The Cost of Obstruction-Freedom

Hagit Attiya	Technion
Rachid Guerraoui	EPFL & MIT
Danny Hendler	Technion & BGU
Petr Kouznetsov	MPI-SWS

Based on the following papers

- **Computing with reads and writes in the absence of step contention**

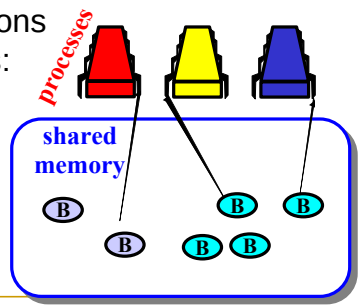
  Attiya, Guerraoui, Kouznetsov, DISC 2005

- **Synchronizing without locks is inherently expensive**

  Attiya, Guerraoui, Hendler, Kouznetsov, PODC 2006

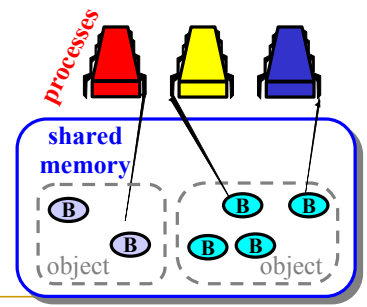
Shared-memory model

- Asynchronous processes
  - cache misses, page fault, quantum used up...
- Apply primitive operations to shared base objects:
  - `read`
  - `write`
  - `Compare & swap`
  - `read-modify-write`



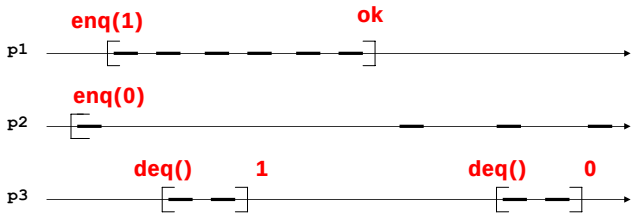
Concurrent Implementations of Shared Objects

- A distributed algorithm providing an illusion of an object implemented in hardware



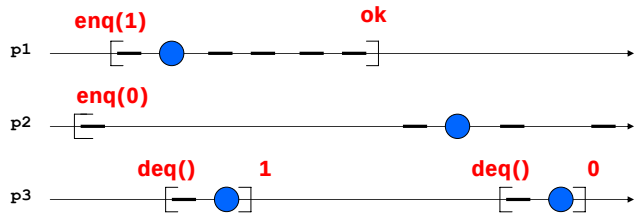
Concurrent Implementations of Shared Objects

- A distributed algorithm providing an illusion of an object implemented in hardware



Safety Property: Linearizability

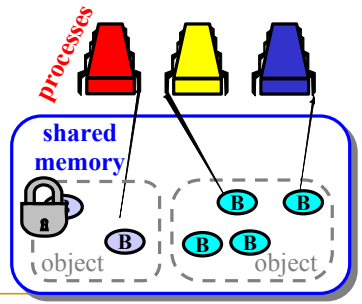
- An operation appears to execute instantaneously between its invocation and response events



Lock-Based Implementations

- Lock the data structure (or parts thereof)
- Apply changes
- Release lock(s)

- Susceptible to:
  - Deadlock
  - Priority inversion
  - Convoying



Lock-Free Implementations

- Ideally: Wait-freedom: Any operation completes in a finite number of its steps
  - regardless of the behavior of other processes

- Alternatively: nonblocking: Some operation completes in a finite number of steps
  - regardless...

Lock-Free Implementations

- Given a distributed shared memory system, which objects can be implemented without locks?
- And at what cost?

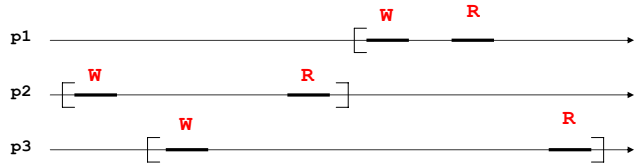
- Depends on the primitive operations available

Any Object from Reads and Writes?

(reads/writes are always available)

- Wait-free / nonblocking **consensus** from reads and writes is impossible

⇨most interesting objects cannot be implemented from reads and writes



Progress only for the Lucky Ones

Why consensus is impossible using reads & writes?

- Steps of concurrent processes interleave

- But step contention is rare in practice
  - Or so we are told…

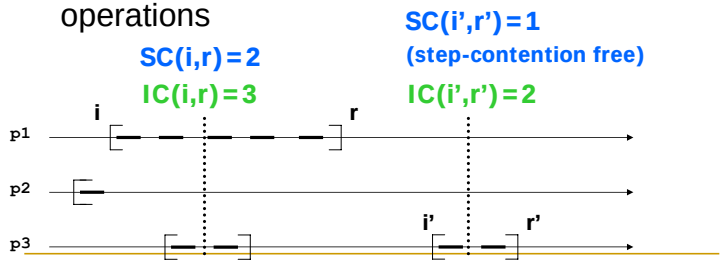
- Make progress only when an operation runs alone, i.e., encounters no **step-contention** (obstruction-freedom)

[Herlihy et al., 2003]

Step Contention

Step contention (**SC**): how many processes take steps concurrently

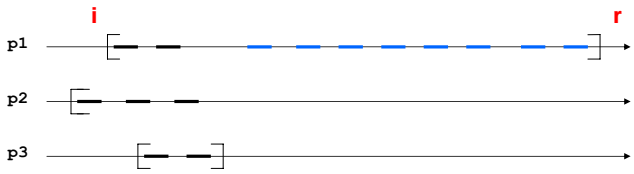
Interval contention (**IC**): number of concurrent operations



When There is No Step Contention

Solo termination: An operation that eventually encounters no step contention must return

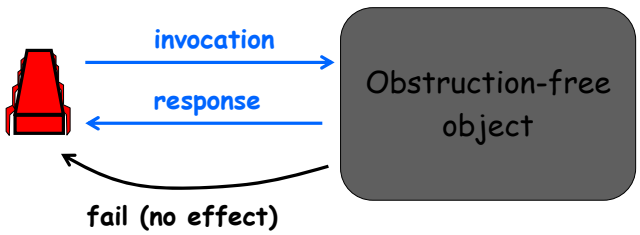
[Fich, Herlihy, Shavit 1998]



When There is Step Contention…

- Keep trying
  - obstruction-free
- Use "expensive" primitives
  - solo-fast
- Fail

Interface to Obstruction-Free Objects



Transaction-like semantics for read/write OF?

Clear Indication of Failure?

Can implement wait-free 2-process consensus with one fail-only consensus object C and one register R

Code for p1:

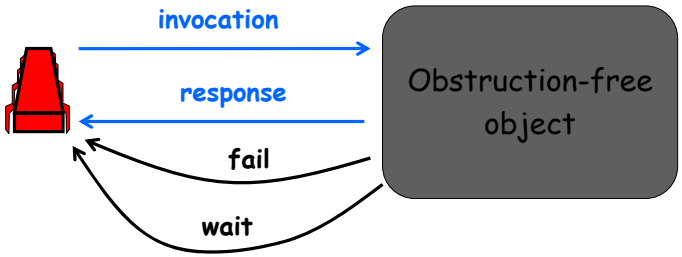
```
propose(v1)
  R := v1
  repeat
     x1=C.propose(v1)
  until x1 <> fail
  return x1
```

Code for p2:

```
propose(v2)
  x2 := C.propose(v2)
  if x2=fail then
     x2:=R
  return x2
```

⇨ Impossible from read/writer registers!

Obstruction-Free Interface (with reads & writes)



In case of step contention:
fail = did not take effect
wait = might have taken effect

Obstruction-Free Objects Exist!

Registers → OF & linearizable implementation of any object

But expensive

- Time complexity
  O(n) steps in a step- contention free operation
- Space complexity
  O(n) registers

And this is optimal!

[Jayanti, Tan & Toueg, 2000]

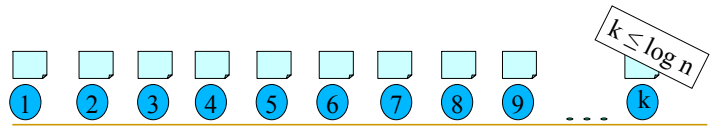
Solo-fast implementations

if no step contention detected **then**
  use registers to terminate
else
  use "expensive" primitives to terminate

Solo-fast implementation of any object:
  linear in time and space

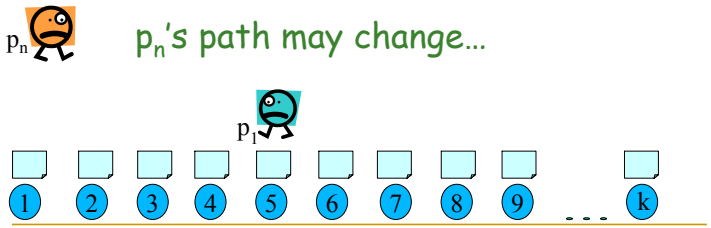
Solo-Fast is not so Fast I

- Given a solo-fast implementation of a counter
- Look at the sequence of base objects p_n accesses in a solo execution of an increment



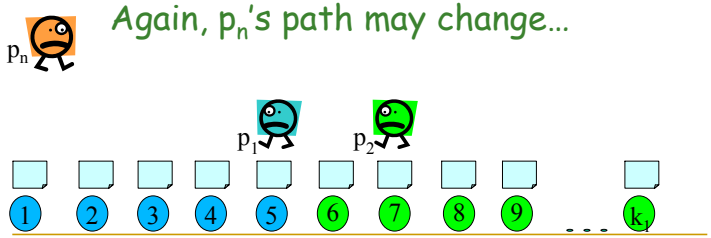
Solo-Fast is not so Fast II

- Let process p_1 execute an increment process
- Must write to an object on p_n'th path
  - Otherwise, counter does not reflect this increment



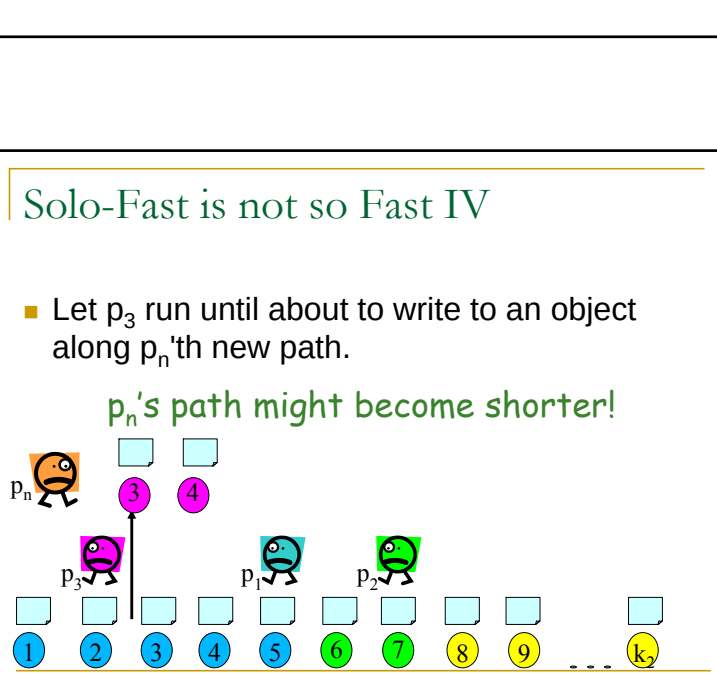
Solo-Fast is not so Fast III

- Let process p_2 run until about to write to an object along p_n'th new path
  - must write to an object other than 5



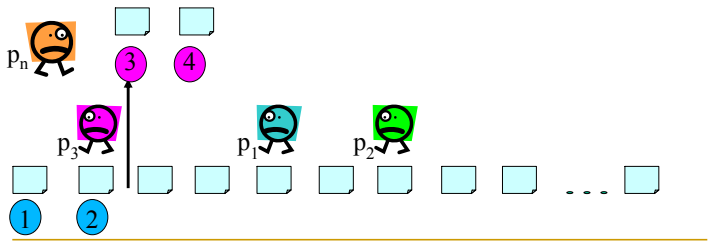
Solo-Fast is not so Fast IV

- Let p_3 run until about to write to an object along p_n'th new path.



Solo-Fast is not so Fast V

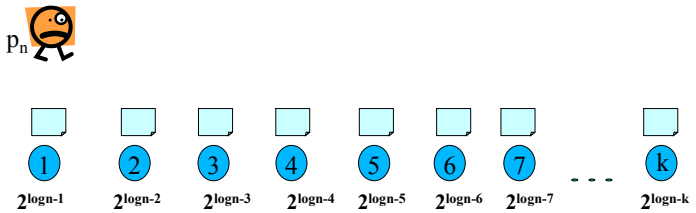
So why must there be a 'long' path?



Logarithmic Lower Bound for Solo-Fast

Define a potential function Ψ on configurations:
$\Psi(c) = \Sigma\ b(i)\ 2^{\log n - i}$
b(i) indicates whether object #i is covered.

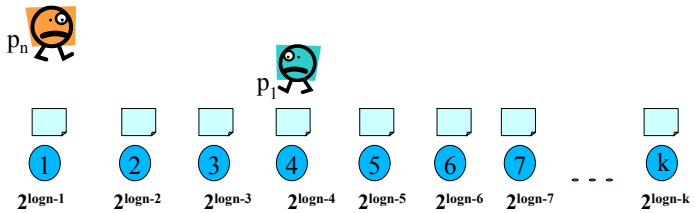
We show that Ψ increases monotonically



Logarithmic Lower Bound for Solo-Fast

$\Psi = 0$

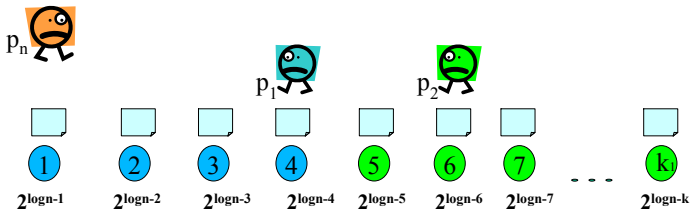
Pick process p_1 and let it run until about to write to an object along p_n'th path.



Logarithmic Lower Bound for Solo-Fast

$\Psi = 2^{\log n-4}$

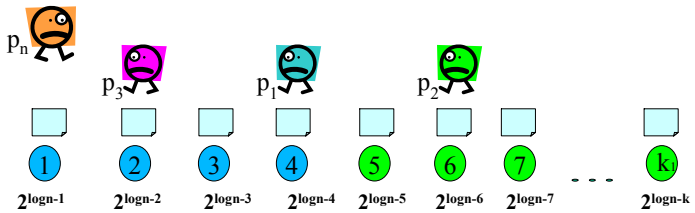
Pick process p_2 and let it run until about to write to an object along p_n'th new path.



Logarithmic Lower Bound for Solo-Fast

$\Psi = 2^{\log n-4} + 2^{\log n-6}$

Pick process p_3 and let it run until about to write to an object along p_n'th new path.

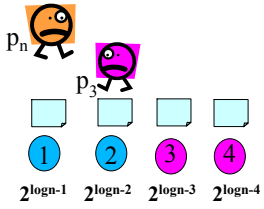


Logarithmic Lower Bound for Solo-Fast

$\Psi = 2^{\log n-2}$

Ψ monotonically increasing in [0,n-1]

We have n-1 processes to deploy



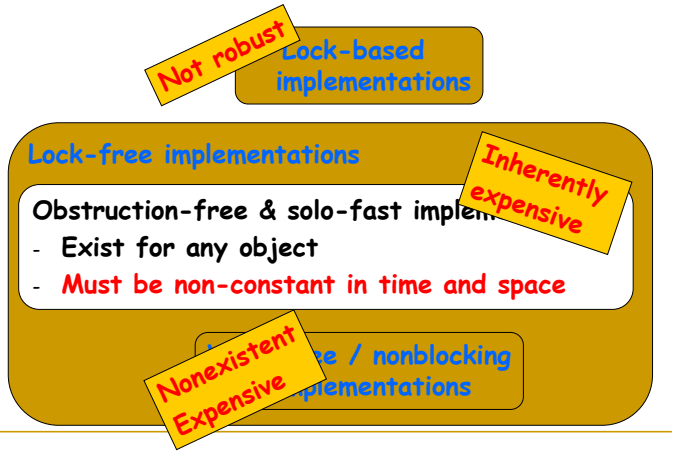
Ψ reaches n-1

Path of length log n exists

More Lower Bounds

- Also a linear space lower bound
- Bounds for obstruction-free implementations from arbitrary primitives:
  - When memory contention is taken into account
  - A $\sqrt{n}$ lower bound
  - A linear time lower bound without failing
- All bounds hold for implementations of perturbable objects from historyless primitives

Bottom Line



What We'd Like to Know…

- Still an exponential gap for solo-fast implementations time complexity of perturbable objects…
- Is the complexity of obstruction-free consensus at least non-constant?
- Can we reduce the complexity of solo-fast implementations using slightly more powerful primitives on the fast path?
  - E.g., queues, fetch & inc