Shared Memory Algorithms (Overview)

Prof R. Guerraoui
Assistant M. Kapalka
Distributed Programming Laboratory

In short

This course introduces a theory of robust concurrent computing

Major chip manufacturers have recently announced what is perceived as a major paradigm shift in computing:

Multiprocessors vs faster processors

May be Moore was wrong…

The clock speed of a processor cannot be increased without overheating

But

More and more processors can fit in the same space

Speed will be achieved by having several processors work on independent parts of a task

But

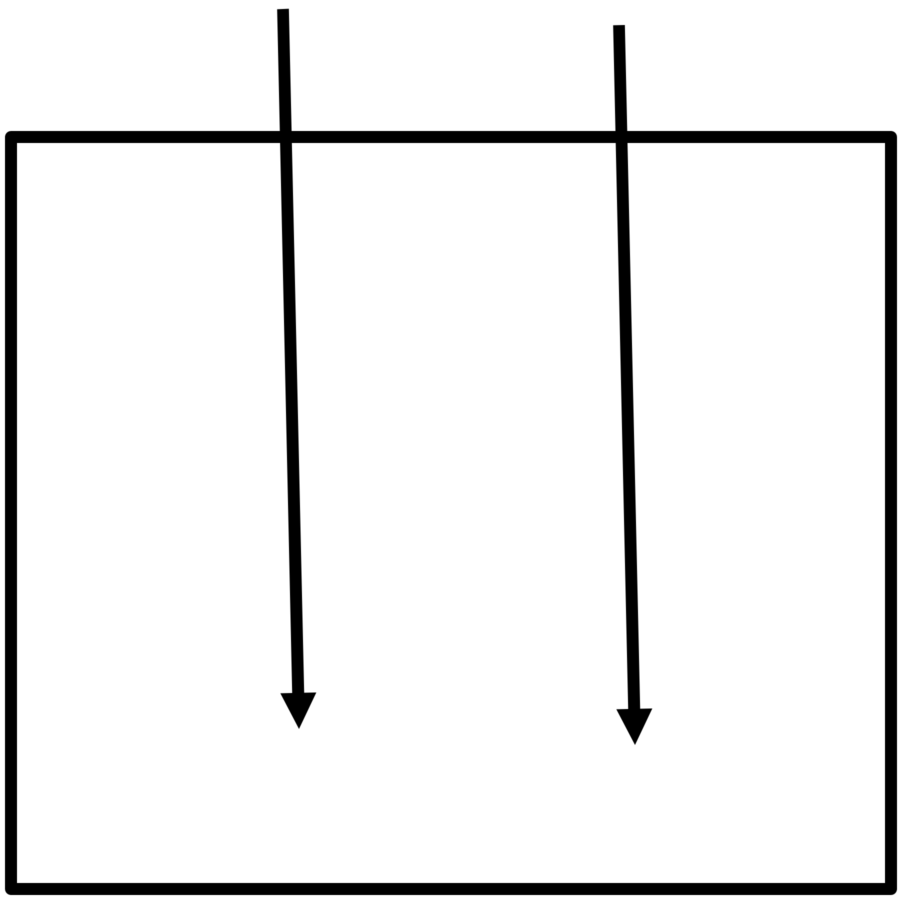
the processors would occasionally need to pause and synchronize

Why synchronize?

But

If the task is indeed common, then pure parallelism is usually impossible and, at best, inefficient

Concurrent processes



Shared object

Concurrent computing for the masses

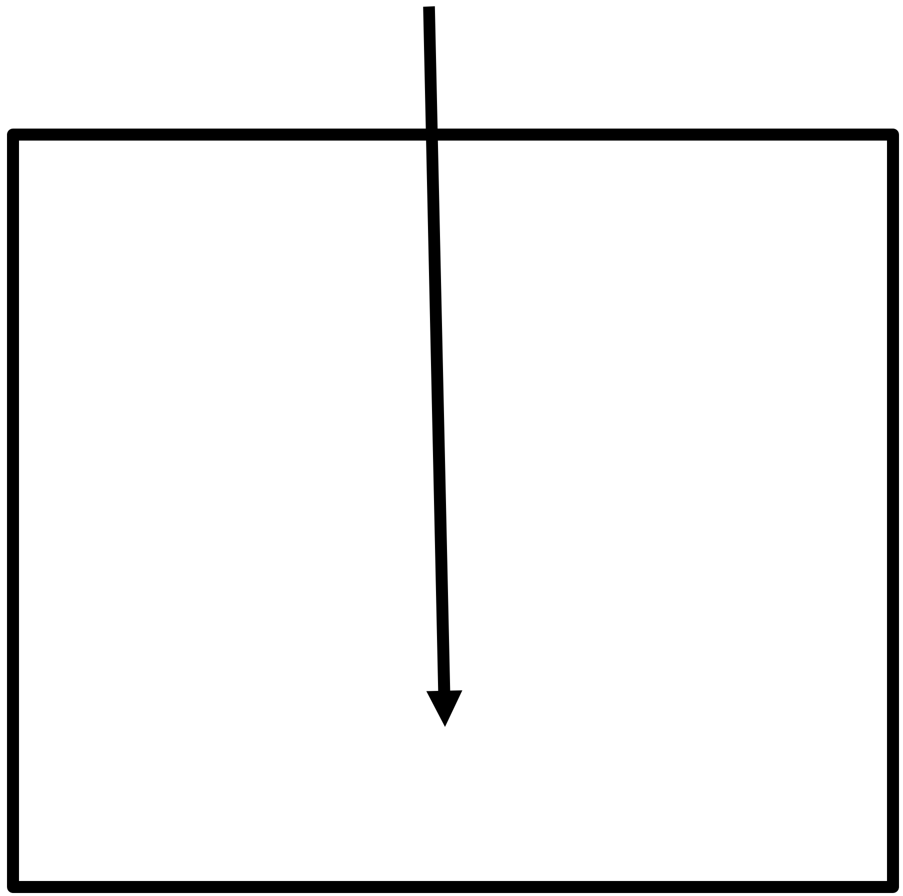
- *Forking processes might become more frequent*
- *But*
- *Concurrent accesses to shared objects might become more problematic*

Locking (mutual exclusion)

- ***Difficult:*** 50% of the bugs reported in Java come from the use of « synchronized »

- ***Fragile:*** a process holding a lock prevents all others from progressing

Locked object

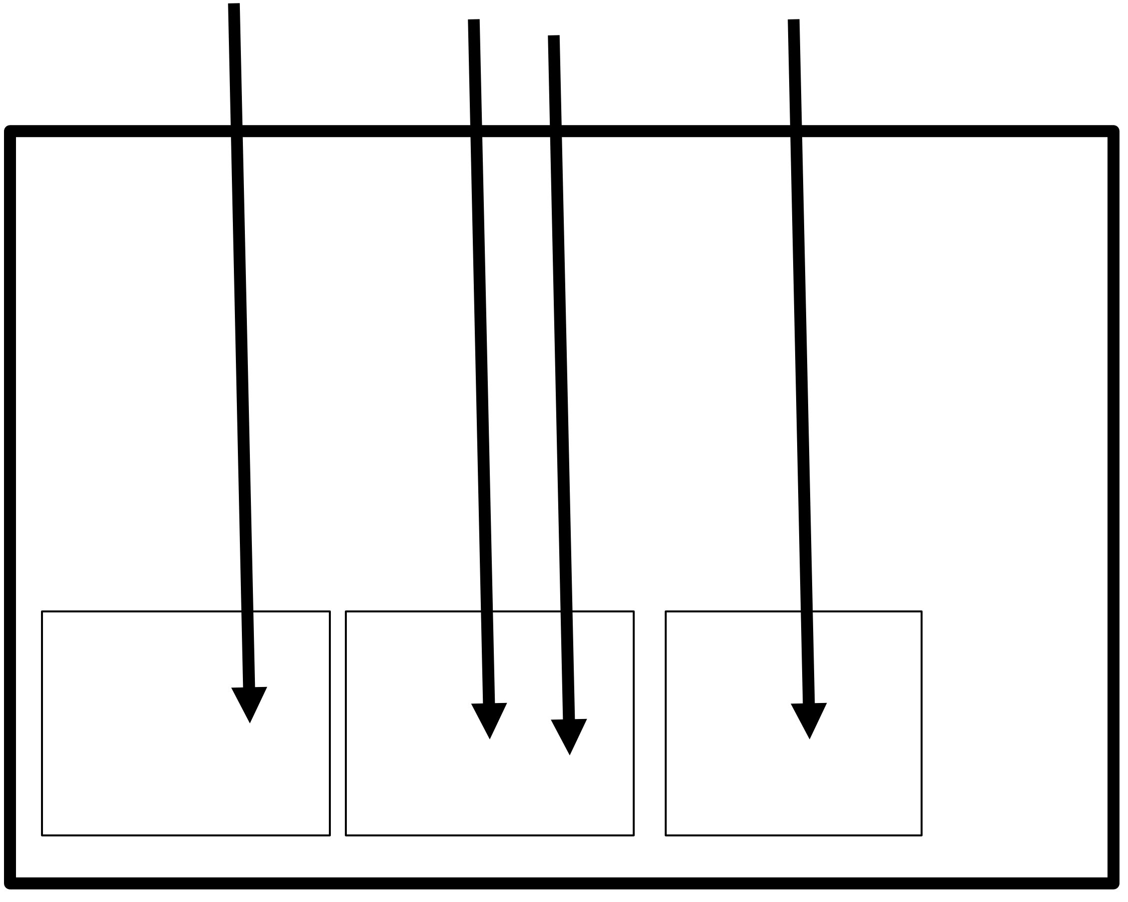


One process at a time

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

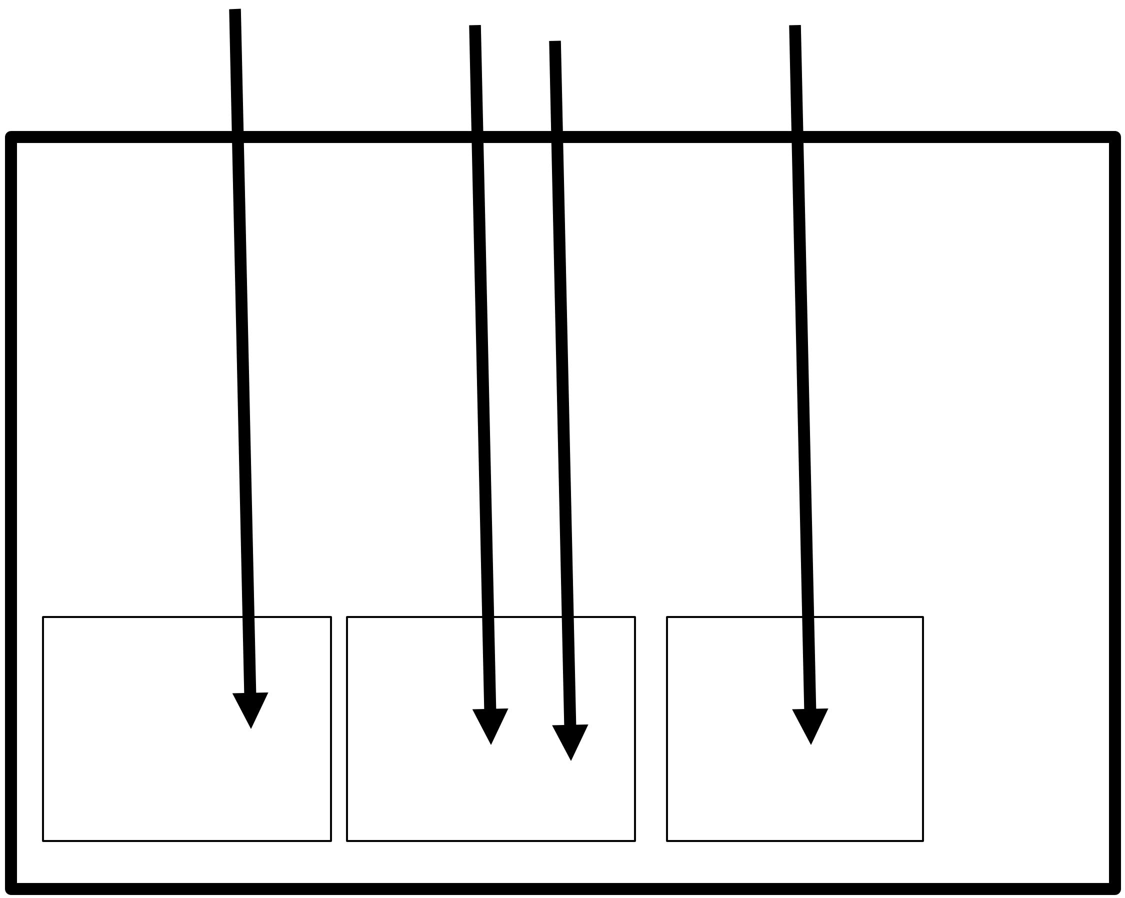
- ***Wait-freedom:*** every process that invokes an operation eventually returns from the invocation (robust … unlike locking)

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

In short

This course shows how to
wait-free implement high-level
atomic objects out of more
primitive base objects

Concurrent processes



Shared object

This course

- *Theoretical*
- *No specific theoretic background*
- *Written exam on Feb 6th*

Roadmap

- *Model*
  - *Processes and objects*
  - *Atomicity and wait-freedom*
- *Examples*
- *Content*

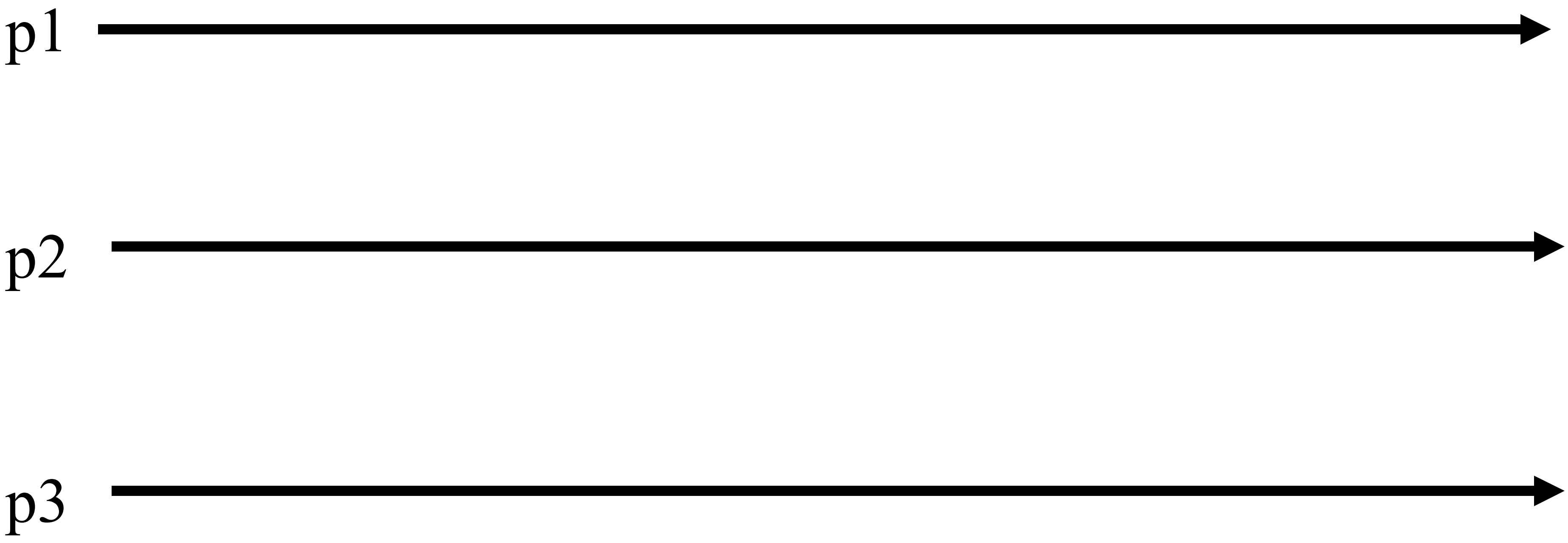
Processes

- 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 explicitely stated otherwise)

Processes

- Processes are ***sequential*** units of computations
- Unless explicitely stated otherwise, we make no assumption on process (relative) speed

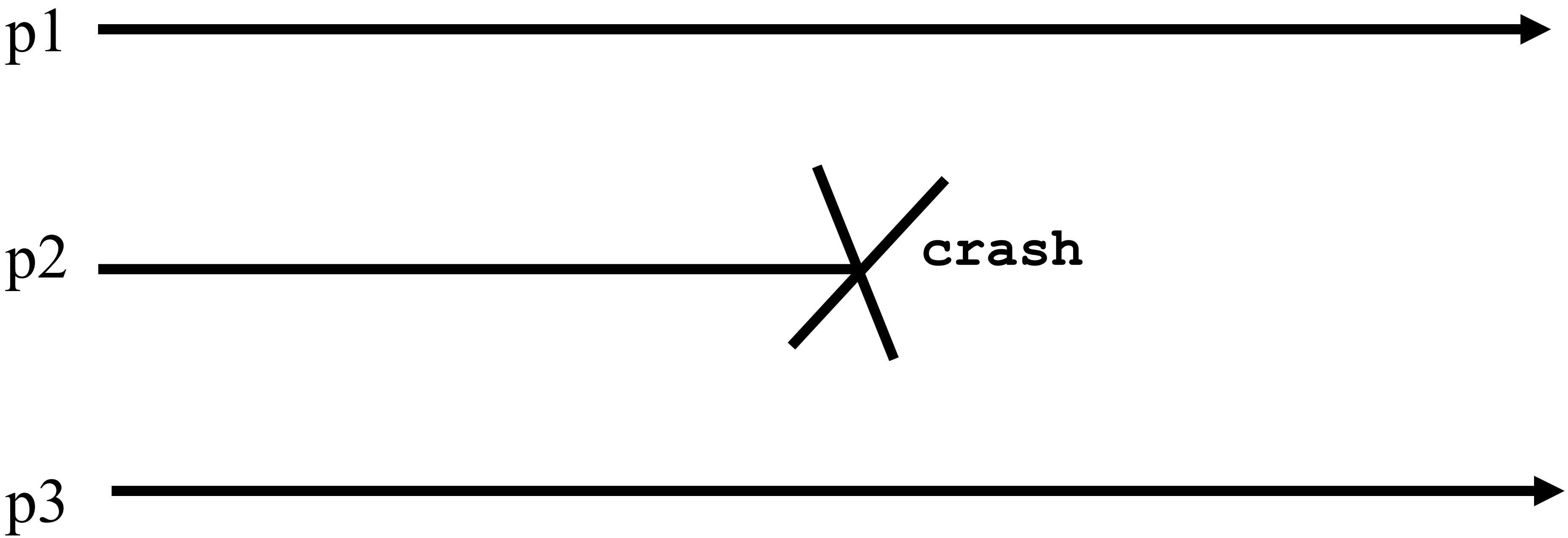
Processes



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)

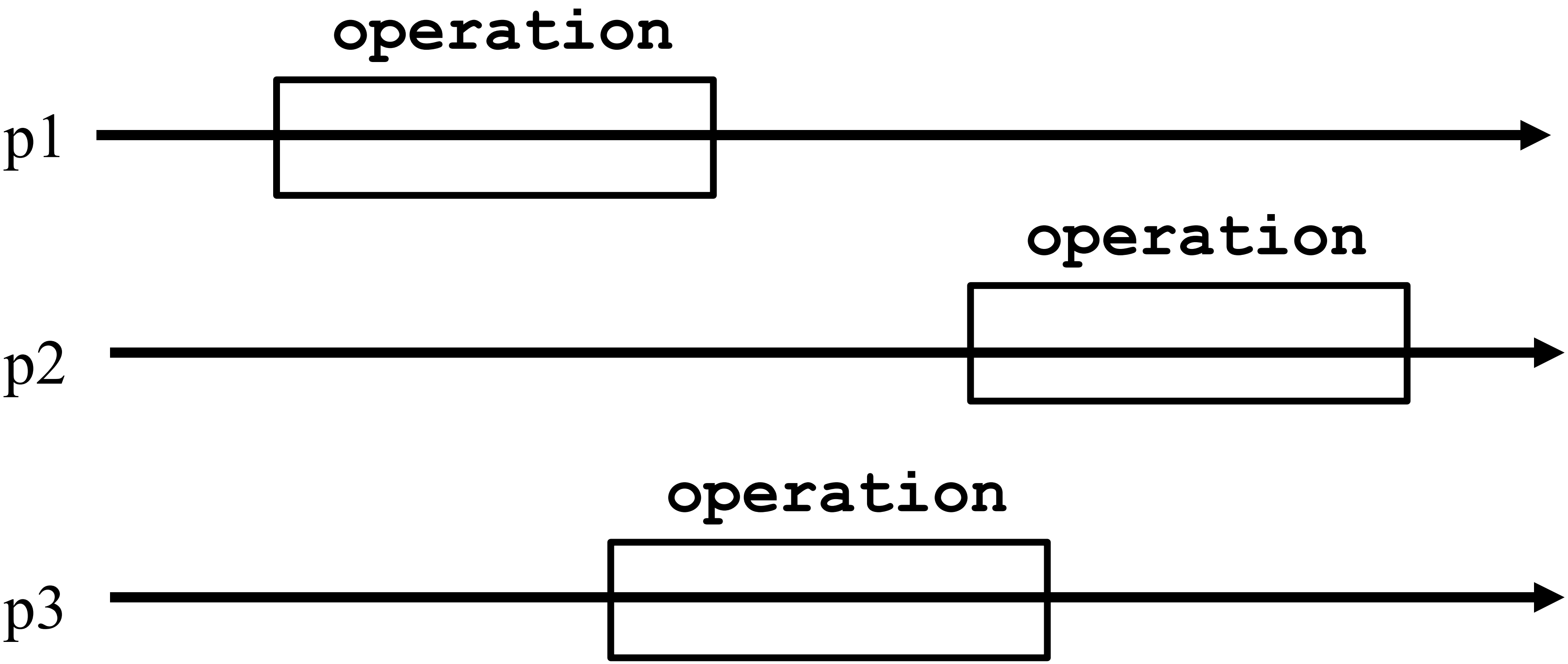
Processes



On objects and processes

- Processes execute local computation or access shared objects through their ***operations***
- Every operation is expected to return a reply

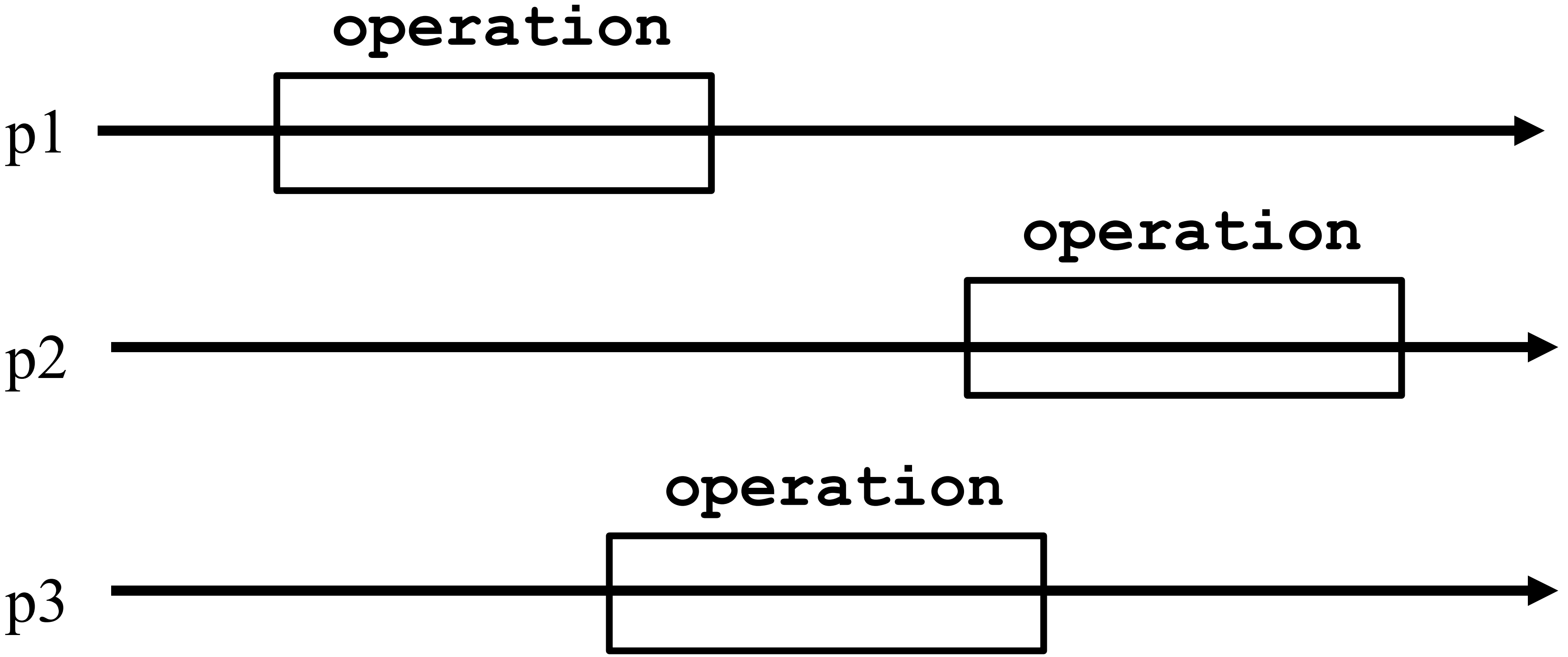
Processes



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

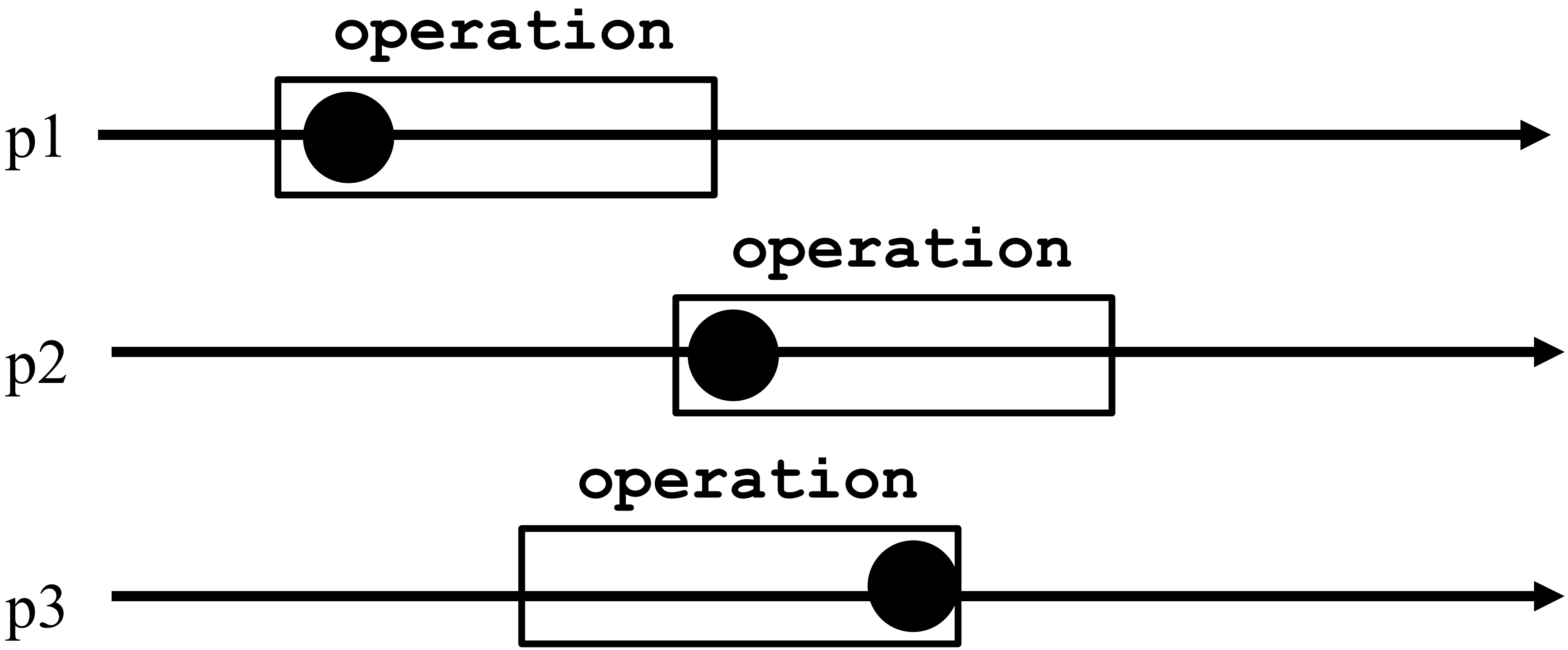
Processes



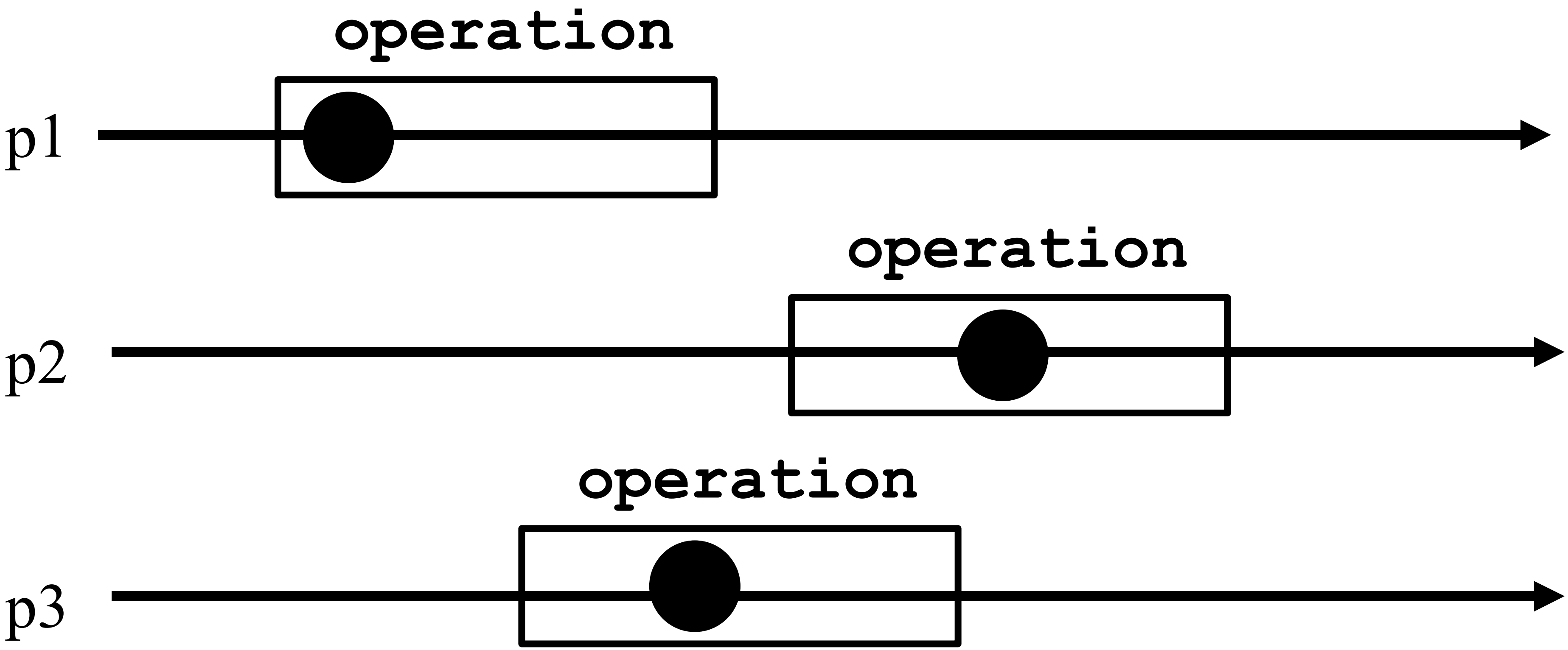
Atomicity

- We mainly focus in this course on how to implement ***atomic*** objects
- ***Atomicity*** means that every operation appears to execute at some indivisible point in time (called linearization point) between the invocation and reply time events

Atomicity



Atomicity

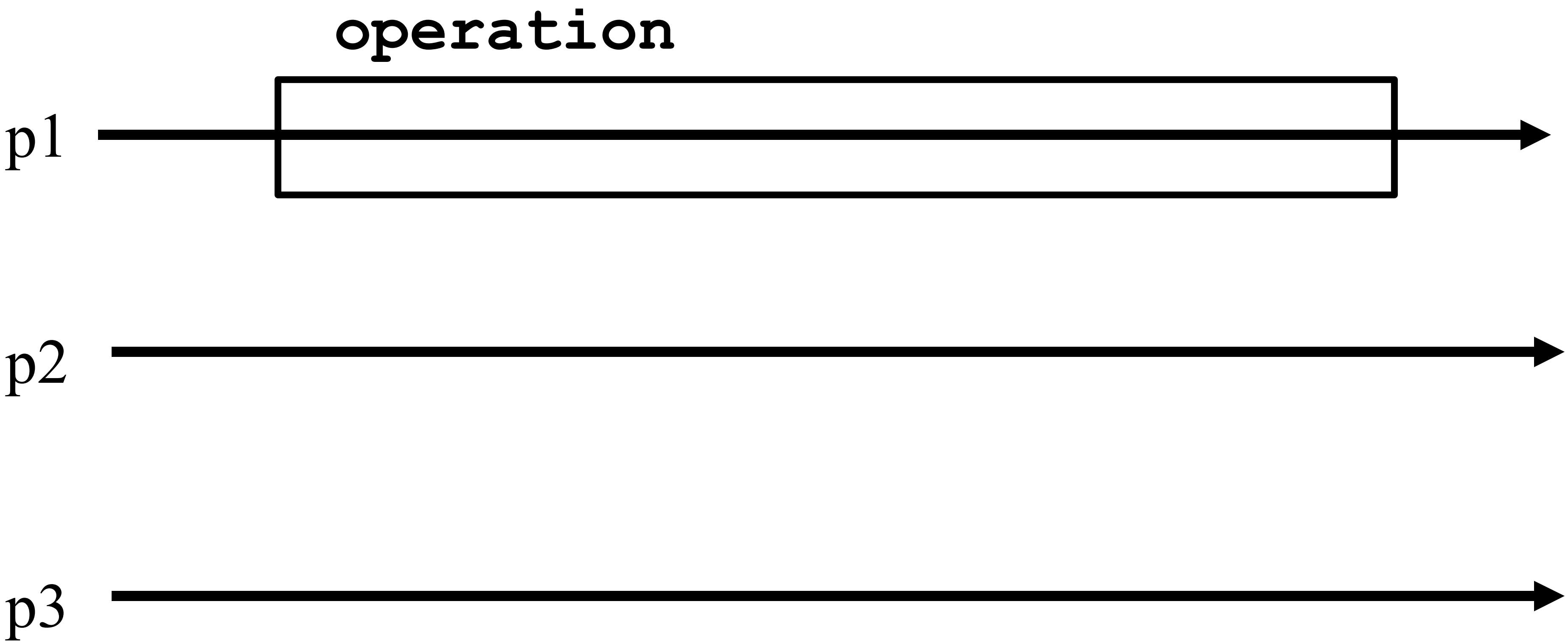


Wait-freedom

- We mainly focus in this course on ***wait-free*** implementations

- An implementation is wait-free if any correct process that invokes an operation eventually gets a reply, no matter what happens to the other processes (crash or very slow)

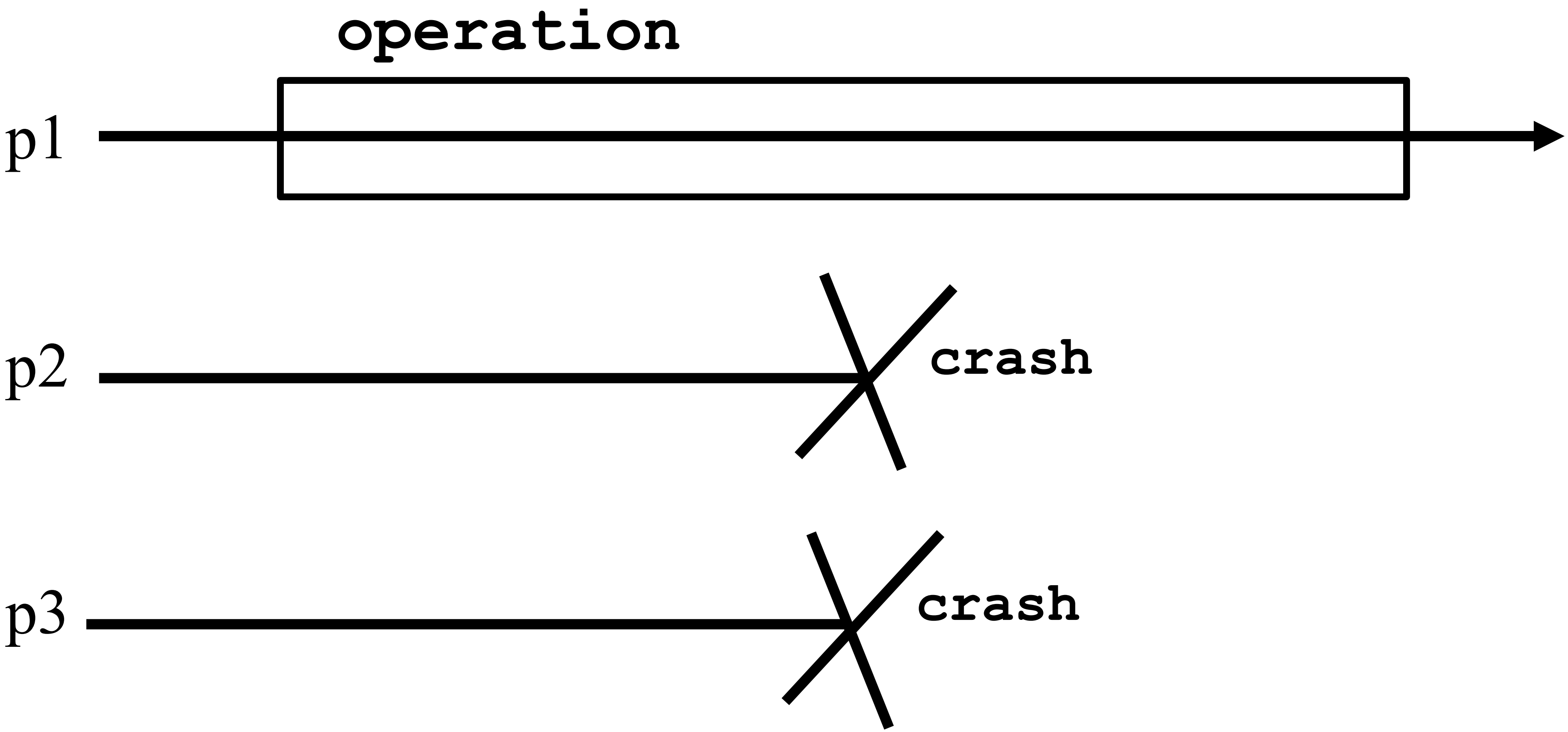
Wait-freedom



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



Roadmap

- *Model*
  - *Processes and objects*
  - *Atomicity and wait-freedom*
- *Examples*
- *Content*

Motivation

- Most synchronization primitives (problems) can be precisely expressed as atomic objects (implementations)
- Studying how to ensure robust synchronization boils down to studying wait-free atomic object implementations

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

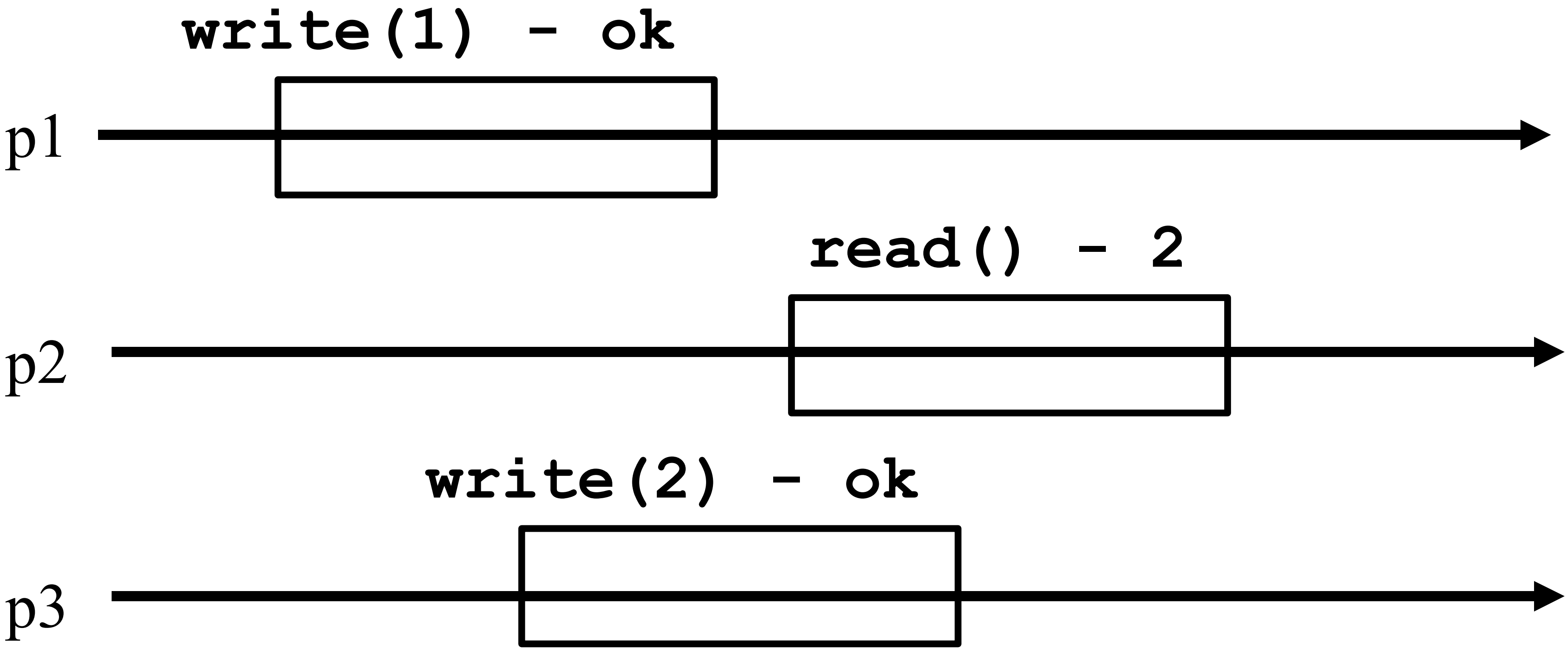
- A ***register*** has two operations: ***read()*** and ***write()***

- 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)

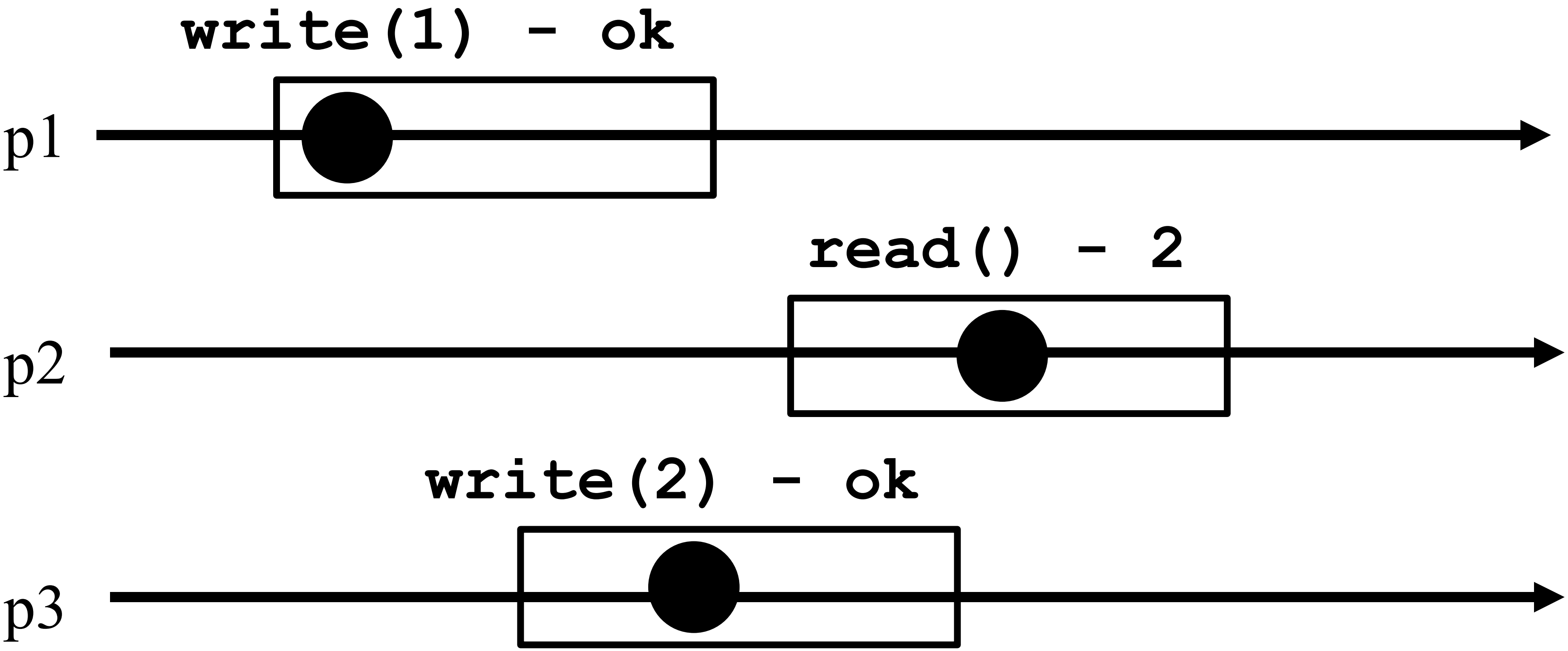
Sequential specification

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

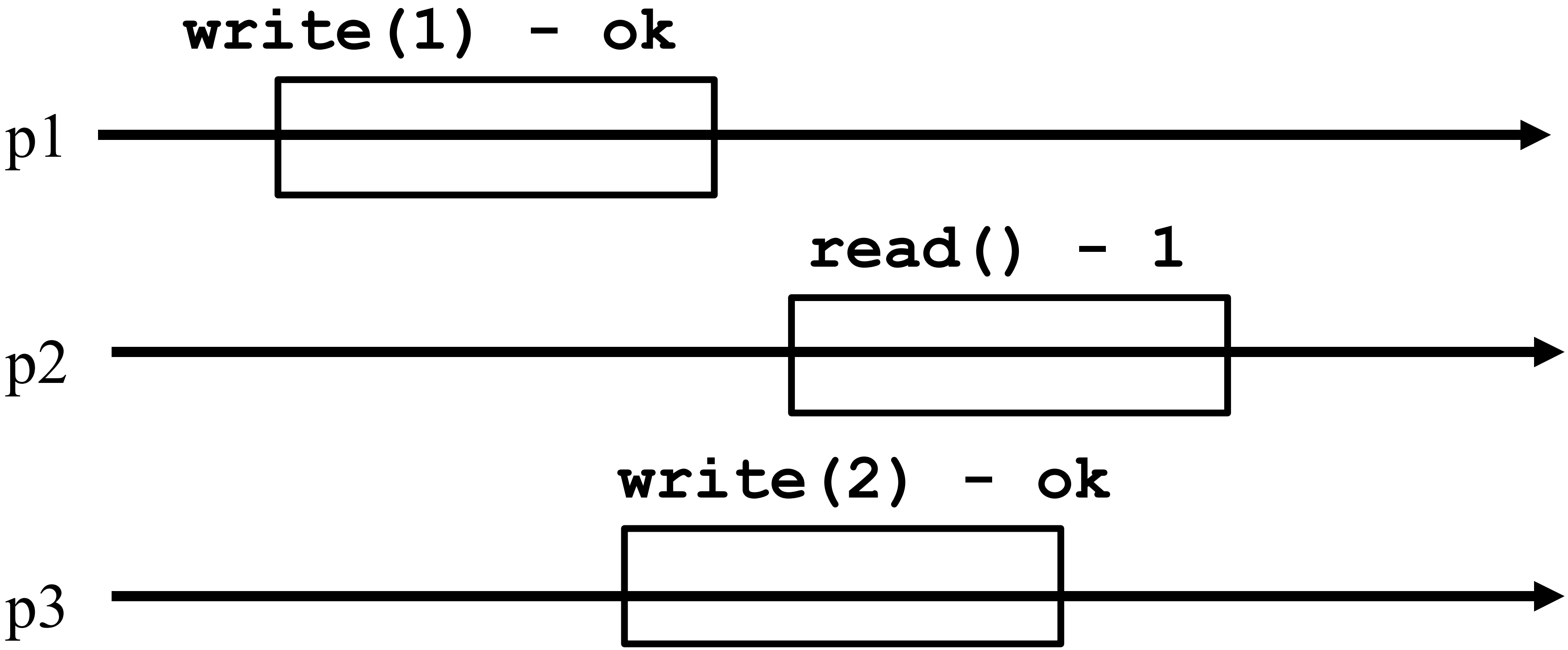
Atomicity?



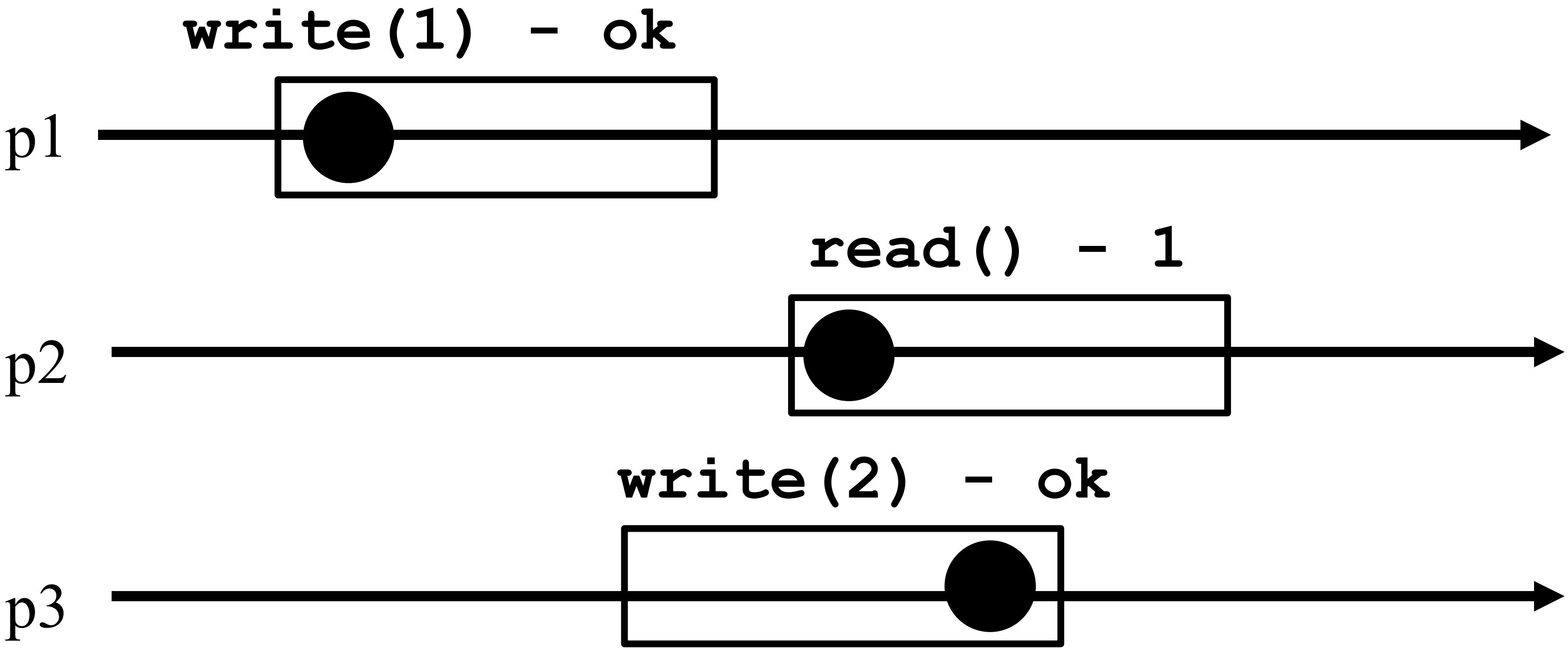
Atomicity?



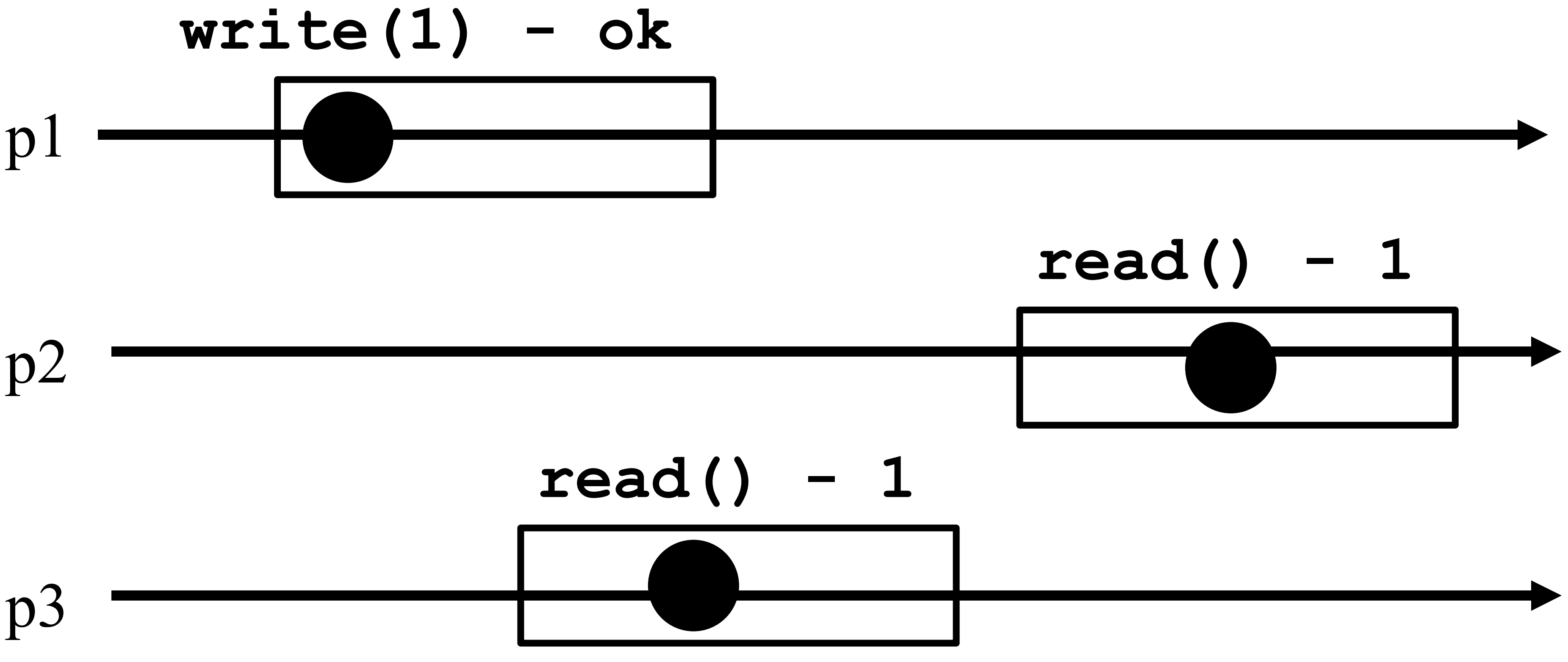
Atomicity?



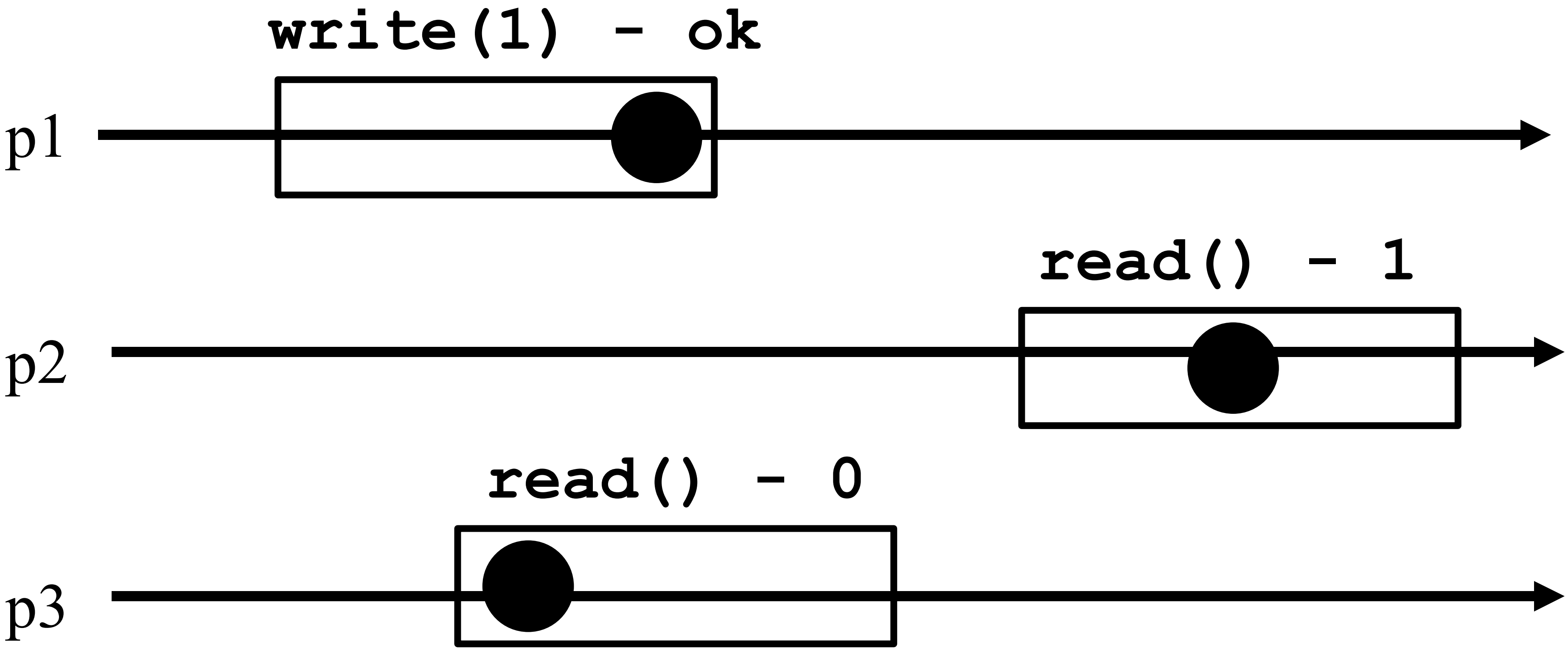
Atomicity?



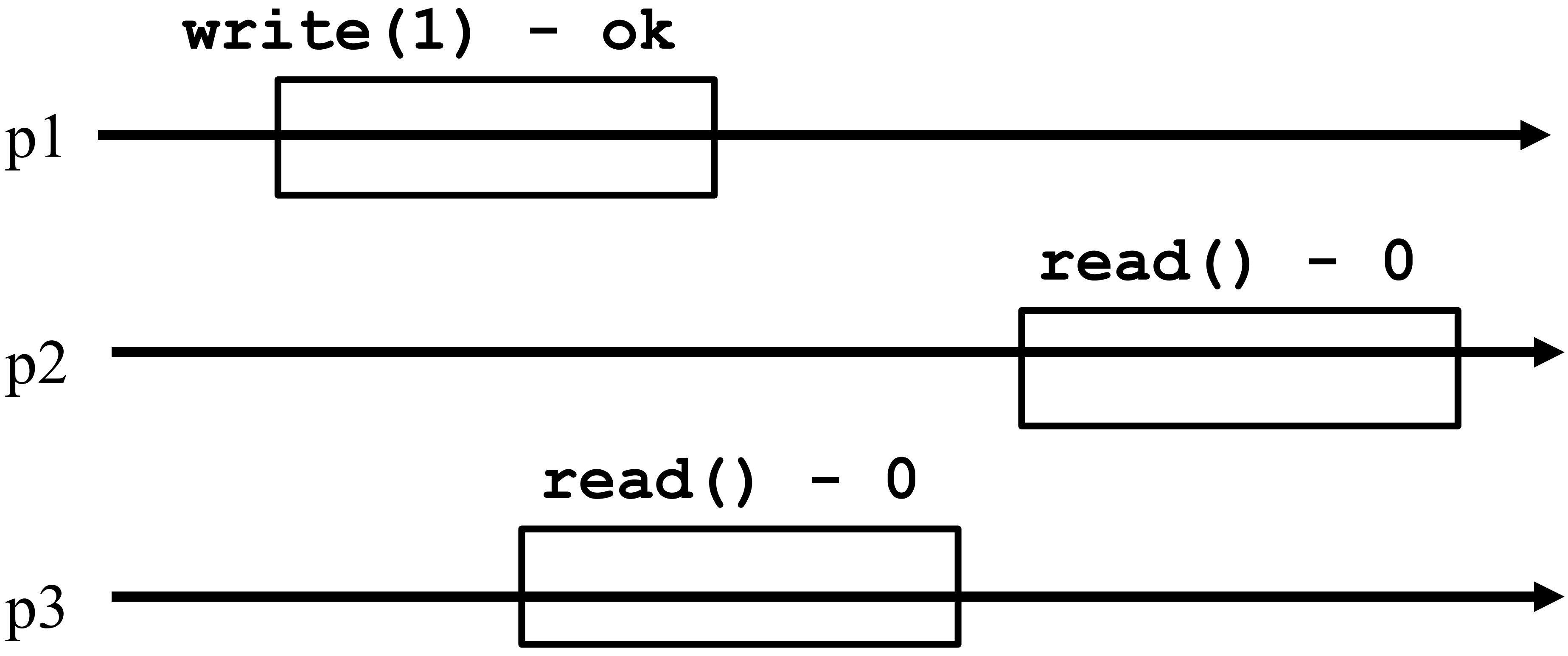
Atomicity?



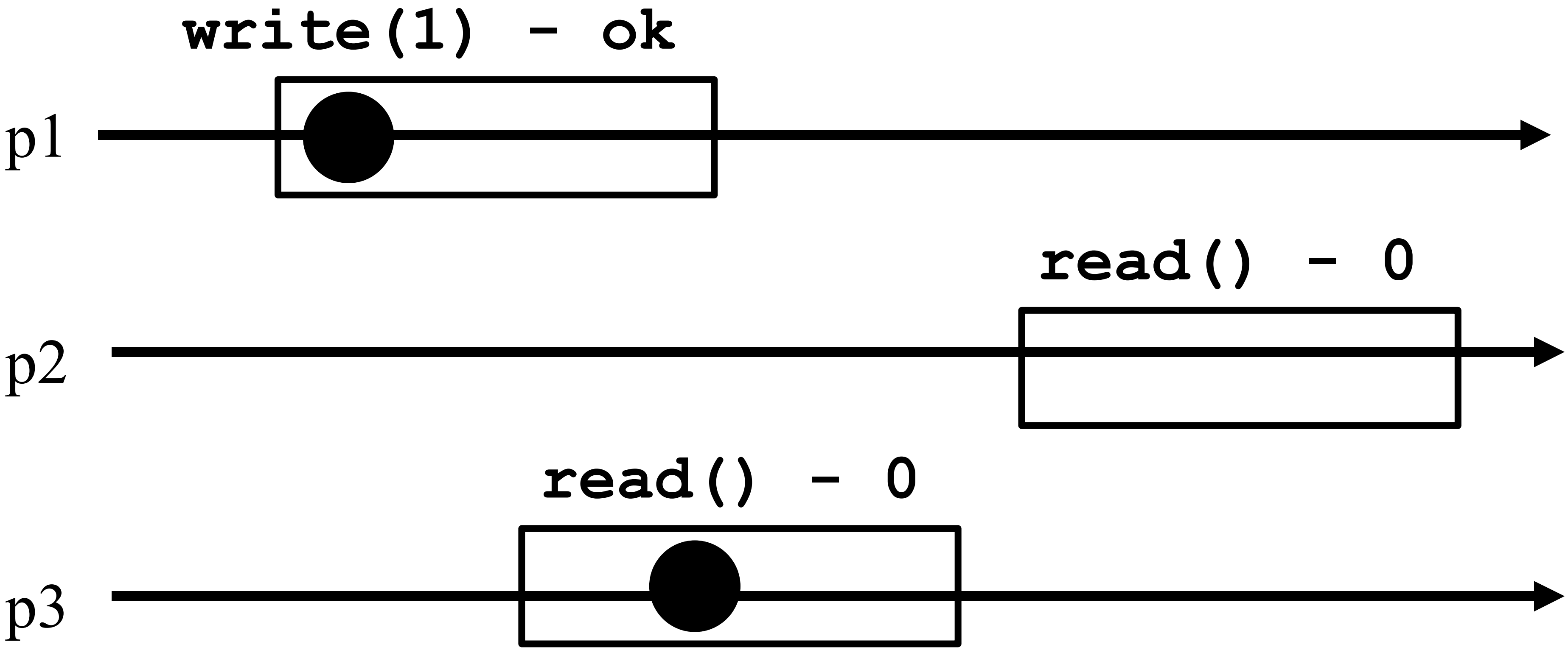
Atomicity?



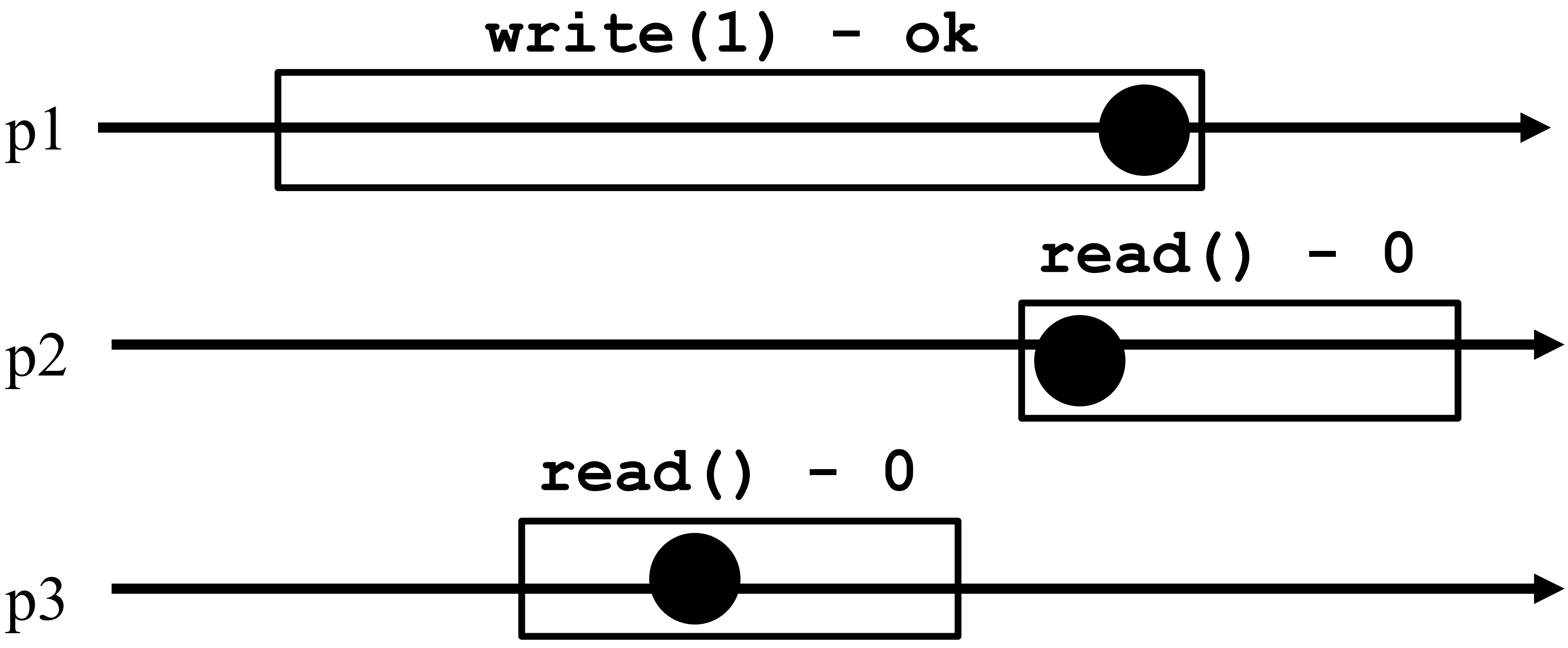
Atomicity?



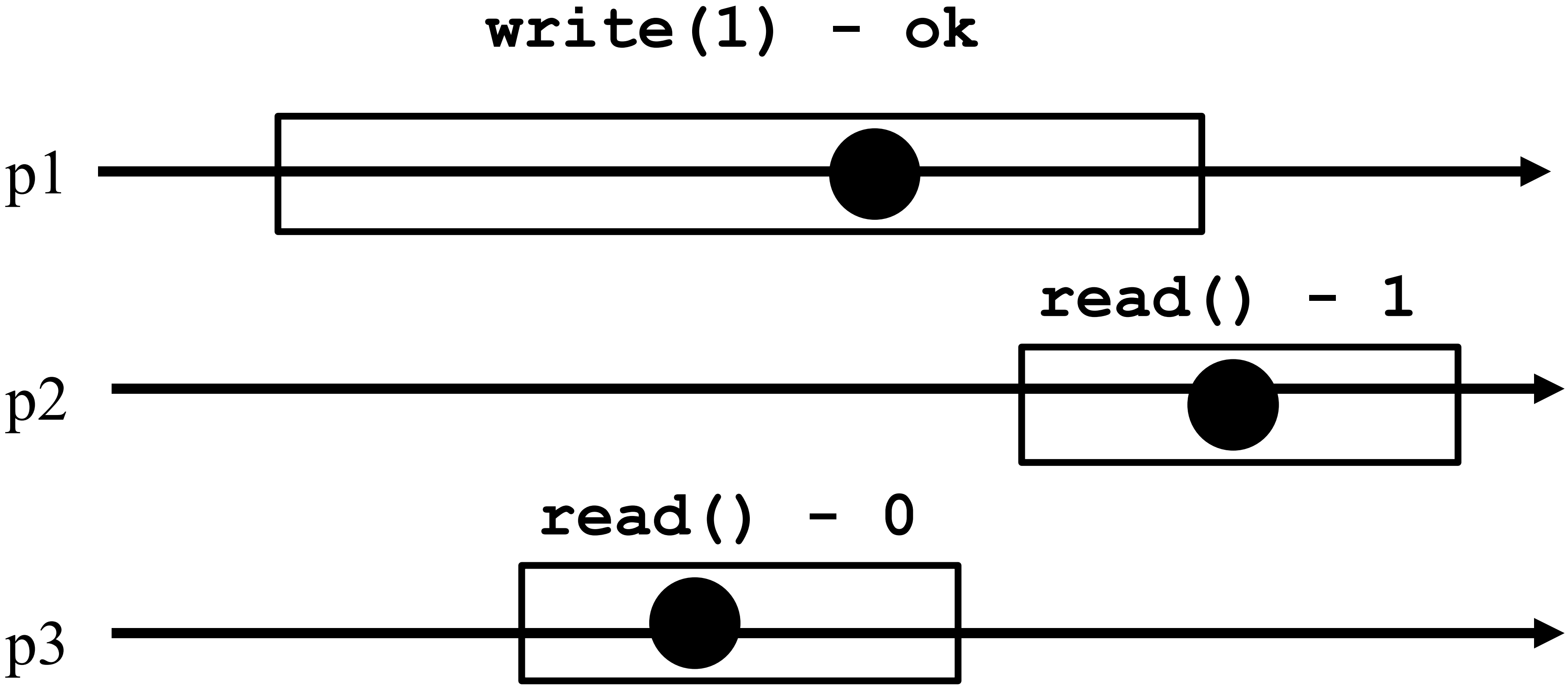
Atomicity?



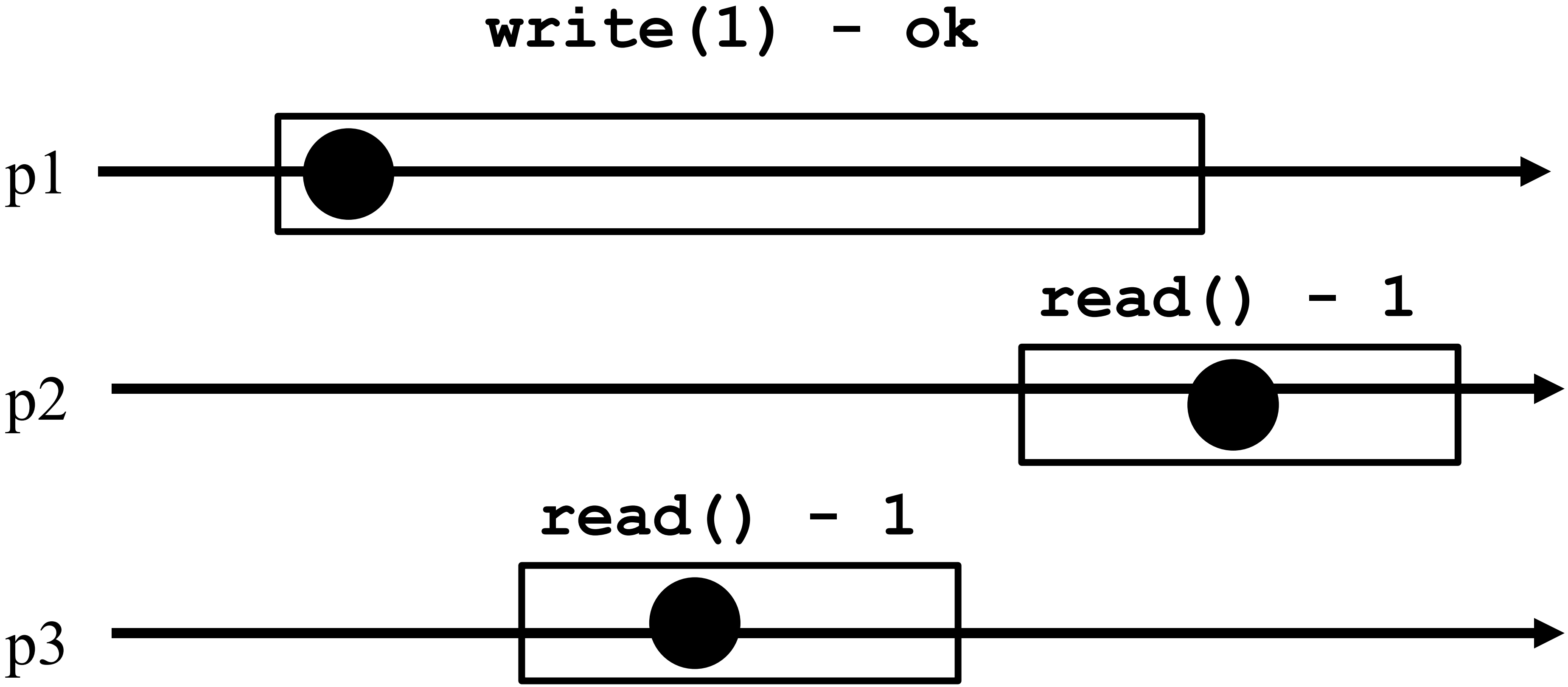
Atomicity?



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

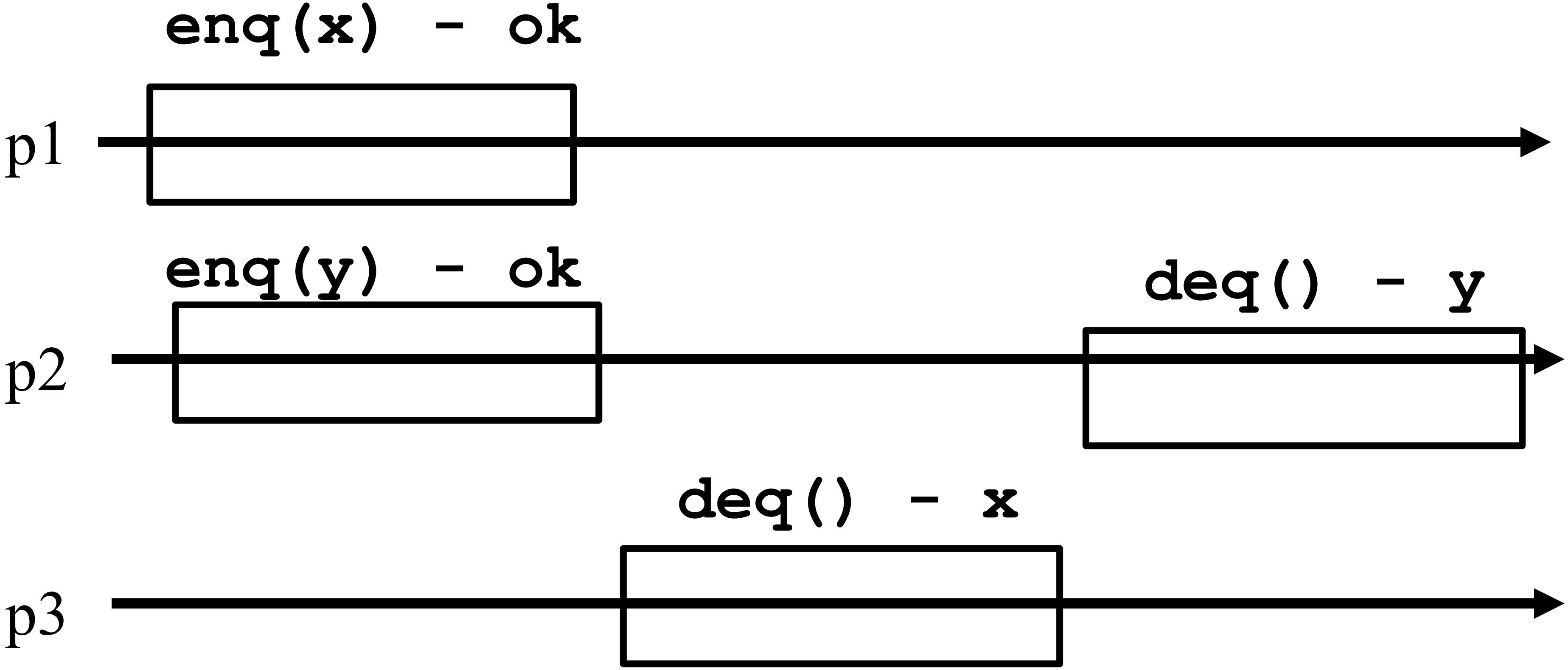
- A ***queue*** has two operations: ***enqueue()*** and ***dequeue()***

- 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

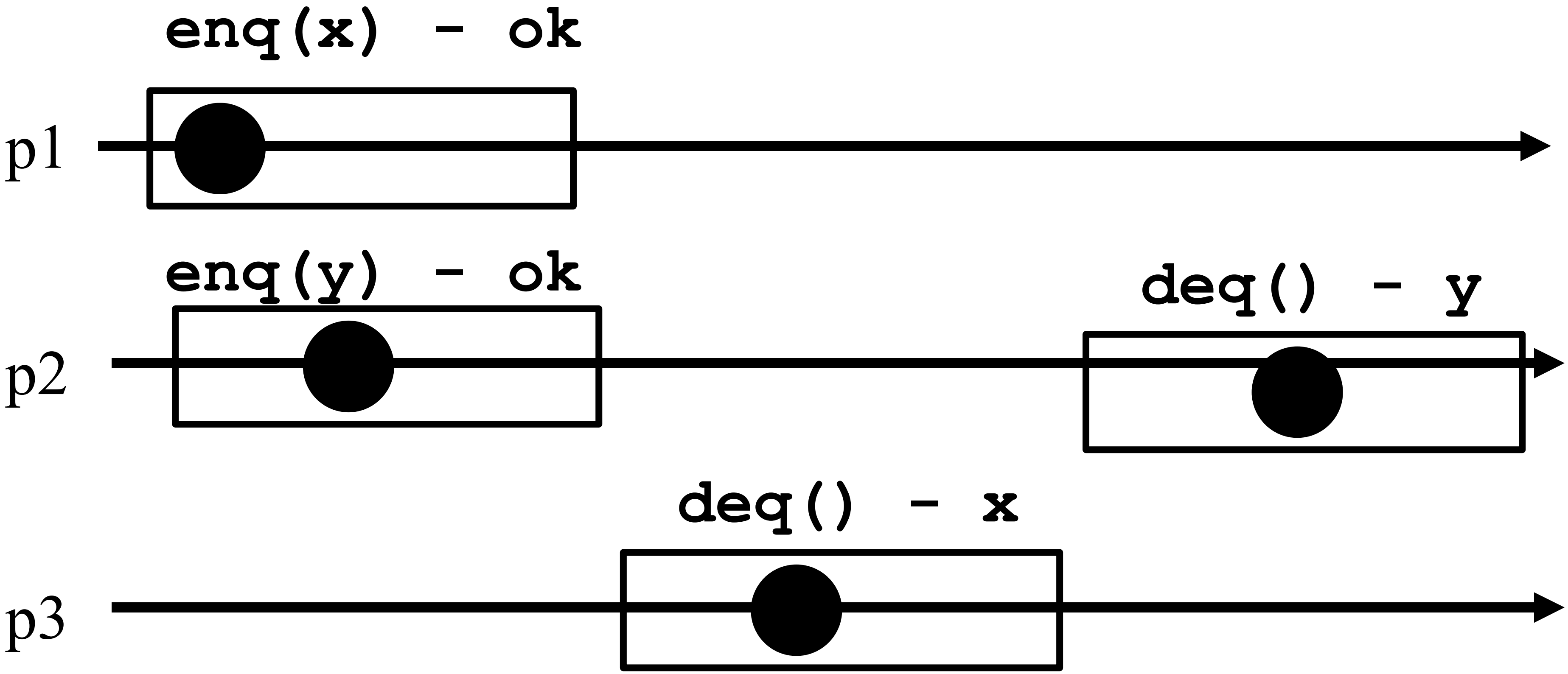
Sequential specification

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

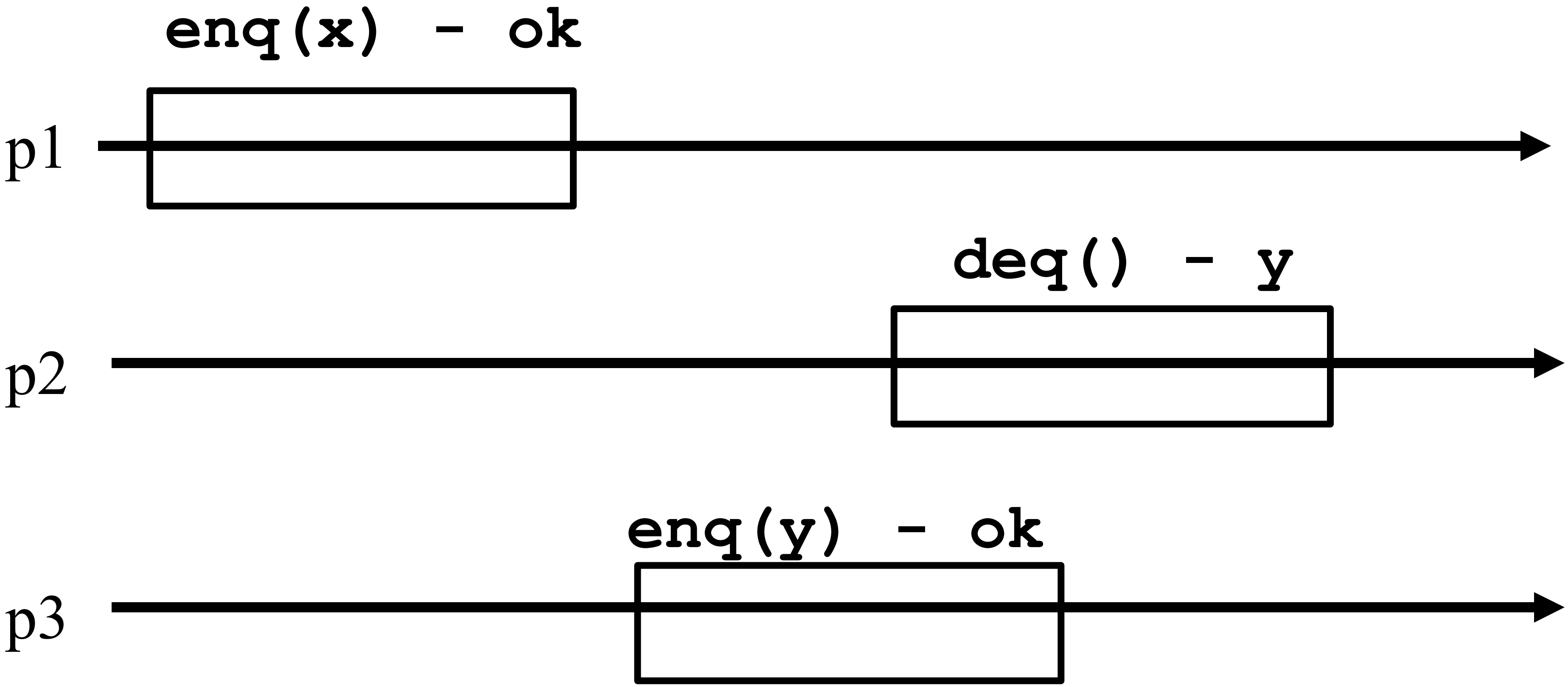
Atomicity?



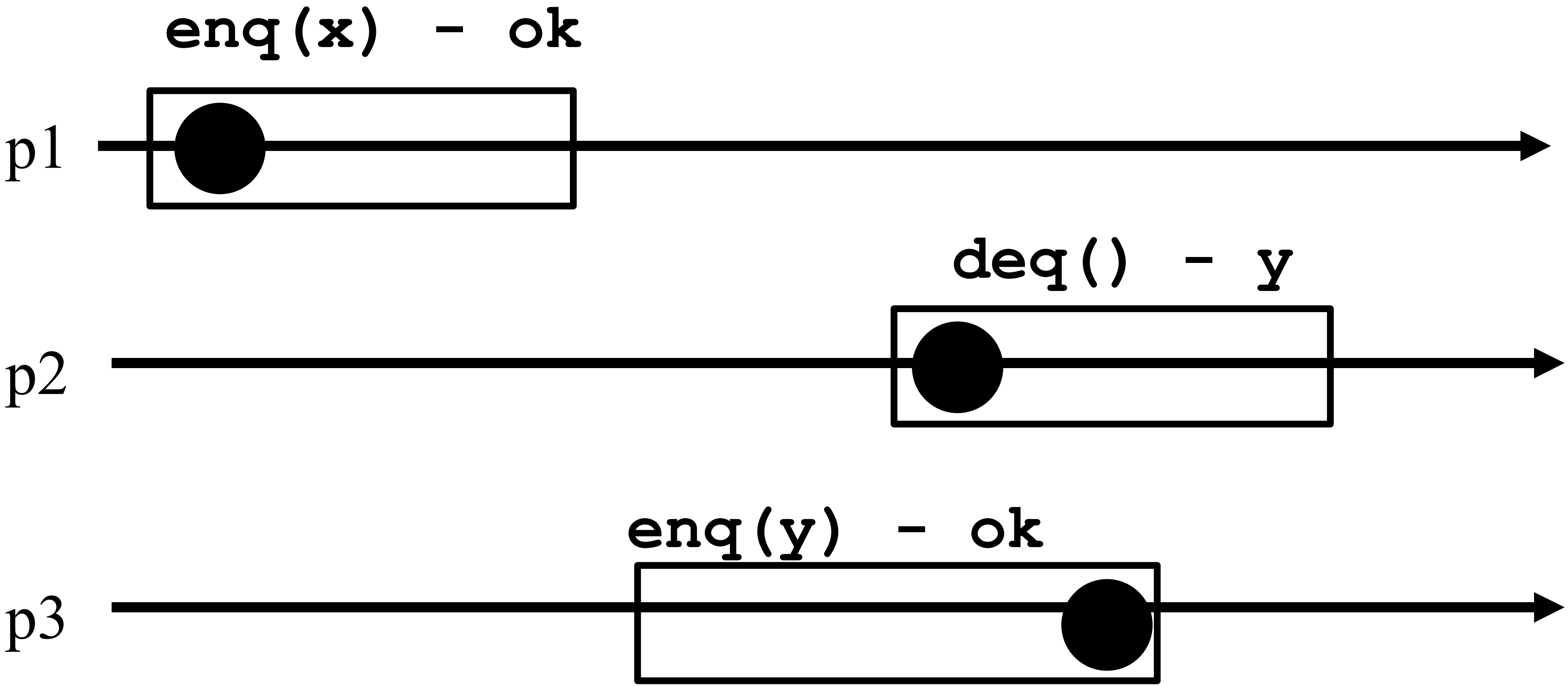
Atomicity?



Atomicity?



Atomicity?



Roadmap

- *Model*
  - *Processes and objects*
  - *Atomicity and wait-freedom*
- *Examples*
- *Content*

Content

- (1) Implementing ***registers***
- (2) The power & limitation of ***registers***
- (3) ***Universal*** objects & synchronization number
- (4) The power of ***time*** & failure detection
- (5) Tolerating ***failure*** prone objects
- (6) ***Anonymous*** implementations
- (7) ***Transaction*** memory

In short

This course shows how to wait-free implement high-level atomic objects out of basic objects

Remark. Unless explicitely stated otherwise, objects mean atomic objects and implementations are wait-free