TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

Replication and Consistency

06 The Relative Power of Synchronization Operations

Annette Bieniusa

AG Softech
FB Informatik
TU Kaiserslautern

Thank you!

These slides are based on companion material of the following books:

- **The Art of Multiprocessor Programming** by Maurice Herlihy and Nir Shavit
- **Synchronization Algorithms and Concurrent Programming** by Gadi Taubenfeld

Motivation

Last lecture: Foundations of Shared Memory

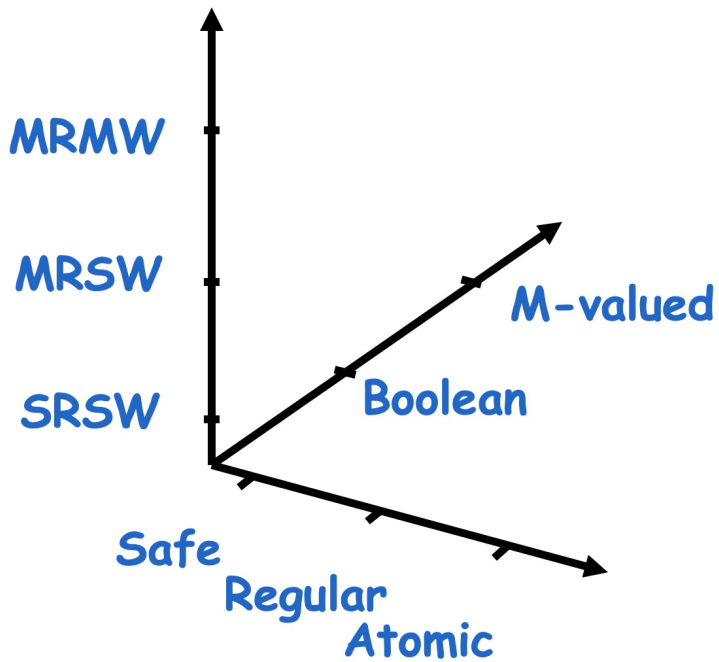
To understand modern multiprocessors we need to ask some basic questions:

What is the **weakest** useful form of shared memory?

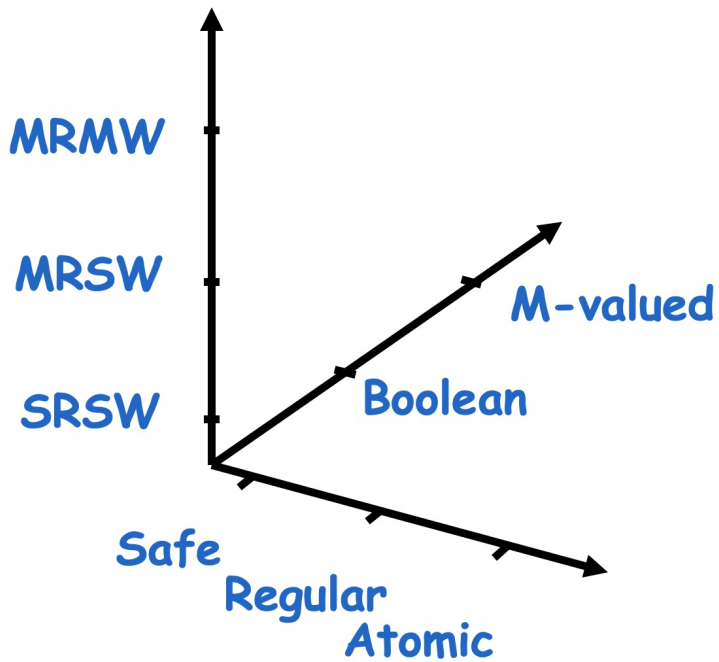
What concurrent problem **is (and what isn't)** computable under a given memory model?[1]

[1] Efficiency is mostly irrelevant here.

Last lecture: From the Weakest Register to Atomic Snapshot!



Last lecture: From the Weakest Register to Atomic Snapshot!



But some synchronization problems require more powerful registers!

Wait-Free Implementations

- Strongest non-blocking progress guarantee
- For every operation / method call, there is a bound on the number of steps that the algorithm will take before the operation completes

Wait-Free Implementations

- Strongest non-blocking progress guarantee
- For every operation / method call, there is a bound on the number of steps that the algorithm will take before the operation completes
- It implies that the algorithm does not rely on mutual exclusion!

The Consensus Problem

The Consensus Problem

- Each process p_i proposes a value v_i
- All processes have to agree on some common value v that is the initial value of some p_i

The Consensus Problem

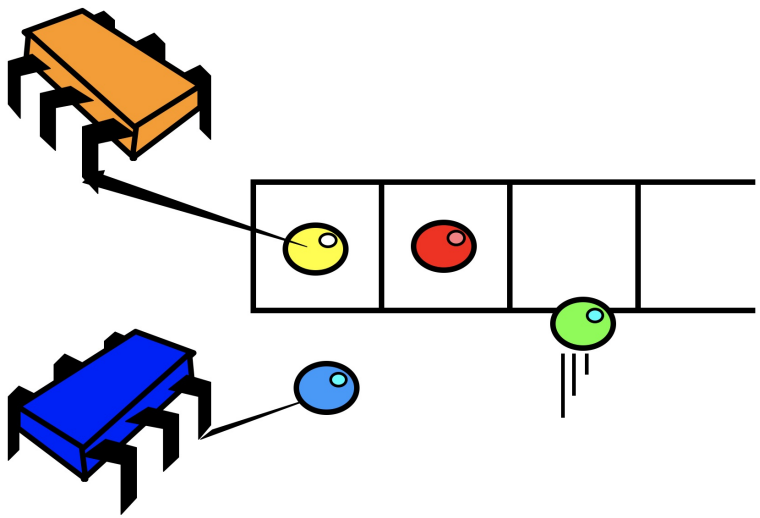
- Each process p_i proposes a value v_i
- All processes have to agree on some common value v that is the initial value of some p_i

Properties of Consensus:

- *Uniform Agreement*: Every process must decide on the same value.
- *Integrity*: Every process decides at most one value, and if it decides some value, then it must have been proposed by some process.
- *Termination*: All processes eventually reach a decision.
- *Validity*: If all processes propose the same value v, then all processes decide v.

Implementing Consensus based on FIFO Queues

- Assume we have a linearizable FIFO-Queue for two dequeuers.
- How can we use it to implement consensus for two threads?



Theorem

- Asynchronous computability is fundamentally different from Turing computability!
- Adapted version of fundamental theorem by Fisher, Lynch, Peterson for distributed computing (Fischer, Lynch, and Paterson 1985) which received the Dijkstra prize for the most influential paper in distributed computing in 2001

There is no wait-free (deterministic) implementation of n-thread consensus ($n > 1$) from read-write registers.

Proof Strategy

- Assume that there **is** a wait-free (deterministic) implementation ...
- Reason about the properties of **any** such protocol
- Derive a contradiction $\Rightarrow$ Done :)

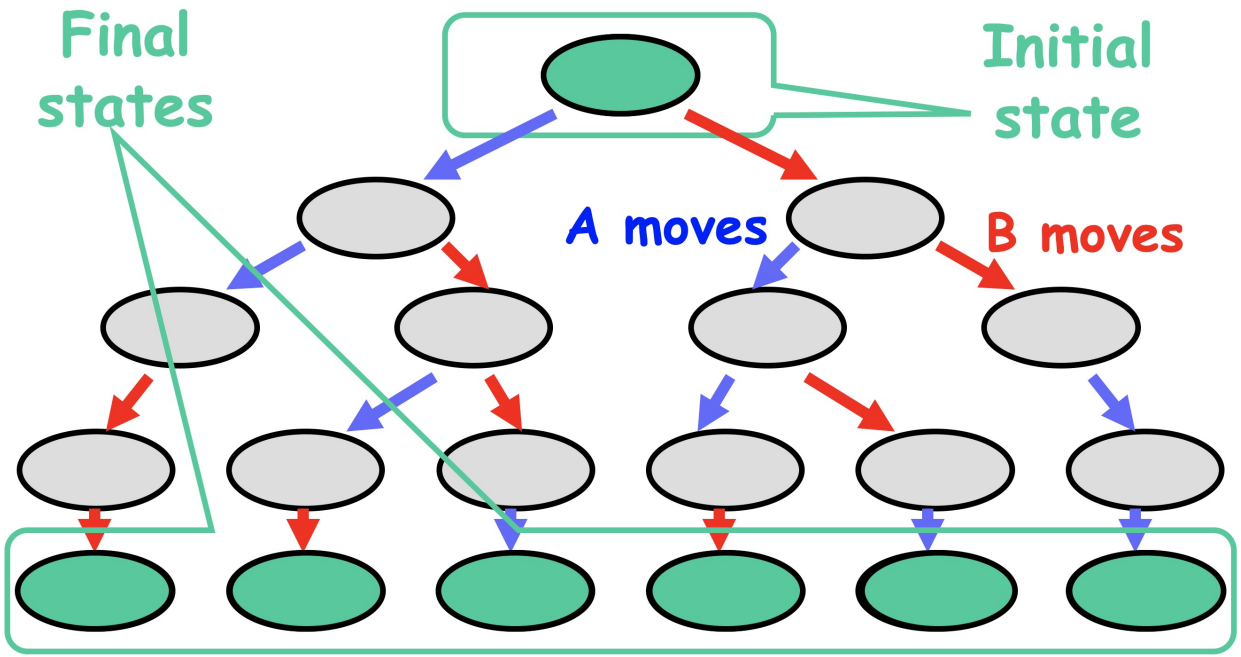
Proof Strategy

- Assume that there **is** a wait-free (deterministic) implementation ...
- Reason about the properties of **any** such protocol
- Derive a contradiction $\Rightarrow$ Done :)

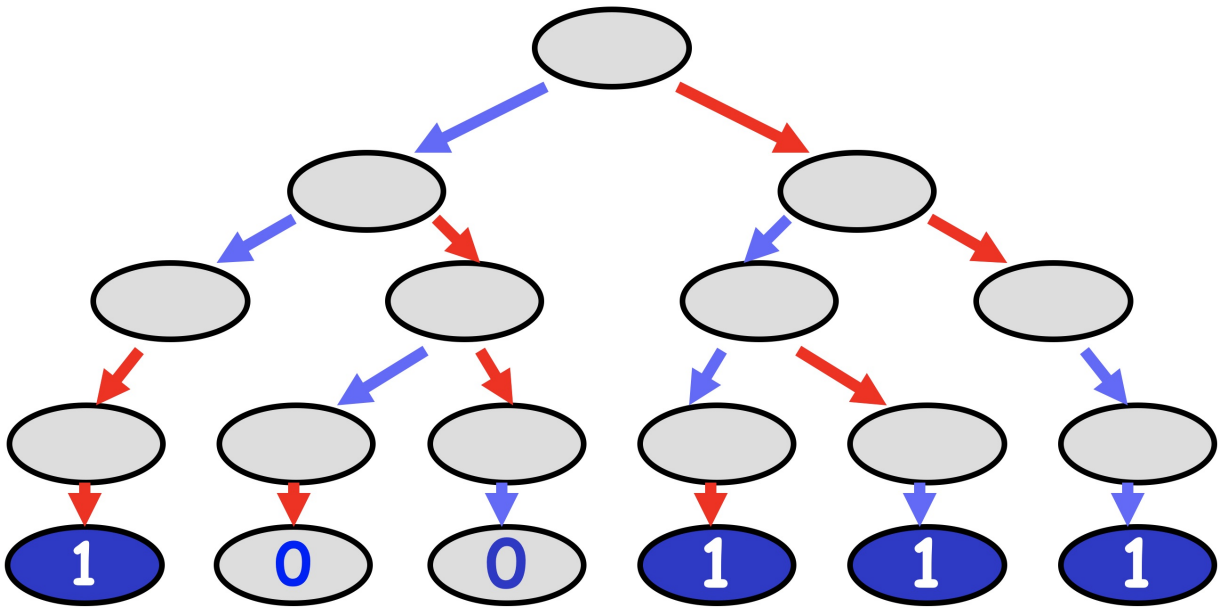
- Suffices to consider $n = 2$ processes and binary consensus (i.e. proposed values are either 0 or 1)

Essence of wait-free computation

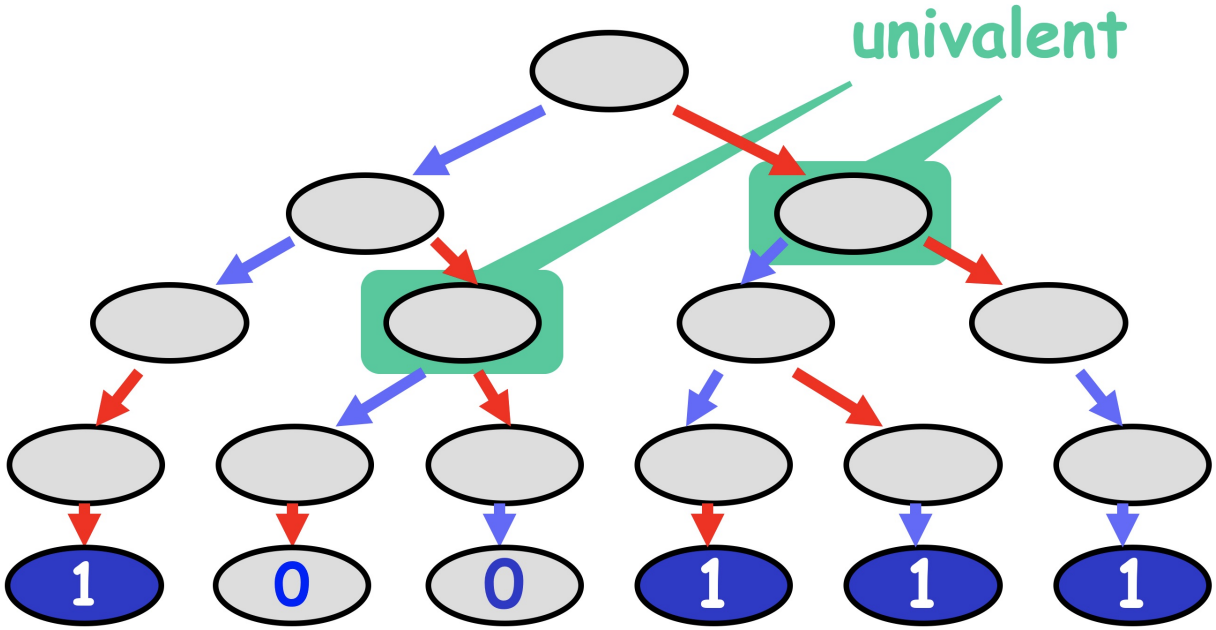
- Either A or B **moves**
- "Moving" here means
  - reads a register, or
  - writes a register
- For two processes, wait-free computations can be modeled as tree



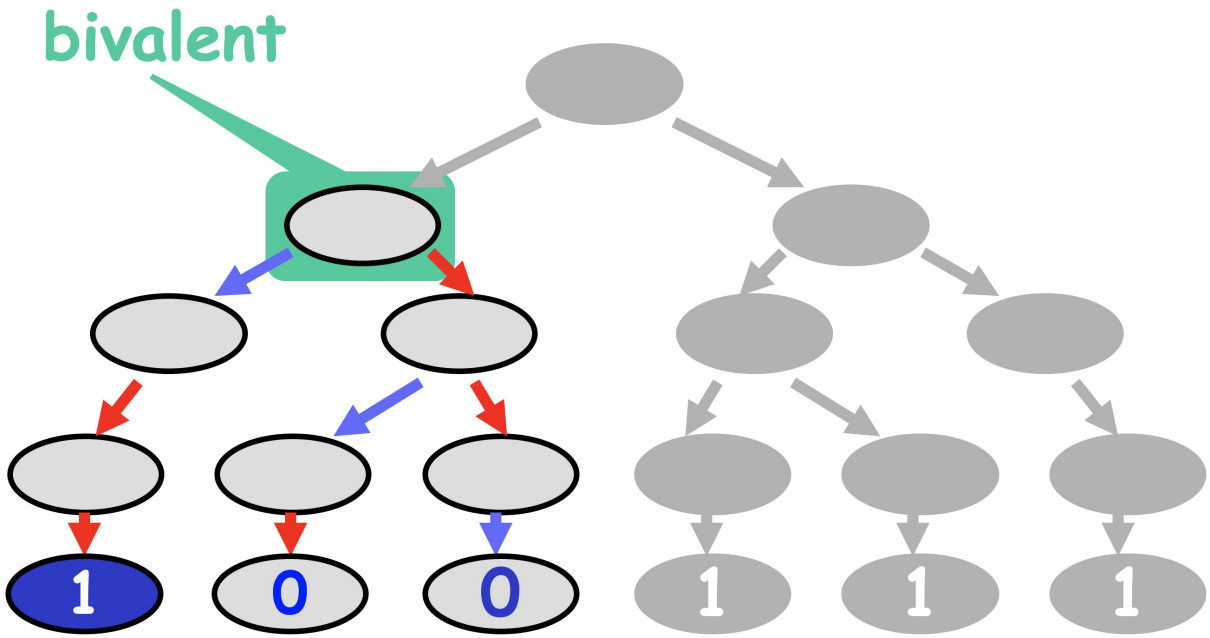
Decision Values



Univalent States: Single Value Possible



Bivalent Statues: Both Values Possible



Proof Part 1: Some initial state is bivalent

If both processors input 0:

- All executions must decide on 0
- Including the solo execution by process A

Proof Part 1: Some initial state is bivalent

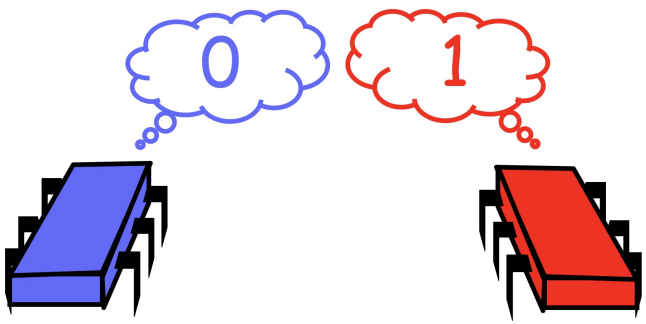
If both processors input 0:

- All executions must decide on 0
- Including the solo execution by process A

If both processors input 1:

- All executions must decide on 1
- Including the solo execution by process B

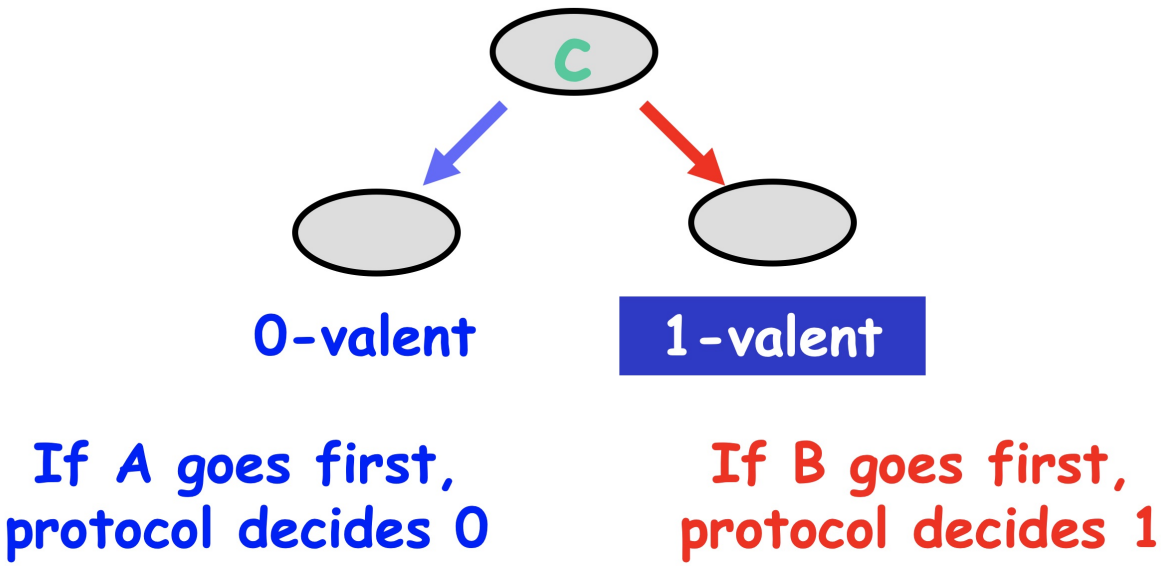
Proof Part 1: Some initial state is bivalent



If the inputs differ:

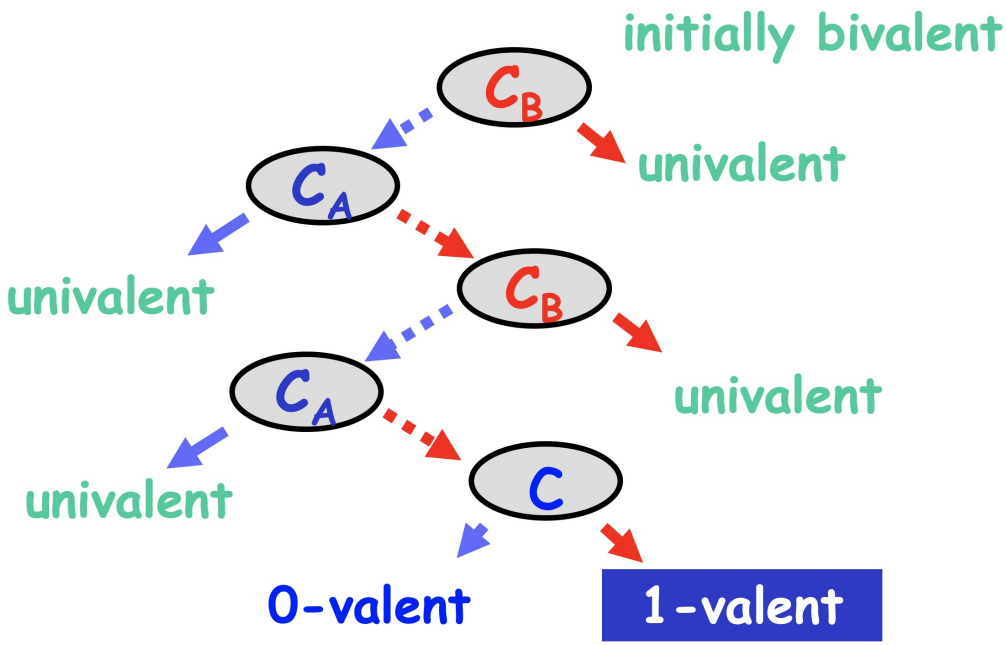
- Solo execution of A decides 0
- Solo execution of B decides 1
- Bivalent state!

Critical State



Proof Part 2: Some state must be a critical state

- Starting from a bivalent initial state, the protocol will reach a critical state
  - Otherwise we could stay bivalent forever



Proof Part 3: Properties of read-write registers

Lets look at executions that:

- Start from a critical state
- In which processes cause state to become univalent by next step, that is reading or writing to same/different registers
- End within a finite number of steps deciding either 0 or 1

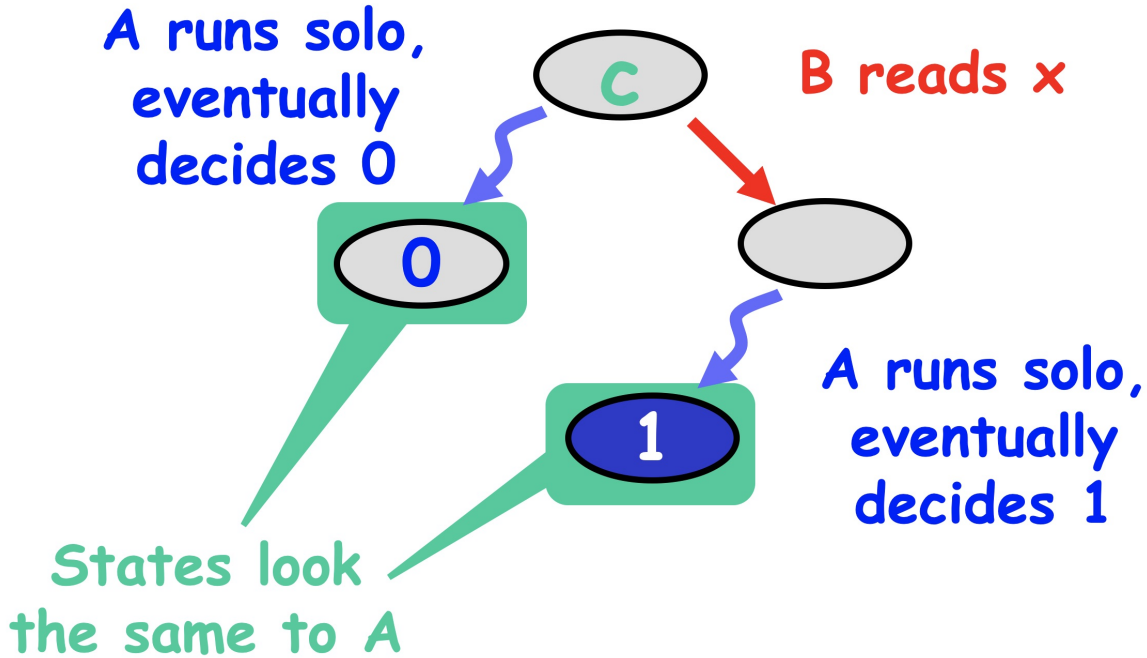
Show that there are no critical states!

$\Rightarrow$ Contradiction

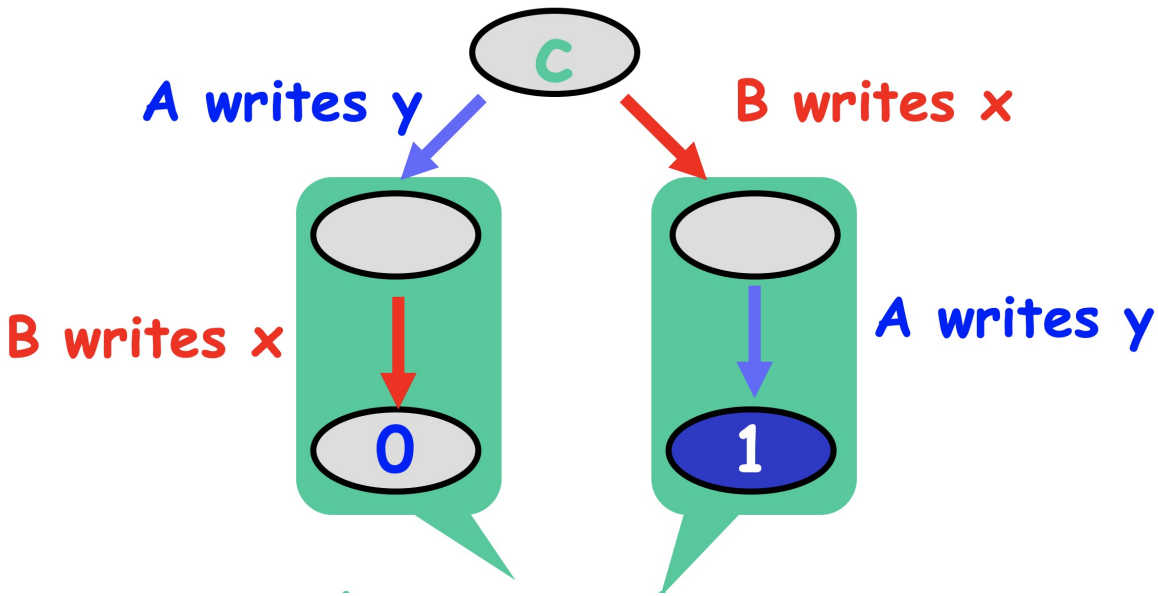
Possible Interactions

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

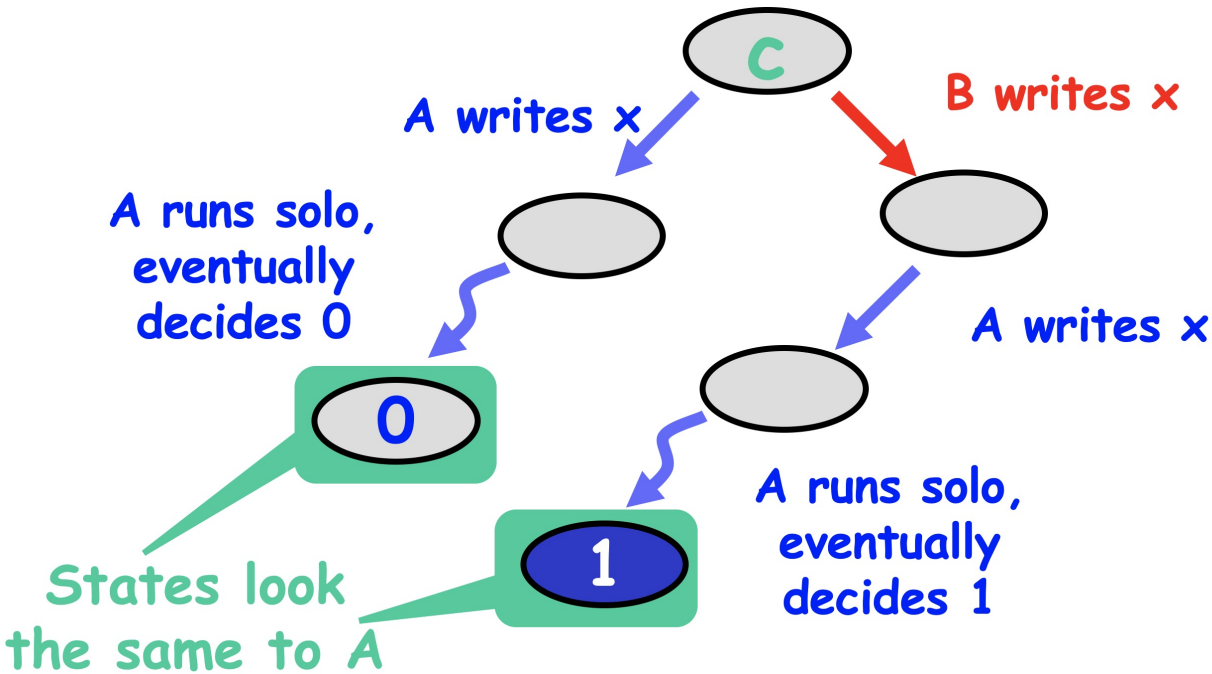
Case 1: Some Thread Reads



Case 2: Threads write distinct registers



Case 3: Threads write same register



Implications

Corollary

It is impossible to implement a two-dequeuer wait-free FIFO queue from read/write registers.

Measuring Synchronization Power

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

If you can implement X from Y and X has consensus number c, then Y has consensus number at least c.

Consensus Protocol for N threads and integer values

```
// For N threads:

abstract class ConsensusProtocol {

  protected int[] proposed = new int[N];

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

  abstract int decide(int value);

}
```

Read-Modify-Write Registers

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

  // here: synchronized indicates atomic execution of all
    instructions in the method; actually implemented using a
    hardware primitive
  public synchronized int getAndMumble() {
    // return prior value
    int prior  = this.value;

    // apply function to current value and replace
    this.value = mumble(this.value);

    return prior;
  }

}
```

Example: getAndSet

```
abstract class RMWRegister {

  private int value;

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

Example: getAndIncrement

```
abstract class RMWRegister {

  private int value;

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

Example: getAndAdd

```
abstract class RMWRegister {

  private int value;

  public synchronized int getAndAdd(int a) {
    int prior  = this.value;
    this.value = this.value + a;
    return prior;
  }
  ...
}
```

Example: get

```
abstract class RMWRegister {

  private int value;

  public synchronized int get() {
    int prior  = this.value;
    // this.value = this.value;
    return prior;
  }
  ...
}
```

Example: compareAndSet

```
abstract class RMWRegister {

  private int value;

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

Definition

An RMW method with function `mumble(x)` is **non-trivial** if there exists a value `v` such that `v != mumble(v)`.

Definition

An RMW method with function `mumble(x)` is **non-trivial** if there exists a value `v` such that `v != mumble(v)`.

Example:

- `getAndIncrement` is

Definition

An RMW method with function `mumble(x)` is **non-trivial** if there exists a value `v` such that `v != mumble(v)`.

Example:

- `getAndIncrement` is non-trivial
- `get` is

Definition

An RMW method with function `mumble(x)` is **non-trivial** if there exists a value `v` such that `v != mumble(v)`.

Example:

- `getAndIncrement` is non-trivial
- `get` is trivial

Theorem

Any non-trivial RMW object has consensus number at least 2.

Proof Sketch: Implementing 2-thread consensus with RMW

```
class RMWConsensus extends ConsensusProtocol {

  // x is arbitrary fixed value with mumble(x) != x
  private RMWRegister r = new RMWRegister(x);

  public int decide(int value) {
    propose(value);

    // first thread reaching the getAndMumble reads x
    if (r.getAndMumble() == x)
      return proposed[ThreadID.get()];
    else
      return proposed[1 - ThreadID.get()];
  }
}
```

Proof Sketch: Implementing 2-thread consensus with RMW

```
class RMWConsensus extends ConsensusProtocol {

  // x is arbitrary fixed value with mumble(x) != x
  private RMWRegister r = new RMWRegister(x);

  public int decide(int value) {
    propose(value);

    // first thread reaching the getAndMumble reads x
    if (r.getAndMumble() == x)
      return proposed[ThreadID.get()];
    else
      return proposed[1 - ThreadID.get()];
  }
}
```

Question

Why doesn't this construction work for more than 2 threads?

Interfering RMW

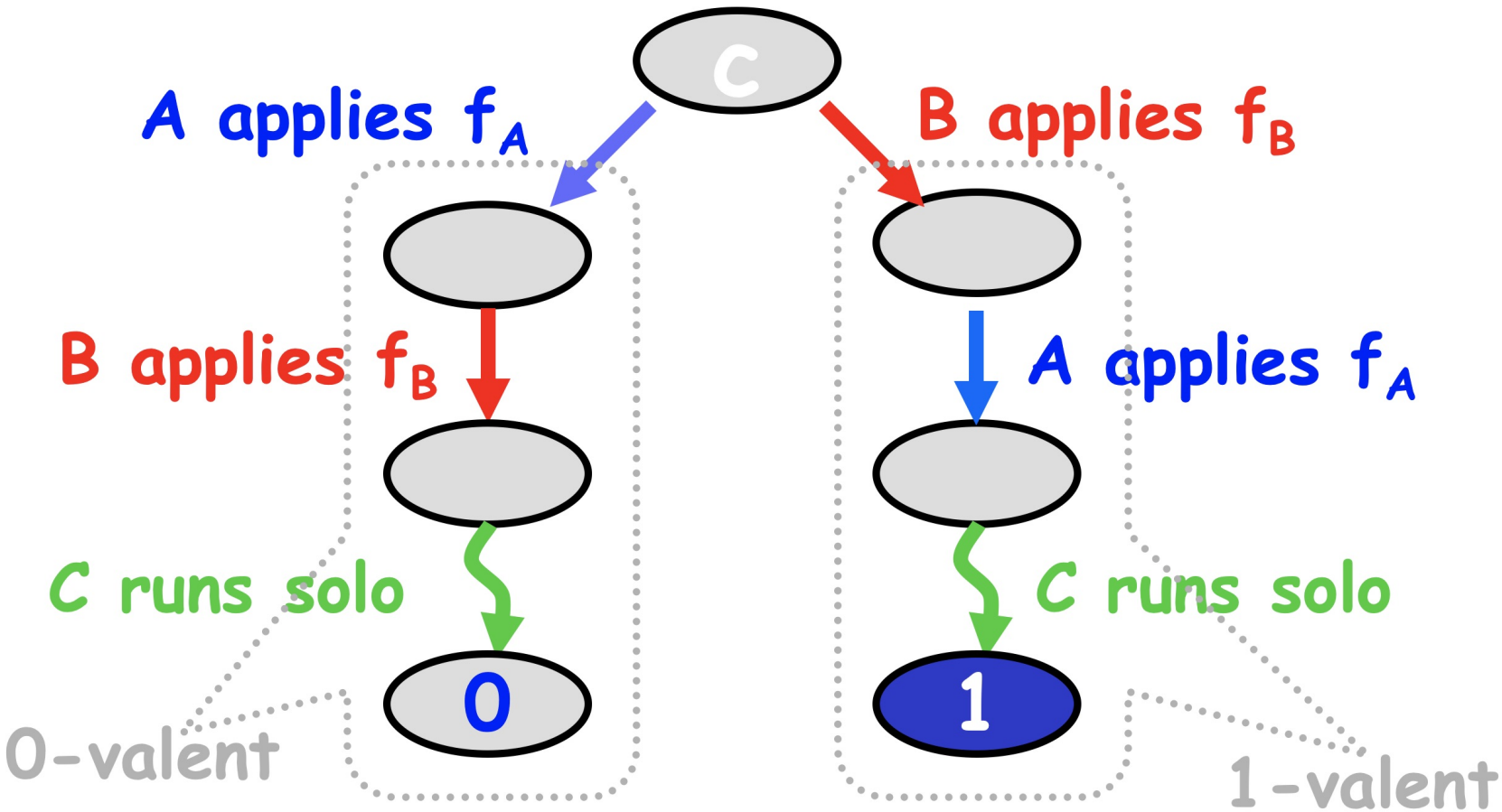
Let F be a set of functions such that for all $f_i, f_j \in F$ either

- commute: $f_i(f_j(v)) = f_j(f_i(v))$
- overwrite: $f_i(f_j(v)) = f_i(v)$

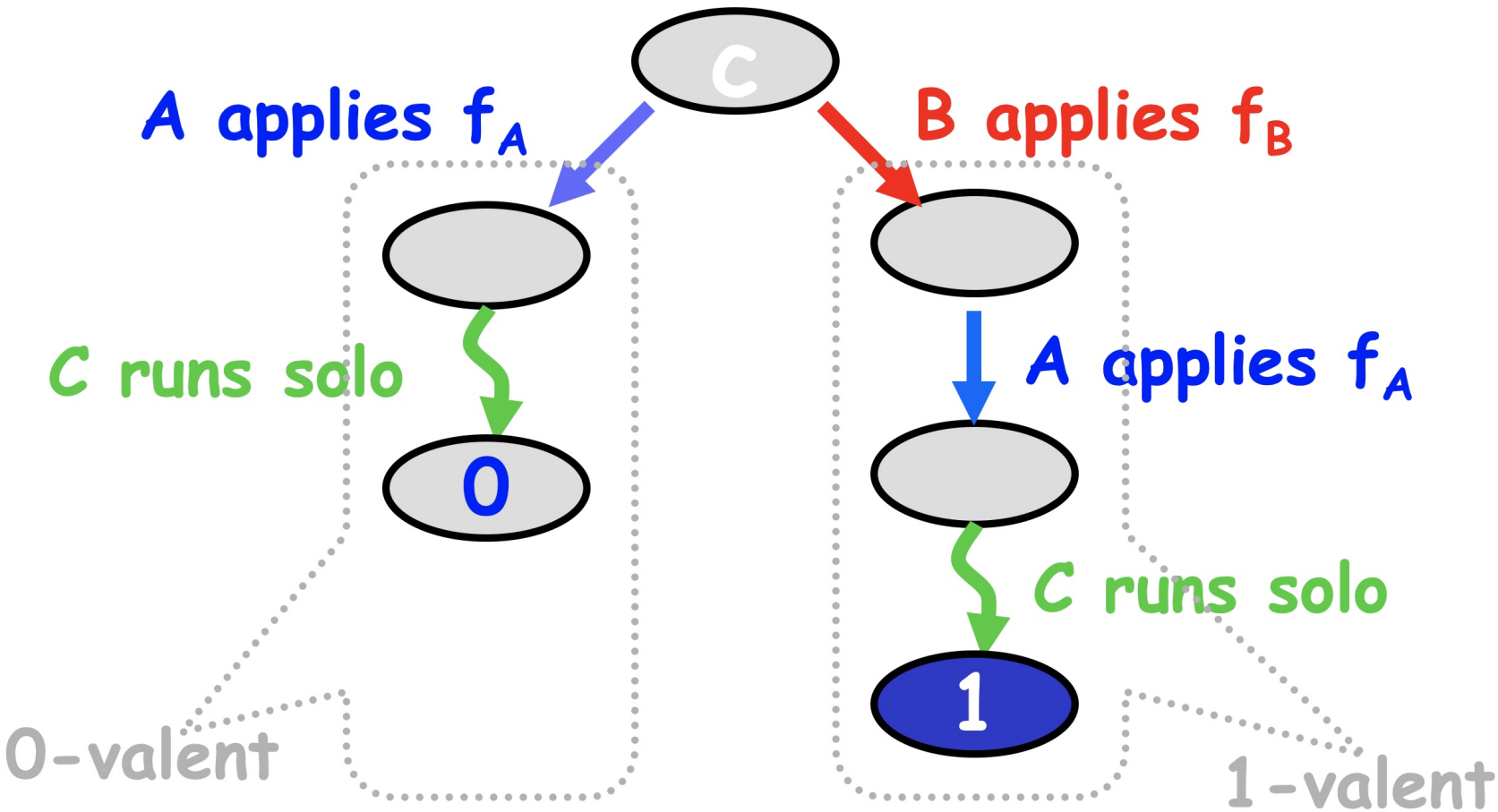
Theorem

Any set of RMW objects with mumble function that commutes or overwrites has consensus number exactly 2.

Proof sketch: Commuting functions



Proof sketch: Overwriting functions



Your turn!

- Using the results so far, derive the consensus numbers for the following objects!
- Justify your answer!

1. Register with getAndIncrement operation
2. Register with getAndSet operation
3. Register with compareAndSwap operation

compareAndSet has Consensus Number ∞

- Construct consensus protocol for **any** number of threads
- Assumption here is that the `AtomicInteger` is not restricted as in Java, but can represent any integral number

```
class RMWConsensus extends ConsensusProtocol {

  private AtomicInteger r = new AtomicInteger(-1);

  int decide(int value) {
    propose(value);

    // first thread executing compareAndSet puts its thread id r
    r.compareAndSet(-1, ThreadID.get());
    return proposed[r.get()];
  }

}
```

Fun Fact: Every consensus number has an object!

- Atomic k-assignment solves consensus for 2k-2 threads
- Can be extended to odd numbers

Impact of these Results

- Many early machines provided these "weak" RMW instructions (i.e. with consensus number ≤ 2)
  - 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?

Theorem: Consensus is Universal!(Herlihy 1991)

From n-process consensus, we can construct a

- wait-free/lock-free
- linearizable
- n-threaded implementation
- of **any** sequentially specified object!

$\Rightarrow$ Next lecture!

Summary

- Consensus problem as classical concurrent problem
- Fundamental impossibility result by Fischer-Lynch-Paterson (adapted by Maurice Herlihy)
- Hierarchy of synchronization operations based on consensus numbers

Further reading

Fischer, Michael J., Nancy A. Lynch, and Mike Paterson. 1985. "Impossibility of Distributed Consensus with One Faulty Process." *J. ACM* 32 (2): 374–82. https://doi.org/10.1145/3149.214121.

Herlihy, Maurice. 1991. "Wait-Free Synchronization." *ACM Trans. Program. Lang. Syst.* 13 (1): 124–49. https://doi.org/10.1145/114005.102808.