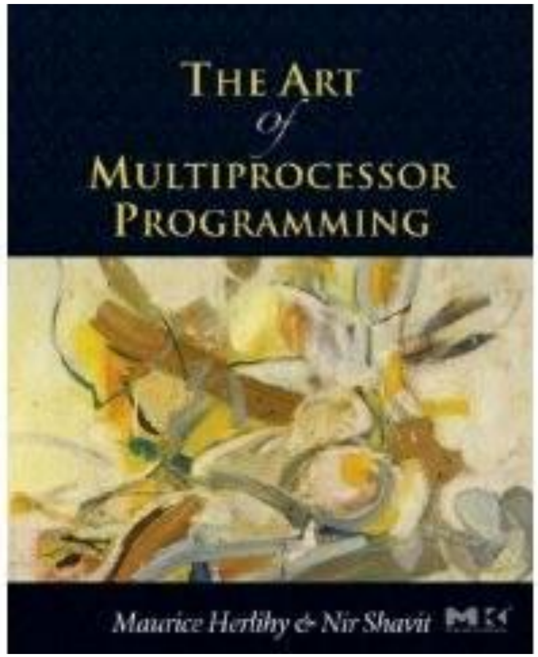
Department of Computer Science

COS 226

Chapter 5

The Relative Power of Primitive Synchronization Operations

Acknowledgement



- Some of the slides are taken from the companion slides for "The Art of Multiprocessor Programming" by Maurice Herlihy & Nir Shavit

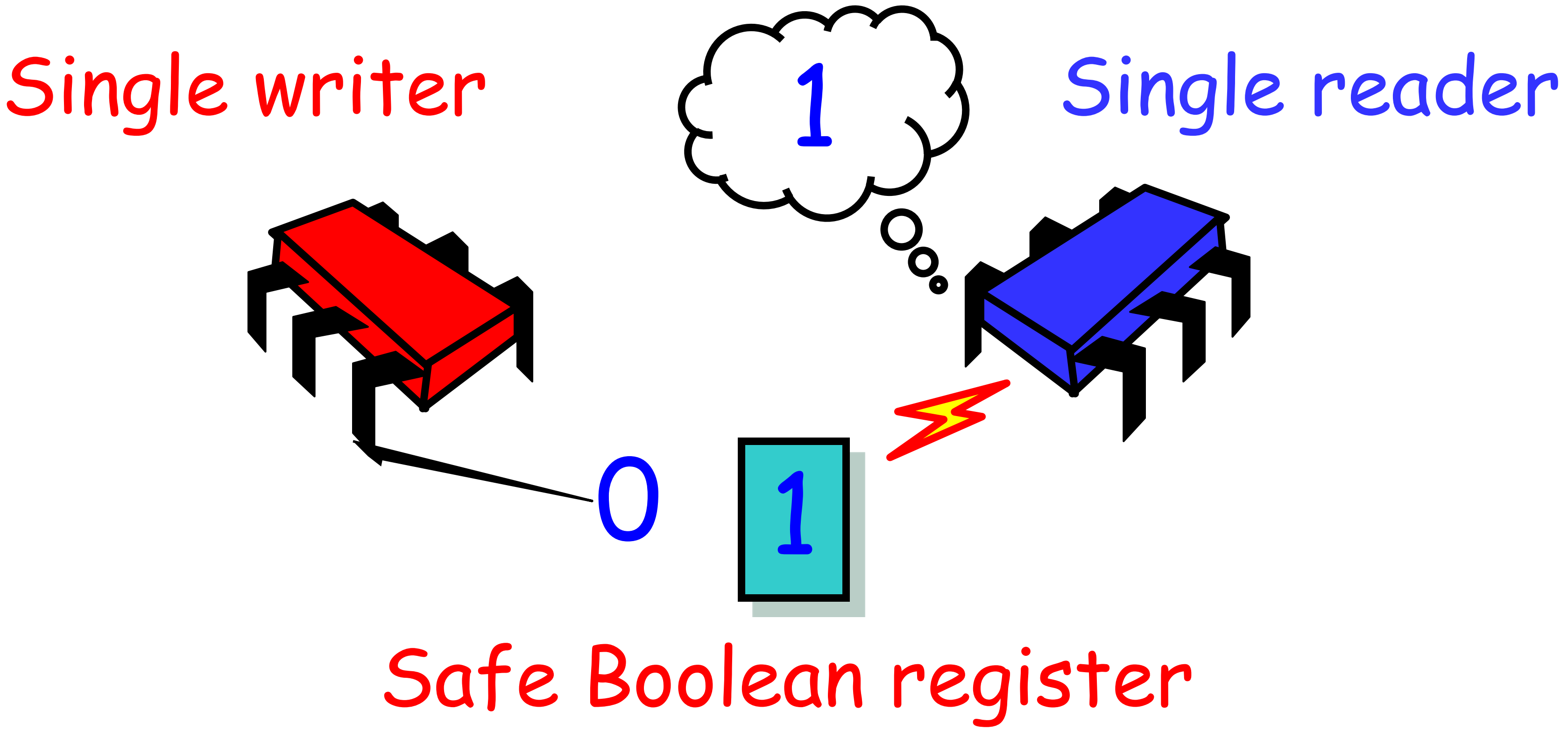
Background

- Which atomic instructions would you include when designing a new multiprocessor?
- Supporting them all would be inefficient

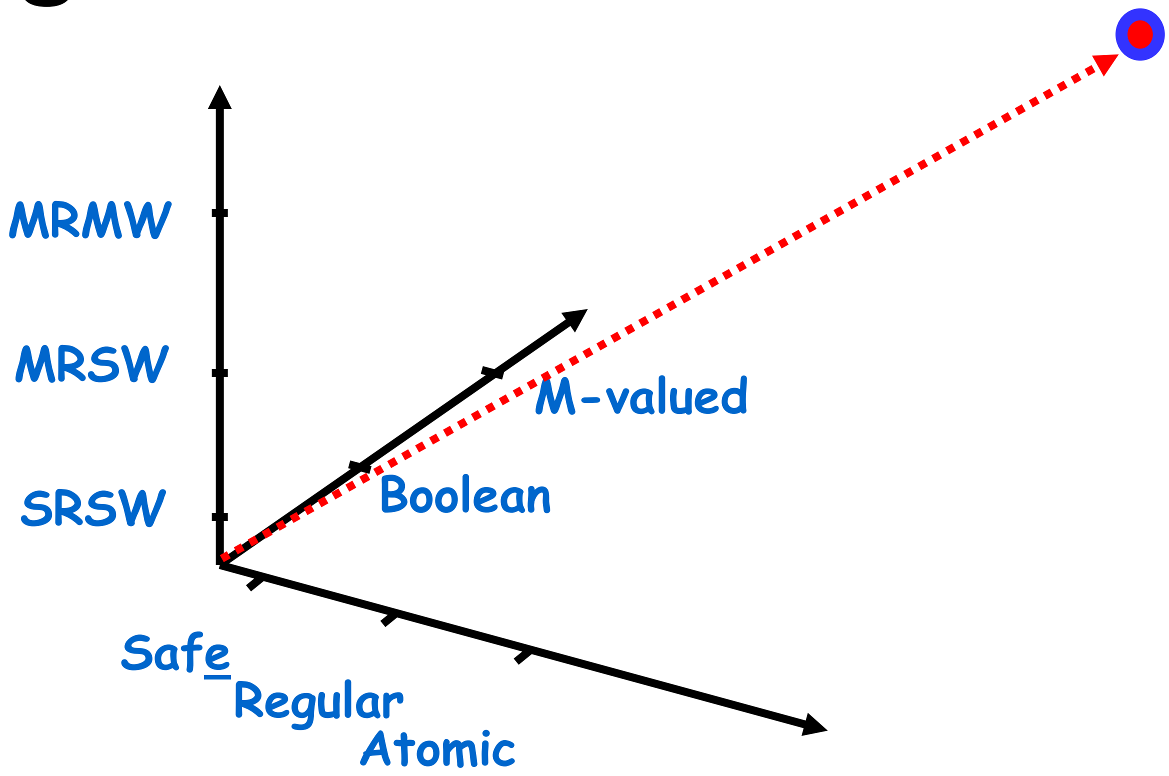
Wait-Free Implementation

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

From Weakest Register

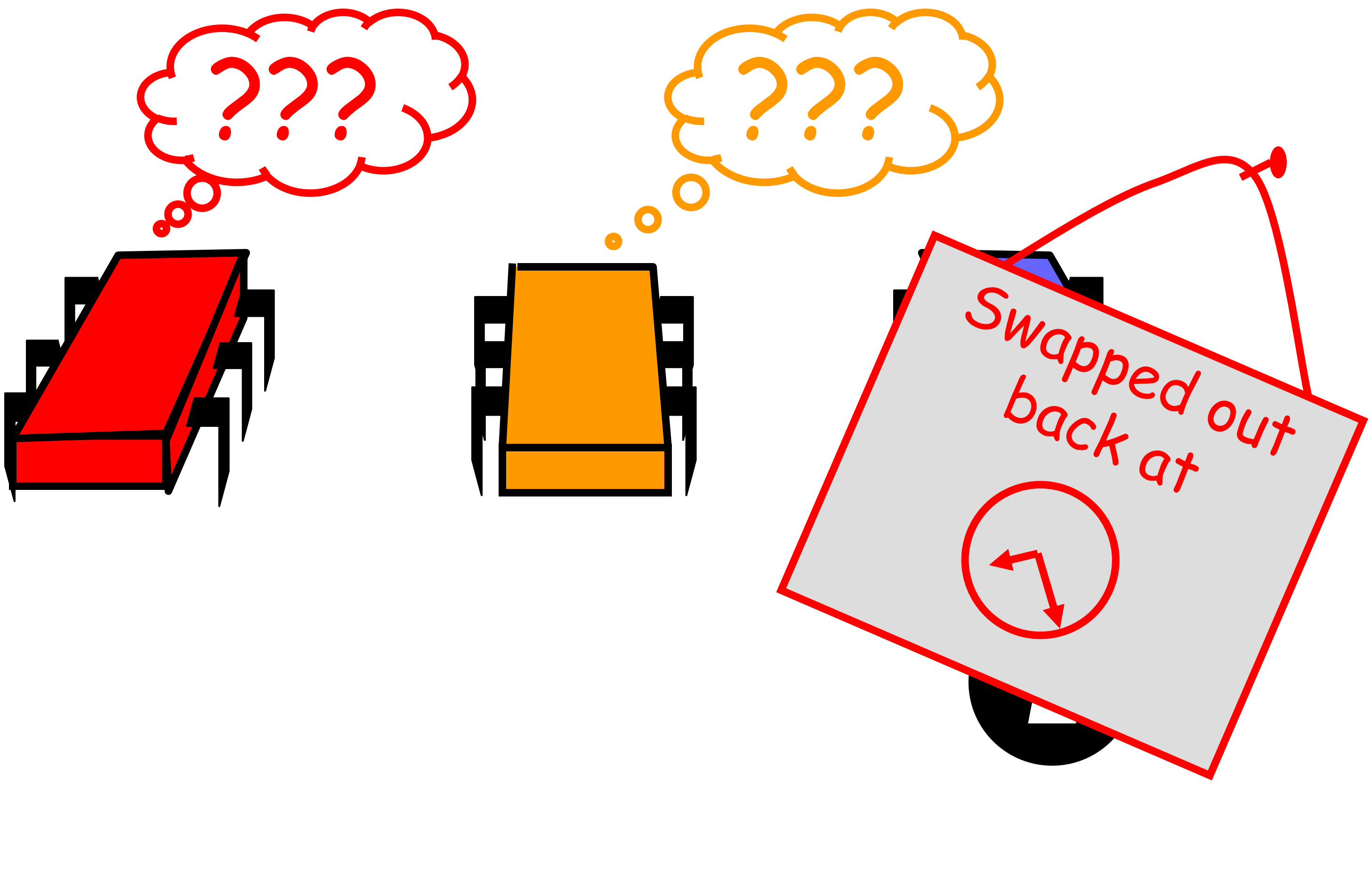


All the way to a Wait-free Implementation of Atomic Registers

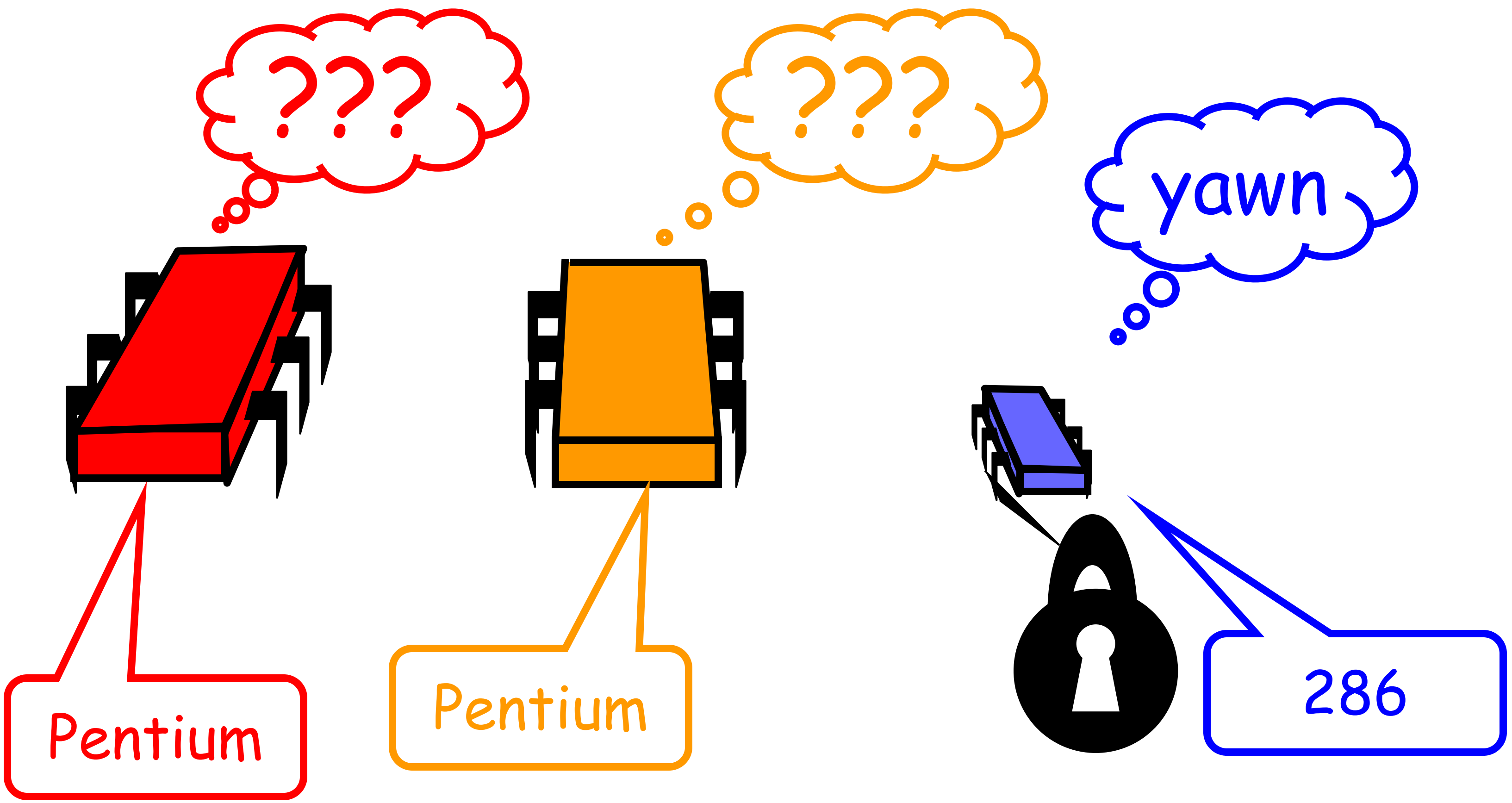


Why is Mutual Exclusion so wrong?

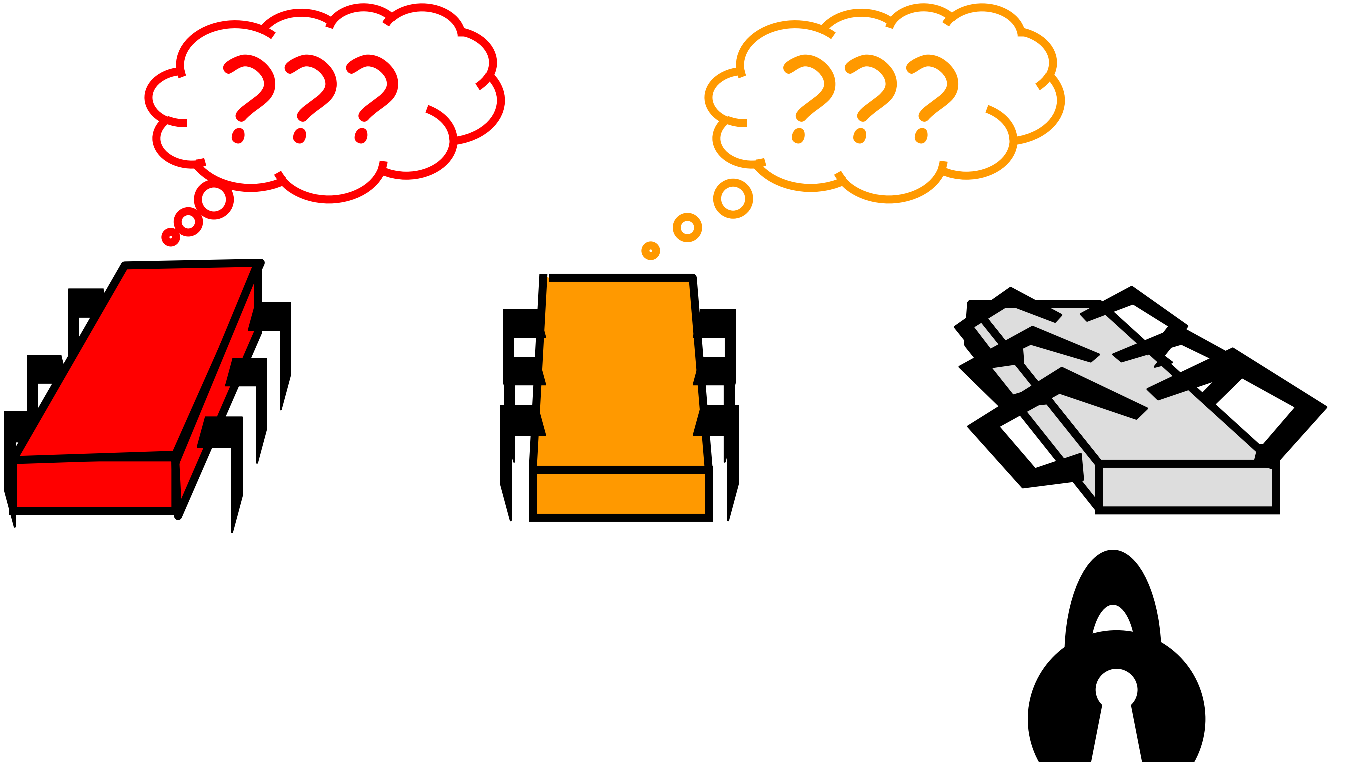
Asynchronous Interrupts



Heterogeneous Processors



Fault-tolerance



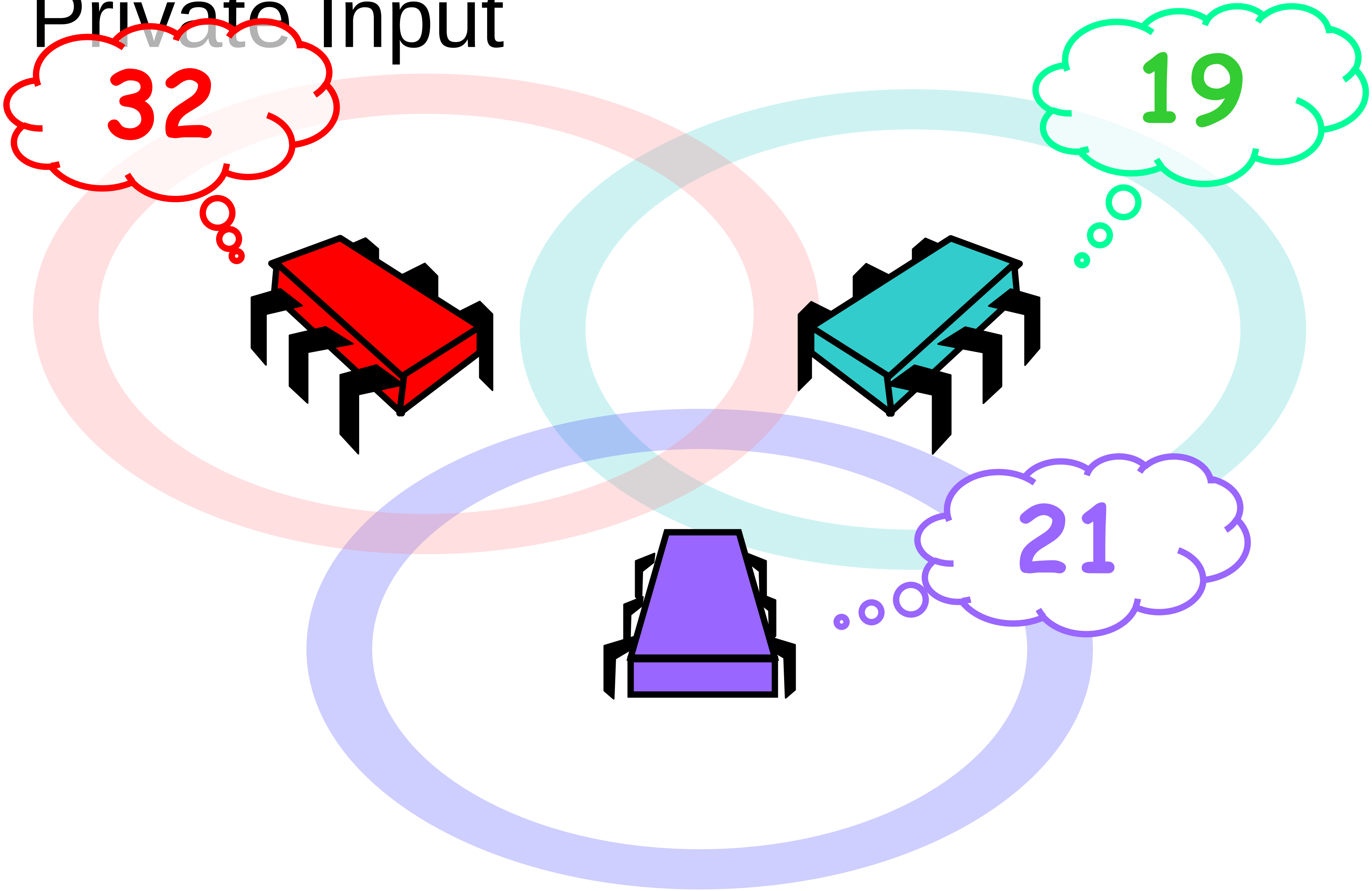
Goal

- To identify a set of primitive synchronization operations powerful enough to solve synchronization problems likely to arise in practise
- To do this we need a way to evaluate the *power* of various synchronization primitives
  - What can they solve and how efficiently?

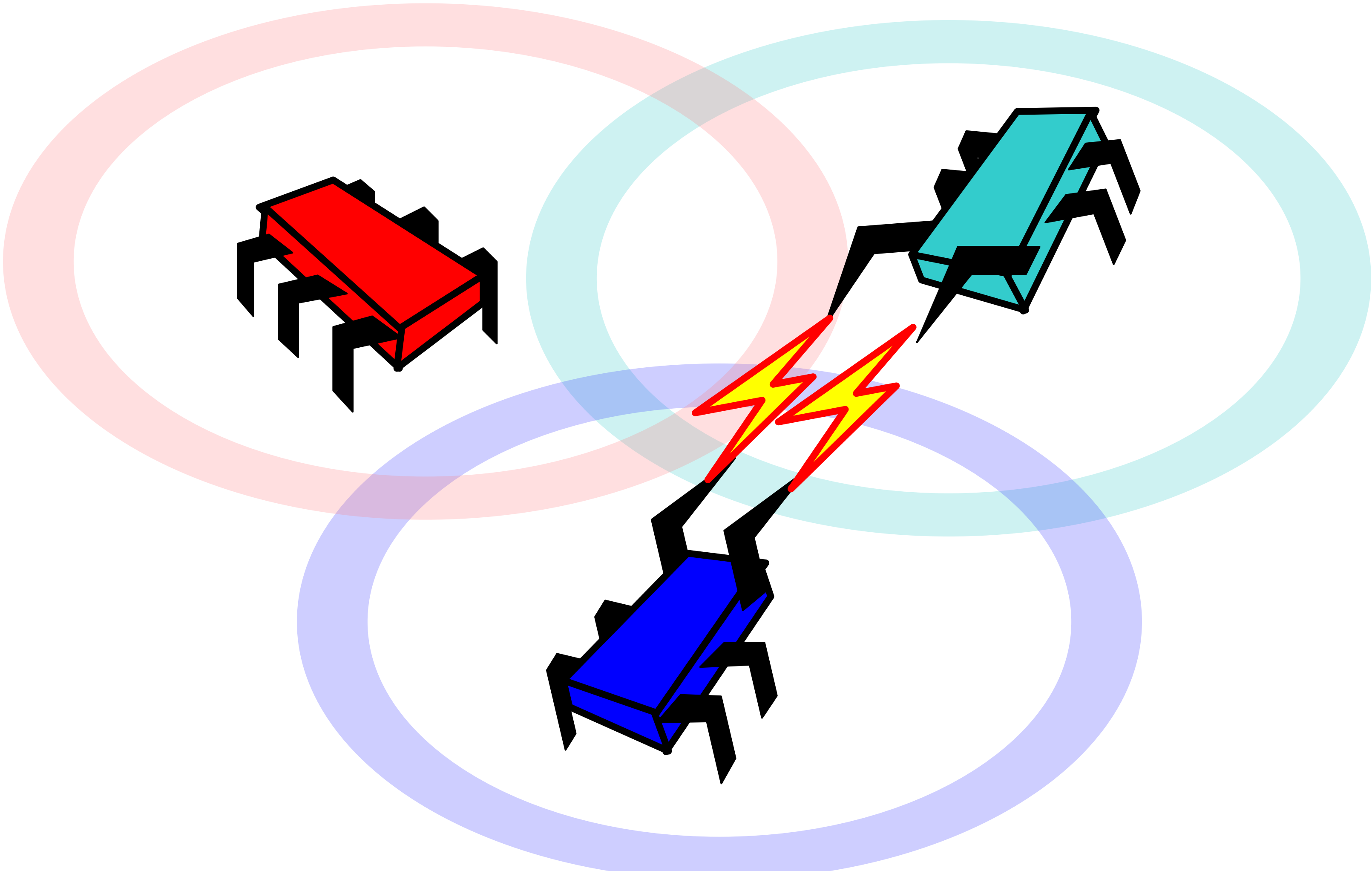
Consensus

- Not all synchronization instructions are created equal
- A hierarchy of synchronization primitives exist
- We are going to use the principle of **consensus** to determine the *power* of synchronization primitives

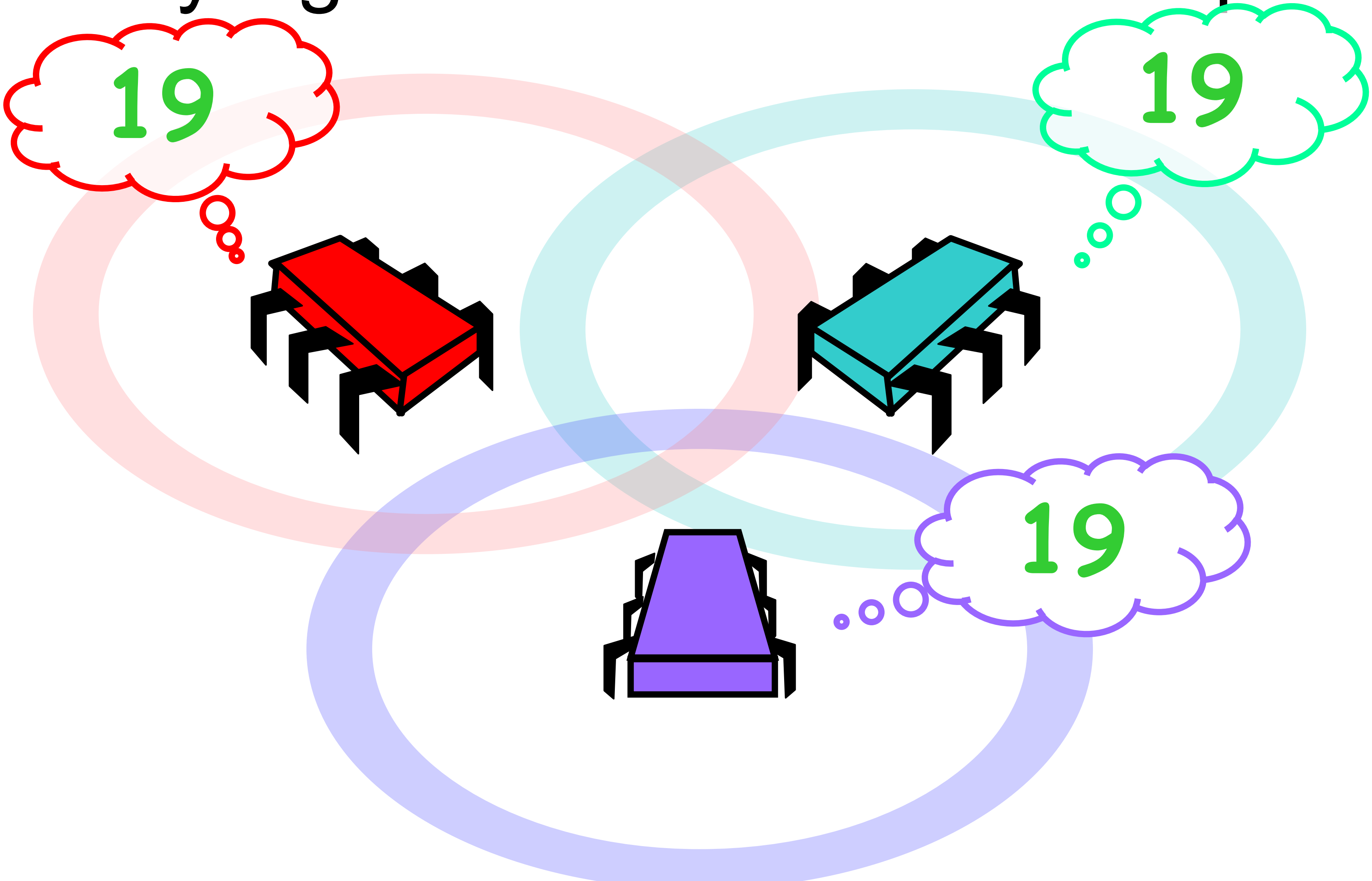
Consensus: Each Thread has a Private Input



They Communicate



They Agree on One Thread's Input

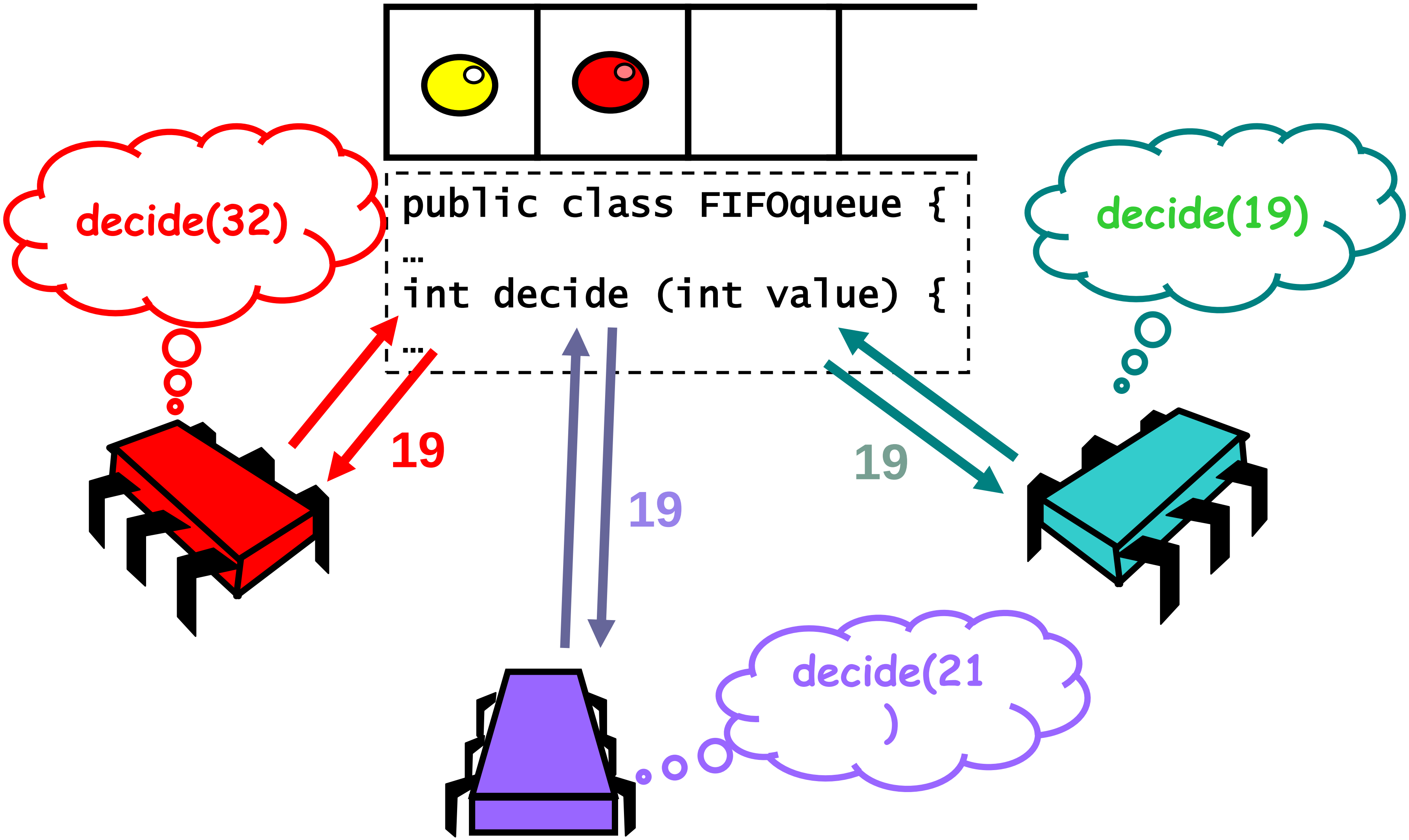


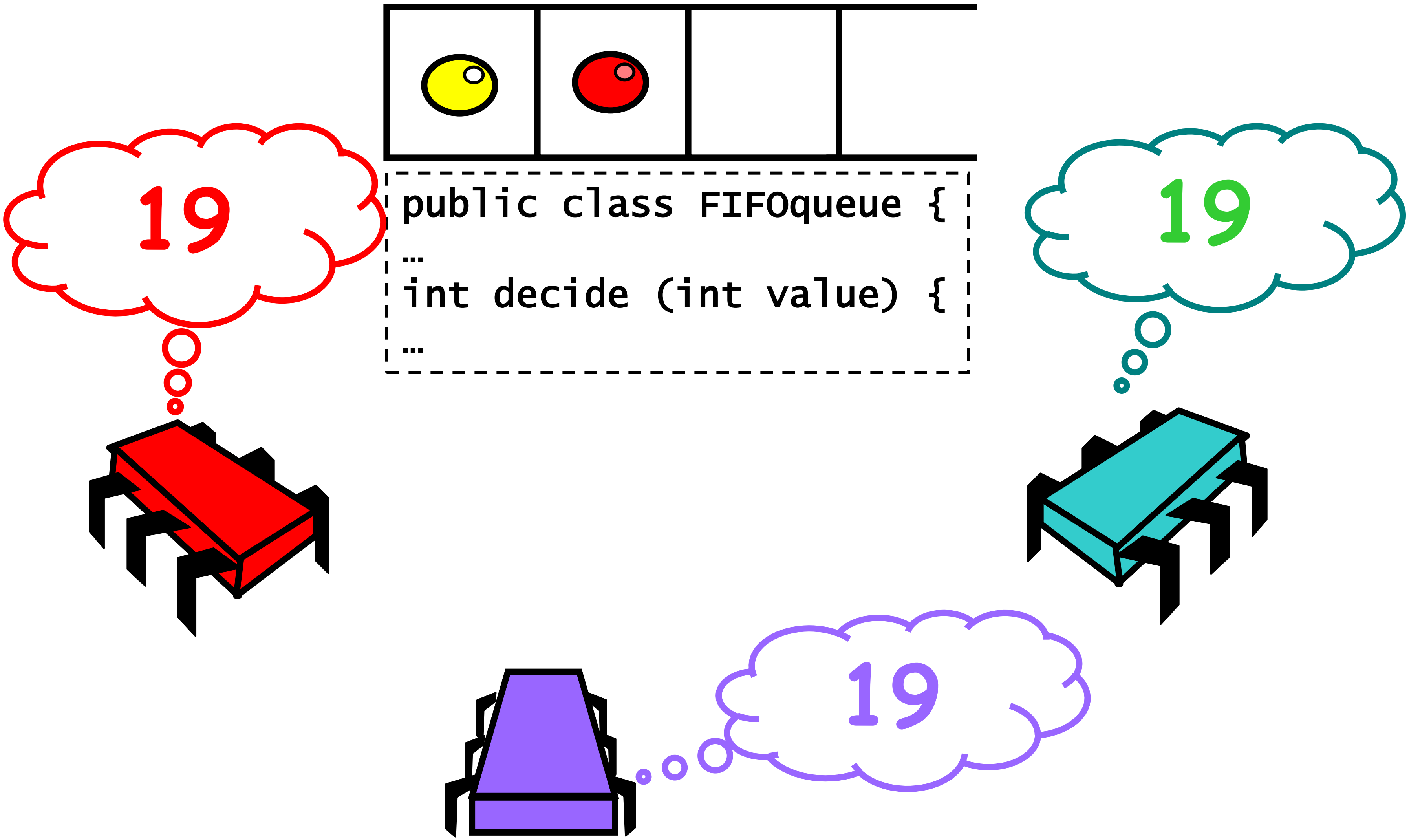
Consensus

- Basic idea:
  - Each class in hierarchy has an associated consensus number
    - Maximum number of threads for which objects can solve consensus

Formally: Consensus

- A consensus object has a decide() method
- Each thread calls the decide() method with its input at most once
- decide() returns a value with the following conditions:
  - Consistent:
    - all threads decide the same value
  - Valid:
    - the common decision value is some thread's input


public class FIFOqueue {
…
int decide (int value) {
…
decide(32)
19
decide(19)
19
19
decide(21
)




```
public class FIFOqueue {
…
int decide (int value) {
…
```

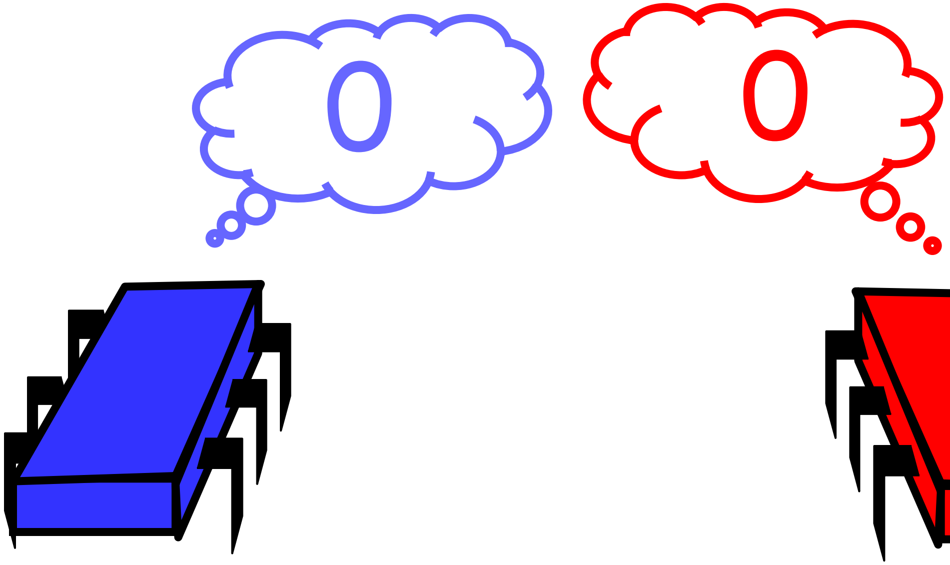
Consensus

- We are now going to look at different concurrent object classes and see whether they solve consensus

Consensus

- Can consensus be reached using atomic registers?

Both Inputs 0

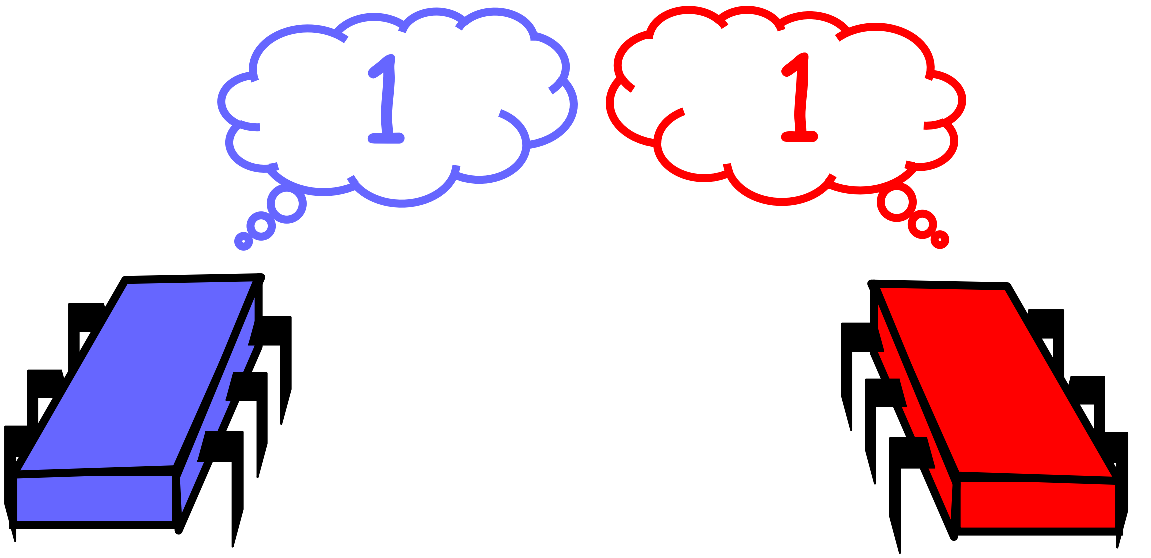


Univalent: all executions must decide 0

Both Inputs 0

Including this solo execution by A

Both Inputs 1

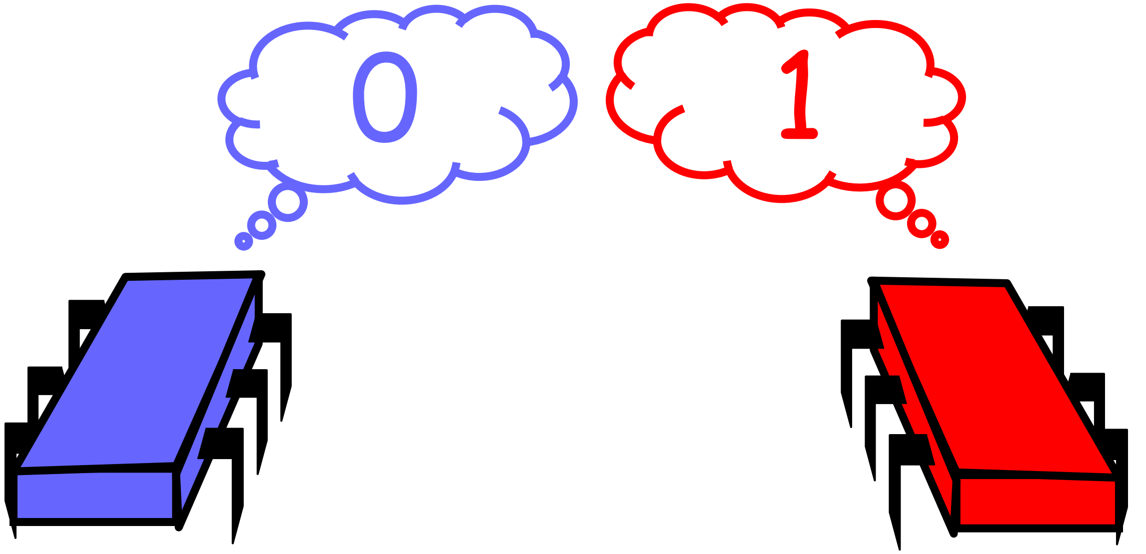


Univalent: all executions must decide 1

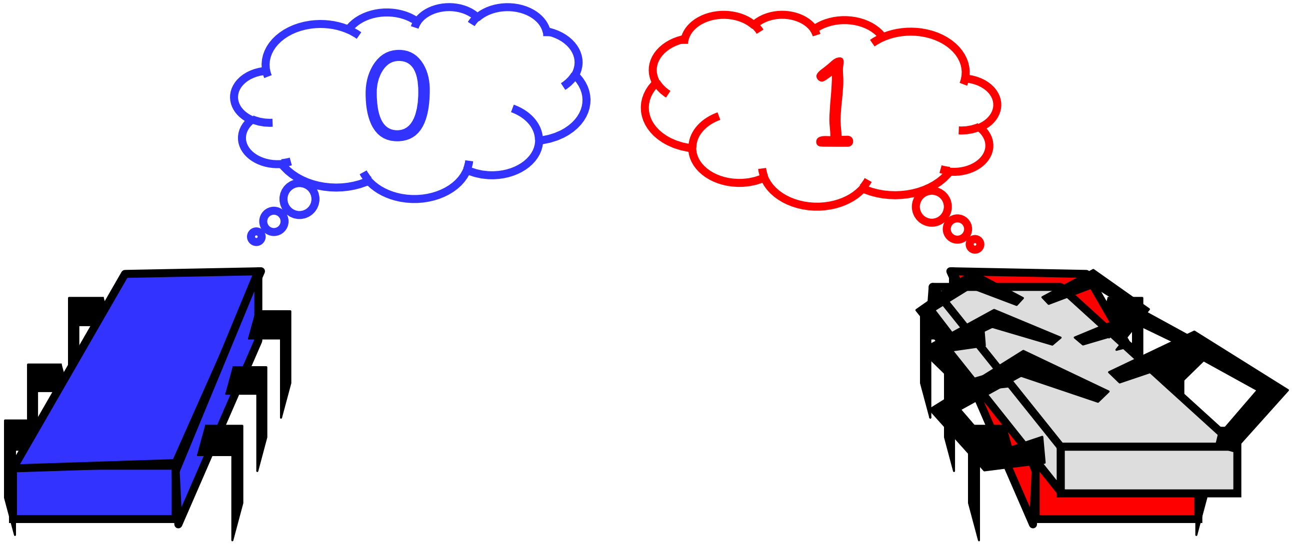
Both Inputs 1

Including this solo execution by B

What if inputs differ?

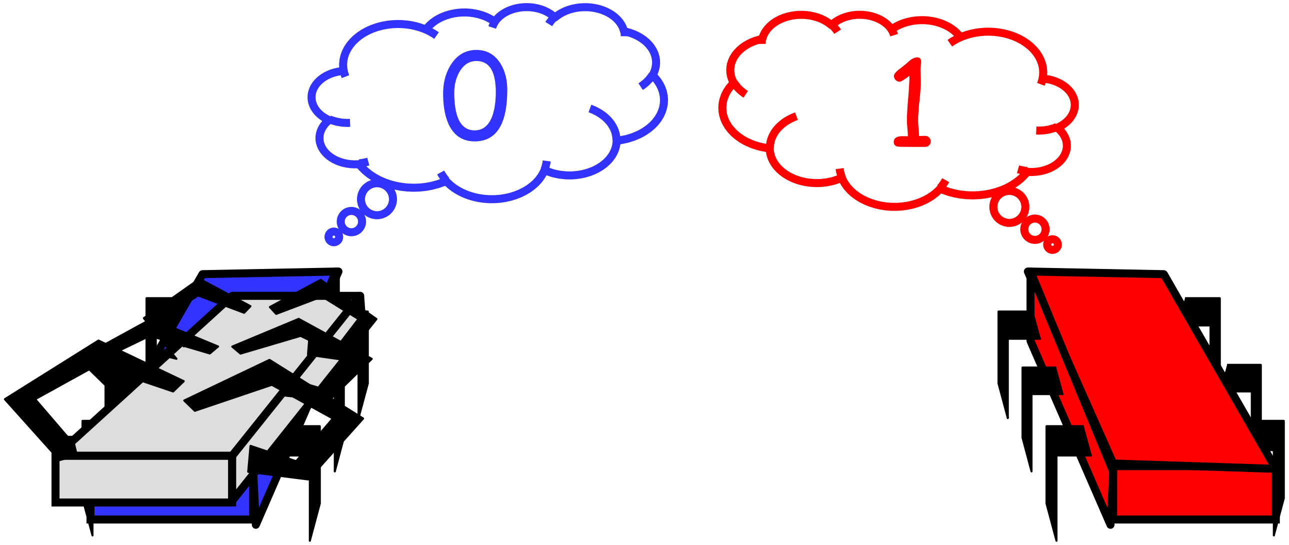


The Possible Executions



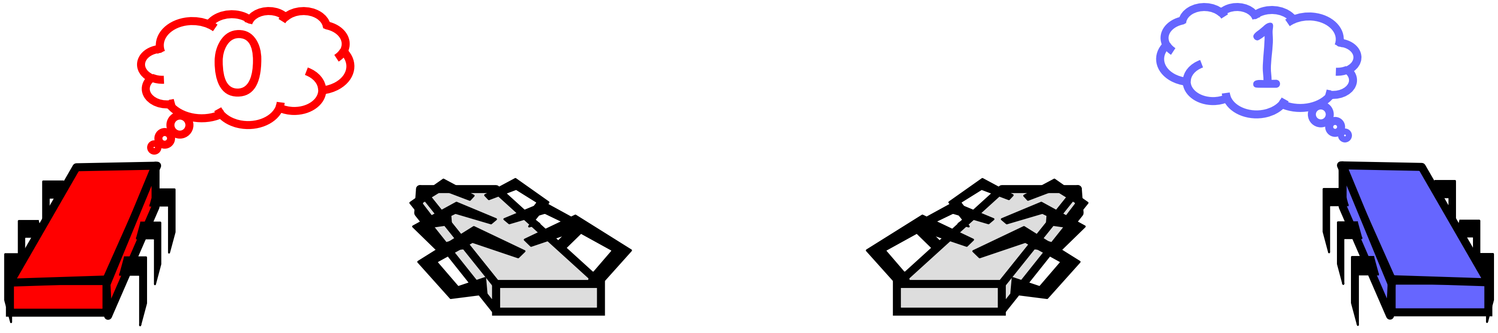
Include the solo execution by A
that decides 0

The Possible Executions



Also include the solo execution by B which decides 1

Possible Executions Include



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

Atomic Registers

- Method calls:
  - One of the threads reads from the register
  - Both threads write to separate registers, or
  - Both threads write to the same registers
- The proofs show that in each case two threads cannot reach consensus on two values using atomic registers

Theorem 5.2.1

- Atomic registers have consensus number 1.

- Consensus numbers:
  - The consensus number of a class is the largest number of threads that can solve consensus using that class.
  - Consensus number 1 means only sequential.

Consensus Object

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

Generic Consensus Protocol

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

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

 abstract public T d
}
```

Each thread's proposed value

Generic Consensus Protocol

```
abstract class ConsensusProtocol<T>
   implements Consensus {
 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

```
ab

  p

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

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

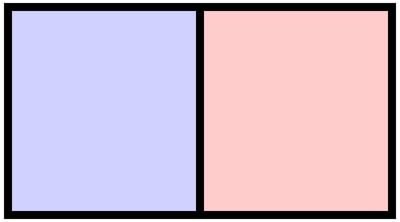
Decide a value: abstract method means subclass does the heavy lifting (real work)

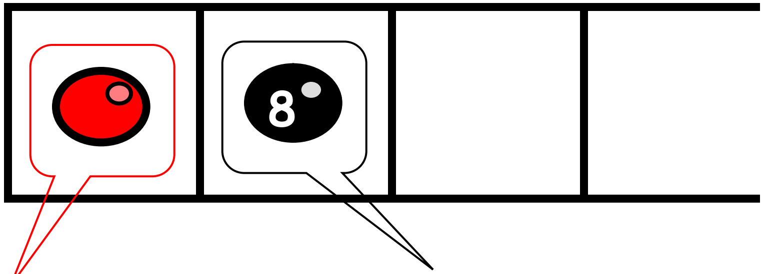
Can a FIFO Queue Implement Consensus?

FIFO Queue

- Let’s start with 2-threads…

FIFO Consensus

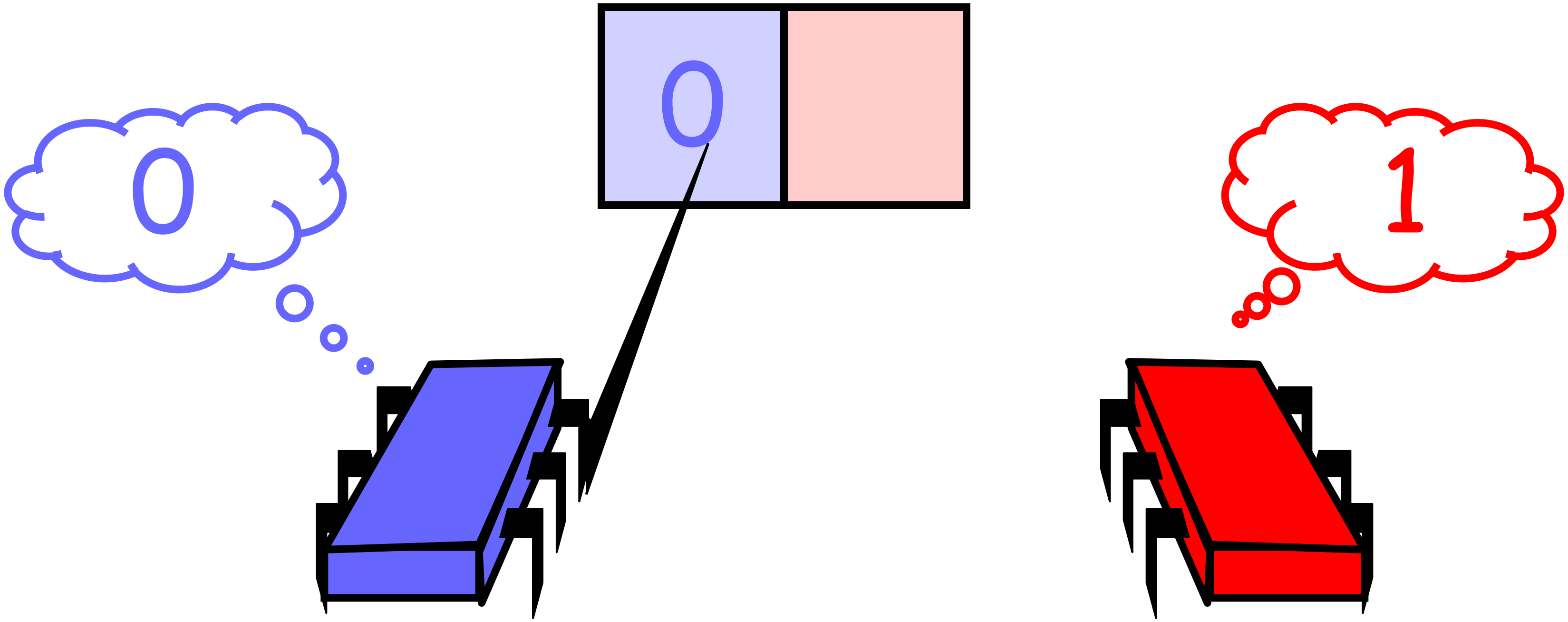

proposed array



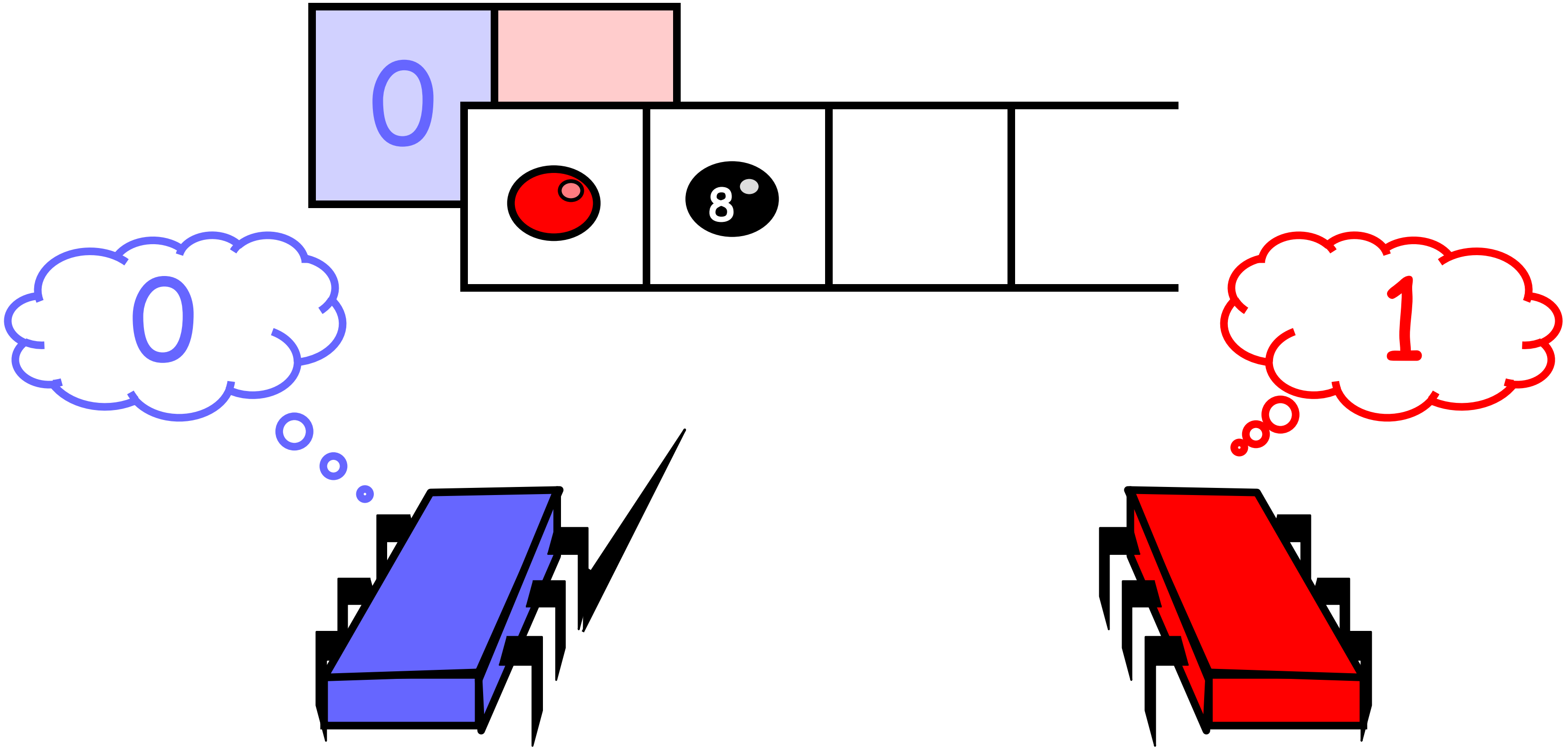
FIFO Queue with red and black balls

Protocol: Write Value to Array

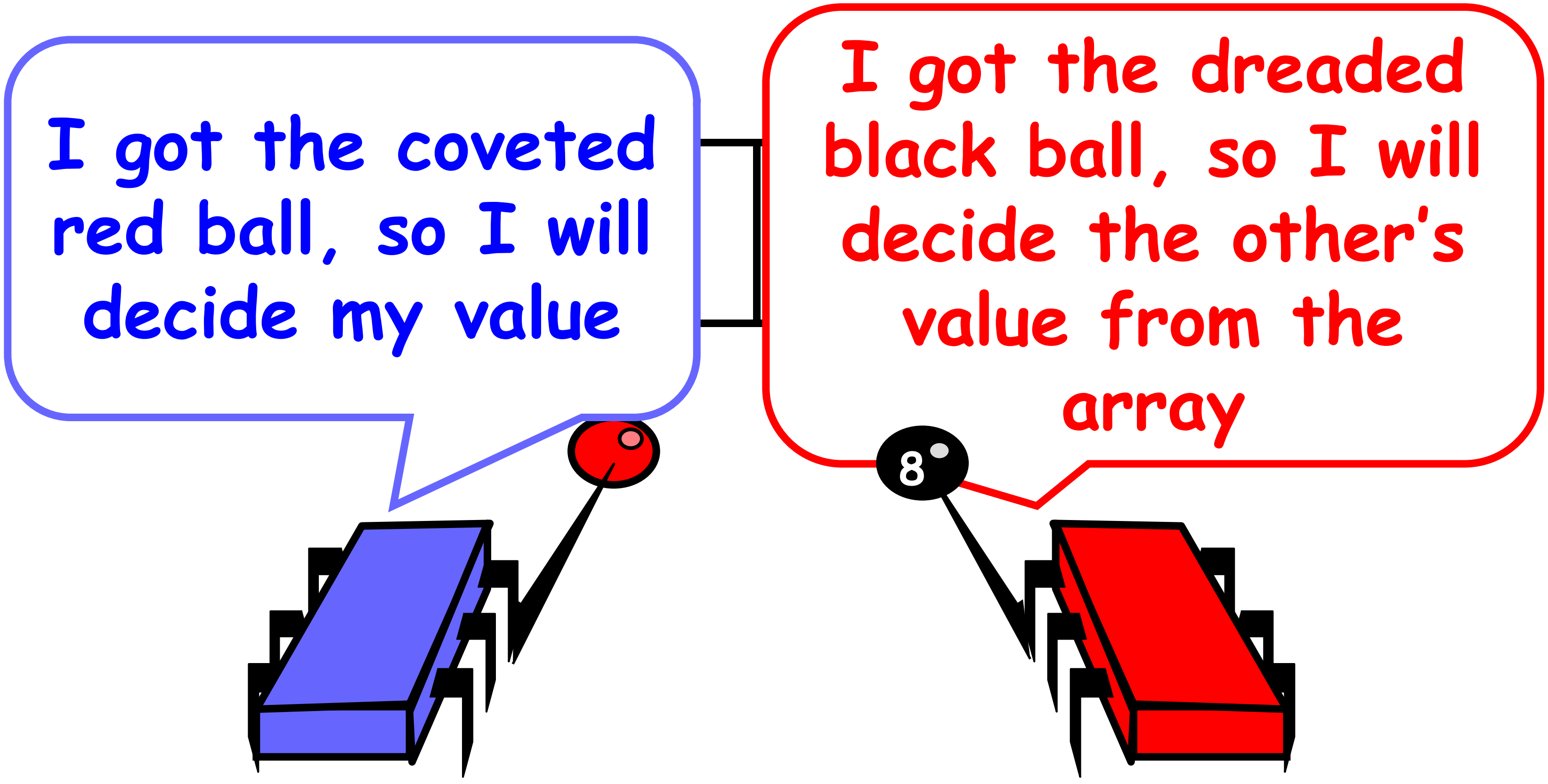
2-Threads attempt to enqueue a value at the same time



Protocol: Take Next Item from Queue



Protocol: Take Next Item from Queue

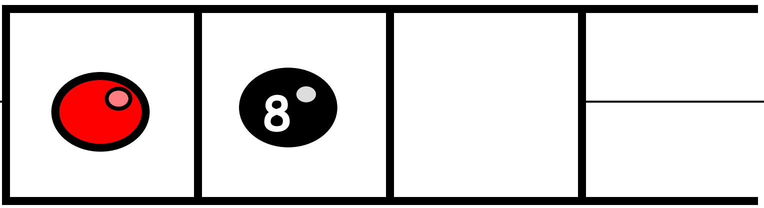


Consensus Using FIFO Queue

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

Initialize Queue

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



Who Won?

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

Who Won?

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

FIFO Queues

- Although FIFO queues solve two-thread consensus, they cannot solve 3-thread consensus.

Theorem 5.4.1

- FIFO queues have consensus number 2.

Read-modify-write operations

- Many synchronization operations can be described as read-modify-write (RMW) operations
- In object form: read-modify-write registers

RMW Methods

- A method is an RMW for the function set *F* if it atomically replaces register value v with *f*(v) for some *f* E *F* and returns v

RMW Methods

- From java.util.concurrent:
  - getAndSet(x):
    - Replaces current value with x and returns prior value
    - $f_v(x) = v$
  - getAndIncrement()
    - Atomically adds 1 to the current value and returns the old value
    - $f_v(x) = v + 1$

RMW Methods

- getAndAdd(k)
  - Atomically adds *k* to the current value and returns the prior value
  - $f_k(x) = x + k$
- get()
  - Returns the register's value

The Exception

- compareAndSet()
  - Takes 2 values – expected value *e* and update value *u*
  - If value is equal to *e*, it replaces it with *u* otherwise it remains unchanged
  - Returns a Boolean value to indicate whether value was changed

Read-Modify-Write

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

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

Read-Modify-Write

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

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

Return prior value

Read-Modify-Write

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

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

Apply function to current value

compareAndSet

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

compareAndSet

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

If value is what was expected, …

compareAndSet

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

… replace it

compareAndSet

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

Report success

compareAndSet

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

Otherwise report failure

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 of 2

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

Implementation

```
public class RMWConsensus
     extends ConsensusProtocol {
 private RMWRegister r = new
 RMWRegister(v);

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

Implementation

```
public class RMWConsensus
     extends ConsensusProtocol {
 private RMWRegister r = new
 RMWRegister(v);

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

Initialized to v

Implementation

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

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

Am I first?

Implementation

```
public class RMWConsensus
     extends ConsensusProtocol {
 private RMWRegister r = new
 RMWRegister(v);

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

Yes, return my input

Implementation

```
public class RMWConsensus
     extends ConsensusProtocol {
 private RMWRegister r = new
 RMWRegister(v);

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

No, return other's input

Common2 RMW Operations

- 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

- getAndSet()
  - Overwrite
- getAndAdd()
  - Commute
- getAndIncrement()
  - Commute

Common2 RMW Registers

- Theorem:
  - Any RMW register in Common2 has consensus number of 2.

compareAndSet()

- A register providing compareAndSet() and get() methods has an infinite consensus number

Implementation

```
public class CASConsensus
     extends ConsensusProtocol {
 private final int FIRST = -1;
 private AtomicInteger r = new
     AtomicInteger(FIRST);

 public Object decide(object value) {
  propose(value);
  int i = ThreadID.get();
  if (r.compareAndSet(FIRST, i))
   return proposed[i];
  else
   return proposed[j];
}}
```

Implementation

```
public class CASConsensus
     extends ConsensusProtocol {
 private final int FIRST = -1;
 private AtomicInteger r = new
     AtomicInteger(FIRST);

 public Object decide(object value) {
  propose(value);
  int i = ThreadID.get();
  if (r.compareAndSet(FIRST, i))
   return proposed[i];
  else
   return proposed[j];
}}
```

Use Atomic Register

Implementation

```
public class CASConsensus
     extends ConsensusProtocol {
 private final int FIRST = -1;
 private AtomicInteger r = new
      AtomicInteger(FIRST);

 public Object decide(object value) {
  propose(value);
  int i = ThreadID.get();
  if (r.compareAndSet(FIRST, i))
   return proposed[i];
  else
   return proposed[j];
}}
```

Add value to proposed array

Implementation

```
public class CASConsensus
     extends ConsensusProtocol {
 private final int FIRST = -1;
 private AtomicInteger r = new
      AtomicInteger(FIRST);

 public Object decide(object value) {
  propose(value);
  int i = ThreadID.get();
  if (r.compareAndSet(FIRST, i))
   return proposed[i];
  else
   return proposed[j];
}}
```

If I am the first thread to access the register