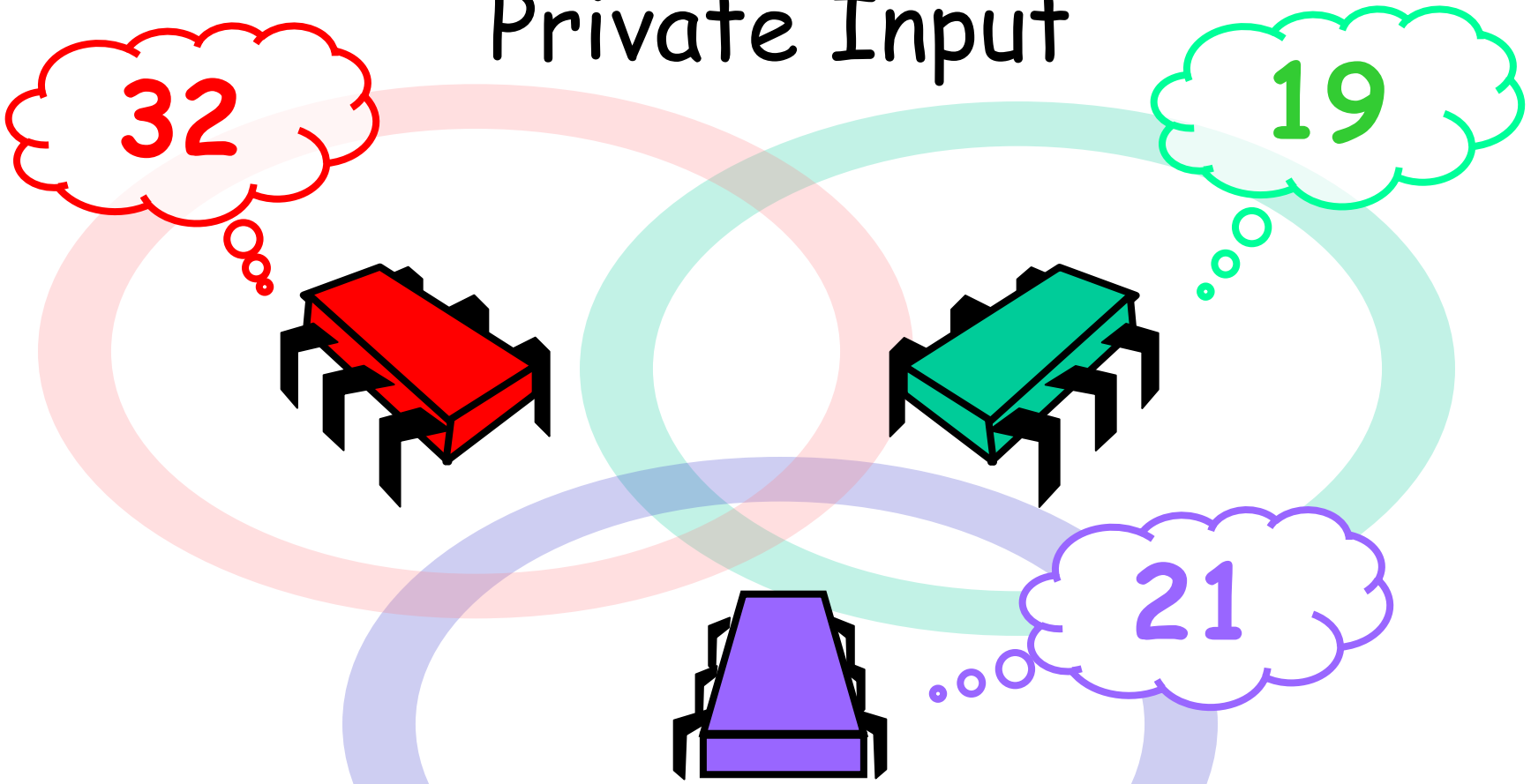
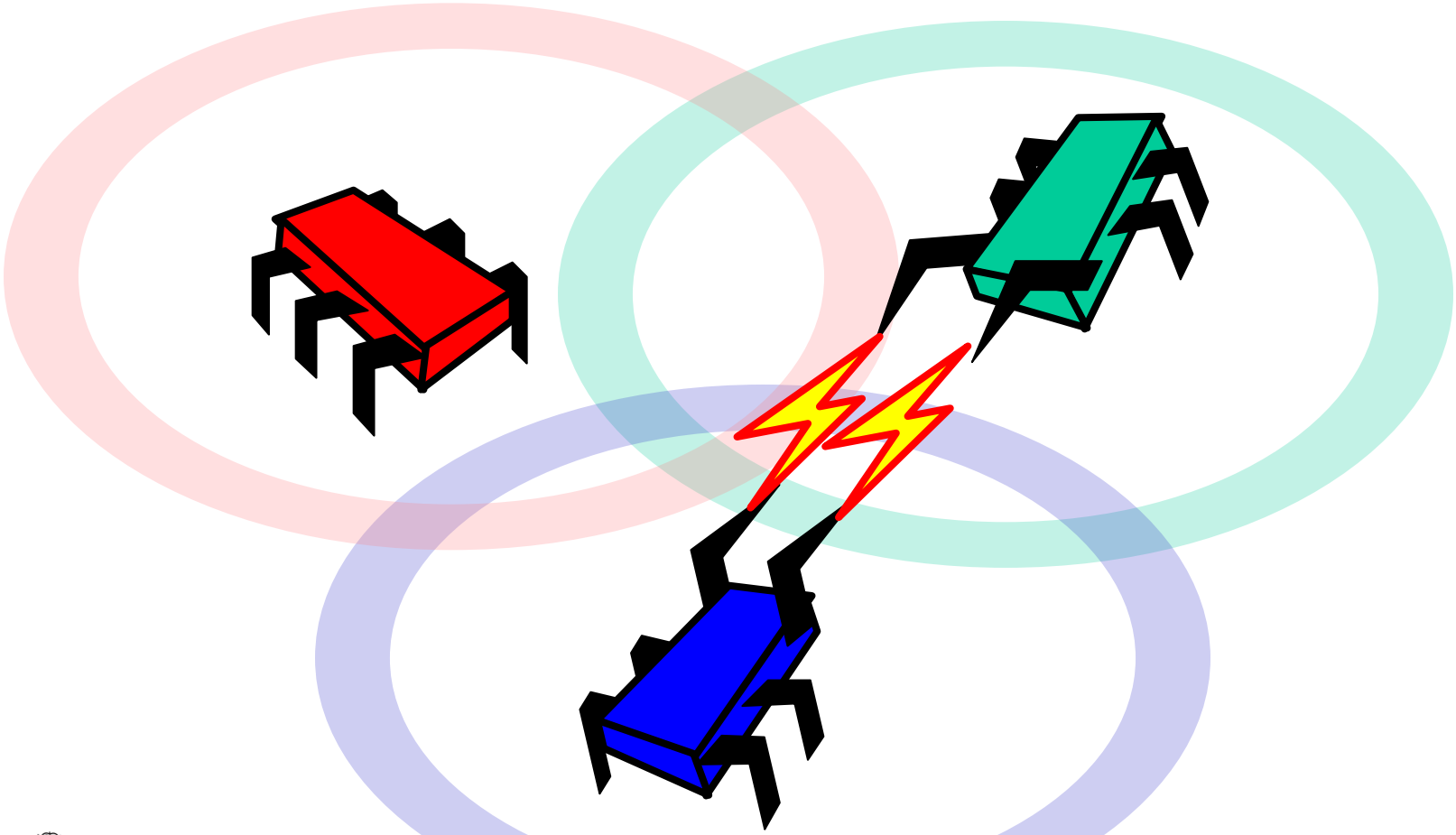
Consensus revisited

from A Comlexity –based Hierarchy
for MP Synchronization
by Faith Ellen, PODC 2016

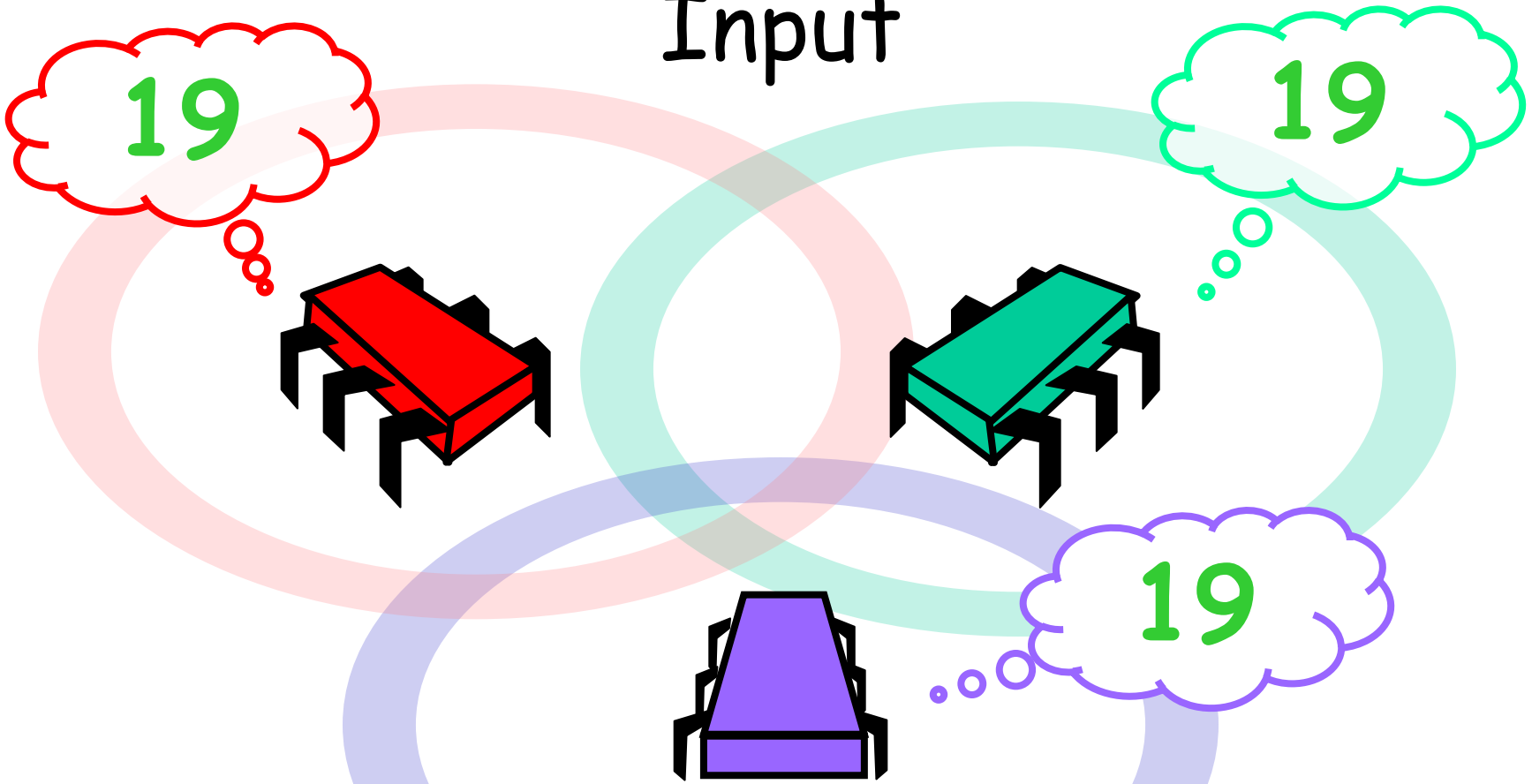
Consensus: Each Thread has a Private Input



They Communicate



They Agree on One Thread's Input

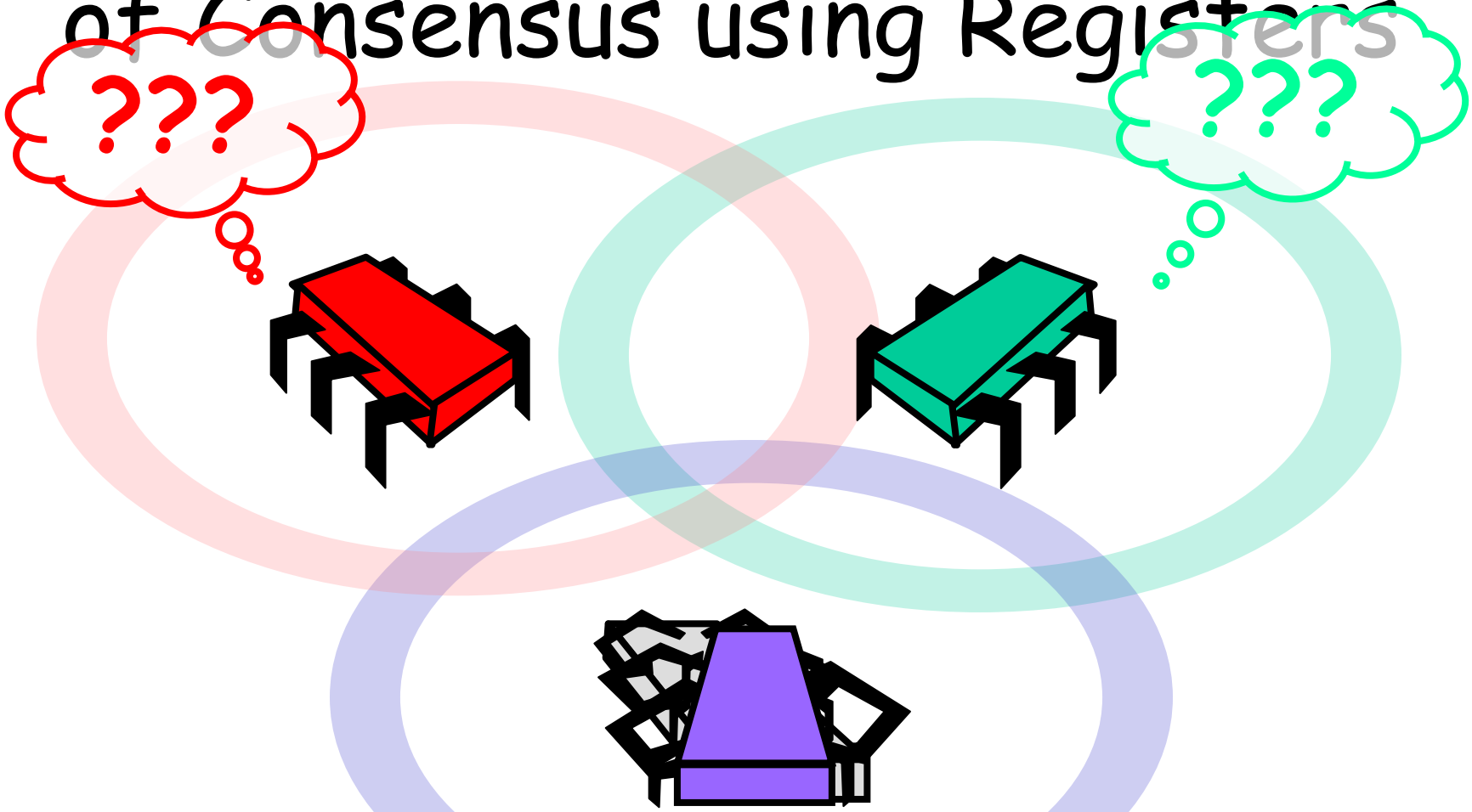


Formally: Consensus

Consistent: all threads decide the same value

Valid: the common decision value is some thread's input

No Wait-Free Implementation of Consensus using Registers



Consensus Numbers

- 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

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 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!

Theoretical Caveat: Certain weird exceptions exist

Examples

- "test-and-set" getAndSet(1) f(v)=1

  Overwrite $f_i(f_j(v))=f_i(v)$
- "swap" getAndSet(x) f(v,x)=x

  Overwrite $f_i(f_j(v))=f_i(v)$
- "fetch-and-inc" getAndIncrement() f(v)=v+1

  Commute $f_i(f_j(v))= f_j(f_i(v))$

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;
  } … }
```

The Consensus Hierarchy

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

Uninterruptible Instructions to Fetch and Update Memory

- Atomic exchange: interchange value in register with one in memory
  - 0 ⇒ Synchronization variable is free
  - 1 ⇒ Synchronization variable is locked and unavailable
  - Set register to 1 & swap
  - New value in register determines success in getting lock
    - 0 if you succeeded in setting lock (you were first)
    - 1 if another processor claimed access first
  - Key: exchange operation is indivisible

Uninterruptible Instruction to Fetch and Update Memory

- Hard to read & write in 1 instruction, so use 2
- Load linked (or load locked) + store conditional
  - Load linked returns initial value
  - Store conditional returns 1 if succeeds (no other store to same memory location since preceding load) and 0 otherwise

Example of atomic swap with LL & SC

```
try:  mov   R3,R4      ; mov exchange value->R3
      ll    R2,0(R1)   ; get old value
      sc    R3,0(R1)   ; store new value
      beqz  R3,try     ; loop if store fails
      mov   R4,R2      ; put old value in R4
```

Example of fetch & inc with LL & SC

```
try:  ll    R2,0(R1)   ; get old value
      addi  R2,R2,#1   ; increment it
      sc    R2,0(R1)   ; store new value
      beqz  R2,try     ; loop if store fails
```

User-Level Synchronization Using LL/SC

- Spin locks: processor continuously tries to acquire lock, spinning around loop trying to get it

```
            li      R2,#1
lockit:     exch    R2,0(R1)        ; atomic exchange
            bnez    R2,lockit       ; loop while locked
```

User-Level Synchronization Using LL/SC

- What about MP with cache coherency?
  - Want to spin on cached copy to avoid full memory latency
  - Likely to get cache hits for such variables
- Problem: exchange includes write
  - Invalidates all other copies
  - Generates considerable bus traffic

User-Level Synchronization Using LL/SC (cont'd)

- Solution to bus traffic: don't try exchange when you know it will fail
  - Keep reading cached copy
  - Lock release will invalidate

```
try:    li    R2,#1
lockit: lw    R3,0(R1)     ;load old
        bnez  R3,lockit    ;≠ 0 ⇒ spin
        exch  R2,0(R1)     ;atomic exchange
        bnez  R2,try       ;spin on failure
```

Strange example

- fetch-and-add(2),
  - returns the number stored in a memory location and increases its value by 2,
- test-and-set(),
  - returns the number stored in a memory location and sets it to 1 if it contained 0

- Objects supporting these instructions have consensus number 2

Wait-free binary consensus for 3 or more processes

- Use a single memory location initialized with 0
- Processes with input 0 perform fetch-and-add(2)
- Processes with input 1 perform test-and-set()

Wait-free binary consensus for 3 or more processes

- If the value returned is odd, decide 1
- If the value 0 is returned from test-and-set(), decide 1
- Otherwise, decide 0

Another example

- read()
  - returns the number stored in a memory location
- decrement()
  - decrements the number stored in a memory location and returns nothing
- multiply(x)
  - multiplies the number stored in a memory location by x and returns nothing.

Read(), decrement(), multiply(x)

- All these have consensus number 1
- It is possible to use these instructions to achieve wait-free binary consensus for any number of processes

Read(), decrement(), multiply(x)

- All these have consensus number 1
- It is possible to use these instructions to achieve wait-free binary consensus for any number of processes

Consensus protocol

- Use a single memory location initialized with 1
- Processes with input 0 perform decrement() and read()
- Processes with input 1 perform multiply(n) and read()
- If the value returned is positive, decide 1
- If the value returned is negative, decide 0

Theorem

- We can solve n-thread consensus using only
  - A single memory location
  - read() and
  - either add(x), multiply(x) or set-bit(x)
  -

Theorem

- It is possible to solve obstruction-free *m*-valued consensus among *n* processes using an *m*-component unbounded counter

- *m*-component unbounded counter has m components, each with an integer value supporting increment() and scan()

Proof (1/4)

- For each possible input value v, there's a separate component c_v initialized to be 0.
- Each process alternates between promoting(c_v++) a value and performing a scan of all m components.
- A process first promotes its input value.

Proof (2/4)

- After performing a scan, if it observes that c_v is at least n larger than other counters, it returns v.
- Otherwise, it promotes the value with largest count(breaking ties arbitrarily).

Proof (3/4)

- If some process returns the value v, then each other process will increment some component at most once before next performing a scan.
- In each of those scans, the count stored in cv will still be larger than the counts stored in all other components.

Proof (4/4)

- From then on, these processes will promote value v and keep incrementing c_v.
- Eventually, the count in component c_v will be at least n larger than the counts in all other components, and these processes will return v, ensuring agreement.

Lemma (bounded counter)

- If each component also supports a decrement(), we can bound the counter by 3n.
- When promoting u, Among the other components (i.e. excluding c_u), let c_v be one that stores the largest count. If $c_v < n$, it increments c_u, as before. If $c_v \geq n$, then, instead of incrementing c_u, it decrements c_v.

Lemma

- A component with value 0 is never decremented. This is because, after the last time some process observed that it stored a count greater than or equal to n, each process will decrement the component at most once before performing a scan().

Lemma

- Similarly, a component c_v never becomes larger than 3n – 1:
- After the last time some process observed it to have count less than 2n, each process can increment c_v at most once before performing a scan().

Lemma

- If $c_v \geq 2n$, then either the other components are less than n, in which case the process returns without incrementing c_v, or the process decrements some other component, instead of incrementing c_v

n-consensus with 2 max-registers

- Initially, both m1 and m2 have value (0, 0).
- Each process alternately performs write-max on one component and takes a scan of both components.
- It begins by performing write-max (0, x′) to m1, where $x' \in \{0, \ldots, n-1\}$ is its input value.

n-consensus with 2 max-registers

- If m1 has value (r+1, x) and m2 has value (r, x) in the scan, then it decides x and terminates.
- If both m1 and m2 have value (r, x) in the scan, then it performs write-max (r+1, x) to m1.
- Otherwise, it performs write-max to m2 with the value of m1 in the scan.