

# Consensus Numbers

魏恒峰

hfwei@nju.edu.cn

2017 年 12 月 14 日



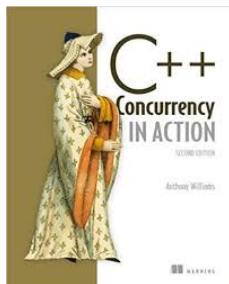
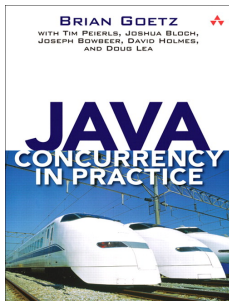




## Concurrent Programming?



## Synchronization



## Synchronization Primitives

Lock  
Semaphore

BlockingQueue  
ConcurrentMap

Phaser  
Barrier

Lock  
Semaphore

BlockingQueue  
ConcurrentMap

Phaser  
Barrier

```
class AtomicInteger: // java.util.concurrent
    get()             // read
    set(int newVal)   // write

    getAndIncrement()
    getAndDecrement()
    getAndSet(int newVal)

    compareAndSet(int expect, int update)
```

Lock  
Semaphore

BlockingQueue  
ConcurrentMap

Phaser  
Barrier

```
class AtomicInteger: // java.util.concurrent
    get()             // read
    set(int newVal)   // write

    getAndIncrement()
    getAndDecrement()
    getAndSet(int newVal)

    compareAndSet(int expect, int update) // atomically
```





(compareAndSet    compareAndSwap)



(compareAndSet    compareAndSwap)

```
// 'appears to' be atomic
compareAndSet(int expect, int update) {
    old = this.val

    if (old == expect)
        this.val = update

    return old
}
```

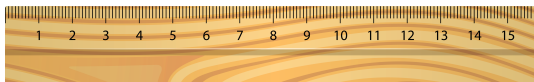


(Relative) *power* of various synchronization primitives

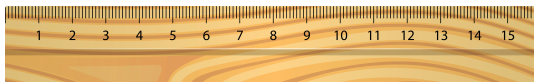
## *Power* measured by the *consensus* problem



*Q* : Is it possible to solve the consensus problem using this synchronization primitive?



*Consensus number:*  $n$ -thread consensus problems



*Consensus number:*  $n$ -thread consensus problems



1, 2,  $n$ ,  $\dots$   $\infty$

## Definition (Consensus Number)

The *consensus number* of a class  $C$  of synchronization primitive is the largest  $n$  for which  $C$  *solves*<sup>a</sup>  $n$ -thread consensus.

If no largest  $n$  exists, the consensus number is said to be *infinite*.

---

<sup>a</sup>defined later

# The beautiful idea of “Consensus Numbers”.

(Maurice Herlihy@TOPLAS'1991)

## Wait-Free Synchronization

MAURICE HERLIHY  
Digital Equipment Corporation

---

A *wait-free* implementation of a concurrent data object is one that guarantees that any process can complete any operation in a finite number of steps, regardless of the execution speeds of the other processes. The problem of constructing a wait-free implementation of one data object from another lies at the heart of much recent work in concurrent algorithms, concurrent data structures, and multiprocessor architectures. First, we introduce a simple and general technique, based on reduction to a consensus protocol, for proving statements of the form, “there is no wait-free implementation of  $X$  by  $Y$ .” We derive a hierarchy of objects such that no object at one level has a wait-free implementation in terms of objects at lower levels. In particular, we show that atomic read/write registers, which have been the focus of much recent attention, are at the bottom of the hierarchy: they cannot be used to construct wait-free implementations of many simple and familiar data types. Moreover, classical synchronization primitives such as *test&set* and *fetch&add*, while more powerful than *read* and *write*, are also computationally weak, as are the standard message-passing primitives. Second, nevertheless, we show that there do exist simple universal objects from which one can construct a wait-free implementation of any sequential object.



## Theorem (Consensus Numbers)

Consensus Number	Object
1	read/write registers
2	test&set, swap, fetch&add, queue, stack
$\vdots$	$\vdots$
$2n - 2$	$n$ -register assignment
$\vdots$	$\vdots$
$\infty$	memory-to-memory move and swap, augmented queue, compare&swap, fetch&cons, sticky byte

## Theorem (Consensus Numbers)

Consensus Number	Object
1	read/write registers
2	test&set, swap, fetch&add, queue, stack
$\vdots$	$\vdots$
$2n - 2$	$n$ -register assignment
$\vdots$	$\vdots$
$\infty$	memory-to-memory move and swap, augmented queue, compare&swap, fetch&cons, sticky byte

$\infty$  : CAS

1 : Atomic read/write register

2 : Queue, getAndSet

# Consensus



“It looks like we have a consensus.”

**Propose**



**Decide**

Propose



Decide

### Definition (The Consensus Problem)

**Consistent** All (**non-faulty**) threads must *agree* on the same value.

**Valid** The common decision value must be the value *proposed* by some thread.

**Wait-free** Each (**non-faulty**) thread must decide in a *finite* number of steps.

# On the Nature of Progress

Maurice Herlihy<sup>1,\*</sup> and Nir Shavit<sup>2</sup>

<sup>1</sup> Brown University Computer Science  
herlihy@cs.brown.edu

<sup>2</sup> MIT CSAIL  
shanir@csail.mit.edu



Wait-free (very informally):

... *regardless of* the execution speeds of other threads

# On the Nature of Progress

Maurice Herlihy<sup>1,\*</sup> and Nir Shavit<sup>2</sup>

<sup>1</sup> Brown University Computer Science  
herlihy@cs.brown.edu

<sup>2</sup> MIT CSAIL  
shanir@csail.mit.edu



Wait-free (very informally):

... *regardless of* the execution speeds of other threads

... *eventually* finishes when it runs *solo*



We next focus on the *binary* consensus problem.



# Consensus Protocol



## Definition (Consensus Protocol (Informally))

A *consensus protocol* is a system of  $n$  *threads* that communicate through a set of *shared objects*.

Propose

$$\{0, 1\}^n$$

Communicate

invocation

response

Decide

$$\{0, 1\}^n$$

## Definition (Consensus Protocol (Informally))

A *consensus protocol* is a system of  $n$  *threads* that communicate through a set of *shared objects*.

Propose

$$\{0, 1\}^n$$

Communicate

invocation

response

Decide

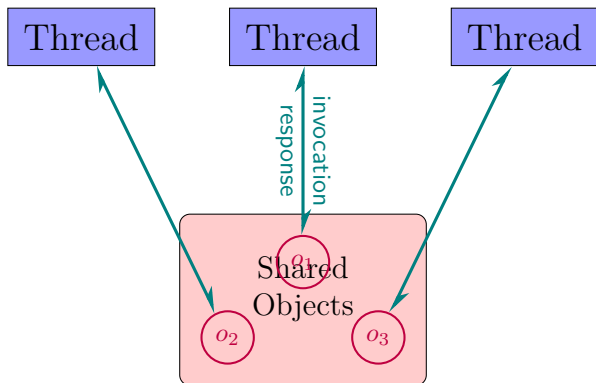
$$\{0, 1\}^n$$

It *correctly* solves the  $n$ -thread consensus problem:

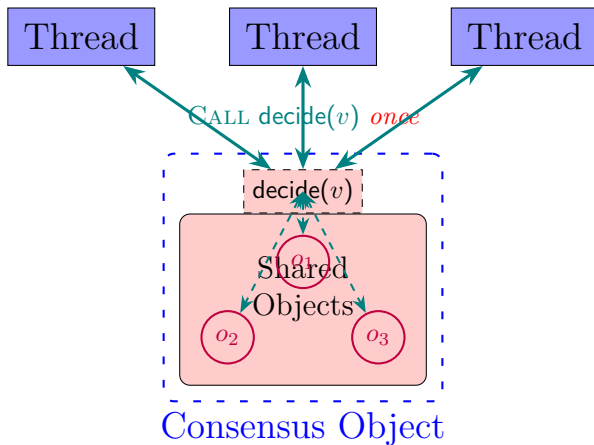
Consistent ...

Valid ...

Wait-free ...



Threads communicate through shared objects.



```
interface Consensus {  
    int decide(int val);  
}
```

## Definition (“Solves”)

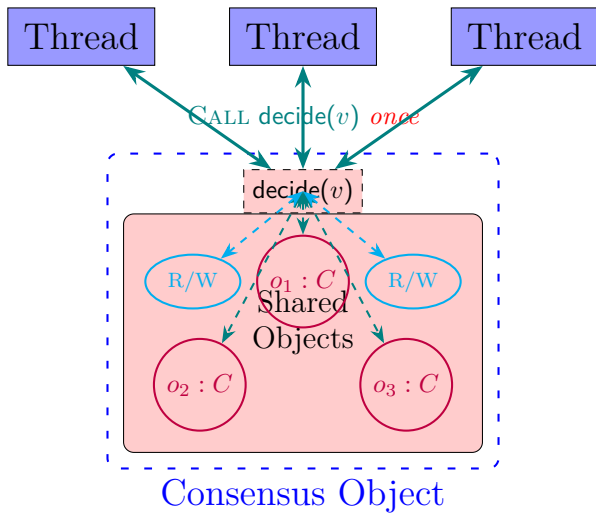
A *class*  $C$  of synchronization primitive *solves*  $n$ -thread consensus if there exist a *consensus protocol* using

- (i) any number of objects of class  $C$
- (ii) any number of atomic read/write registers

## Definition (“Solves”)

A *class*  $C$  of synchronization primitive *solves*  $n$ -thread consensus if there exist a *consensus protocol* using

- (i) any number of objects of class  $C$
- (ii) any number of atomic read/write registers
- (iii) initialized to any state

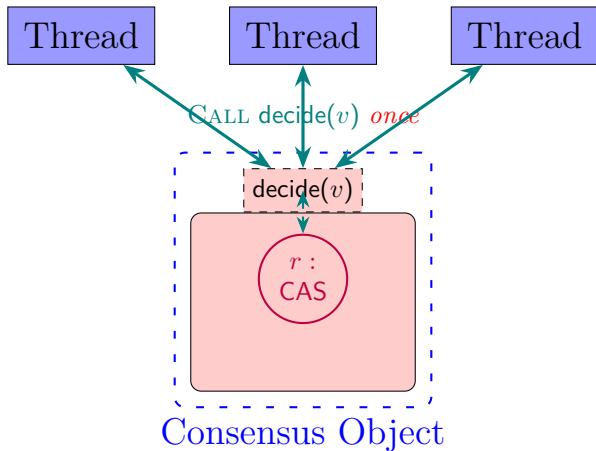


A class  $C$  solves  $n$ -thread consensus.





# $n$ -Consensus Protocol Using (a Single) CAS



```

class CASConsensus implements Consensus {
    // shared CAS object
    CAS r = new CAS( $\perp$ ); // initialized

    int decide(int val) {
        first = r.compareAndSet( $\perp$ , val);

        if (first ==  $\perp$ )
            return val;    // I won
        else
            return first;  // I lose
    }
}

```

```

class CASConsensus implements Consensus {
    // shared CAS object
    CAS r = new CAS( $\perp$ ); // initialized

    int decide(int val) {
        first = r.compareAndSet( $\perp$ , val);

        if (first ==  $\perp$ )
            return val;    // I won
        else
            return first;  // I lose
    }
}

```

The decision value is established by the thread that *succeeds*.

## Theorem (Power of CAS)

A **CAS** register providing the **compareAndSet()** method can solve the consensus problem for **any number** of threads.

## Theorem (Power of CAS)

A **CAS** register providing the **compareAndSet()** method can solve the consensus problem for **any number** of threads.

## Theorem (Consensus Number of CAS)

A **CAS** register providing the **compareAndSet()** method has the consensus number  $\infty$ .







## Theorem (Consensus Number of Atomic Registers)

*Atomic read/write registers have consensus number 1.*

## Theorem (Consensus Number of Atomic Registers)

*Atomic read/write registers have consensus number 1.*



## Theorem (“Weakness” of Atomic Registers)

*Atomic read/write registers **cannot** solve 2-thread consensus.*

# Impossibility Results

# Impossibility Results

By contradiction

$\exists \mathcal{P}$  solves 2-thread consensus

# Impossibility Results

By contradiction

$\exists \mathcal{P}$  solves 2-thread consensus

Consistent

Valid

Wait-free

# Impossibility Results

By contradiction

$\exists \mathcal{P}$  solves 2-thread consensus

Consistent

Valid

Wait-free

(eventually finishes when it runs *solo*)

## Protocol

$$\mathcal{P} = \{\text{All executions of this protocol}\}$$

## Protocol

$$\mathcal{P} = \{\text{All executions of this protocol}\}$$

## Execution

$$e = \sigma_0 \xrightarrow{o_1} \sigma_1 \xrightarrow{o_2} \cdots \xrightarrow{o_{n-1}} \sigma_n$$



## Protocol

$$\mathcal{P} = \{\text{All executions of this protocol}\}$$

## Execution

$$e = \sigma_0 \xrightarrow{o_1} \sigma_1 \xrightarrow{o_2} \cdots \xrightarrow{o_{n-1}} \sigma_n$$

## State

$\sigma_i$  : states of individual threads + states of shared objects

## Protocol

$$\mathcal{P} = \{\text{All executions of this protocol}\}$$

## Execution

$$e = \sigma_0 \xrightarrow{o_1} \sigma_1 \xrightarrow{o_2} \cdots \xrightarrow{o_{n-1}} \sigma_n$$

## State

$\sigma_i$  : states of individual threads + states of shared objects

## Operation

$o_i$  : a method call to some shared object

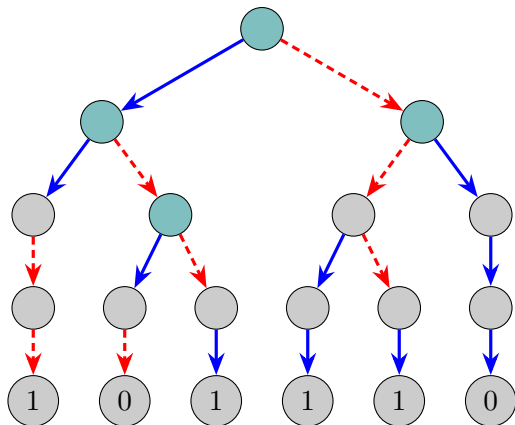
$$e = \sigma_0 \xrightarrow{o_1} \sigma_1 \xrightarrow{o_2} \cdots \xrightarrow{o_{n-1}} \sigma_n$$

- (i) **Sequential execution**: alternates matching invocations and responses.
- (ii) Only interleaved at the granularity of **complete method calls**.

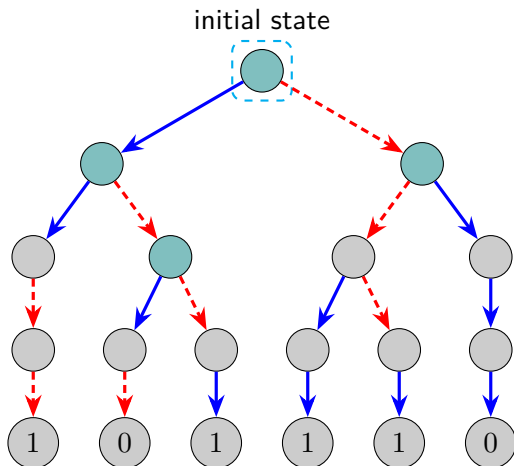
### Theorem

*A consensus protocol is correct iff all its **sequential** executions are correct.*

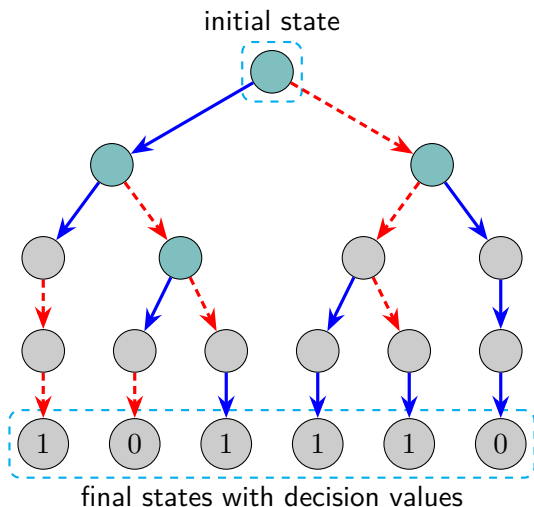
## Modeling $\mathcal{P}$ as an Execution “Tree” (Binary Consensus for 2 Threads)



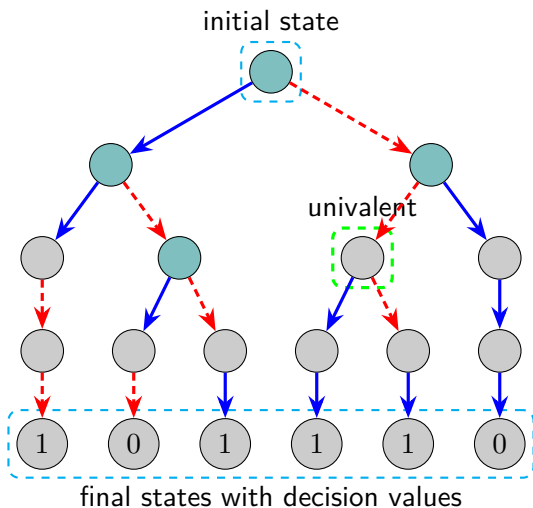
# Modeling $\mathcal{P}$ as an Execution “Tree” (Binary Consensus for 2 Threads)



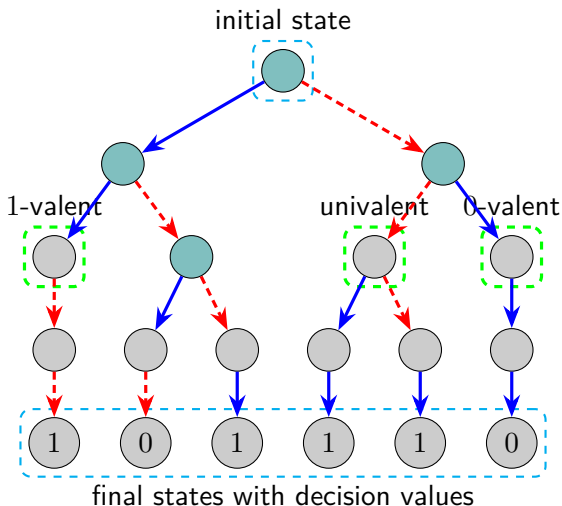
# Modeling $\mathcal{P}$ as an Execution “Tree” (Binary Consensus for 2 Threads)



# Modeling $\mathcal{P}$ as an Execution “Tree” (Binary Consensus for 2 Threads)

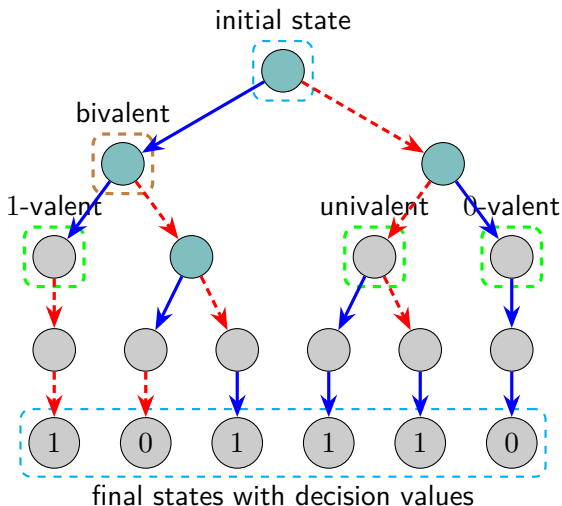


# Modeling $\mathcal{P}$ as an Execution “Tree” (Binary Consensus for 2 Threads)

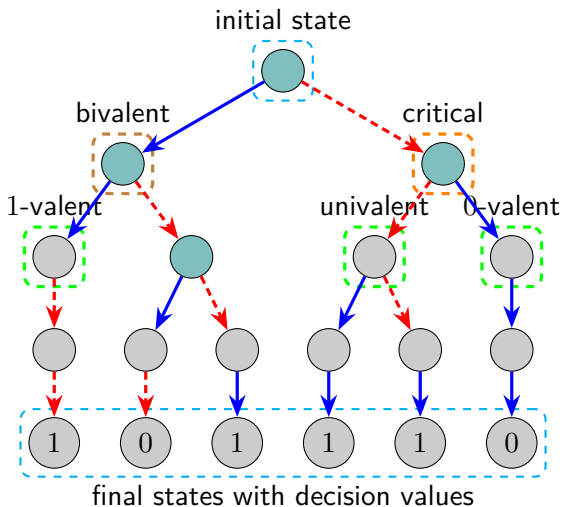




## Modeling $\mathcal{P}$ as an Execution “Tree” (Binary Consensus for 2 Threads)



# Modeling $\mathcal{P}$ as an Execution “Tree” (Binary Consensus for 2 Threads)



## Theorem (Bivalent Initial State)

*Every 2-thread **binary** consensus protocol has a **bivalent initial** state.*

## Theorem (Bivalent Initial State)

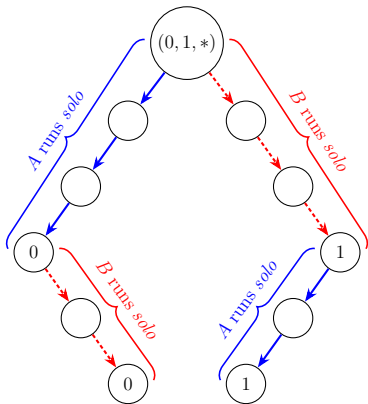
Every 2-thread *binary* consensus protocol has a *bivalent initial* state.

$$(A, B, O) : (0, 0, *) \quad (1, 1, *) \quad (0, 1, *) \quad (1, 0, *)$$

## Theorem (Bivalent Initial State)

Every 2-thread *binary* consensus protocol has a *bivalent initial* state.

$$(A, B, O) : (0, 0, *) \quad (1, 1, *) \quad (0, 1, *) \quad (1, 0, *)$$



## Theorem (Bivalent Initial State)

Every  $n$ -thread consensus protocol has a *bivalent initial* state.

## Theorem (Bivalent Initial State)

*Every  $n$ -thread consensus protocol has a **bivalent initial** state.*

## Theorem (Existence of Critical States)

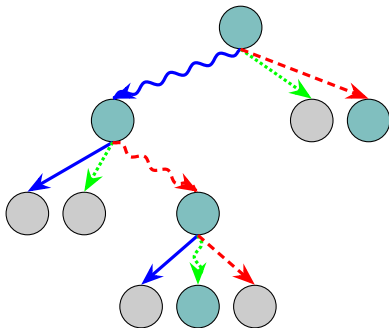
*Every wait-free consensus protocol has a **critical** state.*

## Theorem (Bivalent Initial State)

Every  $n$ -thread consensus protocol has a *bivalent initial* state.

## Theorem (Existence of Critical States)

Every wait-free consensus protocol has a *critical* state.









## Theorem (“Weakness” of Atomic Registers)

*Atomic read/write registers **cannot** solve 2-thread consensus.*

## Theorem (“Weakness” of Atomic Registers)

*Atomic read/write registers **cannot** solve 2-thread consensus.*

Proof.

(i) By contradiction:  $\exists \mathcal{P}$

## Theorem (“Weakness” of Atomic Registers)

*Atomic read/write registers **cannot** solve 2-thread consensus.*

Proof.

- (i) By contradiction:  $\exists \mathcal{P}$
- (ii) Run  $\mathcal{P}$  until it reaches a **critical** state  $s$ .

## Theorem (“Weakness” of Atomic Registers)

Atomic read/write registers *cannot* solve 2-thread consensus.

Proof.

- (i) By contradiction:  $\exists \mathcal{P}$
- (ii) Run  $\mathcal{P}$  until it reaches a **critical** state  $s$ .

*Q* : What are the next decision steps of two threads?

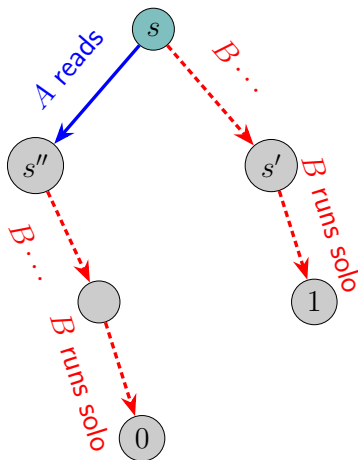


From the critical state  $s$ :

CASE 1  $A$  reads

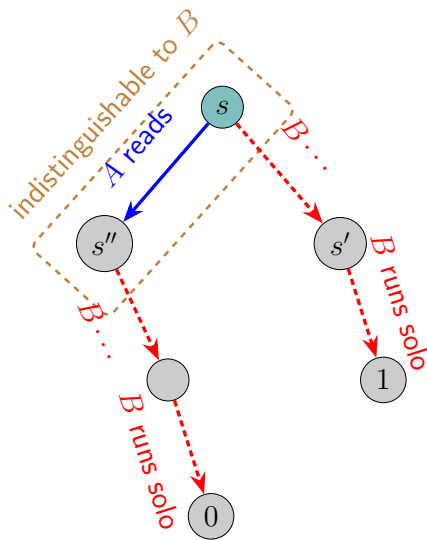
CASE 2  $A$  writes  $r_0$ ;  $B$  writes  $r_1$

CASE 3  $A$  writes  $r$ ;  $B$  writes  $r$

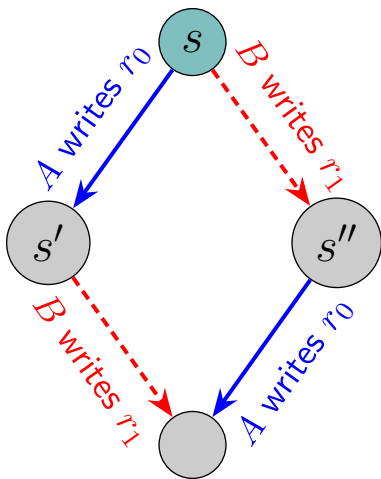


CASE 1: A reads

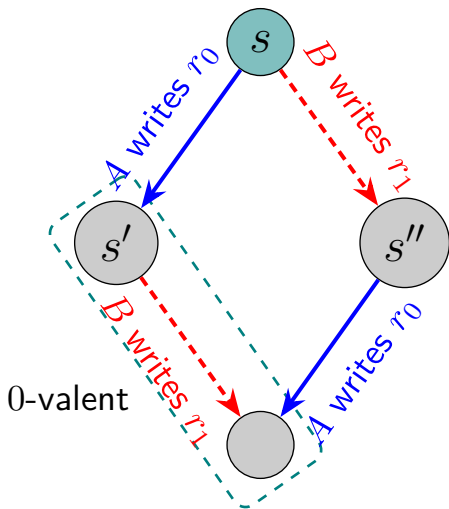




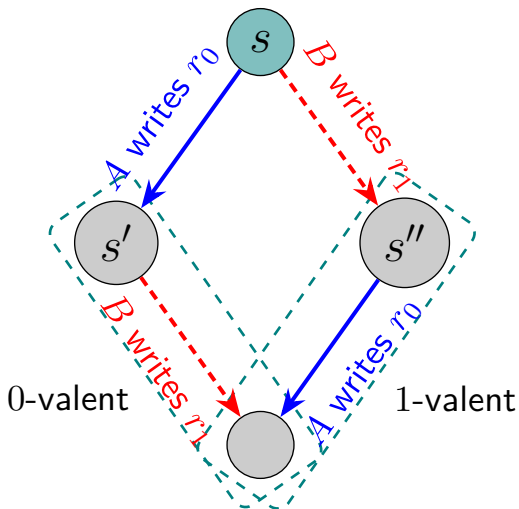
## CASE 1: A reads



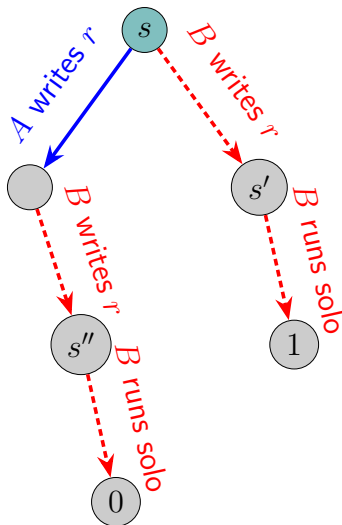
CASE 2:  $A$  writes  $r_0$ ;  $B$  writes  $r_1$



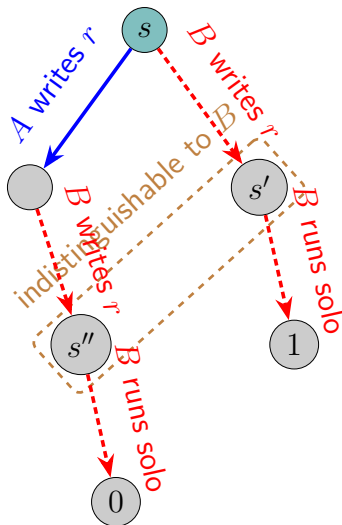
CASE 2:  $A$  writes  $r_0$ ;  $B$  writes  $r_1$



CASE 2:  $A$  writes  $r_0$ ;  $B$  writes  $r_1$



CASE 3:  $A$  writes  $r$ ;  $B$  writes  $r$



CASE 3:  $A$  writes  $r$ ;  $B$  writes  $r$

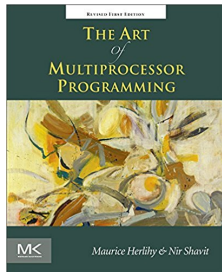
## Theorem (“Weakness” of Atomic Registers)

*It is **impossible** to construct a wait-free implementation of **any object with consensus number  $\geq 2$**  using atomic read/write registers.*

## Theorem (“Weakness” of Atomic Registers)

It is *impossible* to construct a wait-free implementation of *any object with consensus number  $\geq 2$*  using atomic read/write registers.

*“... is perhaps one of the most striking impossibility results in Computer Science.”*







## Theorem (Consensus Number 2)

*The data types **FIFO queue**, **stack**, **set**, **list**, and **priority queue** all have consensus number 2.*

## Theorem (Consensus Number 2 (More))

*The synchronization primitives **get&set** and **get&add** have consensus number 2.*

# References



# The Idea of “Consensus Numbers”. (Maurice Herlihy@TOPLAS'1991)

## Wait-Free Synchronization

MAURICE HERLIHY

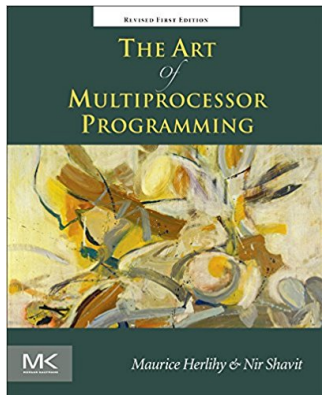
Digital Equipment Corporation

---

A *wait-free* implementation of a concurrent data object is one that guarantees that any process can complete any operation in a finite number of steps, regardless of the execution speeds of the other processes. The problem of constructing a wait-free implementation of one data object from another lies at the heart of much recent work in concurrent algorithms, concurrent data structures, and multiprocessor architectures. First, we introduce a simple and general technique, based on reduction to a consensus protocol, for proving statements of the form, “there is no wait-free implementation of  $X$  by  $Y$ .” We derive a hierarchy of objects such that no object at one level has a wait-free implementation in terms of objects at lower levels. In particular, we show that atomic read/write registers, which have been the focus of much recent attention, are at the bottom of the hierarchy: they cannot be used to construct wait-free implementations of many simple and familiar data types. Moreover, classical synchronization primitives such as *test&set* and *fetch&add*, while more powerful than *read* and *write*, are also computationally weak, as are the standard message-passing primitives. Second, nevertheless, we show that there do exist simple universal objects from which one can construct a wait-free implementation of any sequential object.

# “The Art of Multiprocessor Programming”.

(Maurice Herlihy & Nir Shavit@2008)



# The Idea of “Valency Argument”. (FLP@JACM'1985)

## Impossibility of Distributed Consensus with One Faulty Process

MICHAEL J. FISCHER

*Yale University, New Haven, Connecticut*

NANCY A. LYNCH

*Massachusetts Institute of Technology, Cambridge, Massachusetts*

AND

MICHAEL S. PATERSON

*University of Warwick, Coventry, England*

**Abstract.** The consensus problem involves an asynchronous system of processes, some of which may be unreliable. The problem is for the reliable processes to agree on a binary value. In this paper, it is shown that every protocol for this problem has the possibility of nontermination, even with only one faulty process. By way of contrast, solutions are known for the synchronous case, the “Byzantine Generals” problem.

Thank  
You!