

Distributed Computing with Shared Memory

Introduction

MPRI, P1, 2019

Administrivia

- Module: 2-18-2
- Petr Kuznetsov, Carole Deporte, Hugues Fauconnier
- Language: français
- Lectures: Mondays (16.09-04.11), 8:45-12:00 (1014, Bat. Sophie Germain)
- Web page: wiki
- Homework assignment (to submit by October 14)
- Credit = $0.25 * HW + 0.75 * \text{written exam}$ (18.11)
 - ✓ Bonus for participation/discussion of exercises
 - ✓ Bonus (up to 3 points) for bugs found in slides/lecture notes
- Stages: theory and practice of distributed computing:
contact petr.kuznetsov@telecom-paristech.fr

Literature

- Lecture notes: Algorithms for concurrent systems. R. Guerraoui, P. Kuznetsov, link on the wiki
- M. Herlihy and N. Shavit. The art of multiprocessor programming. Morgan Kaufman, 2008
- H. Attiya, J. Welch. Distributed Computing: Fundamentals, Simulations and Advanced Topics (2nd edition). Wiley. 2004

The field of concurrent computing has gained in importance after major chip manufacturers switched their focus from increasing the speed of individual processors to increasing the number of processors on a chip. The computer industry has thus been calling for a software revolution: the concurrency revolution. A major challenge underlying this paradigm shift is creating a library of abstractions that developers can use for general purpose concurrent programming. We study in this book how to define and build such abstractions in a rigorous manner. We focus on those that are considered the most difficult to get right and have the highest impact on the overall performance of a program: synchronization abstractions, also called shared objects or concurrent data structures. The book is intended for software developers and students. It began as a set of lecture notes for courses given at EPFL and Télécom Paris.

Rachid Guerraoui is Professor of Distributed Computing at Ecole Polytechnique Fédérale de Lausanne. He got a PhD from University of Orsay in 1992 and has been affiliated with HP Labs, MIT and Collège de France.

Petr Kuznetsov is Professor of Computer Science at Télécom ParisTech, Université Paris-Saclay, France. He received his PhD from École Polytechnique Fédérale de Lausanne (EPFL) in 2005. Before joining Télécom ParisTech, he worked at Max Planck Institute for Software Systems and Deutsche Telekom Innovation Labs/Technical University of Berlin.



EPFL
Press

Rachid Guerraoui
Petr Kuznetsov

ALGORITHMS FOR CONCURRENT SYSTEMS

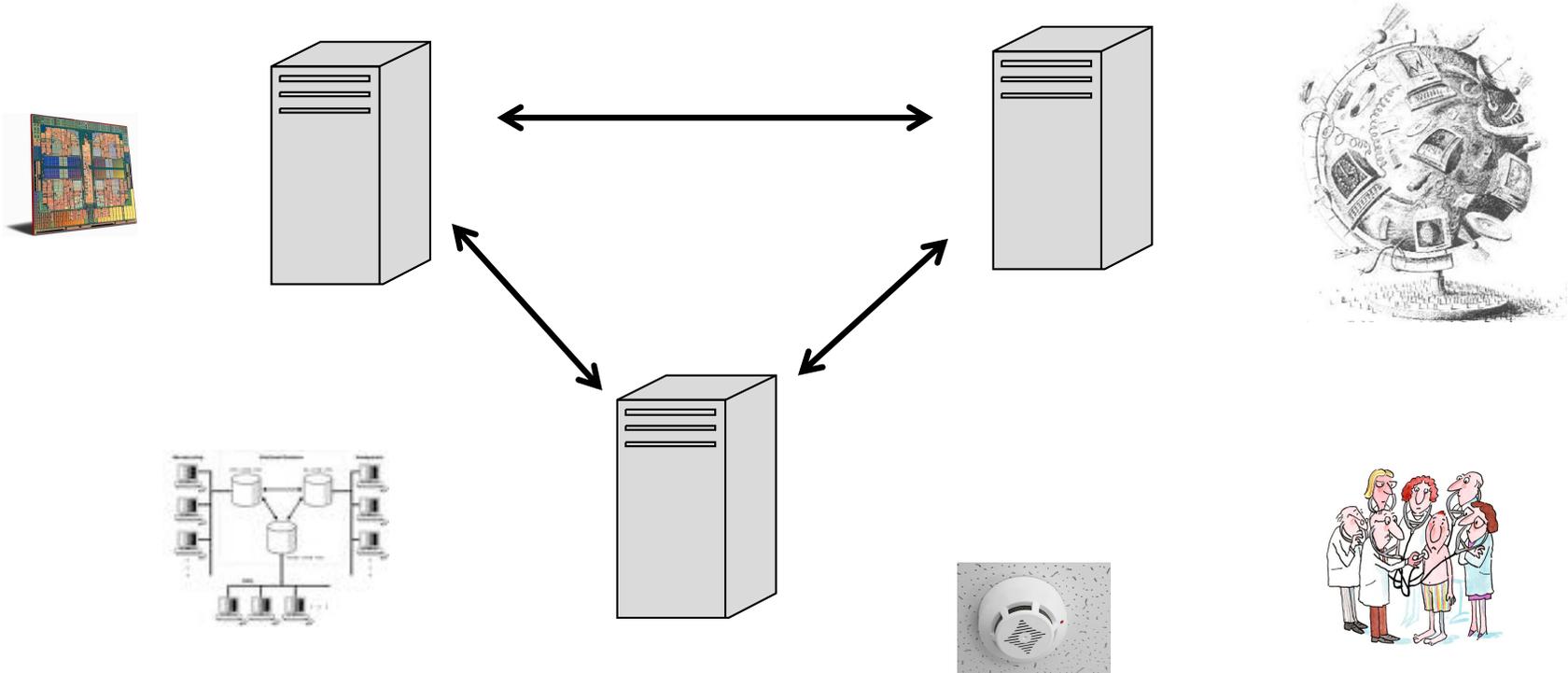
Rachid Guerraoui
Petr Kuznetsov



Librairie Eyrolles, 55-57-61, Blvd Saint-Germain, 75005 Paris
Section «Informatique-Algorithmique»

This course is about distributed
computing:
independent sequential **processes**
that communicate

Concurrency is everywhere!



- Multi-core processors
- Sensor networks
- Internet
- Basically everything related computing

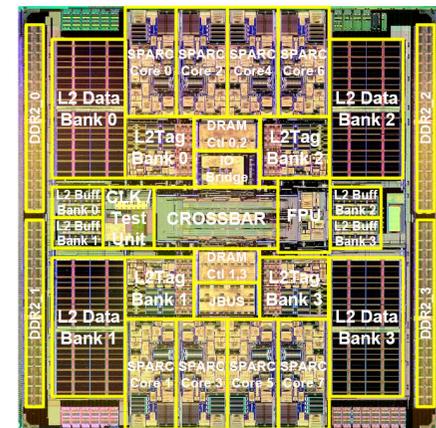
Communication models

- Shared memory
 - ✓ Processes apply (read–write) operations on shared variables
 - ✓ Failures and asynchrony
- Message passing
 - ✓ Processes send and receive messages
 - ✓ Communication graphs
 - ✓ Message delays



- Single-processor performance does not improve
- But we can add more cores
- Run concurrent code on multiple processors

Can we expect a proportional speedup? (ratio between sequential time and parallel time for executing a job)



Amdahl's Law



- p – fraction of the work that can be done in parallel (no synchronization)
- n - the number of processors
- Time one processor needs to complete the job = 1

$$S = \frac{1}{1 - p + p/n}$$

Challenges

- What is a **correct** implementation?
 - ✓ Safety and liveness
- What is the **cost** of synchronization?
 - ✓ Time and space lower bounds
- Failures/asynchrony
 - ✓ Fault-tolerant concurrency?
- How to distinguish possible from impossible?
 - ✓ Impossibility results

Distributed \neq Parallel

- The main challenge is **synchronization**
- “you know you have a distributed system when the crash of a computer you’ve never heard of stops you from getting any work done” (Lamport)



History

- Dining philosophers, mutual exclusion (Dijkstra) ~60' s
- Distributed computing, logical clocks (Lamport), distributed transactions (Gray) ~70' s
- Consensus (Lynch) ~80' s
- Distributed programming models, since ~90' s
- Multicores now

Course outline:

- I. Synchronization, blocking and non-blocking
 - ✓ Introduction, theory and practice of distributed systems
 - ✓ Correctness: Safety and Liveness
 - ✓ Synchronization techniques; mutual exclusion

- I. Read-write synchronization
 - ✓ Safe, regular, and atomic registers
 - ✓ Atomic and immediate snapshot

- II. Consensus
 - ✓ Consensus hierarchy
 - ✓ Distributed tasks: k-set agreement, renaming
 - ✓ Simulation of Borowsky and Gafni, with applications.

Synchronization, blocking and non-blocking

MPRI, period 1,

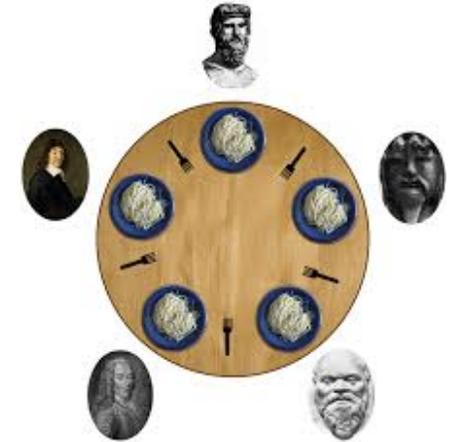
Why synchronize ?

- Concurrent access to a shared resource may lead to an inconsistent state
 - ✓ E. g., concurrent file editing
 - ✓ Non-deterministic result (**race condition**): the resulting state depends on the scheduling of processes
- Concurrent accesses need to be **synchronized**
 - ✓ E. g., decide who is allowed to update a given part of the file at a given time
- Code leading to a race condition is called **critical section**
 - ✓ Must be executed sequentially
- **Synchronization problems**: mutual exclusion, readers-writers, producer-consumer, ...





Dining philosophers (Dijkstra, 1965)



Edsger Dijkstra
1930-2002

- To **make progress** (to eat) each **process** (philosopher) needs two **resources** (forks)
- **Mutual exclusion**: no fork can be shared
- Progress conditions:
 - ✓ Some philosopher does not starve (**deadlock-freedom**)
 - ✓ No philosopher starves (**starvation-freedom**)

Mutual exclusion

- No two processes are in their critical sections (CS) at the same time
- +
 - Deadlock-freedom: **at least one** process eventually enters its CS
 - Starvation-freedom: **every** process eventually enters its CS
 - ✓ Assuming no process **blocks** in **CS** or **Entry section**
- Originally: implemented by reading and writing
 - ✓ Peterson's lock, Lamport's bakery algorithm
- Currently: in hardware (mutex, semaphores)

Peterson's lock: 2 processes

```
bool flag[0] = false;
bool flag[1] = false;
int turn;
```

P0:

```
flag[0] = true;
turn = 1;
while (flag[1] and turn==1)
{
    // busy wait
}
// critical section
...
// end of critical section
flag[0] = false;
```

P1:

```
flag[1] = true;
turn = 0;
while (flag[0] and turn==0)
{
    // busy wait
}
// critical section
...
// end of critical section
flag[1] = false;
```

Peterson's lock: $N \geq 2$ processes

```
// initialization
level[0..N-1] = {-1}; // current level of processes 0..N-1
waiting[0..N-2] = {-1}; // the waiting process in each level
// 0...N-2

// code for process i that wishes to enter CS
for (m = 0; m < N-1; ++m) {
    level[i] = m;
    waiting[m] = i;
    while(waiting[m] == i &&(exists k ≠ i: level[k] ≥ m)) {
        // busy wait
    }
}
// critical section
level[i] = -1; // exit section
```

Bakery [Lamport'74,simplified]

```
// initialization
flag: array [1..N] of bool = {false};
label: array [1..N] of integer = {0}; //assume no bound

// code for process i that wishes to enter CS

flag[i] = true; //enter the "doorway"
label[i] = 1 + max(label[1], ..., label[N]); //pick a ticket
//leave the "doorway"
while (for some k ≠ i: flag[k] and (label[k],k)<<(label[i],i));
// wait until all processes "ahead" are served
...
// critical section
...
flag[i] = false; // exit section
```

Processes are served in the “ticket order”: first-come-first-serve

Bakery [Lamport'74,original]

```
// initialization
flag: array [1..N] of bool = {false};
label: array [1..N] of integer = {0}; //assume no bound

// code for process i that wishes to enter CS
flag[i] = true; //enter the doorway
label[i] = 1 + max(label[1], ..., label[N]); //pick a ticket
flag[i] = false; //exit the doorway
for j=1 to N do {
    while (flag[j]); //wait until j is not in the doorway
    while (label[j]≠0 and (label[j],j)<<(label[i],i));
    // wait until j is not "ahead"
}
...
// critical section
...
label[i] = 0; // exit section
```

Ticket withdrawal is “protected” with flags: a very useful trick:
works with “safe” (non-atomic) shared variables

Black-White Bakery [Taubenfeld'04]

```
// initialization
color: {black,white};
flag: array [1..N] of bool = {false};
label[1..N]: array of type {0,...,N} = {0} //bounded ticket numbers
mycolor[1..N]: array of type {black,white}

// code for process i that wishes to enter CS
flag[i] = true; //enter the "doorway"
mycolor[i] =color;
label[i] = 1 + max({label[j]| j=1,...,N: mycolor[i]=mycolor[j]});
flag[i] = false; //exit the "doorway"
for j=1 to N do
    while (flag[j]);
    if mycolor[j]=mycolor[i] then
        while (label[j]≠0 and (label[j],j)<<(label[i],i) and mycolor[j]=mycolor[i] );
    else
        while (label[j]≠0 and mycolor[i]=color and mycolor[j] ≠ mycolor[i]);
// wait until all processes "ahead" of my color are served
...
// critical section
...
if mycolor[i]=black then color = white else color = black;
label[i] = 0; // exit section
```

Colored tickets => bounded variables!

Readers-writers problem

- Writer updates a file
- Reader keeps itself up-to-date
- Reads and writes are non-atomic!

Why synchronization? Inconsistent values might be read

Writer

```
T=0: write("sell the  
cat")
```

```
T=2: write("wash the  
dog")
```

Reader

```
T=1: read("sell ...")
```

```
T=3: read("... the dog")
```

Sell the dog?

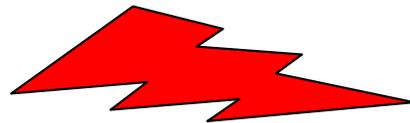
Producer-consumer (bounded buffer) problem

- Producers **put** items in the buffer (of bounded size)
- Consumers **get** items from the buffer
- Every item is consumed, no item is consumed twice
(Client-server, multi-threaded web servers, pipes, ...)

Why synchronization? Items can get lost or consumed twice:

Producer

```
/* produce item */  
while (counter==MAX);  
buffer[in] = item;  
in = (in+1) % MAX;  
counter++;
```



Race!

Consumer

```
/* to consume item */  
while (counter==0);  
item=buffer[out];  
out=(out+1) % MAX;  
counter--;  
/* consume item */
```

Synchronization tools

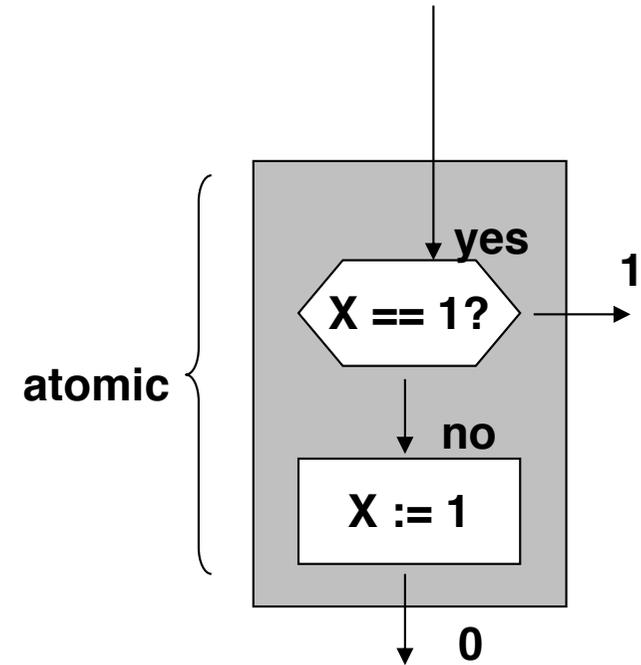
- Busy-waiting (TAS)
- Semaphores (locks), monitors
- Nonblocking synchronization
- Transactional memory

Busy-wait: Test and Set

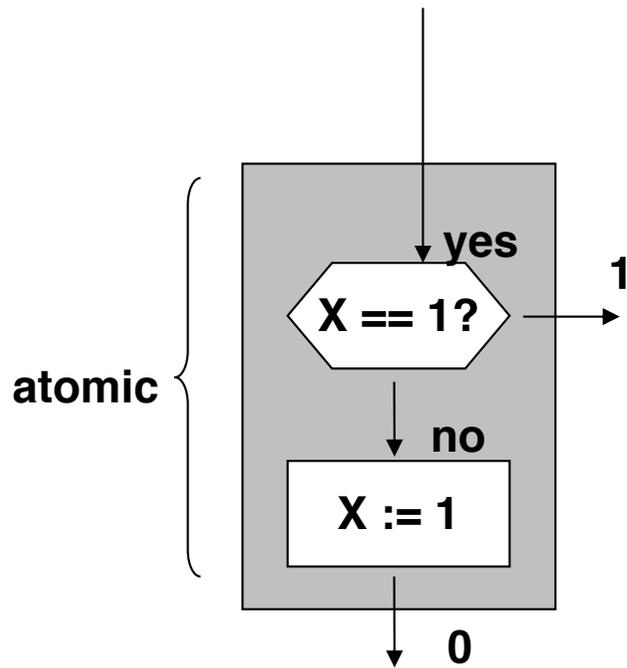
- **TAS(X)** **tests** if $X = 1$, **sets** X to 1 if not, and returns the old value of X
 - ✓ Instruction available on almost all processors

TAS(X):

```
atomic {  
  if X == 1 return 1;  
  X = 1;  
  return 0;  
}
```



Busy-wait: Test and Set



shared X:=0	
Producer	Consumer
<pre> while(counter==MAX); . . . buffer[in] = item; . . . while TAS(X); counter++; X:=0; . . . </pre>	<pre> while (counter==0); . . . item = buffer[out]; . . . while TAS(X); counter--; X:=0; . . . </pre>

Problems:

- busy waiting
- no record of request order (for multiple producers and consumers)

Semaphores [Dijkstra 1968]: specification

- A semaphore S is an integer variable accessed (apart from initialization) with two atomic operations $P(S)$ and $V(S)$
 - ✓ Stands for “passeren” (to pass) and “vrijgeven” (to release) in Dutch
- The value of S indicates the number of resource elements available (if positive), or the number of processes waiting to acquire a resource element (if negative)

```
Init(S,v){ S := v; }
```

```
P(S){  
    while S<=0;    /* wait until a resource is available */  
    S--;          /* pass to a resource */  
}
```

```
V(S){  
    S++;          /* release a resource */  
}
```

Semaphores: implementation

S is associated with a composite object:

- ✓ S.counter: the **value** of the semaphore
- ✓ S.wq: the **waiting queue**, memorizing the processes having requested a resource element

```
Init(S,R_nb) {
    S.counter=R_nb;
    S.wq=empty;
}
P(S) {
    S.counter--;
    if S.counter<0{
        put the process in S.wq and wait until
        READY;}
}
V(S) {
    S.counter++
    if S.counter>=0{
        mark 1st process in S.wq
        as        READY;}
}
```

Lock

- A semaphore initialized to 1, is called a **lock** (or **mutex**)
- When a process is in a critical section, no other process can come in

shared semaphore S := 1

Producer	Consumer
<pre>while (counter==MAX); ... buffer[in] = item; ... P(S); counter++; V(S) ...</pre>	<pre>while (counter==0); ... item = buffer[out]; ... P(S); counter--; V(S); ...</pre>

Problem: still waiting until the buffer is ready

Semaphores for producer-consumer

- 2 semaphores used :
 - ✓ **empty**: indicates empty slots in the buffer (to be used by the producer)
 - ✓ **full**: indicates full slots in the buffer (to be read by the consumer)

```
shared semaphores empty := MAX, full := 0;
```

Producer

```
P(empty)
buffer[in] = item;
in = (in+1) % MAX;
V(full)
```

Consumer

```
P(full);
item = buffer[out];
out=(out+1) % MAX;
V(empty);
```

Potential problems with semaphores/locks

- **Blocking:** progress of a process is conditional (depends on other processes)
- **Deadlock:** no progress ever made

X1:=1; X2:=1

Process 1	Process 2
...	...
P(X1)	P(X2)
P(X2)	P(X1)
critical section	critical section
V(X2)	V(X1)
V(X1)	V(X2)
...	...

- **Starvation:** requests blocked in the waiting queue forever

Other problems of blocking synchronization

- Priority inversion
 - ✓ High-priority threads blocked
- No robustness
 - ✓ Page faults, cache misses etc.
- Not composable

Can we think of anything else?

Non-blocking algorithms

A process makes progress, **regardless** of the other processes

```
shared buffer[MAX]:=empty; head:=0; tail:=0;
```

Producer put(item)

```
if (tail-head == MAX){  
    return(full);  
}  
buffer[tail%MAX]=item;  
tail++;  
return(ok);
```

Consumer get()

```
if (tail-head == 0){  
    return(empty);  
}  
item=buffer[head%MAX];  
head++;  
return(item);
```

Problems:

- works for 2 processes but hard to say why it works 😊
- multiple producers/consumers? Other synchronization pbs?
(stay in class to learn more)

Transactional memory

- Mark sequences of instructions as an **atomic transaction**, e.g., the resulting producer code:

```
atomic {  
    if (tail-head == MAX){  
        return full;  
    }  
    items[tail%MAX]=item;  
    tail++;  
}  
return ok;
```

- A transaction can be either **committed** or **aborted**
 - ✓ Committed transactions are **serializable**
 - ✓ Let the transactional memory (TM) care about the conflicts
 - ✓ Easy to program, but performance may be problematic

Summary

- Concurrency is indispensable in programming:
 - ✓ Every system is now concurrent
 - ✓ Every parallel program needs to synchronize
 - ✓ Synchronization cost is high (“Amdahl’s Law”)
- Tools:
 - ✓ Synchronization primitives (e.g., monitors, TAS, CAS, LL/SC)
 - ✓ Synchronization libraries (e.g., `java.util.concurrent`)
 - ✓ Transactional memory, also in hardware (Intel Haswell, IBM Blue Gene,...)
- Coming later:
 - ✓ Read-write transformations and snapshot memory
 - ✓ Nonblocking synchronization

Quiz 1

- What if we reverse the order of the first two lines the 2-process Peterson's algorithm

P0:

```
turn = 1;
```

```
flag[0] = true;
```

P1:

```
turn = 0;
```

```
flag[1] = true;
```

...

Would it work?

- Prove that Peterson's N-process algorithm ensures:
 - ✓ mutual exclusion: no two processes are in the critical section at a time
 - ✓ starvation freedom: every process in the trying section eventually reaches the critical section (assuming no process fails in the trying, critical, or exit sections)
- **Extra:** show that the bounded (black-white) Bakery algorithm is correct

Correctness of algorithms: safety and liveness

Basic abstractions

- *Process* abstraction – an entity performing independent computation
- Communication
 - ✓ Message-passing: *channel* abstraction
 - ✓ Shared memory: *objects*

How to treat a (computing) system formally

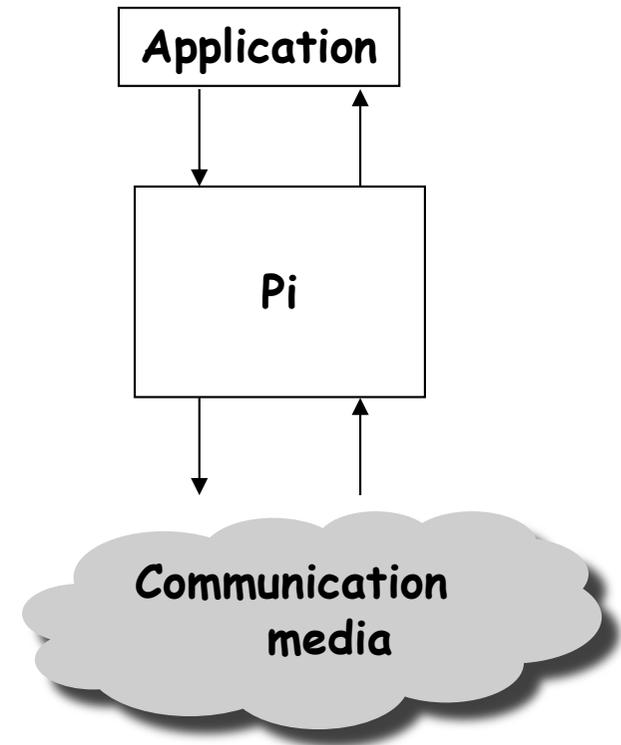
- Define models (tractability, realism)
- Devise abstractions for the system design (convenience, efficiency)
- Devise algorithms and determine complexity bounds

Processes

- Automaton P_i ($i=1, \dots, N$):
 - ✓ States
 - ✓ Inputs
 - ✓ Outputs
 - ✓ Sequential specification

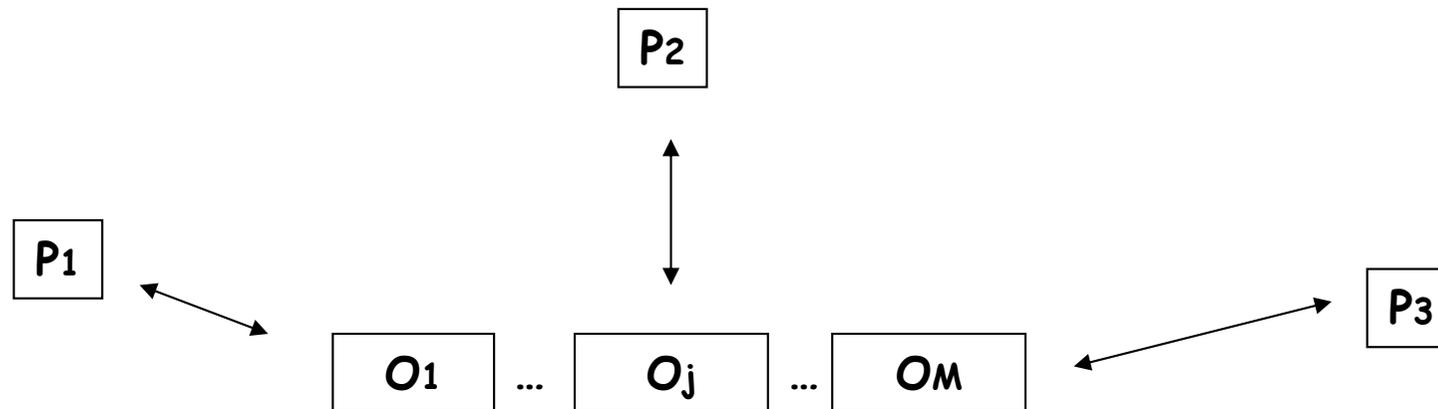
Algorithm = $\{P_1, \dots, P_N\}$

- Deterministic algorithms
- Randomized algorithms



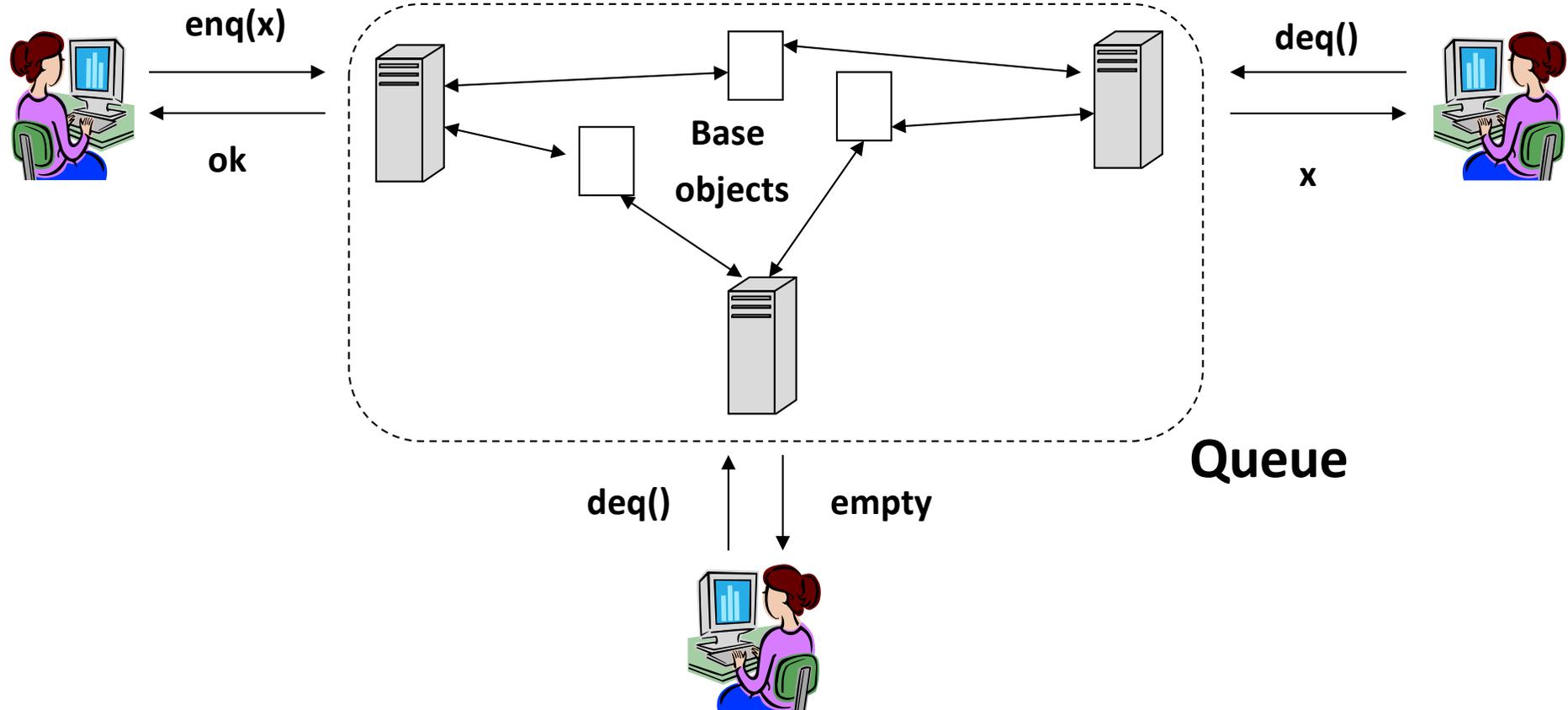
Shared memory

- 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



Correctness

What does it **mean** for an implementation to be correct?

- Safety \approx 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 \approx 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 mutual exclusion) 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^* – the set of finite runs

Σ^ω – 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 $\sigma_0, \sigma_1, \sigma_2, \dots$, 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^*)
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**):

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

- ✓ 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 2: 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 in 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 a mixture 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: linearizability

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;
```

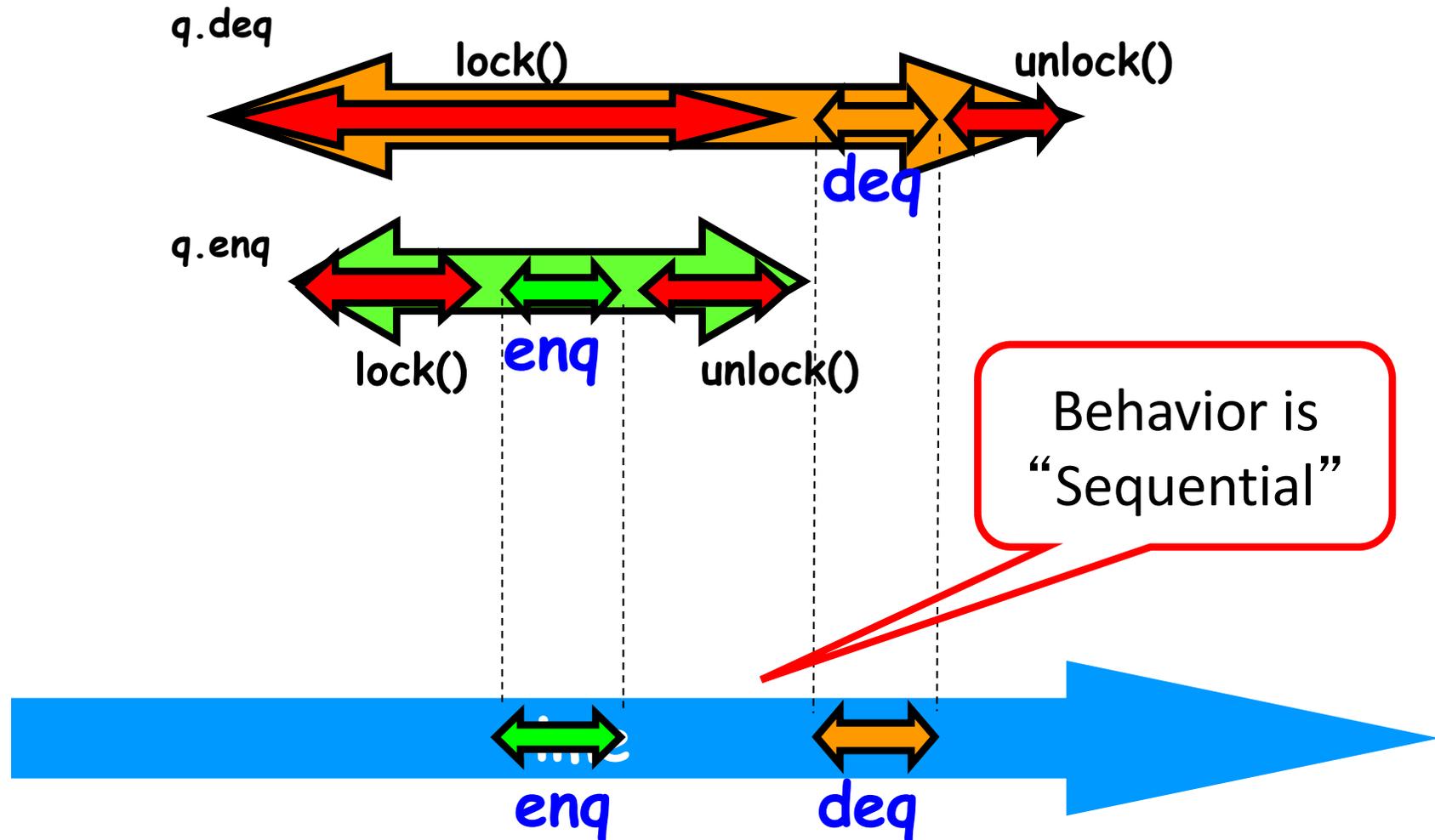
Intuitively...

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



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

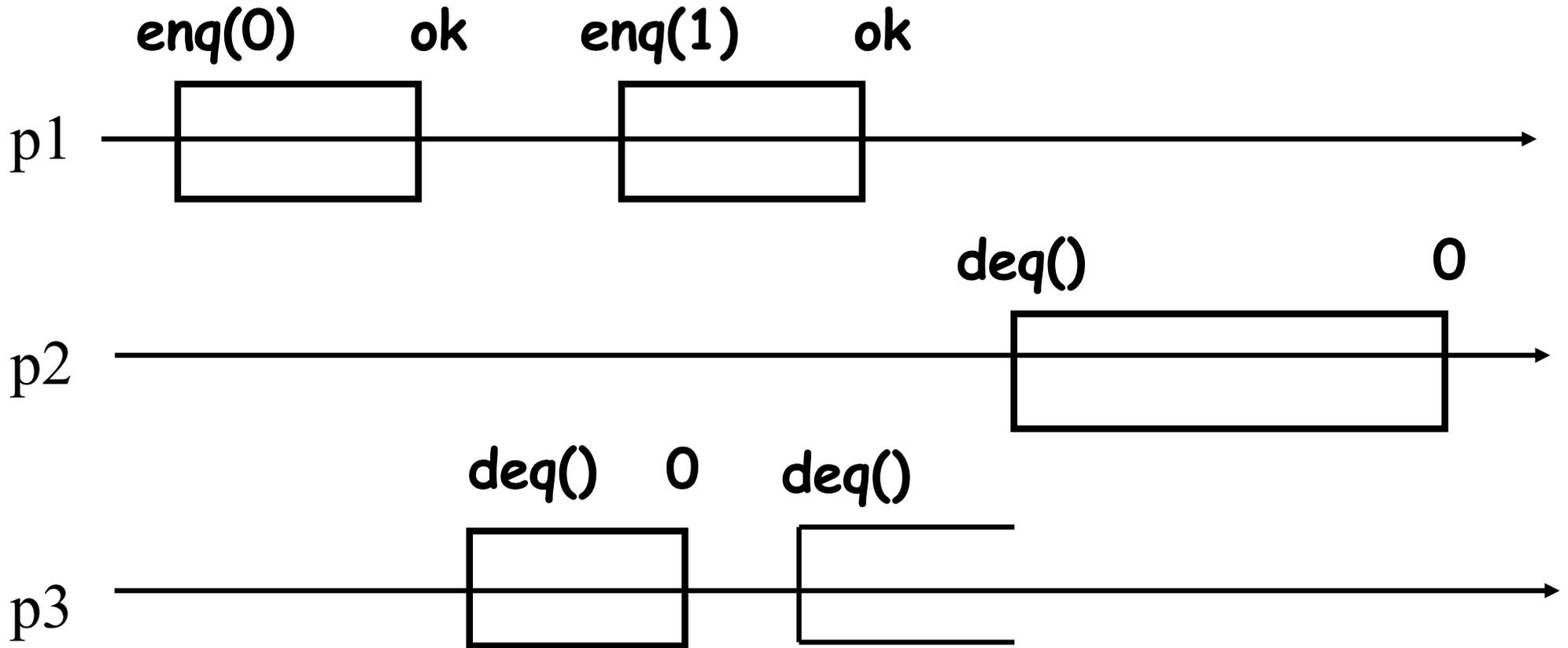
E.g., p1-enq(0), p2-deq(),p1-ok,p2-0,...

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)

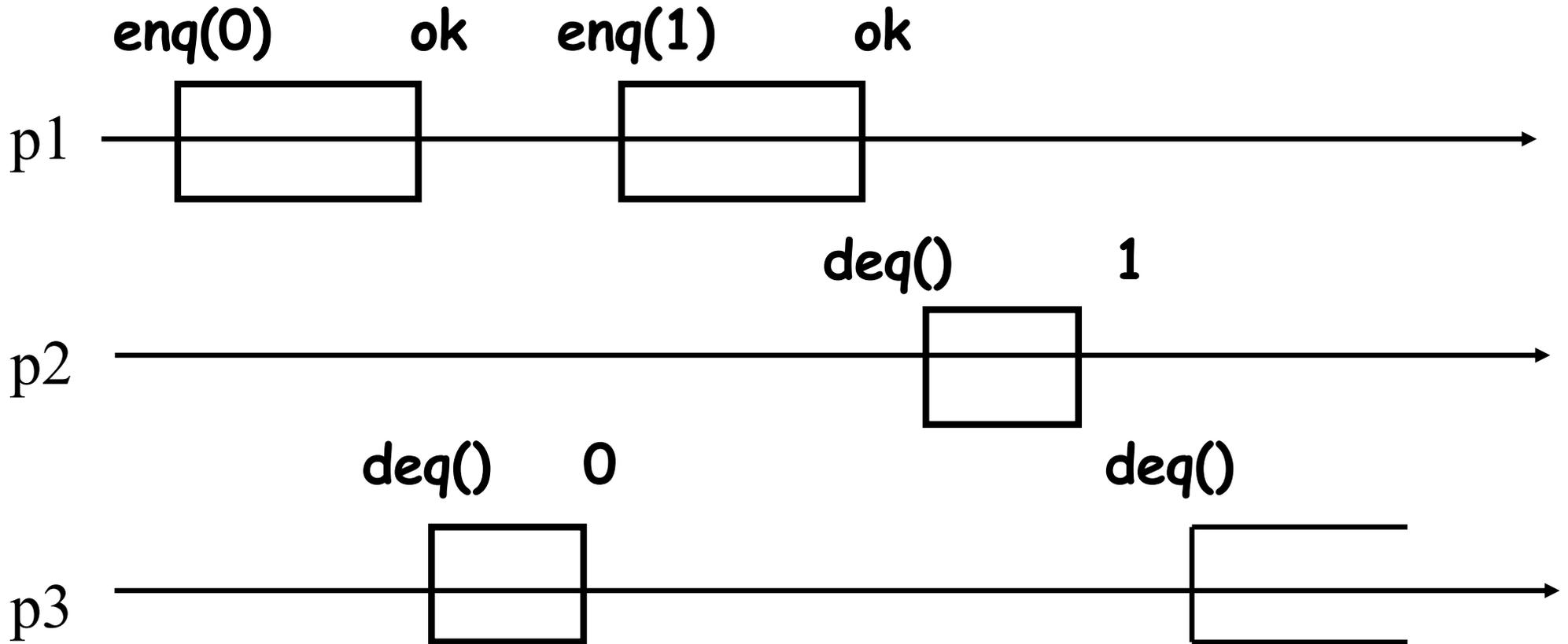
Histories



History:

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

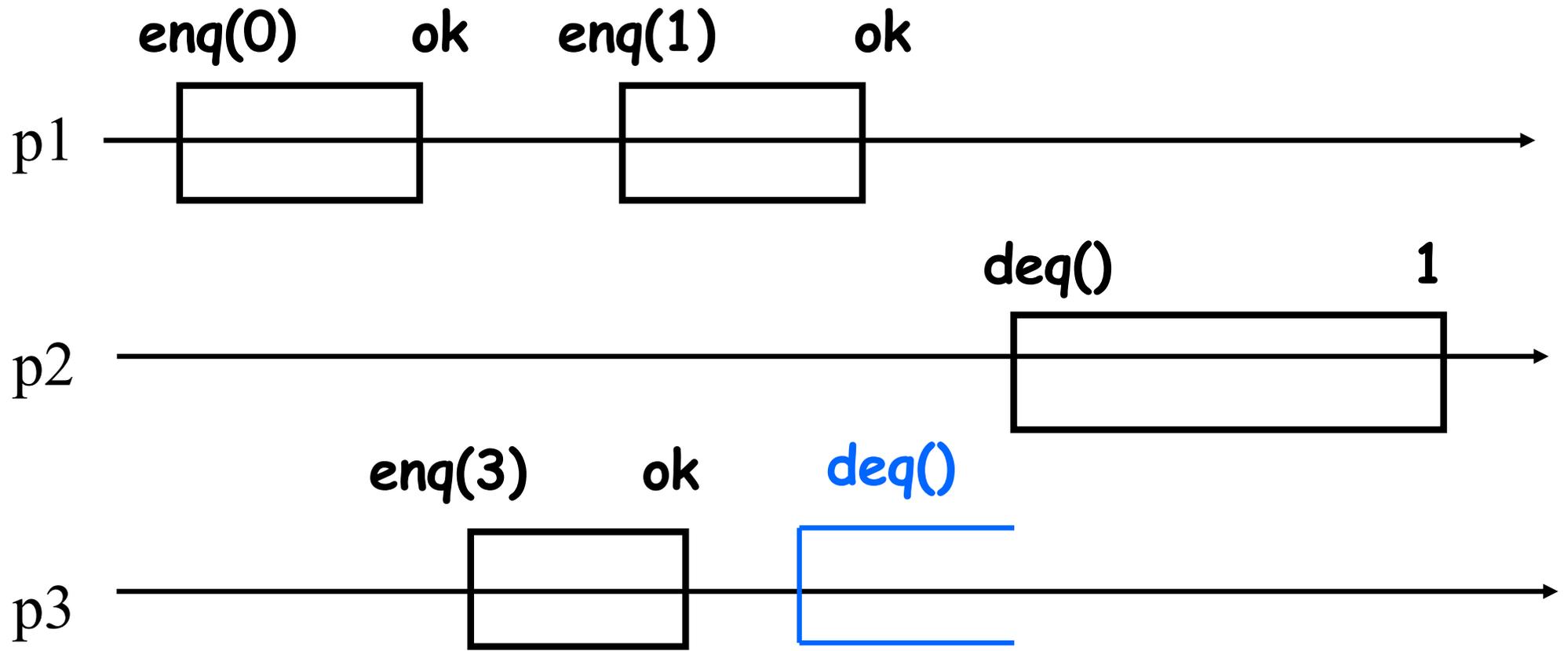
Complete operations and completions

Let H be a history

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

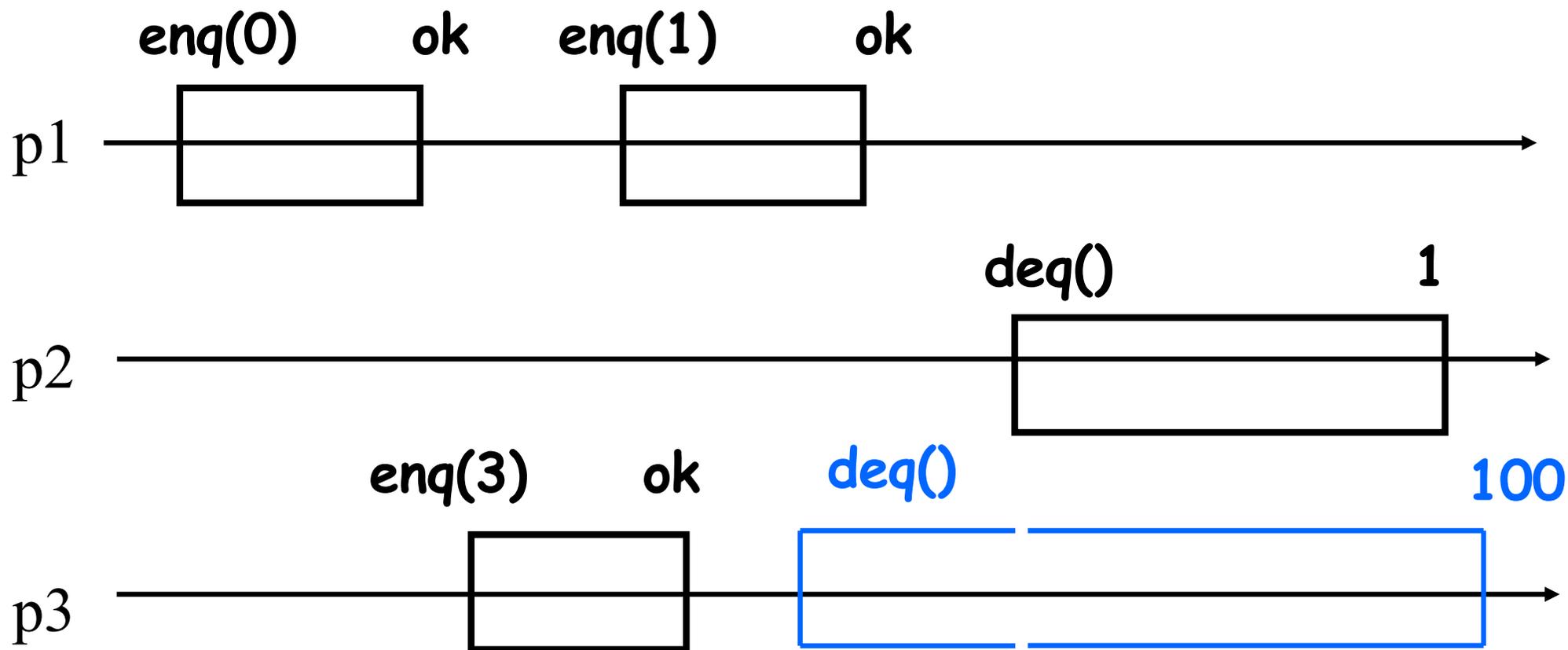
A *completion* of H is a history H' that includes all complete operations of H and a *subset* of incomplete operations of H followed with matching responses

Complete operations and completions



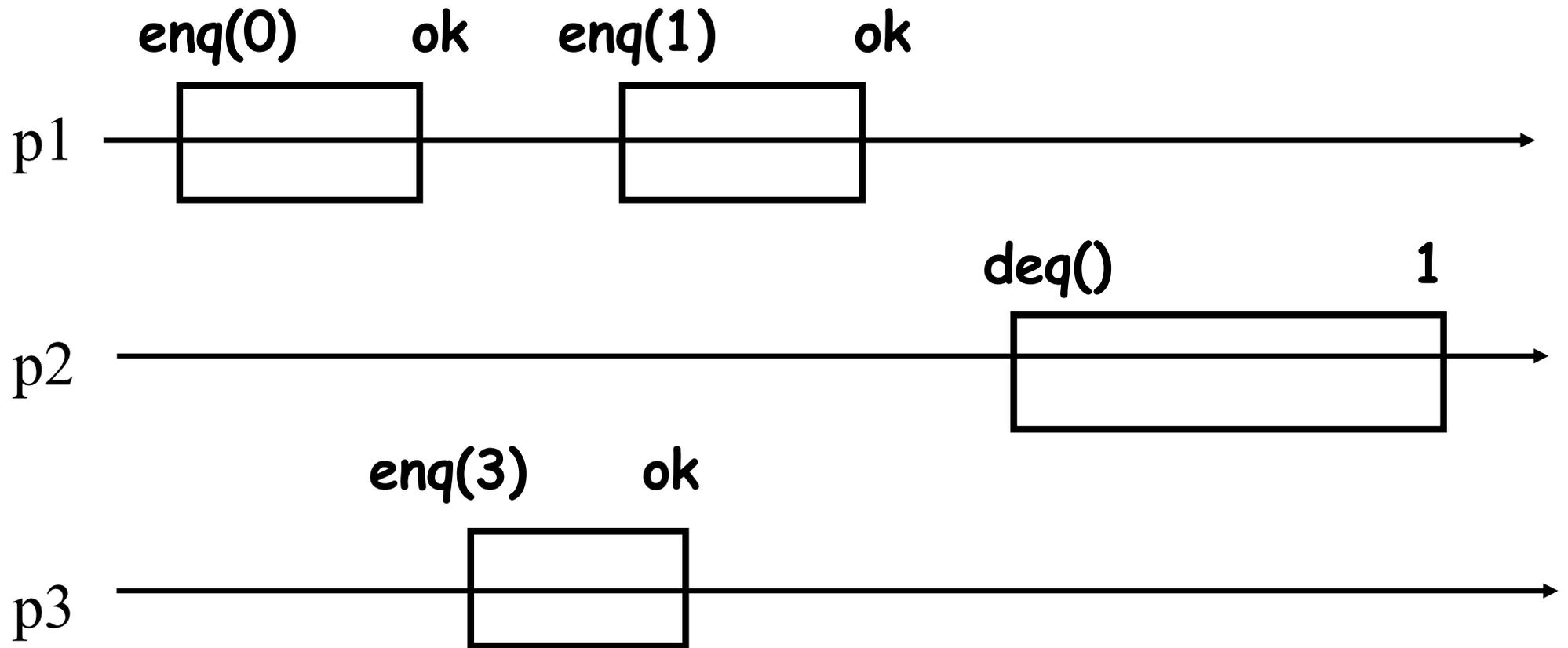
p1-enq(0); p1-ok; p3-enq(3); p1-enq(1); p3-ok;
p3-deq(); p1-ok; p2-deq(); p2-1;

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 p_i

$$H \upharpoonright p_i = H' \upharpoonright p_i$$

E.g.:

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

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

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 \Rightarrow op1 precedes op2 in S

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

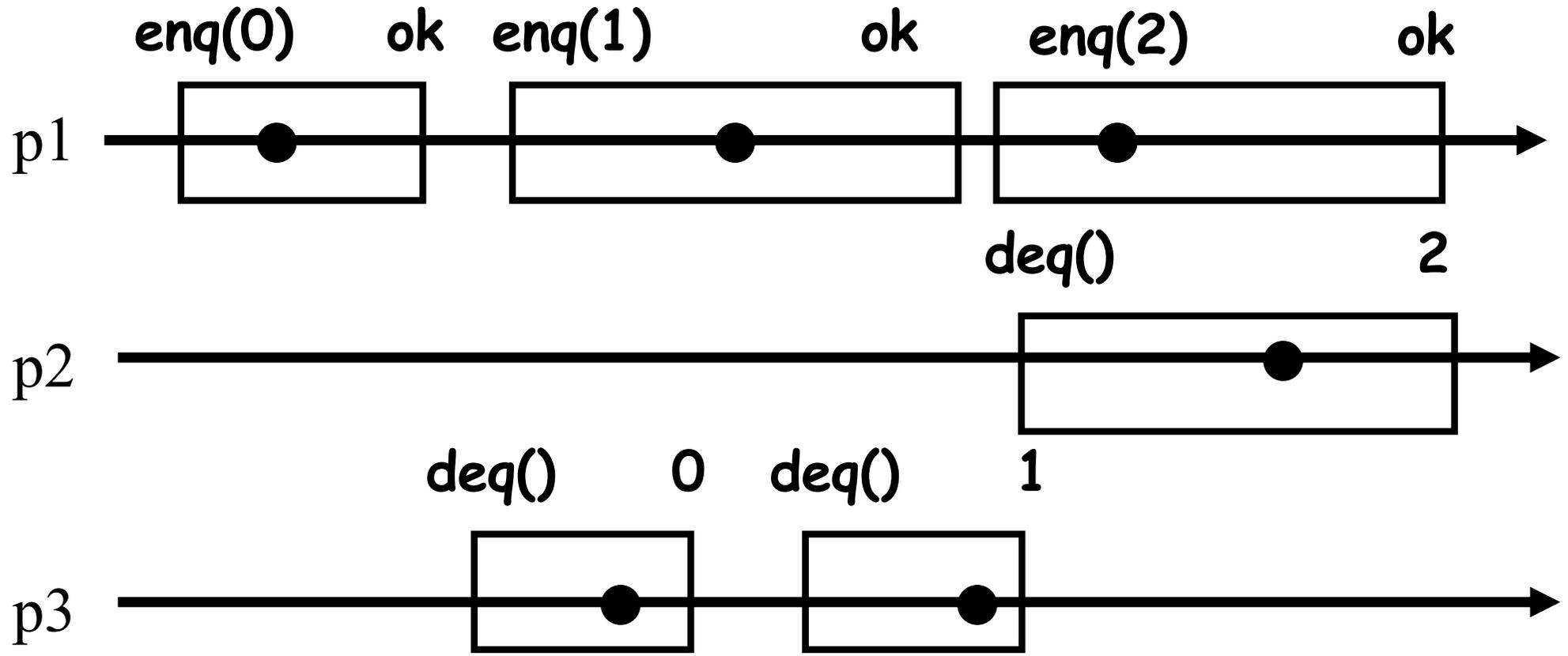
Linearization points

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

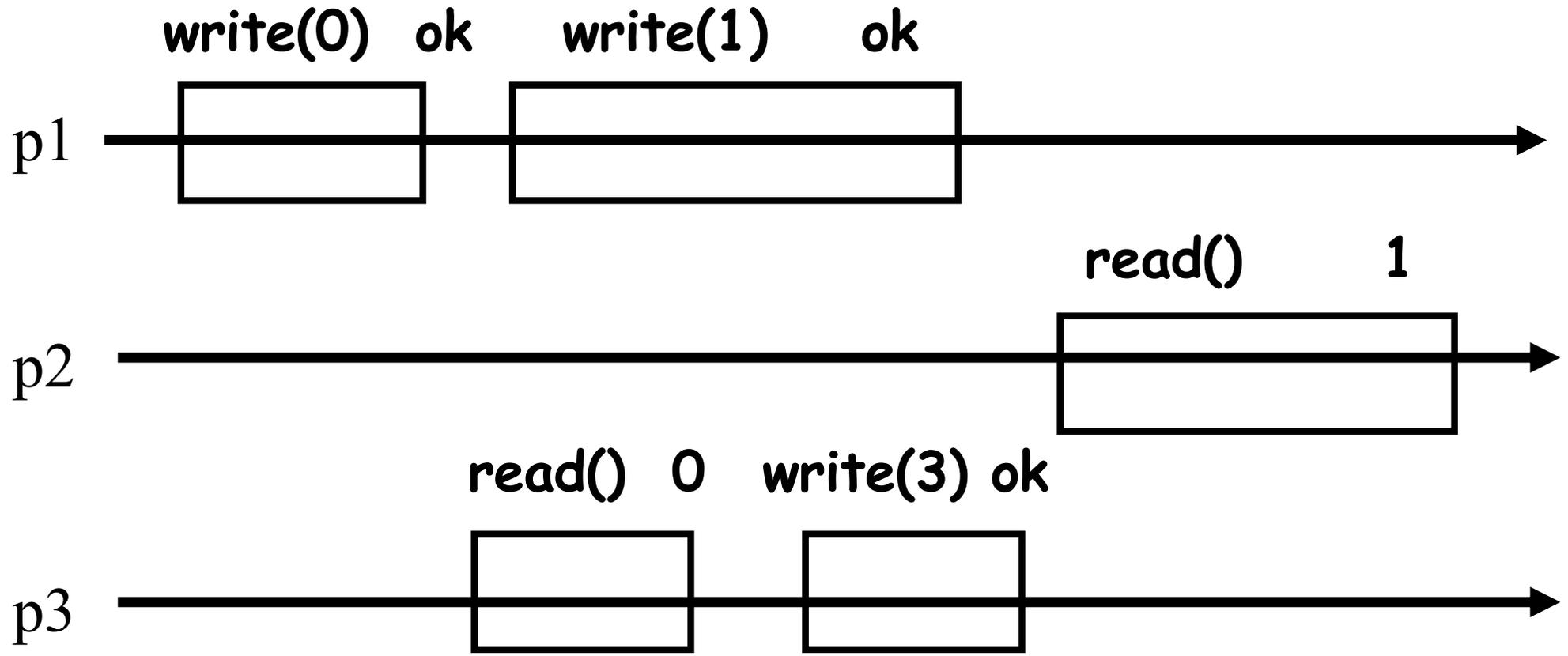
Informally, the complete operations (and some incomplete operations) in a history are seen as *taking effect instantaneously* at some time between their invocations and responses

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

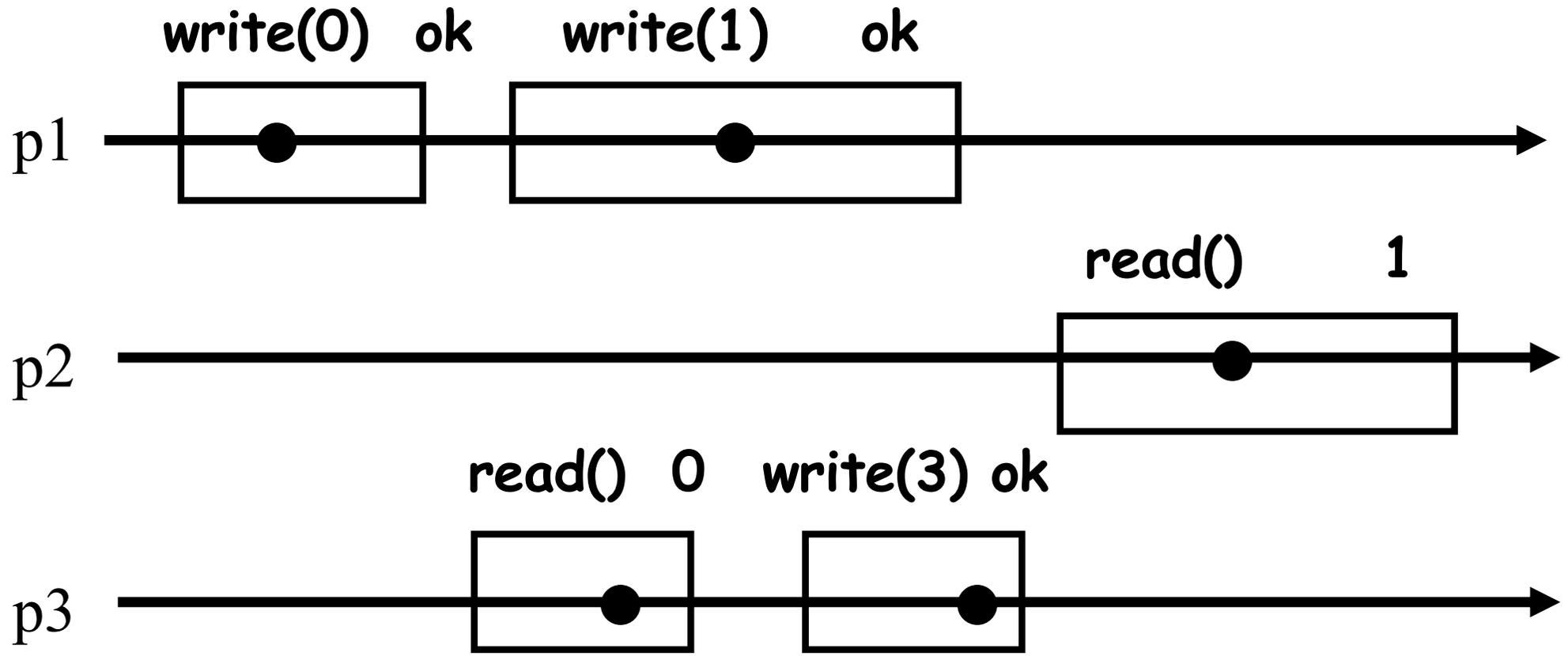
Linearizable?



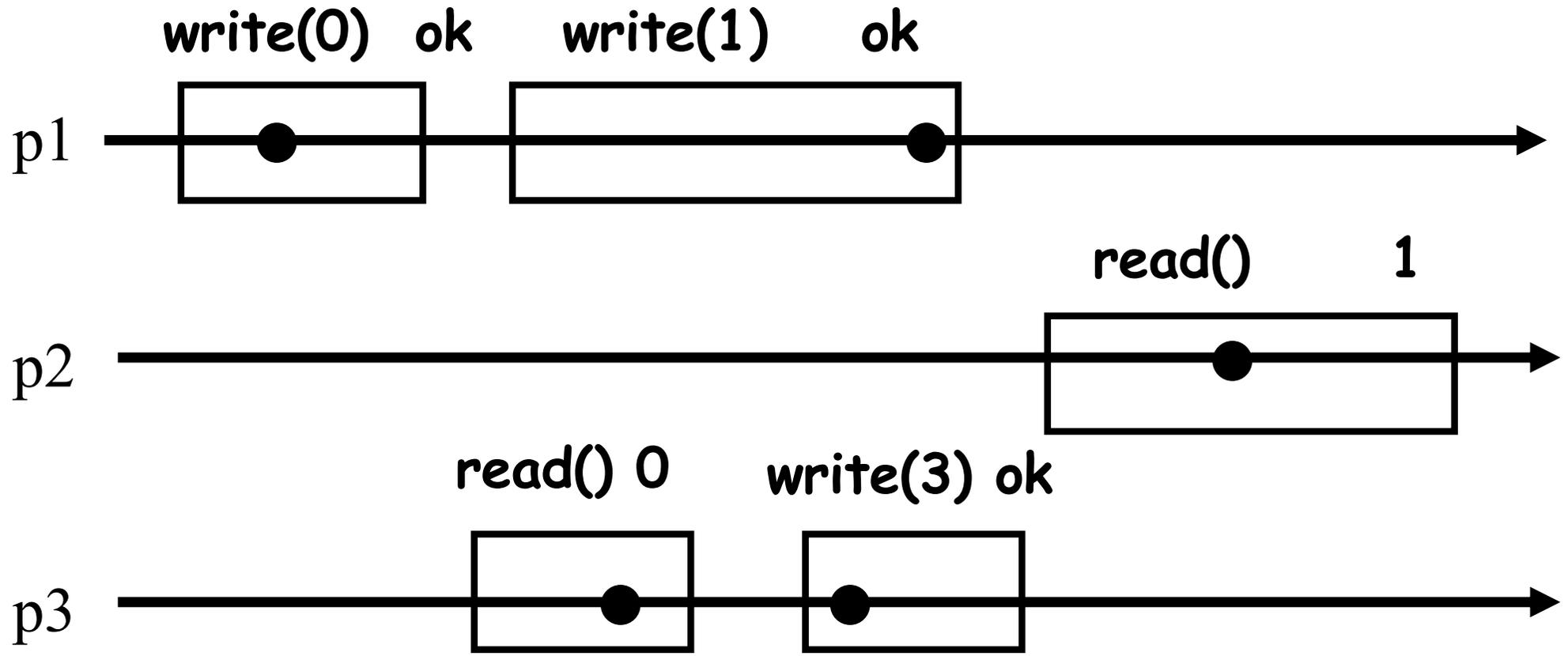
Linearizable?



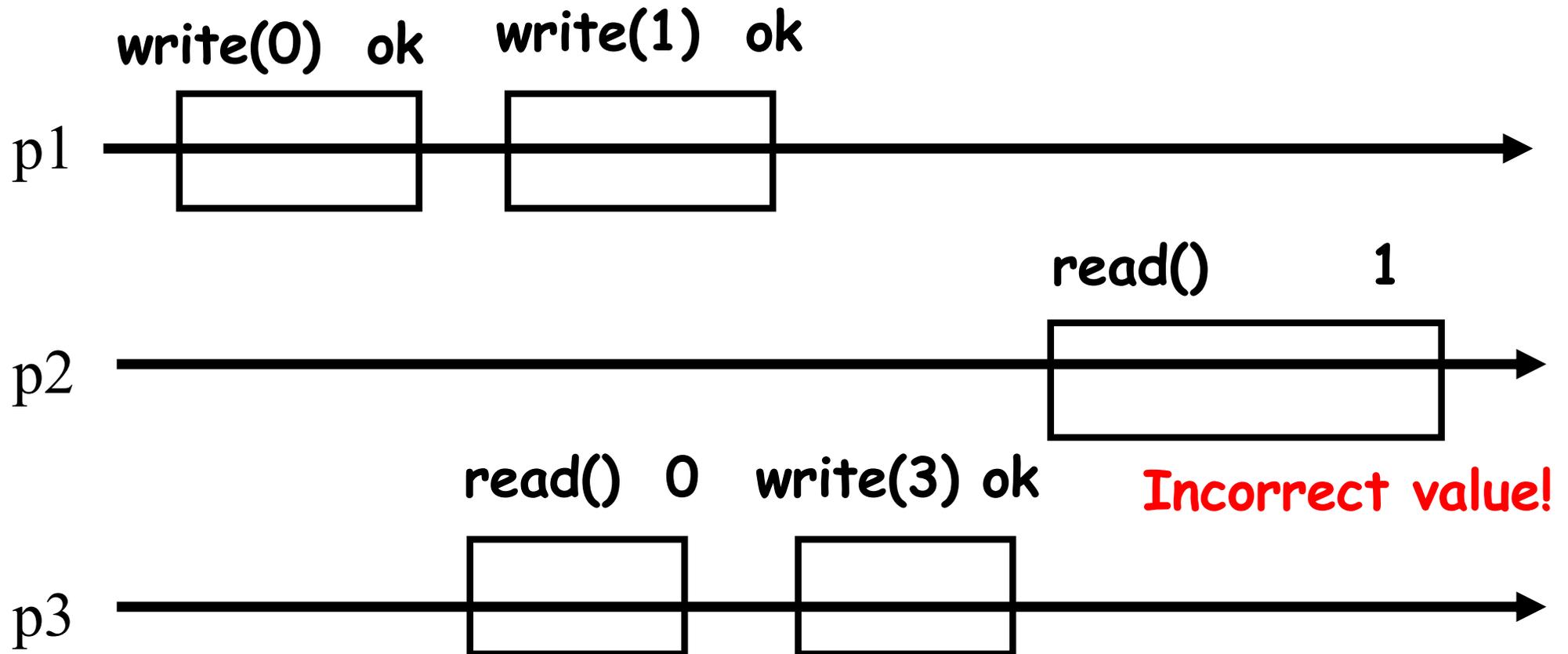
Linearizable?



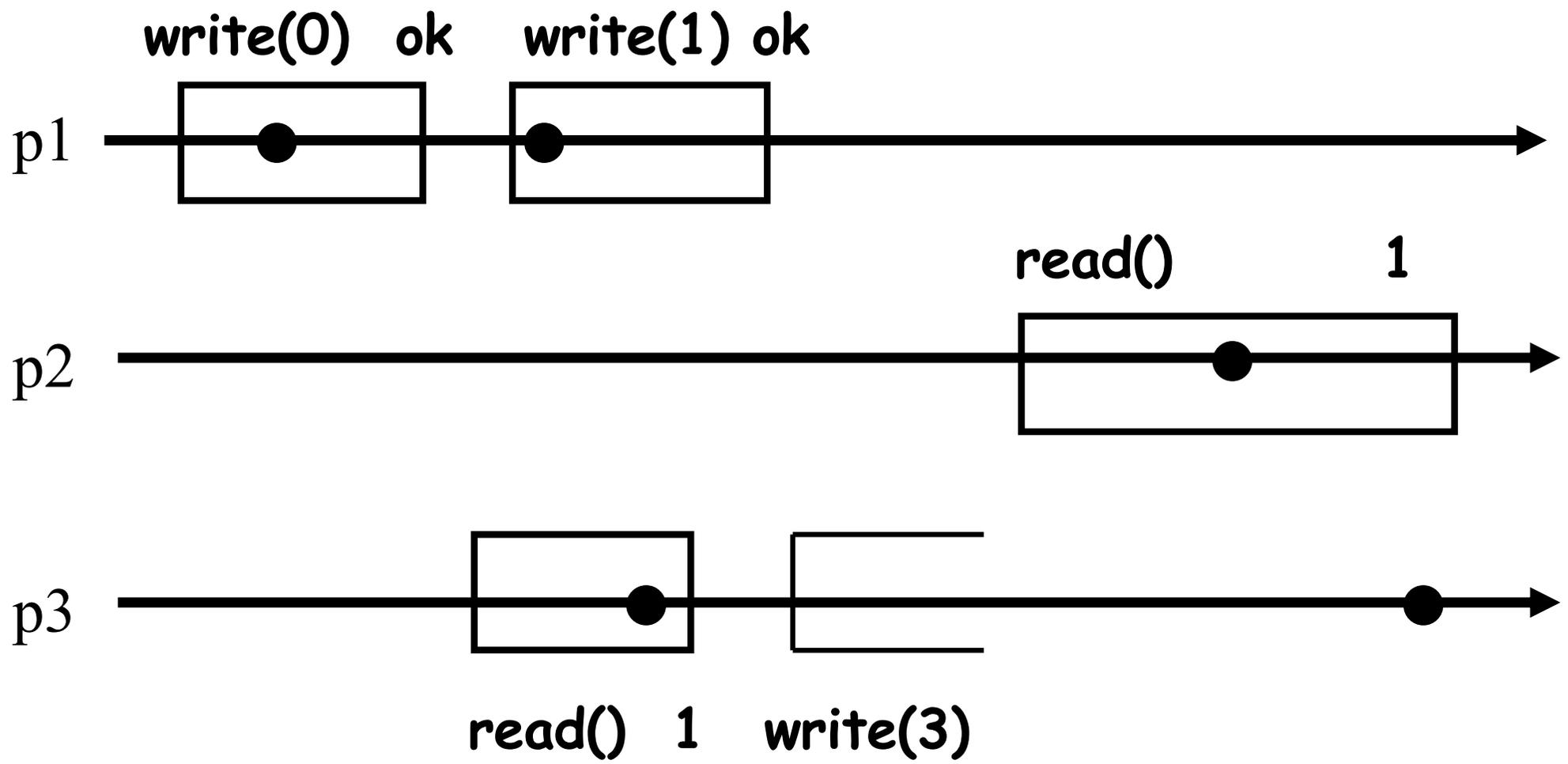
Linearizable?



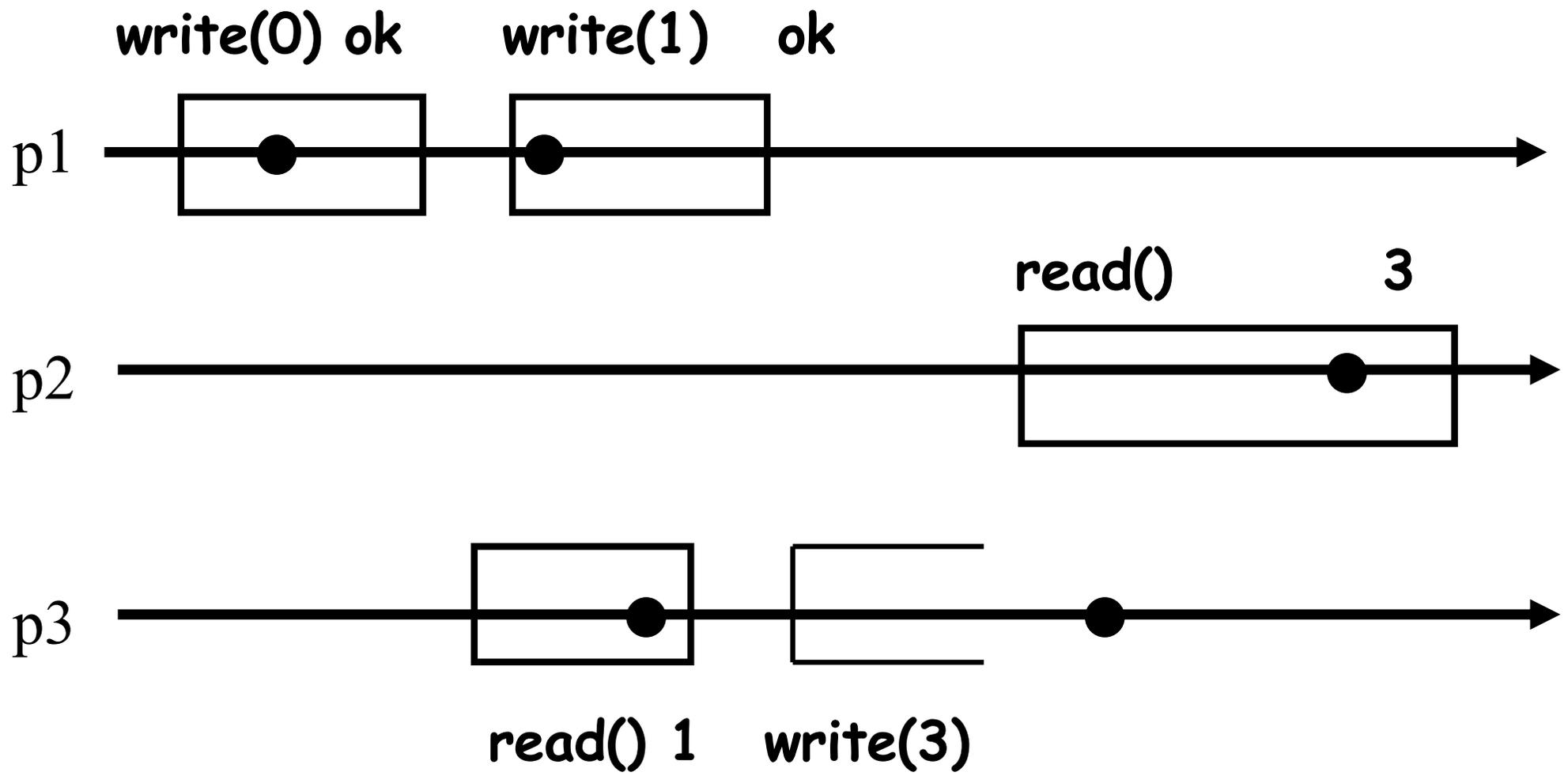
Linearizable?



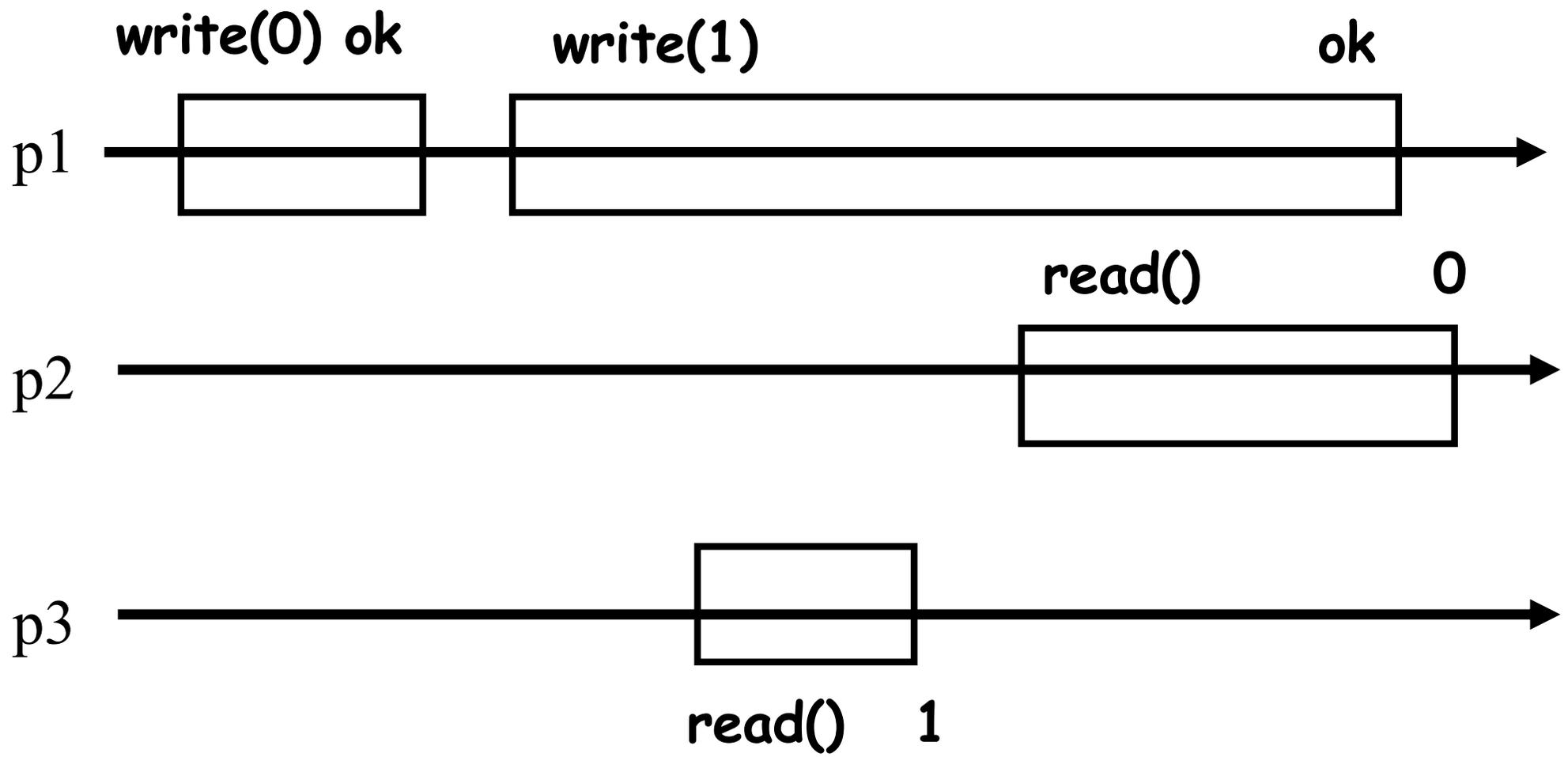
Linearizable?



Linearizable?



Linearizable?



Sequential consistency

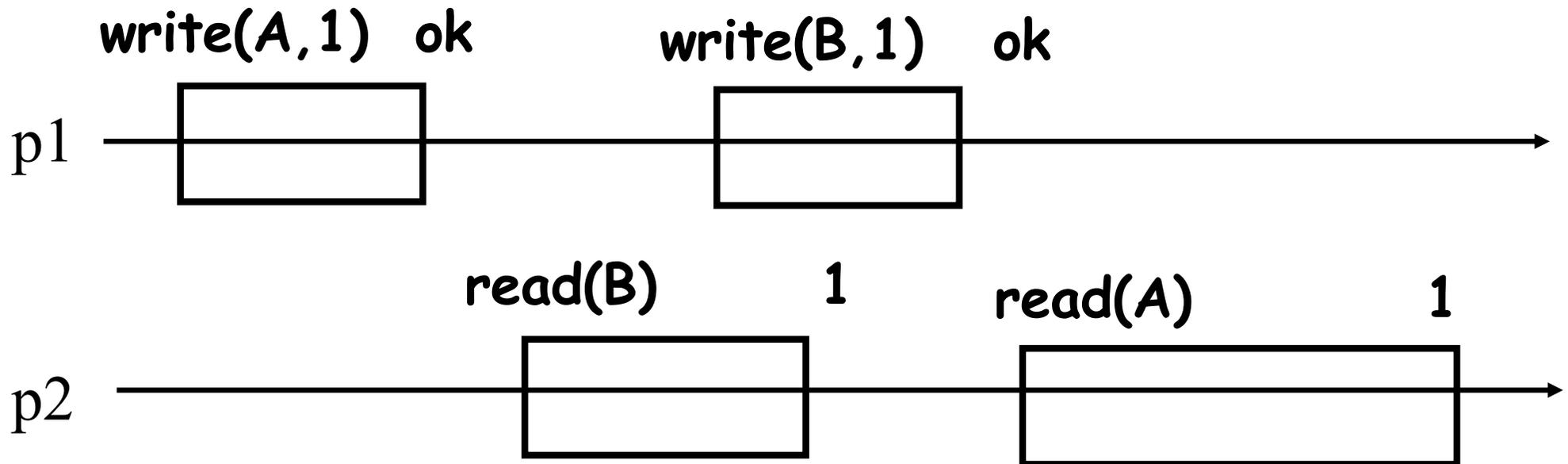
A history H is *sequentially consistent* if there exists a *sequential legal* history S such that:

- S is equivalent to some completion of H
- S preserves the *per-process order* of H :
 p_i executes op_1 before op_2 in $H \Rightarrow p_i$ executes op_1 before op_2 in S

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

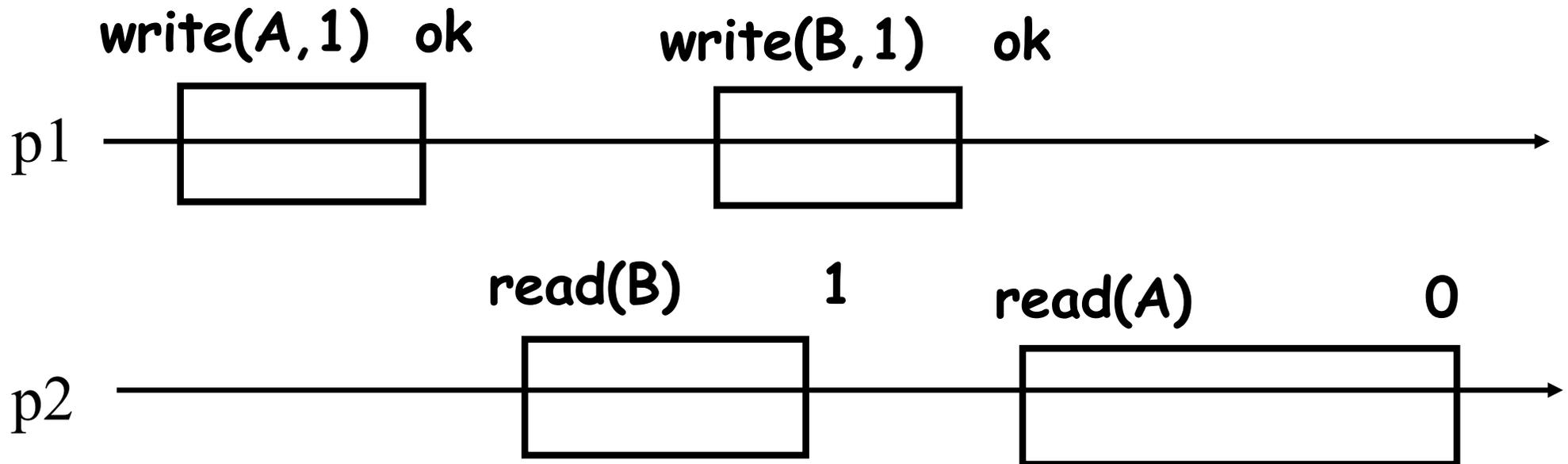
Linearizability is compositional!

- 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)



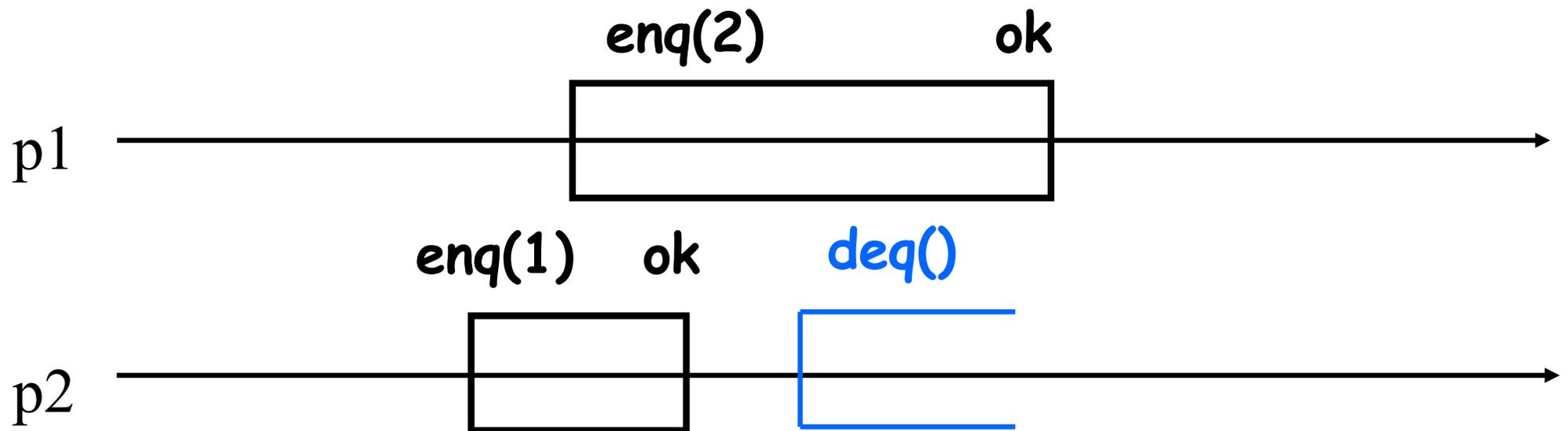
Sequential consistency is not!

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



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 **cooperates (takes enough steps)**, some process makes progress
- *Starvation-free:*
 - ✓ If every process cooperates, every process makes progress
- *Lock-free (sometimes called non-blocking):*
 - ✓ Some active process makes progress
- *Wait-free:*
 - ✓ Every active process makes progress
- *Obstruction-free:*
 - ✓ Every process makes progress **if it executes in isolation**

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 3: linearizability/progress

- Show that linearizability is **compositional**:
 - ✓ A history H on $A \times B$ is linearizable if and only if H_A and H_B are linearizable

- Show how the elements of the “periodic table of progress” are related to each other:
 - ✓ Property P is **weaker** than property P' if P' is a subset of P