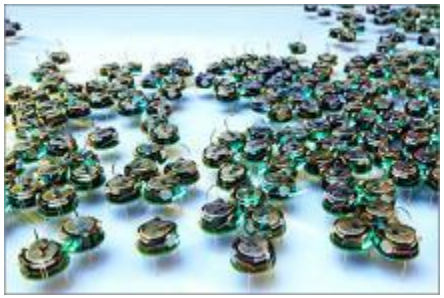
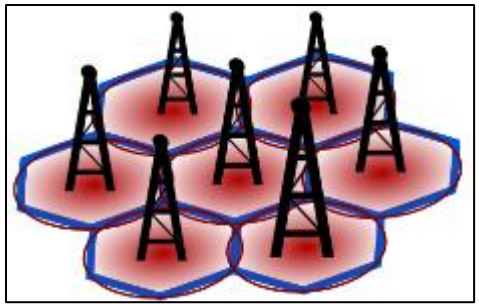
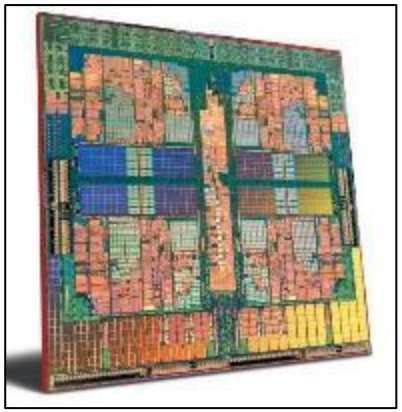
Distributed Computing

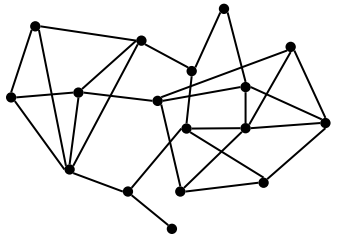






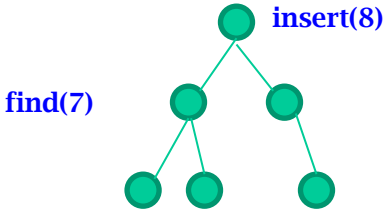
Background: Distributed Computing

- Distributed graph algorithms



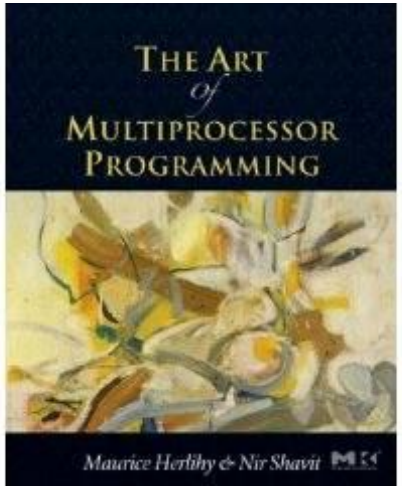
- Large-scale parallel computation

- Concurrent data structures and programming primitives
- Biological systems



The Plan

- Part 1: concurrency in shared memory
- Part 2: distributed graph algorithms
- Part 3: lower bounds & communication complexity



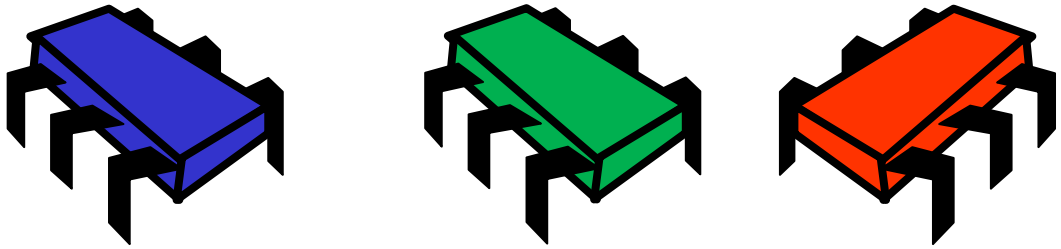
Companion slides for

The Art of Multiprocessor Programming

by Maurice Herlihy & Nir Shavit

Intel x86 Instruction Set

Modeling Concurrency

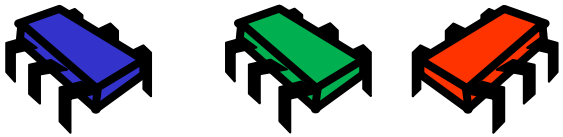


Shared Memory

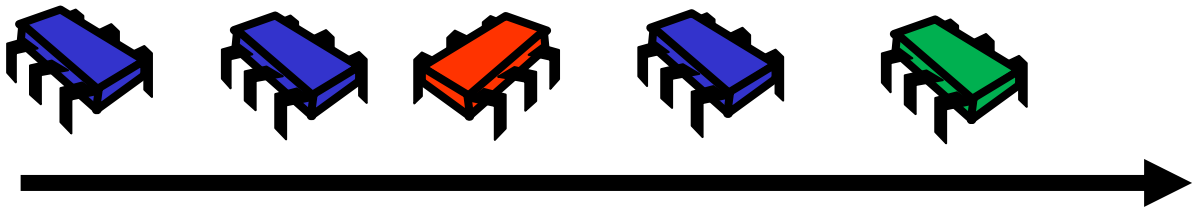
1001101110100

Modeling Concurrency

- n processes



- A run: sequence of **interleaved** process steps



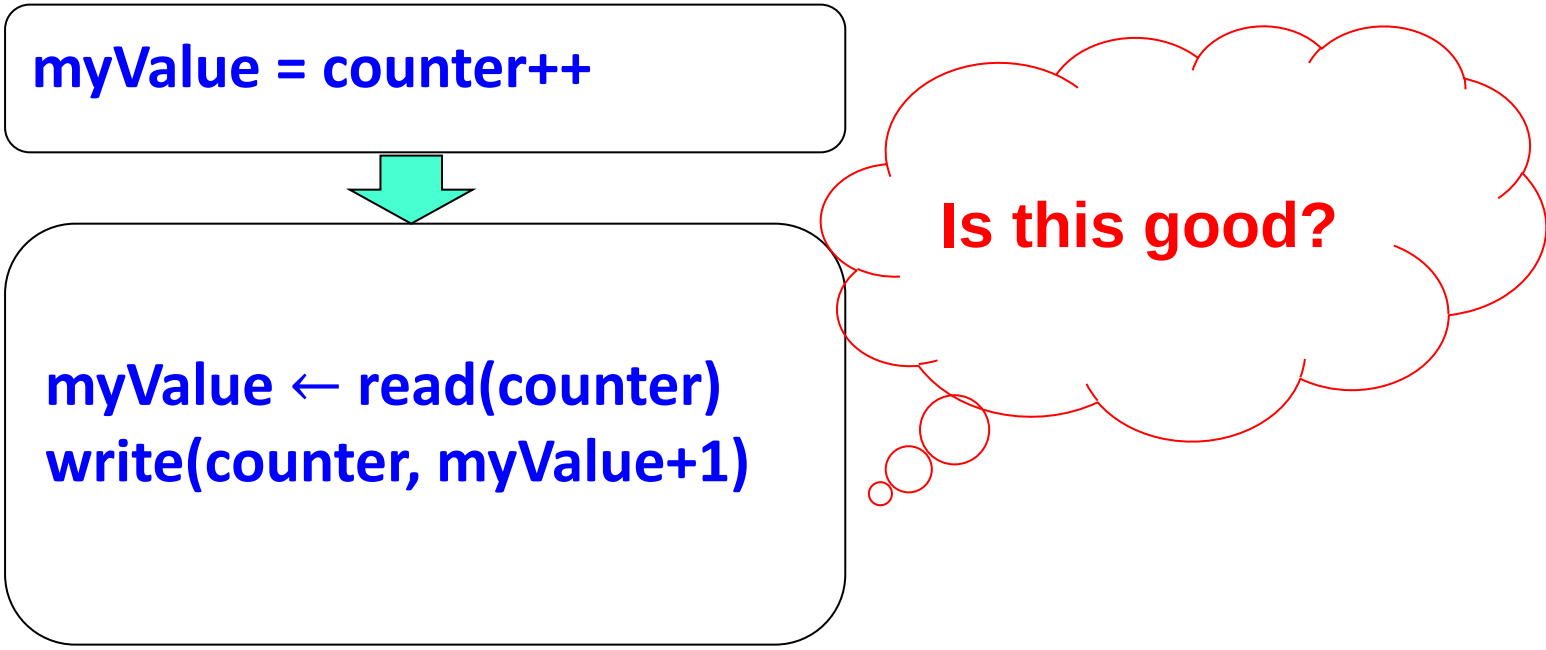
- Order chosen by adversary

Modeling Concurrency

- Shared memory:
  - Collection of objects
- Process step: choose a shared object and *interact* with it
- For now assume: only read/write

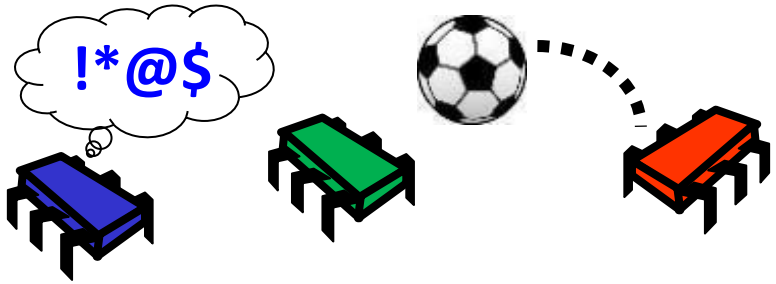
Modeling Concurrency

- Example: shared counter

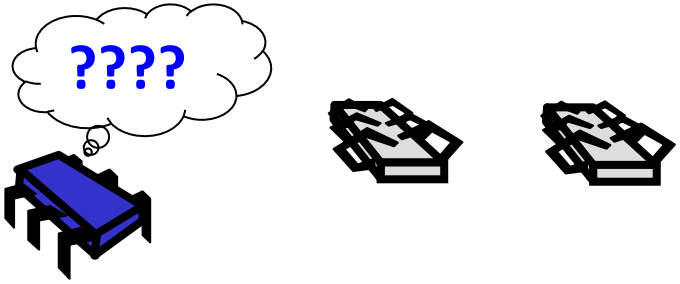


Wait-Free Implementation

- Every process finishes in a finite number of its own steps
  - Even if other processes very active

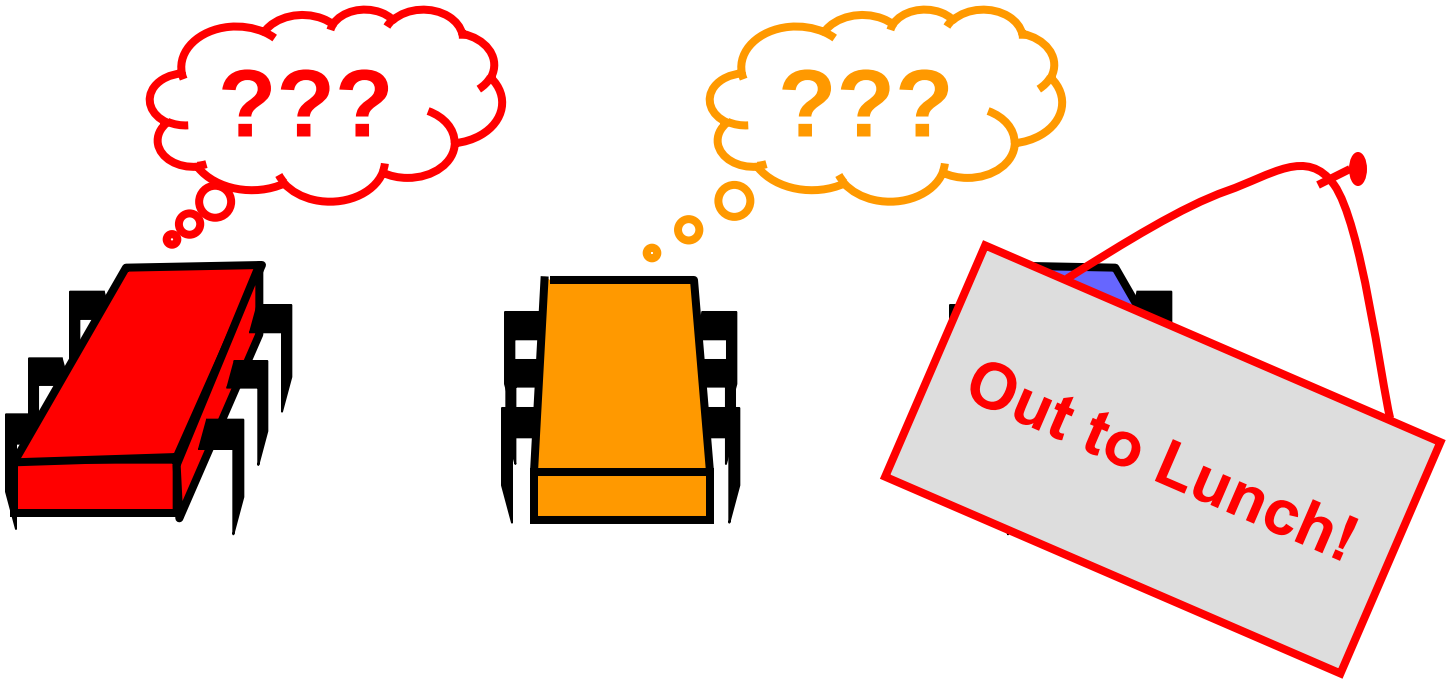


  - Even if other processes die

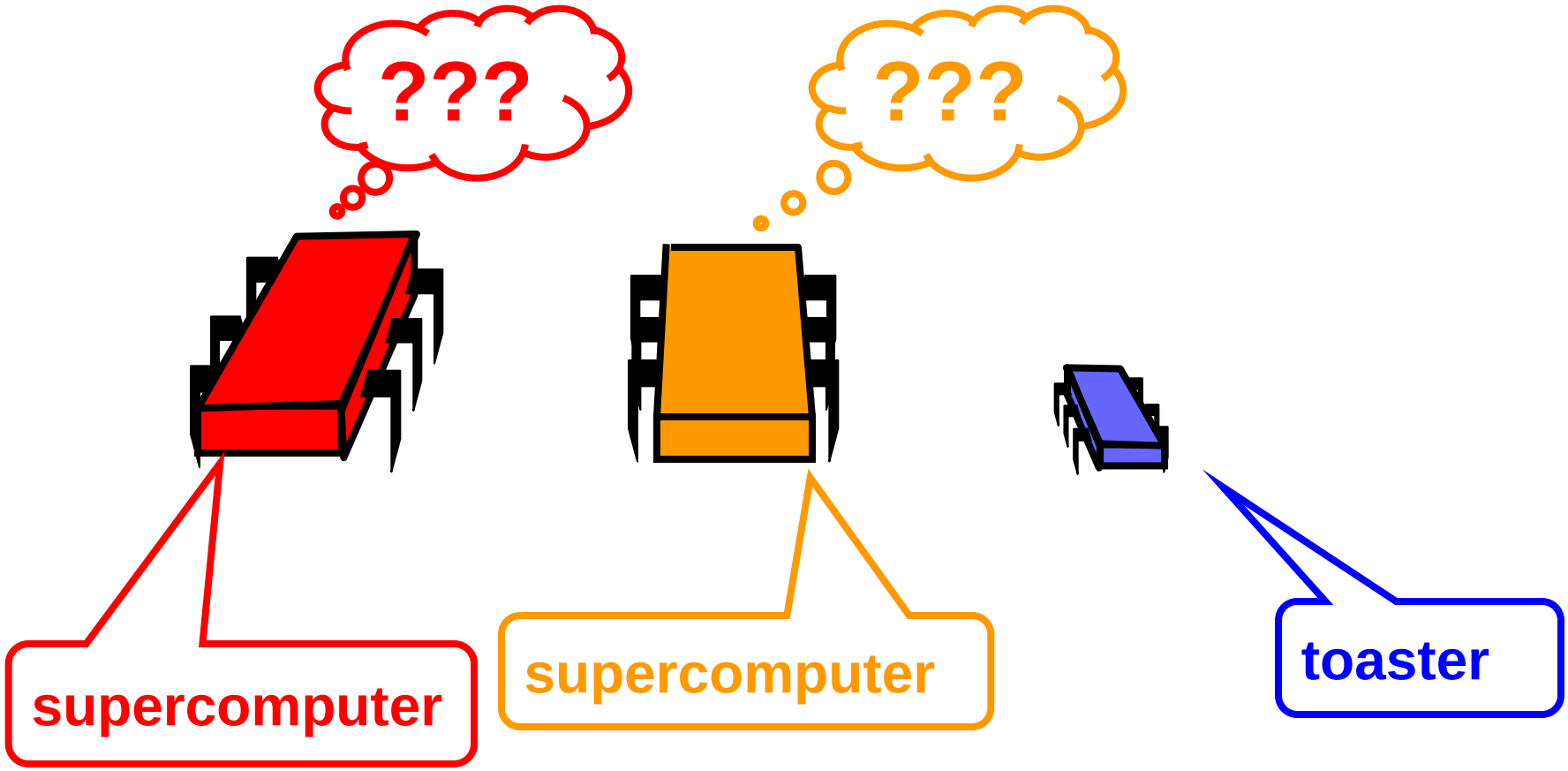


Why wait-freedom?

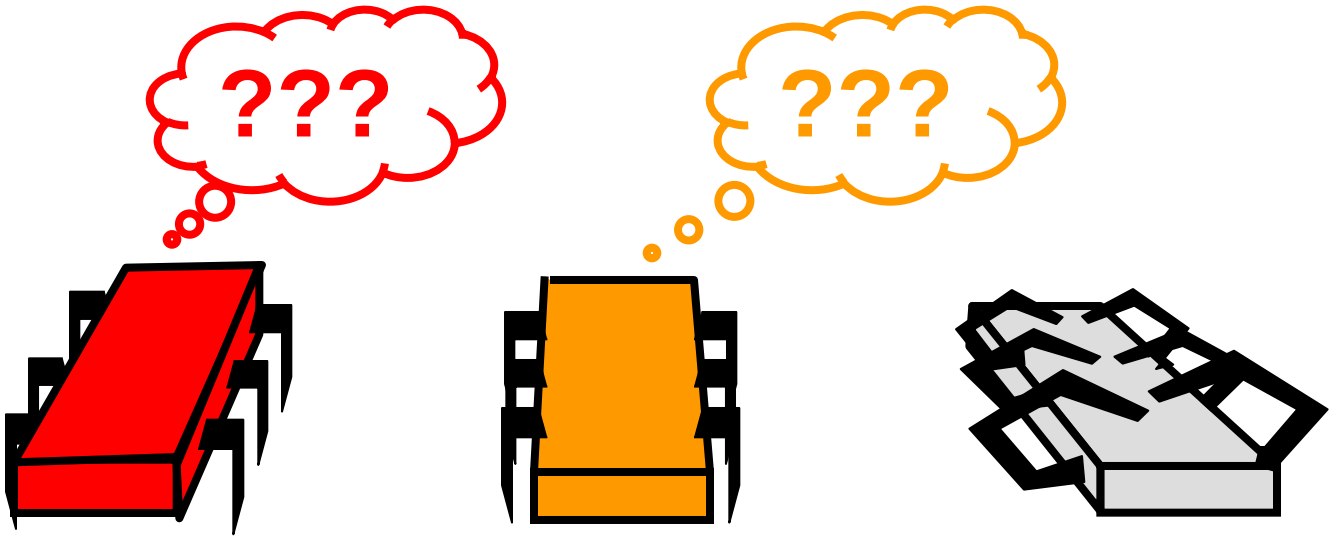
Asynchronous Interrupts



Heterogeneous Processors

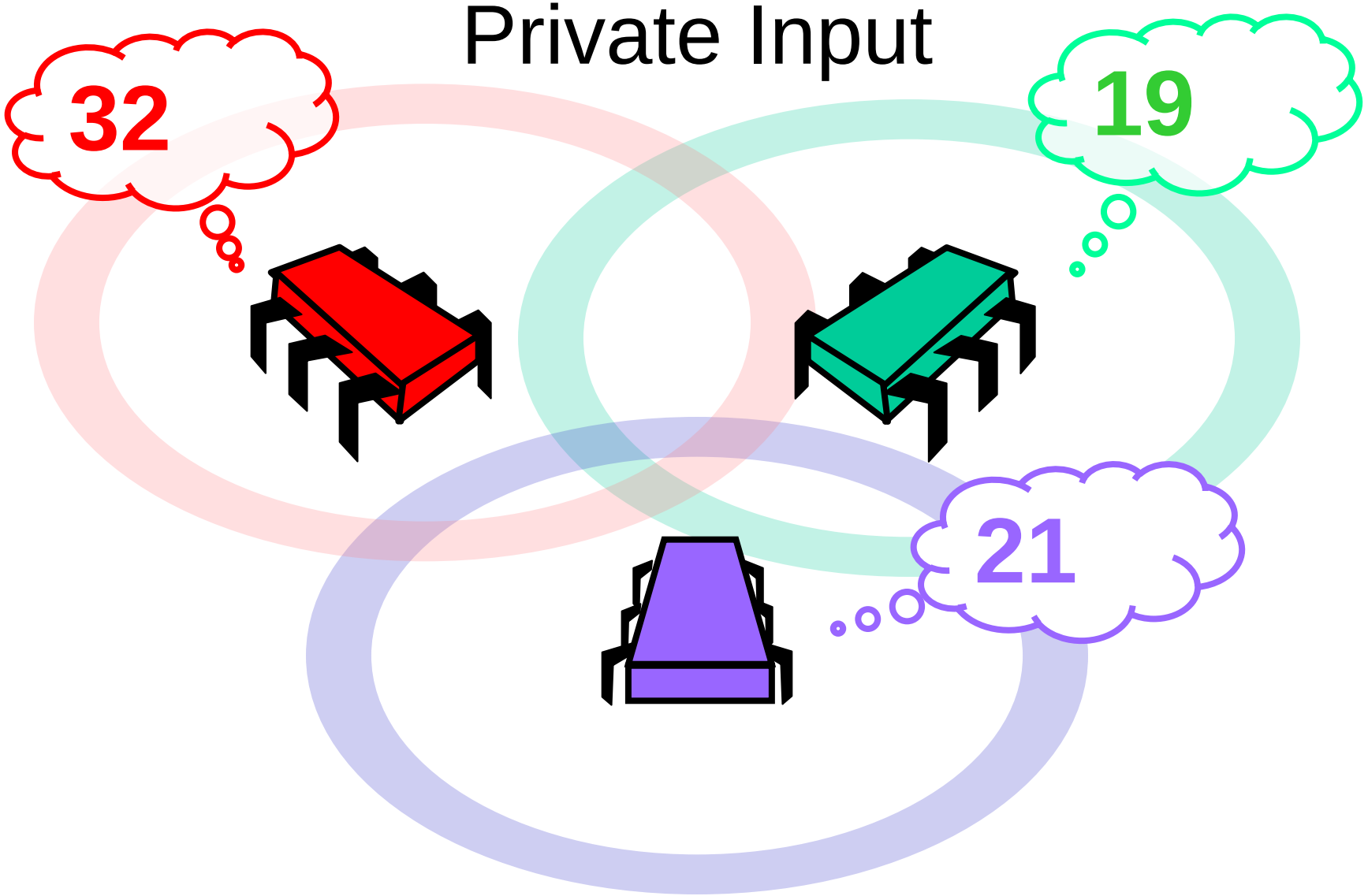


Fault-tolerance

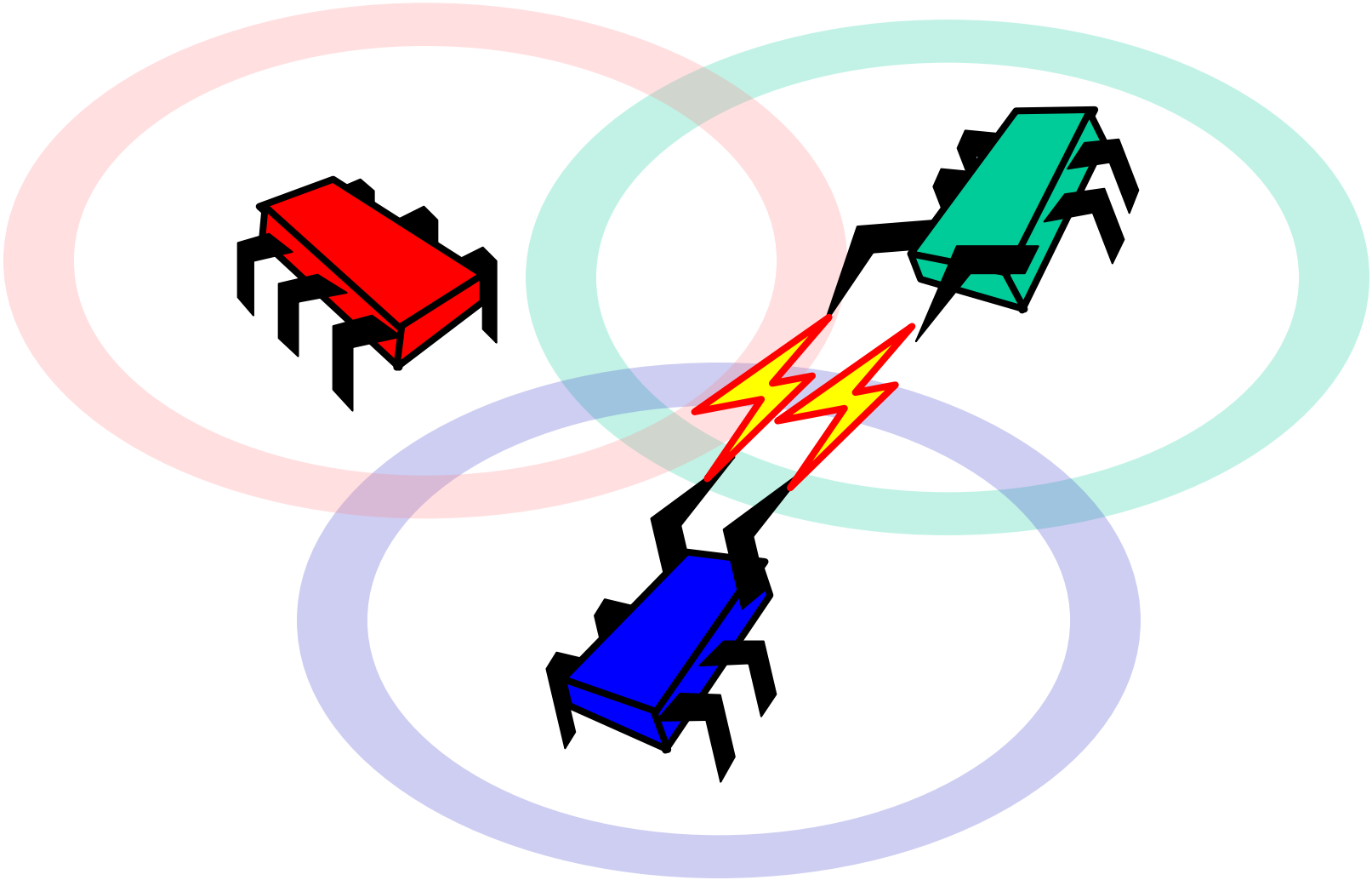


A SIMPLE TASK: CONSENSUS

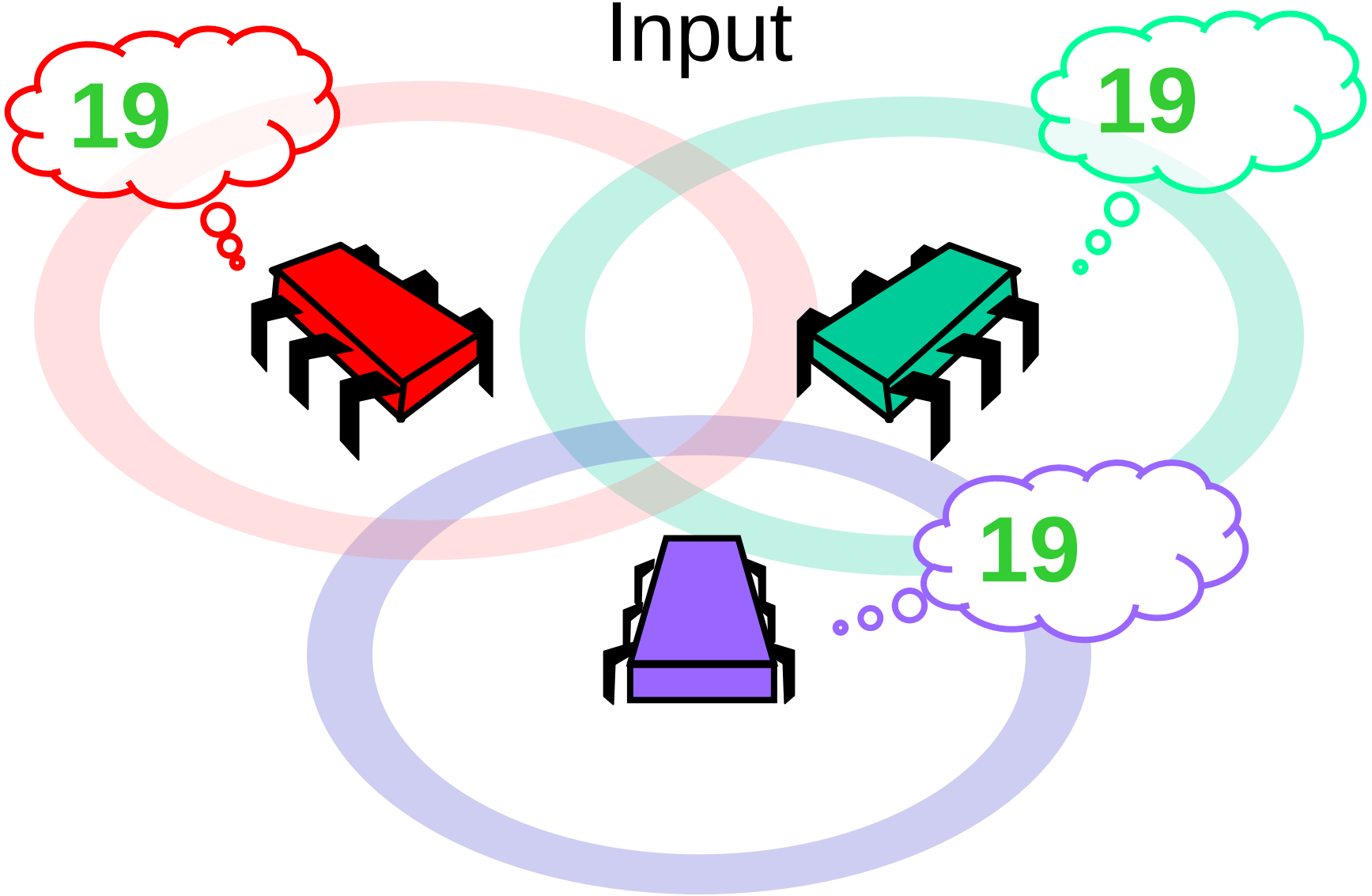
Consensus: Each Thread has a Private Input



They Communicate



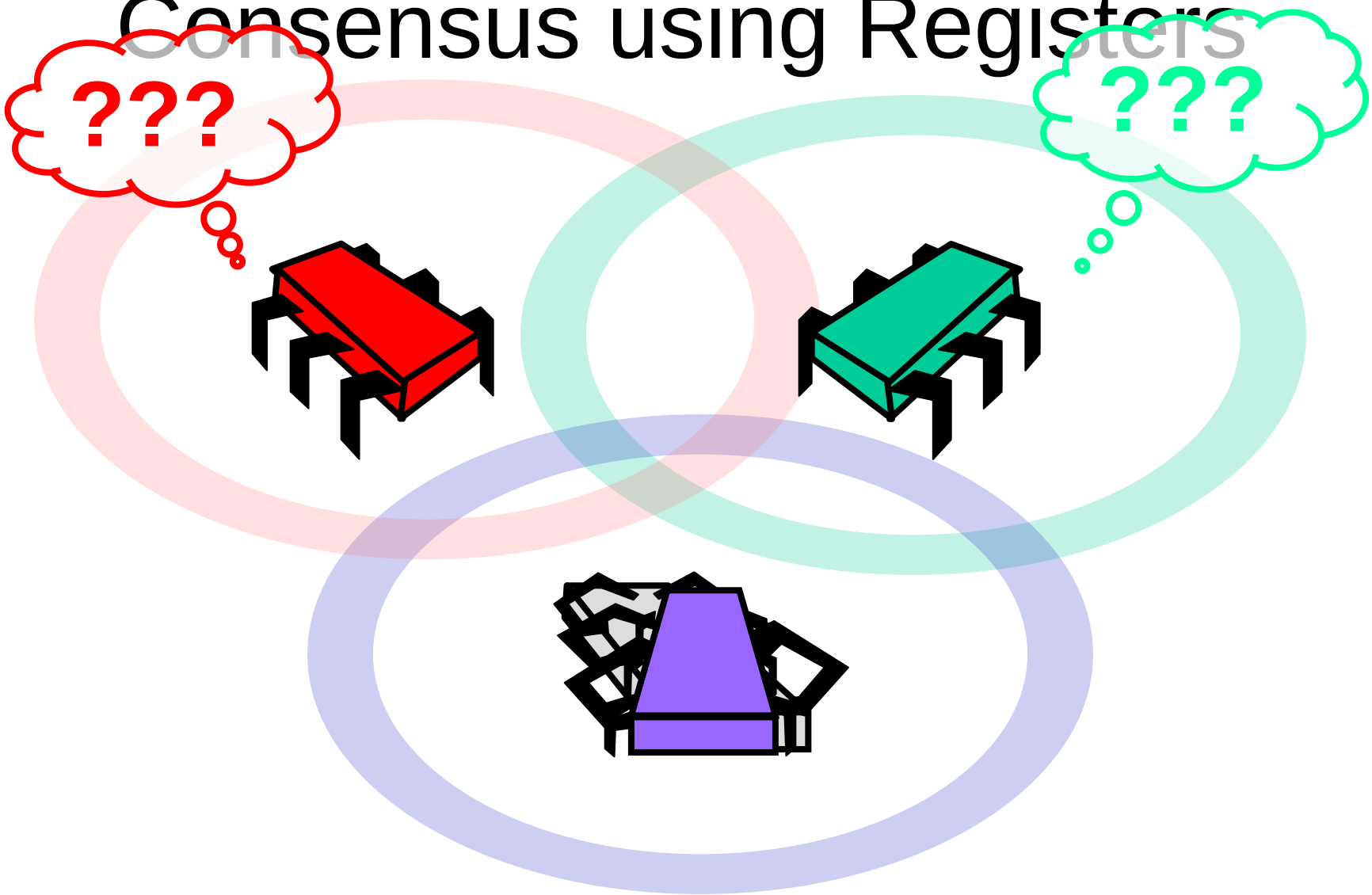
They Agree on One Thread's Input



Formally: Consensus

- Agreement:
  - all threads decide the same value
- Validity:
  - the common decision value is some thread's input

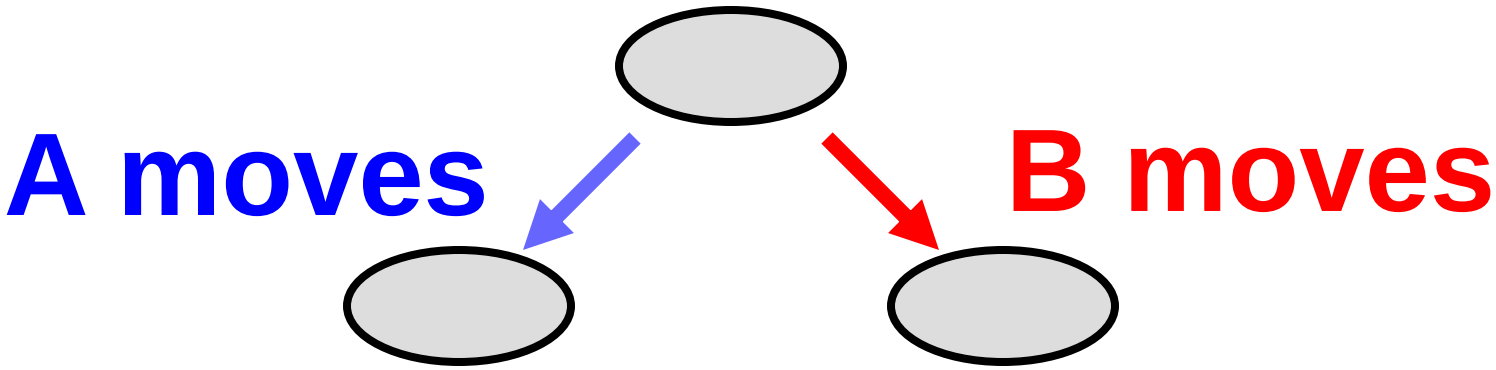
No Wait-Free Implementation of Consensus using Registers



Formally

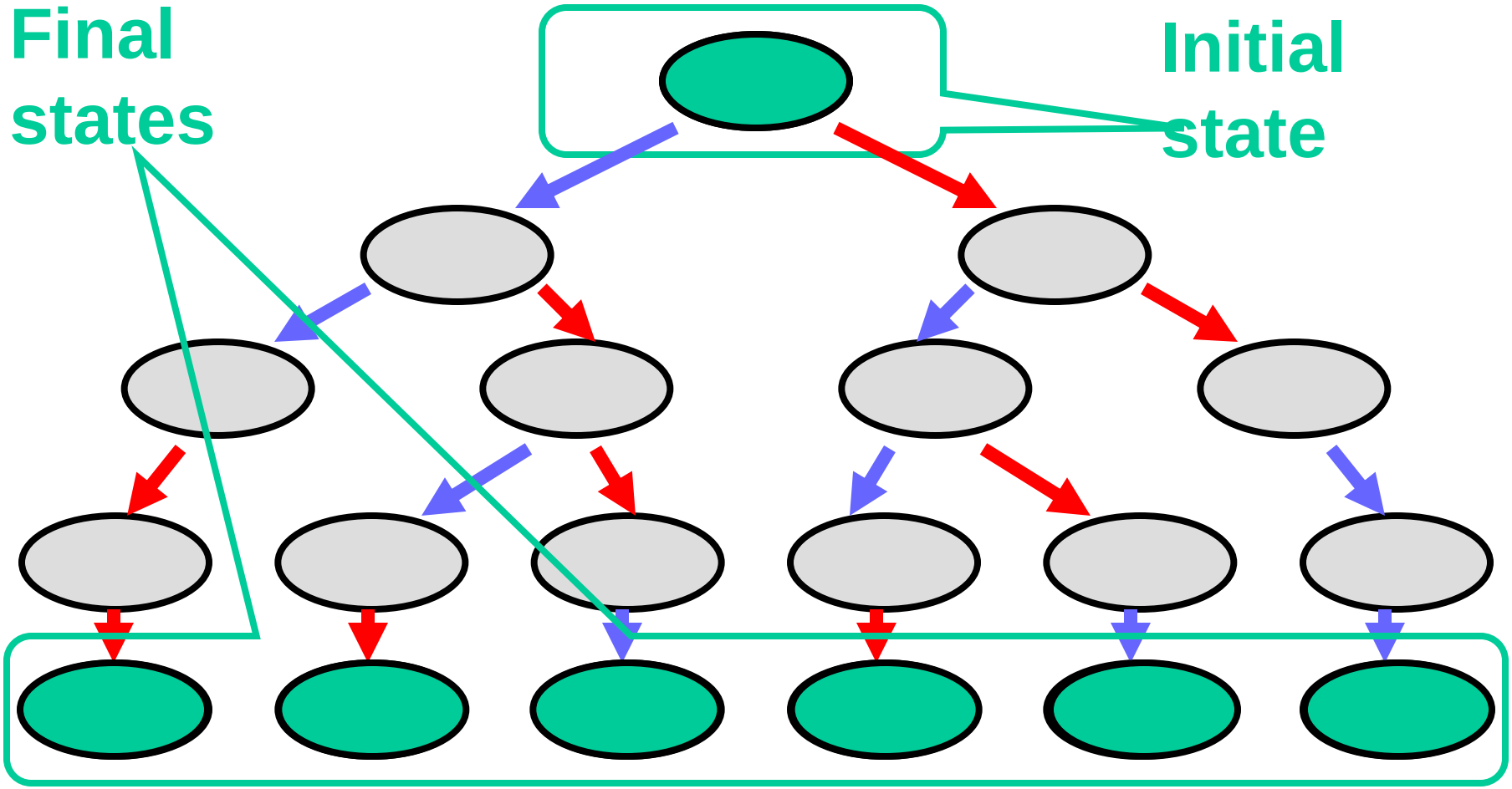
- Theorem
  - There is no wait-free implementation of *n*-thread consensus from read-write registers

Wait-Free Computation

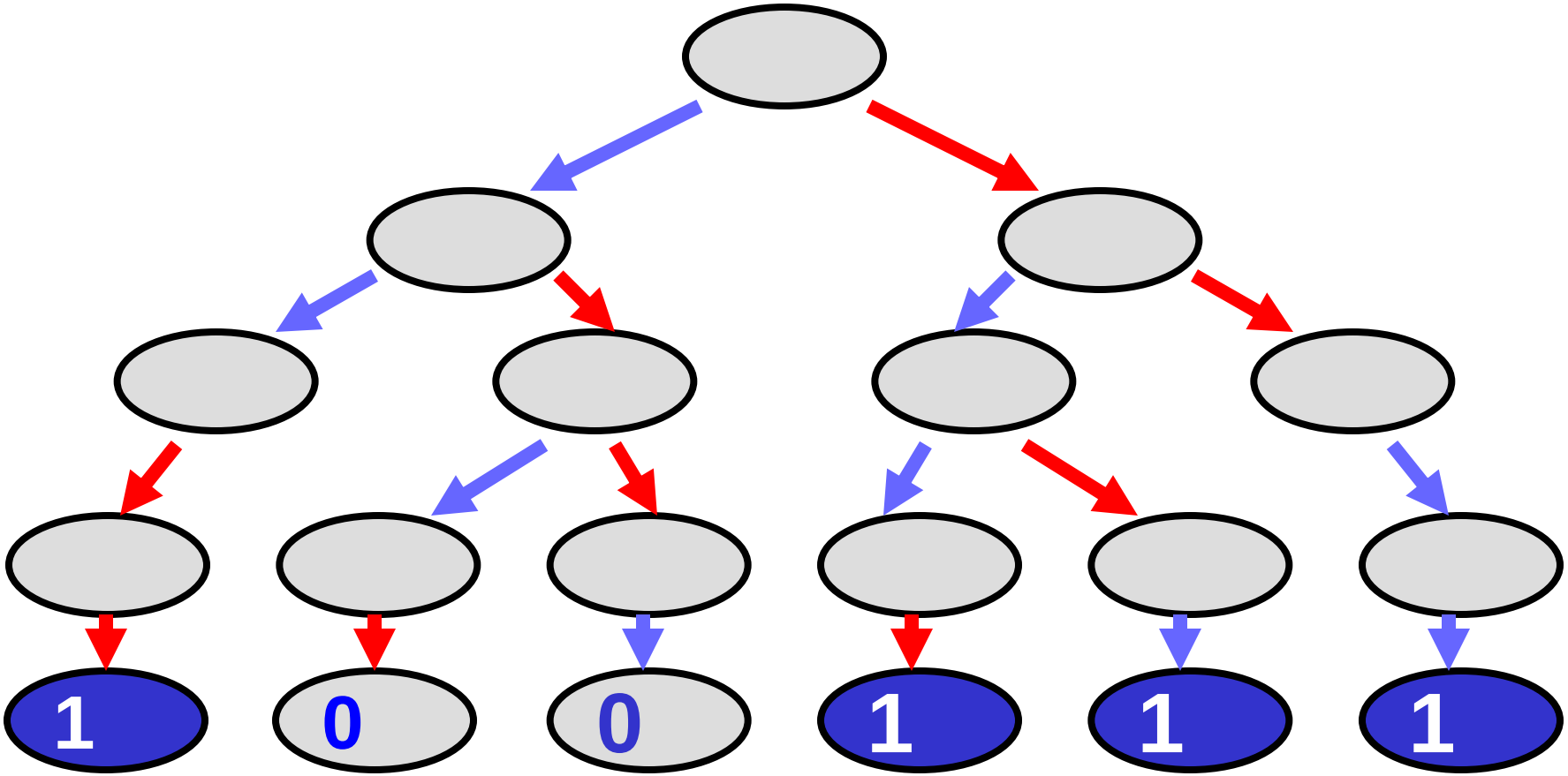


- Either A or B “moves”
- Moving means
  - Register read
  - Register write

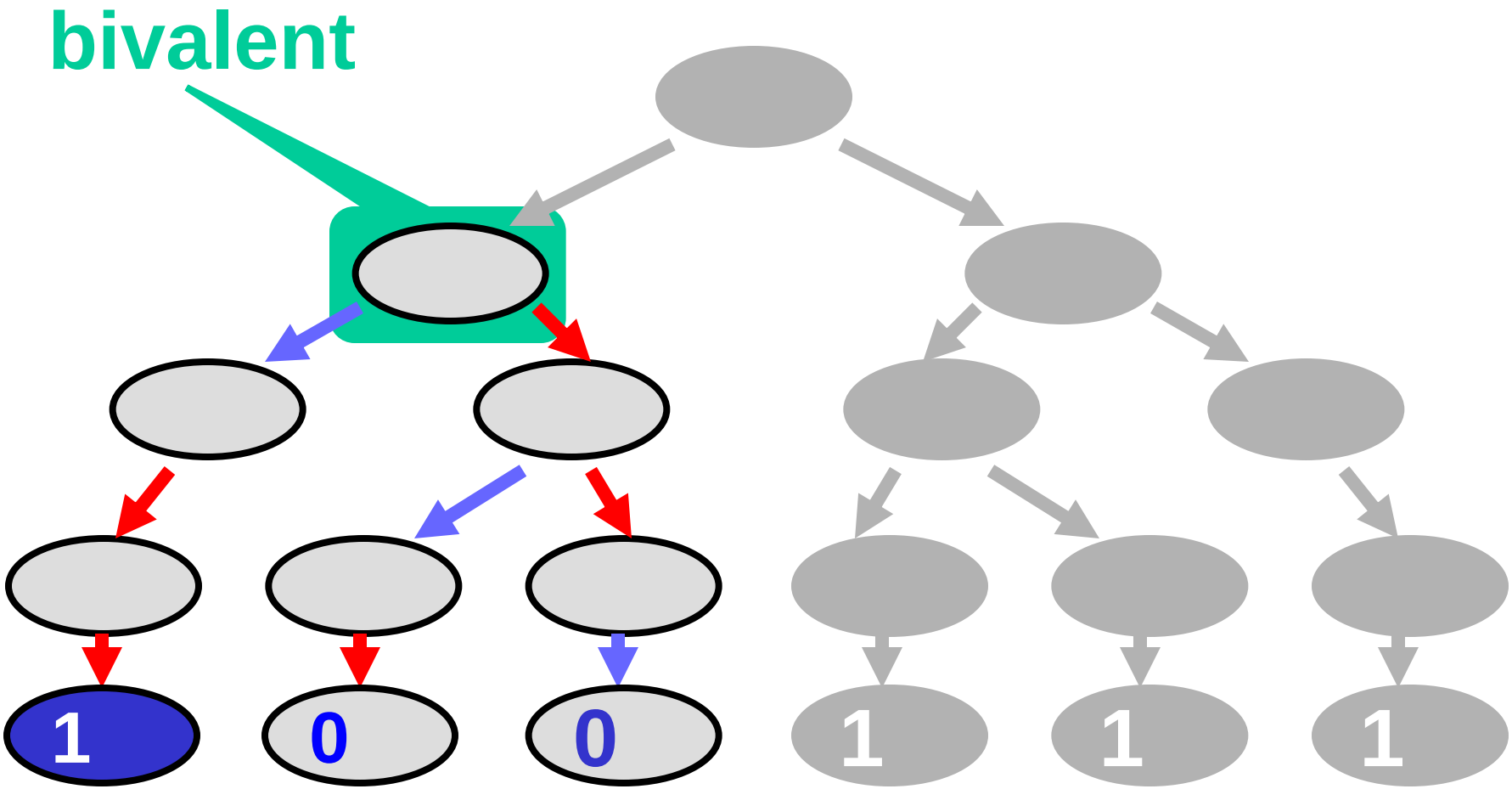
The Two-Move Tree



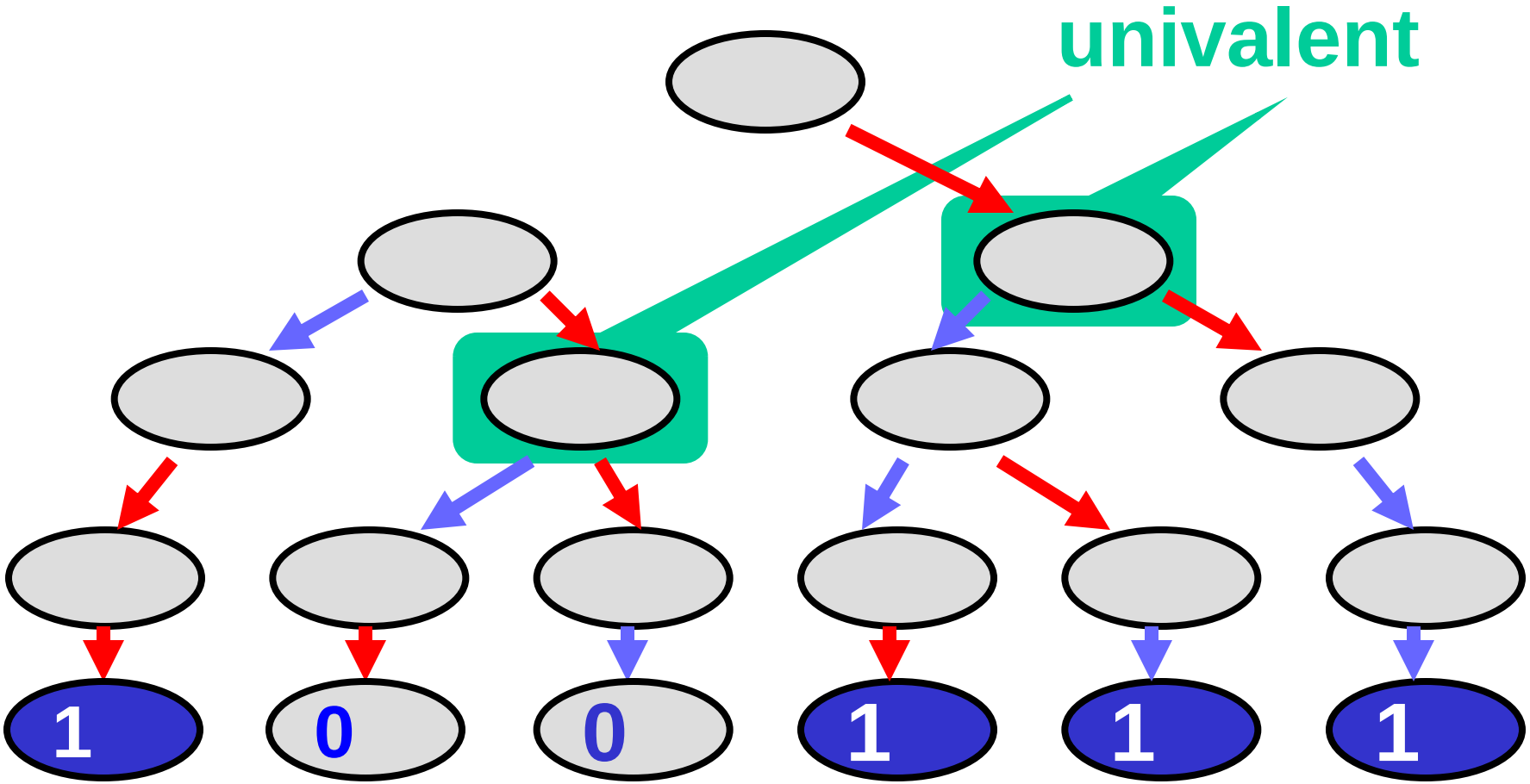
Decision Values



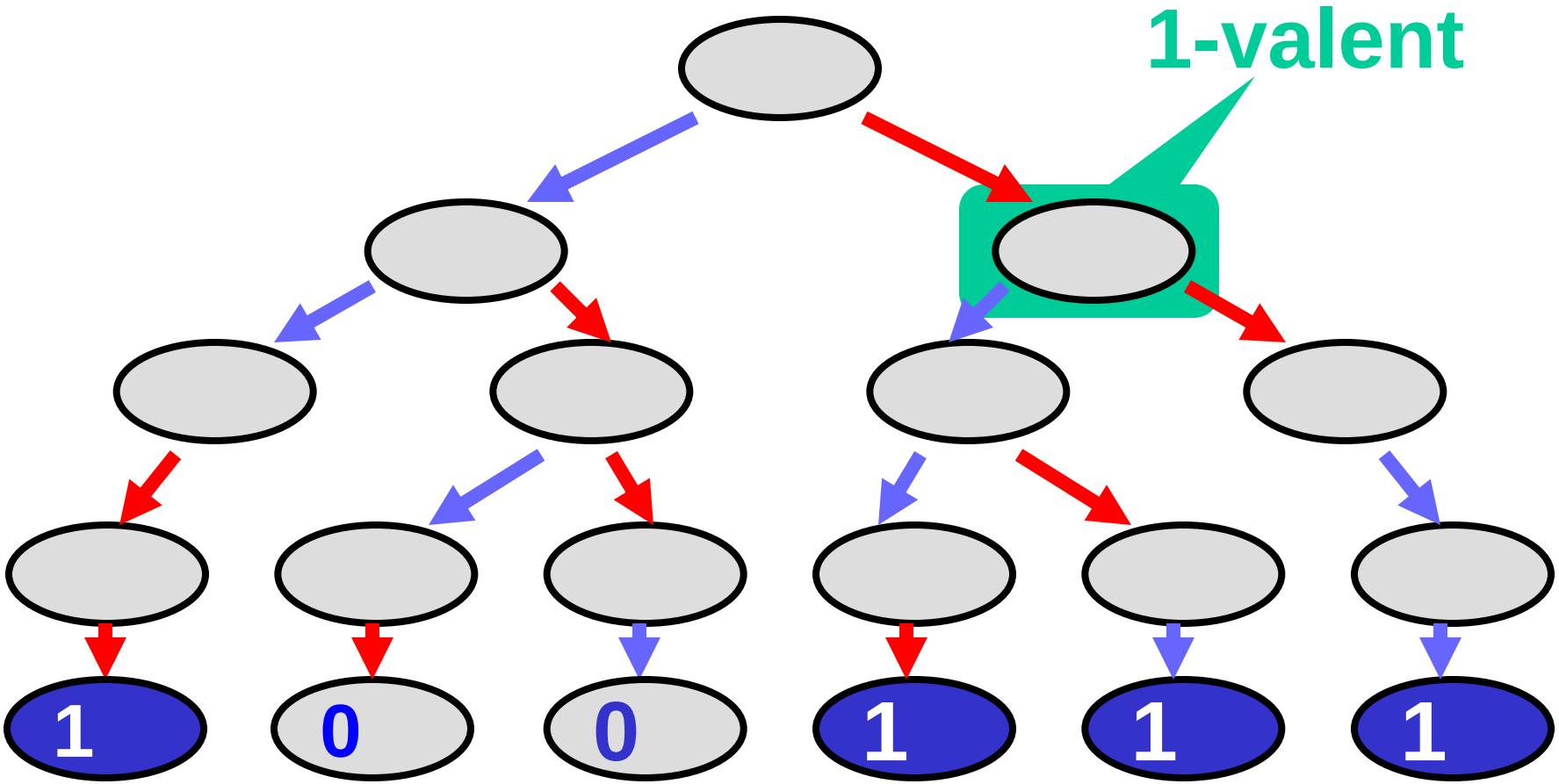
Bivalent: Both Possible



Univalent: Single Value Possible



x-valent: x Only Possible Decision



Summary

- Wait-free computation is a tree
- Bivalent system states
  - Outcome not fixed
- Univalent states
  - Outcome is fixed
  - May not be “known” yet
- 1-Valent and 0-Valent states

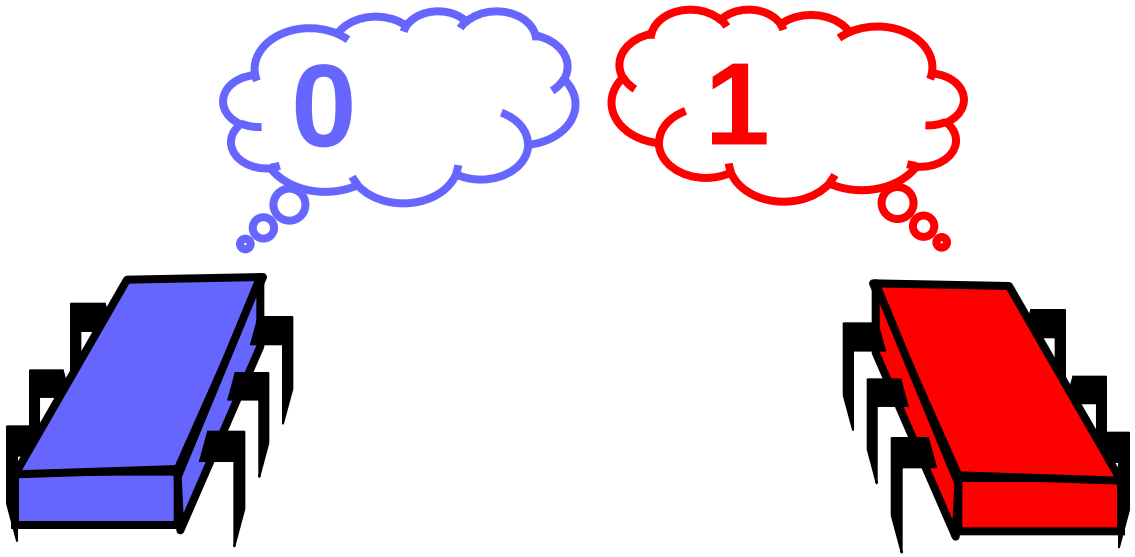
Claim

- Some initial state is bivalent

Claim

- Some initial state is bivalent
- Outcome depends on
  - Chance
  - Whim of the scheduler

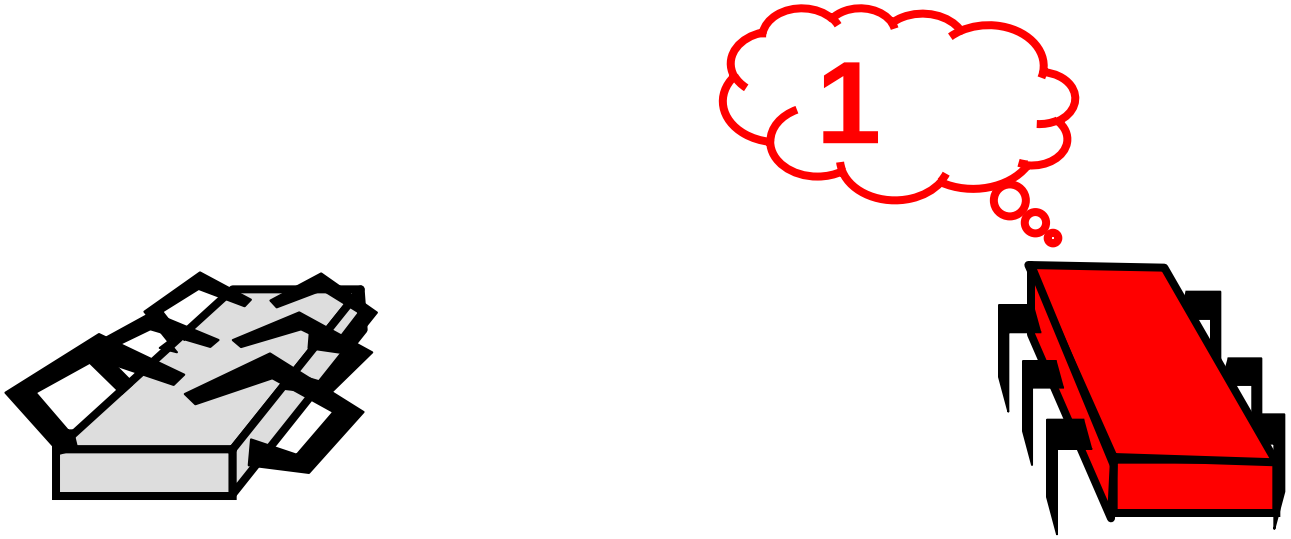
What if inputs differ?



Solo execution by A

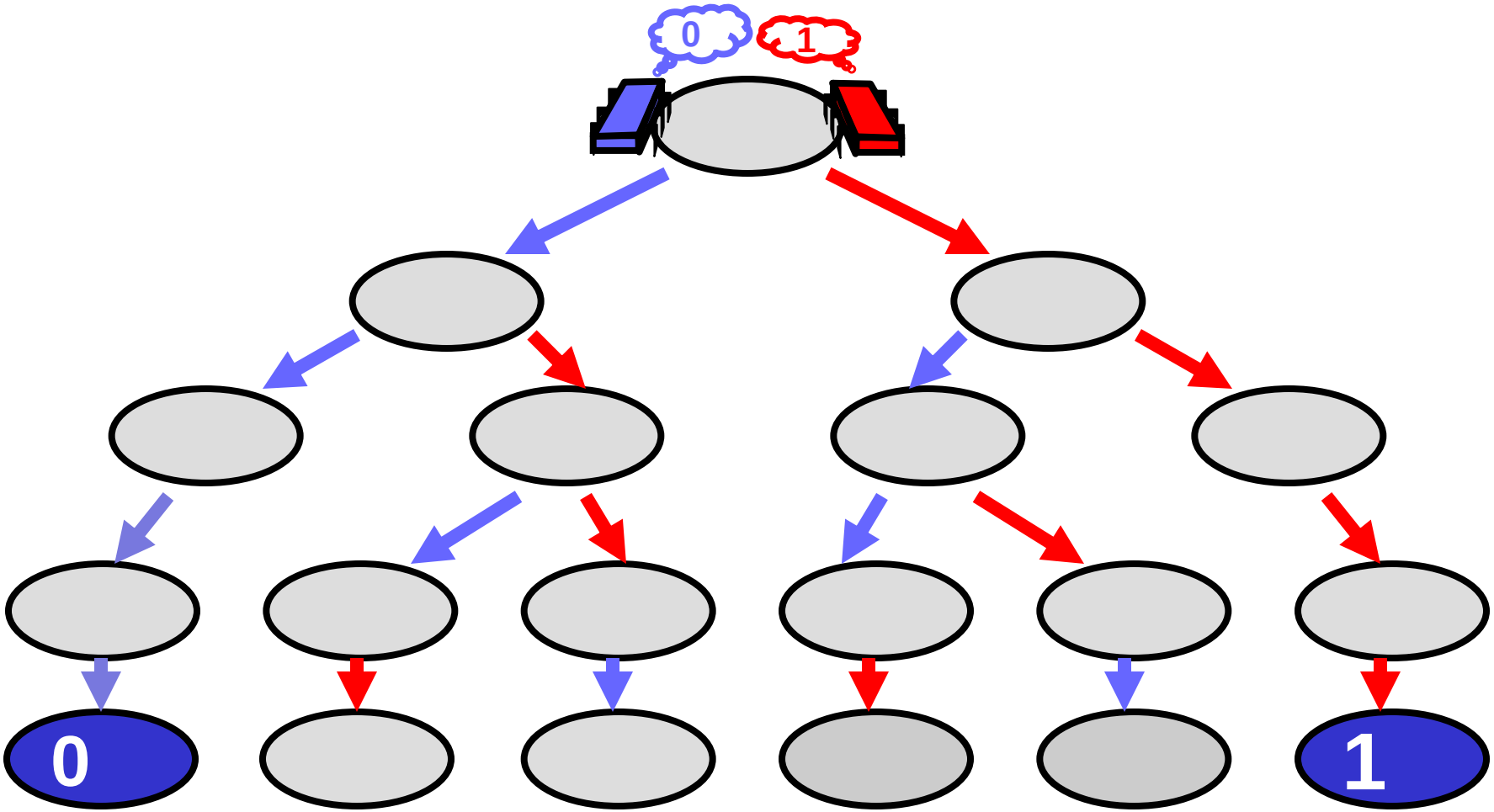
Must decide 0

Solo execution by B

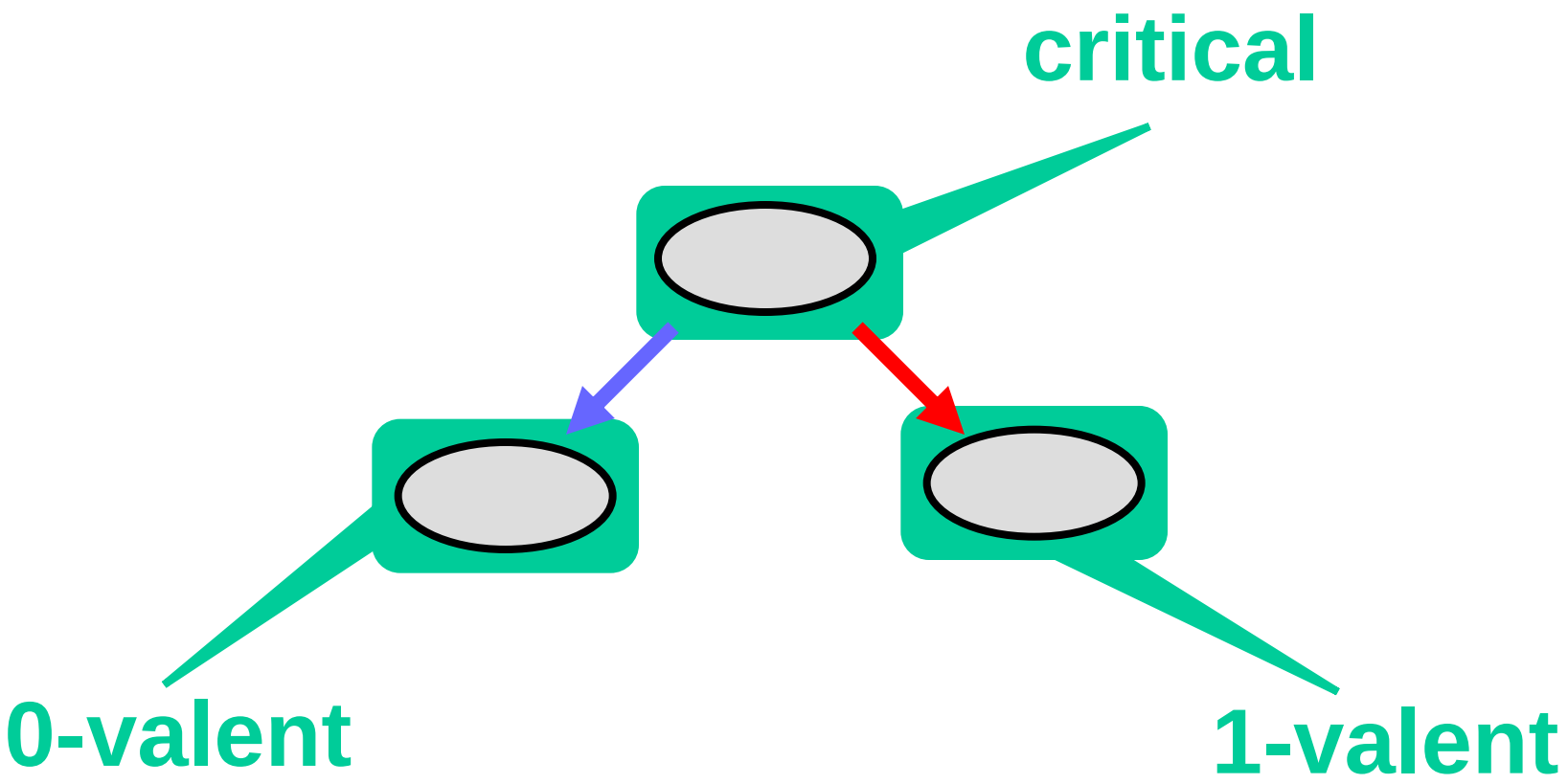


Must decide 1

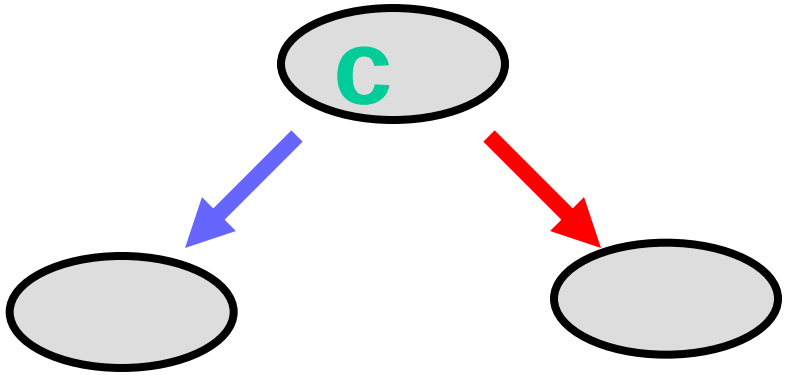
Bivalent initial state



Critical States



From a Critical State



0-valent **1-valent**

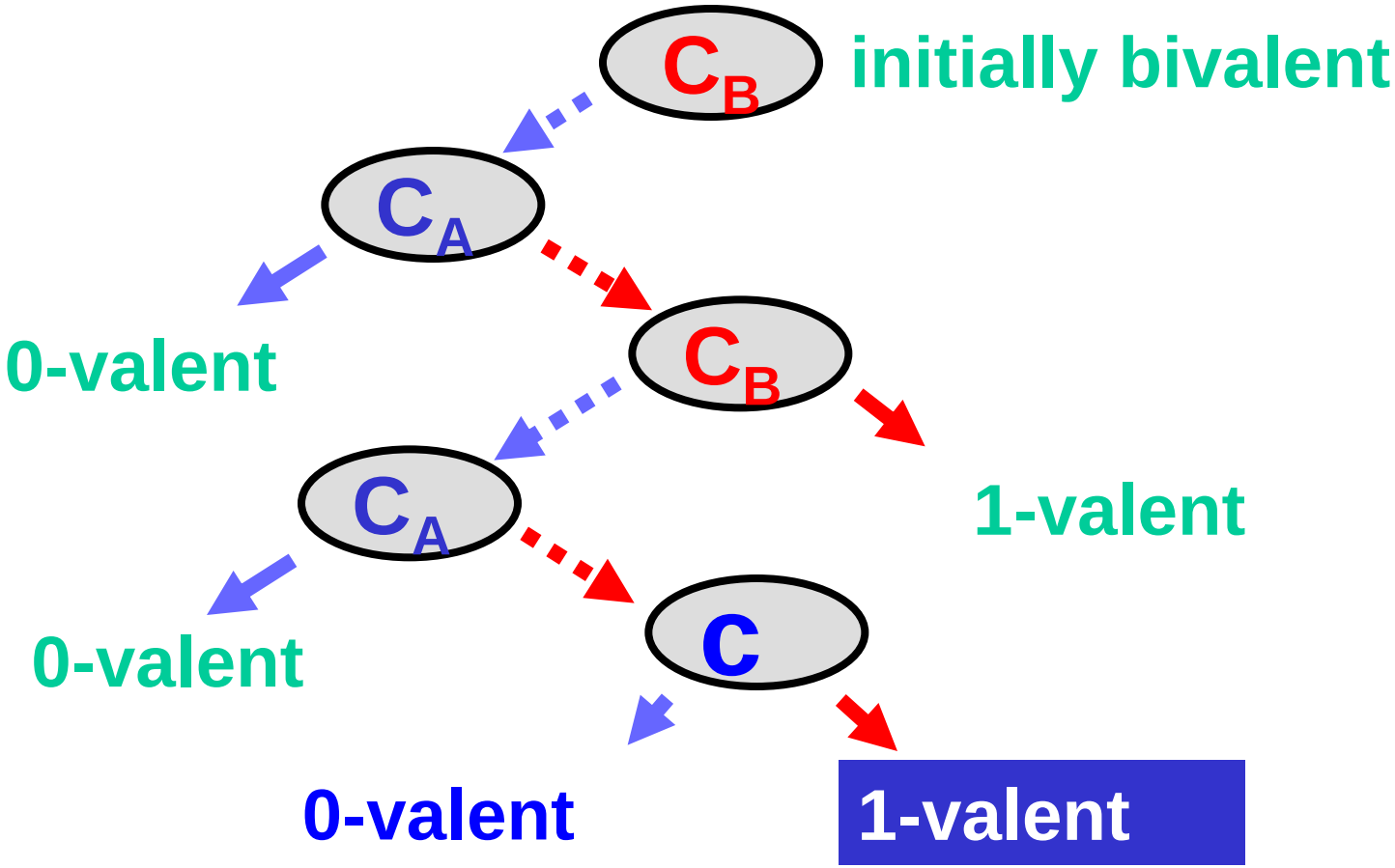
If A goes first, protocol decides 0

If B goes first, protocol decides 1

Critical States

Claim: starting from a bivalent initial state the protocol will reach a critical state

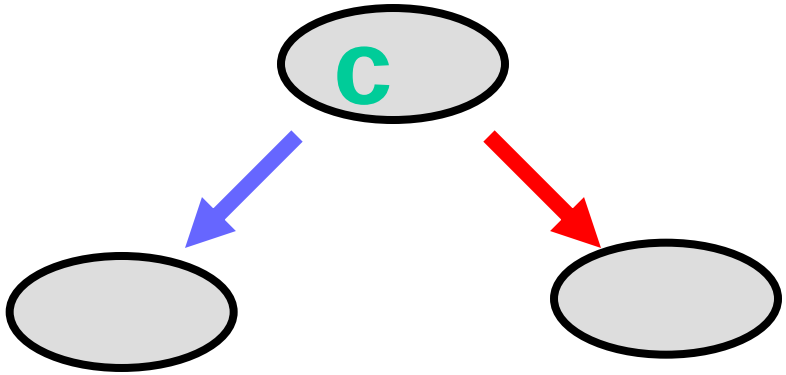
Reaching Critical State



Model Dependency

- So far, memory-independent!
- True for
  - Registers
  - Message-passing
  - Carrier pigeons
  - Any kind of asynchronous computation

From a Critical State



0-valent **1-valent**

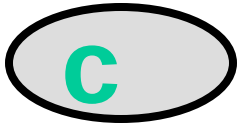
If A goes first, protocol decides 0

If B goes first, protocol decides 1

Possible Interactions

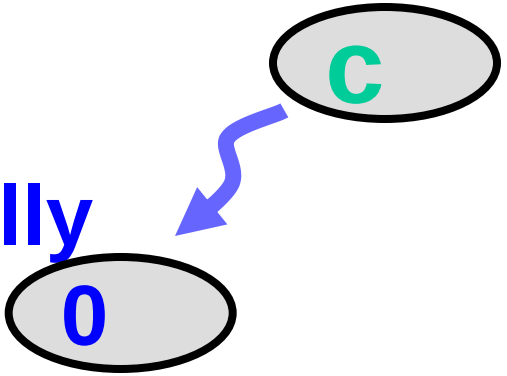
	`x.read()`	`y.read()`	`x.write()`	`y.write()`
`x.read()`	?	?	?	?
`y.read()`	?	?	?	?
`x.write()`	?	?	?	?
`y.write()`	?	?	?	?

B reads some register

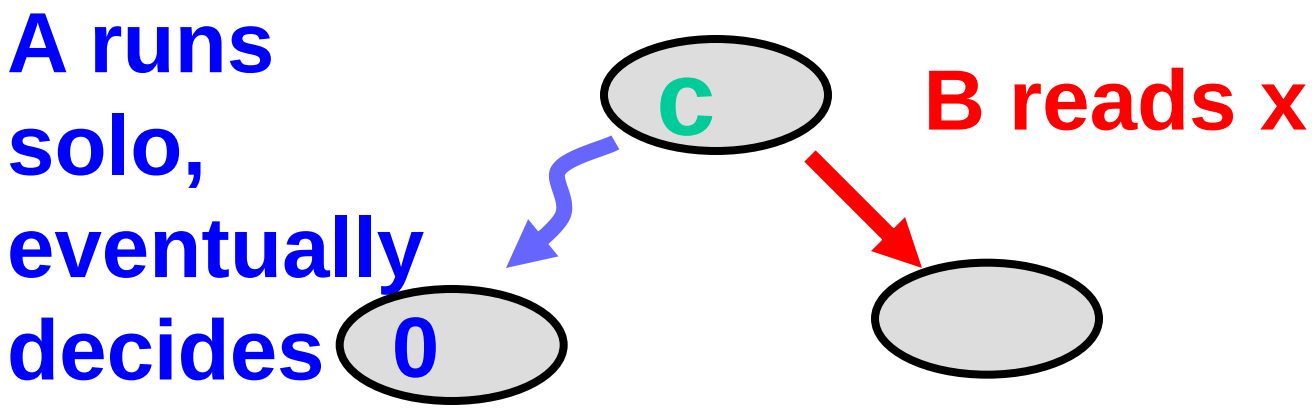


B reads some register

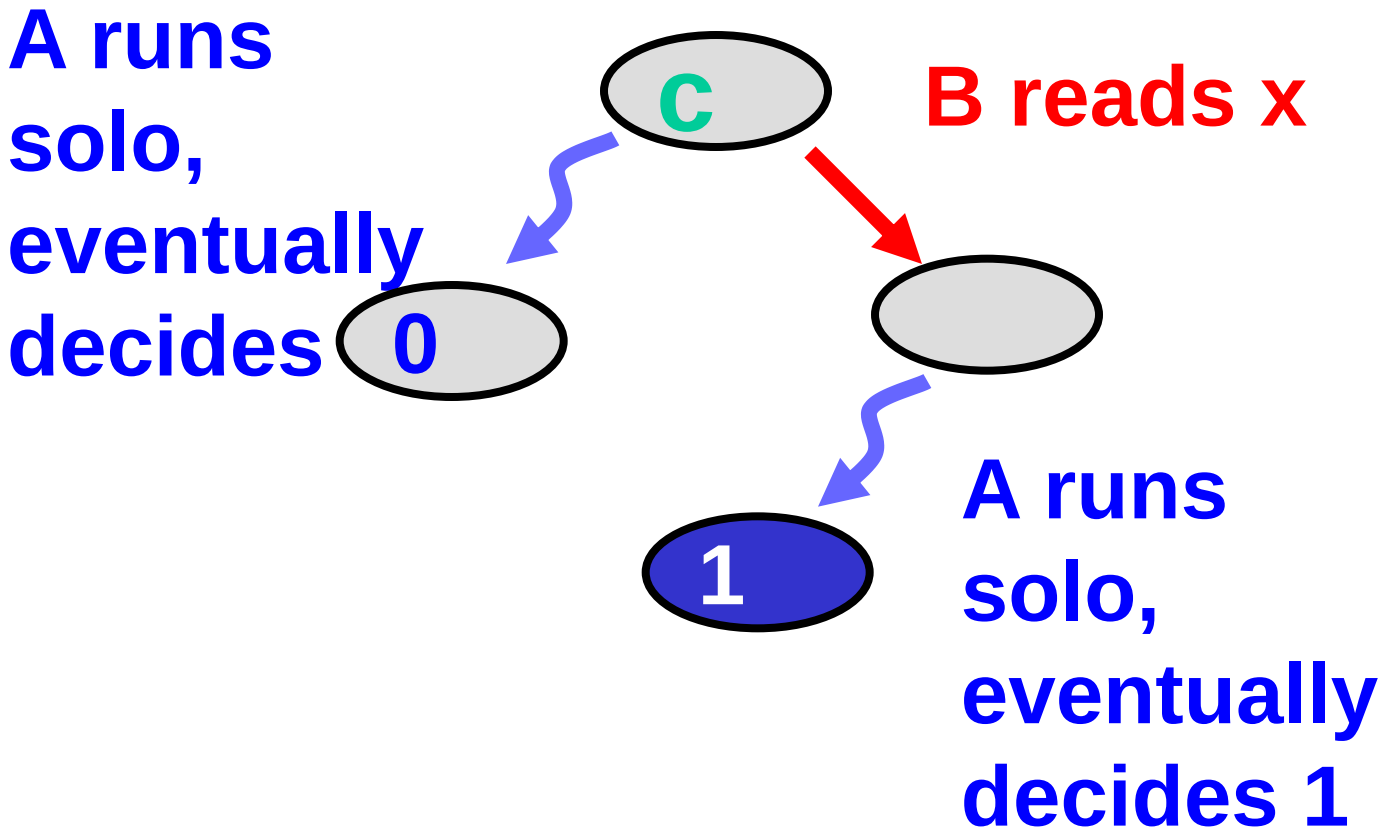
A runs solo, eventually decides



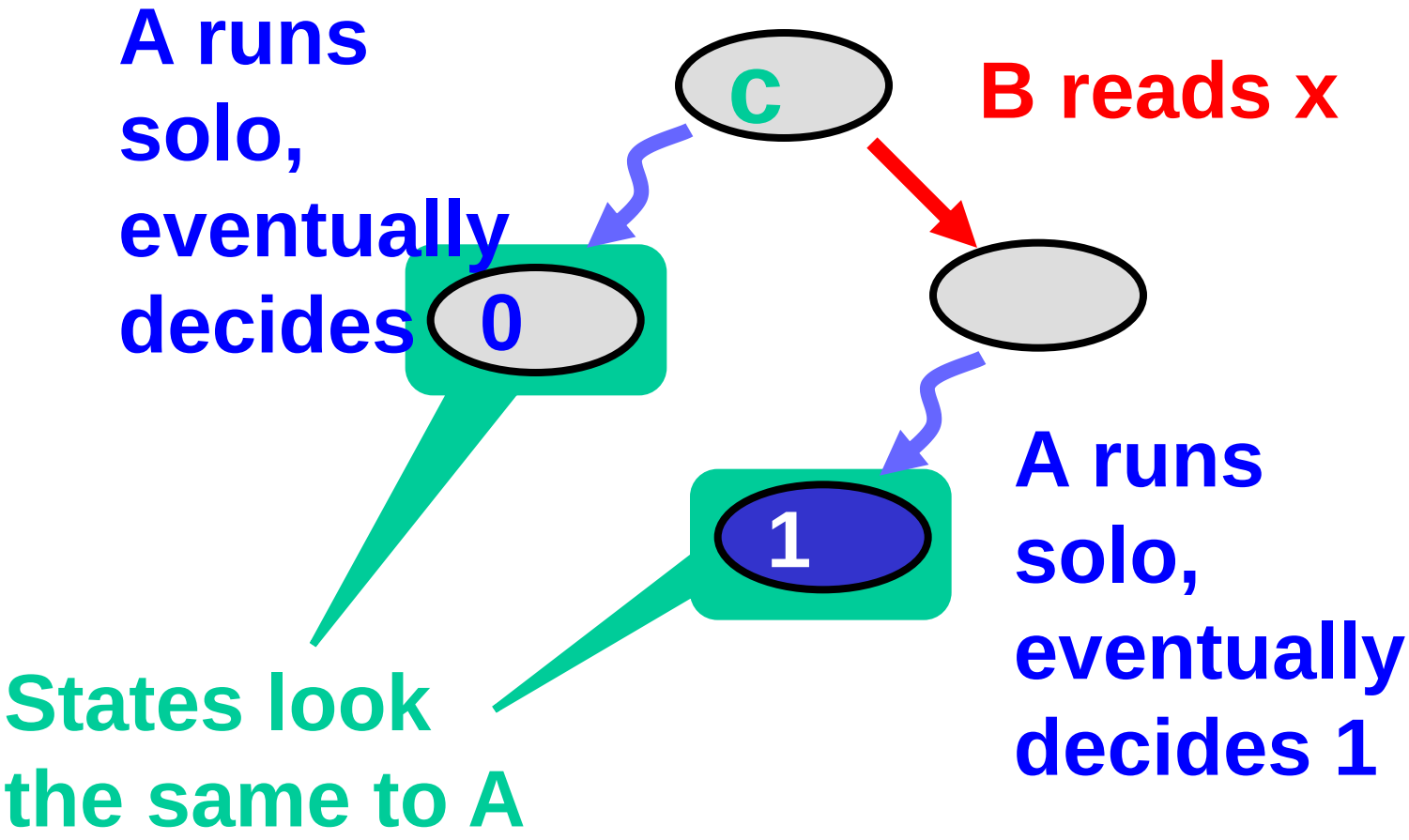
B reads some register



B reads some register



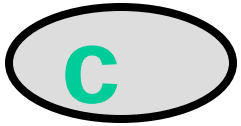
B reads some register



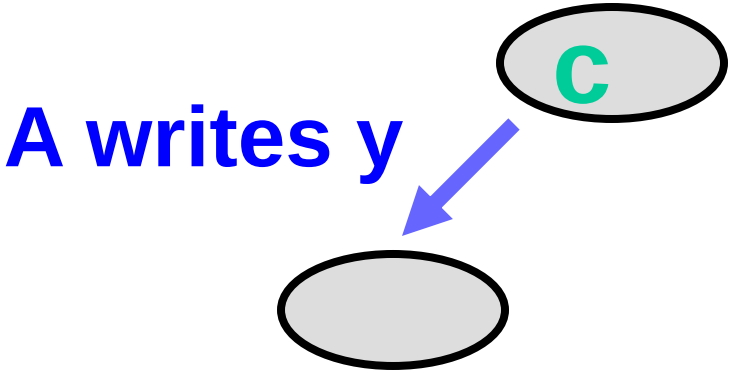
Possible Interactions

	`x.read()`	`y.read()`	`x.write()`	`y.write()`
`x.read()`	no	no	no	no
`y.read()`	no	no	no	no
`x.write()`	no	no	?	?
`y.write()`	no	no	?	?

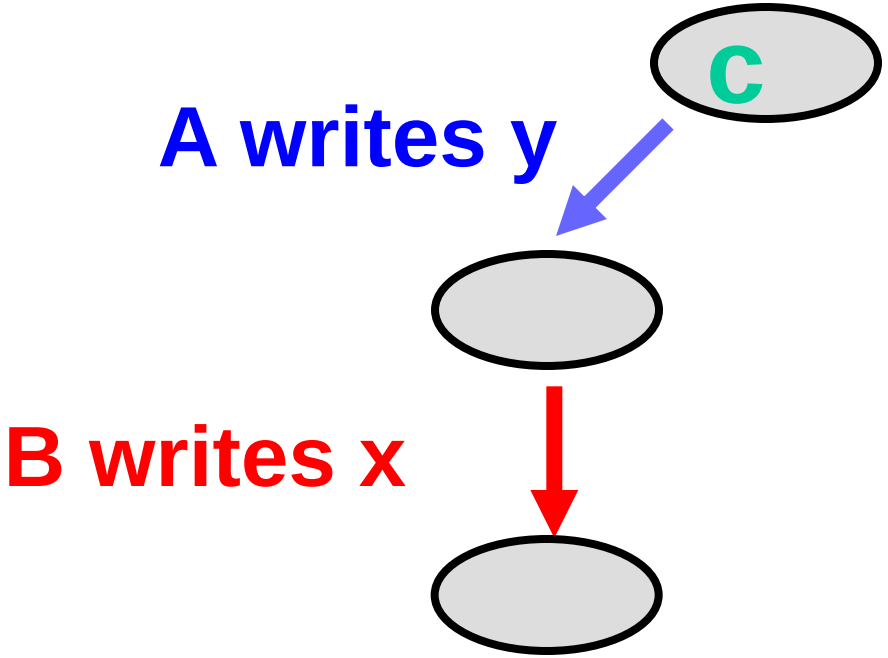
Writing Distinct Registers



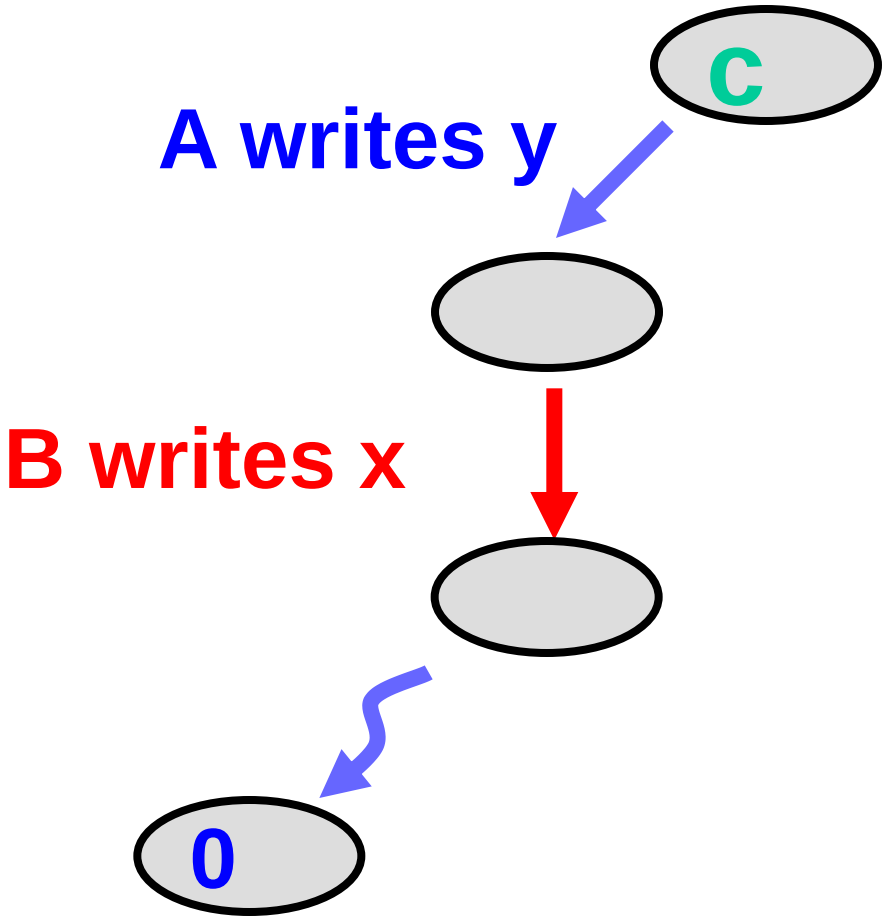
Writing Distinct Registers



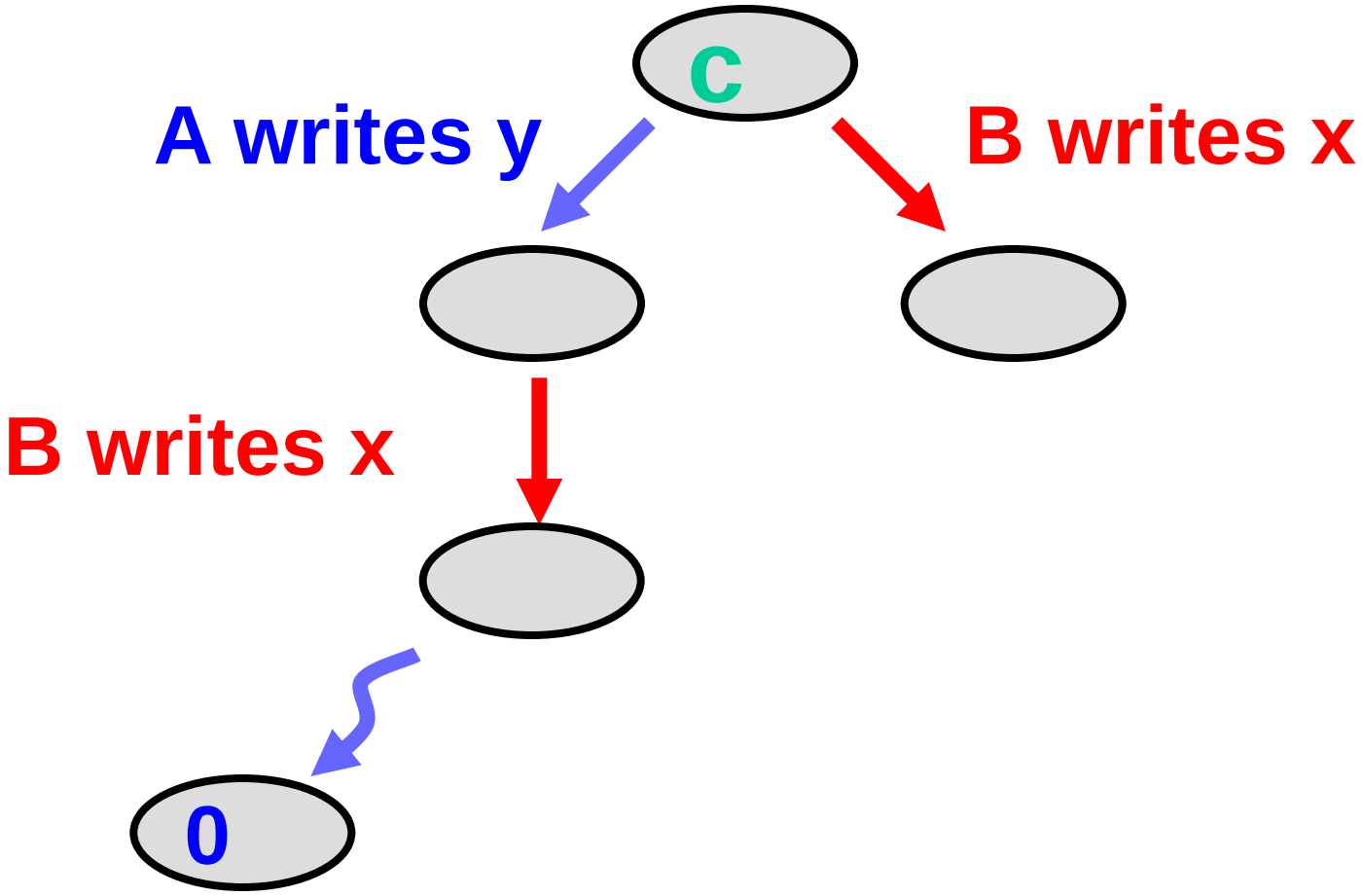
Writing Distinct Registers



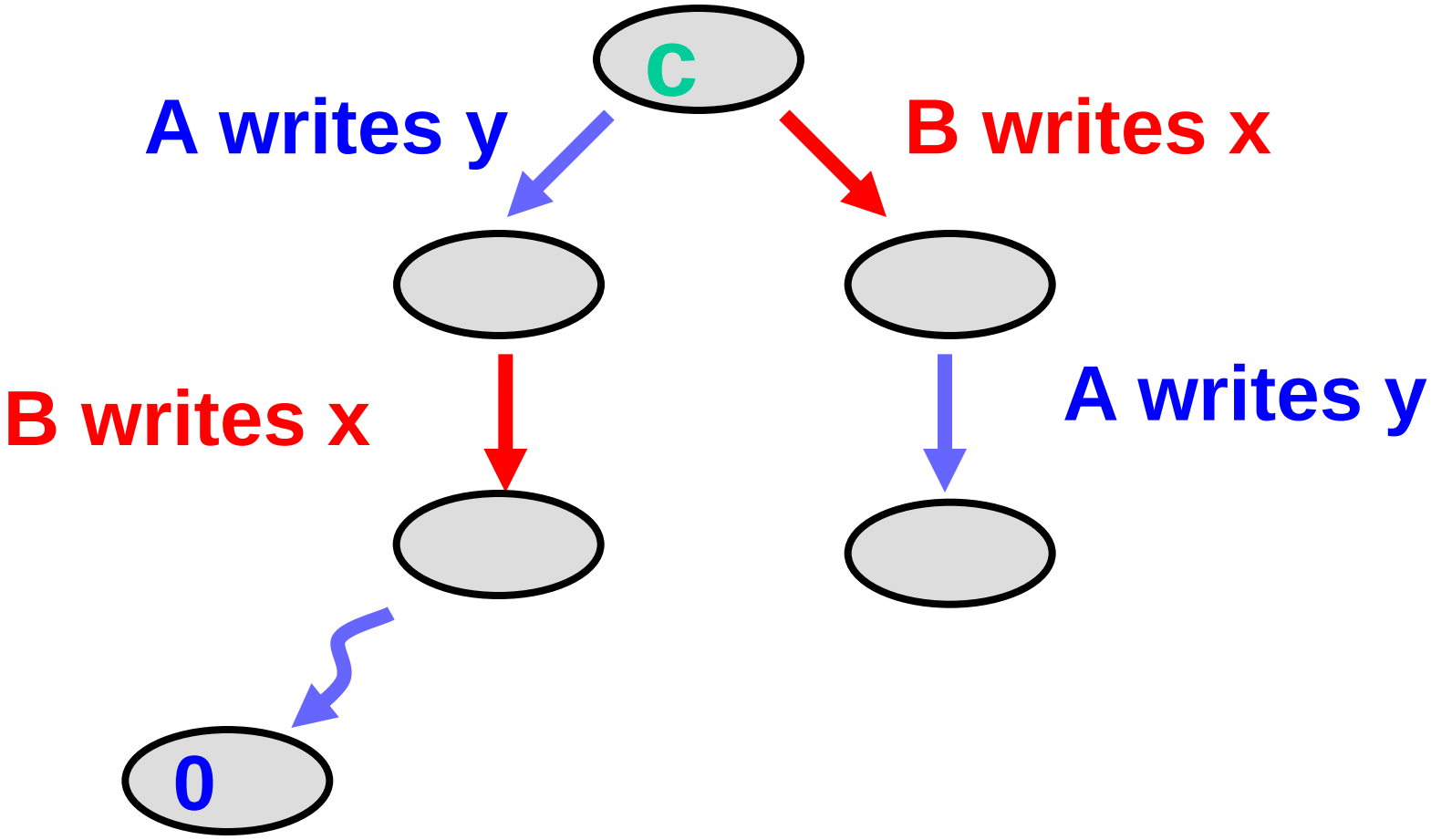
Writing Distinct Registers



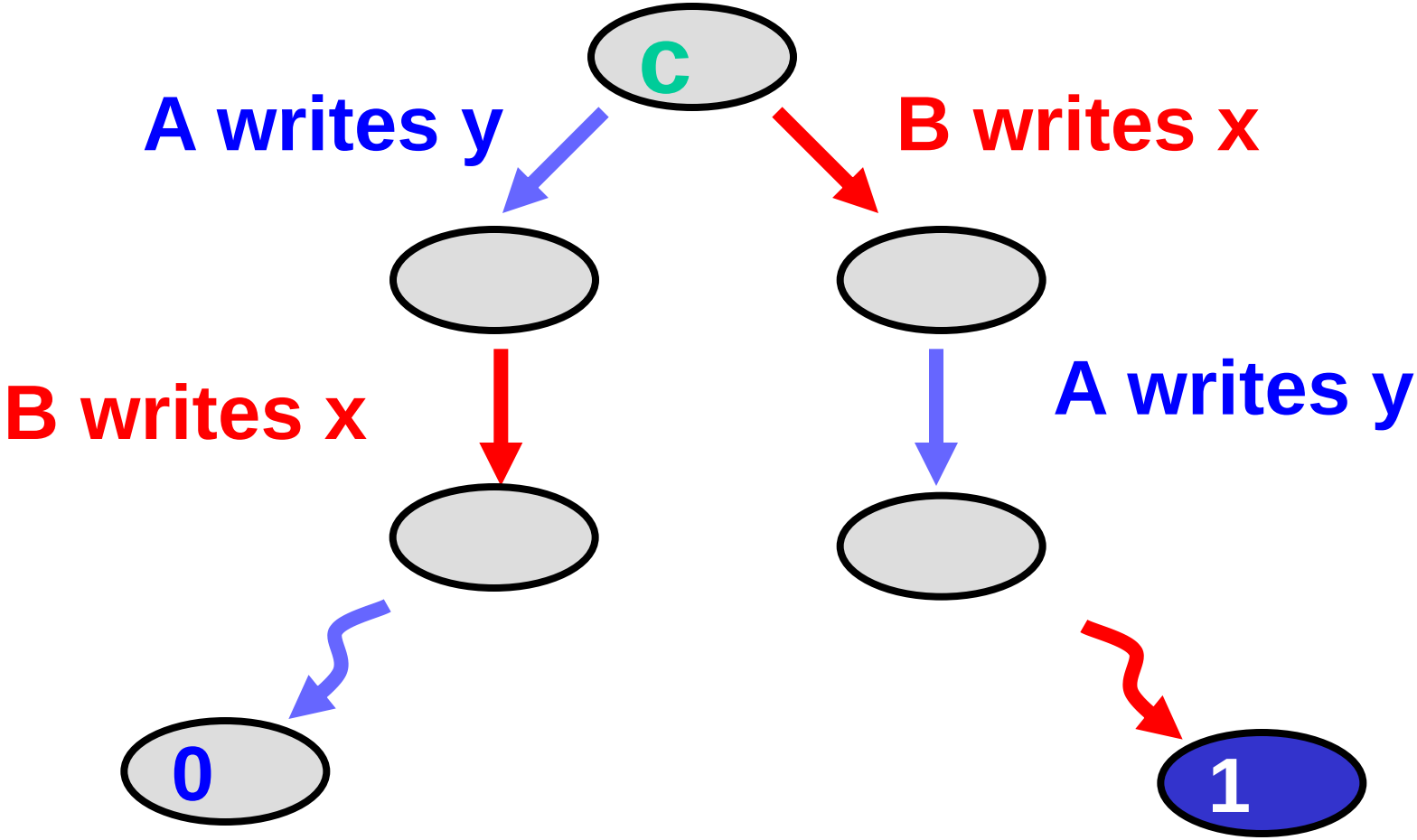
Writing Distinct Registers



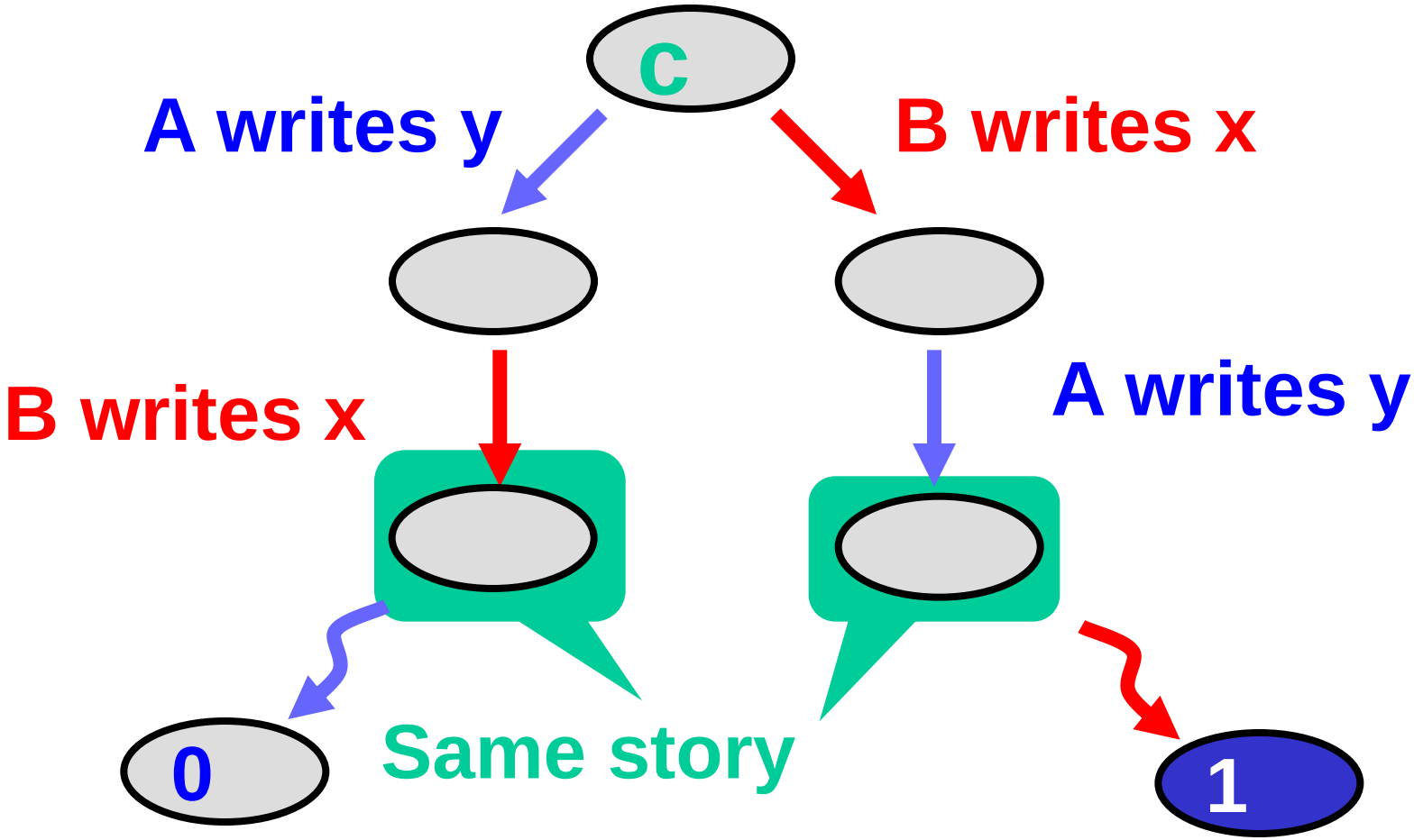
Writing Distinct Registers



Writing Distinct Registers



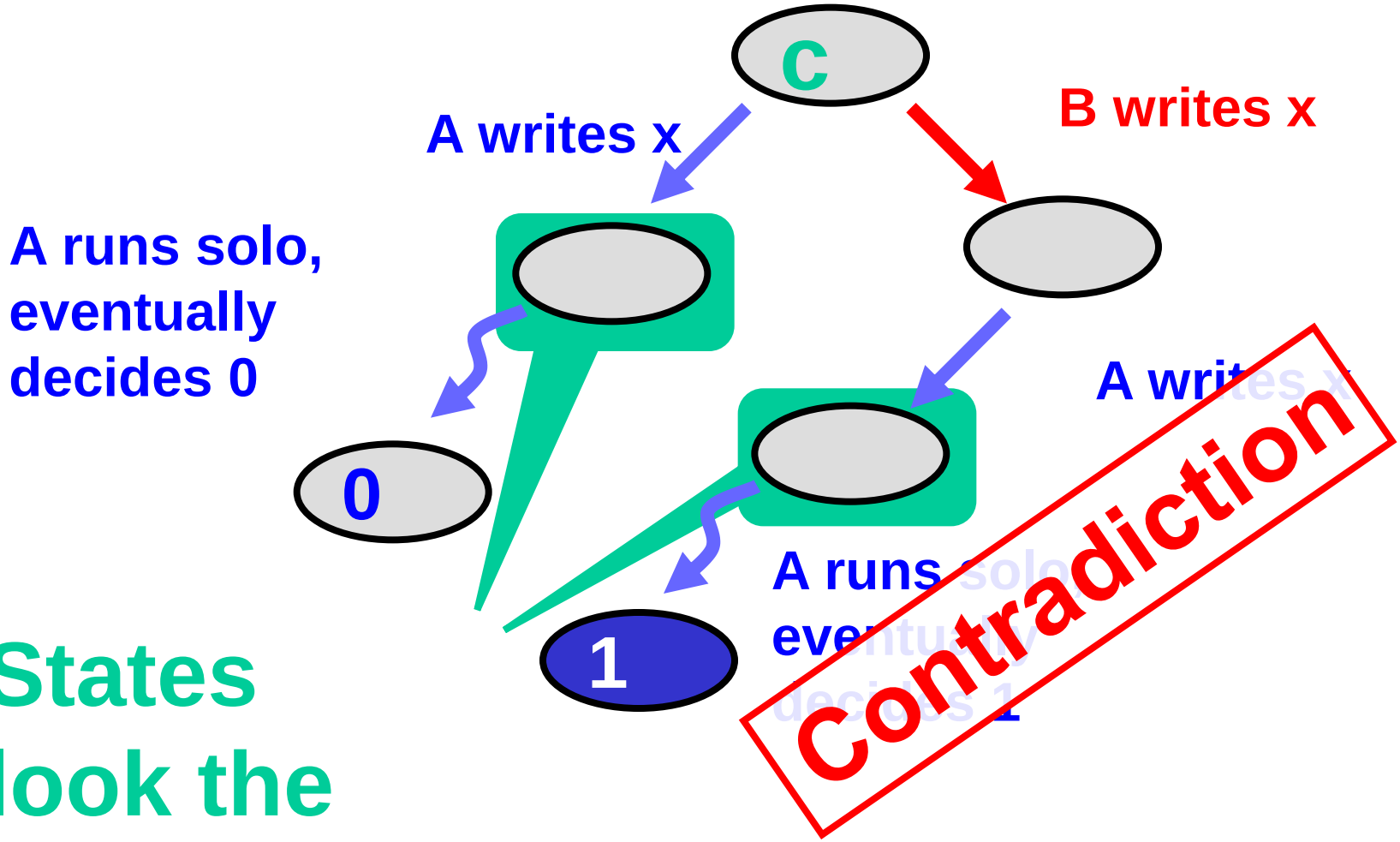
Writing Distinct Registers



Possible Interactions

	`x.read()`	`y.read()`	`x.write()`	`y.write()`
`x.read()`	no	no	no	no
`y.read()`	no	no	no	no
`x.write()`	no	no	?	no
`y.write()`	no	no	no	?

Writing Same Registers



States look the same to A

Recap: Atomic Registers Can't Do Consensus

- If protocol exists
  - It has a bivalent initial state
  - Leading to a critical state
- What's up with the critical state?
  - Case analysis for each pair of methods
  - As we showed, all lead to a contradiction

Consensus Numbers

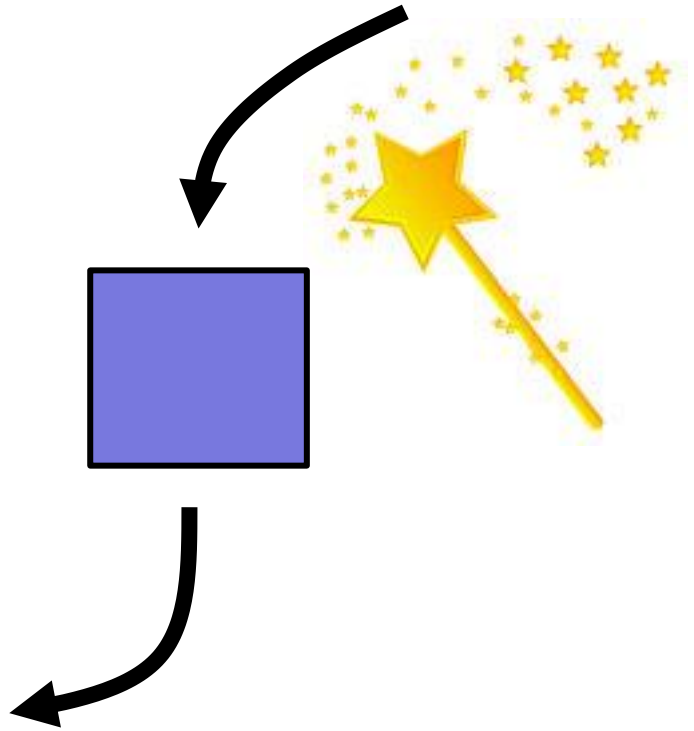
- An object X has consensus number *n*
  - If it can be used to solve *n*-thread consensus
    - Take any number of instances of X
    - together with atomic read/write registers
    - and implement *n*-thread consensus
  - But not (*n*+1)-thread consensus

Consensus Numbers

- Read/write registers have consensus number 1

Intel x86 Instruction Set

Read-Modify-Write Instructions

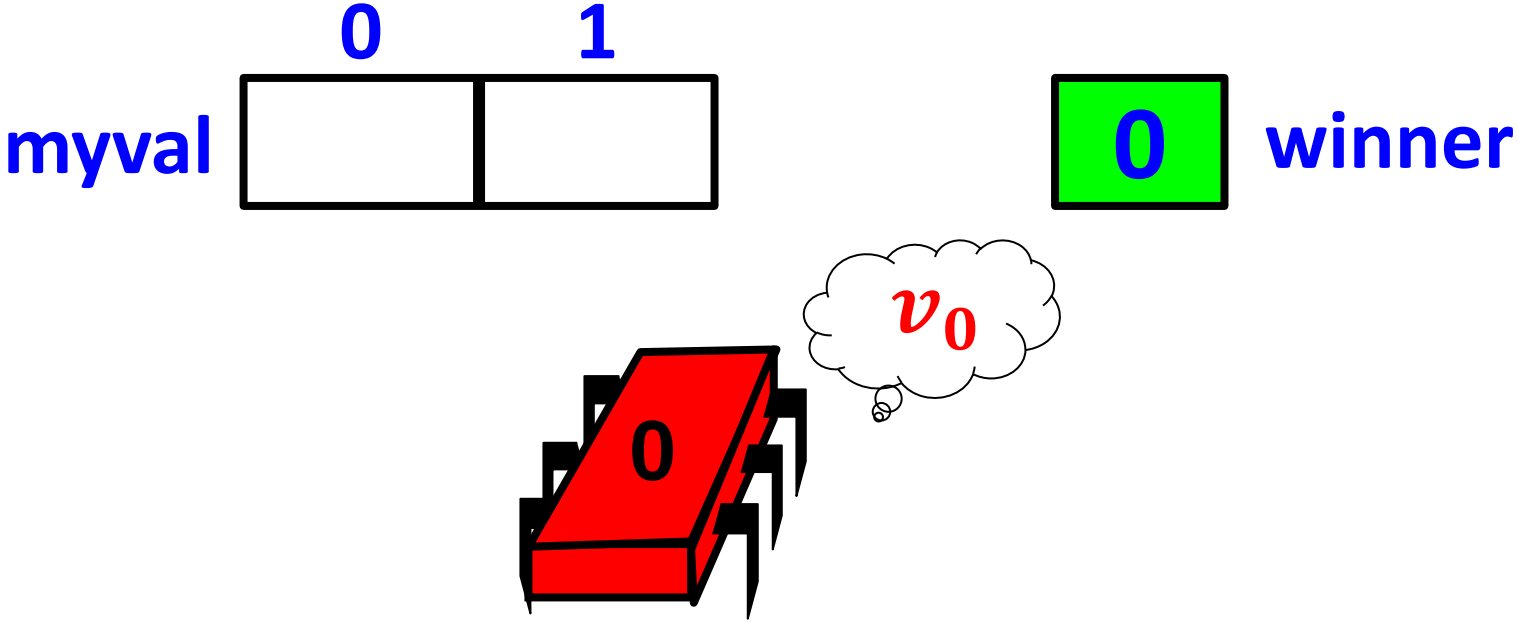


Read-Modify-Write Instructions

- Test-And-Set: t&s(x)
  - If x = 1 return 0
  - If x = 0, set it to 1 and return 1
- Compare-And-Swap: cas(x, old, new)
  - If x = old, set it to new, return “success”
  - Else return value of x

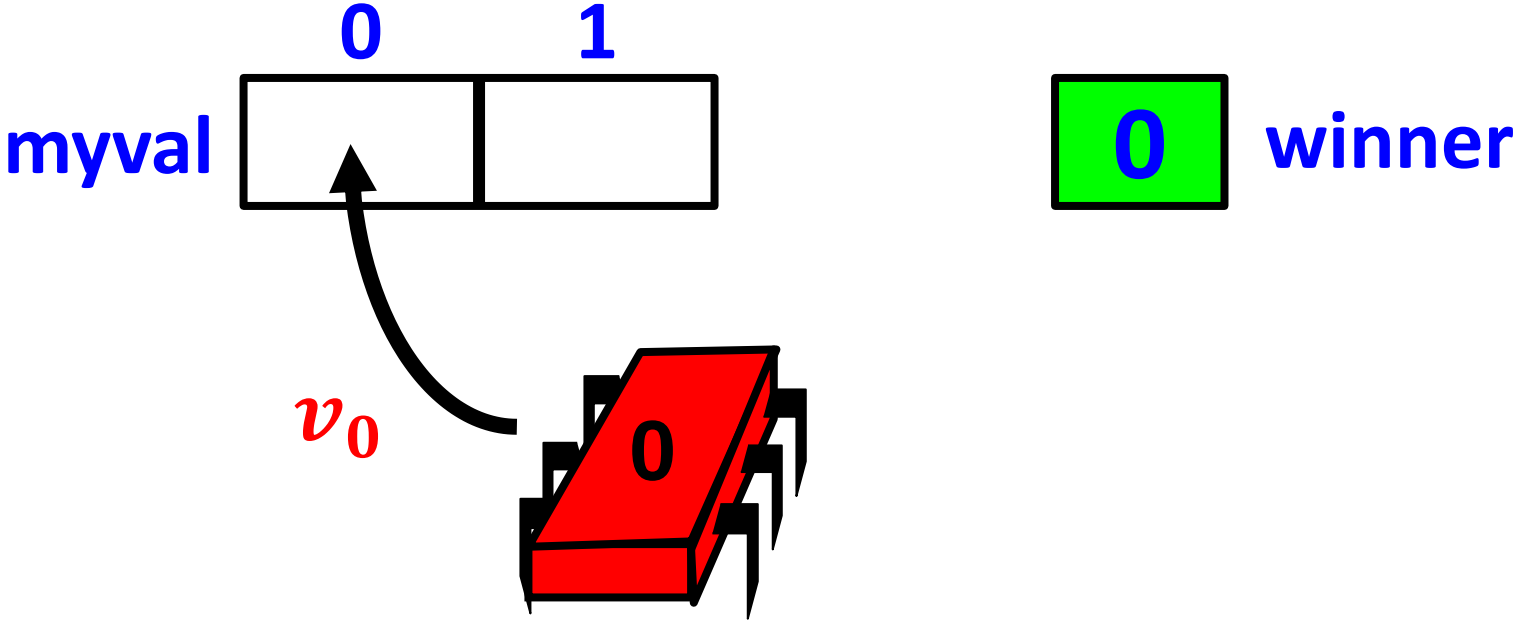
Consensus from Test&Set

- Two processes:



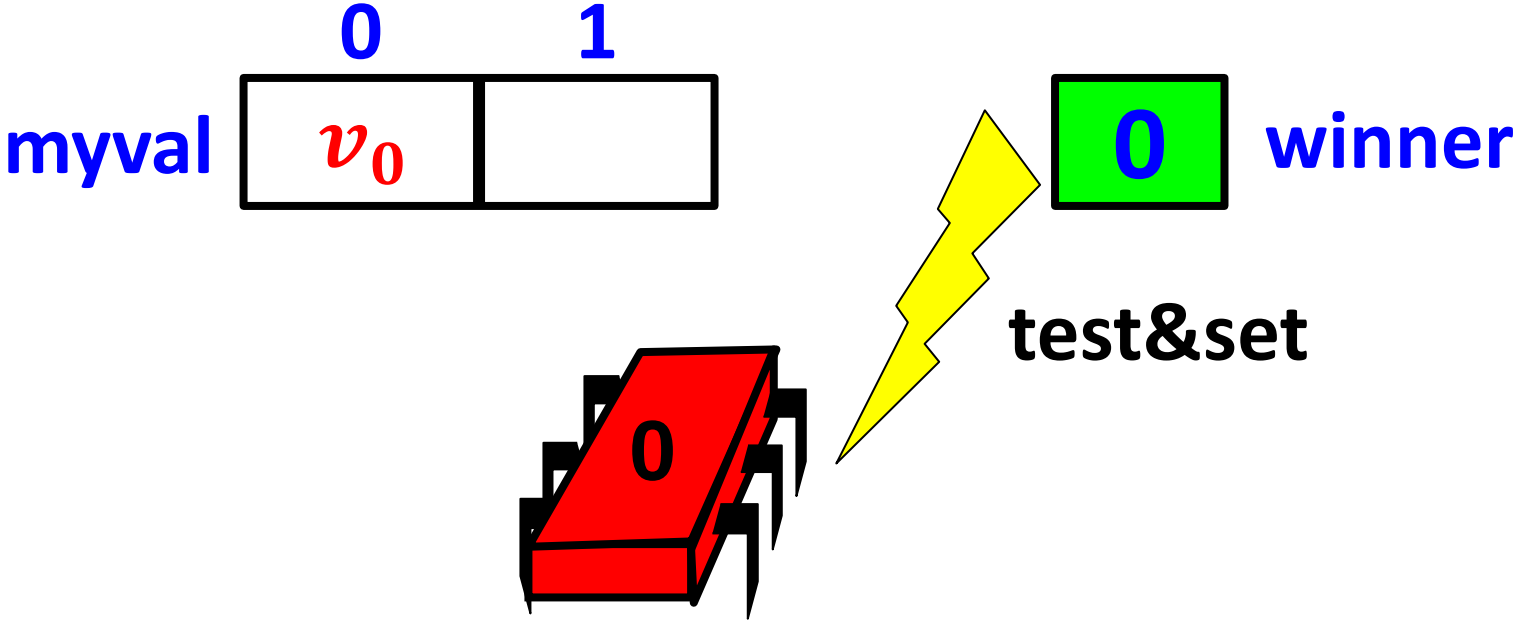
Consensus from Test&Set

- Two processes:



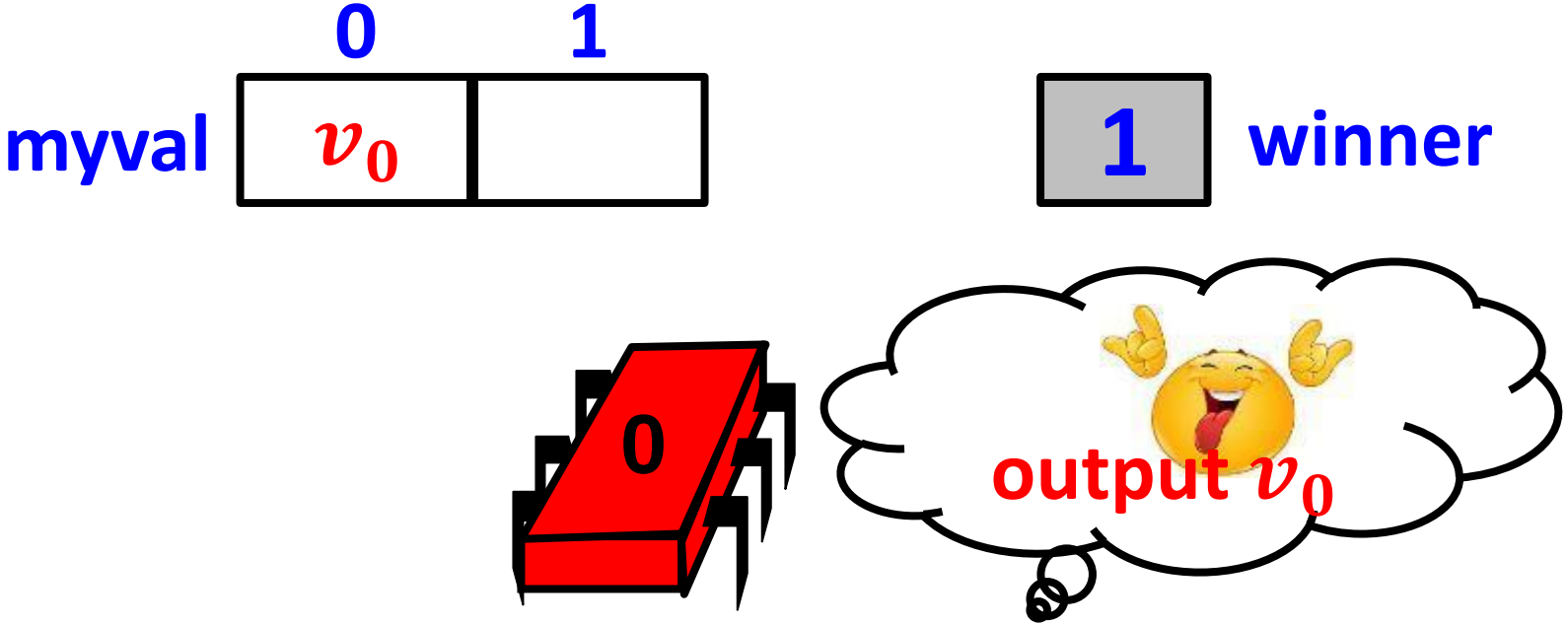
Consensus from Test&Set

- Two processes:



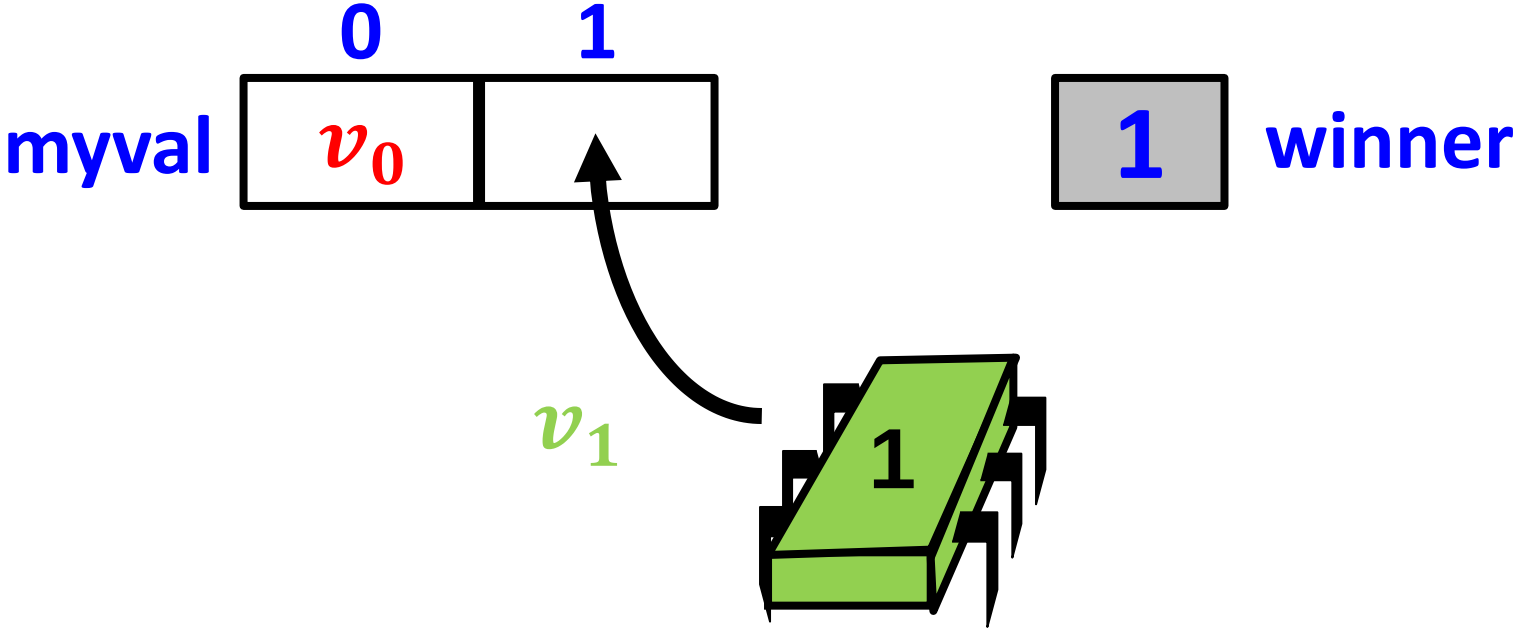
Consensus from Test&Set

- Two processes:



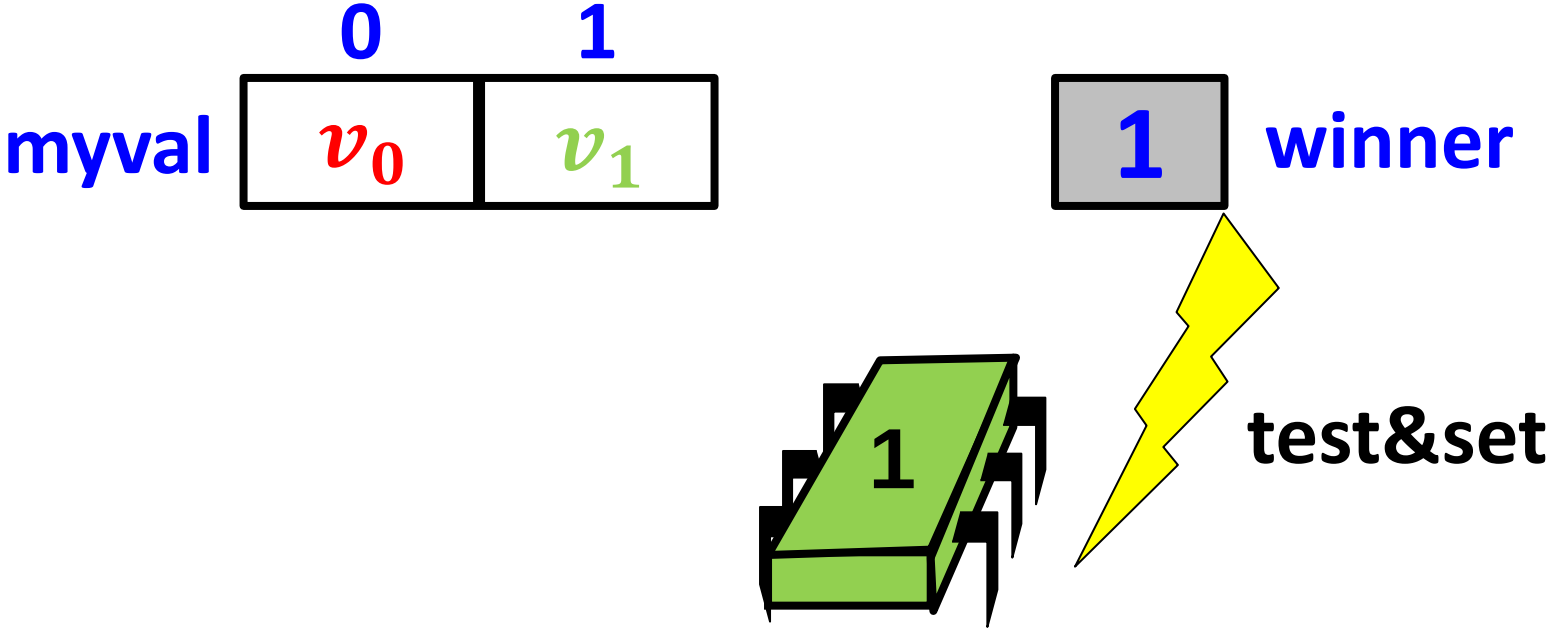
Consensus from Test&Set

- Two processes:



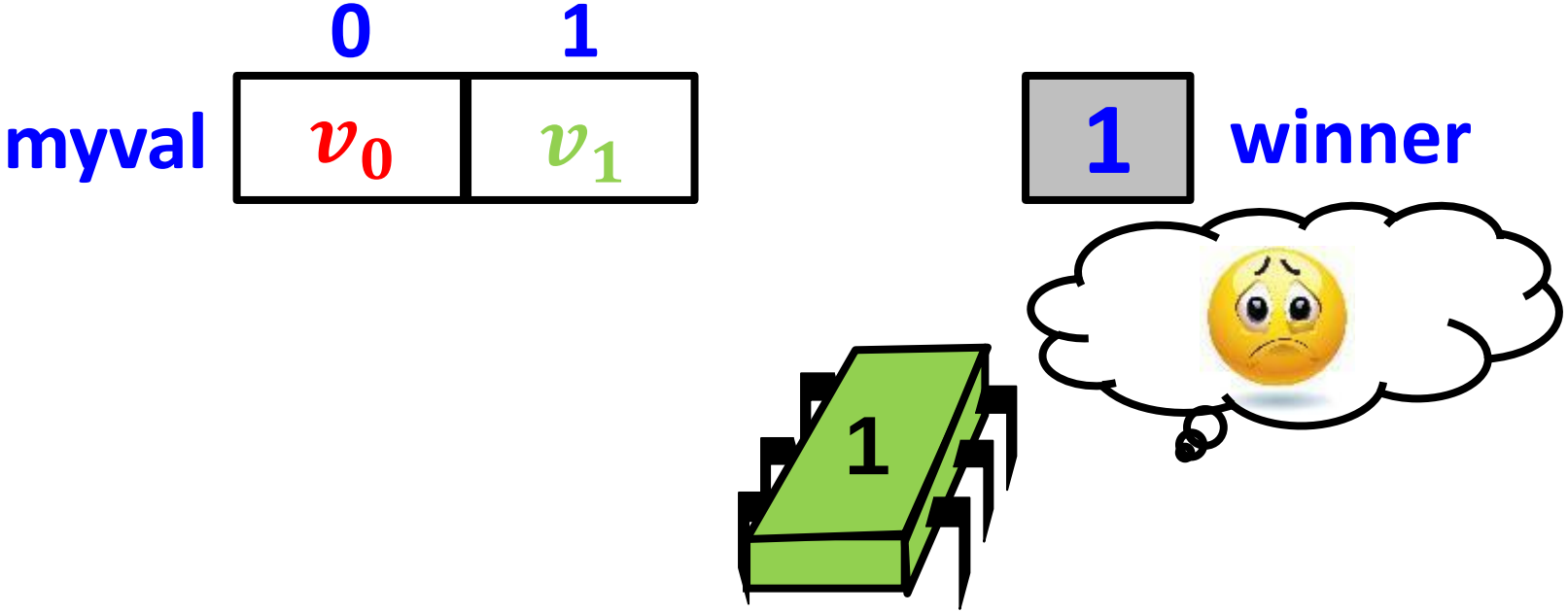
Consensus from Test&Set

- Two processes:



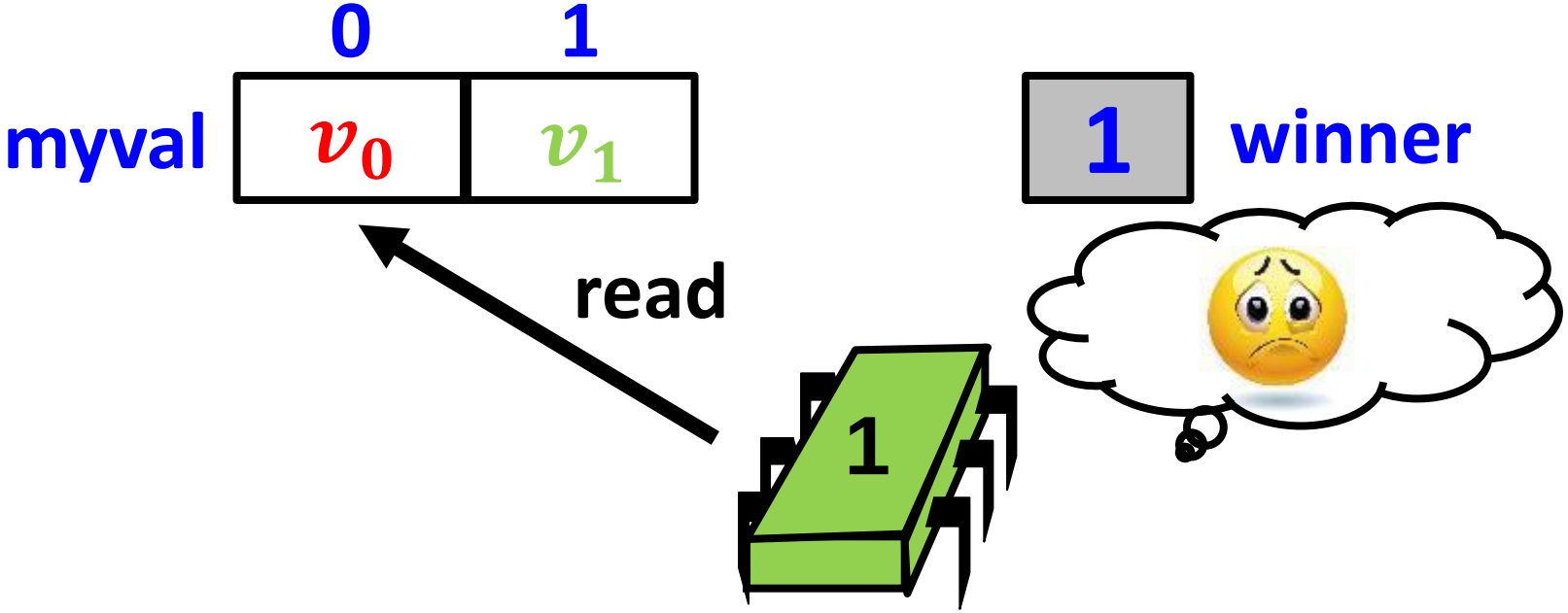
Consensus from Test&Set

- Two processes:



Consensus from Test&Set

- Two processes:



Consensus Number of T&S

- We showed: ≥ 2
- Impossibility for 3:
  - Almost the same as before…

Consensus from CAS

- array myval[n]
- winner, initially ⊥
- process i:
  - write v_i to myval[i]
  - res ← CAS(winner, ⊥, i)
  - If(res = “success”) output v_i
  - else read myval[res]

Transactional Memory

```
atomic {
        …
}
```

MINISTRY OF INNOVATION —

Transactional memory going mainstream with Intel Haswell

Transactional memory is a promising technique for making the development of ...

PETER BRIGHT - 2/9/2012, 4:10 AM