

# The Relative Power of Synchronization Operations

Companion slides for  
The Art of Multiprocessor  
Programming  
by Maurice Herlihy & Nir Shavit

# Last Lecture: Shared-Memory Computability



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

# Wait-Free Implementation

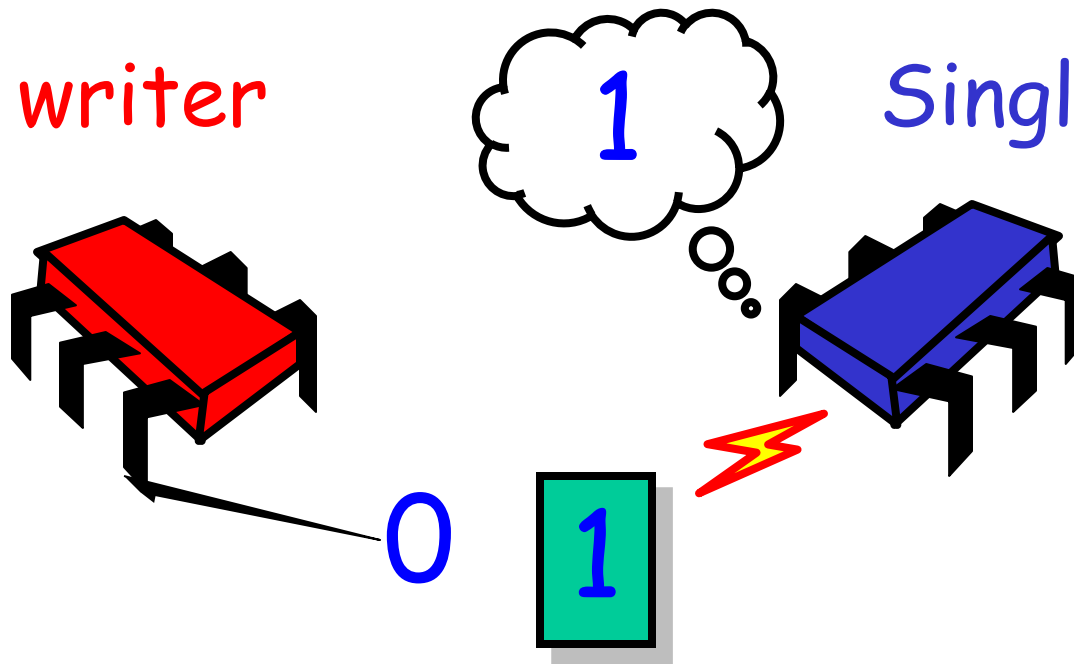
- Every method call completes in finite number of steps
- Implies no mutual exclusion



# From Weakest Register

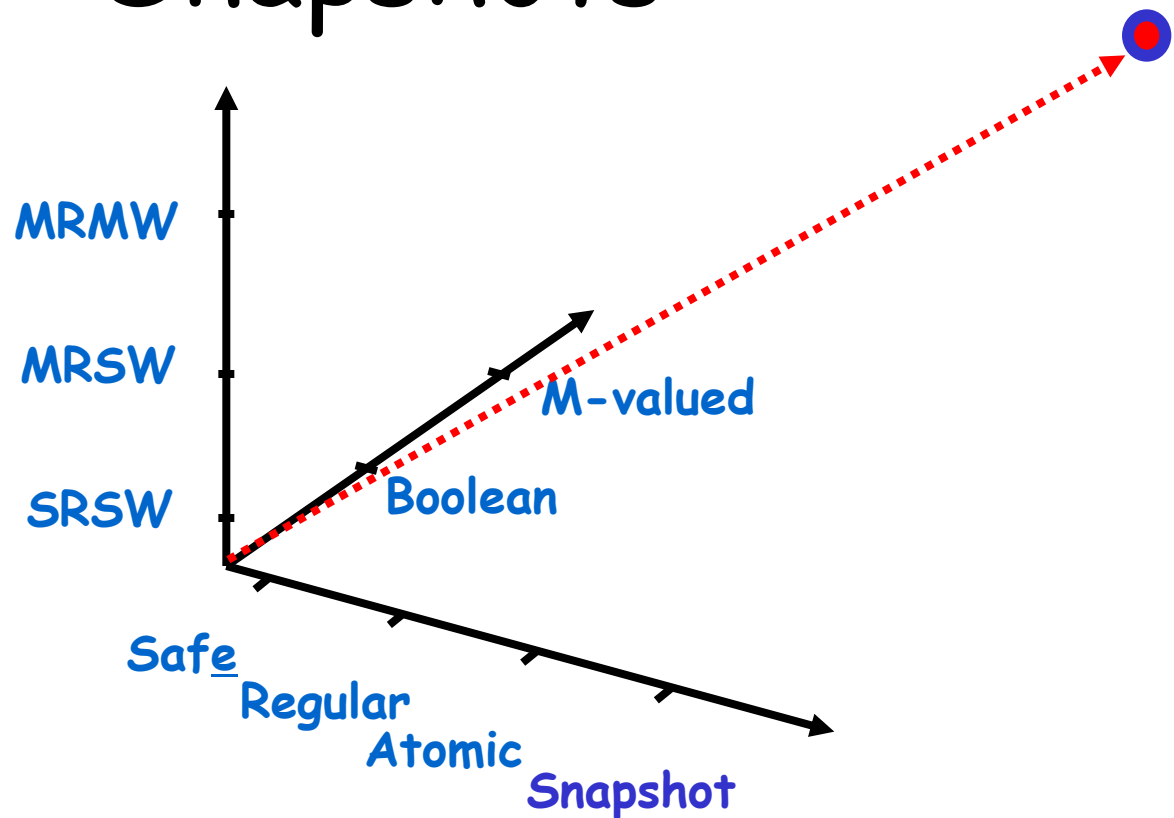
Single writer

Single reader



Safe Boolean register

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



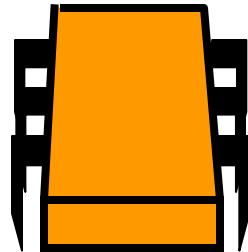
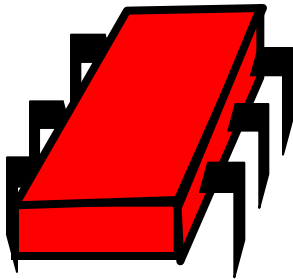
# Rationale for wait-freedom

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

Why is Mutual Exclusion  
so wrong?

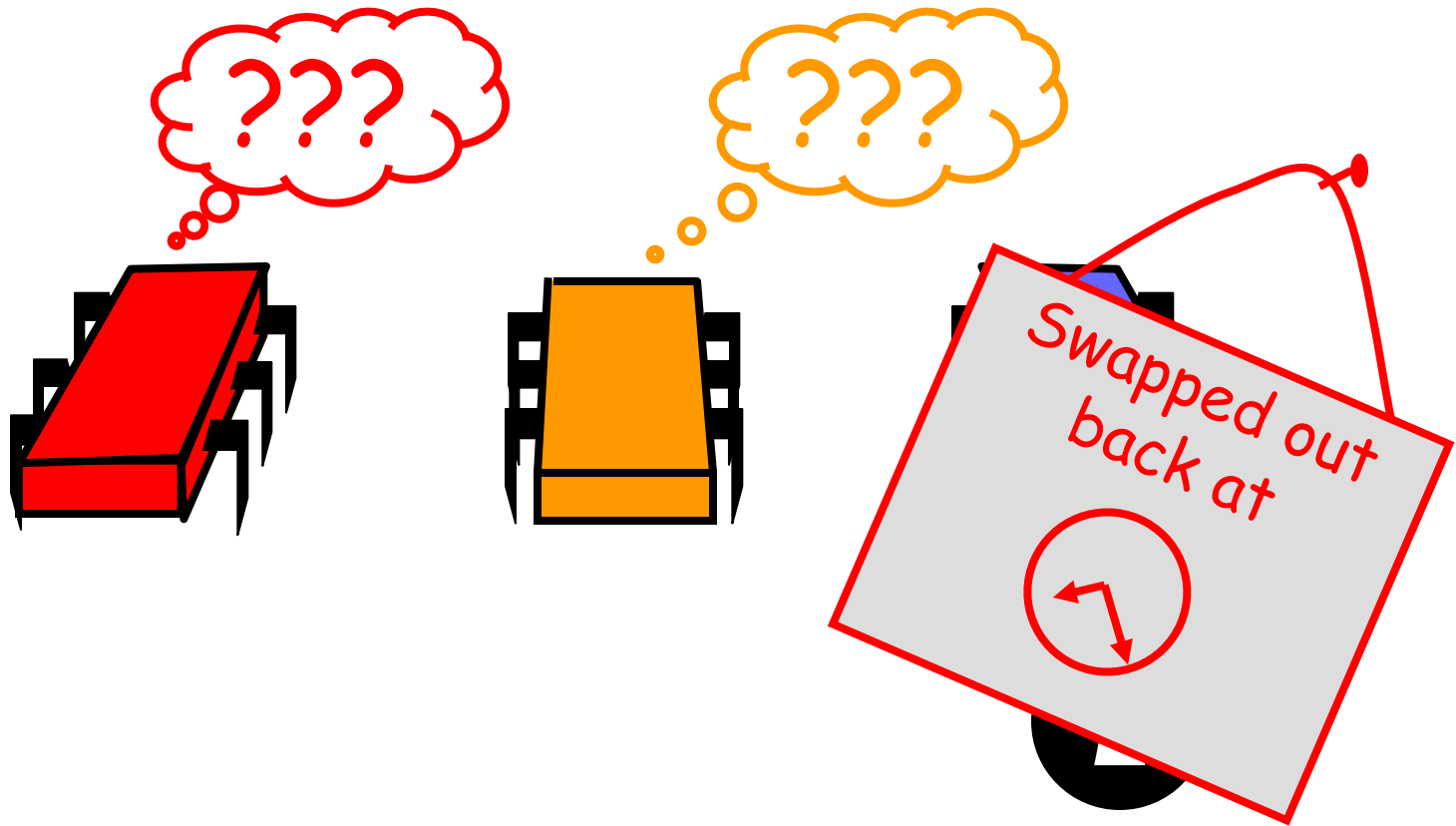


# Asynchronous Interrupts

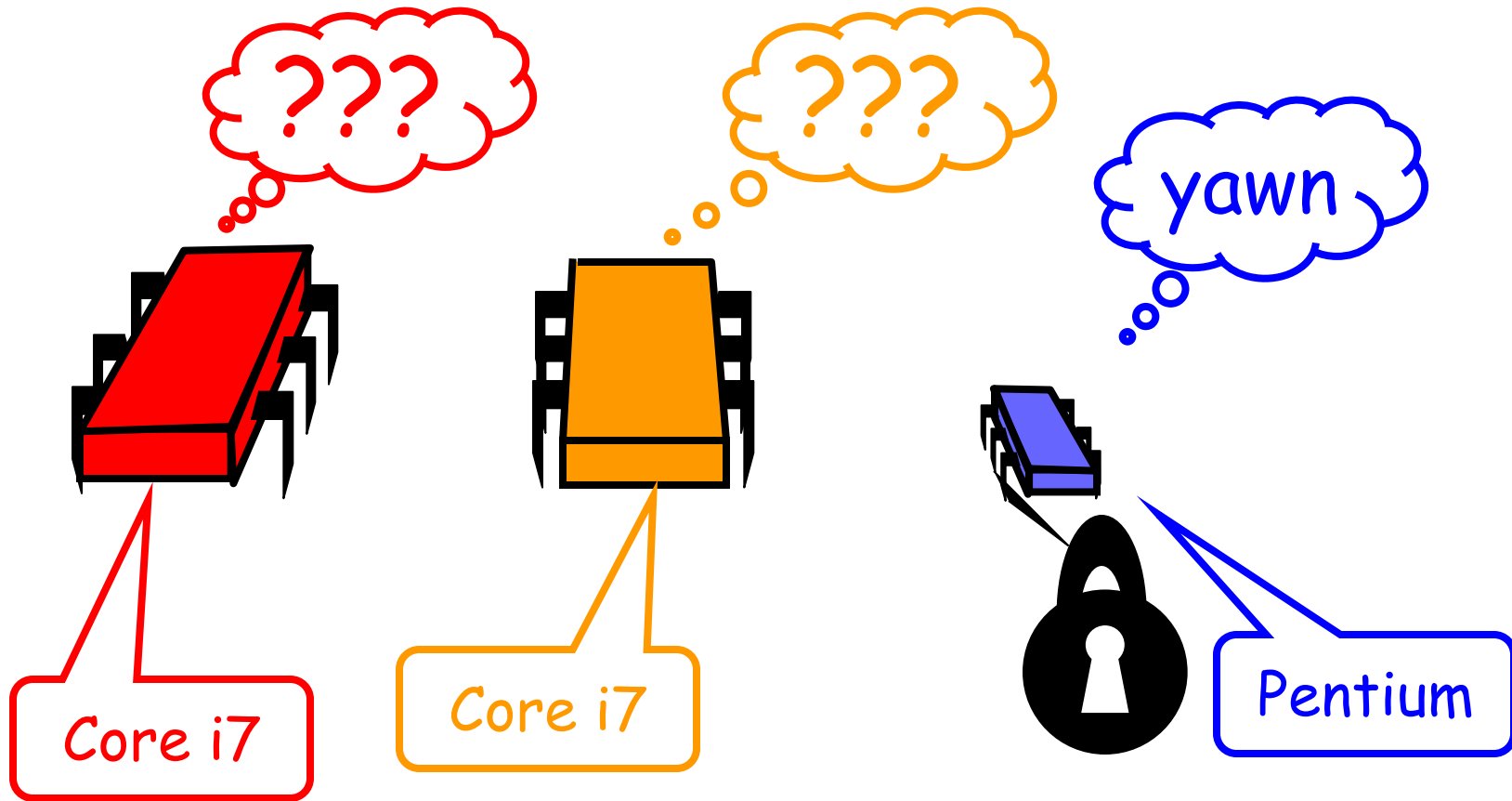




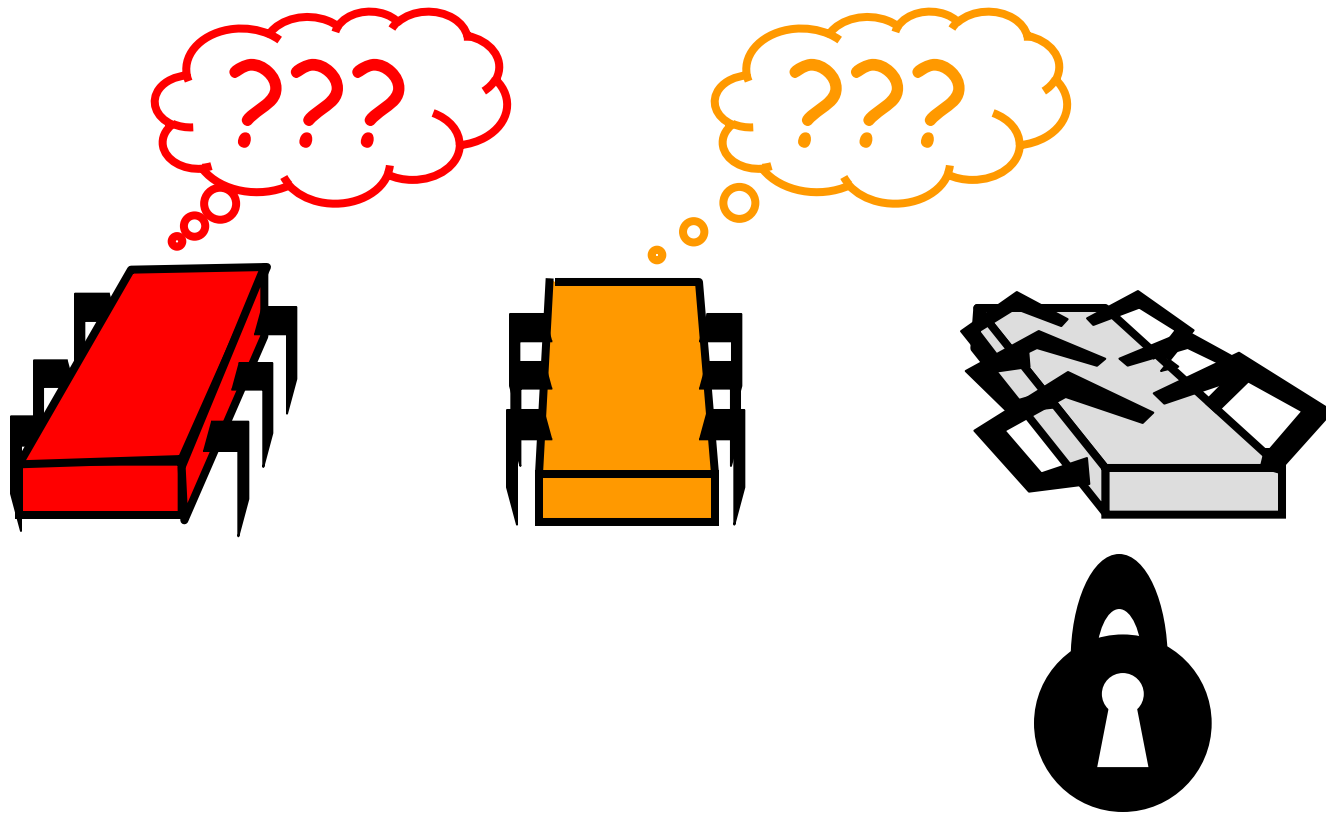
# Asynchronous Interrupts



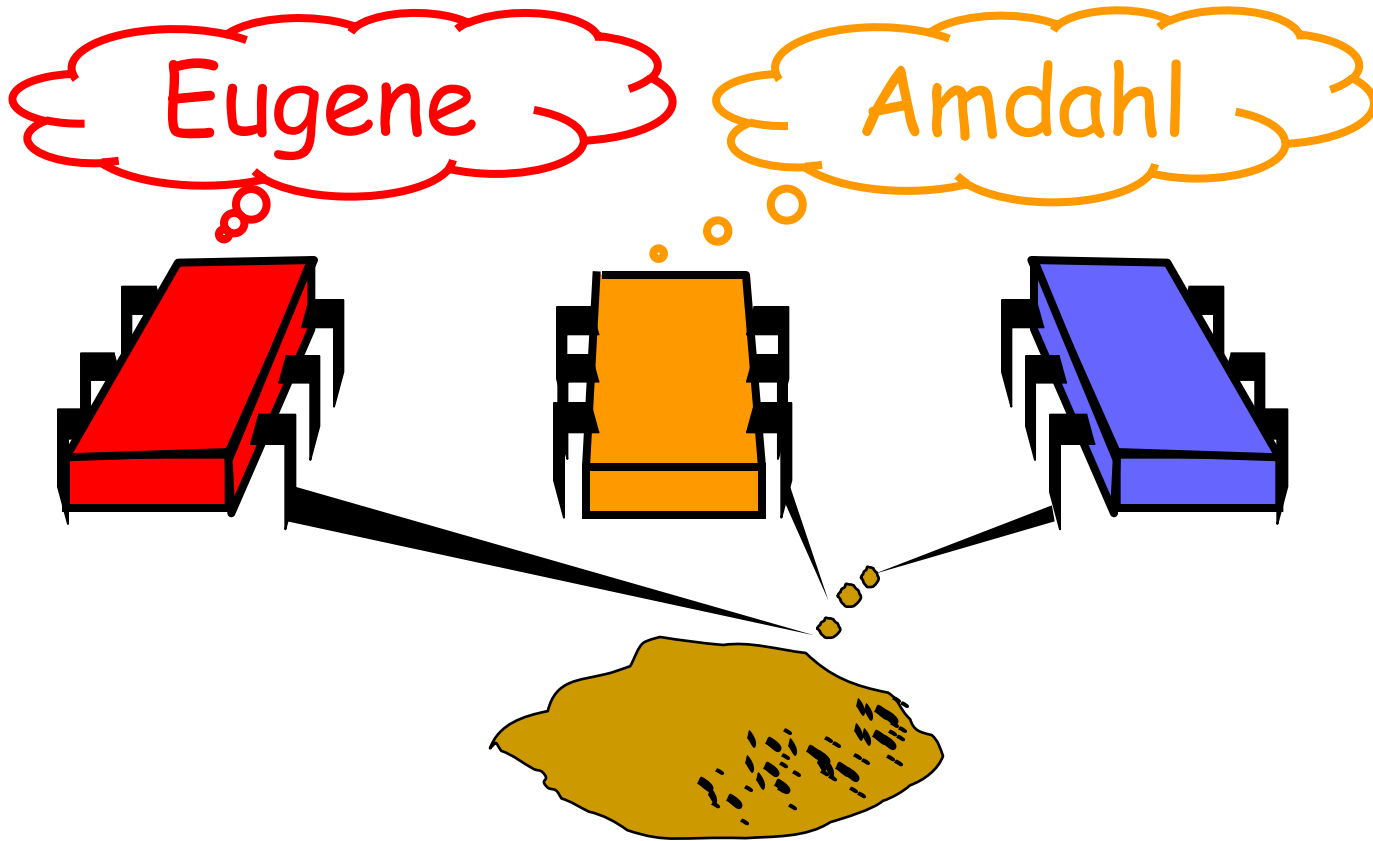
# Heterogeneous Processors



# Fault-tolerance



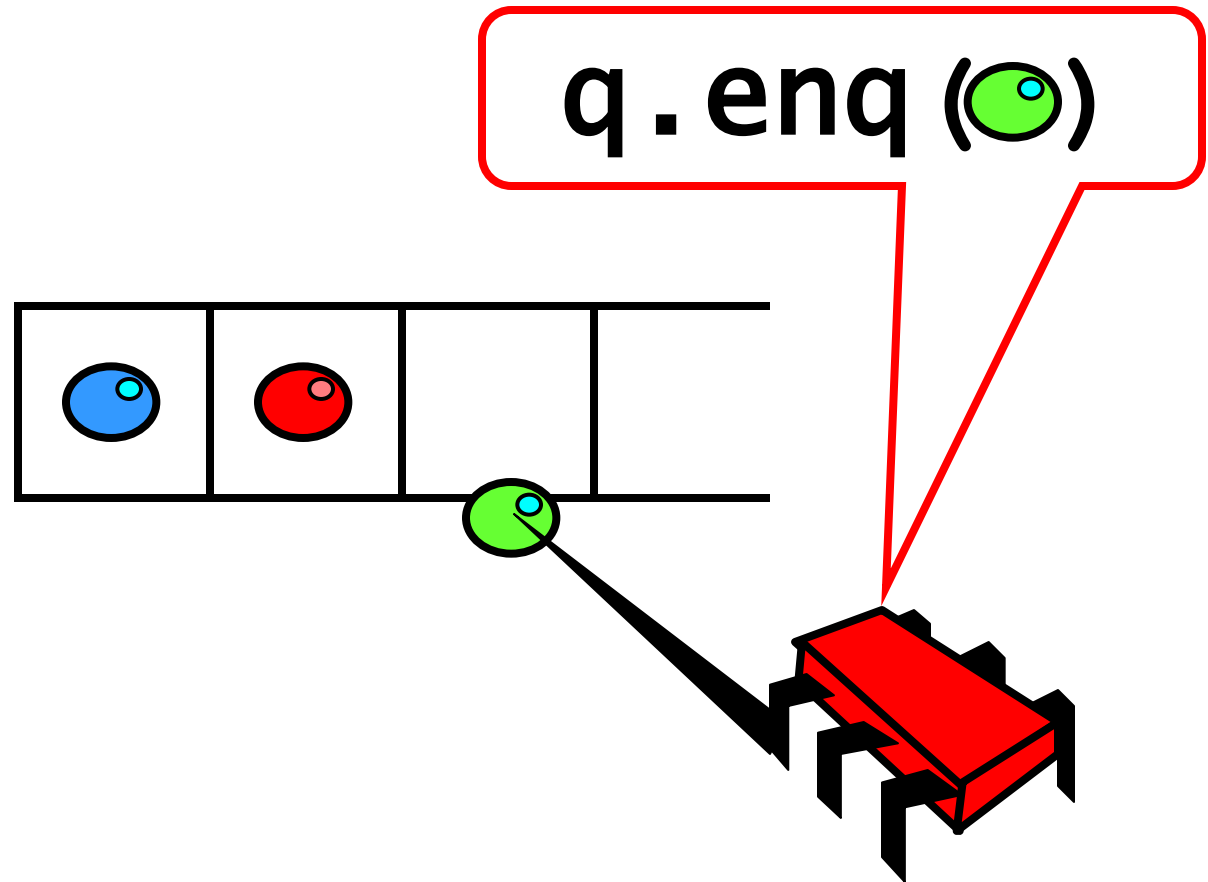
# Machine Level Instruction Granularity



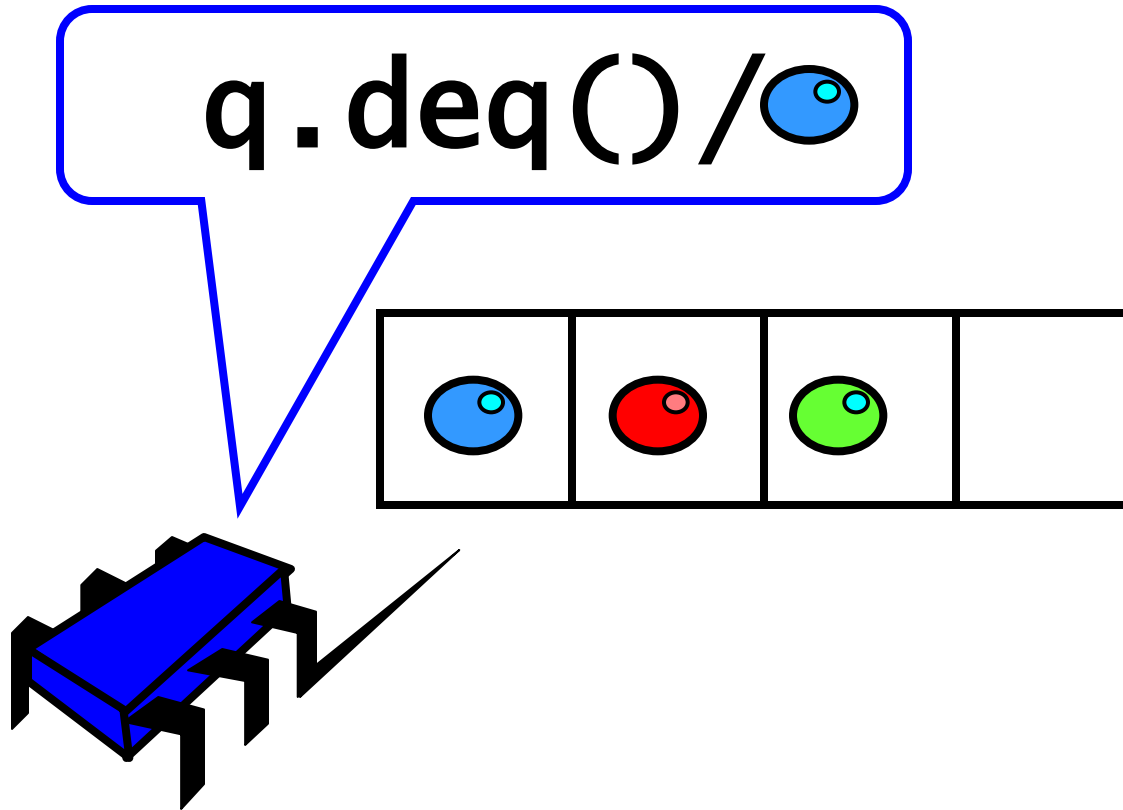
# Basic Questions

- 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

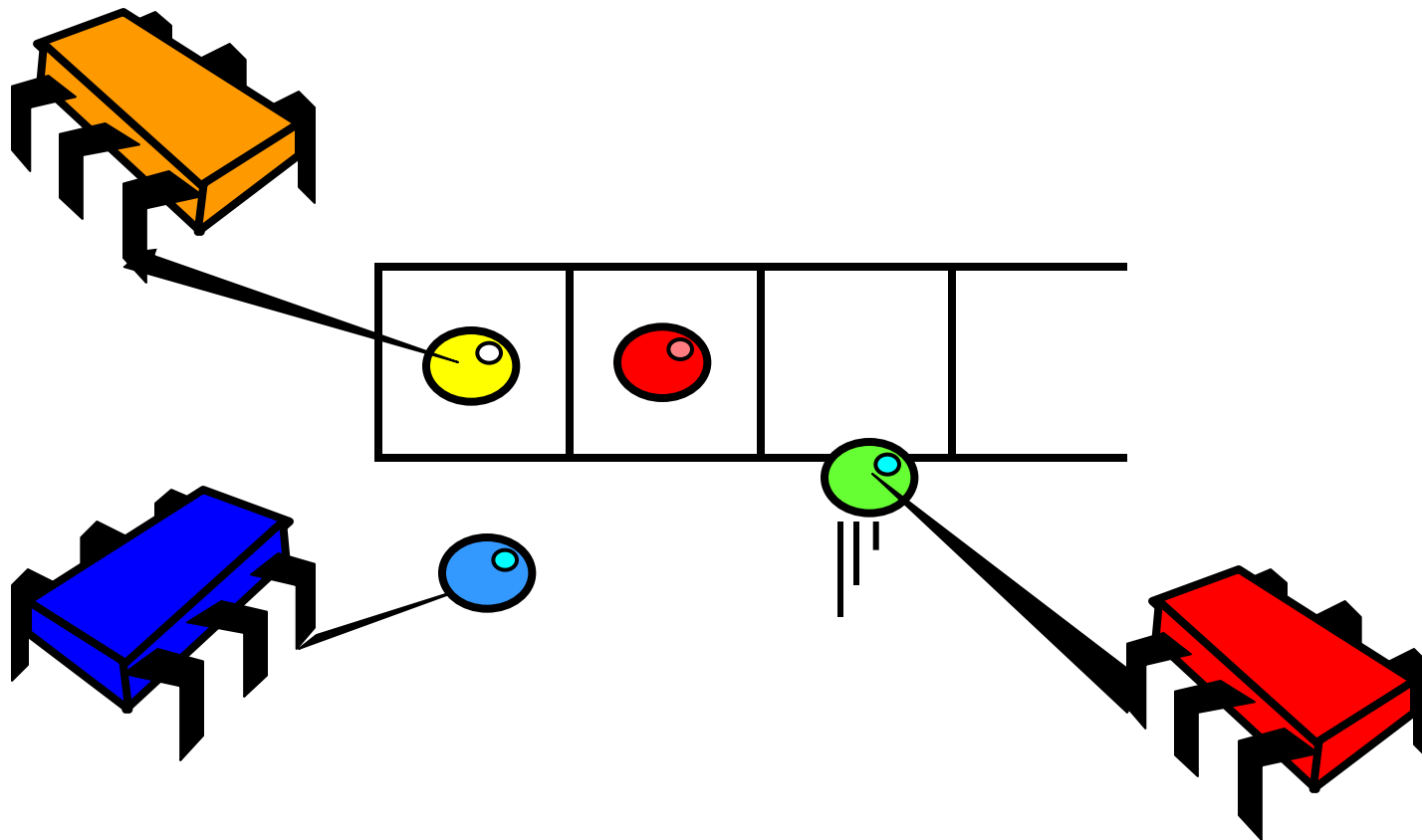


# Two-Thread Wait-Free Queue

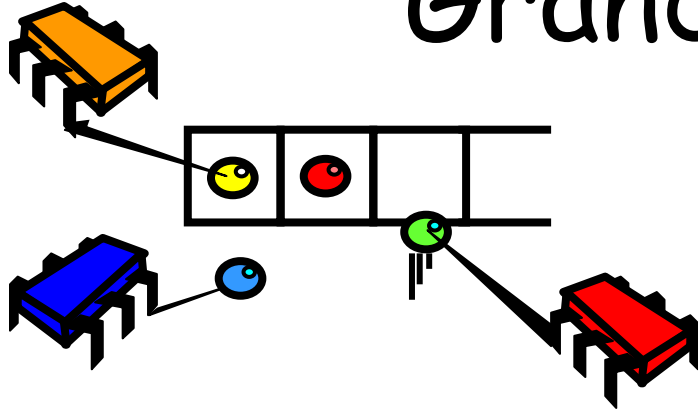
```
public class LockFreeQueue {
    int head = 0, tail = 0;
    Item[QSIZE] items;
    public void enq(Item x) {
        while (tail-head == QSIZE) {};
        items[tail % QSIZE] = x; tail++;
    }
    public Item deq() {
        while (tail-head == 0) {}
        Item item = items[head % QSIZE];
        head++; return item;
    }
}
```



# What About Multiple Dequeueers?



# Grand Challenge



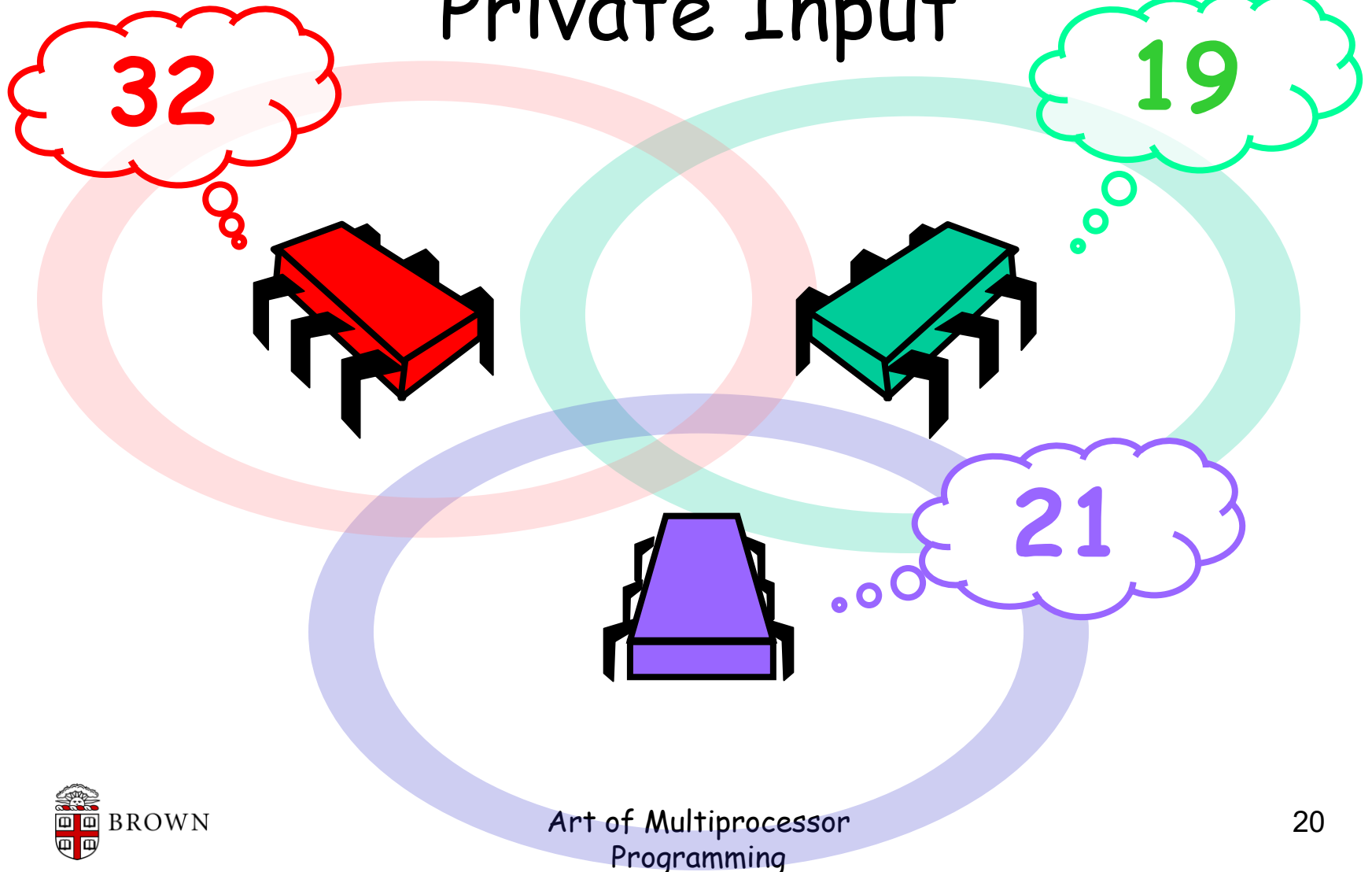
Only new  
aspect

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

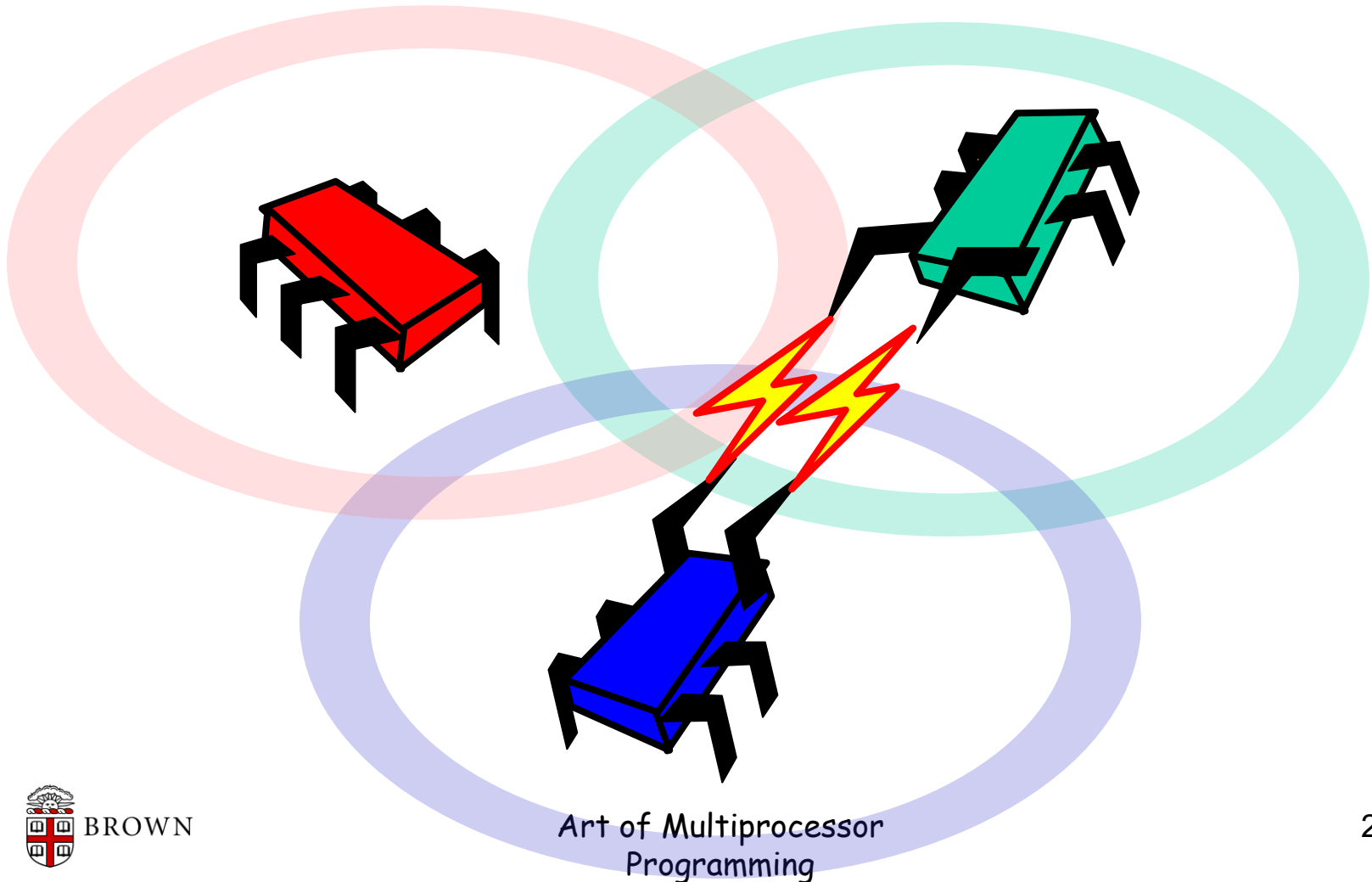
# Consensus

- While you are ruminating on the grand challenge...
- We will give you another puzzle
  - Consensus
  - Will be important ...

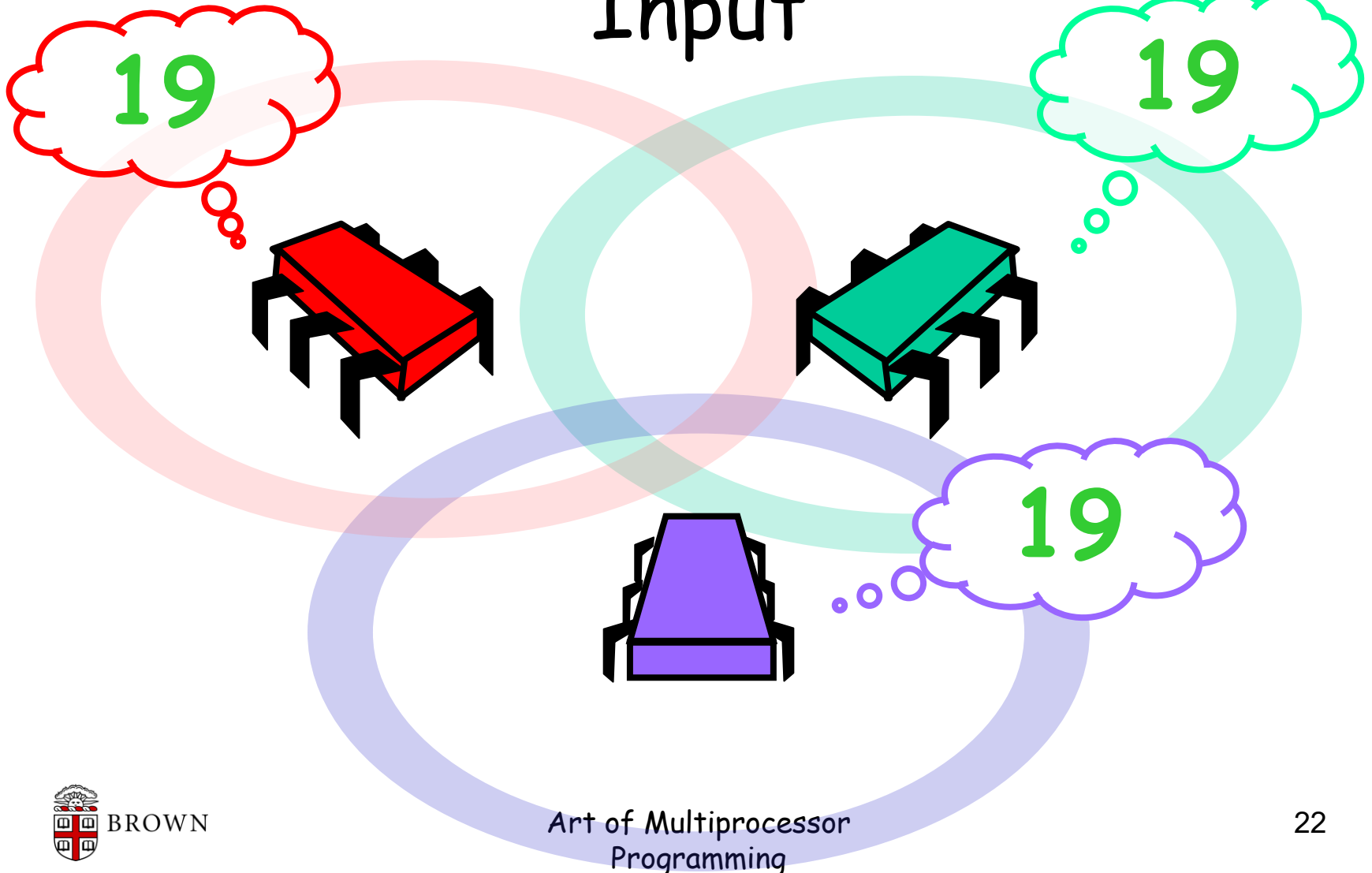
# Consensus: Each Thread has a Private Input



# They Communicate



# They Agree on One Thread's Input

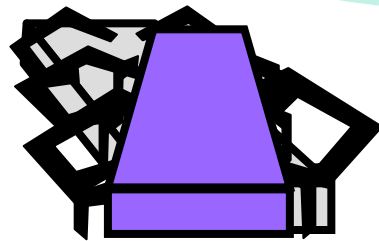
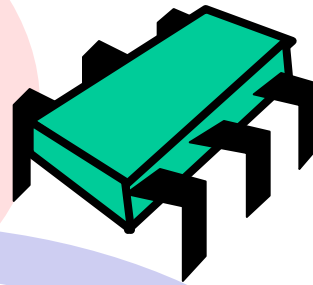
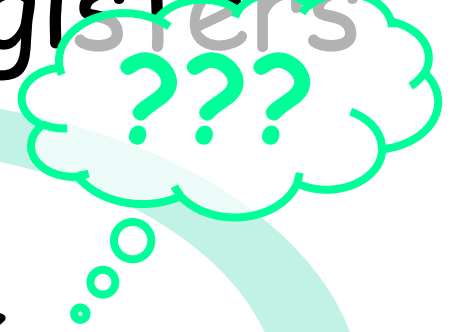
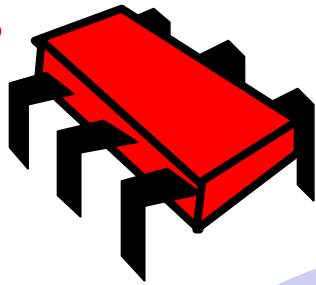


# Formally: Consensus

Consistent: all threads decide the same value

Valid: the common decision value is some thread's input

# No Wait-Free Implementation of Consensus using Registers





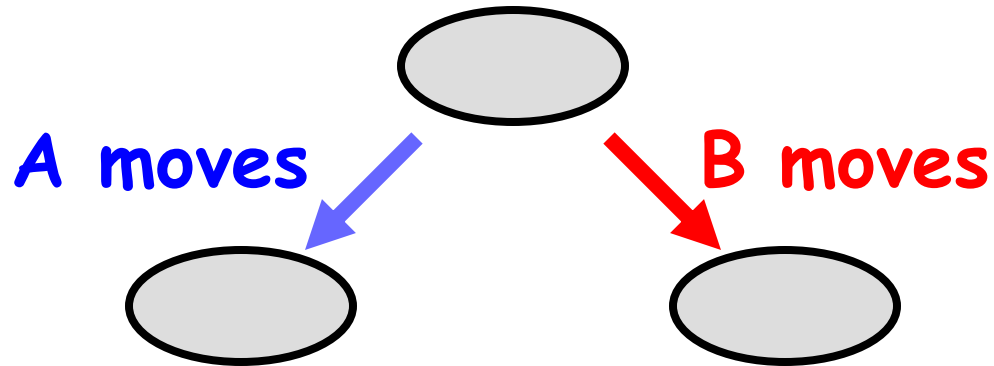
# Formally

- Theorem [adapted from Fischer, Lynch, Paterson]: There is no wait-free implementation of  $n$ -thread consensus,  $n > 1$ , from read-write registers
- Implication: asynchronous computability fundamentally different from Turing computability

# Proof Strategy

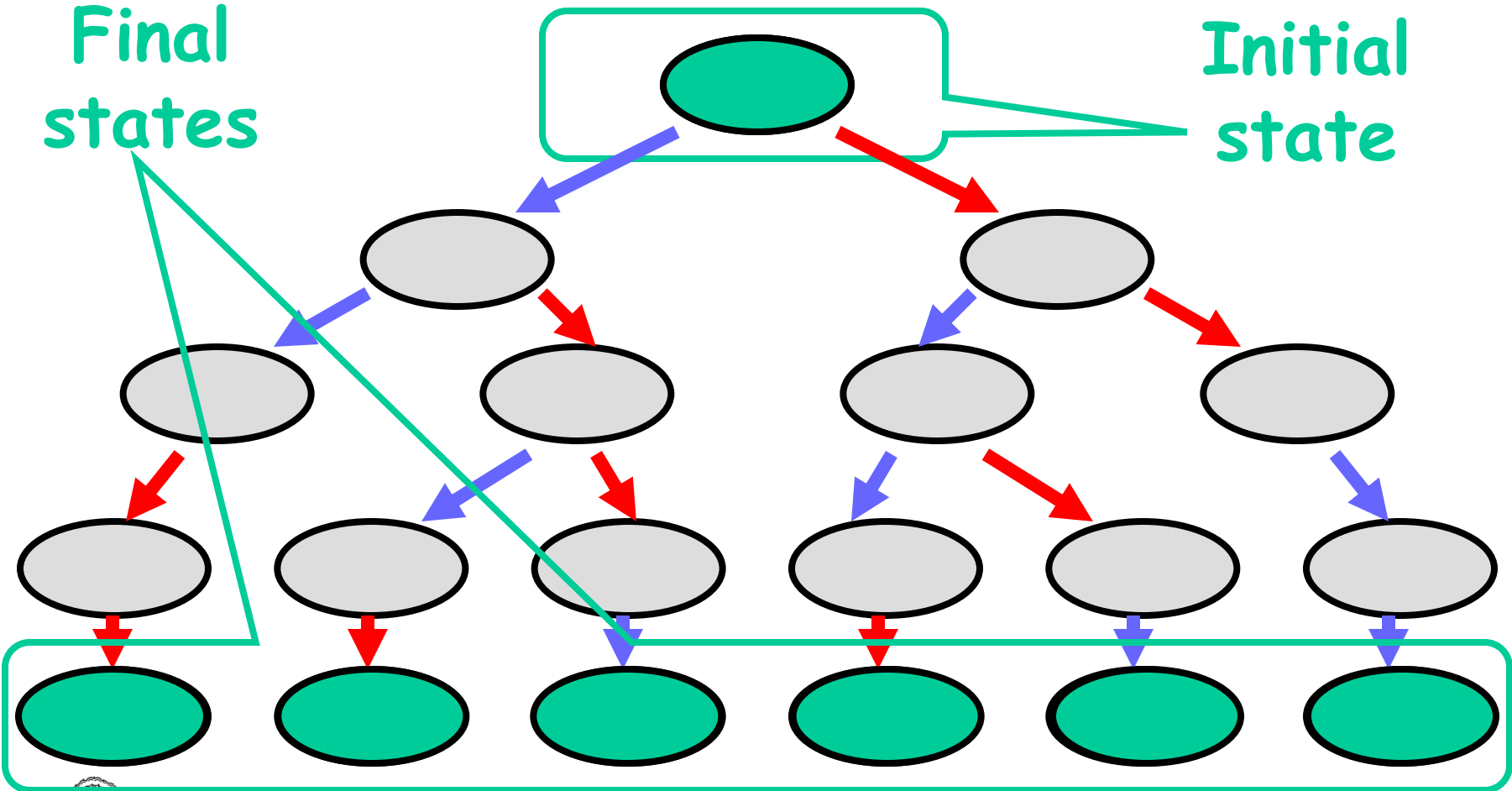
- Assume otherwise
- Reason about the properties of any such protocol
- Derive a contradiction
- Quod Erat Demonstrandum
- Suffices to prove for binary consensus and  $n=2$

# Wait-Free Computation

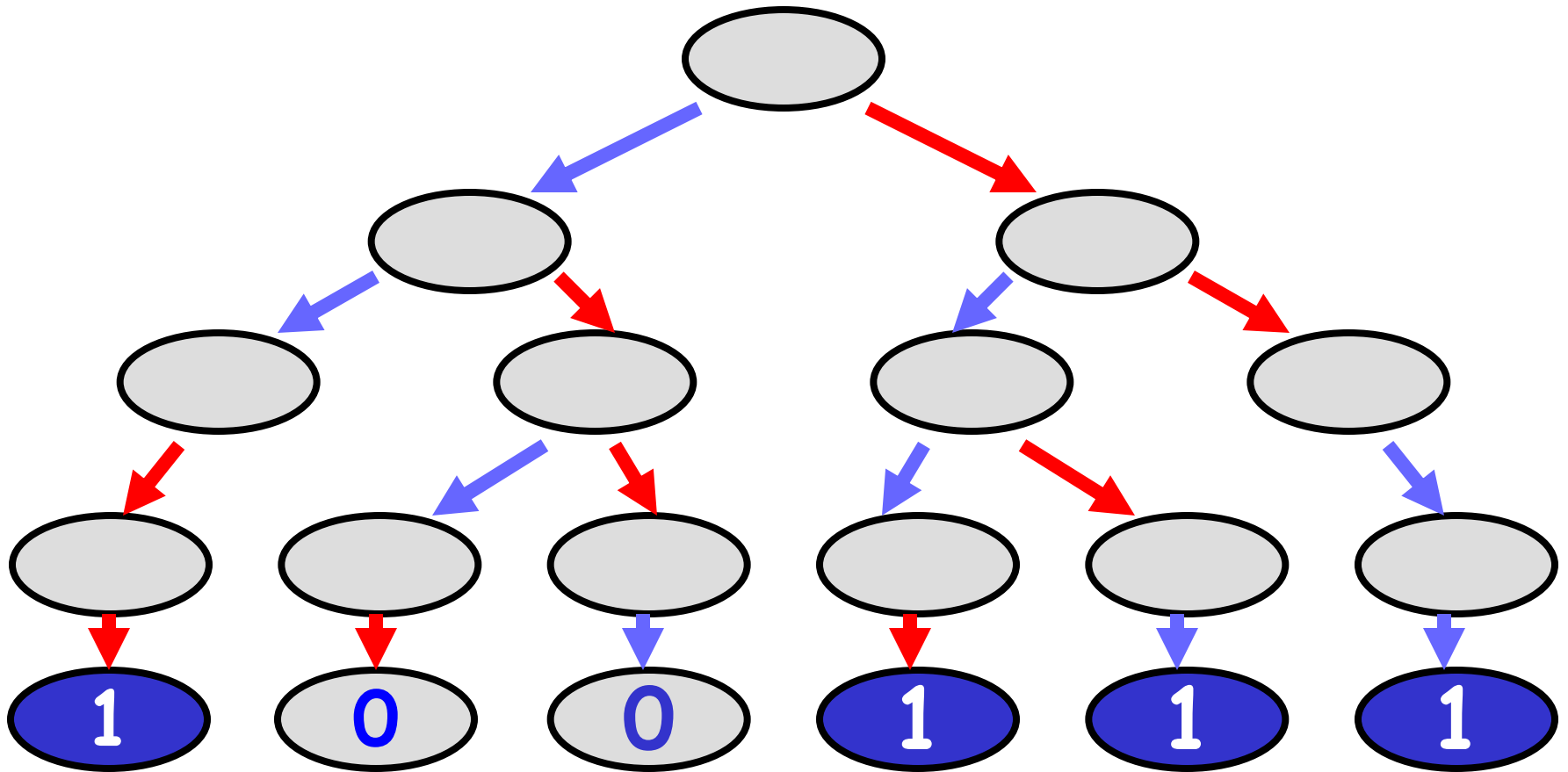


- Either **A** or **B** "moves"
- Moving means
  - Register read
  - Register write

# The Two-Move Tree

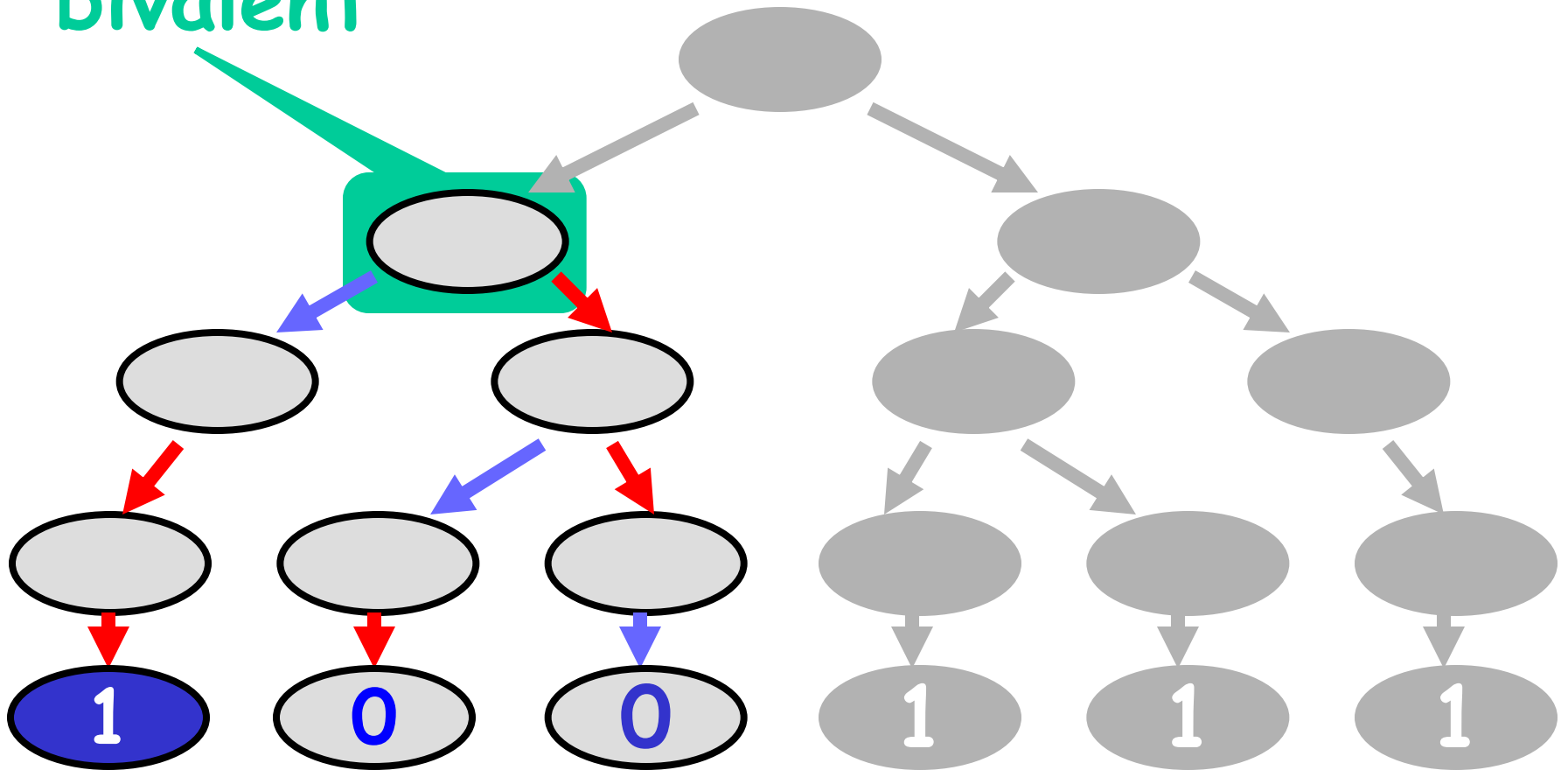


# Decision Values

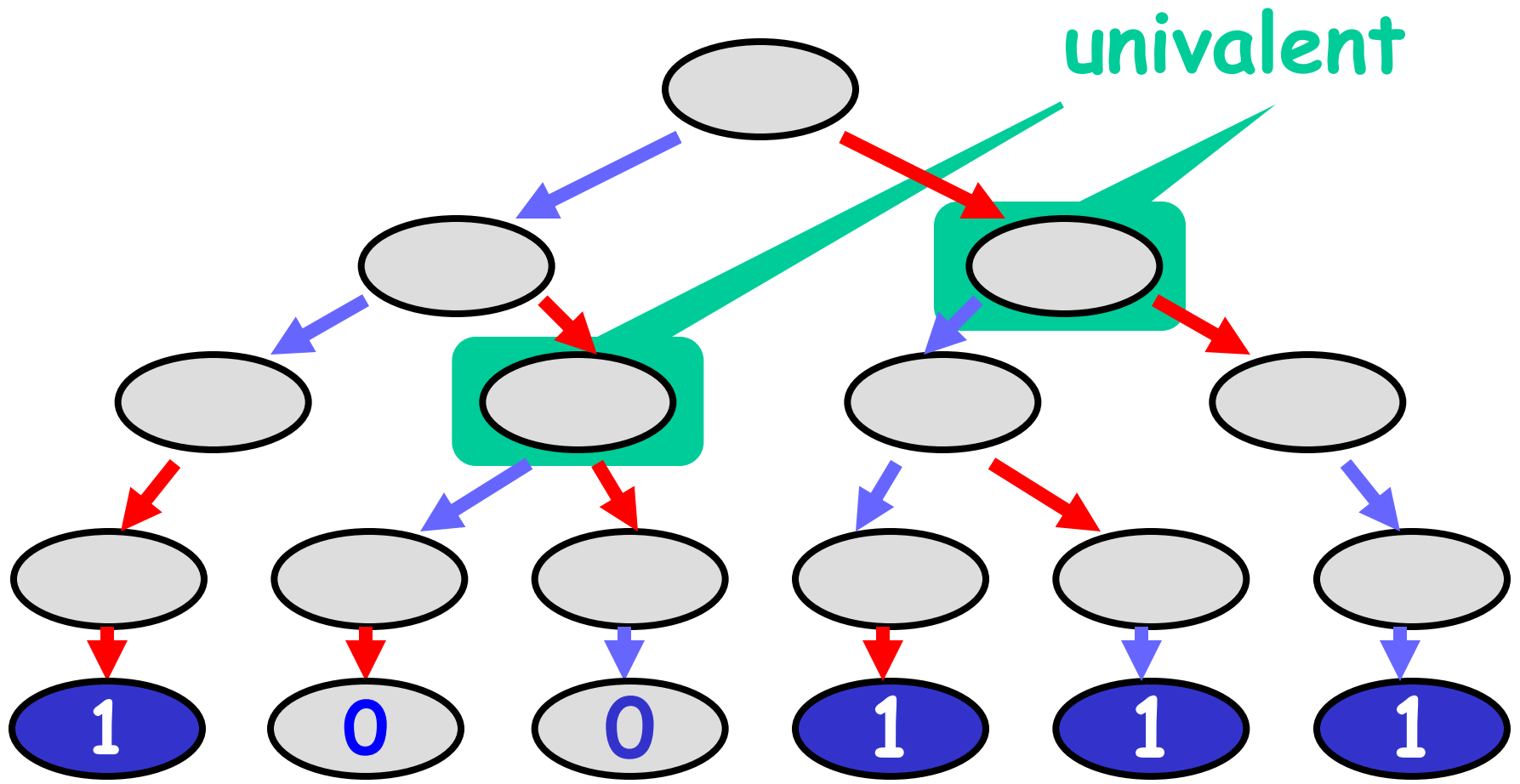


# Bivalent: Both Possible

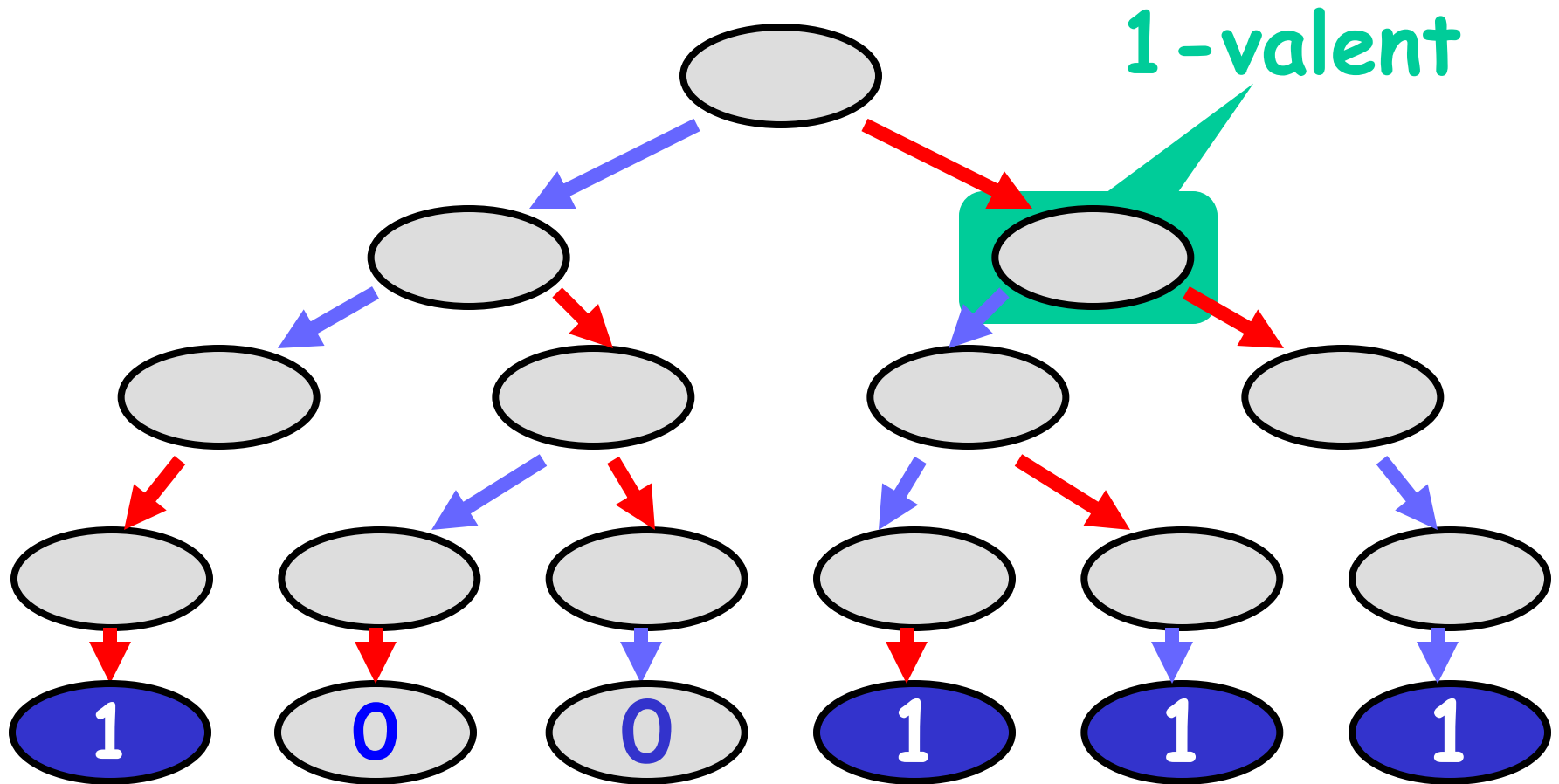
bivalent



# Univalent: Single Value Possible



# x-valent: x Only Possible Decision





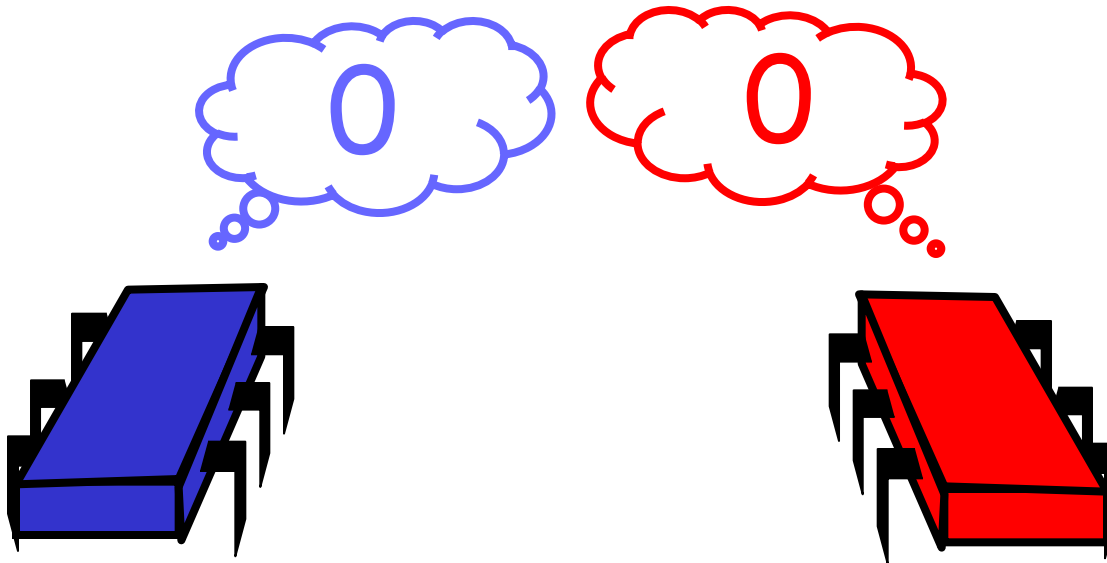
# Summary

- 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
- Outcome depends on
  - Chance
  - Whim of the scheduler
- Multiprocessor gods do play dice
- ...
- Lets prove this claim

# Both Inputs 0



Univalent: all executions must decide 0



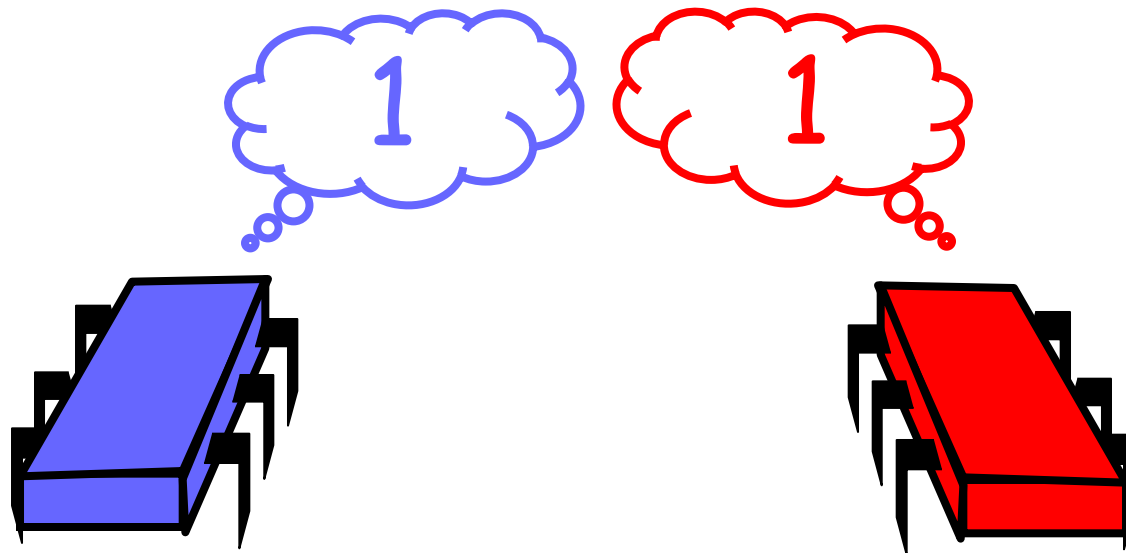
# Both Inputs 0



Including this solo execution by A



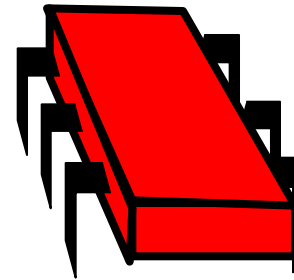
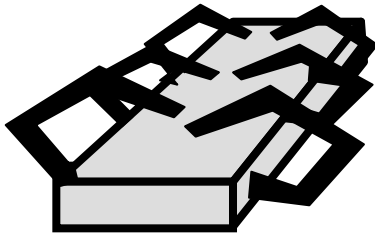
# Both Inputs 1



All executions must decide 1



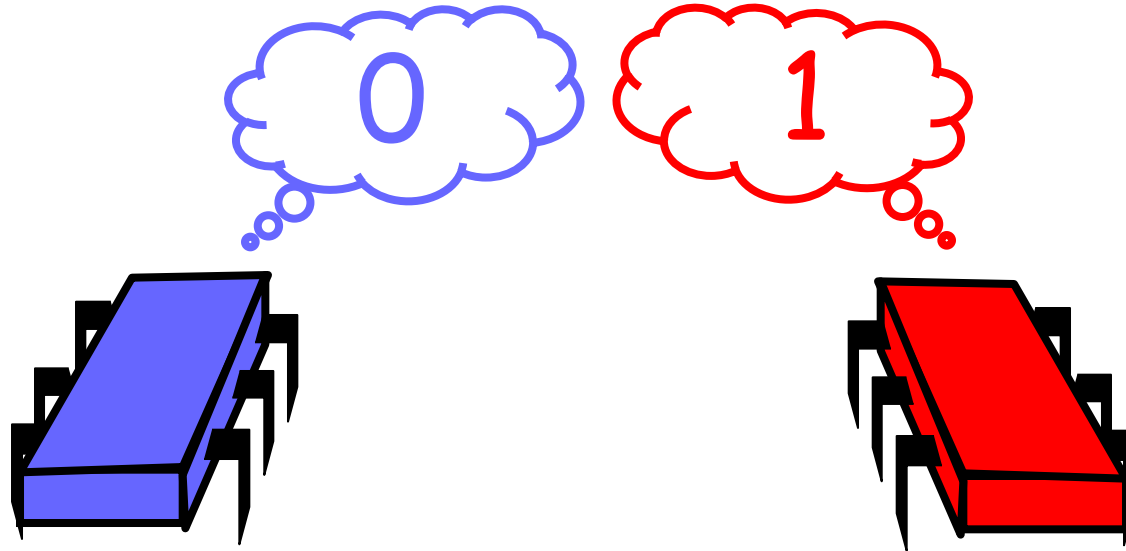
# Both Inputs 1



Including this solo execution by **B**

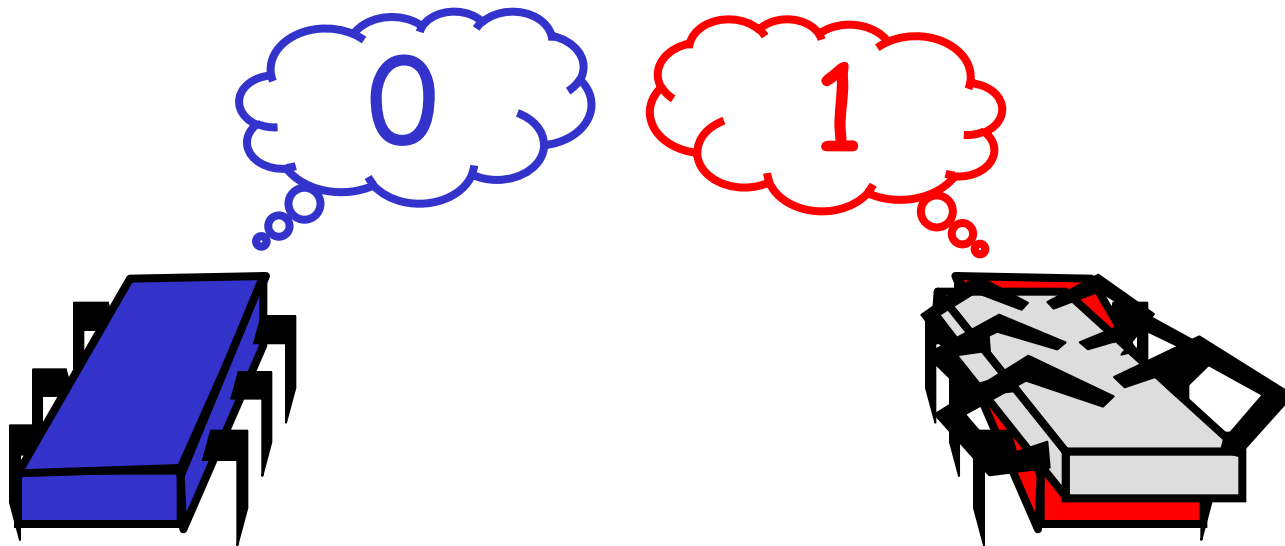


# What if inputs differ?



By Way of contradiction: If univalent  
all executions must decide on same value

# The Possible Executions

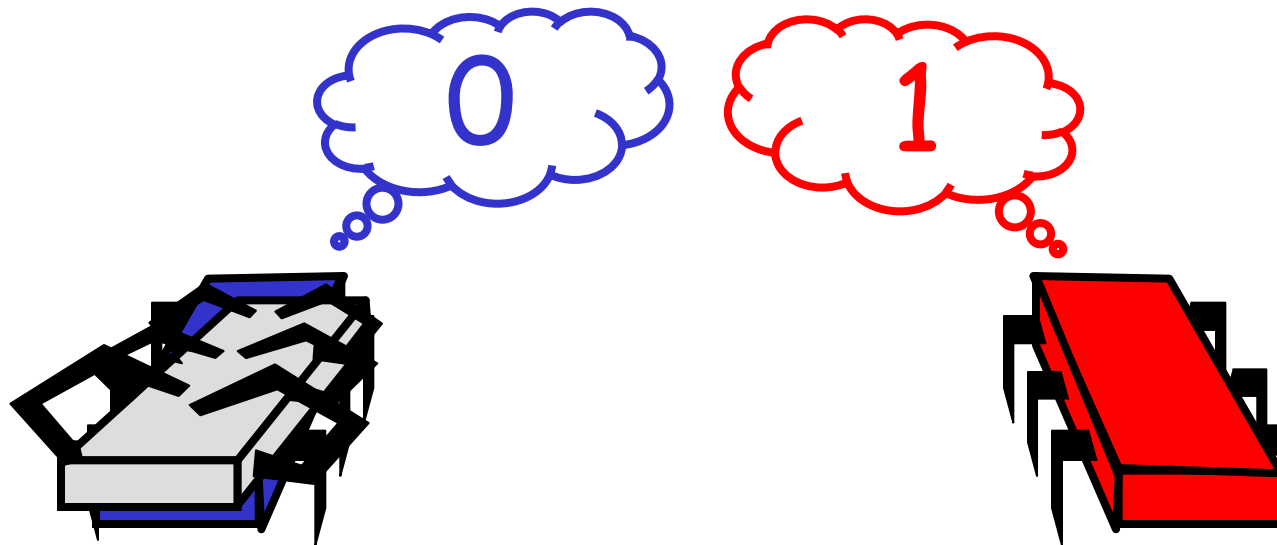


Include the solo execution by A  
that decides 0





# The Possible Executions

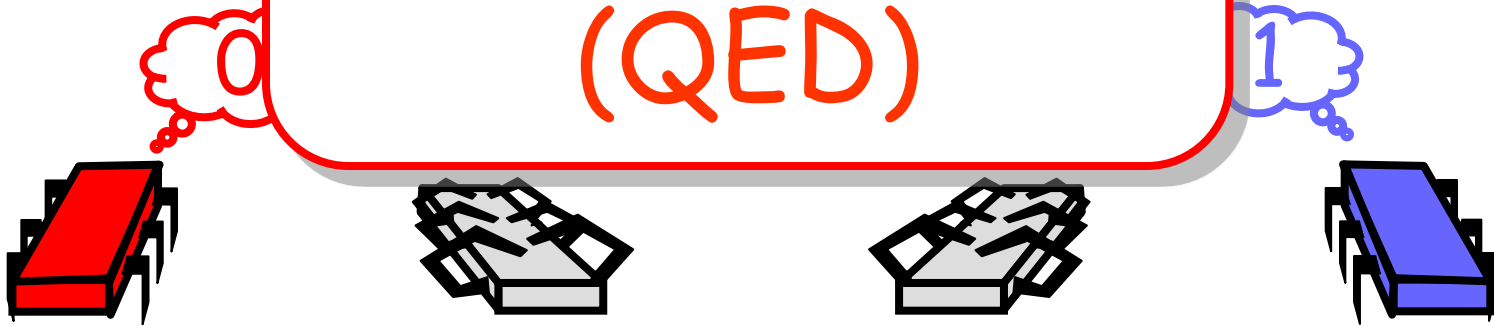


Also include the solo execution by **B**  
which we know decides 1



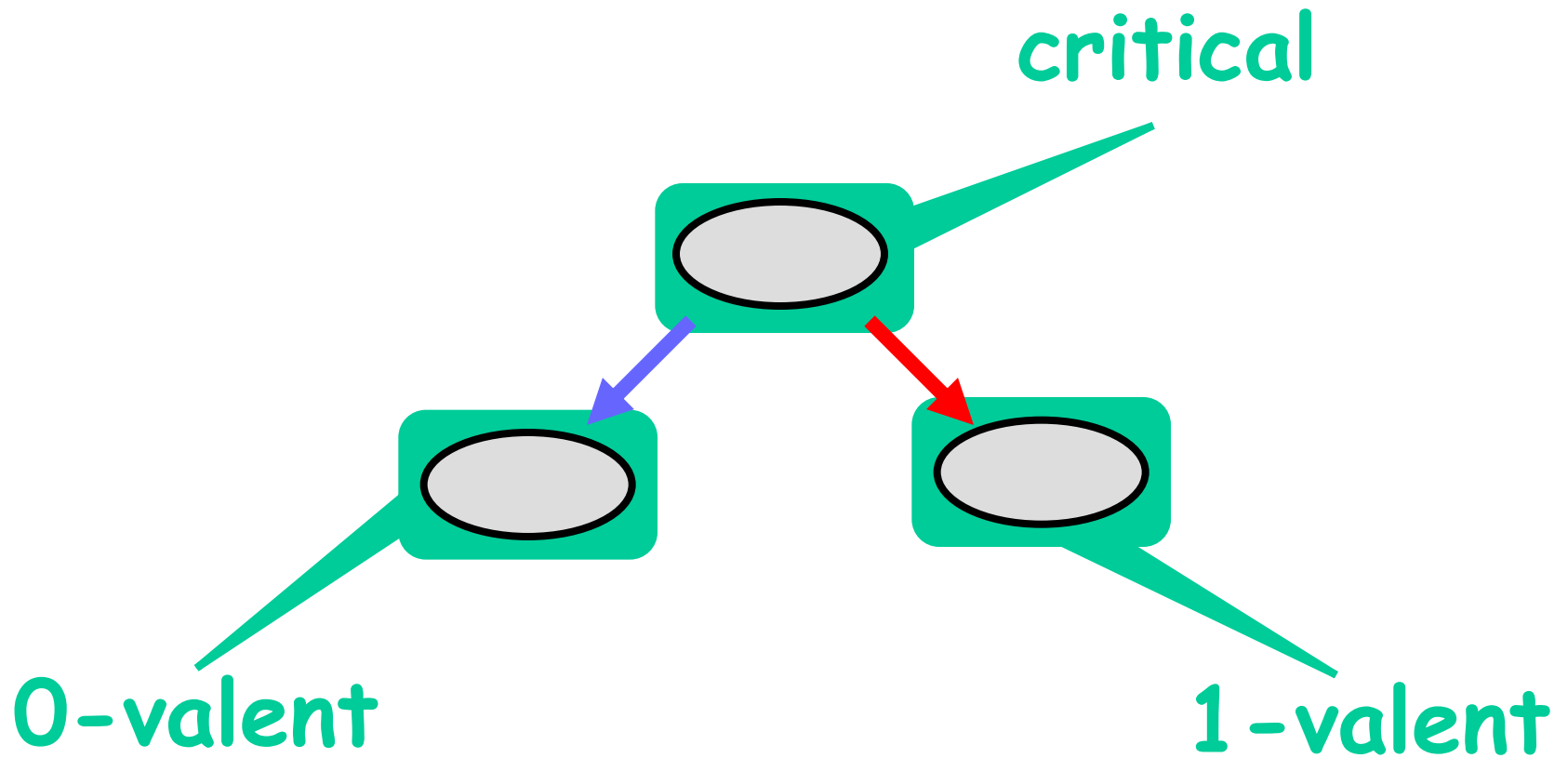
# Possible Executions Include

How univalent  
is that?  
(QED)

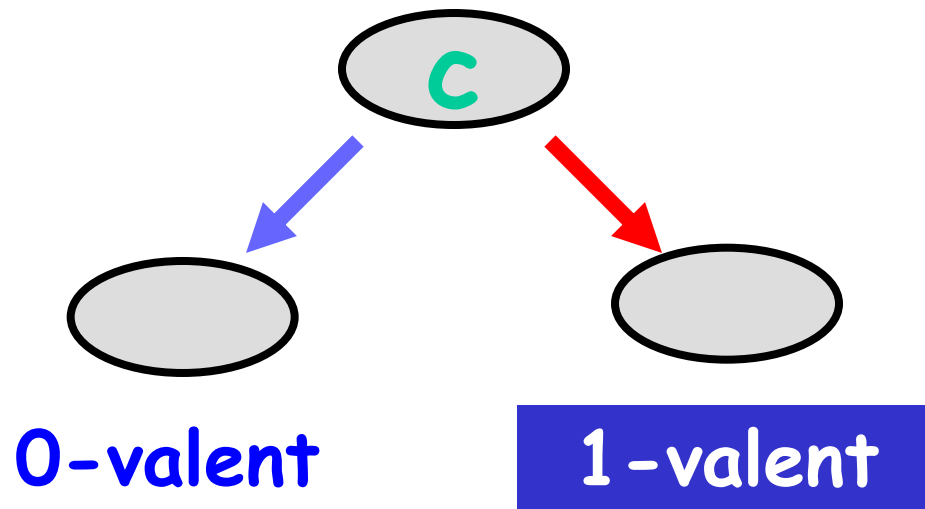


- Solo execution by **A** must decide **0**
- Solo execution by **B** must decide **1**

# Critical States



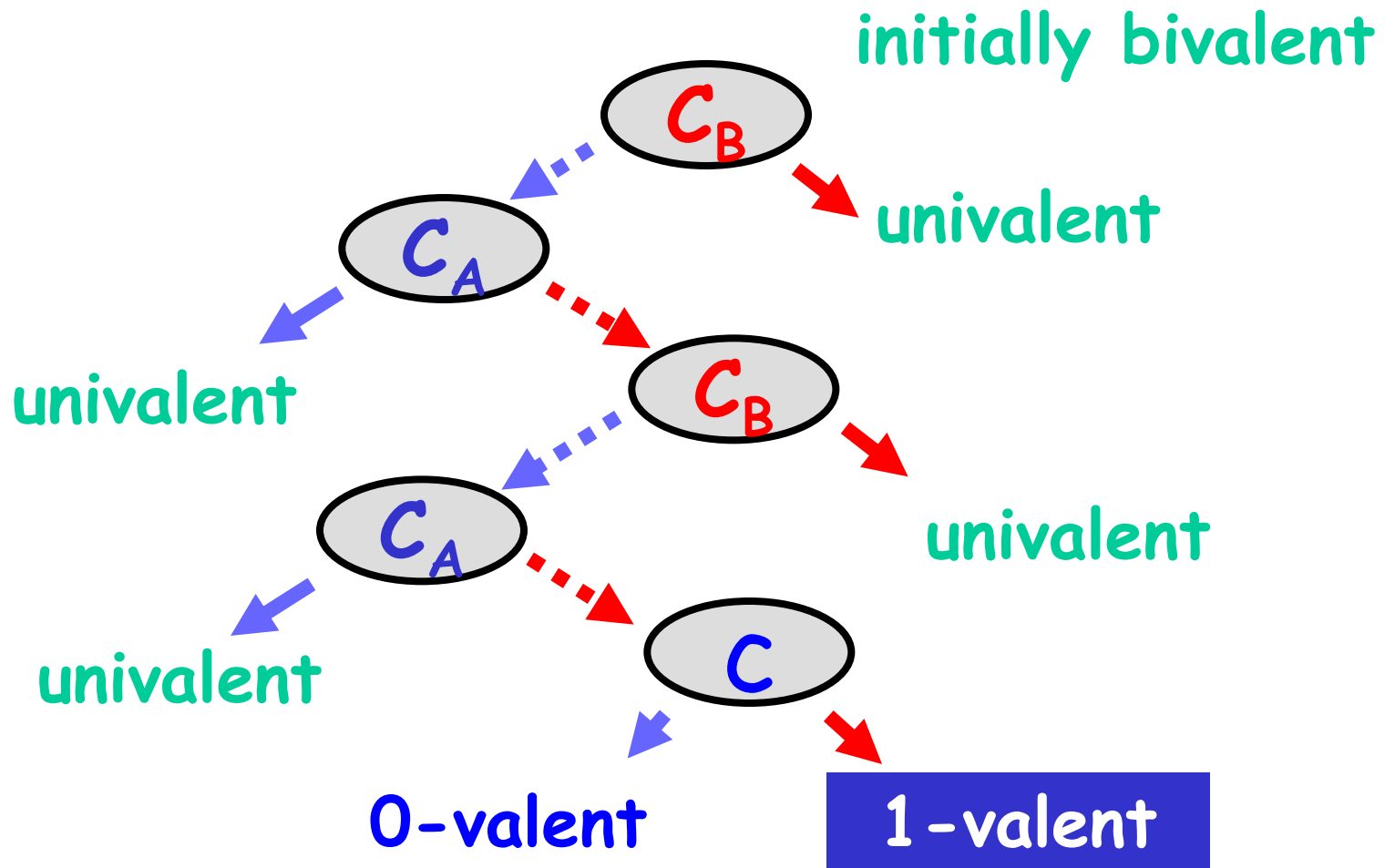
# From a Critical State



**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
- 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

# Read-Write Memory

- Reads and/or writes
- To same/different registers



# Completing the Proof

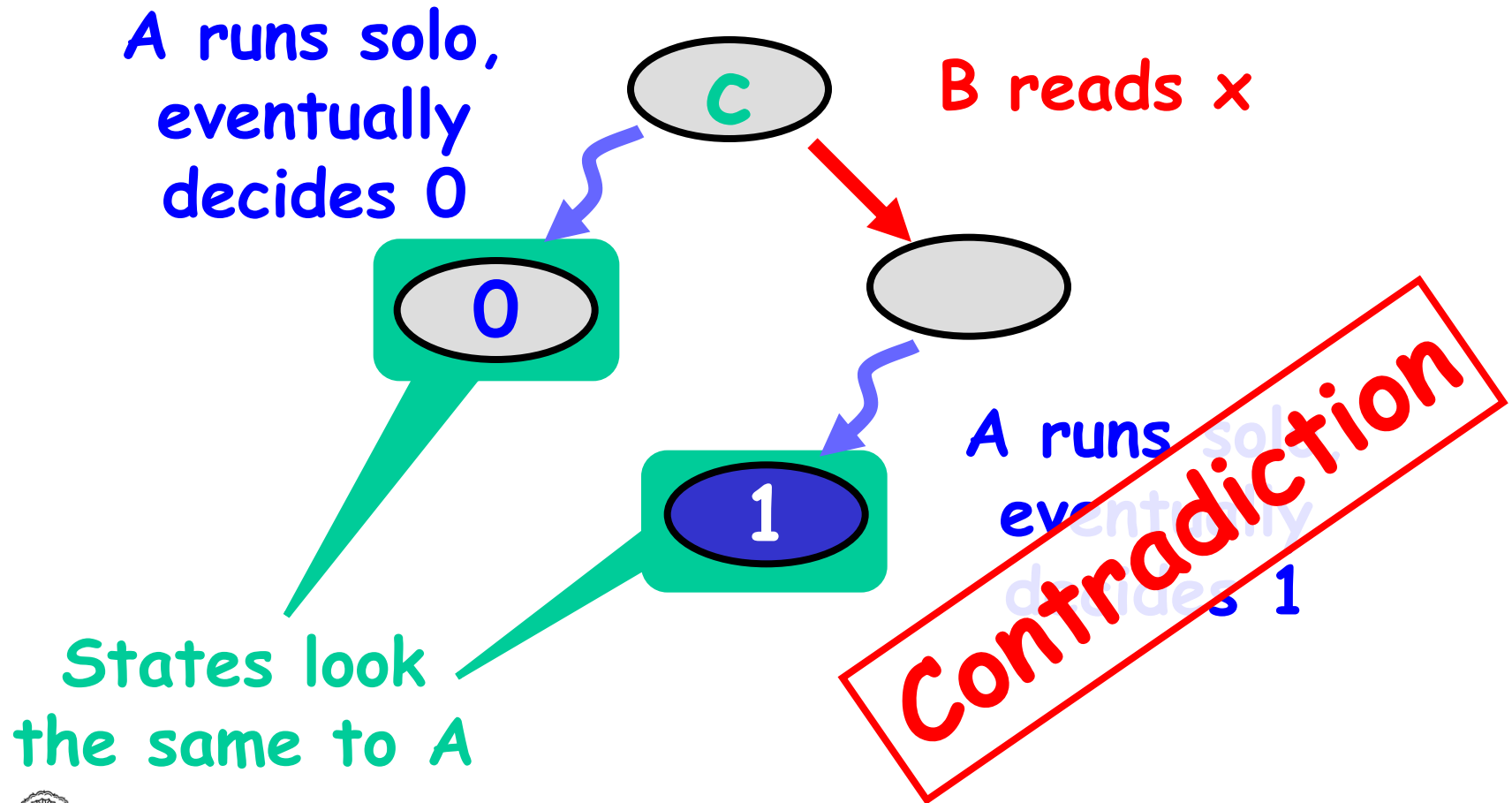
- Lets look at executions that:
  - Start from a critical state
  - Threads cause state to become univalent by reading or writing to same/different registers
  - End within a finite number of steps deciding either 0 or 1
- Show this leads to a contradiction

# Possible Interactions

A reads x  
A reads y

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

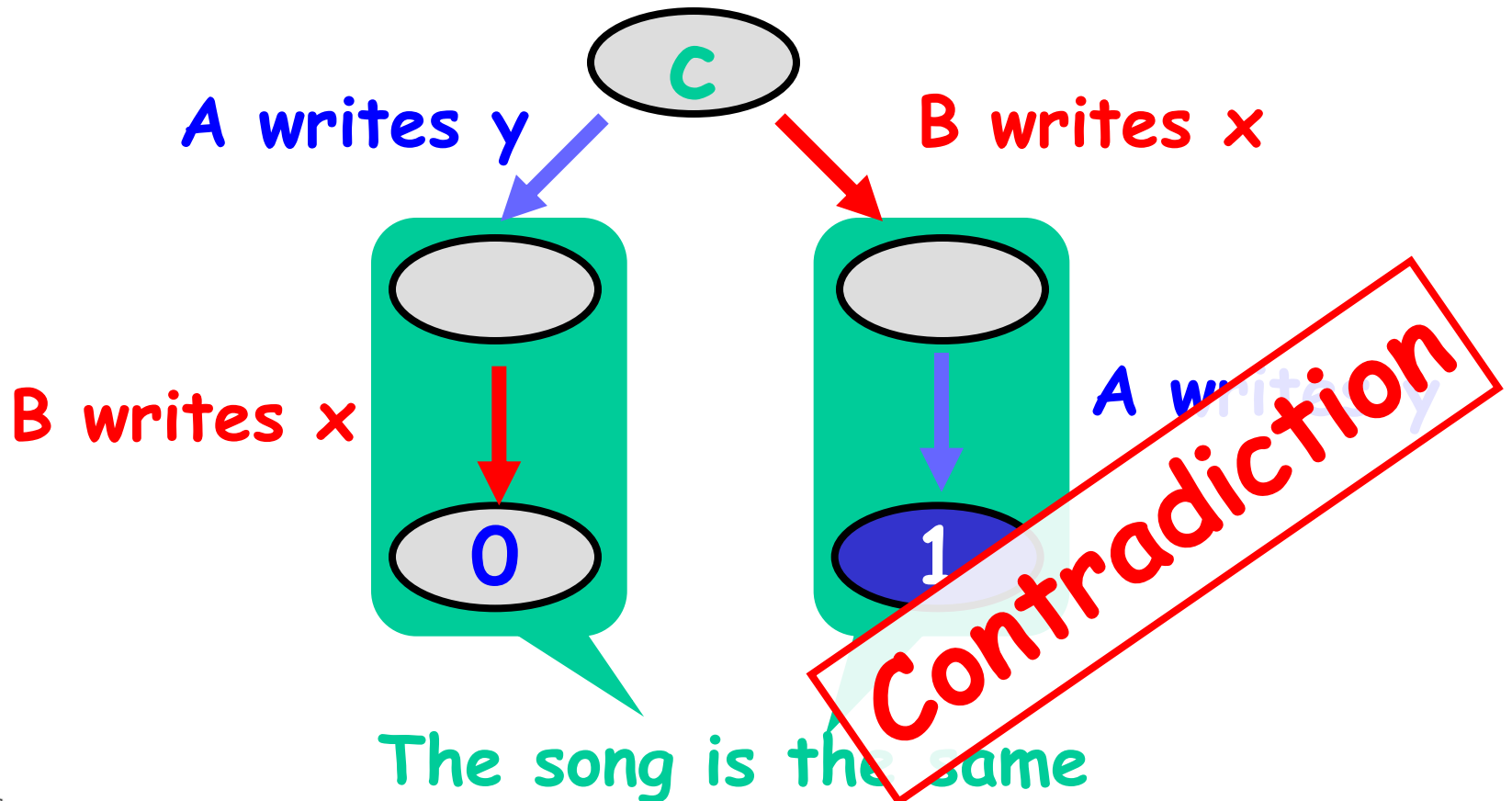
# Some Thread Reads



# 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	?	?

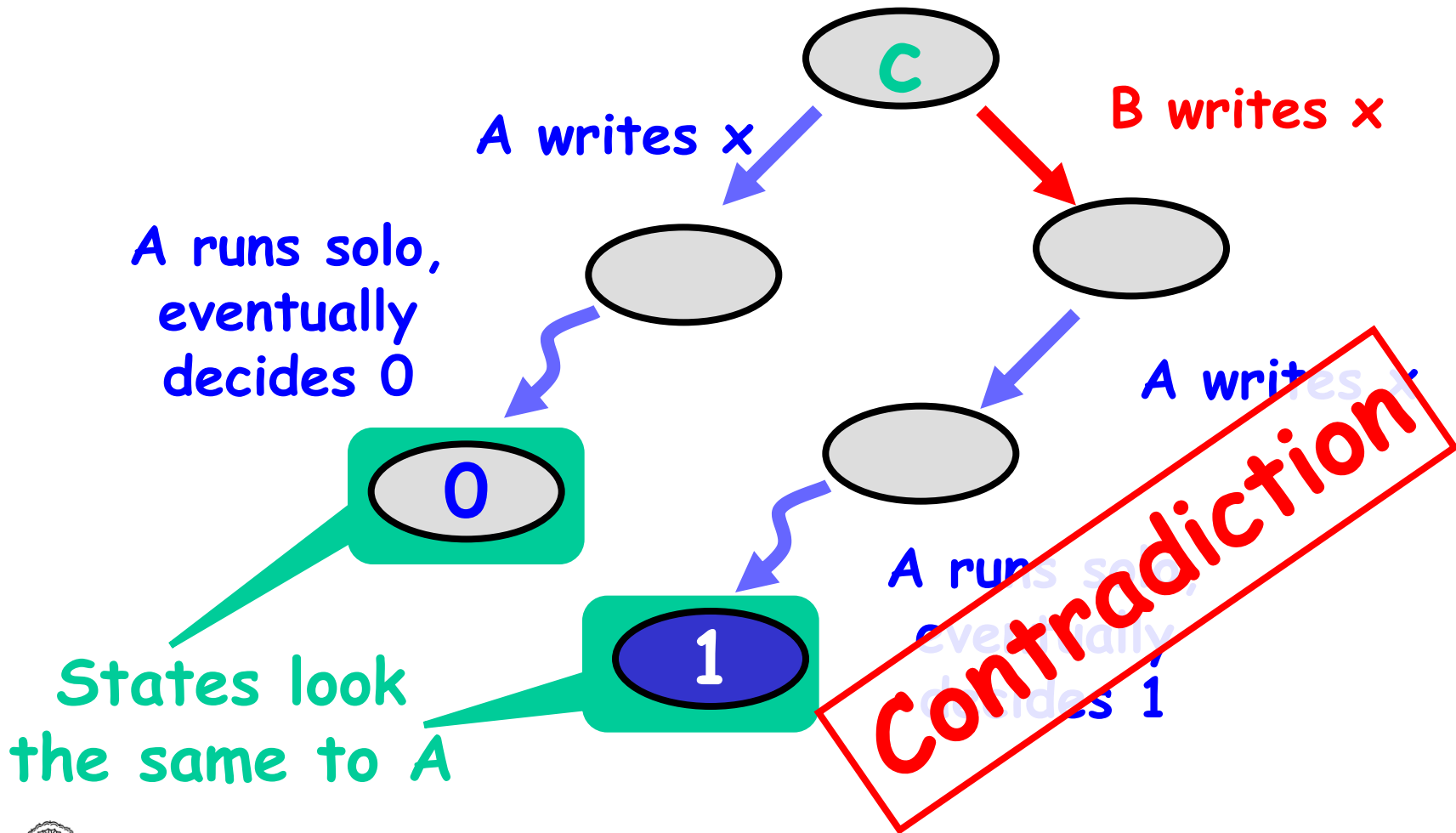
# Writing Distinct Registers



# 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



# 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

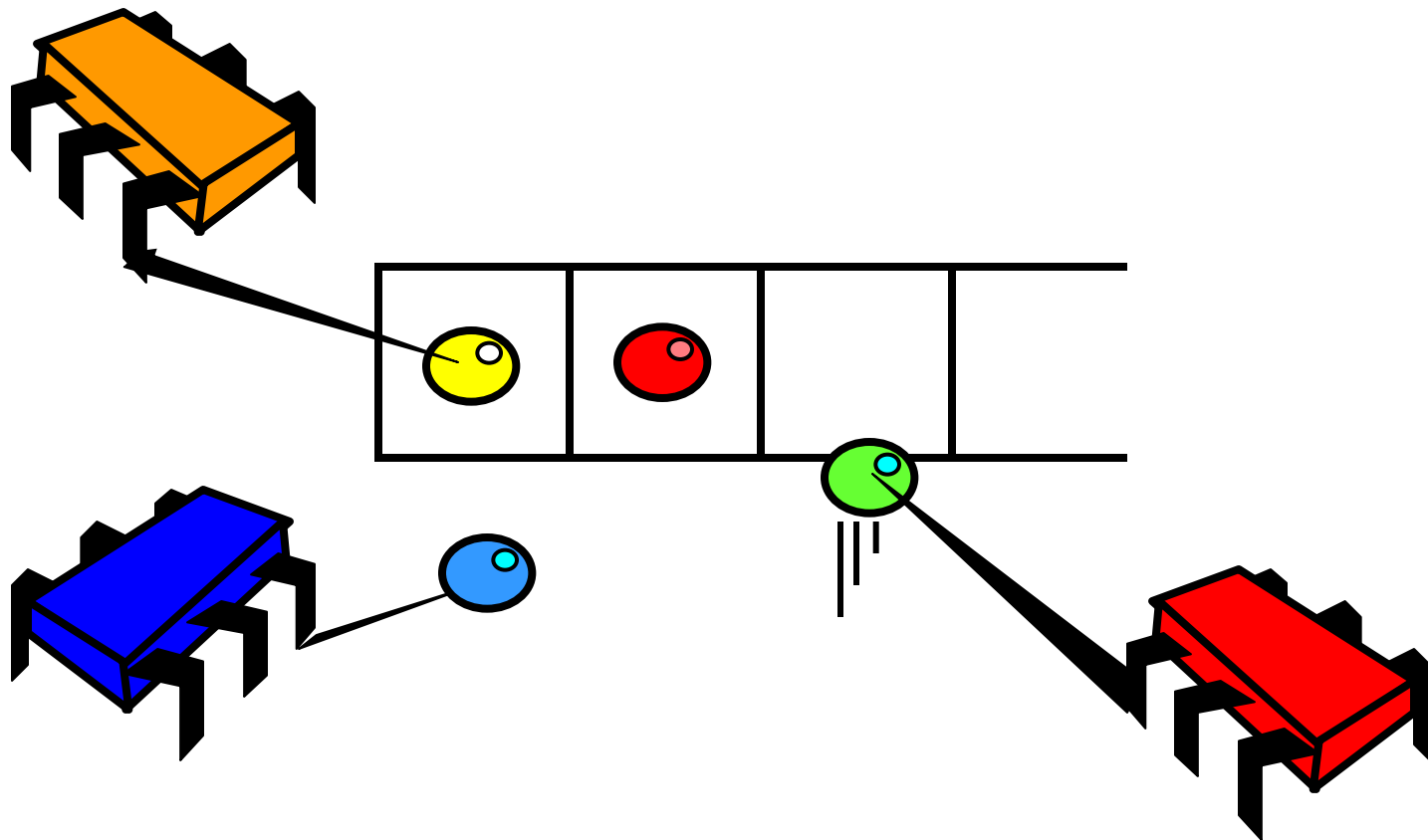
**QED**



# 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

# What Does Consensus have to do with Concurrent Objects?



# Consensus Object

```
public interface Consensus {  
    Object decide(object value);  
}
```



# Concurrent Consensus Object

- We consider only one time objects: each thread can execute a method only once
- Linearizable to sequential consensus object in which
  - the thread who's input was decided on completed its method first

# Java Jargon Watch

- Define Consensus protocol as an abstract class
- We implement some methods
- Leave you to do the rest ...

# Generic Consensus Protocol

```
abstract class ConsensusProtocol
implements Consensus {
    protected Object[] proposed =
        new Object[N];

    private void propose(Object value) {
        proposed[ThreadID.get()] = value;
    }

    abstract public Object
        decide(object value);
}}
```



# Generic Consensus Protocol

```
abstract class ConsensusProtocol
implements Consensus {
    protected Object[] proposed =
        new Object[N];

    private void propose(Object value) {
        proposed[ThreadID.get()] = value;
    }

    abstract public Obj proposed value
        decide(object value);
}}
```

**Each thread's  
proposed value**



# Generic Consensus Protocol

```
abstract class ConsensusProtocol
implements Consensus {
    protected Object[] proposed =
        new Object[N];           Propose a value
    private void propose(Object value) {
        proposed[ThreadID.get()] = value;
    }
    abstract public Object
        decide(object value);
}}
```





# Generic Consensus Protocol

```
abstract class ConsensusProtocol
```

**Decide a value: abstract method  
means subclass does the heavy lifting  
(real work)**

```
private void propose(Object value) {  
    proposed[ThreadID.get()] = value;  
}
```

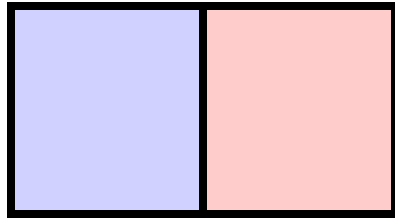
```
abstract public Object  
    decide(object value);
```

```
}}
```

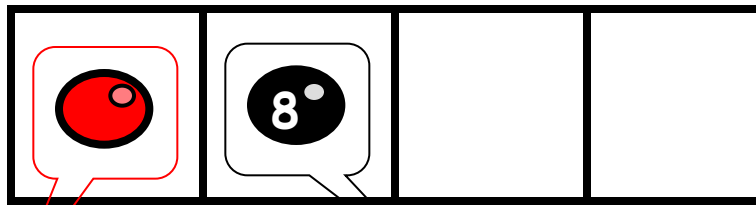


Can a FIFO Queue  
Implement Consensus?

# FIFO Consensus



proposed array

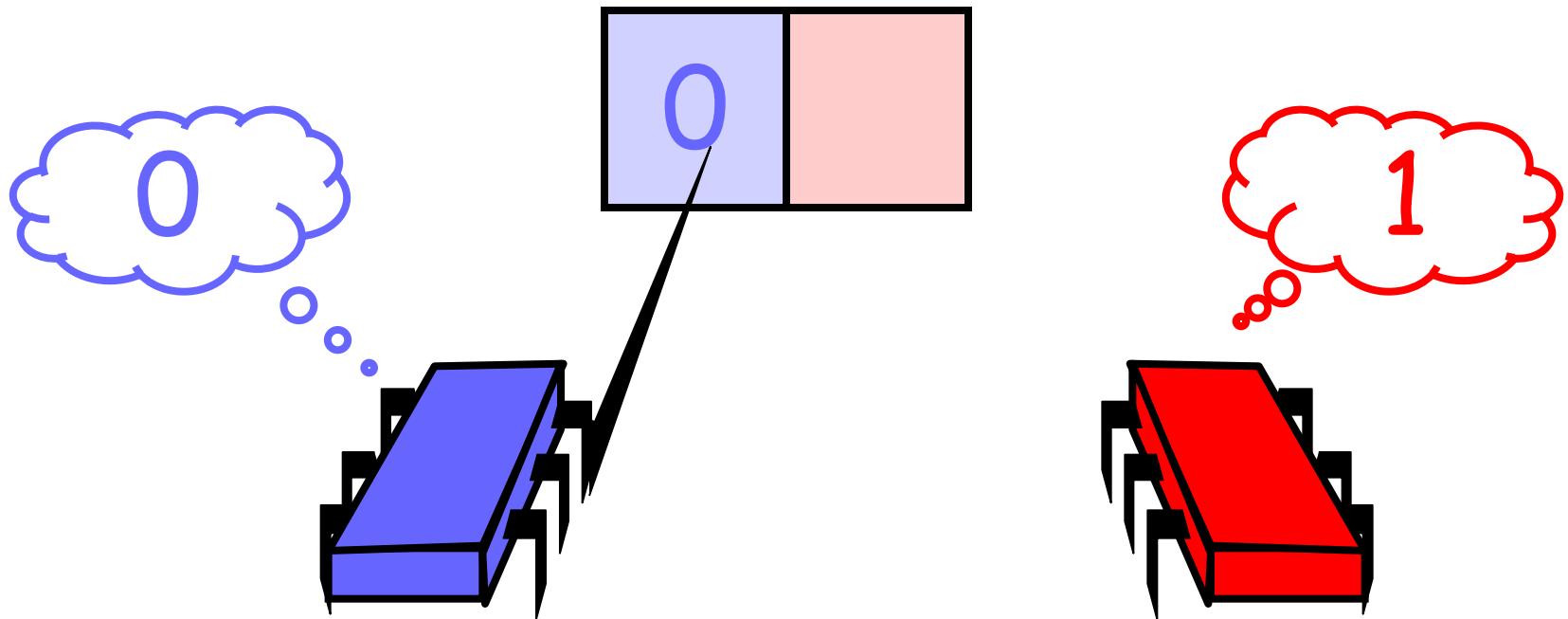


FIFO Queue  
with red and  
black balls

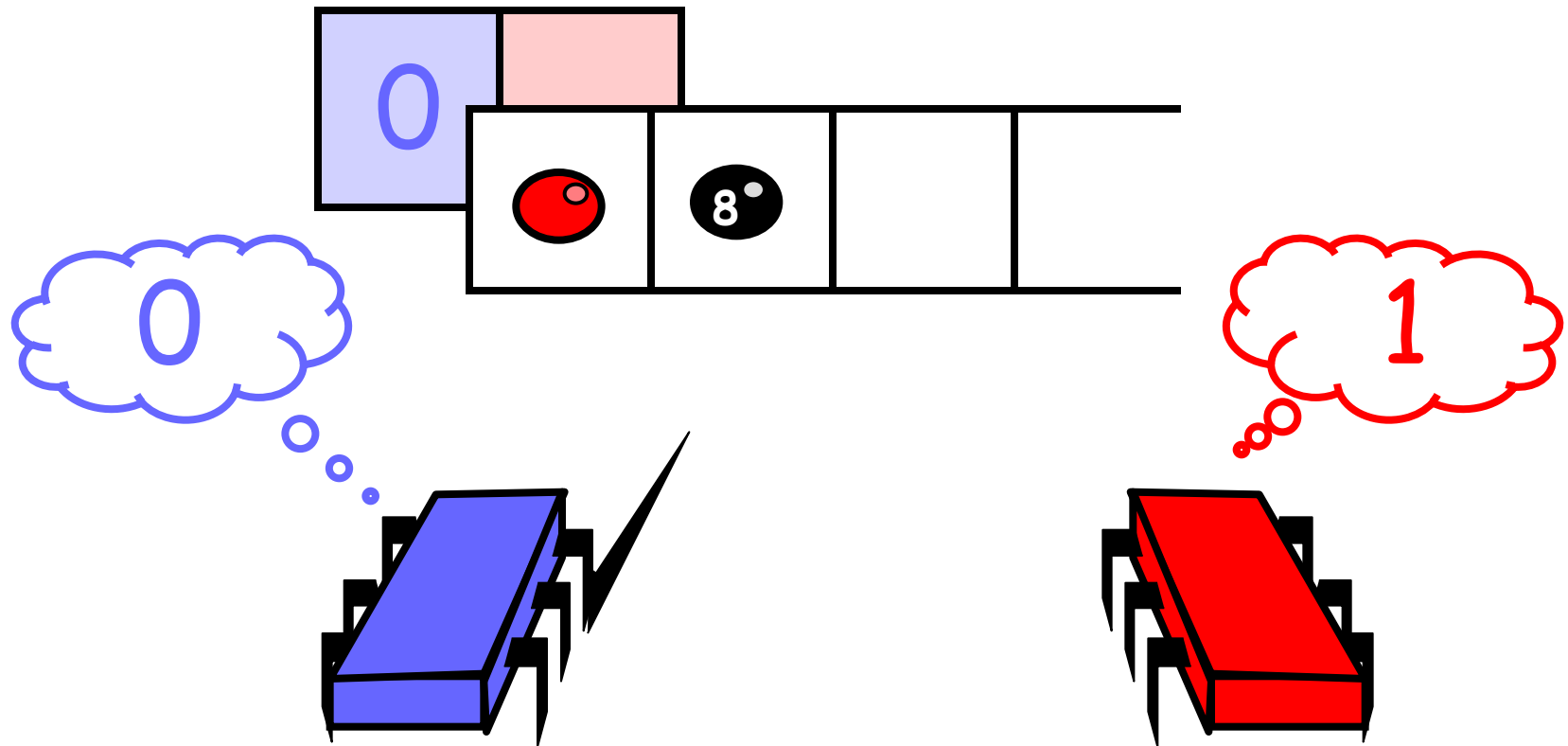
Coveted red ball

Dreaded black ball

# Protocol: Write Value to Array

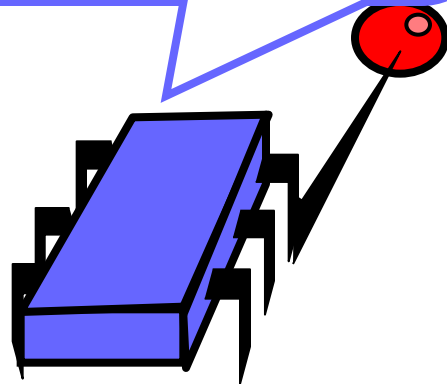


# Protocol: Take Next Item from Queue

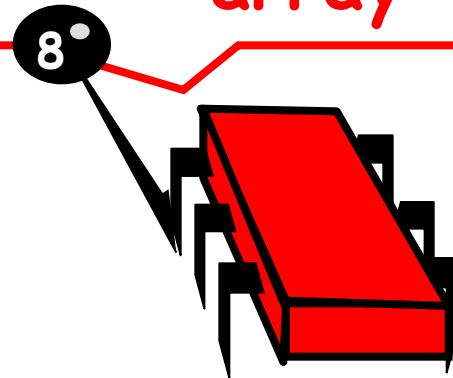


# Protocol: Take Next Item from Queue

I got the coveted red ball, so I will decide my value



I got the dreaded black ball, so I will decide the other's value from the array

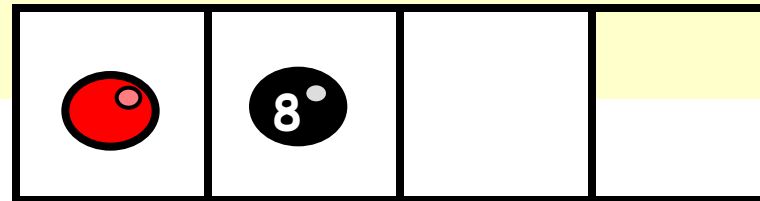


# Consensus Using FIFO Queue

```
public class QueueConsensus
    extends ConsensusProtocol {
    private Queue queue;
    public QueueConsensus() {
        queue = new Queue();
        queue.enq(Ball.RED);
        queue.enq(Ball.BLACK);
    }
    ...
}
```

# Initialize Queue

```
public class QueueConsensus
  extends ConsensusProtocol {
  private Queue queue;
  public QueueConsensus() {
    this.queue = new Queue();
    this.queue.enq(Ball.RED);
    this.queue.enq(Ball.BLACK);
  }
  ...
}
```





# Who Won?

```
public class QueueConsensus
  extends ConsensusProtocol {
  private Queue queue;

  ...

  public decide(object value) {
    propose(value);
    Ball ball = this.queue.deq();
    if (ball == Ball.RED)
      return proposed[i];
    else
      return proposed[1-i];
  }
}
```

# Who Won?

```
public class QueueConsensus
  extends ConsensusProtocol {
  private Queue queue;

  ...
  public decide(object value) {
    propose(value);
    Ball ball = this.queue.deq();
    if (ball == Ball.RED)
      return proposed[i];
    else
      return proposed[1-i];
  }
}
```

**Race to dequeue  
first queue item**

# Who Won?

```
public class QueueConsensus
  extends ConsensusProtocol {
  private Queue queue;
  ...
  public decide(object value) {
    propose(value);
    Ball ball = this.queue.deq();
    if (ball == Ball.RED)
      return proposed[i];
    else
      return proposed[1-i];
  }
}
```

**i = ThreadID.get();  
I win if I was  
first**

# Who Won?

```
public class QueueConsensus
  extends ConsensusProtocol {
  private Queue queue;
  ...
  public decide(object value) {
    propose(value);
    Ball ball = this.queue.deq();
    if (ball == Ball.RED)
      return proposed[i];
    else
      return proposed[1-i];
  }
}
```

Other thread wins if  
I was second

else  
return proposed[1-i];

# Why does this Work?

- If one thread gets the red ball
- Then the other gets the black ball
- Winner decides her own value
- Loser can find winner's value in array
  - Because threads write array
  - Before dequeuing from queue

# Theorem

- We can solve 2-thread consensus using only
  - A two-dequeuer queue, and
  - Some atomic registers

# Implications

- Given
  - A consensus protocol from queue and registers
- Assume there exists
  - A queue implementation from atomic registers
- Substitution yields:
  - A wait-free consensus protocol from atomic registers

**contradiction**

# Corollary

- It is impossible to implement
  - a two-dequeuer wait-free FIFO queue
  - from read/write memory.



# 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

- Theorem
  - Atomic read/write registers have consensus number 1
- Theorem
  - Multi-dequeuer FIFO queues have consensus number at least 2

# Consensus Numbers Measure Synchronization Power

- Theorem
  - If you can implement  $X$  from  $Y$
  - And  $X$  has consensus number  $c$
  - Then  $Y$  has consensus number at least  $c$

# Synchronization Speed Limit

- Conversely

- If  $X$  has consensus number  $d$
- And  $Y$  has consensus number  $d < c$
- Then there is no way to construct a wait-free implementation of  $X$  by  $Y$

- This theorem will be very useful
  - Unforeseen practical implications!

Theoretical  
Caveat: Certain  
weird exceptions  
exist

# Earlier Grand Challenge

- Snapshot means
  - Write any array element
  - Read multiple array elements atomically
- What about
  - Write multiple array elements atomically
  - Scan any array elements
- Call this problem **multiple assignment**

# Multiple Assignment Theorem

- Atomic registers cannot implement multiple assignment
- Weird or what?
  - Single location write/multi location read OK
  - Multi location write/multi location read impossible



# Proof Strategy

- If we can write to 2/3 array elements
  - We can solve 2-consensus
  - Impossible with atomic registers
- Therefore
  - Cannot implement multiple assignment with atomic registers

# Proof Strategy

- Take a 3-element array
  - A writes atomically to slots 0 and 1
  - B writes atomically to slots 1 and 2
  - Any thread can scan any set of locations



# Double Assignment Interface

```
interface Assign2 {  
    public void assign(int i1, int v1,  
                      int i2, int v2);  
    public int read(int i);  
}
```

# Double Assignment Interface

```
interface Assign2 {  
    public void assign(int i1, int v1,  
                      int i2, int v2);  
    public int read(int i);  
}
```

Atomically assign  
value[i<sub>1</sub>] = v<sub>1</sub>  
value[i<sub>2</sub>] = v<sub>2</sub>



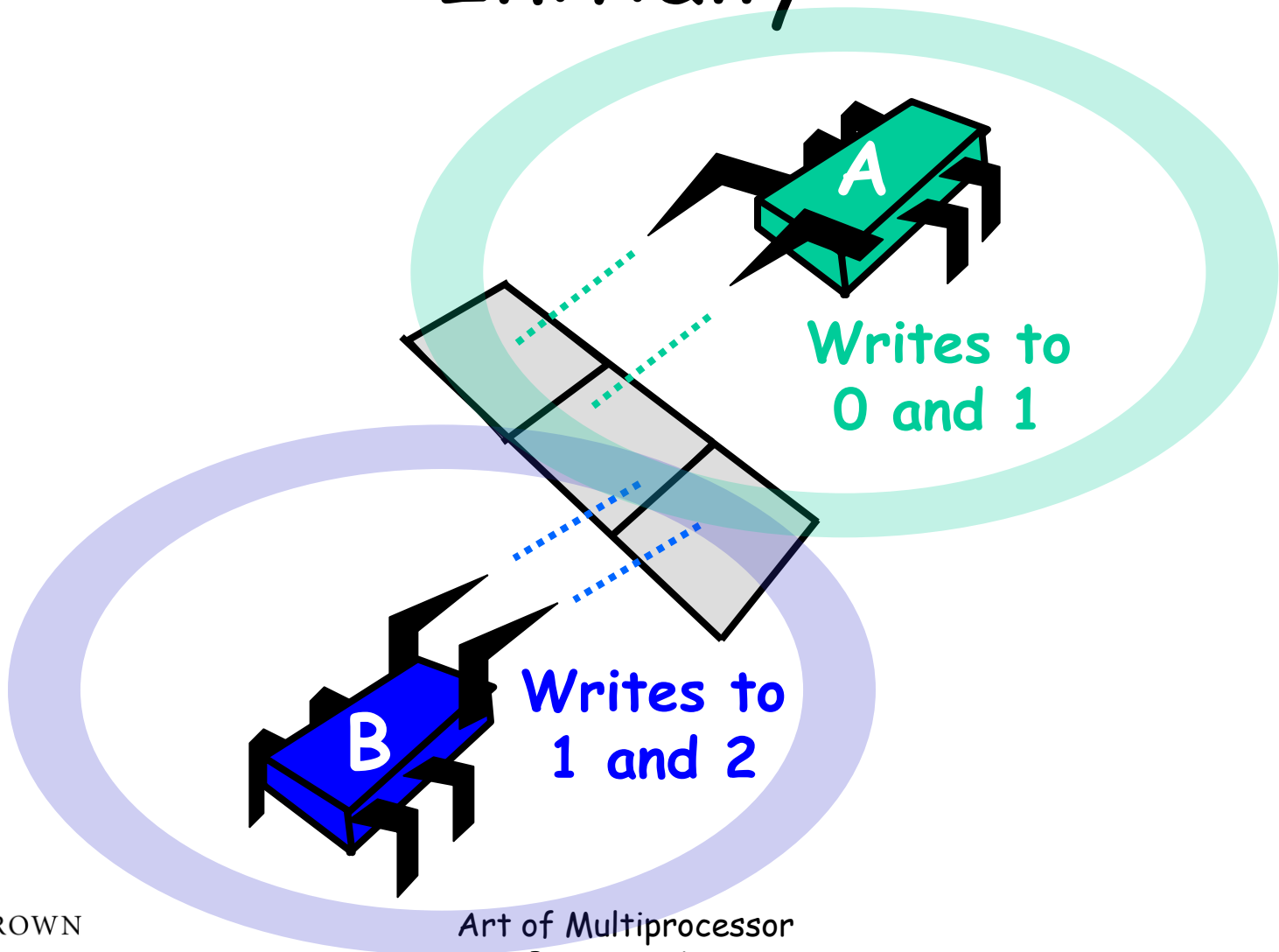
# Double Assignment Interface

```
interface Assign2 {  
    public void assign(int i1, int v1,  
                       int i2, int v2);  
    public int read(int i);  
}
```

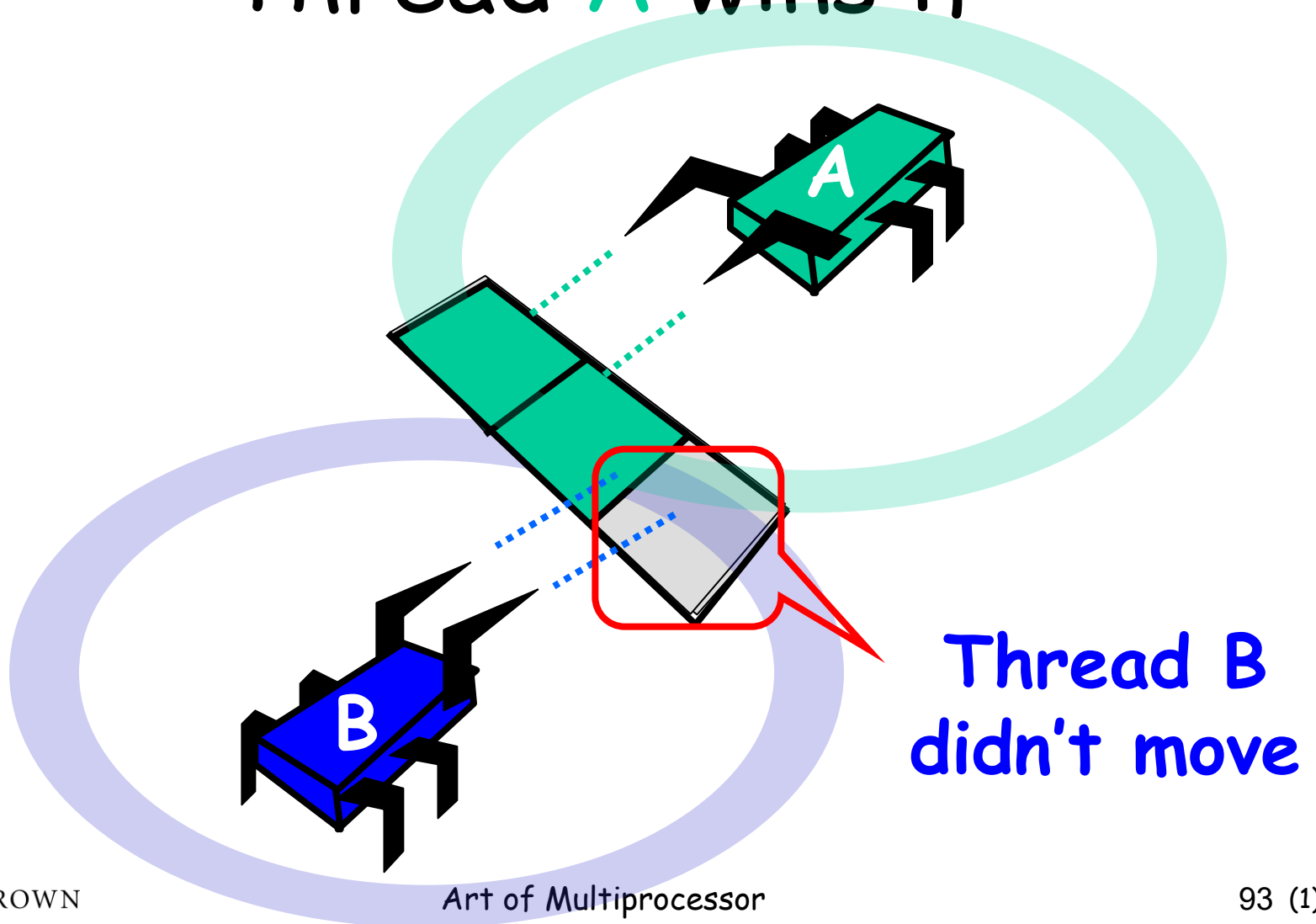
Return *i*-th value



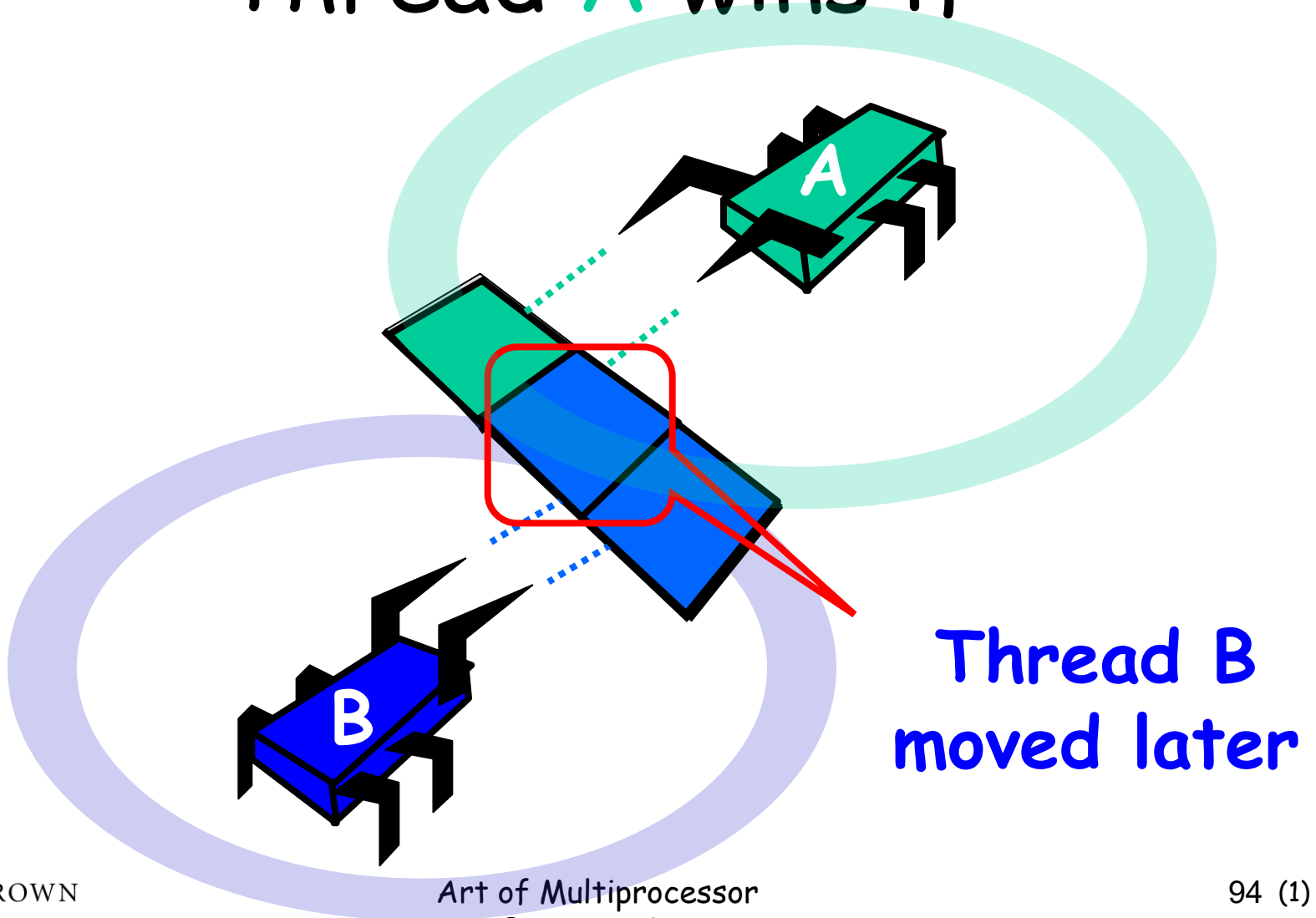
# Initially



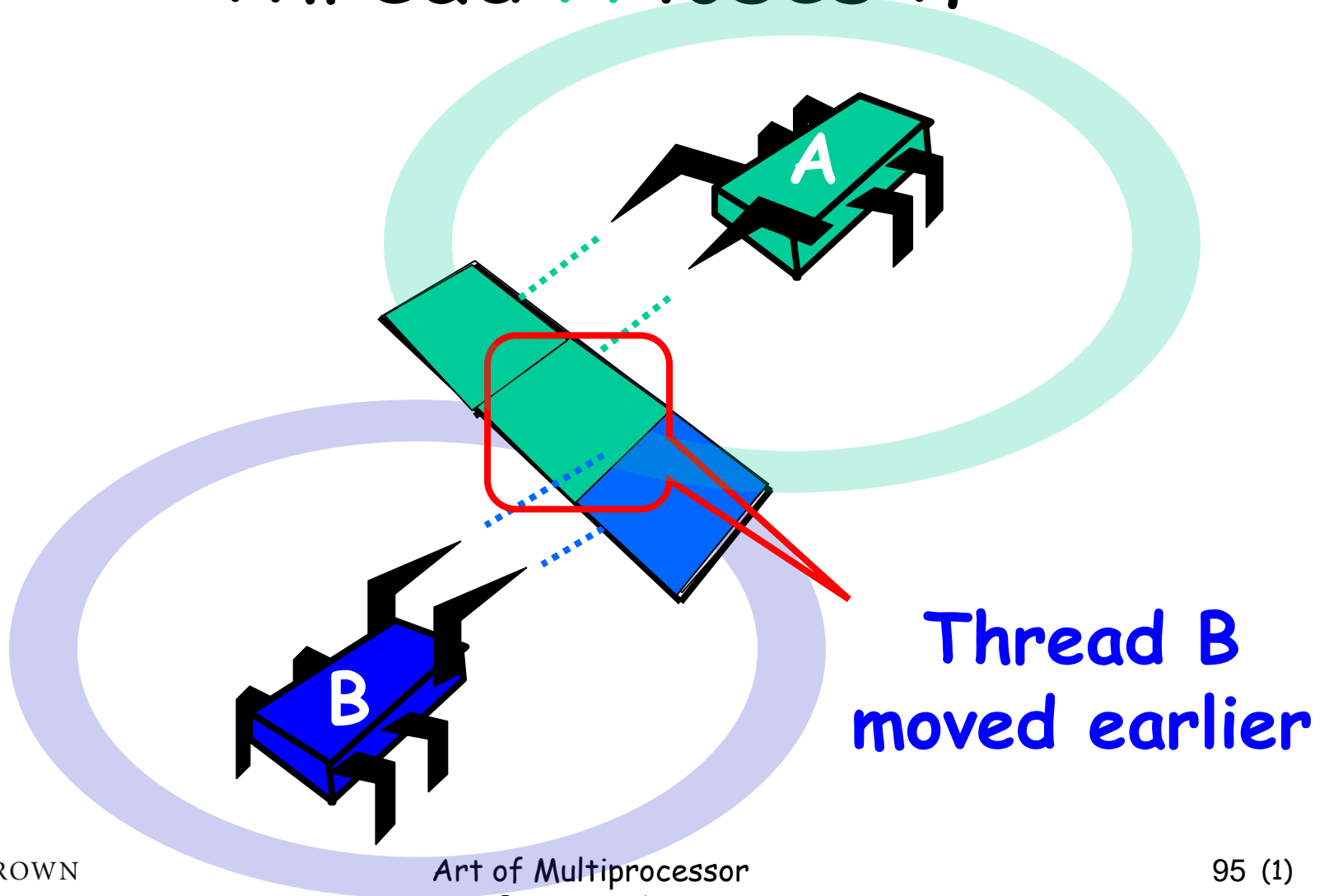
# Thread **A** wins if



# Thread **A** wins if



# Thread **A** loses if



# Multi-Consensus Code

```
class MultiConsensus extends ...{
  Assign2 a = new Assign2(3, EMPTY);
  public Object decide(object value) {
    a.assign(i, i, i+1, i);
    int other = a.read((i+2) % 3);
    if (other==EMPTY || other==a.read(1))
      return proposed[i];
    else
      return proposed[j];
  }
}
```





# Multi-Consensus Code

```
class MultiConsensus extends ...{
    Assign2 a = new Assign2(3, EMPTY);
    public Object decide(Object value) {
        a.assign(i, i, i+1, i);
        int other = a.read((i+2) % 3);
        if (other==EMPTY || other==a.read(1))
            return proposed[i];
        else
            return proposed[j];
    }
}
```

Extends ConsensusProtocol

Decide sets  $j=i-1$  and proposes value



# Multi-Consensus Code

```
class MultiConsensus extends {  
    Assign2 a = new Assign2(3, EMPTY);  
    public Object decide(Object value) {  
        a.assign(i, i, i+1, i);  
        int other = a.read((i+2) % 3);  
        if (other==EMPTY || other==a.read(1))  
            return proposed[i];  
        else  
            return proposed[j];  
    }  
}
```

Three slots  
initialized to  
**EMPTY**



# Multi-Consensus Code

```
class MultiConsensus extends ...{
    Assign2 a = new Assign2(3, EMPTY);
    public Object decide(Object value) {
        a.assign(i, i, i+1, i);
        int other = a.read((i+2) % 3);
        if (other==EMPTY || other==a.read(1))
            return proposed[i];
        else
            return proposed[j];
    }
}
```

**Assign id 0 to  
entries 0,1 (or id 1  
to entries 1,2)**



# Multi-Consensus Code

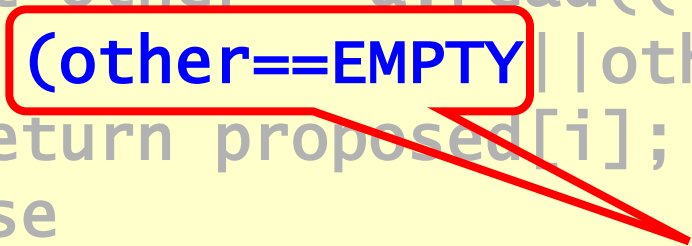
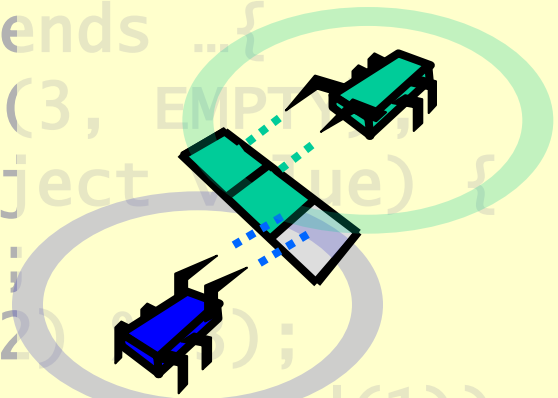
```
class MultiConsensus extends ...{
    Assign2 a = new Assign2(3, EMPTY);
    public Object decide(object value) {
        a.assign(i, i, i+1, i);
        int other = a.read((i+2) % 3);
        if (other==EMPTY || other==a.read(1))
            return proposed[i];
        else
            return proposed[j];
    }
}
```

Read the register my  
thread didn't assign



# Multi-Consensus Code

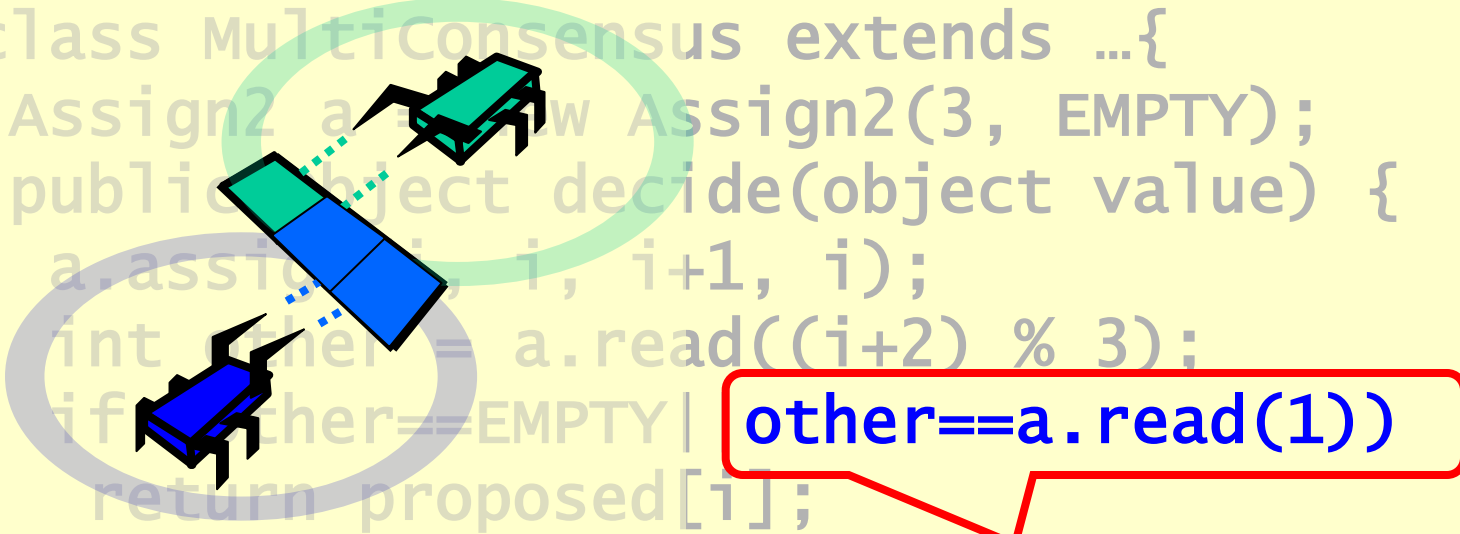
```
class MultiConsensus extends ...{
  Assign2 a = new Assign2(3, EMPTY);
  public Object decide(Object proposed) {
    a.assign(i, i, i+1, i);
    int other = a.read((i+2));
    if (other==EMPTY || other==a.read(1))
      return proposed[i];
    else
      return proposed[j];
  }}
```



Other thread didn't move, so I win

# Multi-Consensus Code

```
class MultiConsensus extends ...{
  Assign2 a = new Assign2(3, EMPTY);
  public Object decide(Object value) {
    a.assign(i, i, i+1, i);
    int other = a.read((i+2) % 3);
    if (other == EMPTY | other == a.read(1))
      return proposed[i];
    else
      return proposed[j];
  }
}
```



Other thread moved later so I win



# Multi-Consensus Code

```
class MultiConsensus extends ...{
  Assign2 a = new Assign2(3, EMPTY);
  public Object decide(object value) {
    a.assign(i, i, i+1, i);
    int other = a.read((i+2) % 3);
    if (other==EMPTY || other==a.read(1))
      return proposed[i];
    else
      return proposed[j];
  }}

```

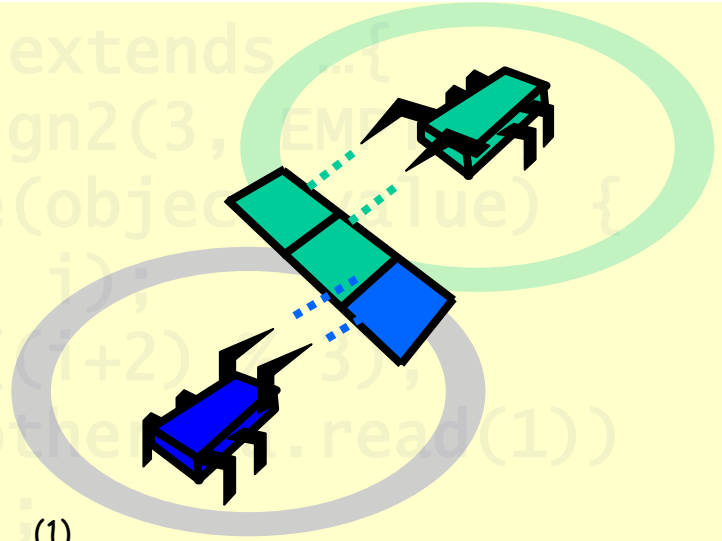
**OK, I win.**

# Multi-Consensus Code

```
class MultiConsensus extends ... {  
    Assign2 a = new Assign2(3, EMPTY);  
    public Object decide(Object value) {  
        a.assign(i, i, i+1, i);  
        int other = a.read(i+2);  
        if (other==EMPTY || other==value) {  
            return proposed[i];  
        }  
        else  
            return proposed[j];  
    }  
}
```

(1)

Other thread moved first, so I lose





# Summary

- If a thread can assign atomically to 2 out of 3 array locations
- Then we can solve 2-consensus
- Therefore
  - No wait-free multi-assignment
  - From read/write registers

# Read-Modify-Write Objects

- Method call
  - Returns object's prior value  $x$
  - Replaces  $x$  with `mumble(x)`

# Read-Modify-Write

```
public abstract class RMWRegister {  
    private int value;  
  
    public int synchronized  
        getAndMumble() {  
        int prior = this.value;  
        this.value = mumble(this.value);  
        return prior;  
    }  
}
```



# Read-Modify-Write

```
public abstract class RMWRegister {  
    private int value;  
  
    public int synchronized  
    getAndMumble() {  
        int prior = this.value;  
        this.value = mumble(this.value);  
        return prior;  
    }  
}
```

**Return prior value**



# Read-Modify-Write

```
public abstract class RMWRegister {  
    private int value;
```

```
    public int synchronized
```

```
        getAndMumble() {
```

```
            int prior = this.value;
```

```
            this.value = mumble(this.value);
```

```
            return prior;
```

```
        }
```

```
    }
```

**Apply function to current value**



# RMW Everywhere!

- Most synchronization instructions
  - are RMW methods
- The rest
  - Can be trivially transformed into RMW methods

# Example: Read

```
public abstract class RMWRegister {  
    private int value;  
  
    public int synchronized read() {  
        int prior = this.value;  
        this.value = this.value;  
        return prior;  
    }  
}
```

# Example: Read

```
public abstract class RMW {  
    private int value;  
  
    public int synchronized read() {  
        int prior = this.value;  
        this.value = this.value;  
        return prior;  
    }  
}
```

**Apply  $f(v)=v$ , the identity function**





# Example: getAndSet

```
public abstract class RMWRegister {  
    private int value;  
  
    public int synchronized  
        getAndSet(int v) {  
        int prior = this.value;  
        this.value = v;  
        return prior;  
    }  
    ...  
}
```

# Example: getAndSet (swap)

```
public abstract class RMWRegister {
    private int value;

    public int synchronized
        getAndSet(int v) {
        int prior = this.value;
        this.value = v;
        return prior;
    }
    ...
}
```

**$F(x)=v$  is constant function**



# getAndIncrement

```
public abstract class RMWRegister {  
    private int value;  
  
    public int synchronized  
        getAndIncrement() {  
        int prior = this.value;  
        this.value = this.value + 1;  
        return prior;  
    }  
    ...  
}
```

# getAndIncrement

```
public abstract class RMWRegister {  
    private int value;  
  
    public int synchronized  
        getAndIncrement() {  
        int prior = this.value;  
        this.value = this.value + 1;  
        return prior;  
    }  
    ...  
}
```

$$F(x) = x+1$$



# getAndAdd

```
public abstract class RMWRegister {  
    private int value;  
  
    public int synchronized  
        getAndAdd(int a) {  
        int prior = this.value;  
        this.value = this.value + a;  
        return prior;  
    }  
    ...  
}
```



# Example: getAndAdd

```
public abstract class RMWRegister {  
    private int value;  
  
    public int synchronized  
        getAndIncrement(int a) {  
        int prior = this.value;  
        this.value = this.value + a;  
        return prior;  
    }  
    ...  
}
```

**$F(x) = x+a$**



# compareAndSet

```
public abstract class RMWRegister {  
    private int value;  
    public boolean synchronized  
        compareAndSet(int expected,  
                       int update) {  
        int prior = this.value;  
        if (this.value==expected) {  
            this.value = update; return true;  
        }  
        return false;  
    } ... }  
}
```

# compareAndSet

```
public abstract class RMWRegister {  
    private int value;  
    public boolean synchronized  
        compareAndSet(int expected,  
                       int update) {  
        int prior = this.value;  
        if (this.value == expected) {  
            this.value = update; return true;  
        }  
        return false;  
    } ... }  
}
```

If value is what was  
expected, ...





# compareAndSet

```
public abstract class RMWRegister {  
    private int value;  
    public boolean synchronized  
        compareAndSet(int expected,  
                      int update) {  
        int prior = this.value;  
        if (this.value==expected) {  
            this.value = update; return true;  
        }  
        return false;  
    } ... }  
    ... replace it
```



# compareAndSet

```
public abstract class RMWRegister {  
    private int value;  
    public boolean synchronized  
        compareAndSet(int expected,  
                       int update) {  
        int prior = this.value;  
        if (this.value==expected) {  
            this.value = update; return true;  
        }  
        return false;  
    } ... }  
}
```

**Report success**



# compareAndSet

```
public abstract class RMWRegister {  
    private int value;  
    public boolean synchronized  
        compareAndSet(int expected,  
                      int update) {  
        int prior = this.value;  
        if (this.value==expected) {  
            this.value = update; return true;  
        }  
        return false;  
    } ... }  
}
```

Otherwise report  
failure



# Read-Modify-Write

```
public abstract class RMWRegister {  
    private int value;  
  
    public void synchronized  
    getAndMumble() {  
        int prior = this.value;  
        this.value = mumble(this.value);  
        return prior;  
    }  
}
```

Lets characterize  $F(x)$ ...



# Definition

- A RMW method
  - With function  $mumble(x)$
  - is non-trivial if there exists a value  $v$
  - Such that  $v \neq mumble(v)$

# Par Example

- $\text{Identity}(x) = x$ 
  - is trivial
- $\text{getAndIncrement}(x) = x+1$ 
  - is non-trivial

# Theorem

- Any non-trivial RMW object has consensus number at least 2
- No wait-free implementation of RMW registers from atomic registers
- Hardware RMW instructions not just a convenience

# Reminder

- Subclasses of consensus have
  - propose(x) method
    - which just stores x into proposed[i]
    - built-in method
  - decide(object value) method
    - which determines winning value
    - customized, class-specific method



# Proof

```
public class RMWConsensus
    extends ConsensusProtocol {
private RMWRegister r = v;
public Object decide(object value) {
    propose(value);
    if (r.getAndMumble() == v)
        return proposed[i];
    else
        return proposed[j];
}}
```



# Proof

```
public class RMWConsensus
    extends ConsensusProtocol {
    private RMWRegister r = v;
    public Object decide(Object value) {
        propose(value);
        if (r.getAndMumble() == v)
            return proposed[i];
        else
            return proposed[j];
    }
}
```

**Initialized to v**



# Proof

```
public class RMWConsensus
    extends Consensus {
    private RMWRegister r = v;

    public Object decide(Object value) {
        if (r.getAndMumble() == v)
            return proposed[i];
        else
            return proposed[j];
    }
}
```

**Am I first?**



# Proof

```
public class RMWConsensus
    extends ConsensusProtocol {
    private RMWRegister r = v;
    public Object decide(object value) {
        propose(value);
        if (r.getAndMumble() == v) Yes, return
            return proposed[i]; my input
        else
            return proposed[j];
    }
}
```



# Proof

```
public class RMWConsensus
    extends ConsensusProtocol {
private RMWRegister r = v;
public Object decide(object value) {
    propose(value);
    if (r.getAndMumble() == v)
        return proposed[i];
    else
        return proposed[j];
}}
```

No, return other's input



# Proof

- We have displayed
  - A two-thread consensus protocol
  - Using any non-trivial RMW object

# Interfering RMW

- Let  $F$  be a set of functions such that for all  $f_i$  and  $f_j$ , either
  - Commute:  $f_i(f_j(v)) = f_j(f_i(v))$
  - Overwrite:  $f_i(f_j(v)) = f_i(v)$
- Claim: Any set of RMW objects that commutes or overwrites has consensus number exactly 2

# Examples

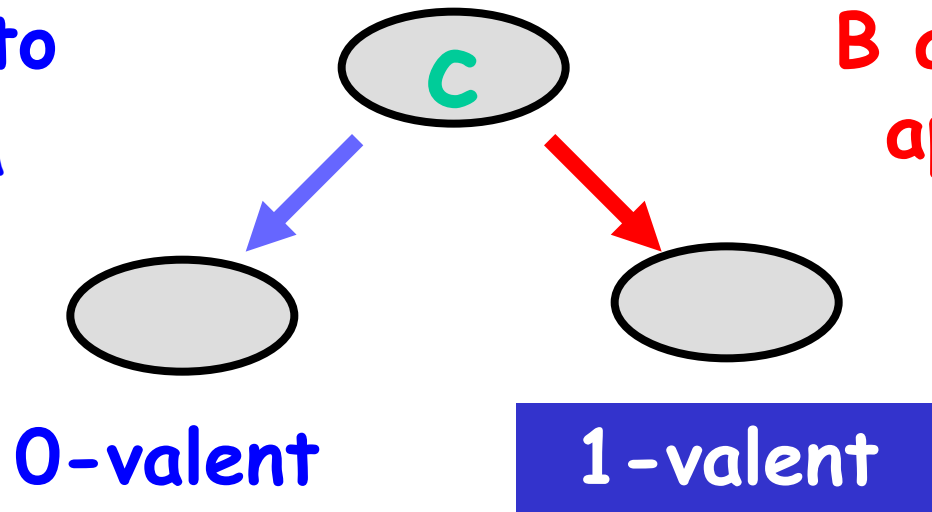
- “test-and-set” `getAndSet(1)`  $f(v)=1$   
**Overwrite**  $f_i(f_j(v))=f_i(v)$
- “swap” `getAndSet(x)`  $f(v,x)=x$   
**Overwrite**  $f_i(f_j(v))=f_i(v)$
- “fetch-and-inc” `getAndIncrement()`  $f(v)=v+1$   
**Commute**  $f_i(f_j(v))= f_j(f_i(v))$



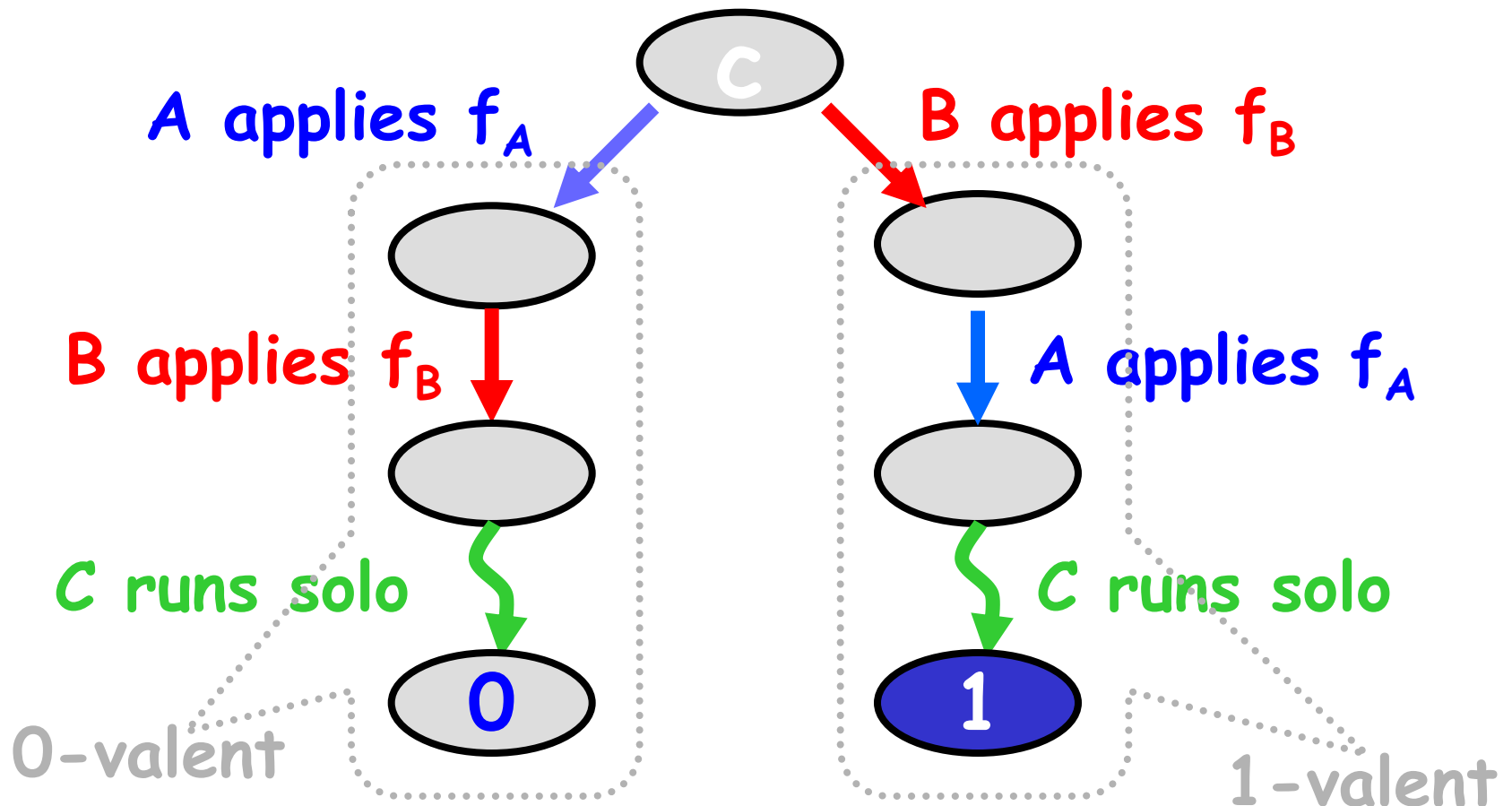
# Meanwhile Back at the Critical State

A about to apply  $f_A$

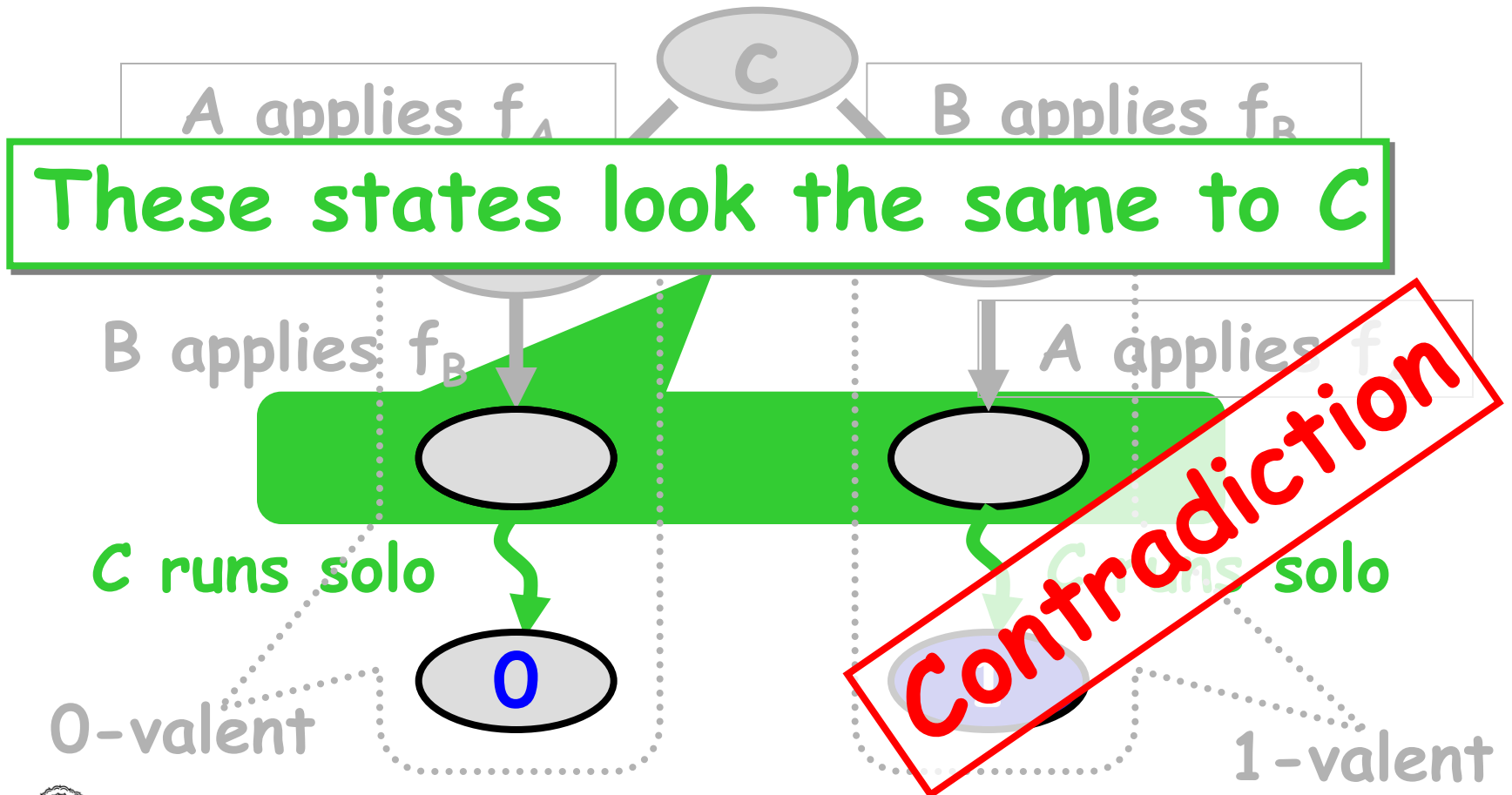
B about to apply  $f_B$



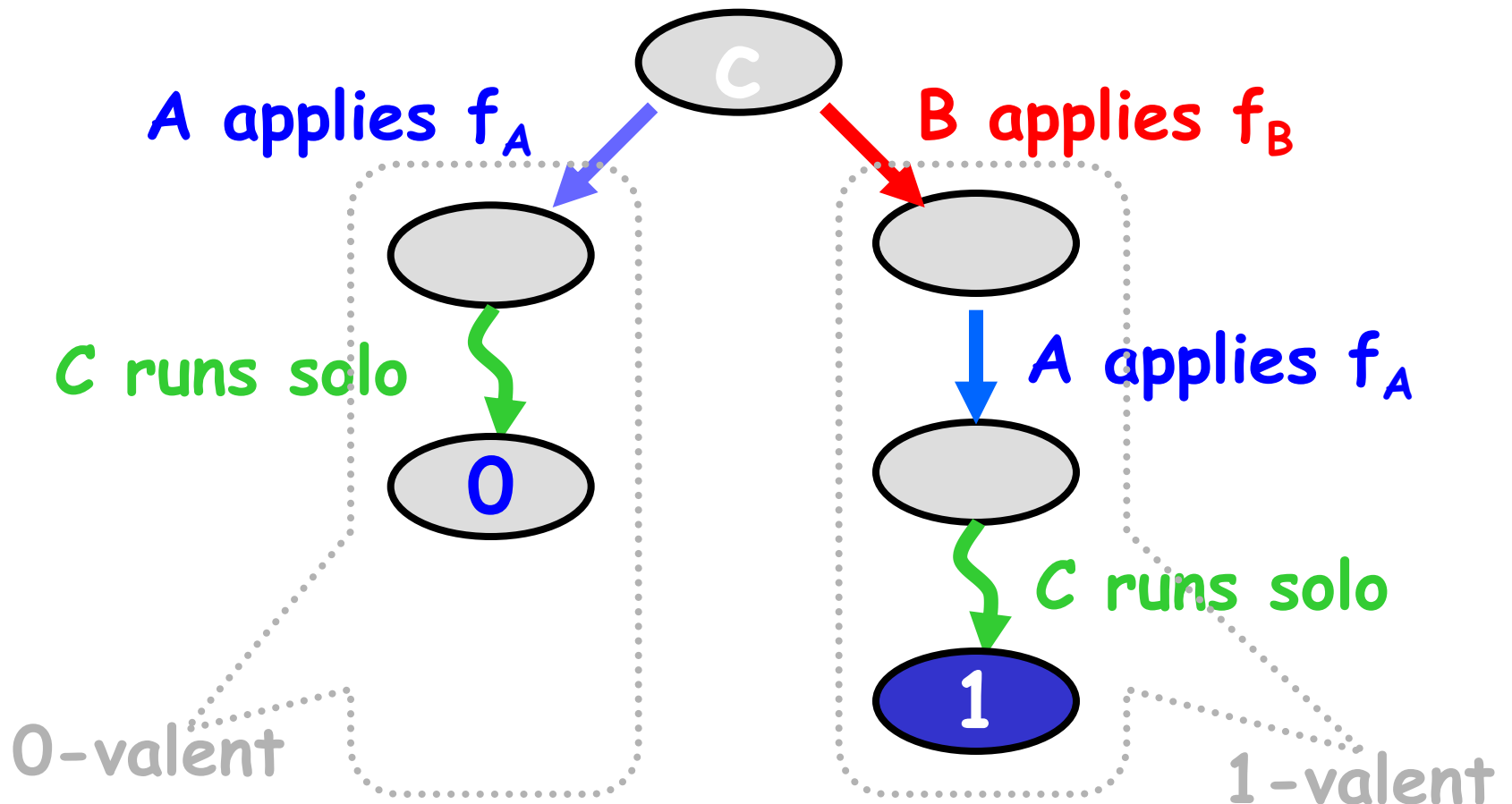
# Maybe the Functions Commute



# Maybe the Functions Commute

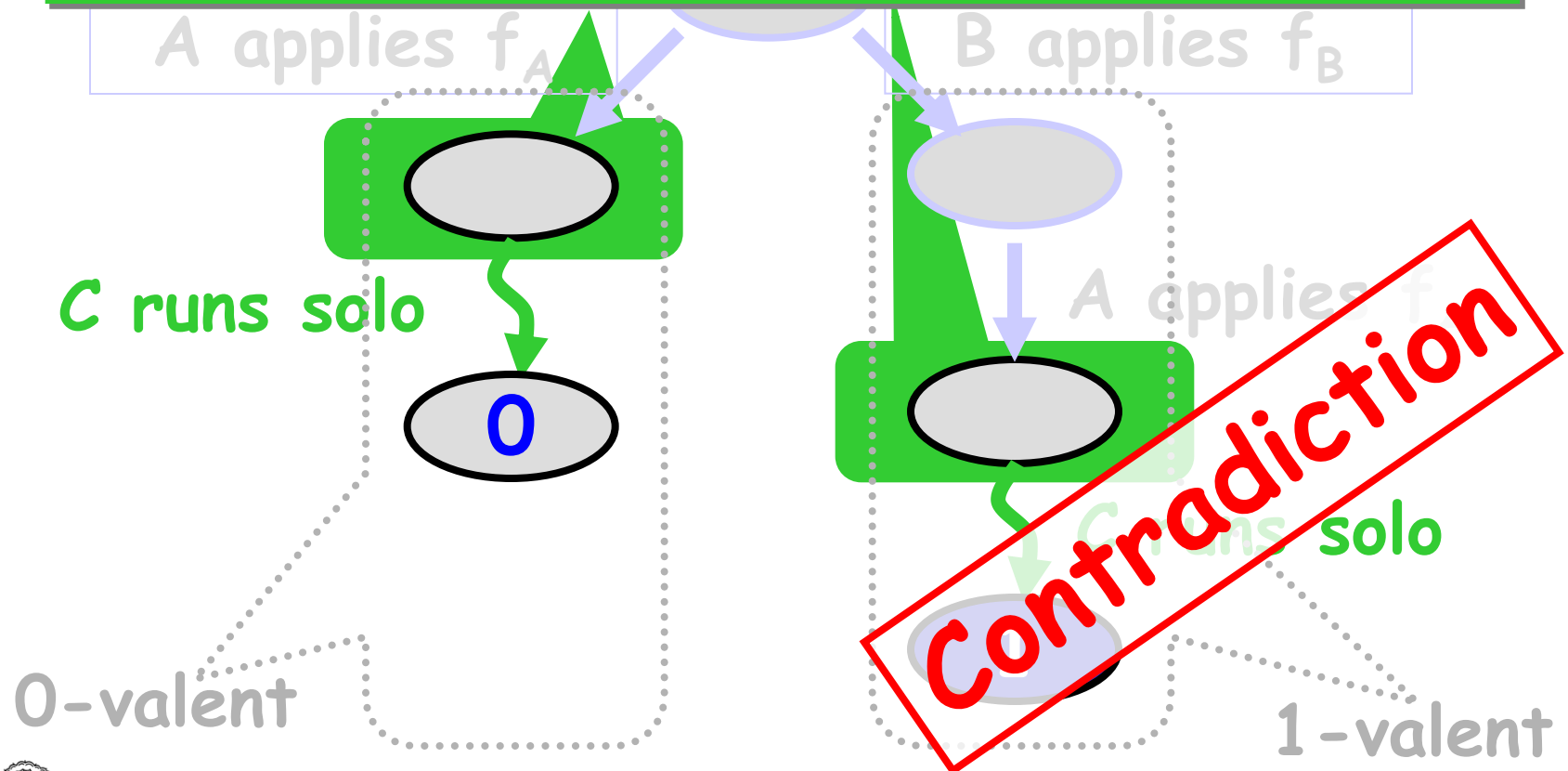


# Maybe the Functions Overwrite



# Maybe the Functions Overwrite

These states look the same to C



# Impact

- Many early machines provided these "weak" RMW instructions
  - Test-and-set (IBM 360)
  - Fetch-and-add (NYU Ultracomputer)
  - Swap (Original SPARCs)
- We now understand their limitations
  - But why do we want consensus anyway?

# compareAndSet

```
public abstract class RMWRegister {
    private int value;
    public boolean synchronized
        compareAndSet(int expected,
                      int update) {
        int prior = this.value;
        if (this.value==expected) {
            this.value = update; return true;
        }
        return false;
    } ... }
```



# compareAndSet

```
public abstract class RMWRegister {  
    private int value;  
    public boolean synchronized  
        compareAndSet(int expected,  
                       int update) {  
        int prior = this.value;  
        if (this.value == expected) {  
            this.value = update; return true;  
        }  
        return false;  
    } ... }  
}
```

replace value if its what we  
expected, ...





# compareAndSet Has $\infty$ Consensus Number

```
public class RMWConsensus
    extends ConsensusProtocol {
    private AtomicInteger r =
        new AtomicInteger(-1);
    public Object decide(object value) {
        propose(value);
        r.compareAndSet(-1, i);
        return proposed[r.get()];
    }
}
```



# compareAndSet Has $\infty$ Consensus Number

```
public class RMWConsensus
    extends ConsensusProtocol {
    private AtomicInteger r =
        new AtomicInteger(-1);
    public Object decide(Object value) {
        propose(value)
        r.compareAndSet(-1, i);
        return proposed[r.get()];
    }
}
```

**Initialized to -1**



# compareAndSet Has $\infty$ Consensus Number

```
public class RMWConsensus
    extends ConsensusProtocol {
    private AtomicInteger r =
        new AtomicInteger(-1);
    public Object decide(object value) {
        propose(value),
        r.compareAndSet(-1, i);
        return proposed[r.get()];
    }
}
```

Try to swap in  
my id



# compareAndSet Has $\infty$ Consensus Number

```
public class RMWConsensus
    extends ConsensusProtocol {
    private AtomicInteger r =
        new AtomicInteger(-1);
    public Object decide(object value) {
        propose(value);
        r.compareAndSet(-1, i);
        return proposed[r.get()];
    }
}
```

**Decide winner's preference**

**return proposed[r.get()];**



# The Consensus Hierarchy

1 Read/Write Registers, Snapshots...

2 getAndSet, getAndIncrement, ...

•  
•  
•

$\infty$  compareAndSet, ...

# Multiple Assignment

- Atomic  $k$ -assignment
- Solves consensus for  $2k-2$  threads
- Every even consensus number has an object (can be extended to odd numbers)

# Lock-Freedom

- Lock-free: in an infinite execution infinitely often some method call finishes (obviously, in a finite number of steps)
- Pragmatic approach
- Implies no mutual exclusion



# Lock-Free vs. Wait-free

- Wait-Free: each method call takes a finite number of steps to finish
- Lock-free: infinitely often some method call finishes





# Lock-Freedom



- Any wait-free implementation is lock-free.
- Lock-free is the same as wait-free if the execution is finite.
- Old saying: "Lock-free is to wait-free as deadlock-free is to lockout-free."

# Lock-Free Implementations

- Lock-free consensus is as impossible as wait-free consensus
- *All the results we presented hold for lock-free algorithms also.*

# There is More: Universality

- Consensus is **universal**
- From **n-thread consensus**
  - Wait-free/Lock-free
  - Linearizable
  - n-threaded
  - Implementation
  - Of any sequential specified object

Stay tuned...

# Uninterruptible Instructions to Fetch and Update Memory

- Atomic exchange: interchange value in register with one in memory
  - 0  $\Rightarrow$  Synchronization variable is free
  - 1  $\Rightarrow$  Synchronization variable is locked and unavailable
  - Set register to 1 & swap
  - New value in register determines success in getting lock
    - 0 if you succeeded in setting lock (you were first)
    - 1 if another processor claimed access first
  - Key: exchange operation is indivisible

# Uninterruptible Instruction to Fetch and Update Memory

- Hard to read & write in 1 instruction, so use 2
- Load linked (or load locked) + store conditional
  - Load linked returns initial value
  - Store conditional returns 1 if succeeds (no other store to same memory location since preceding load) and 0 otherwise

# Example of atomic swap with LL & SC

```
try:  mov    R3,R4    ; mov exchange value->R3
      ll     R2,0(R1) ; get old value
      sc     R3,0(R1) ; store new value
      beqz   R3,try    ; loop if store fails
      mov    R4,R2    ; put old value in R4
```

# Example of fetch & inc with LL & SC

```
try:  ll     R2,0(R1) ; get old value
      addi   R2,R2,#1 ; increment it
      sc     R2,0(R1) ; store new value
      beqz   R2,try    ; loop if store fails
```

# User-Level Synchronization Using LL/SC

- Spin locks: processor continuously tries to acquire lock, spinning around loop trying to get it

```
lockit:      li      R2,#1
             exch   R2,0(R1)      ; atomic exchange
             bnez   R2,lockit     ; loop while locked
```

# User-Level Synchronization Using LL/SC

- What about MP with cache coherency?
  - Want to spin on cached copy to avoid full memory latency
  - Likely to get cache hits for such variables
- Problem: exchange includes write
  - Invalidates all other copies
  - Generates considerable bus traffic



# User-Level Synchronization Using LL/SC (cont'd)

- Solution to bus traffic: don't try exchange when you know it will fail
  - Keep reading cached copy
  - Lock release will invalidate

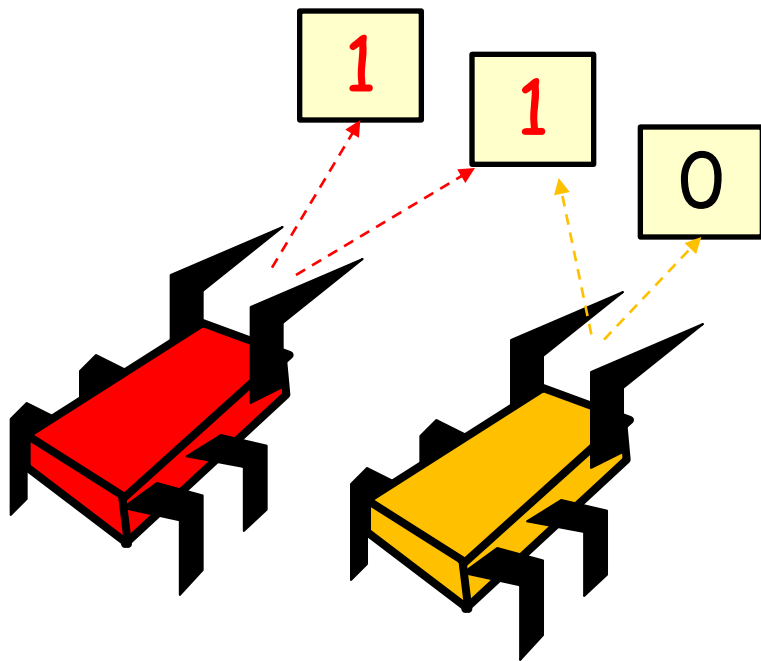
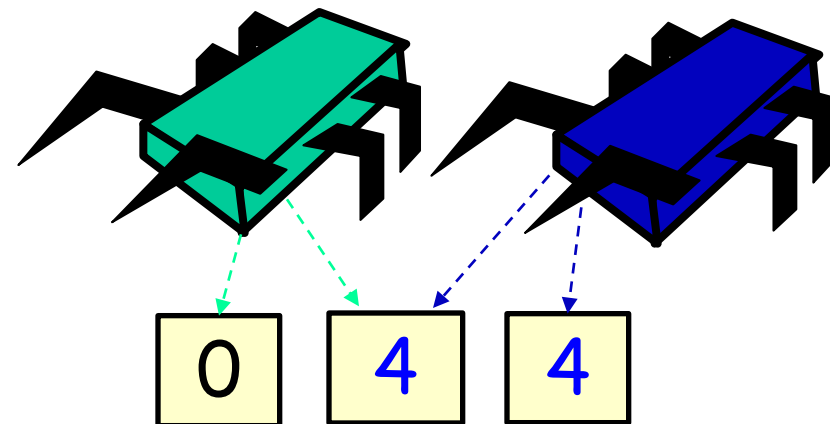
```
try:          li      R2,#1
lockit:      lw      R3,0(R1)      ;load old
             bnez   R3,lockit     ;≠ 0 ⇒ spin
             exch   R2,0(R1)     ;atomic exchange
             bnez   R2,try       ;spin on failure
```

# This work is licensed under a [Creative Commons Attribution-ShareAlike 2.5 License](https://creativecommons.org/licenses/by-sa/3.0/).

- **You are free:**
  - **to Share** — to copy, distribute and transmit the work
  - **to Remix** — to adapt the work
- **Under the following conditions:**
  - **Attribution.** You must attribute the work to “The Art of Multiprocessor Programming” (but not in any way that suggests that the authors endorse you or your use of the work).
  - **Share Alike.** If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.
- For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to
  - <http://creativecommons.org/licenses/by-sa/3.0/>.
- Any of the above conditions can be waived if you get permission from the copyright holder.
- Nothing in this license impairs or restricts the author's moral rights.

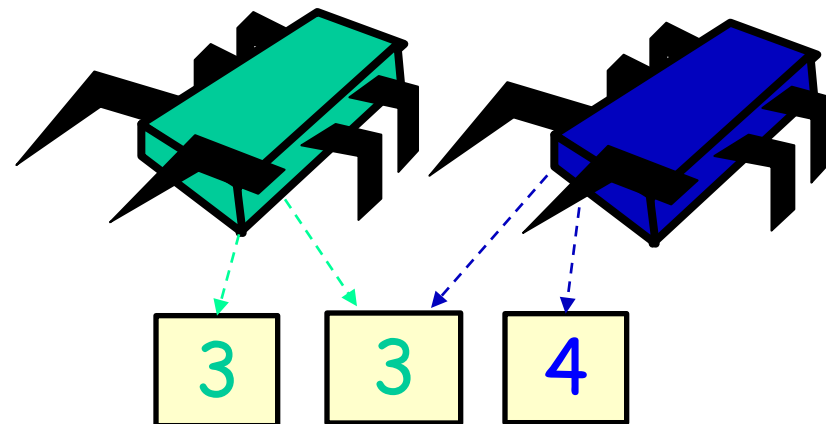
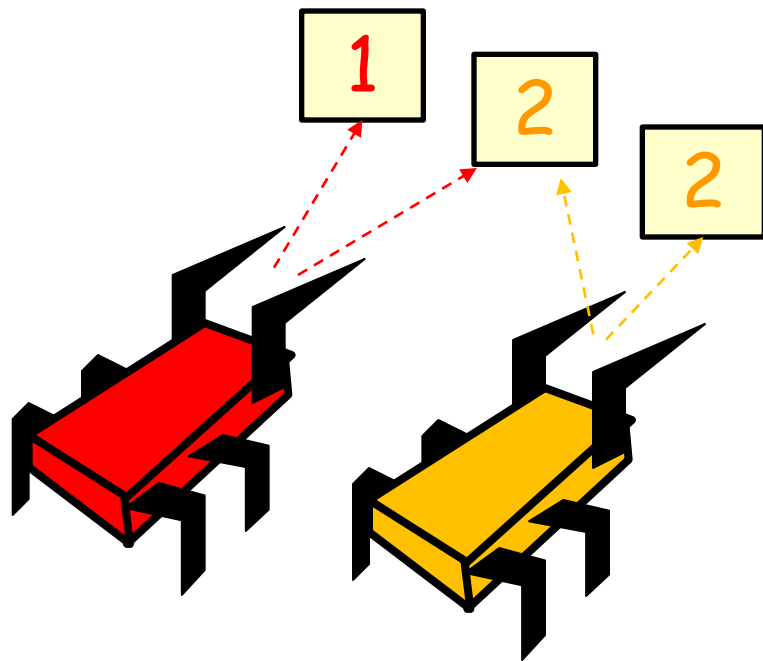
# Atomic 3-assignment $\Rightarrow$ 4-thread consensus

Phase 1 : 2 X 2-consensus



# Atomic 3-assignment => 4-thread consensus

Phase 1 : 2 X 2-consensus

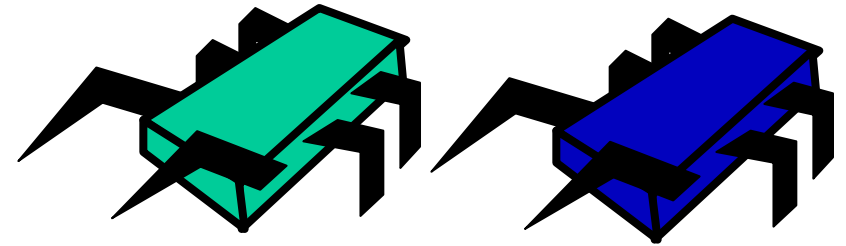


4 won !

1 won !

# Atomic 3-assignment $\Rightarrow$ 4-thread consensus

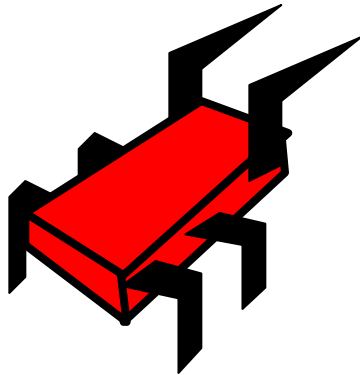
Phase 2 :  
2 group consensus



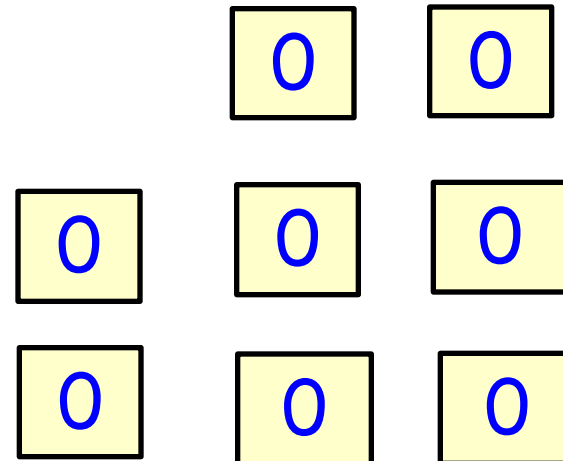
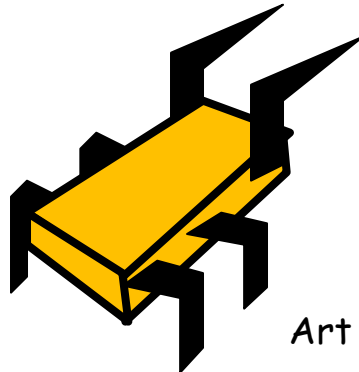
Writes 4

Writes 4

Writes 1

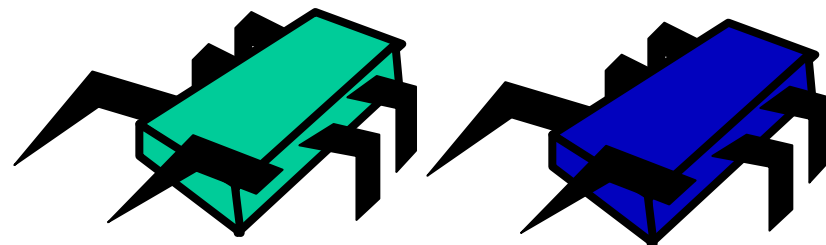


Writes 1

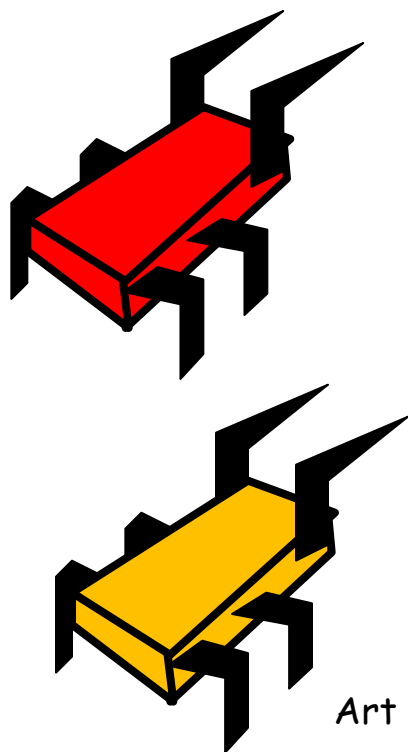


# Atomic 3-assignment $\Rightarrow$ 4-thread consensus

Phase 2 :  
2 group consensus



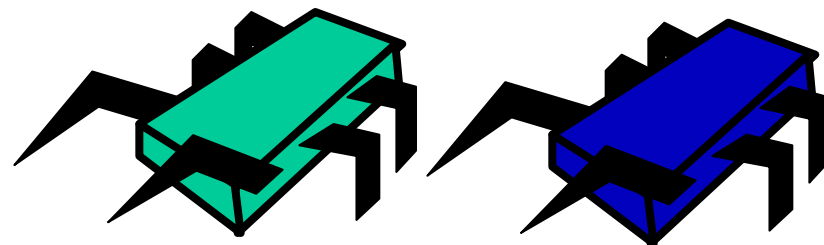
Writes 1



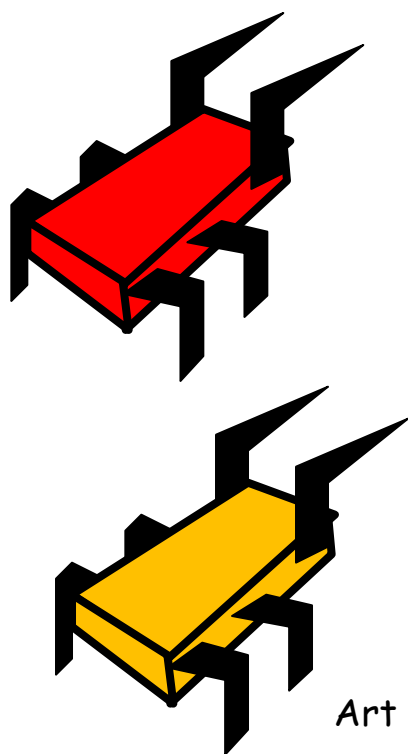
	0	0
1	1	1
0	0	0

# Atomic 3-assignment $\Rightarrow$ 4-thread consensus

Phase 2 :  
2 group consensus



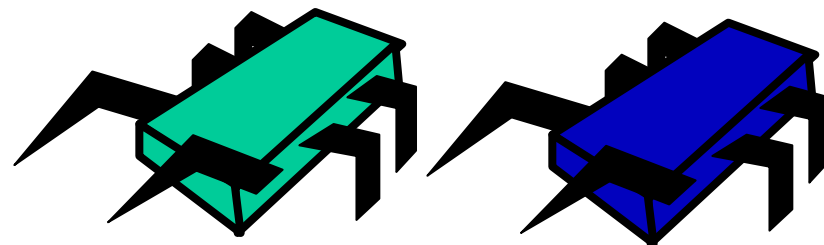
Writes 4



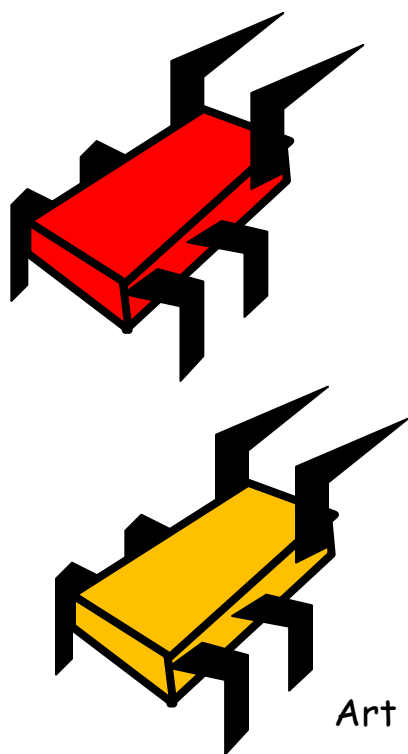
	4	0
1	4	1
0	4	0

# Atomic 3-assignment $\Rightarrow$ 4-thread consensus

Phase 2 :  
2 group consensus



Writes 4

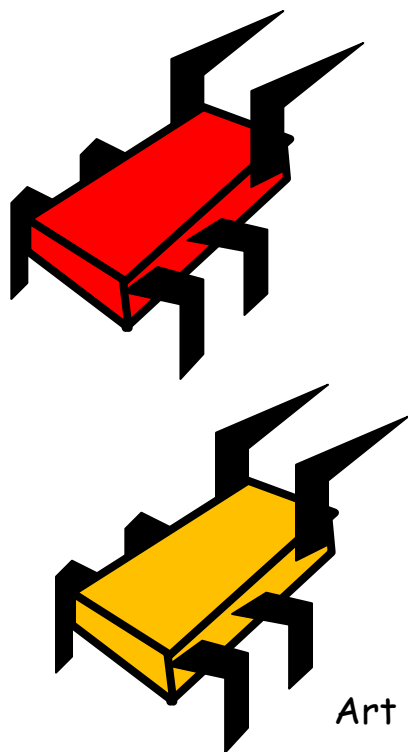
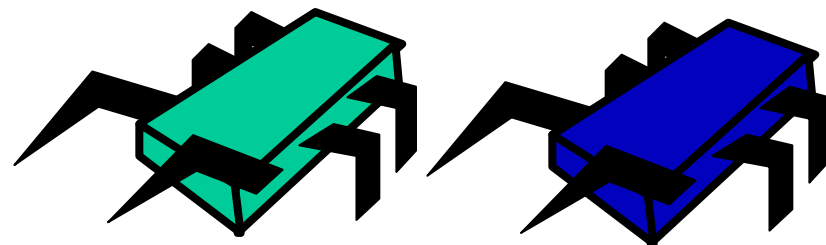


	4	4
1	4	4
0	4	4



# Atomic 3-assignment $\Rightarrow$ 4-thread consensus

Phase 2 :  
2 group consensus

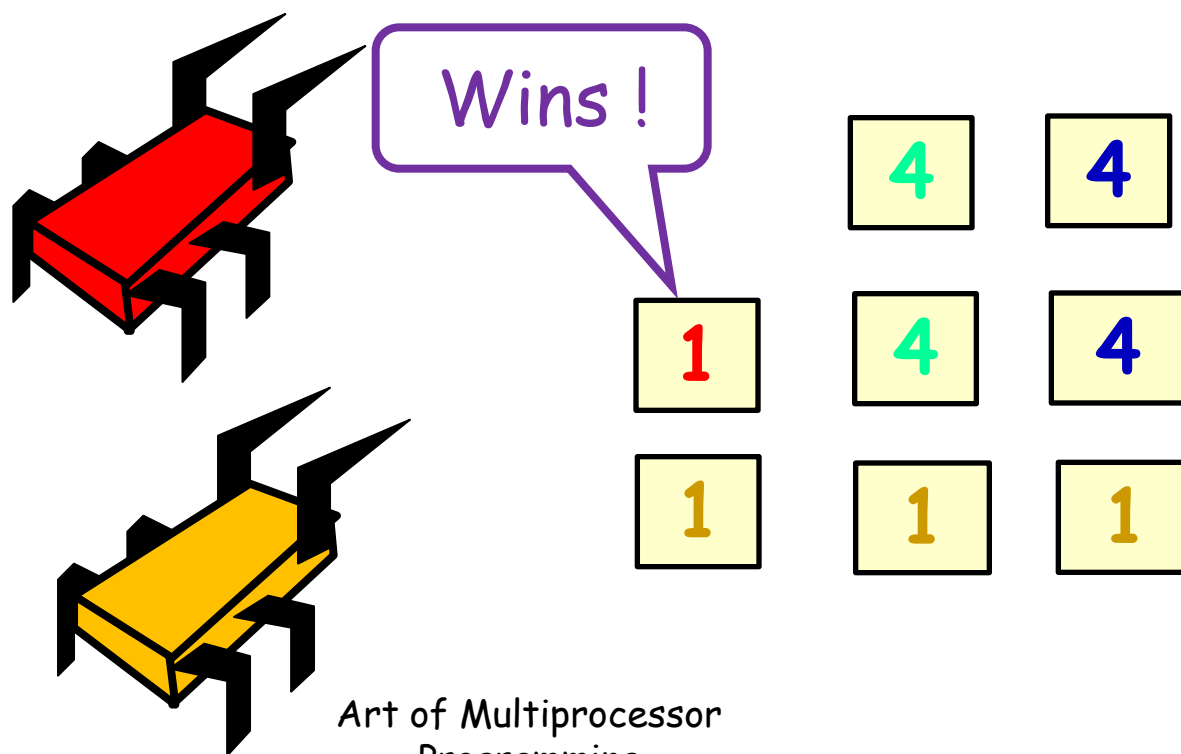
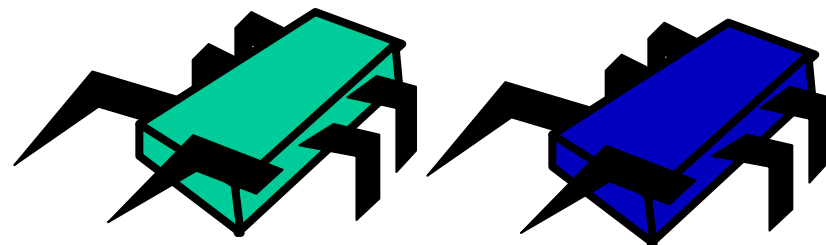


Writes 1

	4	4
1	4	4
1	1	1

# Atomic 3-assignment => 4-thread consensus

Phase 2 :  
2 group consensus



Atomic 64 bit-assignment =>

k-thread consensus ?

byte addressing ?

# Atomic 64 bit-assignment (masking)

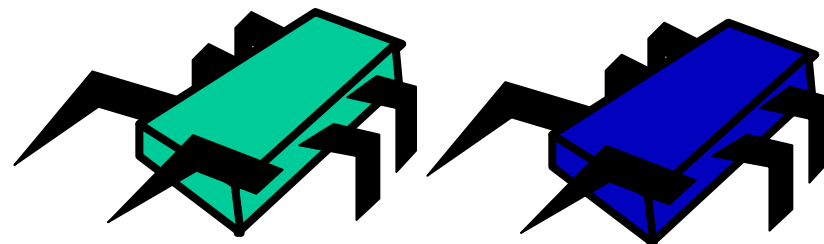
=>

## 4-thread consensus

1	0	0	0	1	1	1	0			
	2	0	0	2	0	0	2	2		
		3	0	0	3	0	3	0	3	
			4	0	0	4	0	4	4	0
1	2	3	4	12	13	14	23	24	34	

# Atomic 3-assignment $\Rightarrow$ 4-thread consensus

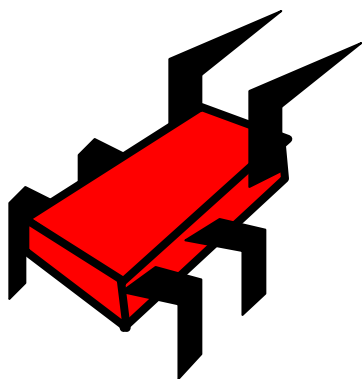
Phase 2 :  
2 group consensus



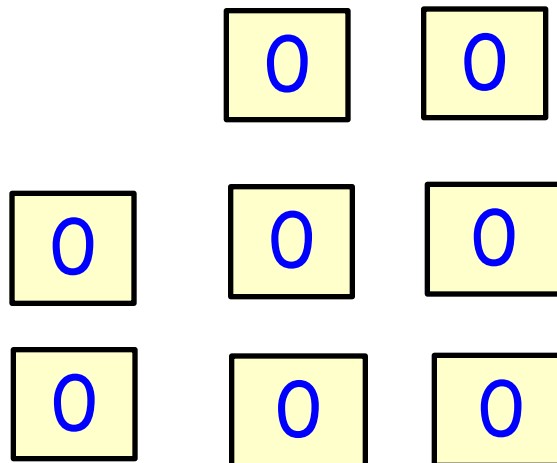
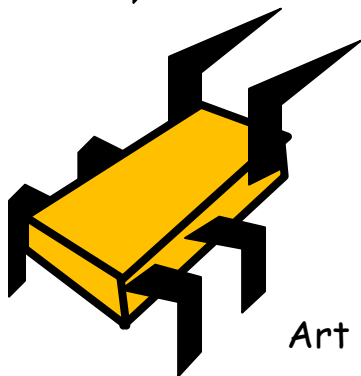
Writes 4

Writes 4

Writes 1



Writes 1

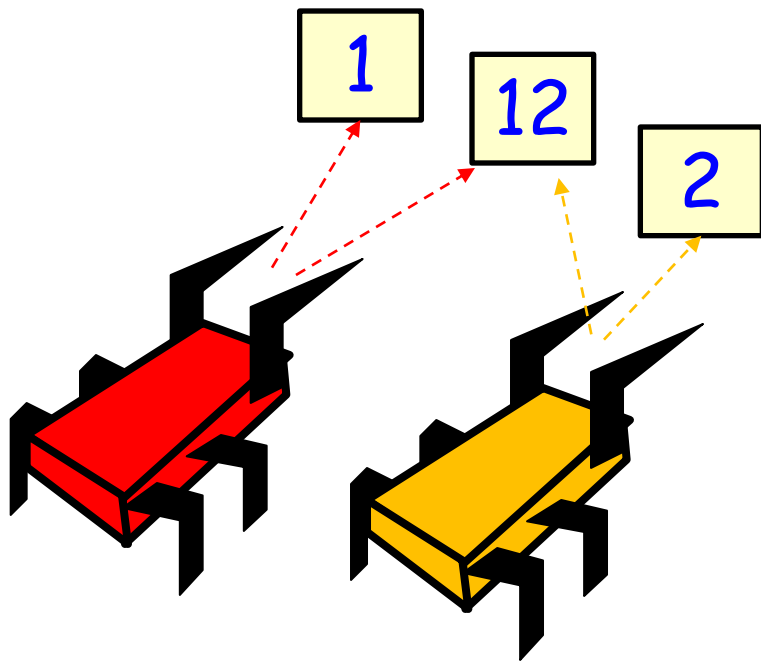
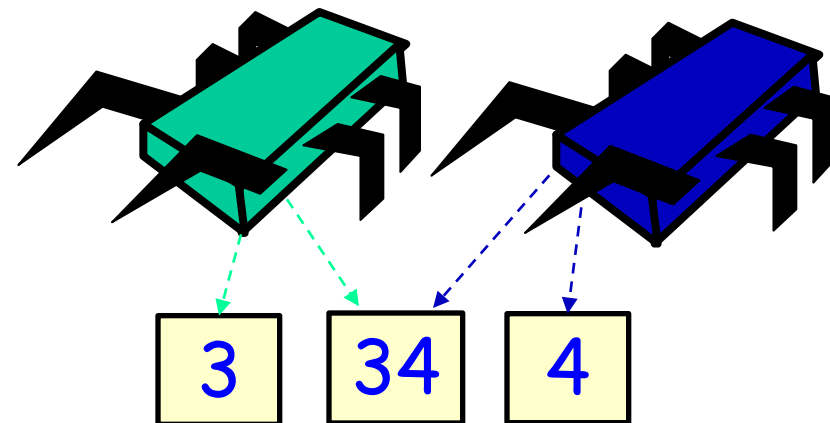


# 4 thread consensus using 2-assignment ?

- Arrange threads in 2 groups/2 phases
- Each group selects a leader using 2-assignment
- The leaders vote using 2-assignment to select the global leader

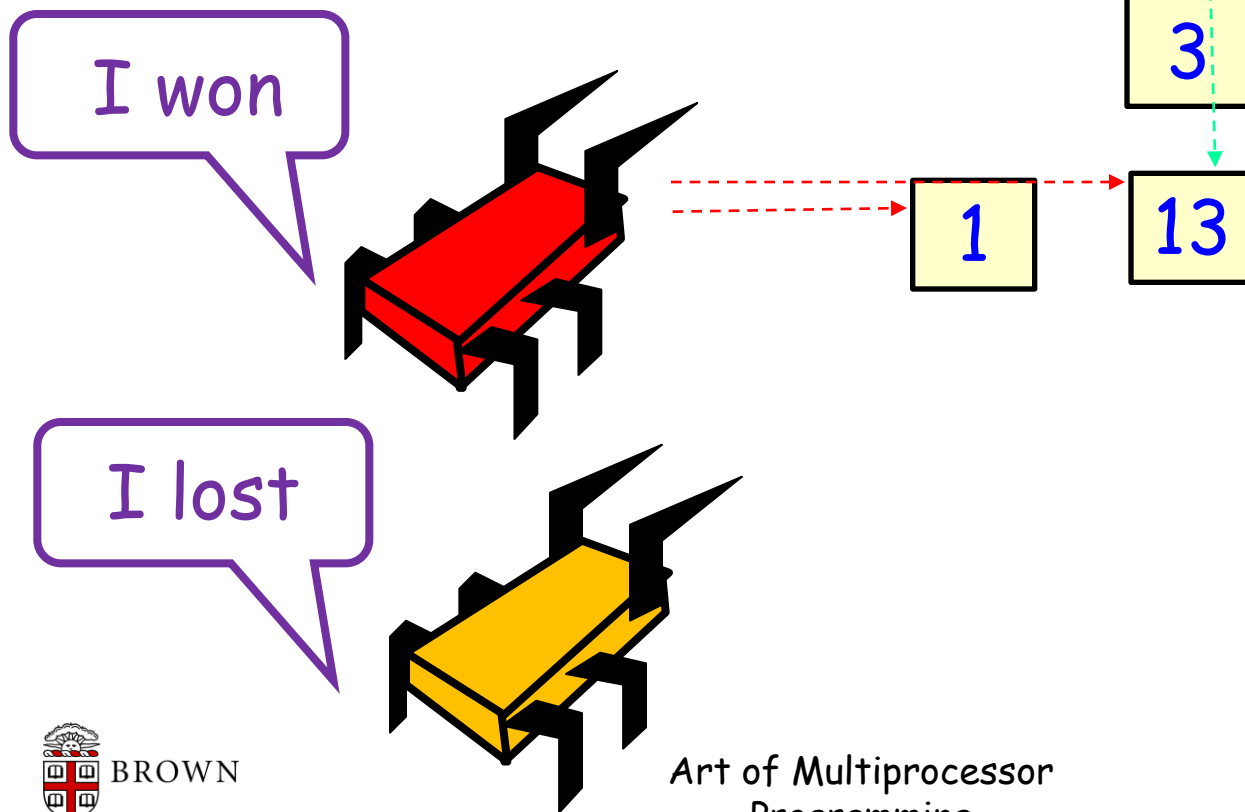
# Atomic 2-assignment $\Rightarrow$ 4-thread consensus

Phase 1 : 2 X 2-consensus



# Atomic 2-assignment $\Rightarrow$ 4-thread consensus

Phase 2 :  
leader consensus

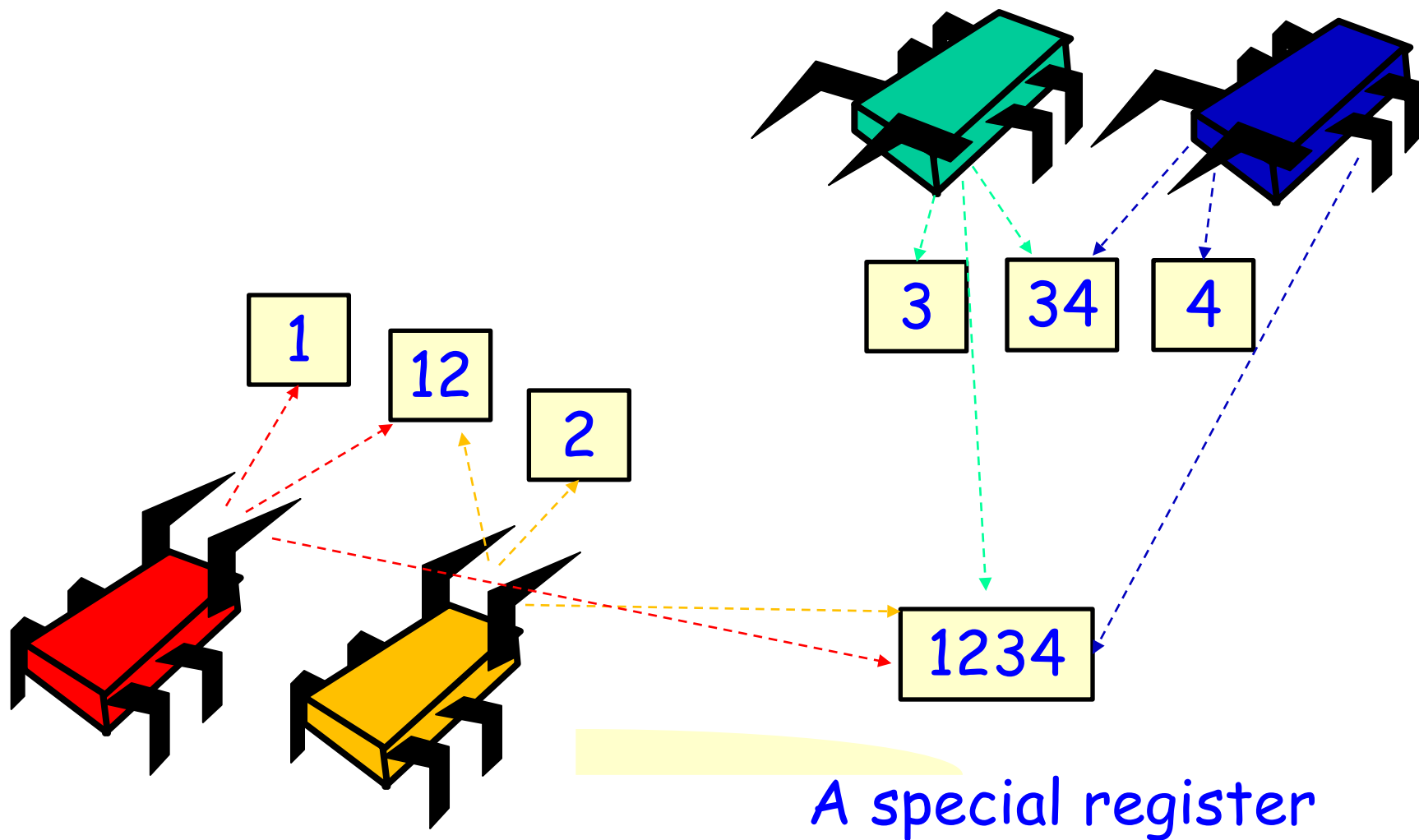




# 4 thread consensus using 2-assignment ?

- What if the global winner dies right after it won ?
  - Examining the registers will reveal it !
- What if both winners did not make it to the 2<sup>nd</sup> phase ?
  - Nobody knows who won !

# Atomic 3-assignment $\Rightarrow$ 4-thread consensus



# 4 thread consensus using 3-assignment ?

- The special register(1234) knows who wrote last.
- Two registers (12), (34) know who won in each group.
- However, it would be impossible who was the first !
- We can first choose the group which did not write last and choose the leader of the group. --- really ?

# 4 thread consensus using 3-assignment ?

- Let's assume the output is (112, 334, 1)
- We know 2→1, 4→3 & 3→1
- Possible cases :
  - 2→4→3→1, 4→2→3→1, 4→3→2→1
- Can we just say 4 wins in this case ?
  
- If 2,4 participate and the output is (022,044,4), then they have to decide 2 won.
- However, 1,3 later participate (112,334,1) then what ?