Consensus and hierarchy of objects

System model

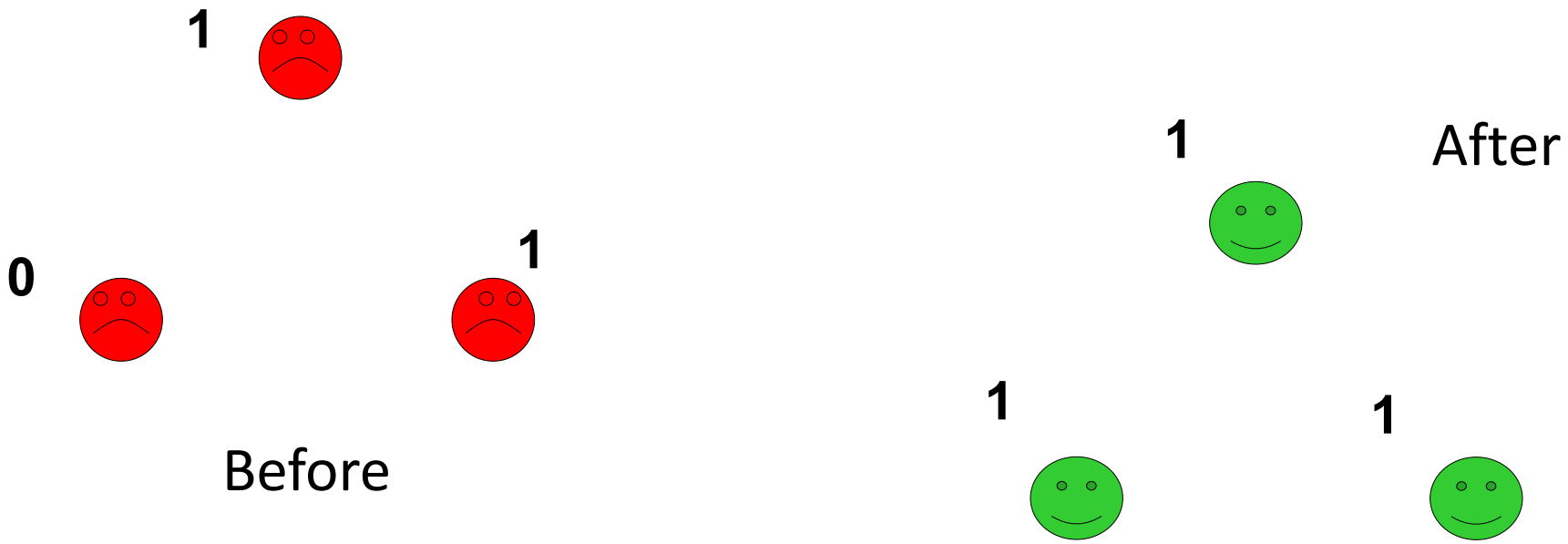
N *asynchronous* (no bounds on relative speeds) processes $p_0,\ldots,p_{N-1}$ ($N \geq 2$) communicate via atomic read-write registers

Processes can fail by crashing

- A crashed process takes only finitely many steps (reads and writes)
- Up to t processes can crash: t-resilient system
- t=N-1: wait-free

Consensus

Processes *propose* values and must *agree* on a common decision value so that the decided value is a proposed value of some process



Consensus: definition

A process *proposes* an *input* value in V (|V|≥2) and tries to *decide* on an *output* value in V

Agreement: No two processes decide on different values

Validity: Every decided value is a proposed value

Termination: No process takes infinitely many steps without deciding

(Every correct process decides)

0-resilient consensus?

Consider the case t=0, no process fails

Shared:

1 register D, initially $\perp$ (default value not in V)

Upon propose(v) by process p_i:

```
if i = 0 then D.write(v)          // if p0 decide on v
wait until D.read() ≠ ⊥           // wait until p0 decides
return D
```

(every process decides on p_0's input)

Impossibility of wait-free consensus [FLP85,LA87]

Theorem No algorithm solves *t-resilient (t≥1)* consensus (even binary) for n≥2 processes with atomic registers

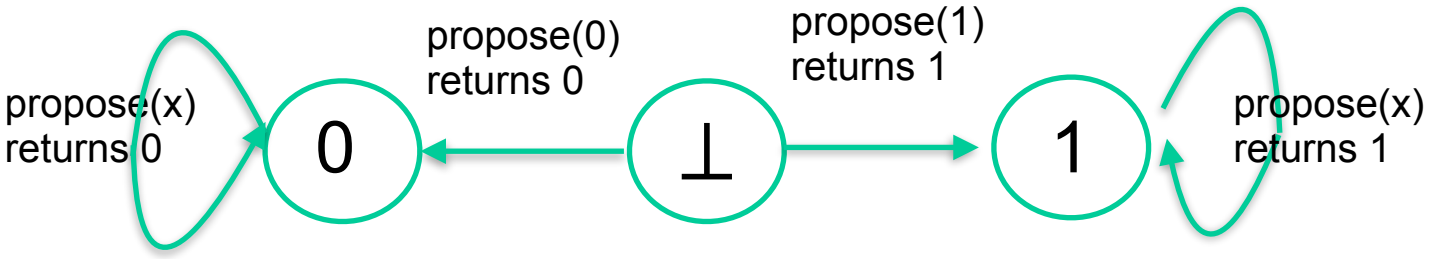
(For the moment we are going to prove that no *wait-free* algorithm solves consensus for 2 processes (and hence there is no wait-free algorithm for n (n≥2) processes))

Objects

- Shared Objects:
  - sequential specification + linearizability (ex: Register, Compare&swap, Stack …)
  - Wait-free implementation (every method call completes in finite number of steps)
  - How powerful are objects (concurrent data structures), like Test&Set, Compare&Swap, Queue, Stack …?
  - Use the consensus to compare objects

Consensus object

As (one-shot) object: method *propose(v)* returns the decided value:



n-Consensus

n-consensus (= consensus between n processes) to get a hierarchy of(wait-free) objects:

We will prove that there is no wait-free implementation of 2-consensus with atomic registers (or no implementation of 2-consensus object)

- But with Queue and registers is 2-consensus solvable?

consensus number: among how many processes an object (and registers) may solve Consensus?

Consensus number

- A class C of objects solves n-consensus if there exists a consensus protocol using any number of objects of class C and atomic registers for n processes
- The consensus number of a class C: (h(C)) is the largest n for which that class solves n-consensus.
- h(C) defines a hierarchy of objects:
  - If one can implement an object of class C from objects of class D (and registers) then $h(C) \leq h(D)$
  - (if $h(C) > h(D)$ there is no implementation of objects of C with objects of D).

Compare objects

- Theorem
  There is no wait-free implementation of n-consensus from read-write registers for n≥2

By contradiction, assume there is such for n=2

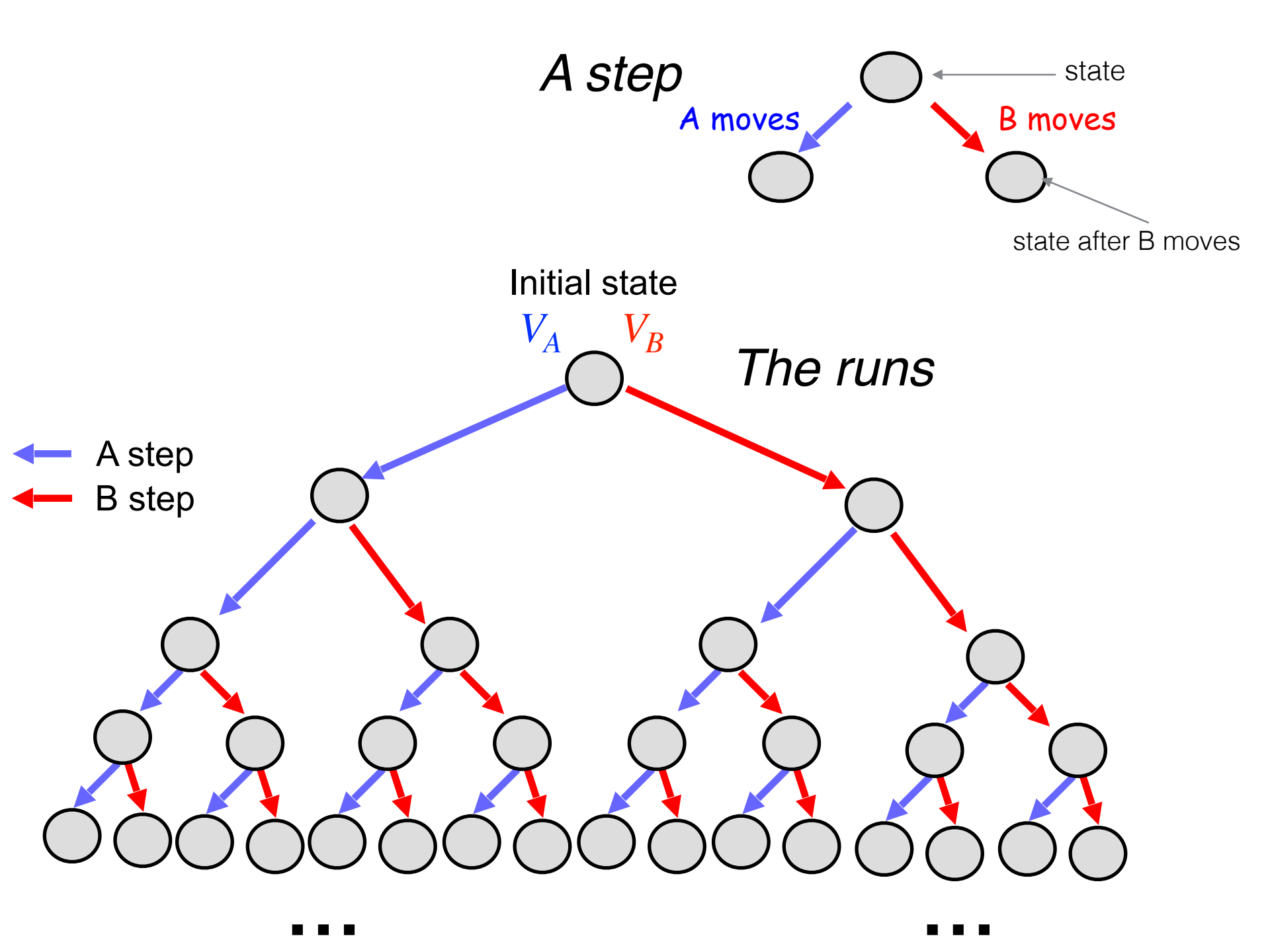
Model…

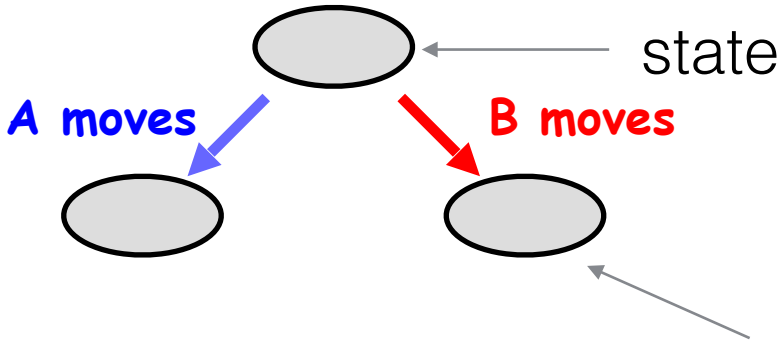
we are going to model the runs of an algorithm

assume only 2 processes (A and B)

assume binary consensus (initial value 0 or 1 only)

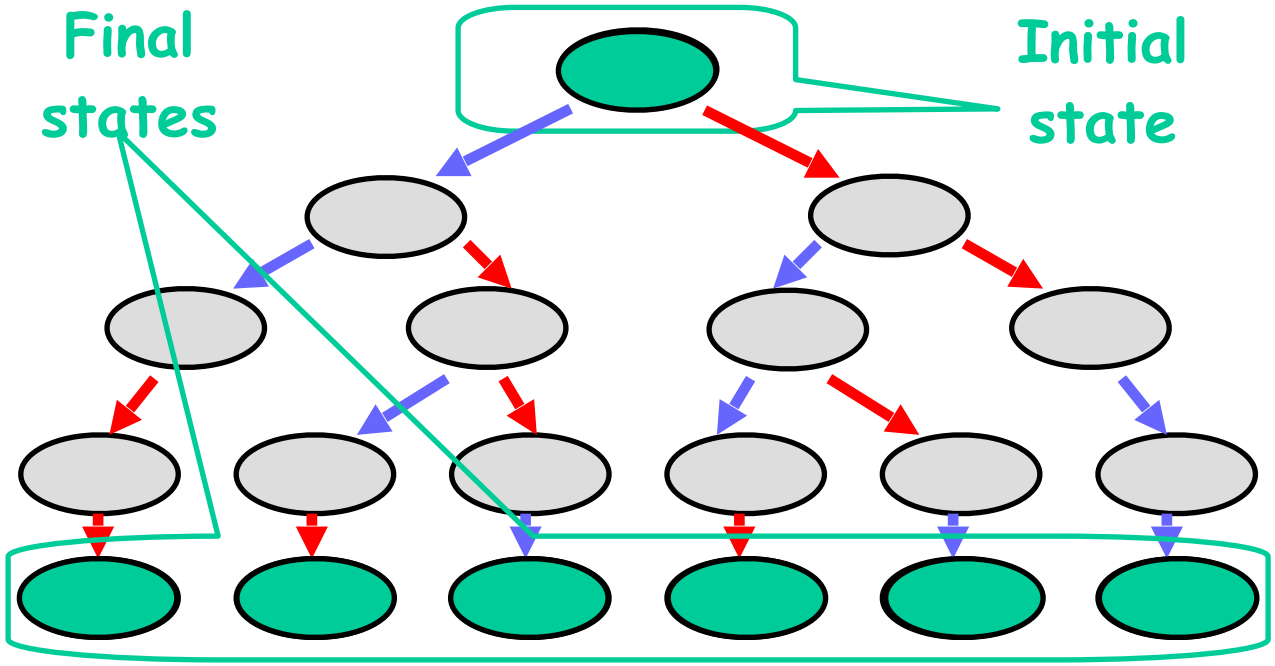
- « step » of a process (here read or write of some register)
- « configuration » (state of memory, state of process…)
- « run » of the algorithm: sequences of configurations + steps
- « schedule »: sequence of process id (wait-free: every sequence of A and B is a schedule)
- an initial configuration and a schedule define a run (algo is deterministic)

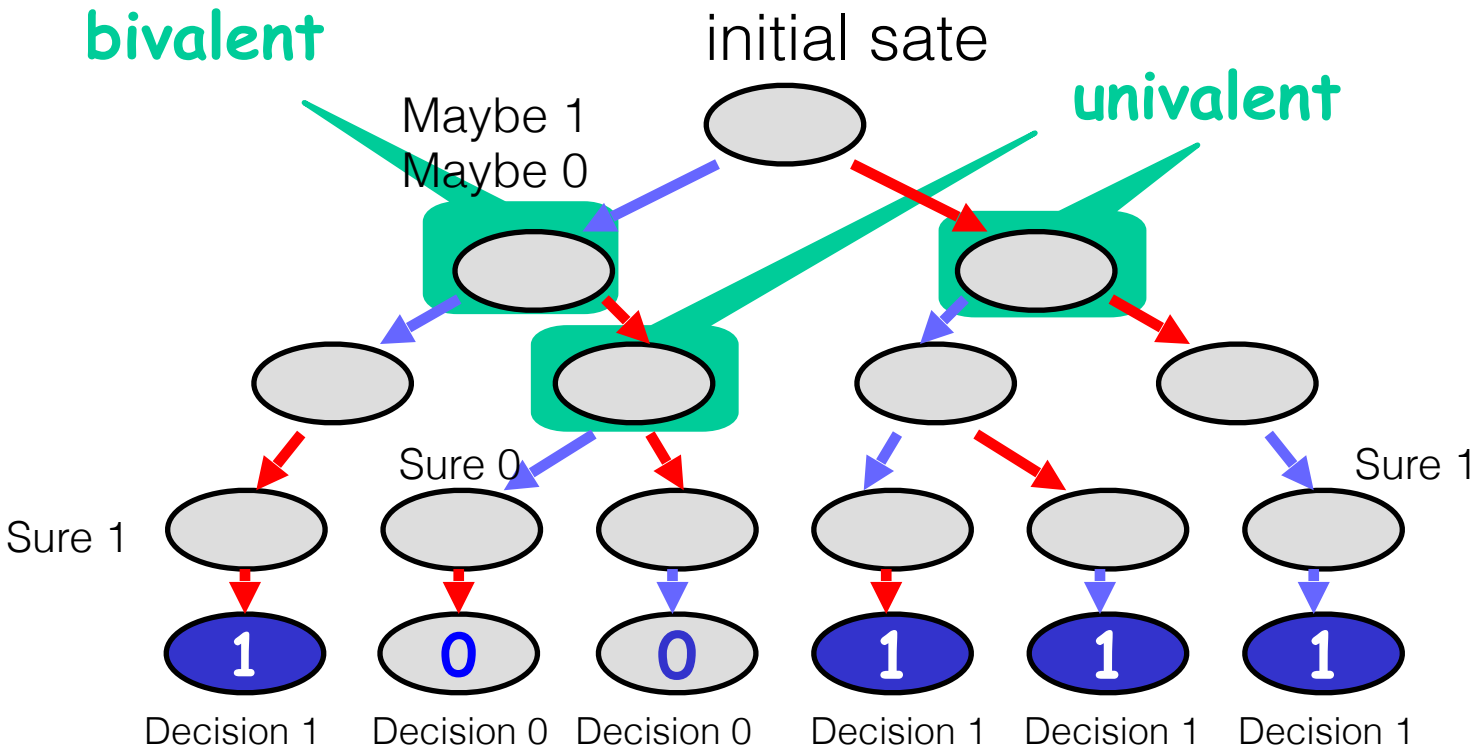

A step
state
A moves
B moves
state after B moves
Initial state
V_A
V_B
The runs
A step
B step



moves: atomic step

state after B moves





bivalent: from here 0 and 1 are possible

0-valent: from here only 0

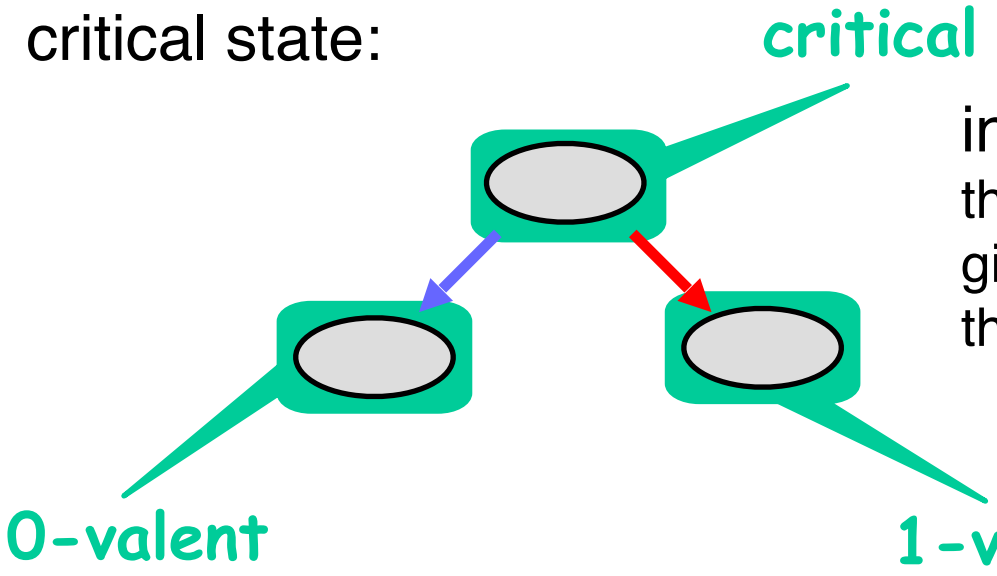
univalent: from here only one value

1-valent: from here only 1

Model

- Wait-free computation is a tree (a forest) (node: configurations link: steps) representing all possible runs from some initial configuration (from all initial configurations)
- (wait free means that in each configuration (exactly) one step of any process is possible)
- Configurations Valences:
  - in *Bivalent* configurations:
    - Outcome not fixed
  - in *Univalent* states
    - Outcome is fixed
    - (but may not be “known” yet)
  - *1-Valent* states = univalent for 1
  - 0-*Valent* states = univalent for 0

critical state:



in a critical state
the next step of any process gives a configuration in which the decision is fixed

Properties:

- Some initial state is bivalent
- If there is a consensus protocol then there is a critical state

There is an initial bivalent state?

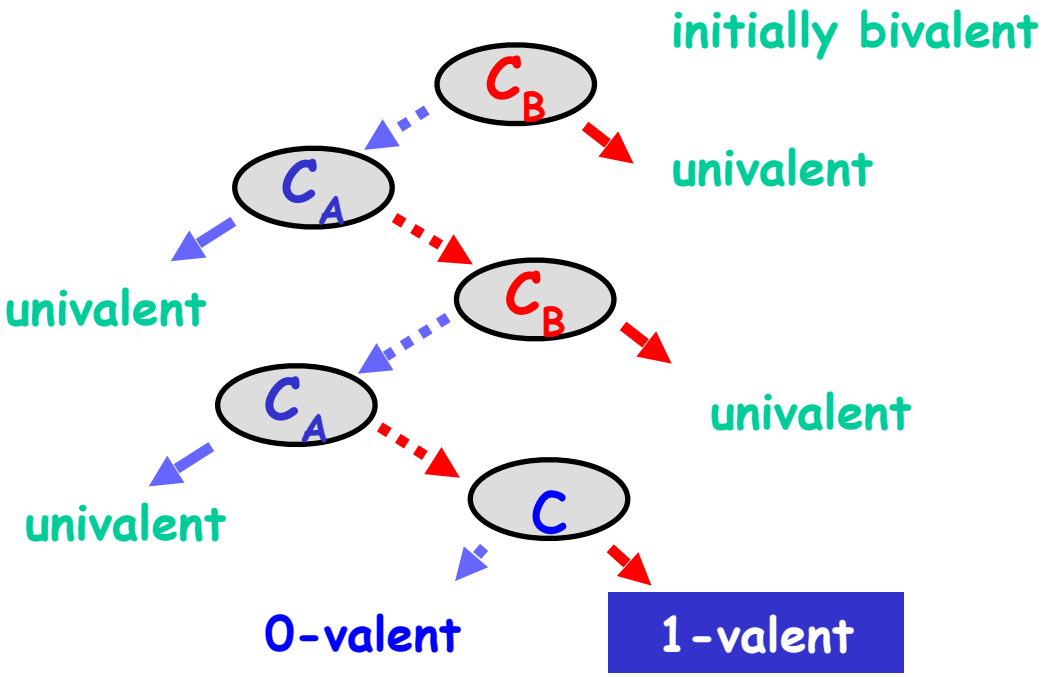
S:A initial value 0, B initial value 1

A solo decides (wait-free), A must decide 0 (the run is indistinguishable from run A alone for A and B initial have both value 0 as initial value)

Same argument B solo decides 1

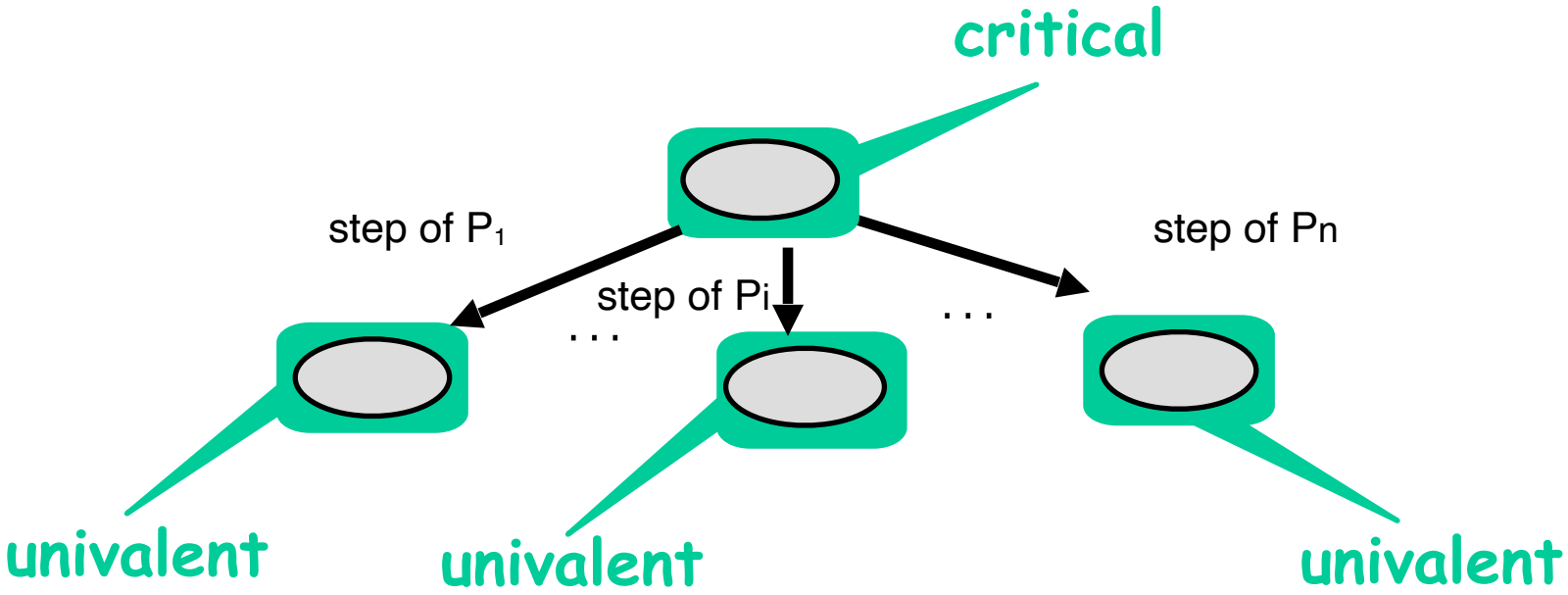
Then S is bivalent.

there is a critical state?

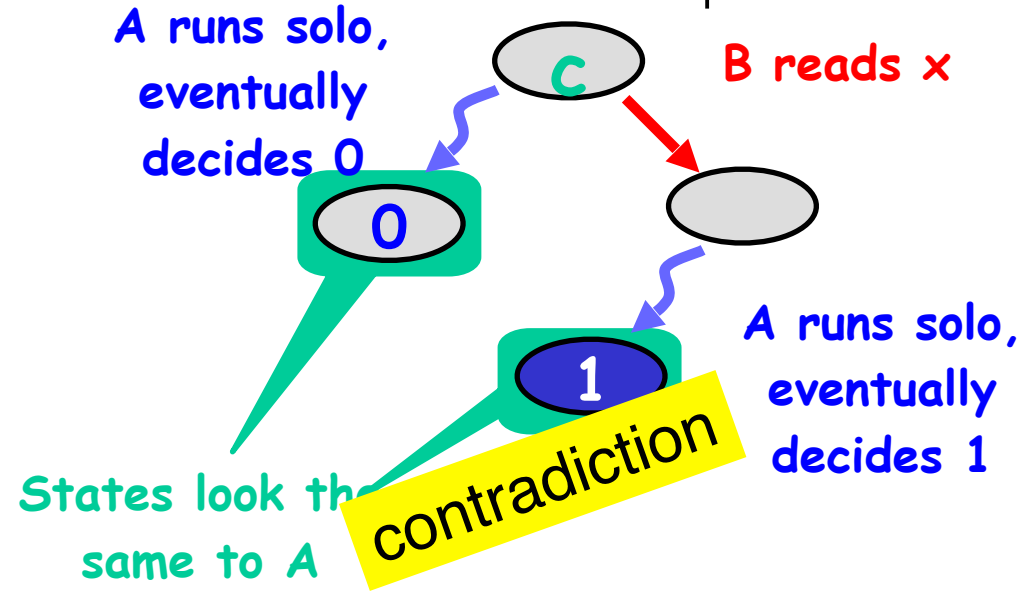


- Starting from a bivalent initial state
- The protocol can reach a critical state
  - Otherwise we could stay bivalent forever
  - And the protocol is not wait-free

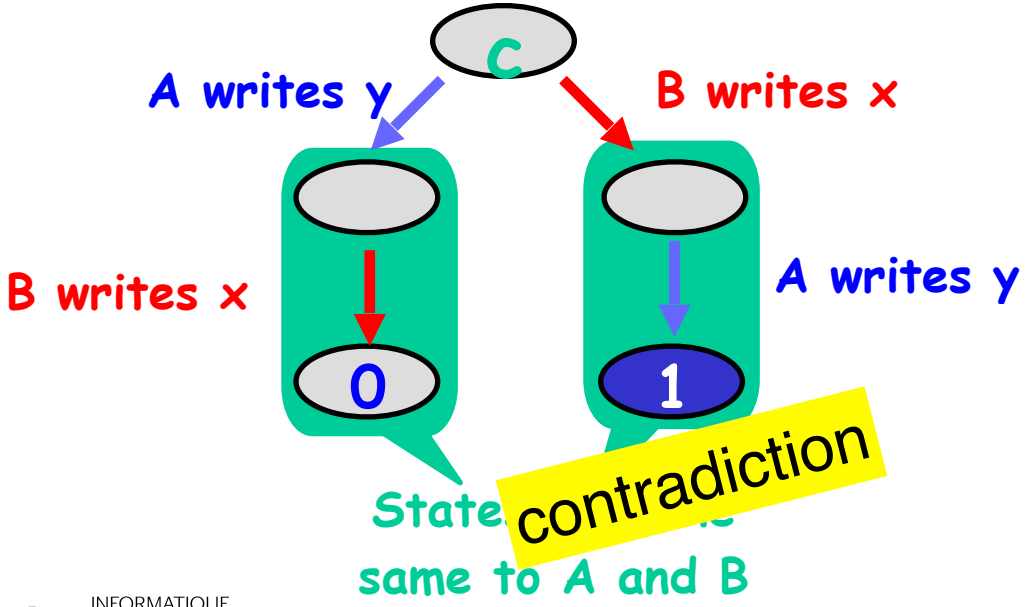
Then: for every consensus protocol there exists a critical step



Critical step

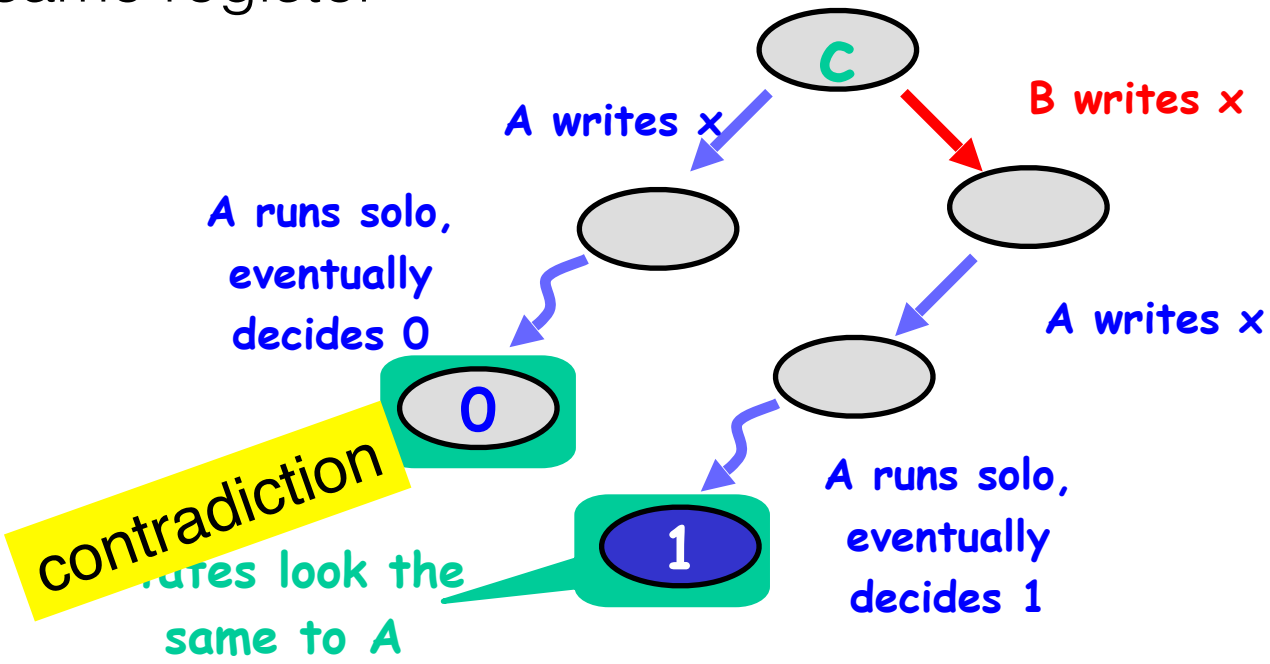


read



write on different registers

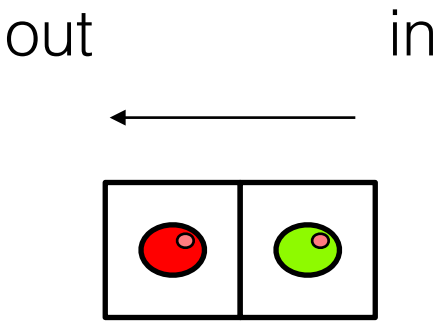
Write on the same register



Theorem: h(Register)=1

no way to get consensus with register!

Fifo -> 2 consensus



- Fifo: enq(v) and deq()
- Algo for 2 processes:

Queue q: initialized to Red,Green

process i :
writes its value v_i in proposed[i]

V:= deq() if V=Red then decide v_i

else decide proposed[1-i]

if you like Java…

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

abstract class ConsensusProtocol<T>
   implements Consensus {
 protected T[] proposed = new T[N]
 protected void propose(T value) {
  proposed[ThreadID.get()] = value;
 }
 abstract public T decide(T value);
}
public class QueueConsensus
  extends ConsensusProtocol {
 private Queue queue;
 public QueueConsensus() {
  queue = new Queue();
  queue.enq(Ball.RED);
  queue.enq(Ball.GREEN);
 }
public decide(object value) {
  propose(value);
  Ball ball = this.queue.deq();
  if (ball == Ball.RED)
   return proposed[i];
  else
   return proposed[1-i];
 }
}
```

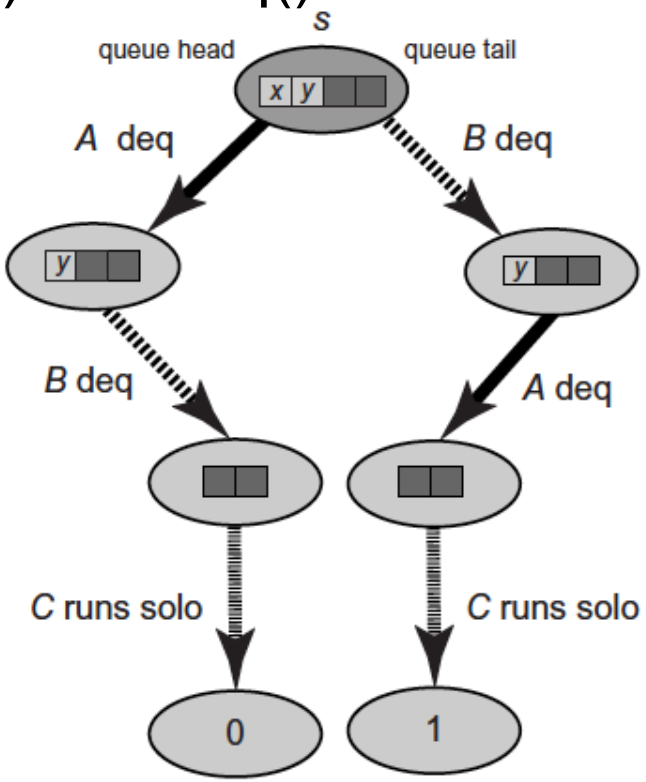
- only one process gets RED it is the winner
- the winner decides its value
- the loser can find the winner's value in the array

theorem: h(Queue)≥2

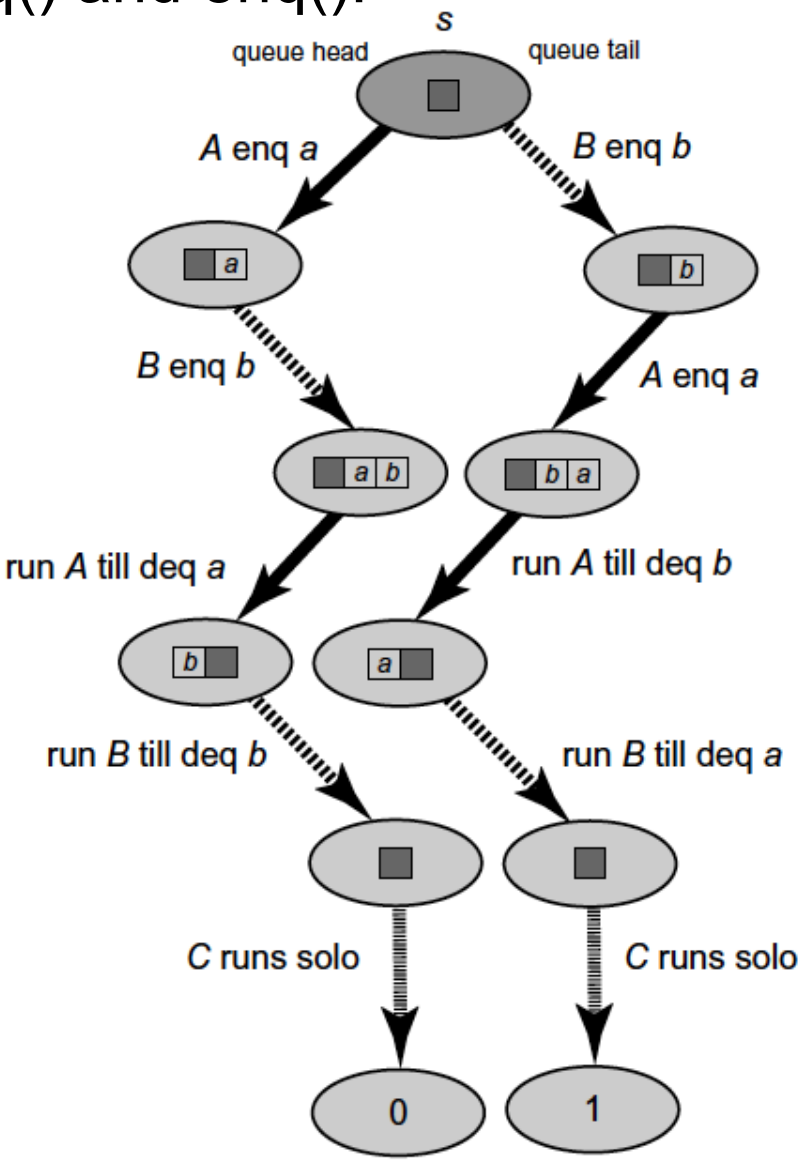
there is no wait-free implementation of queue with registers

enq(a) and deq() ?

deq() and deq():



enq() and enq():



theorem: h(queue)=2

Read-Modify-Write RMW

F a set of functions of integer to integer

method for a RMW object: method(v)

- returns the previous value x
- replace x with $f_v(x)$ for some f_v in F

RMW

Examples:

- getAndSet(v) (or swap) ($f_v(x)=v$)
- getAndIncrement() ($f(x)=x+1$)
- getAndAdd(k) ($f(x)=x+k$)
- compareAndSet(e,u) if the value of the register is equal to e then set the value to u and returns true else doesn't change and returns false
- get() identity

RMW is *non trivial* if the set of functions includes at least one function different from identity.

Theorem

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

(Hence No wait-free implementation of RMW registers from atomic registers)

Proof:

there is a value v and a function f in F such that f(v) ≠ v

```
public class RMWConsensus
                extends ConsensusProtocol {
        private RMWRegister r = v;
        public Object decide(object value) {
                proposed[i]=value;
                if (r.f() == v)
                 return proposed[i];
                else
                 return proposed[j];
}}
```

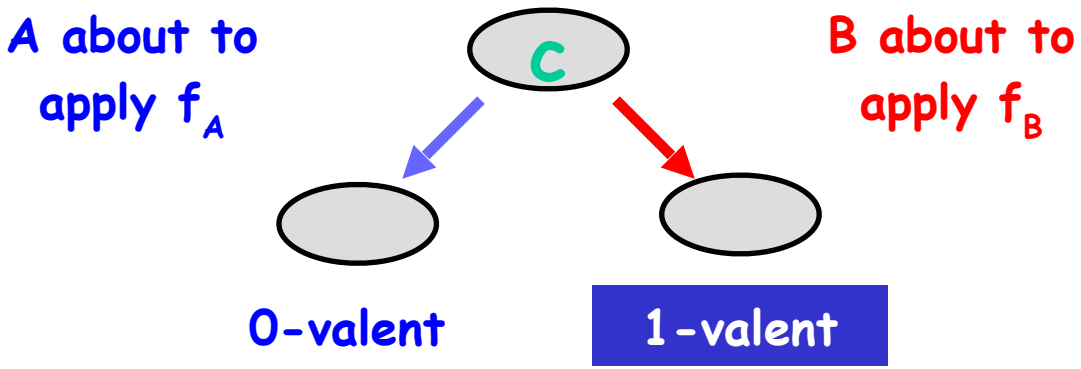
RMW

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

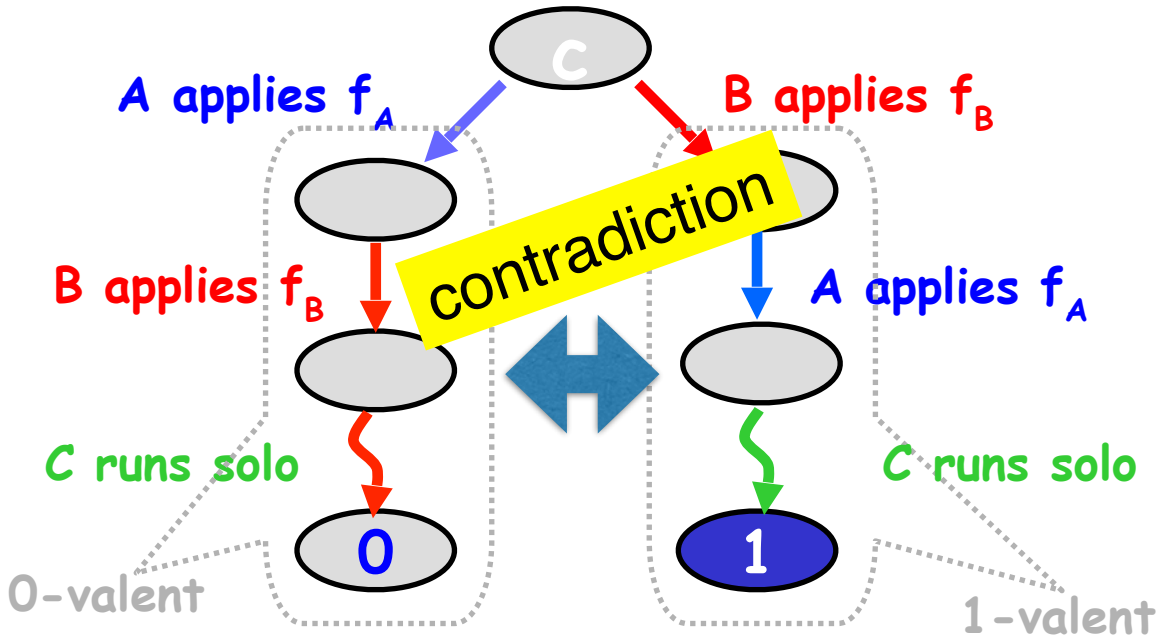
- Theorem : Any set of RMW objects that commutes or overwrites has consensus number exactly 2

RMW

Critical state

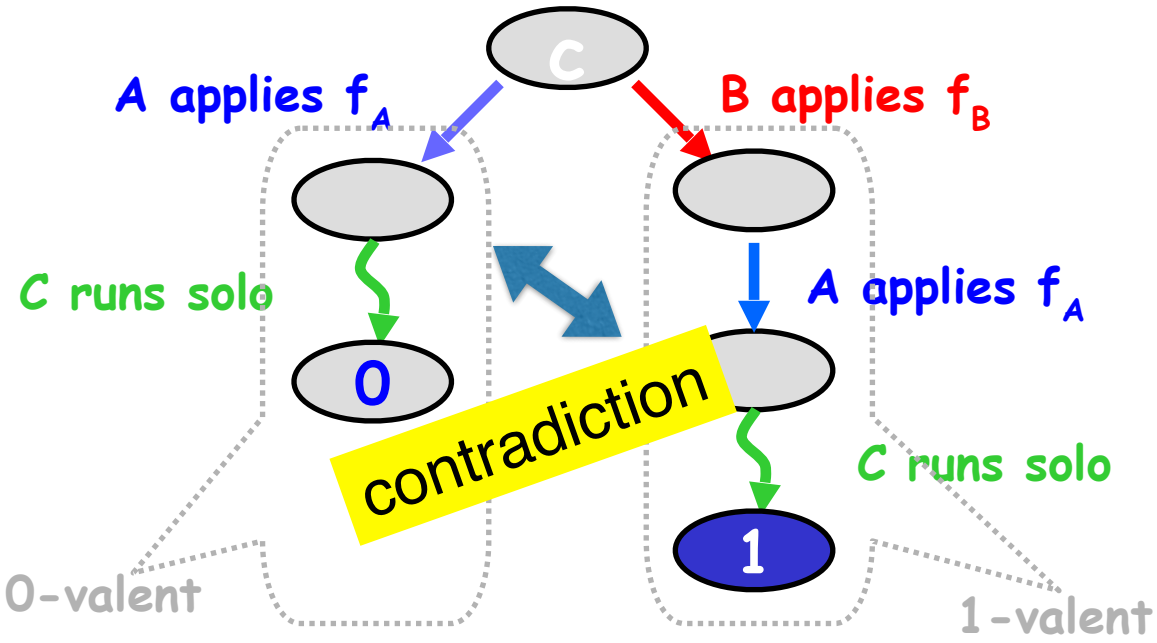


Commute



indistinguishable for C

Overwrite


C
A applies f_A
B applies f_B
C runs solo
A applies f_A
0
contradiction
C runs solo
1
0-valent
1-valent

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;
  }
… }
```

h(CompareAndSet) = ∞

```
public class RMWConsensus  extend ConsensusProtocol {
        private AtomicInteger r = new AtomicInteger(-1);
        public Object decide(object value) {
                proposed[i]=value;
                r.compareAndSet(-1,i);
         return proposed[r.get()];
        }
}
```

1 Read/Write Registers, Snapshots…
2 getAndSet, getAndIncrement, …
. . .
∞ compareAndSet,…

Hierarchy…

There exists (non-deterministic) objects a and b such that h(a)=h(b)=1 and h({a,b})=2

Wai-Kau Lo, Vassos Hadzilacos: All of Us Are Smarter than Any of Us: Nondeterministic Wait-Free Hierarchies Are Not Robust. SIAM J. Comput. 30(3): 689-728 (2000)

(Robustness: given a, b then h({a,b})= max {h(a), b(b)})

Exercices

h(Stack)? (very similar to Queue)

h(test&set) ? (test&set is a compareAndSet but only with boolean values)

From the previous…

we get a hierarchy:

Question

1 Read/Write Registers, Snapshots…
yes?
2 getAndSet, getAndIncrement, …
yes?
.
.
.
yes?
∞ compareAndSet,…

Universality

A class C of objects is *universal* for some set E of classes if and only if any object in E can be implemented with objects of C (and registers)

n-Consensus is universal for n-objects (objects shared by n processes)

- for any sequential object there is a wait-free (linearizable) implementation of this object with atomic registers and n-consensus objects.

Universal Construction

Theorem [Herlihy, 1991] If *n* processes can solve consensus, then *n* processes can (wait-free) implement every object $O=(Q,O,R,\sigma)$ shared by *n* processes

Remarks

n-object:

- O n-object s.t. h(0)=n enables to implement all n-object (all n-objects such that h(0)=n are in that sense « equivalent »)
- But it concerns n-objects:
  - A file (stack) enables to solve 2-consensus
  - 2-consensus enables to implement a 2-file (or a 2-stack) -universality result

  - But that does not prove that with 2-consensus it is possible to implement a (standard) file (or stack!)

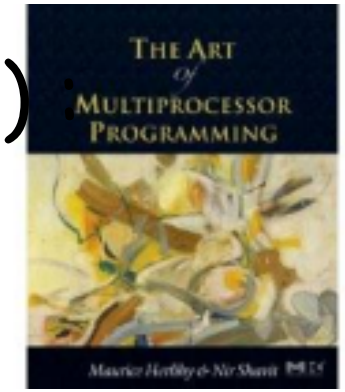
Universality

Principles:

- build a shared list corresponding to the history of the object (each cell contain the operation and a link to the next cell)
- link new cells by consensus (hence there is one list!):
  - if a process wants to add a new operation, it goes to the head of the list and proposes the operation to the consensus
- to ensure wait-freedom a « helping mechanism » (processes propose operations of other processes) (recall that with a consensus object some processes may loose forever)

Universality

an implementation (Herlihy-Shavit)



en java

```
public interface SeqObject {
  public abstract
    Response apply(Invoc invoc);
}
```

```
public class Invoc {
 public String method;
 public Object[] args;
}
```

```
public class Response {
  public Object value;
}
```

Implementation

First: lock-free implementation (infinitely often some method call finishes)

Then wait-free: each method call takes a finite number of steps to finish

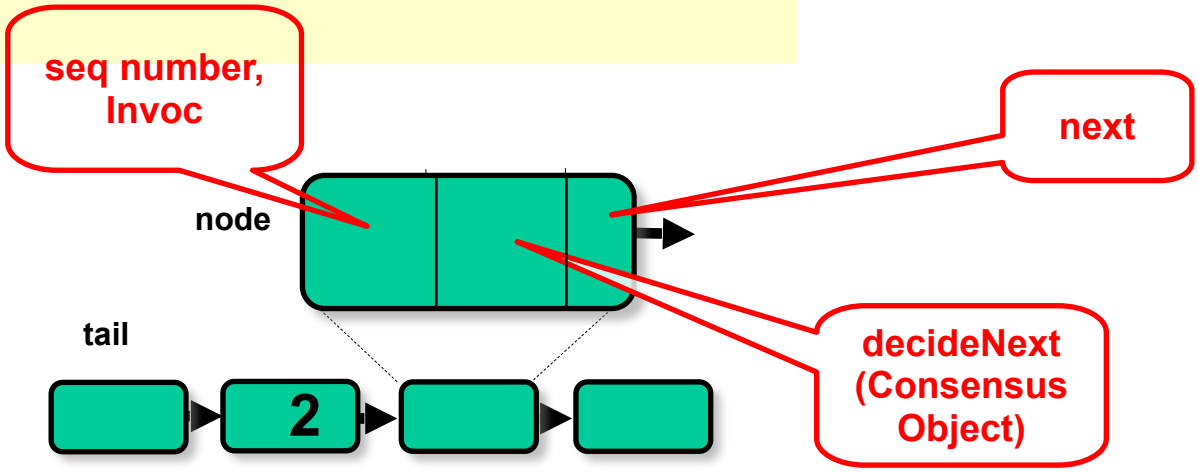
- Object represented as
  - Initial Object State
  - A Log: a linked list of the method calls
- New method call
  - Find end of list
  - Atomically append call (using consensus)
  - Compute response by replaying log

- Threads use *one-time* consensus objects to decide which node goes next
- Threads update actual next field to reflect consensus outcome
  - OK because they all write the same value

Data Structure

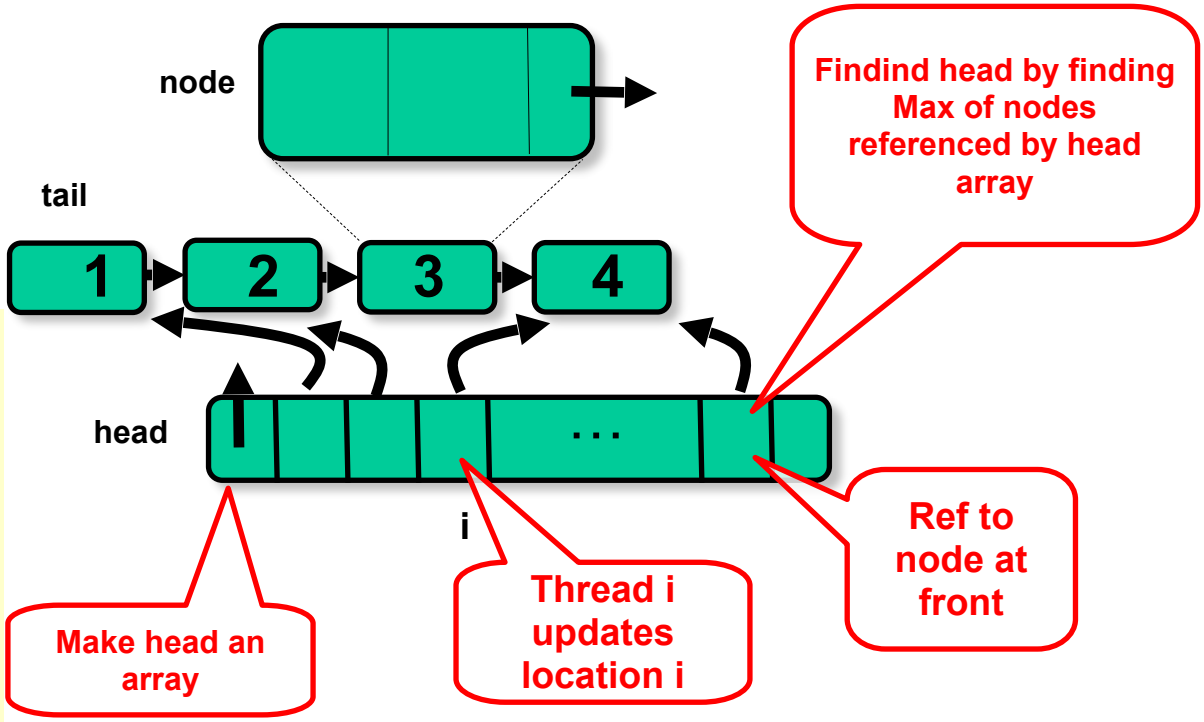
```
public class Node implements
java.lang.Comparable {
 public Invoc invoc;
 public Consensus<Node> decideNext;
 public Node next;
 public int seq;
 public Node(Invoc invoc) {
    invoc = invoc;
    decideNext = new Consensus<Node>()
    seq = 0;
  }
```

warning: in a one-shot consensus a process can do at most one consensus per Node



add the node to the head:
how to maintain the head array ?

head

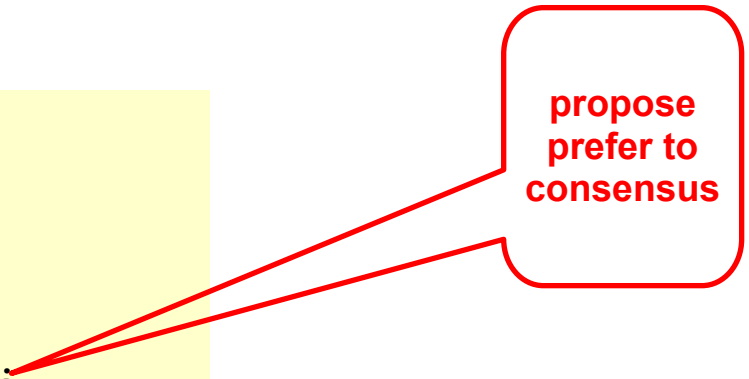



```
public class Universal{
  private Node[] head;
  public Universal(){
    private Node tail = new
                  Node();
   tail.seq = 1;
   for (int j=0; j < n; j++){
          head[j] = tail;
    }
    …
```

```
public static Node max(Node[] array) {
    Node max = array[0];
    for (int i = 1; i < array.length; i++)
      if (max.seq < array[i].seq)
        max = array[i];
    return max;
```

Link the invoc

```
public Response apply(Invoc invoc) {
  int i = myID();
  Node prefer = new node(invoc);
  while (prefer.seq == 0) {
    Node before = Node.max(head);
    Node after =
     before.decideNext.decide(prefer);
    before.next = after;
    after.seq = before.seq + 1;
    head[i] = after;
    }
  …
```



Compute response

```
...
//compute my response
SeqObject MyObject = new SeqObject();
current = tail.next;
while (current != prefer){
    MyObject.apply(current.invoc);
    current = current.next;
  }
return MyObject.apply(current.invoc);
}
```

Correctness

- List defines linearized sequential history
- Thread returns its response based on list order

- Lock-free because
  - A thread moves forward in list
  - Can repeatedly fail to win consensus on “real” head only if another succeeds
  - Consensus winner adds node and completes within a finite number of steps

wait-free

- This implementation is not wait-free: if an invocation is concurrent infinitely often with other invocations this invocation may be forever not chosen in consensus (consensus can chose any proposed value)

- solution: helping
  - Lock-free construction + **`announce`** array
  - Stores (pointer to) node in **`announce`**
    - If a thread doesn't append its node
    - Another thread will see it in array and ***help*** append it

```
public class Universal {
  private Node[] announce;
  private Node[] head;
  public Universal(){
     private Node tail = new node();
     tail.seq = 1;
     for (int j=0; j < n; j++){
       head[j] = tail; announce[j] = tail
     };
```

announce array

```
public Response apply(Invoc invoc) {
  int i = ThreadID.get();
  announce[i] = new Node(invoc);
  head[i] = Node.max(head);
  while (announce[i].seq == 0) {
  …
  // while node not appended to list
  …
  }
```

help!

- Non-zero sequence # means success
- Thread keeps helping append nodes
- Until its own node is appended

```
while (announce[i].seq == 0) {
 Node before = head[i];
 Node help = announce[(before.seq + 1) % n];
 if (help.seq == 0)
     prefer = help;
    else
     prefer = announce[i];
…
```

- Choose a thread to “help”
- If that thread needs help
  - Try to append its node
  - Otherwise append your own
- Worst case
  - Everyone tries to help same pitiful loser
  - Someone succeeds

- When last node in list has sequence number *k*
- All threads check …
  - Whether thread *k*+1 mod *n* wants help
  - If so, try to append her node first

- First time after thread *k*+1 announces
  - No guarantees
- After *n* more nodes appended
  - Everyone sees that thread *k*+1 wants help
  - Everyone tries to append that node
  - Someone succeeds

Then

- After thread A announces its node
- No more than *n* other calls
  - Can start and finish
  - Without appending A's node

wait-free

```
while (announce[i].seq == 0) {
  Node before = head[i];
    Node help = announce[(before.seq + 1) % n];
    if (help.seq == 0)
        prefer = help;
   else
        prefer = announce[i];
    Node after = before.decideNext.decide(prefer);
  before.next = after;
  after.seq = before.seq + 1;
  head[i] = after;
 }
```

same as before

Compute my response

```
...
//compute my response
SeqObject MyObject = new SeqObject();
current = tail.next;
while (current != announce[i]){
     MyObject.apply(current.invoc);
     current = current.next;
  }
return MyObject.apply(current.invoc);
}
```

- Compute result by
  - sequentially applying list's method calls
  - to a private copy of the object
  - starting from the initial state