The Relative Power of Synchronization Primitives

BROWN

Maurice Herlihy

CS176

Fall 2003

Wait-Free Implementation

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

Wait-Free Constructions

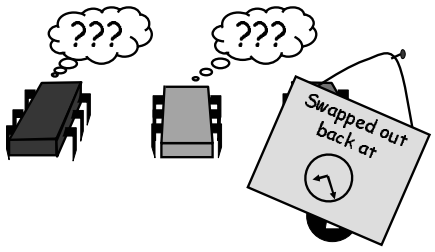
- Wait-free atomic registers
  - From safe registers
- Two-threaded FIFO queue
  - From atomic registers
  - And indirectly from safe registers

Rationale

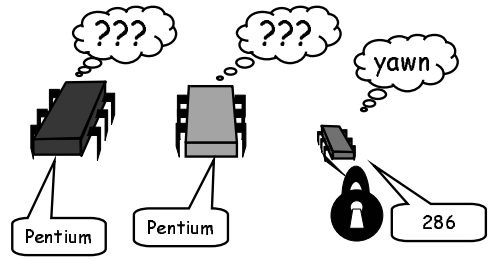
- We wanted atomic registers to implement mutual exclusion
- So we couldn't use mutual exclusion to implement atomic registers
- But wait, there's more!

Why is Mutual Exclusion so wrong?

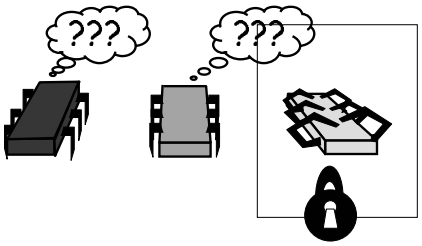
Asynchronous Interrupts



Heterogeneous **processors**



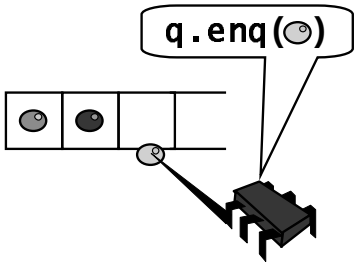
Fault-tolerance



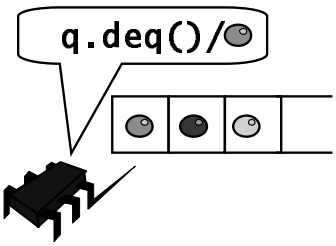
Basic Questions

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

FIFO Queue: Enqueue Method



FIFO Queue: Dequeue Method



Two-Thread Wait-Free Queue

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

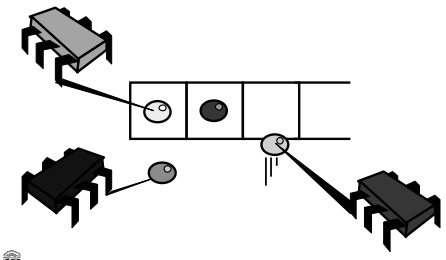
Put object in quue

Two-Thread Wait-Free Queue

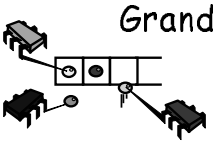
```
public class LockFreeQueue {
 int head = 0, tail = 0;
 Item[QSIZE] items;
 public void enq(Item x) {
  while (tail-head == QSIZE) {};
  items[tail % QSIZE] = x; tail++;
  }
 public Item deq() {
  while (tail == head) {}
  Item item = items[head % QSIZE];
  head++; return item;
}}
```

Increment tail counter

What About Multiple Dequeuers?



Grand Challenge



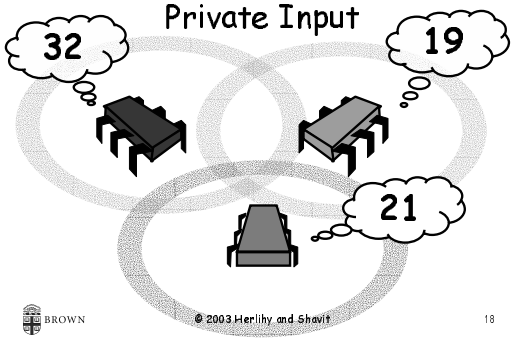
- Implement a FIFO queue
  - Wait-free
  - Linearizable
  - From atomic read-write registers
  - Multiple dequeuers

Only new aspect

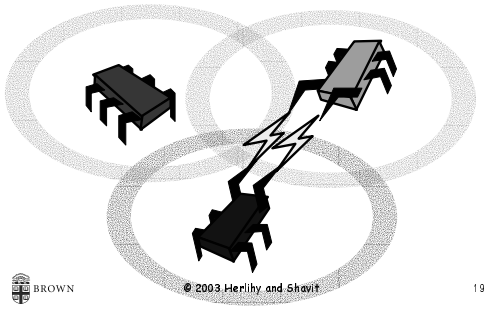
Consensus

- While you are ruminating on the grand challenge…
- We will give you another puzzle
  - Consensus
  - Pretty important …

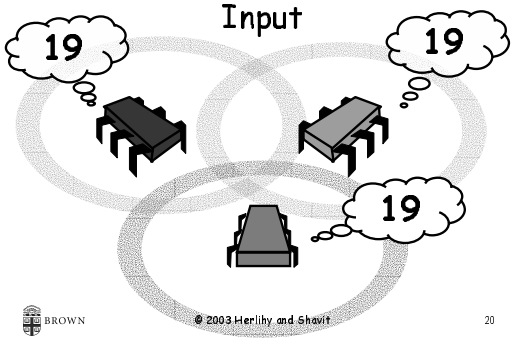
Consensus: Each Thread has a Private Input



They Communicate



They Agree on One Thread's Input



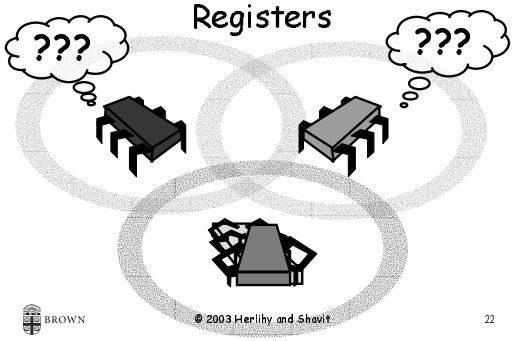
Formally: Consensus

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

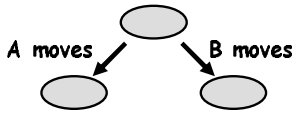
No Wait-Free Consensus using Registers



Proof Strategy

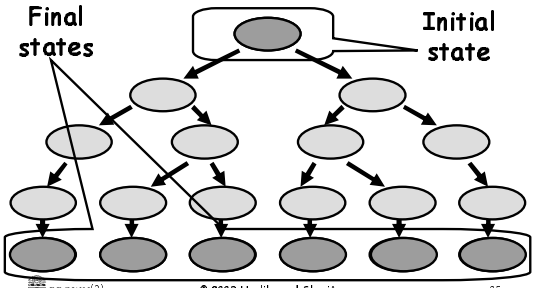
- Assume otherwise
- Reason about the properties of any such protocol
- Derive a contradiction
- Quod Erat Demonstrandum

Wait-Free Computation

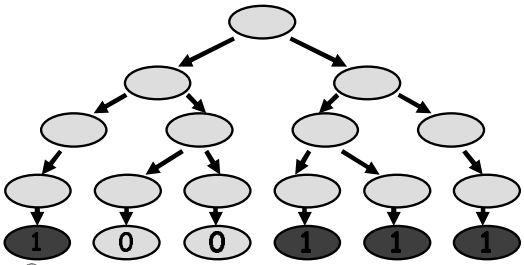


- Either A or B "moves"
- Moving means
  - Register read
  - Register write

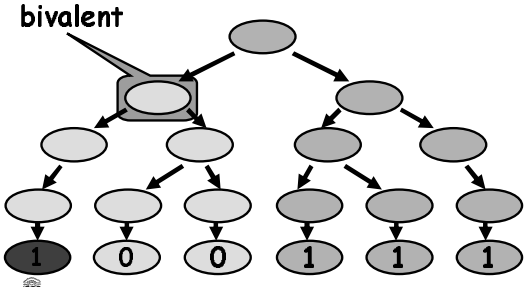
The Two-Move Tree



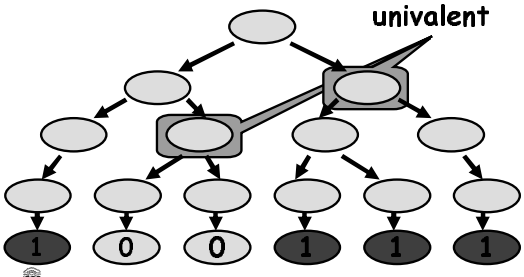
Decision Values



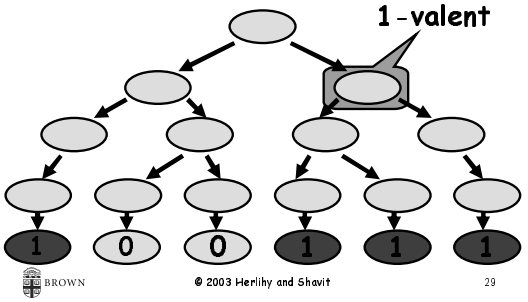
Bivalent: Both Possible



Univalent: Single Value Possible



x-valent: x Only Possible Decision



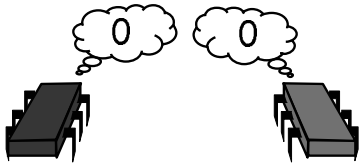
Summary

- Wait-free computation is a tree
- Bivalent system states
  - Outcome not fixed
- Univalent states
  - Outcome is fixed
  - May not be "known" yet
- 1-Valent and 0-Valent states

Claim

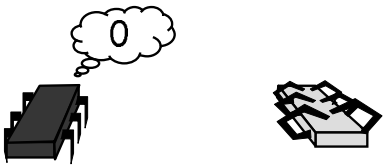
- Some initial state is bivalent
- Outcome depends on
  - Chance
  - Whim of the scheduler
- Multiprocessor gods do play dice …

Univalent Initial State



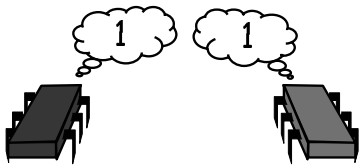
All executions must decide 0

Univalent Initial State



Including this solo execution by A

Univalent Initial State

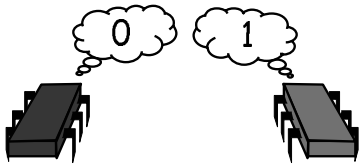


All executions must decide 1

Univalent Initial State

Including this solo execution by B

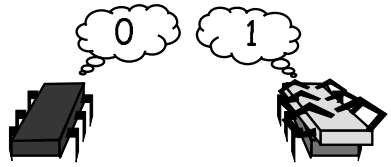
Univalent Initial State?



Imagine all executions deciding alike…

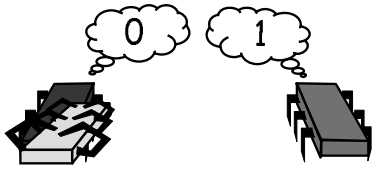
(cue in the John Lennon ..)

Univalent Initial State?



Including this solo execution by A which we know decides 0

Univalent Initial State?

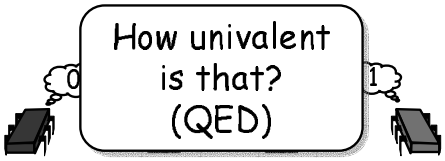


Including this solo execution by B which we know decides 1

Uh-Oh

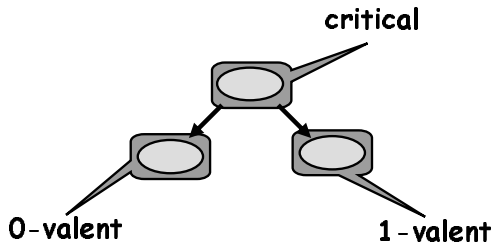
- Solo execution by A must decide 0
- Solo execution by B must decide 1

Uh-Oh

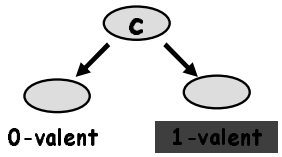


- Solo execution by A must decide 0
- Solo execution by B must decide 1

Critical States



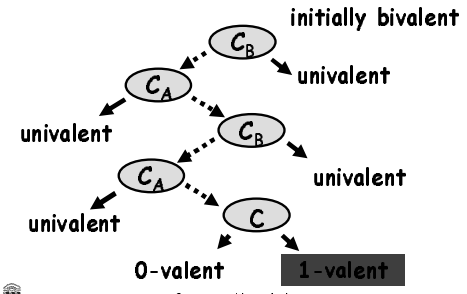
From a Critical State



If A goes first, protocol decides 0

If B goes first, protocol decides 1

Reaching Critical State



Critical States

- Starting from a bivalent initial state
- The protocol can reach a critical state
  - Otherwise we could stay bivalent forever
  - And the protocol is not wait-free

Model Dependency

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

What are the Threads Doing?

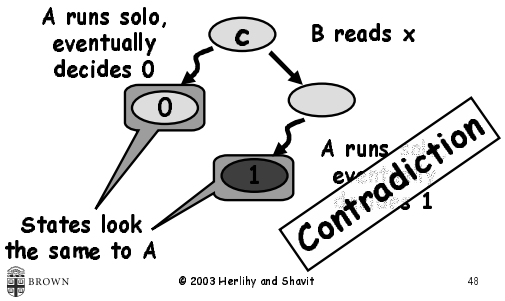
- Reads and/or writes
- To same/different registers

Possible Interactions

A reads x
A reads y

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

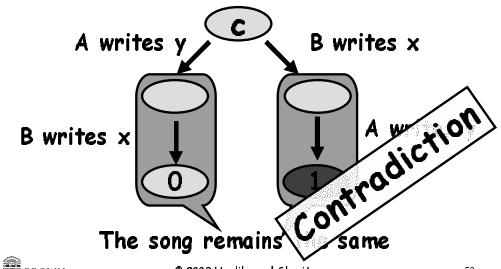
Reading Registers



Possible Interactions

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

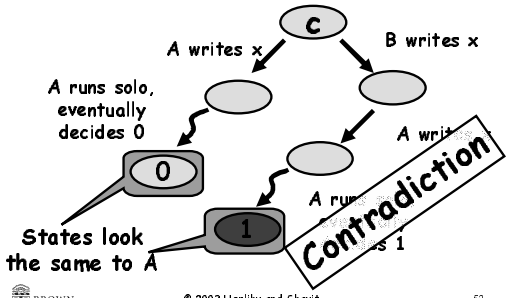
Writing Distinct Registers



Possible Interactions

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

Writing Same Registers



That's All, Folks!

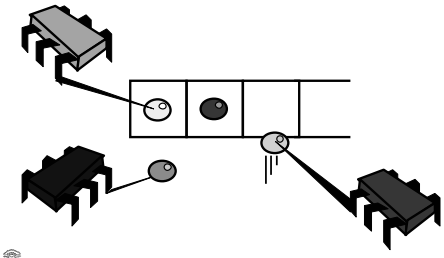
	x.read()	y.read()	x.write()	y.write()
x.read()	no	no	no	no
y.read()	no	no	no	no
x.write()	no	no	no	no
y.write()	no	no	no	no

QED

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

What Does Consensus have to do with Concurrent Objects?



Java Jargon Watch

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

Consensus Object

```
abstract class Consensus {
 private Object[] proposed =
  new Object[N];

 public void propose(Object value) {
  proposed[Thread.myIndex()] = value;
 }

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


Consensus Object

```
abstract class Consensus {
 private Object[] proposed =
  new Object[N];

 public void propose(Object value) {
  proposed[Thread.myIndex()] = value;
 }

 abstract public Obj
 }}
```

Each thread's proposed value

Consensus Object

Propose a value

```
sensus {
proposed =
 new Object[N];

public void propose(Object value) {
 proposed[Thread.myIndex()] = value;
}

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


Consensus Object

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

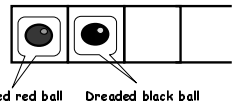
```
public void propose(Object value) {
 proposed[Thread.myIndex()] = value;
}

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

Can FIFO Queue Implement Consensus?

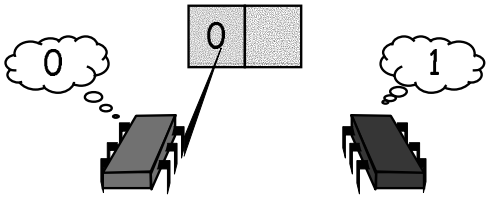
FIFO Consensus

announce array

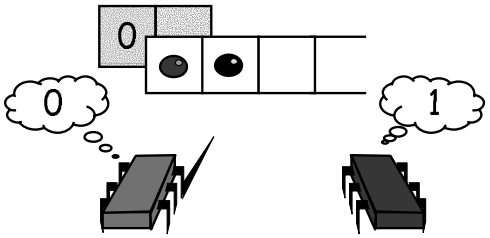


FIFO Queue with red and black balls

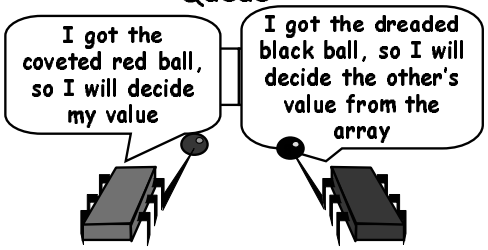
Protocol: Write Value to Array



Protocol: Take Next Item from Queue



Protocol: Take Next Item from Queue

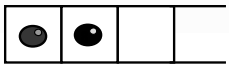


Consensus Using FIFO Queue

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

Initialize Queue

```
public class QueueConsensus
  extends Consensus {
 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 Consensus {
 private Queue queue;
 …
 public decide() {
  Ball ball = this.queue.deq();
  if (ball == Ball.RED)
   return proposed[i];
  else
   return proposed[j];
 }
}
```


Who Won?

```
public class QueueConsensus
  extends Consensus {
 private Queue queue;
 …
 public decide() {
  Ball ball = this.queue.deq();
  if (ball == Ball.RED)
   return proposed[i];
  else
   return proposed[j];
 }
}
```

Race to dequeue first queue item

Who Won?

```
public class QueueConsensus
  extends Consensus {
 private Queue queue;
 …
 public decide() {
  Ball ball = this.queue.deq();
  if (ball == Ball.RED)
   return proposed[i];
  else
   return proposed[j];
 }
}
```

I win if I was first

Who Won?

```
public class QueueConsensus
  extends Consensus {
 private Queue queue;
 …
 public decide() {
  Ball ball = this.queue.deq();
  if (ball == Ball.RED)
   return proposed[j];
  else
   return proposed[j];
 }
}
```

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

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 n
  - And Y has consensus number m
  - 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/multiple read OK
  - Multi write/multiple 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);
}
```

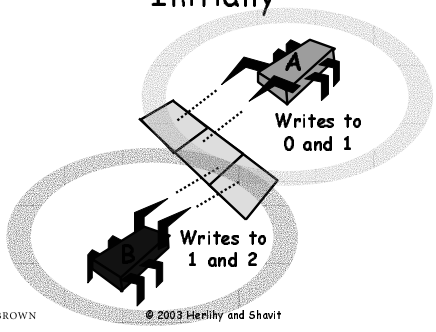
Atomically assign
value[i_1]= v_1
value[i_2]= v_2

Double Assignment Interface

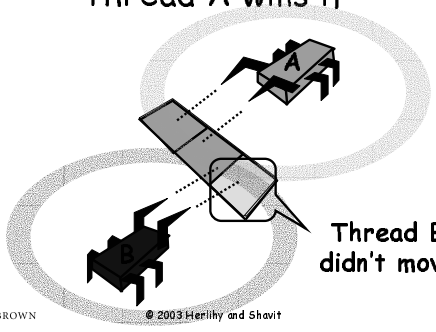
```
interface Assign2 {
 public void assign(int i1, int v1,
                    int i2, int v2);
 public int read(int i);
}
```

Return i-th value

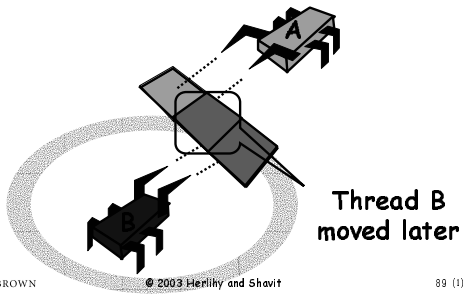
Initially



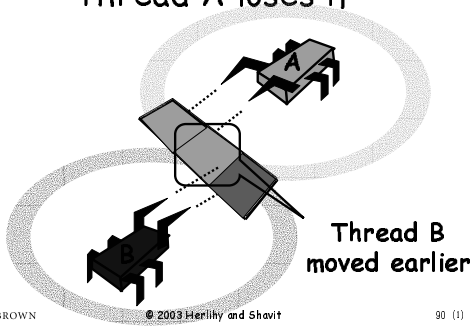
Thread A wins if



Thread A wins if



Thread A loses if



Multi-Consensus Code

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


Multi-Consensus Code

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

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

Multi-Consensus Code

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

OK, I win.

Multi-Consensus Code

```
class MultiConsensus extends Consensus{
 Assign2 a = new Assi
 public Object decide
  a.assign(i, i, i+1,
  int other = a.read(
  if (other==EMPTY||o
   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 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;
  }
}
```

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 void 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

```
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 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)
- Read() is trivial
- getAndIncrement() is non-trivial

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 `this.announce[i]`
    - Built-in method
  - `decide()` method
    - which determines winning value
    - Customized, class-specific method

Proof

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

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


Proof

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

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

Initialized to v

Proof

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

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

Am I first?

Proof

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

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

Yes, return my input

Proof

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

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

No, return other's input

Proof

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

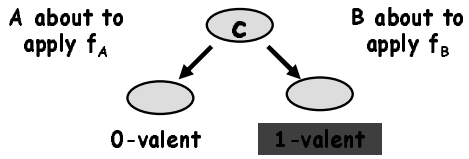
Interfering RMW

- 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 such set of RMW objects has consensus number exactly 2

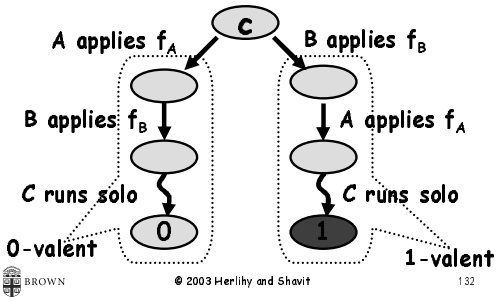
Examples

- Test-and-Set f(v)=1
  Overwrite $f_i(f_j(v))=f_i(v)$
- Swap f(v,x)=x
  Overwrite $f_i(f_j(v))=f_i(v)$
- Fetch-and-inc f(v)=v+1
  Commute $f_i(f_j(v))= f_j(f_i(v))$

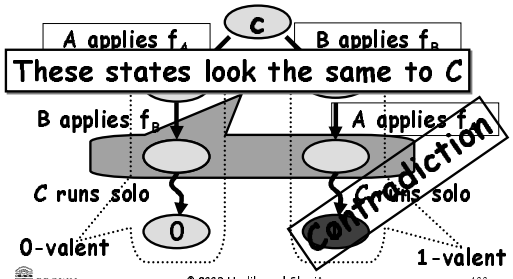
Meanwhile Back at the Critical State



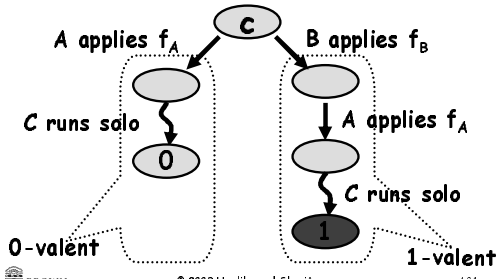
Maybe the Functions Commute



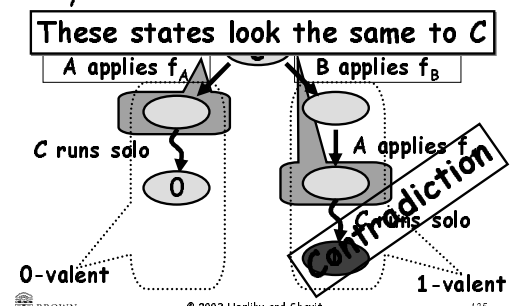
Maybe the Functions Commute



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 expected, …

compareAndSet Has ∞ Consensus Number

```
public class RMWConsensus
     implements Consensus {
 private AtomicInteger r =
   new AtomicInteger(-1);

 public Object decide() {
  r.compareAndSet(-1,i);
  return this.announce[r.get()];
 }
}
```


compareAndSet Has ∞ Consensus Number

```
public class RMWConsensus
     implements Consensus {
 private AtomicInteger r =
   new AtomicInteger(-1);

 public Object decide() {
  r.compareAndSet(-1,i);
  return this.announce[r.get()];
 }
}
```

Initialized to -1

compareAndSet Has ∞ Consensus Number

```
public class RMWConsensus
     implements Consensus {
 private AtomicInteger r =
   new AtomicInteger(-1);

 public Object decide() {
  r.compareAndSet(-1,i);
  return this.announce[r.get()];
 }
}
```

Try to swap in my id

compareAndSet Has ∞ Consensus Number

```
public class RMWConsensus
     implements Consensus {
 private AtomicInteger r =
   new AtomicInteger(-1);

 public Object decide() {
  r.compareAndSet(-1,i);
  return this.announce[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-Free Implementations

- Infinitely often some method call completes in a finite number of steps
- Pragmatic approach
- Implies no mutual exclusion

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
  - Wait-free/Lock-free
  - Linearizable
  - n-threaded
  - Implementation
  - Of any sequentially specified object

The Relative Power of Synchronization Methods

Nir Shavit
Multiprocessor Synchronization
Spring 2003

Notes For The Relative Power of Synchronization Methods

- Students had a lot of questions during lecture so I added a lot of slides…
- Added lock-freedom in the end, especially since we will talk about it when doing unoiversal stuff. It needs more lock-free stuff since it becoime major later
- What about robustness, gotta say somewhere that reduction theorem works only for deterministic data structures
- I updated many slides but hav't listed which yet, sorry

- Added slide for getting toi CS
- Added slide for using only two registers