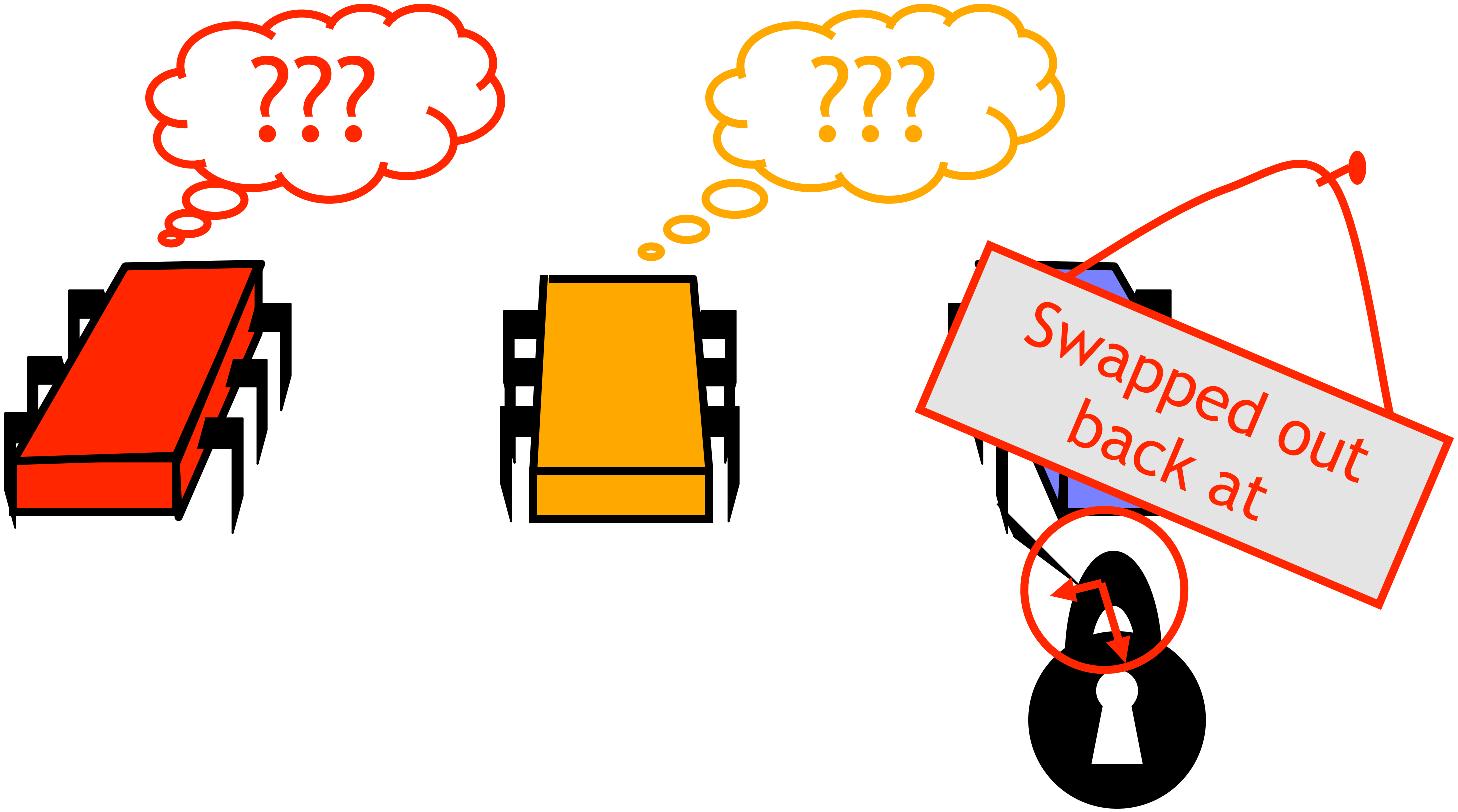
The Relative Power of Synchronization Operations Queues and Stacks

Christof Fetzer, TU Dresden

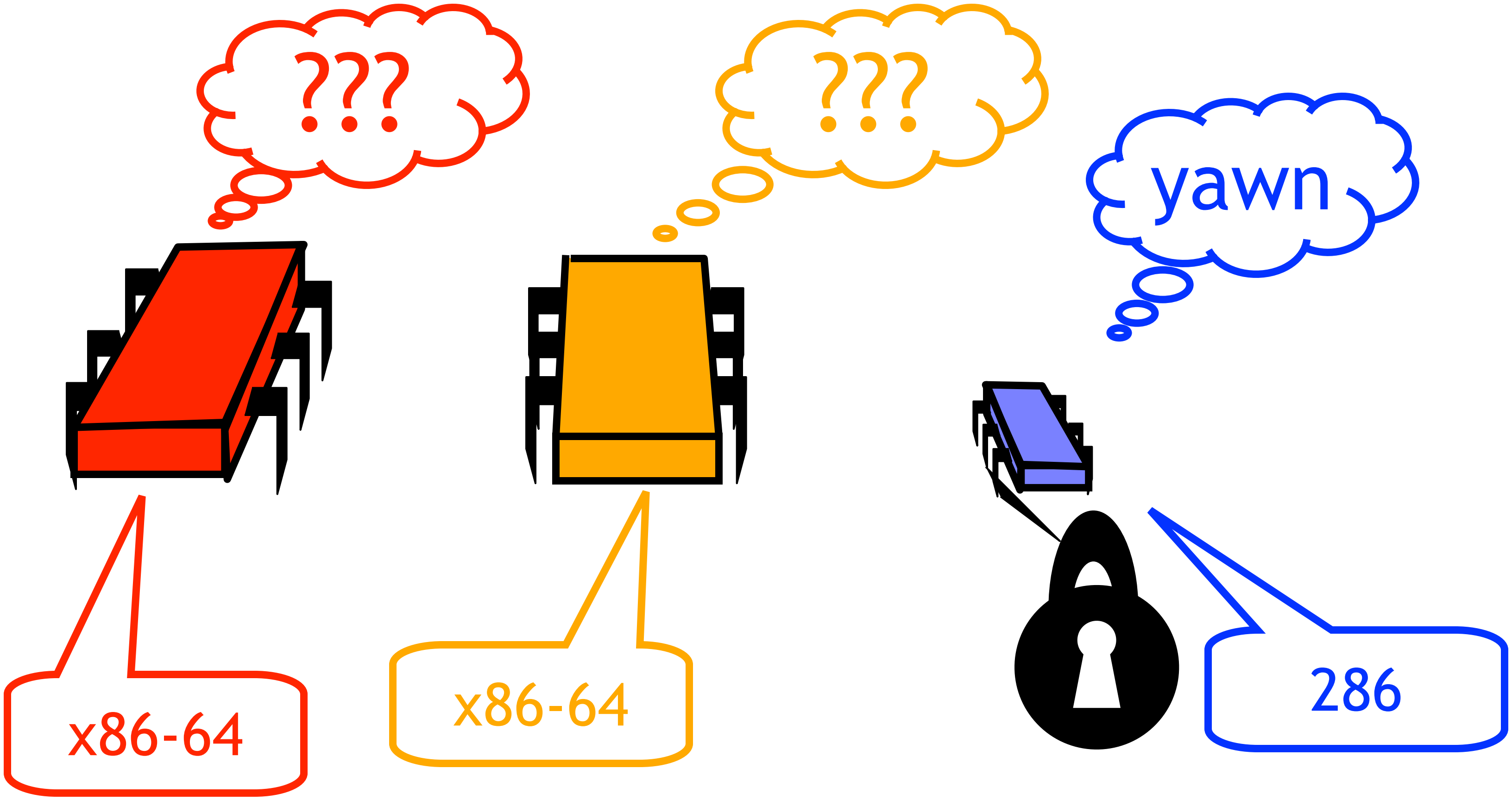
Based on slides by Maurice Herlihy and Nir Shavit

Why is Mutual Exclusion so wrong?

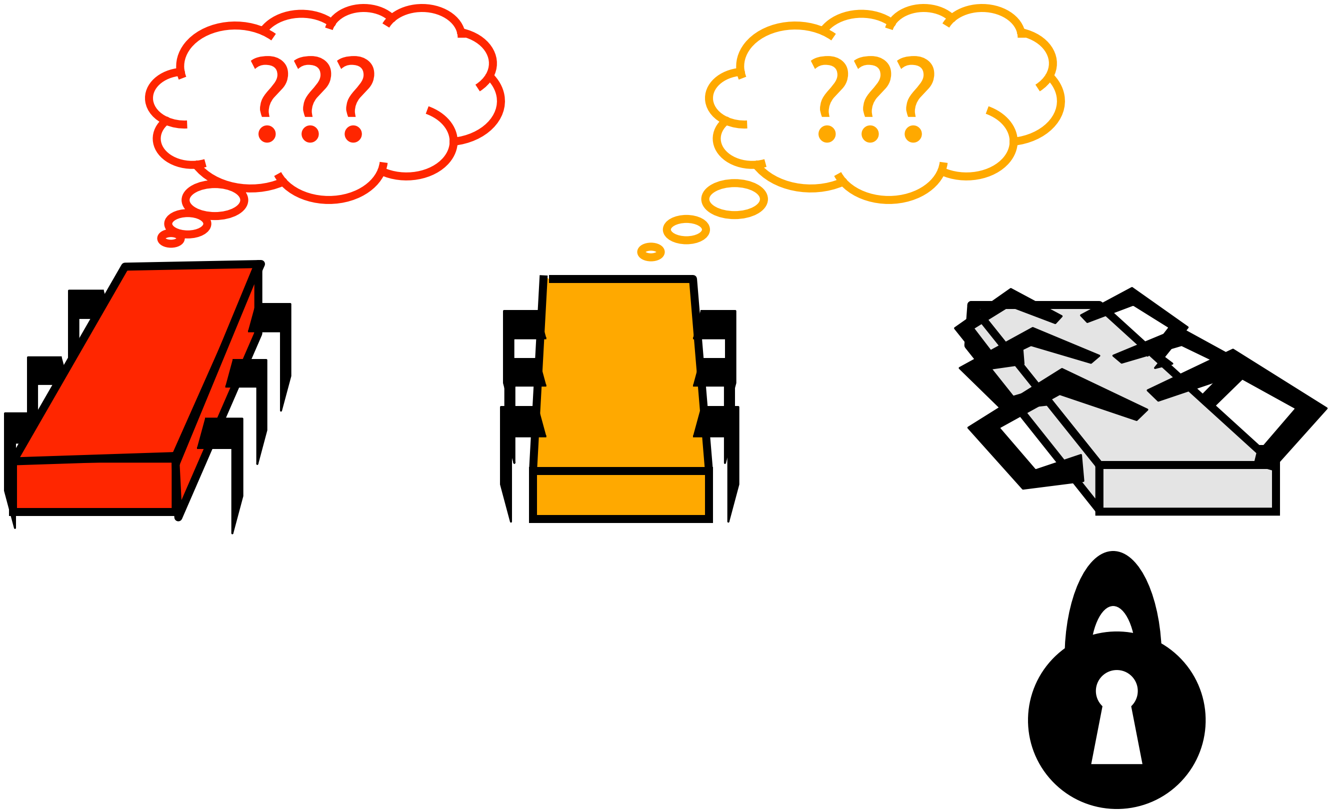
Asynchronous Interrupts



Heterogeneous Processors



Fault-tolerance



Wait-Free Implementations

Definition: An object implementation is **wait-free** if every thread completes a method in a finite number of steps

No mutual exclusion

- Thread could halt in critical section
- Build mutual exclusion from registers

Lock-Free Implementations

Definition: An object implementation is **lock-free** if in an infinite execution infinitely often some method call finishes (obviously, in a finite number of steps)

No difference between lock-free and wait-free for finite executions

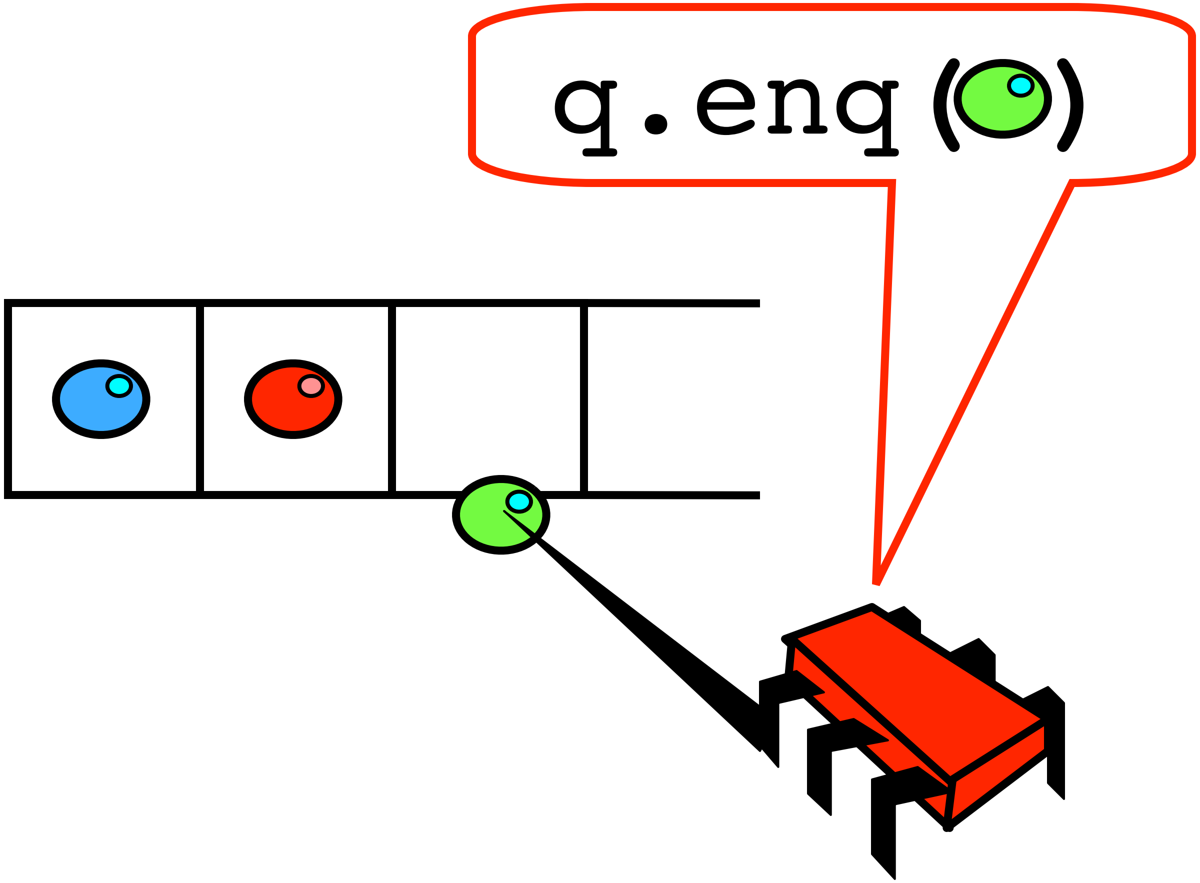
Basic Questions

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

Today: Focus on Wait-free

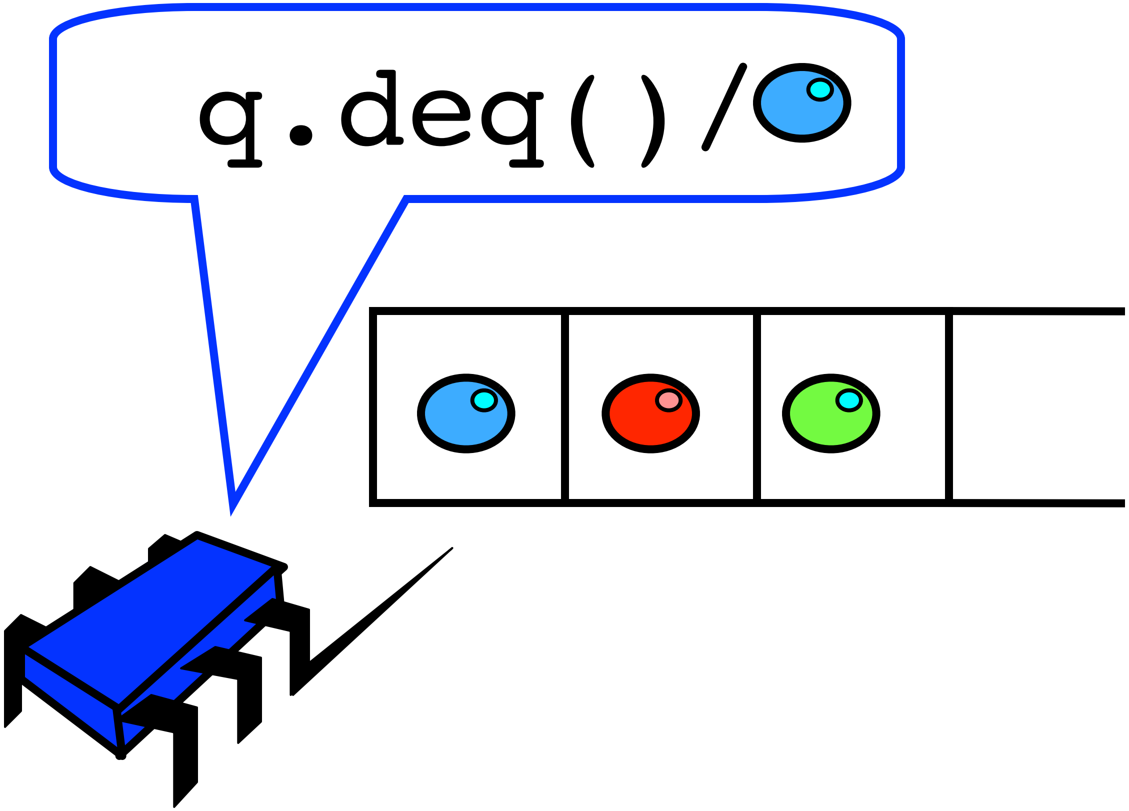
- The rest of today’s discussion will focus on wait-free implementations
- But the results we present apply almost verbatim to lock-free ones

FIFO Queue: Enqueue Method



`q.enq(●)`

FIFO Queue: Dequeue Method



`q.deq()/`●

Two-Thread Wait-Free Queue

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

Two-Thread Wait-Free Queue

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

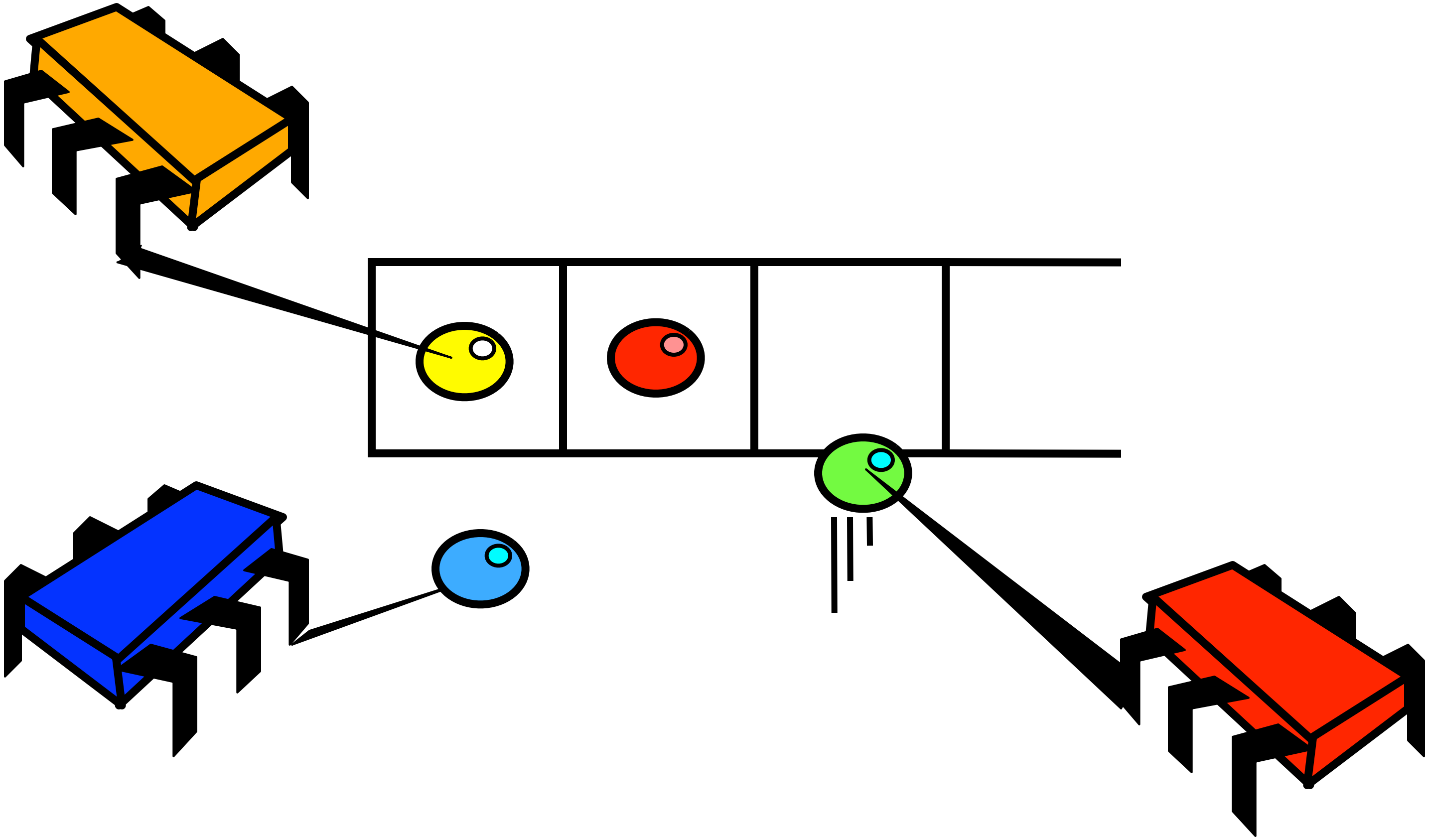
Put object in queue

Two-Thread Wait-Free Queue

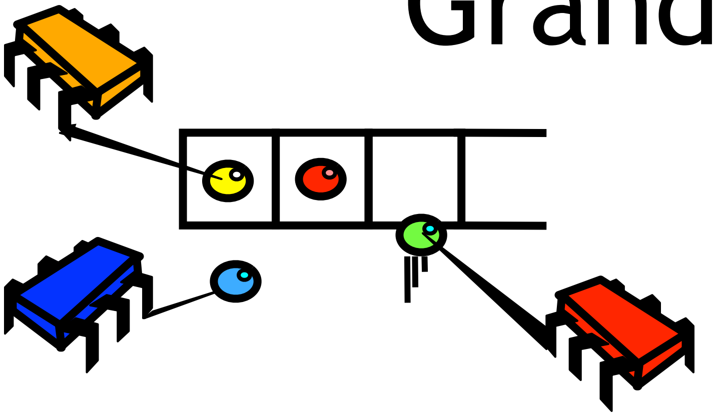
```
public class LockFreeQueue {
 int head = 0, tail = 0;
 Item[QSIZE] items;
 public void enq(Item x) throws… {
  if (tail-head == QSIZE) { throw…};
  items[tail % QSIZE] = x; tail++;
  }
 public Item deq() {
  if (tail-head == 0) { throw…}
  Item item = items[head % QSIZE];
  head++; return item;
}}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

Increment tail counter

What About Multiple Dequeuers?



Grand Challenge



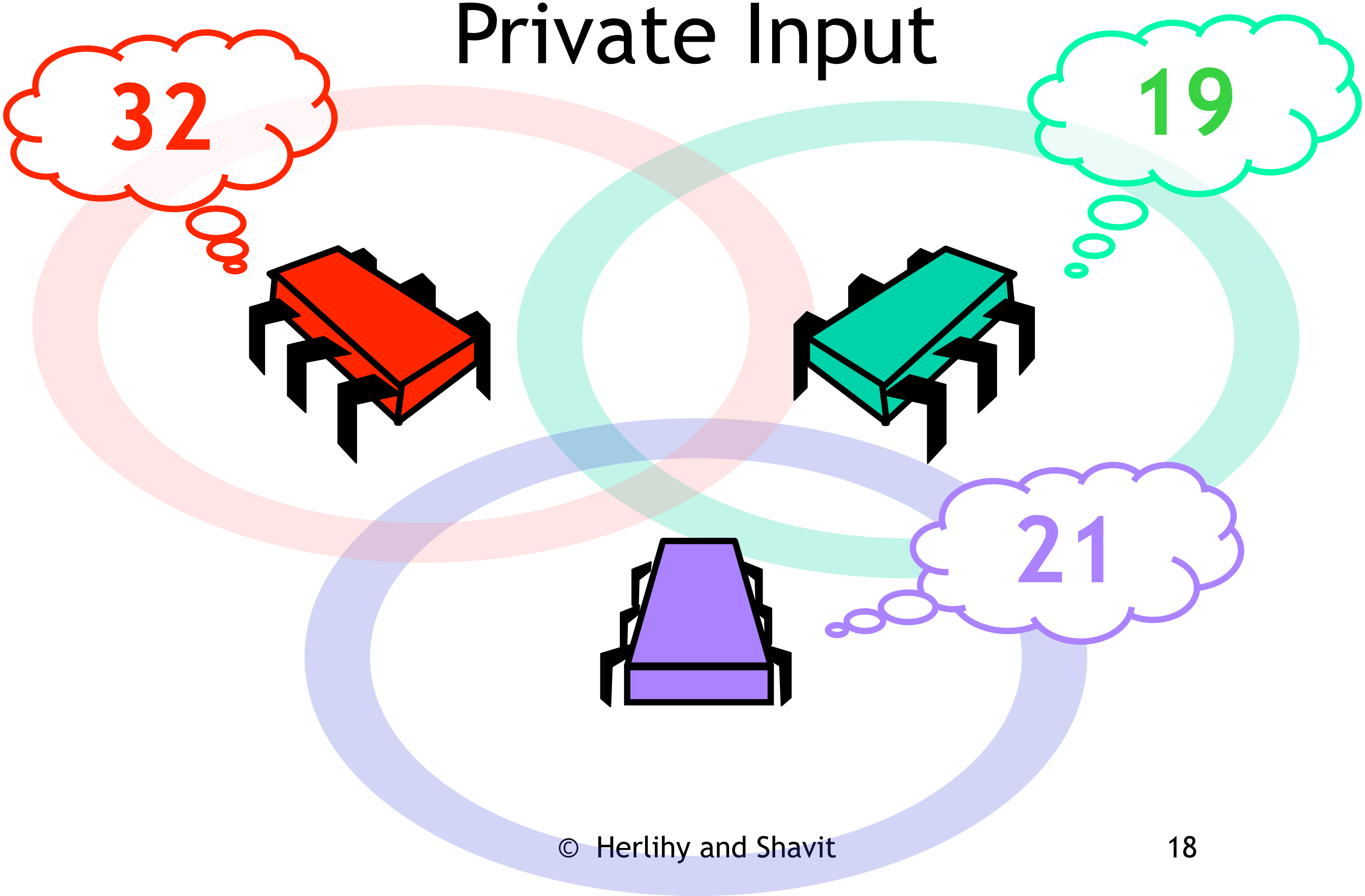
- Implement a FIFO queue
  - Wait-free
  - Linearizable
  - From atomic read-write registers
  - Multiple dequeuers

Only new aspect

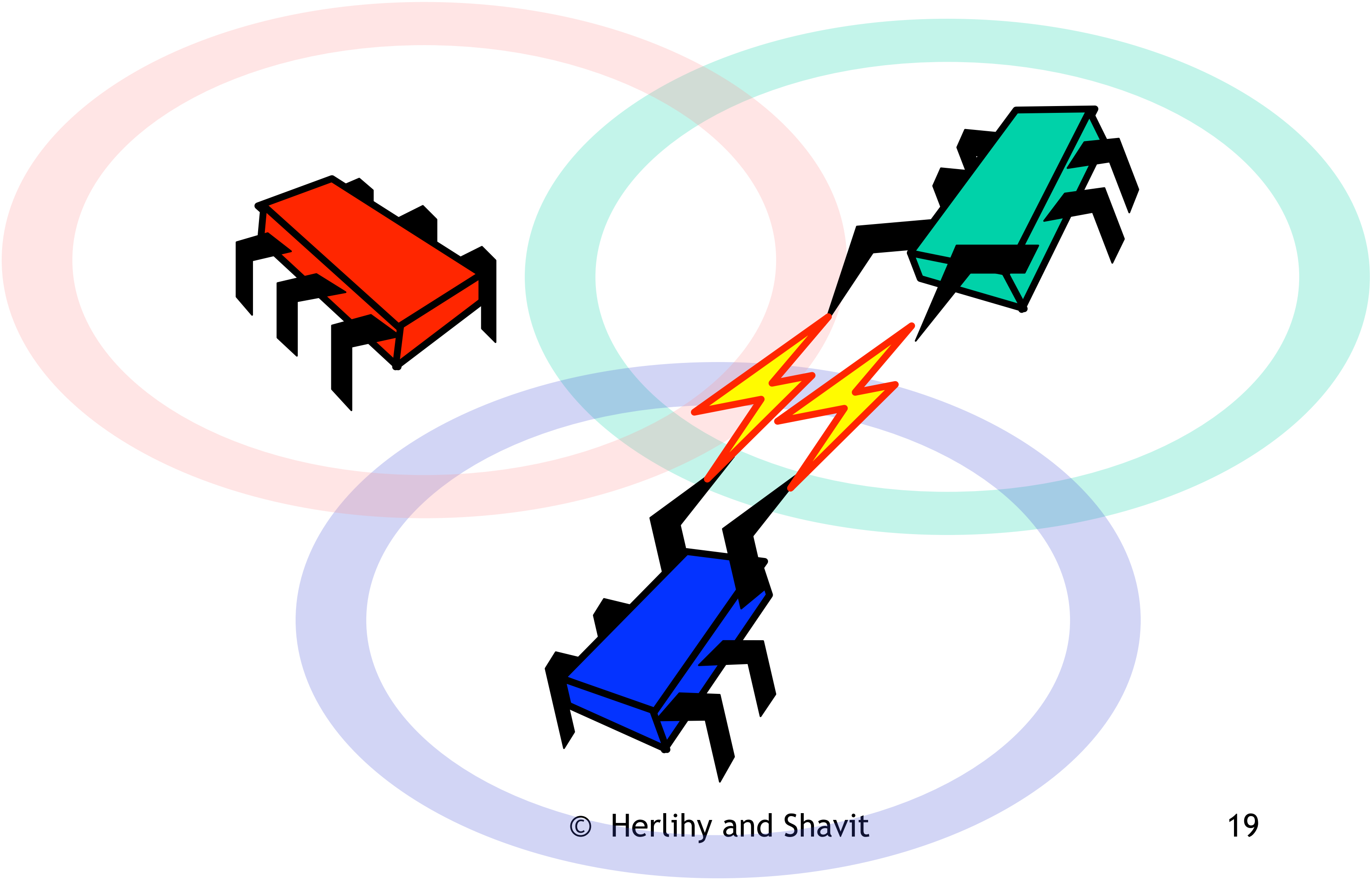
Consensus

- While you are thinking about the grand challenge…
- We will give you another puzzle
  - Consensus
  - Will be important …

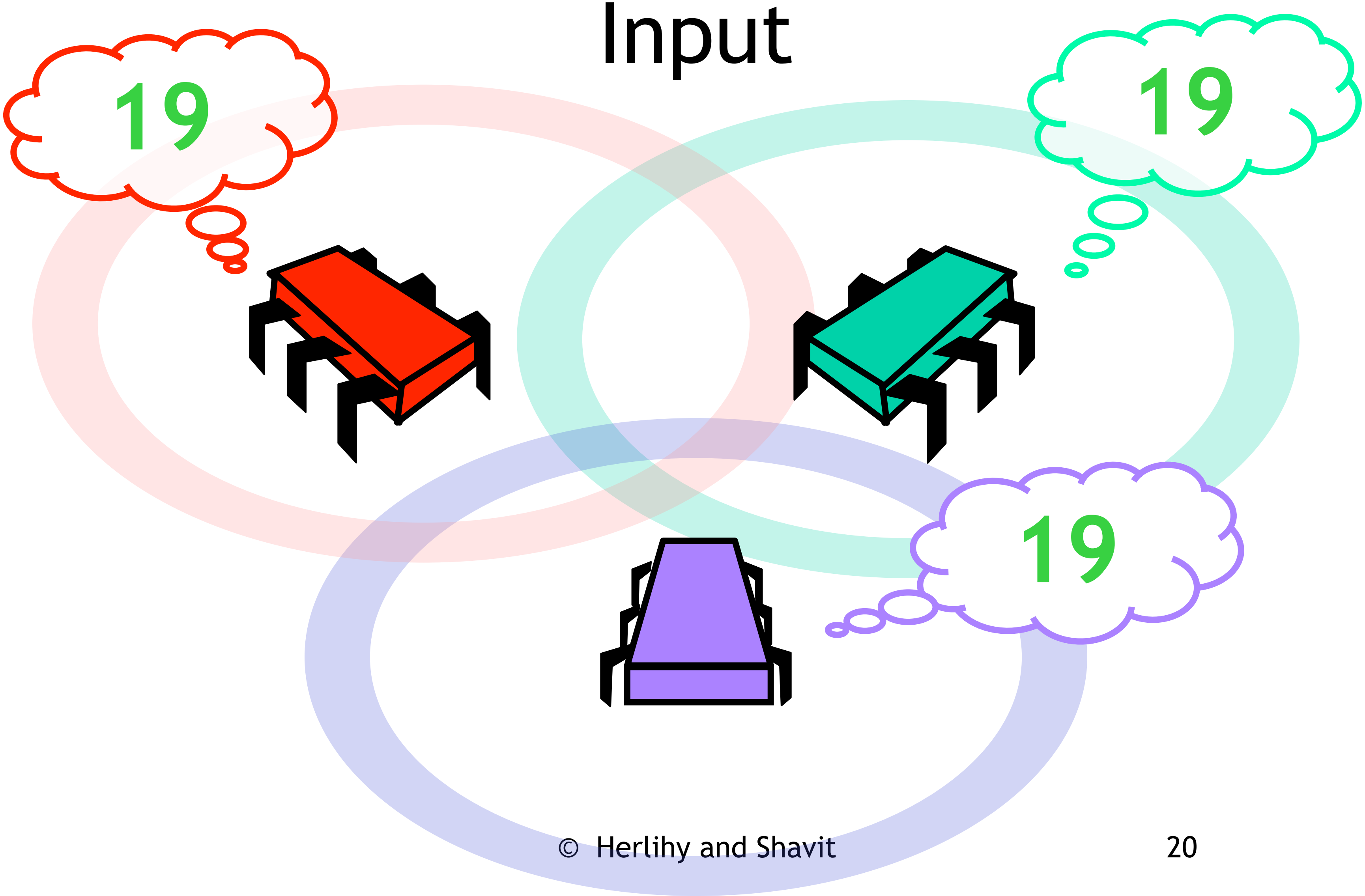
Consensus: Each Thread has a Private Input



They Communicate



They Agree on One Thread's Input



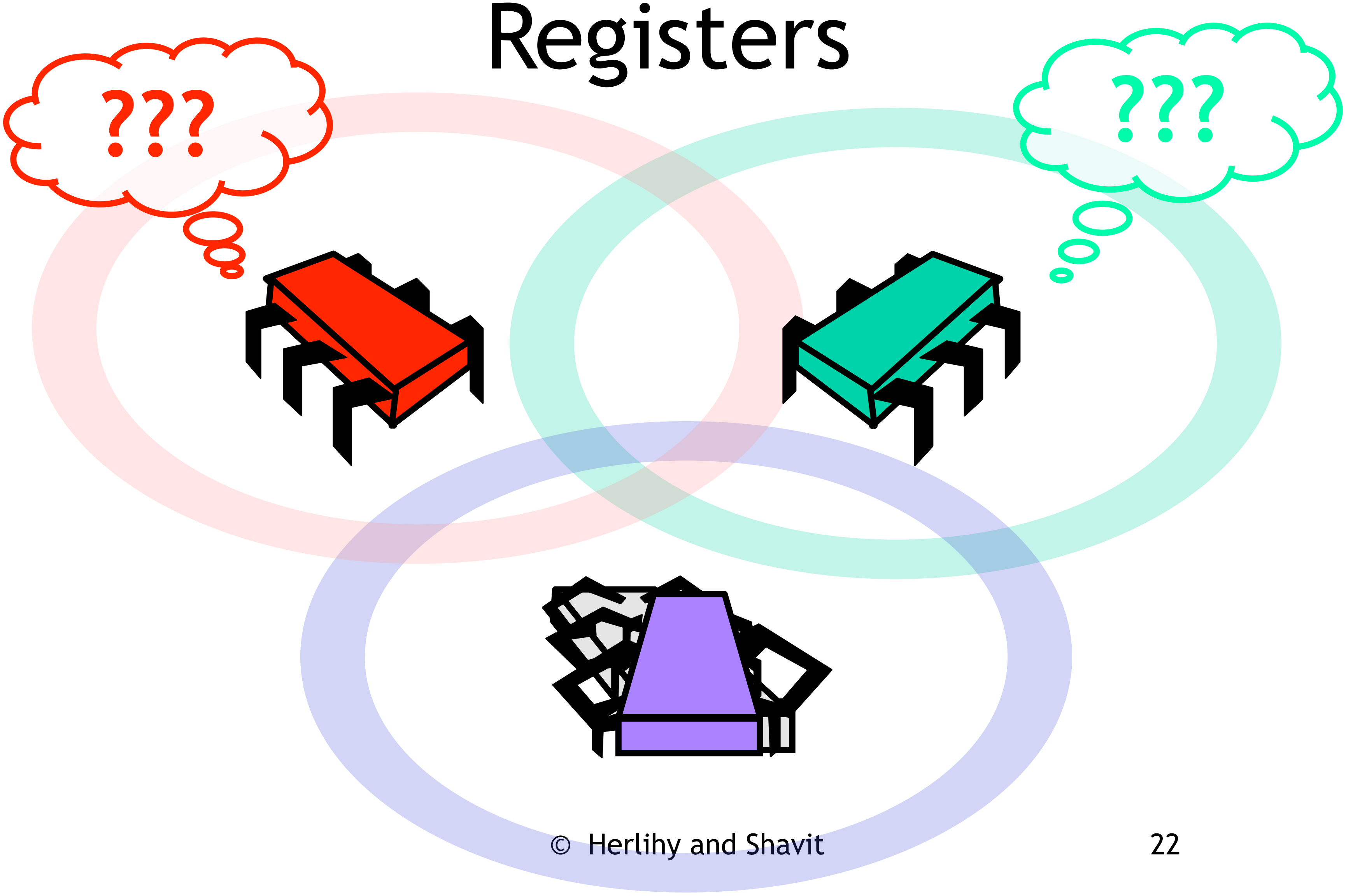
Formally: Consensus

Consistent: all threads decide the same value

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

Wait-free: each thread decides after a finite number of steps

No Wait-Free Consensus using Registers



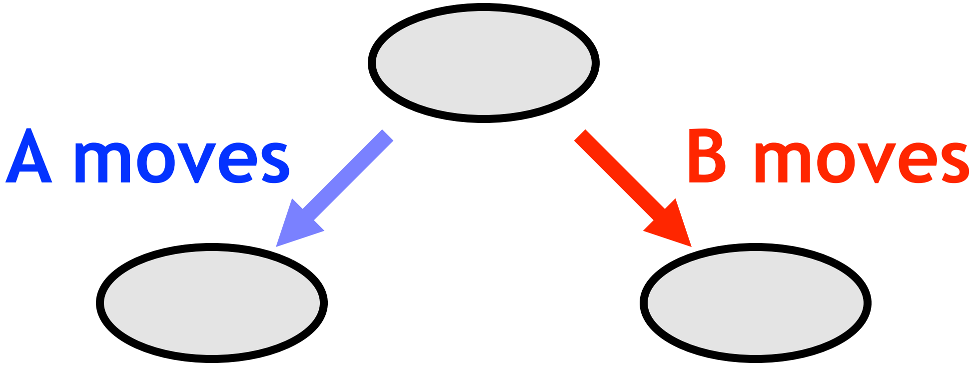
Formally

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

Proof Strategy

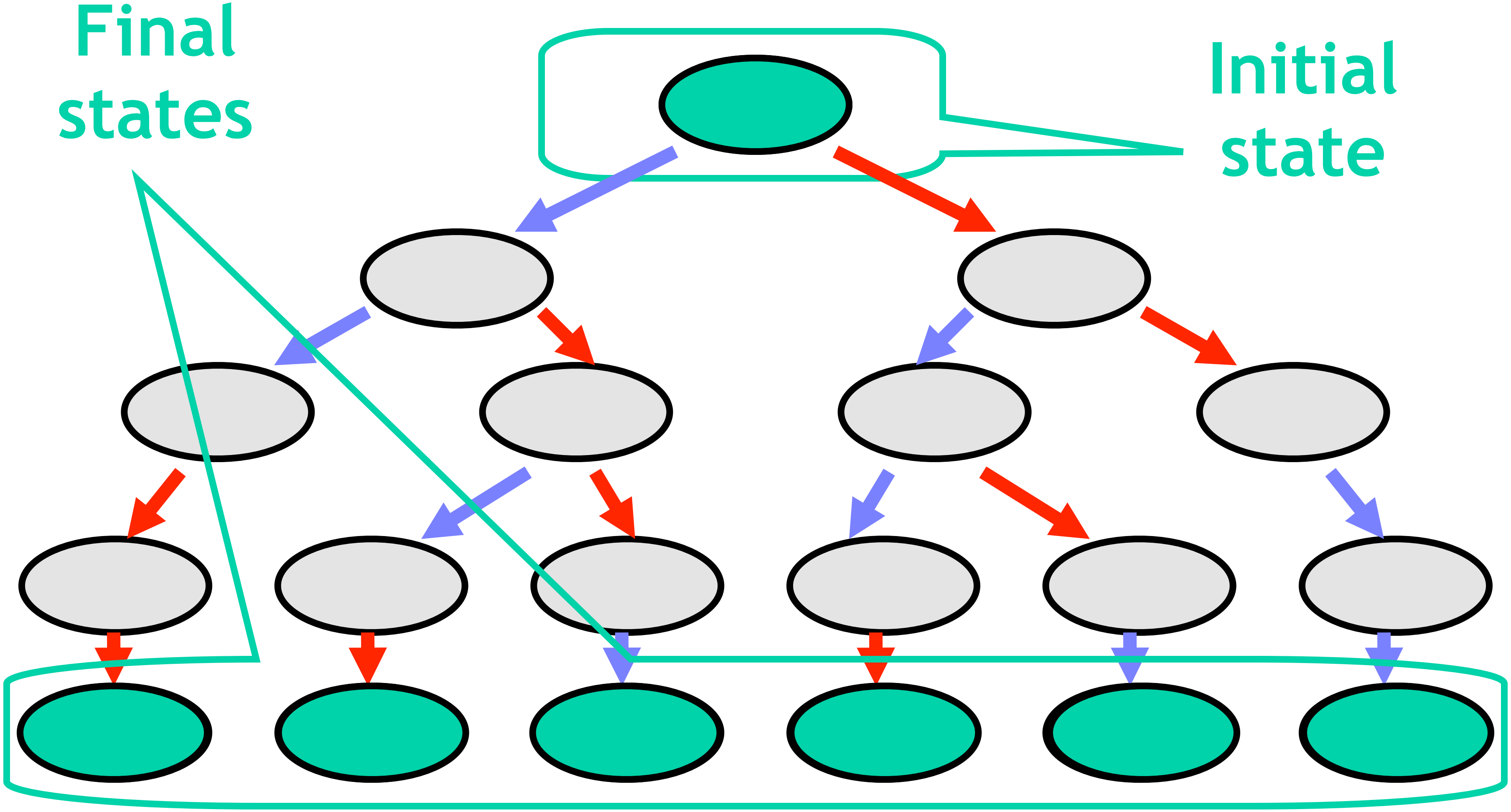
- Assume otherwise
- Reason about the properties of any such protocol
- Derive a contradiction
- Quod Erat Demonstrandum
- Suffices to prove for binary consensus and n=2

Wait-Free Computation



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

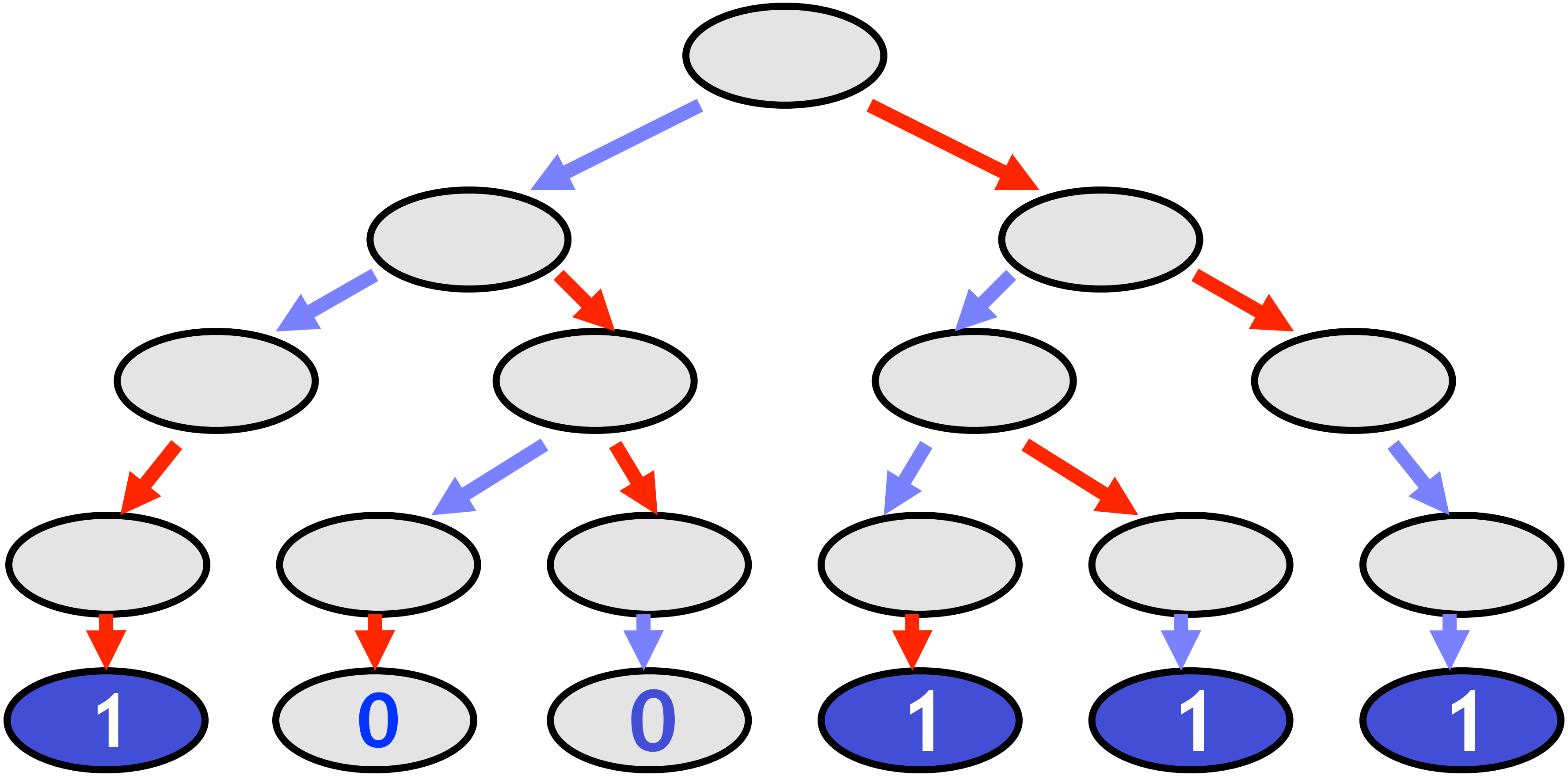
The Two-Move Tree



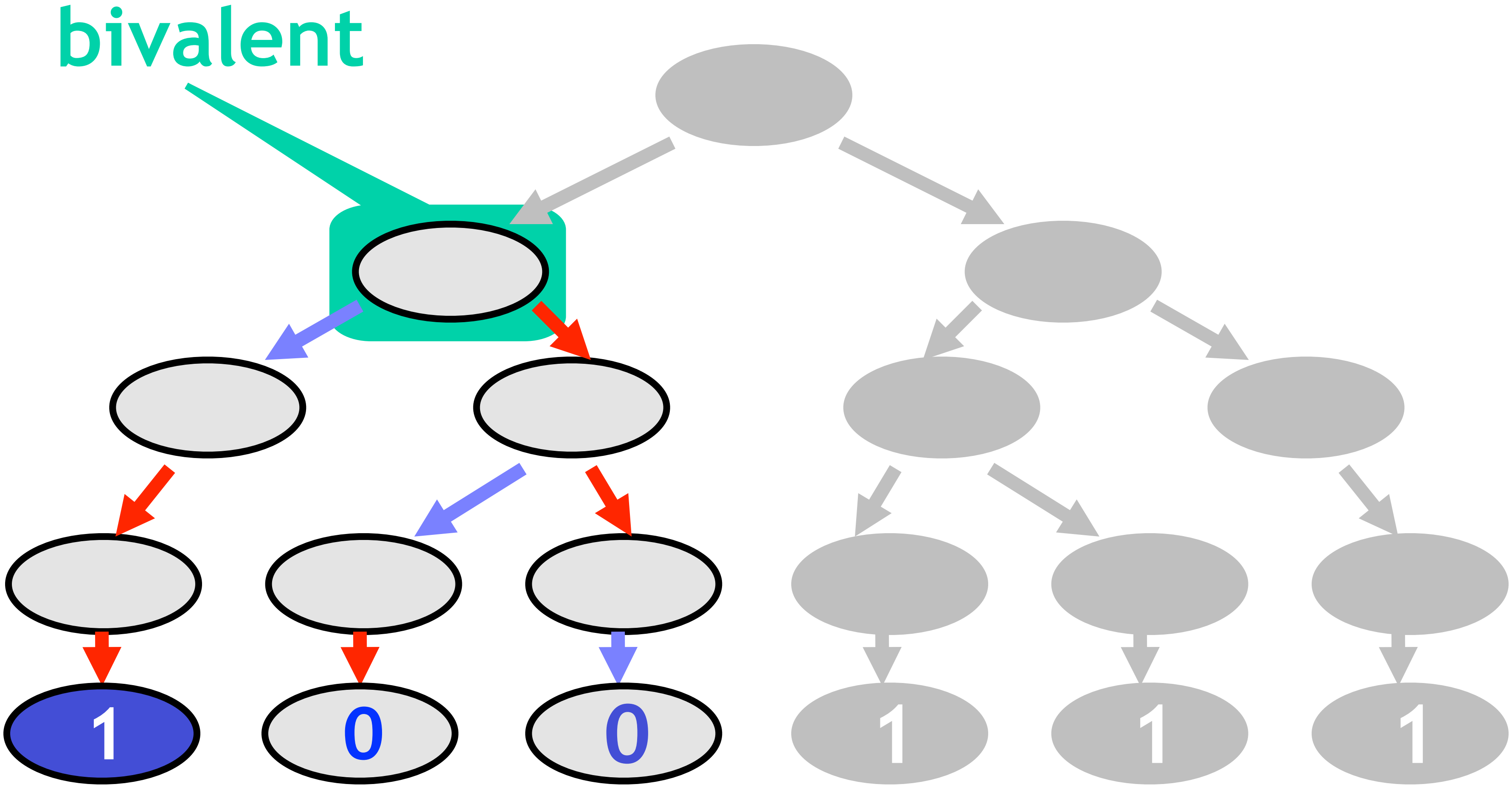
Final states

Initial state

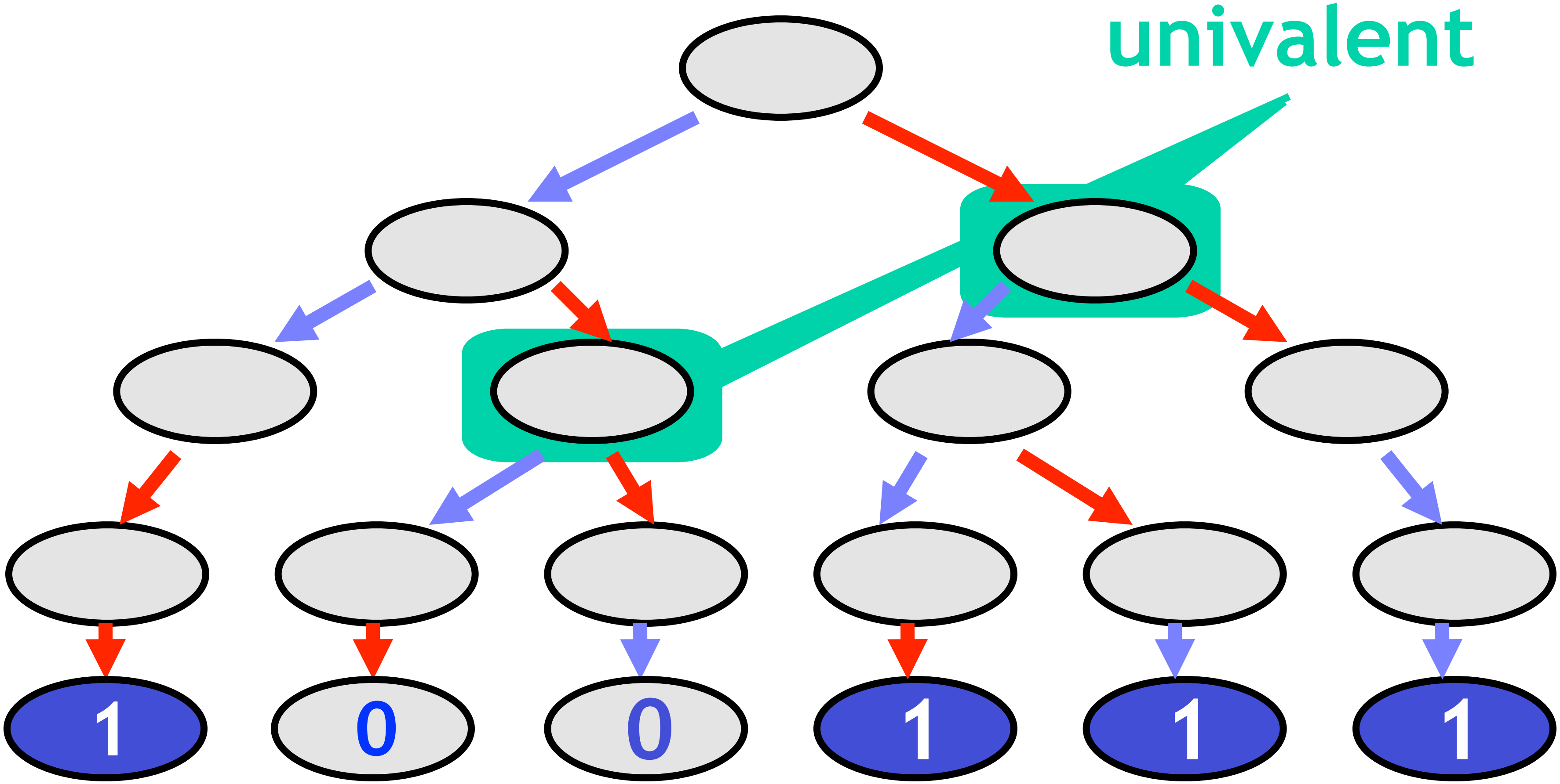
Decision Values



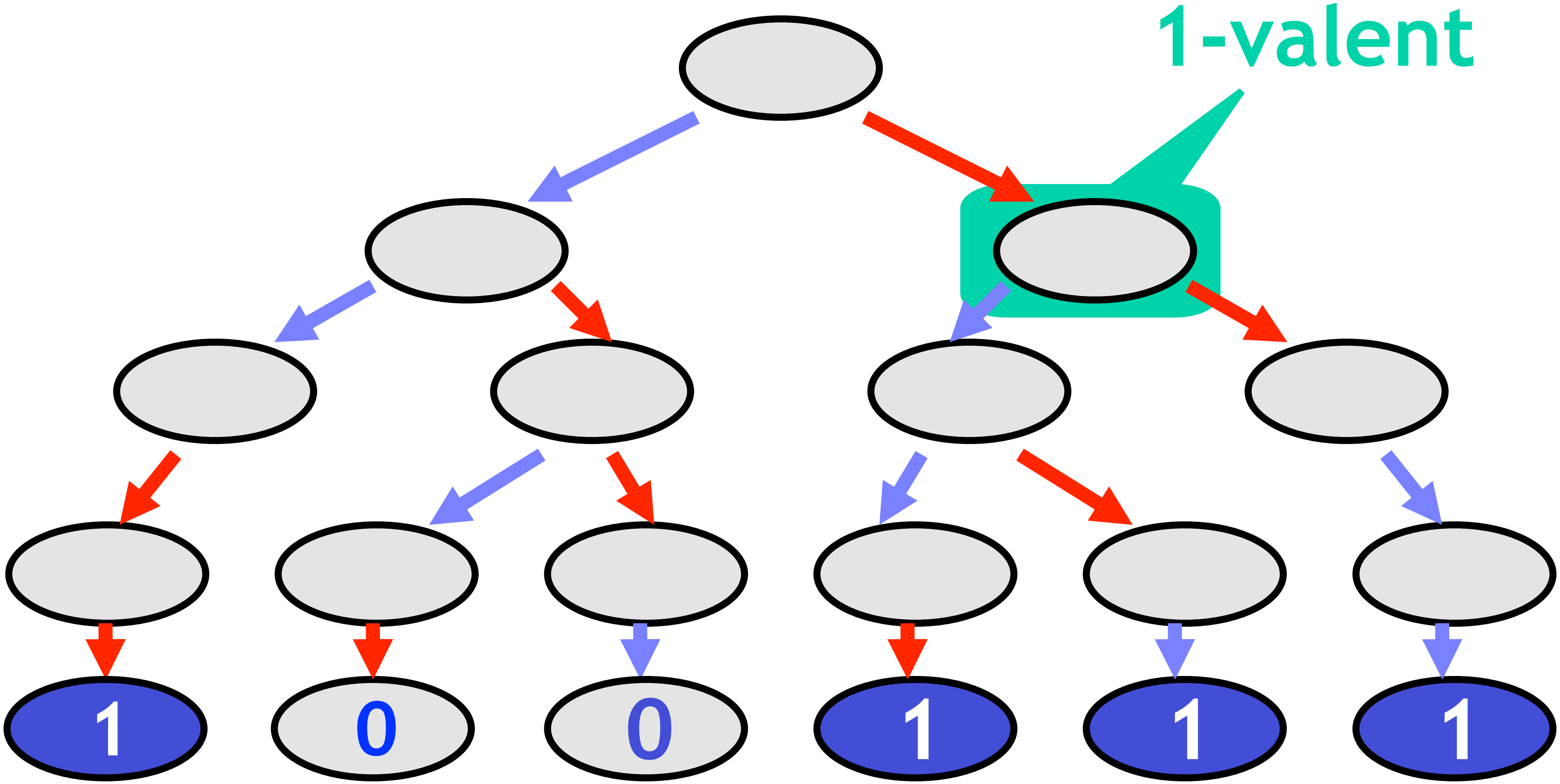
Bivalent: Both Possible



Univalent: Single Value Possible



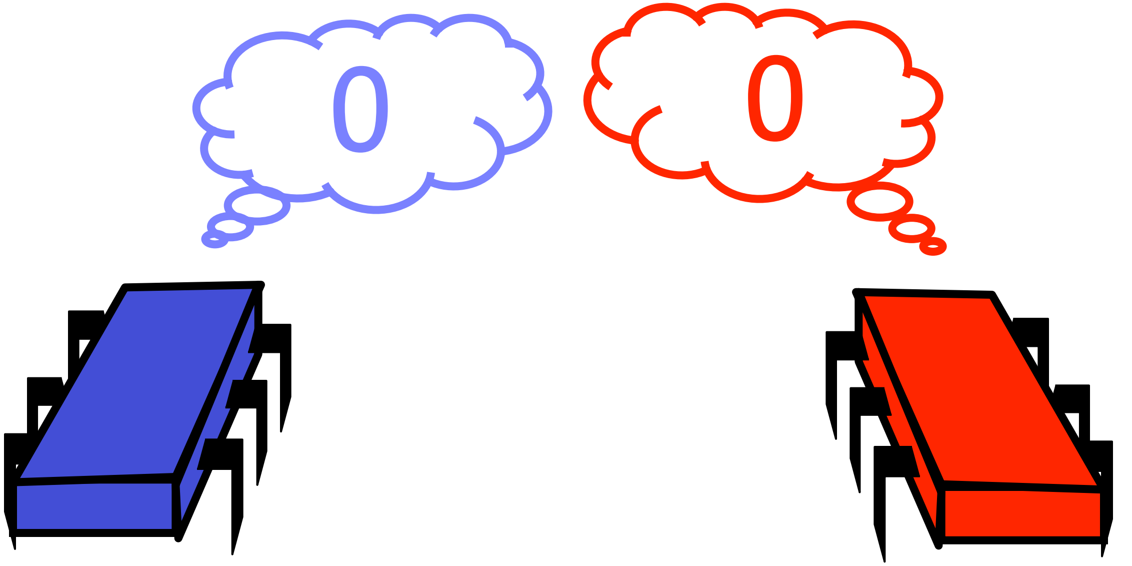
x-valent: x Only Possible Decision



Claim

- Some initial state is bivalent
- Outcome depends on
  - Chance
  - Behavior of the scheduler
- Lets prove this claim

Both Inputs 0

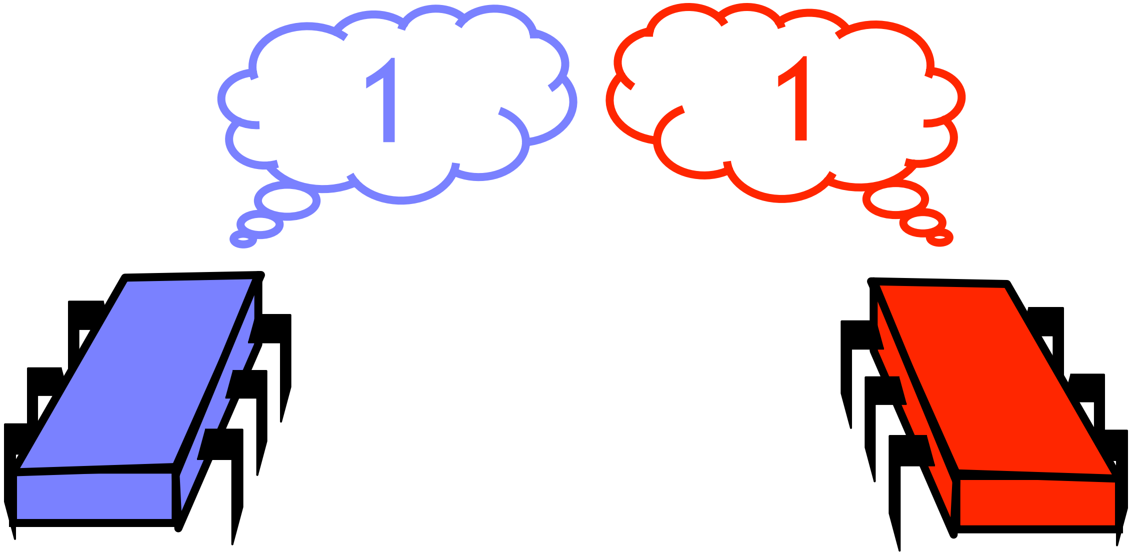


Univalent: all executions must decide 0

Both Inputs 0

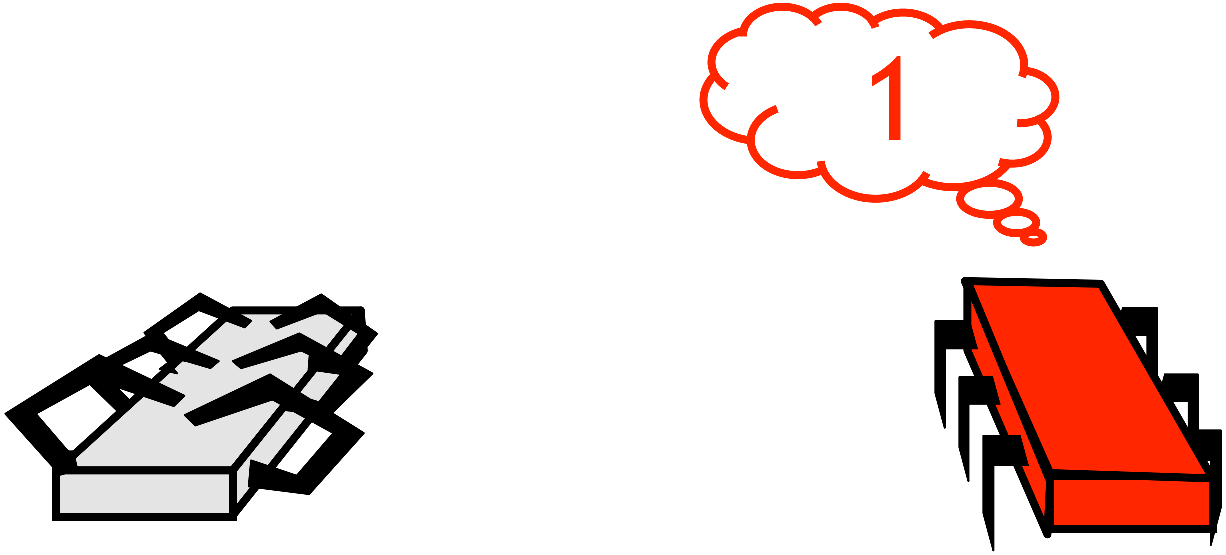
Including this solo execution by A

Both Inputs 1



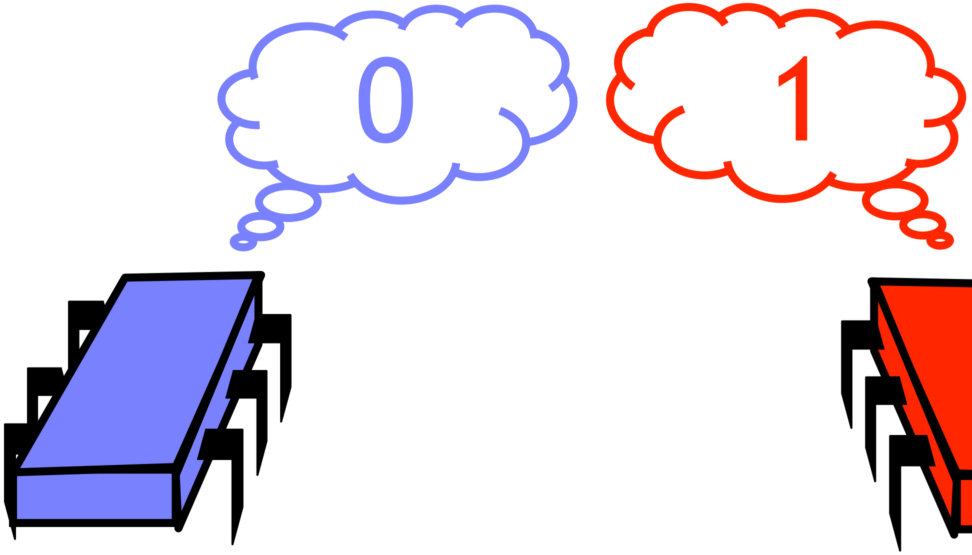
All executions must decide 1

Both Inputs 1



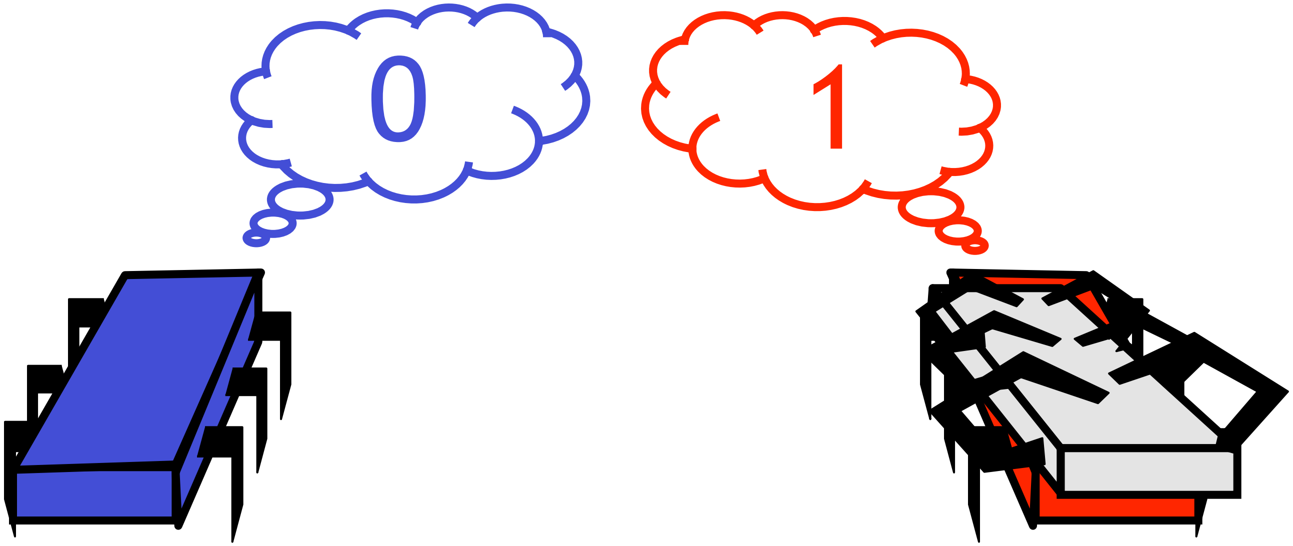
Including this solo execution by B

What if inputs differ?



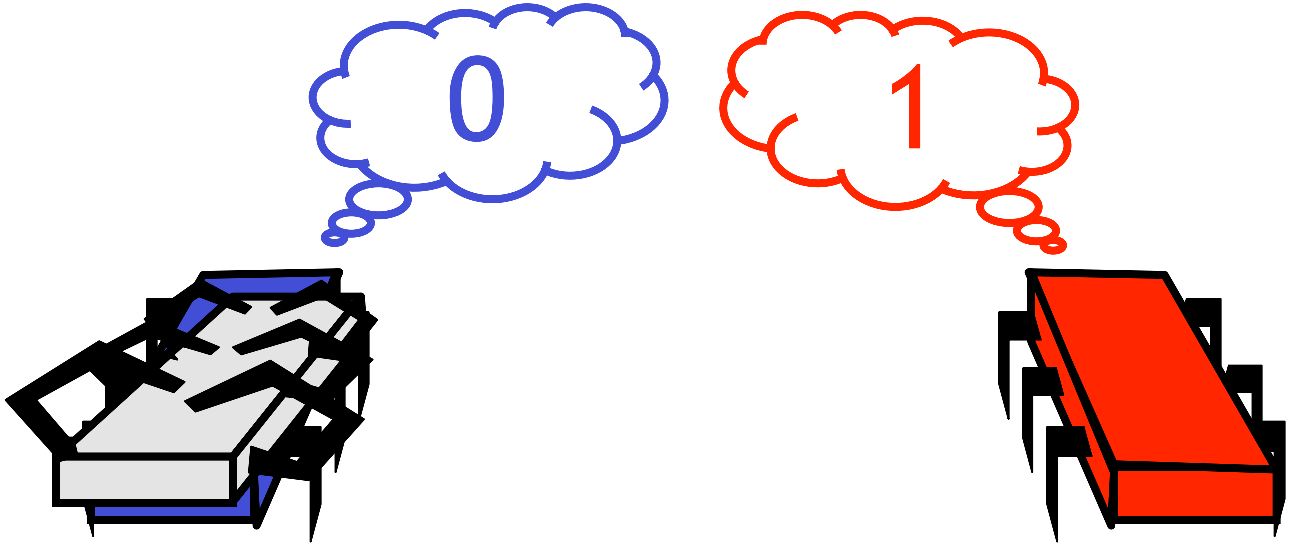
By Way of contradiction: If univalent all executions must decide on same value

The Possible Executions



Include the solo execution by A that decides 0

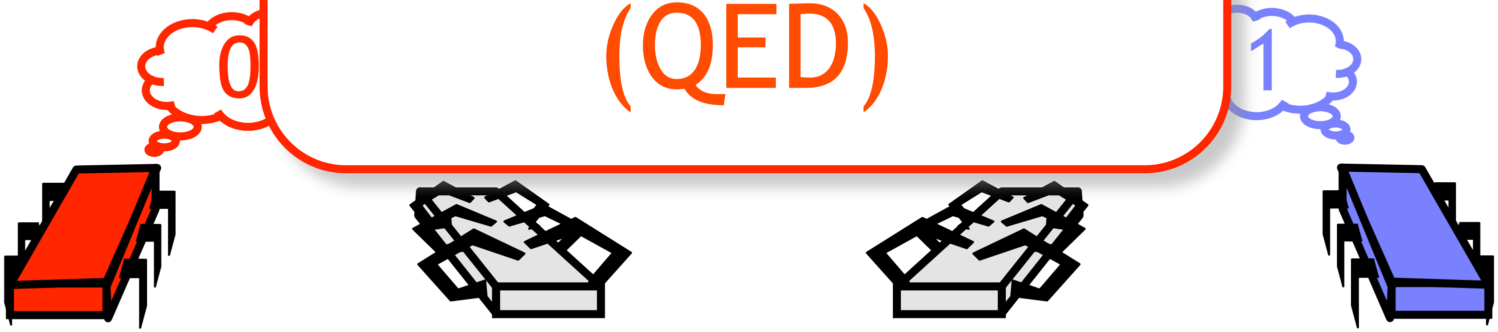
The Possible Executions



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

Possible Executions Include

How univalent is that?
(QED)

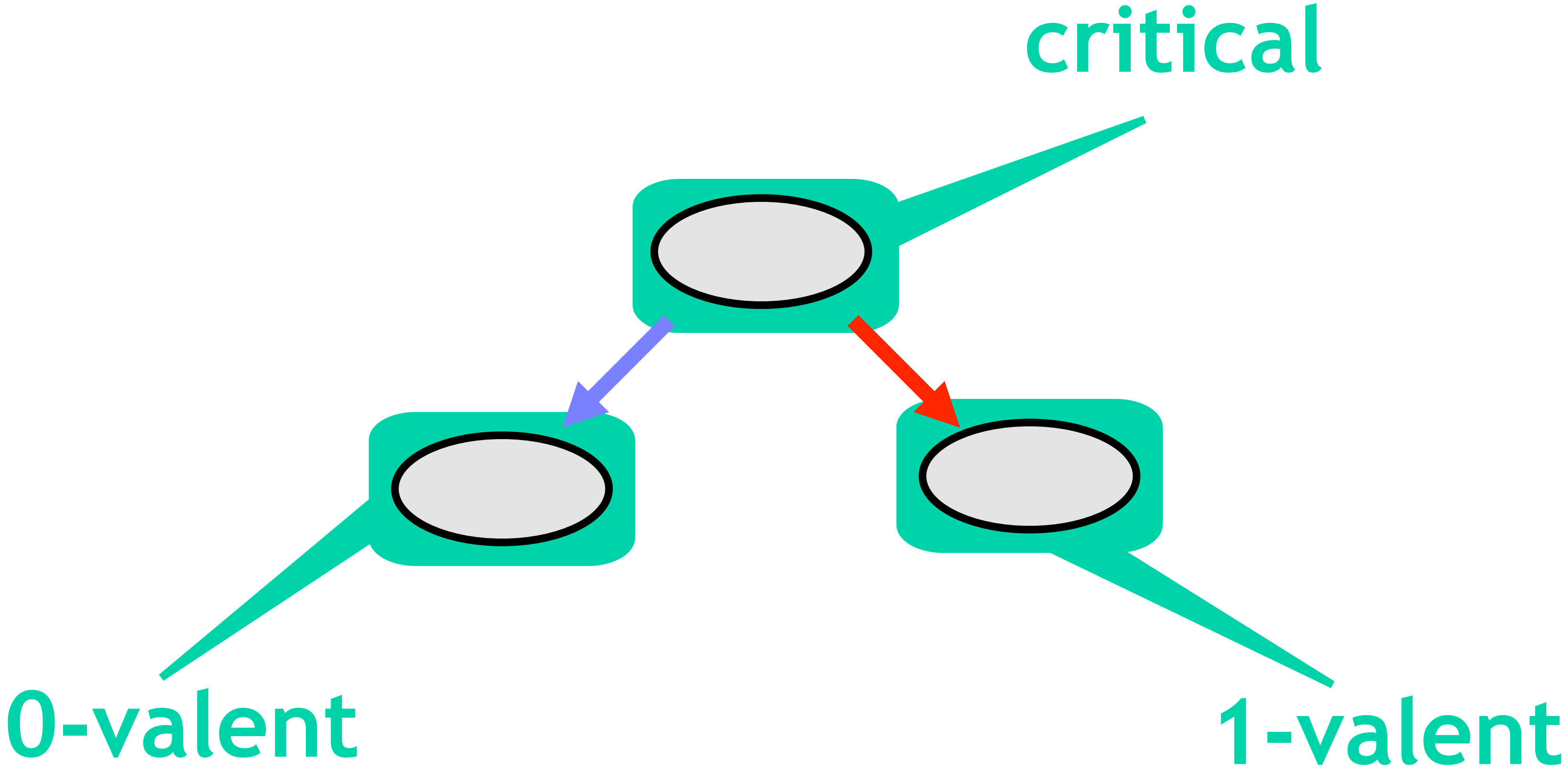


- Solo execution by A must decide 0
- Solo execution by B must decide 1

Summary So Far

- 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

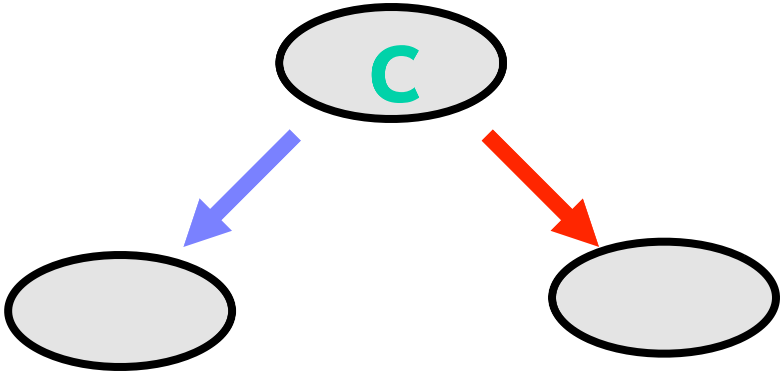
Critical States


critical

0-valent

1-valent

From a Critical State



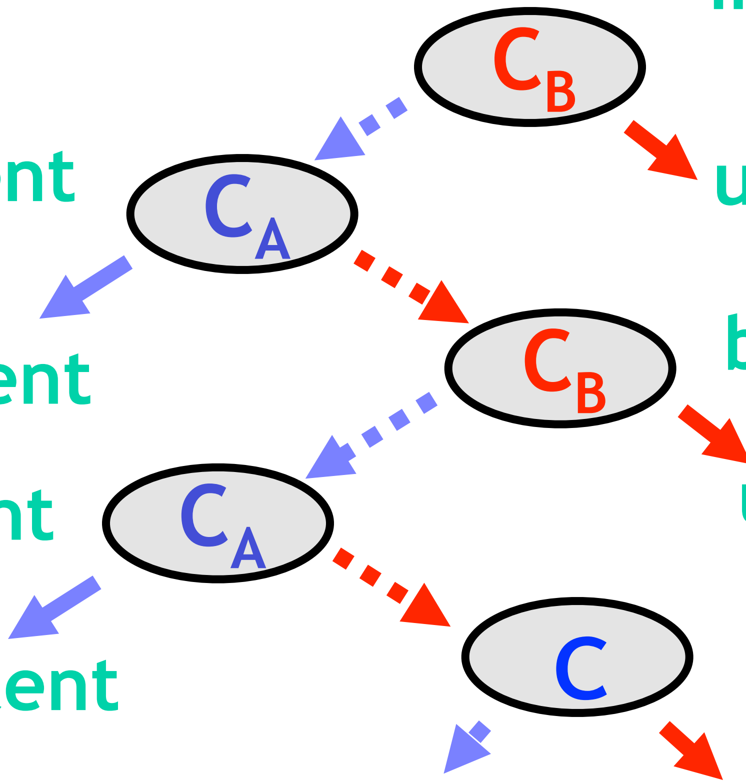
0-valent

1-valent

If A goes first, protocol decides 0

If B goes first, protocol decides 1

Reaching Critical State



initially bivalent

bivalent

univalent

univalent

bivalent

bivalent

univalent

univalent

0-valent

1-valent

© Herlihy and Shavit

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

What are the Threads Doing?

- 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

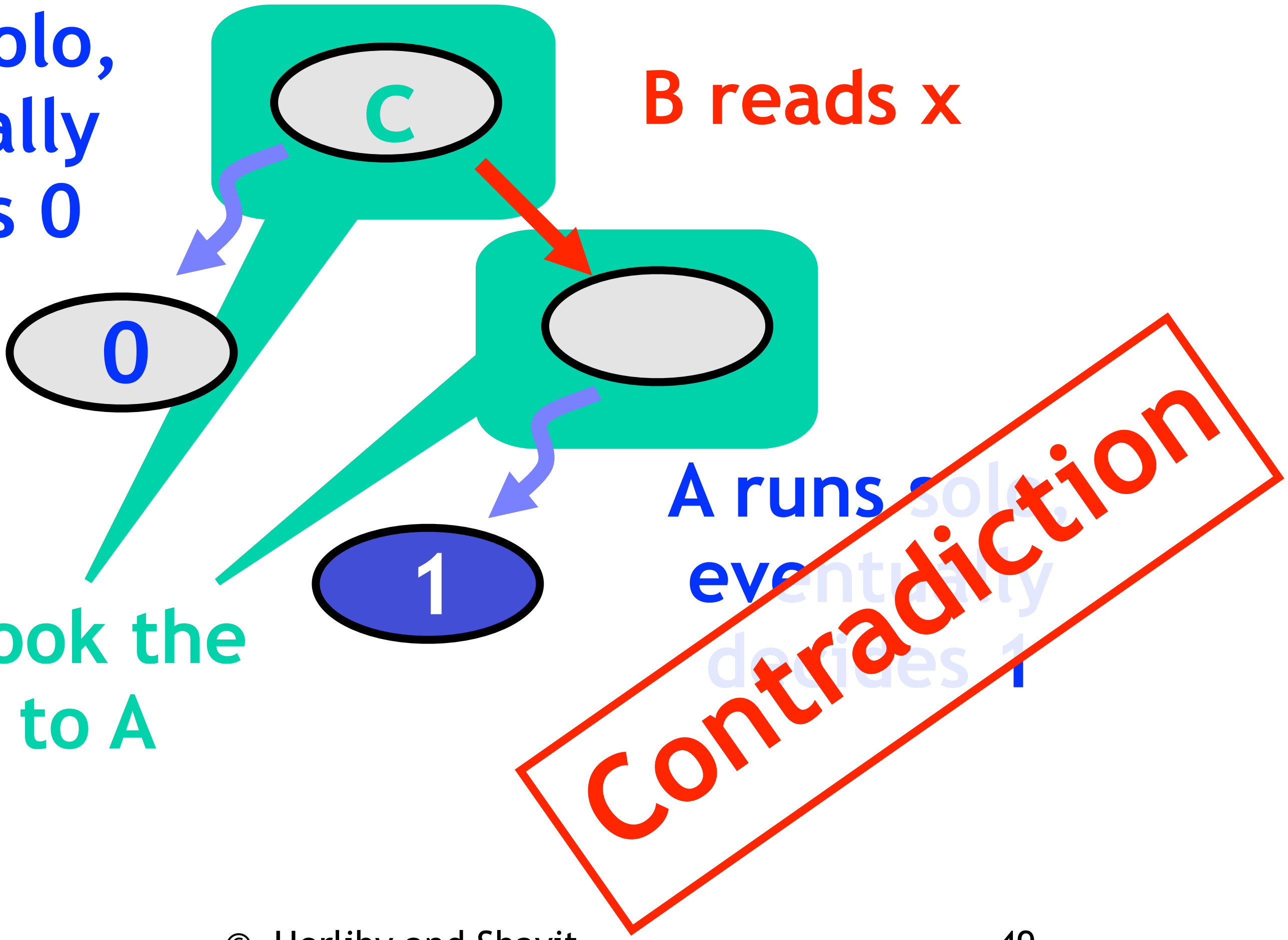
Possible Interactions

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



A runs solo, eventually decides 0

B reads x

States look the same to A

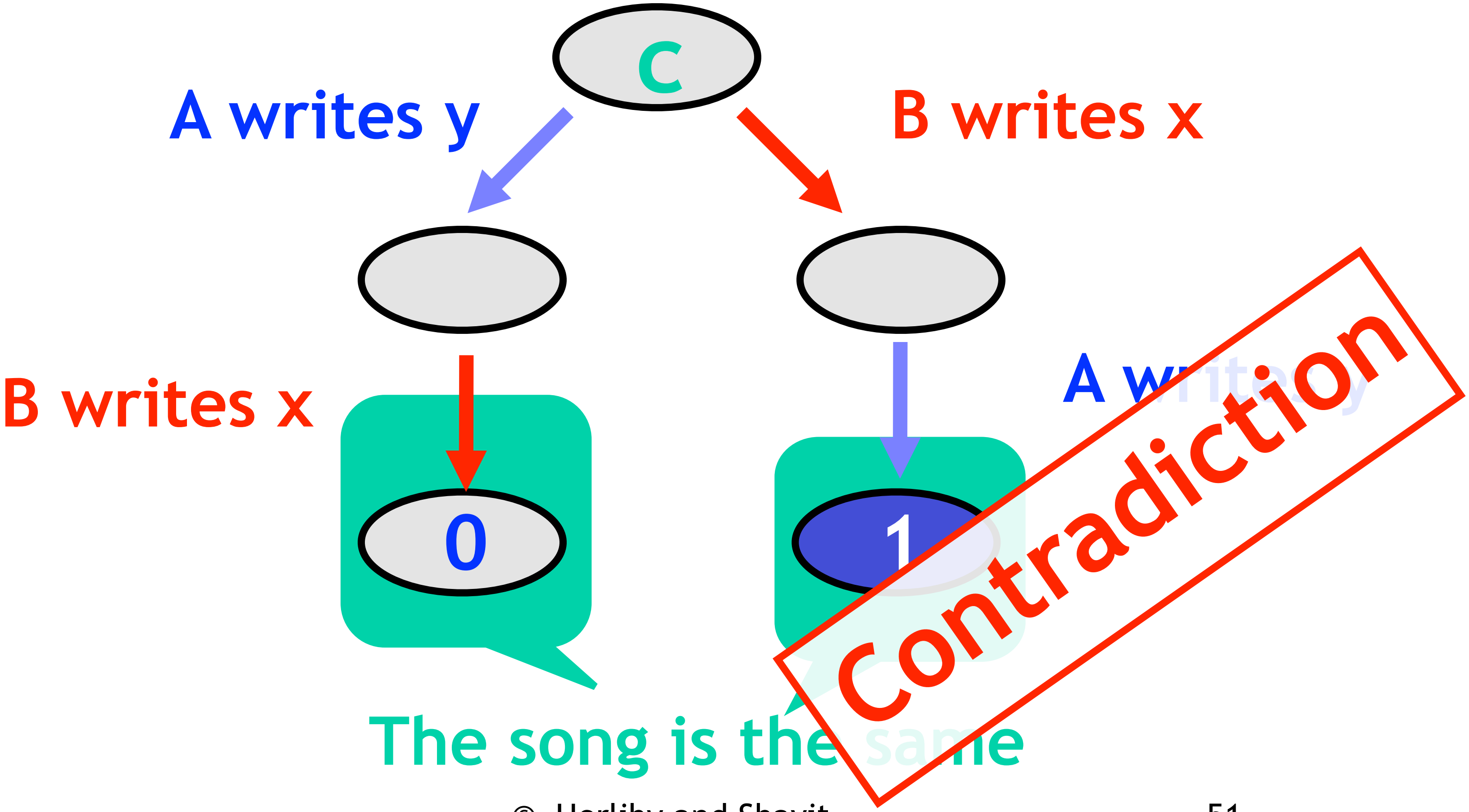
A runs solo, eventually decides 1

Contradiction

Possible Interactions

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

Writing Distinct Registers



C

A writes y

B writes x

B writes x

0

A writes y

1

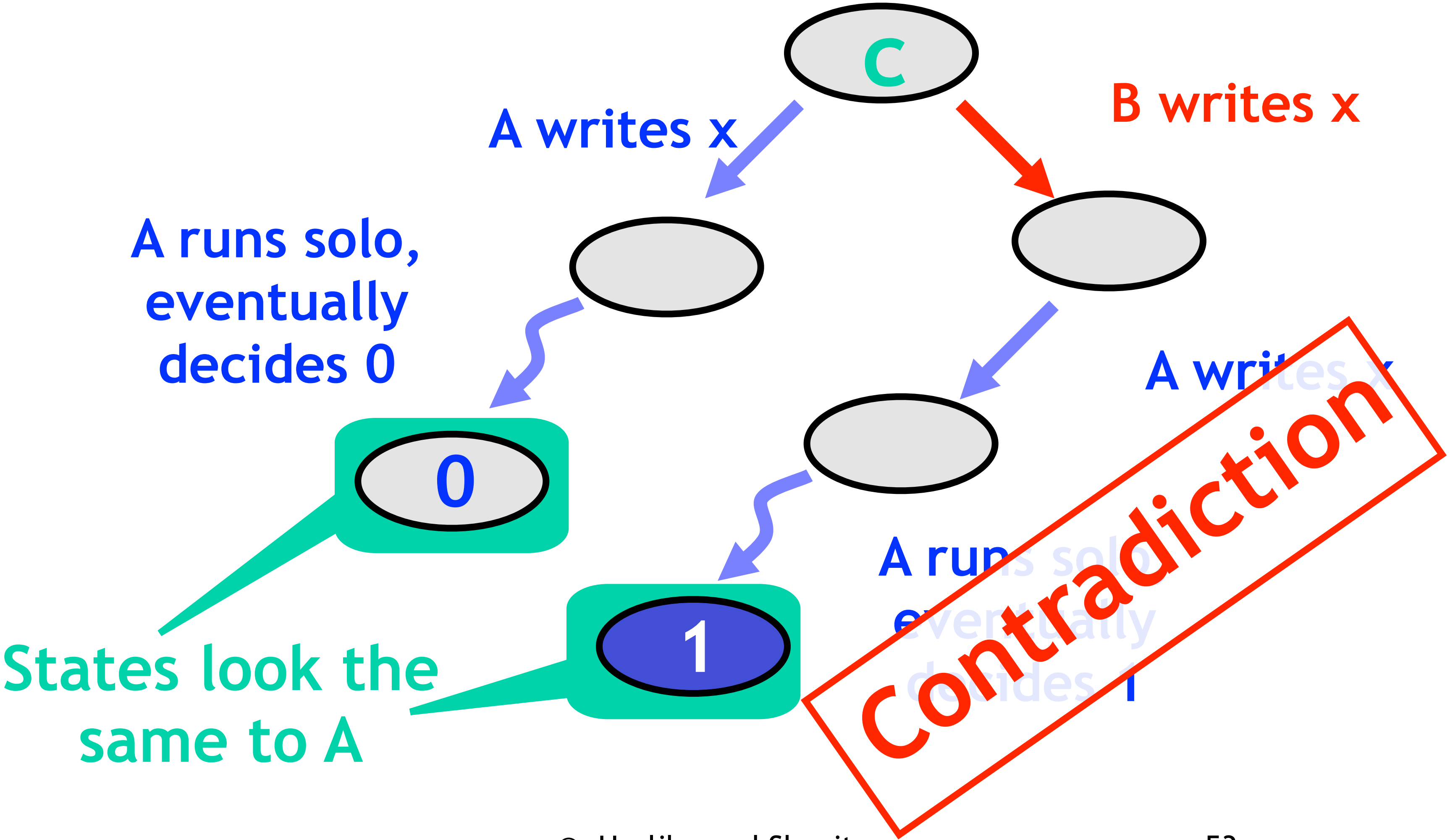
The song is the same

Contradiction

Possible Interactions

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

Writing Same Registers



B writes x

A writes x

A runs solo, eventually decides 0

A writes x

A runs solo, eventually decides 1

States look the same to A

Contradiction

That’s All, Folks!

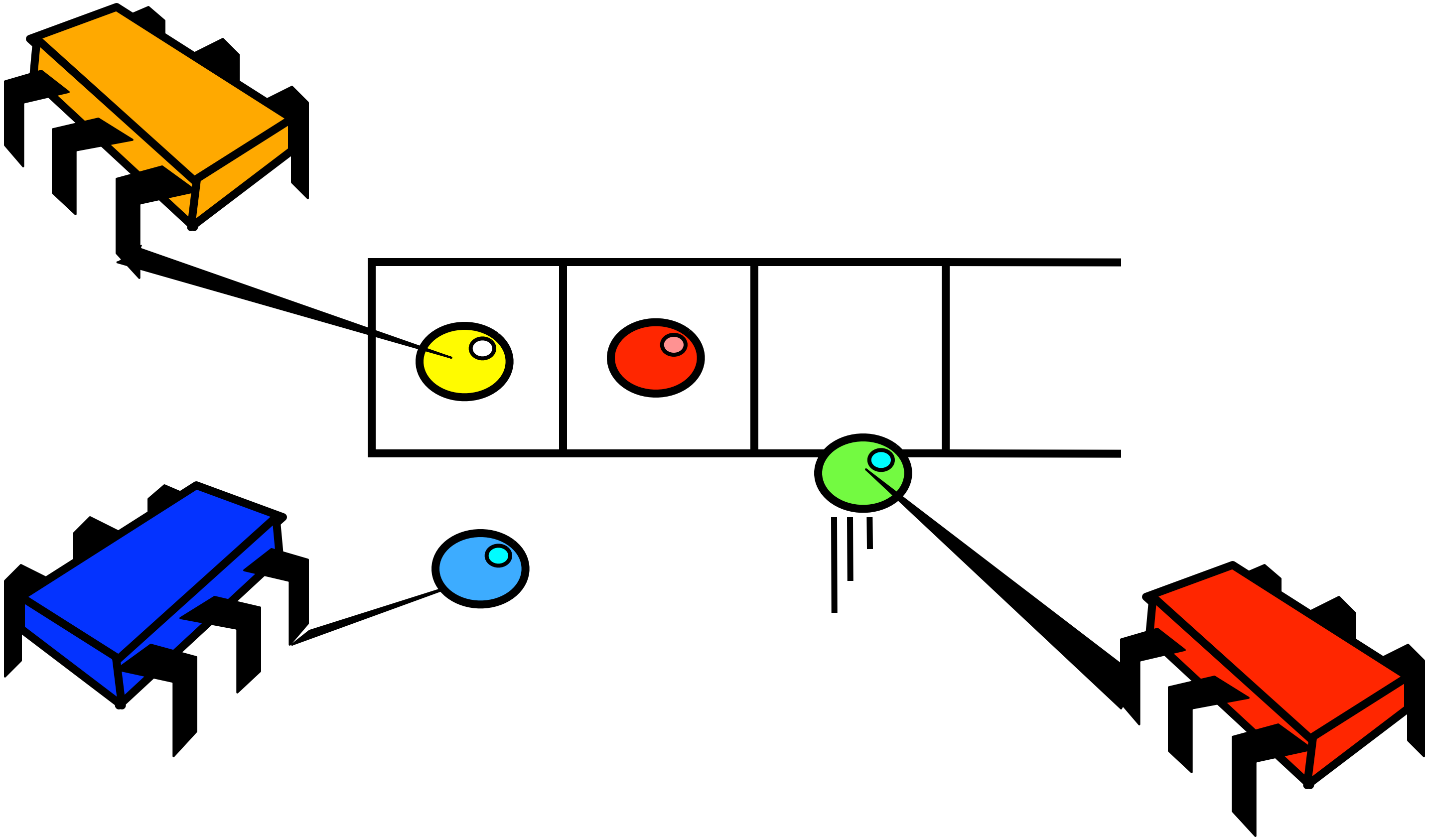
	`x.read()`	`y.read()`	`x.write()`	`y.write()`
`x.read()`	no	no	no	no
`y.read()`	no	no	no	no
`x.write()`	no	no	no	no
`y.write()`	no	no	no	no

QED

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 Obje
                 deci

 }}
```

Each thread's proposed value

Generic Consensus Protocol

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

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

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

Propose a value

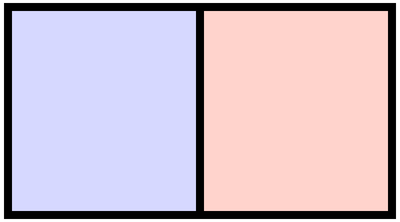
Generic Consensus Protocol

```
abstract class ConsensusProtocol

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

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

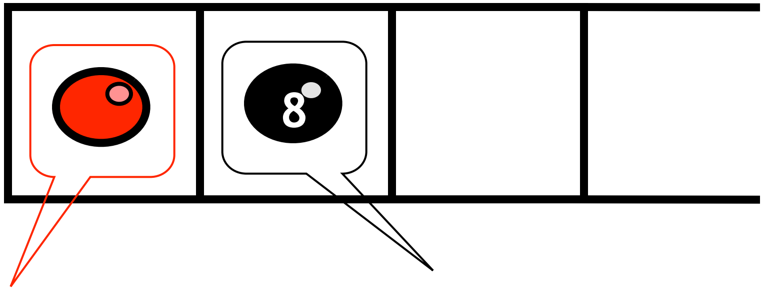
Decide a value: abstract method means subclass does the heavy lifting (real work)

Can FIFO Queue Implement Consensus?

FIFO Consensus



`propose` **array**

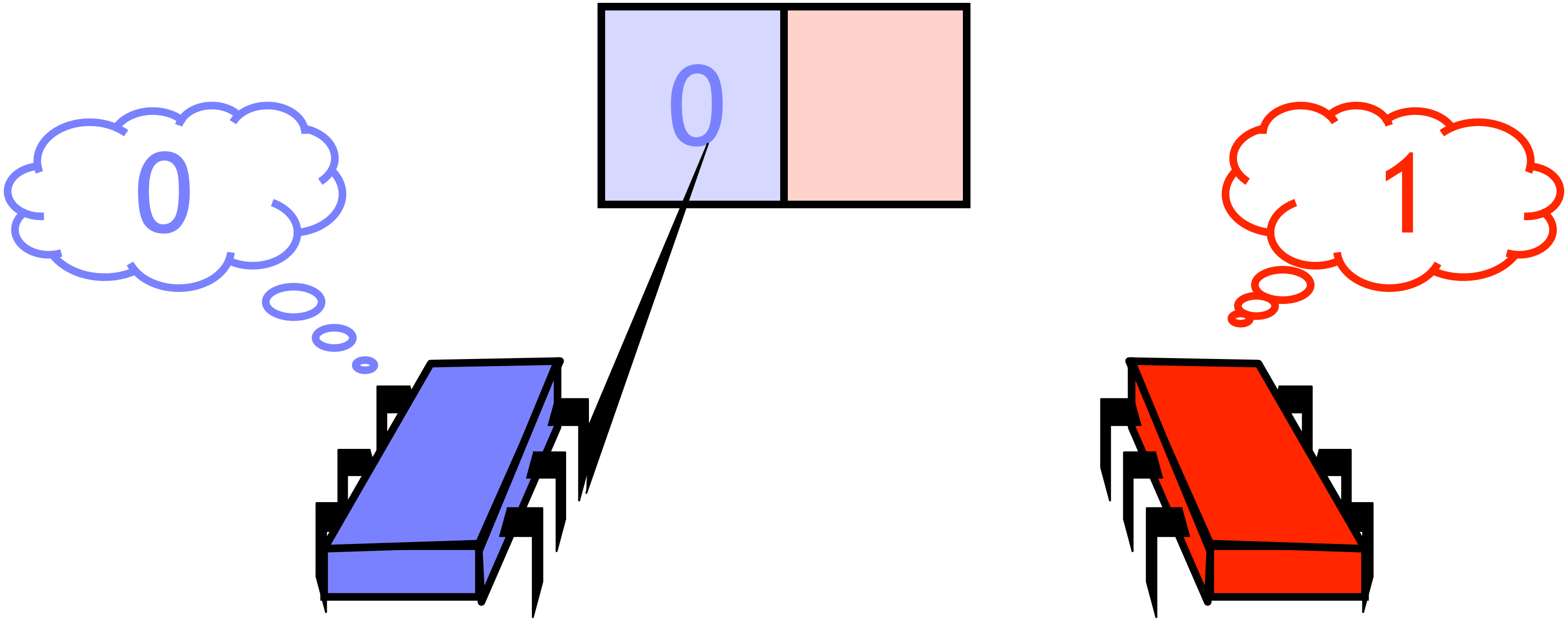


`FIFO Queue` **with red and black balls**

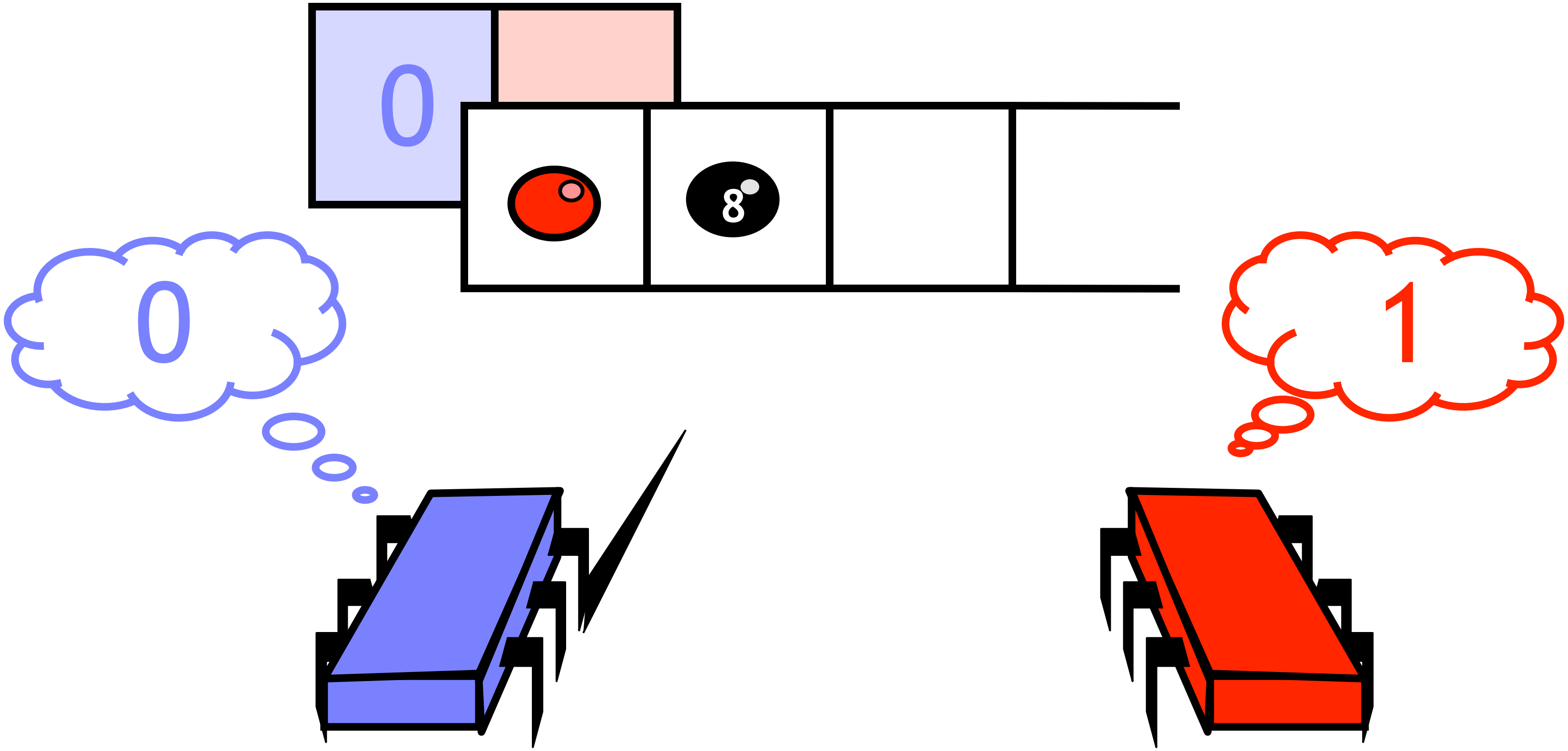
Coveted red ball

Dreaded black ball

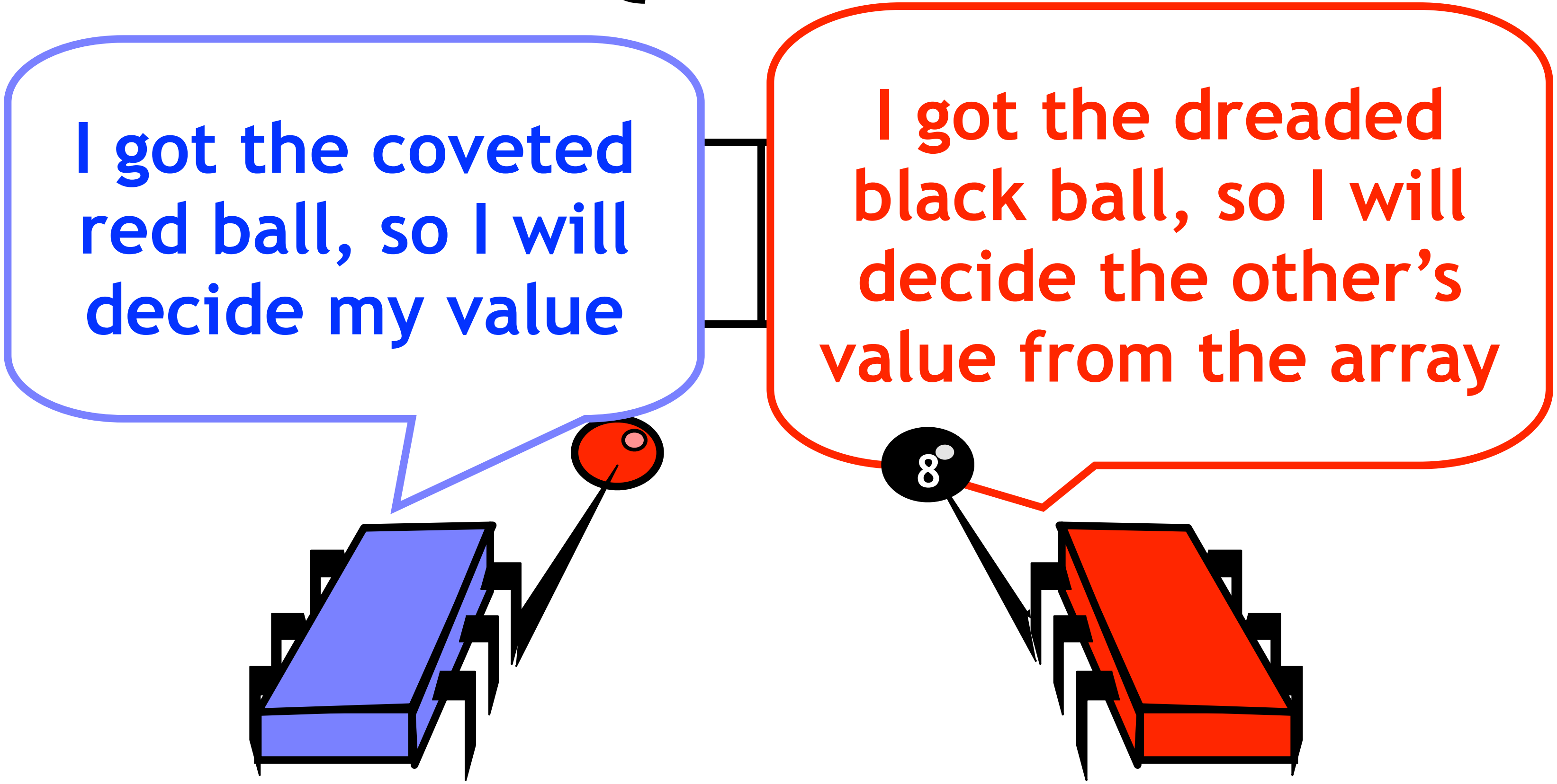
Protocol: Write Value to Array



Protocol: Take Next Item from Queue



Protocol: Take Next Item from Queue



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

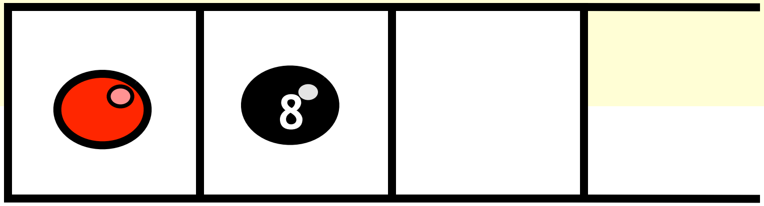
I got the dreaded black ball, so I will decide the other's value from the array

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() {
  queue = new Queue();
  queue.enq(Ball.RED);
  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-ij];
 }
}
```

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

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 to array
  - Before dequeueing 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
    - Taking 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 c
  - 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

Homework

- What is the consensus number of a wait-free FIFO queue with methods:
  - enq(o): enqueue object o
  - deq(): dequeue first object
  - peek(): get a copy of first object

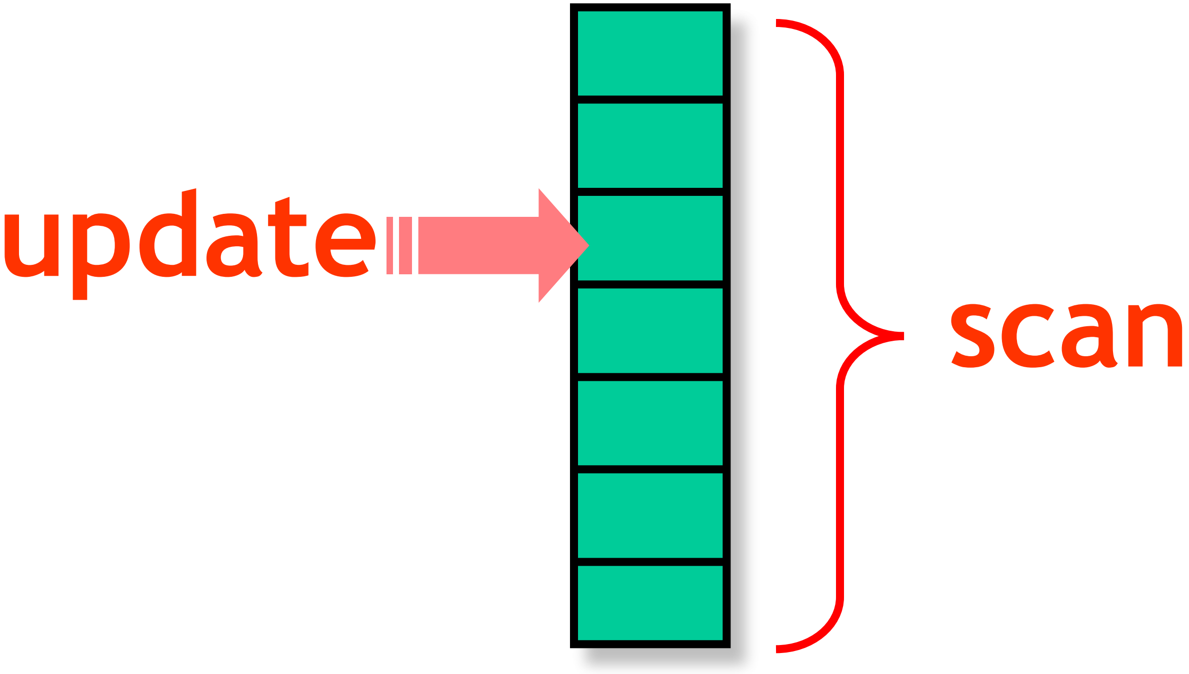
New Grand Challenge

- Consider:
  - 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/multiple location read OK (= Atomic Snapshot)
  - Multi location write/single location read impossible

Atomic Snapshot



Atomic Snapshot

- Array of MRSW atomic registers
- Take instantaneous snapshot of all
- Generalizes to MRMW registers …

Snapshot Interface

```
public interface Snapshot {
  public int update(int v);
  public int[] scan();
}
```

Snapshot Interface

Thread i writes v to its register

```
public interface Snapshot {
  public int update(int v);
  public int[] scan();
}
```

Snapshot Interface

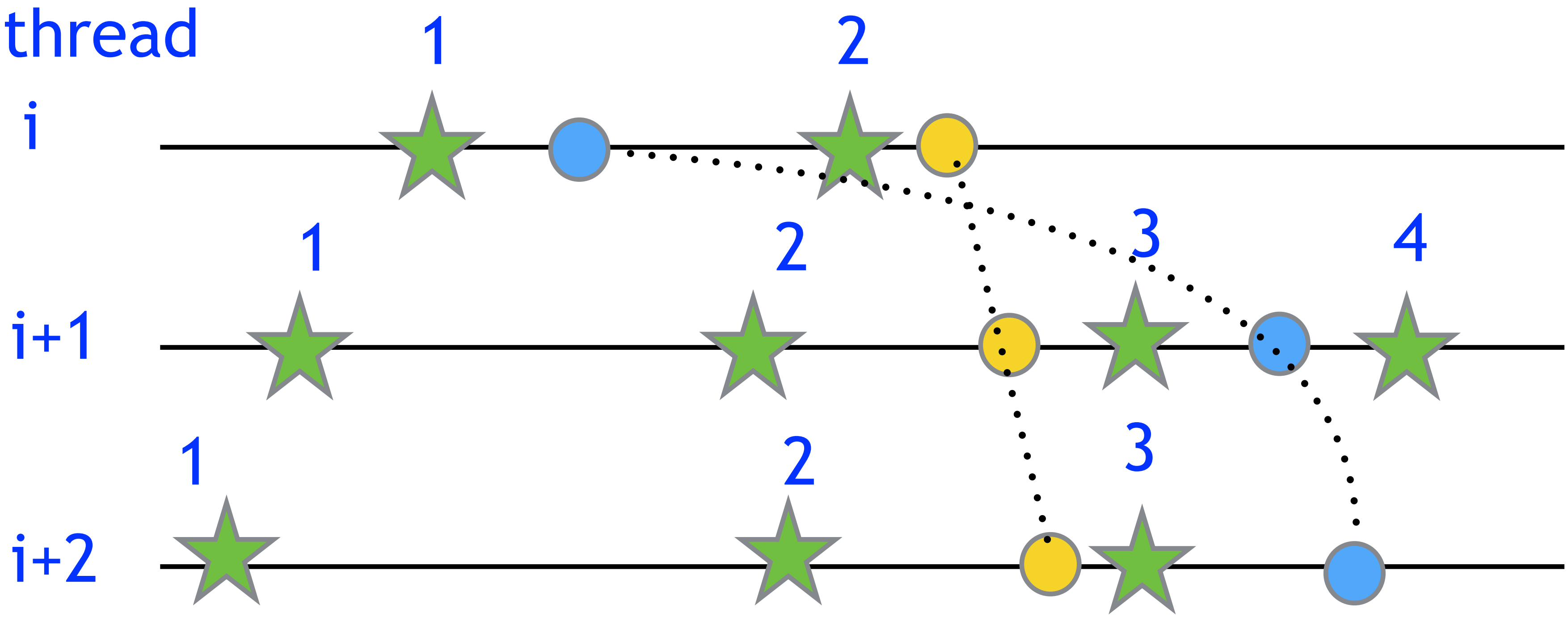
Instantaneous snapshot of all theads’ registers

```
public interface Snapshot {
  public int update(int v);
  public int[] scan();
}
```

Atomic Snapshot

- Collect
  - Read values one at a time
- Problem
  - Incompatible concurrent collects
  - Result not linearizable

Example: Atomic Snapshot MRMW



○ read

☆ update

A: 1, 3, 3

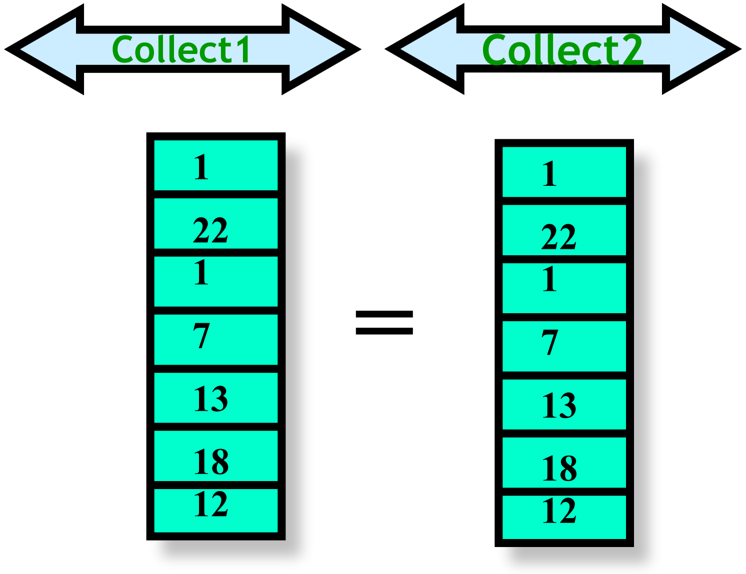
B: 2, 2, 2

Clean Collects

- Clean Collect
  - Collect during which nothing changed
  - Can we make it happen?
  - Can we detect it?

Simple Snapshot

- Put increasing labels on each entry
- Collect twice
- If both agree,
  - We’re done
- Otherwise,
  - Try again



Simple Snapshot: Update

```
public class SimpleSnapshot implements Snapshot {
  private AtomicMRSWRegister[] register;

  public void update(int value) {
    int i = Thread.myIndex();
       LabeledValue oldValue = register[i].read();
    LabeledValue newValue =
     new LabeledValue(oldValue.label+1, value);
    register[i].write(newValue);
  }
```

Simple Snapshot: Update

```
public class SimpleSnapshot implements Snapshot {
  private AtomicMRSWRegister[] register;

  public void update(int value) {
    int i = Thread.myIndex();
    LabeledValue oldValue = register[i].read();
    LabeledValue newValue =
     new LabeledValue(oldValue.label+1, value);
    register[i].write(newValue);
  }
```

One single-writer register per thread

Simple Snapshot: Update

```
public class SimpleSnapshot implements Snapshot {
  private AtomicMRSWRegister[] register;

  public void update(int value) {
    int i = Thread.myIndex();
    LabeledValue oldValue = register[i].read();
    LabeledValue newValue =
     new LabeledValue(oldValue.label+1, value);
    register[i].write(newValue);
  }
```

Write each time with higher label

Simple Snapshot: Collect

```
private LabeledValue[] collect() {
 LabeledValue[] copy =
  new LabeledValue[n];
 for (int j = 0; j < n; j++)
  copy[j] = this.register[j].read();
 return copy;
}
```

Simple Snapshot

```
private LabeledValue[] collect() {
 LabeledValue[] copy =
  new LabeledValue[n];
 for (int j = 0; j < n; j++)
  copy[j] = this.register[j].read();
 return copy;
}
```

Just read each register into array

Simple Snapshot: Scan

```
public int[] scan() {
  LabeledValue[] oldCopy, newCopy;
  oldCopy = collect();
  collect: while (true) {
    newCopy = collect();
    if (!equals(oldCopy, newCopy)) {
      oldCopy = newCopy;
      continue collect;
     }
    return getValues(newCopy);
  }
}
```

Simple Snapshot: Scan

```
public int[] scan() {
  LabeledValue[] oldCopy, newCopy;
  oldCopy = collect();
  collect: while (true) {
    newCopy = collect();
    if (!equals(oldCopy, newCopy)) {
      oldCopy = newCopy;
      continue collect;
    }
    return getValues(newCopy);
  }
}
```

Collect once

Simple Snapshot: Scan

```
public int[] scan() {
  LabeledValue[] oldCopy, newCopy;
  oldCopy = collect();
  collect: while (true) {
    newCopy = collect();
    if (!equals(oldCopy, newCopy)) {
      oldCopy = newCopy;
      continue collect;
    }
    return getValues(newCopy);
  }
}
```

Collect once

Collect twice

Simple Snapshot: Scan

```
public int[] scan() {
  LabeledValue[] oldCopy, newCopy;
  oldCopy = collect();
  collect: while (true) {
    newCopy = collect();
    if (!equals(oldCopy, newCopy)) {
      oldCopy = newCopy;
      continue collect;
    }
    return getValues(newCopy);
  }
}
```

Collect once

Collect twice

On mismatch, try again

(1)

Simple Snapshot: Scan

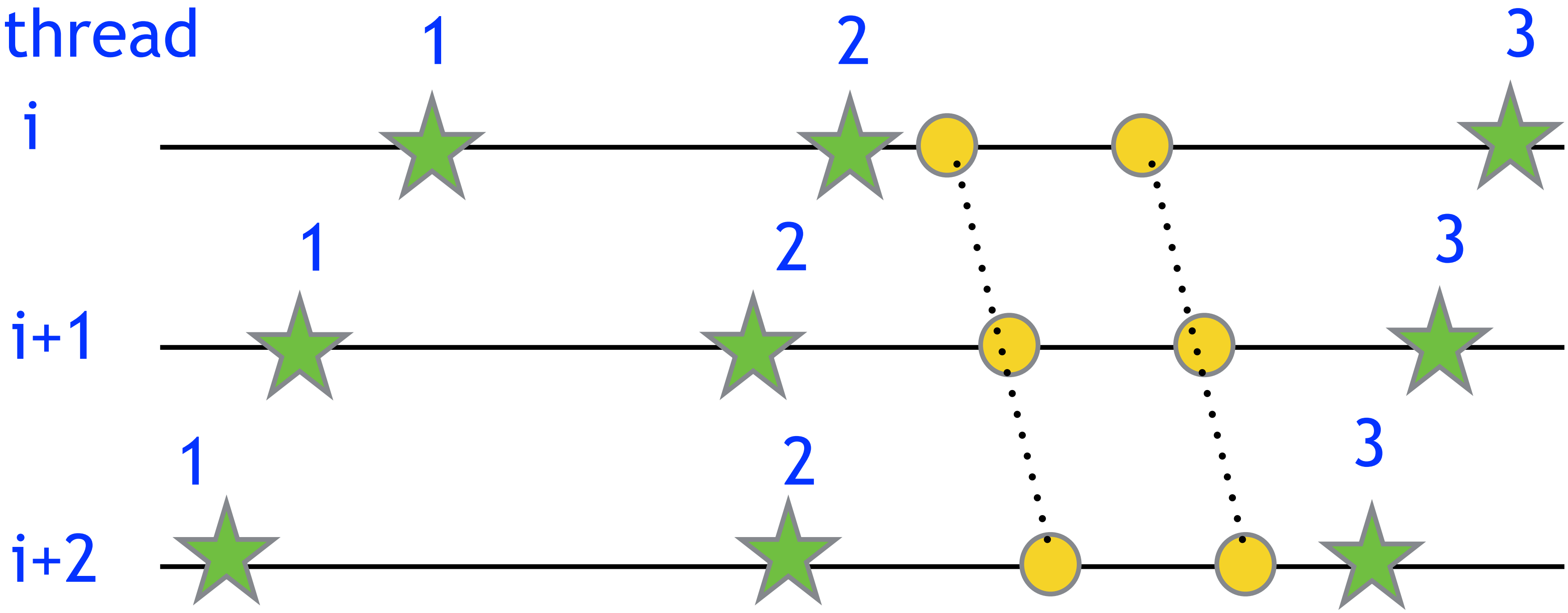
```
public int[] scan() {
  LabeledValue[] oldCopy, newCopy;
  oldCopy = collect();
  collect: while (true) {
    newCopy = collect();
    if (!equals(oldCopy, newCopy)) {
      oldCopy = newCopy;
      continue collect;
    }
    return getValues(newCopy);
  }
}
```

Collect once

Collect twice

On match, return values

Example



○ read

☆ update

B: 2, 2, 2

B: 2, 2, 2

Simple Snapshot

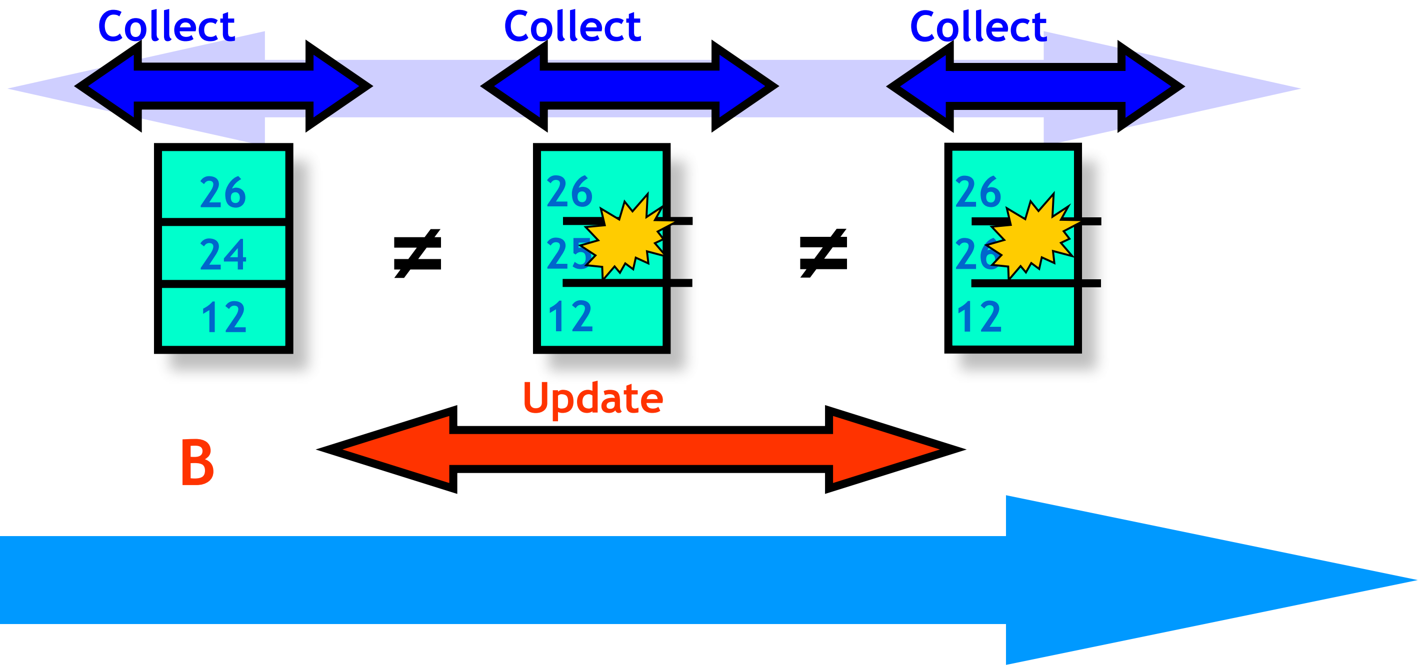
- Linearizable
- Update is wait-free
  - No unbounded loops
- But Scan can starve
  - If interrupted by concurrent update

Wait-Free Snapshot

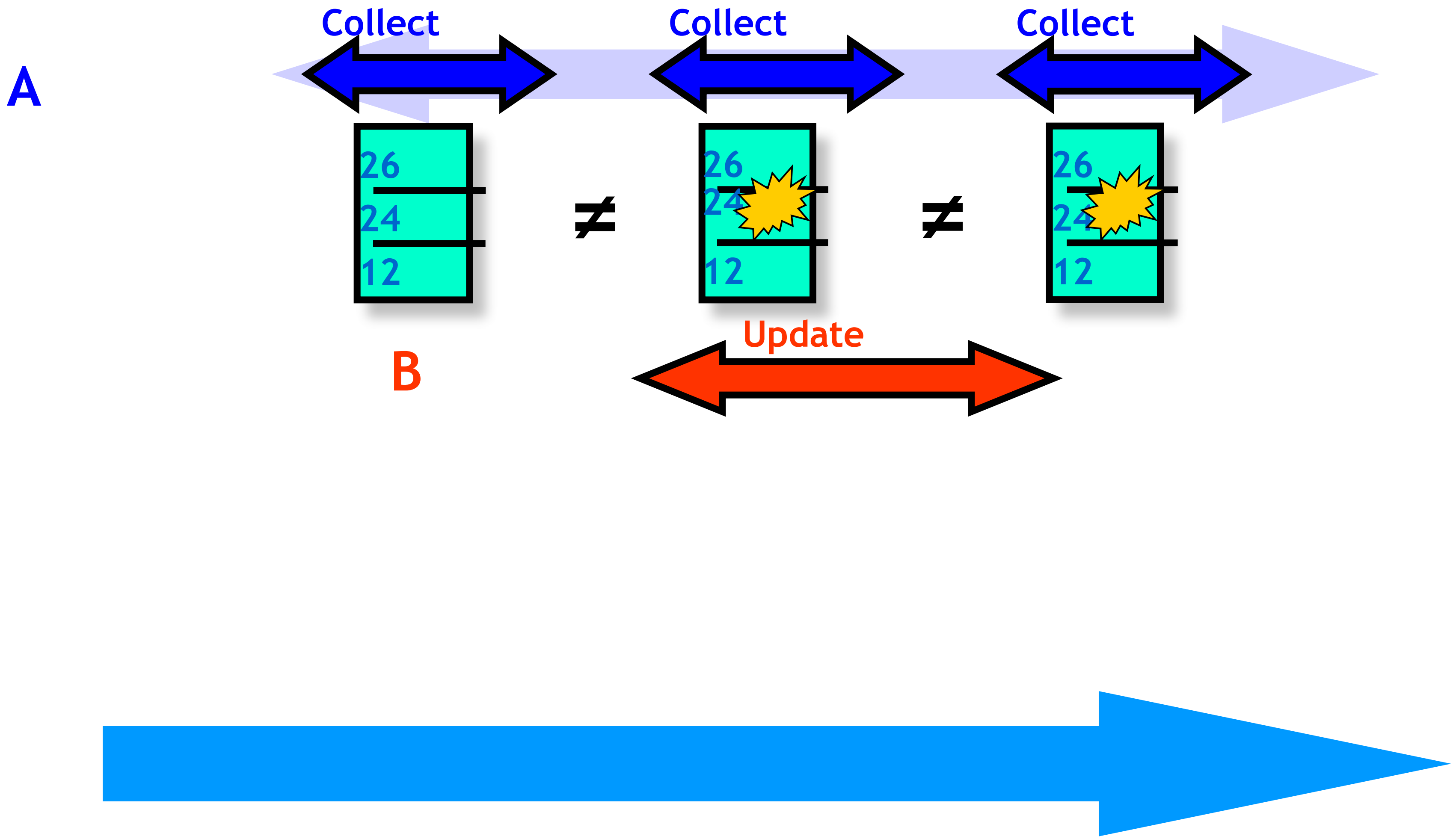
- Add a scan before every update
- Write resulting snapshot together with update value
- If scan is continuously interrupted by updates, scan can take the update's snapshot

Wait-free Snapshot

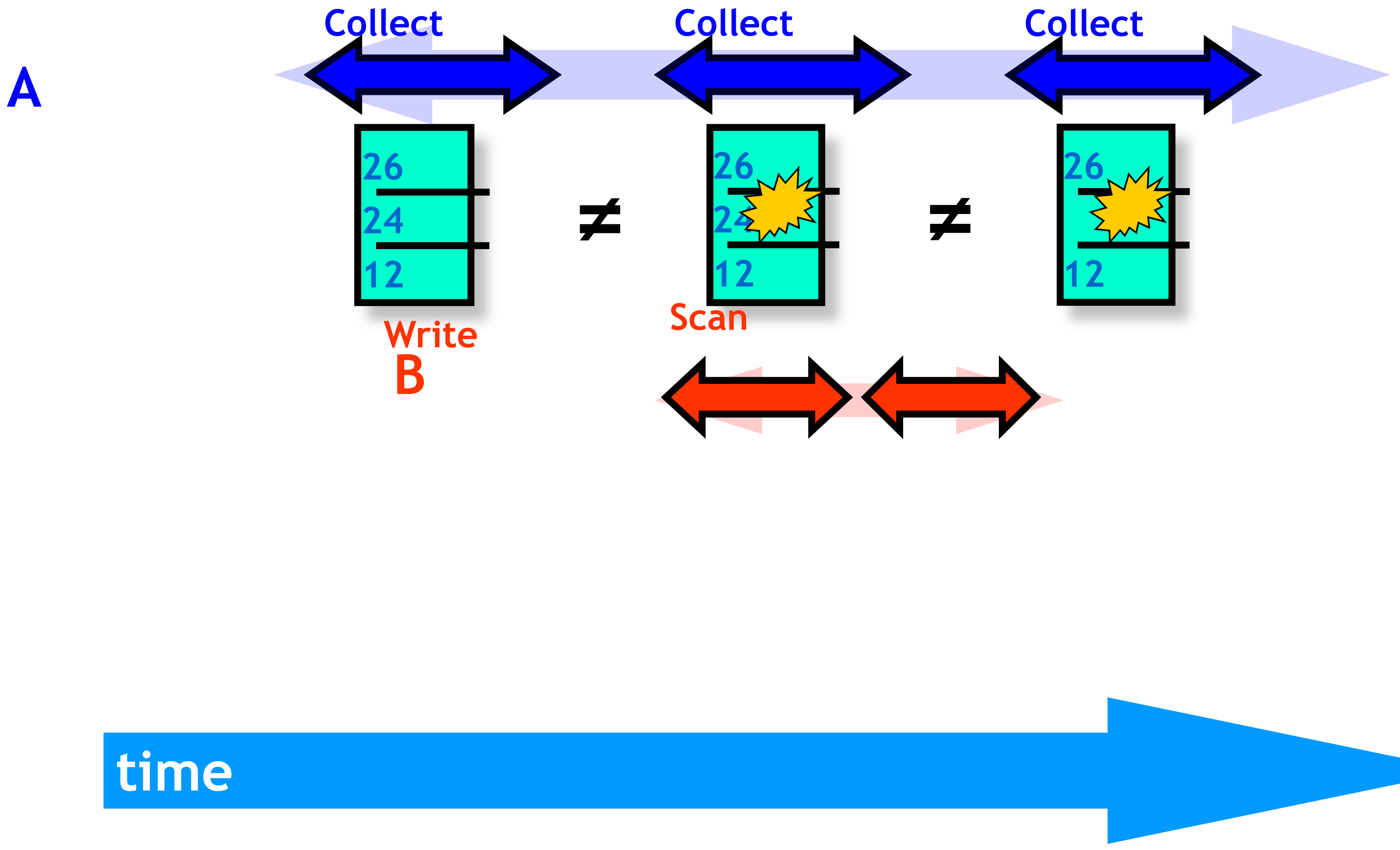
If A's scan observes that B moved <u>twice</u>, then B completed an update while A's scan was in progress



Wait-free Snapshot

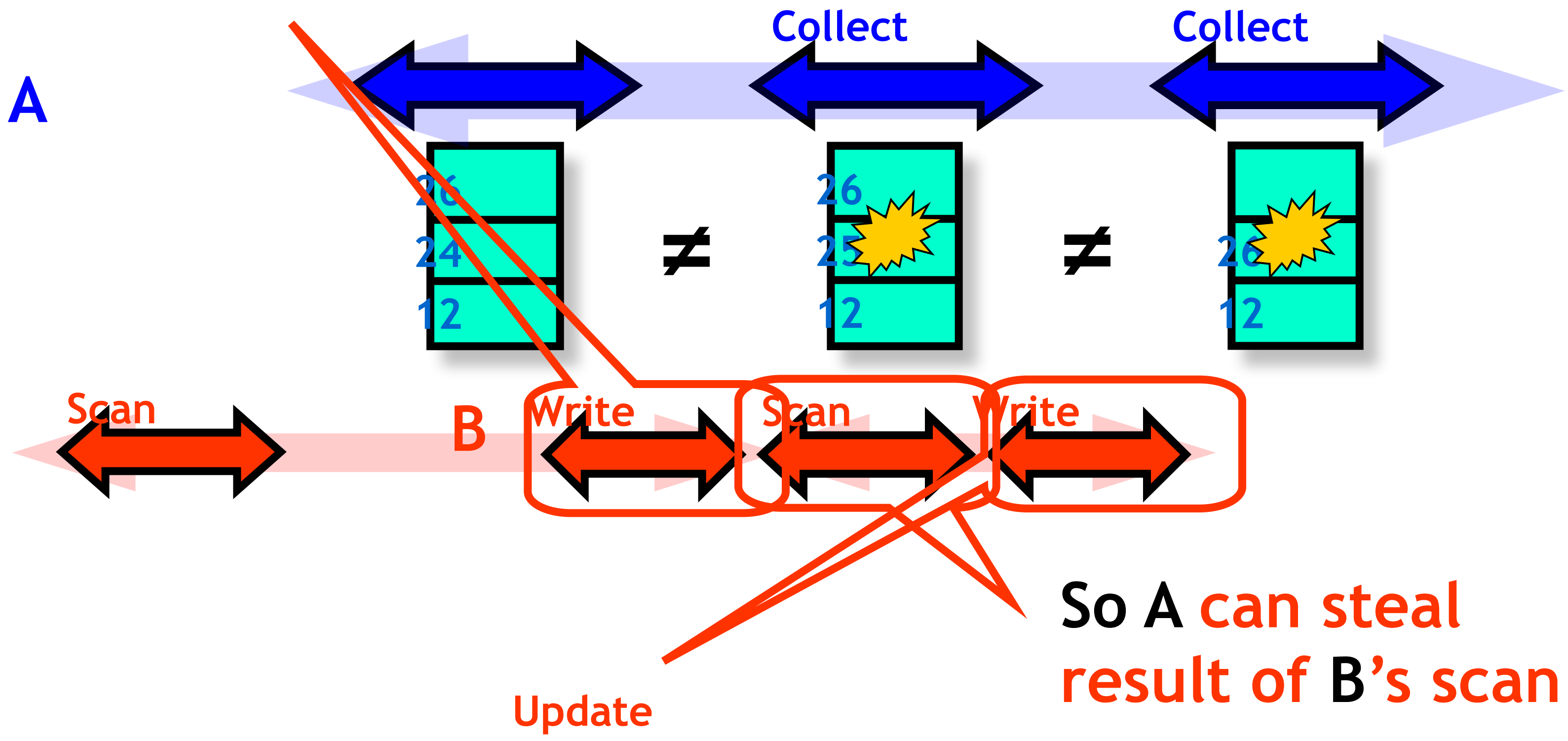


Wait-free Snapshot



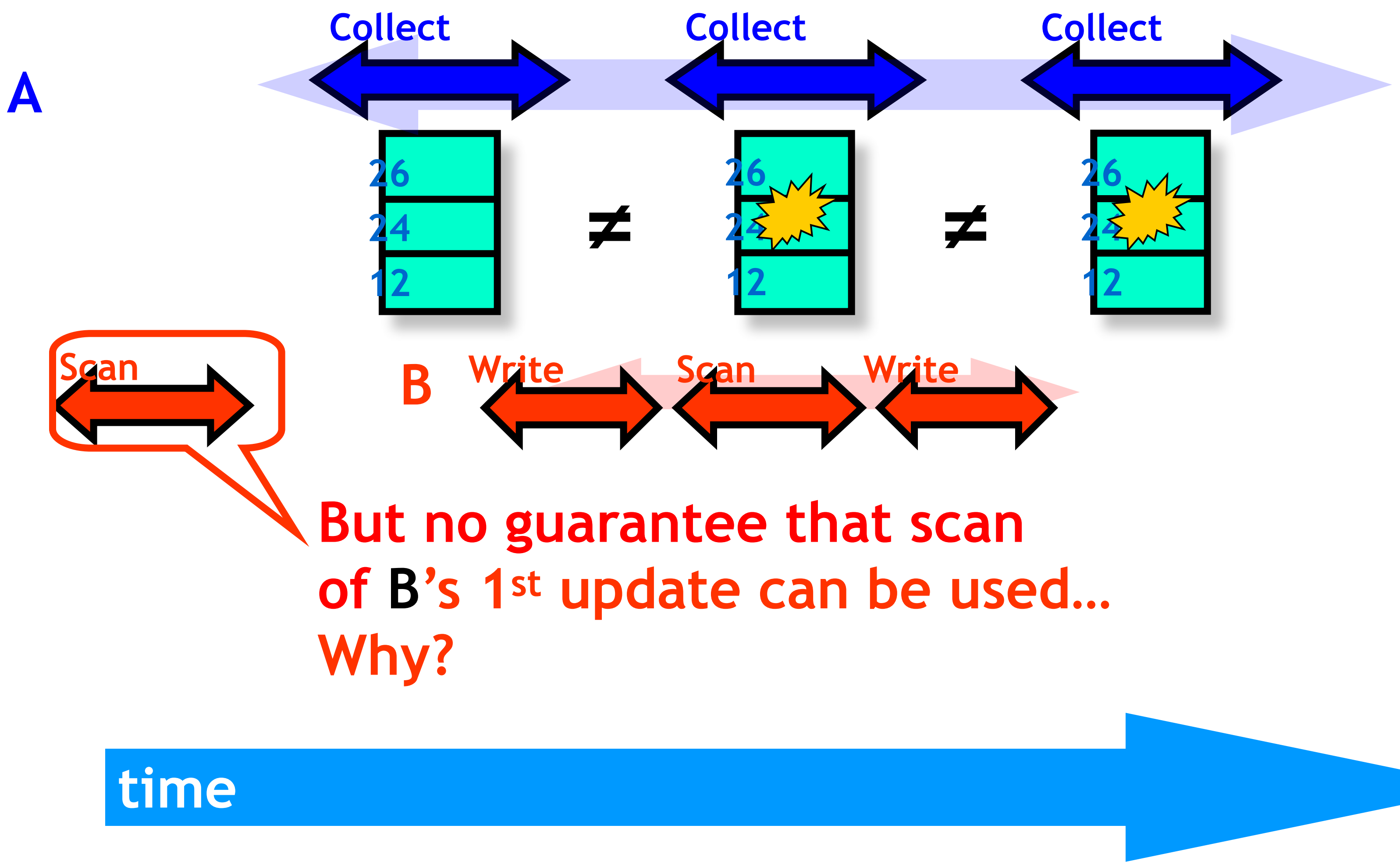
Wait-free Snapshot

B’s 1st update must have written during 1st collect



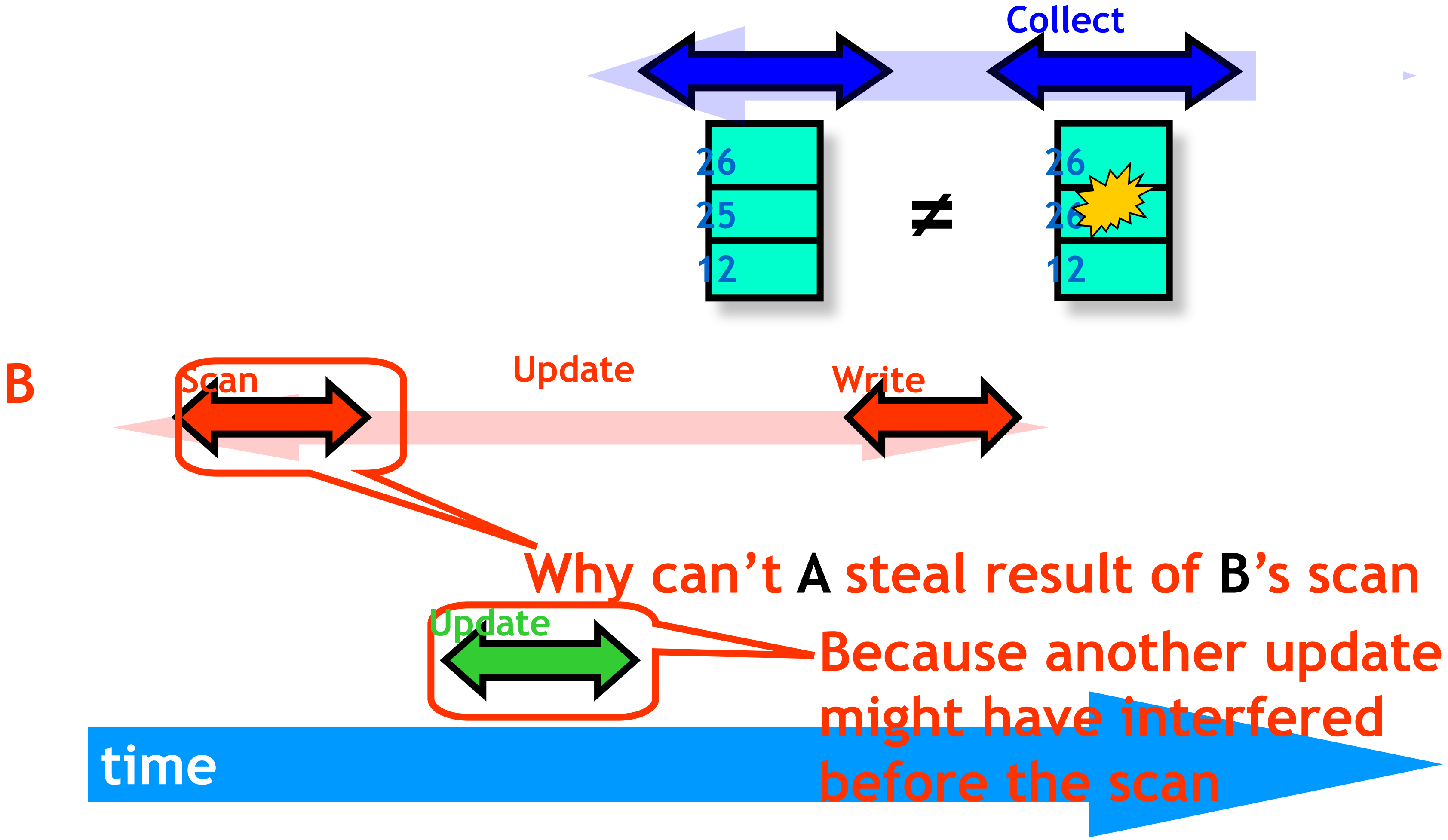
So A can steal result of B’s scan

Wait-free Snapshot



But no guarantee that scan of B’s 1st update can be used… Why?

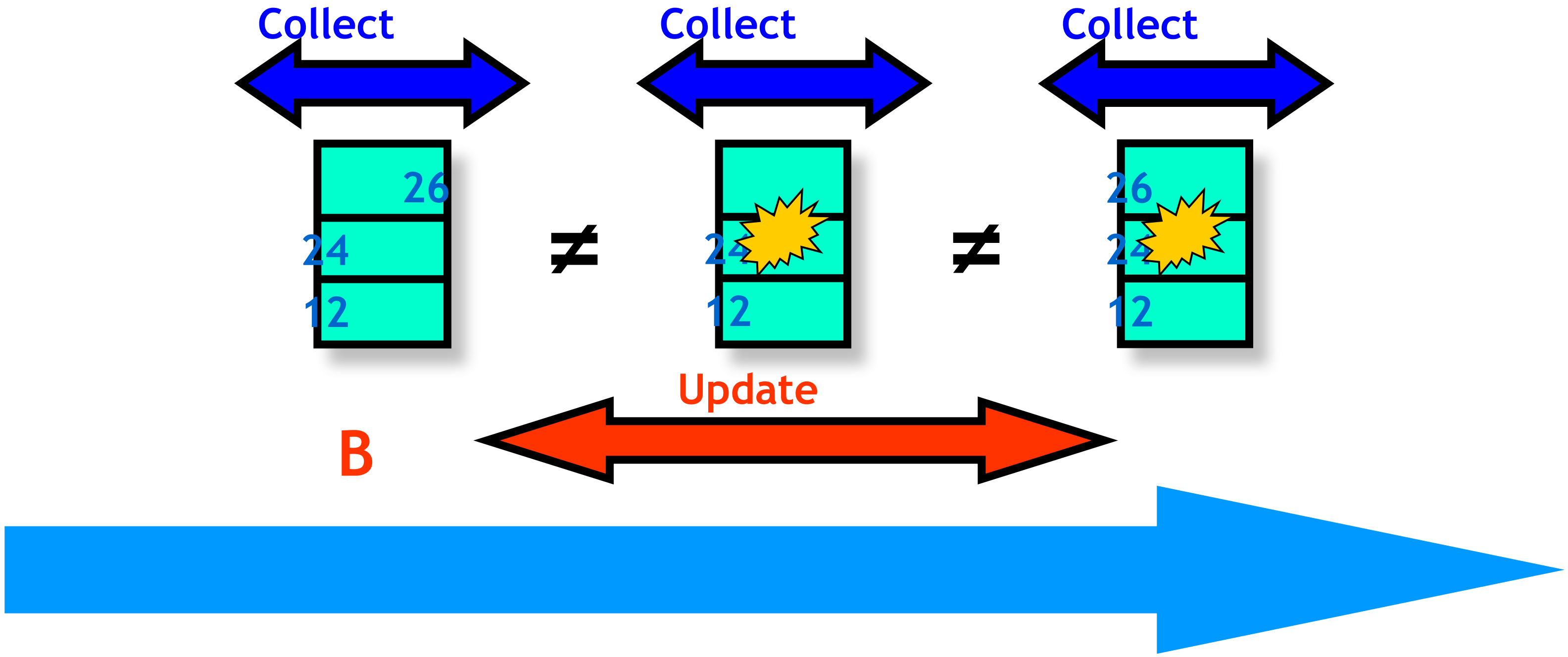
Once is not Enough



Why can't **A** steal result of **B**'s scan

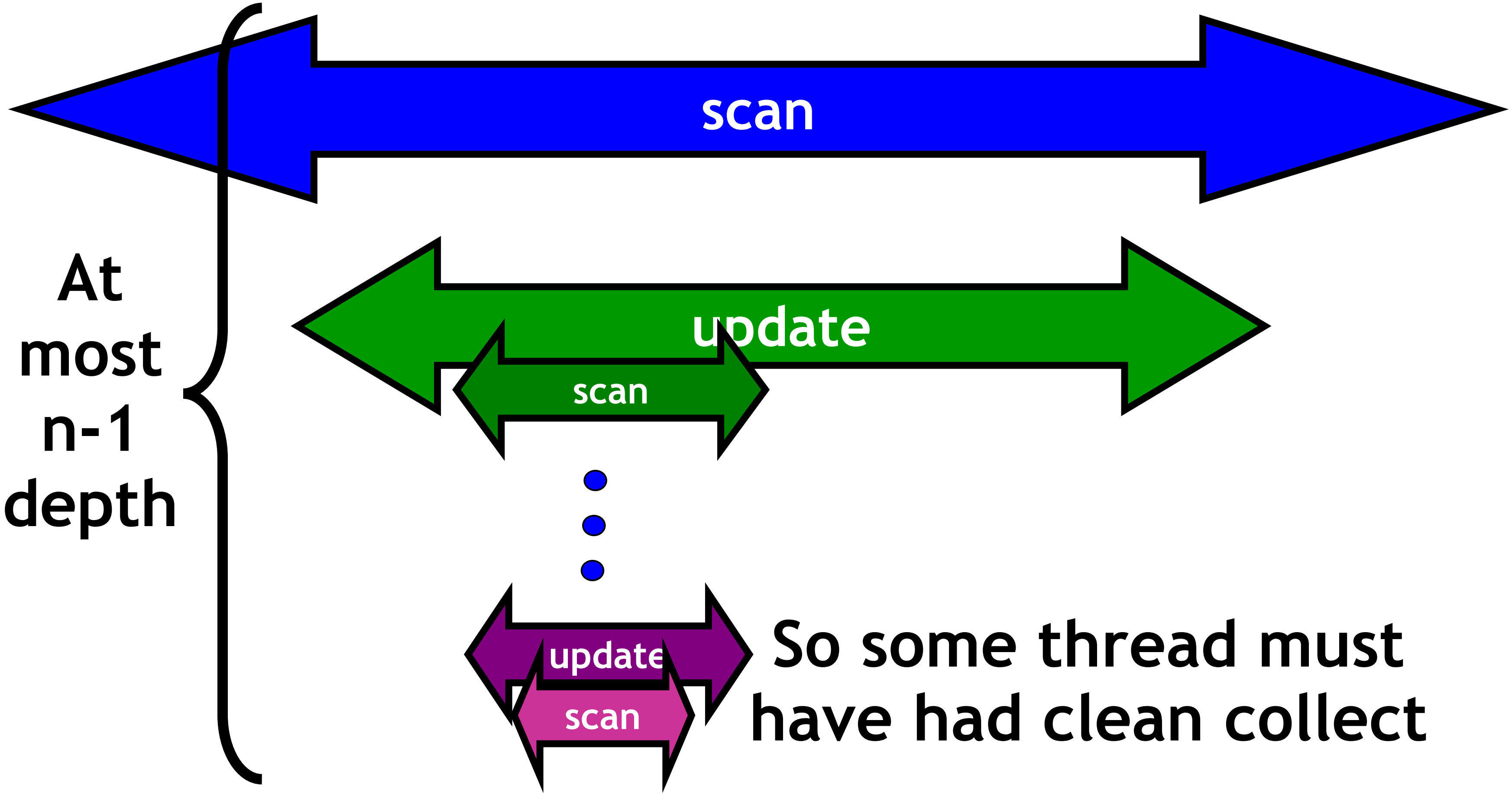
Because another update might have interfered before the scan

Someone Must Move Twice



If we collect n times…some thread
Must move twice (Pigeon hole)

Scan is Wait-free



So some thread must have had clean collect

Wait-Free Snapshot Label

```
public class SnapValue {
 public int   label;
 public int   value;
 public int[] snap;
}
```

Wait-Free Snapshot Label

```
public class SnapValue {
 public int    label;
 public int    value;
 public int[] snap;
}
```

Counter incremented with each snapshot

Wait-Free Snapshot Label

```
public class SnapValue {
 public int   label;
 public int   value;
 public int[] snap;
}
```

Actual value

Wait-Free Snapshot Label

```
public class SnapValue {
 public int    label;
 public int    value;
 public int[] snap;
}
```

most recent snapshot

Wait-free Update

```
public void update(int value) {
 int i = Thread.myIndex();
 int[] snap = this.scan();
 SnapValue oldValue = r[i].read();
 SnapValue newValue =
  new SnapValue(oldValue.label+1,
                value, snap);
 r[i].write(newValue);
 }
```

Wait-free Scan

```
public void update(int value) {
 int i = Thread.myIndex();
 int[] snap = this.scan();
 SnapValue oldValue = r[i].read();
 SnapValue newValue =
  new SnapValue(oldValue.label+1,
                value, snap);
 r[i].write(newValue);
 }
```

Take scan

Wait-free Scan

```
public void update(int value) {
 int i = Thread.myIndex();
 int[] snap = this.scan();
 SnapValue oldValue = r[i].read();
 SnapValue newValue =
  new SnapValue(oldValue.label+1,
                value, snap);
 r[i].write(newValue);
 }
```

Take scan

Label value with scan

Wait-free Scan

```
 public int[] scan() {
  SnapValue[] oldCopy, newCopy;
  boolean[] moved = new boolean[n];
  oldCopy = collect();
  collect: while (true) {
  newCopy = collect();
  for (int j = 0; j < n; j++) {
    if (oldCopy[j].label != newCopy[j].label) {
      …
  }}
  return getValues(newCopy);
}}}
```

Wait-free Scan

```
public int[] scan() {
 SnapValue[] oldCopy, newCopy;
 boolean[] moved = new boolean[n];
 oldCopy = collect();
 collect: while (true) {
 newCopy = collect();
 for (int j = 0; j < n; j++) {
   if (oldCopy[j].label != newCopy[j].label) {
     …
 }}
 return getValues(newCopy);
}}}
```

Keep track of who moved

Wait-free Scan

```
public int[] scan() {
 SnapValue[] oldCopy, newCopy;
 boolean[] moved = new boolean[n];
 oldCopy = collect();
 collect: while (true) {
 newCopy = collect();
 for (int j = 0; j < n; j++) {
   if (oldCopy[j].label != newCopy[j].label) {
     …
 }}
 return getValues(newCopy);
}}}
```

Repeated double collect

Wait-free Scan

```
public int[] scan() {
 SnapValue[] oldCopy, newCopy;
 boolean[] moved = new boolean[n];
 oldCopy = collect();
 collect: while (true) {
 newCopy = collect();
 for (int j = 0; j < n; j++) {
   if (oldCopy[j].label != newCopy[j].label) {
     …
 }}
 return getValues(newCopy);
}}}
```

If mismatch detected…lets expand here…

Mismatch Detected

```
  if (oldCopy[j].label != newCopy[j].label) {
    if (moved[j]) {       // second move
     return newCopy[j].snap;
    } else {
     moved[j] = true;
     oldCopy = newCopy;
     continue collect;
  }}}
  return getValues(newCopy);
}}}
```

Mismatch Detected

```
  if (oldCopy[j].label != newCopy[j].label) {
   if (moved[j]) {
    return newCopy[j].snap;
   } else {
    moved[j] = true;
    oldCopy = newCopy;
    continue collect;
  }}}
  return getValues(newCopy);
}}}
```

If thread moved twice, just steal its second snapshot

Mismatch Detected

```
  if (oldCopy[j].label != newCopy[j].label) {
   if (moved[j]) {      // second move
    return newCopy[j].snap;
   } else {
    moved[j] = true;
    oldCopy = newCopy;
    continue collect;
 }}}
 return getValues(newCopy);
}}}
```

Remember that thread moved

Snapshot Summary

- We saw that we can build wait-free atomic snapshot from atomic registers

Multiple Assignment Theorem

- Atomic registers cannot implement multiple assignment
- Weird or what?
  - Single location write/multi location read OK
    (= Atomic Snapshot)
  - Multi location write/single 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

$value[i_1]= v_1$

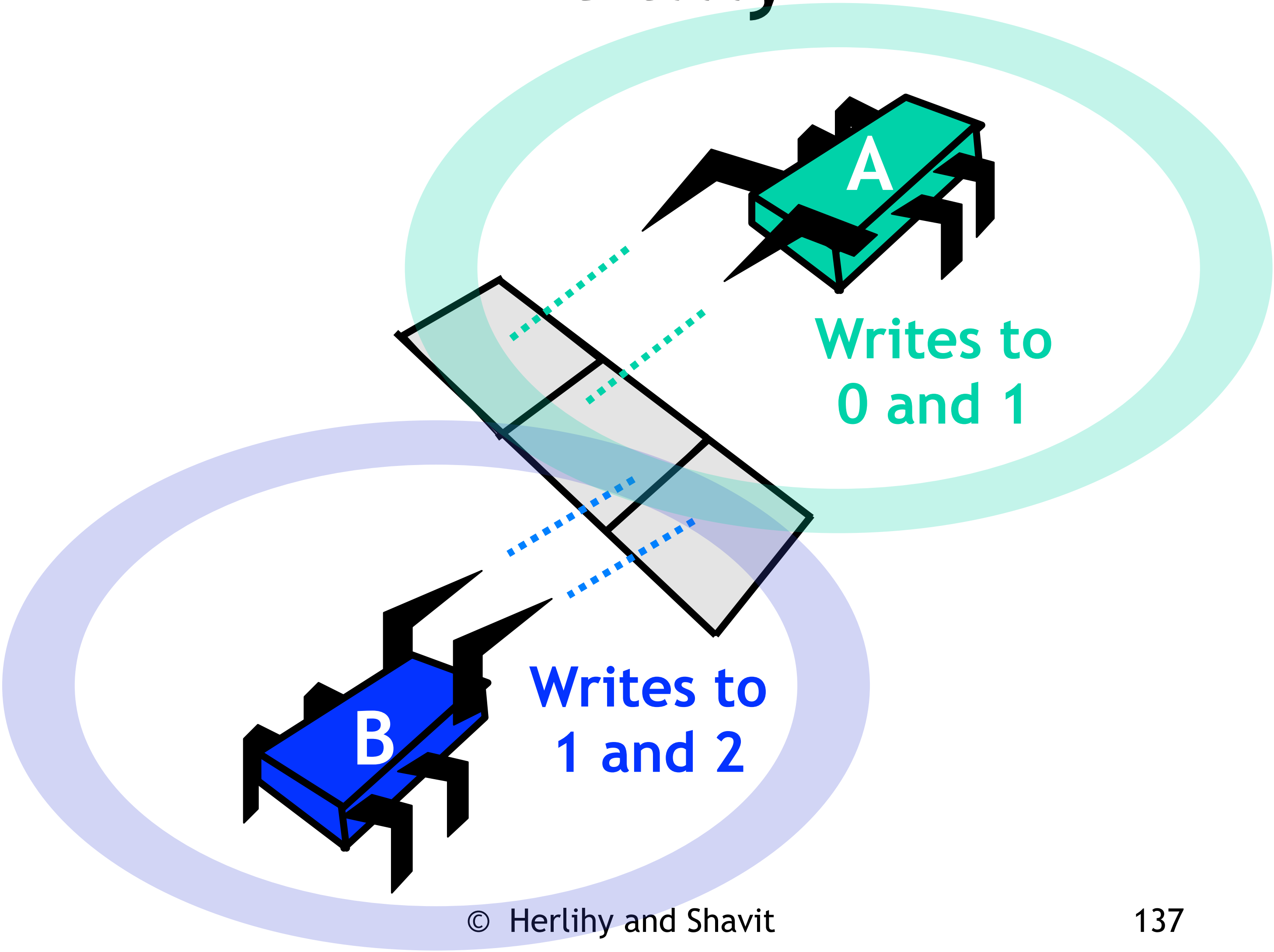
$value[i_2]= v_2$

Double Assignment Interface

```
interface Assign2 {
 public void assign(int i_1, int v_1,
                    int i_2, int v_2);
 public int read(int i);
}
```

Return i-th value

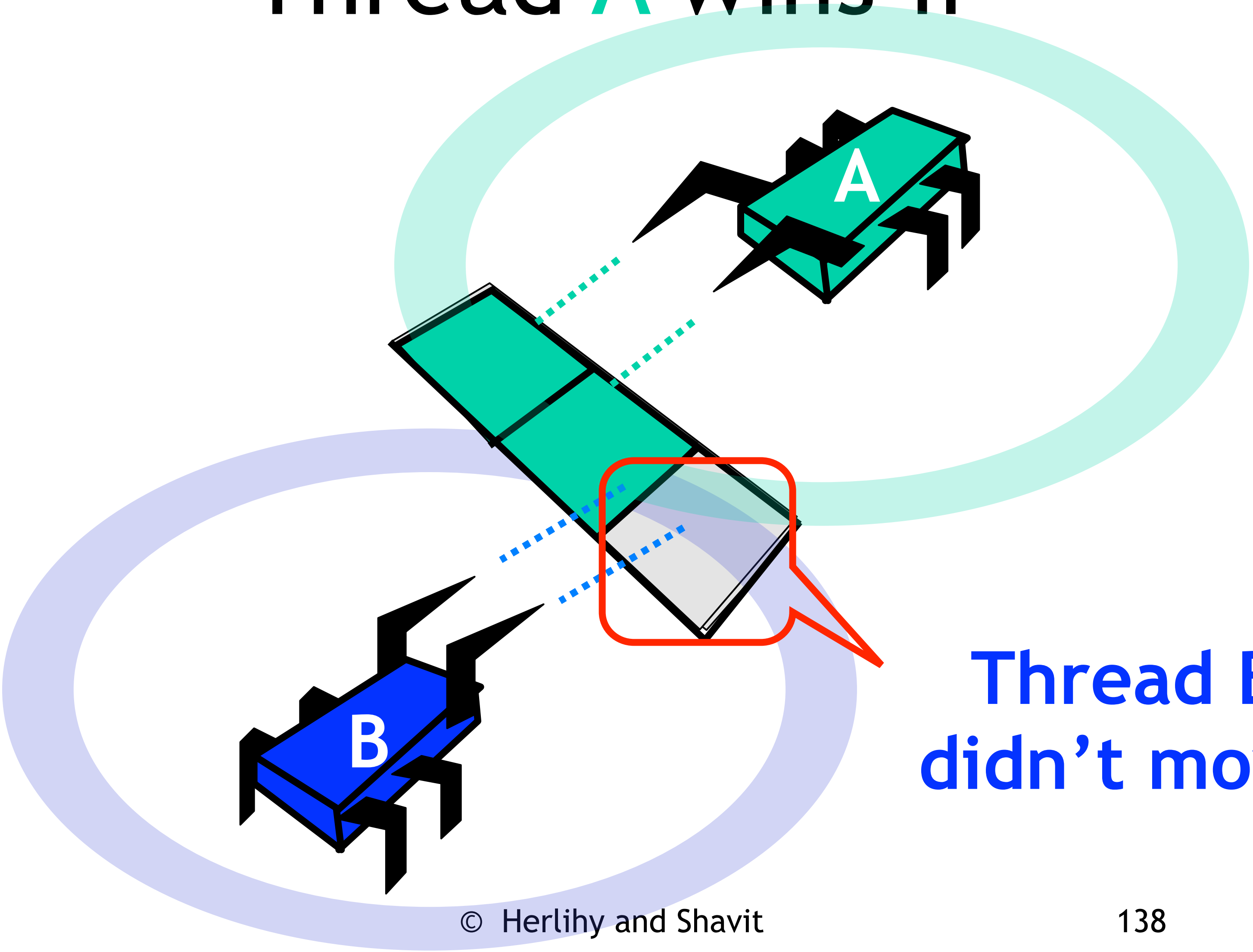
Initially



Writes to
0 and 1

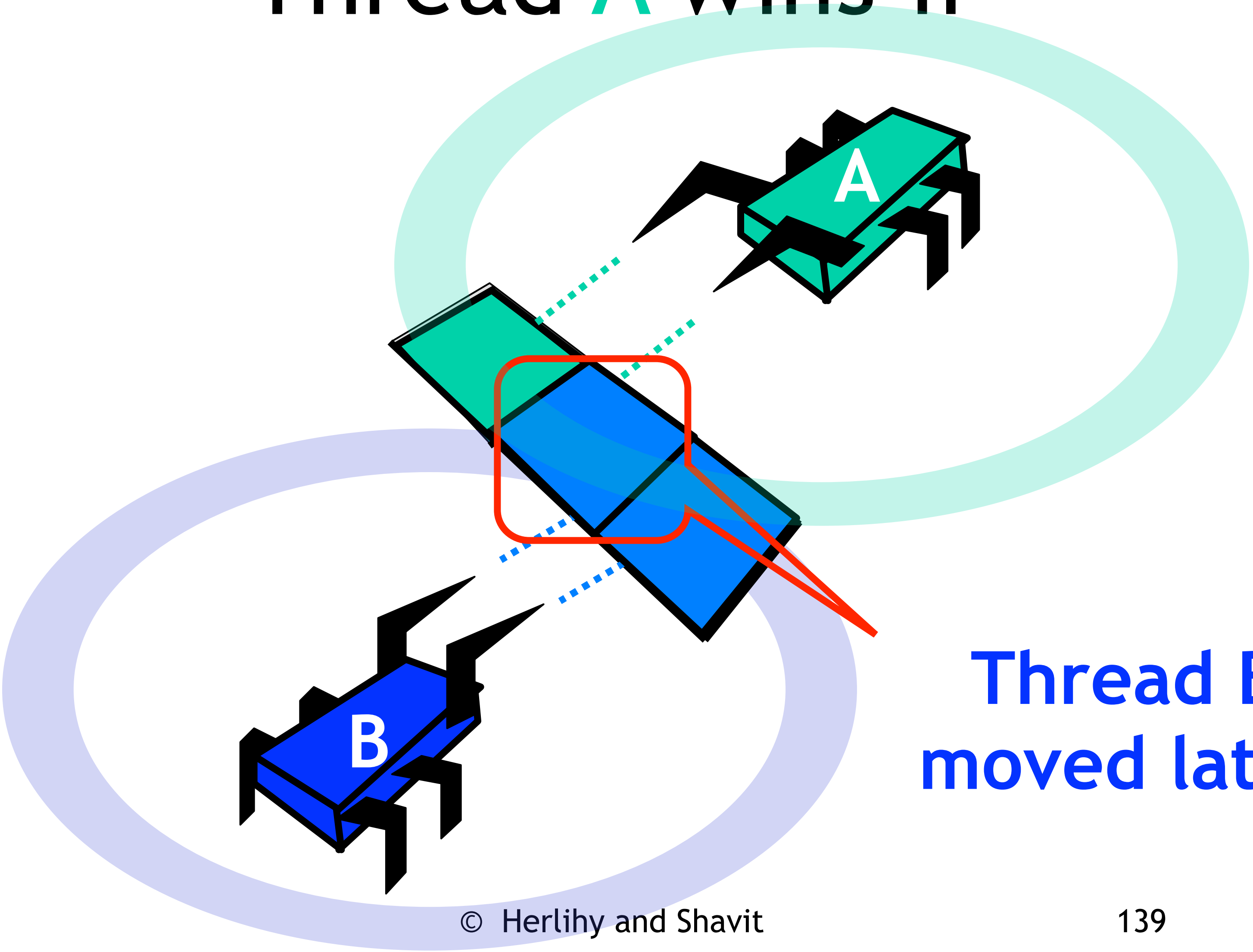
Writes to
1 and 2

Thread A wins if



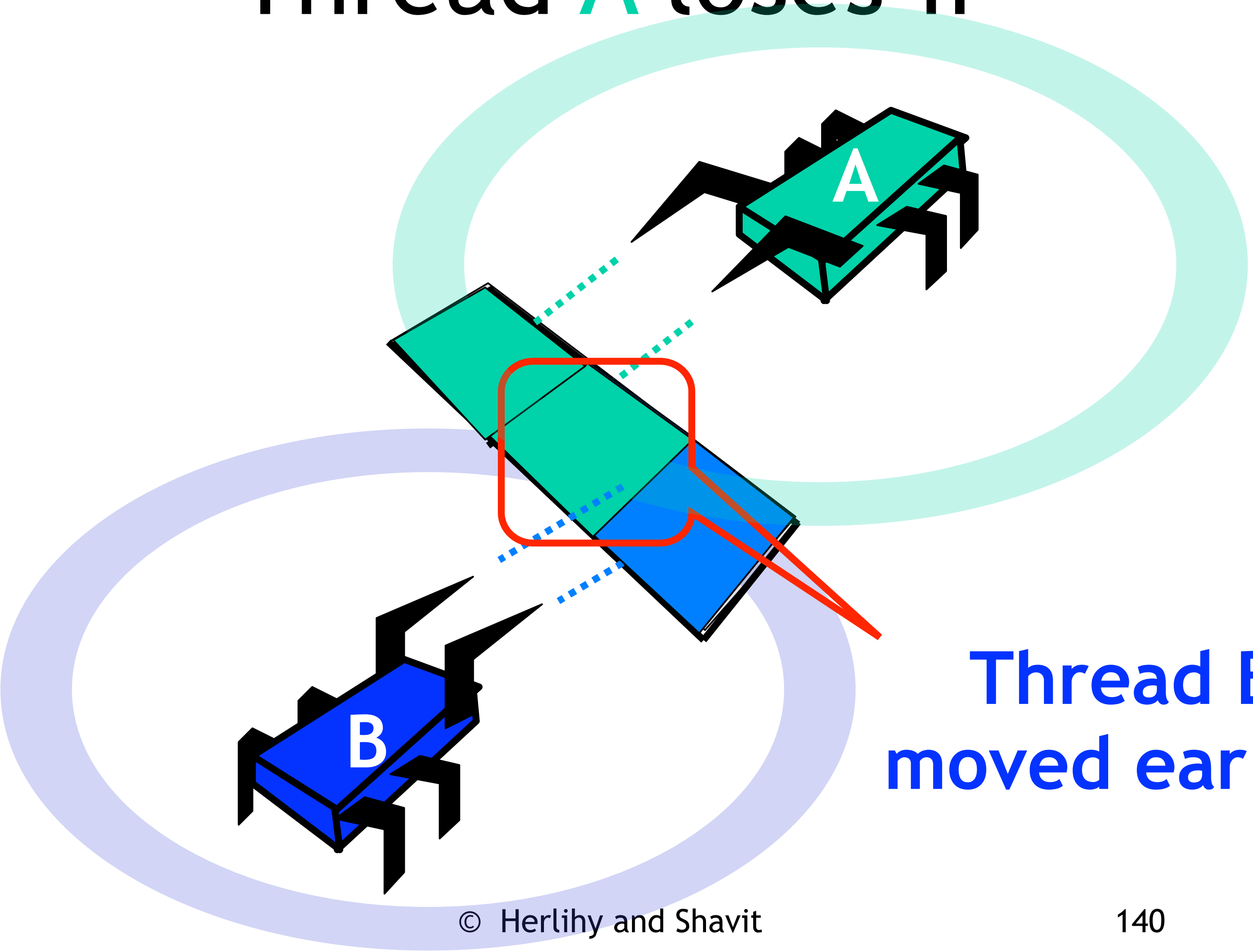
Thread B didn't move

Thread A wins if



Thread B moved later

Thread A loses if



Thread B moved earlier

Multi-Consensus Code

```
class MultiConsensus extends Consensus{
 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 Consensus{
 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=1-i and proposes value

Multi-Consensus Code

```
class MultiConsensus extends Consensus{
 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 Consensus{
 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)

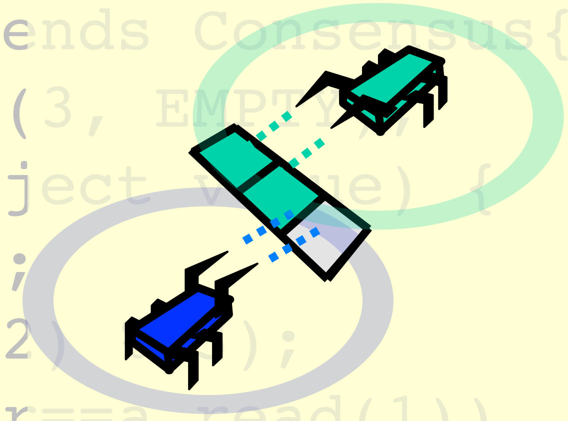
Multi-Consensus Code

```
class MultiConsensus extends Consensus{
 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

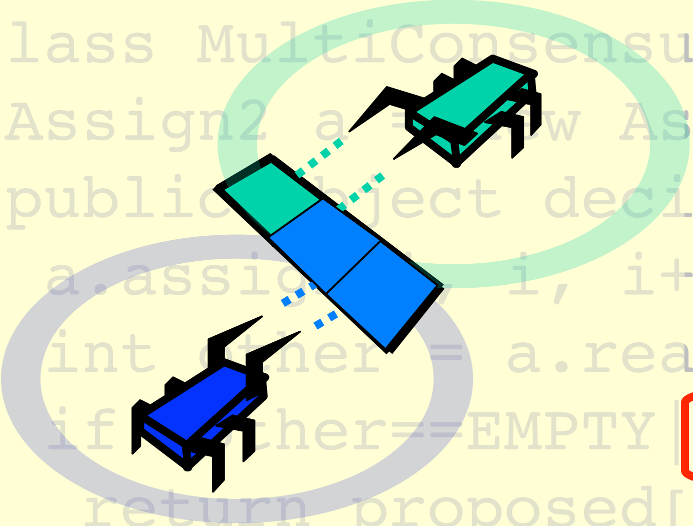
Multi-Consensus Code

```
class MultiConsensus extends Consensus{
 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 didn't move, so I win

Multi-Consensus Code



```
class MultiConsensus extends Consensus{
 Assign2 a = new Assign2(3, EMPTY);
 public Object decide(object 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];
}}
```

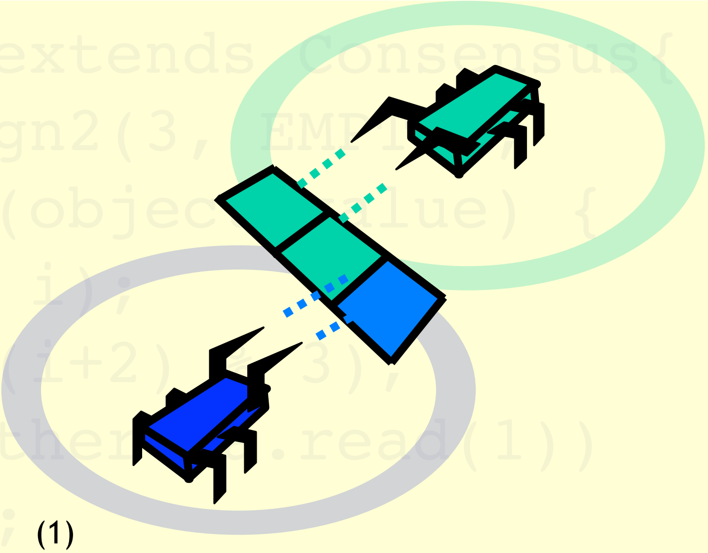
Other thread moved later so I win

Multi-Consensus Code

```
class MultiConsensus extends Consensus{
 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 Consensus{
 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]; (1)
  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 void synchronized
  getAndMumble() {
     int prior  = this.value;
     this.value = mumble(this.value);
     return prior;
  }
}
```

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;
  }
}
```

Return prior value

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;
  }
}
```

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

Example: getAndSet (swap)

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

 public void 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 void synchronized
   getAndIncrement() {
  int prior  = this.value;
  this.value = this.value + 1;
  return prior;
 }
 …
}
```

getAndIncrement

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

 public void 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 void 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 void 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 false;
  } … }
```

```
return true;
```

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;
  }
}
```

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

Am I first?

Proof

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

Yes, return my input

Proof

```
public class RMWConsensus
     implements 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

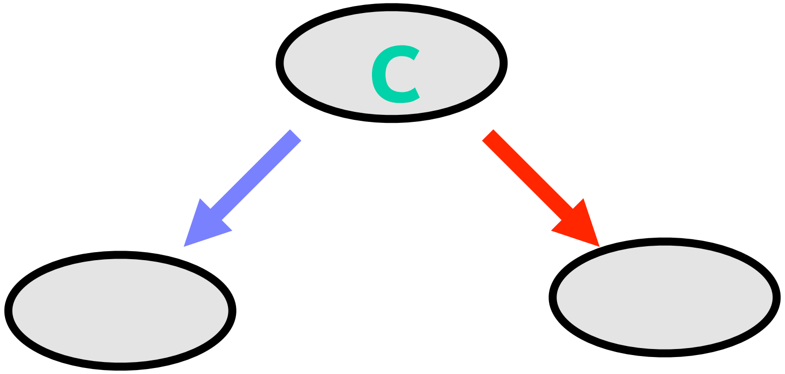
- “test-and-set” getAndSet(1) $f(v)=1$

  Overwrite $f_i(f_j(v))=f_i(v)$
- “swap” getAndSet(x) $f(v)=x$

  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



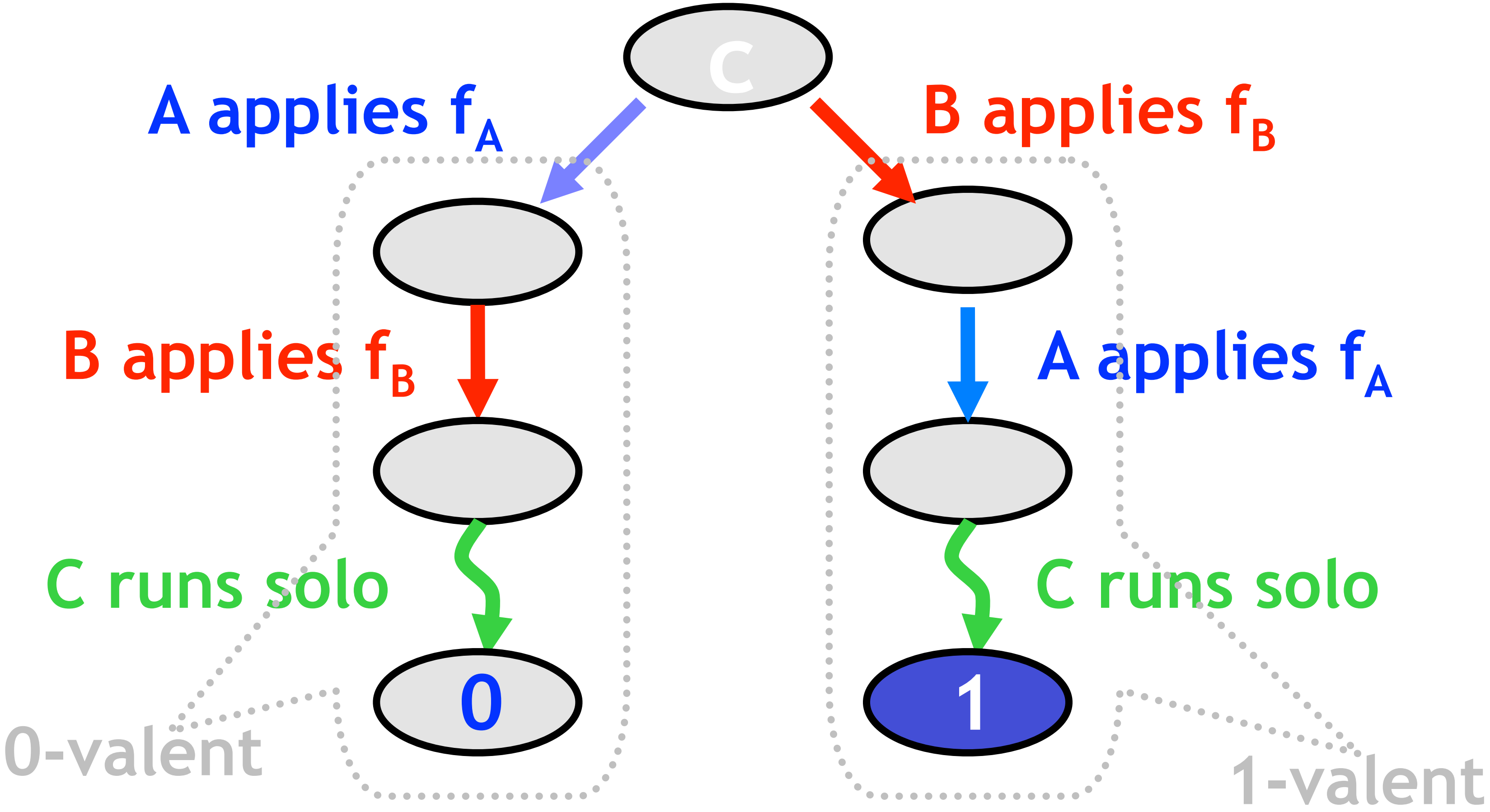
A about to apply f_A

B about to apply f_B

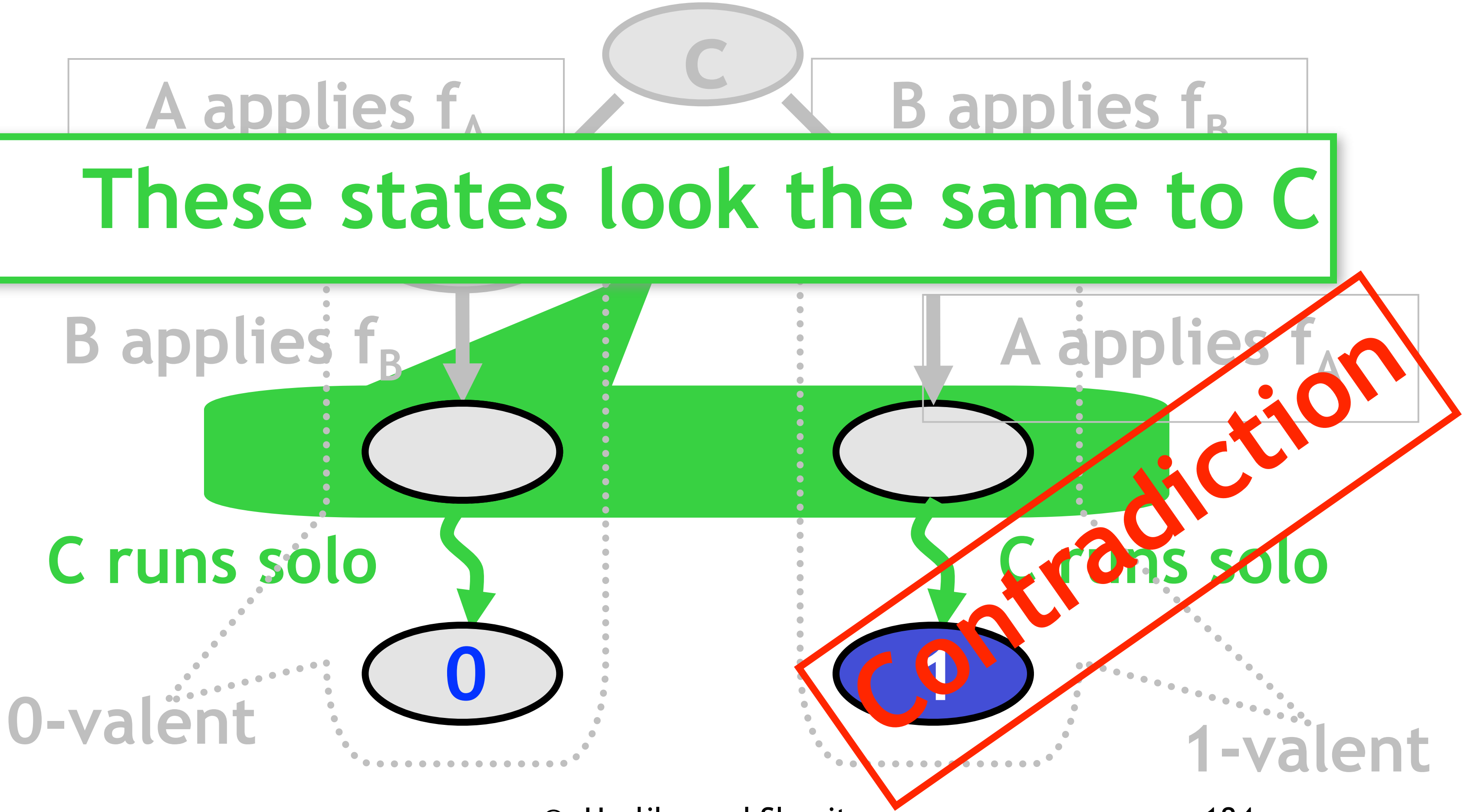
0-valent

1-valent

Maybe the Functions Commute

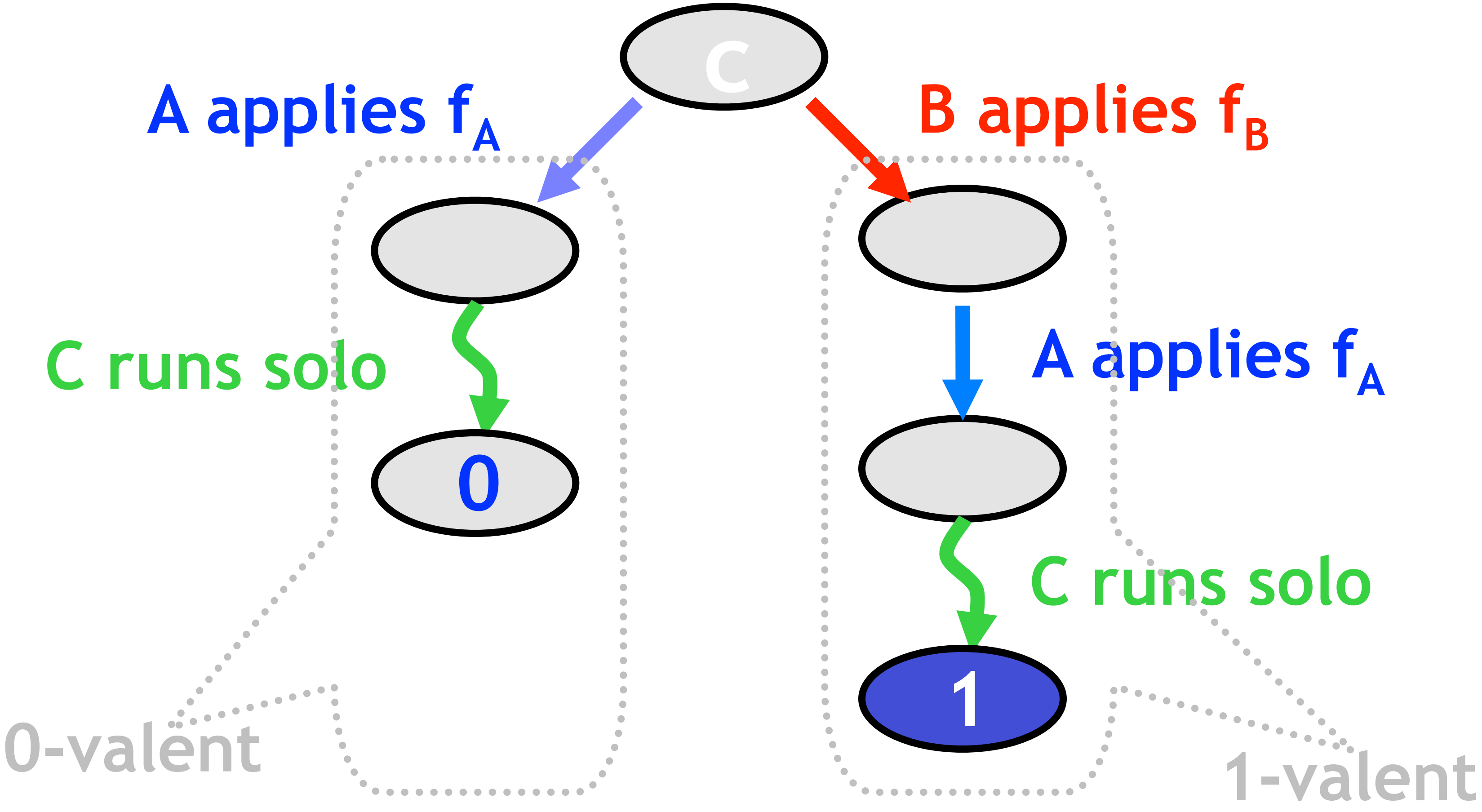


Maybe the Functions Commute

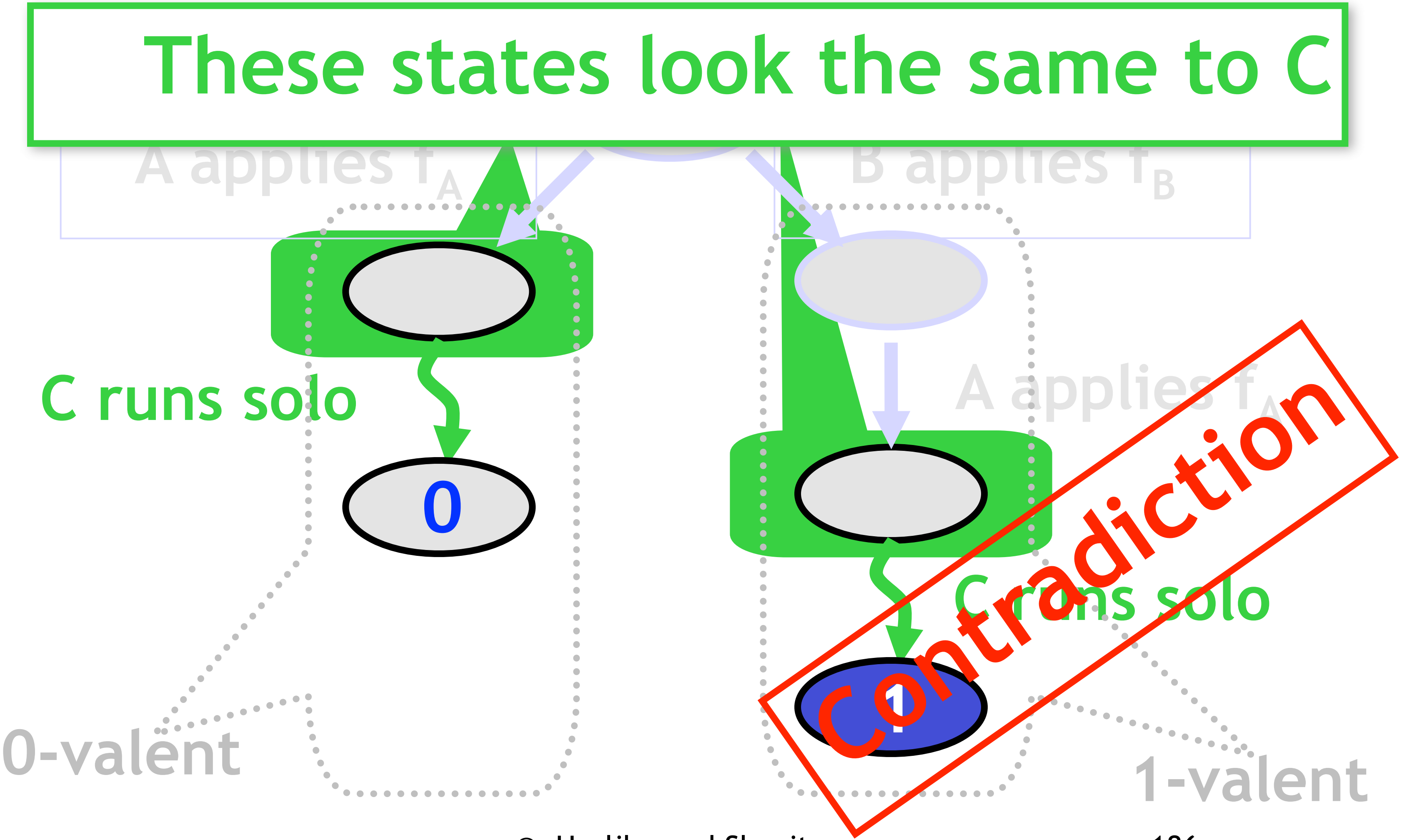


These states look the same to C

Maybe the Functions Overwrite



Maybe the Functions Overwrite



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 ∞ Consensus Number

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

compareAndSet Has ∞ Consensus Number

```
public class RMWConsensus
     implements 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 ∞ Consensus Number

```
public class RMWConsensus
     implements 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 ∞ Consensus Number

```
public class RMWConsensus
     implements 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

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 (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: in an infinite execution 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 just as impossible
- Lock-free = Wait-free for finite executions
- **All the results we presented hold for lock-free algorithms also.**

There is More: Universality

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