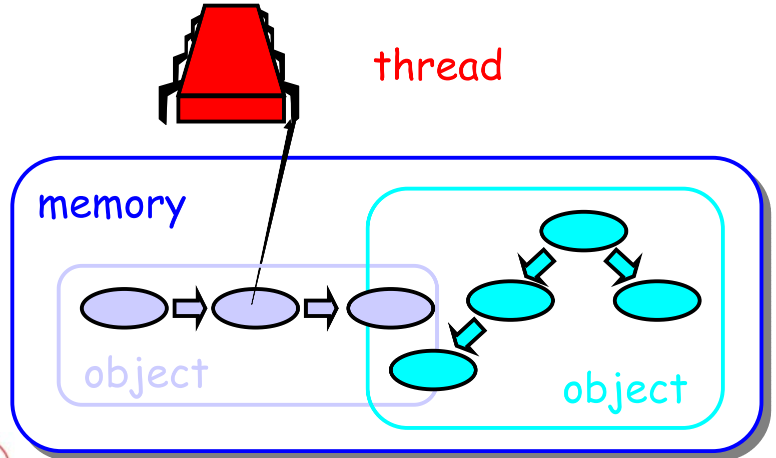
The Consensus Problem

Roger Wattenhofer

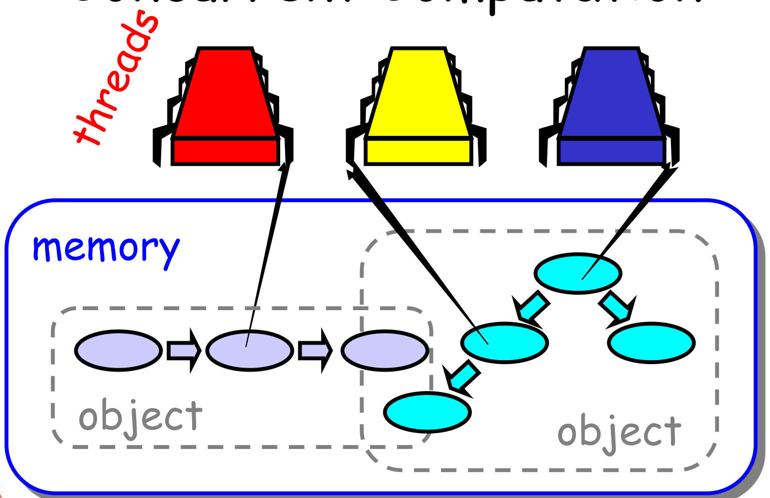
Distributed Computing Group

a lot of kudos to Maurice Herlihy and Costas Busch for some of their slides

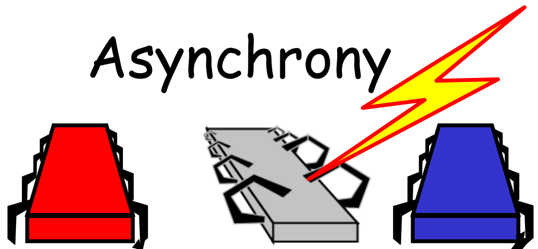
Sequential Computation



Concurrent Computation



Asynchrony



Sudden unpredictable delays
- Cache misses (*short*)
- Page faults (*long*)
- Scheduling quantum used up (*really long*)

Model Summary

- Multiple *threads*
  - Sometimes called *processes*
- Single shared *memory*
- *Objects* live in memory
- Unpredictable asynchronous delays

Road Map

- We are going to focus on principles
  - Start with idealized models
  - Look at a simplistic problem
  - Emphasize correctness over pragmatism
  - "Correctness may be theoretical, but incorrectness has practical impact"

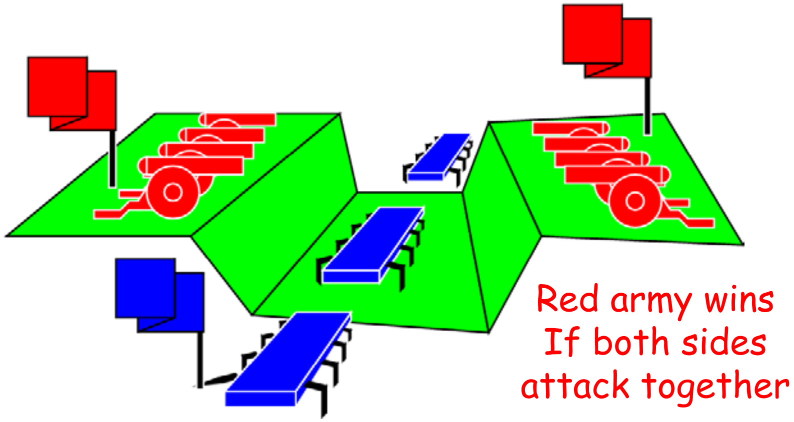
You may ask yourself …

I'm no theory weenie - why all the theorems and proofs?

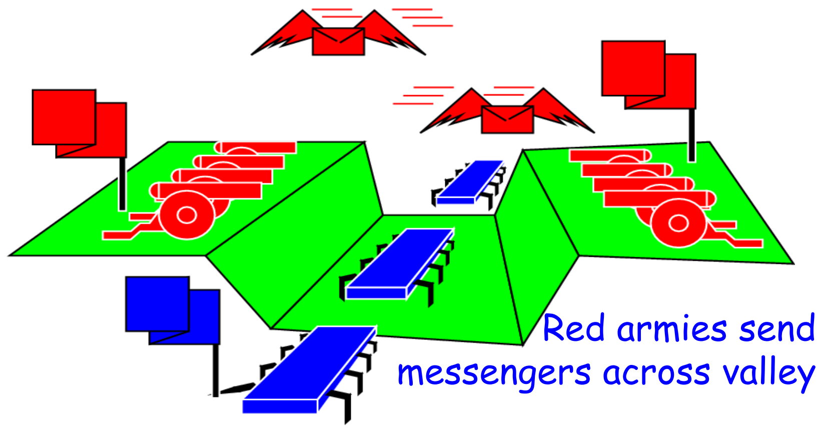
Fundamentalism

- Distributed & concurrent systems are *hard*
  - Failures
  - Concurrency
- Easier to go from theory to practice than vice-versa

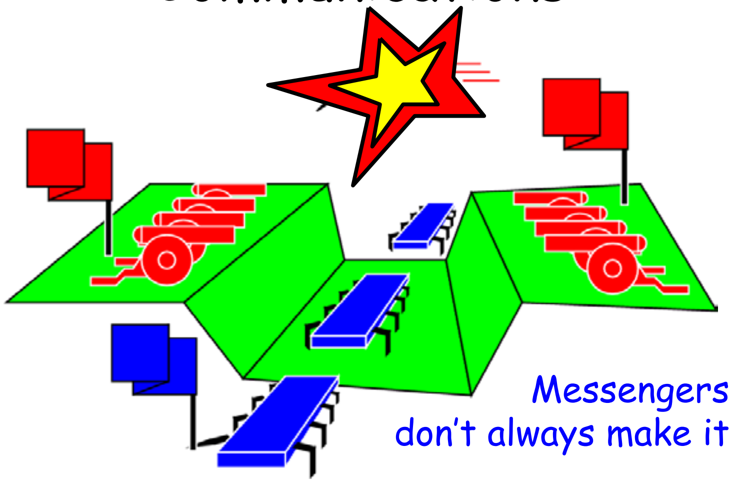
The Two Generals



Communications



Communications



Your Mission

Design a protocol to ensure that red armies attack simultaneously

Real World Generals

Date: Wed, 11 Dec 2002 12:33:58 +0100
From: Friedemann Mattern <mattern@inf.ethz.ch>
To: Roger Wattenhofer <wattenhofer@inf.ethz.ch>
Subject: Vorlesung

Sie machen jetzt am Freitag, 08:15 die Vorlesung Verteilte Systeme, wie vereinbart. OK? (Ich bin jedenfalls am Freitag auch gar nicht da.) Ich uebernehme das dann wieder nach den Weihnachtsferien.

Real World Generals

Date: Mi 11.12.2002 12:34
From: Roger Wattenhofer <wattenhofer@inf.ethz.ch>
To: Friedemann Mattern <mattern@inf.ethz.ch>
Subject: Re: Vorlesung

OK. Aber ich gehe nur, wenn sie diese Email nochmals bestaetigen... :-)

Gruesse -- Roger Wattenhofer

Real World Generals

Date: Wed, 11 Dec 2002 12:53:37 +0100
From: Friedemann Mattern <mattern@inf.ethz.ch>
To: Roger Wattenhofer <wattenhofer@inf.ethz.ch>
Subject: Naechste Runde: Re: Vorlesung ...

Das dachte ich mir fast. Ich bin Praktiker und mache es schlauer: Ich gehe nicht, unabhaengig davon, ob Sie diese email bestaetigen (beziehungsweise rechtzeitig erhalten). (:-)

Real World Generals

Date: Mi 11.12.2002 13:01
From: Roger Wattenhofer <wattenhofer@inf.ethz.ch>
To: Friedemann Mattern <mattern@inf.ethz.ch>
Subject: Re: Naechste Runde: Re: Vorlesung ...

Ich glaube, jetzt sind wir so weit, dass ich diese Emails in der Vorlesung auflegen werde...

Real World Generals

```
Date: Wed, 11 Dec 2002 18:55:08 +0100
From: Friedemann Mattern <mattern@inf.ethz.ch>
To: Roger Wattenhofer <wattenhofer@inf.ethz.ch>
Subject: Re: Naechste Runde: Re: Vorlesung ...

Kein Problem. (Hauptsache es kommt raus, dass der
Prakiker am Ende der schlauere ist... Und der
Theoretiker entweder heute noch auf das allerletzte
Ack wartet oder wissend das das ja gar nicht gehen
kann alles gleich von vornherein bleiben laesst...
(:-))
```


Theorem

There is no non-trivial protocol that ensures the red armies attacks simultaneously

Proof Strategy

- Assume a protocol exists
- Reason about its properties
- Derive a contradiction

Proof

1. Consider the protocol that sends fewest messages
2. It still works if last message lost
3. So just don't send it
   - Messengers' union happy
4. But now we have a shorter protocol!
5. Contradicting #1

Fundamental Limitation

- Need an unbounded number of messages
- Or possible that no attack takes place

You May Find Yourself …

I want a real-time YAFA compliant Two Generals protocol using UDP datagrams running on our enterprise-level fiber tachyion network ...

You might say

I want a real-time YAFA

p

r

fiber tachyion netwo

Yes, Ma'am, right away!

You might say

Advantage:

- Buys time to find another job
- No one expects software to work anyway

fiber tachyion netwo

You might say

Advantage:
- Buys time to find another job

Disadvantage:
- You're doomed
- Without this course, you may not even know you're doomed

You might say

I can't find a fault-tolerant algorithm, I guess I'm just a pathetic loser.

You might say

Advantage:
- No need to take course

algorithm, I guess I'm just a pathetic loser

You might say

Advantage:
- No need to take course

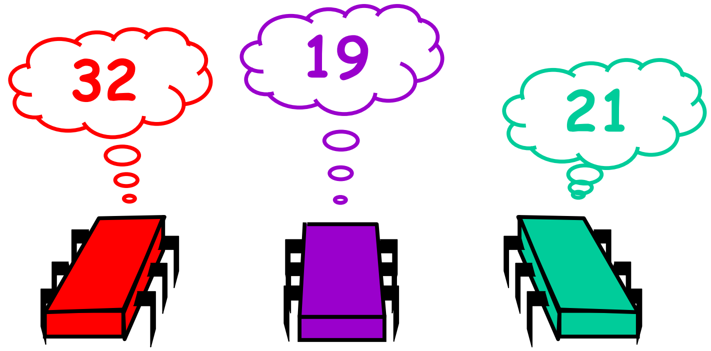
Disadvantage:
- Boss fires you, hires University St. Gallen graduate

You might say

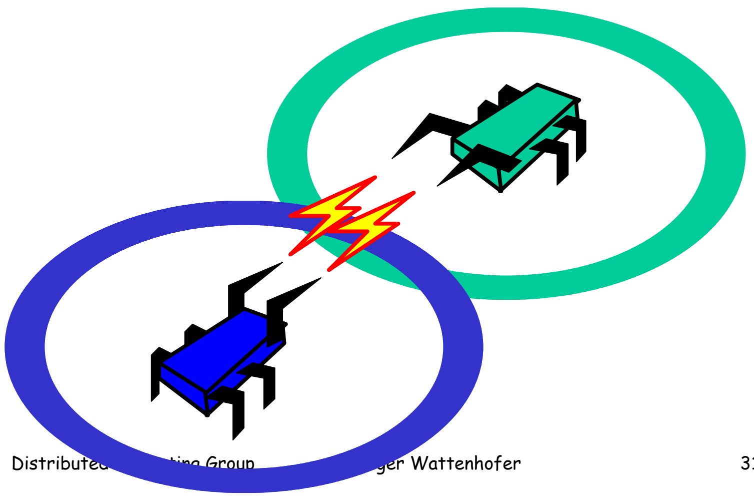
I want a real-time YAFA

Using skills honed in course, I can avert certain disaster!
- Rethink problem spec, or
- Weaken requirements, or
- Build on different platform

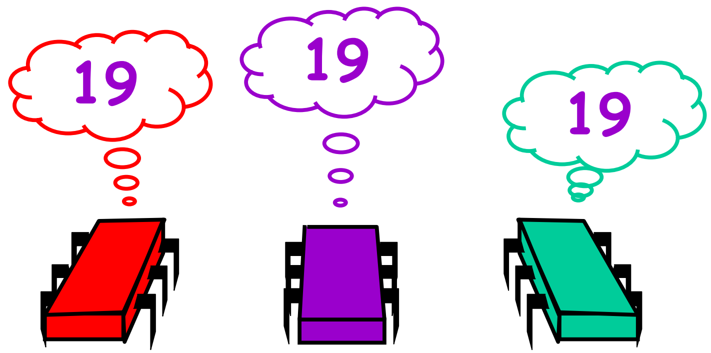
Consensus: Each Thread has a Private Input



They Communicate



They Agree on Some Thread's Input



Consensus is important

- With consensus, you can implement anything you can imagine…
- Examples: with consensus you can decide on a leader, implement mutual exclusion, or solve the two generals problem

You gonna learn

- In some models, consensus is possible
- In some other models, it is not
- Goal of this and next lecture: to learn whether for a given model consensus is possible or not … and prove it!

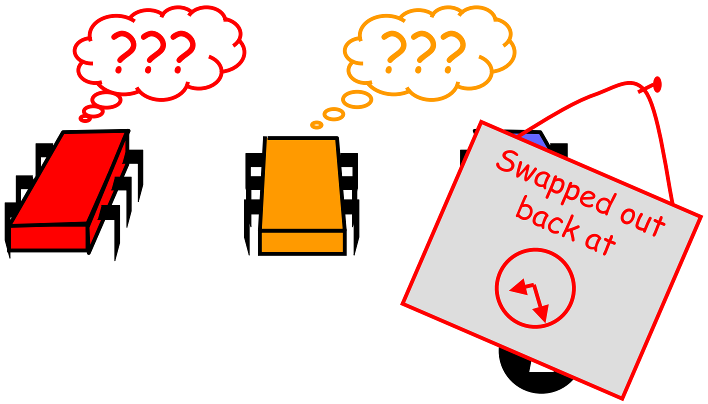
Consensus #1 shared memory

- n processors, with n > 1
- Processors can atomically *read* or *write* (not both) a shared memory cell

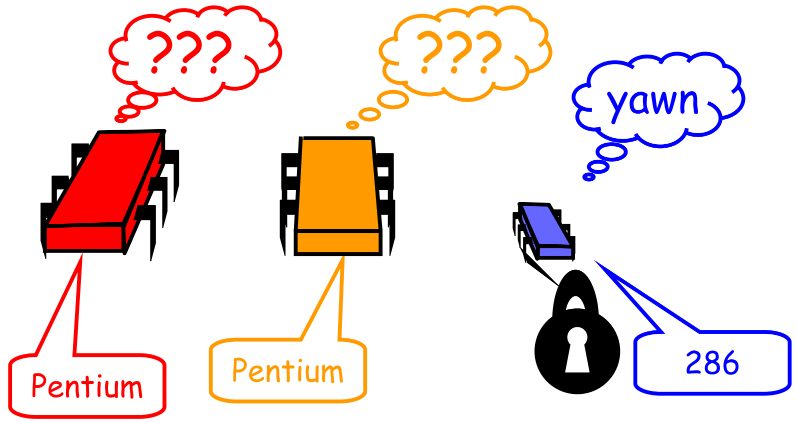
Protocol (Algorithm?)

- There is a designated memory cell c.
- Initially c is in a special state "?"
- Processor 1 writes its value v_1 into c, then decides on v_1.
- A processor j (j not 1) reads c until j reads something else than "?", and then decides on that.

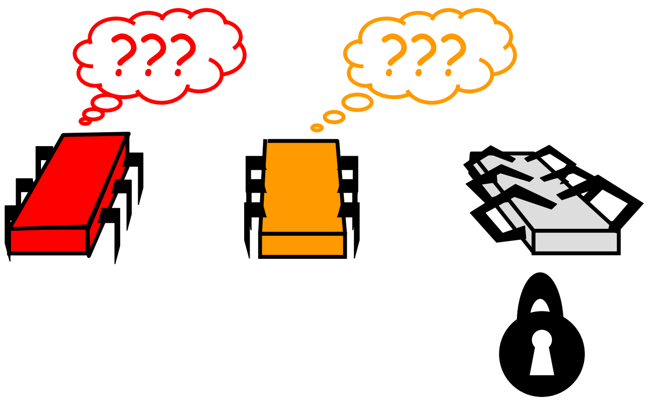
Unexpected Delay



Heterogeneous Architectures



Fault-Tolerance



Consensus #2
wait-free shared memory

- n processors, with n > 1
- Processors can atomically *read* or *write* (not both) a shared memory cell
- Processors might crash (halt)
- Wait-free implementation… huh?

Wait-Free Implementation

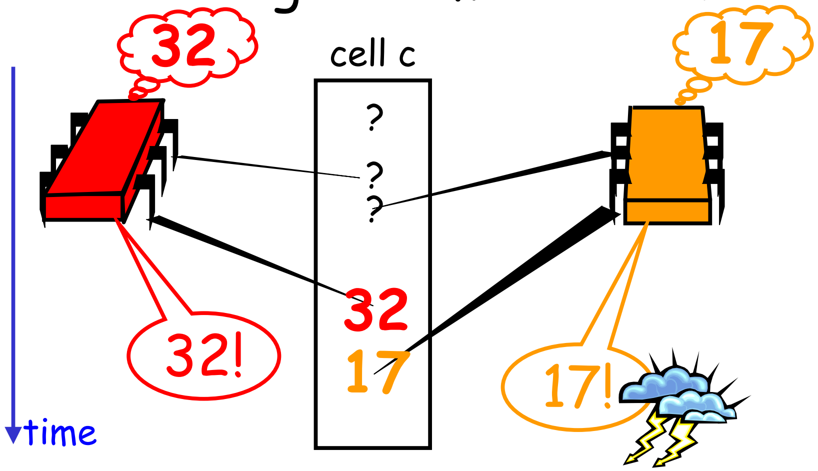
- Every process (method call) completes in a finite number of steps
- Implies no mutual exclusion
- We assume that we have wait-free atomic registers (that is, reads and writes to same register do not overlap)

A wait-free algorithm…

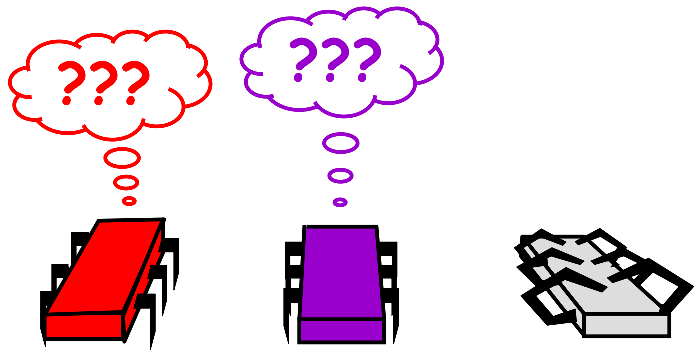
- There is a cell c, initially c="?"
- Every processor i does the following

```
r = Read(c);
if (r == "?") then
      Write(c, v_i); decide v_i;
else
      decide r;
```


Is the algorithm correct?



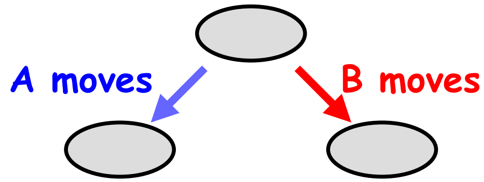
Theorem: No wait-free consensus



Proof Strategy

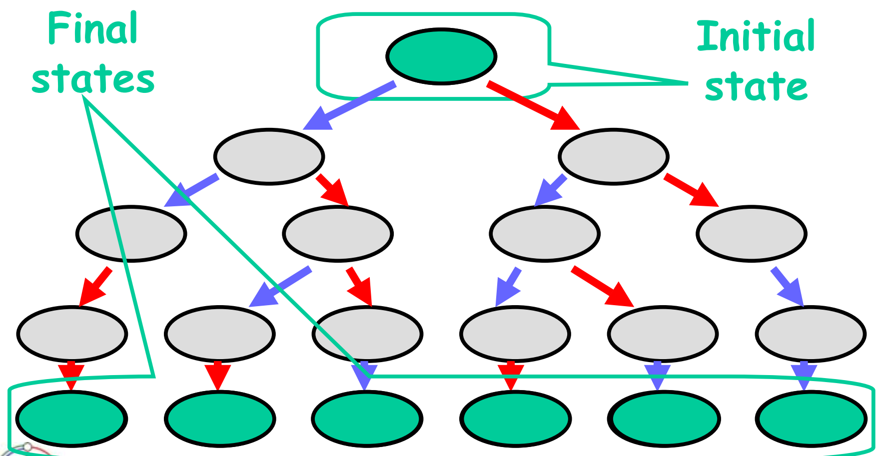
- Make it simple
  - n = 2, binary input
- Assume that there is a protocol
- Reason about the properties of any such protocol
- Derive a contradiction

Wait-Free Computation

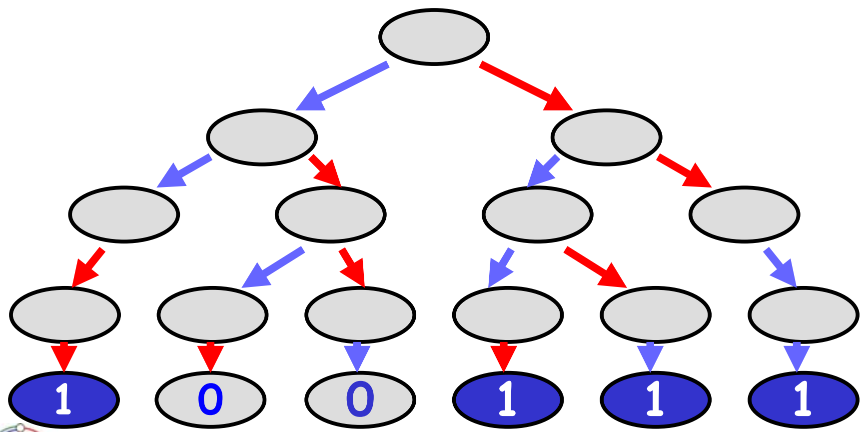


- Either A or B "moves"
- Moving means
  - Register read
  - Register write

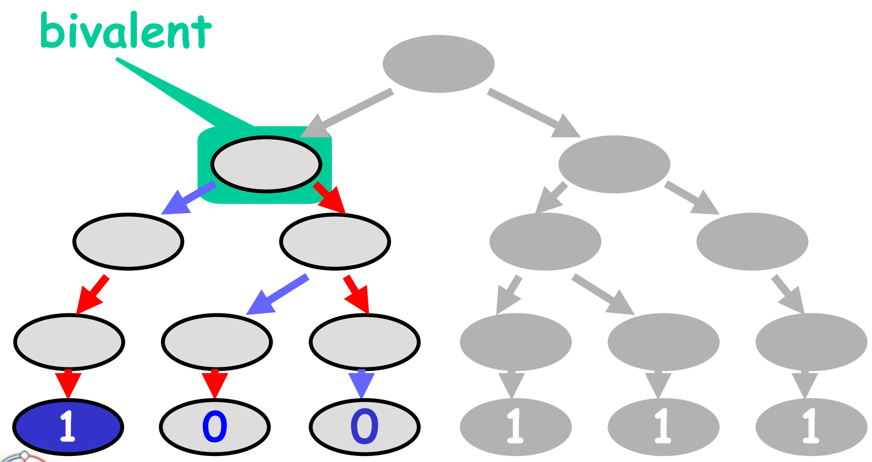
The Two-Move Tree



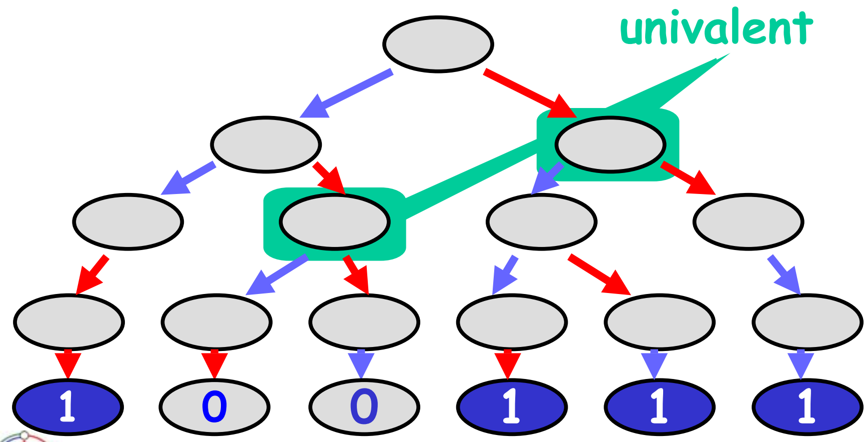
Decision Values



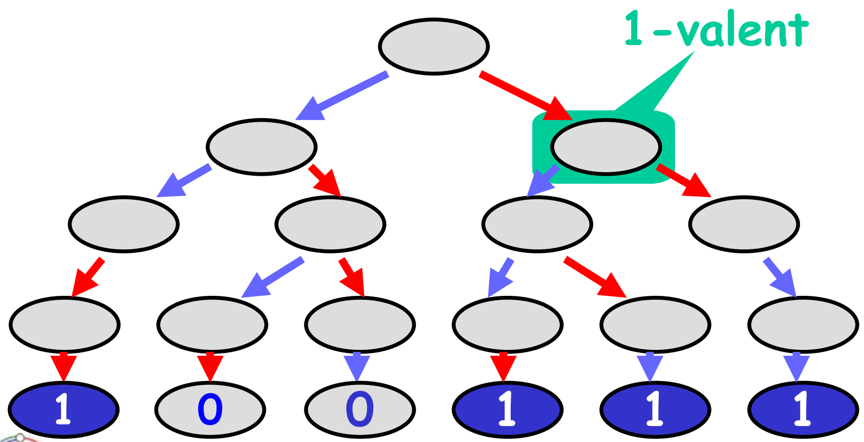
Bivalent: Both Possible



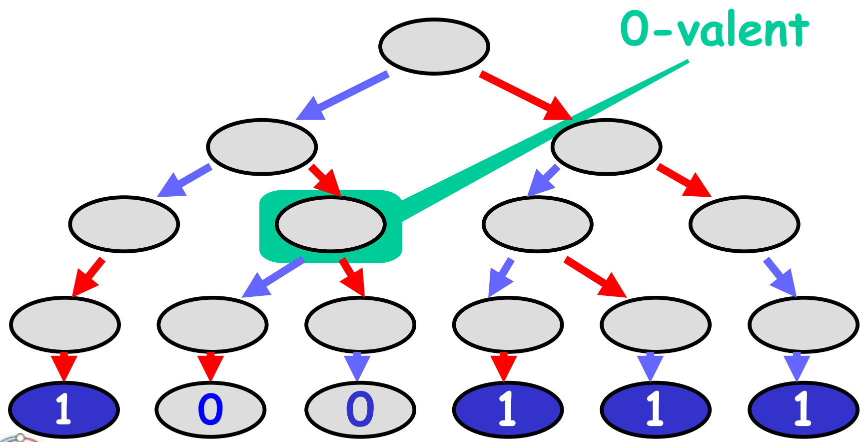
Univalent: Single Value Possible



1-valent: Only 1 Possible



0-valent: Only 0 possible



Summary

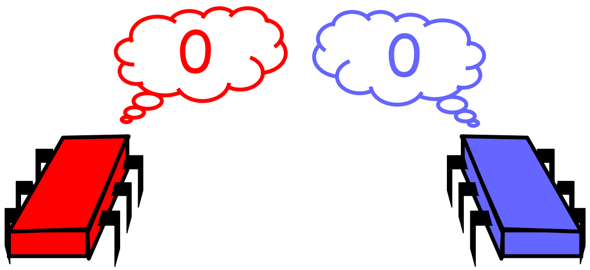
- Wait-free computation is a tree
- Bivalent system states
  - Outcome not fixed
- Univalent states
  - Outcome is fixed
  - May not be "known" yet
  - 1-Valent and 0-Valent states

Claim

Some initial system state is bivalent

(The outcome is not always fixed from the start.)

A 0-Valent Initial State

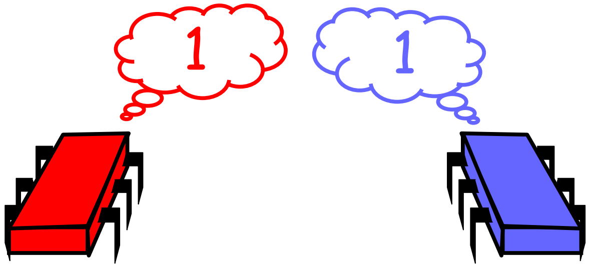


- All executions lead to decision of 0

A 0-Valent Initial State

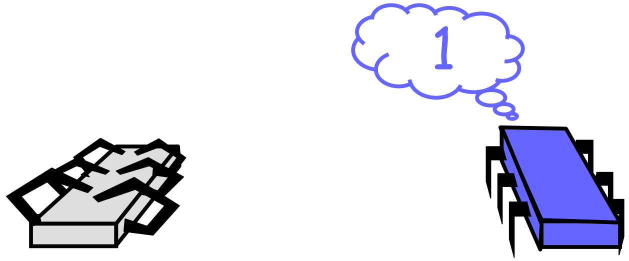
- Solo execution by A also decides 0

A 1-Valent Initial State



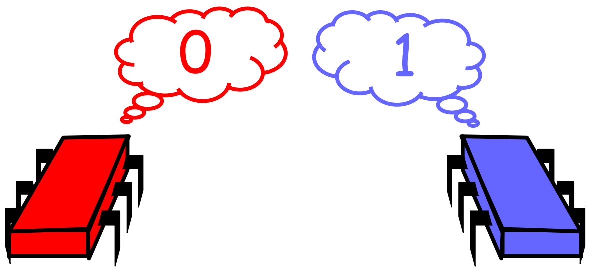
- All executions lead to decision of 1

A 1-Valent Initial State



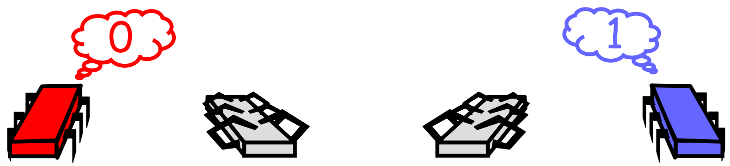
- Solo execution by B also decides 1

A Univalent Initial State?



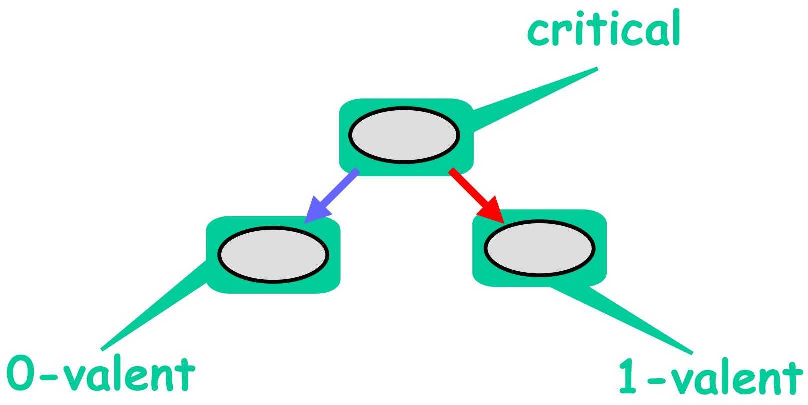
- Can all executions lead to the same decision?

State is Bivalent



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

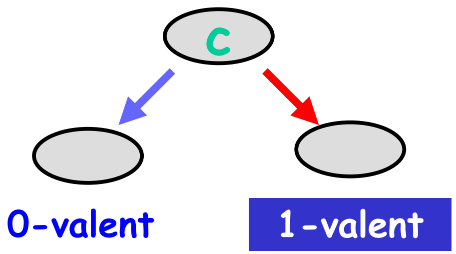
Critical States



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

From a Critical State



If A goes first, protocol decides 0

If B goes first, protocol decides 1

Model Dependency

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

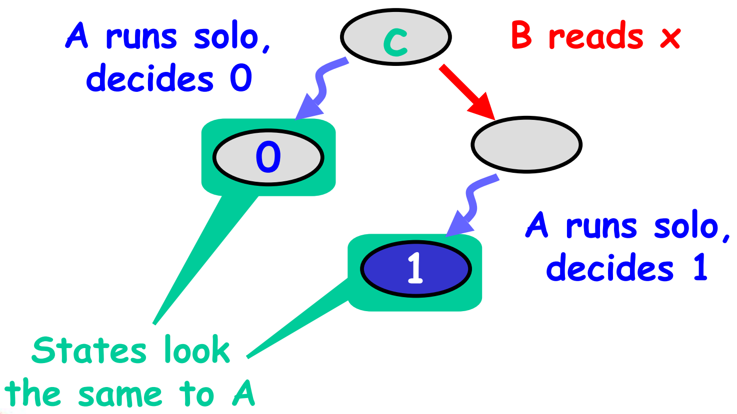
What are the Threads Doing?

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

Possible Interactions

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

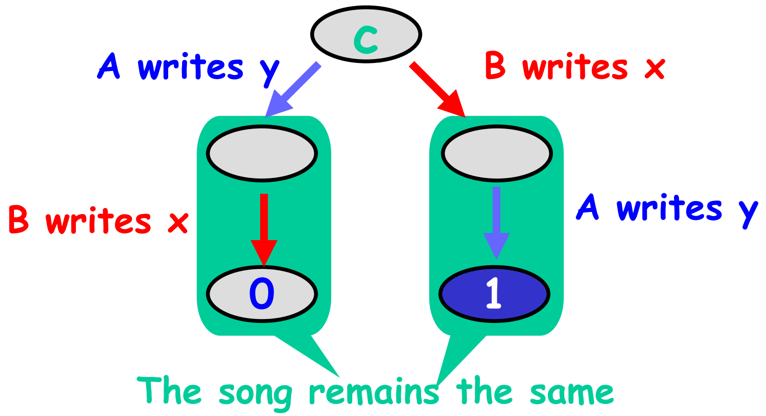
Reading Registers



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	?	?

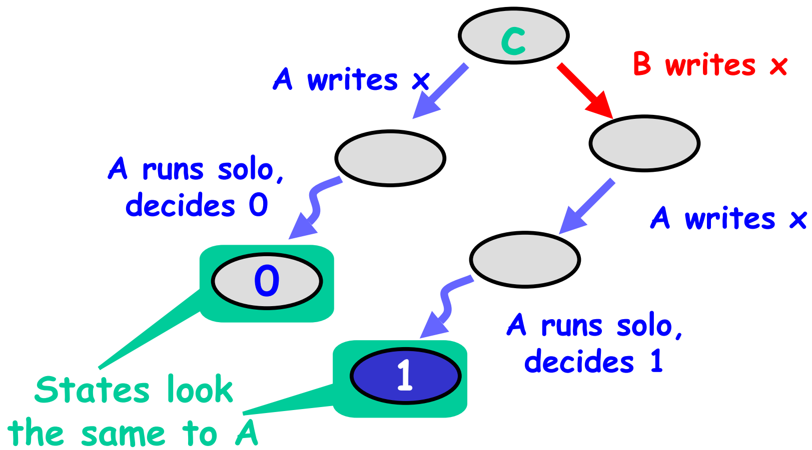
Writing Distinct Registers



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



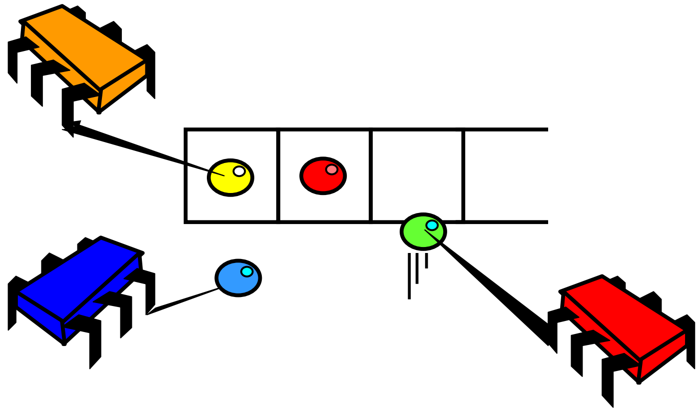
That's All, Folks!

	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

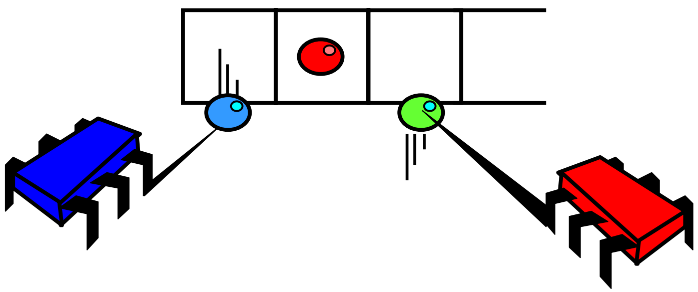
Theorem

- It is impossible to solve consensus using read/write atomic registers
  - Assume protocol exists
  - It has a bivalent initial state
  - Must be able to reach a critical state
  - Case analysis of interactions
    - Reads vs others
    - Writes vs writes

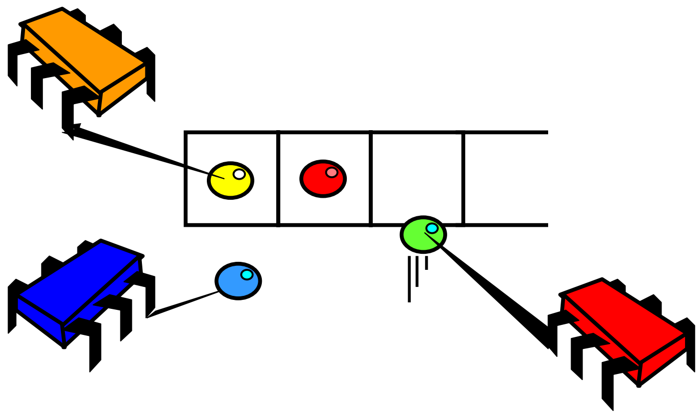
What Does Consensus have to do with Distributed Systems?



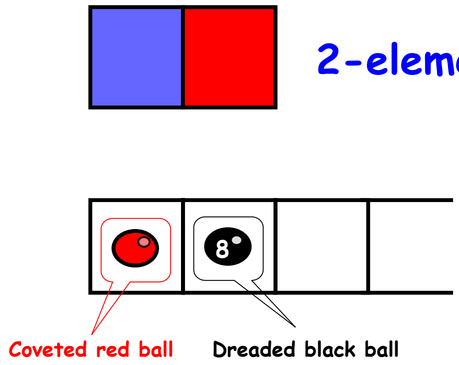
We want to build a Concurrent FIFO Queue



With Multiple Dequeuers!



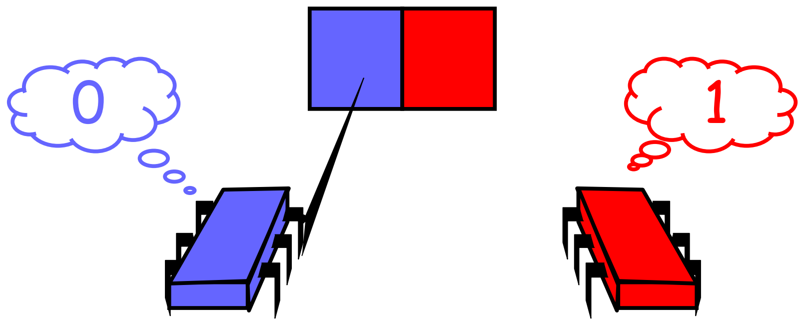
A Consensus Protocol



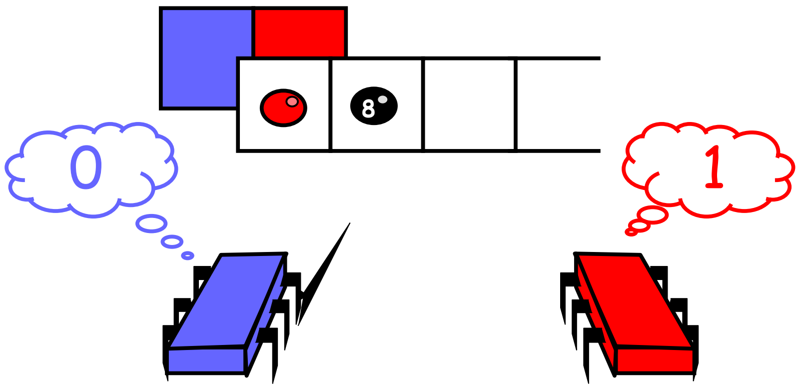
2-element array

FIFO Queue with red and black balls

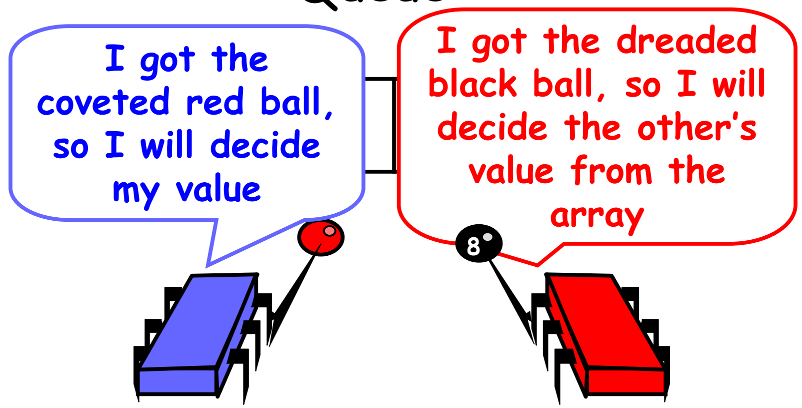
Protocol: Write Value to Array



Protocol: Take Next Item from Queue



Protocol: Take Next Item from Queue



Why does this Work?

- If one thread gets the red ball
- Then the other gets the black ball
- Winner can take her own value
- Loser can find winner's value in array
  - Because threads write array before dequeuing from queue

Implication

- We can solve 2-thread consensus using only
  - A two-dequeuer queue
  - Atomic registers

Implications

- Assume there exists
  - A queue implementation from atomic registers
- Given
  - A consensus protocol from queue and 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 with read/write shared memory.
- This was a proof by reduction; important beyond NP-completeness…

Consensus #3
read-modify-write shared mem.

- n processors, with n > 1
- Wait-free implementation
- Processors can atomically read *and* write a shared memory cell in one atomic step: the value written can depend on the value read
- We call this a RMW register

Protocol

- There is a cell c, initially c="?"
- Every processor i does the following

```
RMW(c), with
  if (c == "?") then
      write(c, v_i); decide v_i;
  else
      decide c;
```

atomic step

Discussion

- Protocol works correctly
  - One processor accesses c as the first; this processor will determine decision
- Protocol is wait-free
- RMW is quite a strong primitive
  - Can we achieve the same with a weaker primitive?

Read-Modify-Write
more formally

- Method takes 2 arguments:
  - Variable x
  - Function f
- Method call:
  - Returns value of x
  - Replaces x with f(x)

Read-Modify-Write

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

  public void rmw(function f) {
    int prior  = this.value;
    this.value = f(this.value);
    return prior;
  }

}
```

Return prior value

Apply function

Example: Read

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

  public void read() {
    int prior  = this.value;
    this.value = this.value;
    return prior;
  }

}
```

identity function

Example: test&set

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

  public void TAS() {
    int prior  = this.value;
    this.value = 1;
    return prior;
  }

}
```

constant function

Example: fetch&inc

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

  public void fai() {
    int prior  = this.value;
    this.value = this.value+1;
    return prior;
  }

}
```

increment function

Example: fetch&add

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

  public void faa(int x) {
    int prior  = this.value;
    this.value = this.value+x;
    return prior;
  }

}
```

addition function

Example: swap

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

  public void swap(int x) {
    int prior  = this.value;
    this.value = x;
    return prior;
  }

}
```

constant function

Example: compare&swap

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

  public void CAS(int old, int new) {
    int prior  = this.value;
    if (this.value == old)
      this.value = new;
    return prior;
  }

}
```

complex function

"Non-trivial" RMW

- Not simply read
- But
  - test&set, fetch&inc, fetch&add, swap, compare&swap, general RMW
- Definition: A RMW is non-trivial if there exists a value v such that $v \neq f(v)$

Consensus Numbers (Herlihy)

- An object has consensus number n
  - If it can be used
    - Together with atomic read/write registers
  - To implement n-thread consensus
    - But not (n+1)-thread consensus

Consensus Numbers

- Theorem
  - Atomic read/write registers have consensus number 1
- Proof
  - Works with 1 process
  - We have shown impossibility with 2

Consensus Numbers

- Consensus numbers are a useful way of measuring 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!

Theorem

- Any non-trivial RMW object has consensus number at least 2
- Implies no wait-free implementation of RMW registers from read/write registers
- Hardware RMW instructions not just a convenience

Proof

Initialized to v

```
public class RMWConsensusFor2
    implements Consensus {
  private RMW r;

  public Object decide() {
    int i = Thread.myIndex();
    if (r.rmw(f) == v)
      return this.announce[i];
    else
      return this.announce[1-i];
  }}
```

Am I first?

Yes, return my input

No, return other's input

Proof

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

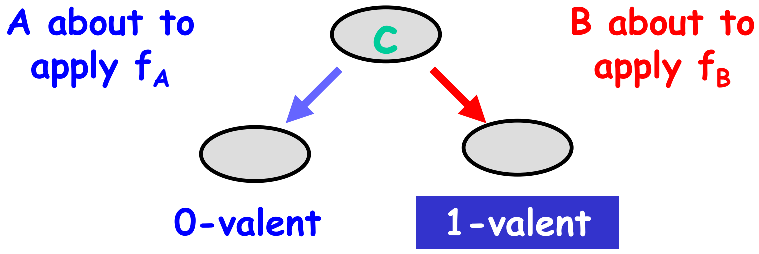
Interfering RMW

- Let F be a set of functions such that for all f_i and f_j, either
  - They commute: $f_i(f_j(x))=f_j(f_i(x))$
  - They overwrite: $f_i(f_j(x))=f_i(x)$
- Claim: Any such set of RMW objects has consensus number exactly 2

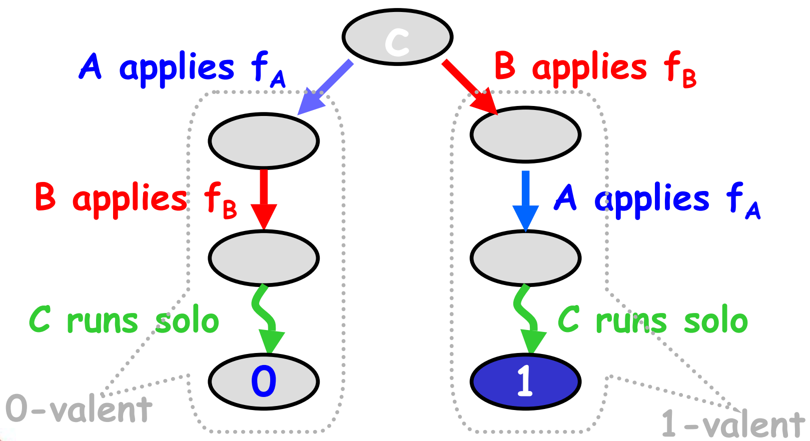
Examples

- Test-and-Set
  - Overwrite
- Swap
  - Overwrite
- Fetch-and-inc
  - Commute

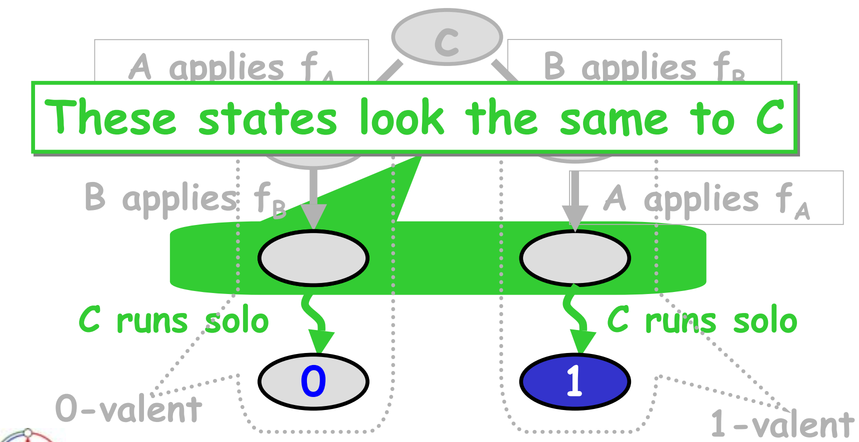
Meanwhile Back at the Critical State



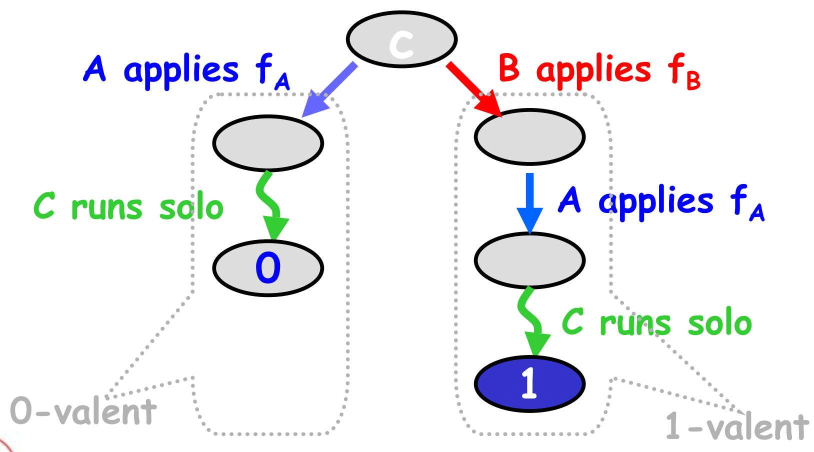
Maybe the Functions Commute



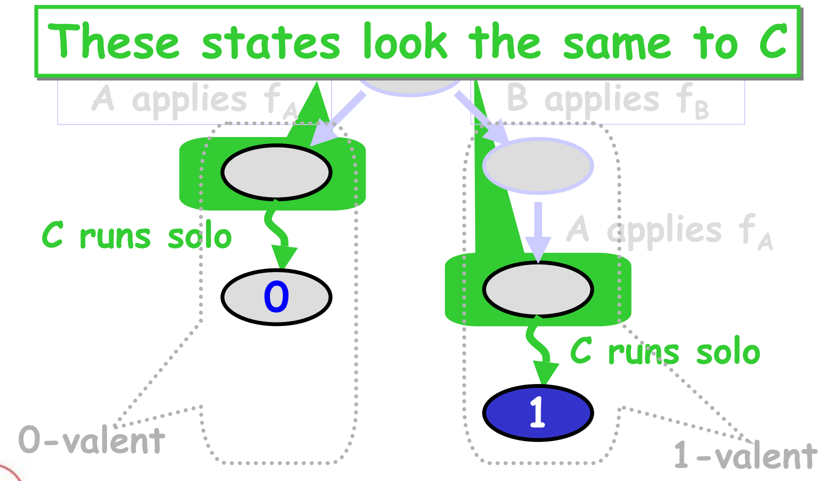
Maybe the Functions Commute



Maybe the Functions Overwrite



Maybe the Functions Overwrite



Impact

- Many early machines used these "weak" RMW instructions
  - Test-and-set (IBM 360)
  - Fetch-and-add (NYU Ultracomputer)
  - Swap
- We now understand their limitations
  - But why do we want consensus anyway?

CAS has Unbounded Consensus Number

```
public class RMWConsensus
    implements Consensus {
  private RMW r;

  public Object decide() {
    int i = Thread.myIndex();
    int j = r.CAS(-1,i);
    if (j == -1)
      return this.announce[i];
    else
      return this.announce[j];
  }}
```

Initialized to -1

Am I first?

Yes, return my input

No, return other's input