Foundations of Distributed Algorithms

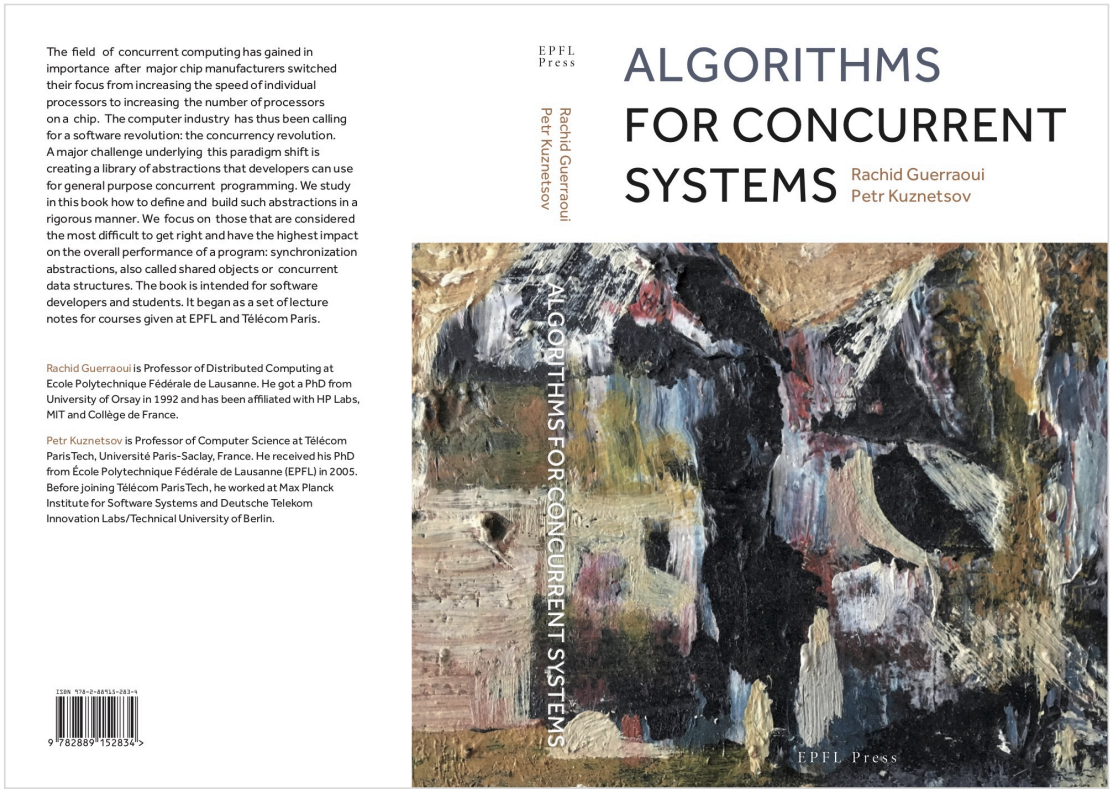
SLR206, P1, 2020-2021

Administrivia

- Language: (fr)anglais?
- Lectures: Wednesdays (09.09-21.10), 13:30-16:45, 0A214
- TP: 23/09, 1A22
- Web page: ecampus, temporary page
- Project: concurrent list implementation and analysis (teams by two)
- Office hours:
  - ✓4D48, appointments by email to petr.kuznetsov@telecom-paris.fr
- Credit = 0.7*written exam+0.3*project
  - ✓Bonus for participation/discussion of exercises
  - ✓Bonus for bugs found in slides/lecture notes

Literature (links in ecampus)

- Algorithms for Concurrent Systems . R. Guerraoui, P. Kuznetsov
- M. Herlihy and N. Shavit. The art of multiprocessor programming. Morgan Kaufman, 2008 (library)
- Lynch, N: Distributed Algorithms. Morgan Kaufmann Publishers, 1996.
- H. Attiya, J. Welch. Distributed Computing: Fundamentals, Simulations and Advanced Topics (2nd edition). Wiley. 2004



Librairie Eyerolles, 55-57-61, Blvd Saint-Germain
75005 Paris
Section « Informatique-Algorithmique » (sous-sol)

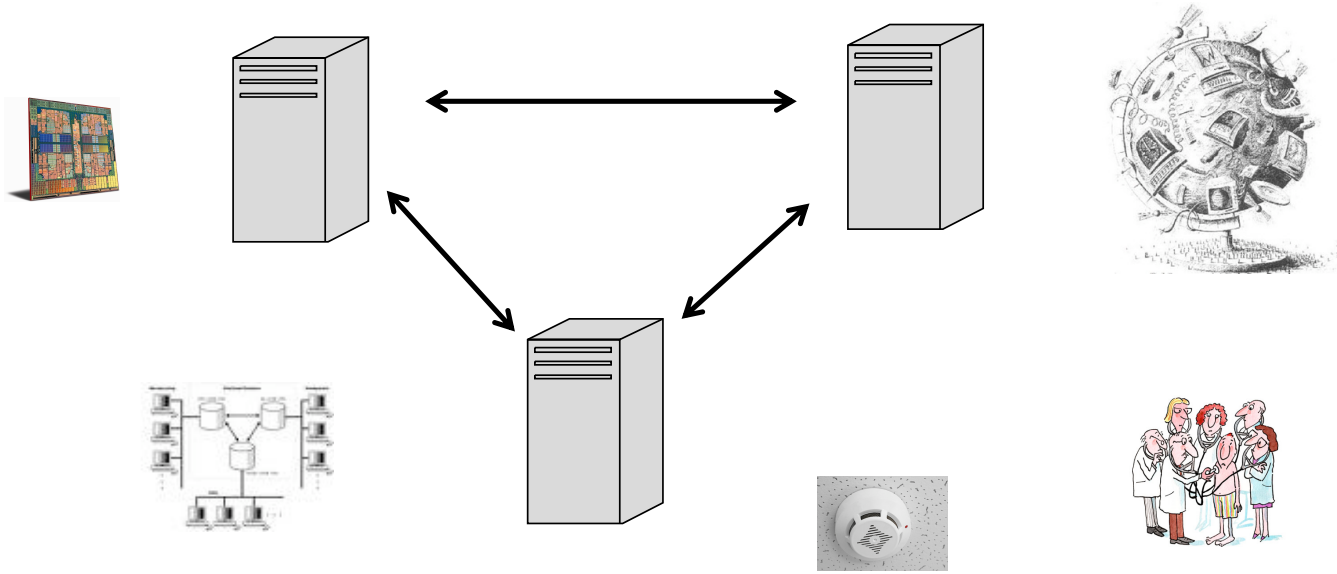
Roadmap

- Synchronization and concurrency
- Correctness in distributed systems
- Optimistic, lazy and non-blocking synchronization
  - ✓Lab
- Basics of read-write communication
- Consensus and universal construction
- Transactional memory

This course is about distributed computing:

independent sequential processes that communicate

Concurrency is everywhere!

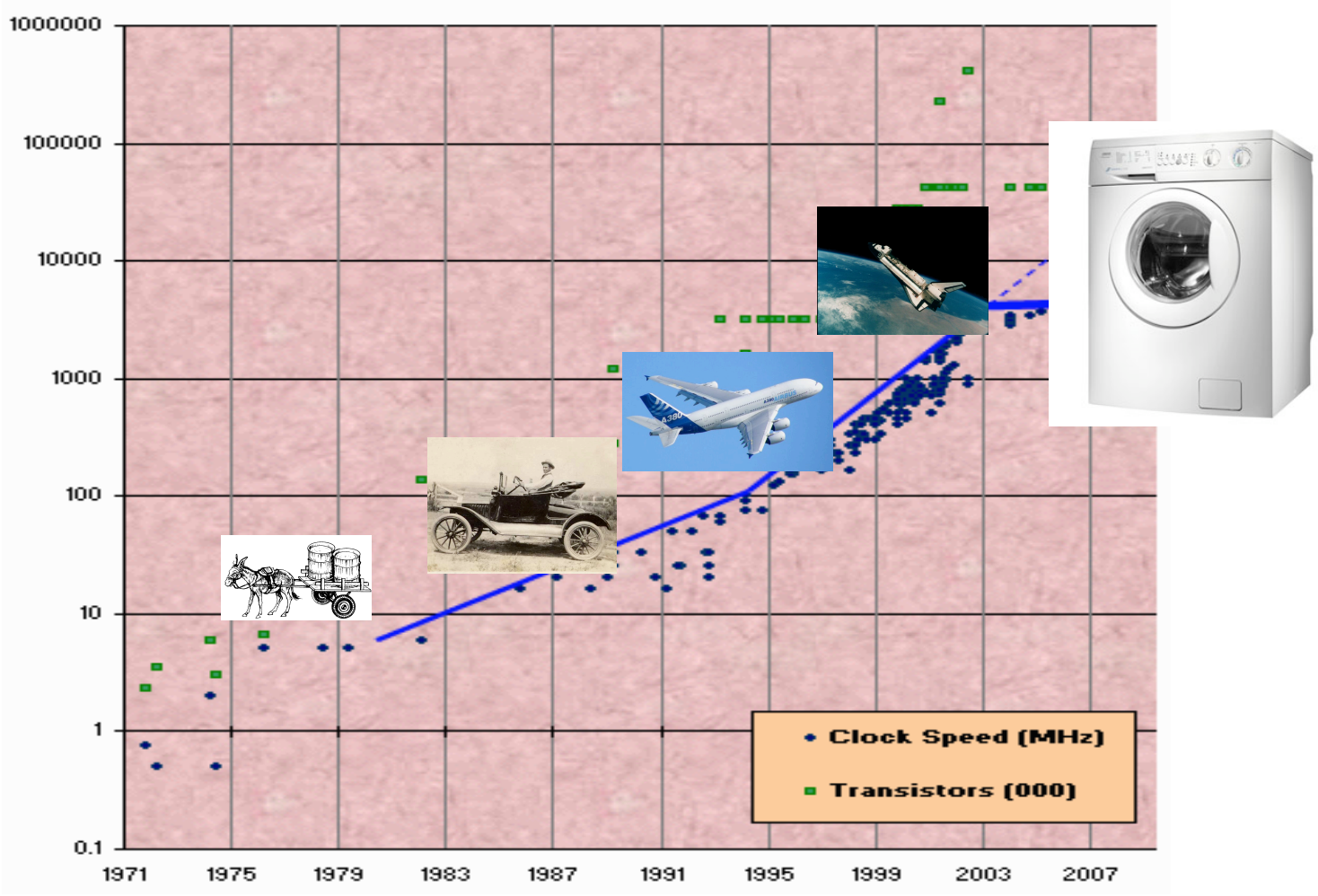


- Multi-core processors
- Sensor networks
- Internet
- …

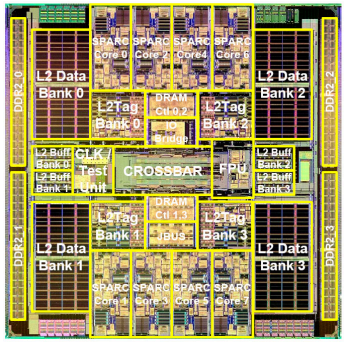
Communication models

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

Moore's Law and CPU speed



- 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

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

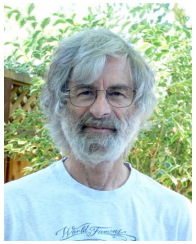
Challenges

- 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 ≠ 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 and large-scale distributed services now

Real concurrency--in which one program actually continues to function while you call up and use another--is more amazing but of small use to the average person. How many programs do you have that take more than a few seconds to perform any task?

New York Times, 25 April 1989, in an article on new operating systems for IBM PC

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, …

Edsger Dijkstra
1930-2002

Dining philosophers (Dijkstra, 1965)

- 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 ≥ 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
```

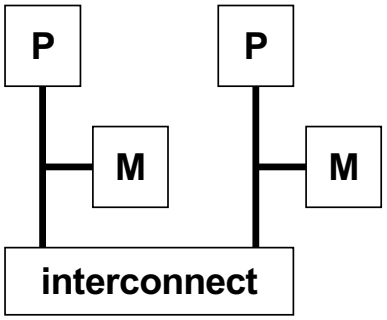
What about time complexity?

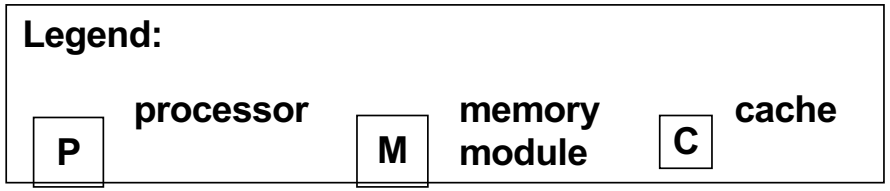
- **Number of steps is not a good metric**
  Alur & Taubenfeld [RTSS'92]
- **Processes may need to busy-wait** 

of <u>remote</u> memory references/accesses is a more realistic measure

Remote memory references (RMRs)

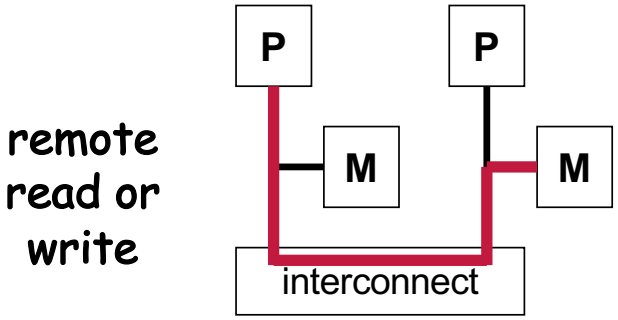
Distributed
Shared Memory
(DSM)





Remote memory references (RMRs)

Distributed Shared Memory (DSM)



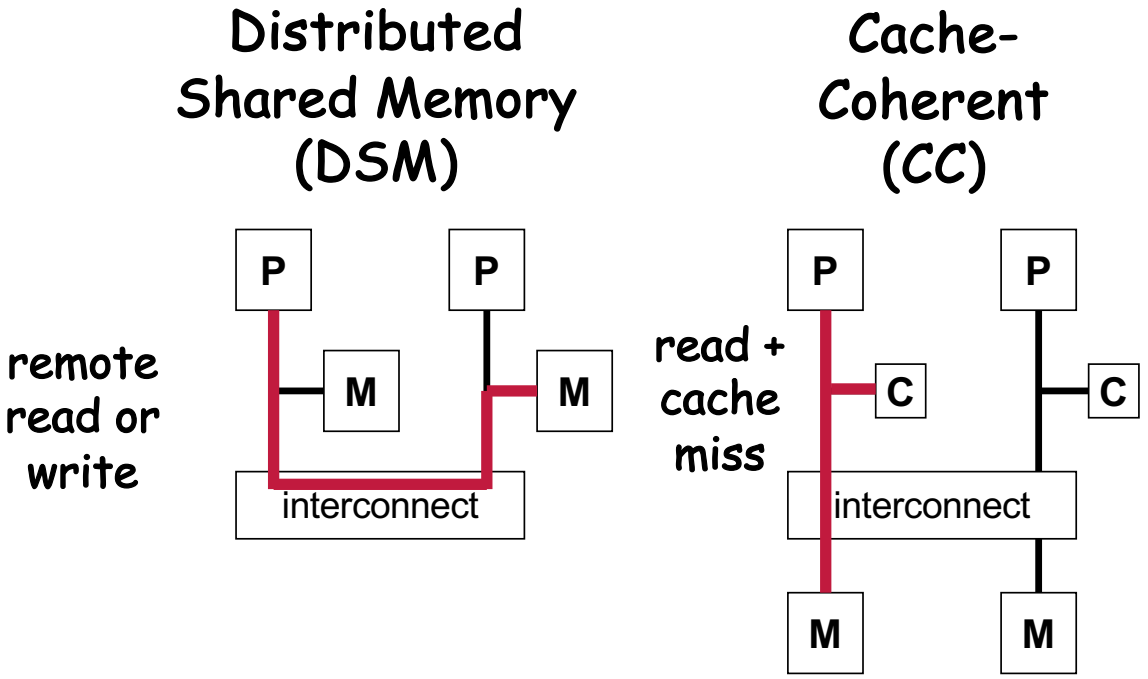
Legend:

P processor

M memory module

C cache

Remote memory references (RMRs)



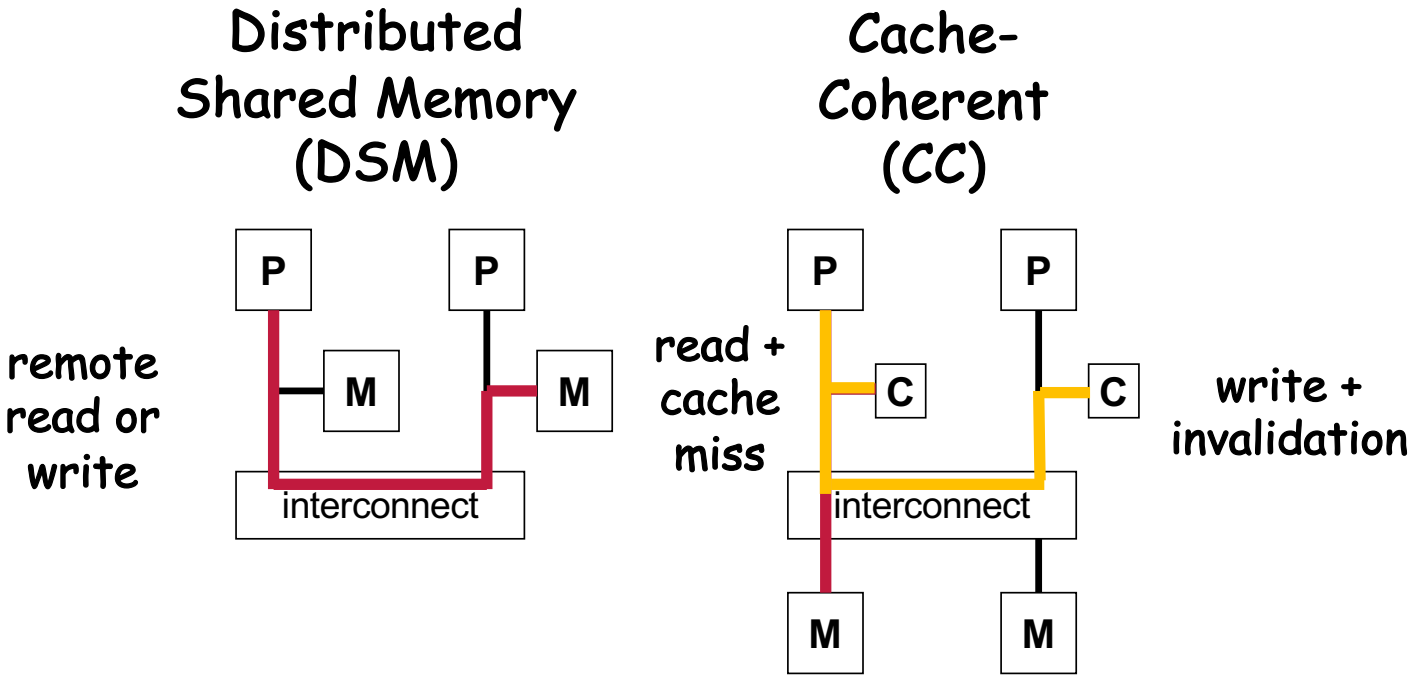
Legend:

P processor

M memory module

C cache

Remote memory references (RMRs)



Peterson's lock: N ≥ 2 processes

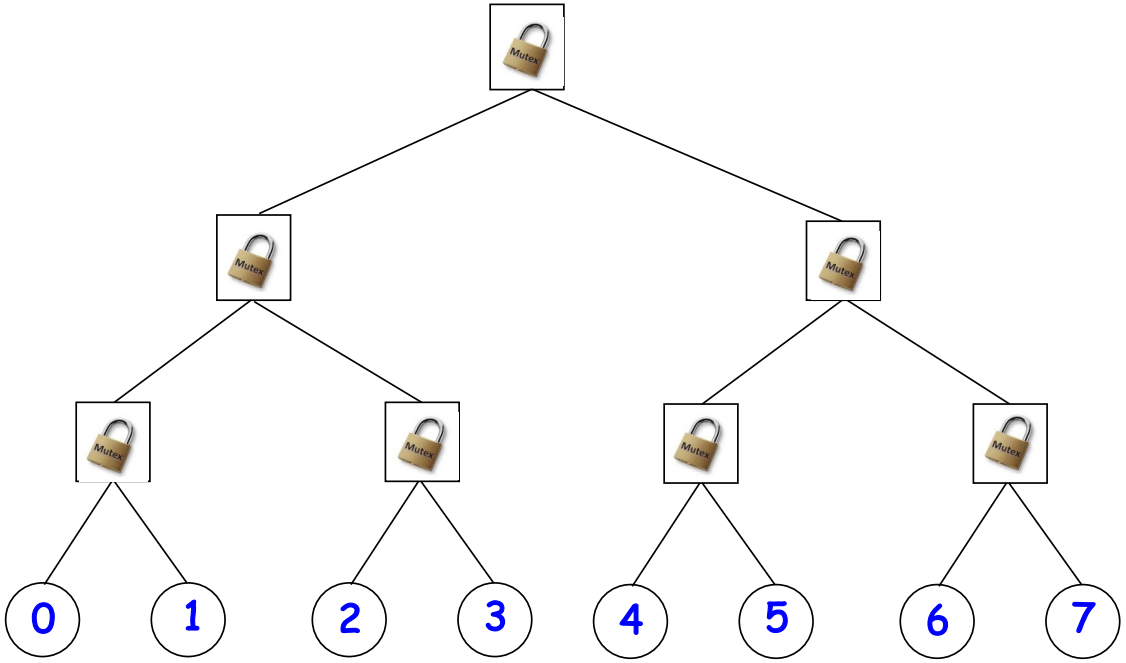
```
// initialization
level[0..N-1] = {-1};      // current level of processes 0...N-1
waiting[0..N-2] = {-1}; // the waiting process in each level
                            // 0...N-2

// 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
```

$O(N^3)$ RMRs!

n-process tournament tree alg.

Hofri [OSR'90]



Satisfies mutex and starvation-freedom

O(log N) RMRs

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!

RMR complexity bounds

Deterministic read/write mutual exclusion	**O(log N)** Yang, Anderson [DC'95] **Ω(log N)** Attiya, Hendler, Woelfel [STOC'08]
Randomized read/write mutual exclusion (strong adversary)	**O(log N / log log N)** Hendler, Woelfel [DC'11] **Ω(log N / log log N)** Woelfel, Giakkoupis [DC'12]

Quiz 1.1

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

```
P0:                          P1:
turn = 1;                    turn = 0;
flag[0] = true;              flag[1] = true;
…                            …
```

Would it work?

- Prove that the Tournament Tree algorithm satisfies:
  - ✓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)

Foundations of Distributed Algorithms

Correctness: safety and liveness

SLR206 2020-2021

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

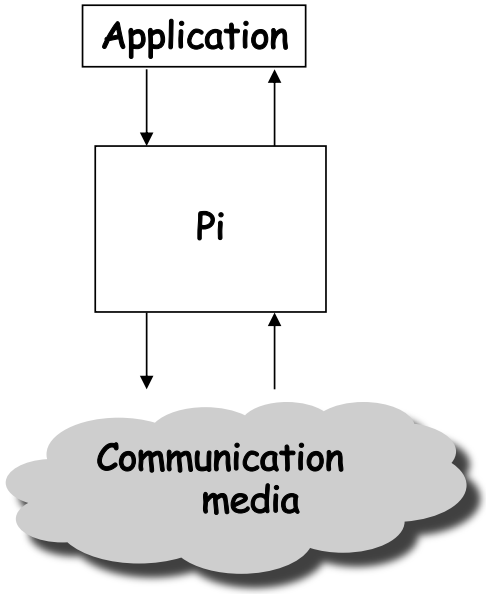
Basic abstractions

- *Process* abstraction – an entity performing independent computation

- Communication
  - ✓Message-passing: *channel* abstraction
  - ✓Shared memory: *objects*

Processes

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

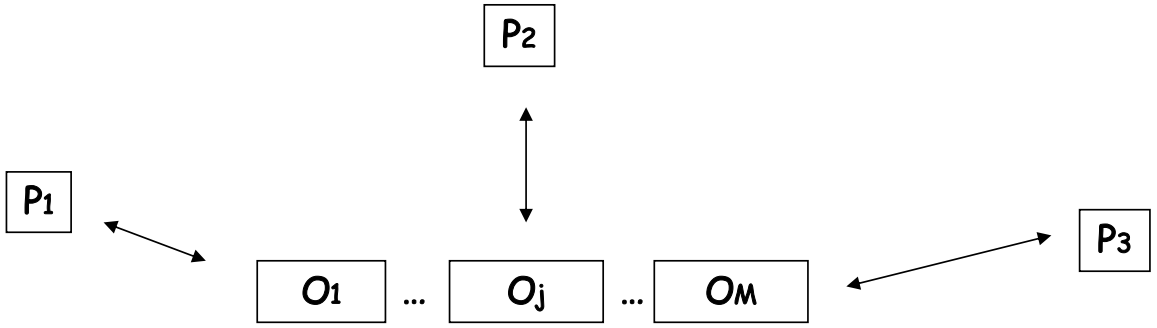


Algorithm = {P_1,…,P_N}

- Deterministic algorithms
- Randomized algorithms

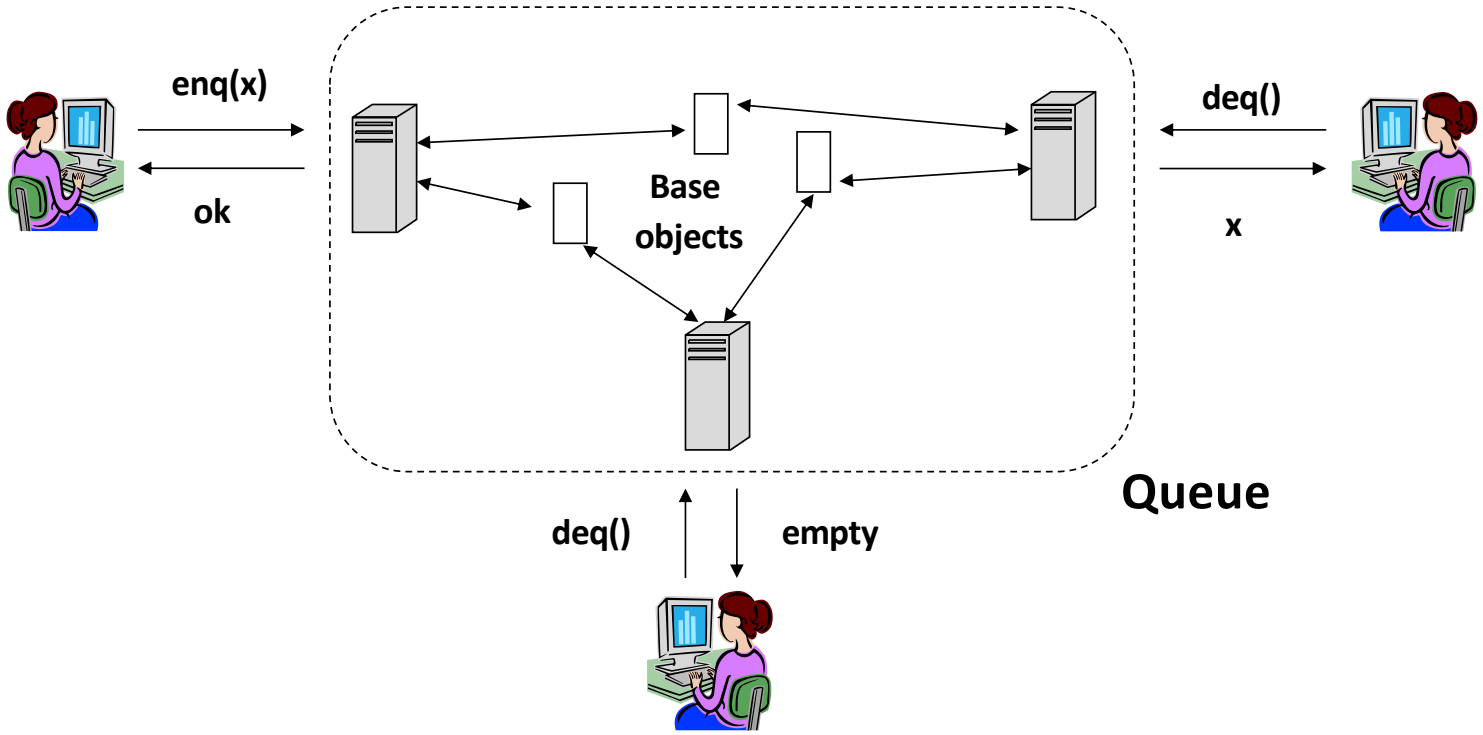
Shared memory

- 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 ≈ 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., read-write buffer, a queue, a mutex) by invoking operations

- An operation is implemented using a sequence of accesses to base objects
  - E.g.: a queue 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 σ_0, σ_1, σ_2,…, 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):

$\Sigma = U_{i,v}\{propose_i(v), decide_i(v)\} U \{base\text{-}object\ accesses\}$

✓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 1.2: 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 are safe

2. Show that every unsafe run σ has an unsafe finite prefix σ' : every extension of σ' is unsafe

1. 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;
```

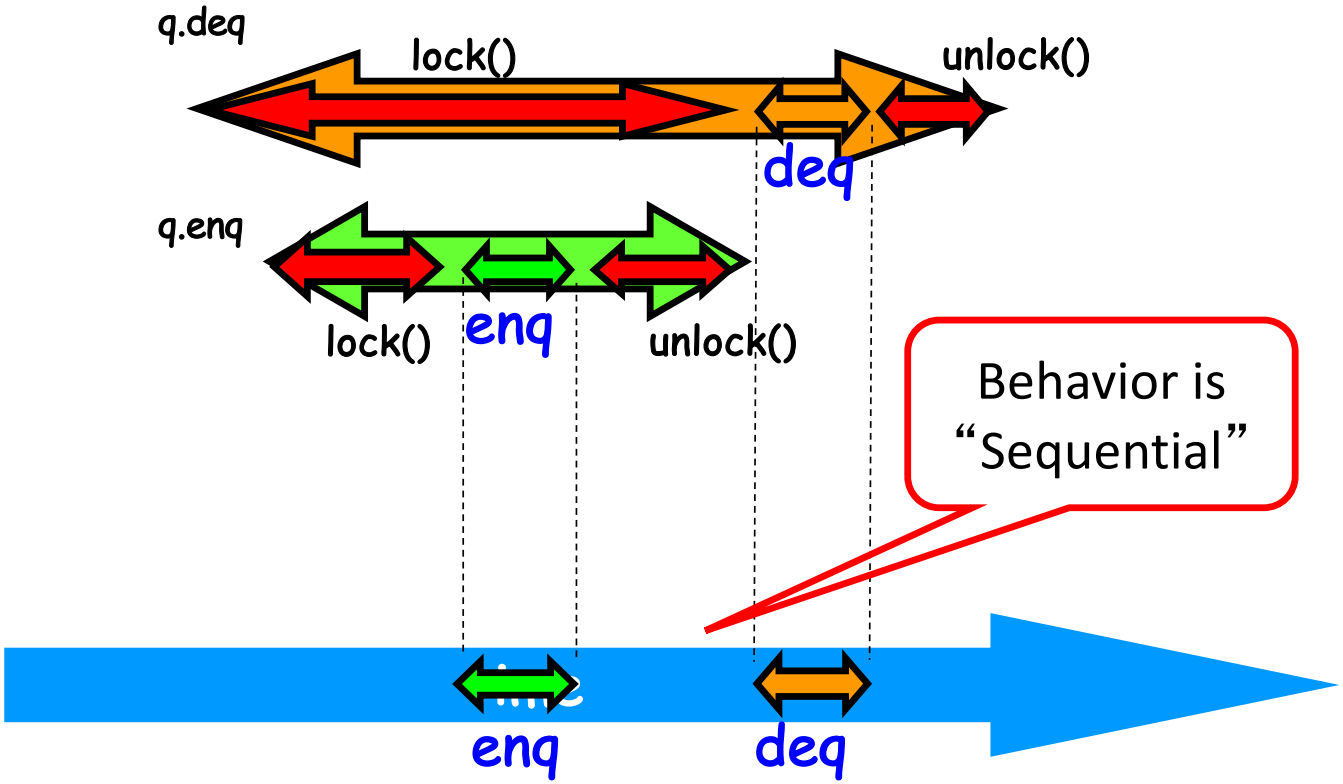
Intuitively…

```
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



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

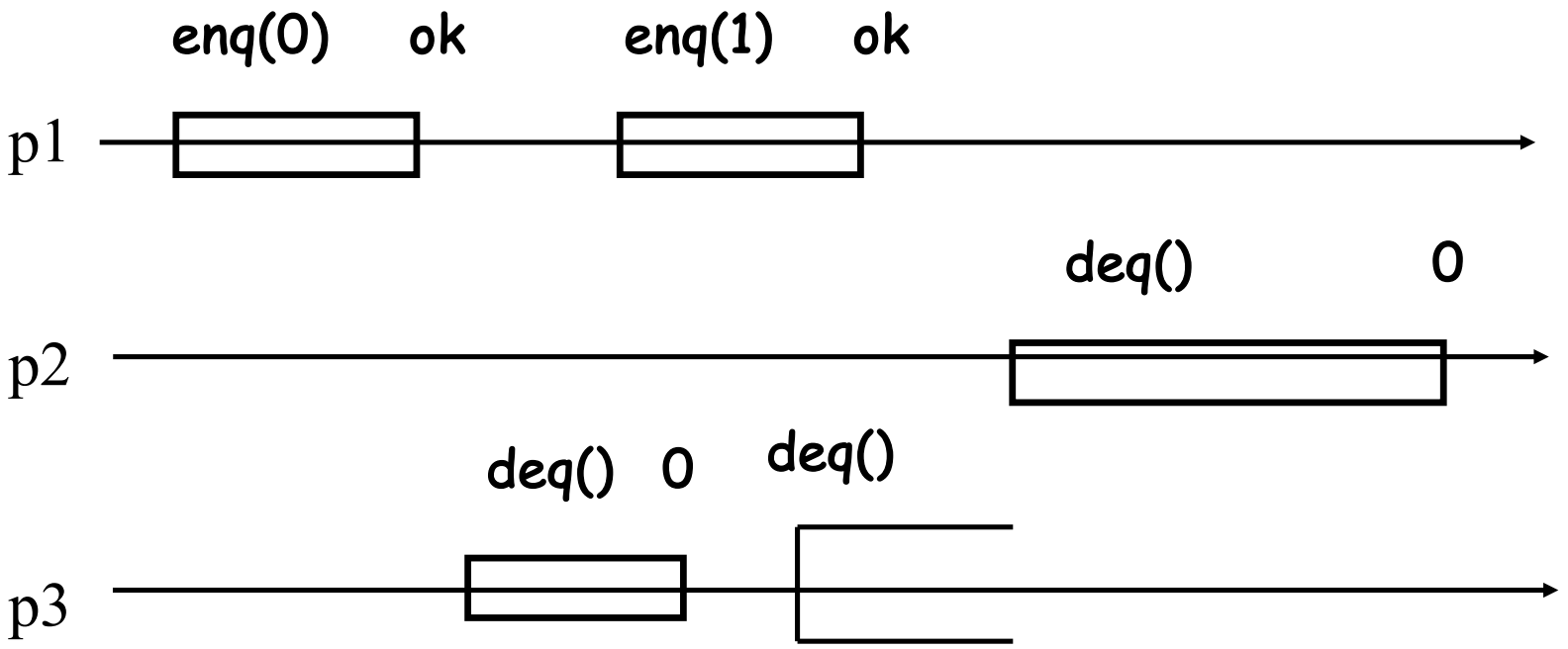
E.g., p1-enq(0), p2-deq(),p1-ok,p2-0,…

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)

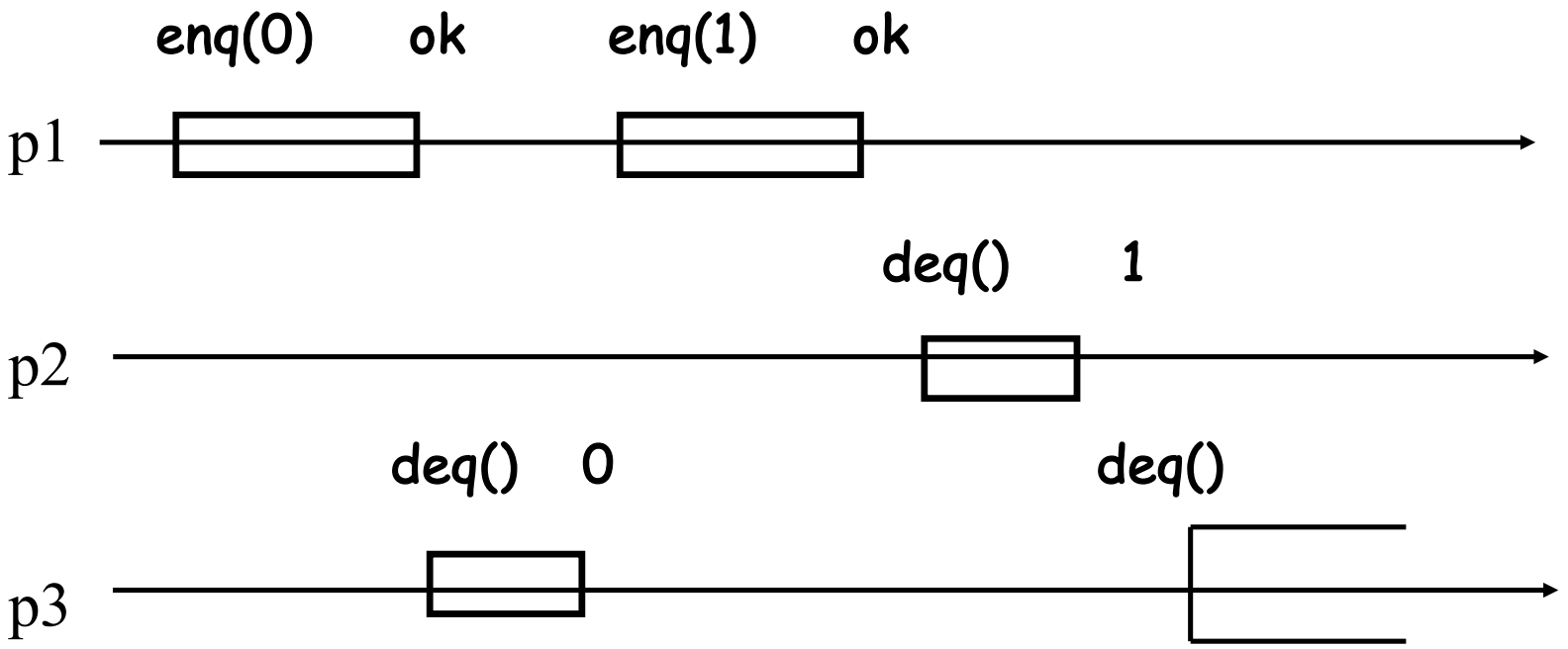
Histories



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)

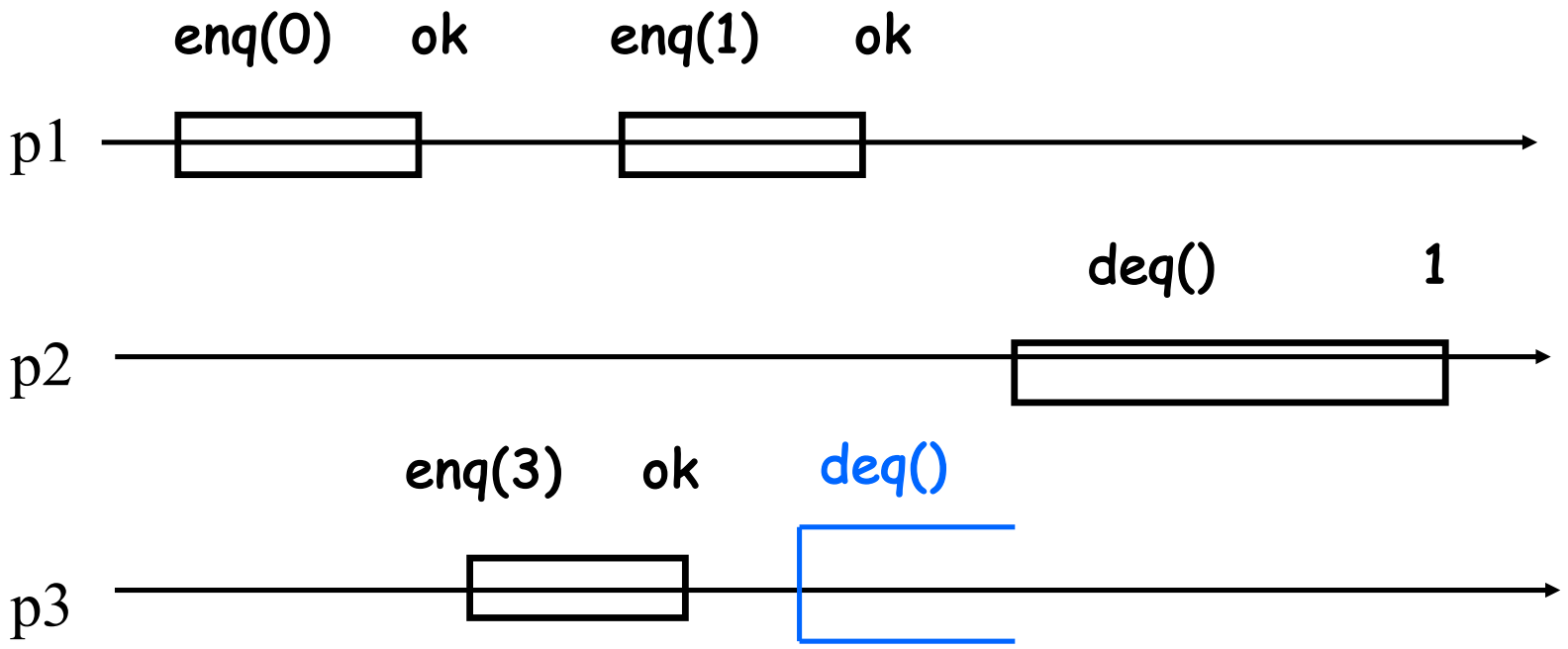
Complete operations and completions

Let H be a history

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

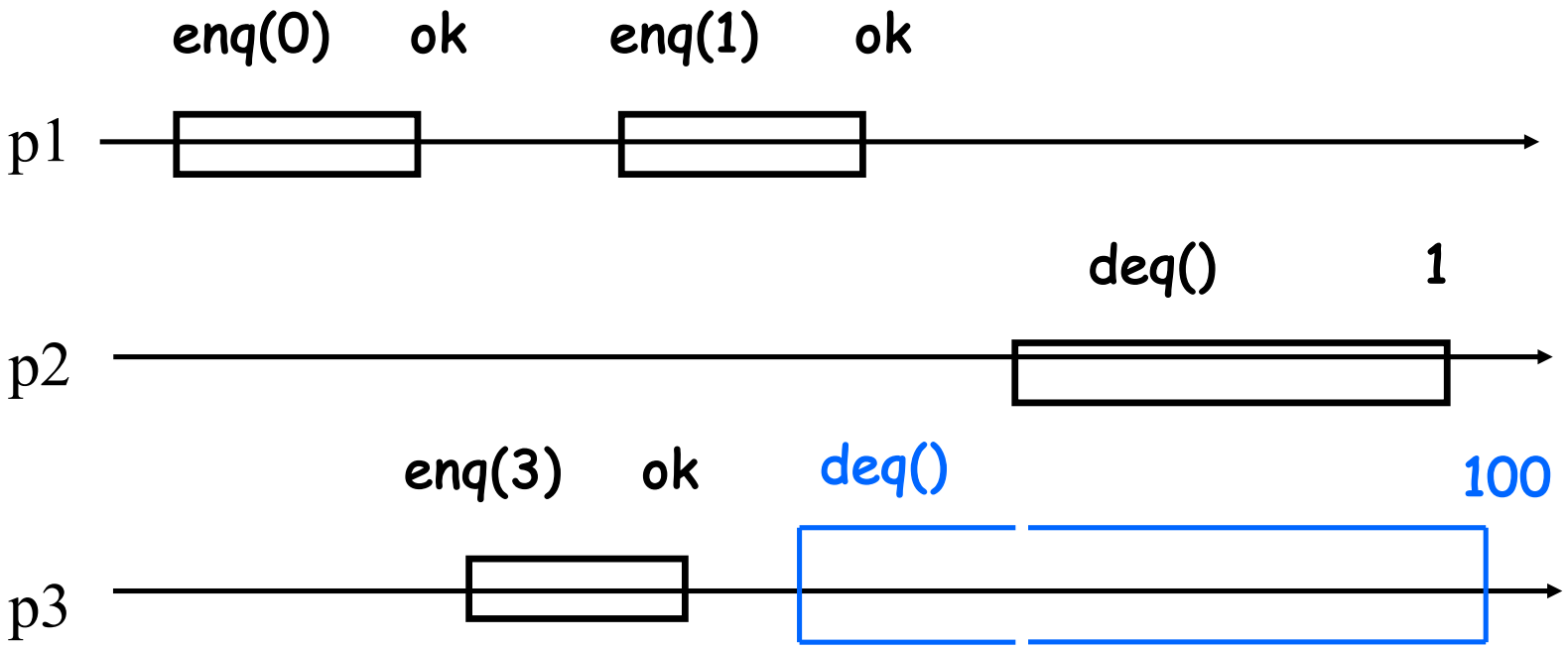
A *completion* of H is a history H' that includes all complete operations of H and a subset of incomplete operations of H followed with matching responses

Complete operations and completions



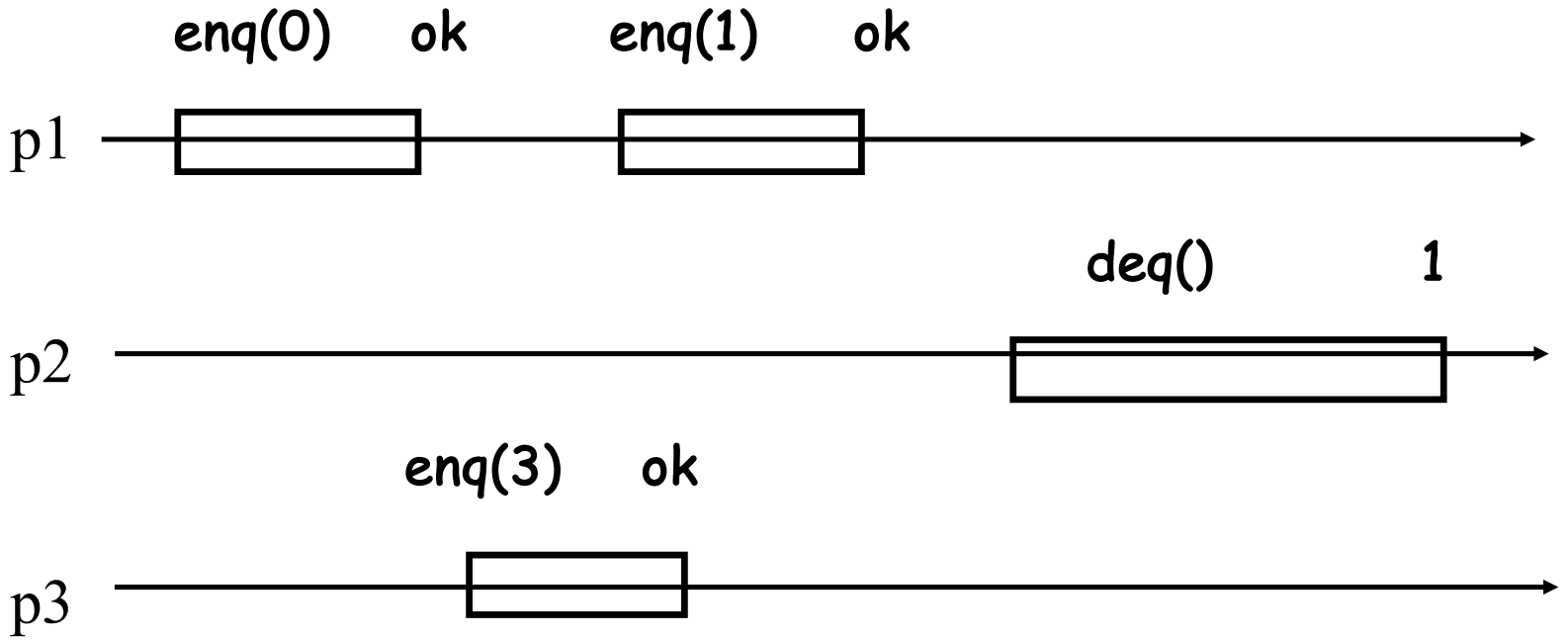
p1-enq(0); p1-ok; p3-enq(3); p1-enq(1); p3-ok; p3-deq();
p1 -ok; p2-deq(); p2-1;

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.:

$H = p_1$-enq(0); p_1-ok; p3-deq(); p_3-3

$H' = p_1$-enq(0); p_3-deq(); p_1-ok; p_3-3

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?

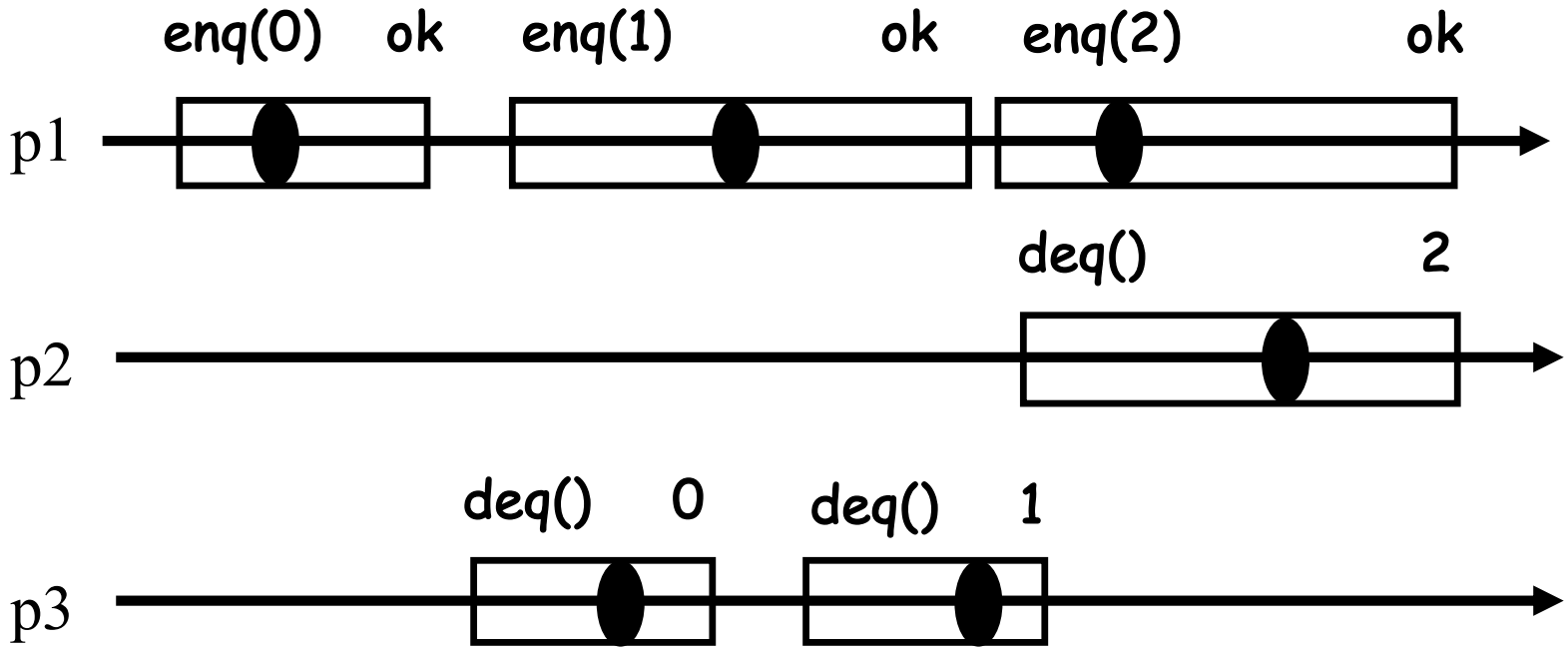
Linearization points

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

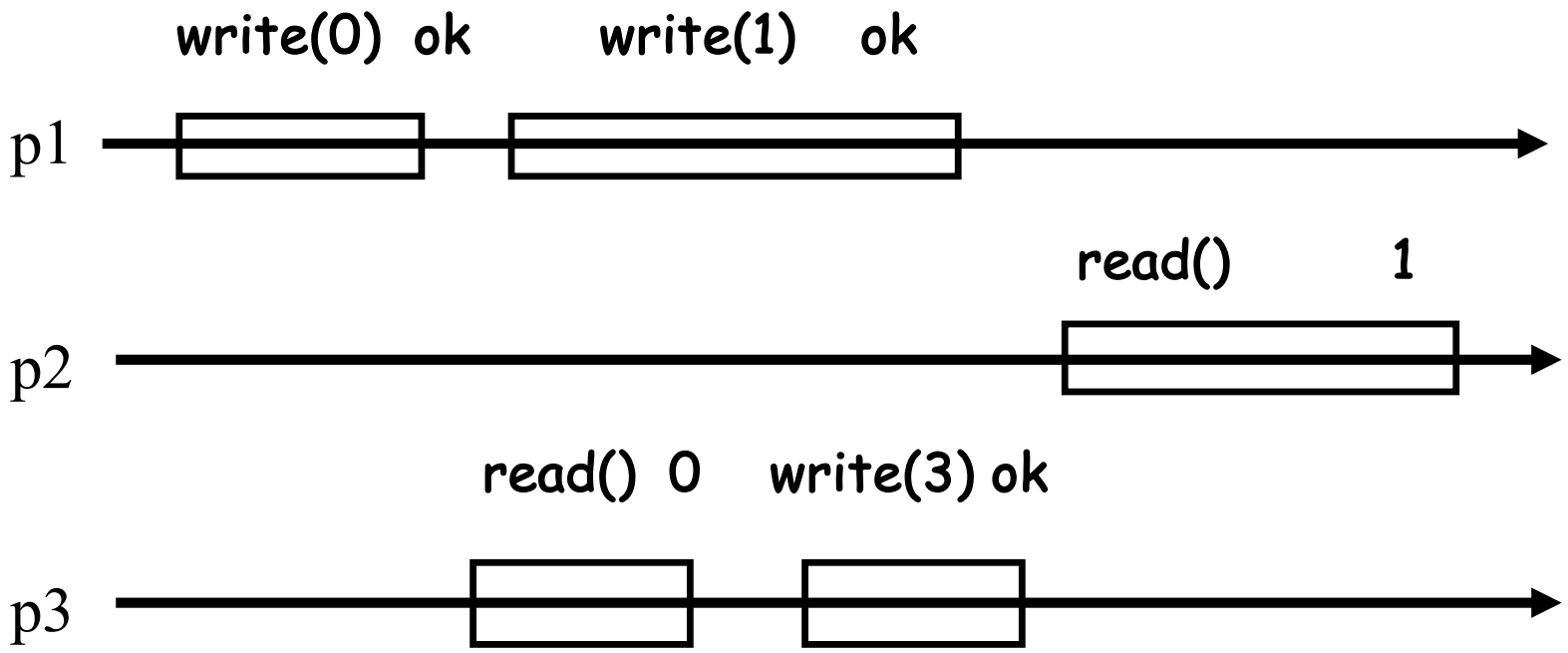
Informally, the complete operations (and some incomplete operations) in a history are seen as taking effect instantaneously at some time between their invocations and responses

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

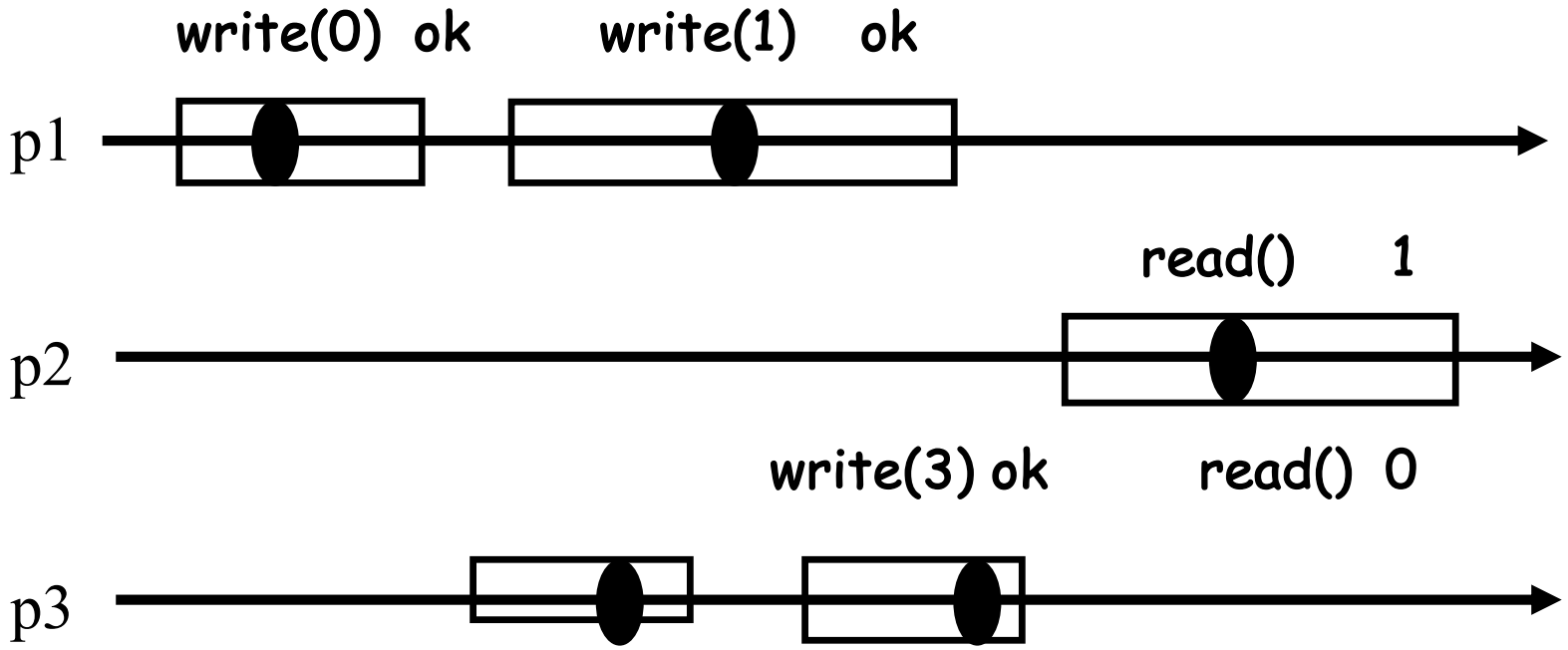
Linearizable?



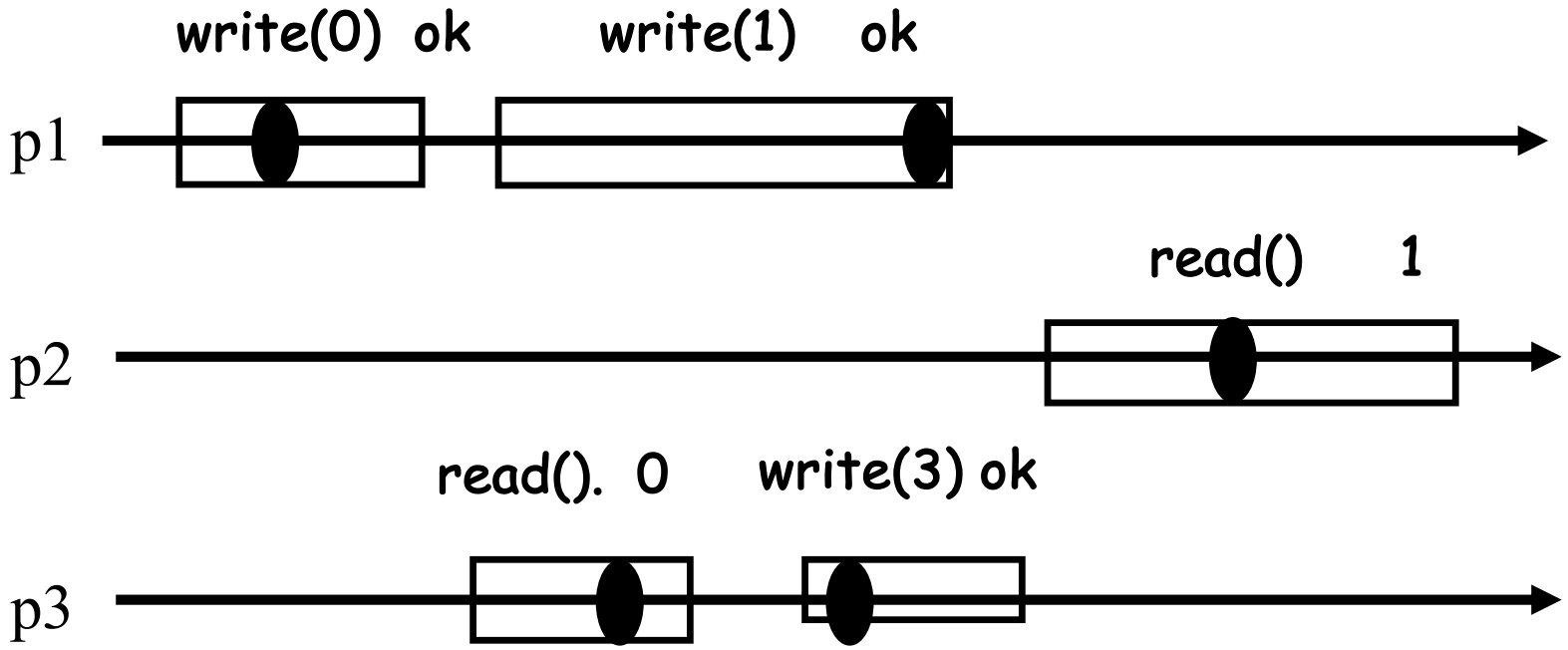
Linearizable?



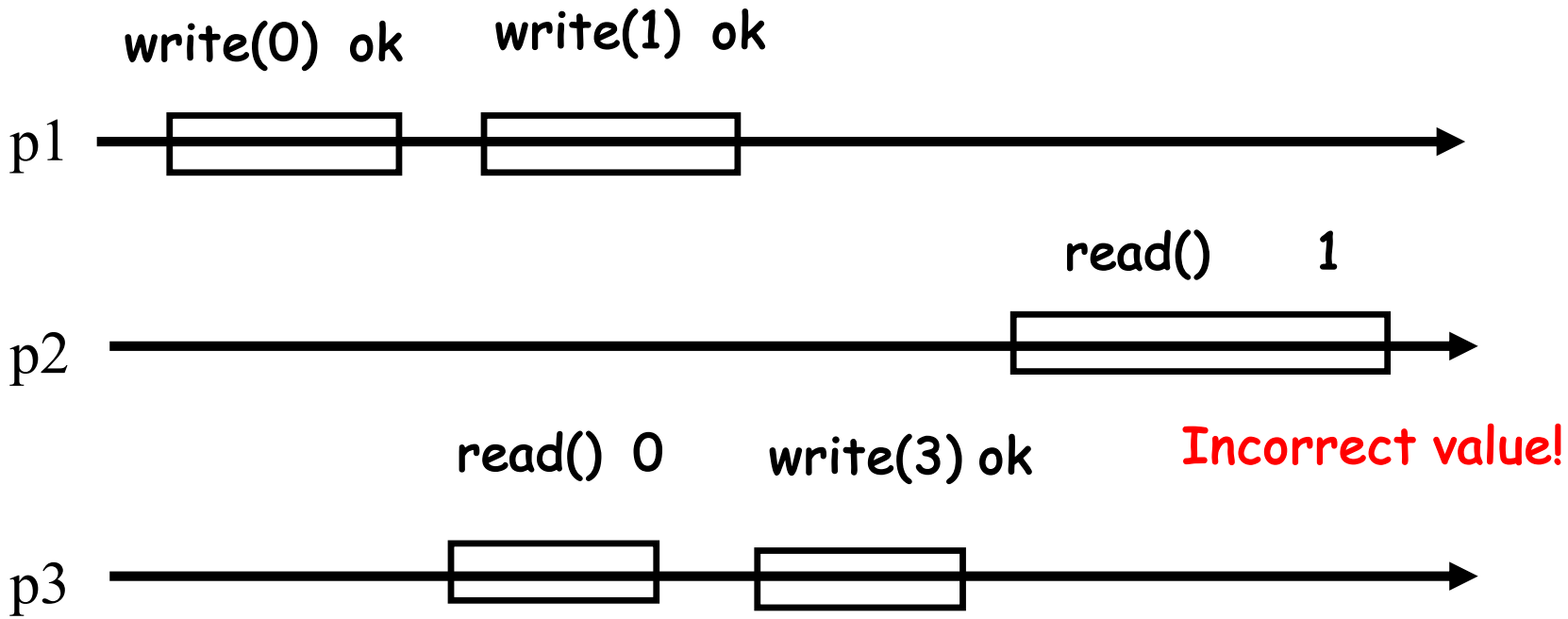
Linearizable?



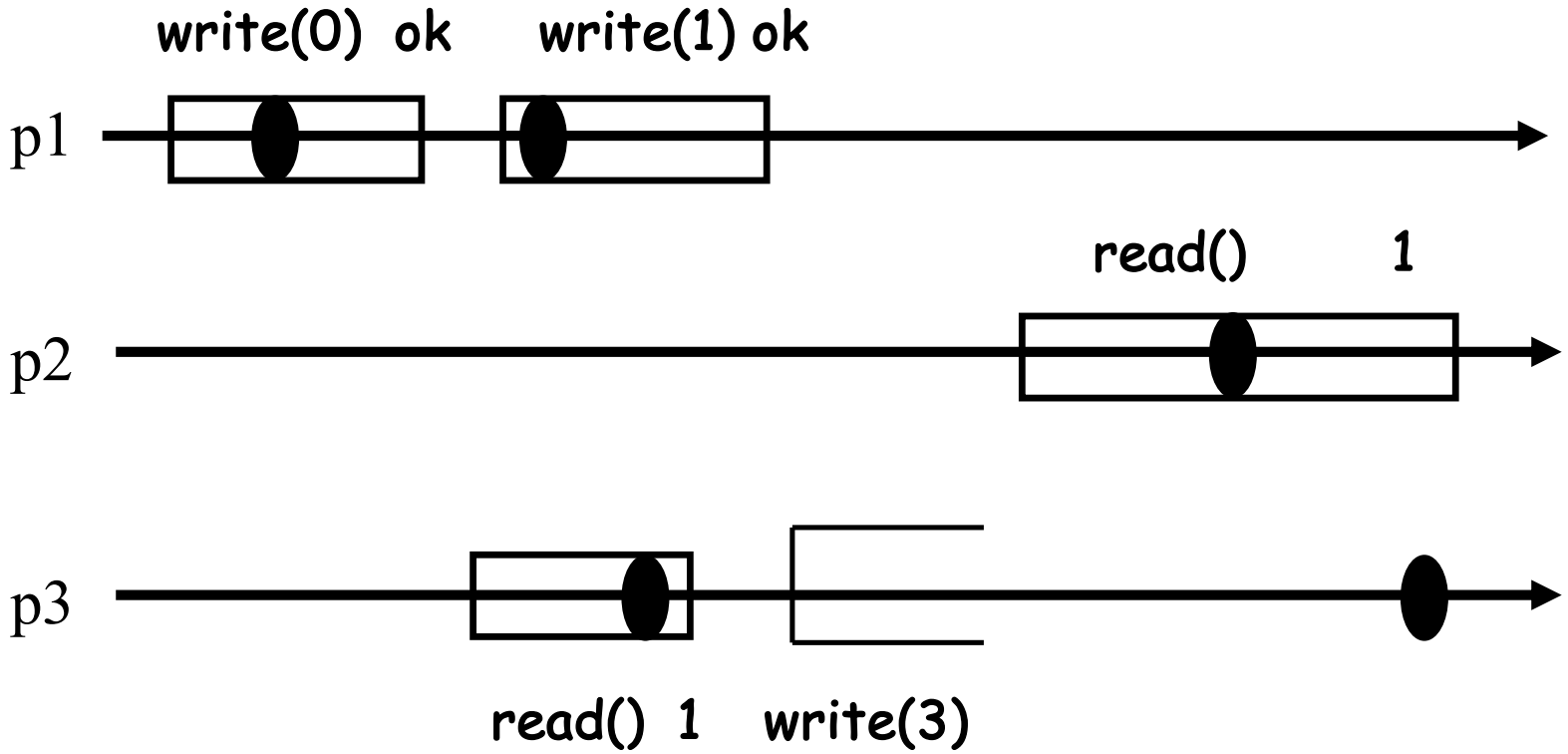
Linearizable?



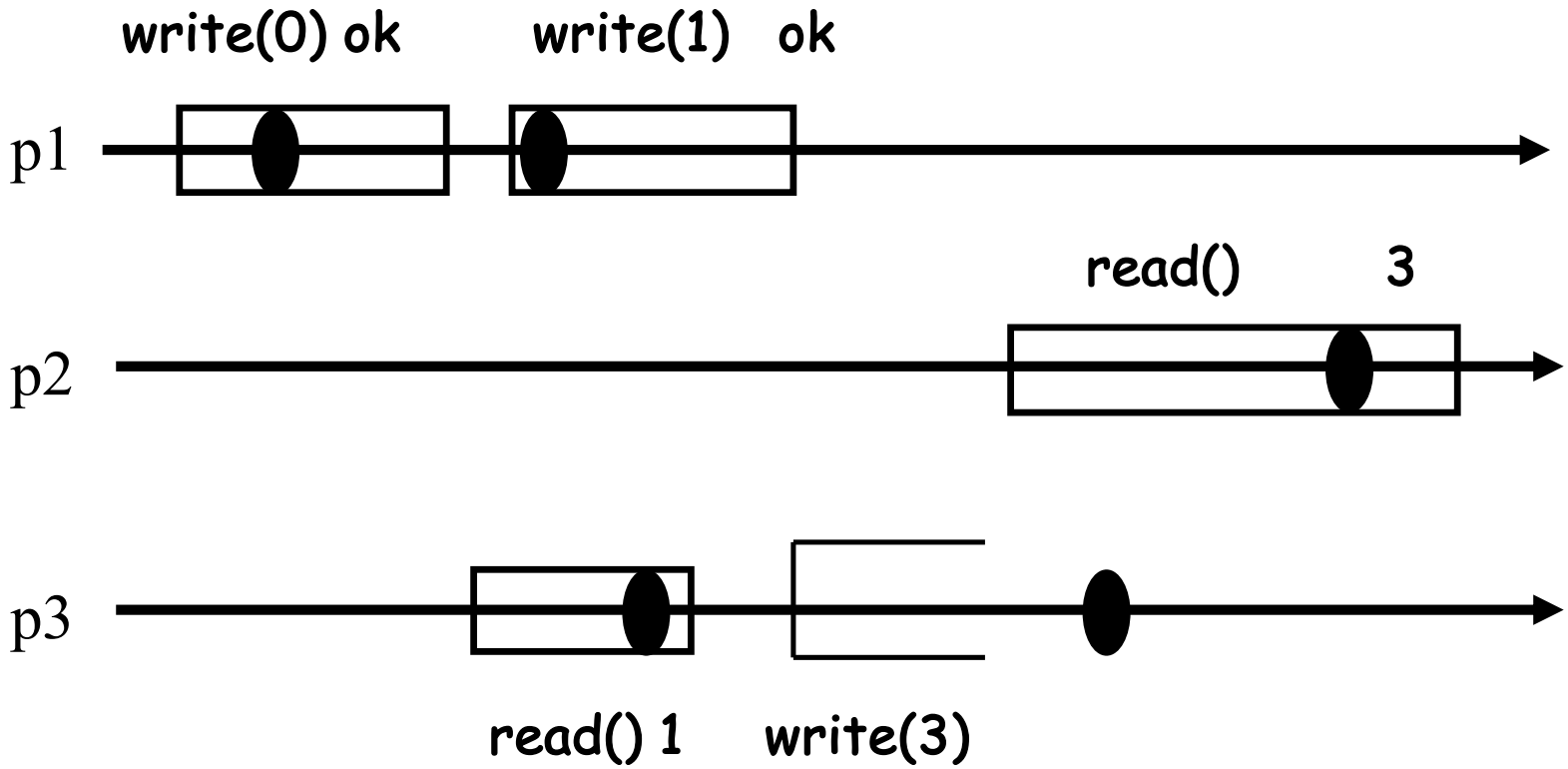
Linearizable?



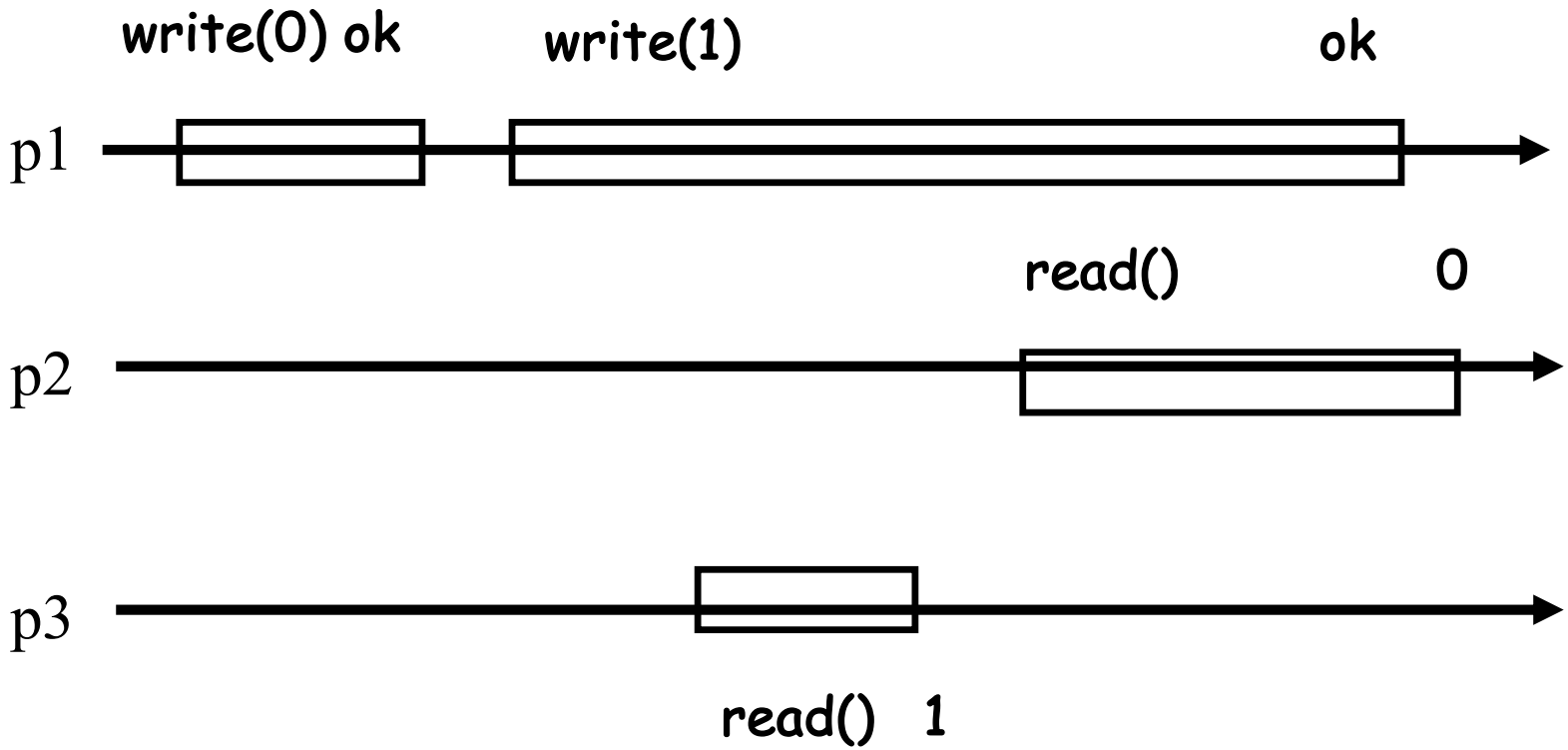
Linearizable?



Linearizable?



Linearizable?



Sequential consistency

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

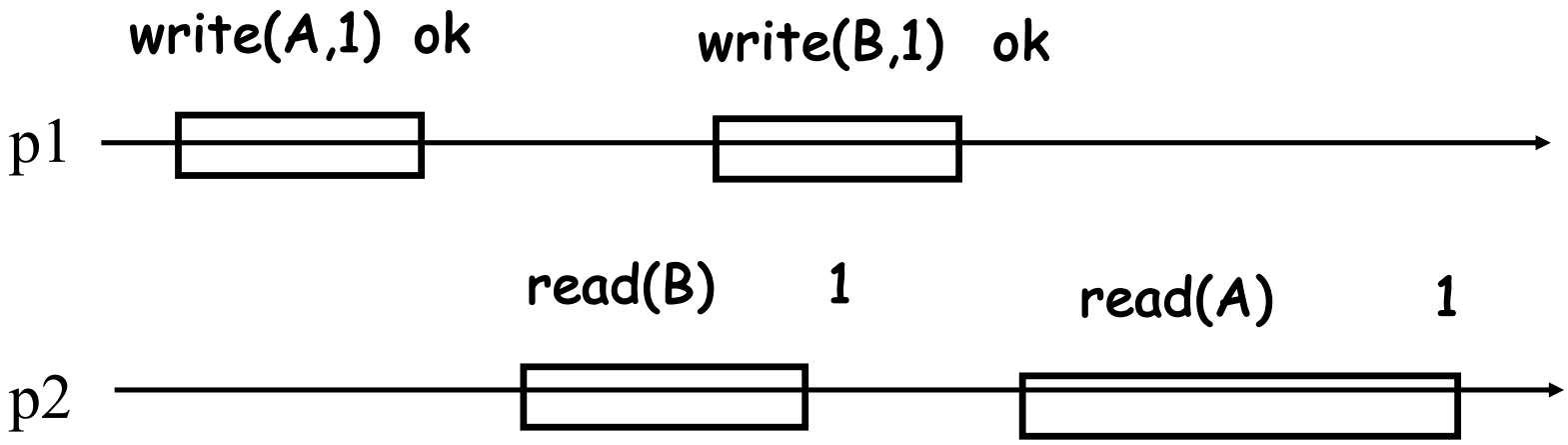
- S is equivalent to some completion of H
- S preserves the per-process order of H:

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

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

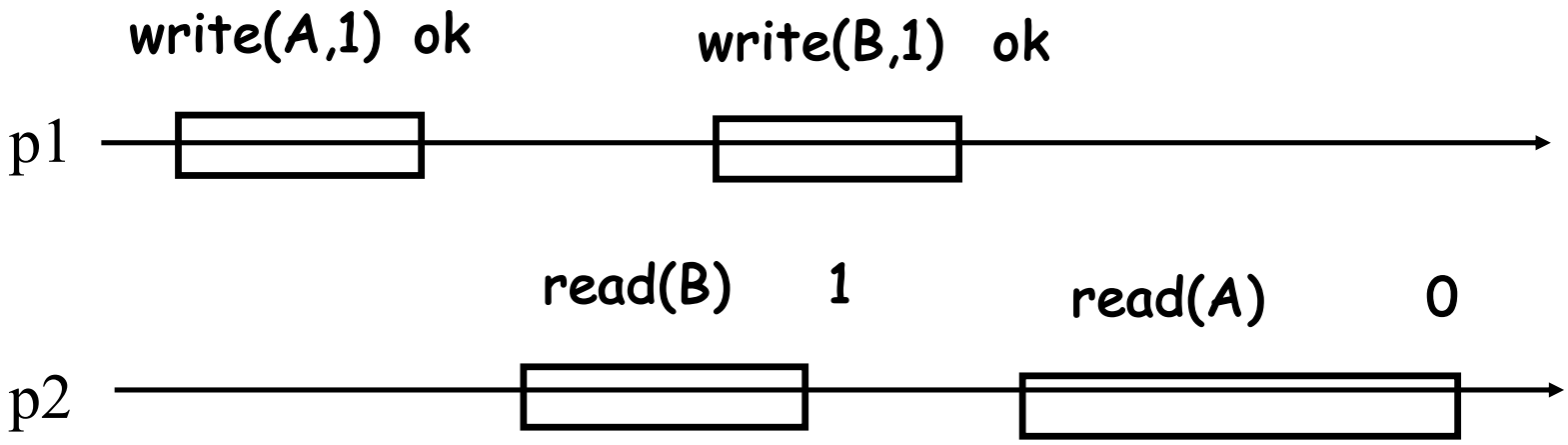
Linearizability is compositional!

- 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)



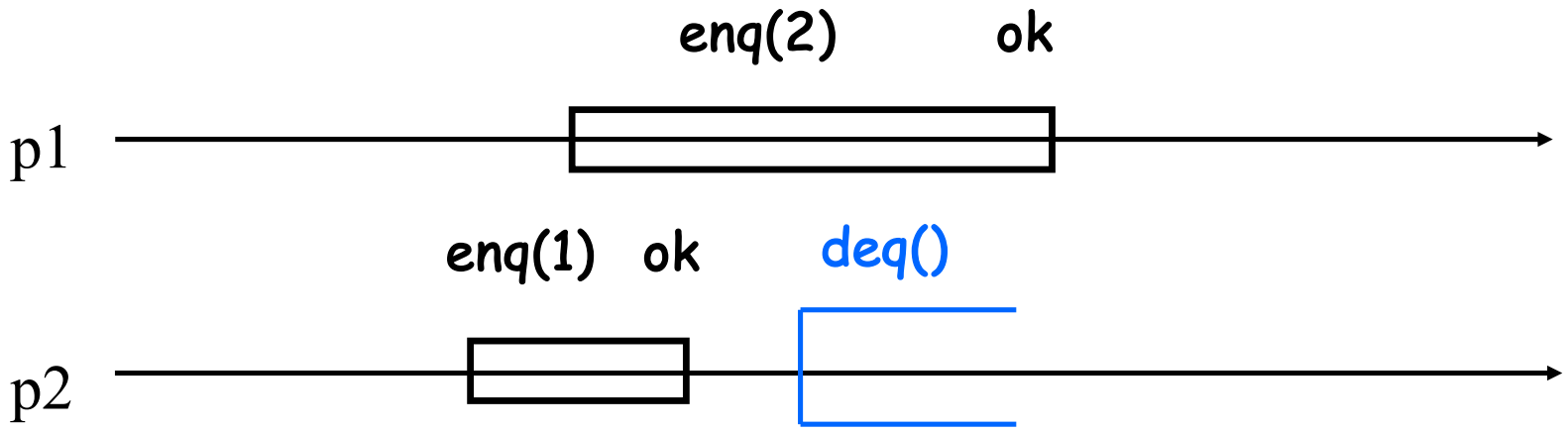
Sequential consistency is not!

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



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

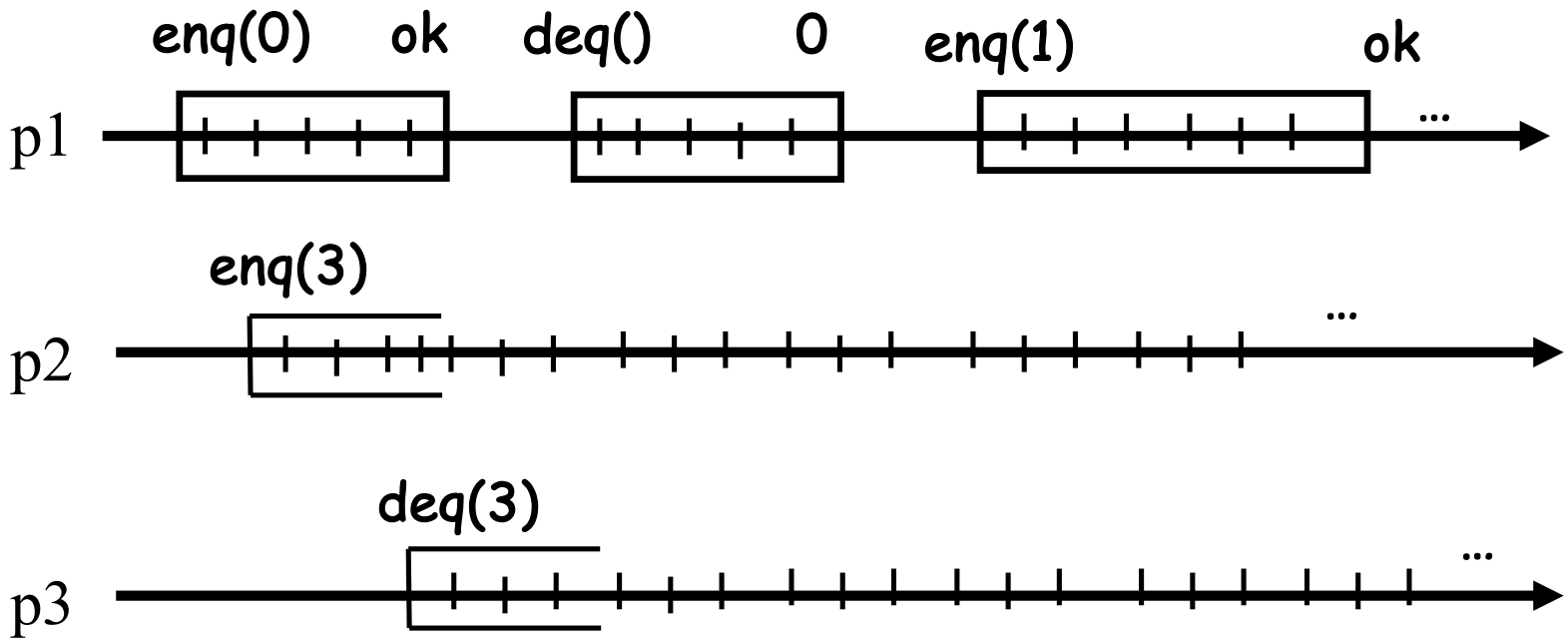
- *Deadlock-free:*
  - ✓ If every process is correct*, some process makes progress**
- *Starvation-free:*
  - ✓ If every process is correct, every process makes progress

- *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, assuming that every process always has some operations to execute

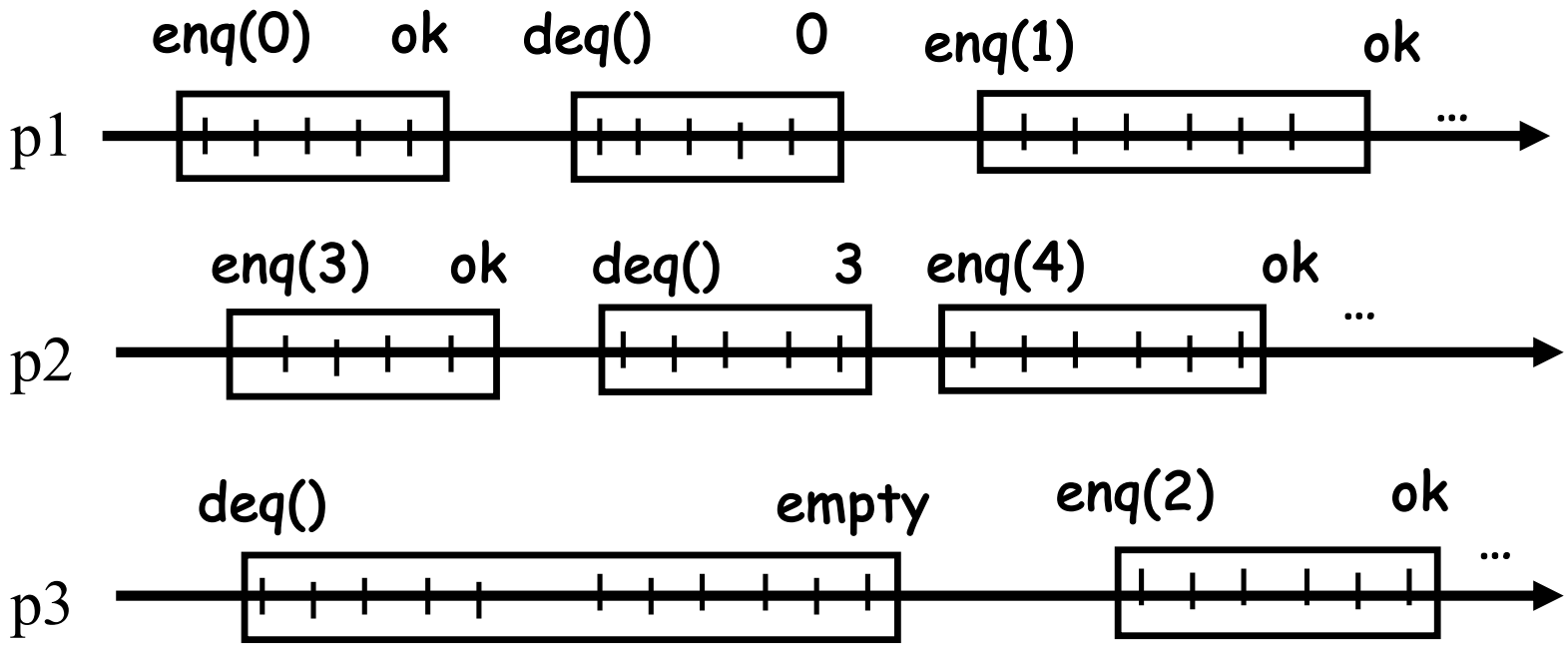
** Completes infinitely many operations

Deadlock-freedom



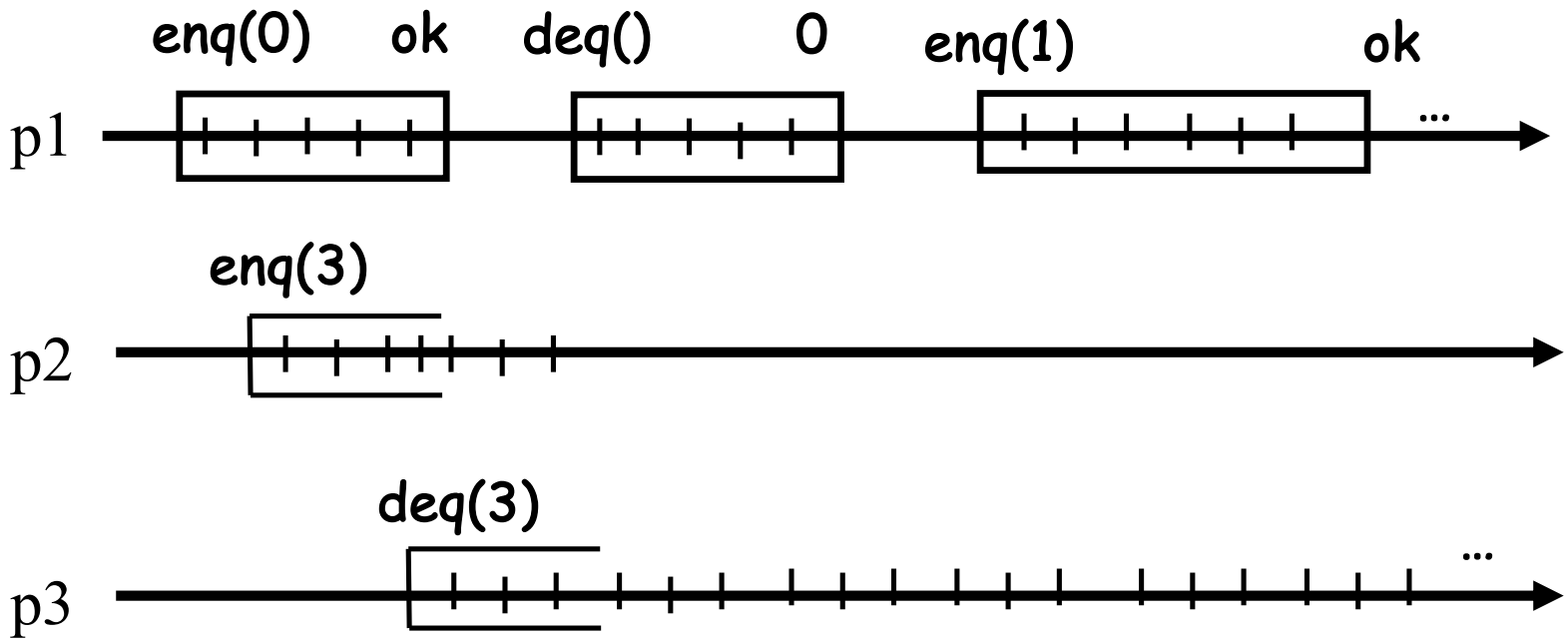
Only p1 makes progress, p2 and p3 busy-wait

Starvation-freedom



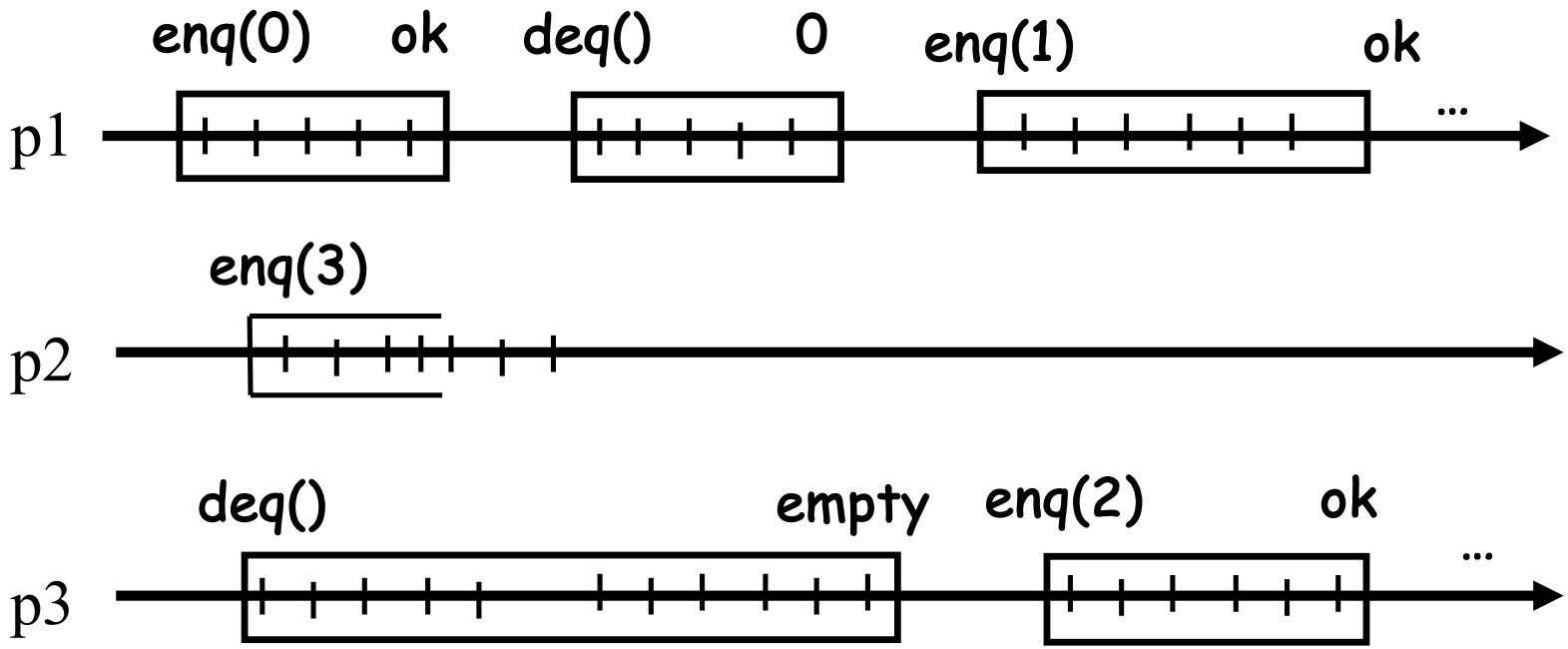
All processes make progress, assuming every process is correct

Lock-freedom



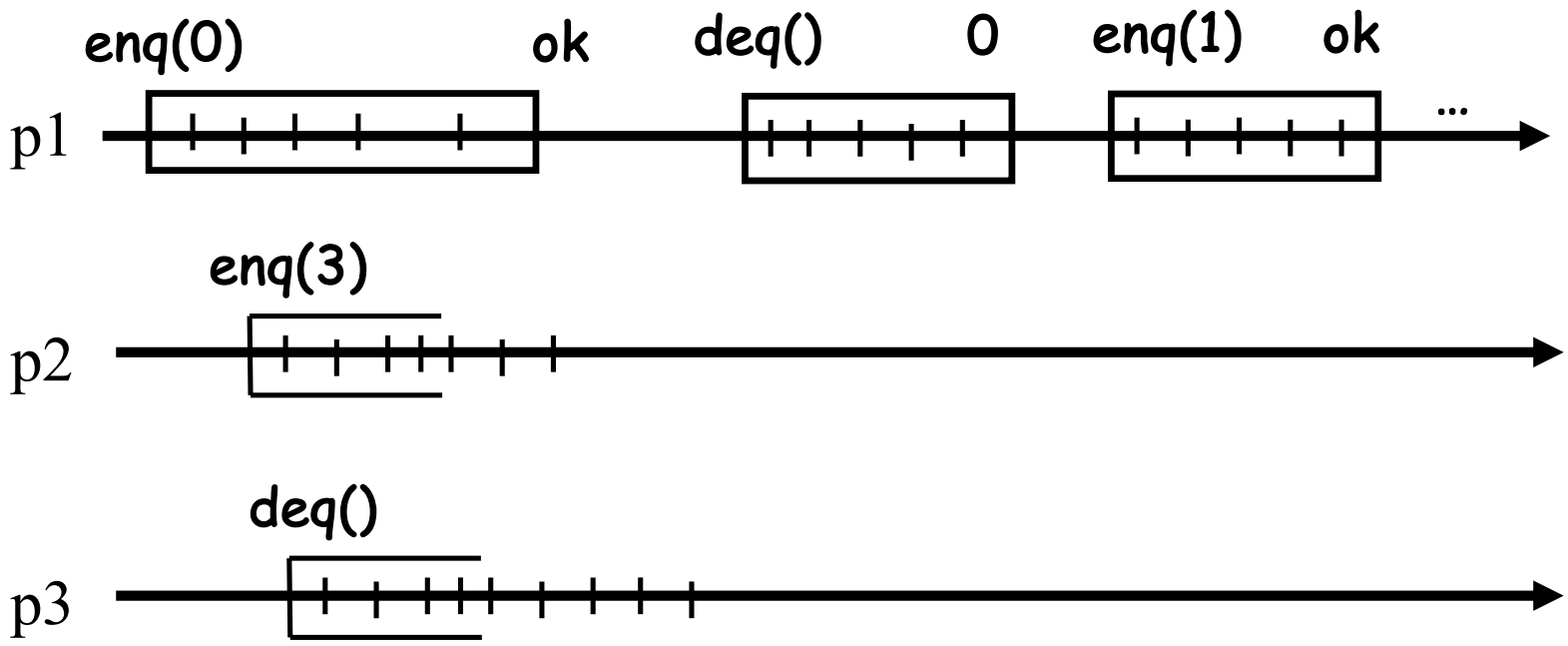
Only p1 makes progress, p2 crashes, p3 busy-waits

Wait-freedom



Both p1 and p3 make progress, even though p2 crashes

Obstruction-freedom



p1 eventually runs solo and makes progress, p2 and p3 crash (suspend taking steps)

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?

Quiz 1.3: 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

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 1.4: linearizability

- Show that the sequential queue implementation considered before is linearizable and wait-free *as is* if used by two processes: one performing only enqueue operations and one performing only dequeue operations

- Devise a simple queue implementation shared by any number of processes in which enqueue and dequeue operations can run concurrently (data races between these operations do not affect correctness)