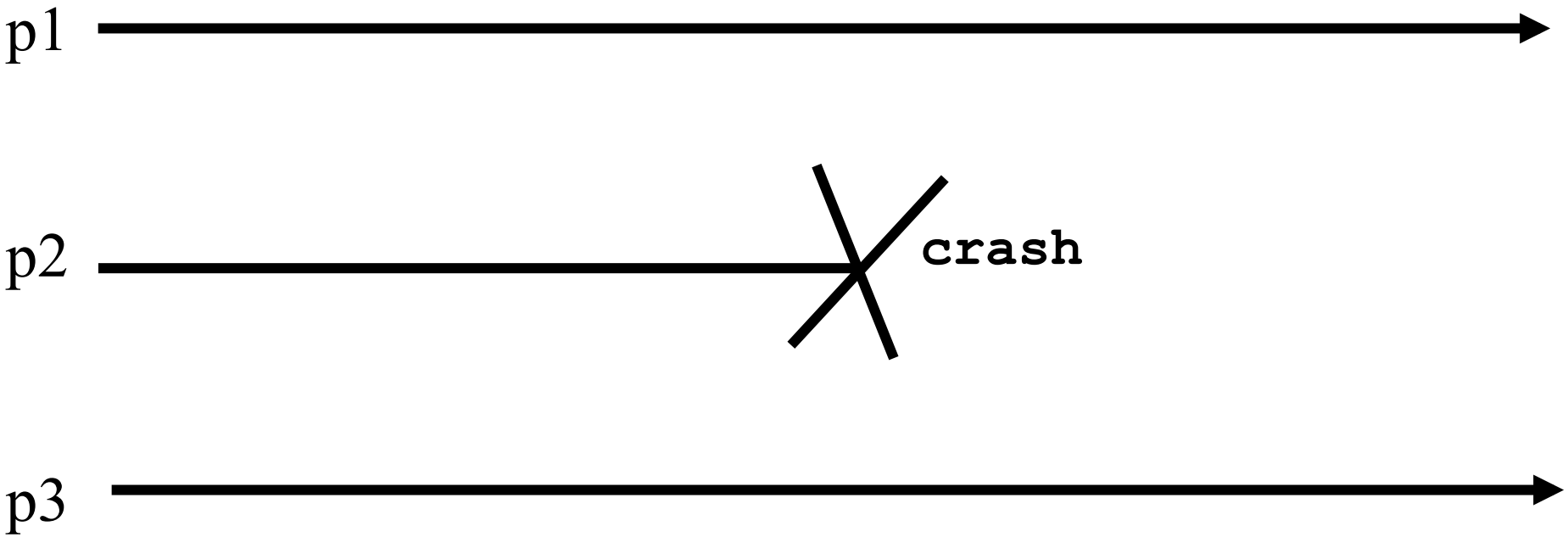
Processes



Processes

- A process either executes the algorithm assigned to it or crashes
- A process that crashes does not recover (in the context of the considered computation)
- A process that does not crash in a given execution (computation or run) is called correct (in that execution)

Processes

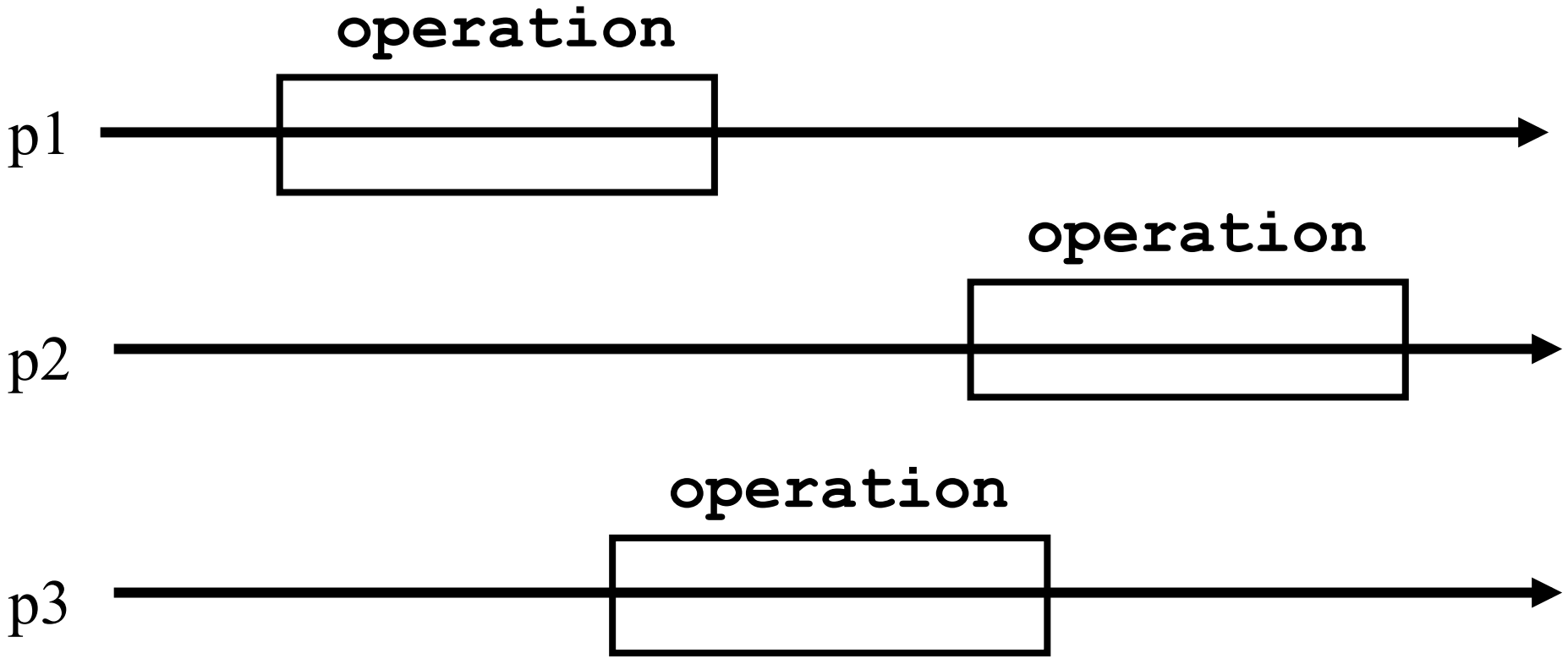


On objects and processes

- Processes execute local computation or access shared objects through their ***operations***

- Every operation is expected to return a reply

Processes

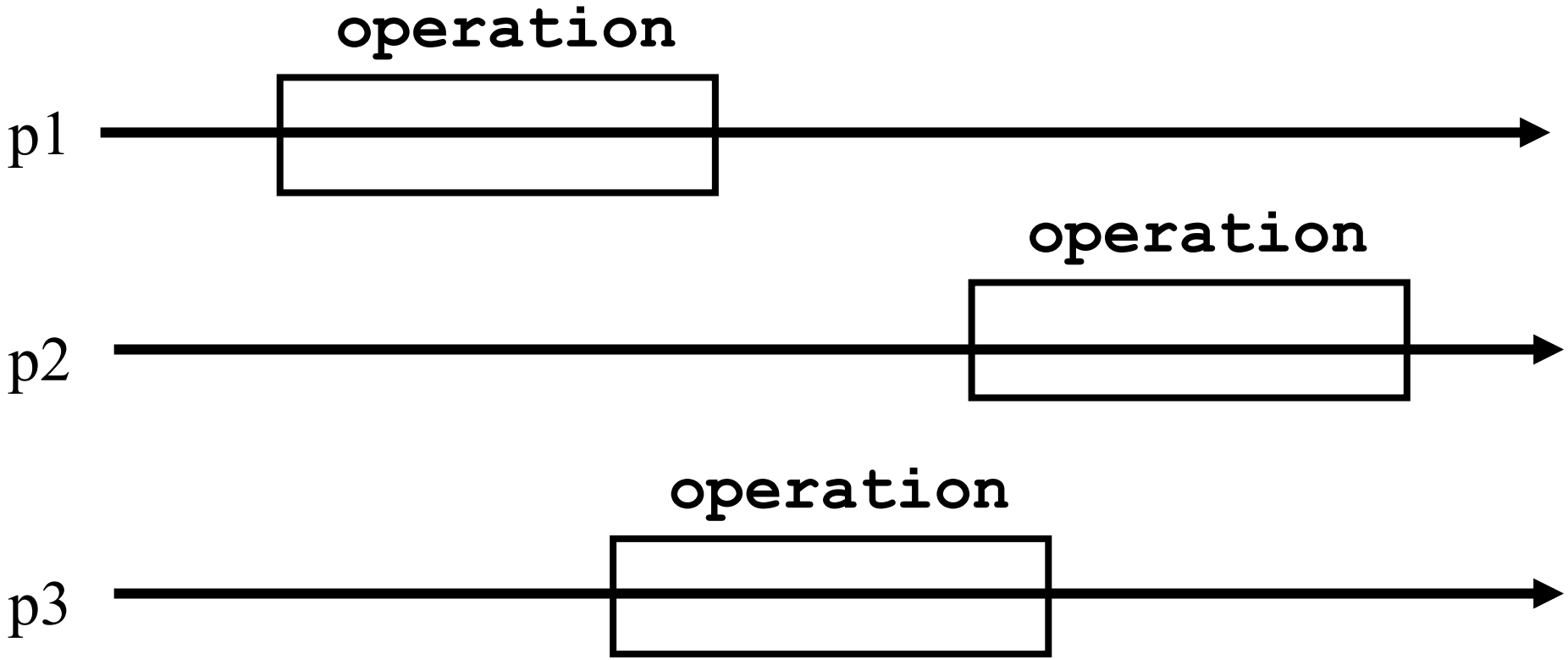


On objects and processes

- ***Sequentiality*** means here that, after invoking an operation op1 on some object O1, a process does not invoke a new operation (on the same or on some other object) until it receives the reply for op1

- ***Remark***. Sometimes we talk about operations when we should be talking about operation invocations

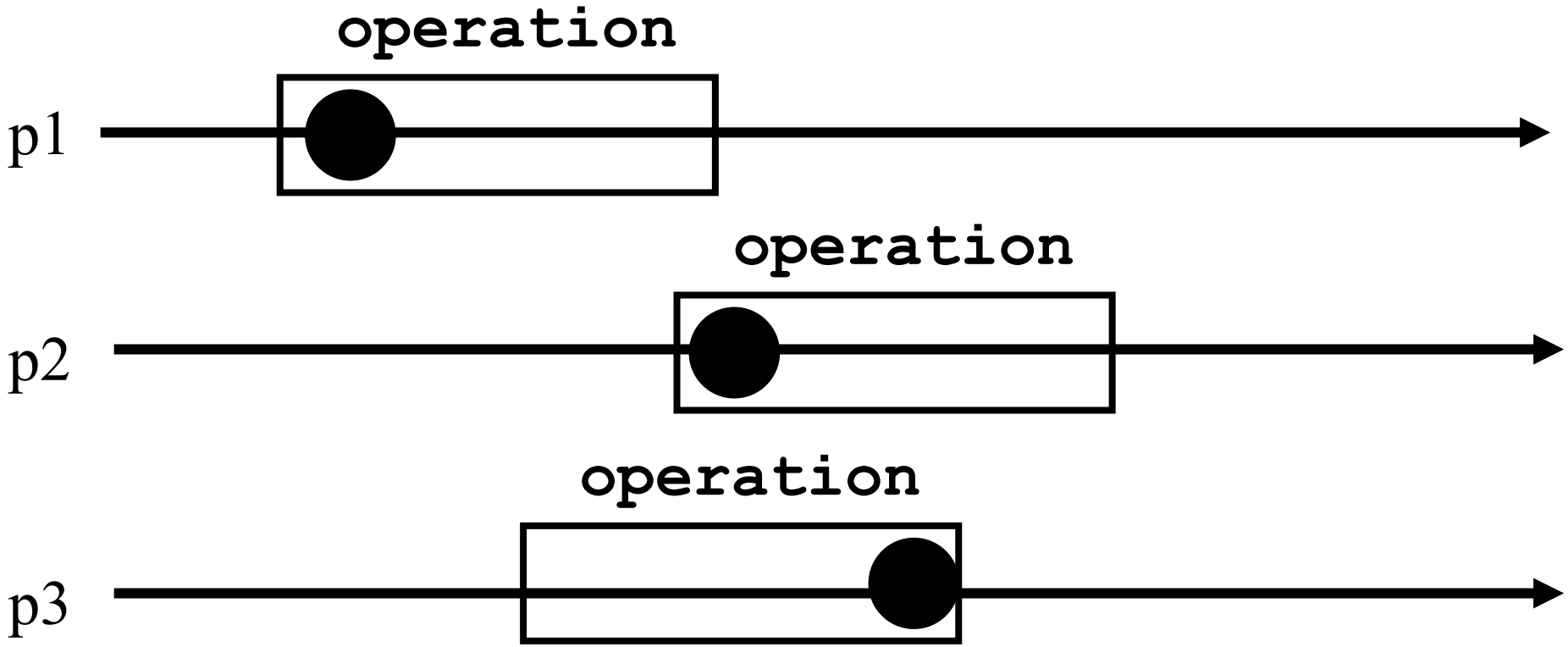
Processes



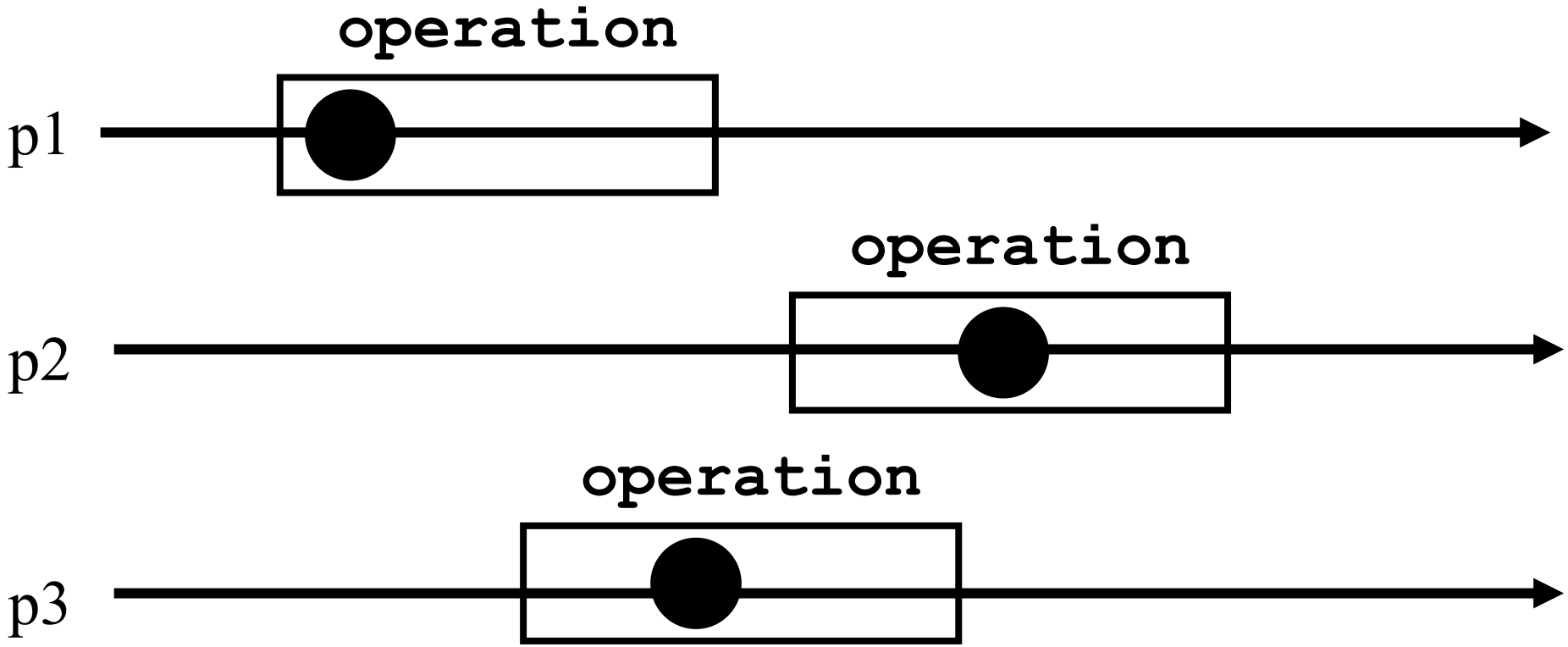
Atomicity

- Every operation appears to execute at some indivisible point in time (called linearization point) between the invocation and reply time events

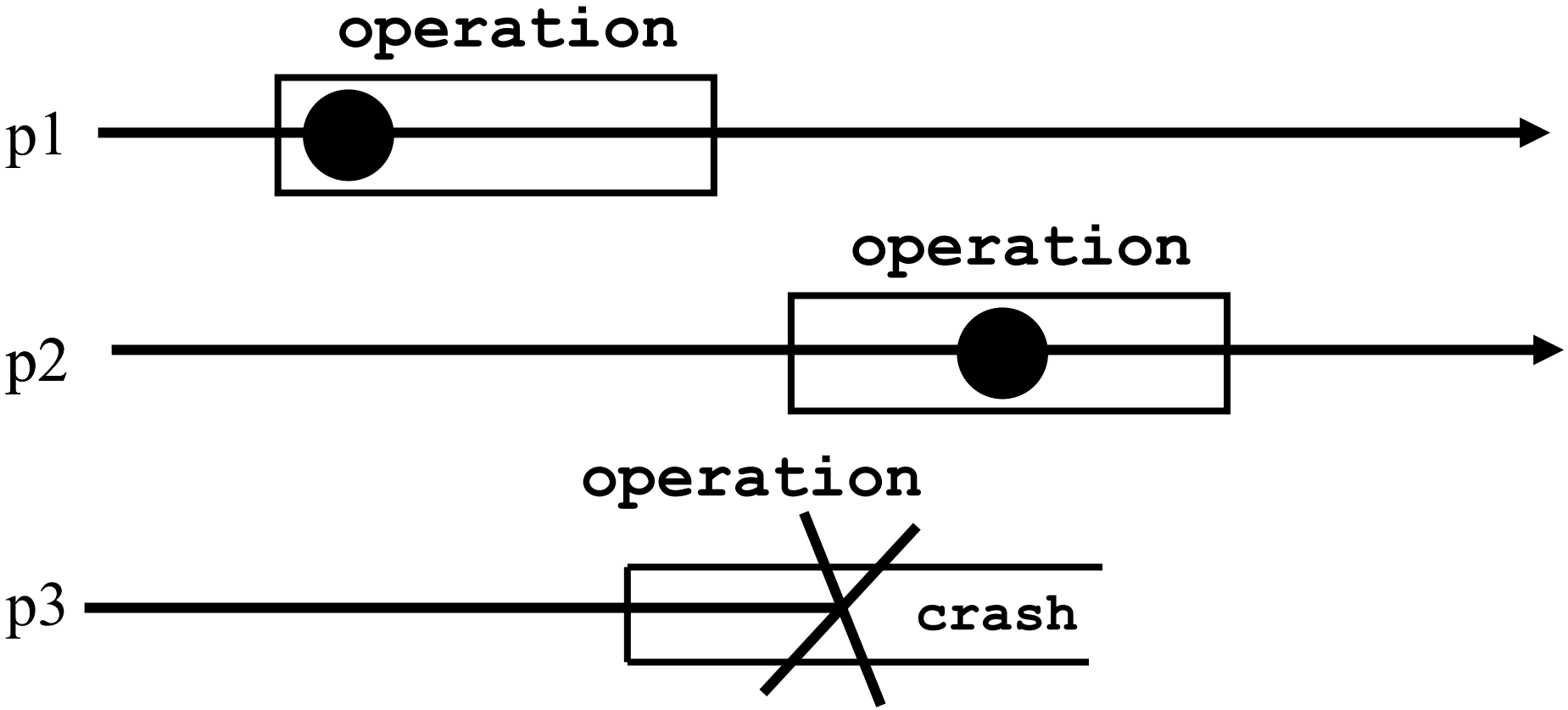
Atomicity



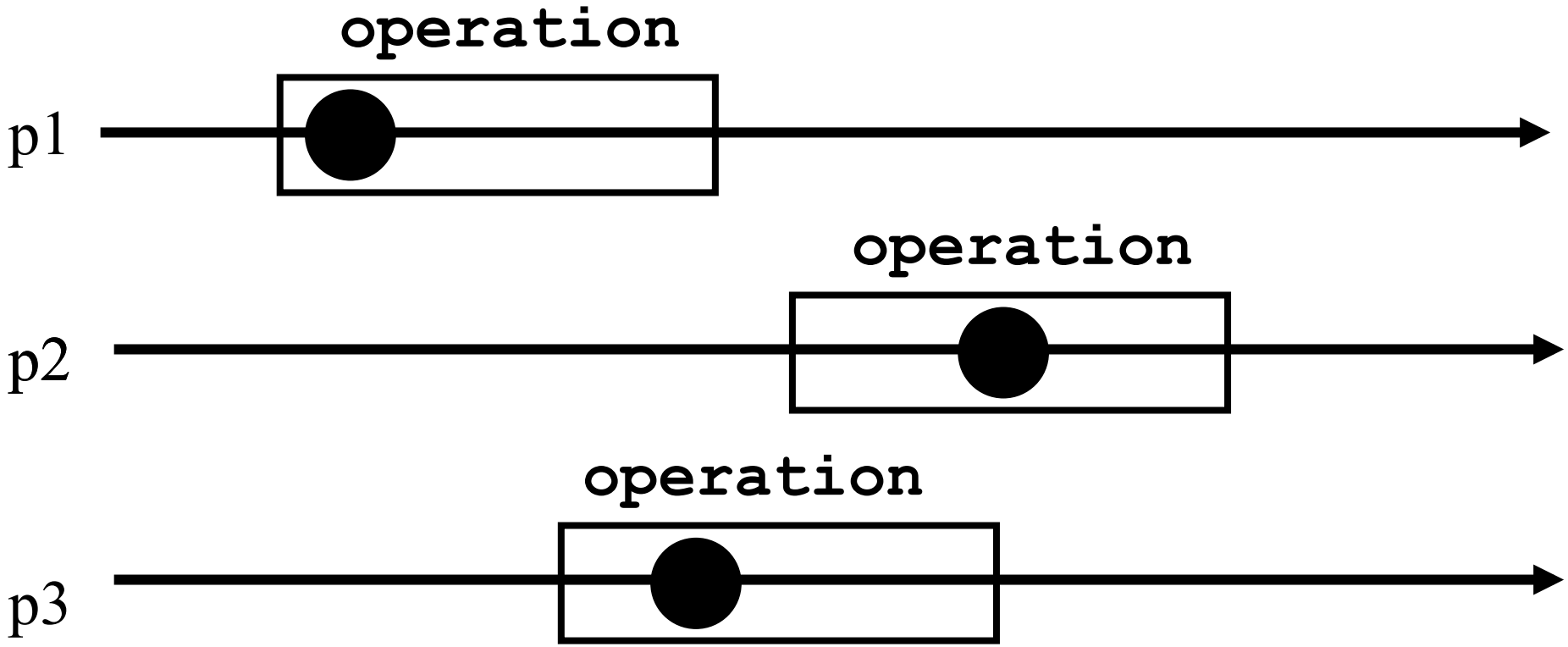
Atomicity



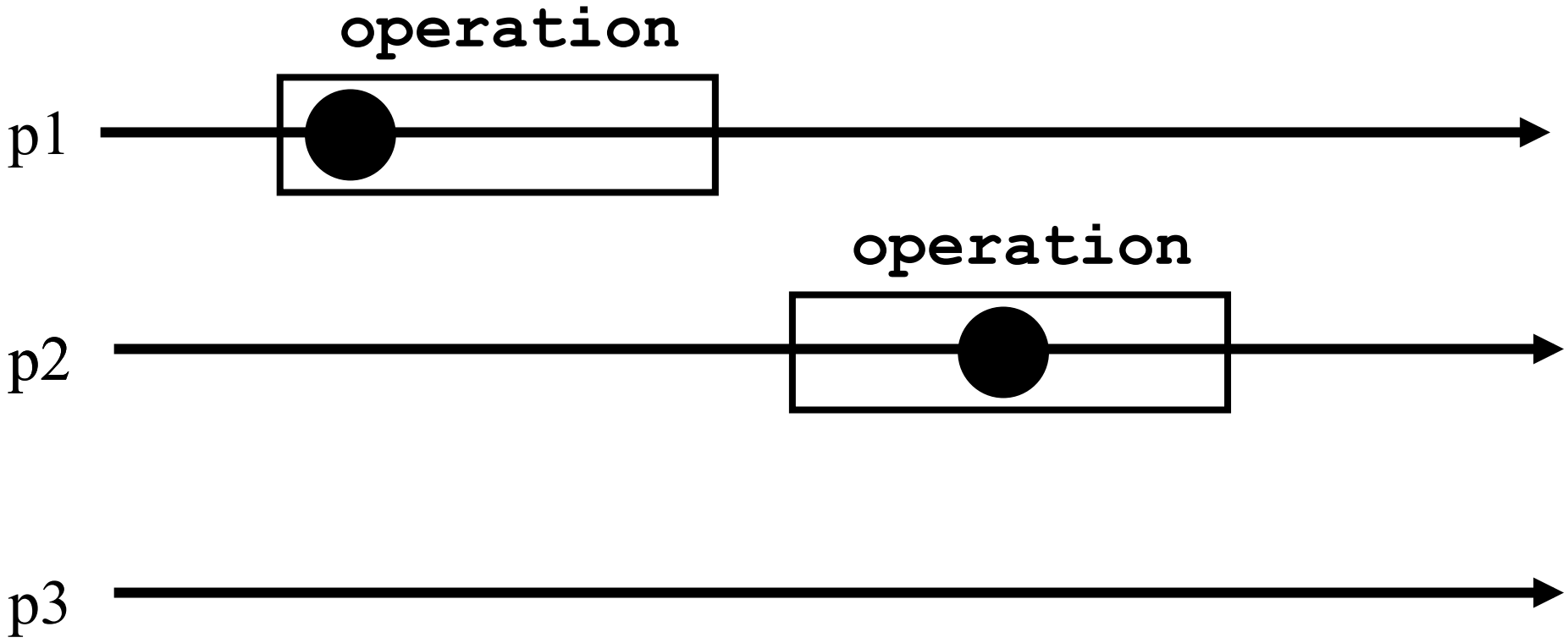
Atomicity (the crash case)



Atomicity (the crash case)



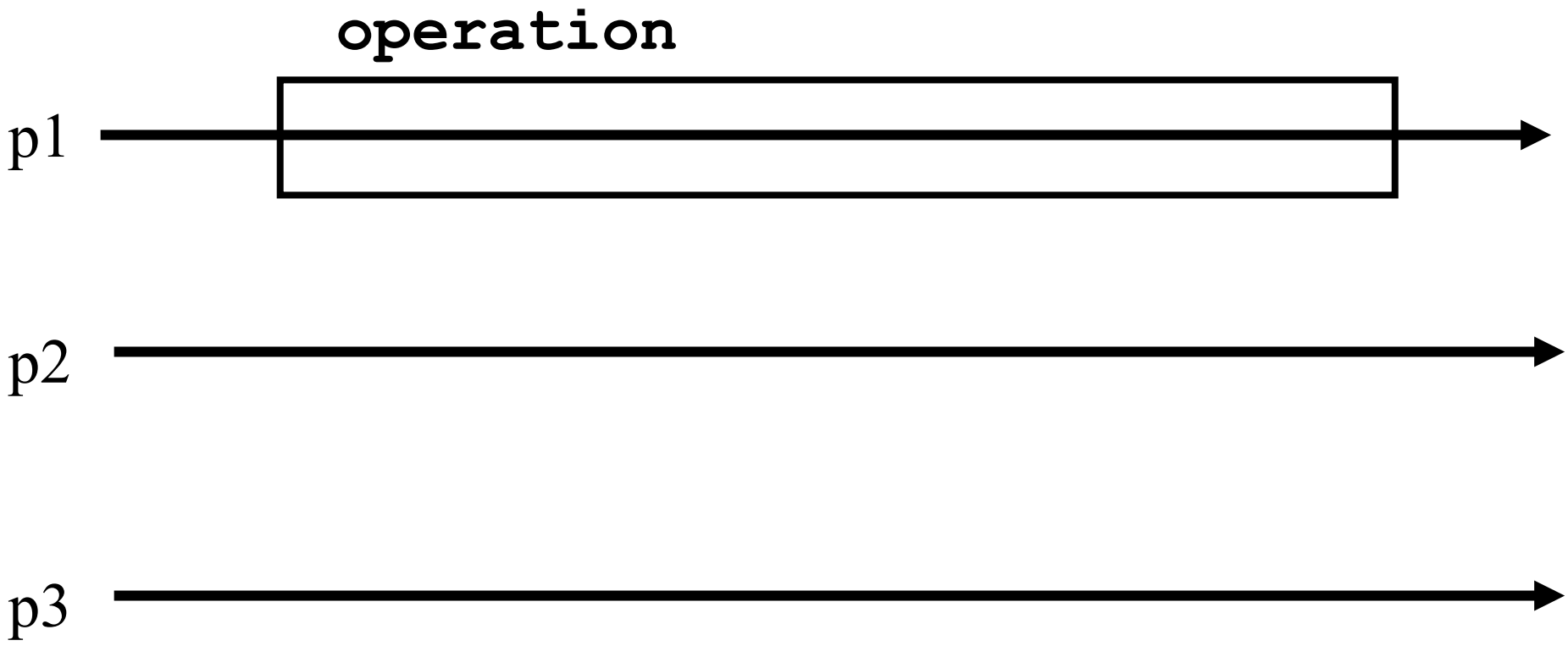
Atomicity (the crash case)



Wait-freedom

- Any correct process that invokes an operation eventually gets a reply, no matter what happens to the other processes (crash or very slow)

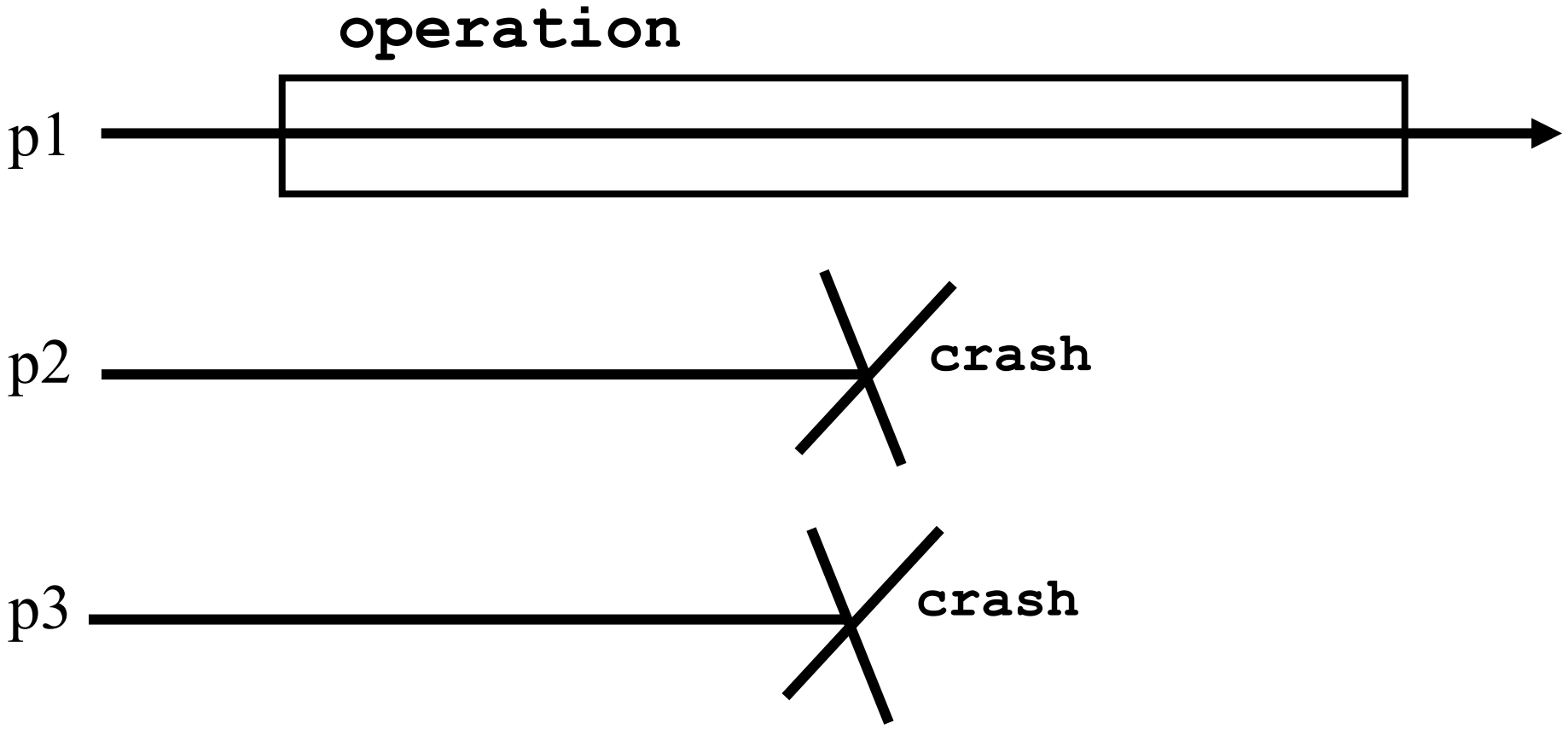
Wait-freedom



Wait-freedom

- Wait-freedom conveys the robustness of the implementation
- With a wait-free implementation, a process gets replies despite the crash of the n-1 other processes
- Note that this precludes implementations based on locks (mutual exclusion)

Wait-freedom



Example 1

- The reader/writer synchronization problem corresponds to the ***register*** object
- Basically, the processes need to read or write a shared data structure such that the value read by a process at a time t, is the last value written before t

Register

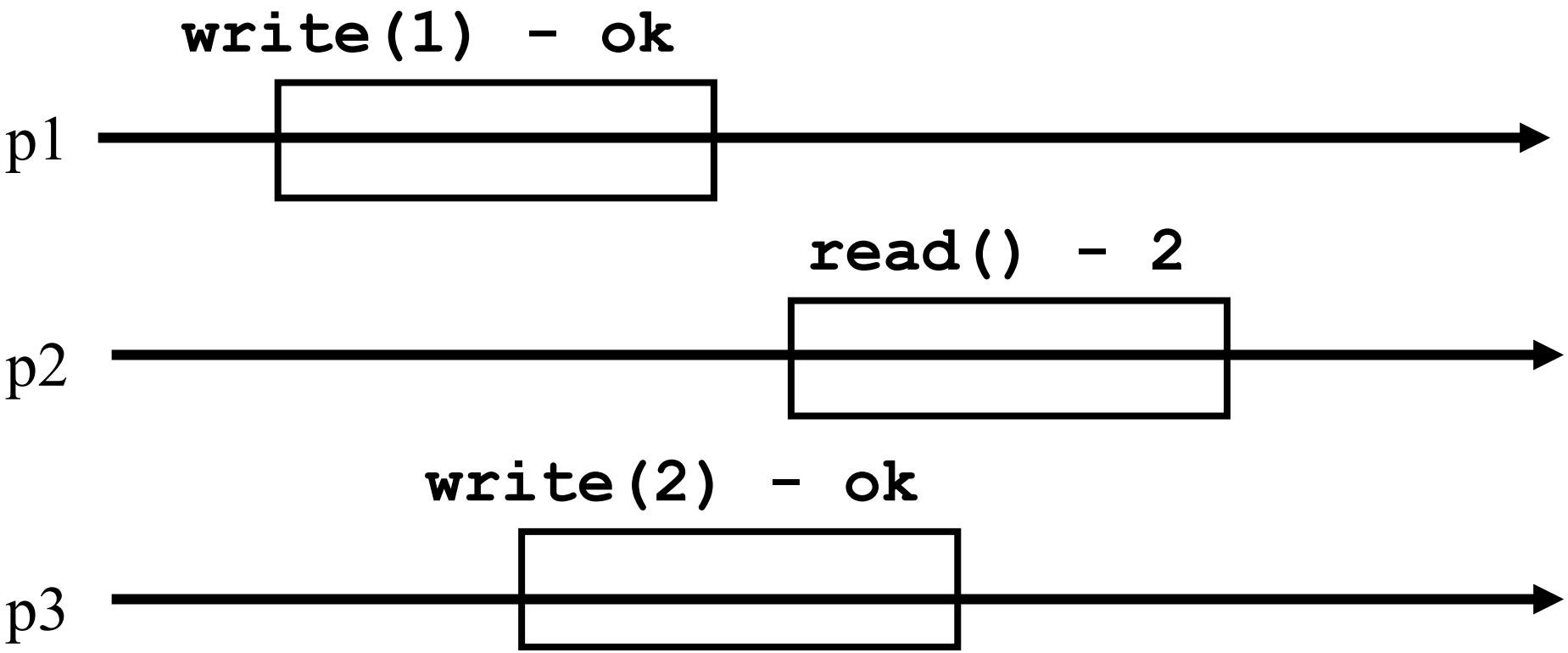
- A ***register*** has two operations: ***read()*** and ***write()***

- We assume that a ***register*** contains an integer for presentation simplicity, i.e., the value stored in the ***register*** is an integer, denoted by *x* (initially 0)

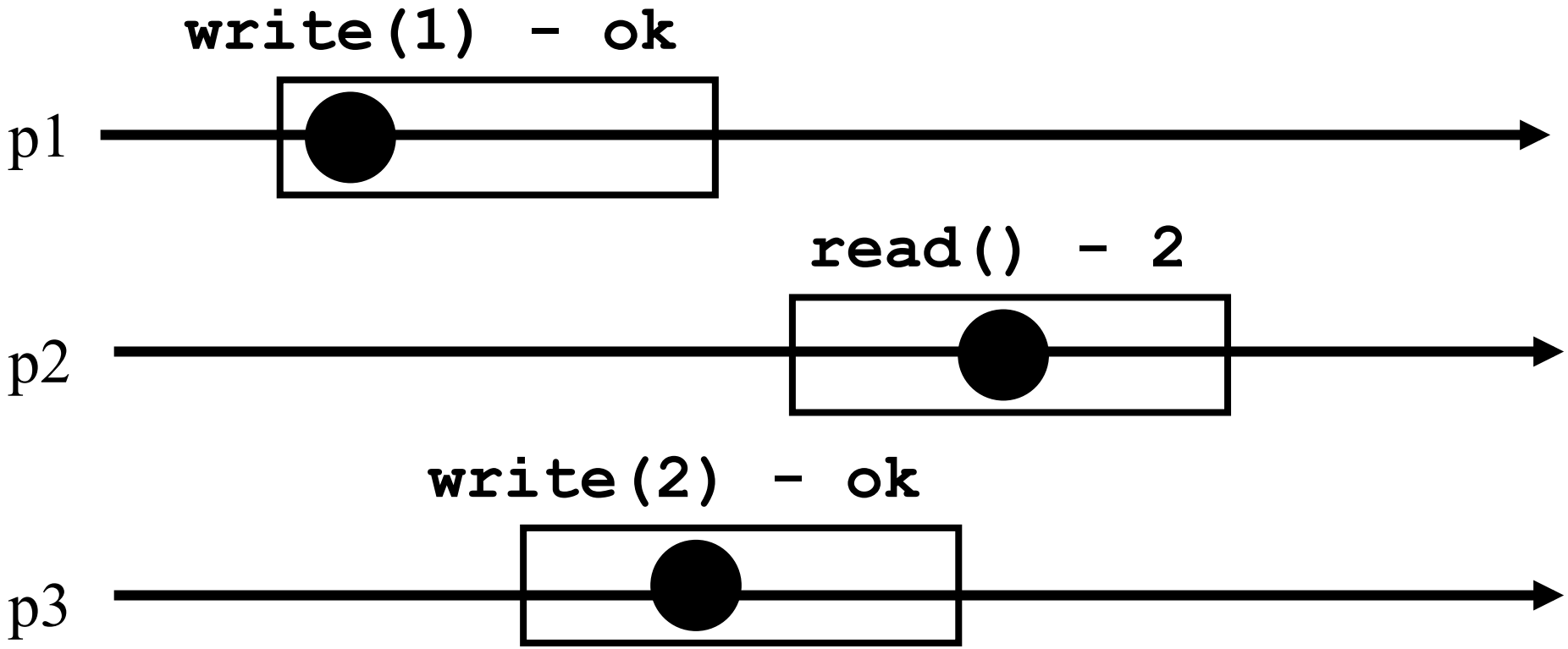
Sequential specification

- Sequential specification
  - ***read()***
    - return(x)
  - ***write(v)***
    - x <- v;
    - return(ok)

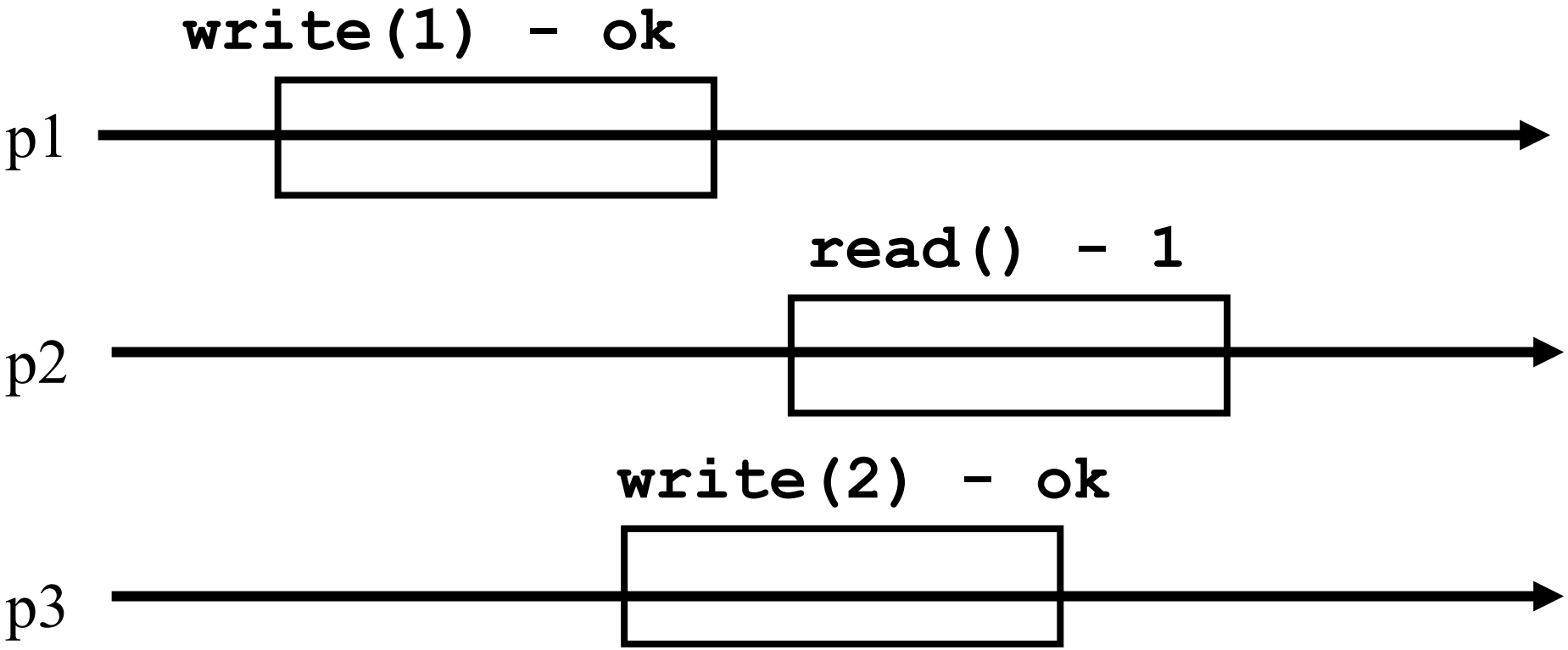
Atomicity?



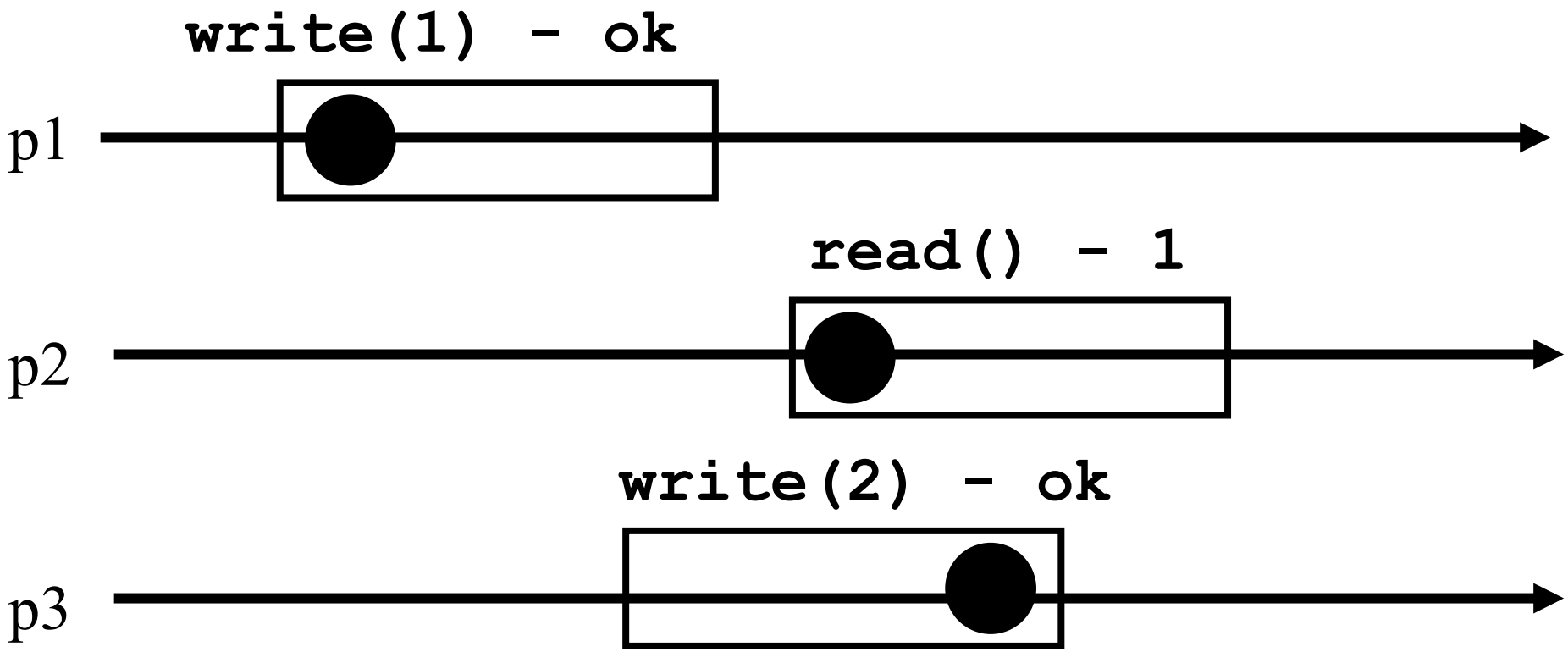
Atomicity?



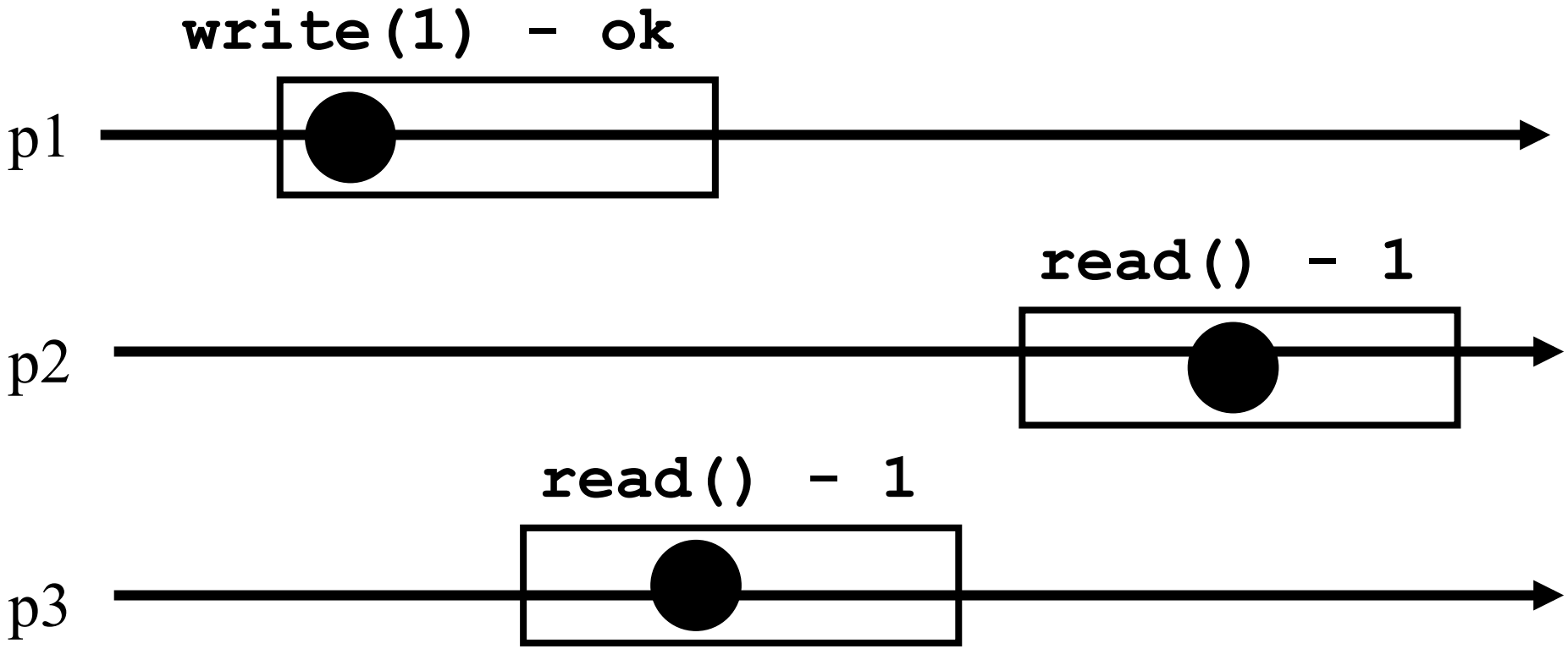
Atomicity?



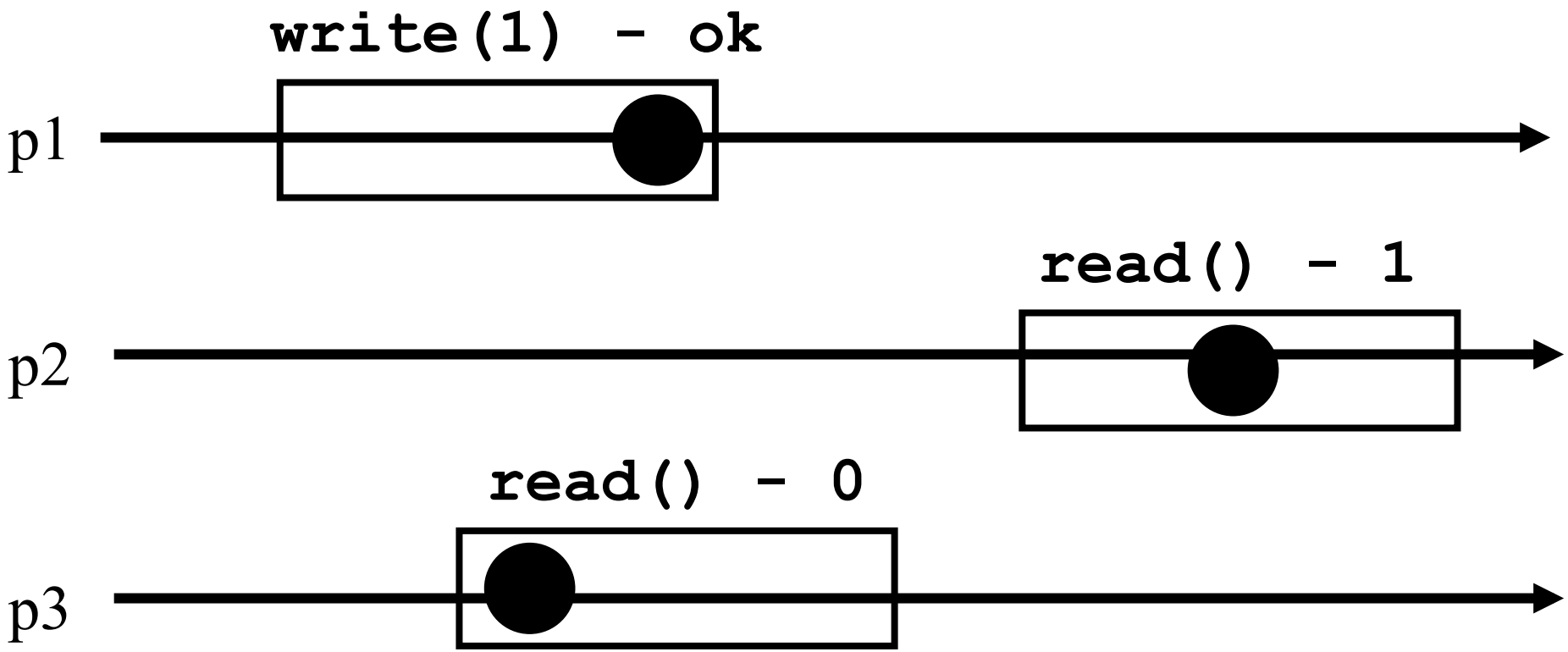
Atomicity?



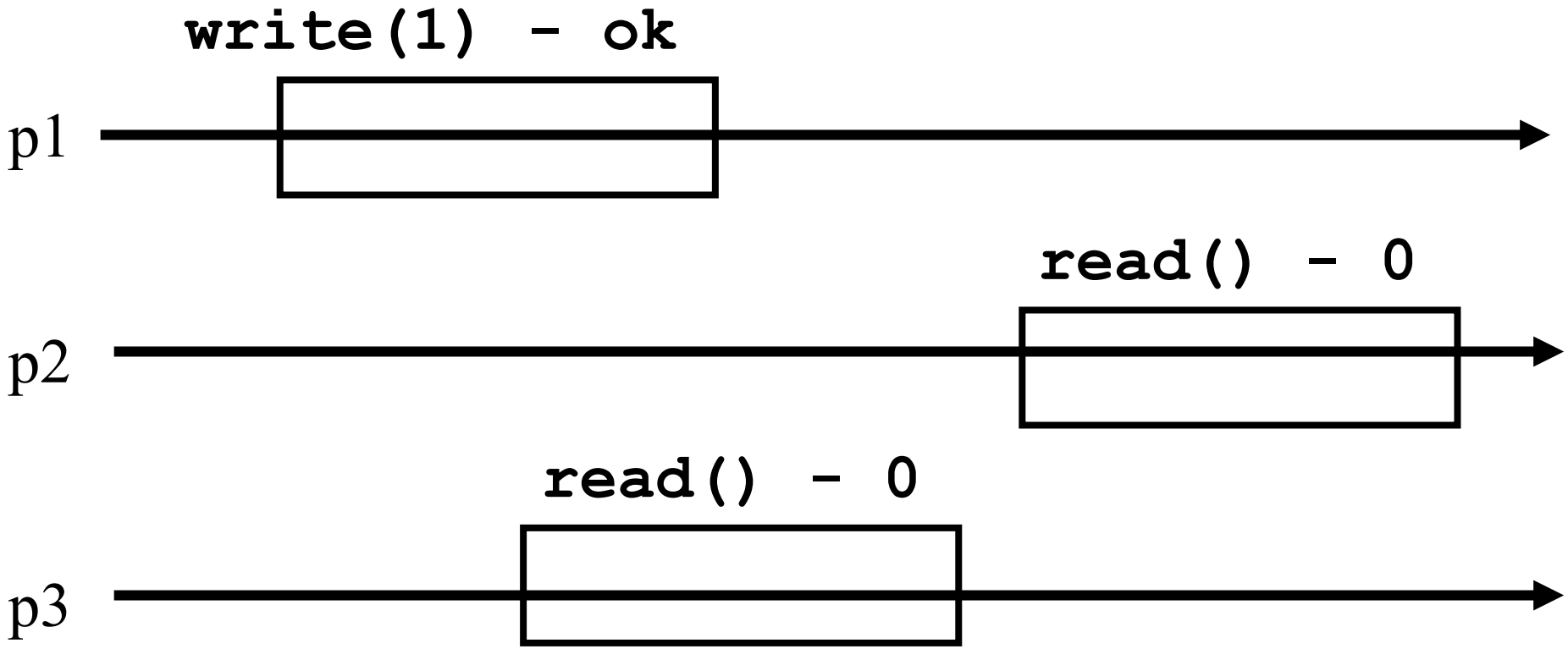
Atomicity?



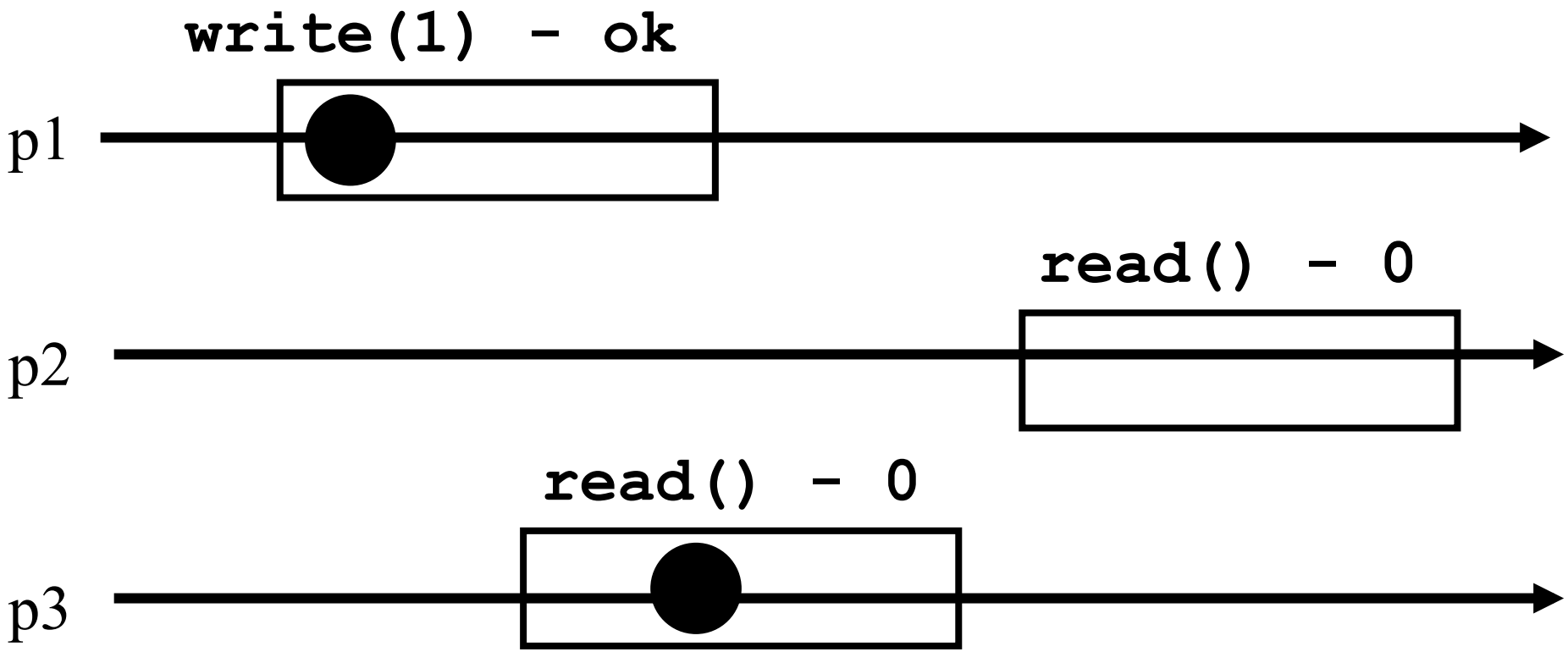
Atomicity?



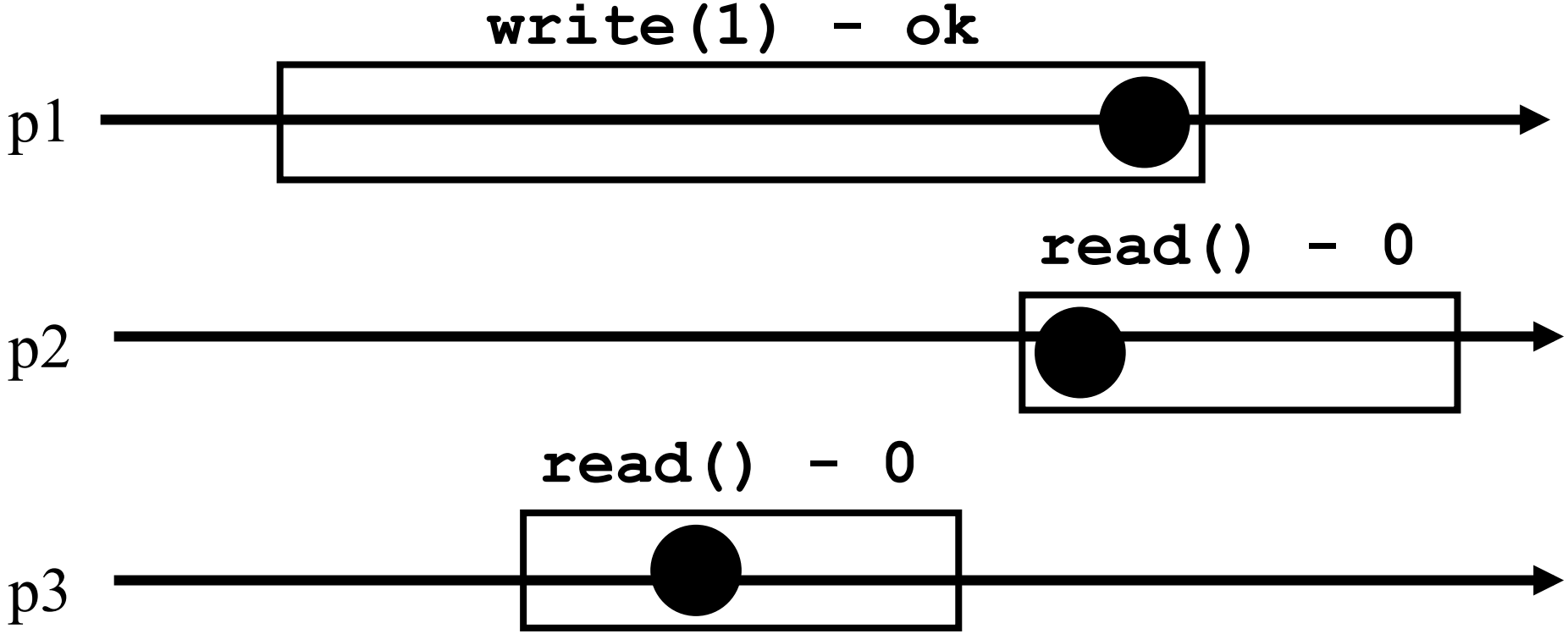
Atomicity?



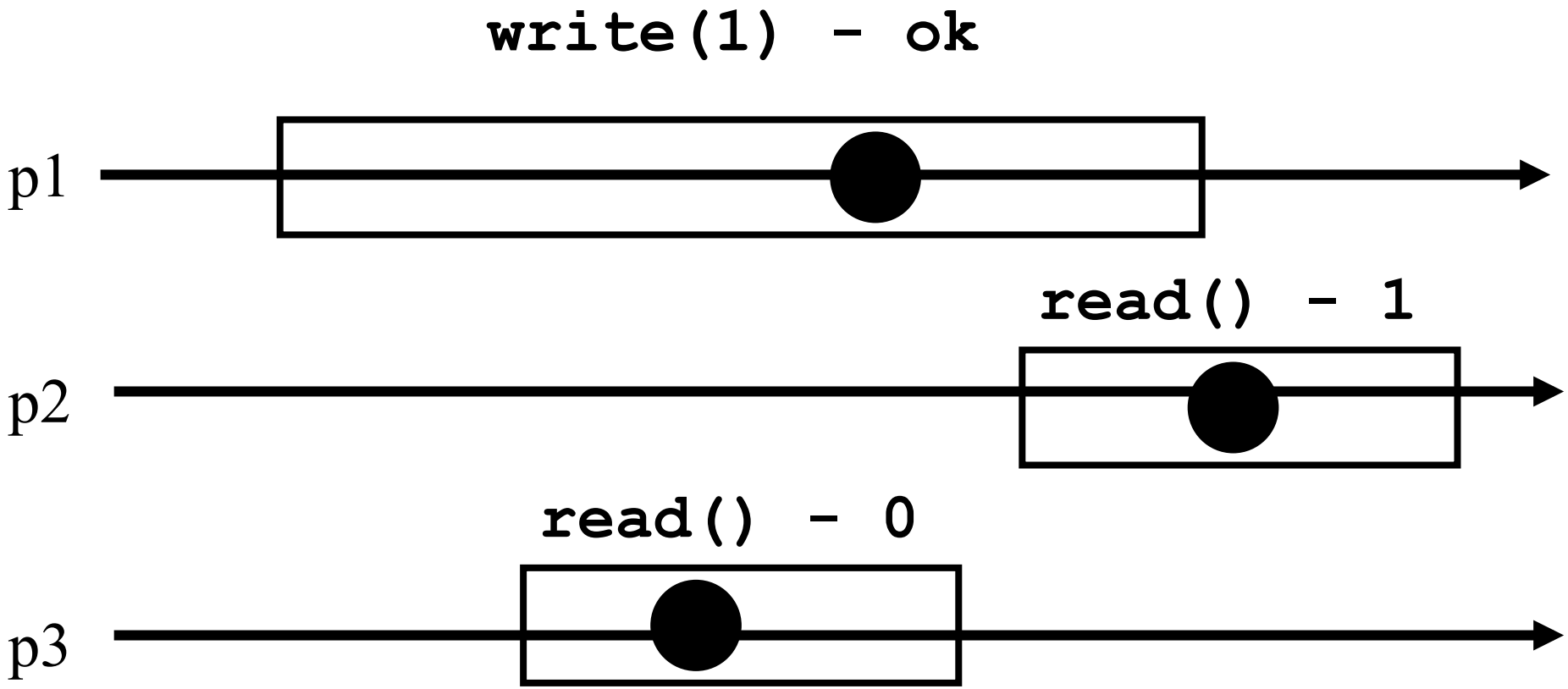
Atomicity?



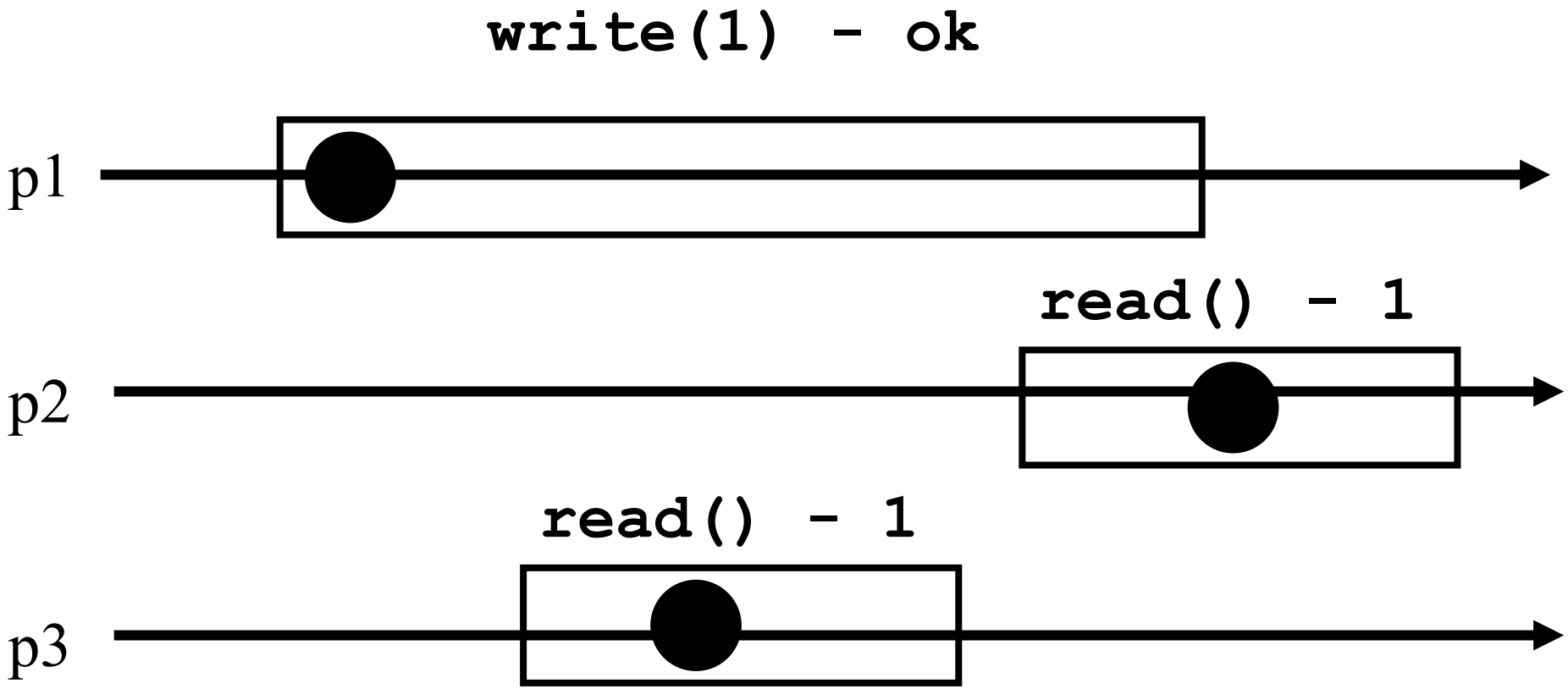
Atomicity?



Atomicity?



Atomicity?



Example 2

- The producer/consumer synchronization problem corresponds to the ***queue*** object
- Producer processes create items that need to be used by consumer processes
- An item cannot be consumed by two processes and the first item produced is the first consumed

Queue

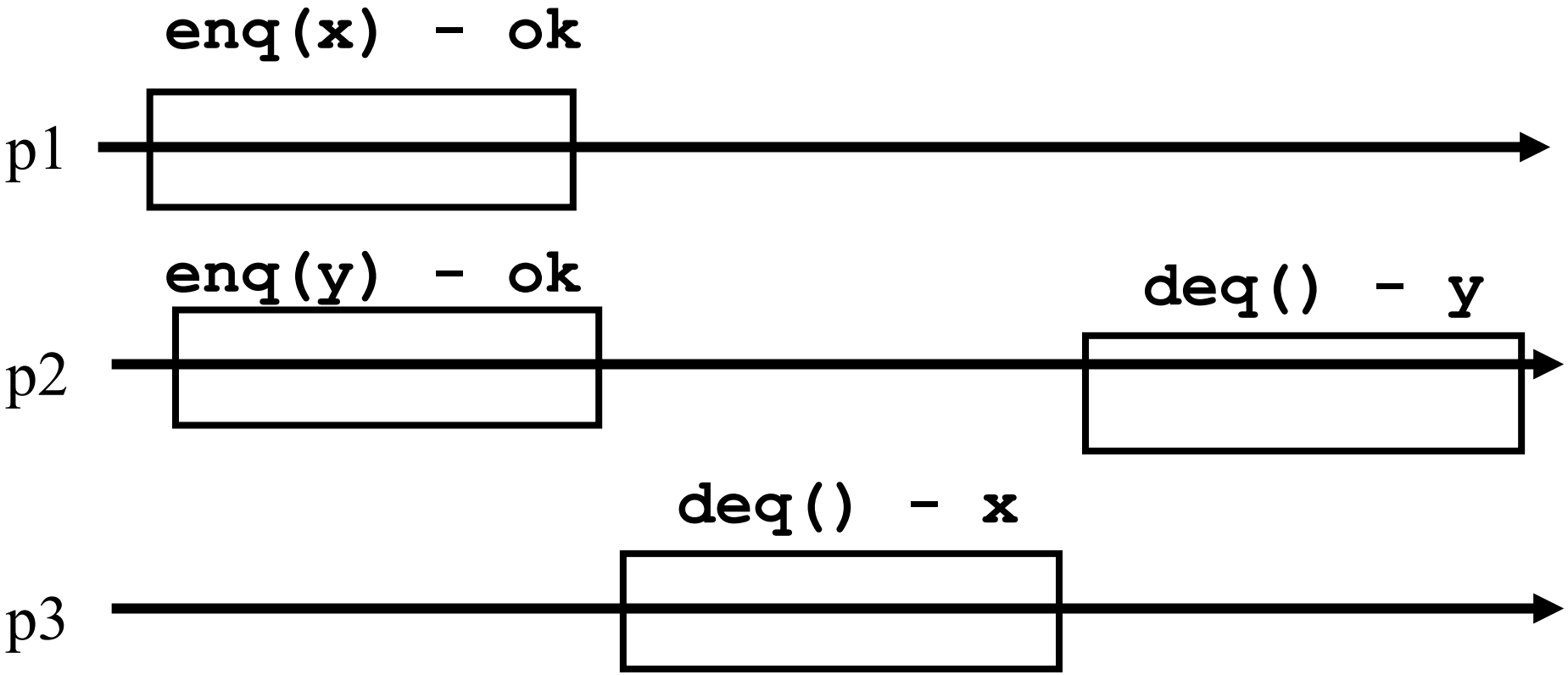
- A ***queue*** has two operations: ***enqueue()*** and ***dequeue()***

- We assume that a ***queue internally*** maintains a list *x* which exports operation *appends()* to put an item at the end of the list and *remove()* to remove an element from the head of the list

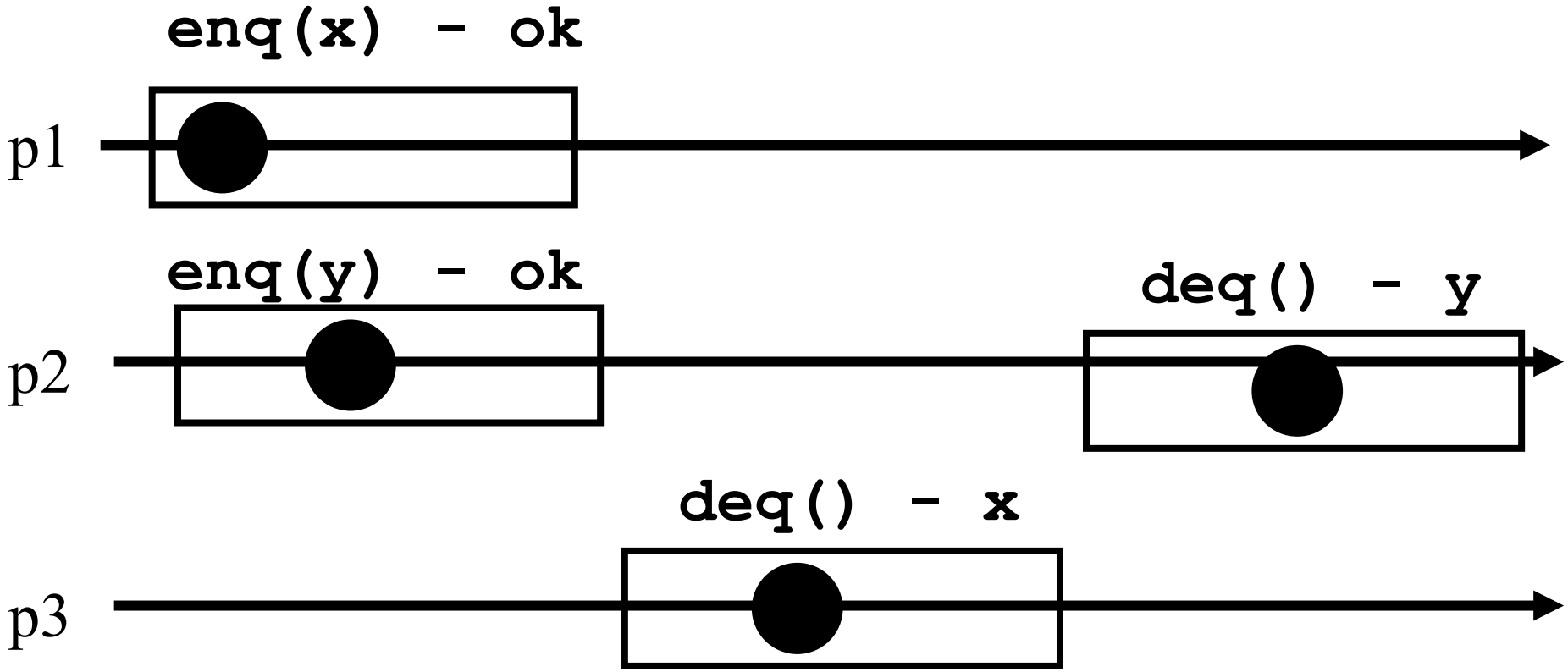
Sequential specification

- ***dequeue()***
  - if(x=0) then return(nil);
  - else return(x.remove())
- ***enqueue(v)***
  - x.append(v);
  - return(ok)

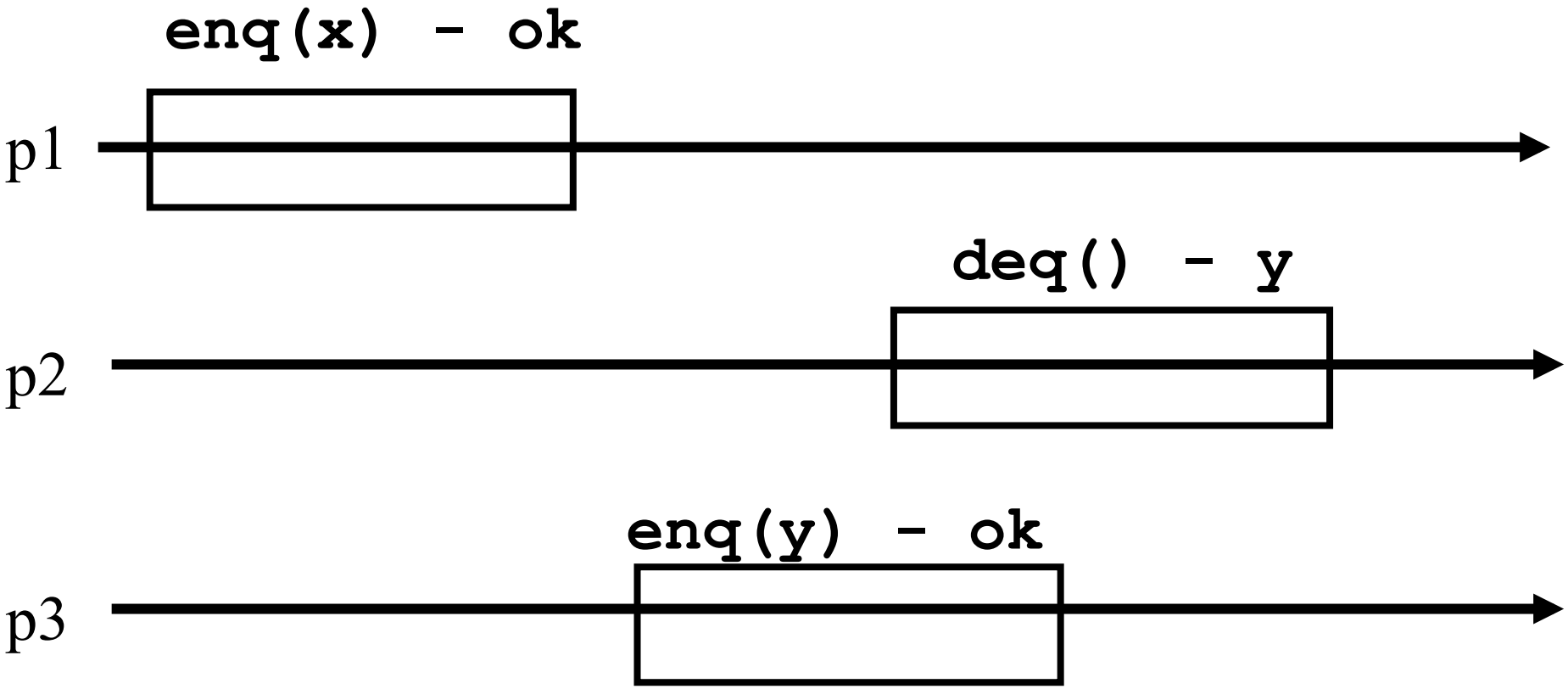
Atomicity?



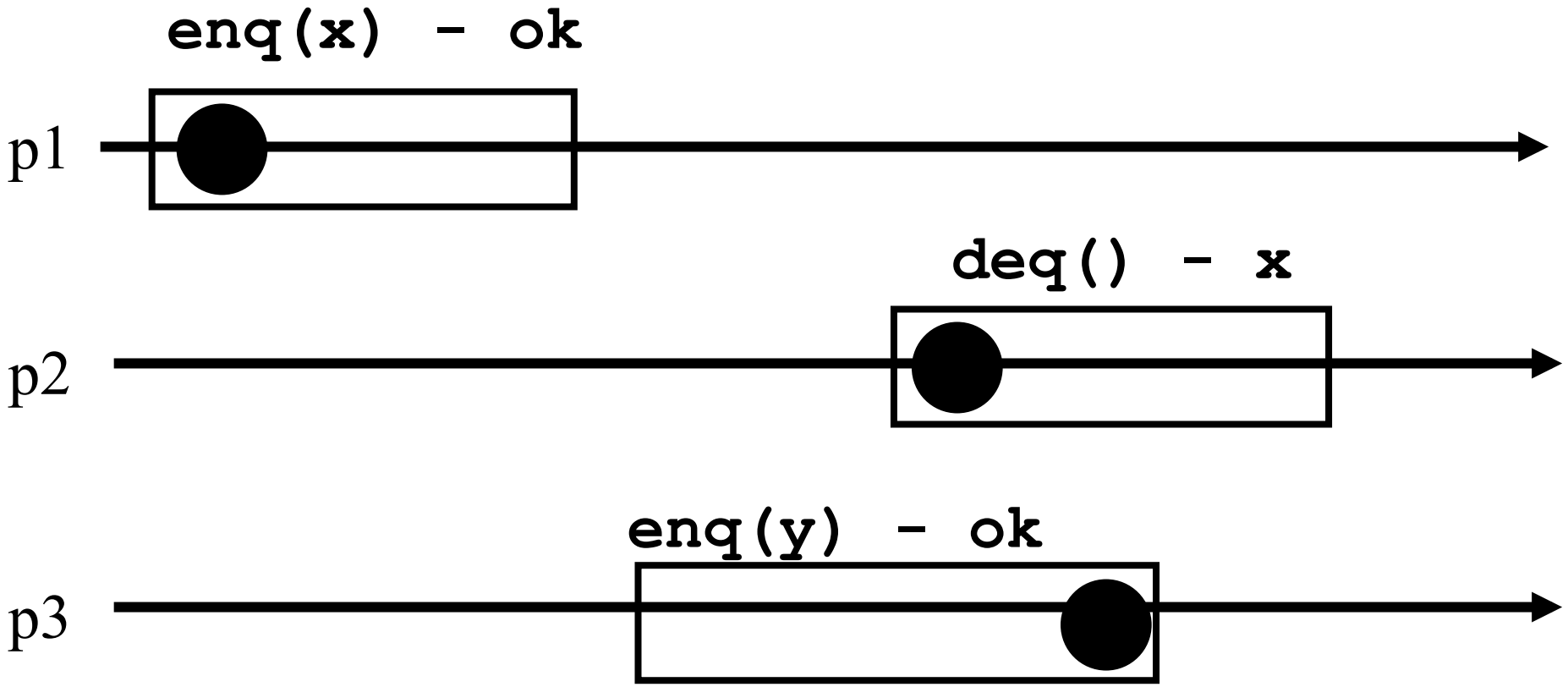
Atomicity?



Atomicity?

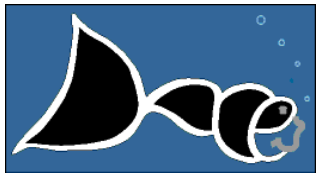


Atomicity?



Registers

Prof R. Guerraoui
Distributed Programming Laboratory



Register

- A ***register*** has two operations: ***read()*** and ***write()***
- Sequential specification
- - ***read()***
    - return(x)
  - ***write(v)***
    - x <- v; return(ok)

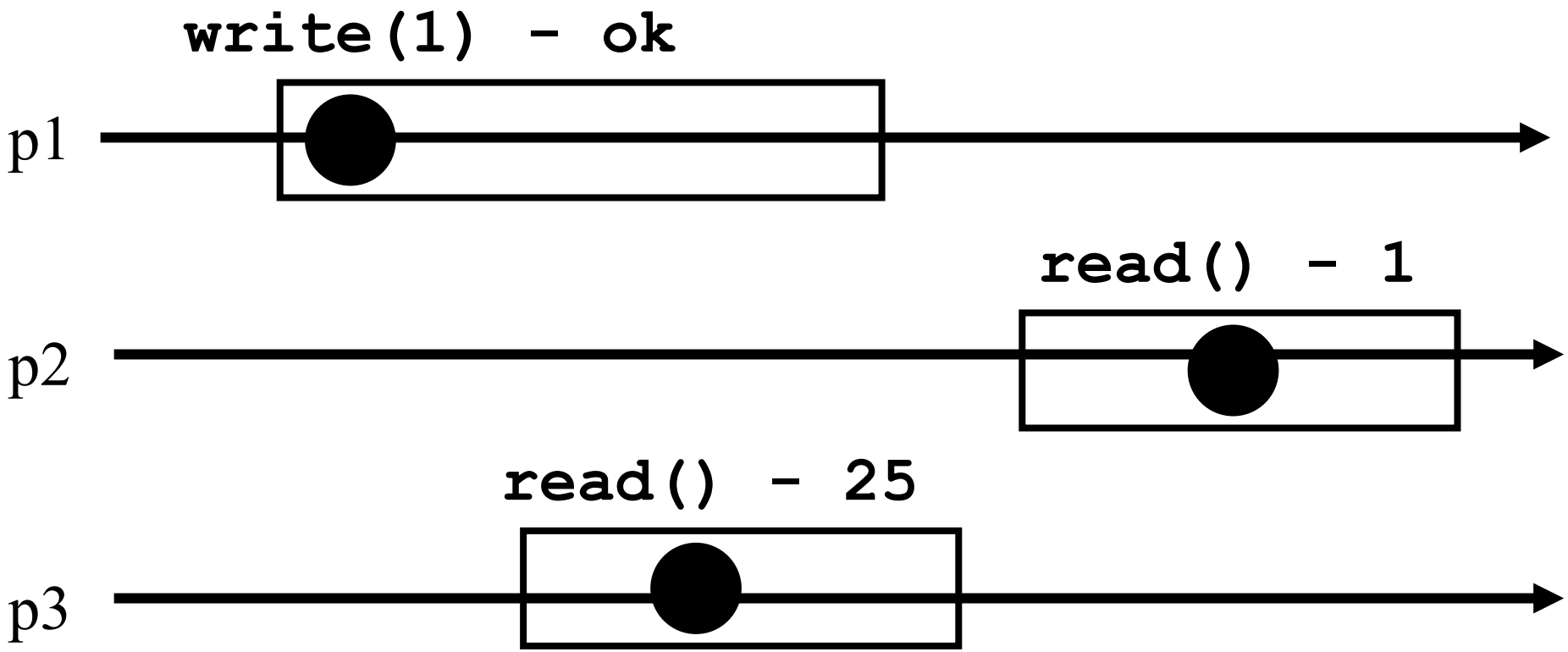
Simplifications

- We assume that ***registers*** contain only integers
- Unless explicitely stated otherwise, ***registers*** are initially supposed to contain 0

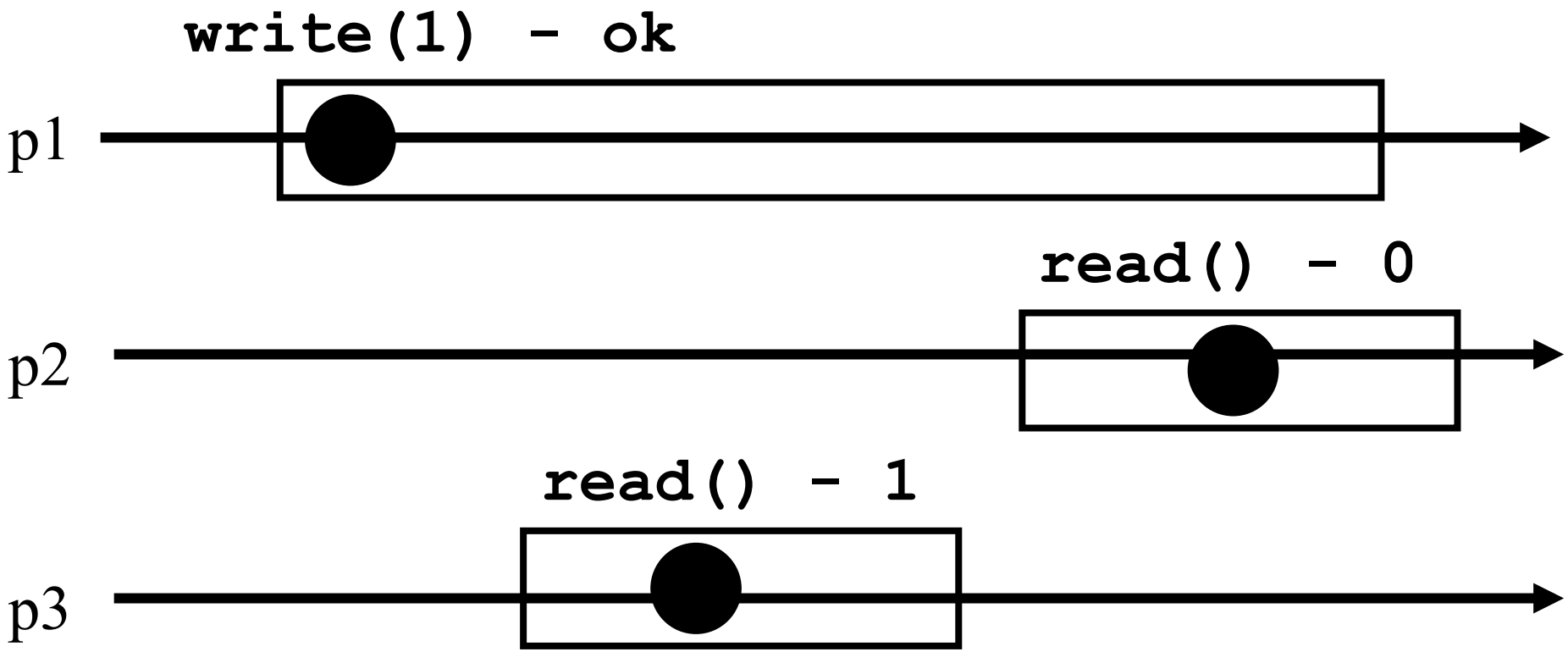
Space of registers

- Dimension 1: binary (boolean) – multivalued
- Dimension 2:
  - SRSW (single reader, single writer)
  - MRSW (multiple reader, single writer)
  - MRMW (multiple reader, multiple writer)
- Dimension 3: safe – regular – atomic

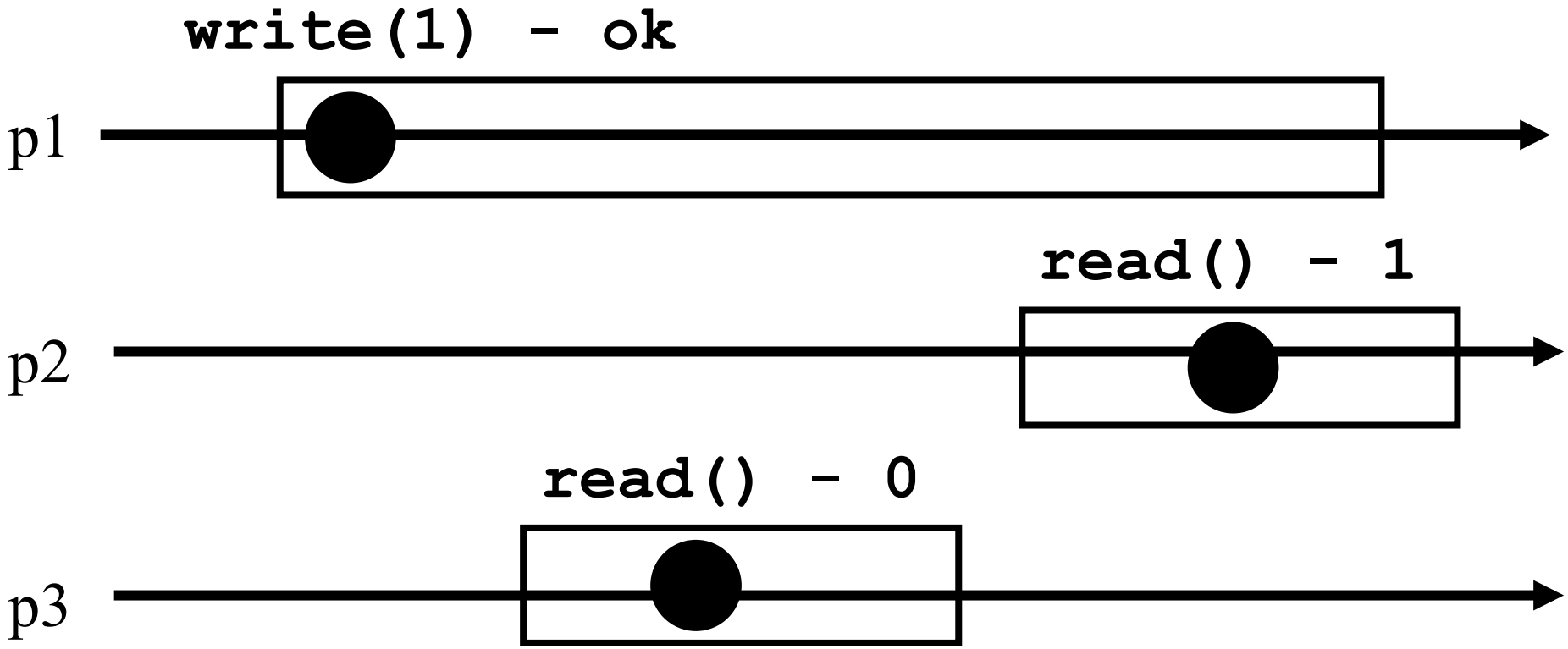
Safe execution



Regular execution



Atomic execution



2 decades of hard work

- Theorem: A multivalued MRMW atomic ***register*** can be implemented with binary SRSW safe ***register***

Algorithms

- The process executing the code is implicitely assumed to be pi
- We assume a system of N processes
- NB. We distinguish base and high-level registers

Conventions

- The operations to be implemented are denoted ***Read()*** and ***Write()***
- Those of the base registers are denoted ***read()*** and ***write()***

- We omit the ***return(ok)*** instruction at the end of ***Write()*** implementations

(1) From (binary) SRSW safe to (binary) MRSW safe

- We use an array of SRSW ***registers***

  Reg[1,..,N]

- **Read()**
  - return (Reg[i].read());

- **Write(v)**
  - for j = 1 to N
    - Reg[j].write(v);

(2) From binary MRSW safe to binary MRSW regular

- We use one MRSW safe register
- **Read()**
  - return(Reg.read());

- **Write(v)**
  - if old ≠ v then
    - Reg.write(v);
    - old := v;

(3) From *binary* to *M-Valued* MRSW regular

- We use an array of MRSW registers Reg[0,1,..,M] init to [1,0,..,0]

- **Read()**
  - for j = 0 to M
    - if Reg[j].read() = 1 then return(j)
- **Write(v)**
  - Reg[v].write(1);
  - for j=v-1 downto 0
    - Reg[j].write(0);

(4) From SRSW *regular* to SRSW *atomic*

- We use one SRSW register Reg and two local variables t and x

- **Read()**
  - (t',x') = Reg.read();
  - if t' > t then t:=t'; x:=x';
  - return(x)
- **Write(v)**
  - t := t+1;
  - Reg.write(v,t);

(5) From SRSW atomic to MRSW atomic

- We use N*N SRSW atomic registers RReg[(1,1),(1,2),..,(k,j),..(N,N)] to communicate among the readers
  - In RReg[(k,j)] the reader is pk and the writer is pj

- We also use n SRSW atomic ***registers*** WReg[1,..,N] to store new values
  - the writer in all these is p1
  - the reader in WReg[k] is pk

(5) From SRSW atomic to MRSW atomic (cont'd)

- **Write(v)**
  - t1 := t1+1;
  - for j = 1 to N
    - WReg.write(v,t1);

(5) From SRSW atomic to MRSW atomic (cont'd)

- **Read()**
  - for j = 1 to N do
    - (t[j],x[j]) = RReg[i,j].read();
  - (t[0],x[0]) = WReg[i].read();
  - (t,x) := highest(t[..],x[..]);
  - for j = 1 to N do
    - RReg[j,i].write(t,x);
  - return(x)

Value with highest timestamp

(6) From *MRSW* atomic to *MRMW* atomic

- We use N MRSW atomic registers Reg[1,..,N]; the writer of Reg[j] is pj

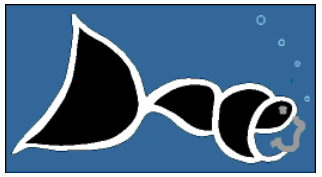
- **Write(v)**
  - for j = 1 to N do
    - (t[j],x[j]) = Reg[j].read();
  - (t,x) := highest(t[..],x[..]);
  - t := t+1;
  - Reg[i].write(t,v);

(6) From MRSW atomic to MRMW atomic (cont'd)

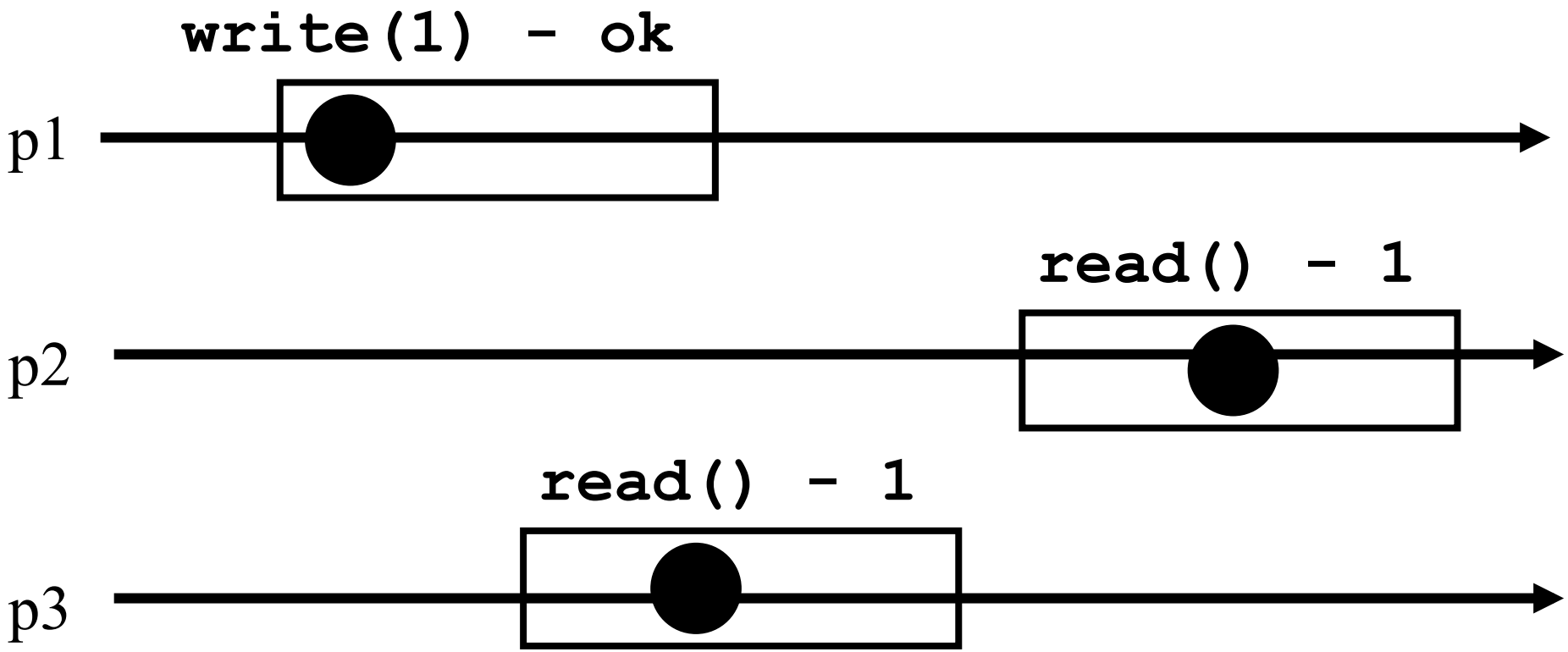
- **Read()**
  - for j = 1 to N do
    - (t[j],x[j]) = Reg[j].read();
  - (t,x) := highest(t[..],x[..]);
  - return(x)

The Power of Registers

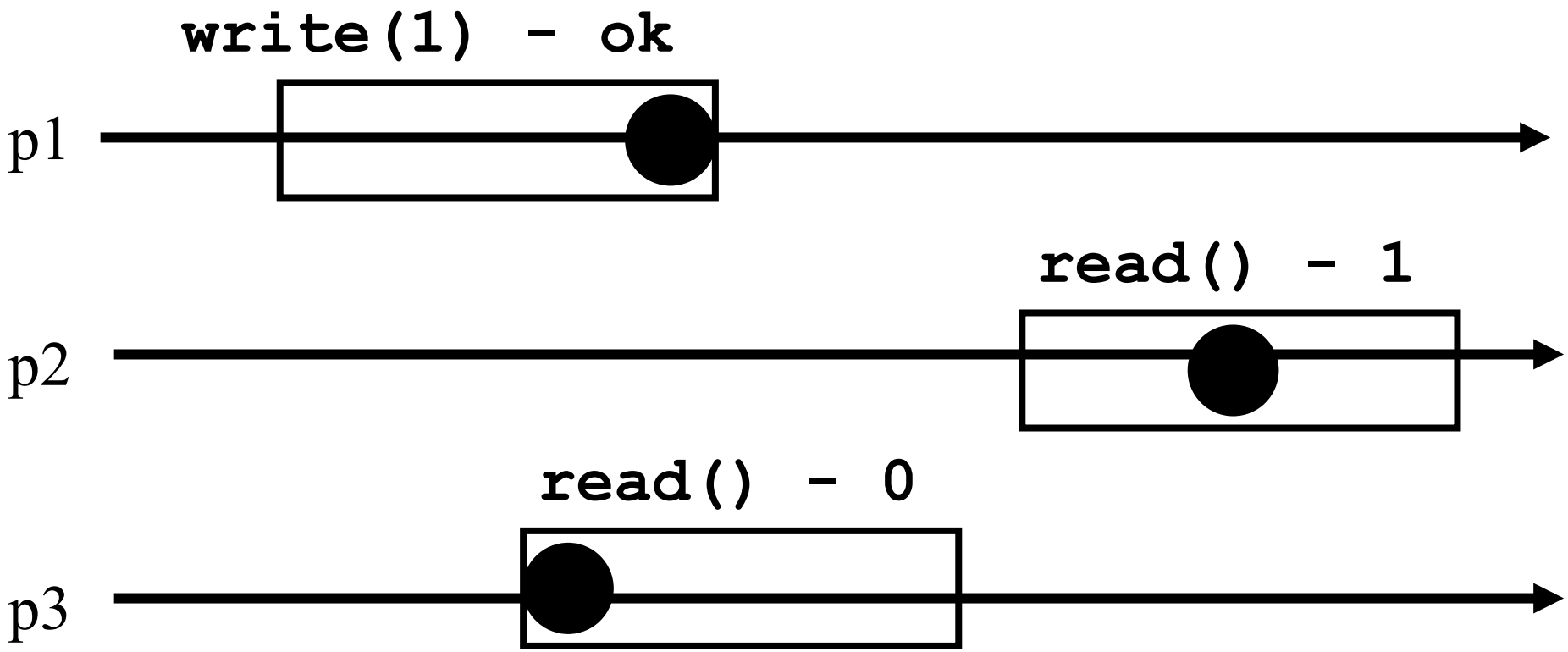
Prof R. Guerraoui
Distributed Programming Laboratory



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)

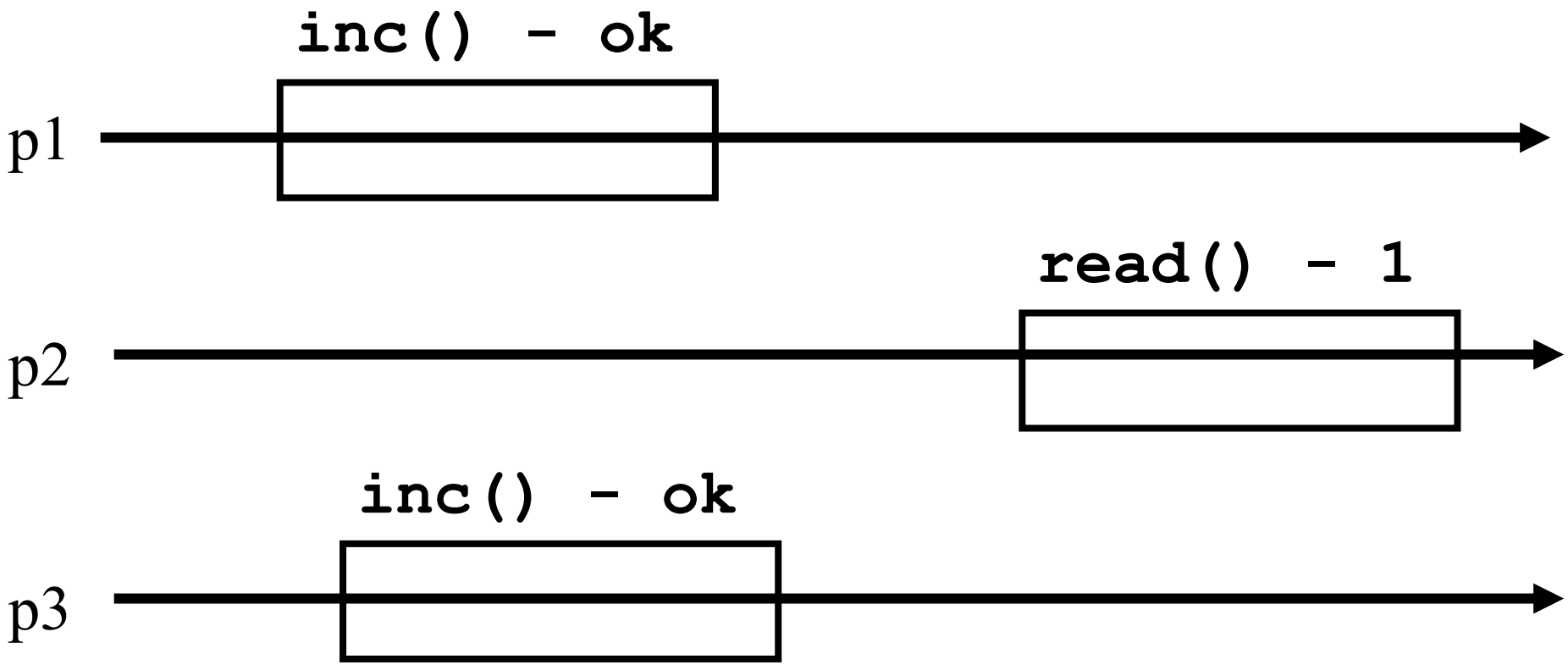
- A ***counter*** has two operations ***inc()*** and ***read()*** and maintains an integer *x init to 0*

- ***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

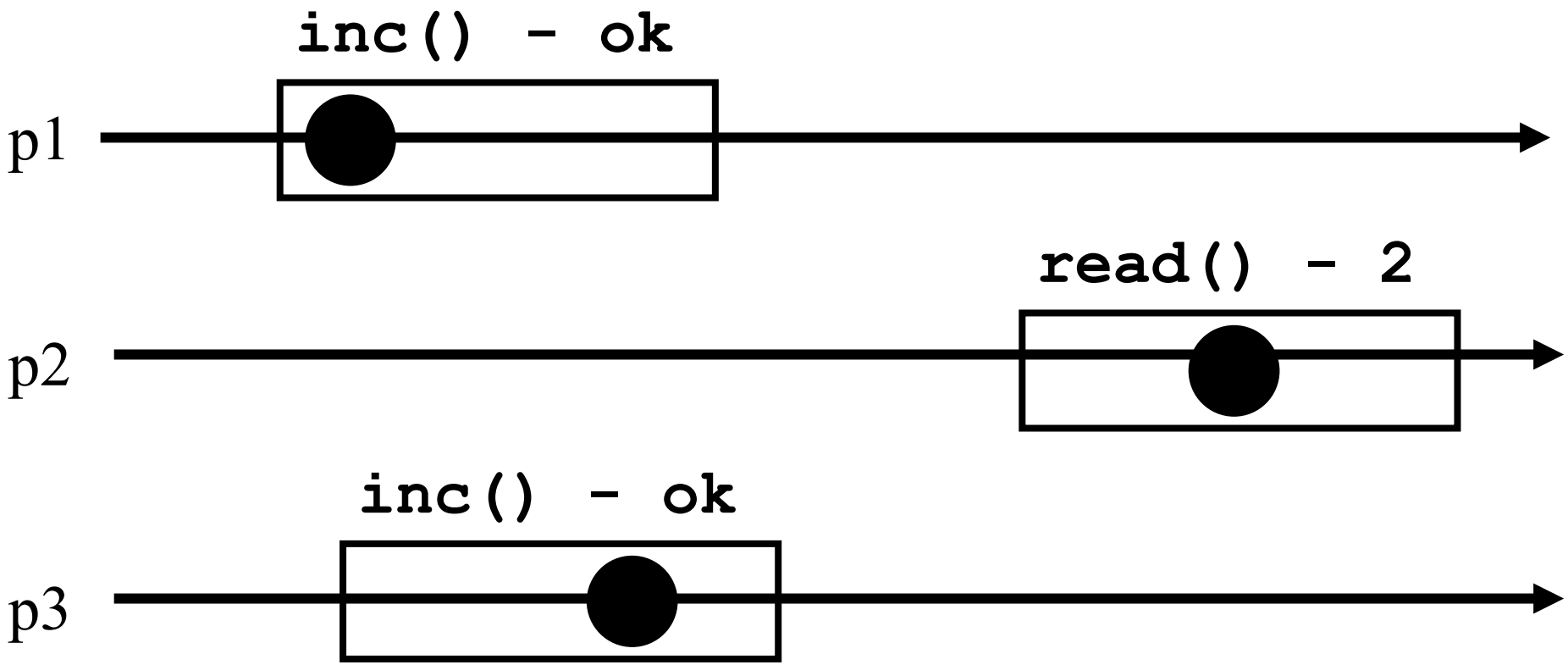
- The processes share an array of registers Reg[1,..,n]

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

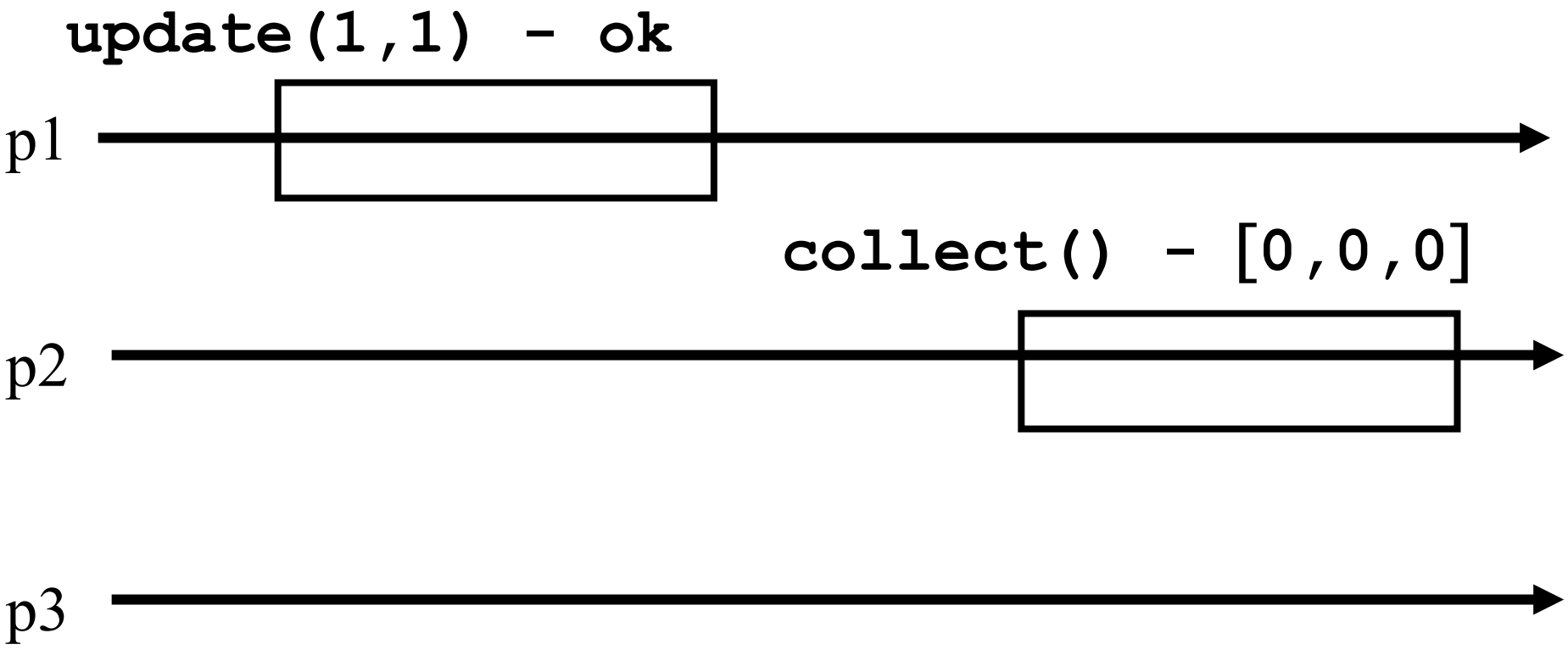
- ***scan():***
  - return(x)
- ***update(i,v):***
  - x[i] := v;
  - return(ok)

Very naive implementation

- Each process maintains an array of integer variables x init to [0,..,0]

- ***scan():***
  - return(x)
- ***update(i,v):***
  - x[i] := v;
  - return(ok)

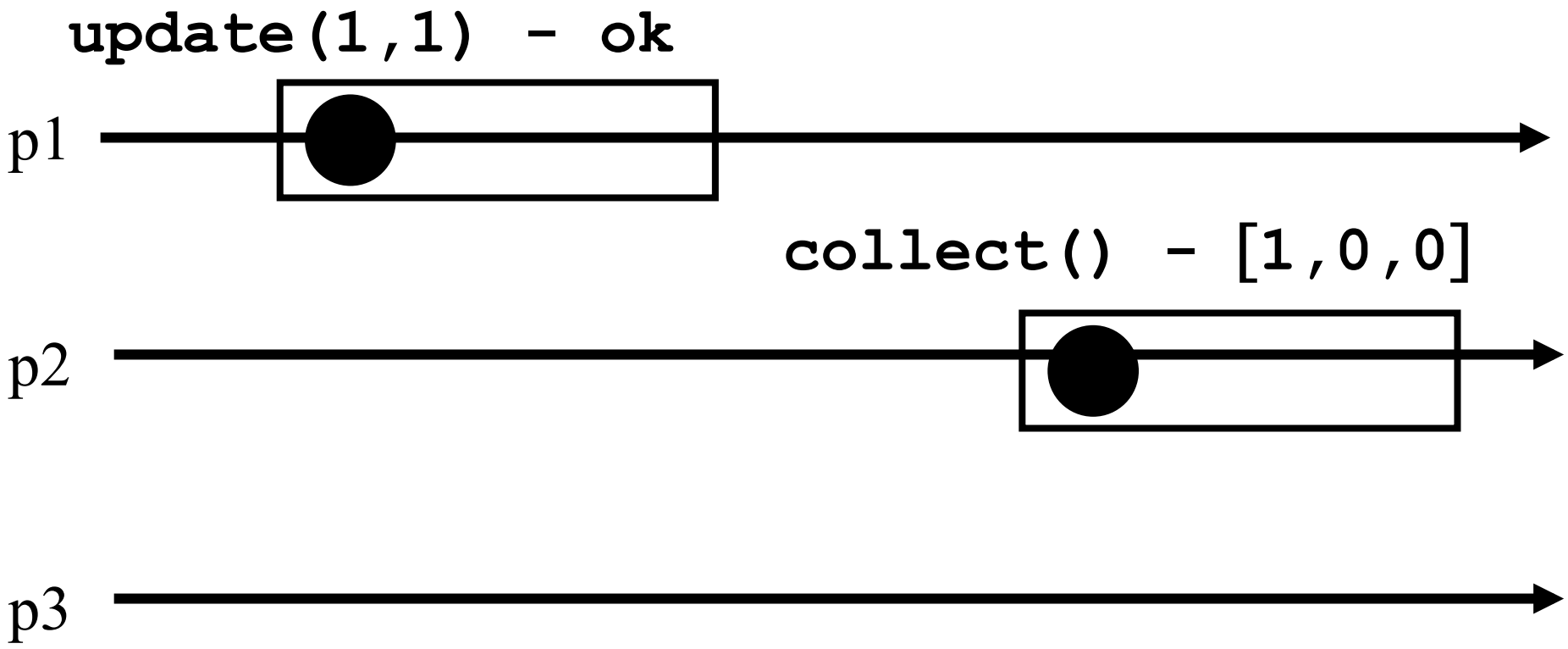
Atomic execution?



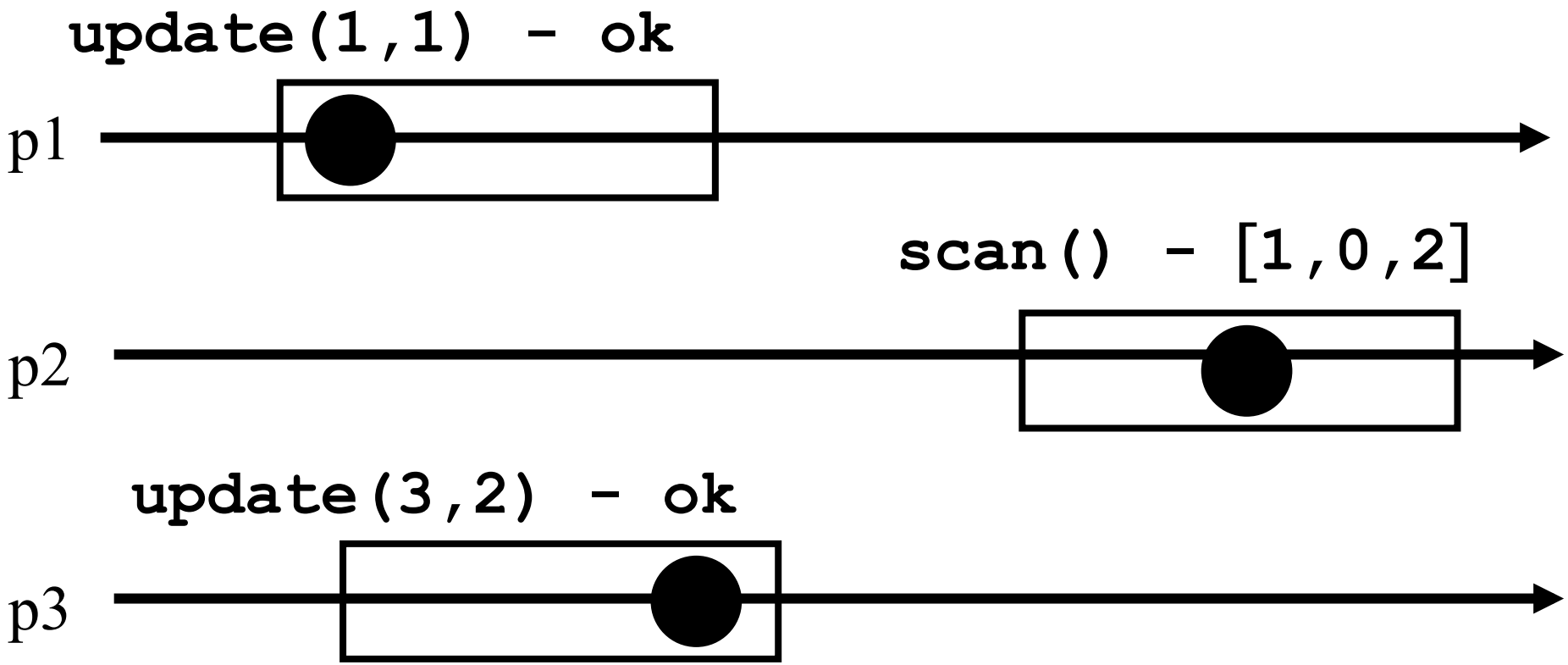
Less naive implementation

- 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)

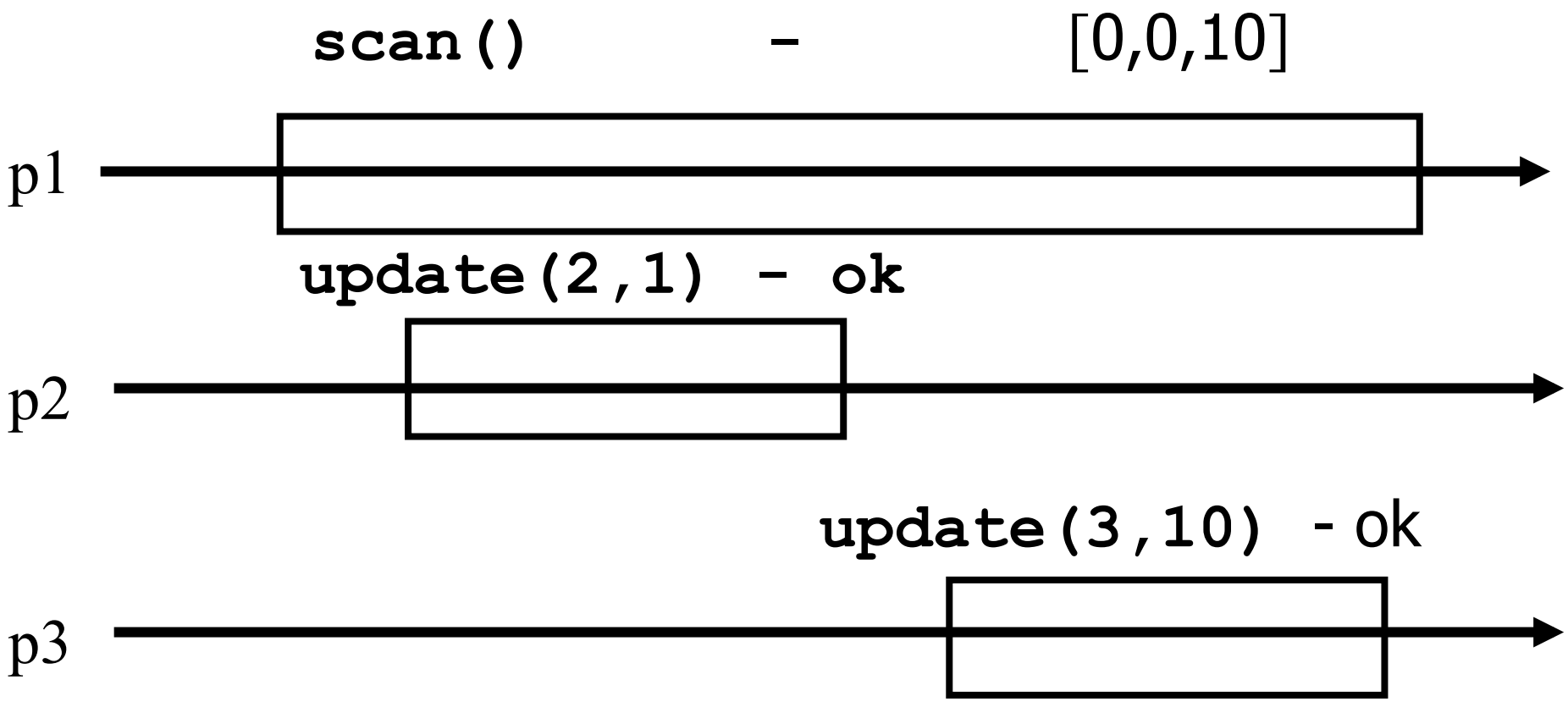
Atomic execution?



Atomic execution?



Atomic execution?



Non-atomic vs atomic snapshot

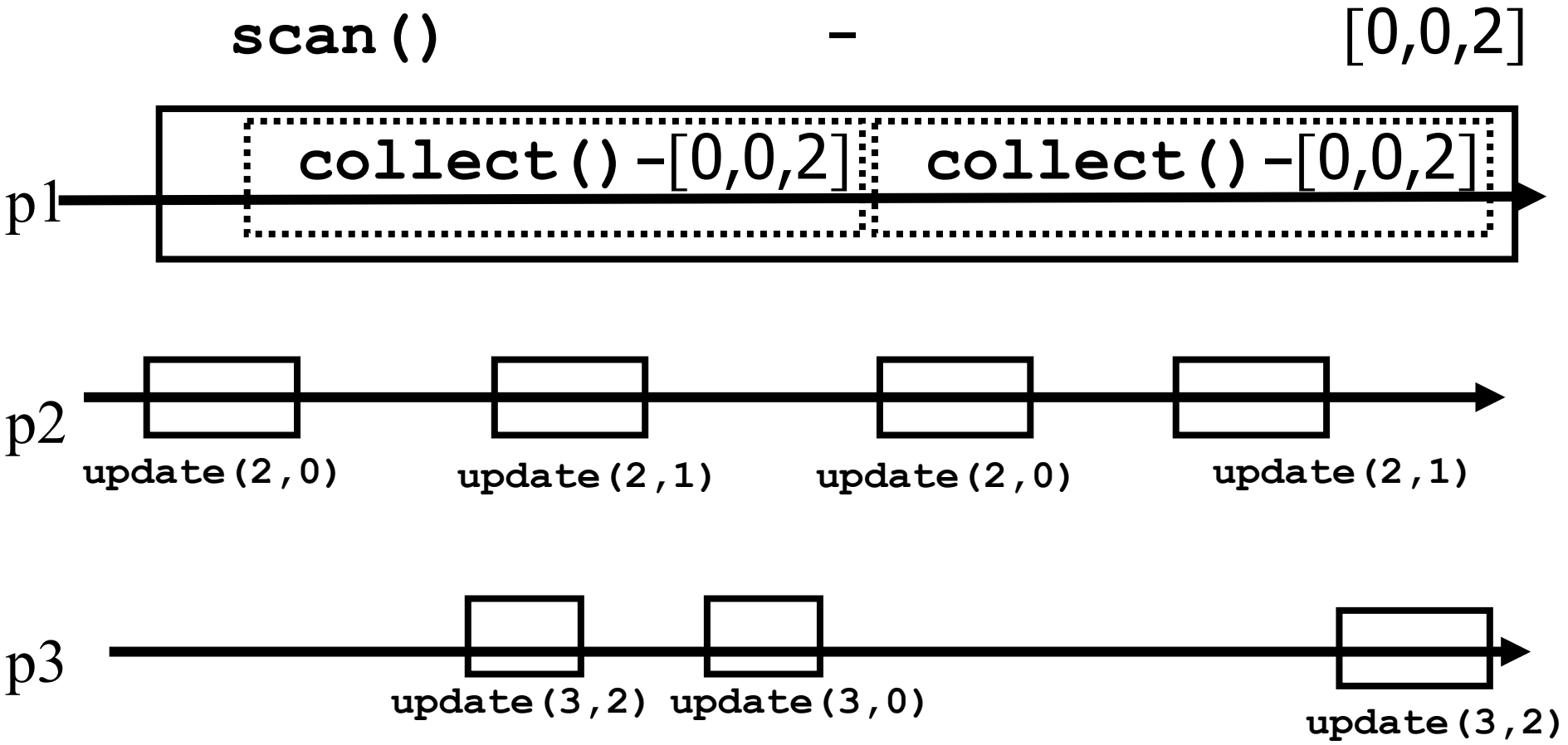
- What we implement here is some kind of ***regular*** snapshot:
  - A ***scan*** returns, for every index of the snapshot, the last written values or the value of any concurrent update
  - We call it ***collect***

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



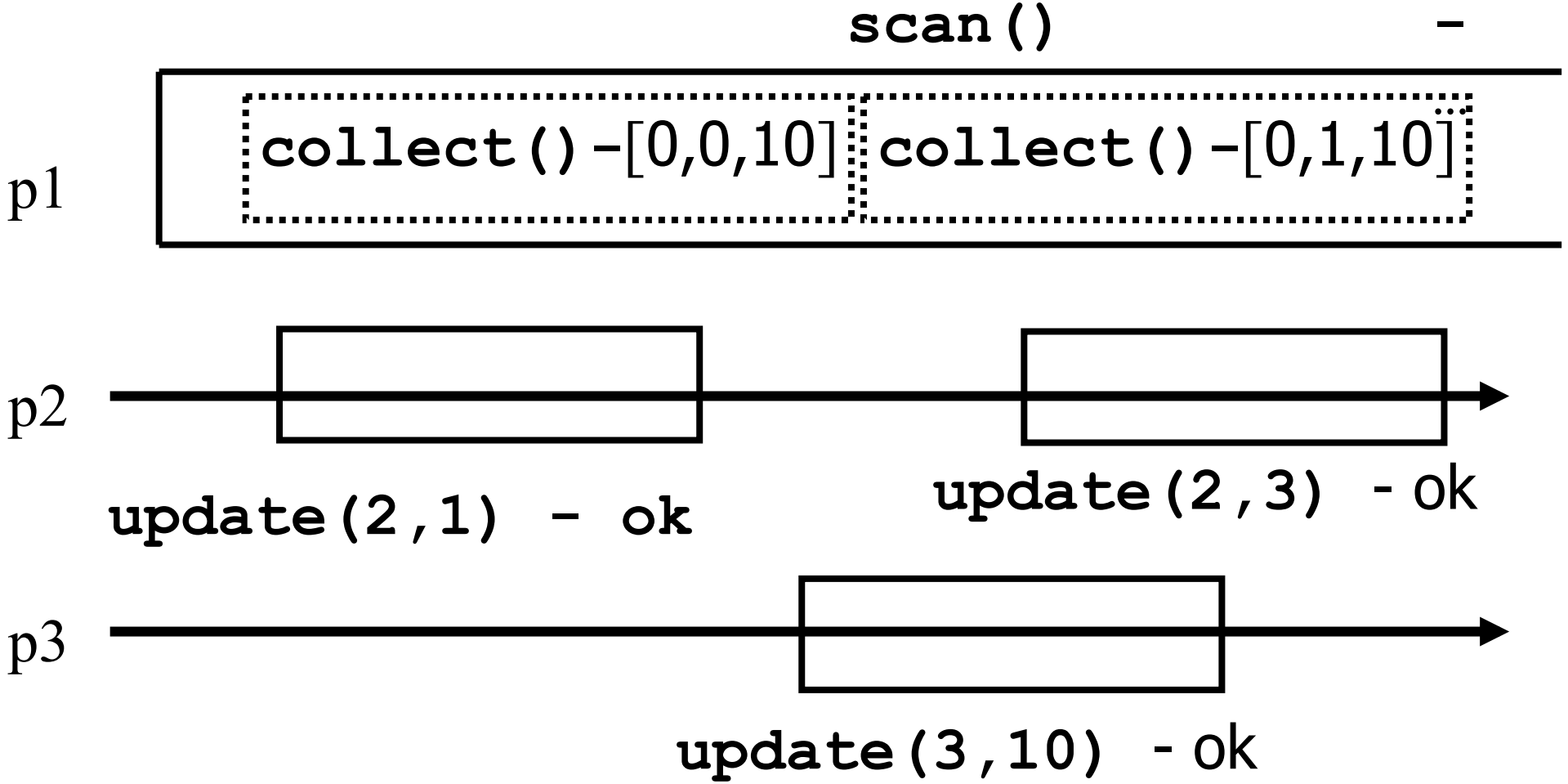
Enforcing atomicity

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

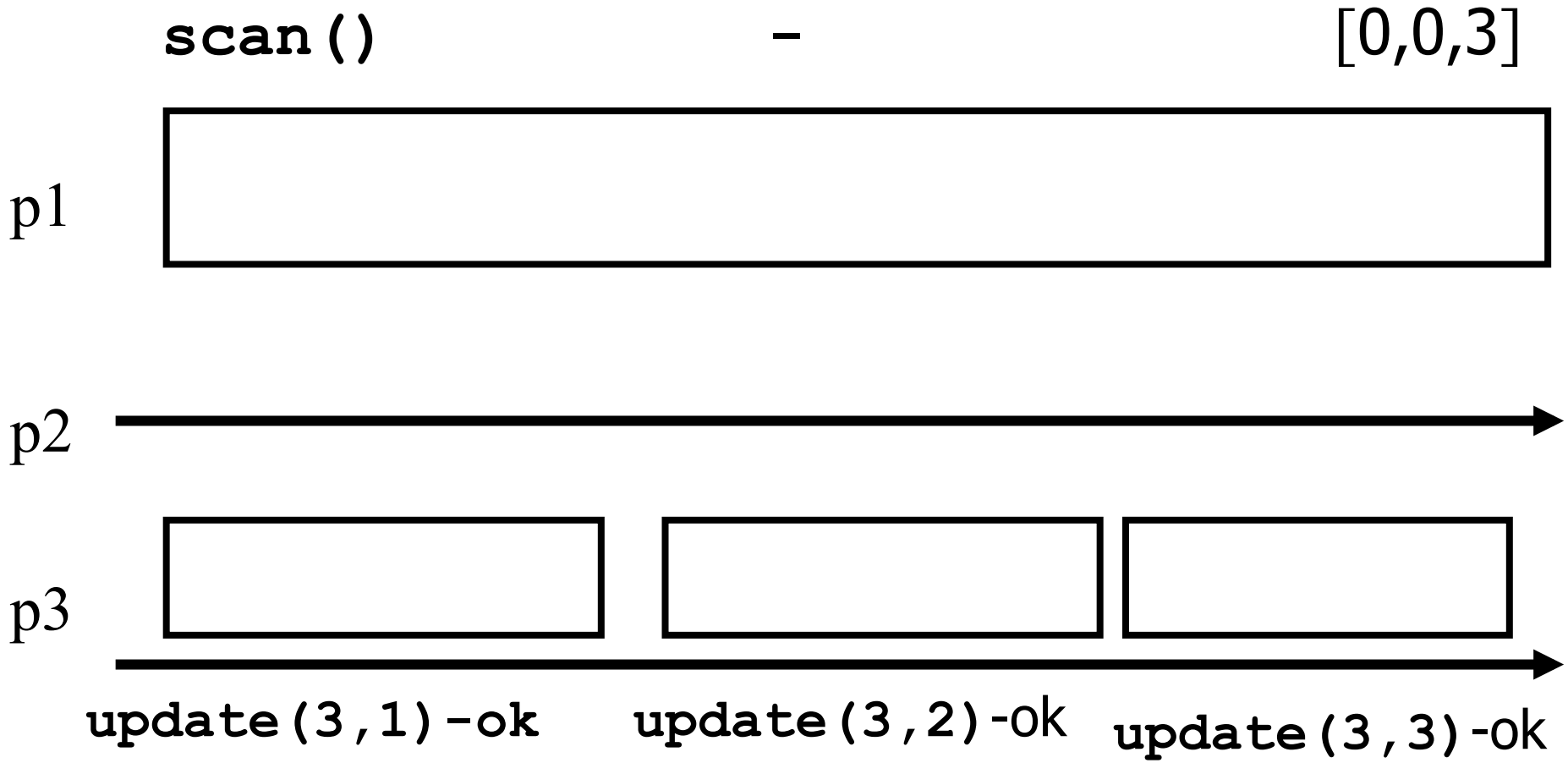
- ***update(i,v):***
  - ts := ts + 1;
  - Reg[i].write(v,ts,self.scan());
  - return(ok)

Snapshot implementation

- ***scan():***
  - t1 := self.collect(); t2:= t1
  - while(true) do
    - t3:= self.collect();
    - if (t3 = t2) then return (t3);
    - for j = 1 to N do
    - if(t3[j,2] ≥ t1[j,2]+2) then
      - return (t3[j,3])
    - t2 := t3

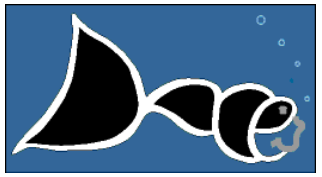
Return the first value in each cell in t3

Possible execution?



The Limitations of Registers

R. Guerraoui
Distributed Programming Laboratory

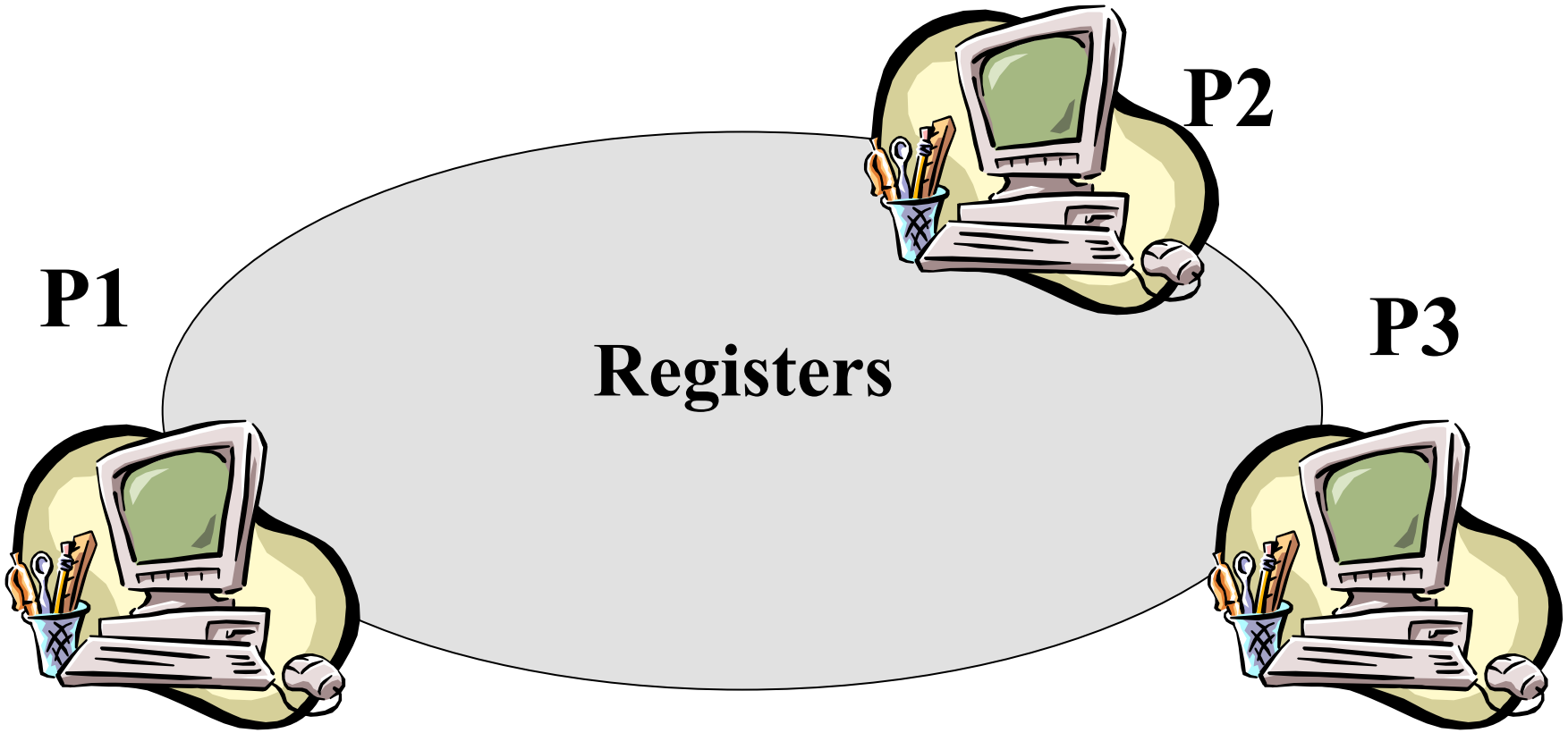


Registers

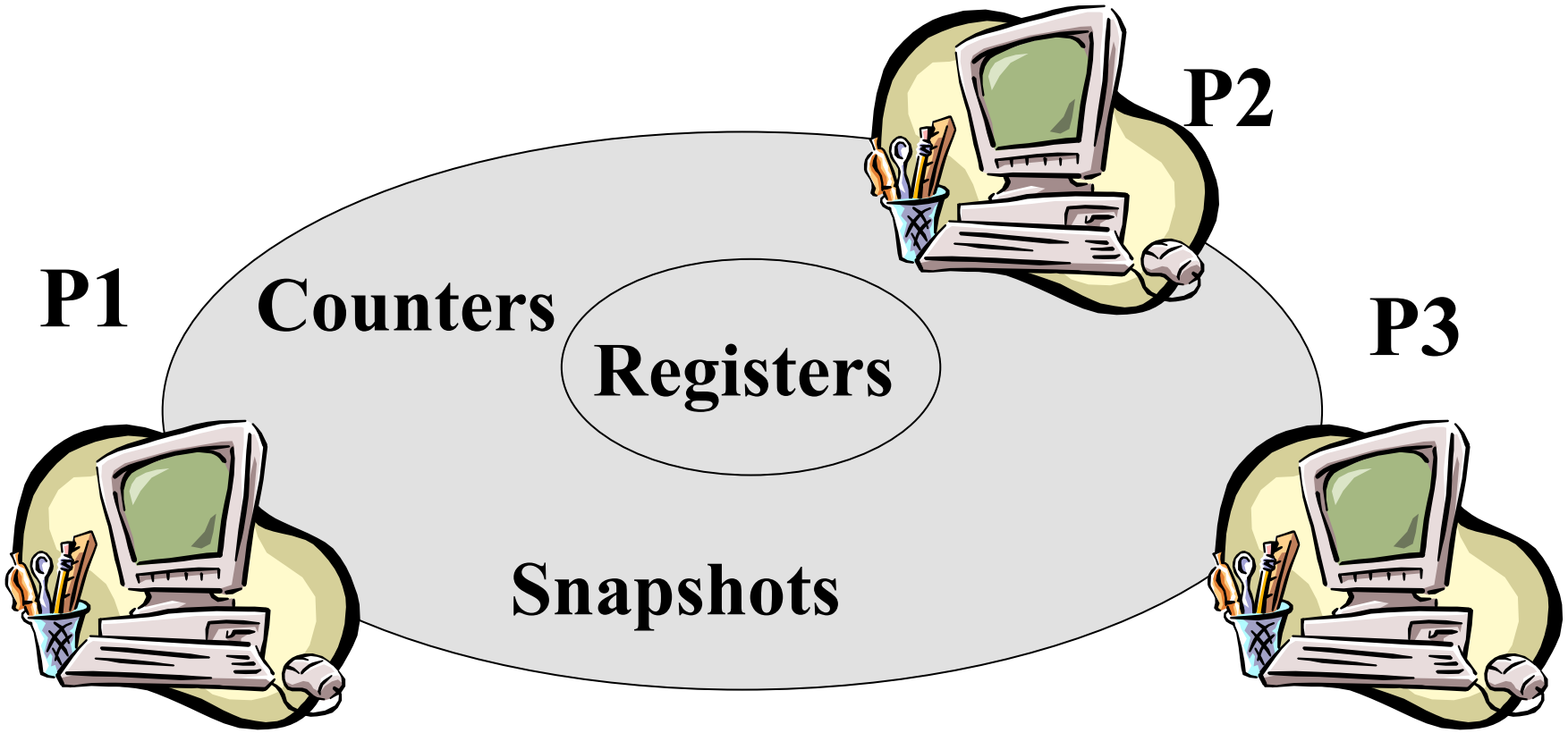
- ***Question 1:*** what objects can we implement with registers? ***Counters*** and ***snapshots*** (previous lecture)

- ***Question 2:*** what objects we cannot implement? (this lecture)

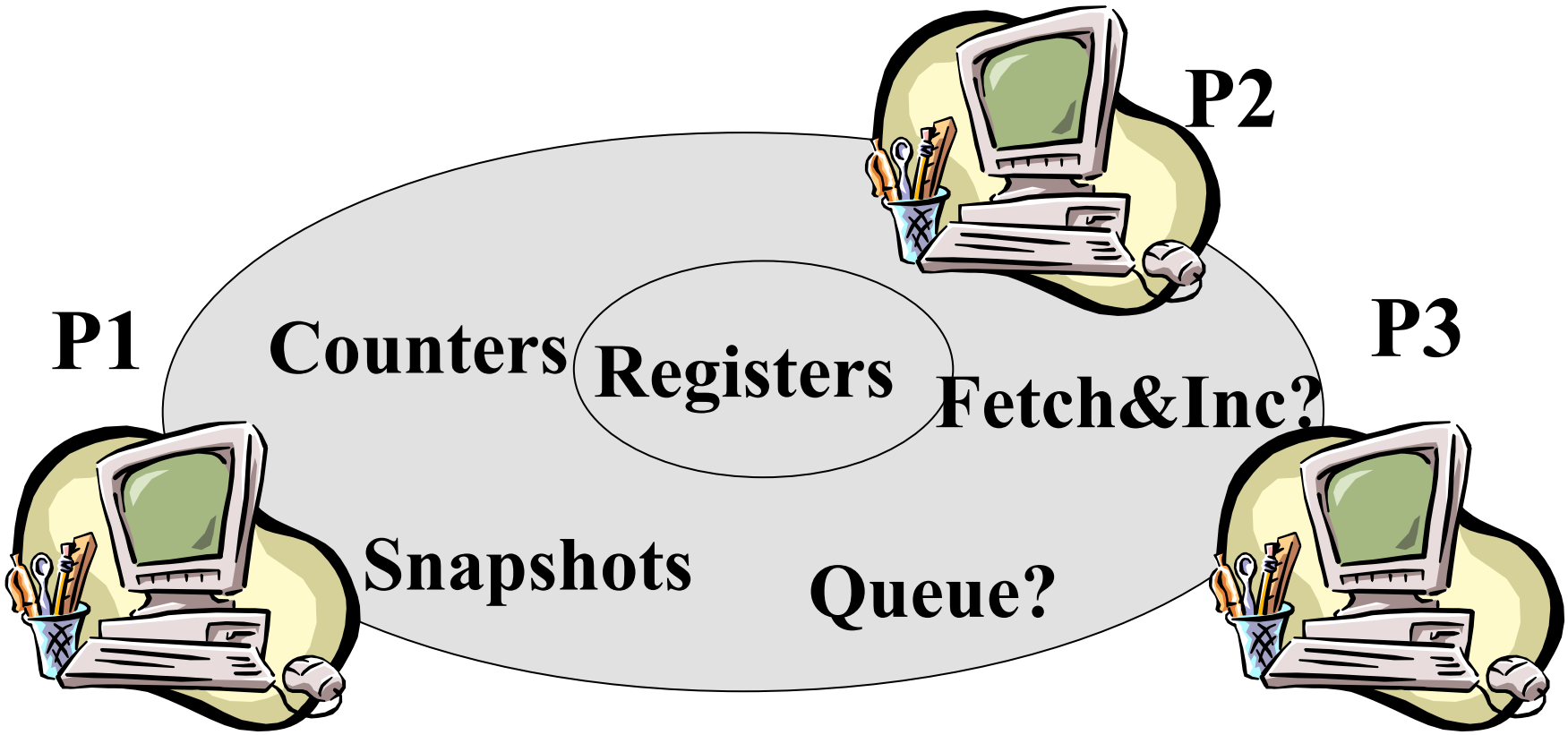
Shared memory model



Shared memory model



Shared memory model



Fetch&Inc

- **A counter that contains an integer**
- **Operation fetch&inc() increments the counter and returns the new value**

The consensus object

- One operation ***propose()*** which returns a value***.*** When a propose operation returns, we say that the process decides
- No two processes decide differently
- Every decided value is a proposed value

The consensus object

- ***Proposition:***
  - ✓ ***Consensus* can be implemented among two processes with *Fetch&Inc* and *registers***

- **Proof (algorithm): consider two processes p0 and p1 and two *registers* R0 and R1 and a *Fetch&Inc* C.**

2-Consensus with Fetch&Inc

- **Uses two registers R0 and R1, and a Fetch&Inc object C (with one fetch&inc() operation that returns its value)**
- **(NB. The value in C is initialized to 0)**

- **Process pI:**

- **propose(vI)**
- **RI.write(vI)**
- **val := C.fetch&inc()**
- **if(val = 1) then**
  - ✓ **return(vI)**
  - – **else return(R{1-I}.read())**

Impossibility [FLP85,LA87]

- ***Proposition:*** **there is no *asynchronous deterministic* algorithm that implements *consensus* among two processes using only *registers***

- ***Corollary:*** **there is no algorithm that implements *Fetch&Inc* among two processes using only *registers***

Queue

- **The queue is an object container with two operations: *enq()* and *deq()***

- **Can we implement a (atomic wait-free) *queue*?**

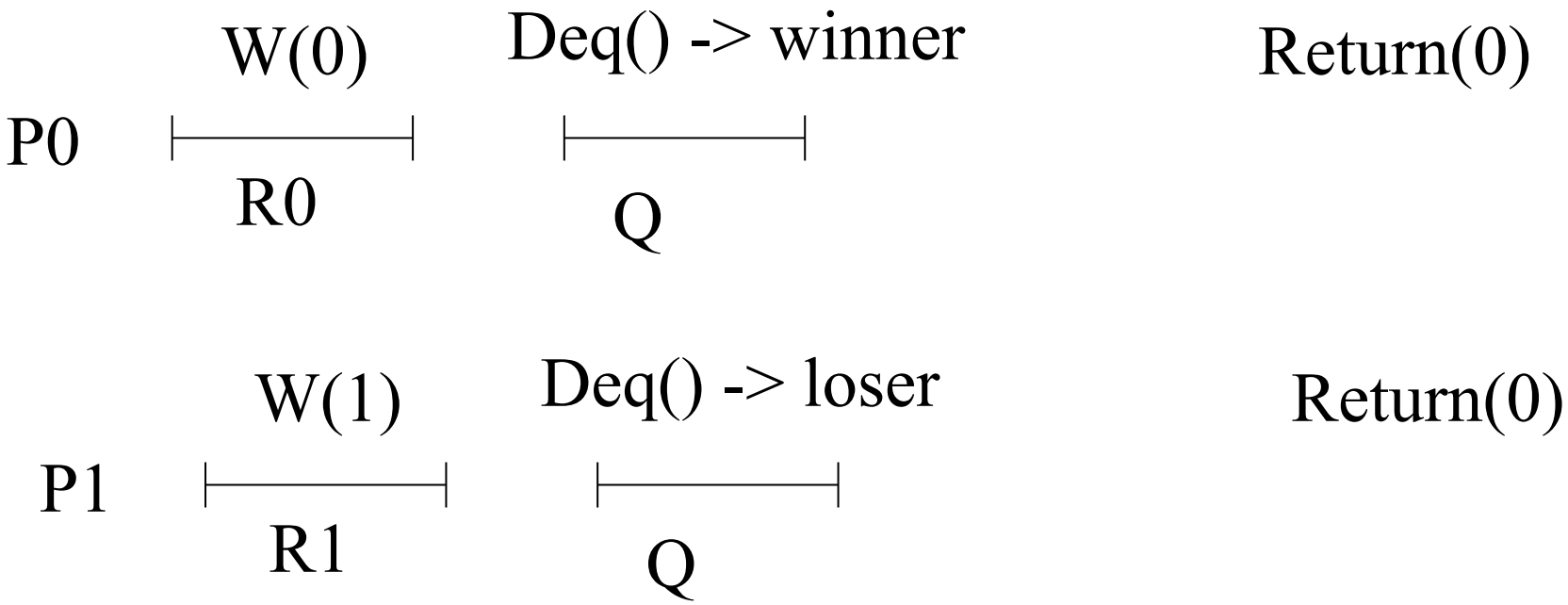
2-Consensus with queues

Uses two registers R0 and R1, and a queue Q

Q is initialized to {winner, loser}

Process pI:

```
propose(vI)
    RI.write(vI)
    item := Q.dequeue()
    if item = winner return(vI)
    return(R{1-I}.read())
```


W(0)
Deq() -> winner
Return(0)
P0
R0
Q
W(1)
Deq() -> loser
Return(0)
P1
R1
Q

Correctness

Proof (algorithm):

- (wait-freedom) by the assumption of a wait-free register and a wait-free queue plus the fact that the algorithm does not contain any wait statement
- (validity) If pI dequeues winner, it decides on its own proposed value. If pI dequeues loser, then the other process pJ dequeued winner before. By the algorithm, pJ has previously written its input value in RJ. Thus, pI decides on pJ's proposed value;
- (agreement) if the two processes decide, they decide on the value written in the same register.

More consensus implementations

- A ***Test&Set*** object maintains binary values x, init to 0, and y; it provides one operation: ***test&set()***
  - ✓ Sequential spec:
  - ✓ test&set() {y := x; x: = 1; return(y);}

- A ***Compare&Swap*** object maintains a value x, init to ⊥, and provides one operation: ***compare&swap(v,w);***
  - ✓ Sequential spec:
    - c&s(old,new) {if x = old then x := new; return(x)}

2-Consensus with Test&Set

- Uses two registers R0 and R1, and a Test&Set object T
-

- Process pI:

- **propose(vI)**
- **RI.write(vI)**
- **val := T.test&set()**
- **if(val = 0) then**
  - ✓ **return(vI)**
    else return(R{1-I}.read())

N-Consensus with C&S

- Uses a C&S object C
-
- Process pI:
-
- **propose(vI)**
- **val := C.c&s(⊥,vI)**
- **if(val = ⊥) then**
  - ✓ **return(vI)**
  - – **else return(val)**

Impossibility [FLP85,LA87]

- ***Proposition:*** there is no ***asynchronous deterministic*** algorithm that implements ***consensus*** among two processes using only ***registers***

- ***Corollary:*** there is no algorithm that implements a ***queue*** (***Fetch&Inc,…***) among two processes using only ***registers***

Registers

- ***Question 1:*** what objects can we implement with registers? ***Counters*** and ***snapshots*** (previous lecture)

- ***Question 2:*** what objects we cannot implement? All objects that (together with ***registers***) can implement ***consensus*** (this lecture)

Impossibility (Proof)

- ***Proposition:*** there is no algorithm that implements ***consensus*** among two processes using only ***registers***

- Proof (by contradiction): consider two processes p0 and p1 and any number of ***registers***, R1..Rk..

  Assume that a consensus algorithm A for p0 and p1 exists.

Elements of the model

- A ***configuration*** is a global state of the distributed system

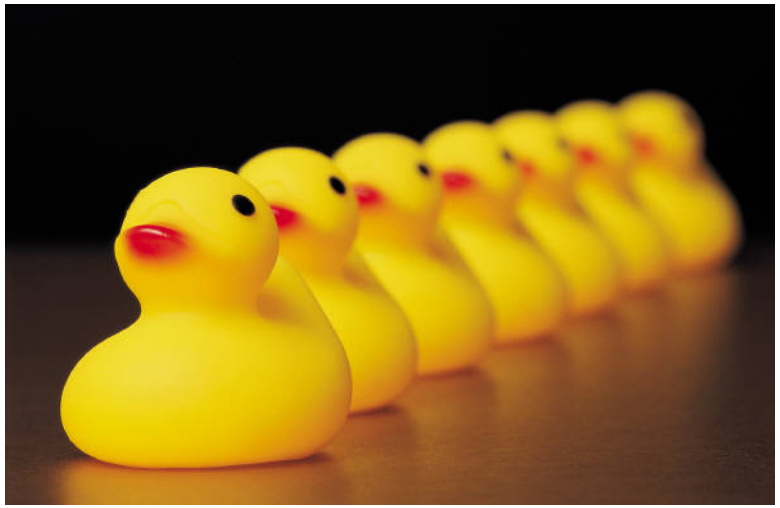
- A new configuration is obtained by executing a ***step*** on a previous configuration: the step is the unit of execution

Elements of the model

- **The adversary decides which process executes the next step and the algorithm deterministically decides the next configuration based on the current one**

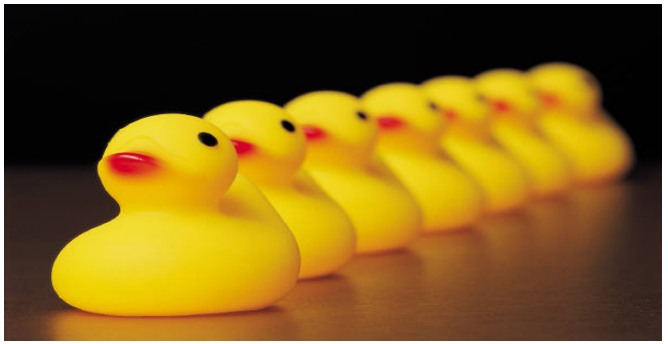
What is distributed computing?
A game

A game between an adversary and a set of processes



The adversary decides which process goes next

The processes take steps



Elements of the model

- The adversary decides which process executes the next step and the algorithm deterministically decides the next configuration based on the current one

Elements of the model

- ***Schedule:*** a sequence of steps represented by process ids
- The schedule is chosen by the system

- An asynchronous system is one with no constraint on the schedules: any sequence of process ids is a schedule

Consensus

- The algorithm must ensure that *agreement* and *validity* are satisfied in every schedule
- Every process that executes an infinite number of steps eventually decides

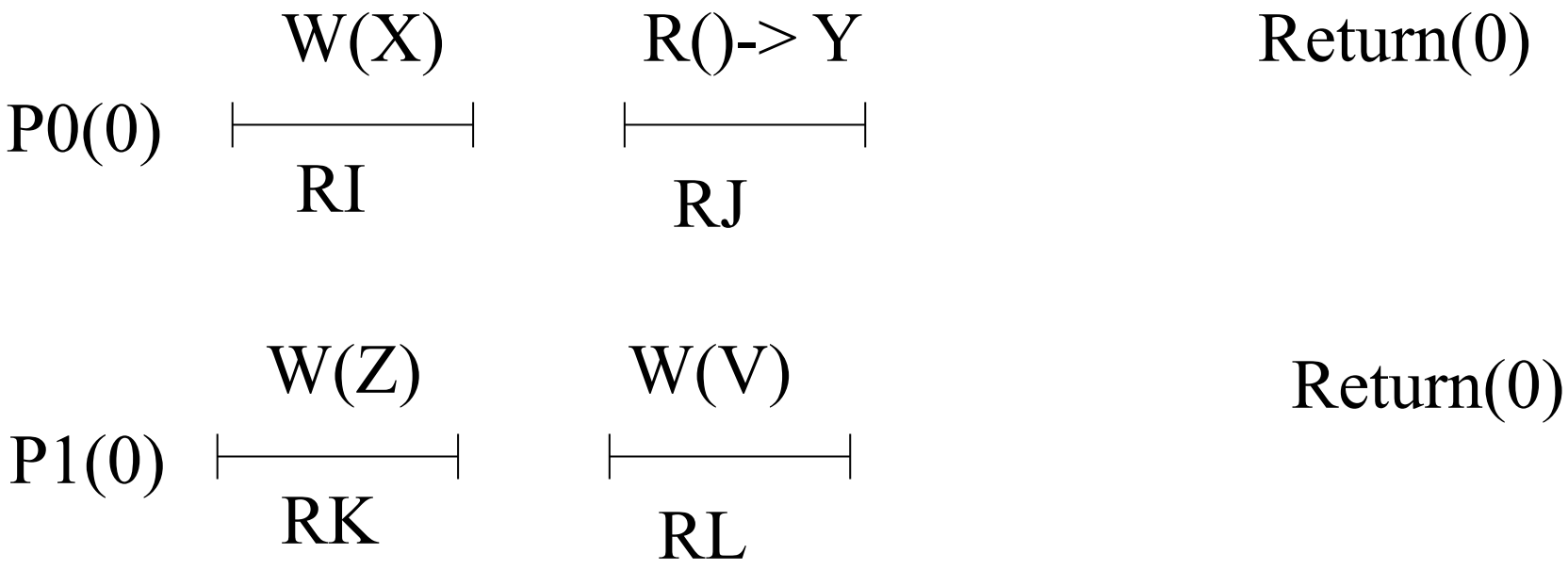
Impossibility (elements)

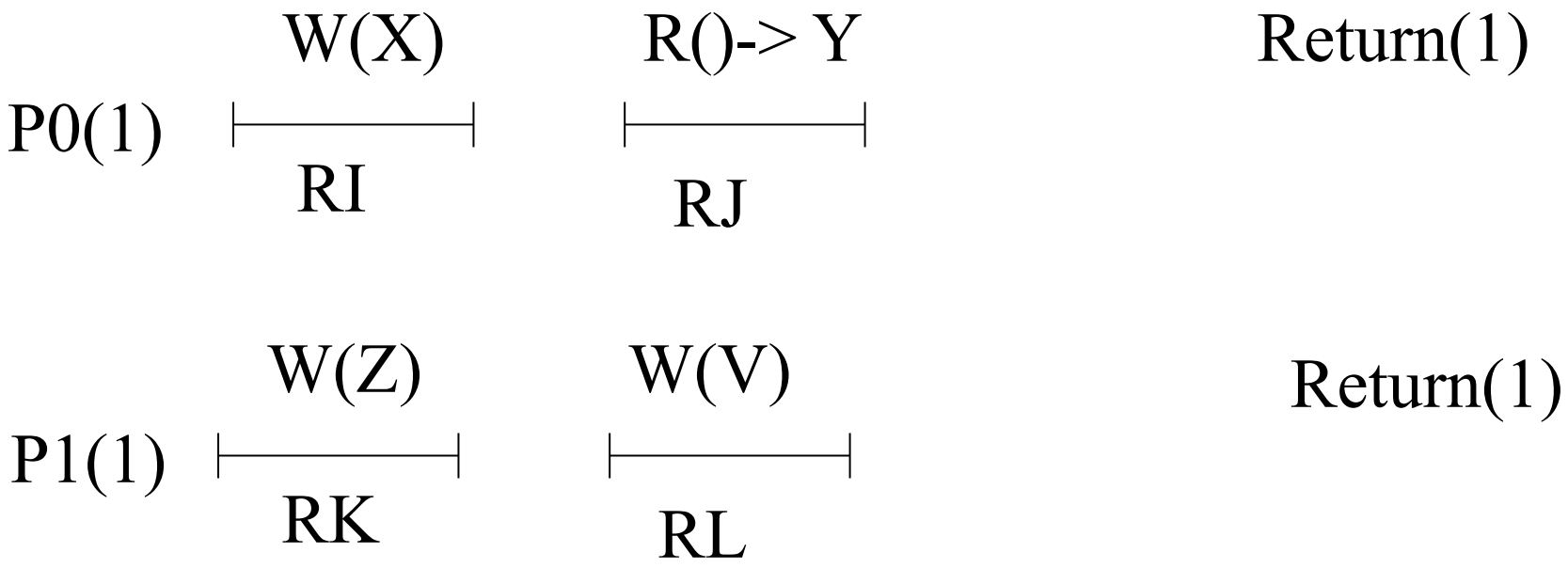
- (1) a (initial) ***configuration*** C is a set of (initial) values of p0 and p1 together with the values of the registers: R1..Rk,..;
- (2) a ***step*** is an elementary action executed by some process pI: it consists in reading or writing a value in a register and changing pI's state according to the algorithm A;
- (3) a ***schedule*** S is a sequence of steps; S(C) denotes the configuration that results from applying S to C.

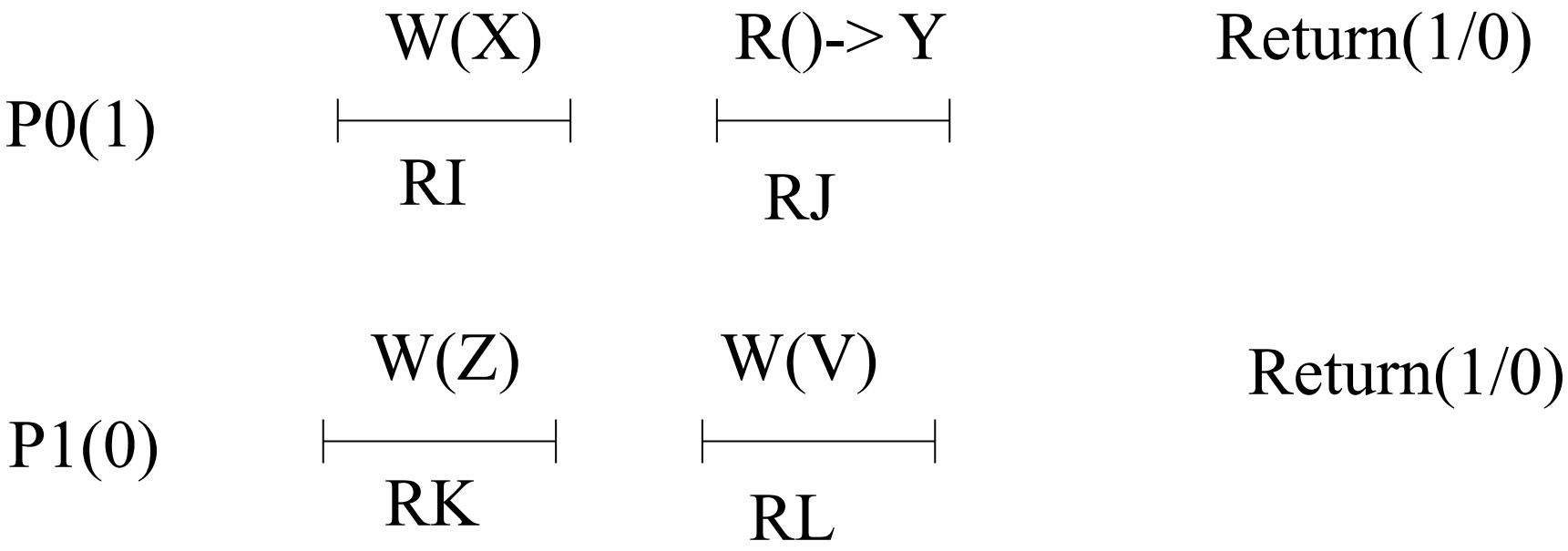
Impossibility (elements)

- Consider u to be 0 or 1; a configuration C is ***u-valent*** if, starting from C, no matter how the processes behave, no decision other than u is possible

- We say that the configuration is ***univalent***. Otherwise, the configuration is called ***bivalent***


W(X)
R()-> Y
Return(0)
P0(0)
RI
RJ
W(Z)
W(V)
Return(0)
P1(0)
RK
RL


W(X)
R()-> Y
Return(1)
P0(1)
RI
RJ
W(Z)
W(V)
Return(1)
P1(1)
RK
RL


W(X)
R()-> Y
Return(1/0)
P0(1)
RI
RJ
W(Z)
W(V)
Return(1/0)
P1(0)
RK
RL

Impossibility (structure)

- ***Lemma 1:*** there is at least one initial ***bivalent*** configuration

- ***Lemma 2:*** given any bivalent configuration C, there is an ***arbitrarily long schedule*** S(C) that leads to another bivalent configuration

The conclusion

- Lemmas 1 and 2 imply that there is a configuration C and an *infinite* schedule S such that, for any prefix S' of S, S'(C) is bivalent.

- In infinite schedule S, at least one process executes an infinite number of steps and does not decide

- A contradiction with the assumption that A implements consensus.

Lemma 1

The initial configuration C(0,1) is bivalent

Proof: consider C(0,0) and p1 not taking any step: p0 decides 0; p0 cannot distinguish C(0,0) from C(0,1) and can hence decides 0 starting from C(0,1); similarly, if we consider C(1,1) and p0 not taking any step, p1 eventually decides 1; p1 cannot distinguish C(1,1) from C(0,1) and can hence decides 1 starting from C(0,1). Hence the bivalency.

Lemma 2

Given any bivalent configuration C, there is an arbitrarily long schedule S such that S(C) is bivalent

Proof (by contradiction): let S be the schedule with the maximal length such as D= S(C) is bivalent; p0(D) and p1(D) are both univalent: one of them is 0-valent (say p0(D)) and the other is 1-valent (say p1(D))

Lemma 2

- Proof (cont'd): To go from D to p0(D) (vs p1(D)) p0 (vs p1) accesses a register; the register must be the same in both cases; otherwise p1(p0(D)) is the same as p0(p1(D)): in contradiction with the very fact that p0(D) is 0-valent whereas p1(D) is 1-valent

Lemma 2

- Proof (cont'd): To go from D to p0(D), p0 cannot read R; otherwise R has the same state in D and in p0(D) ; in this case, the registers and p1 have the same state in p1(p0(D)) and p1(D); if p1 is the only one executing steps, then p1 eventually decides 1 in both cases: a contradiction with the fact that p0(D) is 0-valent; the same argument applies to show that p1 cannot read R to go from D to p1(D)

  Thus both p0 and p1 write in R to go from D to p0(D) (resp., p1(D)). But then p0(p1(D))= p0(D) (resp. p1(p0(D))= p1(D)) --- a contradiction.

The conclusion (bis)

Lemmas 1 and 2 imply that there is a configuration C and an *infinite* schedule S such that, for any prefix S' of S, S'(C) is bivalent.

In infinite schedule S, at least one process executes an infinite number of steps and does not decide

A contradiction with the assumption that A implements consensus.