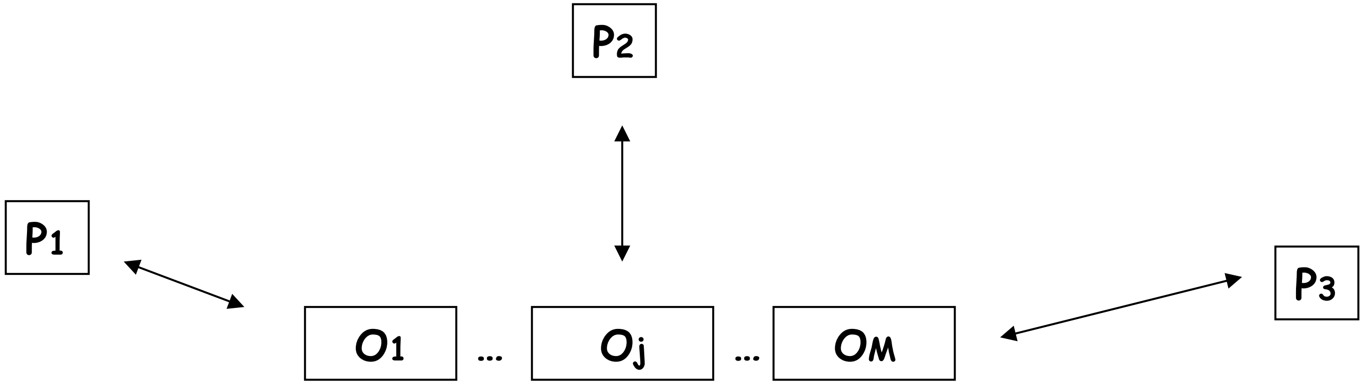
Concurrent systems

Correctness: safety and liveness

SE205, P1, 2017

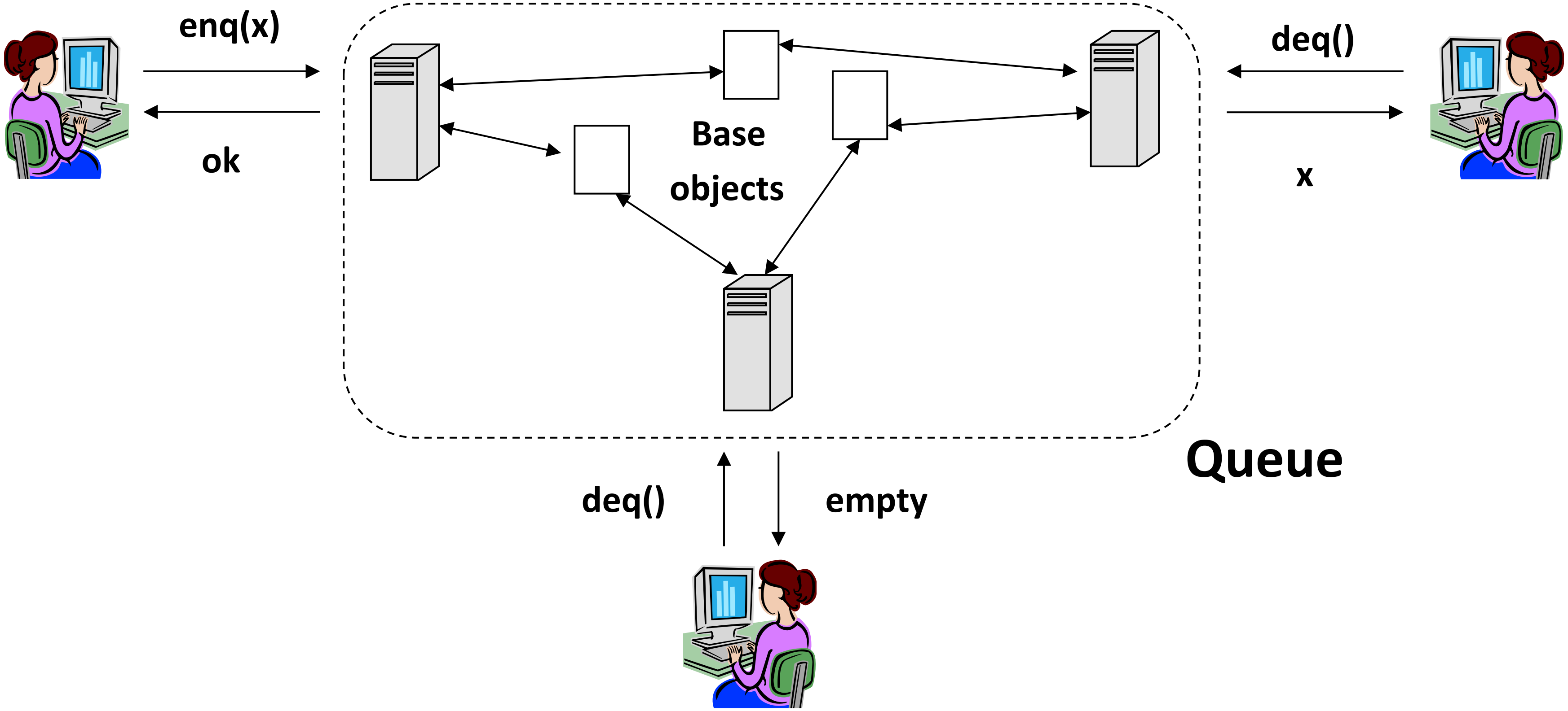
Shared memory

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



Implementing an object

Using *base* objects, create an illusion that an object O is available



enq(x)

ok

Base objects

deq()

x

Queue

deq()

empty

Correctness

What does it mean for an implementation to be correct?

- Safety ≈ nothing bad ever happens
  - ✓Can be violated in a finite execution, e.g., by producing a wrong output or sending an incorrect message
  - ✓What the implementation is allowed to output
- Liveness ≈ something good eventually happens
  - ✓Can only be violated in an *infinite* execution, e.g., by never producing an expected output
  - ✓Under which condition the implementation outputs

In our context

Processes access an (implemented) abstraction (e.g., bounded buffer, a queue, a mutex) by invoking operations

- An operation is implemented using a sequence of accesses to base objects
  - E.g.: a bounded-buffer using reads, writes, TAS, etc.
- A process that never fails (stops taking steps) in the middle of its operation is called correct
  - We typically assume that a correct process invokes infinitely many operations, so a process is correct if it takes infinitely many steps

Runs

A system run is a sequence of events

- ✓E.g., actions that processes may take

Σ – event alphabet

- ✓ E.g., all possible actions

Σ^ω is the set all finite and infinite runs

A property P is a subset of Σ^ω

An implementation satisfies P if every its run is in P

Safety properties

P is a safety property if:

- P is prefix-closed: if σ is in P, then each prefix of σ is in P
- P is limit-closed: for each infinite sequence of traces $\sigma_0, \sigma_1, \sigma_2, \ldots$, such that each σ_i is a prefix of σ_{i+1} and each σ_i is in P, the limit trace σ is in P

(Enough to prove safety for all finite traces of an algorithm)

Liveness properties

P is a liveness property if every finite σ (in Σ*, the set of all finite histories) has an extension in P

(Enough to prove liveness for all infinite runs)

A liveness property is dense: intersects with extensions of every finite trace

Safety? Liveness?

- Processes propose *values* and decide on *values* (distributed tasks):

$\Sigma = \cup_{i,v}\{\mathtt{propose}_i(v), \mathtt{decide}_i(v)\} \cup \{\mathtt{base\text{-}object\ accesses}\}$

✓Every decided value was previously proposed

✓No two processes decide differently

✓Every correct (taking infinitely many steps) process eventually decides

✓No two correct processes decide differently

Quiz 1: safety

1. Let S be a safety property. Show that if all finite runs of an implementation I are safe (belong to S) then all runs of I are safe
2. Show that every unsafe run σ has an unsafe finite prefix σ’: every extension of σ’ is unsafe
3. Show that every property is an intersection of a safety property and a liveness property

How to distinguish safety and liveness: rules of thumb

Let P be a property (set of runs)

- If every run that violates P is infinite
  - ✓P is liveness
- If every run that violates P has a finite prefix that violates P
  - ✓P is safety
- Otherwise, P is a mixture of safety and liveness

Example: implementing a concurrent queue

What *is* a concurrent FIFO queue?

✓FIFO means strict temporal order

✓Concurrent means ambiguous temporal order

When we use a lock…

```
shared
     items[];
     tail, head := 0

deq()

  lock.lock();
    if (tail = head)
       x := empty;
    else
       x := items[head];
       head++;
  lock.unlock();
  return x;
```

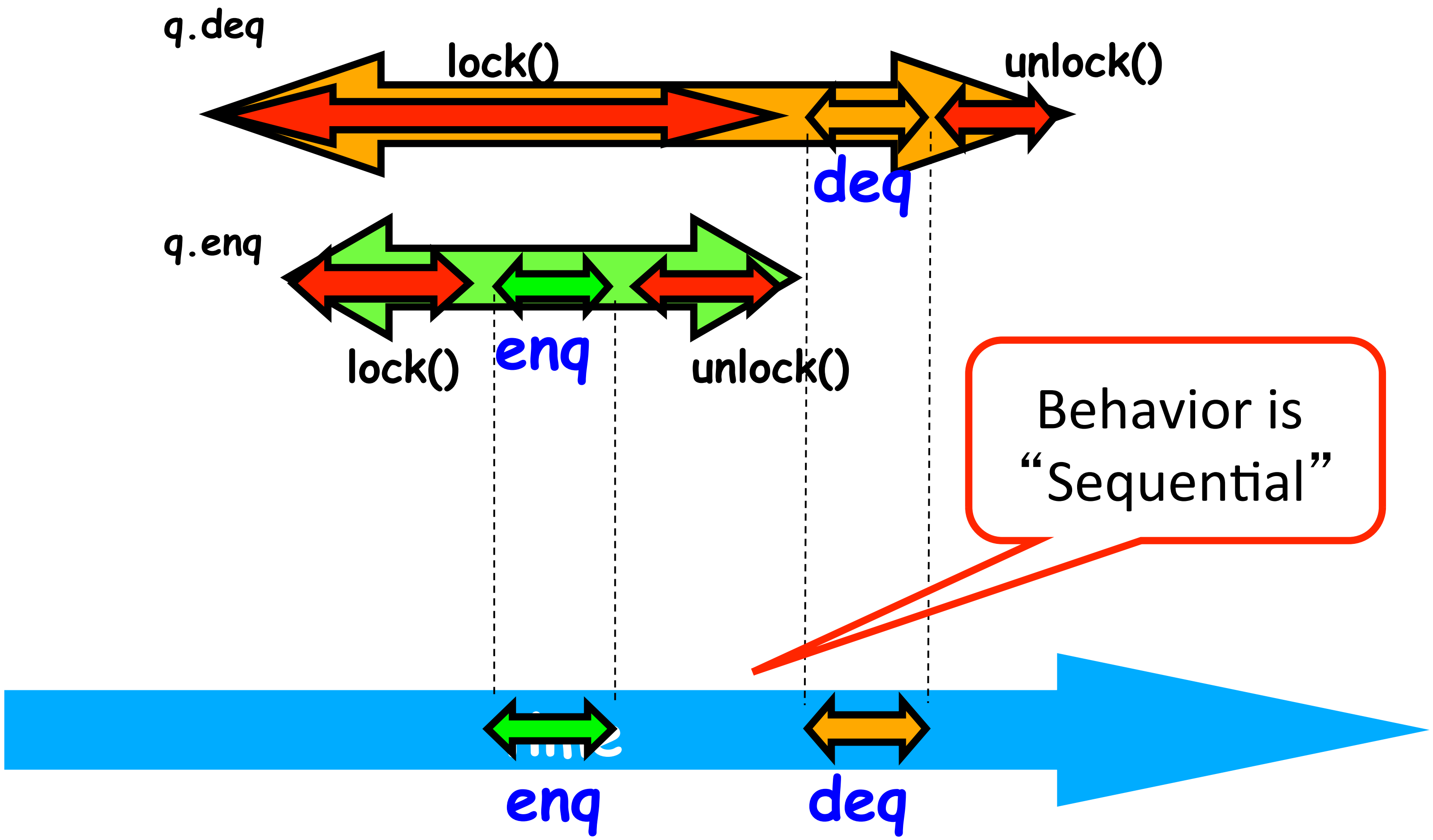
Intuitively…

```
deq()

  lock.lock();
    if (tail = head)
       x := empty;
    else
       x := items[head];
       head++;
  lock.unlock();
  return x;
```

All modifications
of queue are done
in mutual exclusion

We describe the concurrent via the sequential



Behavior is "Sequential"

Linearizability (atomicity): A Safety Property

- Each complete operation should
  - ✓"take effect"
  - ✓Instantaneously
  - ✓Between invocation and response events

- The history of a concurrent execution is correct if its "sequential equivalent" is correct

- Need to define histories first

Histories

A history is a sequence of invocation and responses

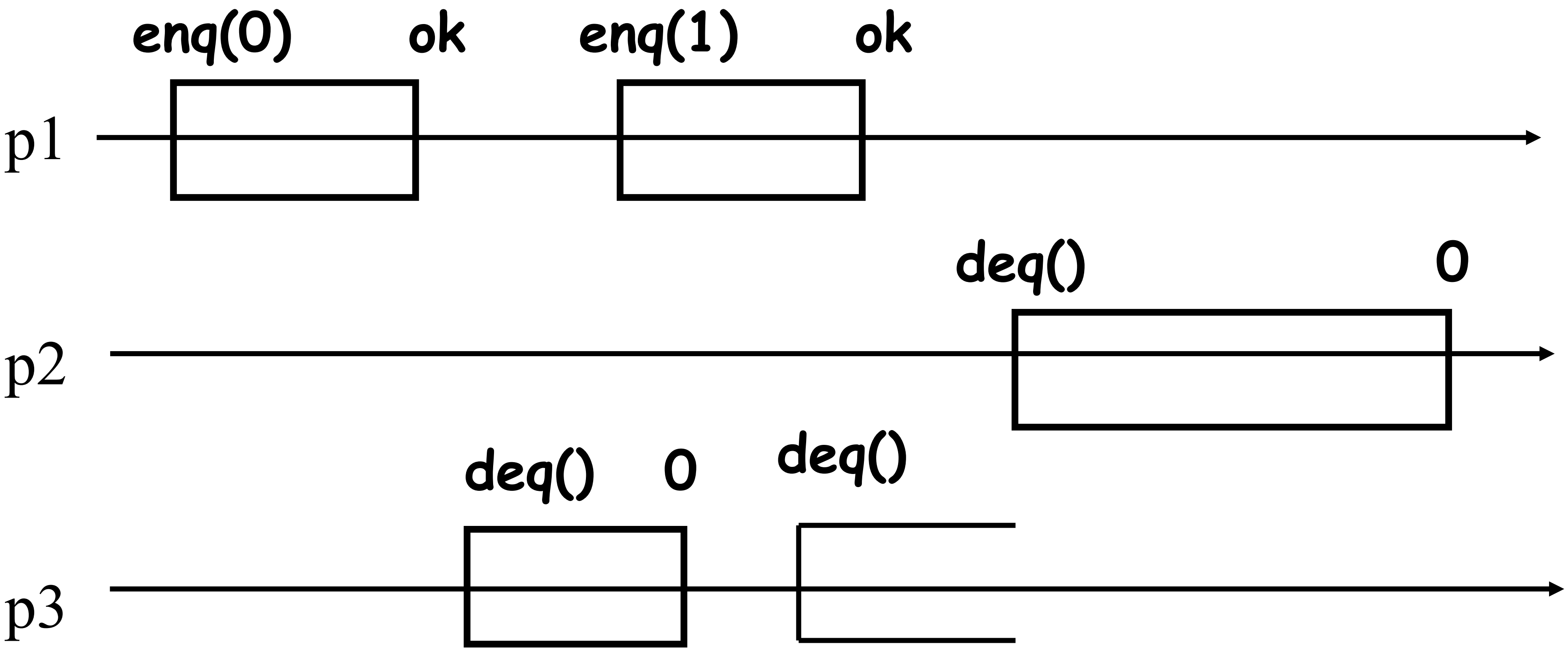
E.g., p1-enq(0), p2-deq(),p1-ok,p2-0,…

A history is sequential if every invocation is immediately followed by a corresponding response

E.g., p1-enq(0), p1-ok, p2-deq(),p2-0,…

(A sequential history has no concurrent operations)

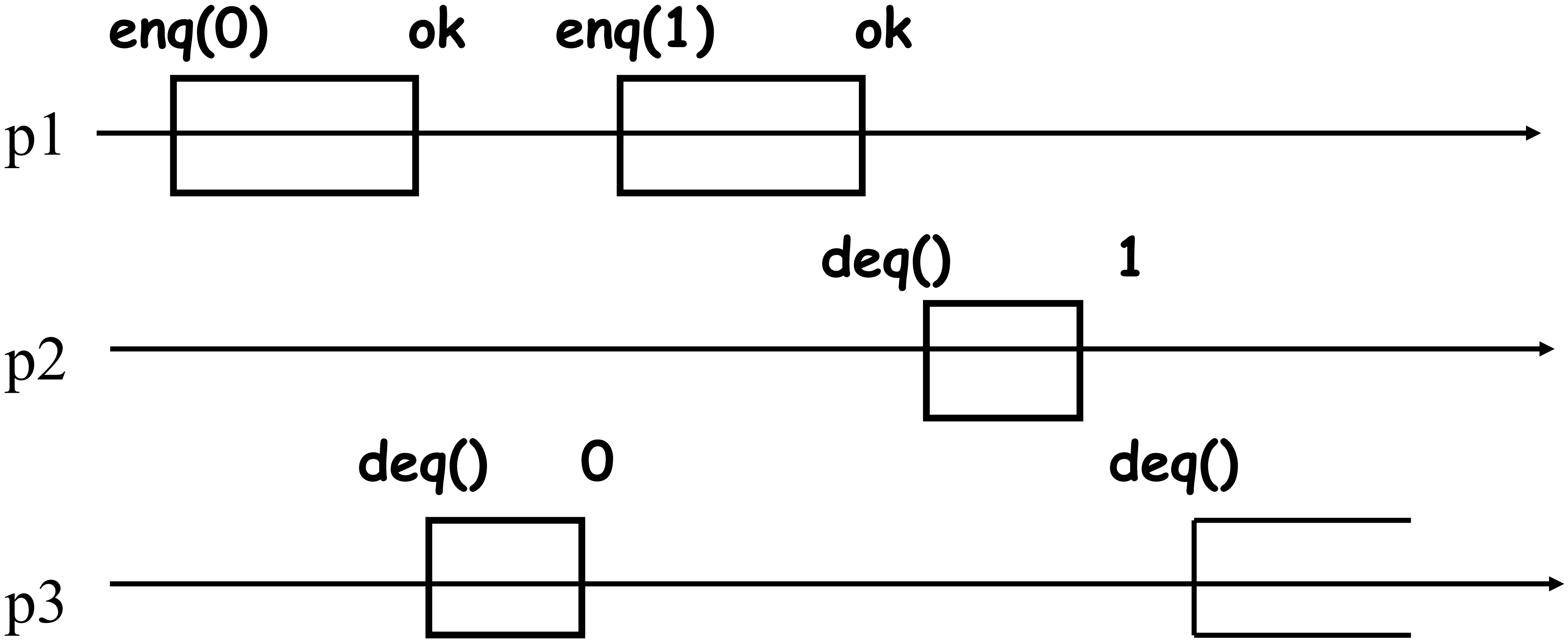
Histories



History:

p1-enq(0); p1-ok; p3-deq(); p1-enq(); p3-0; p3-deq(); p1-ok; p2-deq(); p2-0

Histories



History:

p1-enq(0); p1-ok; p3-deq(); p3-0; p1-enq(1); p1-ok; p2-deq(); p2-1; p3-deq();

Legal histories

A sequential history is *legal* if it satisfies the sequential specification of the shared object

- (FIFO) queues:
  Every deq returns the first not yet dequeued value
- Read-write registers:
  Every read returns the last written value

(well-defined for sequential histories)

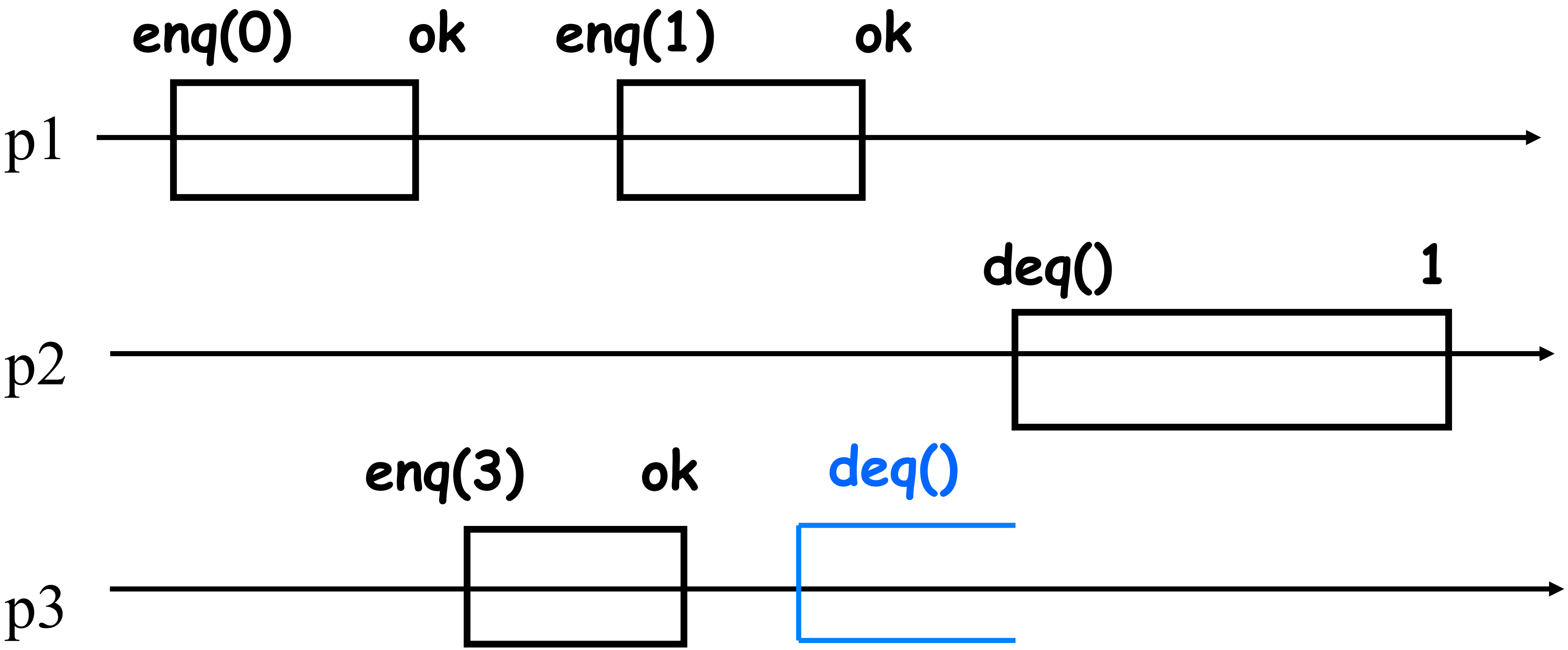
Complete operations and completions

Let H be a history

An operation op is *complete* in H if H contains both the invocation and the response of op

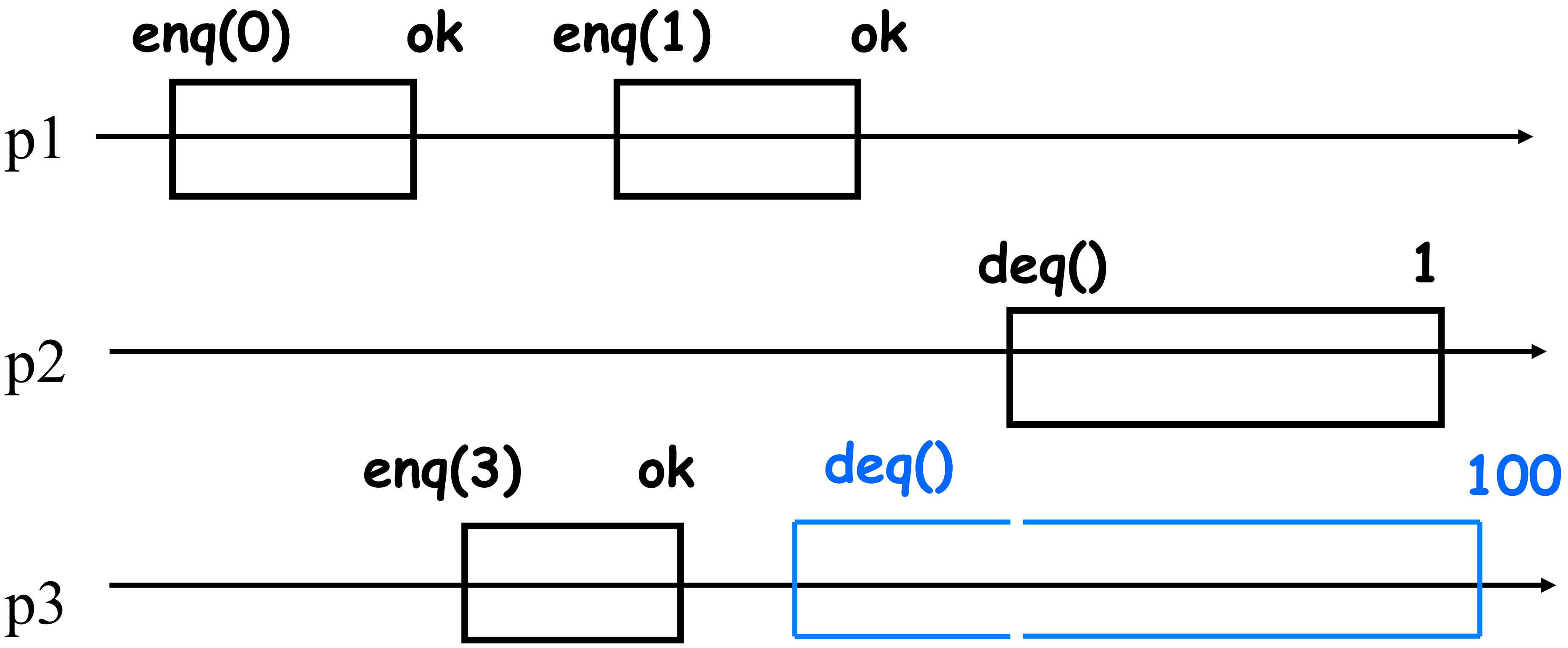
A *completion* of H is a history H' that includes all complete operations of H and a subset of incomplete operations of H followed with matching responses

Complete operations and completions



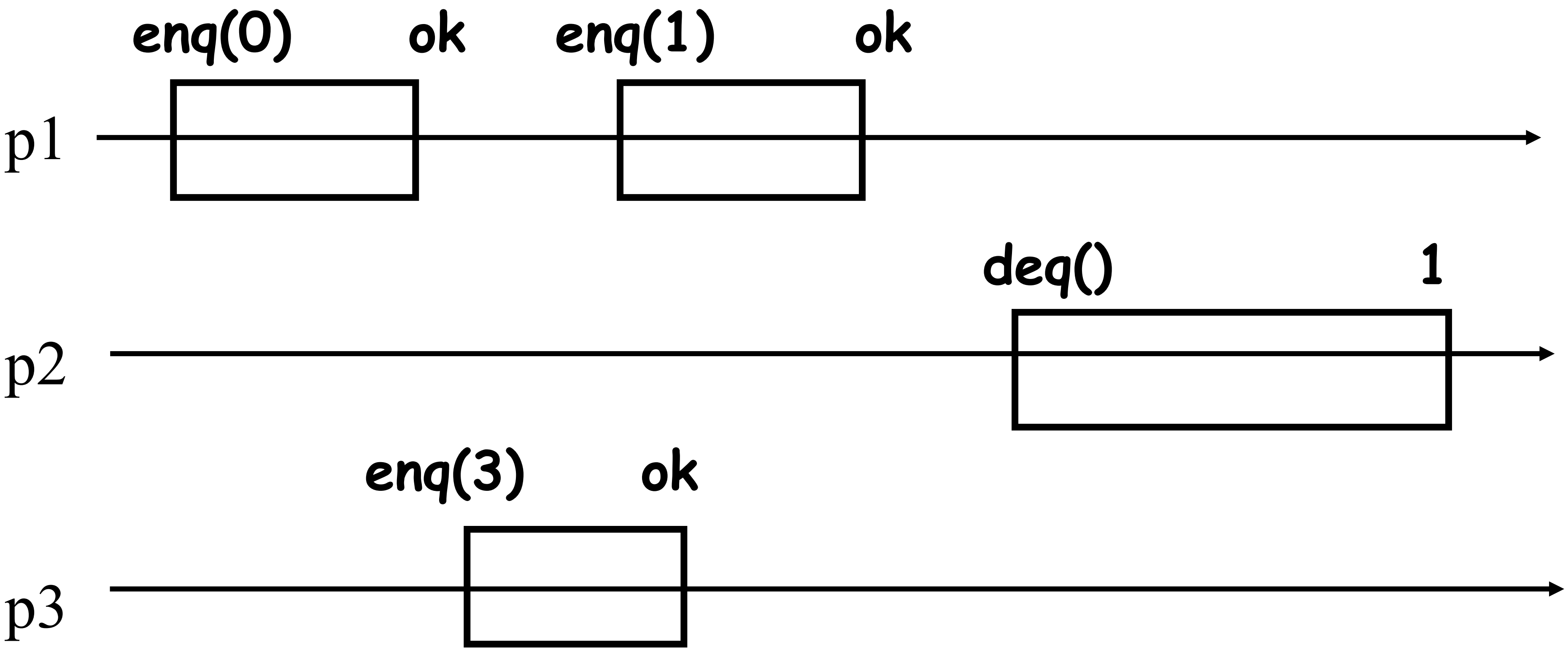
p1-enq(0); p1-ok; p3-enq(3); p1-enq(1); p3-ok; p3-deq(); p1 –ok; p2-deq(); p2-1;

Complete operations and completions



**p1-enq(0); p1-ok; p3-enq(3); p1-enq(1); p3-ok;
p3-deq(); p1 –ok; p2-deq(); p2-1; p3-100**

Complete operations and completions



p1-enq(0); p1-ok; p3-enq(3); p1-enq(1); p3-ok; p1 –ok; p2-deq(); p2-1;

Equivalence

Histories H and H' are *equivalent* if for all pi

$H \mid p_i = H' \mid p_i$

E.g.:

$H = p_1\text{-enq}(0);\ p_1\text{-ok};\ p3\text{-deq}();\ p_3\text{-}3$

$H' = p_1\text{-enq}(0);\ p_3\text{-deq}();\ p_1\text{-ok};\ p_3\text{-}3$

Linearizability (atomicity)

A history H is *linearizable* if there exists a sequential legal history S such that:

- S is equivalent to some completion of H
- S preserves the precedence relation of H:

op1 precedes op2 in H => op1 precedes op2 in S

What if: define a completion of H as any complete extension of H?

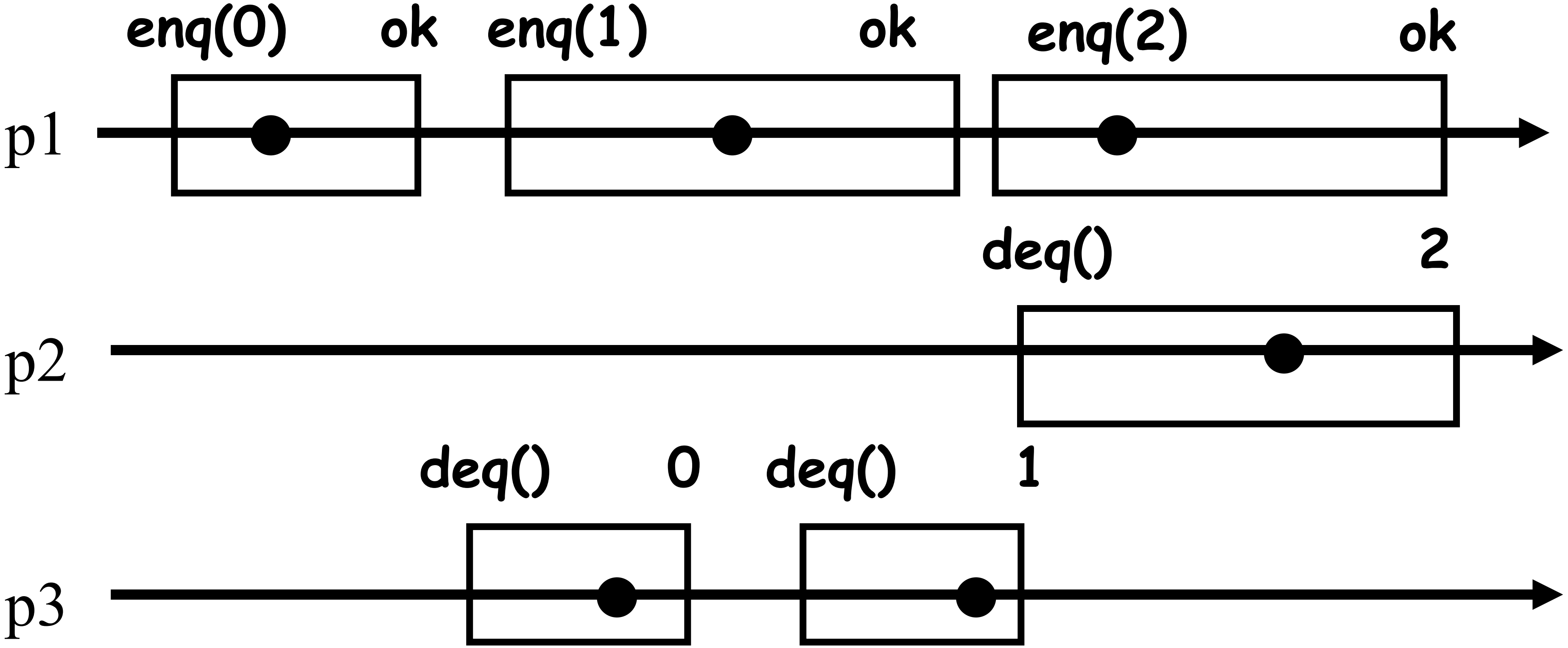
Linearization points

An implementation is *linearizable* if every history it produces is linearizable

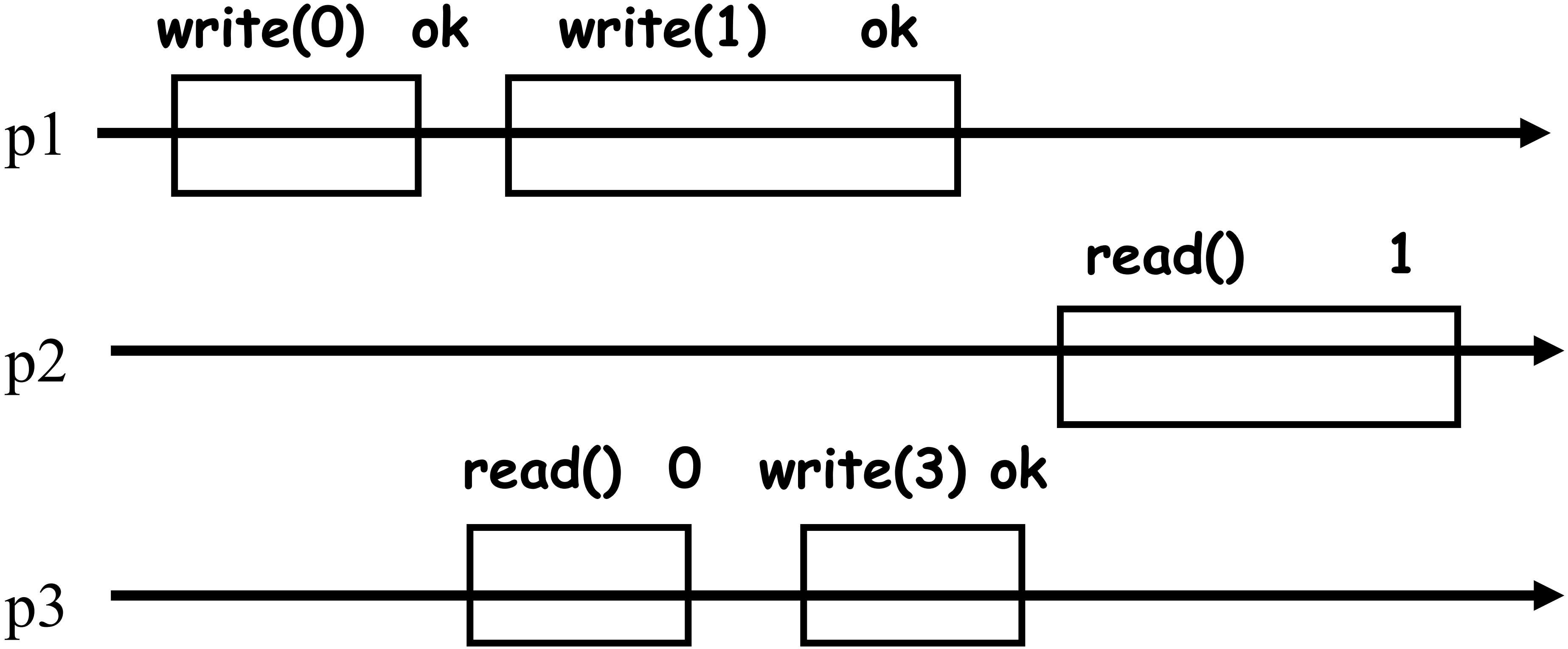
Informally, the complete operations (and some incomplete operations) in a history are seen as taking effect instantaneously at some time between their invocations and responses

Operations ordered by their linearization points constitute a legal (sequential) history

Linearizable?

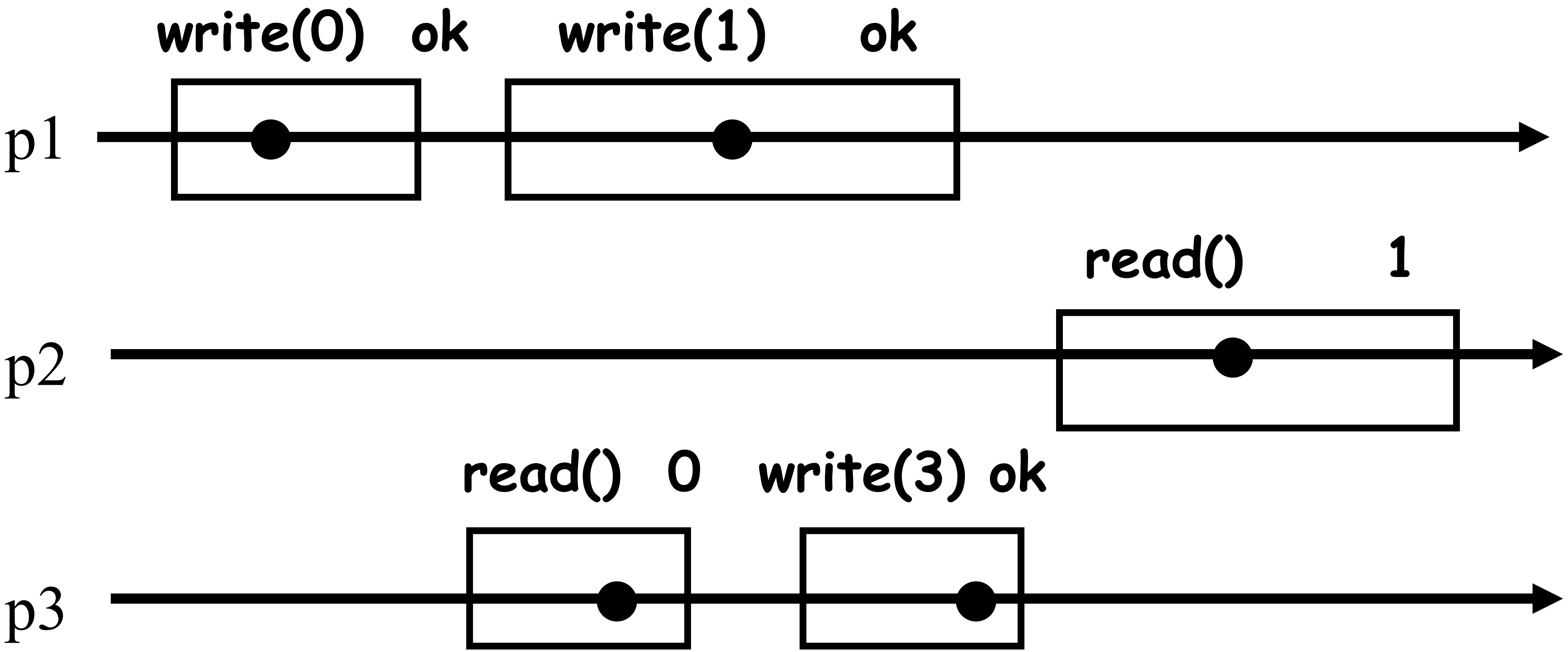


Linearizable?

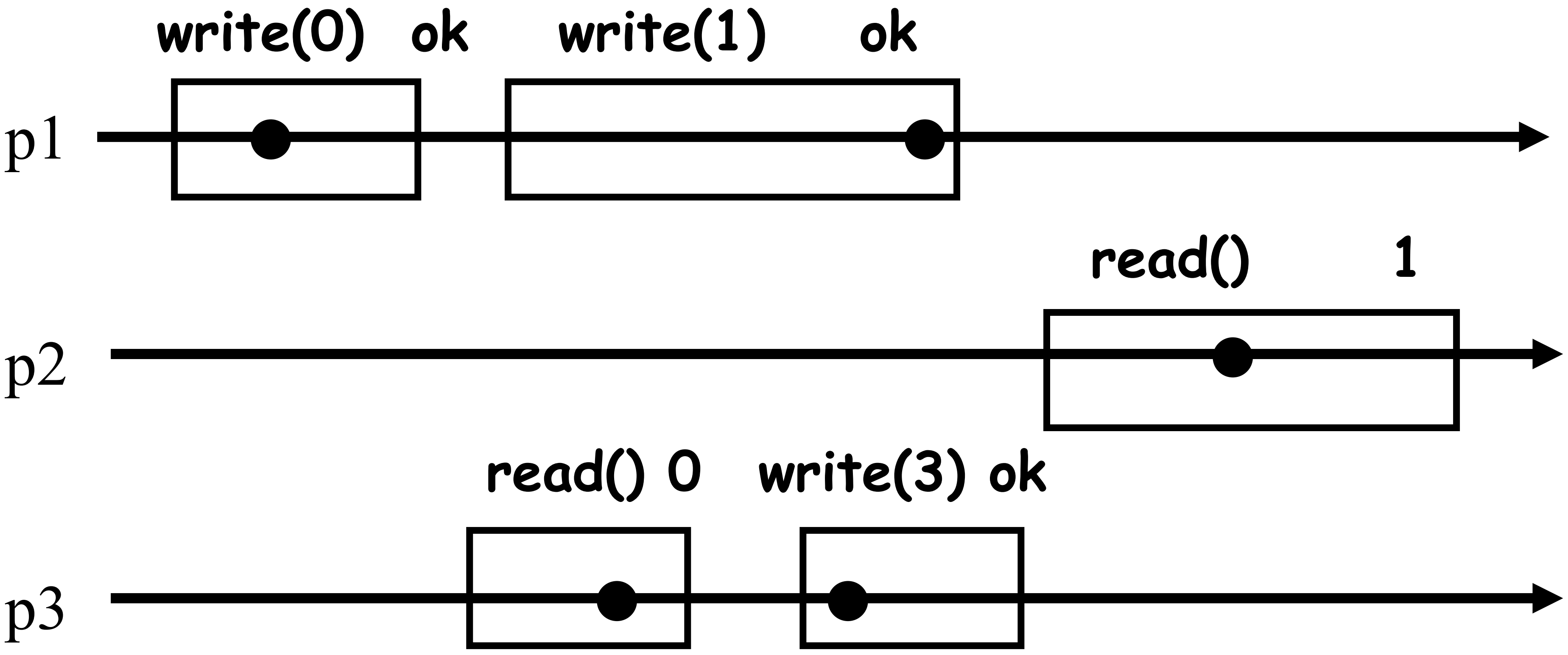


© 2017 P. Kuznetsov

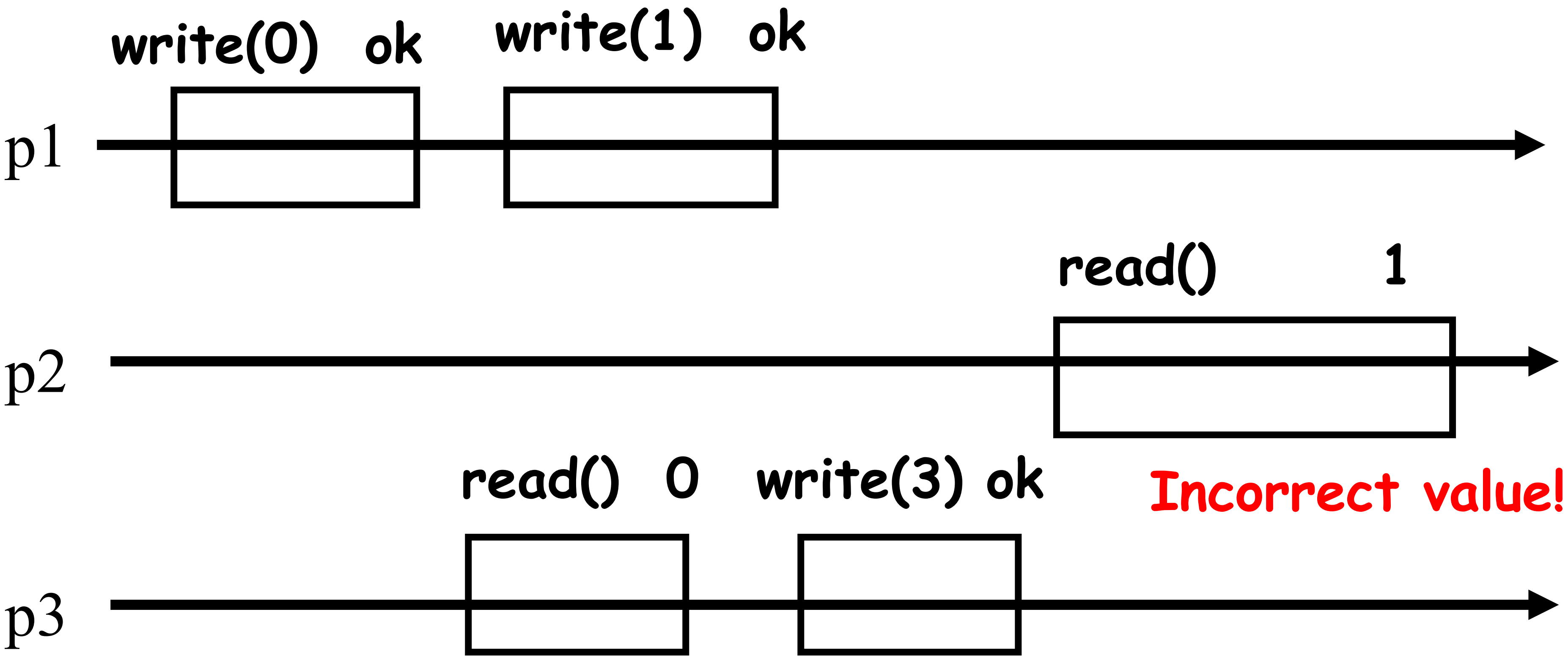
Linearizable?



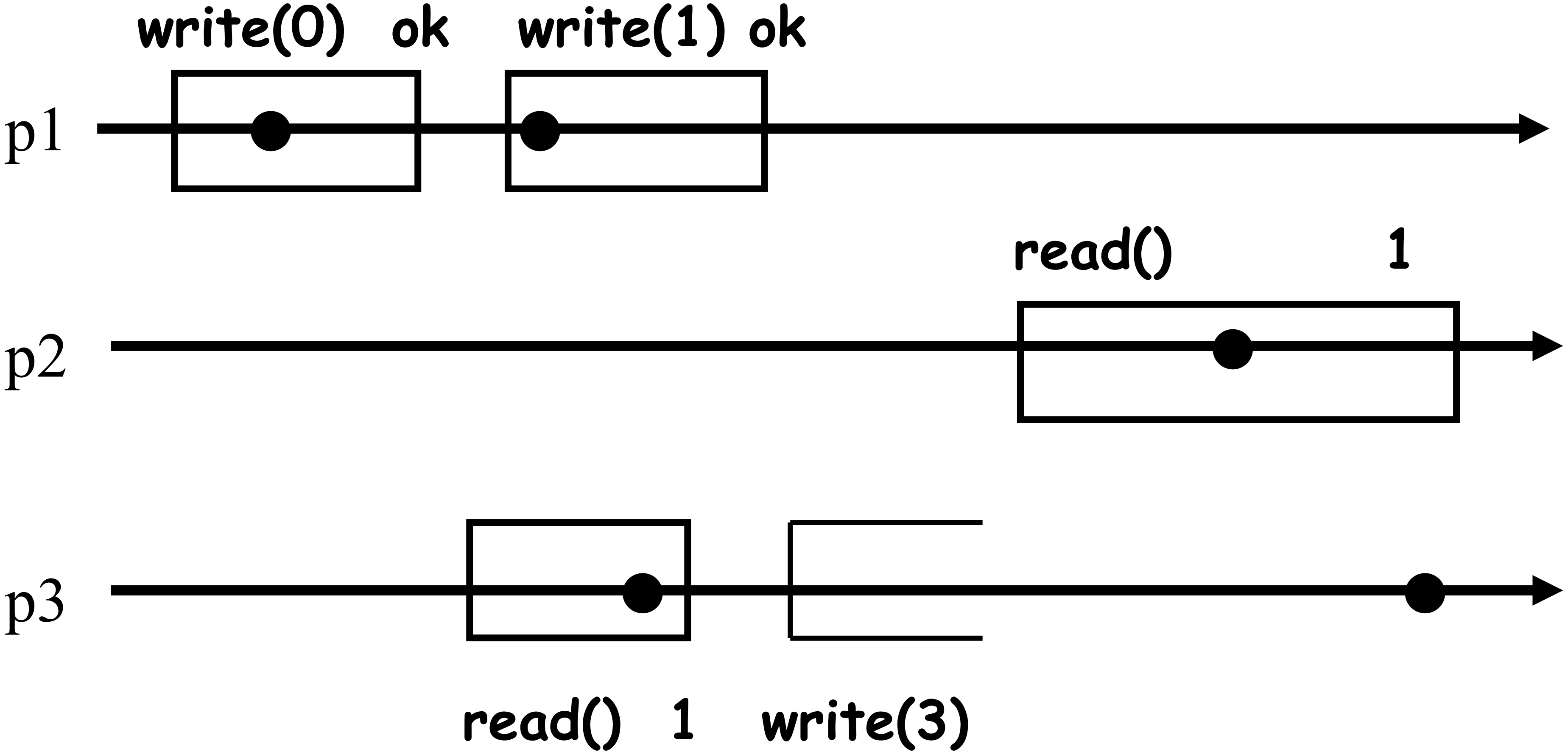
Linearizable?



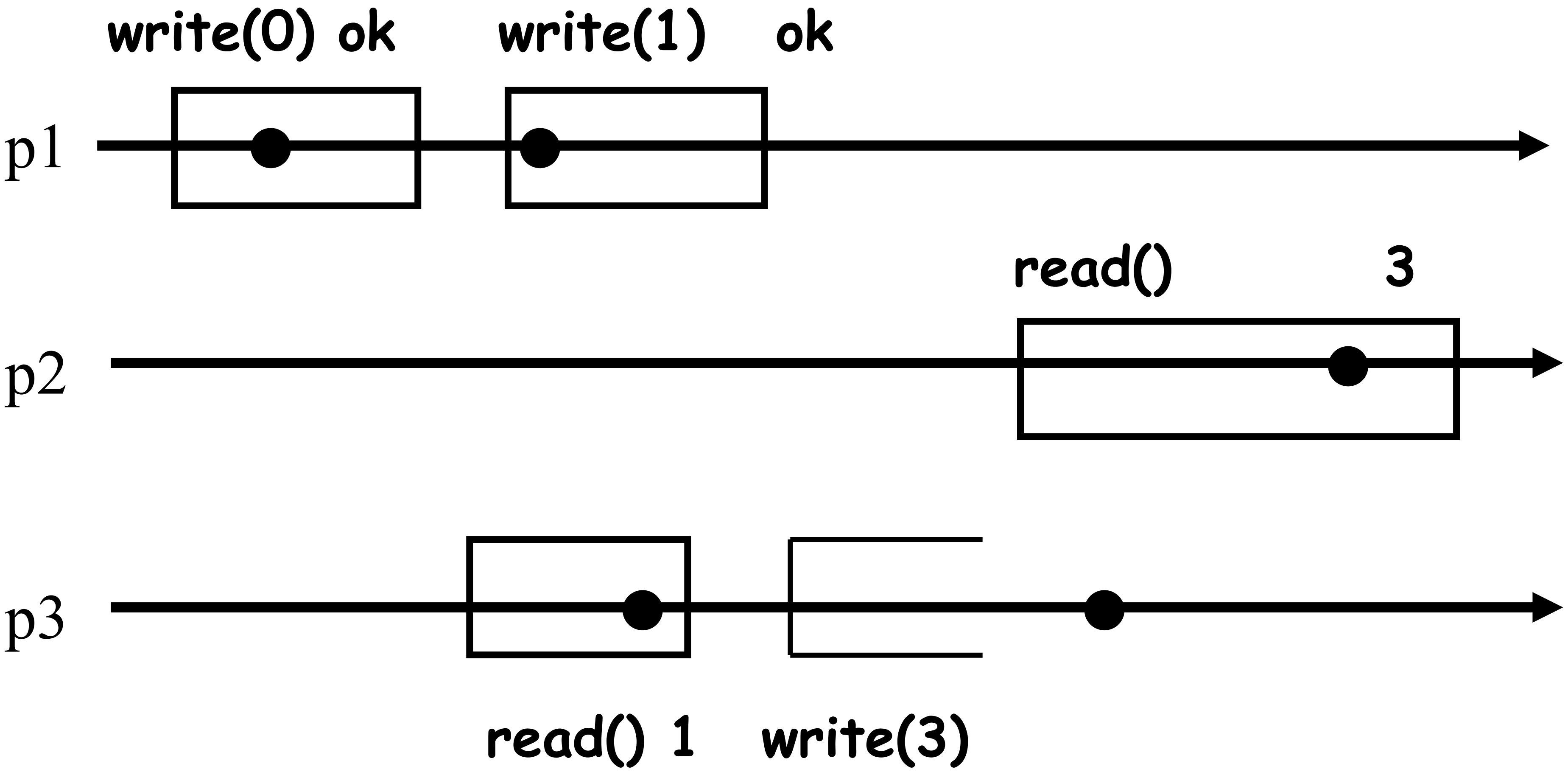
Linearizable?



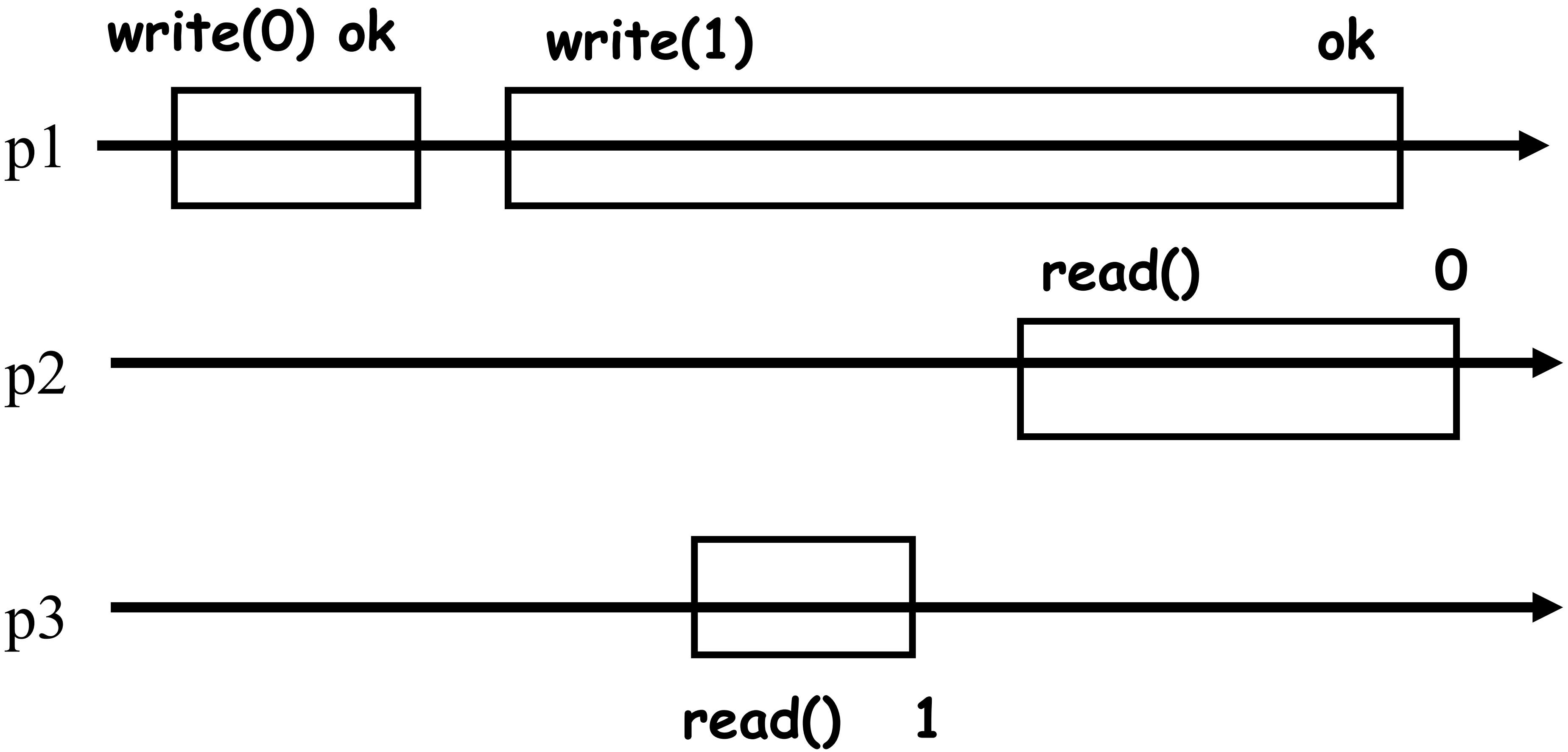
Linearizable?



Linearizable?



Linearizable?



Sequential consistency

A history H is *sequentially consistent* if there exists a sequential legal history S such that:

- S is equivalent to some completion of H
- S preserves the per-process order of H:

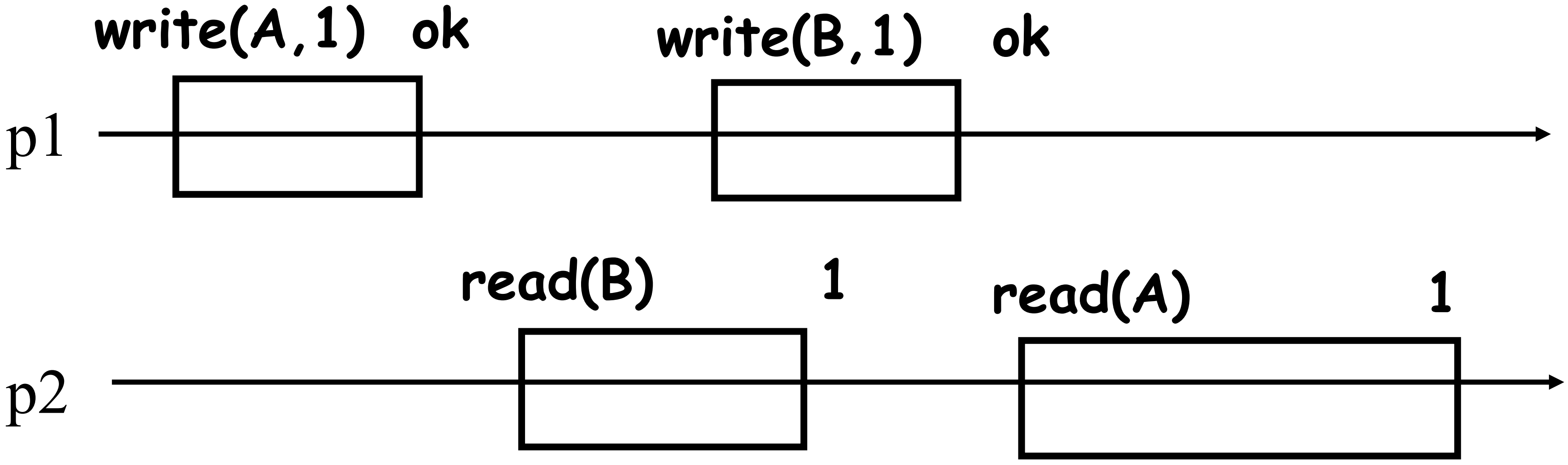
pi executes op1 before op2 in H => pi executes op1 before op2 in S

Why (strong) linearizability and not (weak) sequential consistency?

Linearizability is compositional!

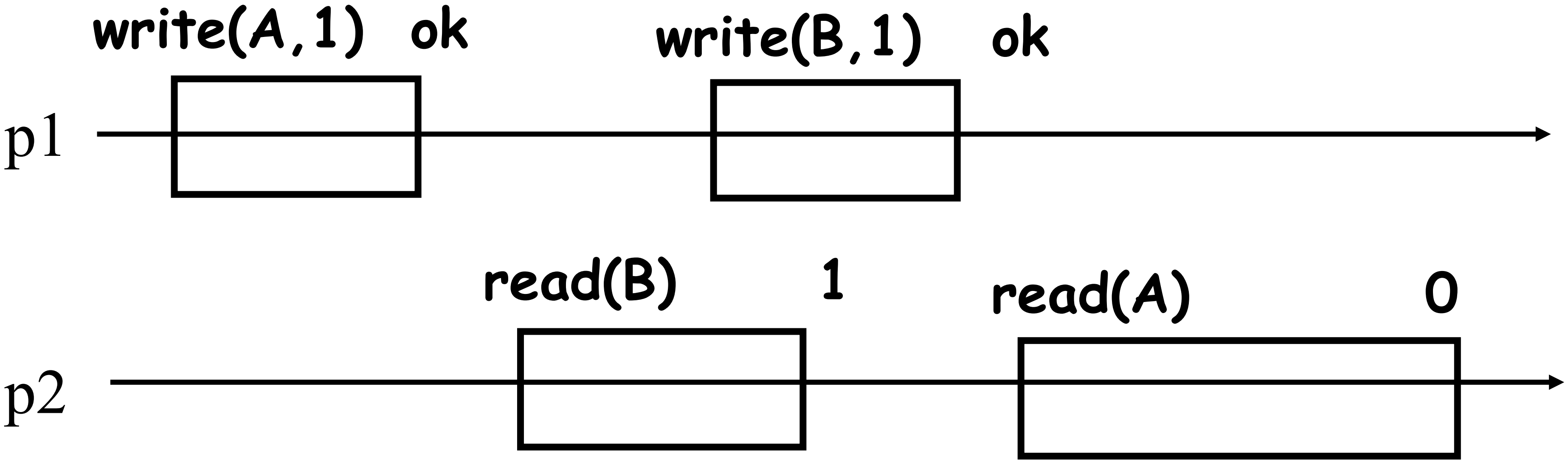
- Any history on two linearizable objects A and B is a history of a linearizable composition (A,B)

- A composition of two registers A and B is a two-field register (A,B)



Sequential consistency is not!

- A composition of sequential consistent objects is not always sequentially consistent!



write(A,1) ok

write(B,1) ok

p1

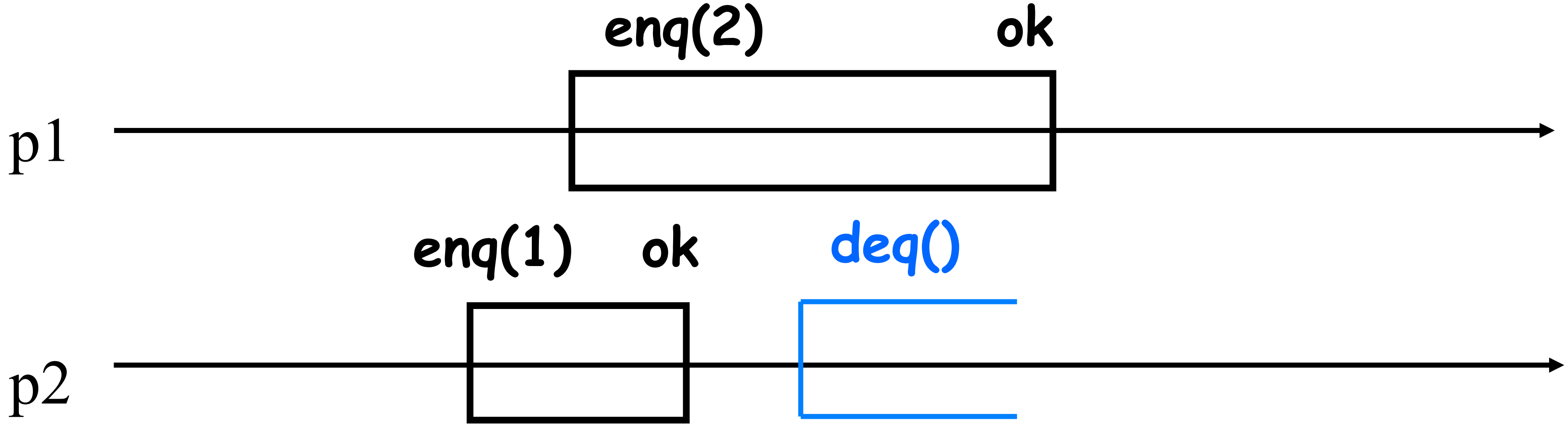
read(B) 1

read(A) 0

p2

Linearizability is nonblocking

Every incomplete operation in a finite history can be independently completed



What safety property is blocking?

Linearizability as safety

- Prefix-closed: every prefix of a linearizable history is linearizable
- Limit-closed: the limit of a sequence of linearizable histories is linearizable

(see Chapter 2 of the lecture notes)

An implementation is linearizable if and only if all its finite histories are linearizable

Why not using one lock?

- Simple – automatic transformation of the sequential code
- Correct – linearizability for free
- In the best case, starvation-free: if the lock is "fair" and every process cooperates, every process makes progress
- Not robust to failures/asynchrony
  - ✓ Cache misses, page faults, swap outs
- Fine-grained locking?
  - ✓ Complicated/prone to deadlocks

Liveness properties

- *Deadlock-free:*
  - ✓If every process is correct*, some process makes progress**
- *Starvation-free:*
  - ✓If every process is correct, every process makes progress

- *Lock-free* (sometimes called *non-blocking):*
  - ✓Some correct process makes progress
- *Wait-free:*
  - ✓Every correct process makes progress
- *Obstruction-free:*
  - ✓Every process makes progress if it executes in isolation (it is the only correct process)

* A process is correct if it takes infinitely many steps.

** Completes infinitely many operations.

Periodic table of liveness properties

[© 2013 Herlihy&Shavit]

	independent non-blocking	**dependent non-blocking**	**dependent blocking**
every process makes progress	wait-freedom	obstruction-freedom	starvation-freedom
some process makes progress	lock-freedom	?	deadlock-freedom

What are the relations (weaker/stronger) between these progress properties?

Liveness properties: relations

Property A is stronger than property B if every run satisfying A also satisfies B (A is a subset of B).

A is strictly stronger than B if, additionally, some run in B does not satisfy A, i.e., A is a proper subset of B.

For example:

- WF is stronger than SF

Every run that satisfies WF also satisfies SF: every correct process makes progress (regardless whether processes cooperate or not).

WF is actually strictly stronger than SF. Why?

- SF and OF are incomparable (none of them is stronger than the other)

There is a run that satisfies SF but not OF: the run in which p1 is the only correct process but does not make progress.

There is a run that satisfies OF but not SF: the run in which every process is correct but no process makes progress

Quiz 2: liveness

- Show how the elements of the “periodic table of progress” are related to each other

  - ✓Hint: for each pair of properties, A and B, check if any run of A is a run of B (A is stronger than B), or if there exists a run of A that is not in B (A is not stronger than B)
  - ✓Can be shown by transitivity: if A is stronger than B and B is stronger than C, then A is stronger than C