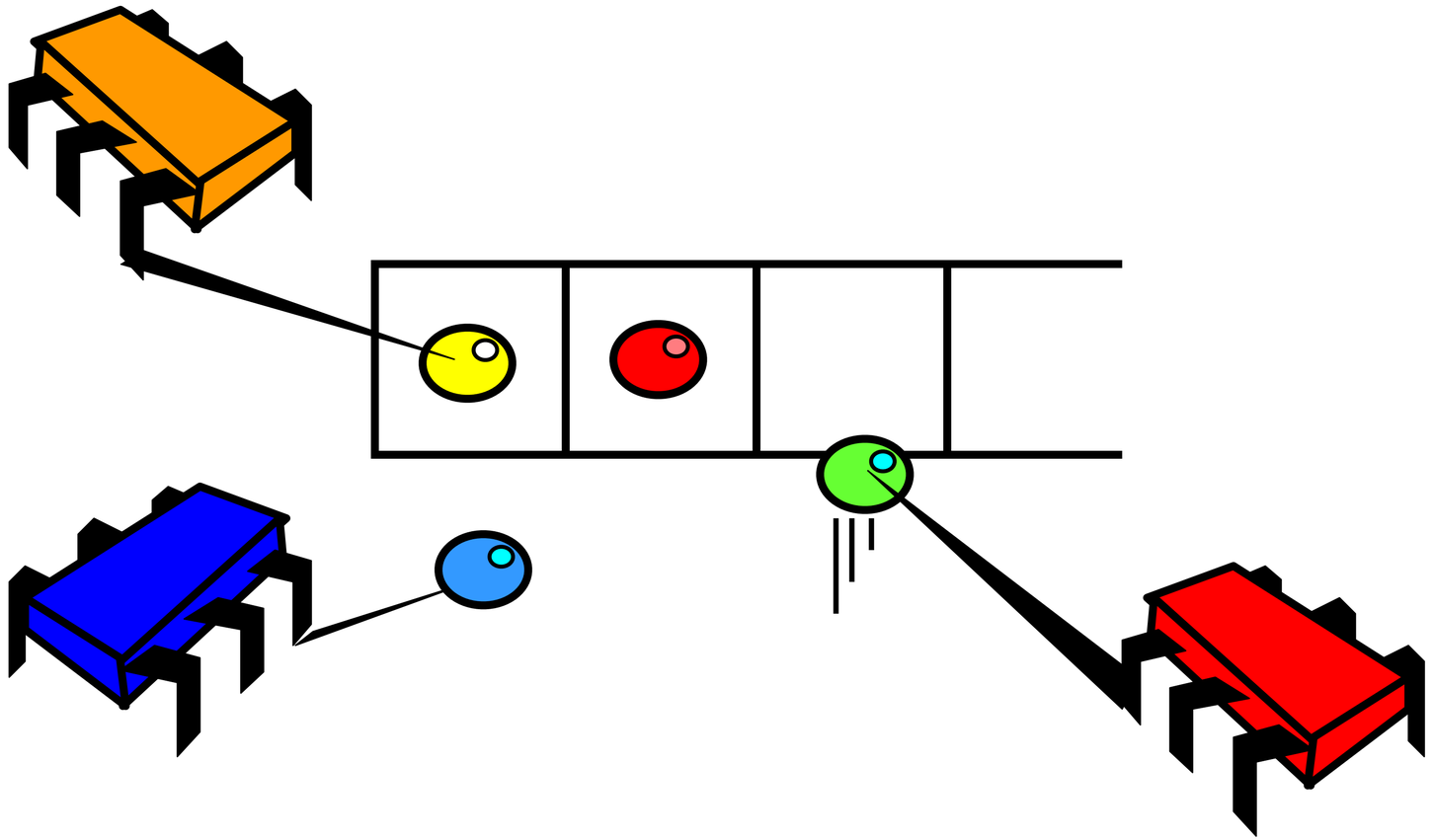
YSC4231: Parallel, Concurrent and Distributed Programming

Concurrent Consensus and Read-Modify Write Operations

What Does Consensus have to do with Concurrent Objects?



Consensus Object

```
trait Consensus[T] {

  def decide(value: T): T

}
```

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

Scala Jargon Watch

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

Generic Consensus Protocol

```
abstract class ConsensusProtocol[T] extends Consensus[T] {

  private val THREADS_NUM = 3248
  var proposed = new Array[T](THREADS_NUM)

  protected def propose(value: T): Unit = {
    val i = ThreadID.get
    proposed(i) = value
  }

  def decide(value: T): T
}
```

Generic Consensus Protocol

```
abstract class ConsensusProtocol[T] extends Consensus[T] {

  private val THREADS_NUM = 2019
  var proposed = new Array[T](THREADS_NUM)

  protected def propose(value: T): Unit = {
    val i = ThreadID.get
    proposed(i) = value
  }

  def decide(value: T): T
}
```

Each thread's proposed value

Generic Consensus Protocol

```
abstract class ConsensusProtocol[T] extends Consensus[T] {

  private val THREADS_NUM = 2019
  var proposed = new Array[T](THREADS_NUM)

  protected def propose(value: T): Unit = {
    val i = ThreadID.get
    proposed(i) = value
  }

  def decid
}
```

Propose a value

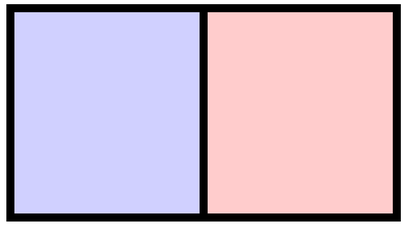
Generic Consensus Protocol

```
abstract class ConsensusProtocol[T] extends Consensus[T] {

  private val THREADS_NUM = 2019
  var proposed = new Array[T](THREADS_NUM)

  def decide(value: T): T = {
    val i = ThreadID.get
    proposed(i) = value
  }
}
```

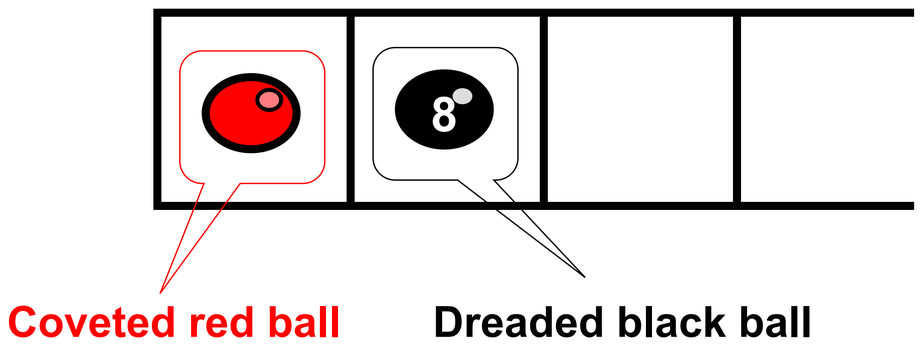
Decide a value: abstract method means subclass does the real work

Can a FIFO Queue Implement Consensus?

FIFO Consensus

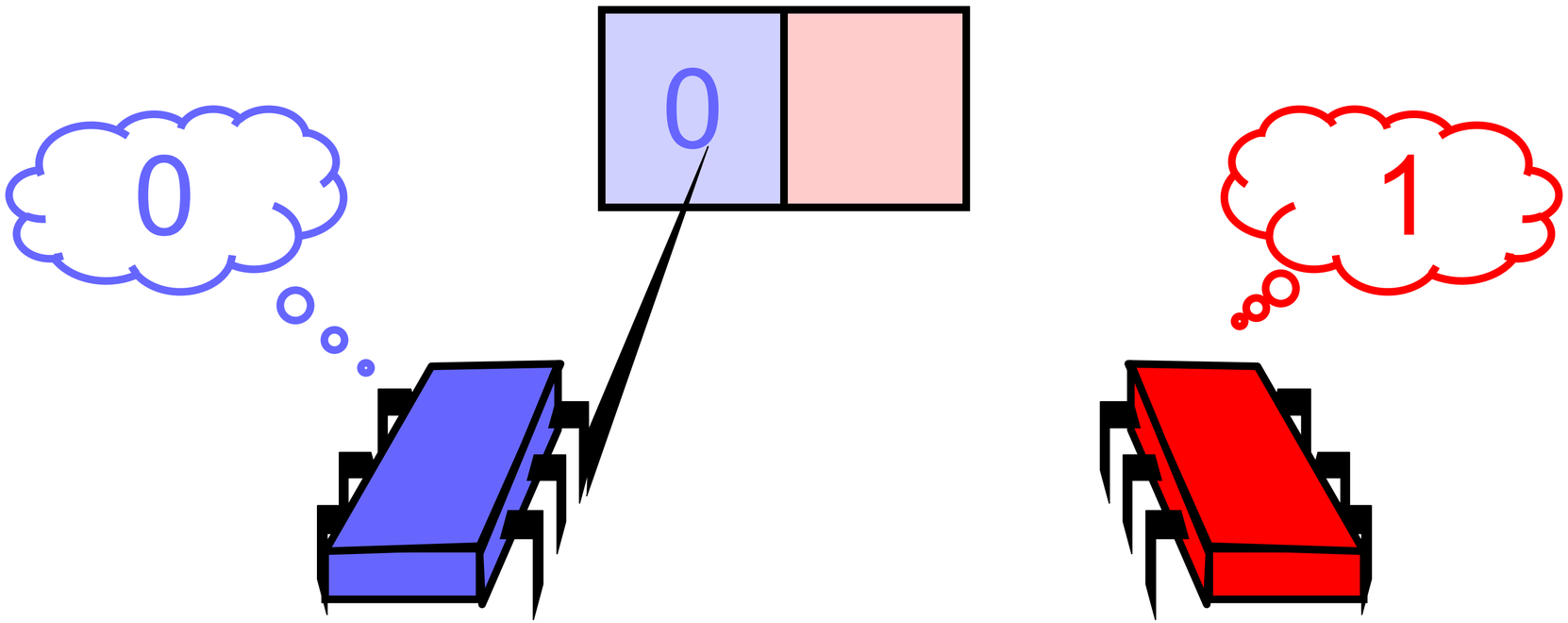


proposed array

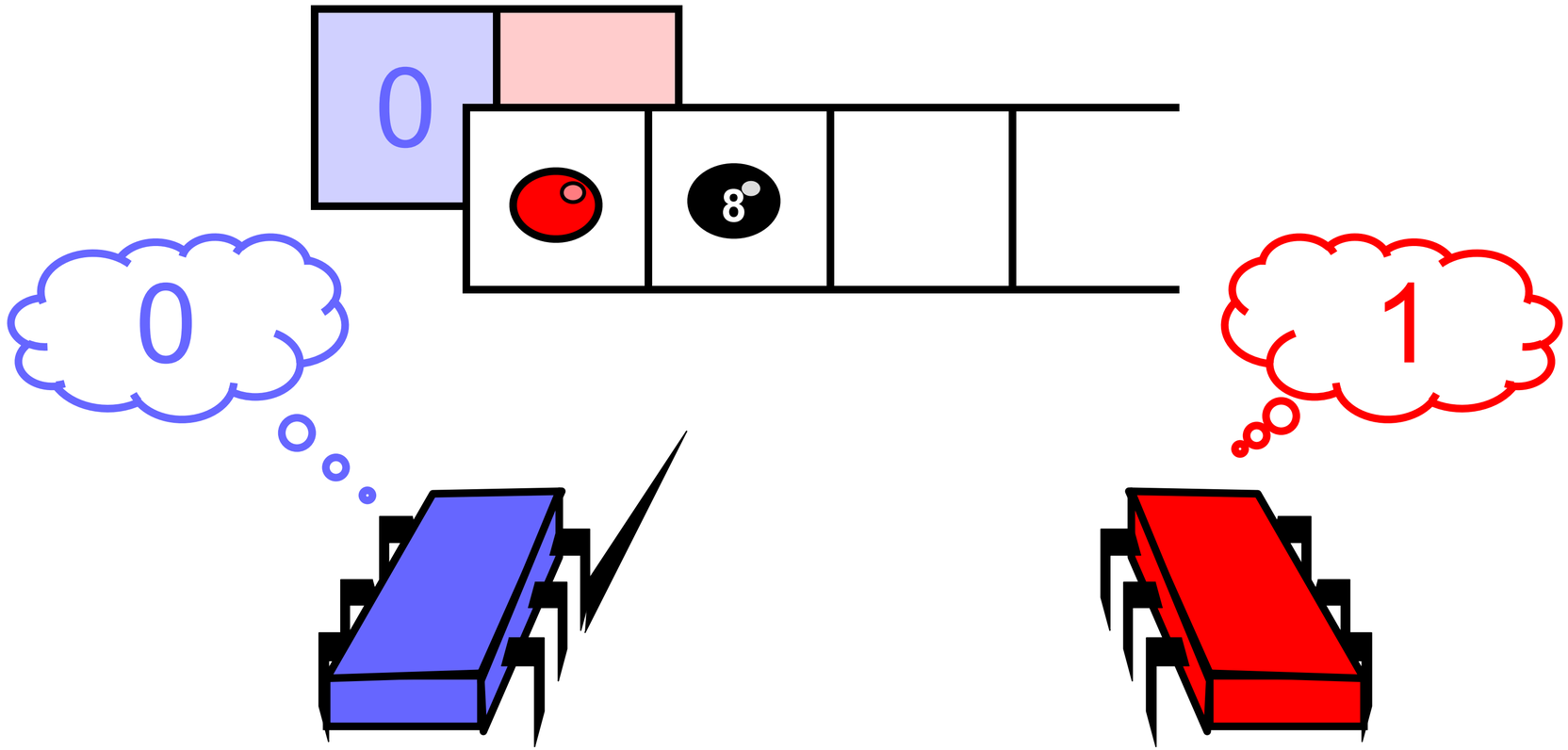


FIFO Queue with red and black balls

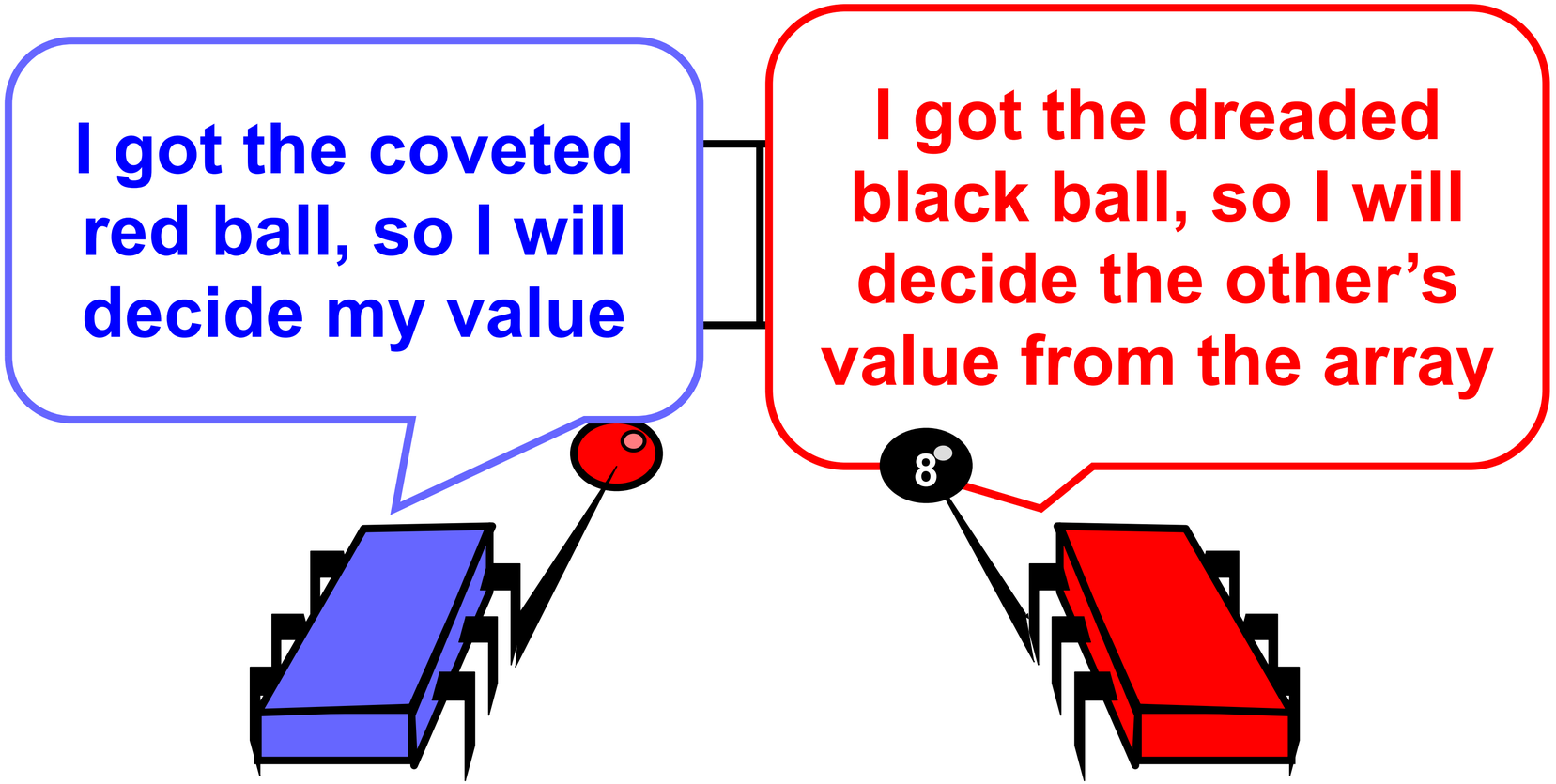
Protocol: Write Value to Array



Protocol: Take Next Item from Queue



Protocol: Take Next Item from Queue

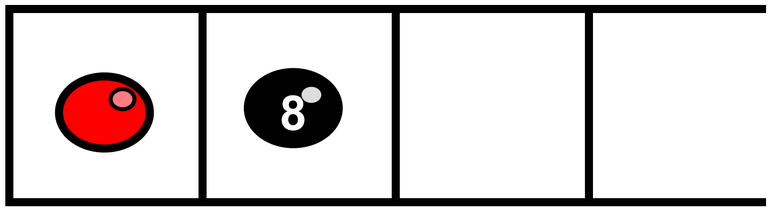


Consensus Using FIFO Queue

```
public class QueueConsensus[T]
   extends ConsensusProtocol[T] {
   val queue : Queue = new Queue()
   queue.enq(Ball.RED)
   queue.enq(Ball.BLACK)
…
}
```

Initialize Queue

```
public class QueueConsensus[T]
  extends ConsensusProtocol[T] {
  val queue : Queue = new Queue()
  queue.enq(Ball.RED)
  queue.enq(Ball.BLACK)
…
}
```



Who Won?

```
public class QueueConsensus[T]
  extends ConsensusProtocol[T] {
  val queue : Queue = new Queue()

  override def decide(value: T) = {
    propose(value)
    val ball = queue.deq()
    val i = ThreadID.get
    if (ball == Ball.RED) {
      proposed(i).get()
    } else {
      proposed(1 - i).get()
    }
  }}
```

Who Won?

```
public class QueueConsensus[T]
  extends ConsensusProtocol[T] {
  val queue : Queue = new Queue()

  override def decide(value: T) = {
    propose(value)
    val ball = queue.deq()
    val i = ThreadID.get
    if (ball == Ball.RED) {
      proposed(i).get()
    } else {
      proposed(1 - i).get()
    }
  }}
```

Race to dequeue first queue item

Who Won?

```
public class QueueConsensus[T]
  extends ConsensusProtocol[T] {
  val queue : Queue = new Queue()

  override def decide(value: T) = {
    propose(value)
    val ball = queue.deq()
    val i = ThreadID.get
    if (ball == Ball.RED) {
      proposed(i).get()
    } else {
      proposed(1 - i).get()
    }
  }}
```

I win if I was first

Who Won?

```
public class QueueConsensus[T]
  extends ConsensusProtocol[T] {
  val queue : Queue = new Queue()

  override def decide(value: T) = {
    propose(value)
    val ball = queue.deq()
    val i = ThreadID.get
    if (ball == Ball.RED) {
      proposed(i).get()
    } else {
      proposed(1 - i).get()
    }
  }}
```

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 array
  - Before dequeueing from queue

Demo

- Testing Queue-based consensus

Theorem

- We can solve 2-thread consensus using only
  - A two-dequeuer wait-free 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 only

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 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!

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

Double Assignment Interface

```
class Assign23[T](val init: T) {

  val r: Array[AtomicReference[T]] =
         Array.fill(3)(new AtomicReference(init))

  def assign(v0: T, v1: T, i0: Int, i1: Int): Unit =
    this.synchronized {
      r(i0).set(v0)
      r(i1).set(v1)
    }

  def read(i: Int): T = this.synchronized {
    r(i).get()
  }
}
```

Double Assignment Interface

```
class Assign23[T](val init: T) {

  val r: Array[AtomicReference[T]] =
         Array.fill(3)(new AtomicReference(init))

  def assign(v0: T, v1: T, i0: Int, i1: Int): Unit =
    this.synchronized {
      r(i0).set(v0)
      r(i1).set(v1)
    }

  def read(i: Int): T = this.synchronized {
    r(i).get()
  }
}
```

Atomically assign

$r(i_0) = v_0$

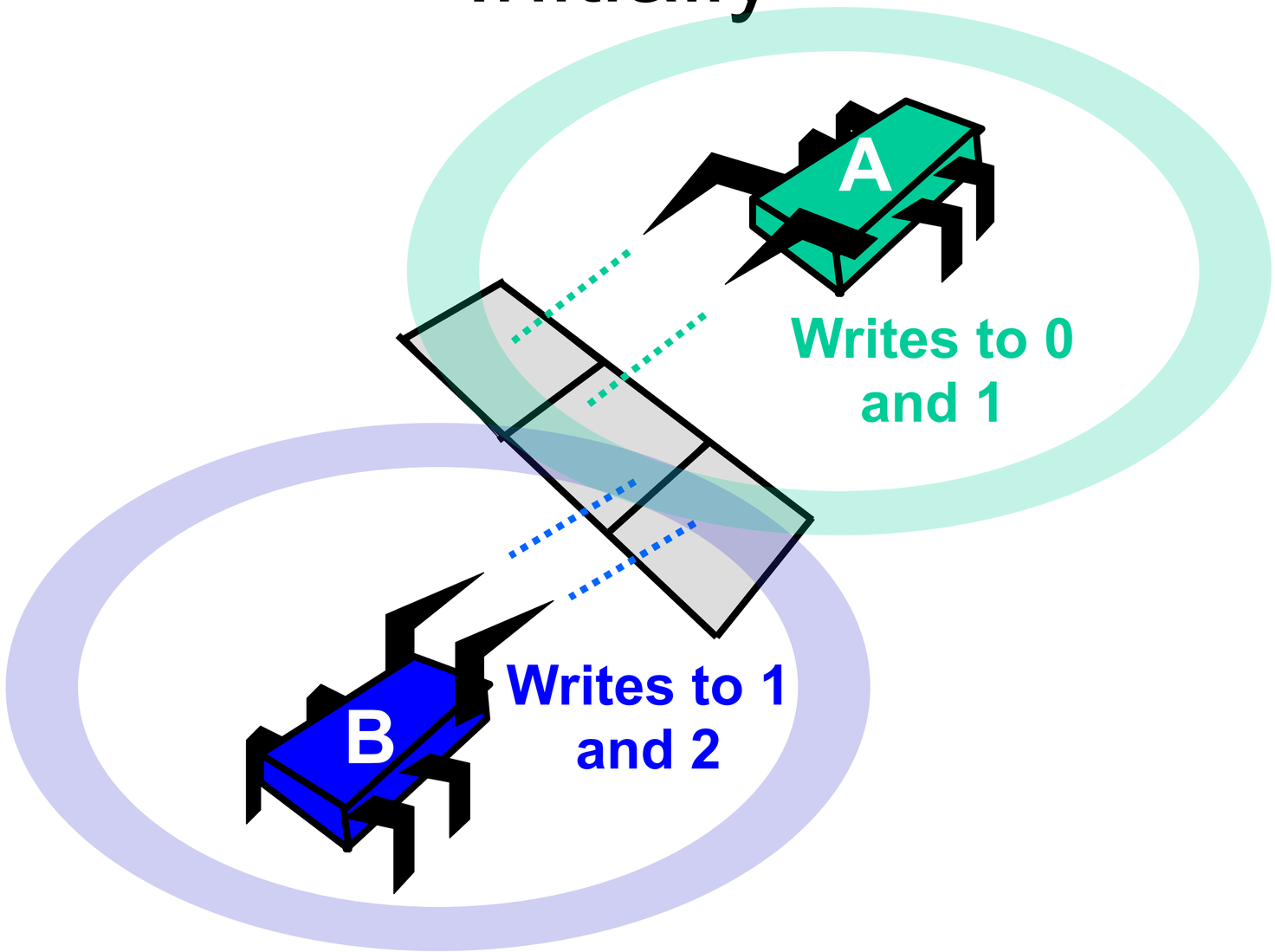
$r(i_1) = v_1$

Double Assignment Interface

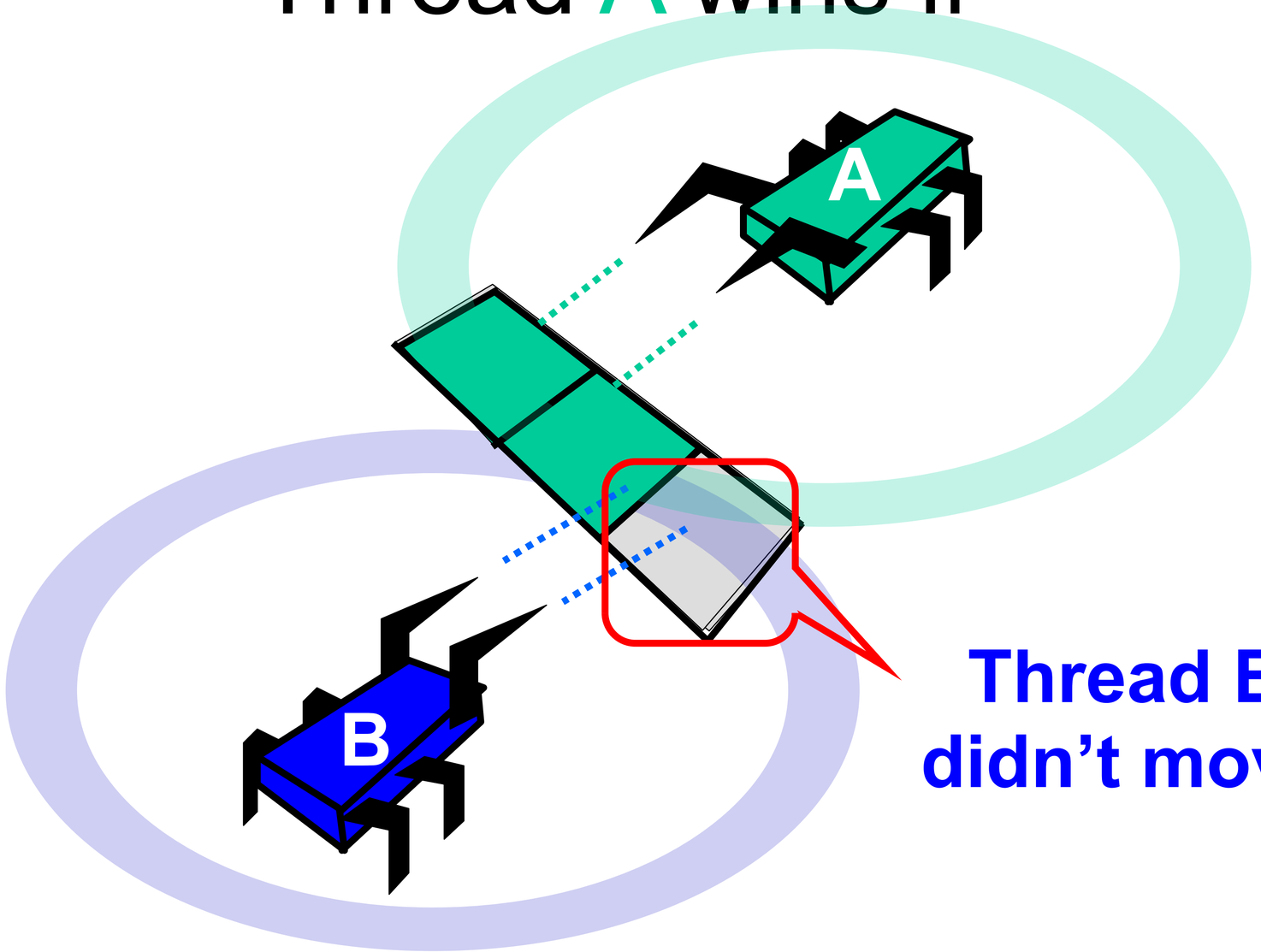
```
class Assign23[T](val init: T) {

  val r: Array[AtomicReference[T]] =
        Array.fill(3)(new AtomicReference(init))

  def assign(v0: T, v1: T, i0: Int, i1: Int): Unit =
    this.synchronized {
      r(i0).set(v0)
      r(i1).set(v1)
    }

  def read(i: Int): T = this.synchronized {
    r(i).get()
  }
}
```

Return i^{th} value

Initially

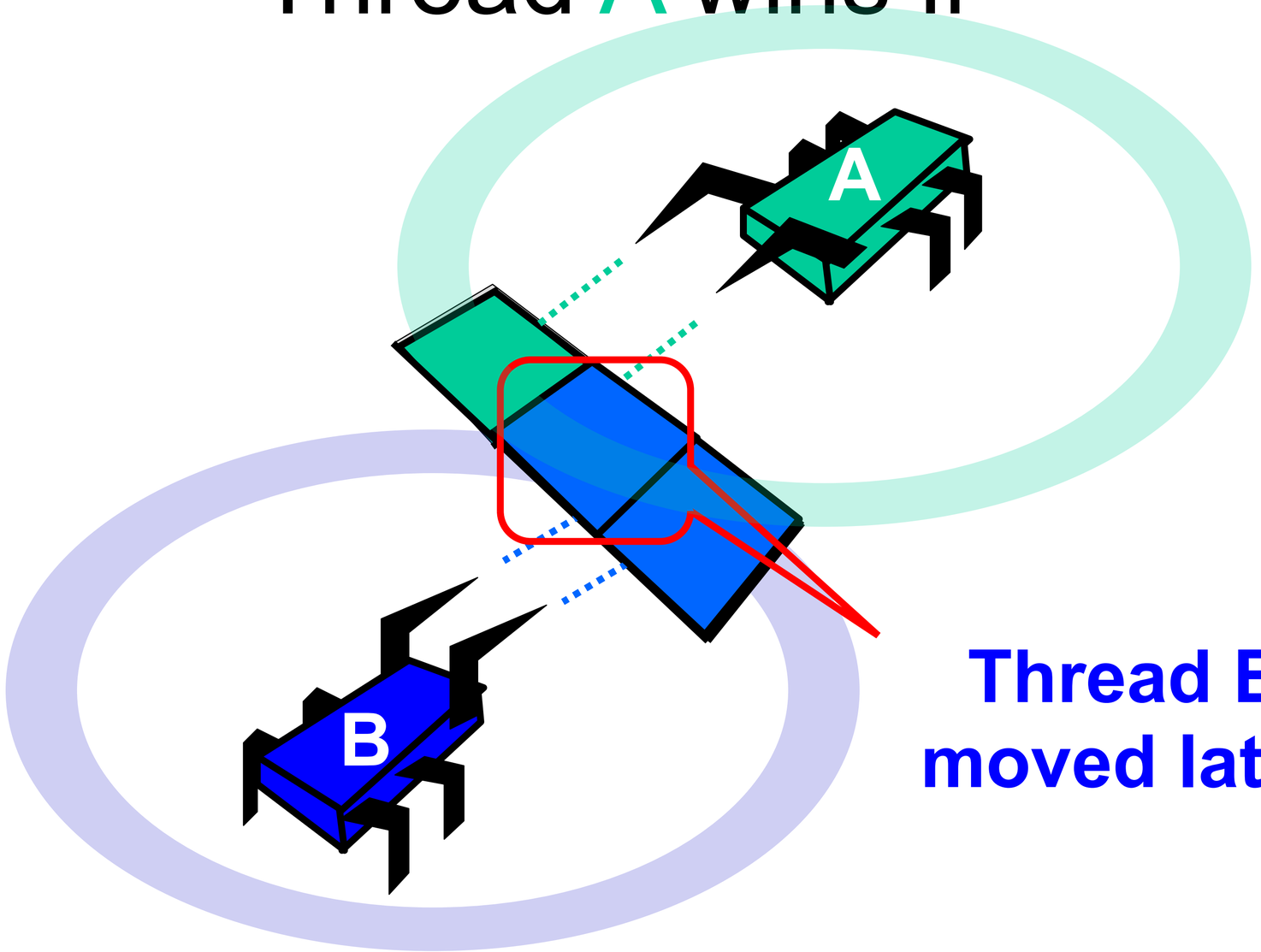


Thread A wins if



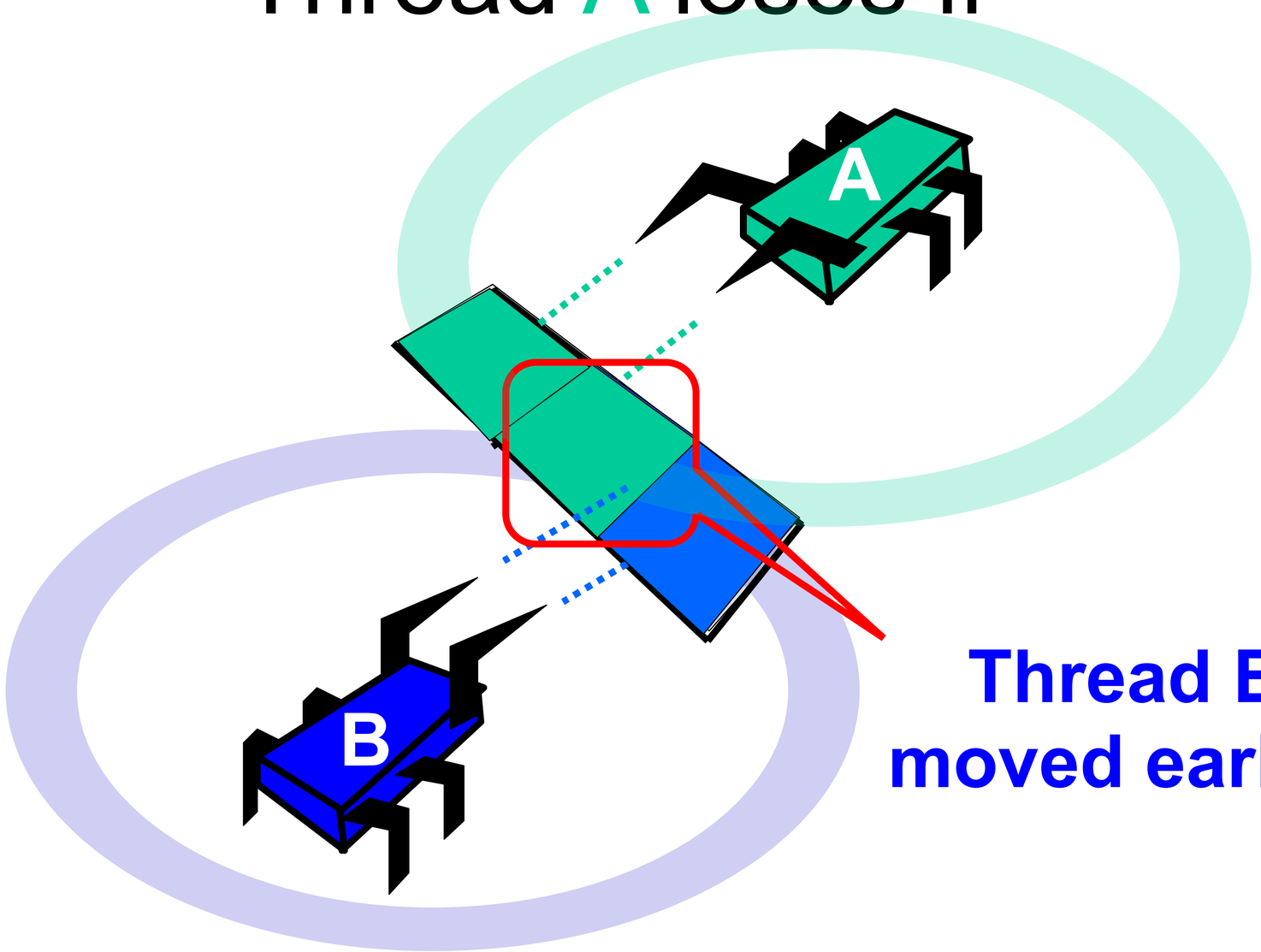
Thread B didn’t move

Thread A wins if



Thread B moved later

Thread A loses if



Thread B moved earlier

Multi-Consensus Code

```
class MultiConsensus extends … {
  private val NULL = -1
  private val assign23 = new Assign23(NULL)

  override def decide(value: T) = {
    propose(value)
    val i = ThreadID.get
    assign23.assign(i, i, i, i + 1)
    val other = assign23.read((i + 2) % 3)
    if (other == NULL || other == assign23.read(1)) {
      proposed(i).get()      // I win
    } else {
      proposed(1 - i).get() // I lose
    }
  }}
```

Multi-Consensus Code

```
class MultiConsensus extends … {
  private val NULL = -1
  private val assign23 = new Assign23(NULL)

  override def decide(value: T) = {
    propose(value)
    val i = ThreadID.get
    assign23.assign(i, i, i, i + 1)
    val other = assign23.read((i + 2) % 3)
    if (other == NULL || other == assign23.read(1)) {
      proposed(i).get()      // I win
    } else {
      proposed(1 - i).get() // I lose
    }
  }}
```

Extends ConsensusProtocol

“decide” sets 1-i and proposes value

Multi-Consensus Code

```
class MultiConsensus extends … {
  private val NULL = -1
  private val assign23 = new Assign23(NULL)

  override def decide(value: T) = {
    propose(value)
    val i = ThreadID.get
    assign23.assign(i, i, i, i + 1)
    val other = assign23.read((i + 2) % 3)
    if (other == NULL || other == assign23.read(1)) {
      proposed(i).get()      // I win
    } else {
      proposed(1 - i).get() // I lose
    }
  }}
```

Three slots initialized to NULL

Multi-Consensus Code

```
class MultiConsensus extends … {
  private val NULL = -1
  private val assign23 = new Assign23(NULL)

  override def decide(value: T) = {
    propose(value)
    val i = ThreadID.get
    assign23.assign(i, i, i, i + 1)
    val other = assign23.read((i + 2) % 3)
    if (other == NULL || other == assign23.read(1)) {
      proposed(i).get()      // I win
    } else {
      proposed(1 - i).get() // I lose
    }
  }}
```

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

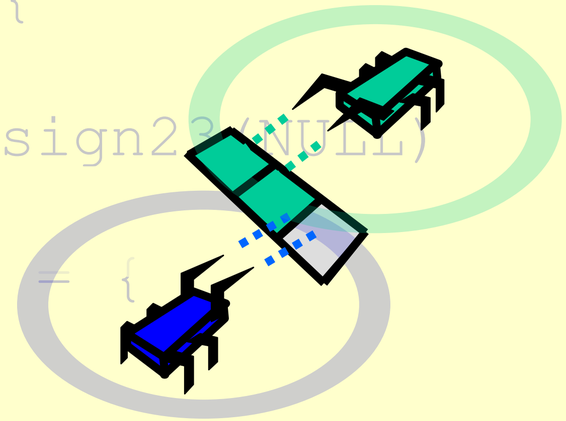
Multi-Consensus Code

```
class MultiConsensus extends … {
  private val NULL = -1
  private val assign23 = new Assign23(NULL)

  override def decide(value: T) = {
    propose(value)
    val i = ThreadID.get
    assign23.assign(i, i, i, i + 1)
    val other = assign23.read((i + 2) % 3)
    if (other == NULL || other == assign23.read(1)) {
      proposed(i).get()      // I win
    } else {
      proposed(1 - i).get() // I lose
    }
  }}
```

Read the register my thread didn't assign

Multi-Consensus Code

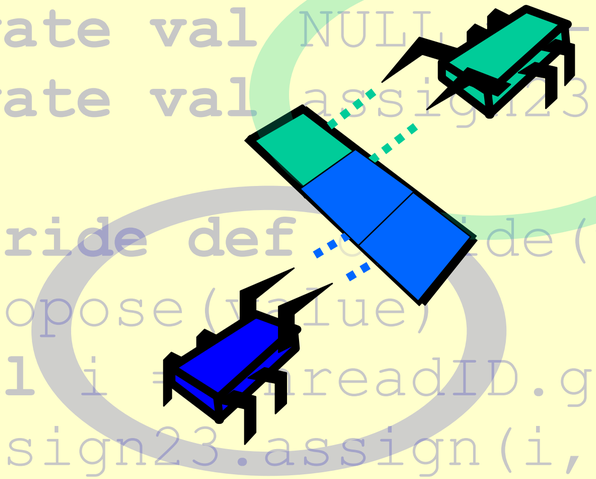
```
class MultiConsensus extends … {
  private val NULL = -1
  private val assign23 = new Assign23(NULL)

  override def decide(value: T) = {
    propose(value)
    val i = ThreadID.get
    assign23.assign(i, i, i, i + 1)
    val other = assign23.read((i + 2) % 3)
    if (other == NULL || other == assign23.read(1)) {
      proposed(i).get()      // I win
    } else {
      proposed(1 - i).get() // I lose
    }
  }}
```



Other thread didn't move, so I win

Multi-Consensus Code

```
class MultiConsensus extends … {
  private val NULL = -1
  private val assign23 = new Assign23(NULL)

  override def decide(value: T) = {
    propose(value)
    val i = ThreadID.get
    assign23.assign(i, i, i, i + 1)
    val other = assign23.read((i + 2) % 3)
    if (other == NULL || other == assign23.read(1)) {
      proposed(i).get()      // I win
    } else {
      proposed(1 - i).get() // I lose
    }
  }}
```



Other thread moved later so I win

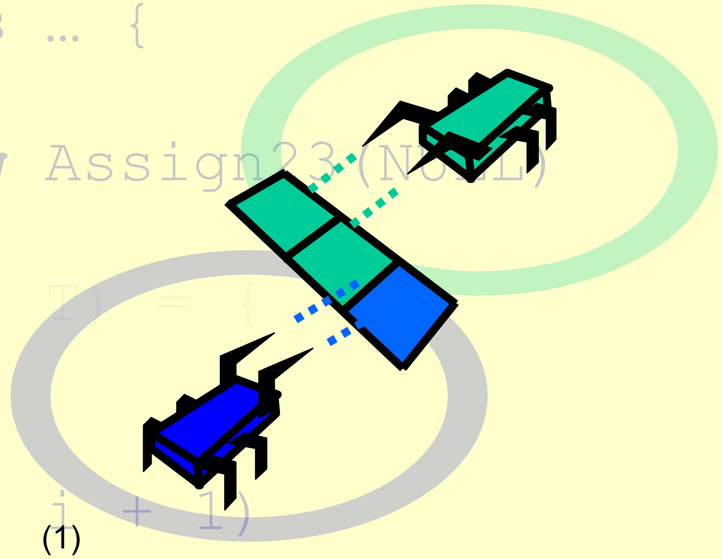
Multi-Consensus Code

```
class MultiConsensus extends … {
  private val NULL = -1
  private val assign23 = new Assign23(NULL)

  override def decide(value: T) = {
    propose(value)
    val i = ThreadID.get
    assign23.assign(i, i, i, i + 1)
    val other = assign23.read((i + 2) % 3)
    if (other == NULL || other == assign23.read(1)) {
      proposed(i).get()      // I win
    } else {
      proposed(1 - i).get() // I lose
    }
  }}
```

OK, I win.

Multi-Consensus Code

```
class MultiConsensus extends … {
  private val NULL = -1
  private val assign23 = new Assign23(NULL)

  override def decide(value: T) = {
    propose(value)
    val i = ThreadID.get
    assign23.assign(i, i, i, j + 1)
    val other = assign23.read((i + 2) % 3)
    if (other == NULL || other == assign23.read(1)) {
      proposed(i).get()      // I win
    } else {
      proposed(1 - i).get()  // I lose
    }
  }}
```



Other thread moved first, so I lose

Demo

- Testing multi-consensus

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

We need better concurrent primitives!

Read-Modify-Write Objects

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

Read-Modify-Write

```
class RMWRegister(private val init: Int) {
  private var value: Int = init

  def getAndMumble() = this.synchronized {
    val prior = value
    value = mumble(value)
    prior
  }

}
```

Read-Modify-Write

```
class RMWRegister(private val init: Int) {
  private var value: Int = init

  def getAndMumble() = this.synchronized {
    val prior = value
    value = mumble(value)
    prior
  }

}
```

Return prior value

Read-Modify-Write

```
class RMWRegister(private val init: Int) {
  private var value: Int = init

  def getAndMumble() = this.synchronized {
    val prior = value
    value = mumble(value)
    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

```
class RMWRegister(private val init: Int) {
  private var value: Int = init

  def read: Int = this.synchronized {
    val prior = value
    value = value
    prior
  }
}
```

Example: Read

```
class RMWRegister(private val init: Int) {
  private var value: Int = init

  def read: Int = this.synchronized {
    val prior = value
    value = value
    prior
  }
}
```

apply f(x)=x, the identity function

Example: getAndSet

```
class RMWRegister(private val init: Int) {
  private var value: Int = init

  def getAndSet(v: Int): Int =
    this.synchronized {
      val prior = value
      value = v
      prior
   }
}
```

Example: getAndSet (swap)

```
class RMWRegister(private val init: Int) {
  private var value: Int = init

  def getAndSet(v: Int): Int =
    this.synchronized {
      val prior = value
      value = v
      prior
    }
}
```

f(x)=v is constant

getAndIncrement

```
class RMWRegister(private val init: Int) {
  private var value: Int = init

  def getAndIncrement: Int =
    this.synchronized {
      val prior = value
      value = value + 1
      prior
   }
}
```

getAndIncrement

```
class RMWRegister(private val init: Int) {
  private var value: Int = init

  def getAndIncrement: Int =
    this.synchronized {
      val prior = value
      value = value + 1
      prior
    }
}
```

f(x) = x+1

getAndAdd

```
class RMWRegister(private val init: Int) {
  private var value: Int = init

  def getAndAdd(a: Int): Int =
    this.synchronized {
      val prior = value
      value = value + a
      prior
   }
}
```

Example: getAndAdd

```
class RMWRegister(private val init: Int) {
  private var value: Int = init

  def getAndAdd(a: Int): Int =
    this.synchronized {
      val prior = value
      value = value + a
      prior
    }
}
```

f(x) = x+a

compareAndSet

```
class RMWRegister(private val init: Int) {
  private var value: Int = init

  def compareAndSet(expected: Int, update: Int) =
    this.synchronized {
      if (value == expected) {
        value = update
        true
      } else {
        false
      }
    }
}
```

compareAndSet

```
class RMWRegister(private val init: Int) {
  private var value: Int = init

  def compareAndSet(expected: Int, update: Int) =
    this.synchronized {
      if (value == expected) {
        value = update
        true
      } else {
        false
      }
    }
}
```

If value is as expected, …

compareAndSet

```
class RMWRegister(private val init: Int) {
  private var value: Int = init

  def compareAndSet(expected: Int, update: Int) =
    this.synchronized {
      if (value == expected) {
        value = update
        true
      } else {
        false
      }
    }
}
```

… replace it

compareAndSet

```
class RMWRegister(private val init: Int) {
  private var value: Int = init

  def compareAndSet(expected: Int, update: Int) =
    this.synchronized {
      if (value == expected) {
        value = update
        true
      } else {
        false
      }
    }
}
```

Report success

compareAndSet

```
class RMWRegister(private val init: Int) {
  private var value: Int = init

  def compareAndSet(expected: Int, update: Int) =
    this.synchronized {
      if (value == expected) {
        value = update
        true
      } else {
        false
      }
    }
}
```

Otherwise report failure

Read-Modify-Write

```
class RMWRegister(private val init: Int) {
  private var value: Int = init

  def getAndMumble() = this.synchronized {
    val prior = value
    value = mumble(value)
    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

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

```
class RMWRegister(private val init: Int) {
  private var value: Int = init
  def compareAndSet(expected: Int, update: Int): Boolean =
    this.synchronized {
      if (value == expected) {
        value = update
        true
      } else {
        false
      }
    }
}
```

compareAndSet

```
class RMWRegister(private val init: Int) {
  private var value: Int = init
  def compareAndSet(expected: Int, update: Int): Boolean =
    this.synchronized {
      if (value == expected) {
        value = update
        true
      } else
        false
      }
    }
}
```

replace value if it's what we expected, …

compareAndSet Has ∞ Consensus Number

```
class CASConsensus extends ConsensusProtocol[Int] {
  private val FIRST = -1
  private val r = new RMWRegister(FIRST)

  override def decide(value: Int) = {
    propose(value)
    val i = ThreadID.get
    if (r.compareAndSet(FIRST, i)) {
      proposed(i).get() // I won
    } else {
      proposed(r.read).get()
    }
  }
}
```

compareAndSet Has ∞ Consensus Number

```
class CASConsensus extends ConsensusProtocol[Int] {
  private val FIRST = -1
  private val r = new RMWRegister(FIRST)

  override def decide(value: Int) = {
    propose(value)
    val i = ThreadID.get
    if (r.compareAndSet(FIRST, i)) {
      proposed(i).get() // I won
    } else {
      proposed(r.read).get()
    }
  }
}
```

Initialized to -1

compareAndSet Has ∞ Consensus Number

```
class CASConsensus extends ConsensusProtocol[Int] {
  private val FIRST = -1
  private val r = new RMWRegister(FIRST)

  override def decide(value: Int) = {
    propose(value)
    val i = ThreadID.get
    if (r.compareAndSet(FIRST, i)) {
      proposed(i).get() // I won
    } else {
      proposed(r.read).get()
    }
  }
}
```

Try to swap in my id

compareAndSet Has ∞ Consensus Number

```
class CASConsensus extends ConsensusProtocol[Int] {
  private val FIRST = -1
  private val r = new RMWRegister(FIRST)

  override def decide(value: Int= {
    propose(value)
    val i = ThreadID.get
    if (r.compareAndSet(FIRST, i)) {
      proposed(i).get() // I won
    } else {
      proposed(r.read).get()
    }
  }
}
```

Try to swap in my id

compareAndSet Has ∞ Consensus Number

```
class CASConsensus extends ConsensusProtocol[Int] {
  private val FIRST = -1
  private val r = new RMWRegister(FIRST)

  override def decide(value: Int) = {
    propose(value)
    val i = ThreadID.get
    if (r.compareAndSet(FIRST, i)) {
      proposed(i).get() // i won
    } else {
      proposed(r.read).get()
    }
  }
}
```

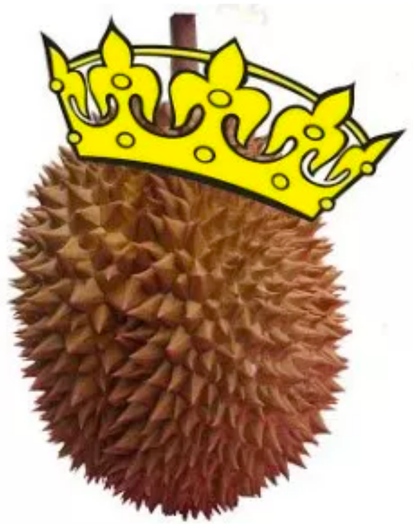
Decide winner's preference

Demo

- CAS-based consensus

The Consensus Hierarchy

1	Read/Write Registers, Snapshots…
2	getAndSet, getAndIncrement, …
.	
.	
.	
∞	compareAndSet,…



Summary

- Wait-free mutual exclusion requires synchronization
- WF synchronization = consensus (deciding who goes first)
- RMW operations allow to implement consensus for an *arbitrary* number of threads
  - Supporting RMW = supporting proper concurrency

Next week:

- Implementing practical locks via RMW operations
- Monitors and Blocking Synchronisation