YSC3248: Parallel, Concurrent and Distributed Programming

Wait-Free Implementations and Consensus

Last Lecture

- Defined concurrent objects using linearizability and sequential consistency
- Fact: implemented linearizable objects (Two thread FIFO Queue) in read-write memory without mutual exclusion
- Fact: hardware does not provide linearizable read-write memory

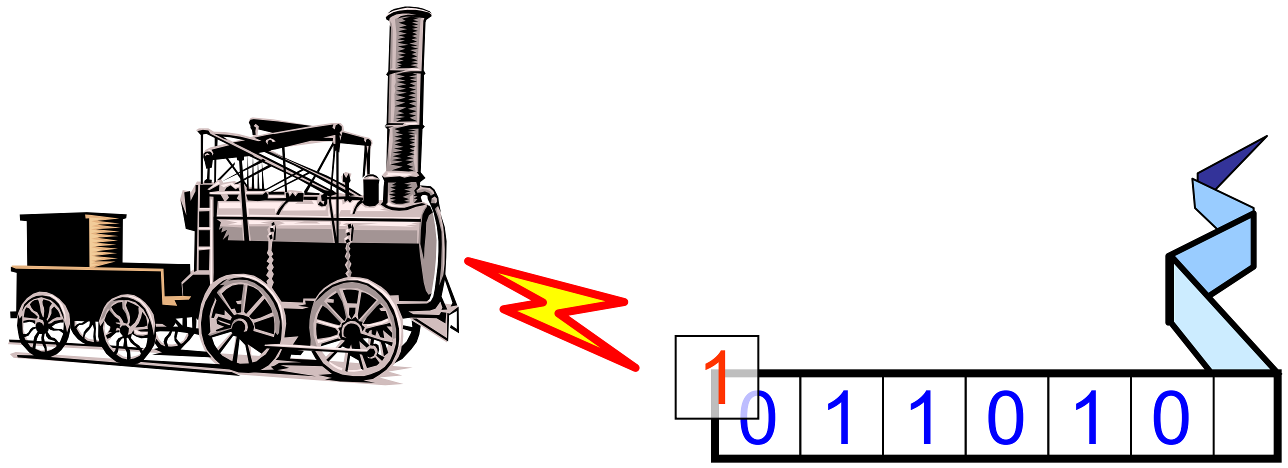
Fundamentals

- What is the weakest form of communication that supports mutual exclusion?
- What is the weakest shared object that allows shared-memory computation?

Alan Turing

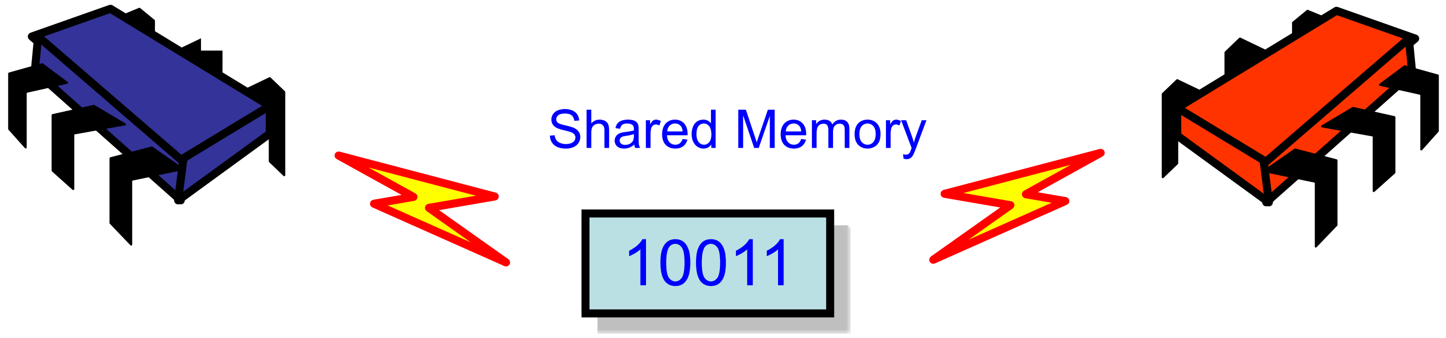
- Showed what is and is not computable on a sequential machine.
- Still best model there is.

Turing Computability



- Mathematical model of computation
- What is (and is not) computable
- Efficiency (mostly) irrelevant

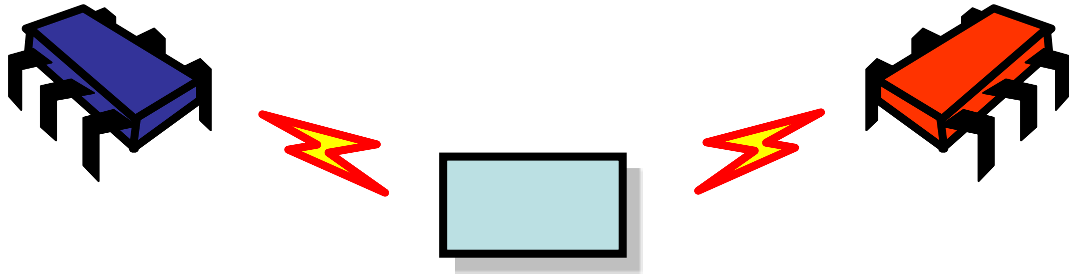
Shared-Memory Computability?



- Mathematical model of concurrent computation
- What is (and is not) concurrently computable
- Efficiency (mostly) irrelevant

Foundations of Shared Memory

To understand modern multiprocessors we need to ask some basic questions …



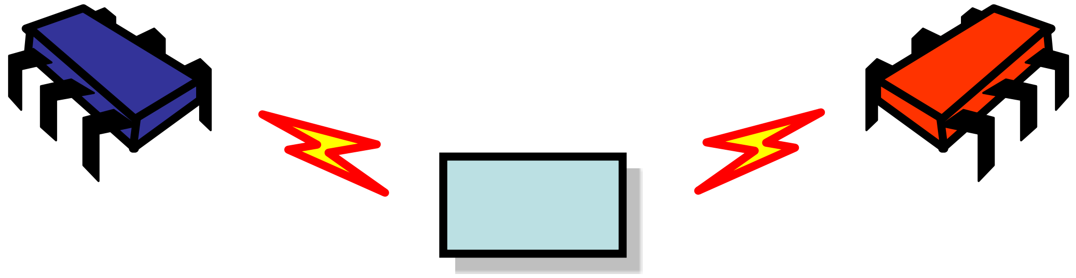
Foundations of Shared Memory

To understand modern

r

basic questions …

What is the weakest useful form of shared memory?



Foundations of Shared Memory

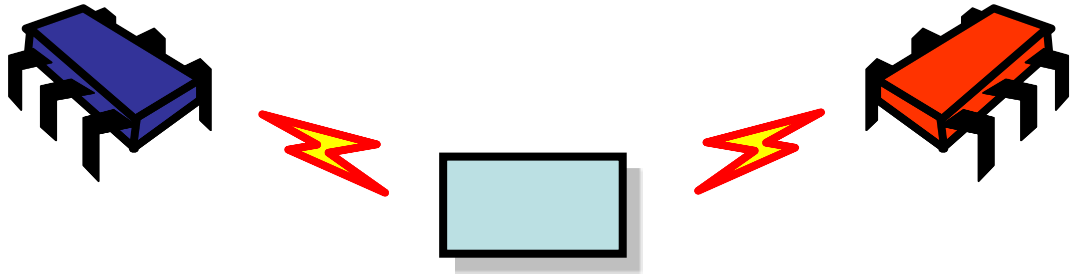
To understand modern

What is the weakest useful form of

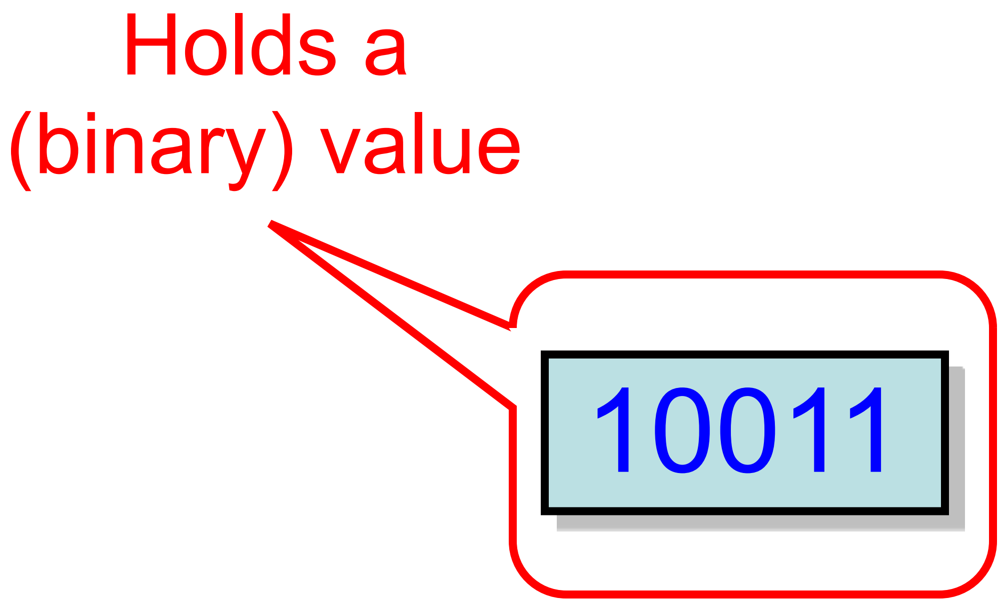
What can it do?

shared memory?

basic questions …

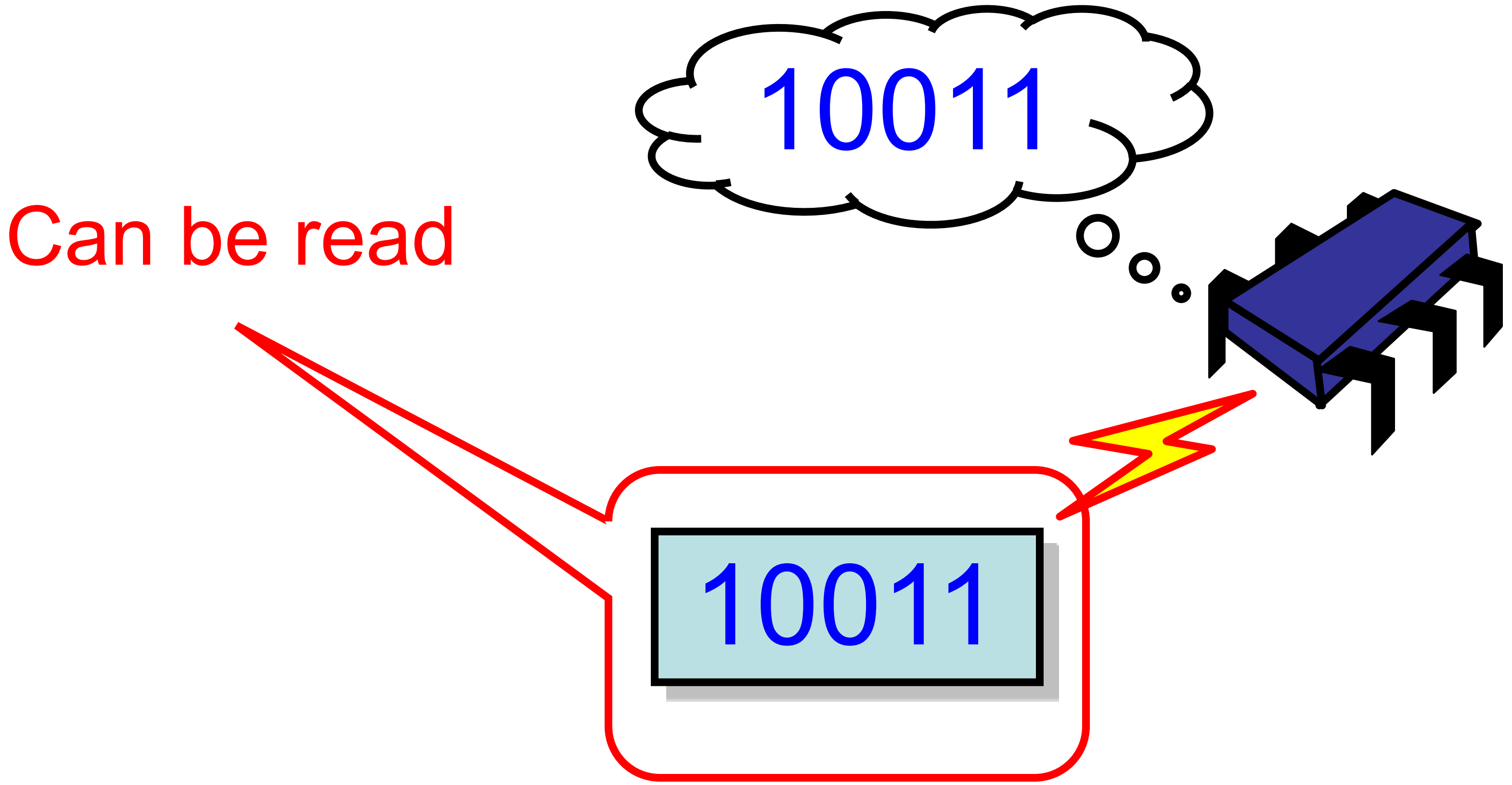


Register*

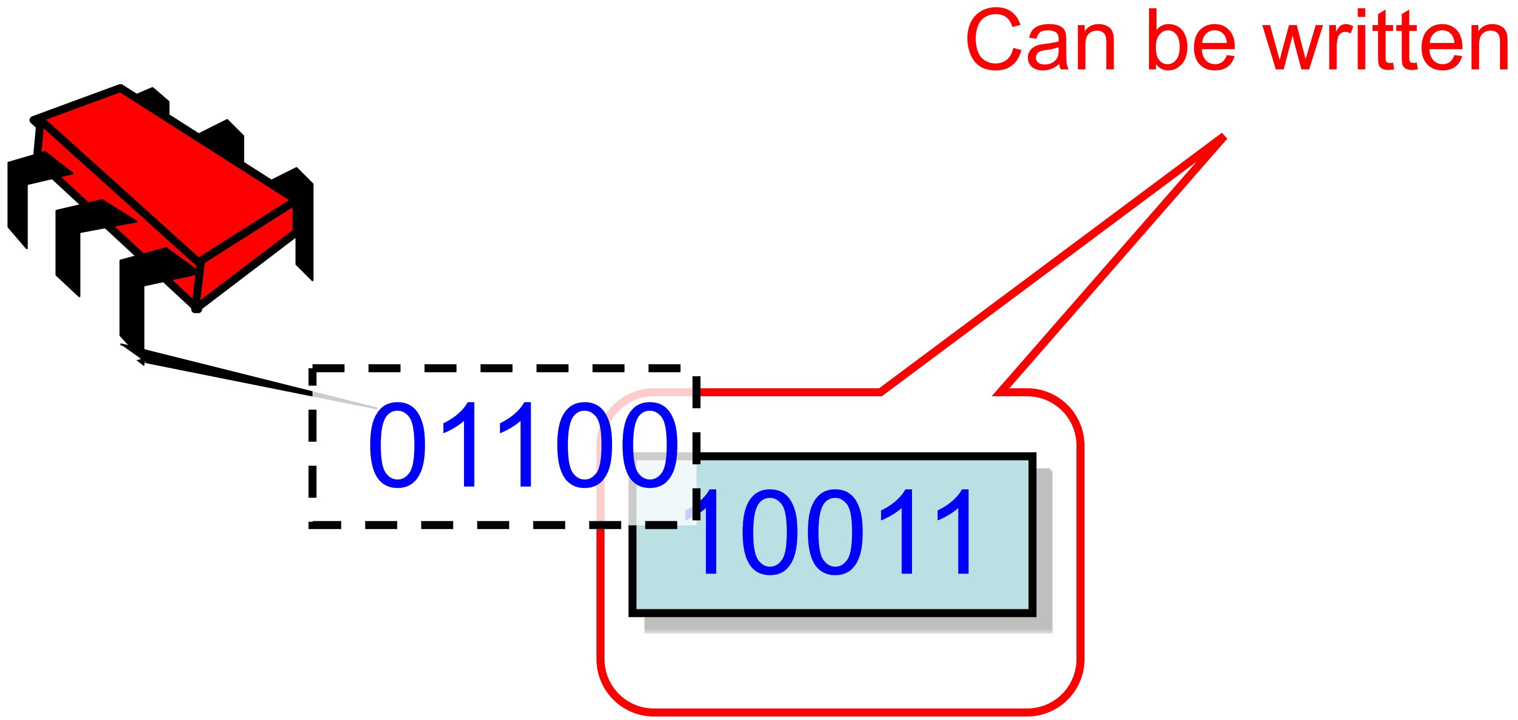


* **A memory location: name is historical**

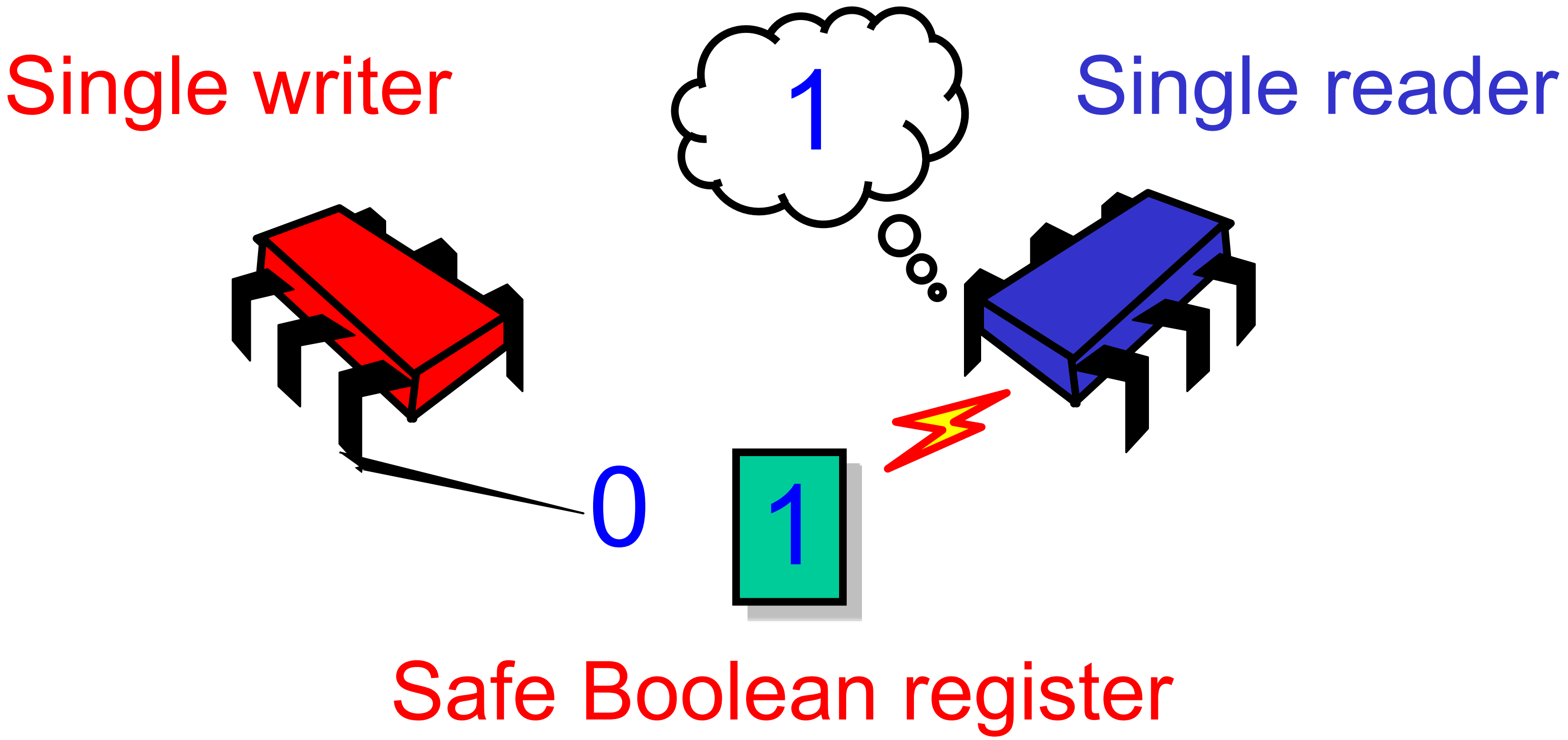
Register



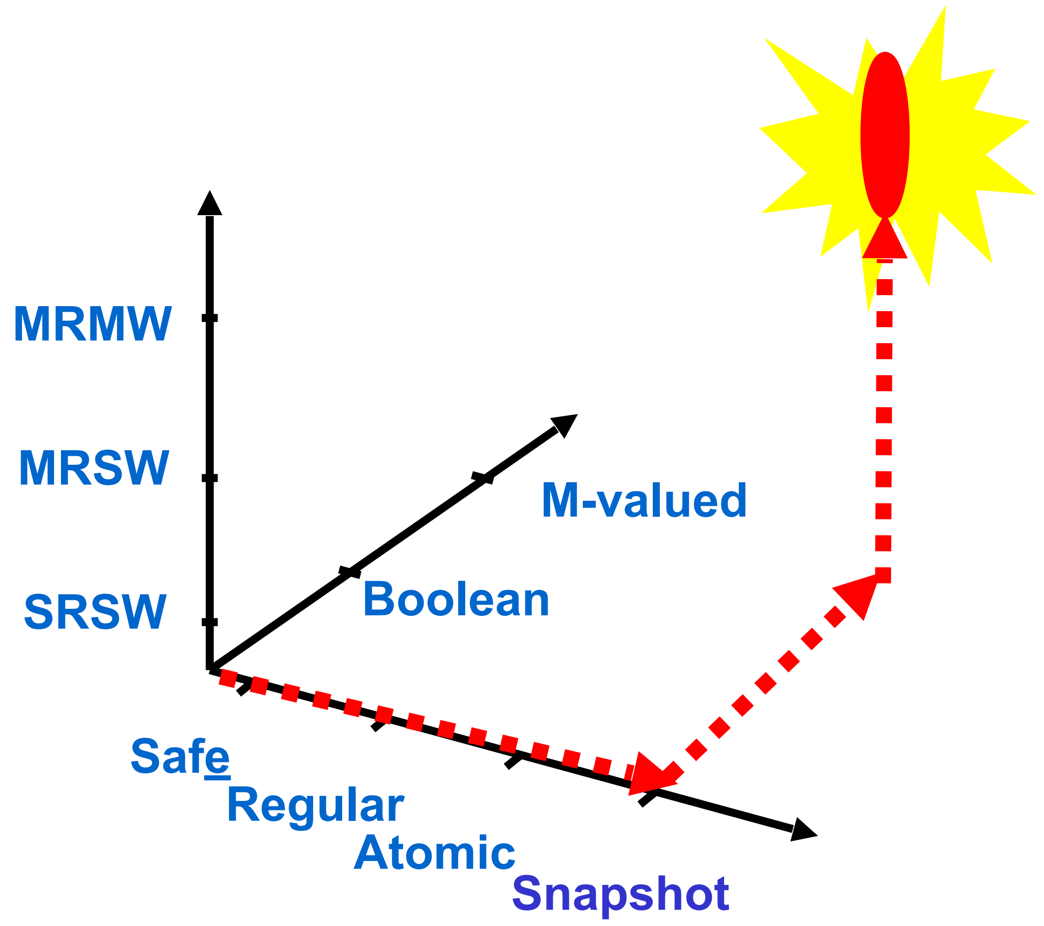
Register

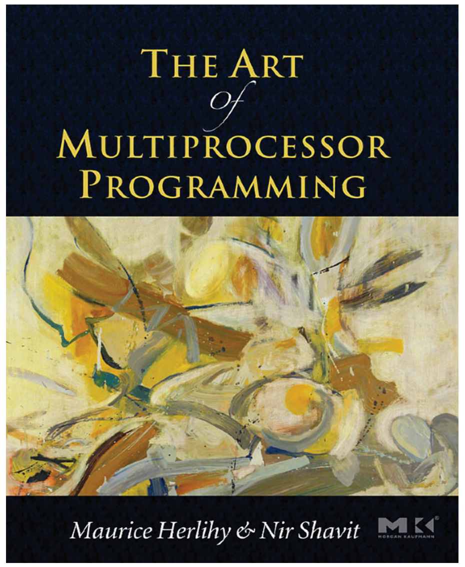


From Weakest Register



All the way to a Wait-free Implementation of Atomic Snapshots




The Art of Multiprocessor Programming
Maurice Herlihy & Nir Shavit

Chapter 4

Wait-Free Implementation

- Every method call completes in finite number of steps
- Implies no mutual exclusion

Rationale for wait-freedom

- We wanted atomic registers to implement mutual exclusion

Rationale for wait-freedom

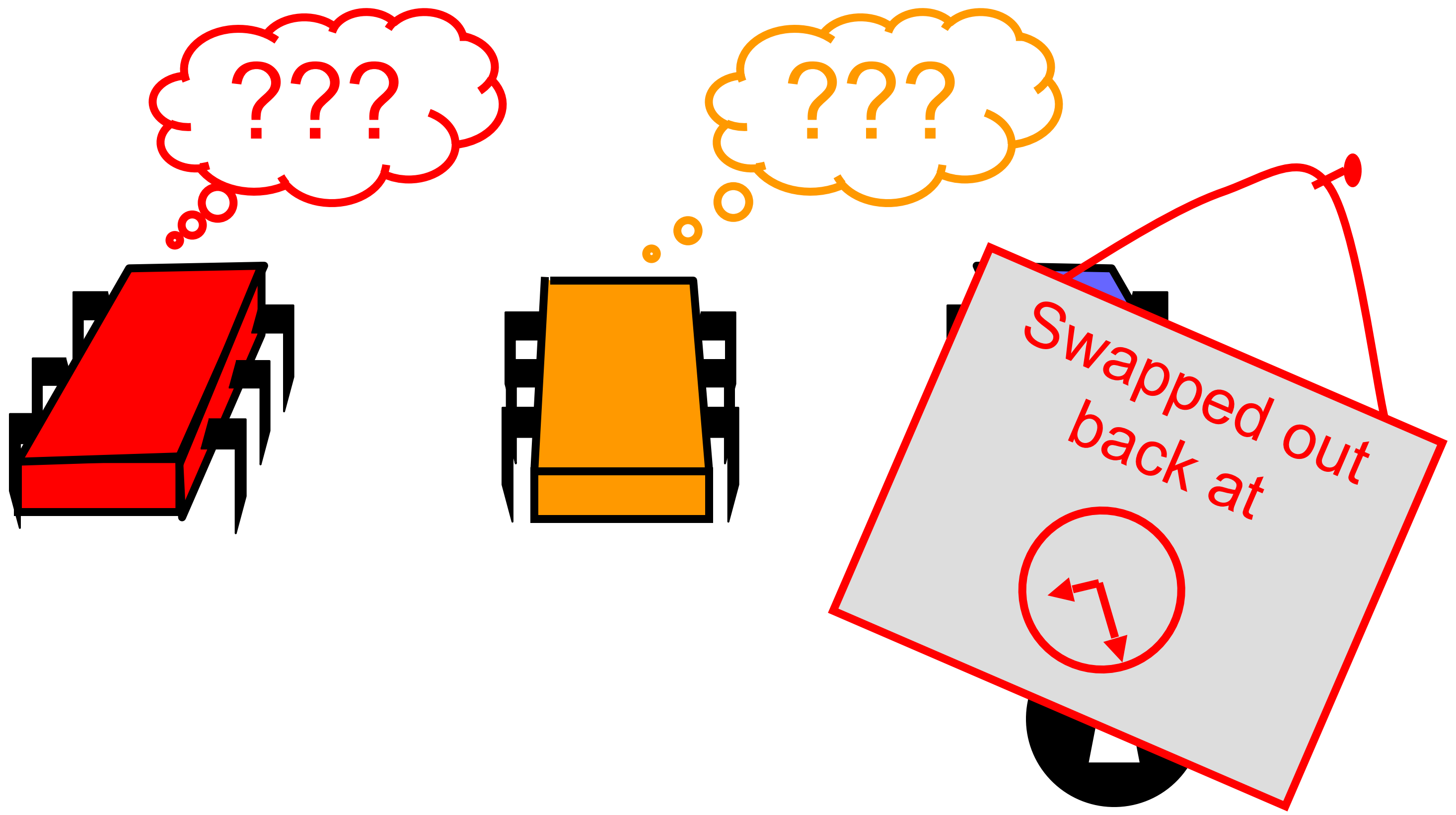
- We wanted atomic registers to implement mutual exclusion
- So we couldn’t use mutual exclusion to implement atomic registers

Rationale for wait-freedom

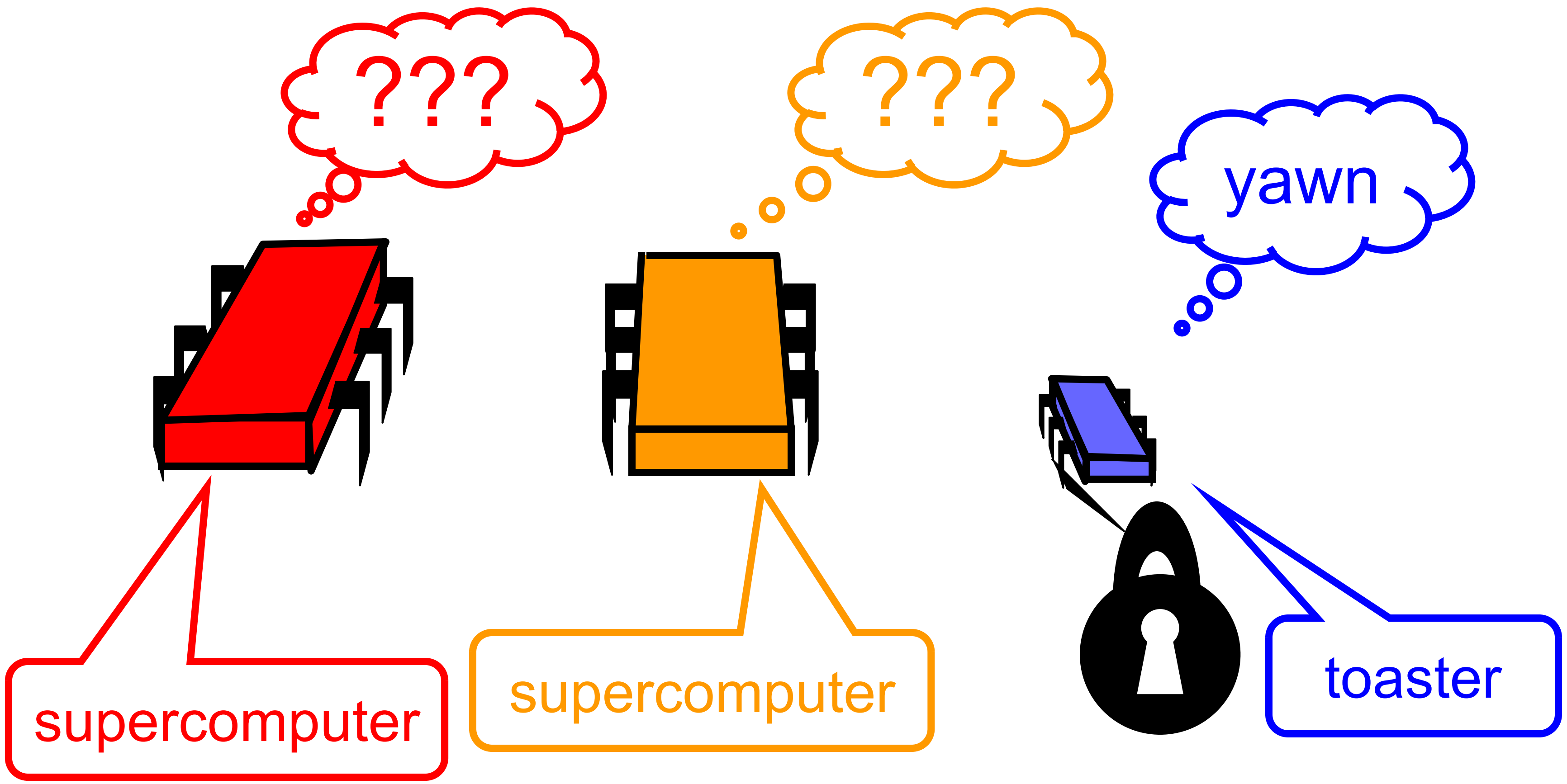
- We wanted atomic registers to implement mutual exclusion
- So we couldn't use mutual exclusion to implement atomic registers
- But wait, there's more!

What’s the problem with Mutual Exclusion?

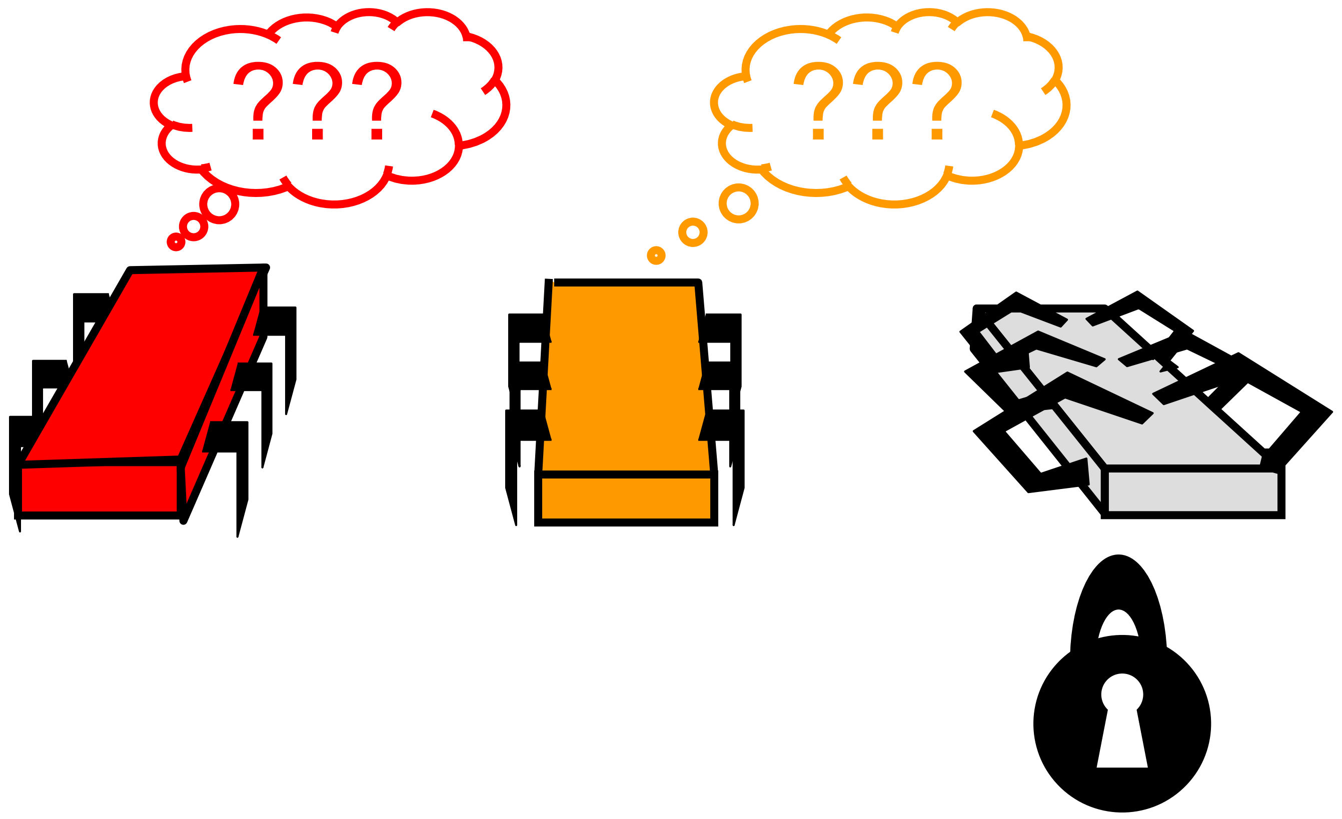
Asynchronous Interrupts



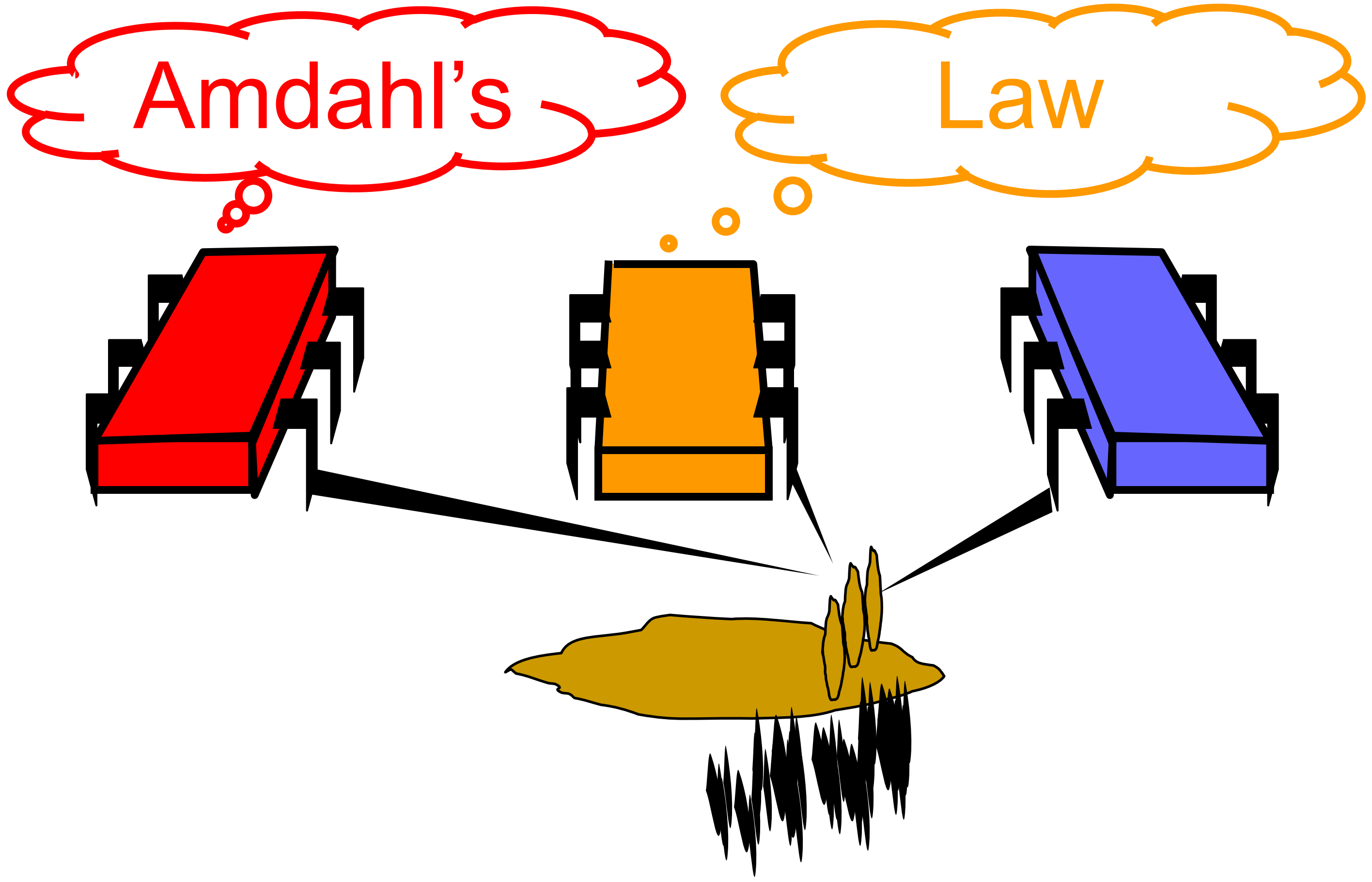
Heterogeneous Processors



Fault-tolerance



Machine Level Instruction Granularity



Basic Questions

- Wait-Free synchronization might be a good idea in principle

Basic Questions

- Wait-Free synchronization might be a good idea in principle
- But how do you do it …

Basic Questions

- Wait-Free synchronization might be a good idea in principle
- But how do you do it …
  - Systematically?

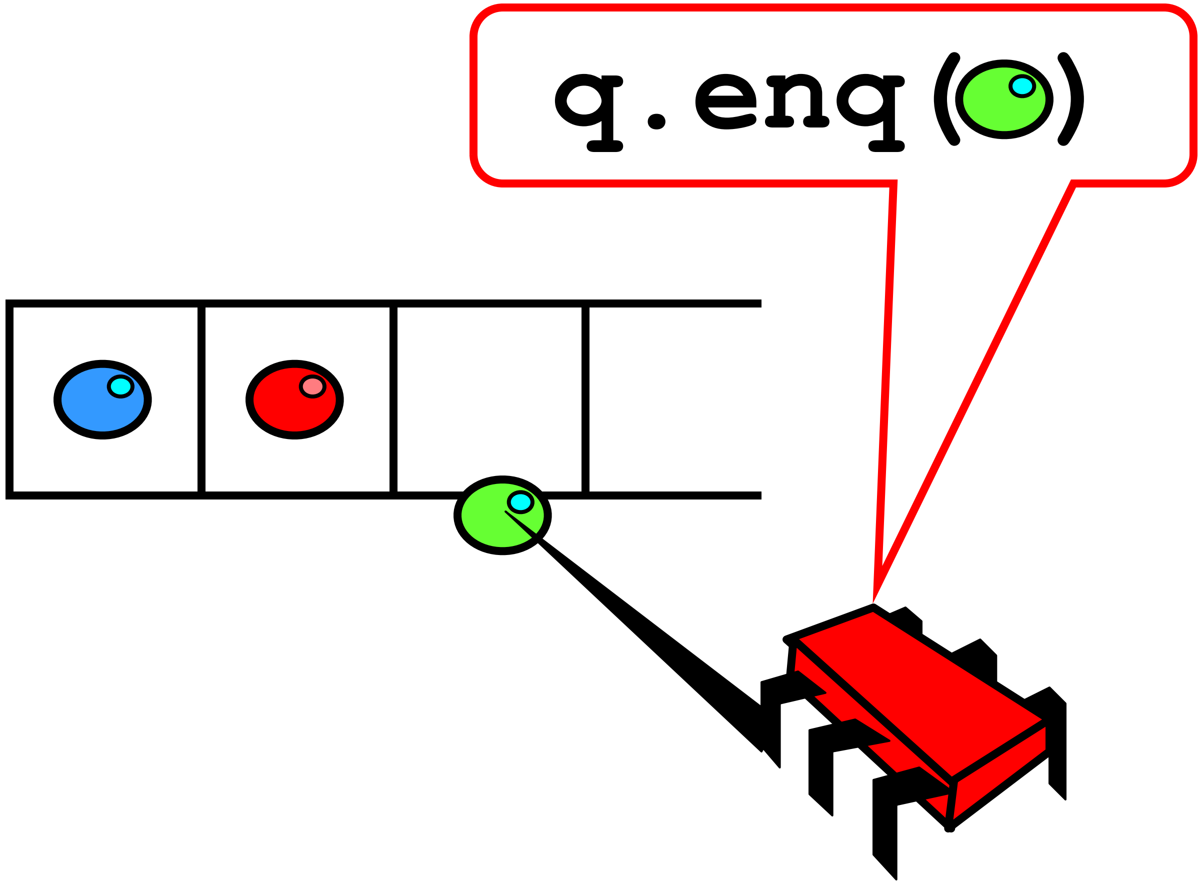
Basic Questions

- Wait-Free synchronization might be a good idea in principle
- But how do you do it …
  - Systematically?
  - Correctly?

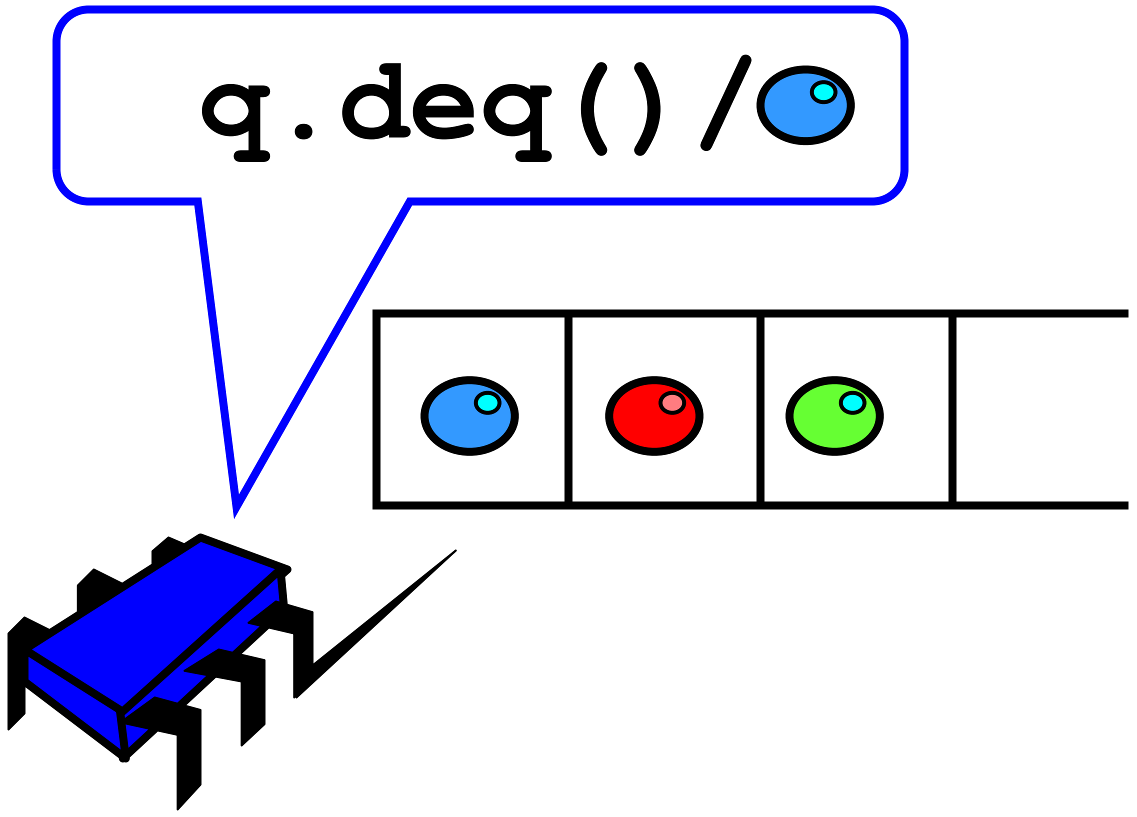
Basic Questions

- Wait-Free synchronization might be a good idea in principle
- But how do you do it …
  - Systematically?
  - Correctly?
  - Efficiently?

FIFO Queue: Enqueue Method

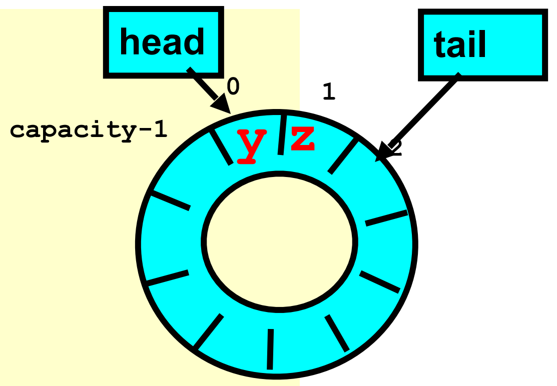


FIFO Queue: Dequeue Method

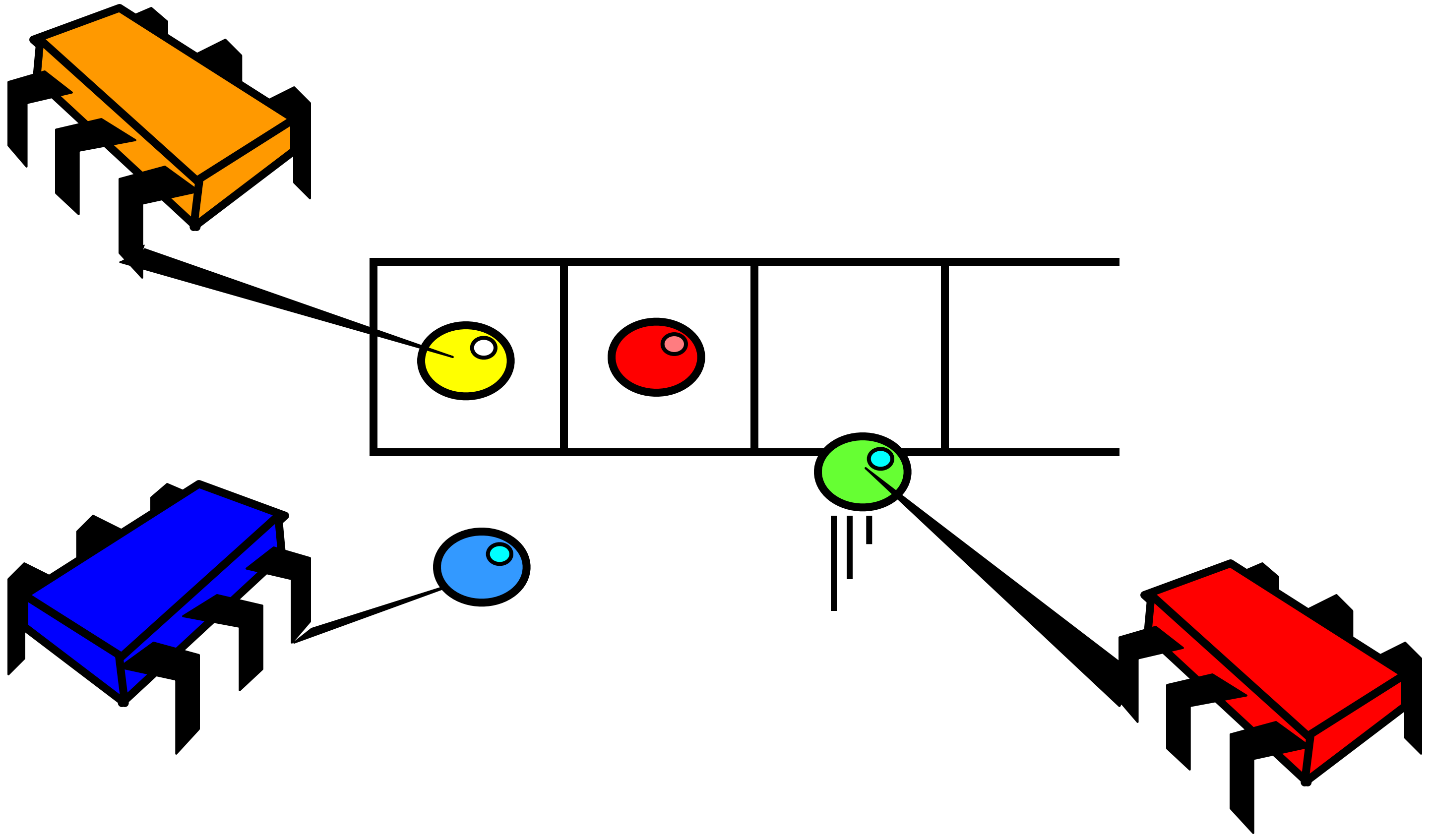


Two-Thread Wait-Free Queue

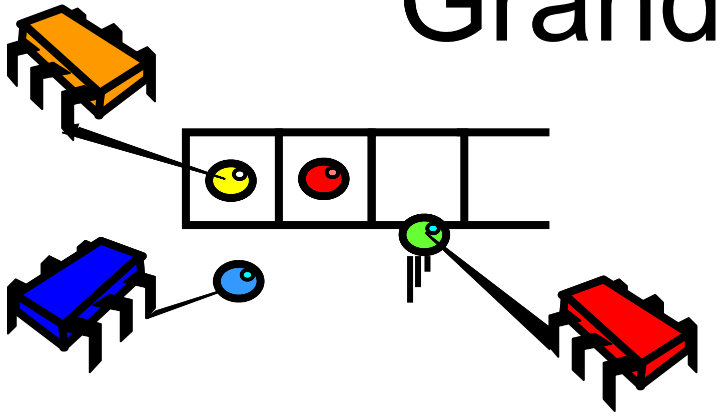
```
class LockFreeQueue[T: ClassTag](val capacity: Int) {

  @volatile
  private var head, tail: Int = 0
  private val items = new Array[T](capacity)

  def enq(x: T): Unit = {
    if (tail - head == items.length) throw FullException
    items(tail % items.length) = x
    tail = tail + 1
  }

  def deq(): T = {
    if (tail == head) throw EmptyException
    val x = items(head % items.length)
    head = head + 1
    x
  }
}
```



What About Multiple Dequeuers?

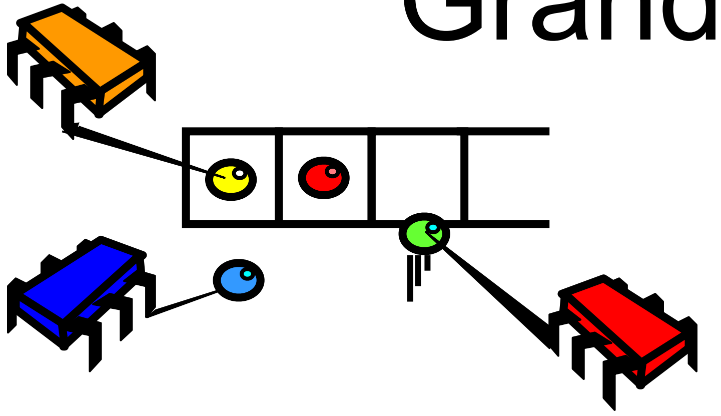


Grand Challenge



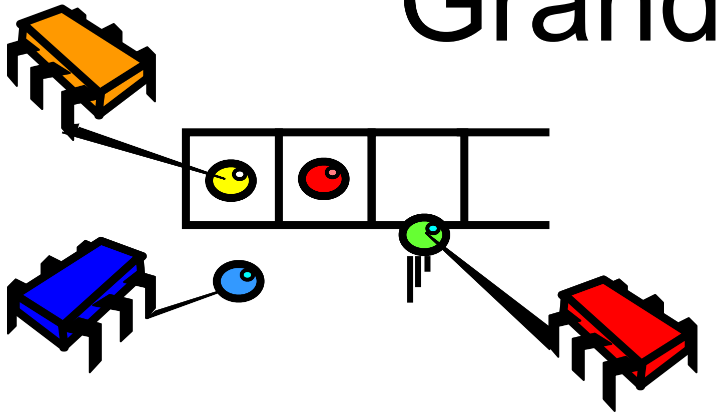
- Implement a FIFO queue

Grand Challenge



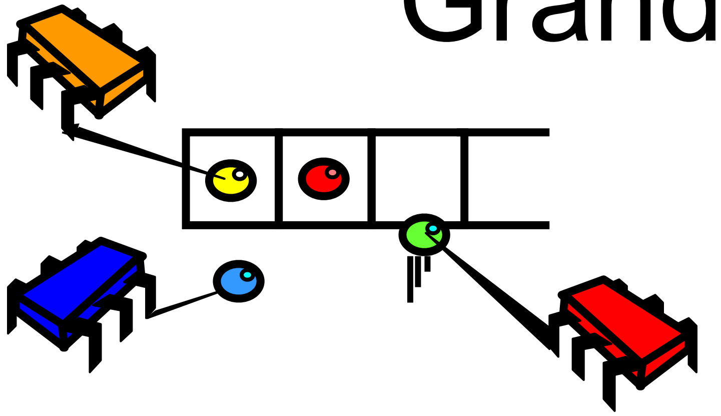
- Implement a FIFO queue
  - Wait-free

Grand Challenge



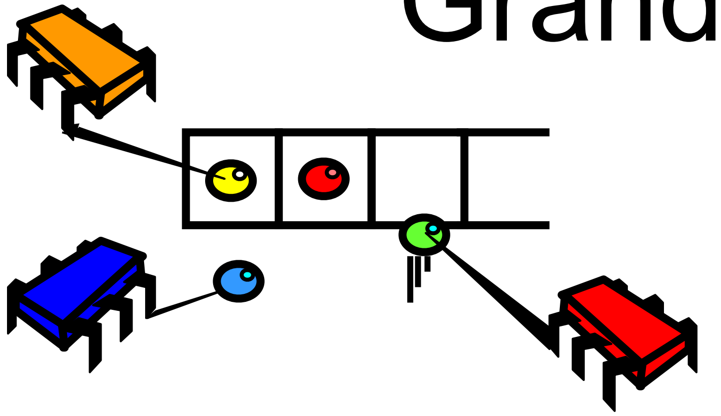
- Implement a FIFO queue
  - Wait-free
  - Linearizable

Grand Challenge

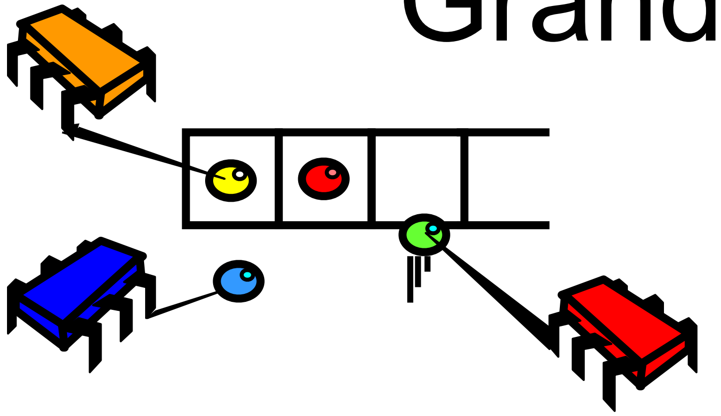


- Implement a FIFO queue
  - Wait-free
  - Linearizable
  - From atomic read-write registers

Grand Challenge



- Implement a FIFO queue
  - Wait-free
  - Linearizable
  - From atomic read-write registers
  - Multiple dequeuers



Grand Challenge

- Implement a FIFO queue
  - Wait-free
  - Linearizable
  - From atomic read-write registers
  - Multiple dequeuers

Only new aspect

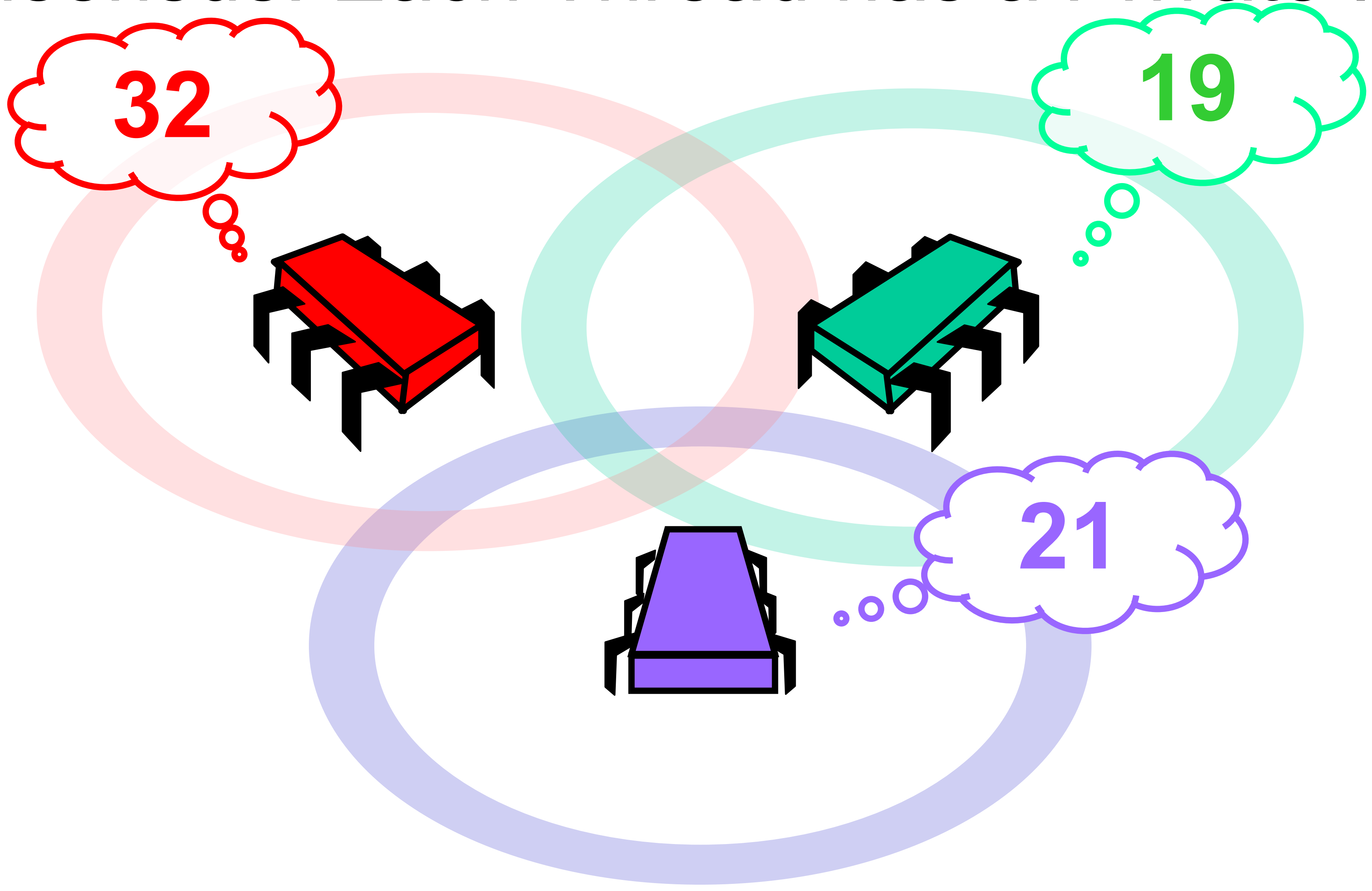
Puzzle

While you are ruminating on the grand challenge …

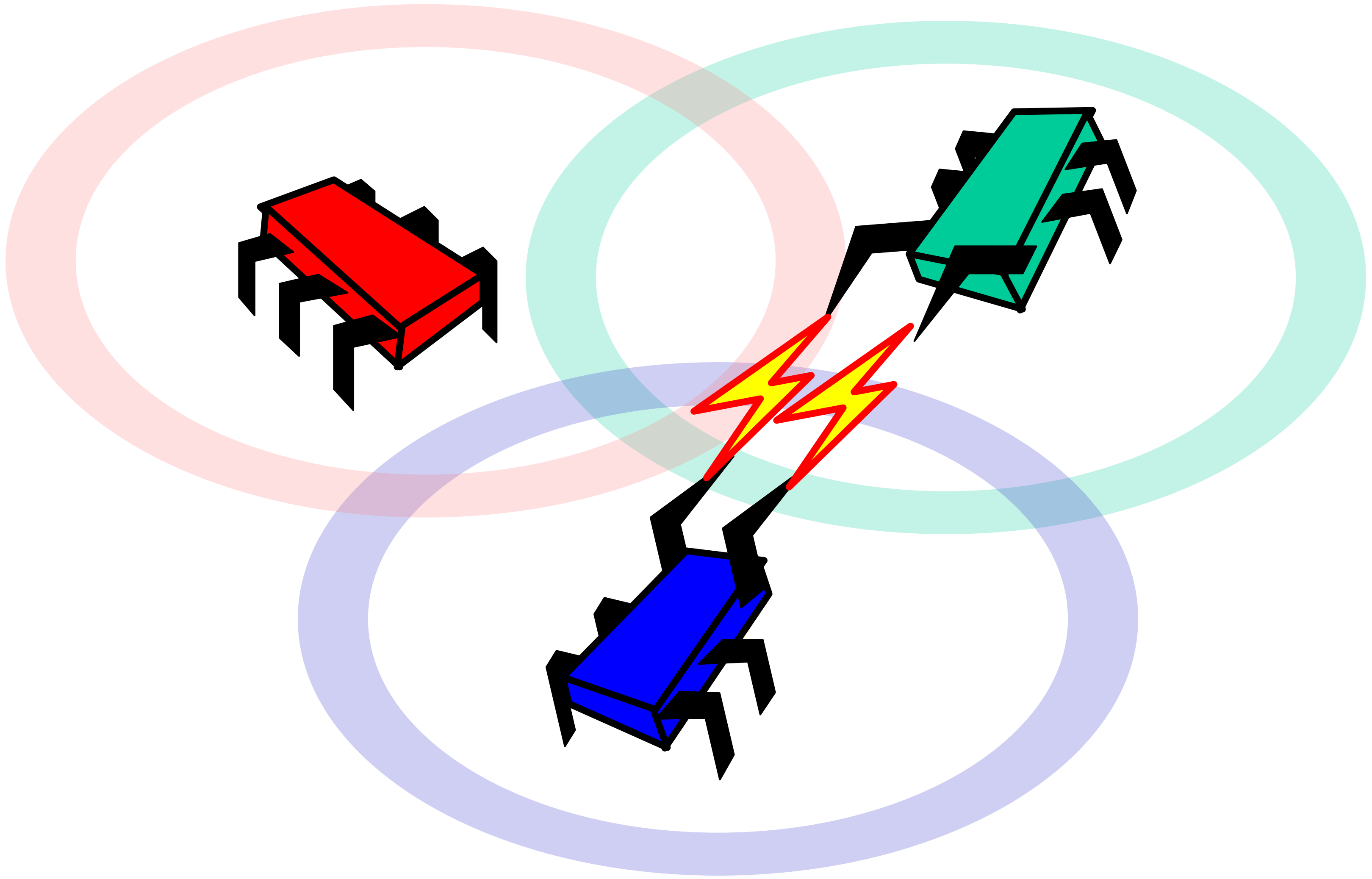
We will give you another puzzle …

Consensus!

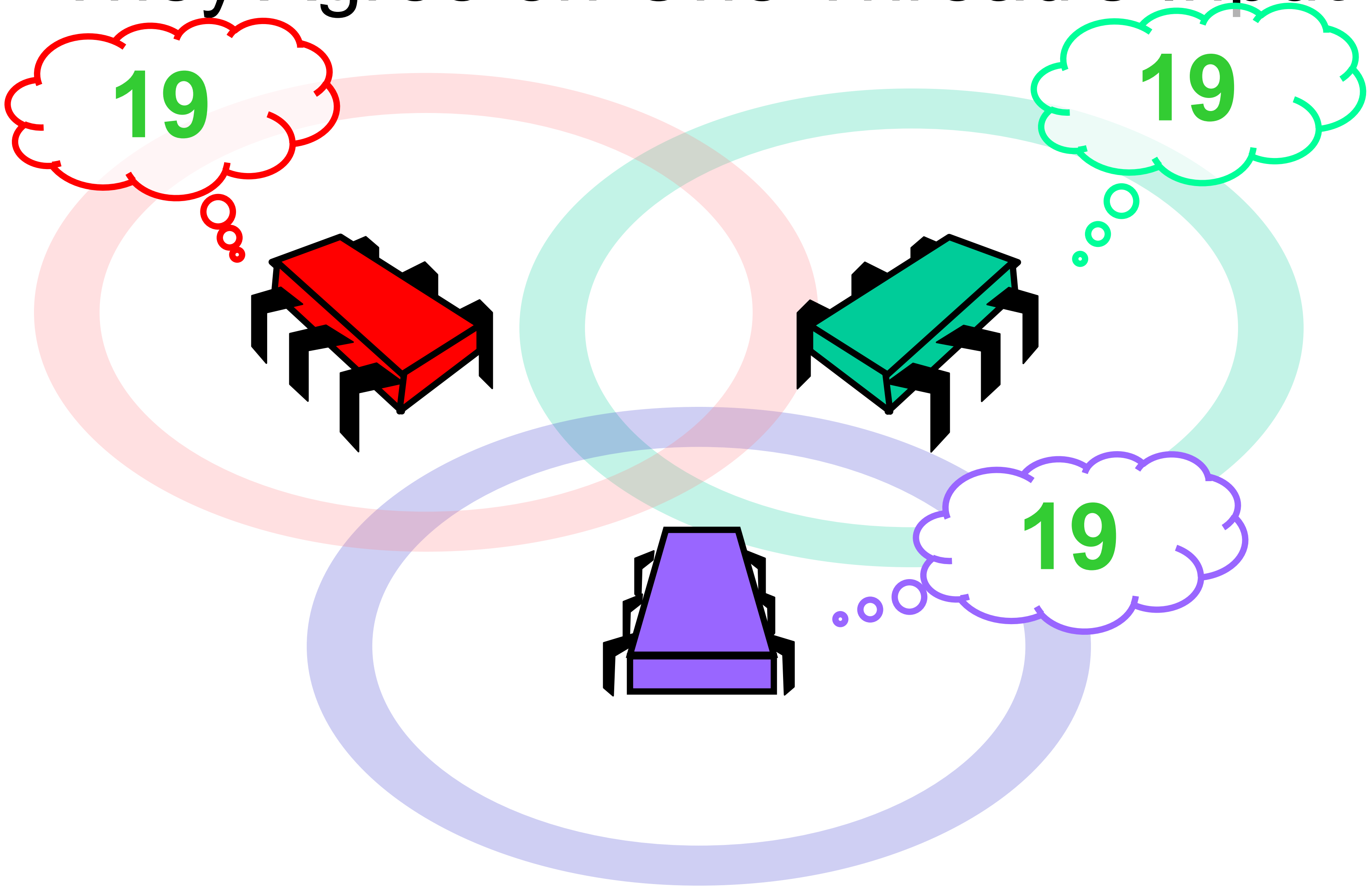
Consensus: Each Thread has a Private Input



They Communicate



They Agree on One Thread's Input



Formally: Consensus

- Consistent:
  - all threads decide the same value

Formally: Consensus

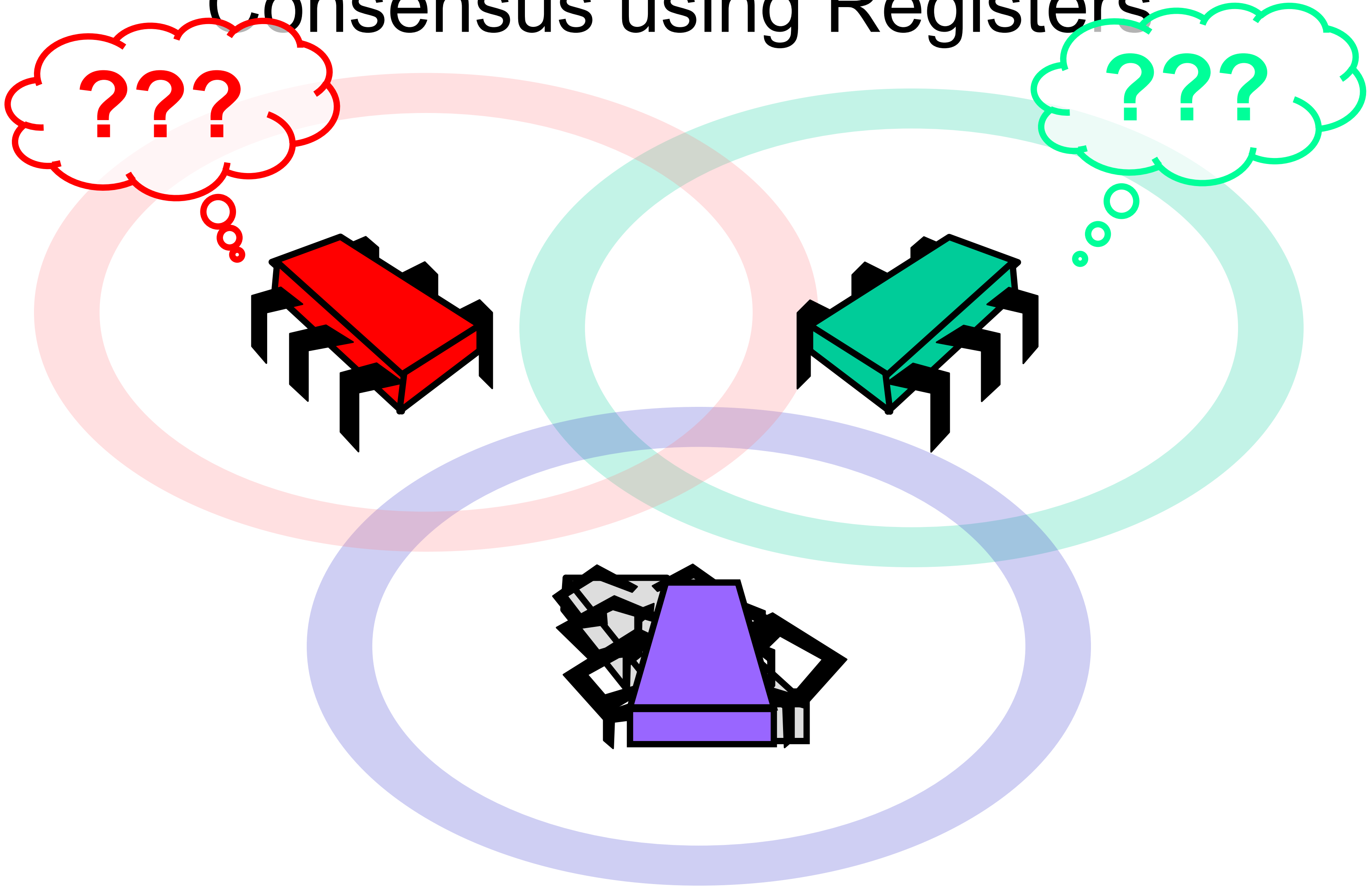
- Consistent:
  - all threads decide the same value
- Valid:
  - the common decision value is some thread's input

Let’s play into Consensus

- Two of you need to agree on a value, e.g., A or B
- You need to devise *a protocol to reach a consensus*
- Tell me *the maximal number of steps* for each thread (<= 5, please)
- We are going to communicate using the white board
- Rules: *either* reading or writing **one** register (not both)
- *No other communication,*
- **No priorities** in “thread” identities or values:
  - Either of the values can be chosen (non-triviality)
  - One of the thread’s suggestions need to be chosen (validity)

Game

No Wait-Free Implementation of Consensus using Registers



Formally

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

Formally

- Theorem
  - There is no wait-free implementation of *n*-thread consensus from read-write registers
- Implication
  - Asynchronous computability different from Turing computability

Proof Strategy

Assume otherwise …

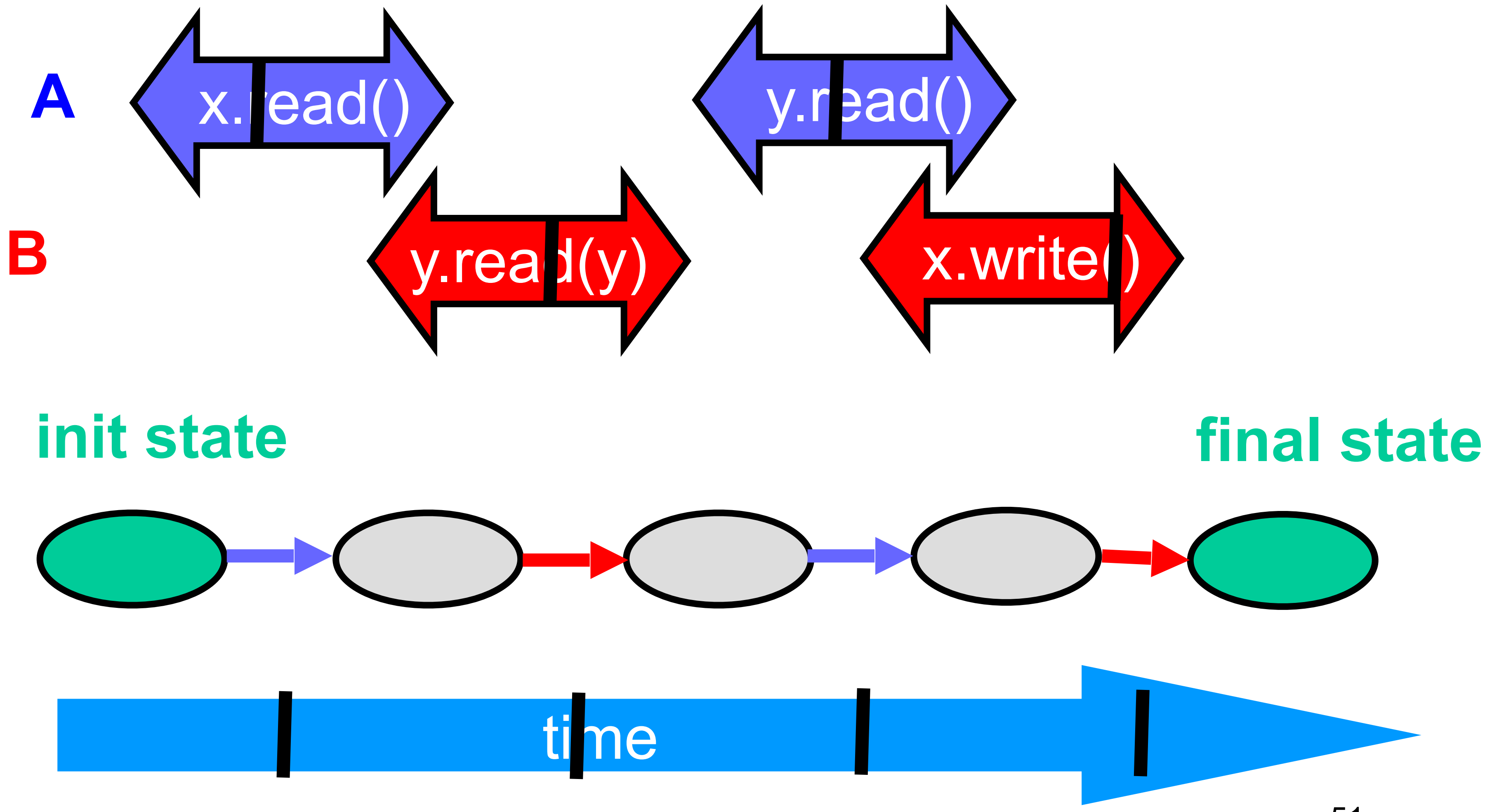
Reason about the properties of any such protocol …

Derive a contradiction

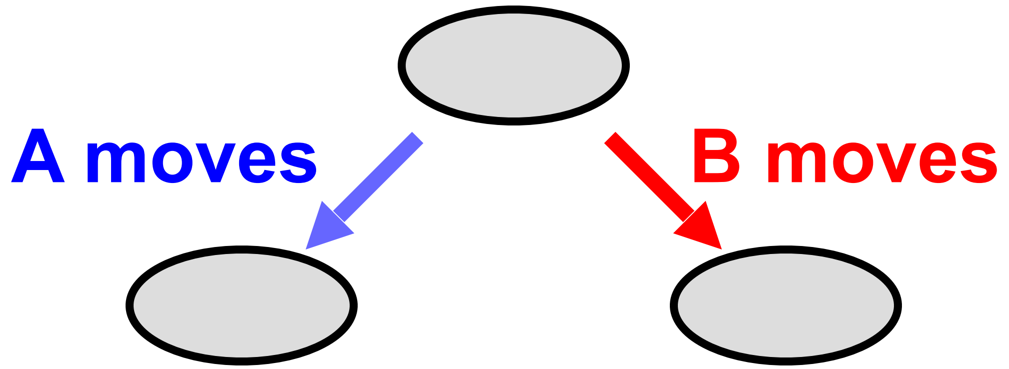
Quod
Erat
Demonstrandum

Enough to consider binary consensus and $n=2$

Protocol Histories as State Transitions

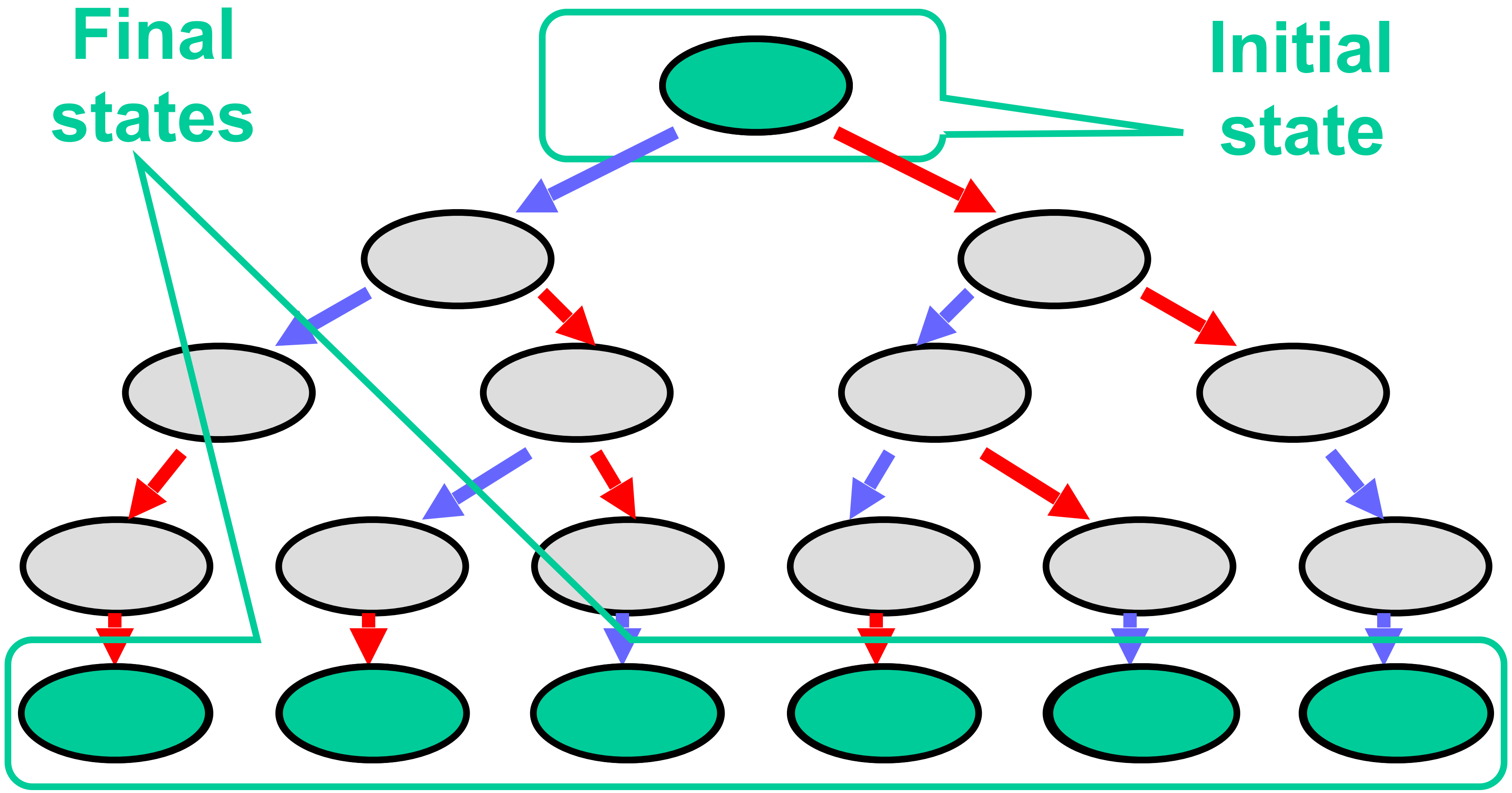


Wait-Free Computation

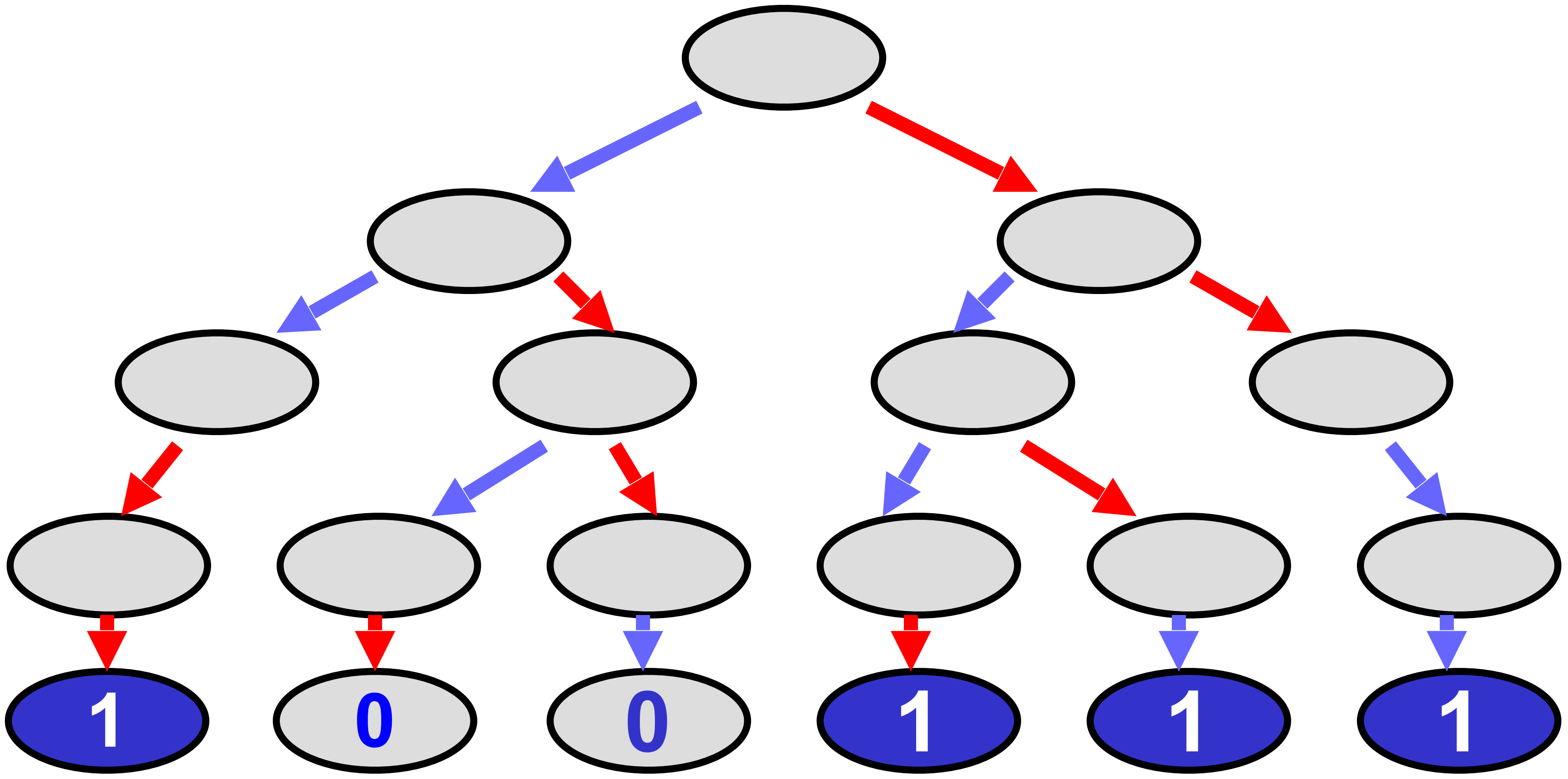


- Either A or B "moves"
- Moving means
  - Register read
  - Register write

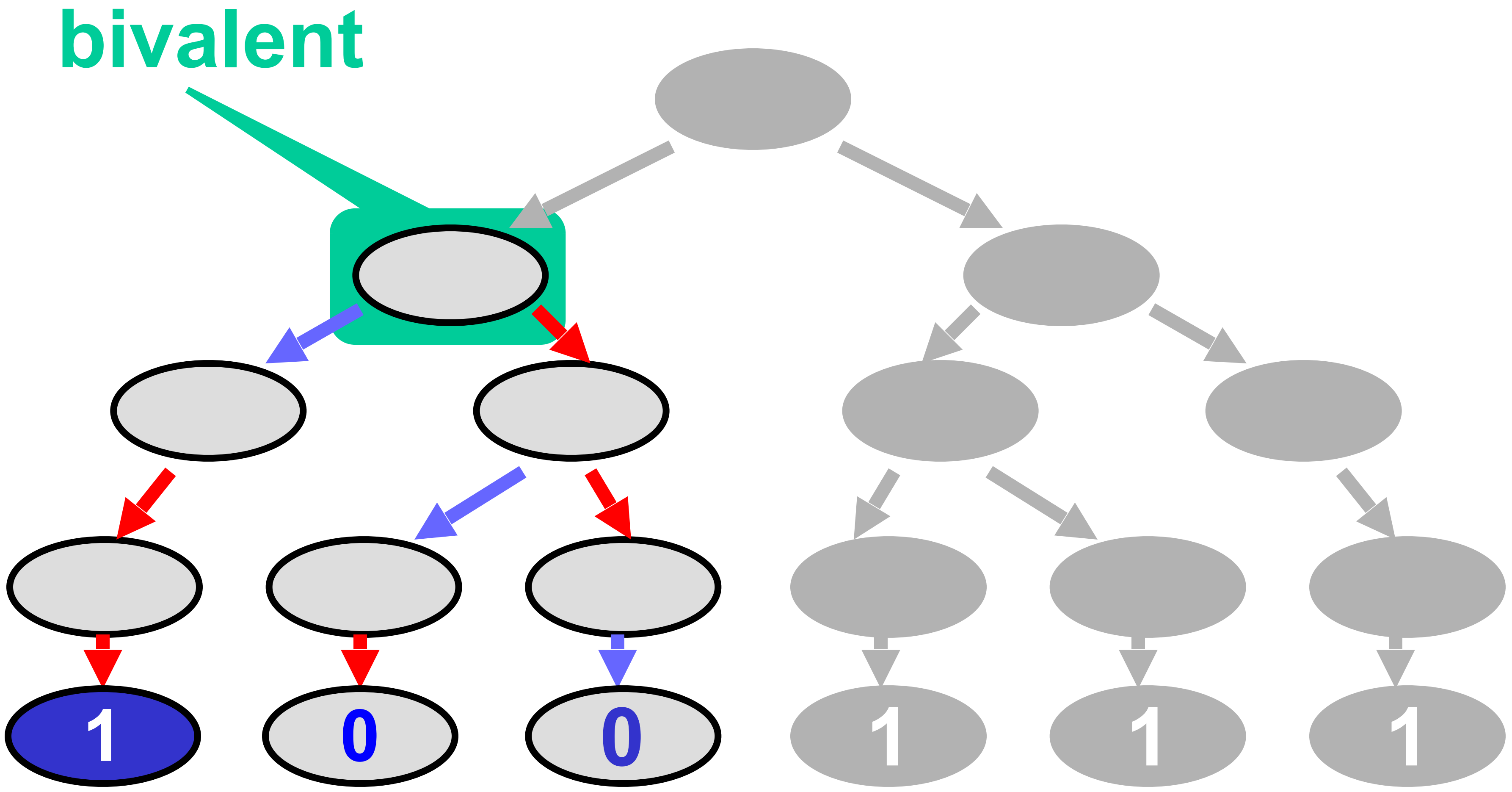
The Two-Move Tree



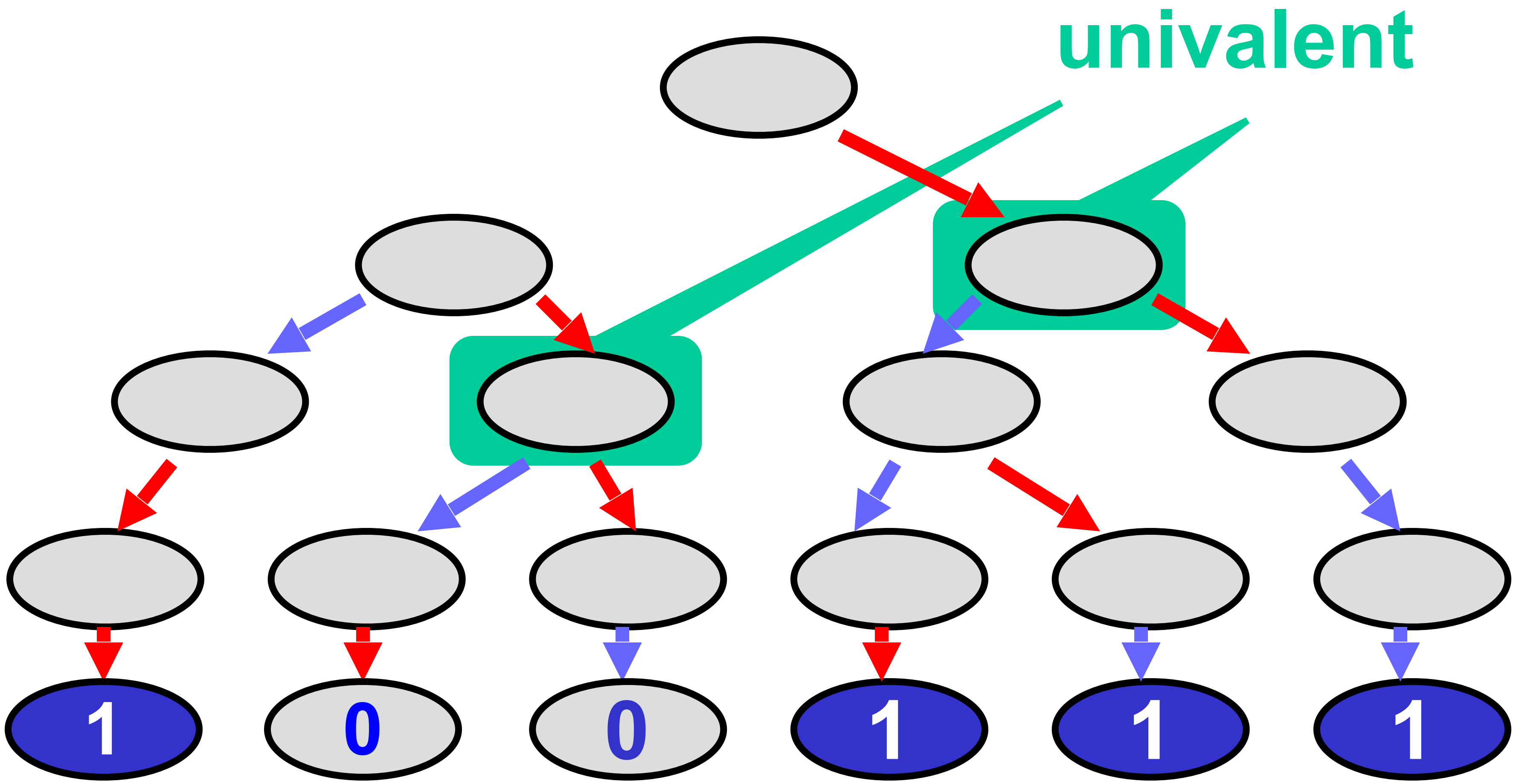
Decision Values



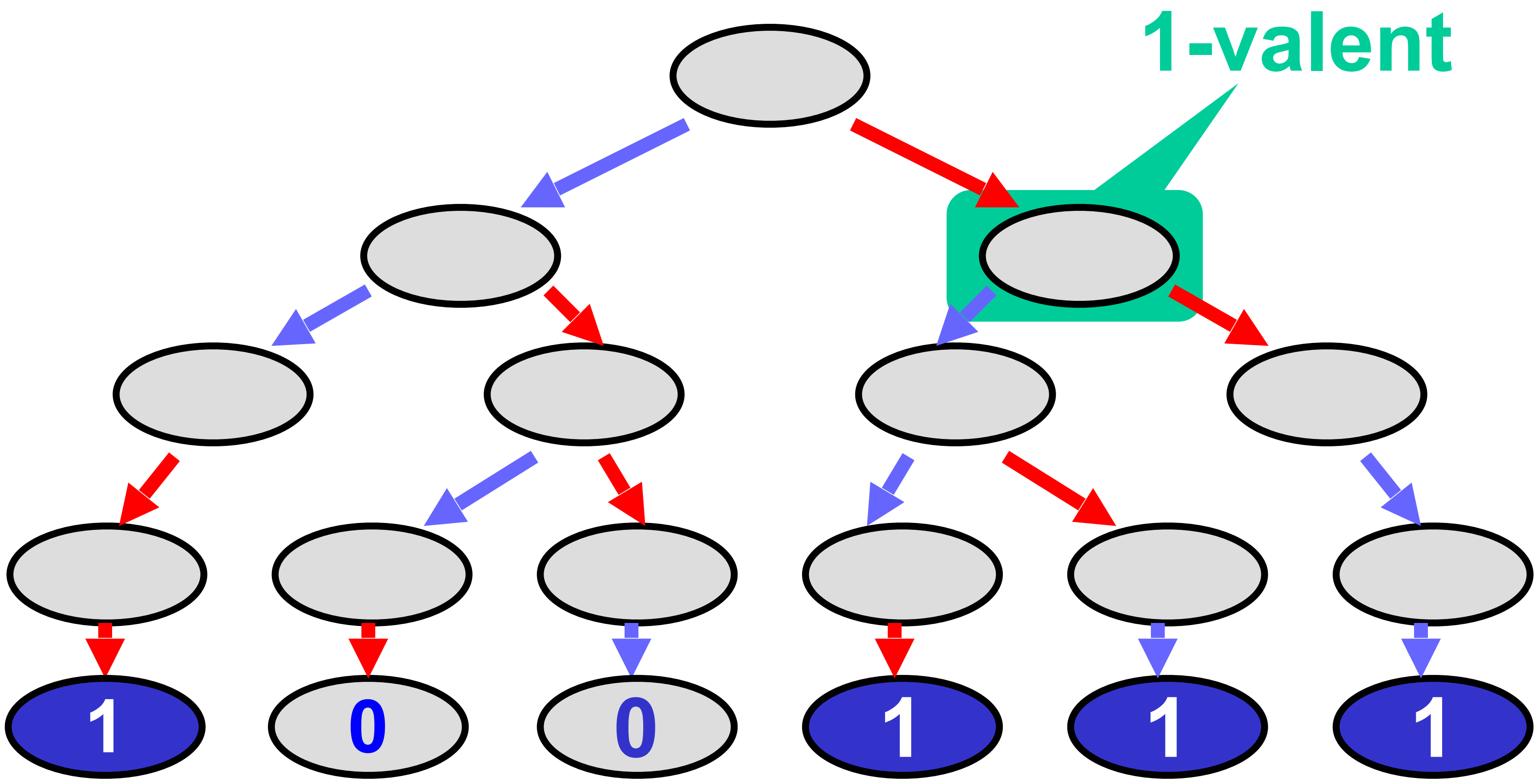
Bivalent: Both Possible



Univalent: Single Value Possible



x-valent: x Only Possible Decision



Summary

- Wait-free computation is a tree

Summary

- Wait-free computation is a tree
- Bivalent system states
  - Outcome not fixed

Summary

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

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

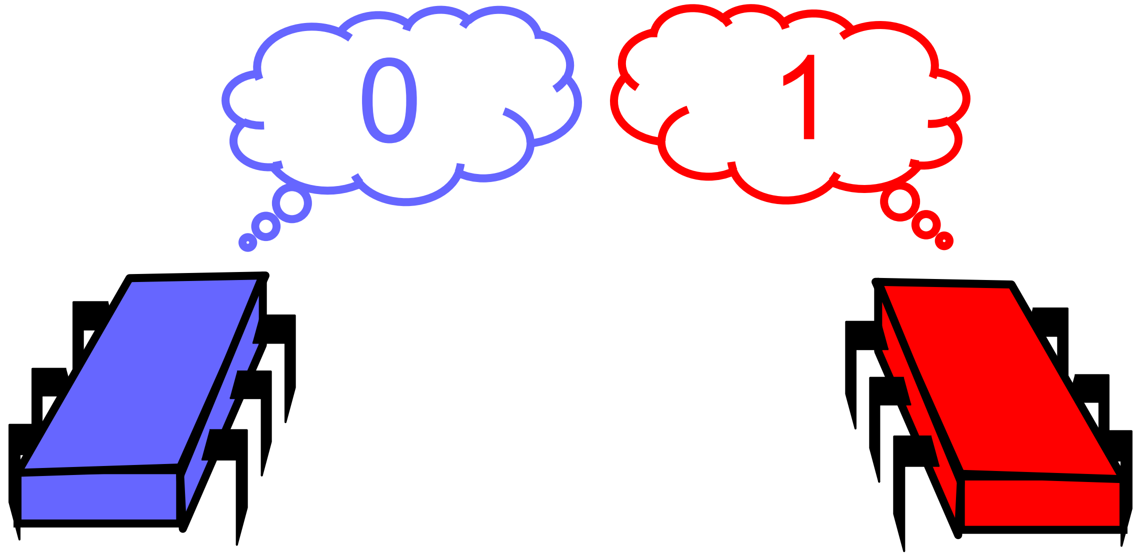
Claim

- Some initial state is bivalent
- Outcome depends on
  - Chance
  - Whim of the scheduler
- Multicore gods procrastinate …

Claim

- Some initial state is bivalent
- Outcome depends on
  - Chance
  - Whim of the scheduler
- Multicore gods procrastinate …
- Let's prove it …

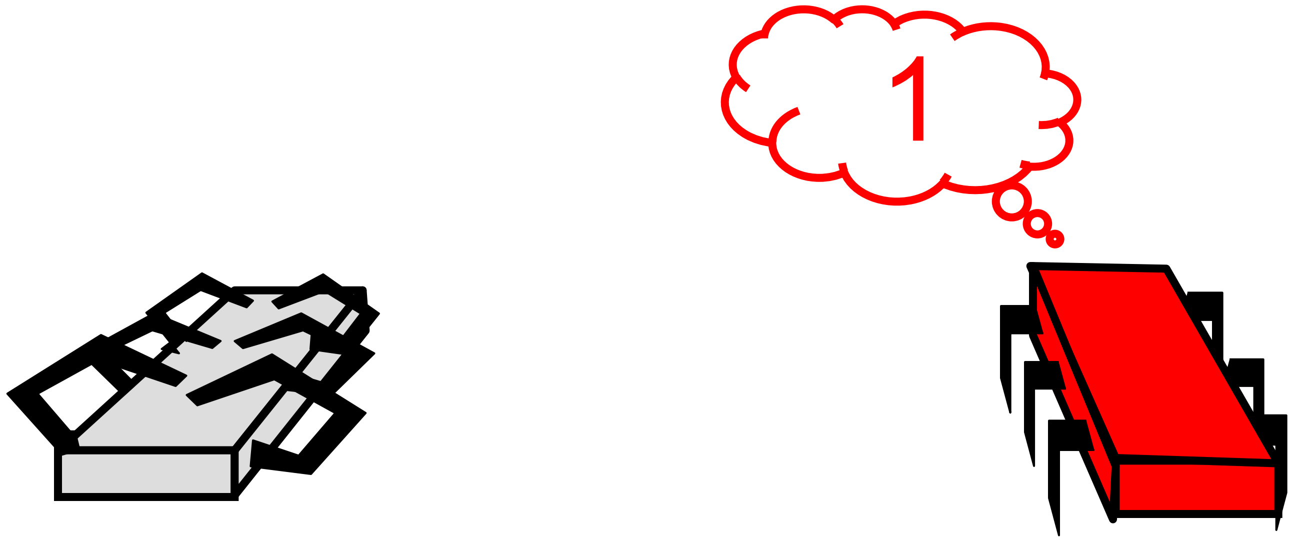
What if inputs differ?



Must Decide 0

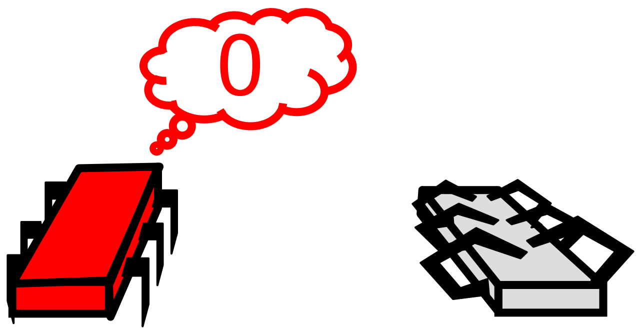
In this solo execution by A

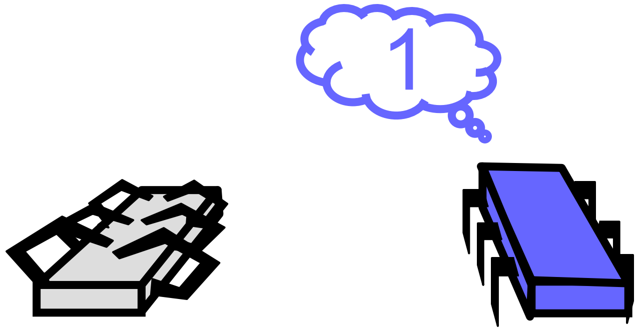
Must Decide 1



In this solo execution by B

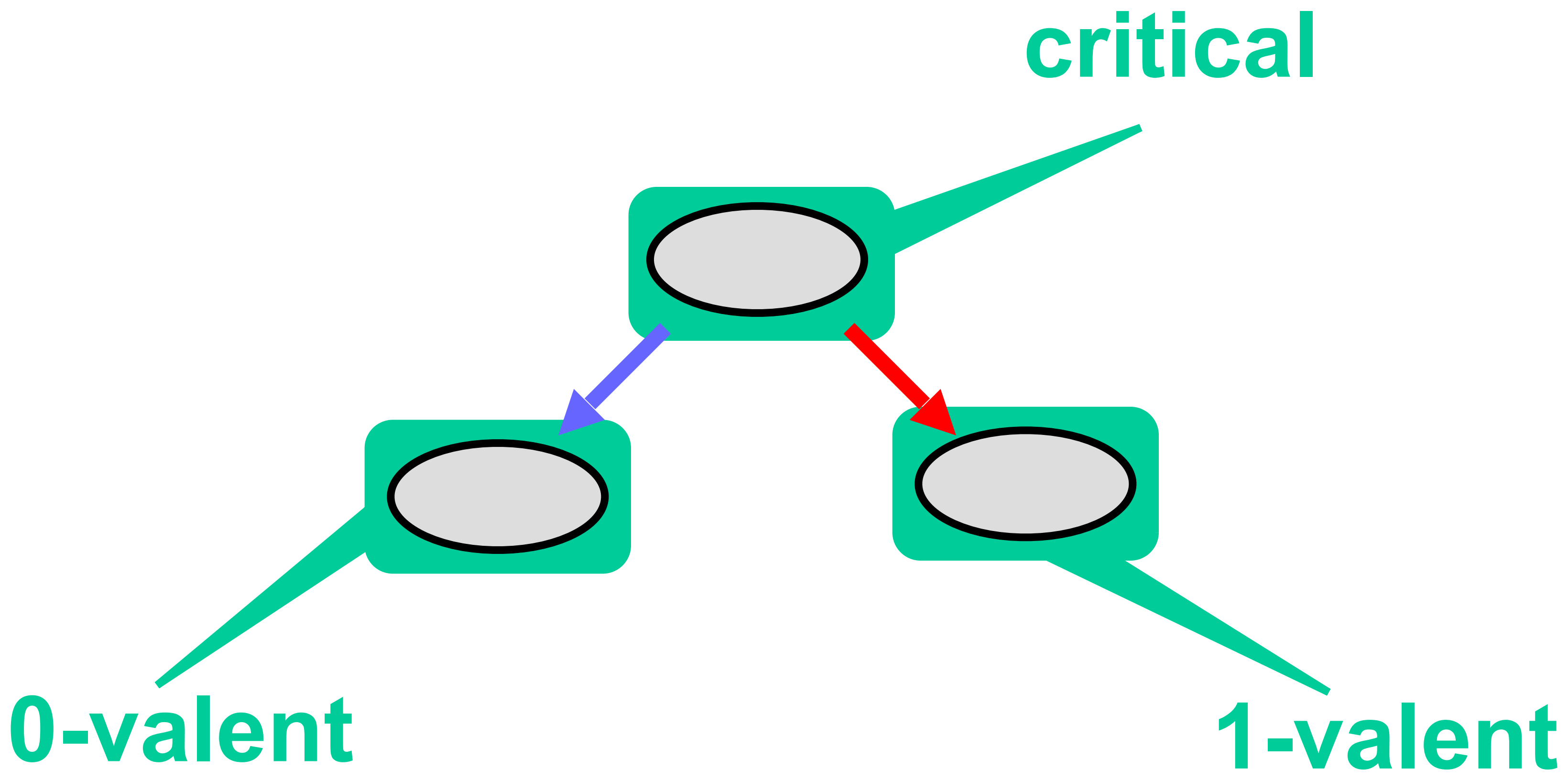
Mixed Initial State Bivalent



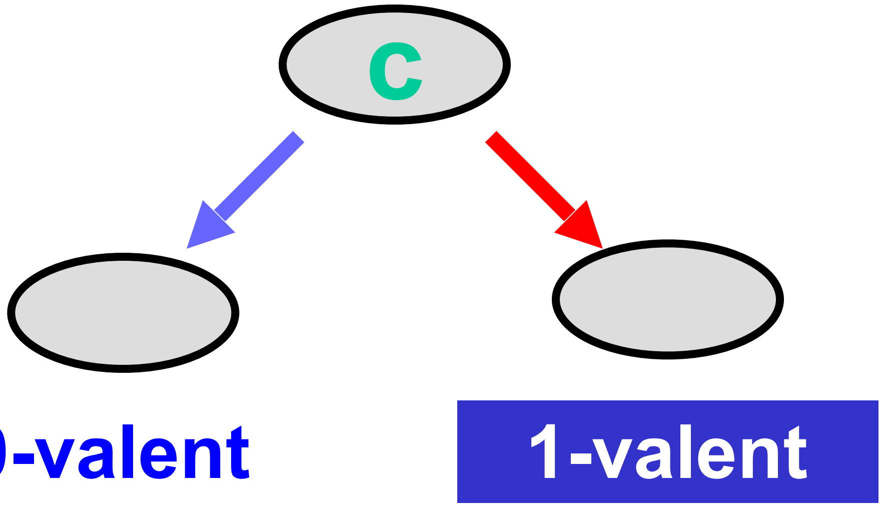


- Solo execution by A must decide 0
- Solo execution by B must decide 1

Critical States



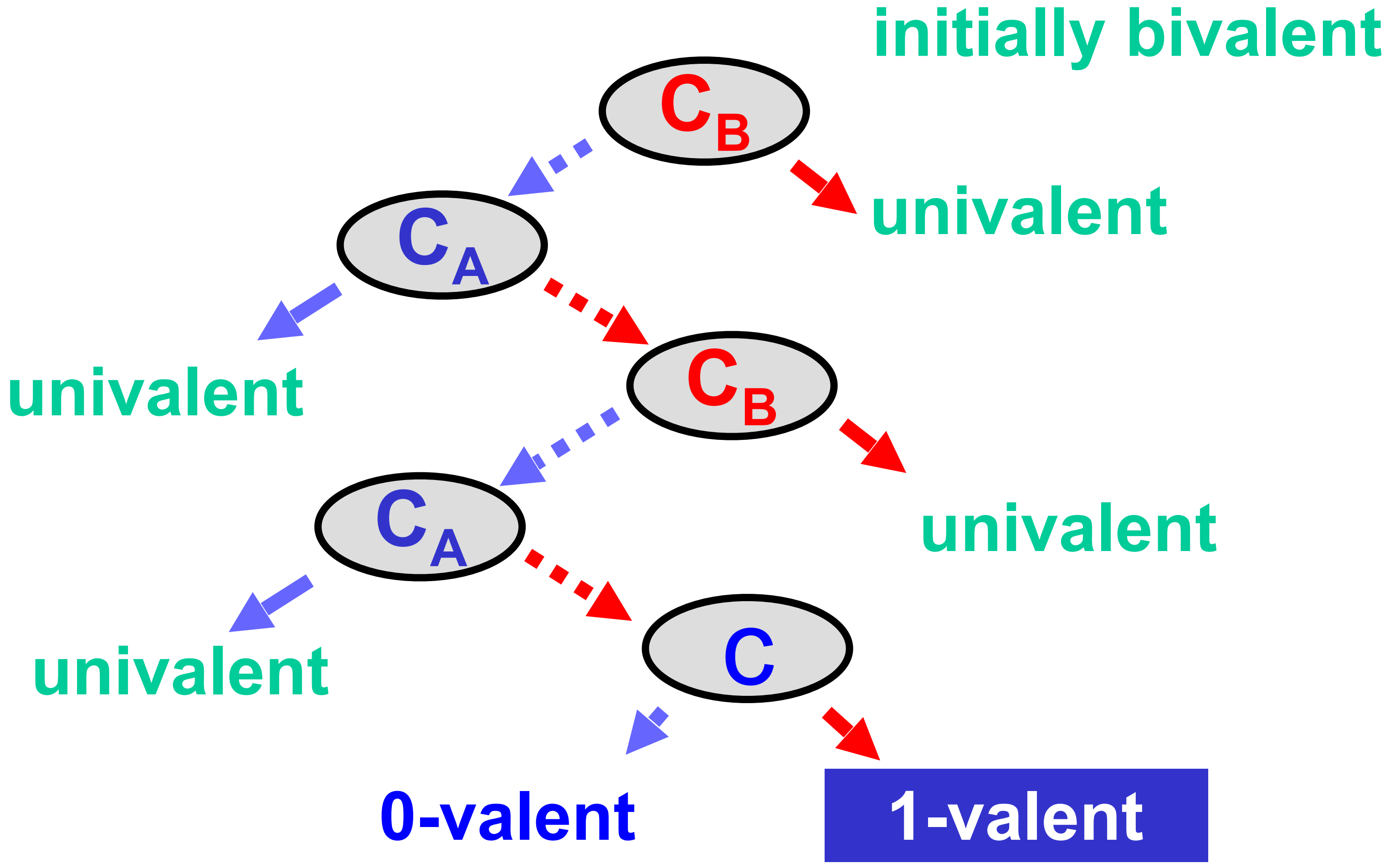
From a Critical State



If A goes first, protocol decides 0

If B goes first, protocol decides 1

Reaching Critical State



Critical States

- Starting from a bivalent initial state

Critical States

- Starting from a bivalent initial state
- The protocol can reach a critical state

Critical States

- Starting from a bivalent initial state
- The protocol can reach a critical state
  - Otherwise we could stay bivalent forever
  - And the protocol is not wait-free

Model Dependency

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

Closing the Deal

- Start from a critical state

Closing the Deal

- Start from a critical state
- Each thread fixes outcome by
  - Reading or writing …
  - Same or different registers

Closing the Deal

- Start from a critical state
- Each thread fixes outcome by
  - Reading or writing …
  - Same or different registers
- Leading to a 0 or 1 decision …

Closing the Deal

- Start from a critical state
- Each thread fixes outcome by
  - Reading or writing …
  - Same or different registers
- Leading to a 0 or 1 decision …
- And a contradiction.

Possible Interactions

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

Possible Interactions

A reads x

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

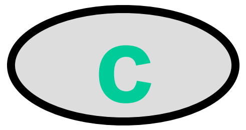
Possible Interactions

A reads x

A reads y

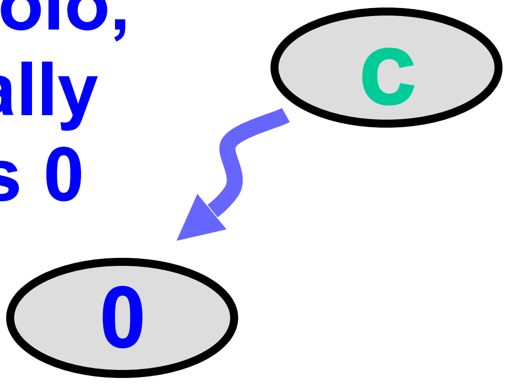
	x.read()	y.read()	x.write()	y.write()
x.read()	?	?	?	?
y.read()	?	?	?	?
x.write()	?	?	?	?
y.write()	?	?	?	?

Some Thread Reads

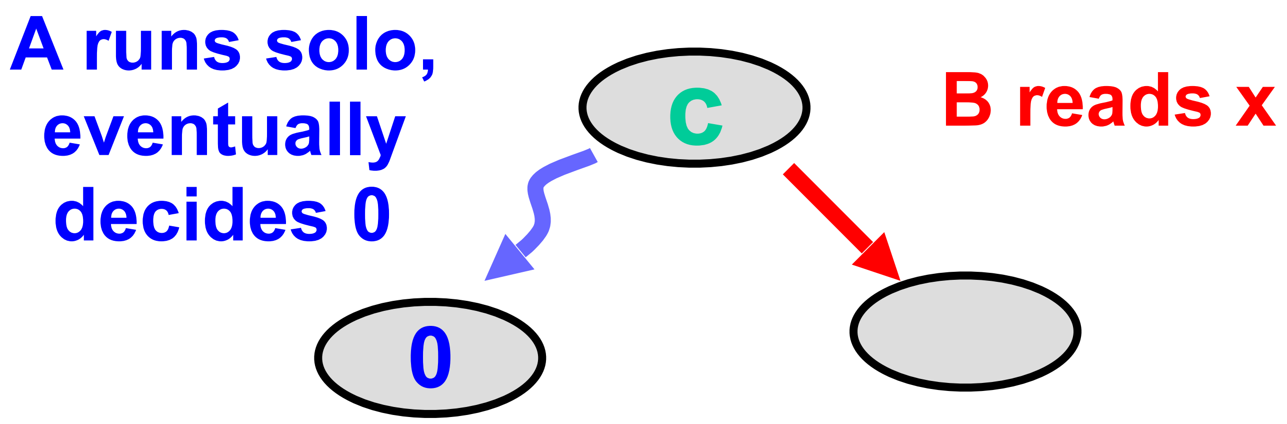


Some Thread Reads

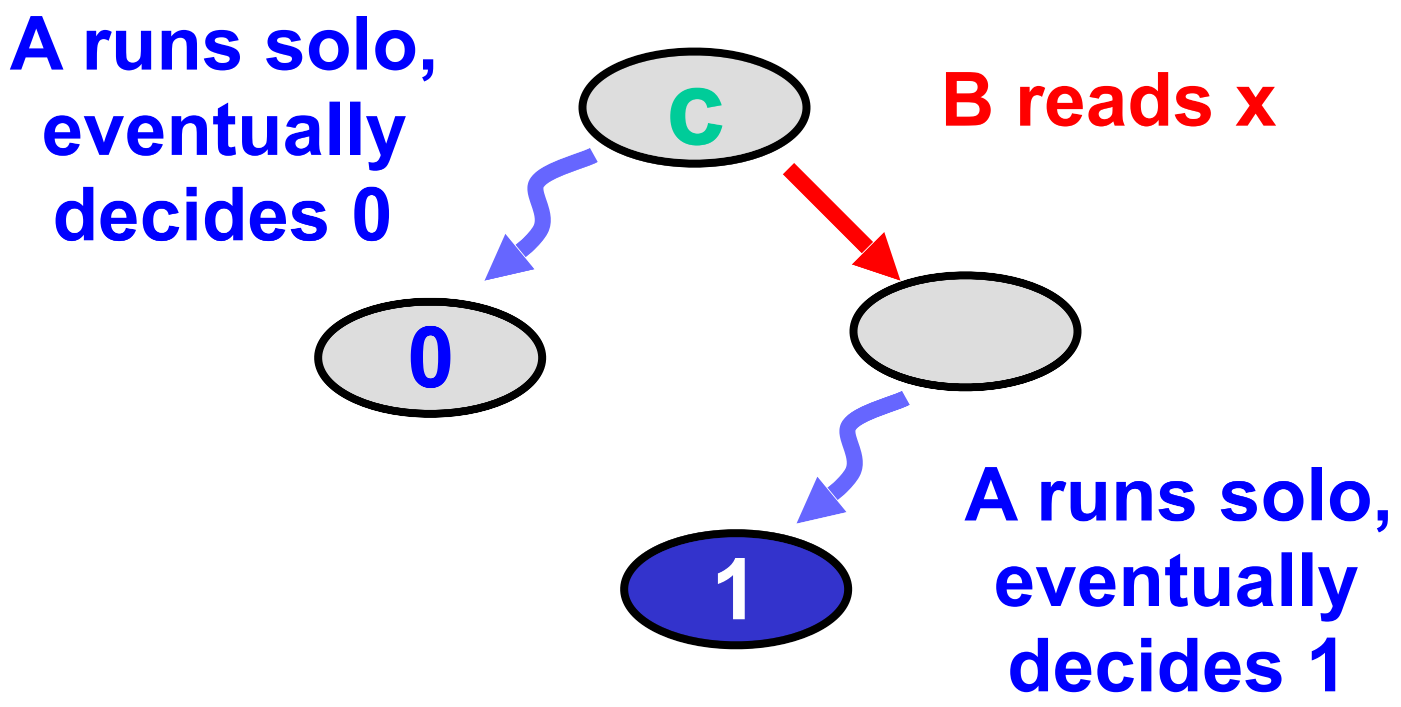
A runs solo, eventually decides 0



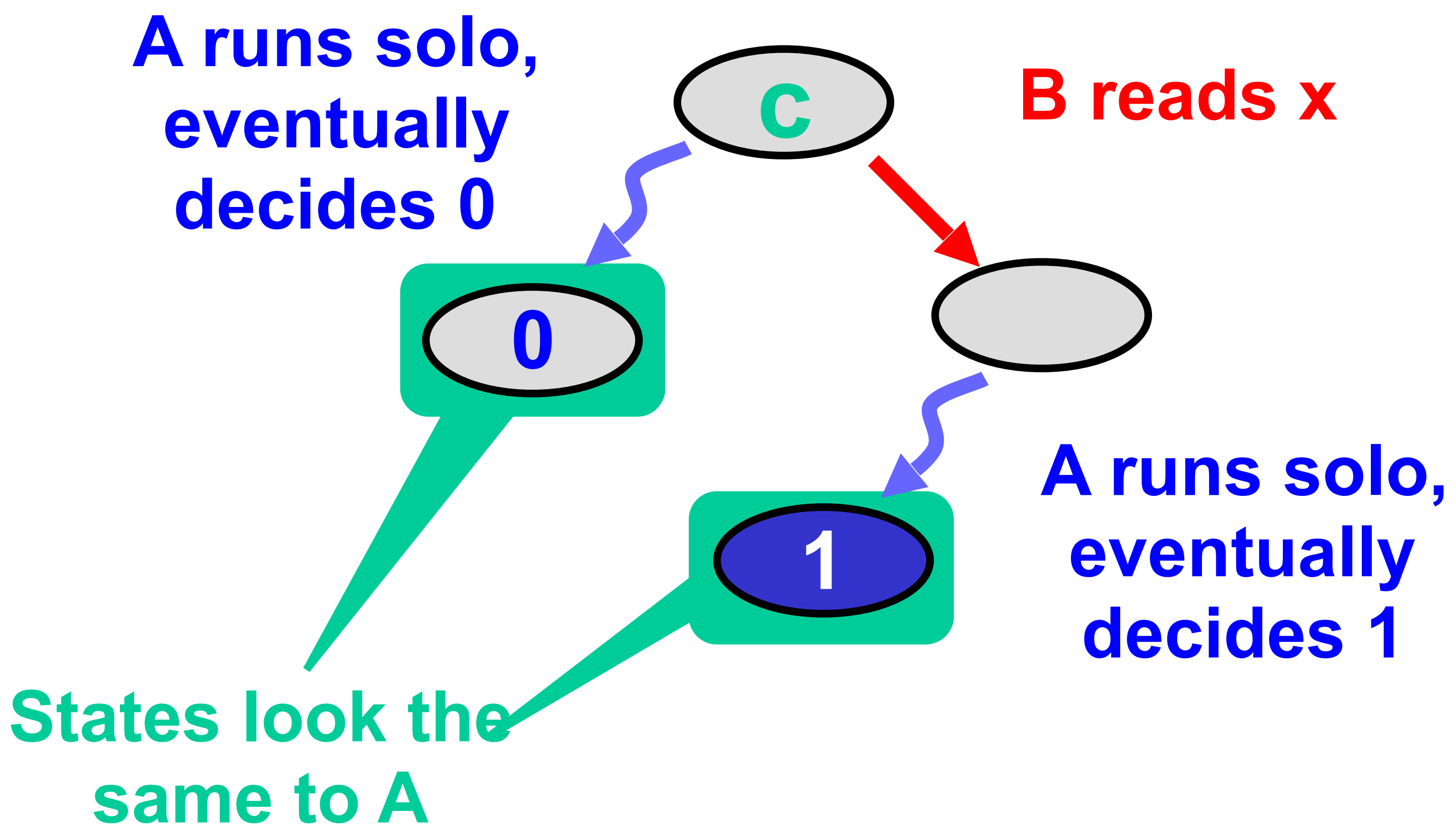
Some Thread Reads



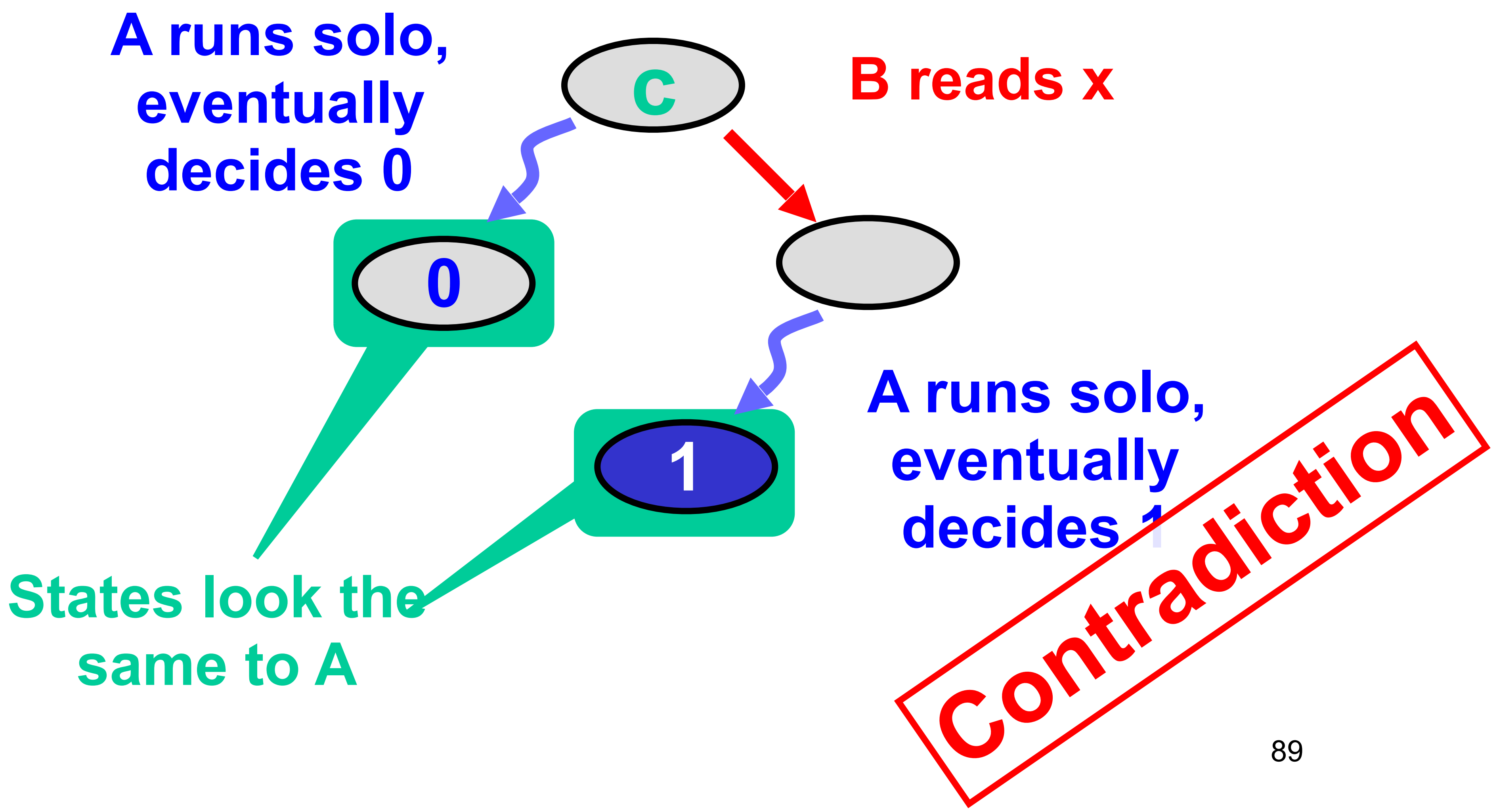
Some Thread Reads


A runs solo, eventually decides 0
c
B reads x
0
1
A runs solo, eventually decides 1

Some Thread Reads


A runs solo, eventually decides 0
c
B reads x
0
1
A runs solo, eventually decides 1
States look the same to A

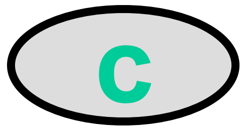
Some Thread Reads


A runs solo, eventually decides 0
c
B reads x
0
1
A runs solo, eventually decides 1
States look the same to A
Contradiction

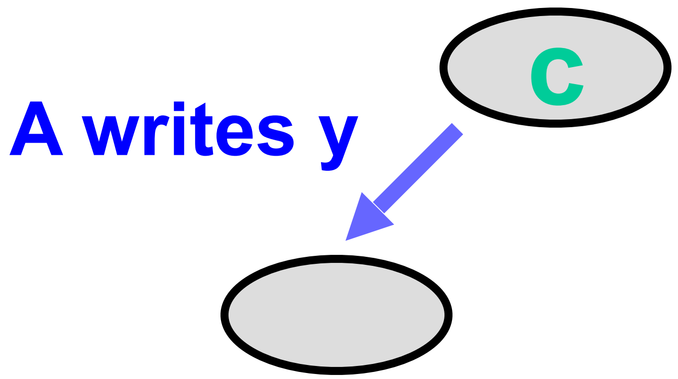
Possible Interactions

	`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	?	?

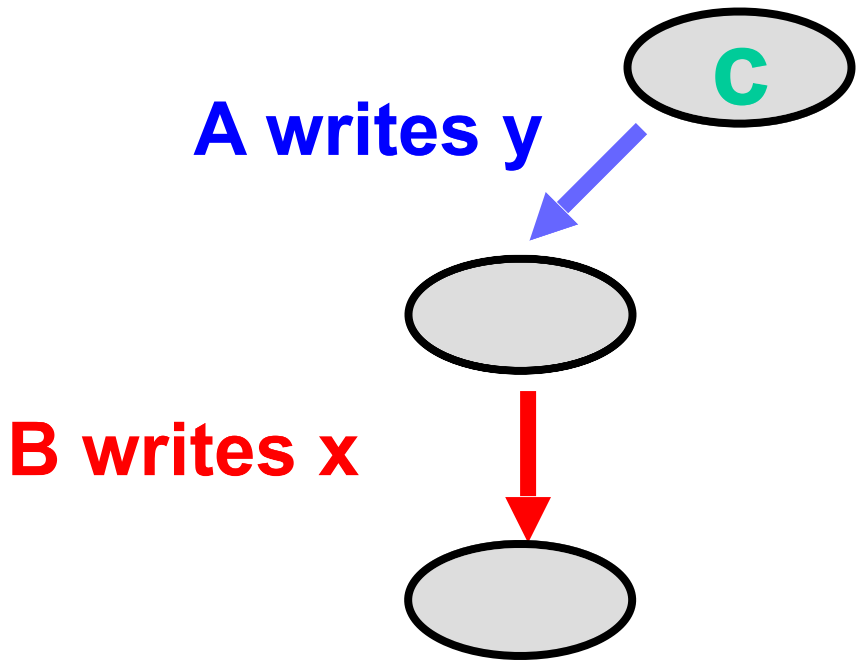
Writing Distinct Registers



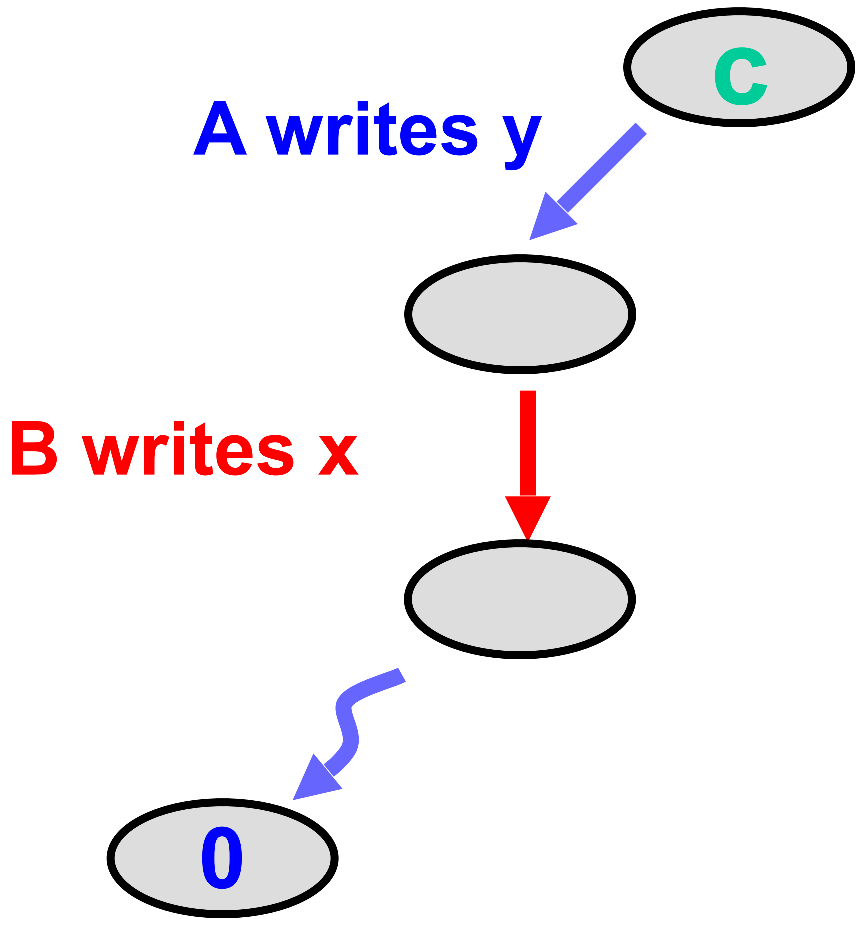
Writing Distinct Registers



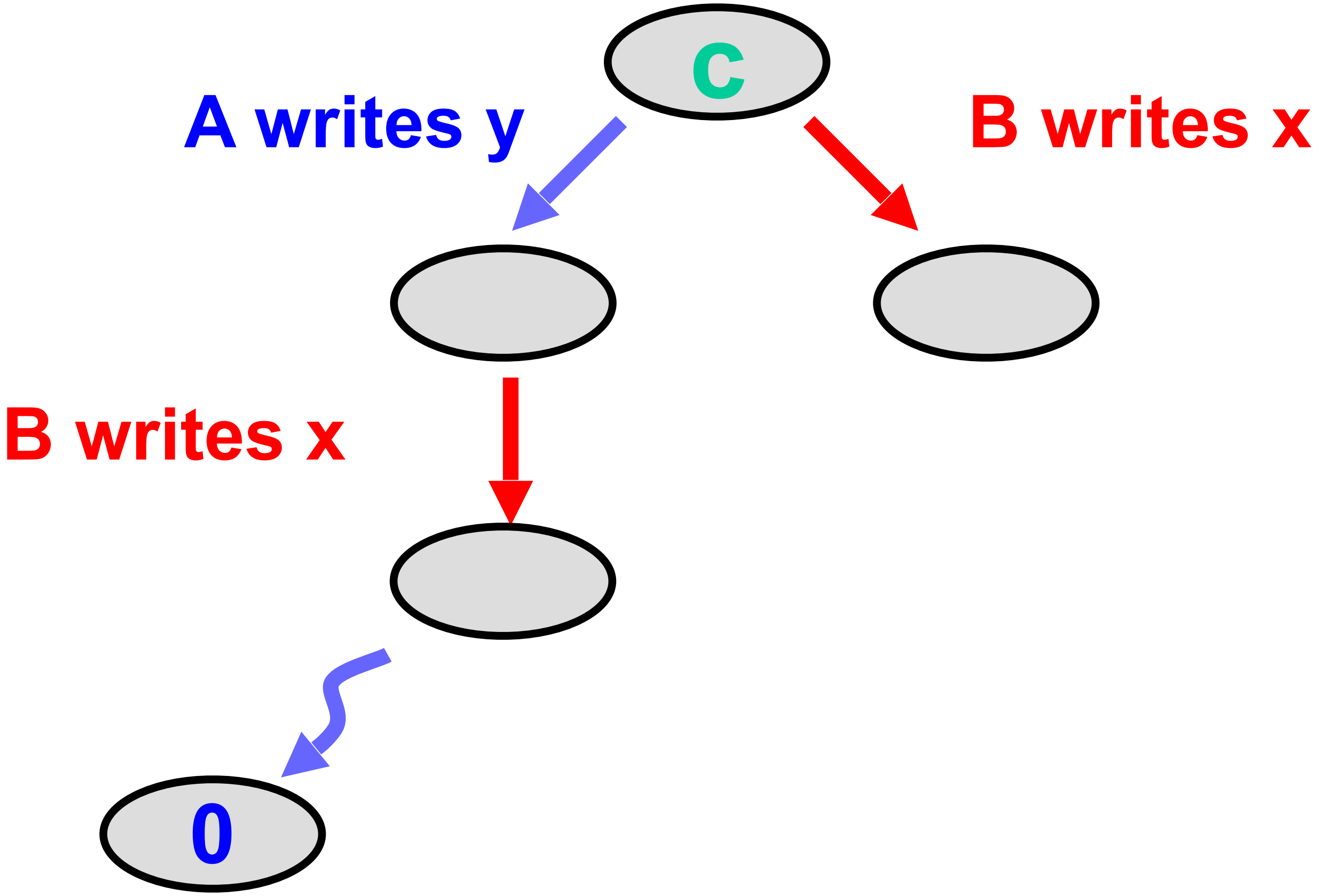
Writing Distinct Registers



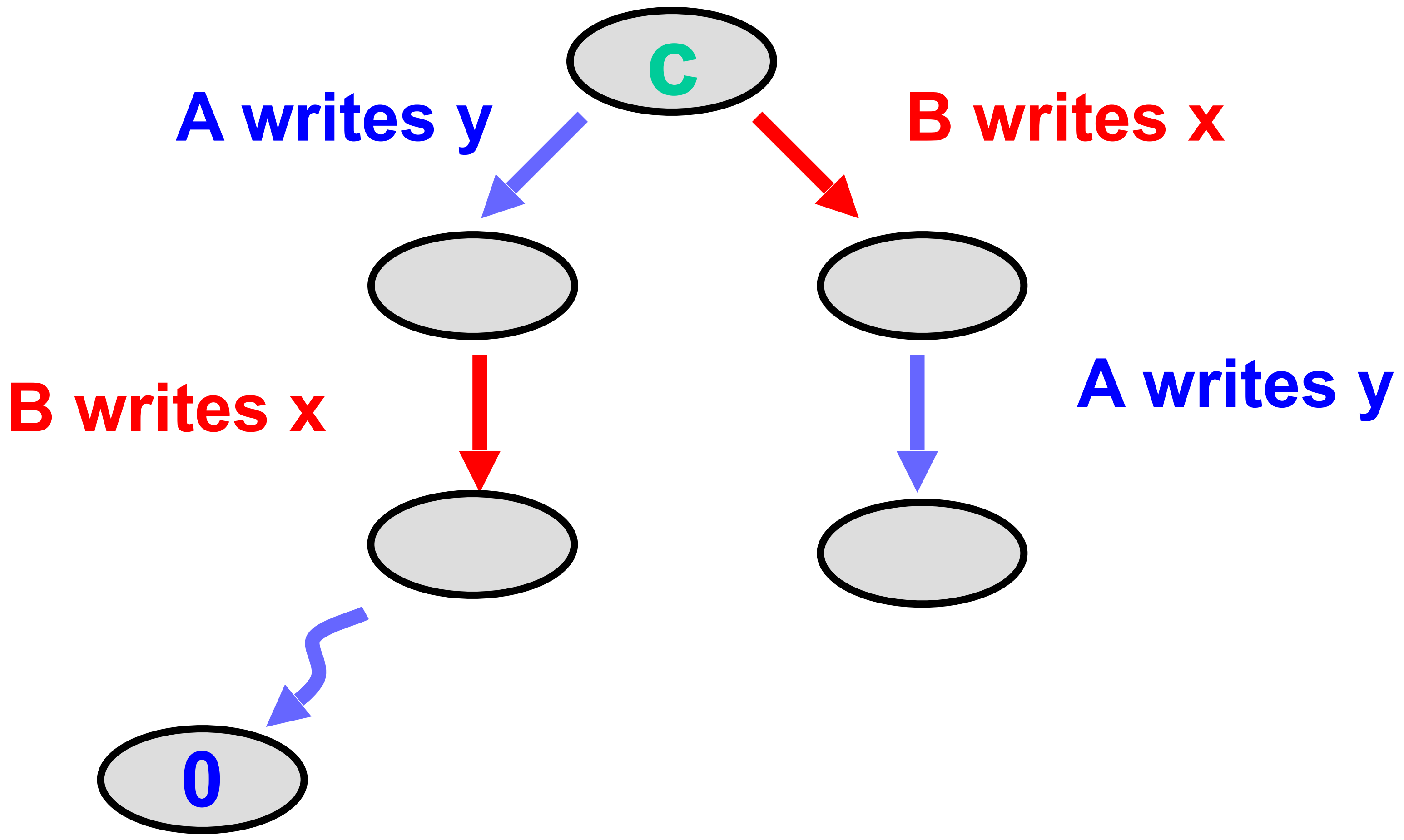
Writing Distinct Registers



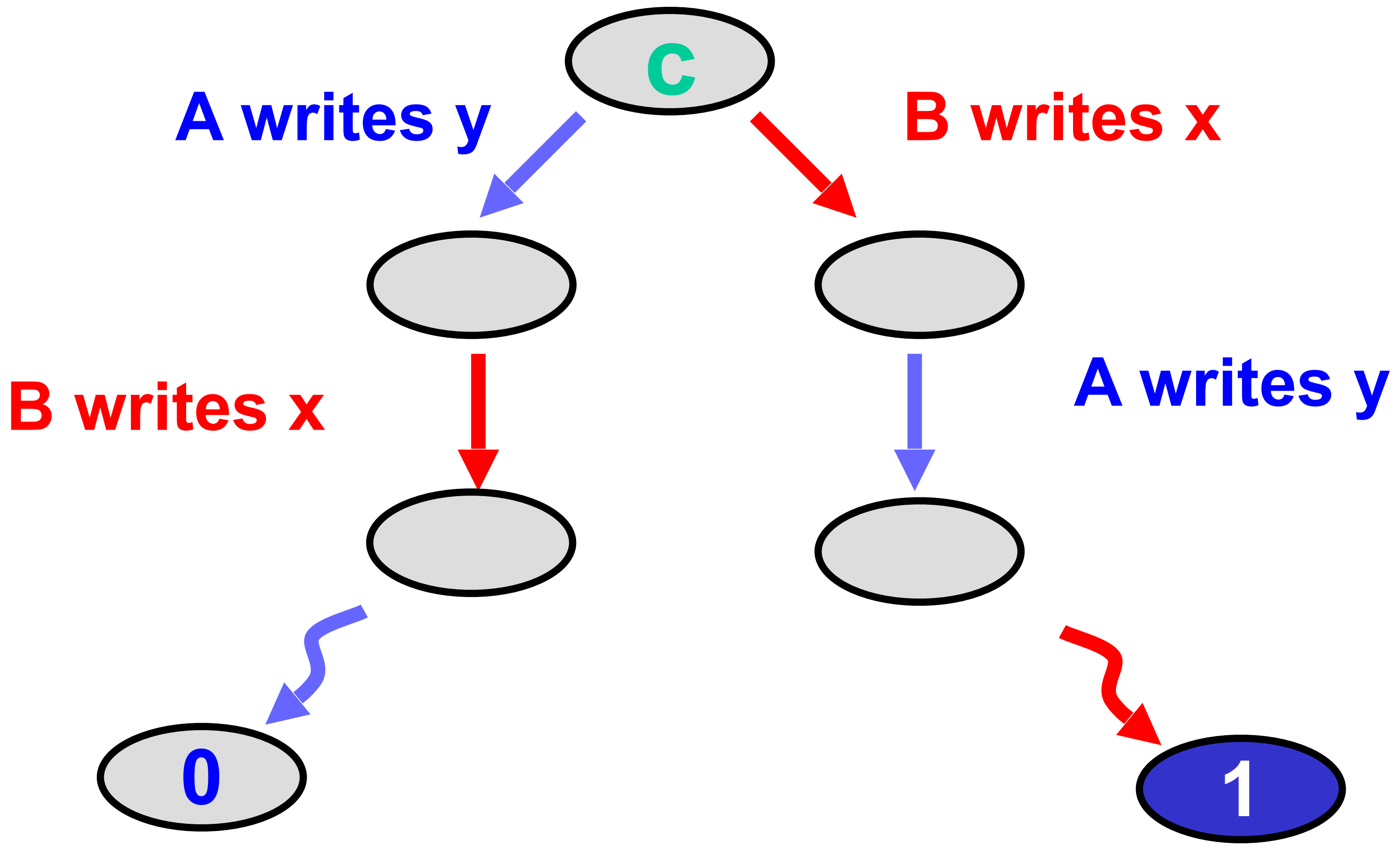
Writing Distinct Registers



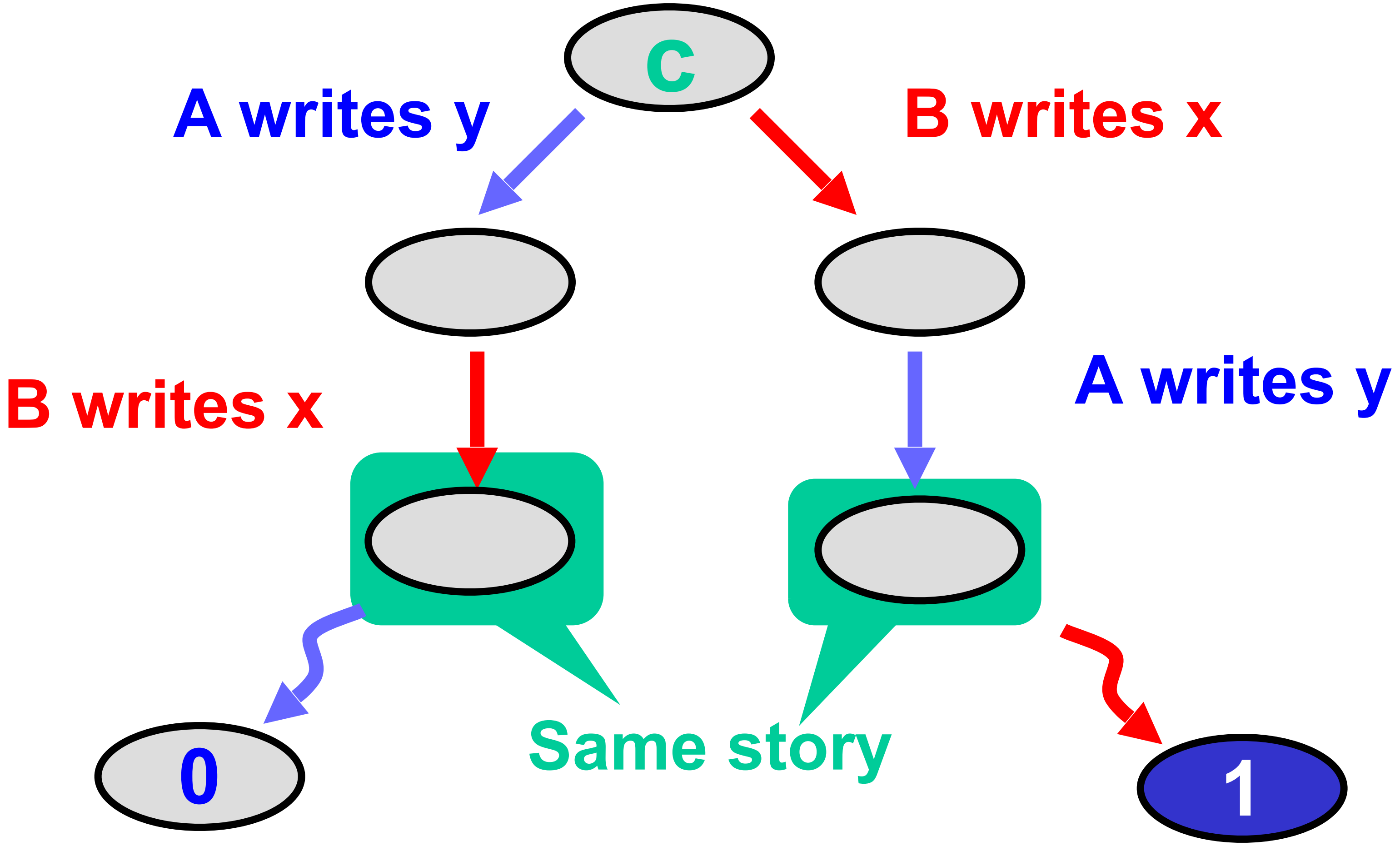
Writing Distinct Registers



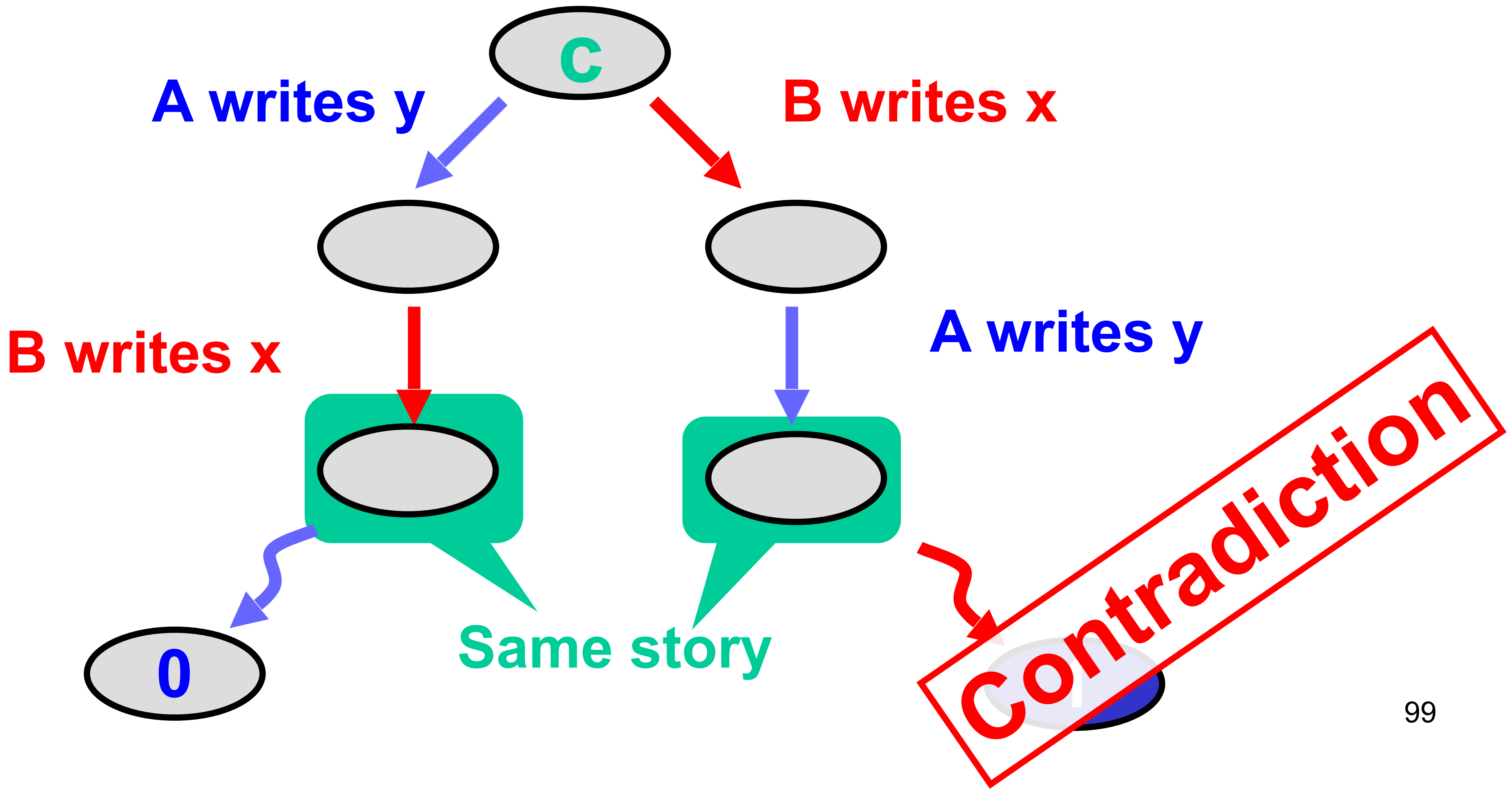
Writing Distinct Registers



Writing Distinct Registers



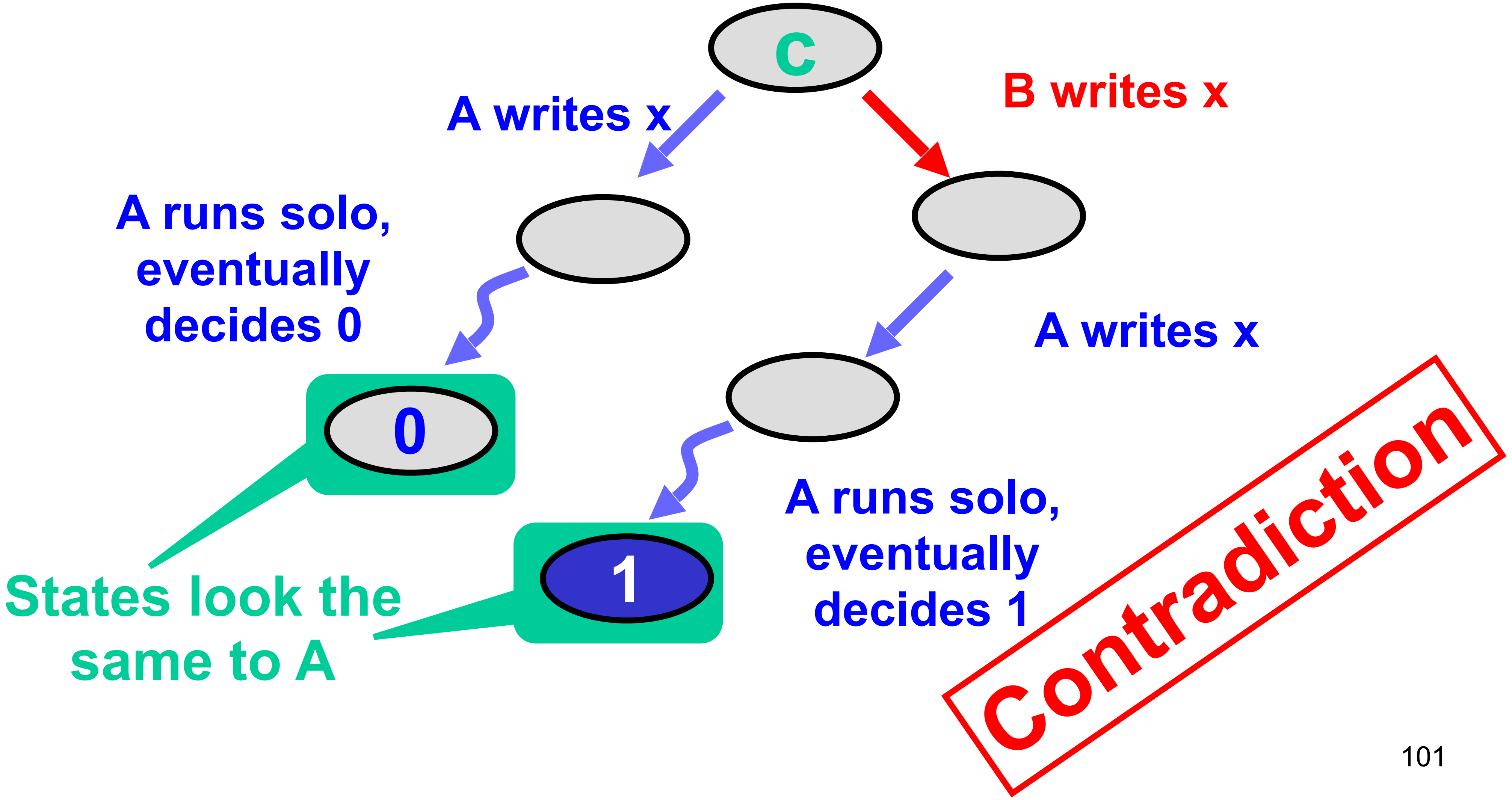
Writing Distinct Registers



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


c
A writes x
B writes x
A runs solo, eventually decides 0
A writes x
0
A runs solo, eventually decides 1
1
States look the same to A
Contradiction

That’s All, Folks!

	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	no
y.write()	no	no	no	no

QED

To Take Away

- Read/Write registers allow for *wait-free* implementations of *some* concurrent objects (yay!)
- Using RW registers, can implement atomic snapshots…
- … but *cannot* implement wait-free mutual exclusion!
- … or, in general any wait-free consensus.
- What should we do?

Let's stop here for today

Next lecture:

better primitives for wait-free synchronisation