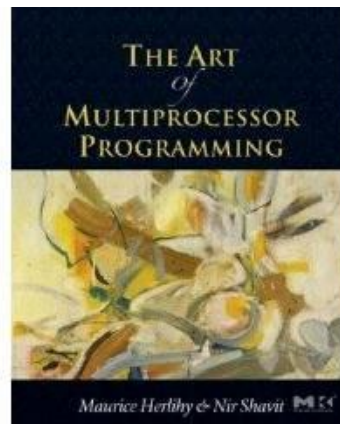
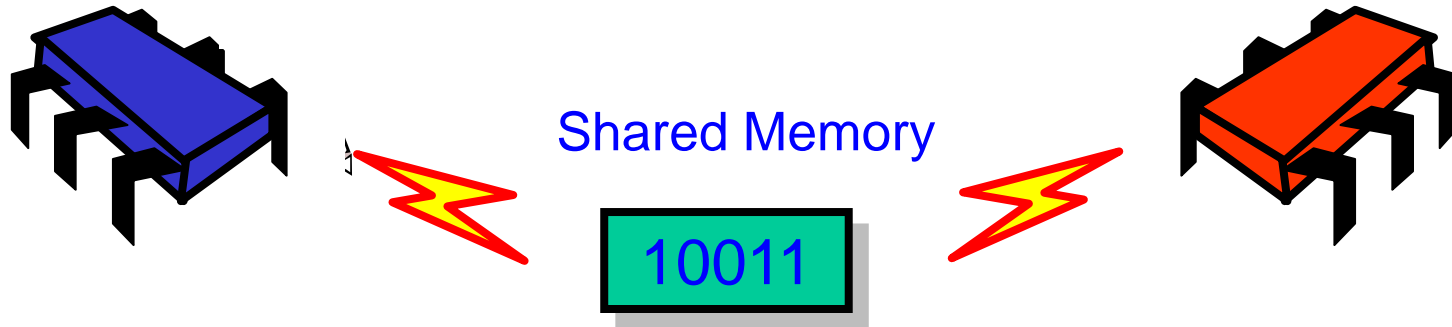


The Relative Power of Synchronization Operations

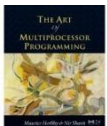


Companion slides for
The Art of Multiprocessor Programming
by Maurice Herlihy & Nir Shavit

Shared-Memory Computability



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



Wait-Free Implementation

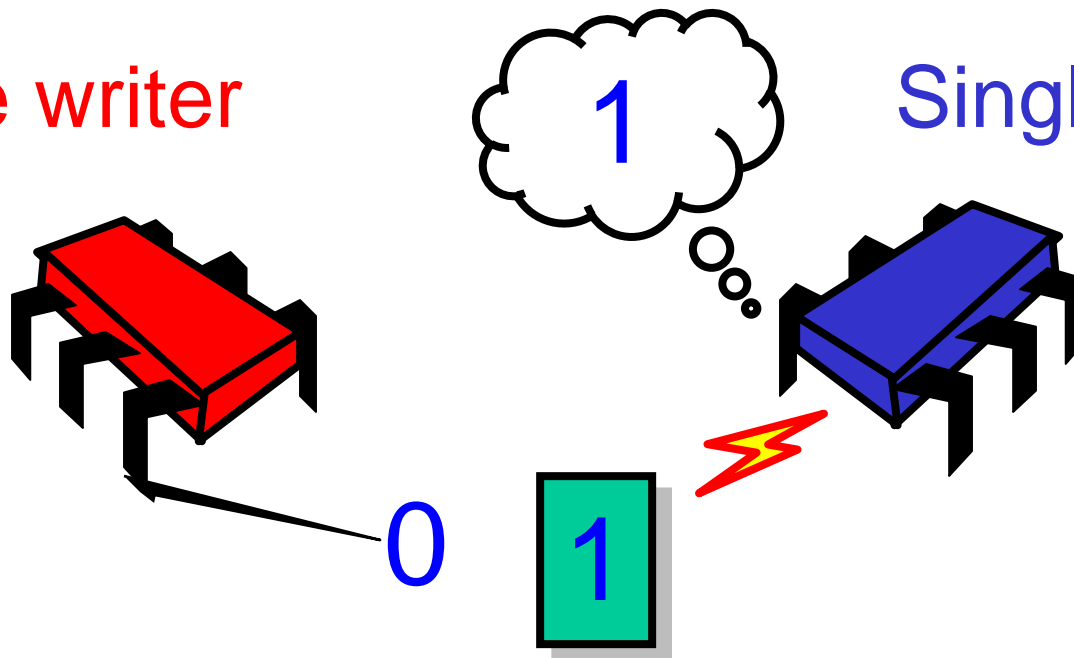
- Every method call completes in finite number of steps
- Implies no mutual exclusion



From Weakest Register

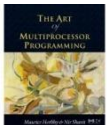
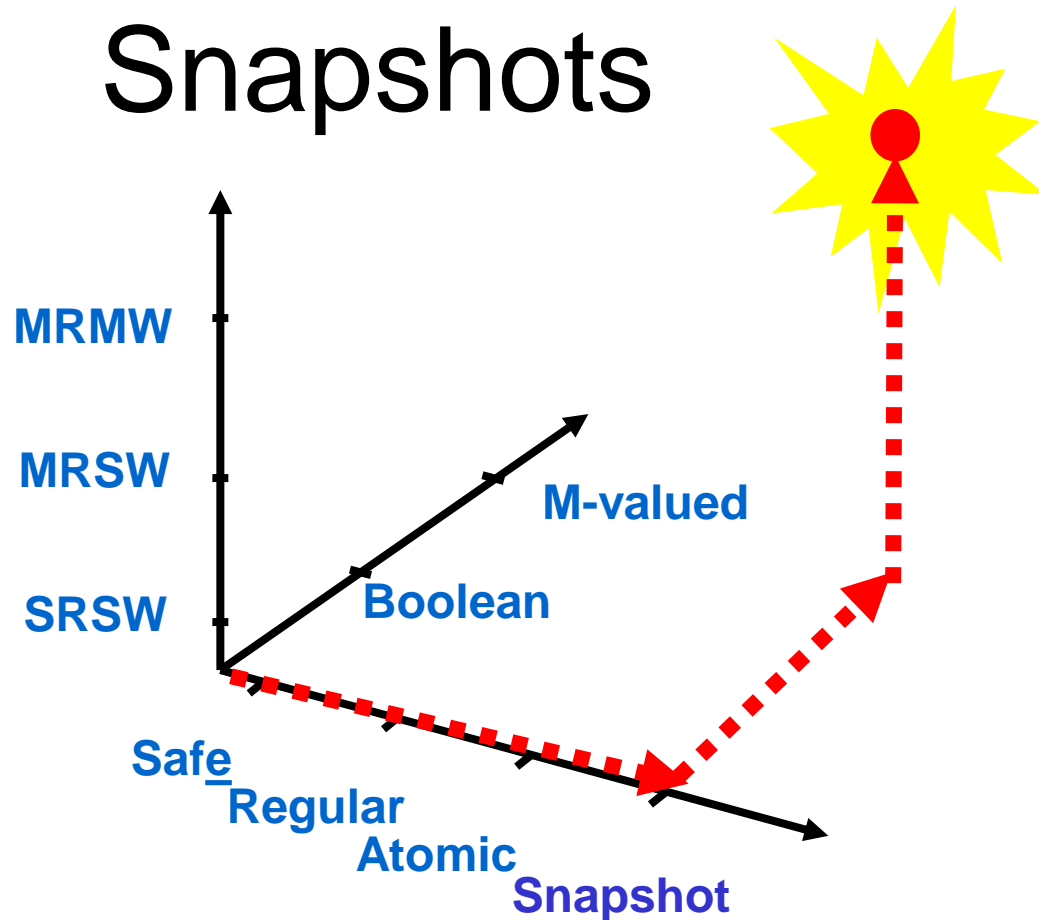
Single writer

Single reader



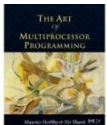
Safe Boolean register

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



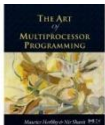
Rationale for wait-freedom

- We wanted atomic registers to implement mutual exclusion



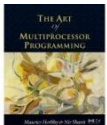
Rationale for wait-freedom

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

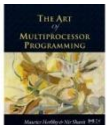


Rationale for wait-freedom

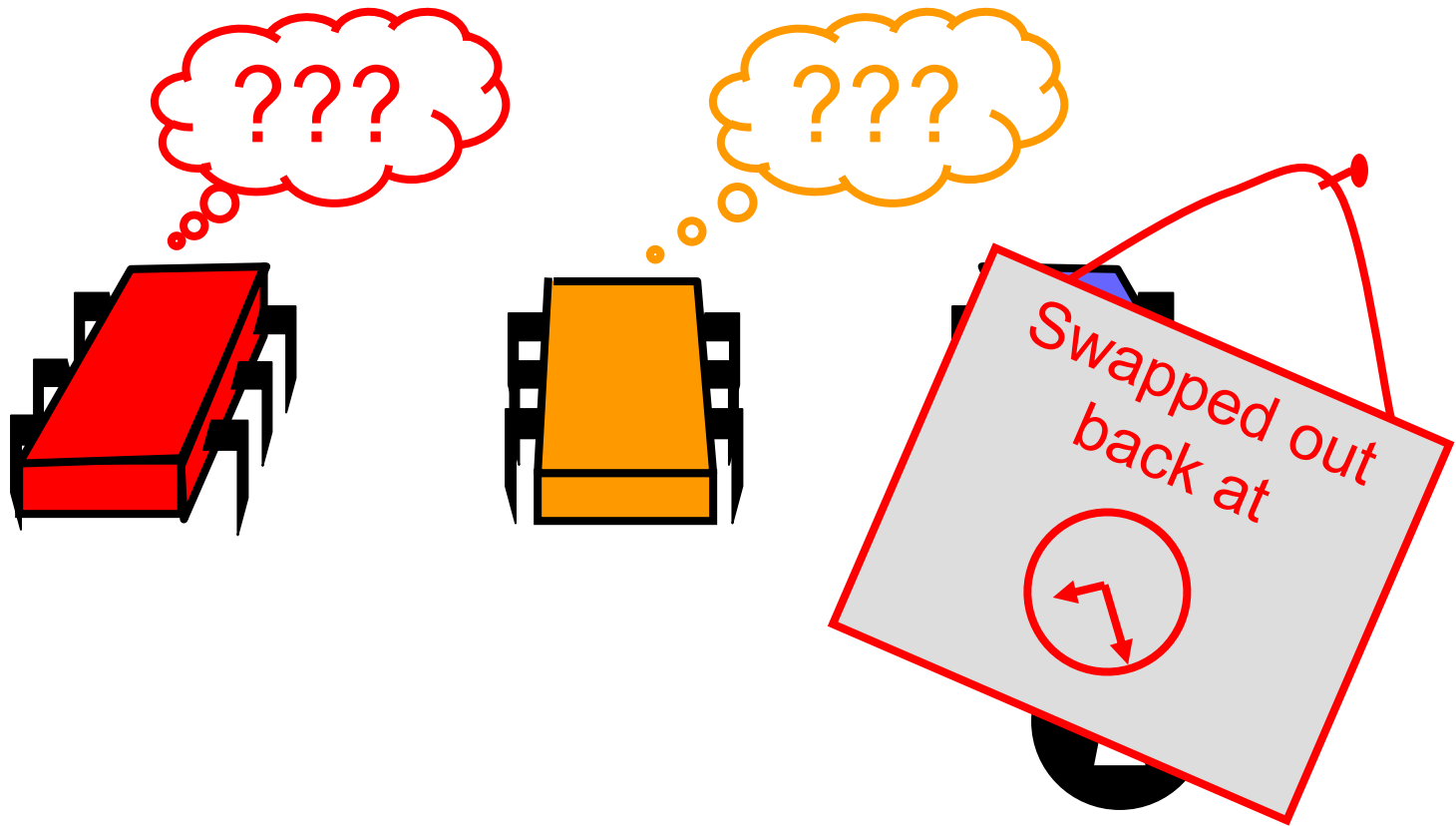
- We wanted atomic registers to implement mutual exclusion
- So we couldn't use mutual exclusion to implement atomic registers
- **But wait, there's more!**



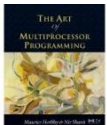
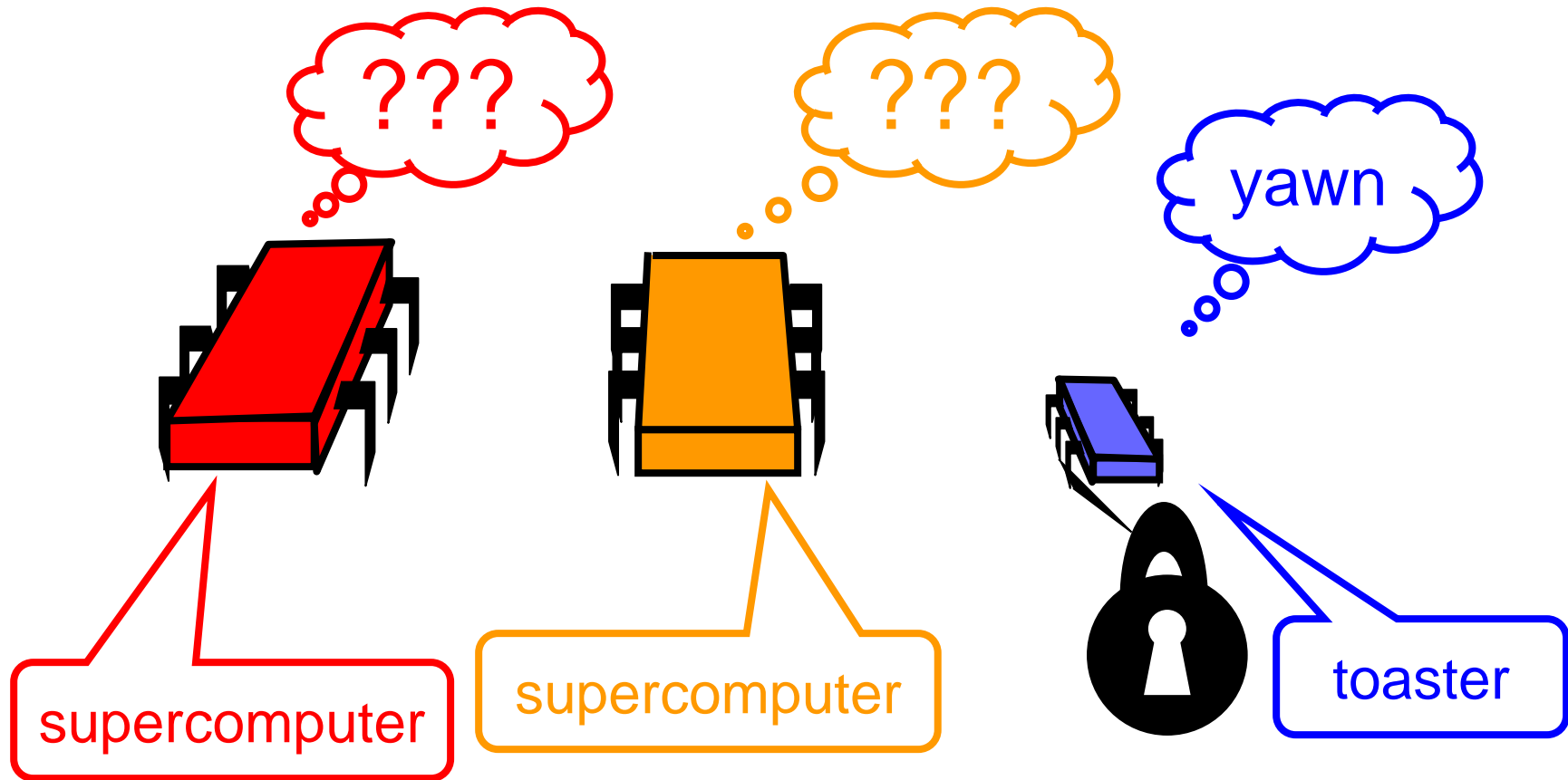
Why is Mutual Exclusion so wrong?



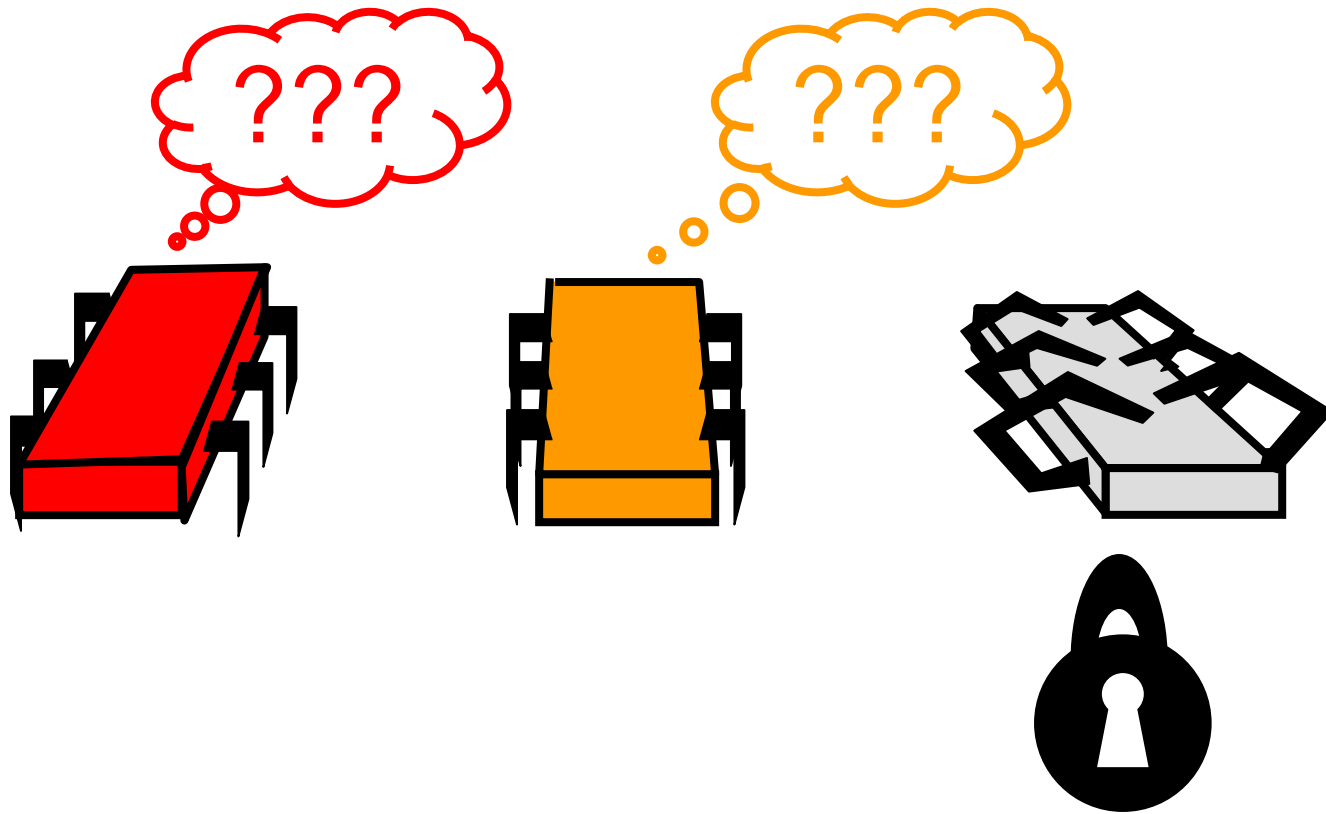
Asynchronous Interrupts



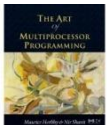
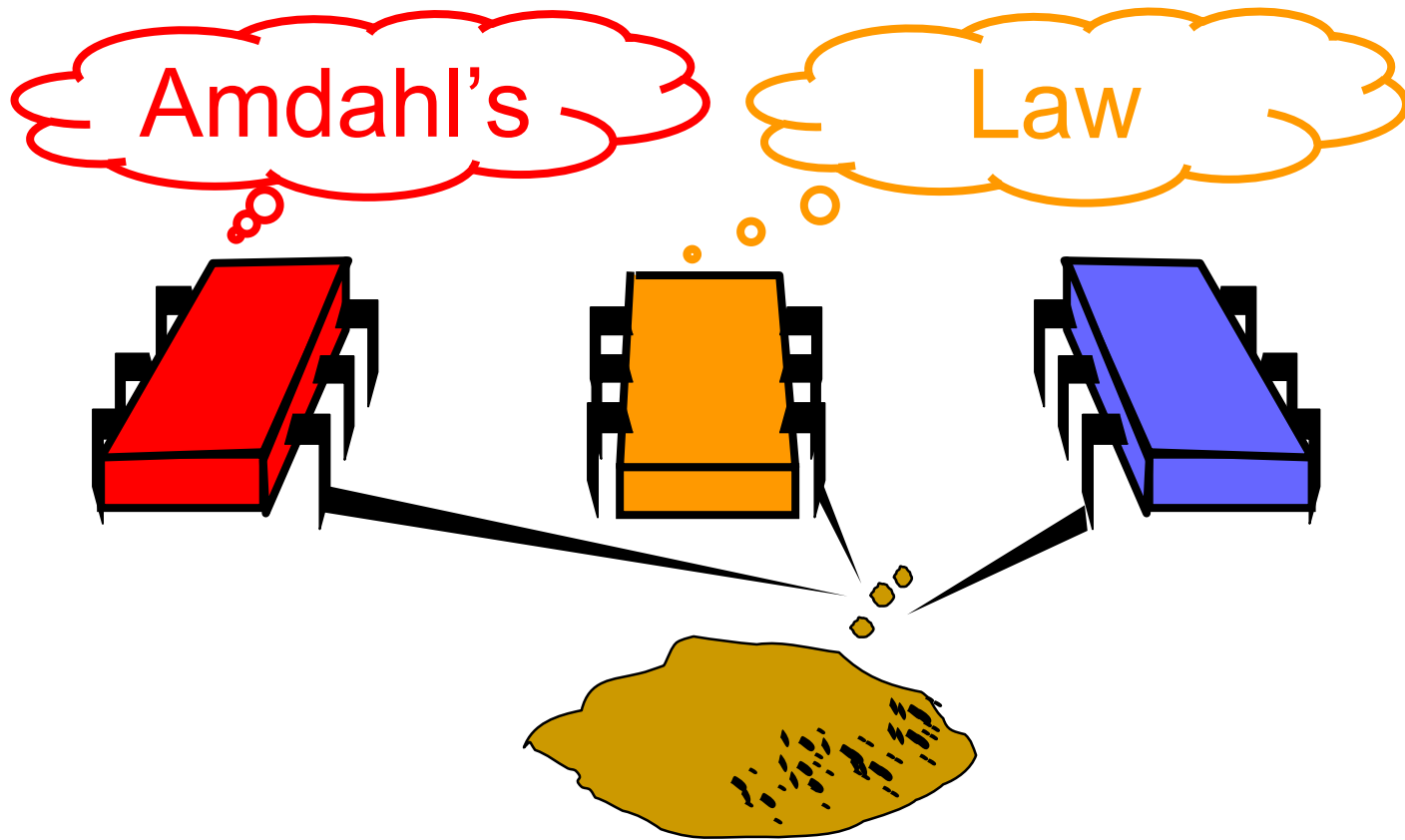
Heterogeneous Processors



Fault-tolerance

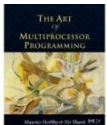


Machine Level Instruction Granularity



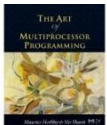
Basic Questions

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



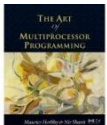
Basic Questions

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



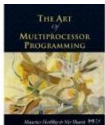
Basic Questions

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



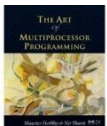
Basic Questions

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

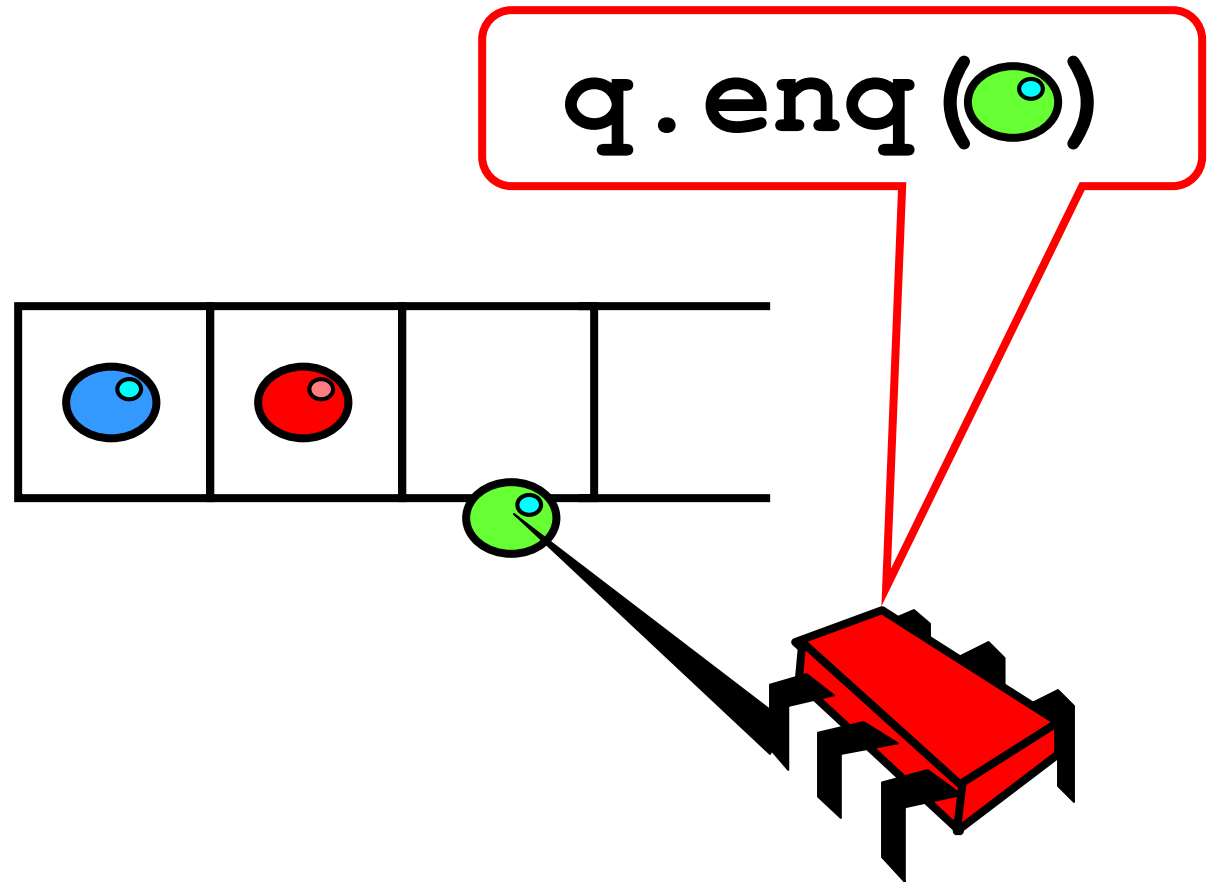


Basic Questions

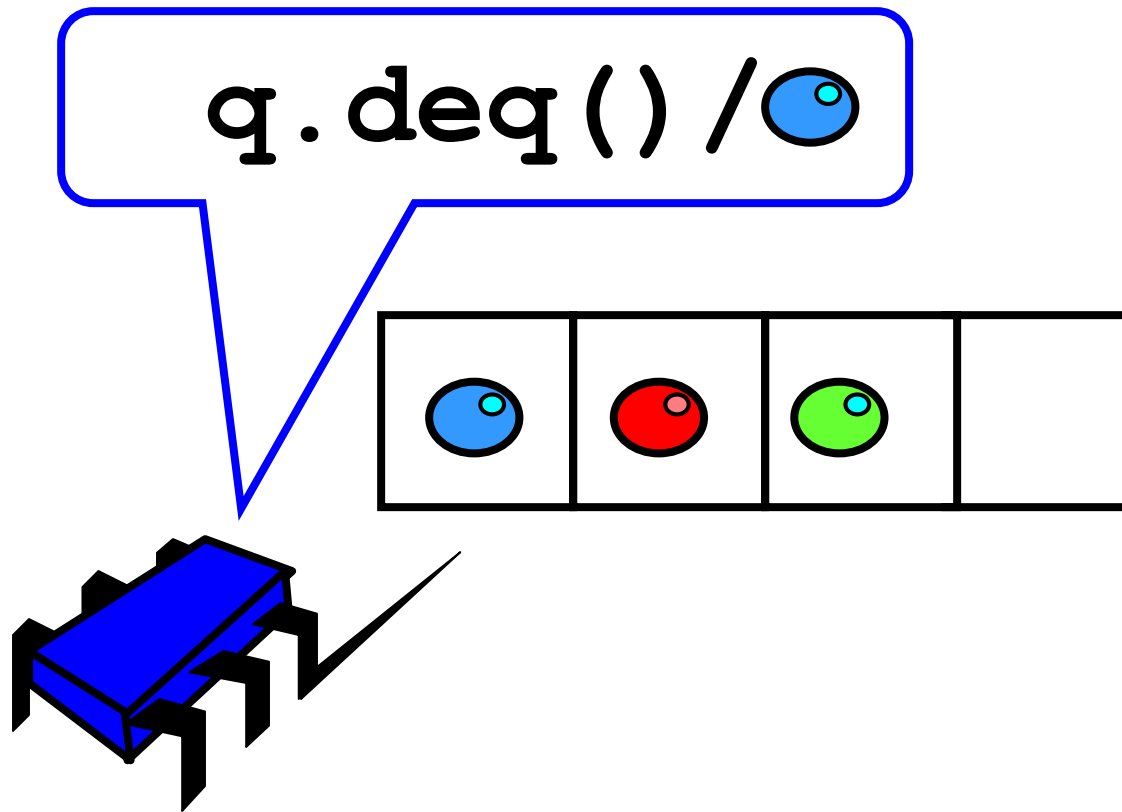
- Wait-Free synchronization might be a good idea in principle
- But how do you do it ...
 - Systematically?
 - Correctly?
 - Efficiently?



FIFO Queue: Enqueue Method

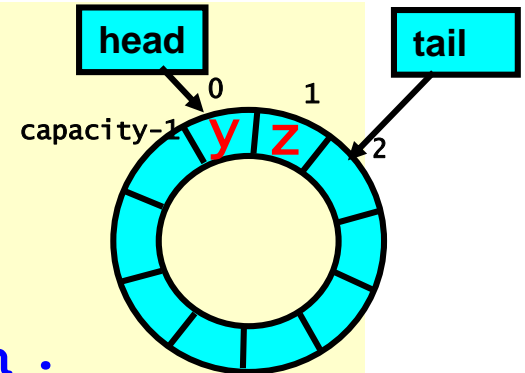


FIFO Queue: Dequeue Method

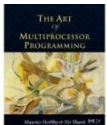
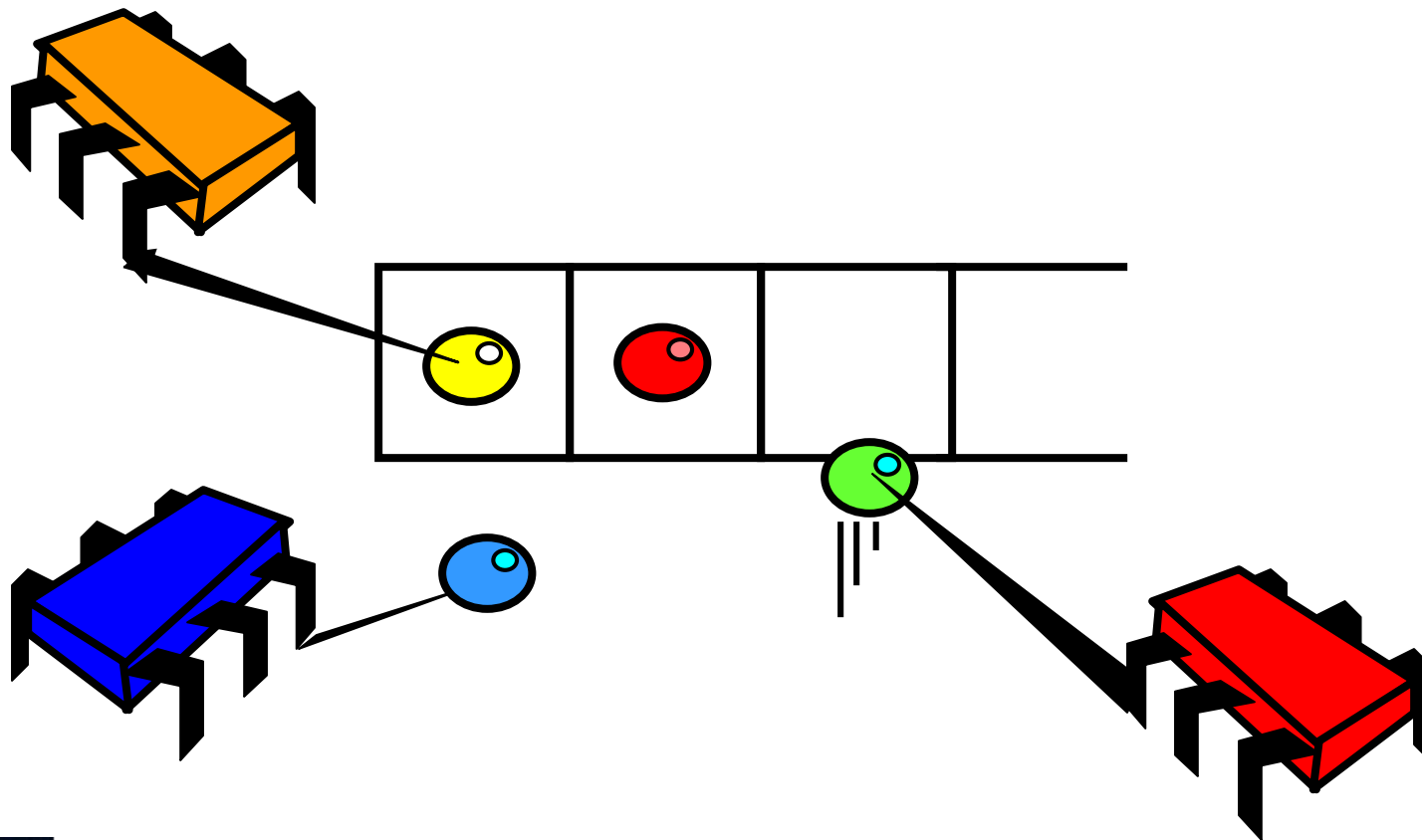


Two-Thread Wait-Free Queue

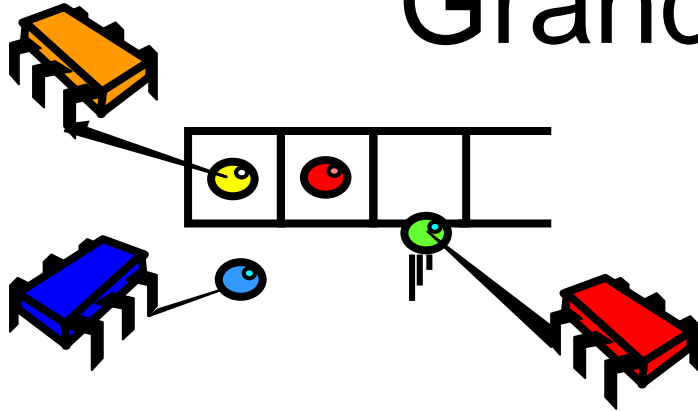
```
public class WaitFreeQueue {
    int head = 0, tail = 0;
    Item[QSIZE] items;
    public void enq(Item x) {
        while (tail-head == QSIZE) {};
        items[tail % QSIZE] = x; tail++;
    }
    public Item deq() {
        while (tail-head == 0) {}
        Item item = items[head % QSIZE];
 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



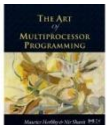
What About Multiple Dequeueers?



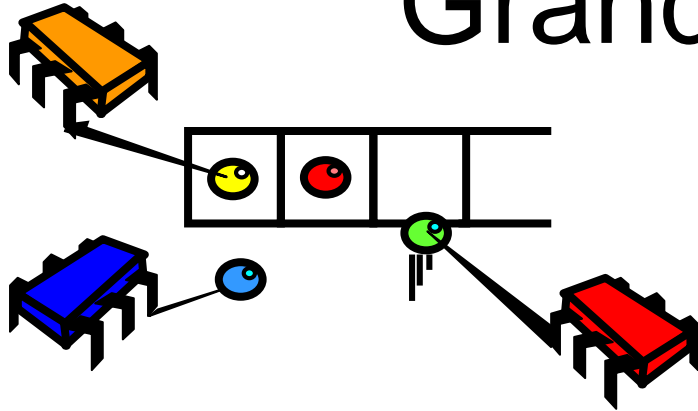
Grand Challenge



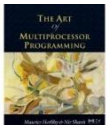
- Implement a FIFO queue



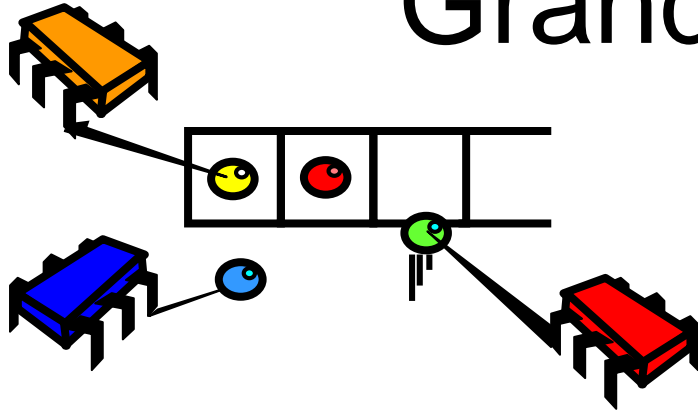
Grand Challenge



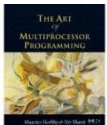
- Implement a FIFO queue
 - Wait-free



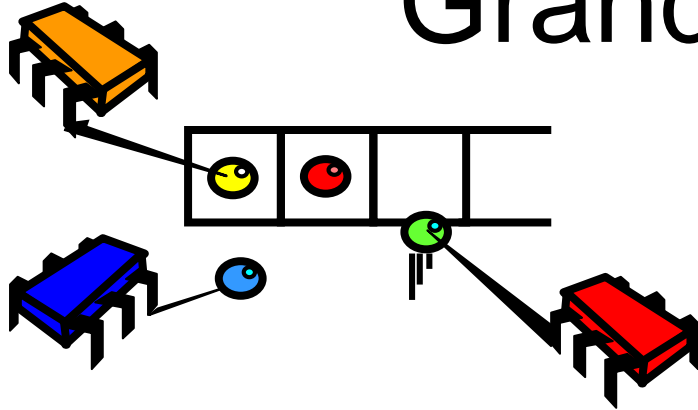
Grand Challenge



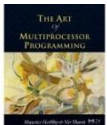
- Implement a FIFO queue
 - Wait-free
 - Linearizable



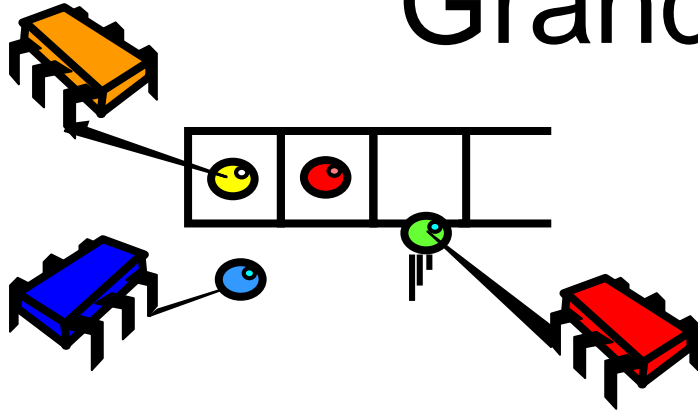
Grand Challenge



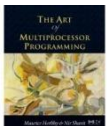
- Implement a FIFO queue
 - Wait-free
 - Linearizable
 - From atomic read-write registers



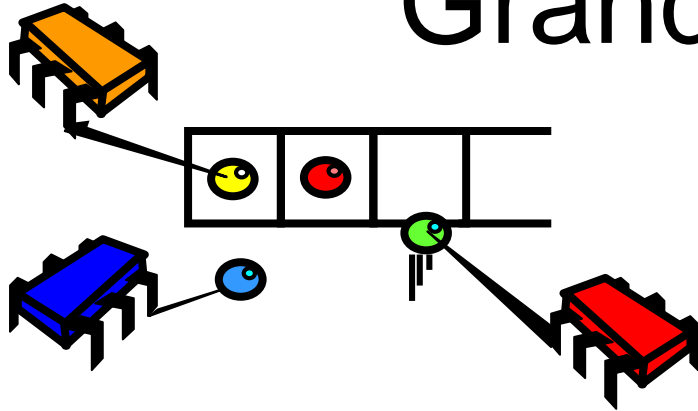
Grand Challenge



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

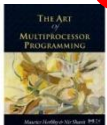


Grand Challenge



Only new
aspect

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

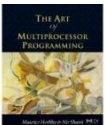


Puzzle

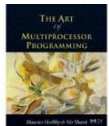
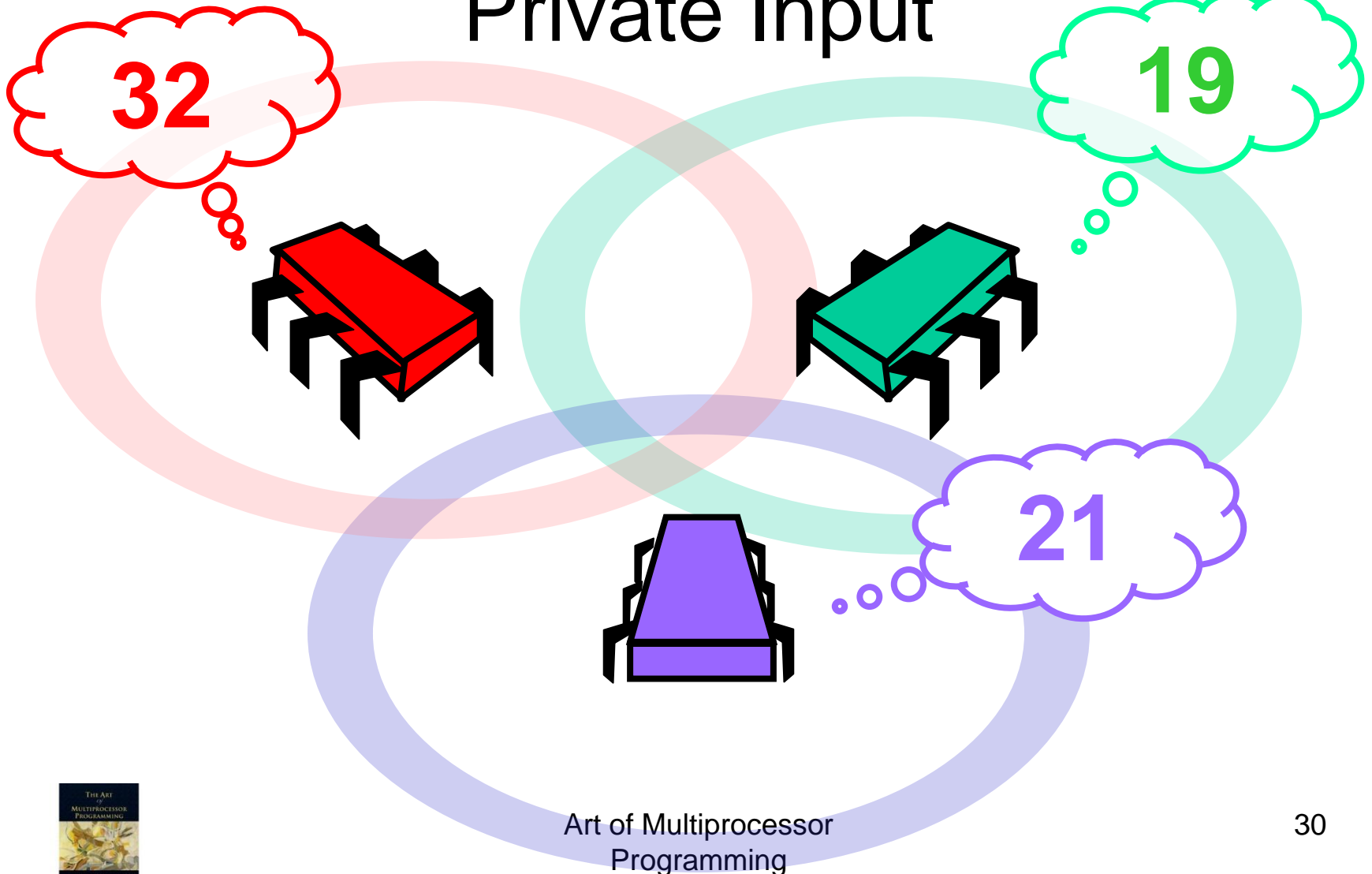
While you are ruminating on the grand challenge ...

We will give you another puzzle ...

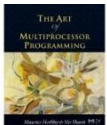
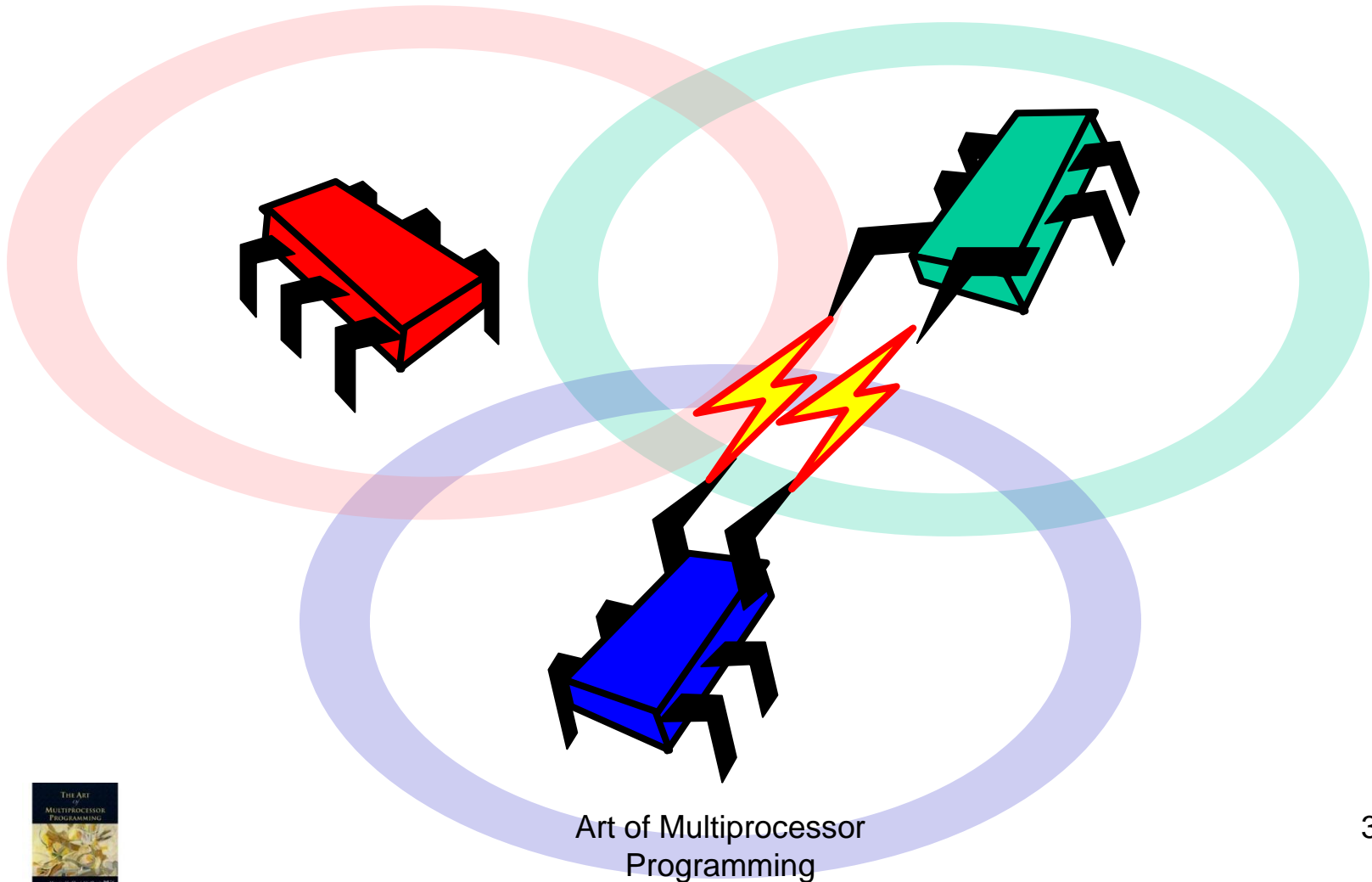
Consensus!



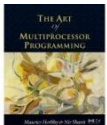
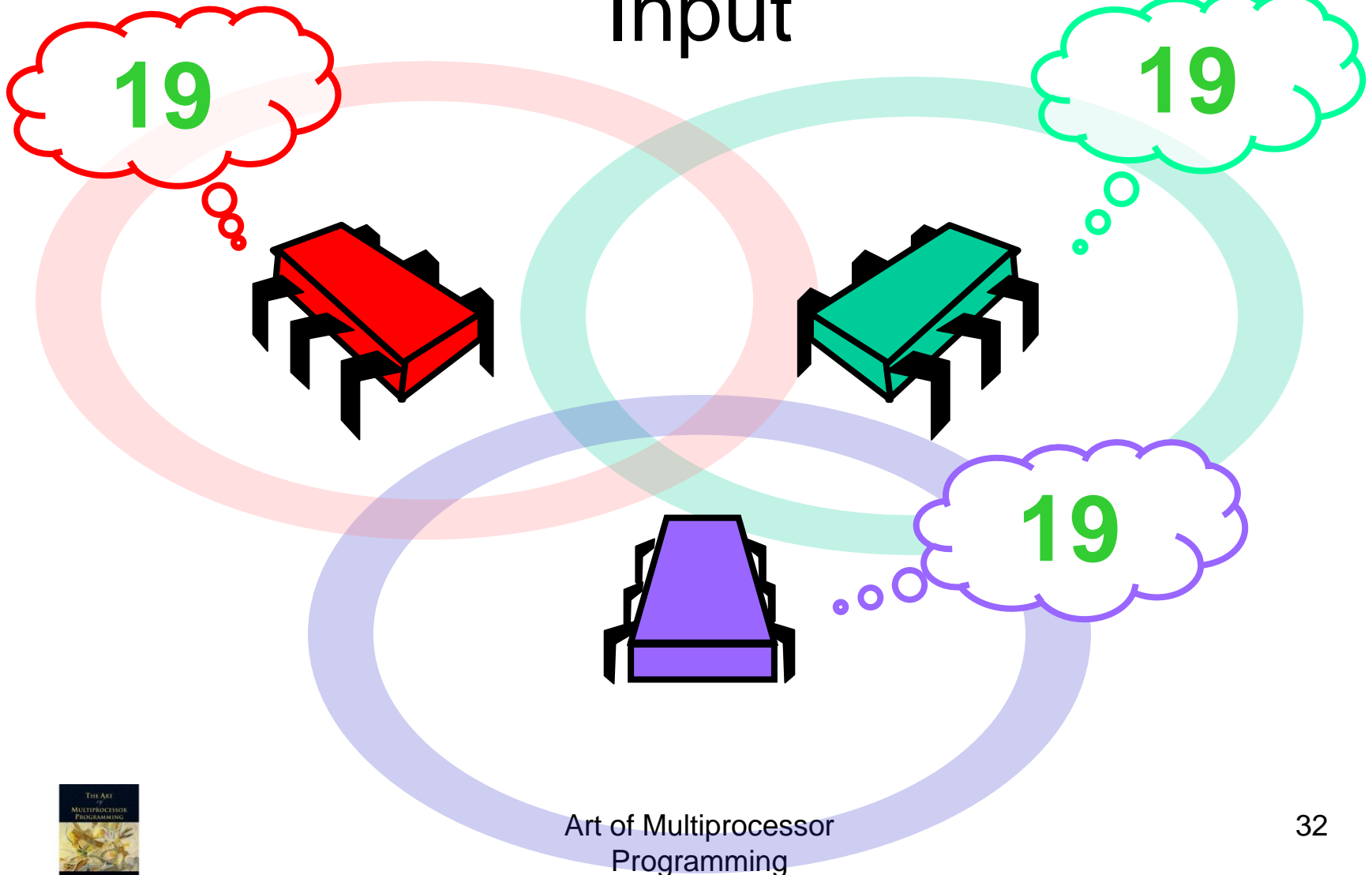
Consensus: Each Thread has a Private Input



They Communicate

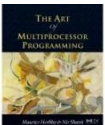


They Agree on One Thread's Input



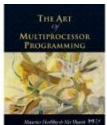
Formally: Consensus

- Consistent:
 - all threads decide the same value

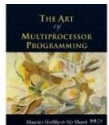
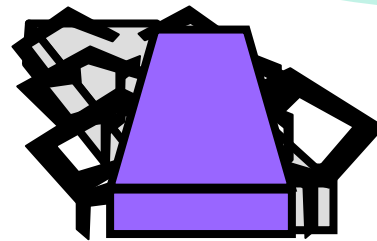
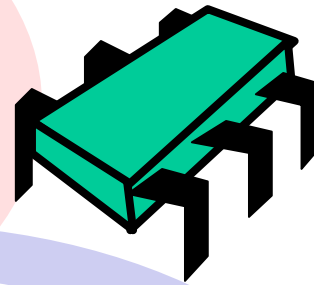
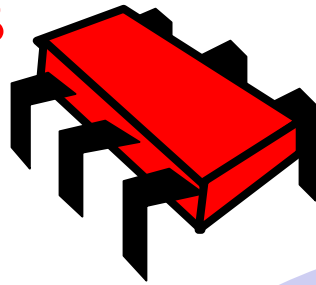


Formally: Consensus

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

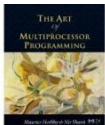


No Wait-Free Implementation of Consensus using Registers



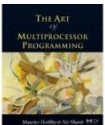
Formally

- 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 ...

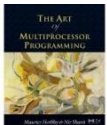
Derive a contradiction

Quod

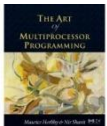
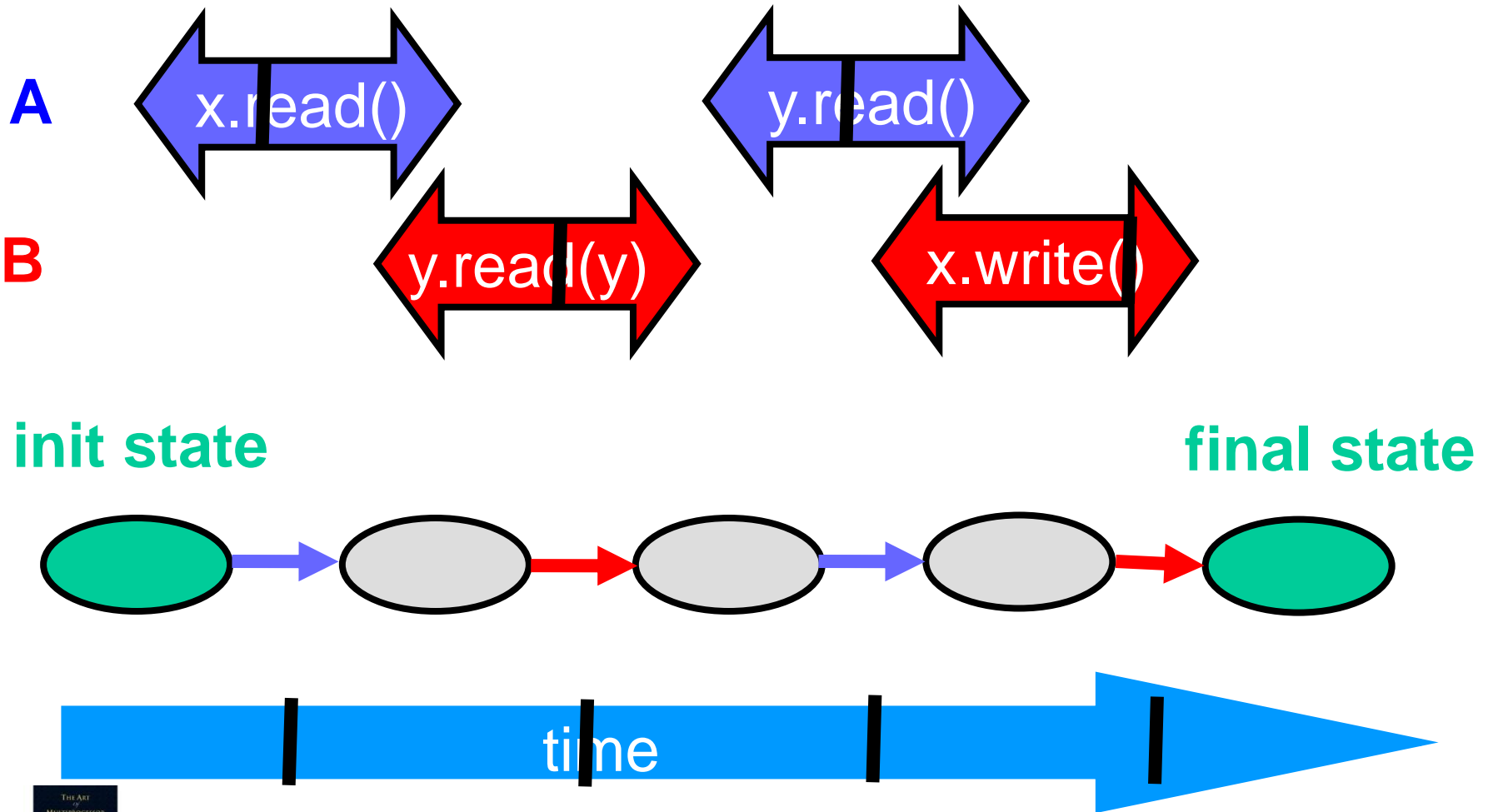
Erat

Demonstrandum

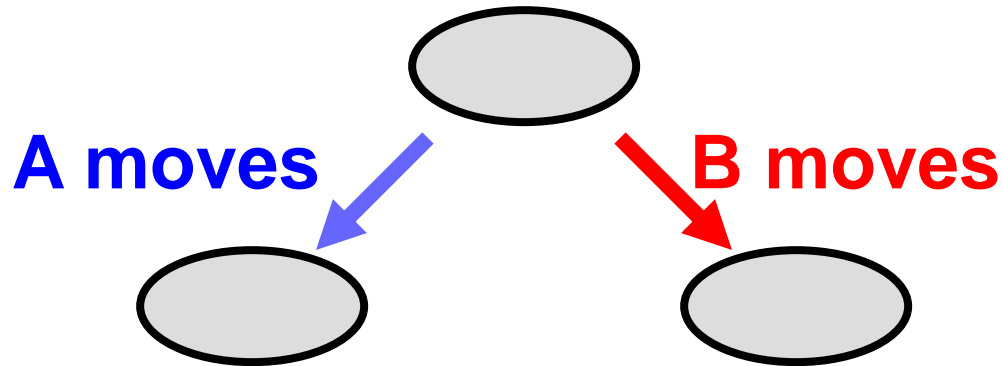
Enough to consider binary
consensus and $n=2$



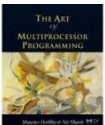
Protocol Histories as State Transitions



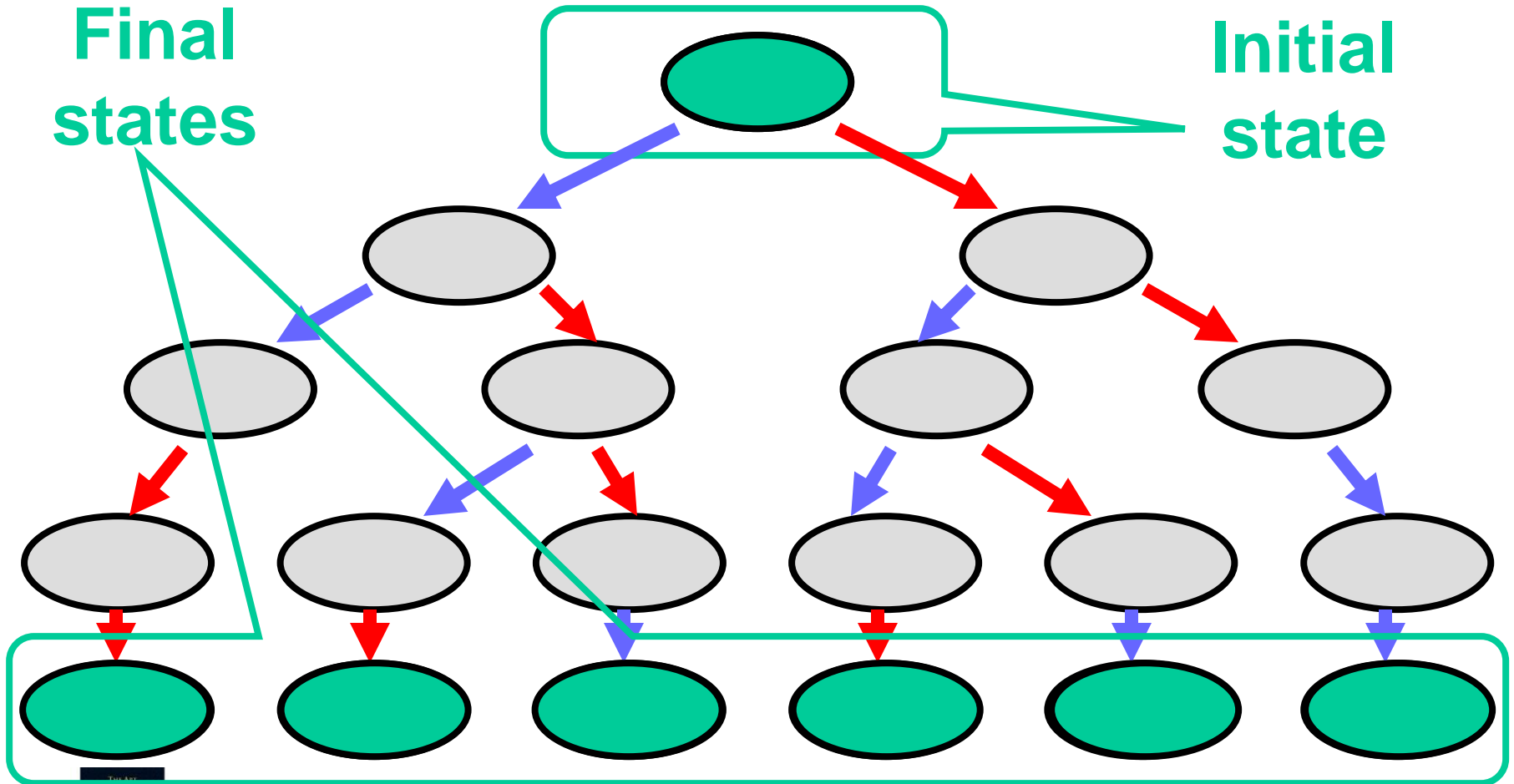
Wait-Free Computation



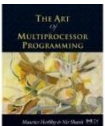
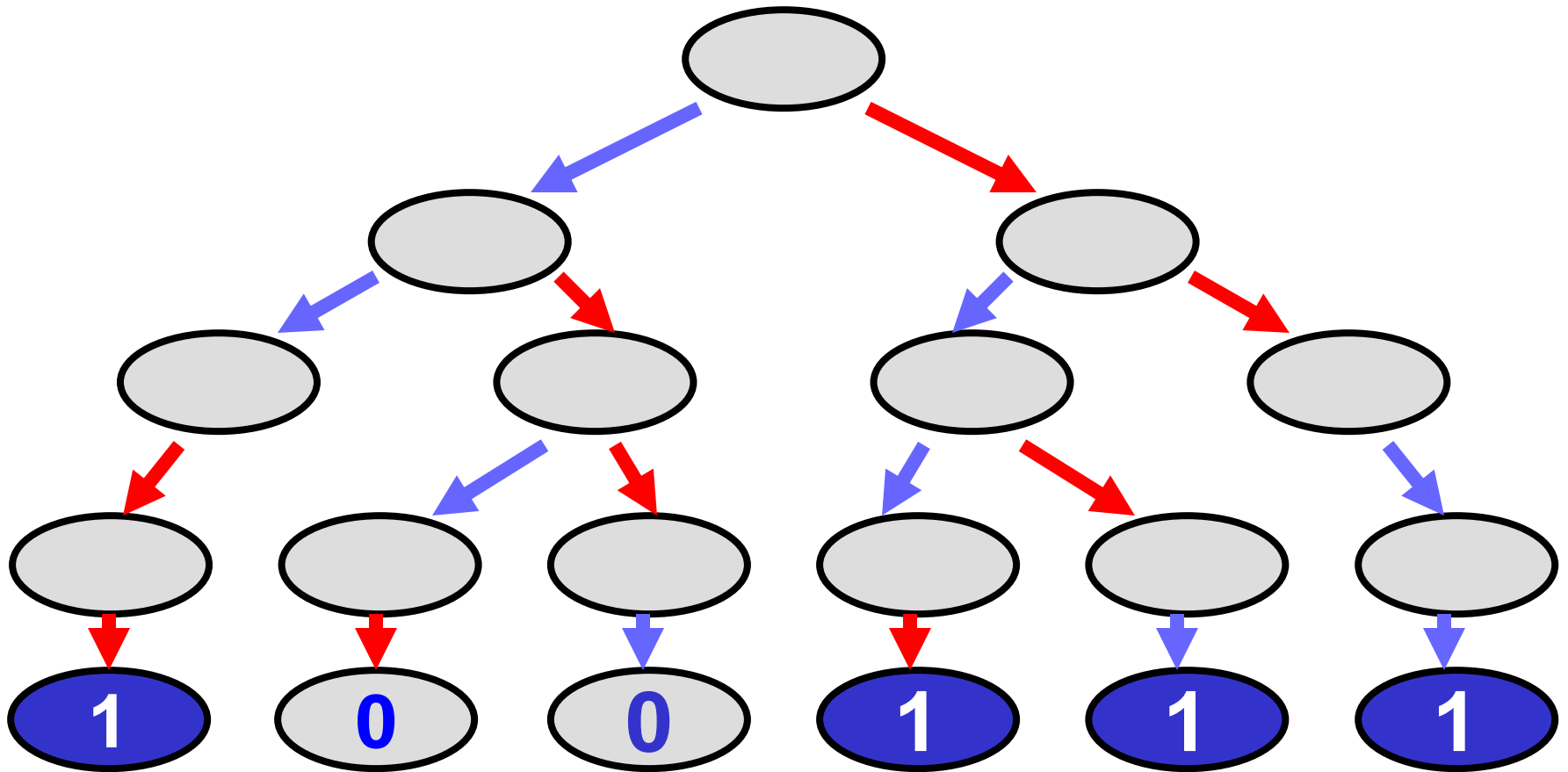
- Either A or B “moves”
- Moving means
 - Register read
 - Register write



The Two-Move Tree

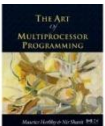
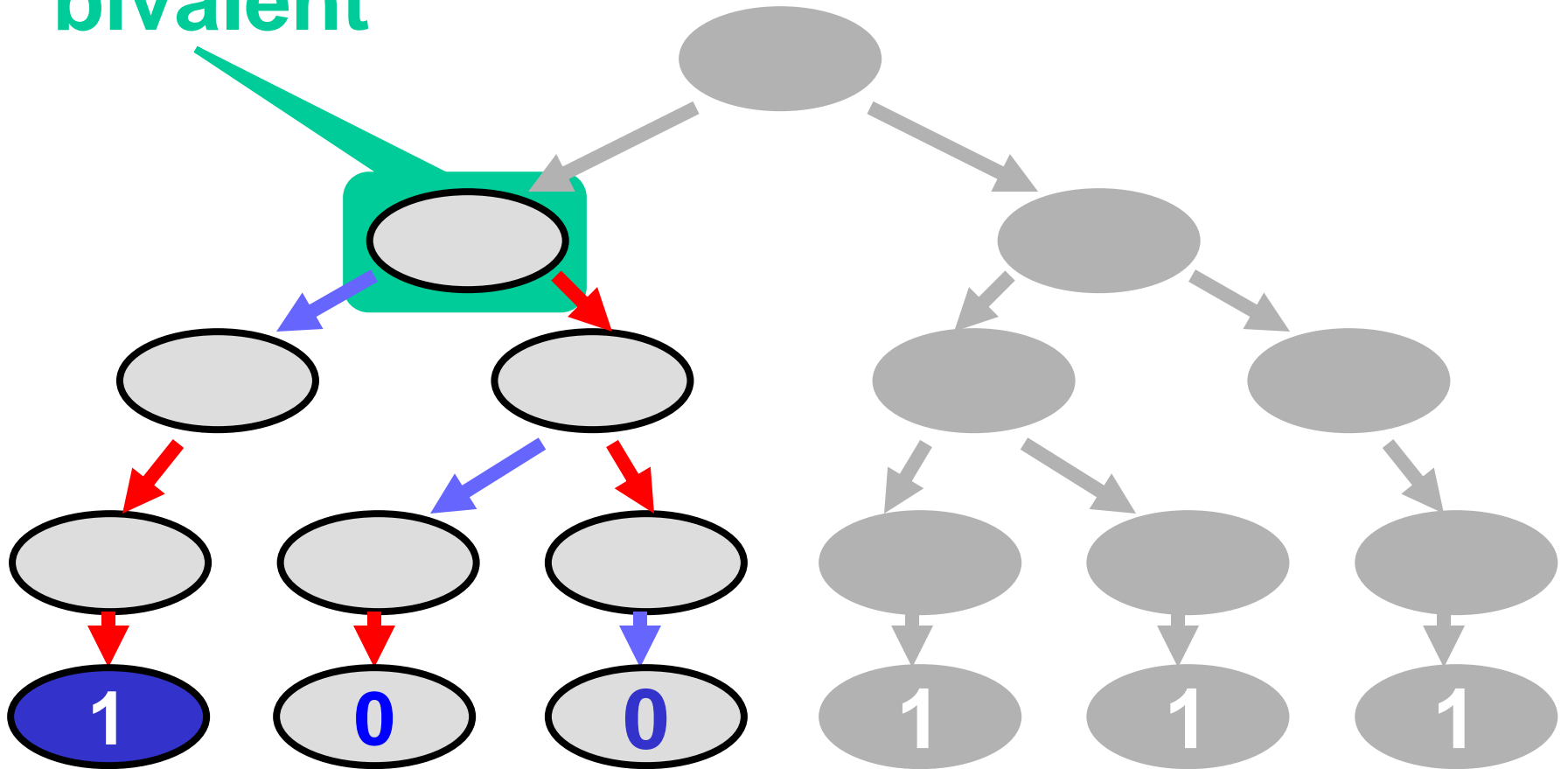


Decision Values

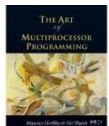
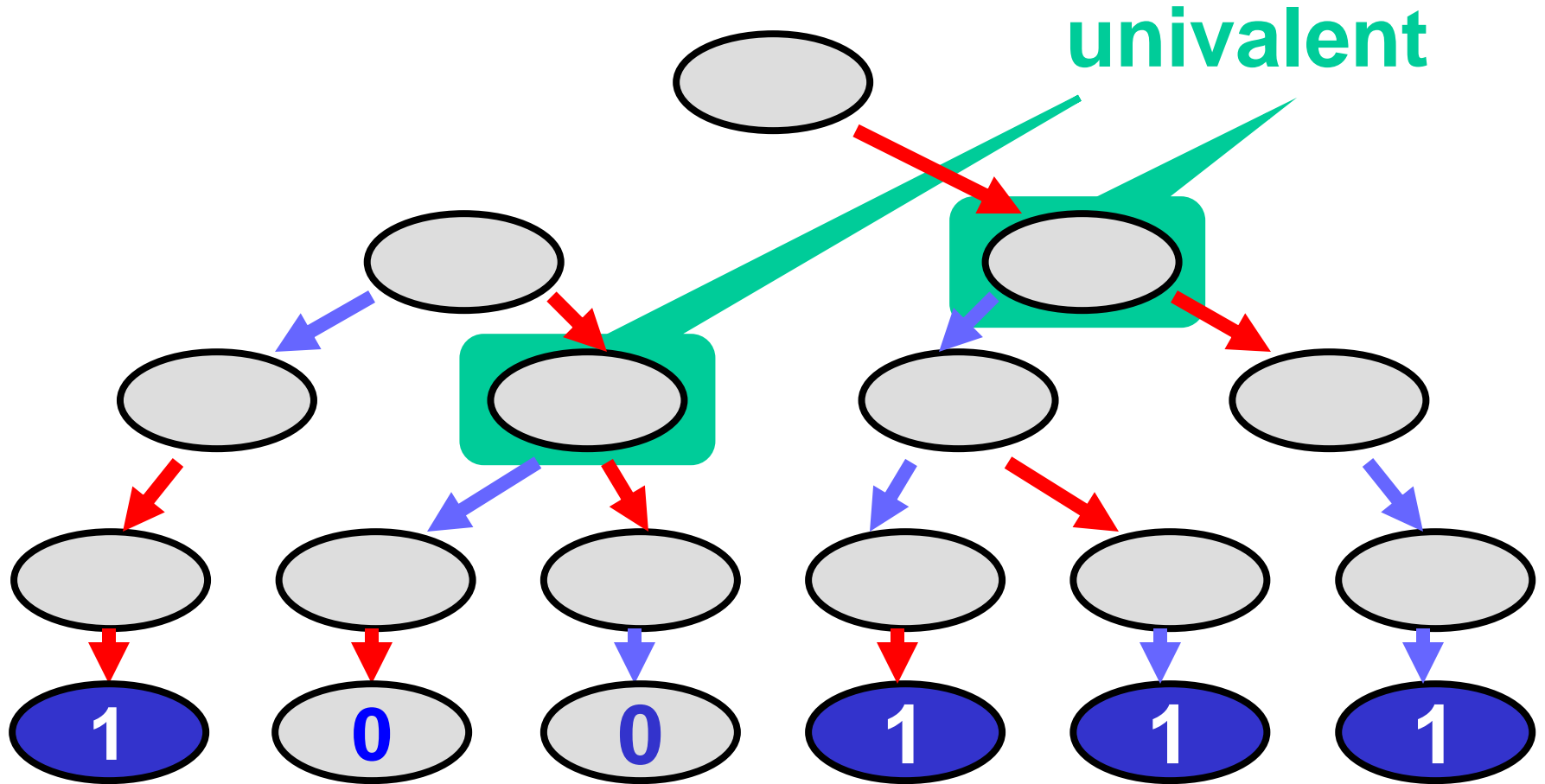


Bivalent: Both Possible

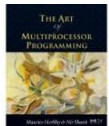
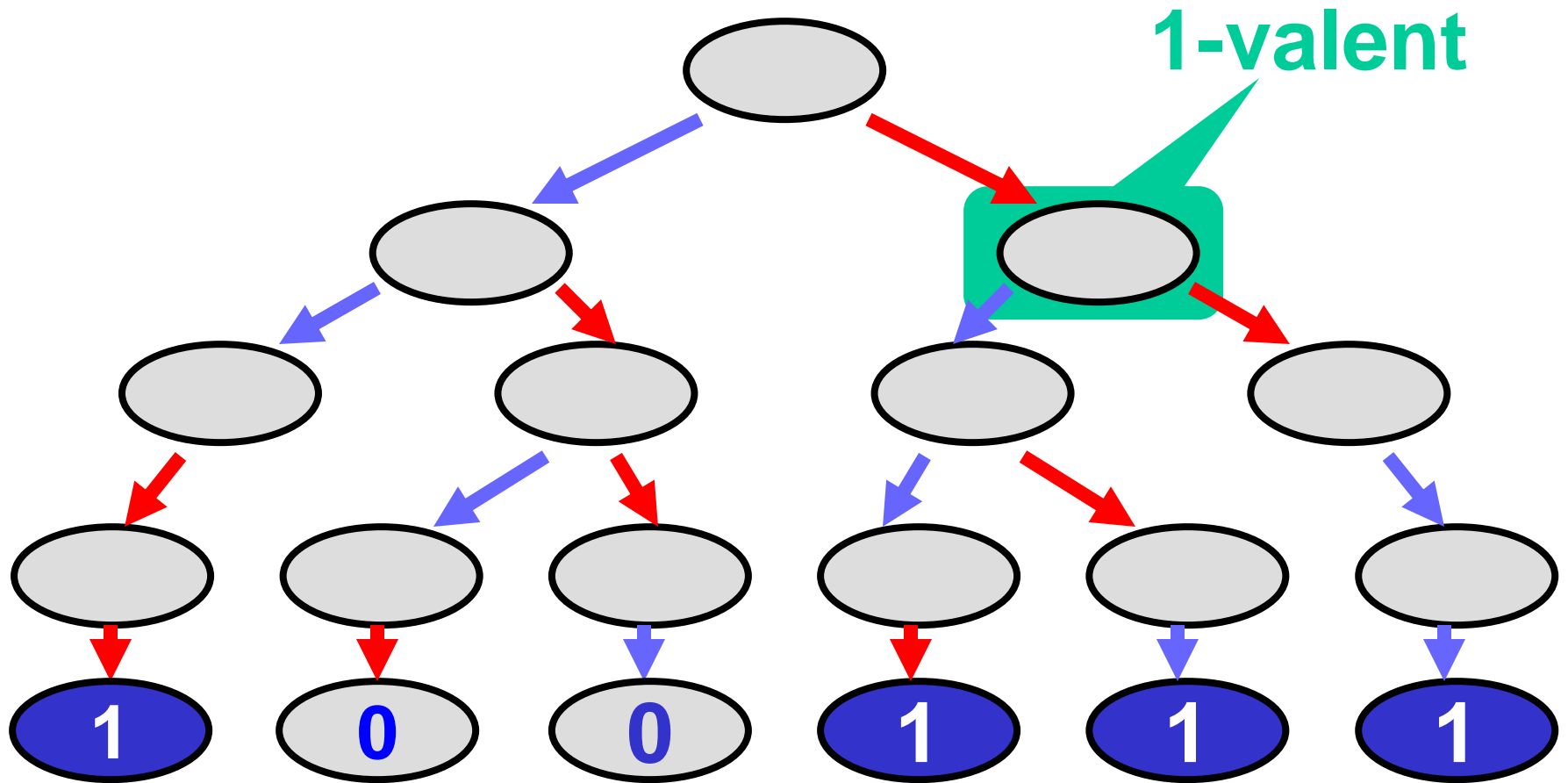
bivalent



Univalent: Single Value Possible

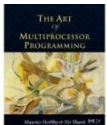


x-valent: x Only Possible Decision



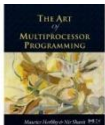
Summary

- Wait-free computation is a tree



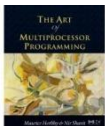
Summary

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



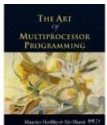
Summary

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



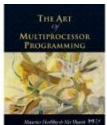
Summary

- 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**



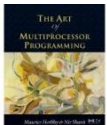
Claim

- Some initial state is bivalent



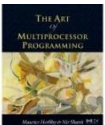
Claim

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



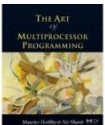
Claim

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

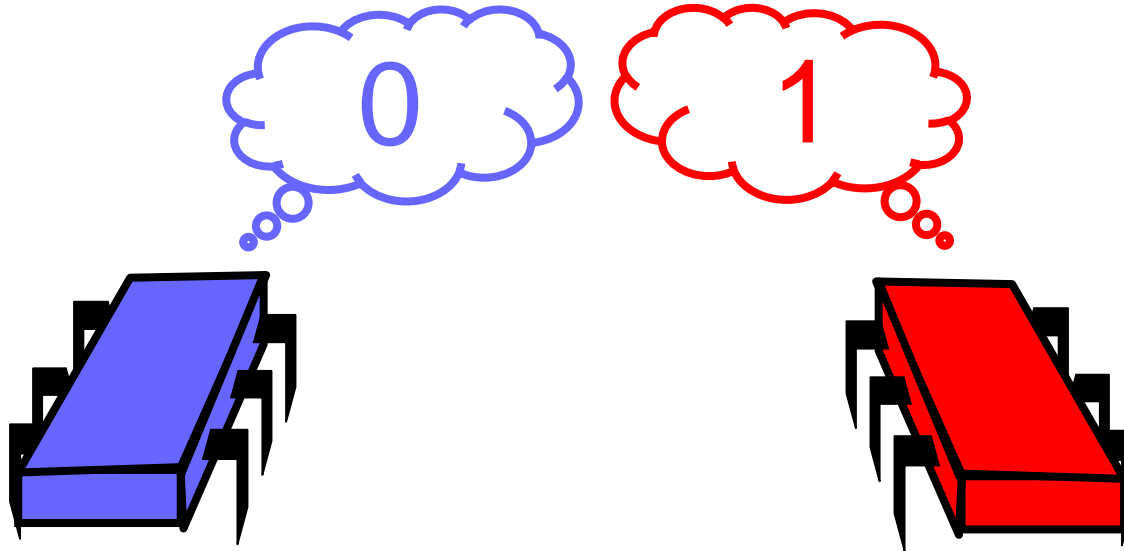


Claim

- 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



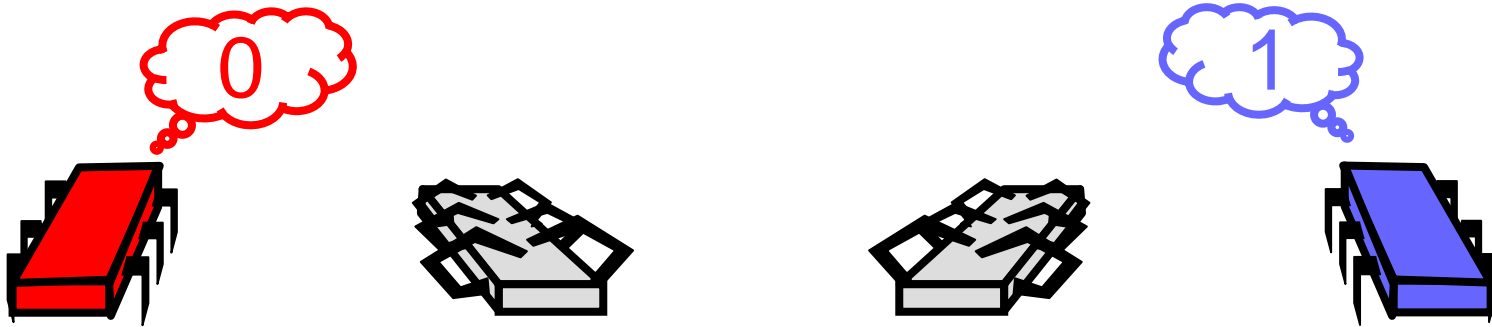
In this solo execution by A

Must Decide 1



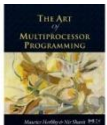
In this solo execution by B

Mixed Initial State Bivalent

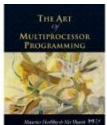
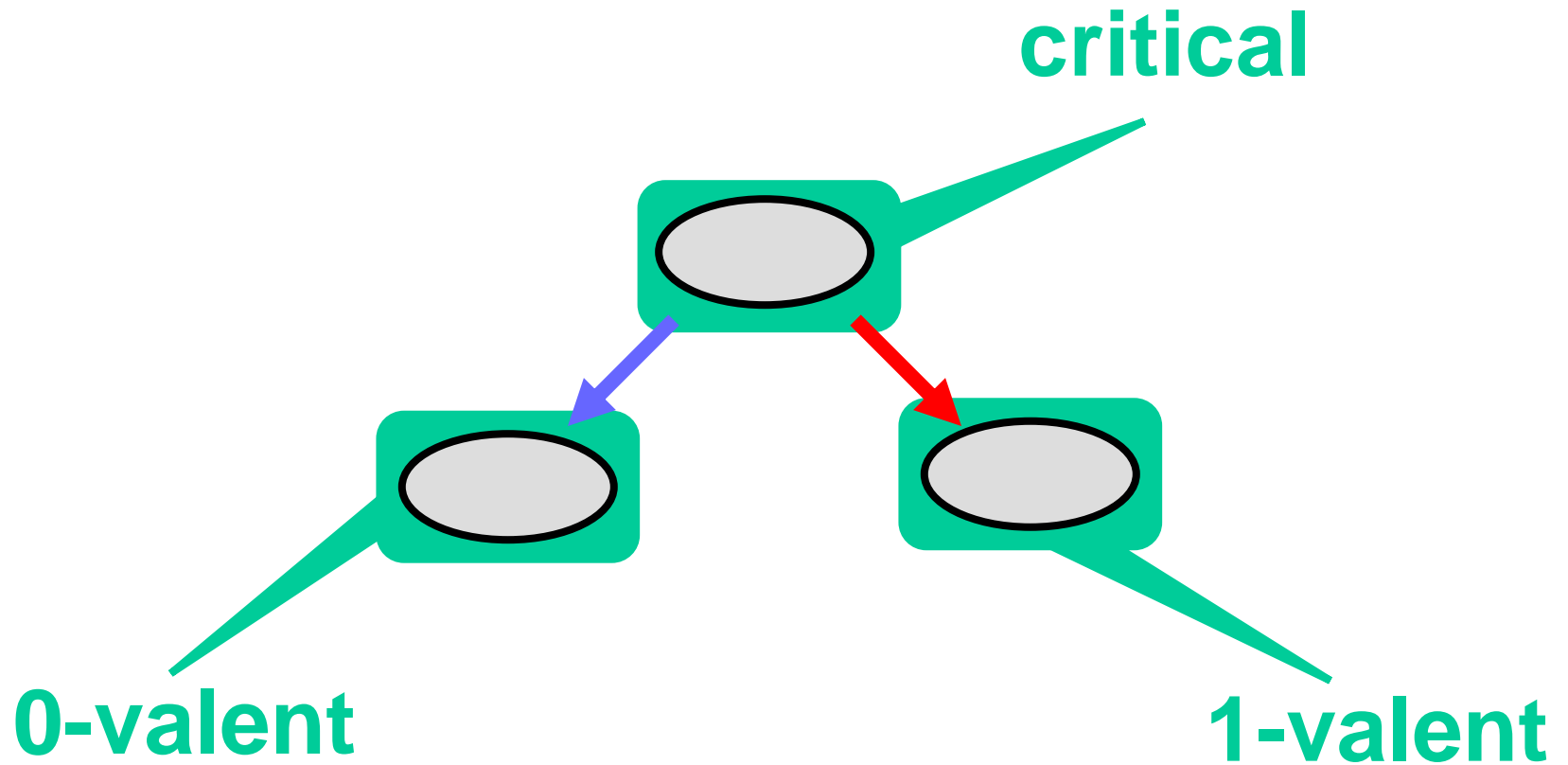


- Solo execution by A must decide 0

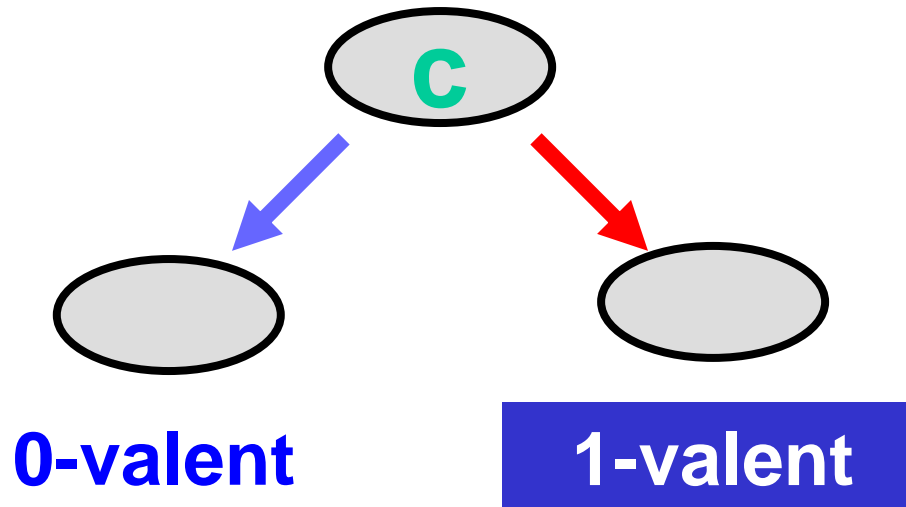
- Solo execution by B must decide 1



Critical States

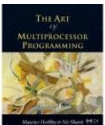


From a Critical State

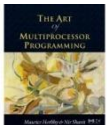
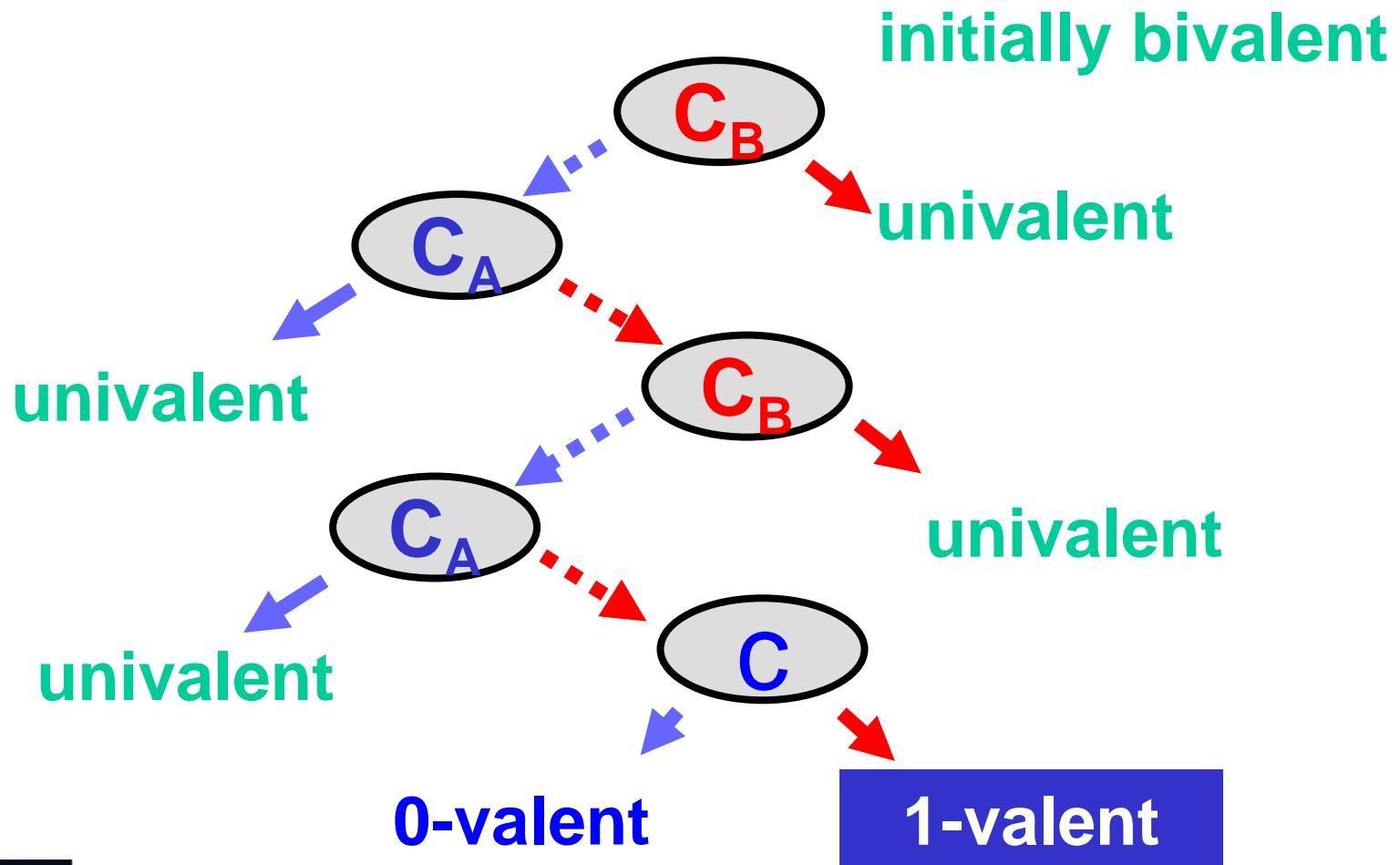


**If A goes first,
protocol decides 0**

**If B goes first,
protocol decides 1**

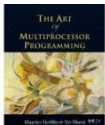


Reaching Critical State



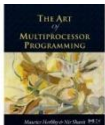
Critical States

- Starting from a bivalent initial state



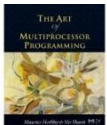
Critical States

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



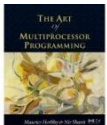
Critical States

- 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



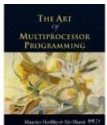
Model Dependency

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



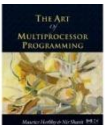
Closing the Deal

- Start from a critical state



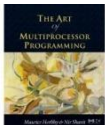
Closing the Deal

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



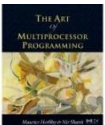
Closing the Deal

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



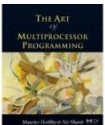
Closing the Deal

- 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

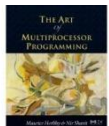
	<code>x.read()</code>	<code>y.read()</code>	<code>x.write()</code>	<code>y.write()</code>
<code>x.read()</code>	?	?	?	?
<code>y.read()</code>	?	?	?	?
<code>x.write()</code>	?	?	?	?
<code>y.write()</code>	?	?	?	?



Possible Interactions

A reads x

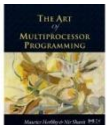
	<code>x.read()</code>	<code>y.read()</code>	<code>x.write()</code>	<code>y.write()</code>
<code>x.read()</code>	?	?	?	?
<code>y.read()</code>	?	?	?	?
<code>x.write()</code>	?	?	?	?
<code>y.write()</code>	?	?	?	?



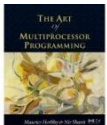
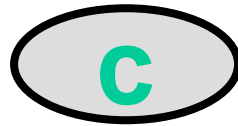
Possible Interactions

A reads x
A reads y

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

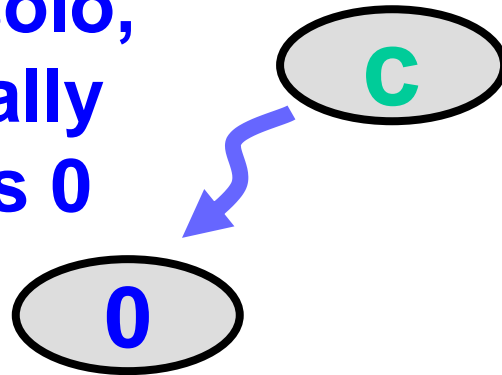


Some Thread Reads



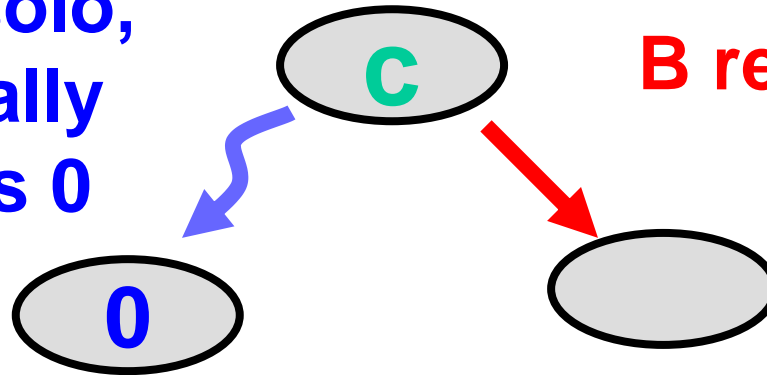
Some Thread Reads

**A runs solo,
eventually
decides 0**

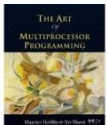


Some Thread Reads

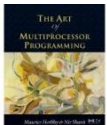
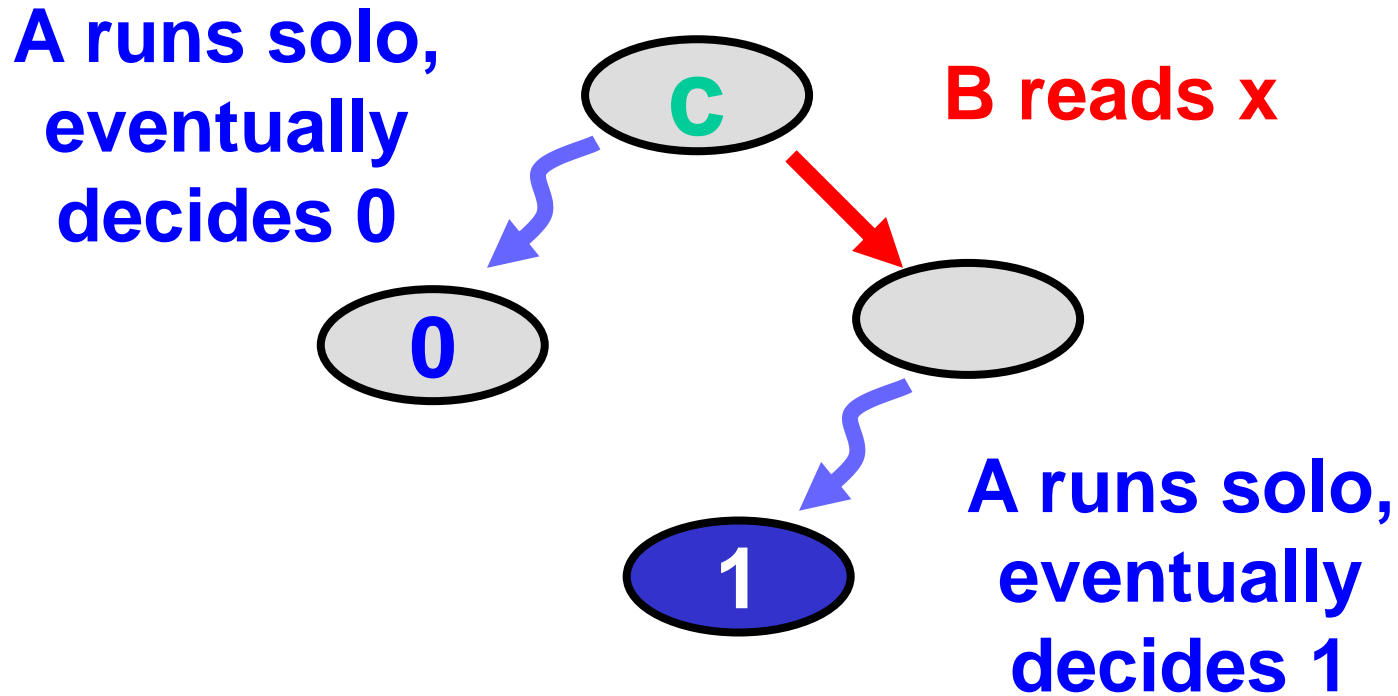
**A runs solo,
eventually
decides 0**



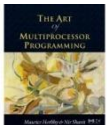
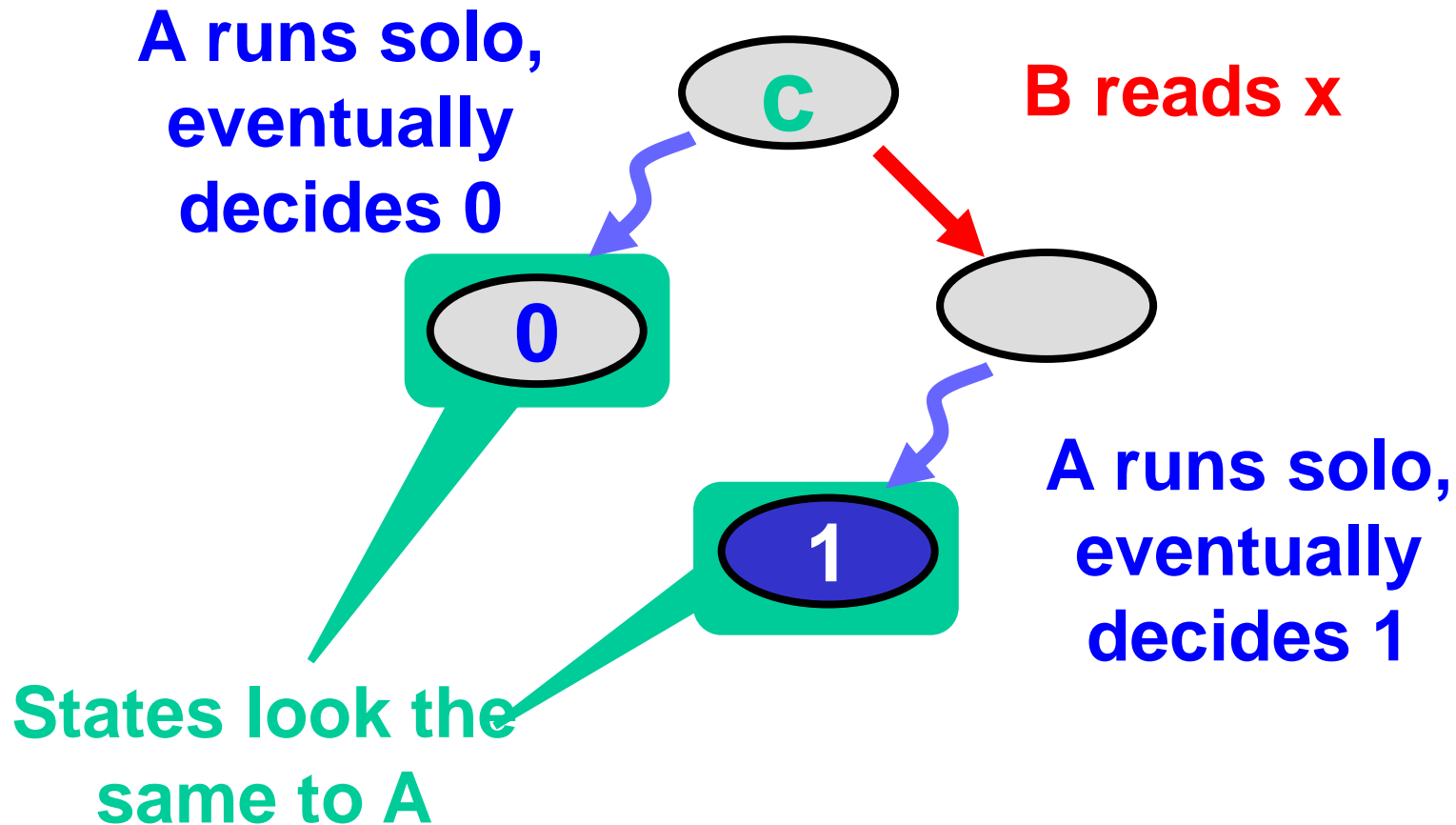
B reads x



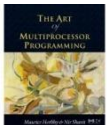
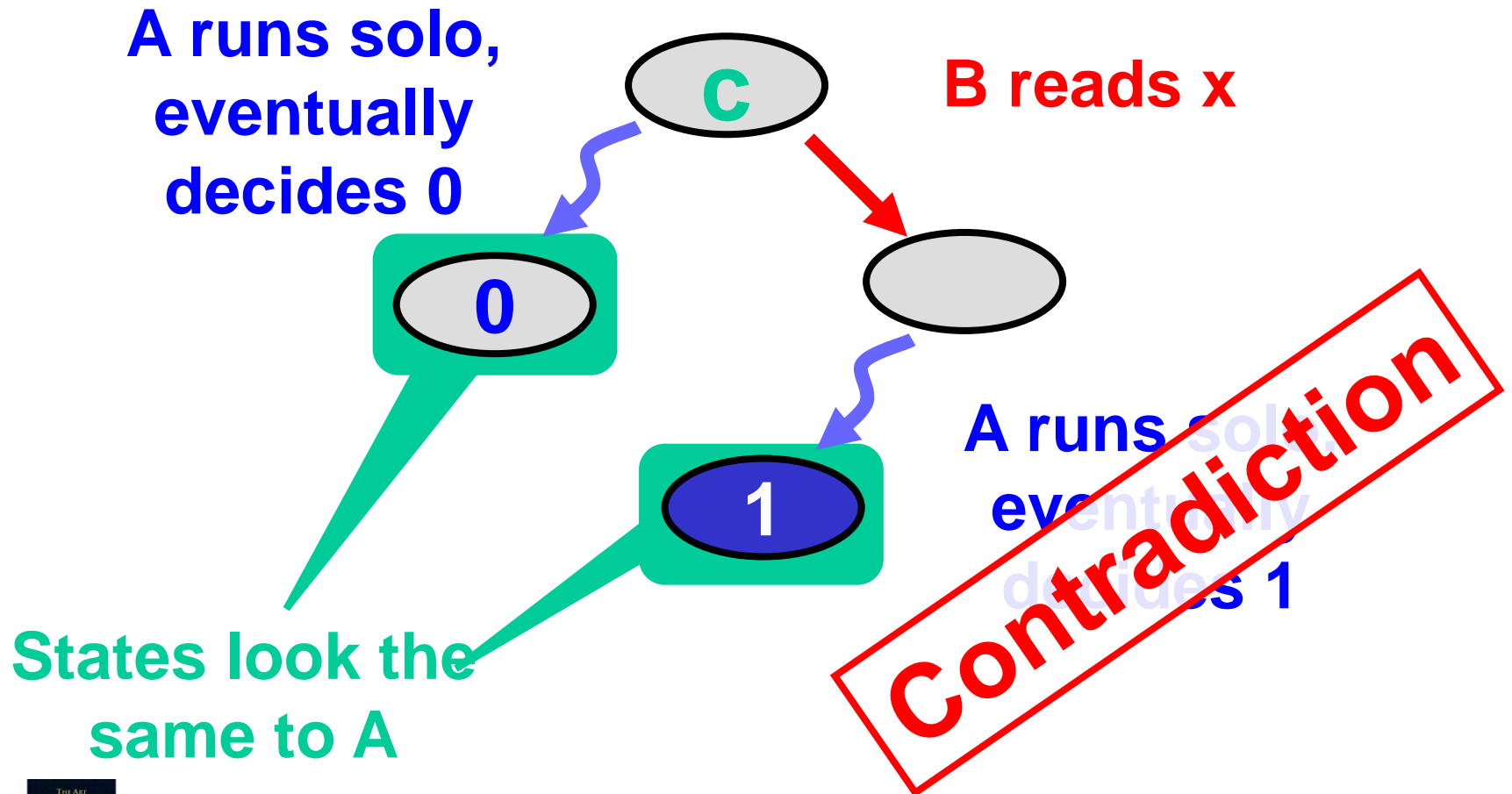
Some Thread Reads



Some Thread Reads

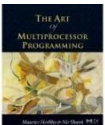


Some Thread Reads

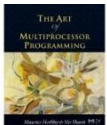
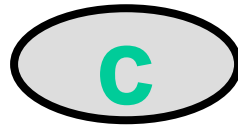


Possible Interactions

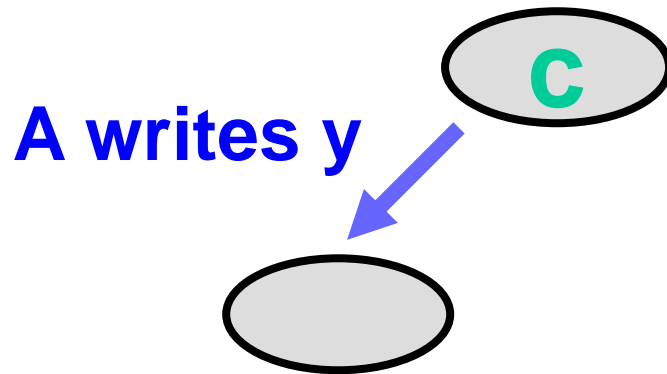
	<code>x.read()</code>	<code>y.read()</code>	<code>x.write()</code>	<code>y.write()</code>
<code>x.read()</code>	no	no	no	no
<code>y.read()</code>	no	no	no	no
<code>x.write()</code>	no	no	?	?
<code>y.write()</code>	no	no	?	?



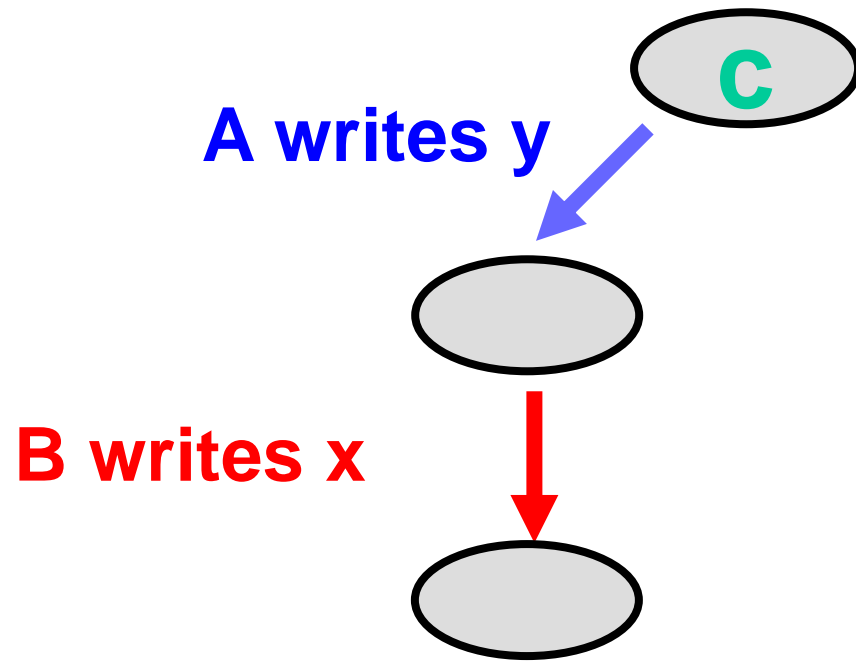
Writing Distinct Registers



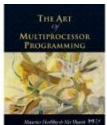
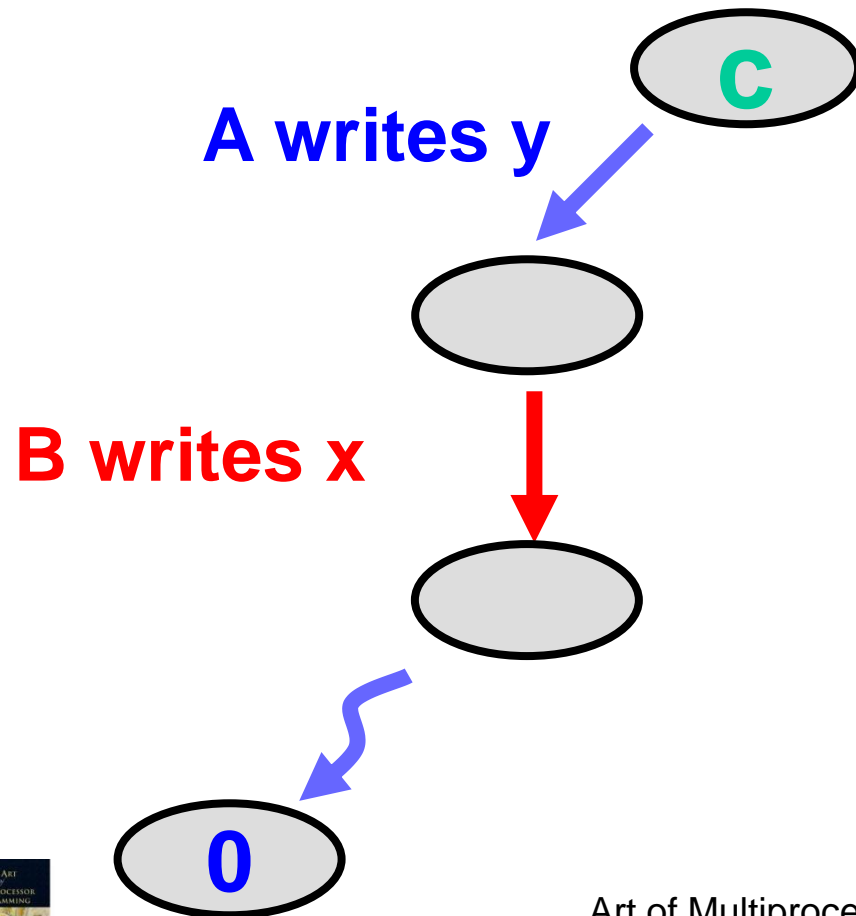
Writing Distinct Registers



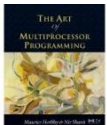
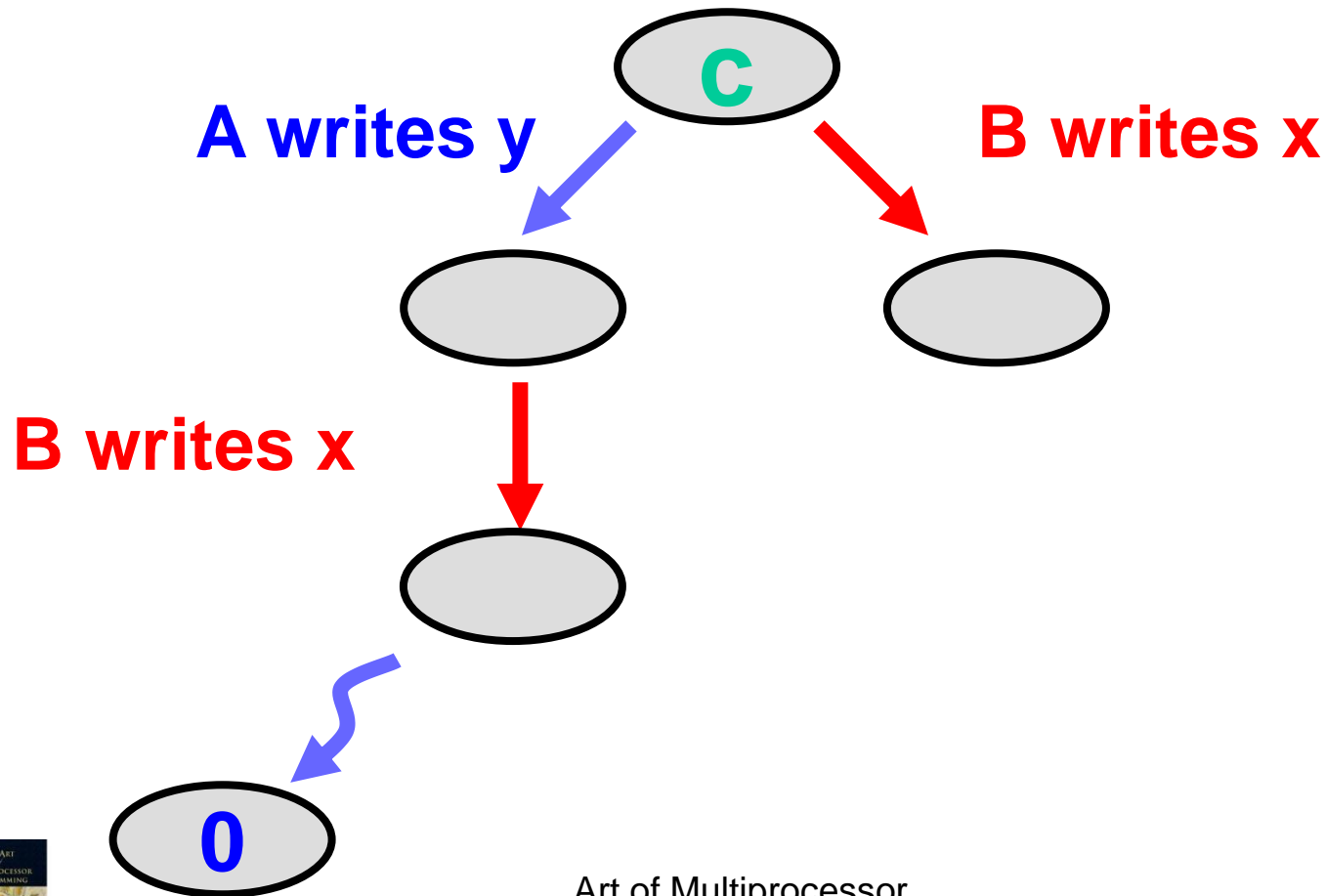
Writing Distinct Registers



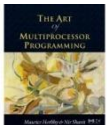
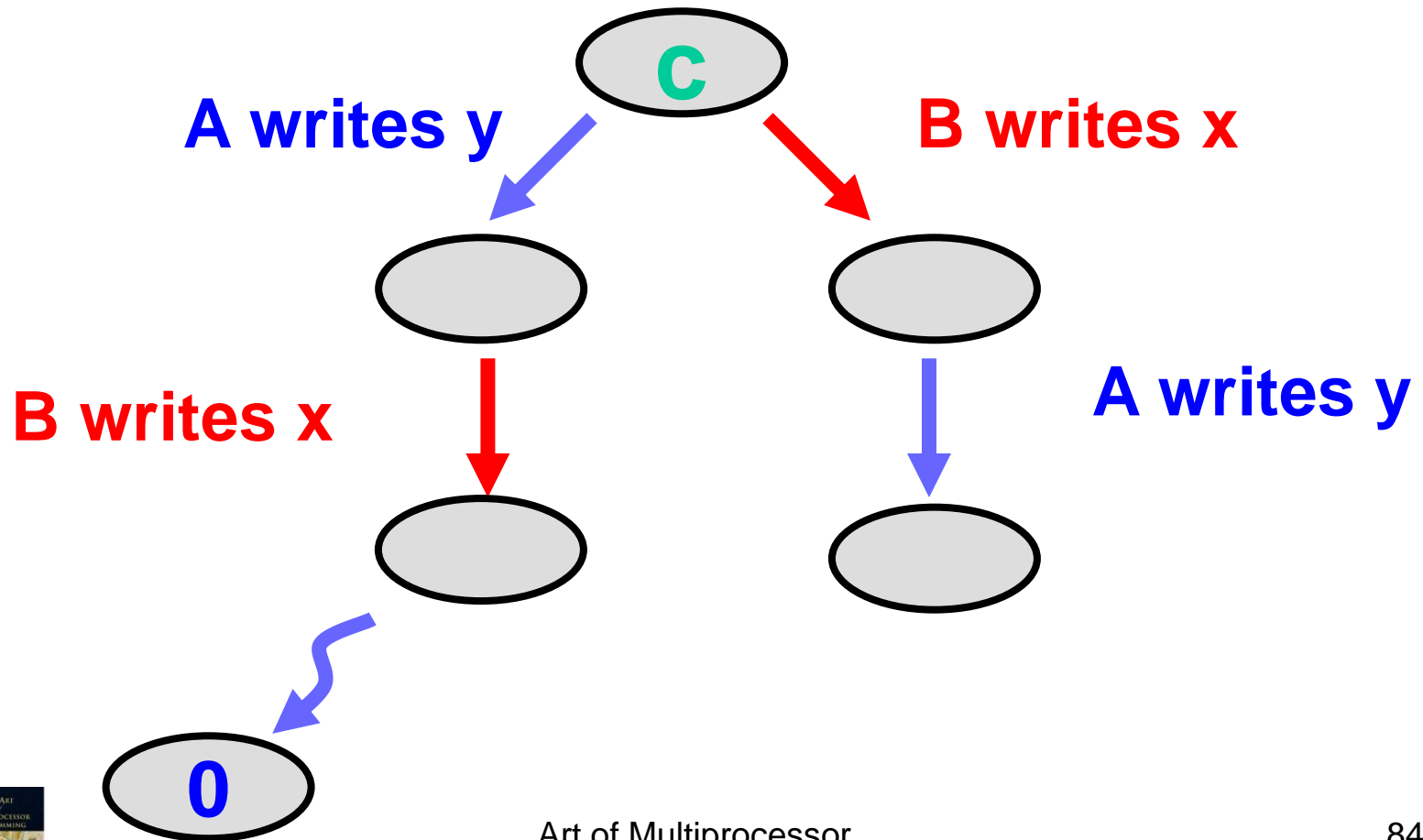
Writing Distinct Registers



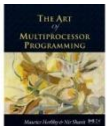
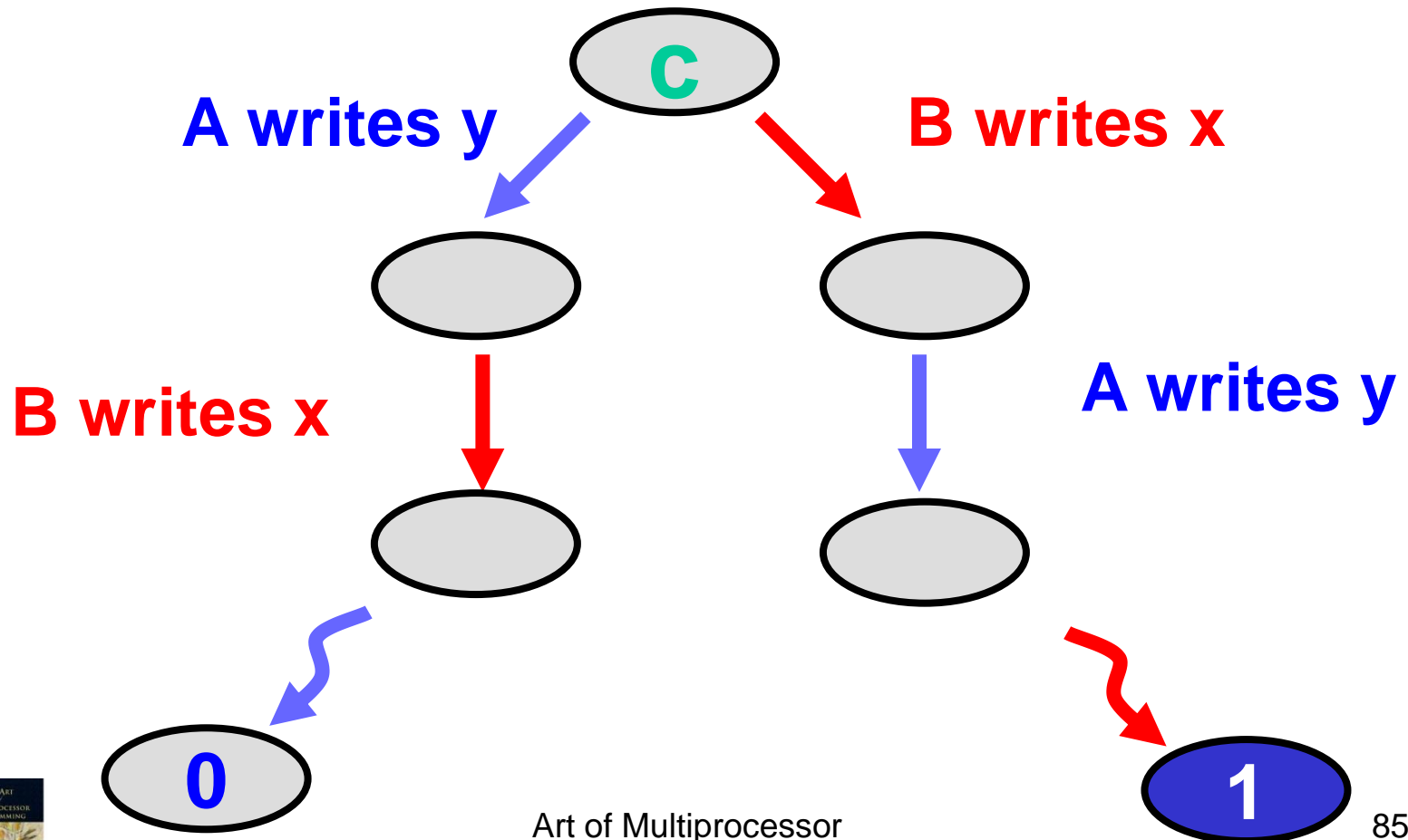
Writing Distinct Registers



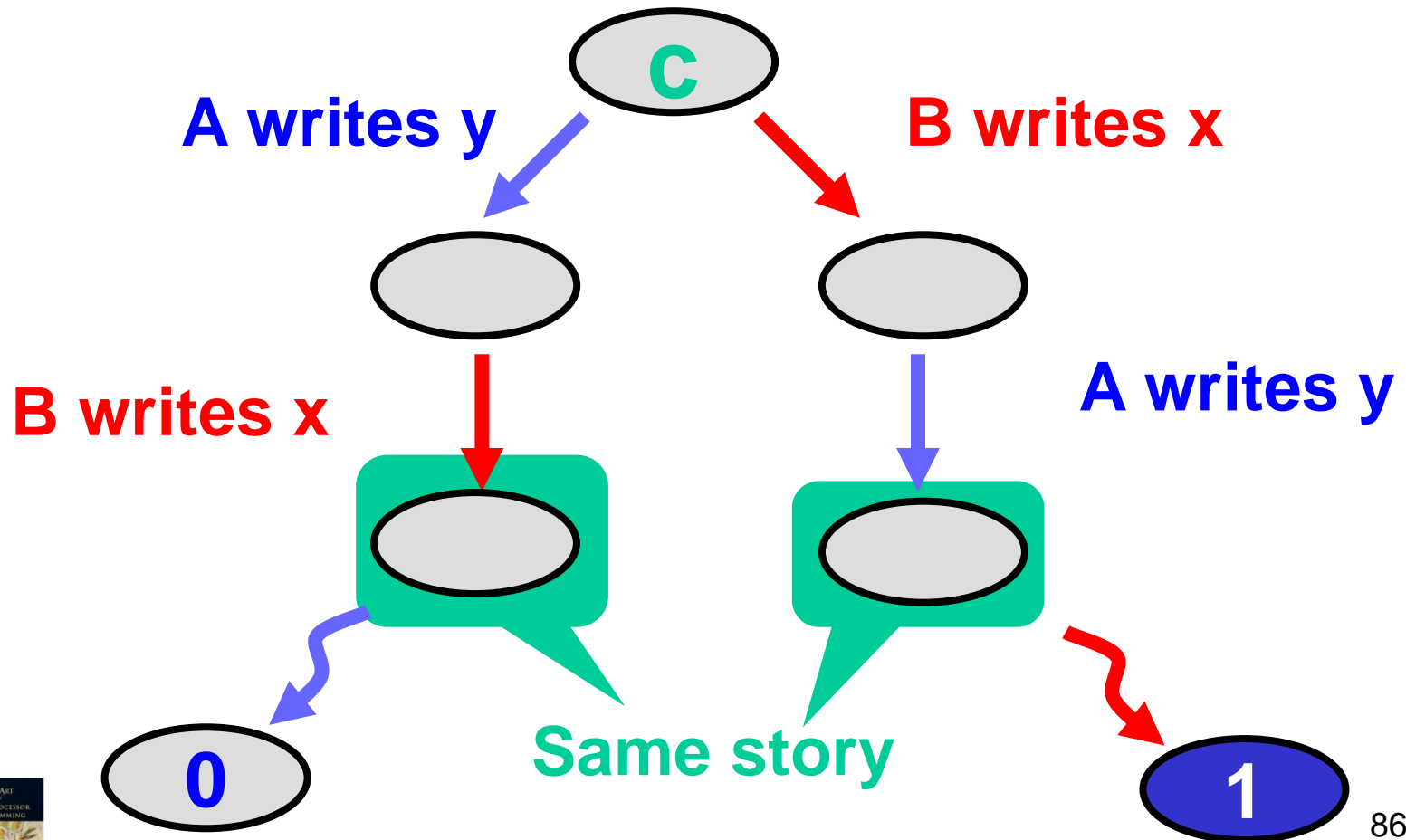
Writing Distinct Registers



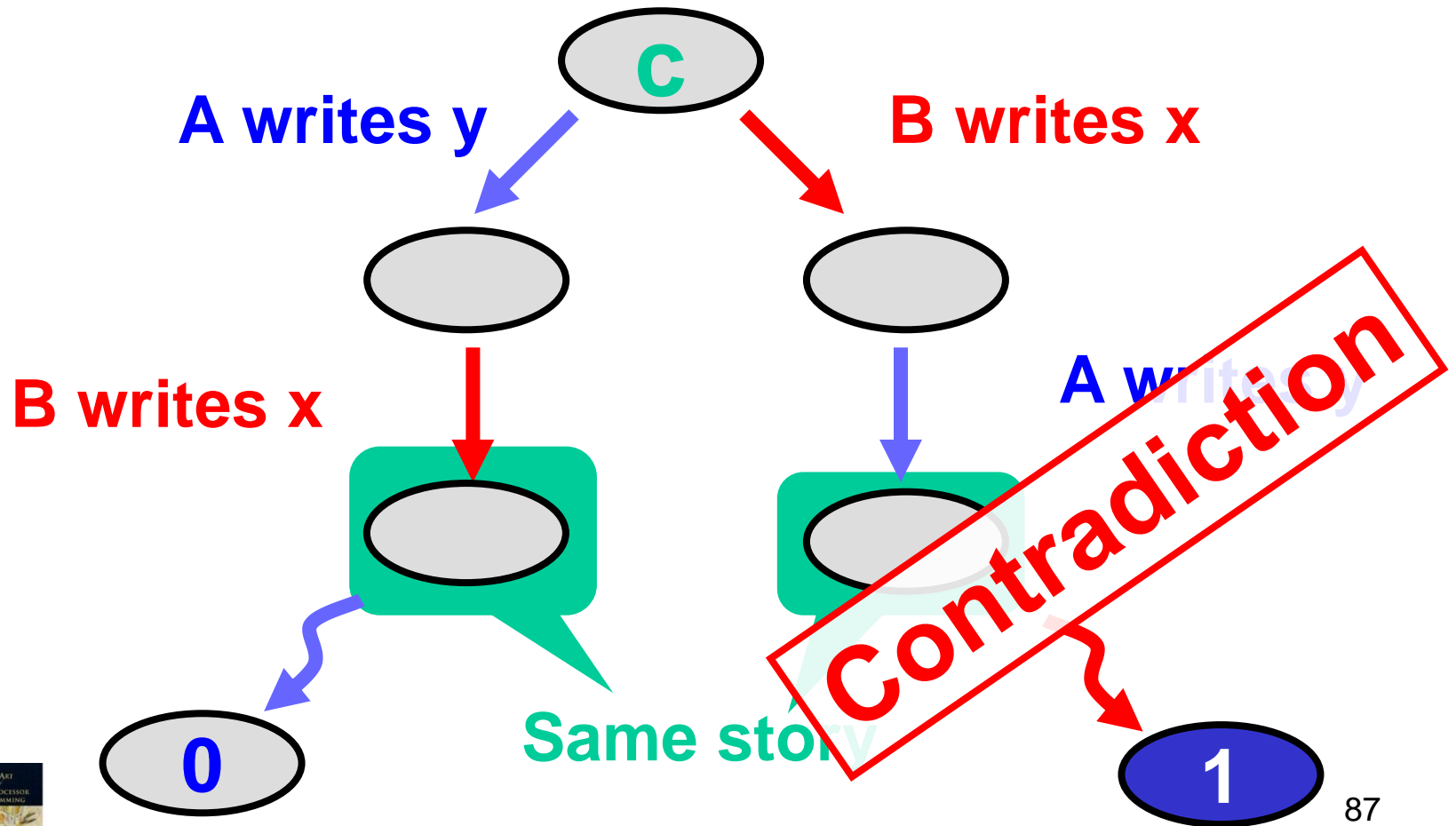
Writing Distinct Registers



Writing Distinct Registers

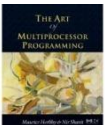


Writing Distinct Registers

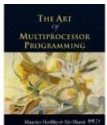
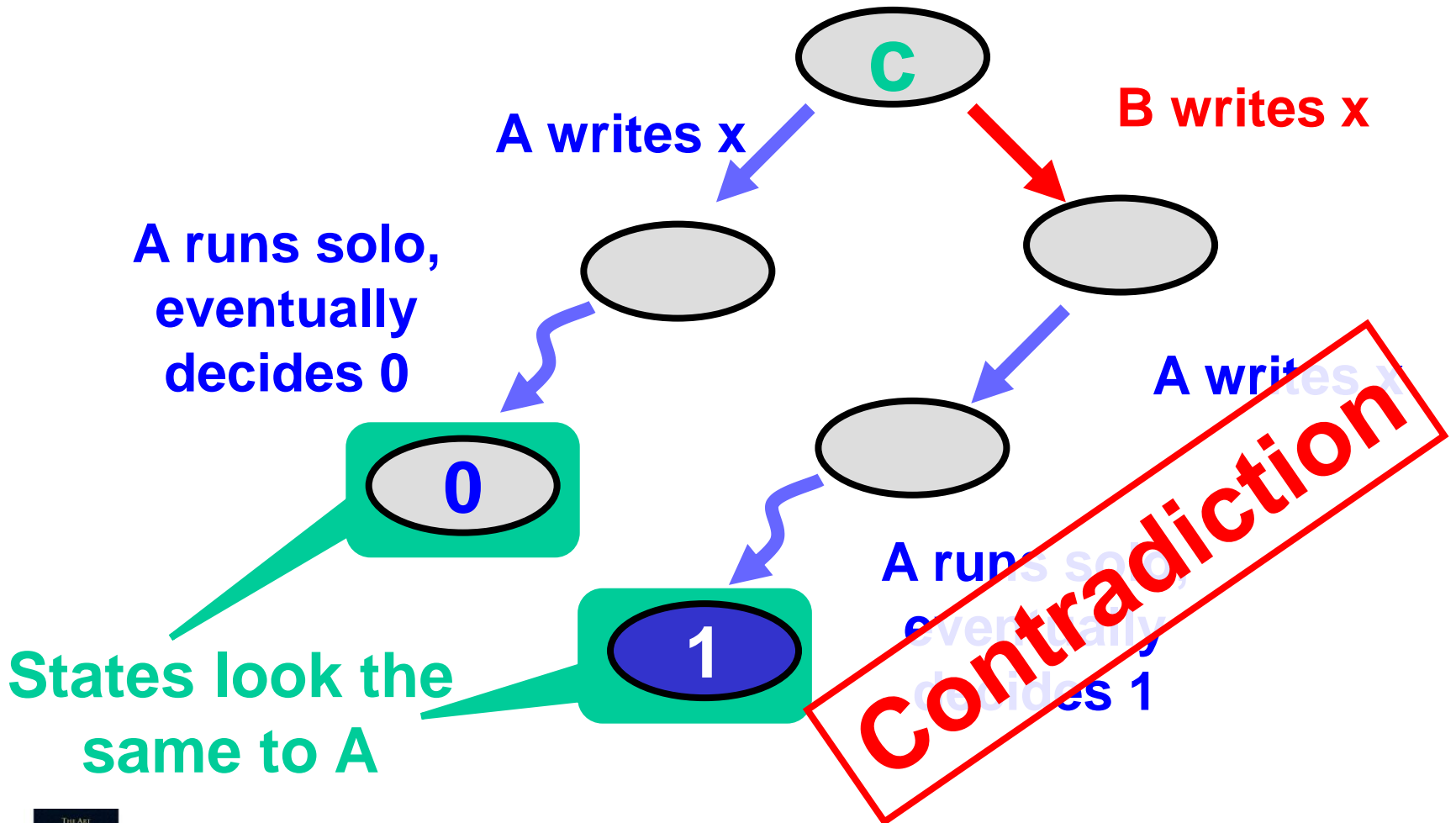


Possible Interactions

	<code>x.read()</code>	<code>y.read()</code>	<code>x.write()</code>	<code>y.write()</code>
<code>x.read()</code>	no	no	no	no
<code>y.read()</code>	no	no	no	no
<code>x.write()</code>	no	no	?	no
<code>y.write()</code>	no	no	no	?



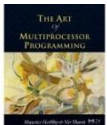
Writing Same Registers



That's All, Folks!

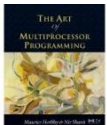
	x.read()	y.read()	x.write()	y.write()
x.read()	no	no	no	no
y.read()	no	no	no	no
x.write()	no	no	no	no
y.write()	no	no	no	no

QED

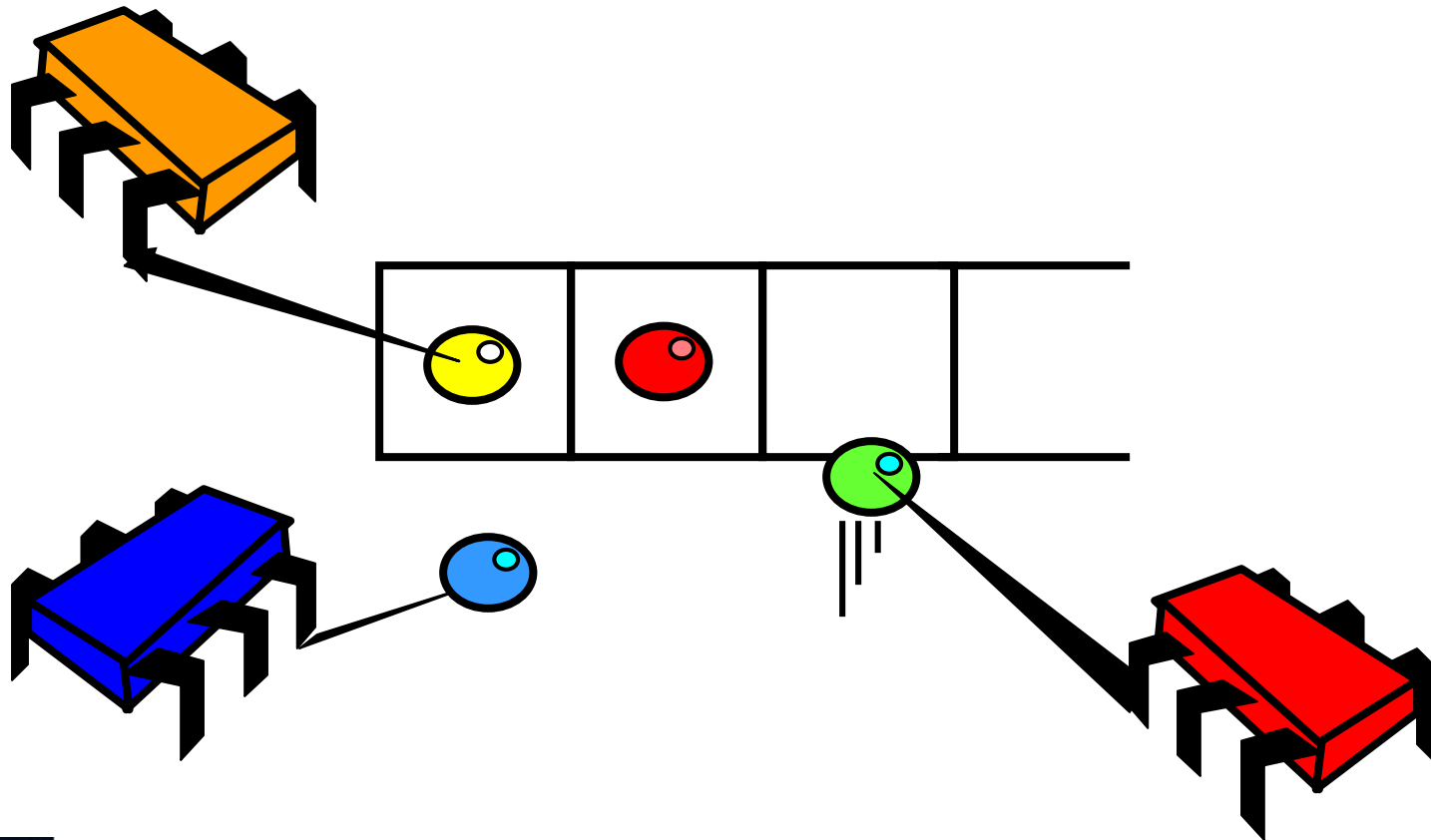


Recap: Atomic Registers Can't Do Consensus

- 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

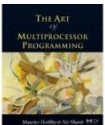


What Does Consensus have to do with Concurrent Objects?



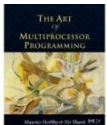
Consensus Object

```
public interface Consensus<T> {  
    T decide(T value);  
}
```



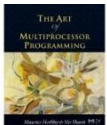
Concurrent Consensus Object

- We consider only one time objects:
 - each thread calls method only once
- Linearizable to sequential consensus object:
 - Winner's call went first



Java Jargon Watch

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

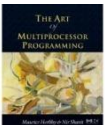


Generic Consensus Protocol

```
abstract class ConsensusProtocol<T>
    implements Consensus<T> {
    protected T[] proposed = new T[N];

    protected void propose(T value) {
        proposed[ThreadID.get()] = value;
    }

    abstract public T decide(T value);
}
```



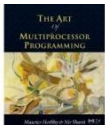
Generic Consensus Protocol

```
abstract class ConsensusProtocol<T>
  implements Consensus<T> {
  protected T[] proposed = new T[N];

  protected void propose(T value) {
    proposed[ThreadID.get()] = value;
  }

  abstract public T decide();
}
```

**Each thread's
proposed value**



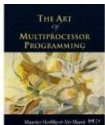
Generic Consensus Protocol

```
abstract class ConsensusProtocol<T>  
    implements Consensus<T> {  
    protected T[] proposed = new T[N];
```

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

```
abstract public T decide(T value):  
}
```

Propose a value



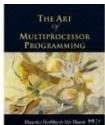
Generic Consensus Protocol

ak

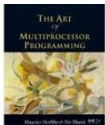
**Decide a value: abstract method
means subclass does the real work**

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

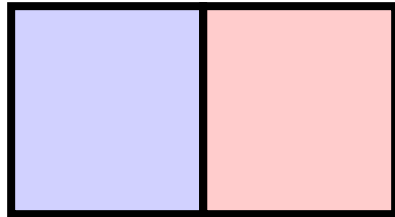
```
abstract public T decide(T value);
```



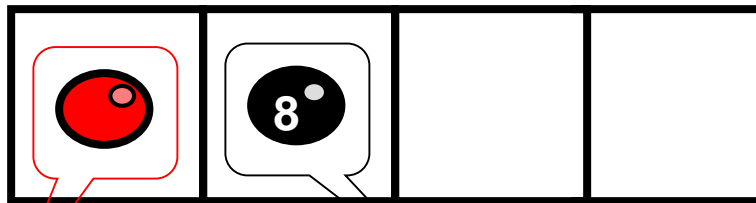
Can a FIFO Queue Implement Consensus?



FIFO Consensus



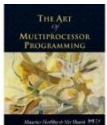
proposed array



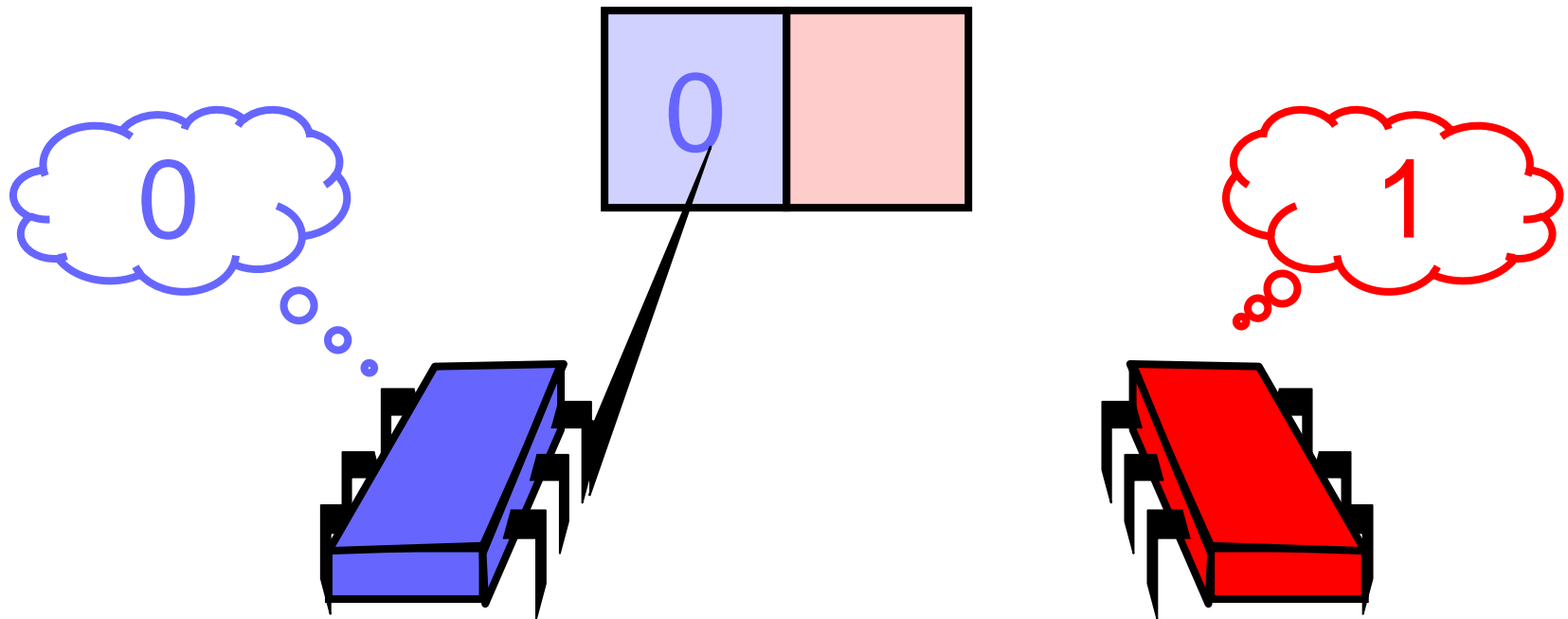
**FIFO Queue
with red and
black balls**

Coveted red ball

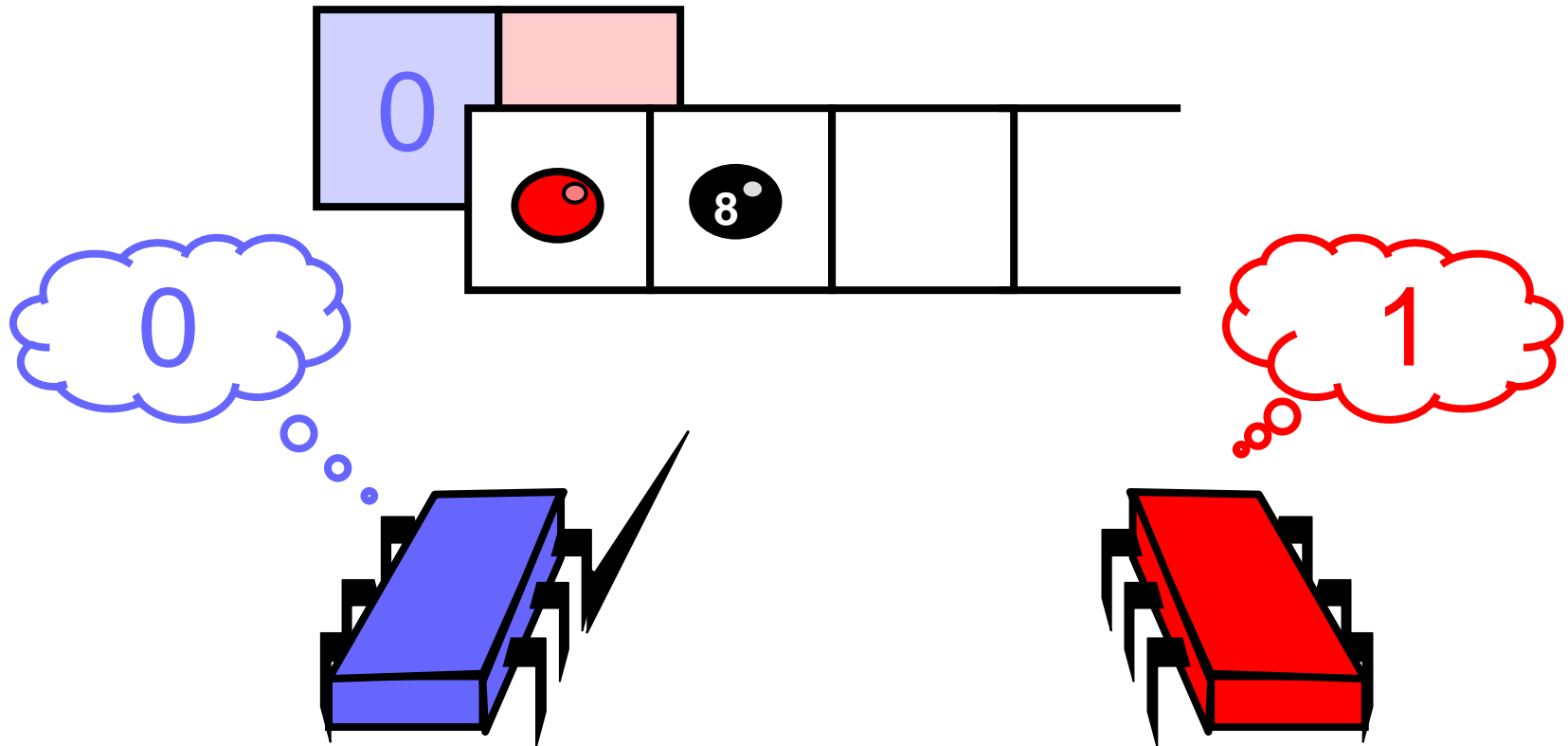
Dreaded black ball



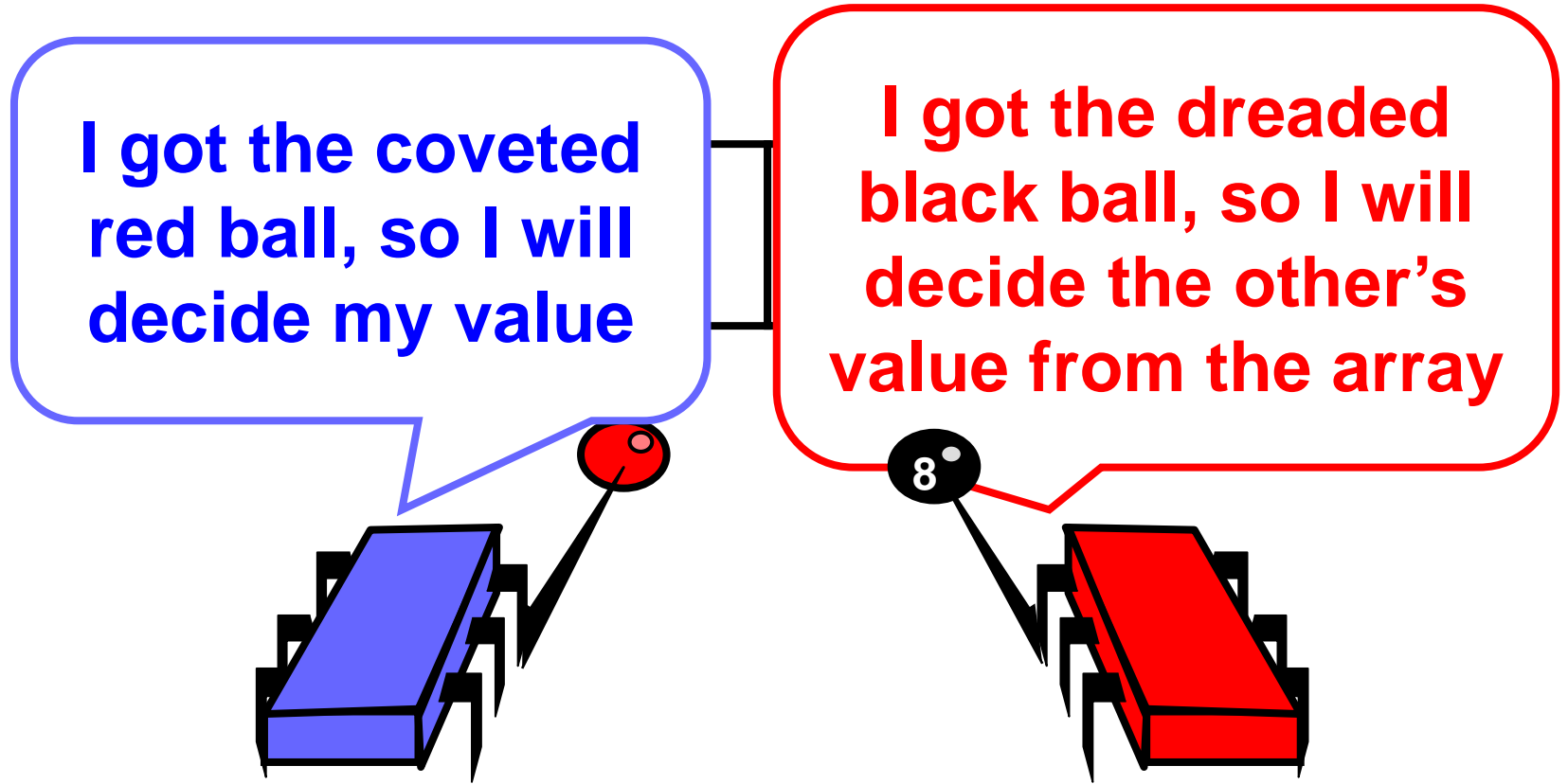
Protocol: Write Value to Array



Protocol: Take Next Item from Queue

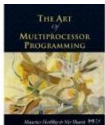


Protocol: Take Next Item from Queue



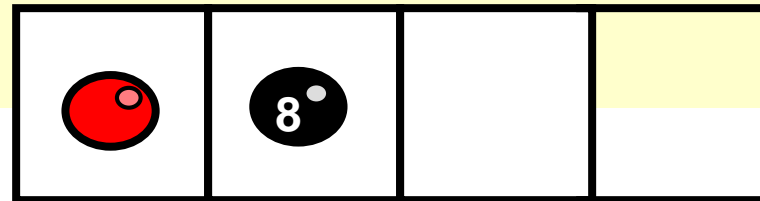
Consensus Using FIFO Queue

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



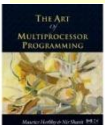
Initialize Queue

```
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?

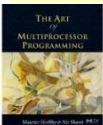
```
public class QueueConsensus<T>
    extends ConsensusProtocol<T> {
    private Queue queue;
    ...
    public T decide(T value) {
        propose(value);
        Ball ball = queue.deq();
        if (ball == Ball.RED)
            return proposed[i];
        else
            return proposed[1-i];
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



Who Won?

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

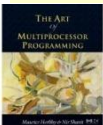
**Race to dequeue
first queue item**



Who Won?

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

I win if I was first

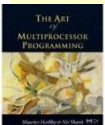


Who Won?

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

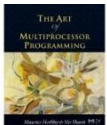
**Other thread wins if I
was second**

**else
return proposed[1-i];**



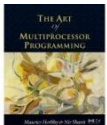
Why does this Work?

- 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

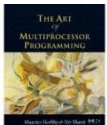
- We can solve 2-thread consensus using only
 - A two-dequeuer queue, and
 - Some atomic registers



Implications

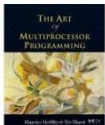
- Given
 - A consensus protocol from queue and registers
- Assume there exists
 - A queue implementation from atomic registers
- Substitution yields:
 - A wait-free consensus protocol from atomic registers

contradiction



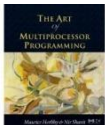
Corollary

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



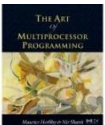
Consensus Numbers

- 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



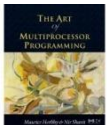
Consensus Numbers

- 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



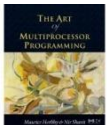
Synchronization Speed Limit

- Conversely

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

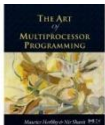
- This theorem will be very useful
 - Unforeseen practical implications!

**Theoretical
Caveat: Certain
weird exceptions
exist**



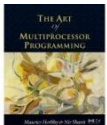
Earlier Grand Challenge

- 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**



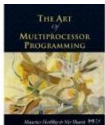
Multiple Assignment Theorem

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



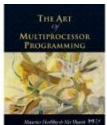
Proof Strategy

- 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

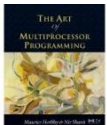
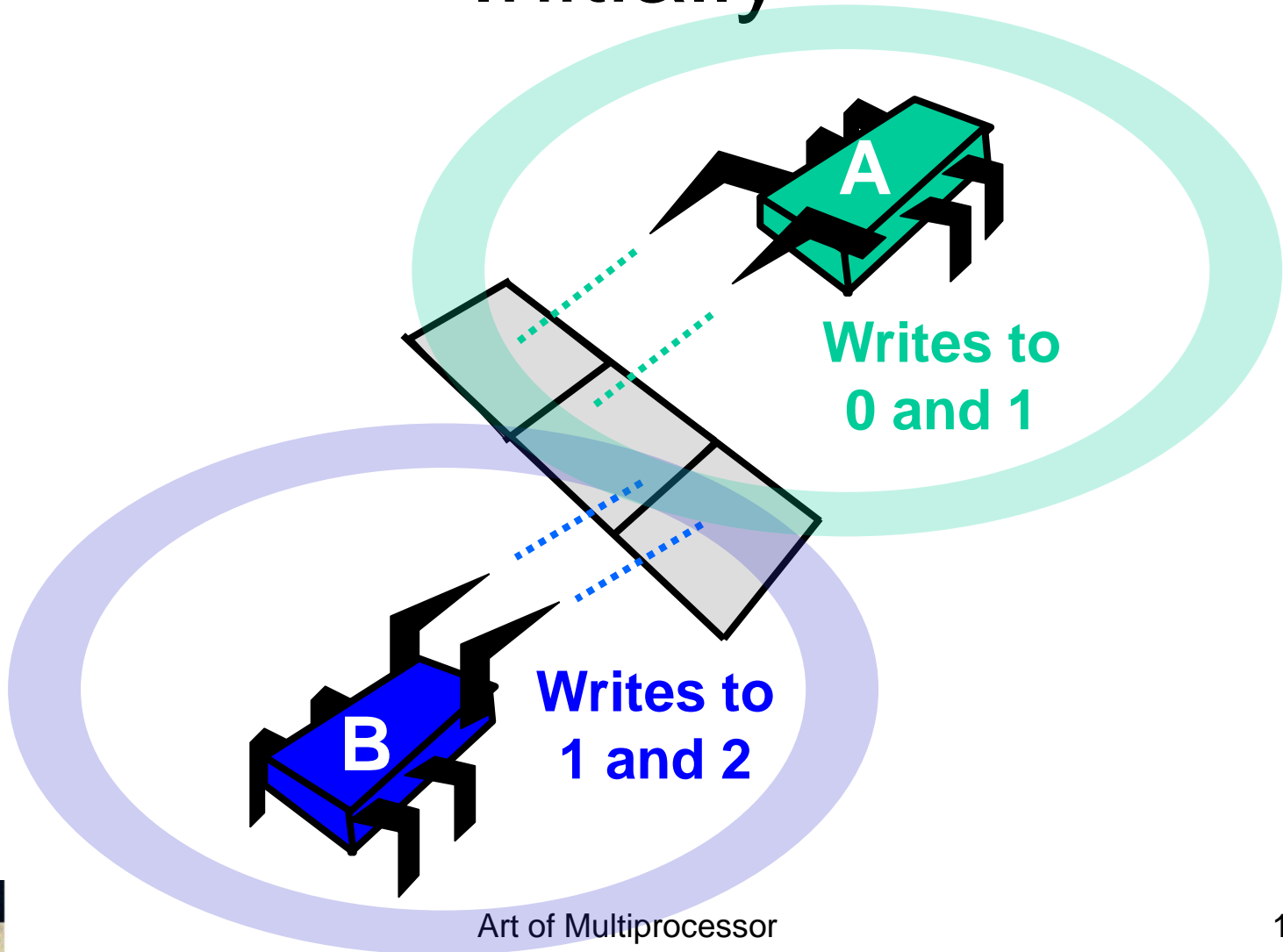


Proof Strategy

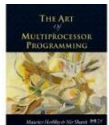
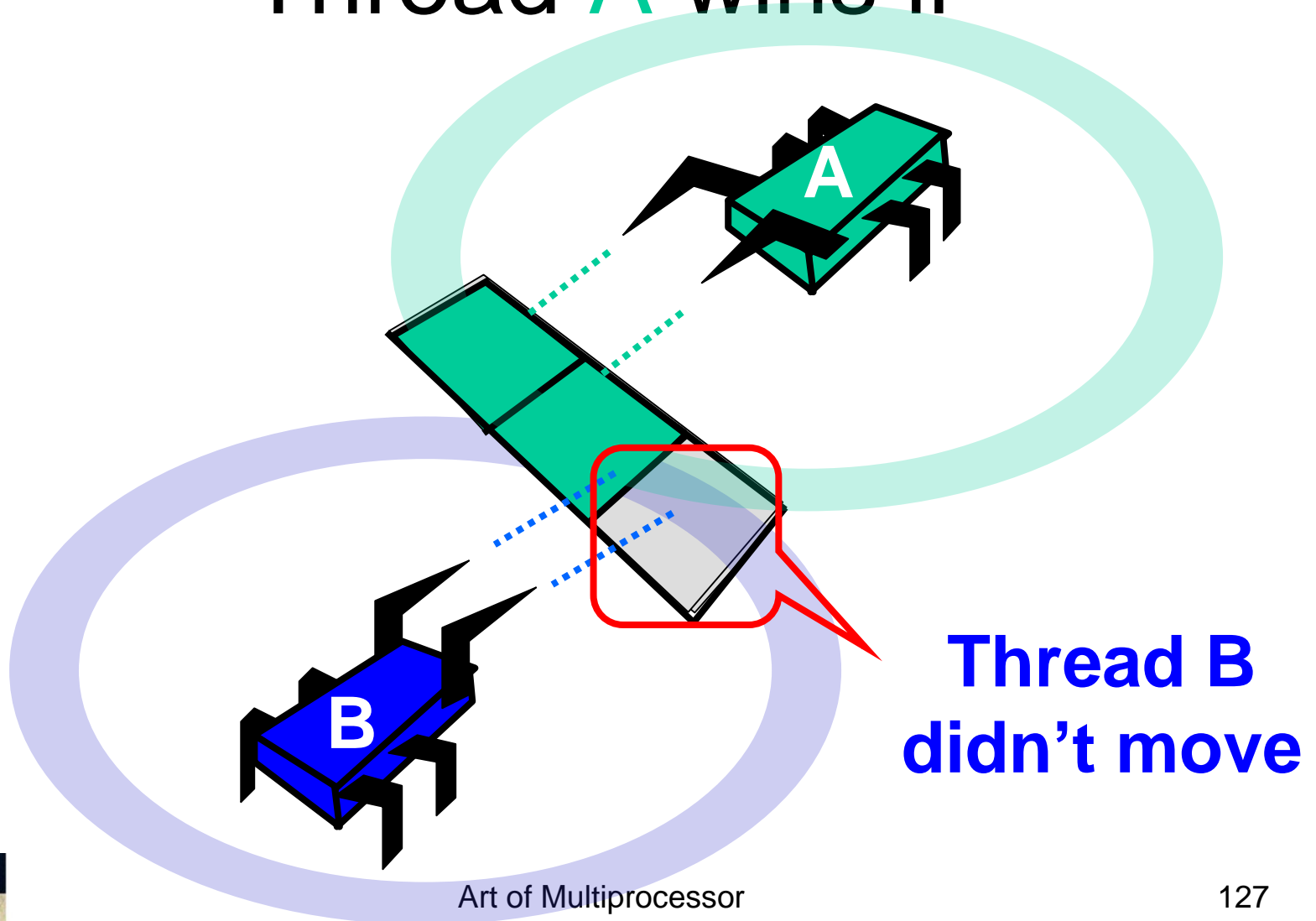
- Take a 3-element array
 - A writes atomically to slots 0 and 1
 - B writes atomically to slots 1 and 2
 - Any thread can scan any set of locations



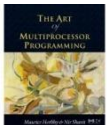
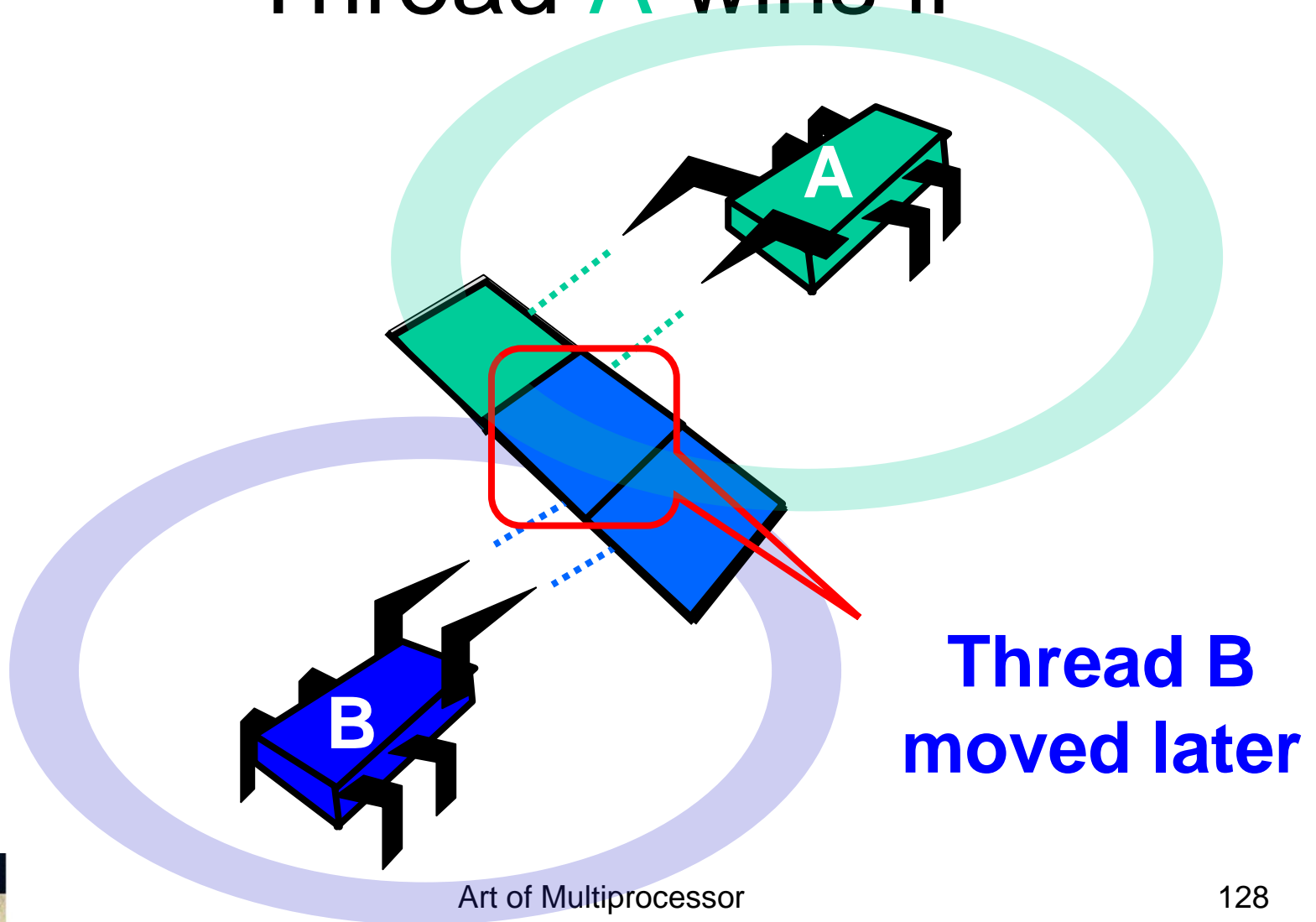
Initially



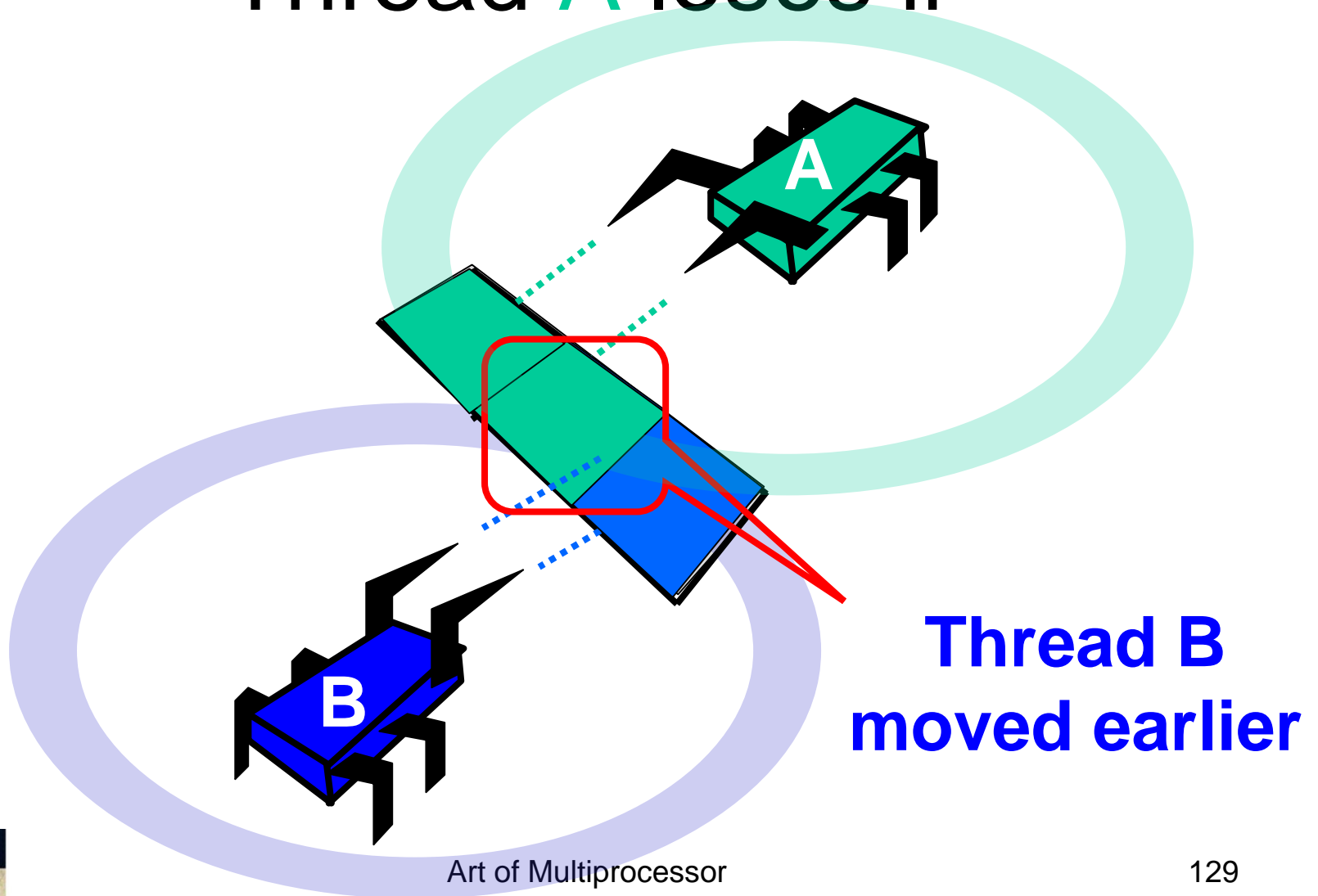
Thread **A** wins if



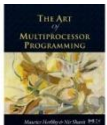
Thread **A** wins if



Thread A loses if

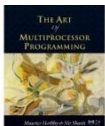


**Thread B
moved earlier**



Multi-Consensus Code

```
class MultiConsensus extends ...{
    Assign2 a = new Assign2(3, EMPTY);
    public T decide(T 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];
    }
}
```

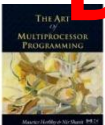


Multi-Consensus Code

```
class MultiConsensus extends ...{
    Assign2 a = new Assign2(3, EMPTY);
    public T decide(T 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

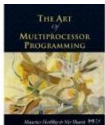
Decide sets $j=i-1$ and proposes value



Multi-Consensus Code

```
class MultiConsensus extends ...{  
Assign2 a = new Assign2(3, EMPTY);  
public T decide(T 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];  
}}
```

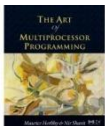
**Three slots
initialized to
EMPTY**



Multi-Consensus Code

```
class MultiConsensus extends ...{
    Assign2 a = new Assign2(3, EMPTY);
    public T decide(T 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];
    }
}
```

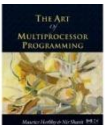
**Assign ID 0 to entries 0,1
(or ID 1 to entries 1,2)**



Multi-Consensus Code

```
class MultiConsensus extends ...{
    Assign2 a = new Assign2(3, EMPTY);
    public T decide(T 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];
    }
}
```

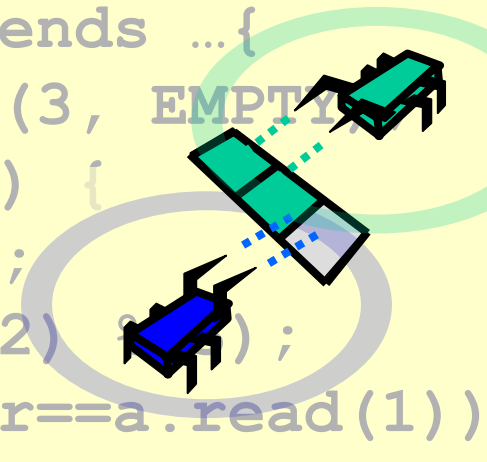
**Read the register my
thread didn't assign**



Multi-Consensus Code

```
class MultiConsensus extends ...{
  Assign2 a = new Assign2(3, EMPTY);
  public T decide(T 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];
  }}

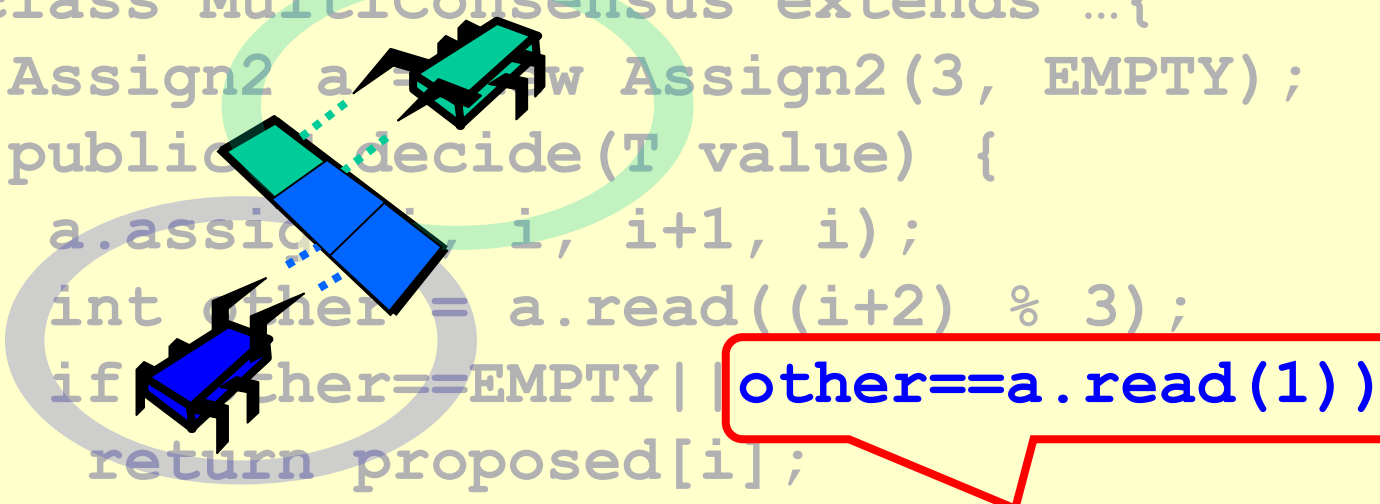
```



**Other thread didn't
move, so I win**

Multi-Consensus Code

```
class MultiConsensus extends ...{
  Assign2 a = new Assign2(3, EMPTY);
  public T decide(T value) {
    a.assign(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];
  }
}
```

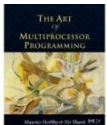


Other thread moved later so I win

Multi-Consensus Code

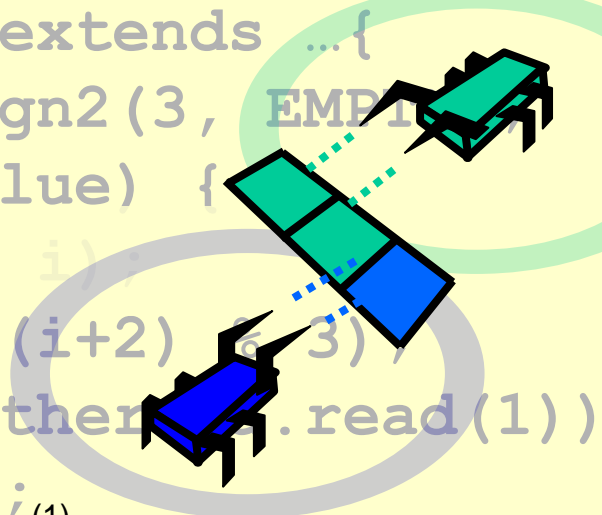
```
class MultiConsensus extends ...{
    Assign2 a = new Assign2(3, EMPTY);
    public T decide(T value) {
        a.assign(i, i, i+1, i);
        int other = a.read((i+2) % 3);
        if (other==EMPTY||other==a.read(1))
            return proposed[i];
        else
            return proposed[j];
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

OK, I win.



Multi-Consensus Code

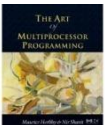
```
class MultiConsensus extends ...{
  Assign2 a = new Assign2(3, EMPTY);
  public T decide(T value) {
    a.assign(i, i, i+1, i);
    int other = a.read((i+2) % 3);
    if (other==EMPTY || other==a.read(1))
      return proposed[i];
    else
      return proposed[j];
  }
}
```



Other thread moved first, so I lose

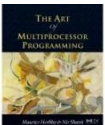
Summary

- 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

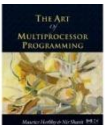
- Method call
 - Returns object's prior value **x**
 - Replaces **x** with **mumble(x)**



Read-Modify-Write

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

    public int synchronized
    getAndMumble() {
        int prior = value;
        value = mumble(value);
        return prior;
    }
}
```

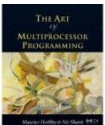


Read-Modify-Write

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

    public int synchronized
    getAndMumble() {
        int prior = value;
        value = mumble(value);
        return prior;
    }
}
```

Return prior value

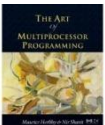


Read-Modify-Write

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

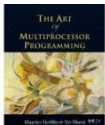
    public int synchronized
    getAndMumble() {
        int prior = value;
        value = mumble(value);
        return prior;
    }
}
```

Apply function to current value



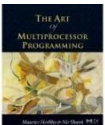
RMW Everywhere!

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



Example: Read

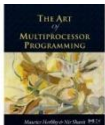
```
public abstract class RMWRegister {  
    private int value;  
  
    public int synchronized read() {  
        int prior = value;  
        value = value;  
        return prior;  
    }  
}
```



Example: Read

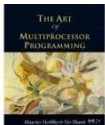
```
public abstract class RMW {  
    private int value;  
  
    public int synchronized read() {  
        int prior = this.value;  
        value = value;  
        return prior;  
    }  
}
```

apply $f(x)=x$, the identity function



Example: getAndSet

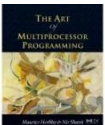
```
public abstract class RMWRegister {  
    private int value;  
  
    public int synchronized  
        getAndSet(int v) {  
        int prior = value;  
        value = v;  
        return prior;  
    }  
    ...  
}
```



Example: getAndSet (swap)

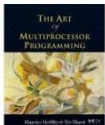
```
public abstract class RMWRegister {  
    private int value;  
  
    public int synchronized  
        getAndSet(int v) {  
        int prior = value;  
        value = v;  
        return prior;  
    }  
    ...  
}
```

f(x)=v is constant



getAndIncrement

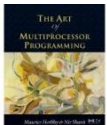
```
public abstract class RMWRegister {  
    private int value;  
  
    public int synchronized  
        getAndIncrement() {  
        int prior = value;  
        value = value + 1;  
        return prior;  
    }  
    ...  
}
```



getAndIncrement

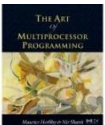
```
public abstract class RMWRegister {  
    private int value;  
  
    public int synchronized  
        getAndIncrement() {  
        int prior = value;  
        value = value + 1;  
        return prior;  
    }  
    ...  
}
```

$f(x) = x+1$



getAndAdd

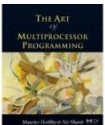
```
public abstract class RMWRegister {  
    private int value;  
  
    public int synchronized  
        getAndAdd(int a) {  
        int prior = value;  
        value = value + a;  
        return prior;  
    }  
    ...  
}
```



Example: getAndAdd

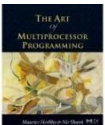
```
public abstract class RMWRegister {  
    private int value;  
  
    public int synchronized  
        getAndIncrement(int a) {  
        int prior = value;  
        value = value + a;  
        return prior;  
    }  
    ...  
}
```

$f(x) = x+a$



compareAndSet

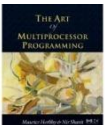
```
public abstract class CASObject {  
    private int value;  
    public boolean synchronized  
        compareAndSet(int expected,  
                       int update) {  
        if (value==expected) {  
            value = update; return true;  
        }  
        return false;  
    } ... } 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



compareAndSet

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

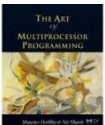
If value is as expected, ...



compareAndSet

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

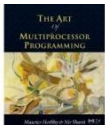
... replace it



compareAndSet

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

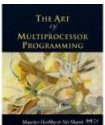
Report success



compareAndSet

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

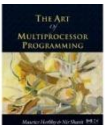
Otherwise report failure



Read-Modify-Write

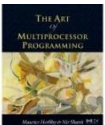
```
public abstract class RMWRegister {  
    private int value;  
  
    public void synchronized  
    getAndMumble() {  
        int prior = value;  
        value = mumble(value) ;  
        return prior;  
    }  
}
```

Lets characterize f(x)...



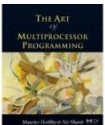
Definition

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



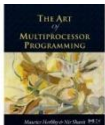
Par Example

- `Identity(x) = x`
 - is trivial
- `getAndIncrement(x) = x+1`
 - is non-trivial



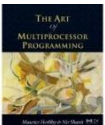
Theorem

- 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



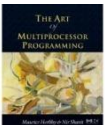
Reminder

- 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

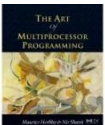
```
public class RMWConsensus
    extends ConsensusProtocol {
    private RMWRegister r = v;
    public T decide(T value) {
        propose(value);
        if (r.getAndMumble() == v)
            return proposed[i];
        else
            return proposed[j];
    }
}
```



Proof

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

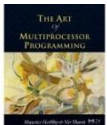
Initialized to v



Proof

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

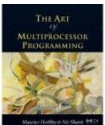
Am I first?



Proof

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

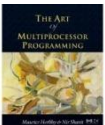
Yes, return my input



Proof

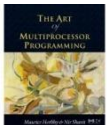
```
public class RMWConsensus
    extends ConsensusProtocol {
private RMWRegister r = v;
public T decide(T value) {
    propose(value);
    if (r.getAndMumble() == v)
        return proposed[i];
    else
        return proposed[j];
}}
```

**No, return
other's input**



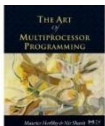
Proof

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



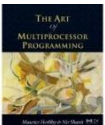
Interfering RMW

- 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

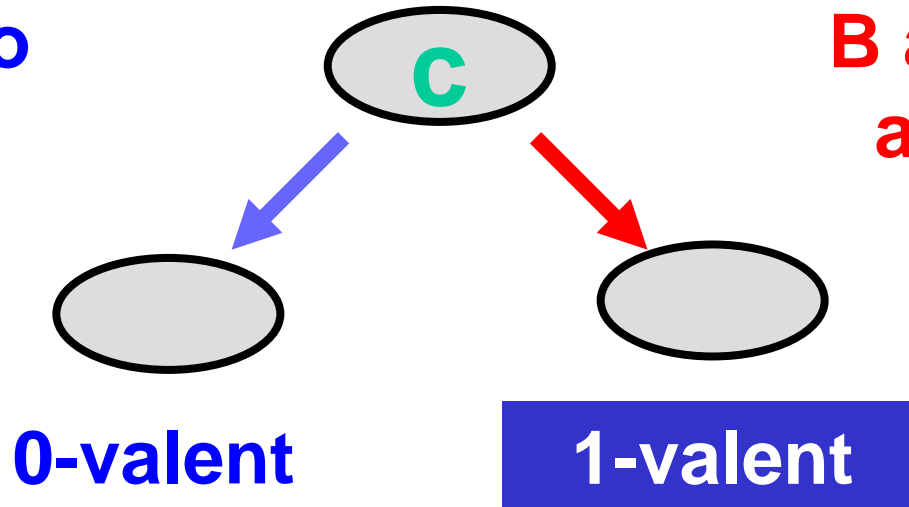
- “test-and-set” `getAndSet(1)` $f(v)=1$
Overwrite $f_i(f_j(v))=f_i(v)$
- “swap” `getAndSet(x)` $f(v,x)=x$
Overwrite $f_i(f_j(v))=f_i(v)$
- “fetch-and-inc” `getAndIncrement()` $f(v)=v+1$
Commute $f_i(f_j(v))=f_j(f_i(v))$



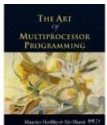
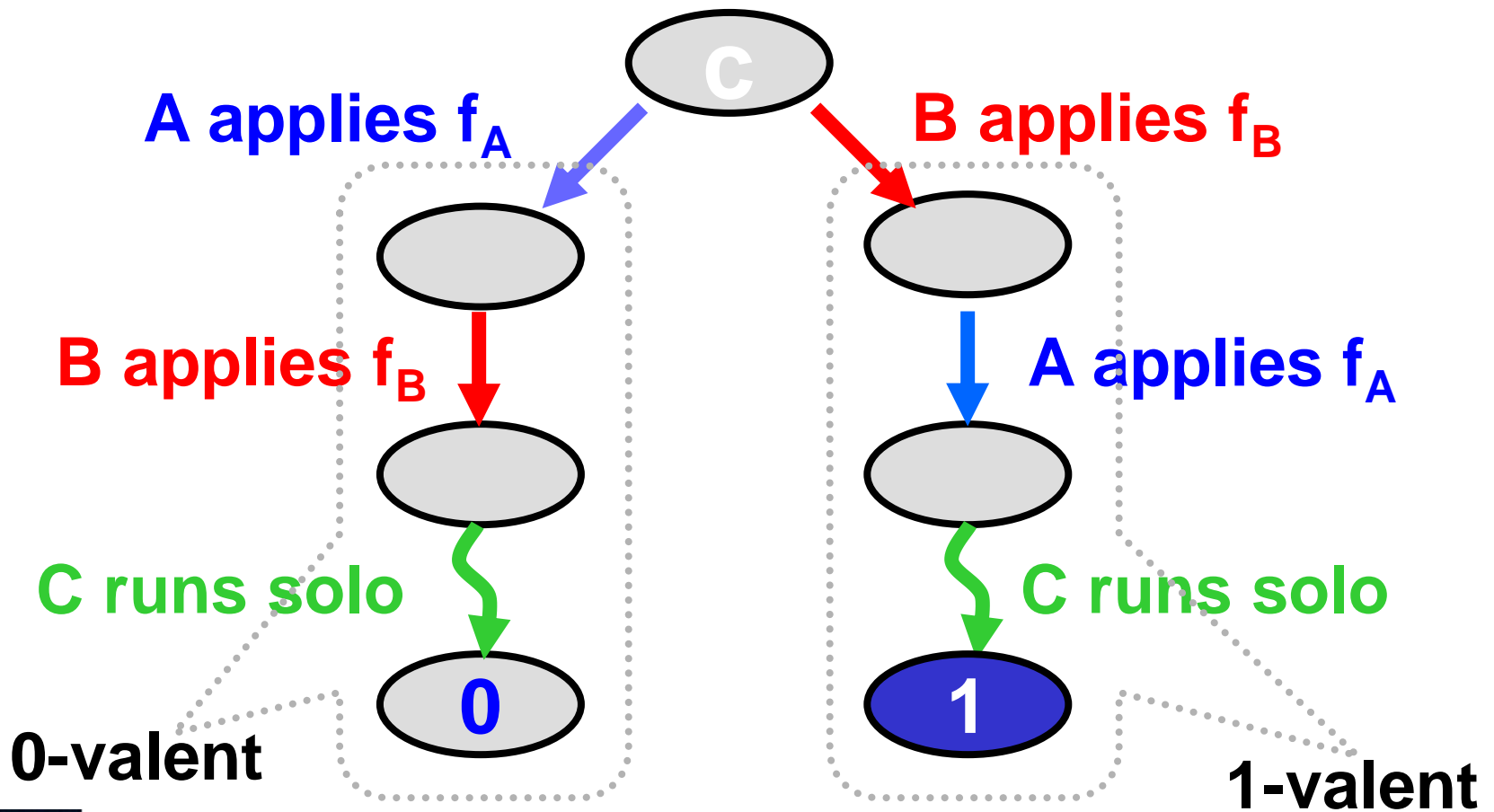
Meanwhile Back at the Critical State

A about to apply f_A

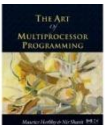
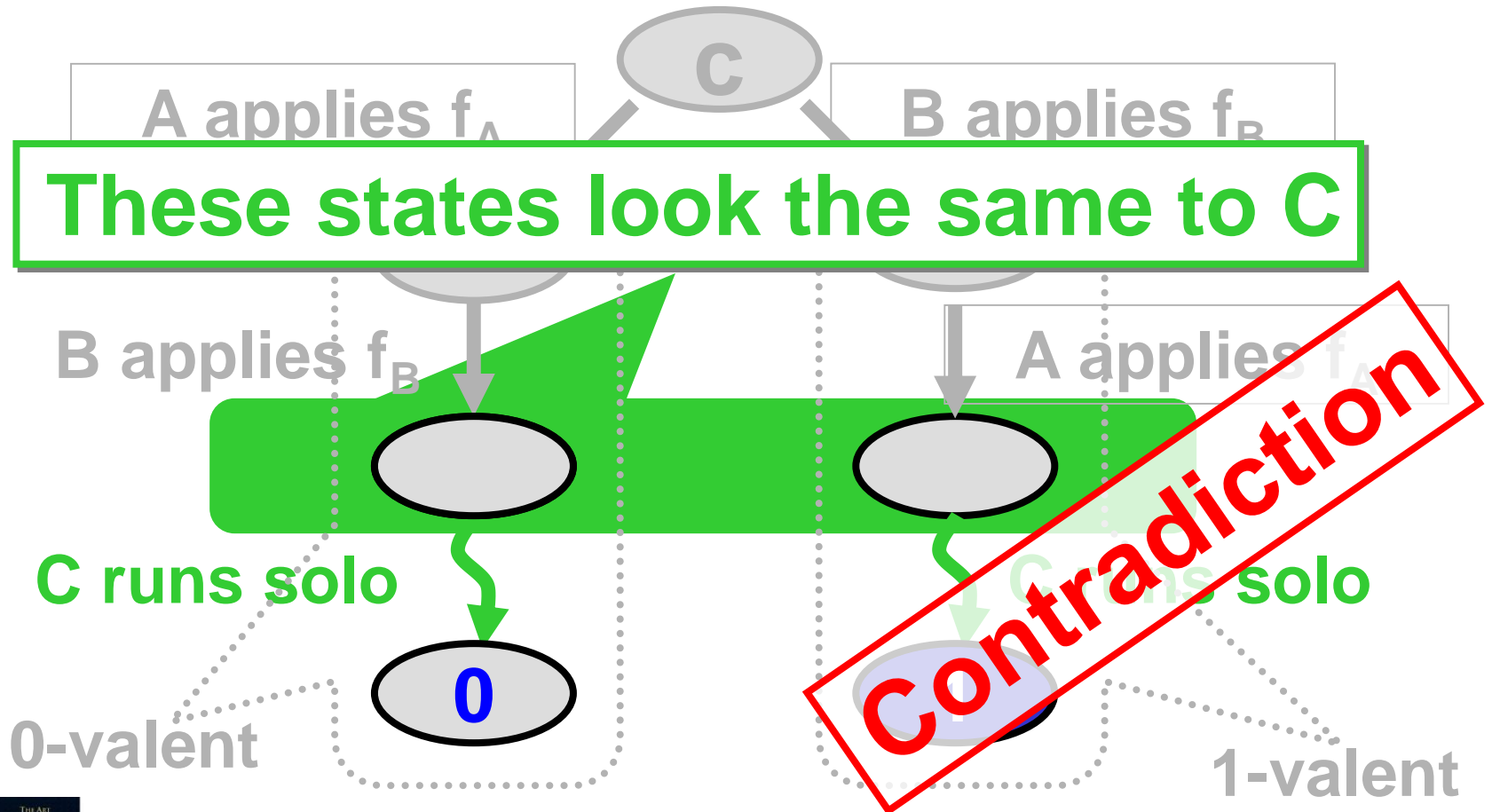
B about to apply f_B



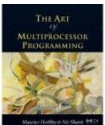
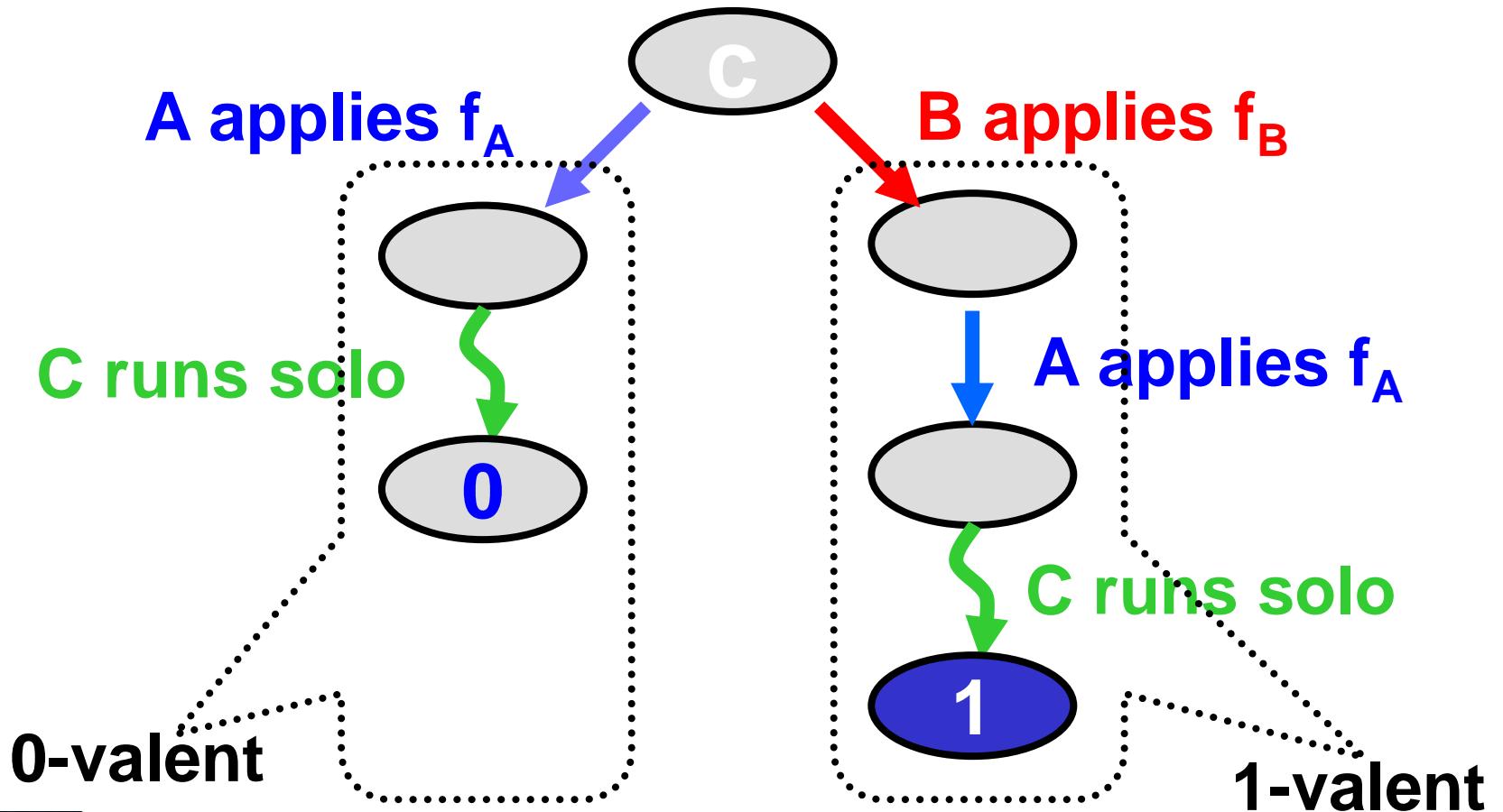
Maybe the Functions Commute



Maybe the Functions Commute

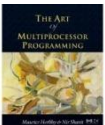
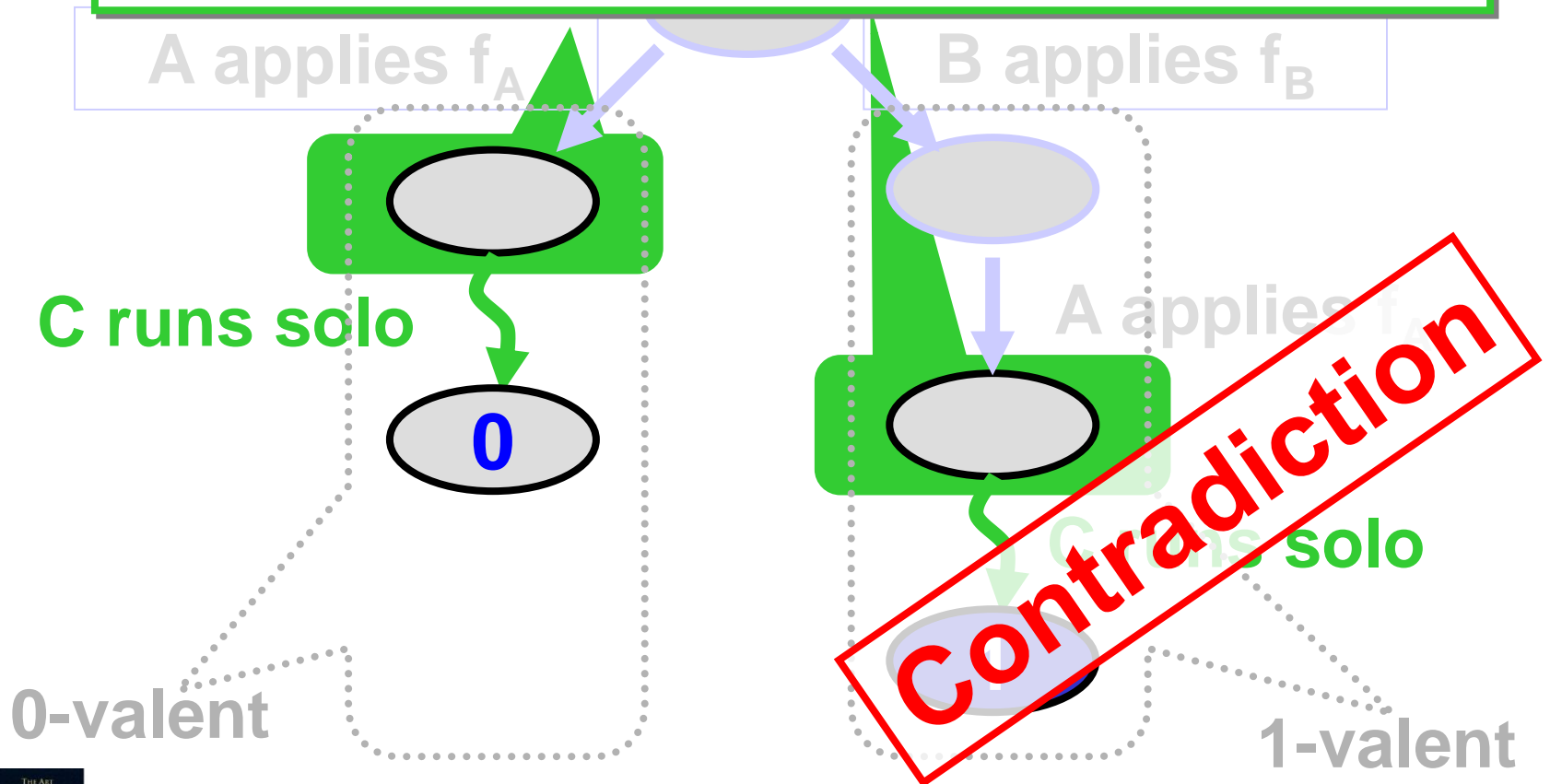


Maybe the Functions Overwrite



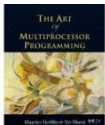
Maybe the Functions Overwrite

These states look the same to C



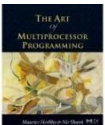
Impact

- Many early machines provided only 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 = value;
        if (value==expected) {
            value = update; return true;
        }
        return false;
    } ... }
```

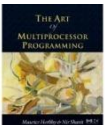


compareAndSet

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

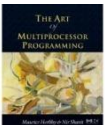
```

**replace value if it's what we
expected, ...**



compareAndSet Has ∞ Consensus Number

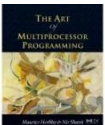
```
public class RMWConsensus
    extends ConsensusProtocol {
    private AtomicInteger r =
        new AtomicInteger(-1);
    public T decide(T value) {
        propose(value);
        r.compareAndSet(-1, i);
        return proposed[r.get()];
    }
}
```



compareAndSet Has ∞ Consensus Number

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

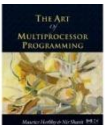
Initialized to -1



compareAndSet Has ∞ Consensus Number

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

Try to swap in my
id

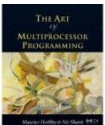


compareAndSet Has ∞ Consensus Number

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

Decide winner's preference

return proposed[r.get()];



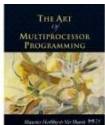
The Consensus Hierarchy

1 Read/Write Registers, Snapshots...

2 getAndSet, getAndIncrement, ...

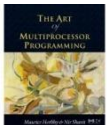
▪
▪
▪

∞ 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
- Pragmatic approach
- Implies no mutual exclusion



Lock-Free vs. Wait-free

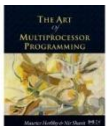
- Wait-Free: each method call takes a finite number of steps to finish
- Lock-free: infinitely often some method call finishes



Lock-Freedom

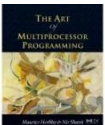


- Any wait-free implementation is lock-free.
- Lock-free is the same as wait-free if the execution is finite.



Lock-Free Implementations

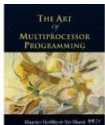
- Lock-free consensus is as impossible as wait-free consensus
- ***All these results hold for lock-free algorithms also.***



There is More: Universality

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

Stay tuned...



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.

