Can we do better?

A little more formally

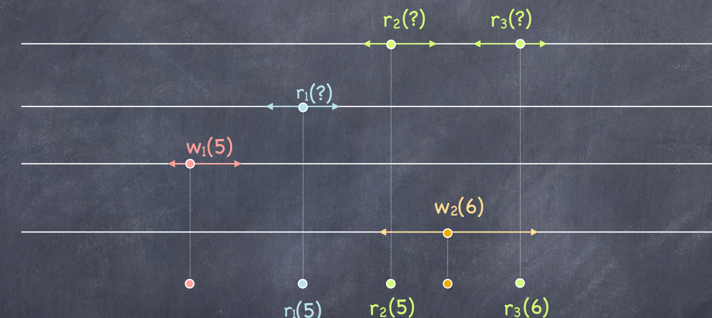
- Execution of a concurrent object modeled by a history
  - a finite sequence of operation invocation and responses
- A history H is sequential if:
  - the first event of H is an invocation
  - each invocation (but possibly the last) is immediately followed by a matching response, followed by a matching invocation
- A history that is not sequential is concurrent

Linearizability

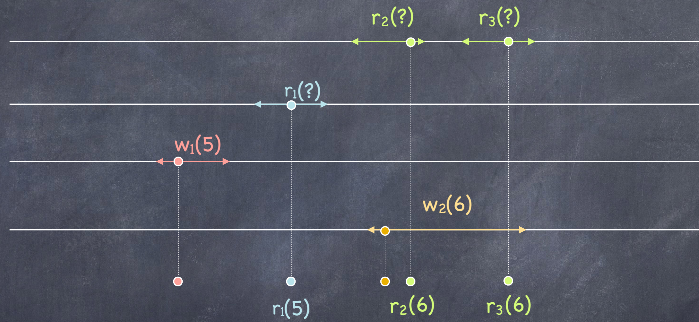
(Herlihy and Wing 1990)

- A history H is linearizable if there exists a permutation π of the operations in H such that
  - For each object O, the subhistory $\pi|O$ respects the sequential specification of O
  - If the response of operation o_1 occurs in H before the invocation of operation o_2, then o_1 appears before o_2 in π
    - in other words, π respects the real-time ordering of non-overlapping operations

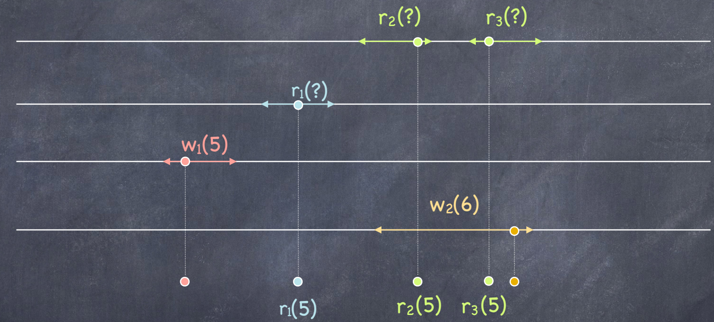
Example: Registers



Example: Registers

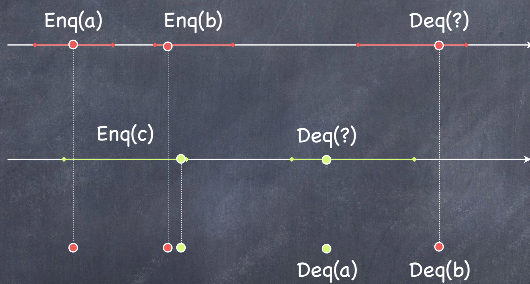


Example: Registers

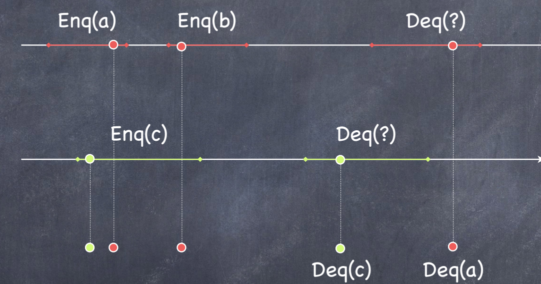


Linearizable registers are called atomic

Example: a queue



Example: a queue



Think global, act local

- A property P of a concurrent system is local if the system satisfies P whenever each individual object satisfies P
  - Given two objects O_1 and O_2 each satisfying P, the composite object $[O_1, O_2]$ satisfies P

Linearizability is a local property

- Theorem

  A history H is linearizable iff, for each object O, H restricted to the operations in O is linearizable

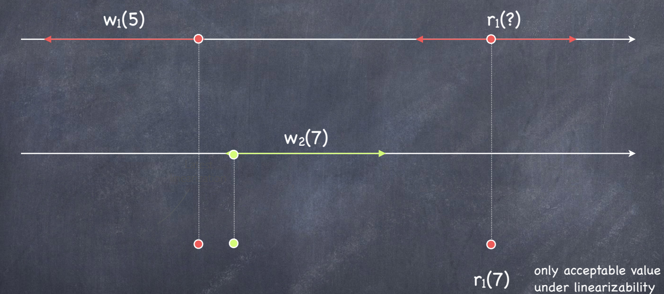
  - Because linearizability is a local property, objects can be implemented independently from each other

Sequential consistency

(Lamport "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs", IEEE TOC C-28,9 (Sept. 1979), 690-691)

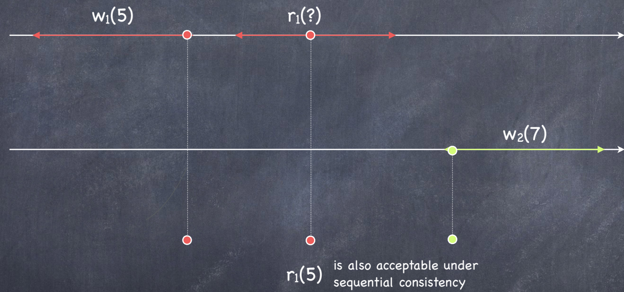
- A history H is sequentially consistent if there exists a permutation π of the operations in H such that
  - For each object O, $\pi|O$ respects the sequential specification of O
  - If the response for operation o_1 at p_i occurs in H before the invocation for operation o_2 at p_i then o_1 appears before o_2 in π
    - order of non-overlapping operations is preserved only for operations by the same thread

Example



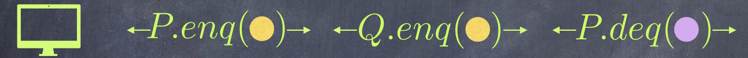
but under sequential consistency...

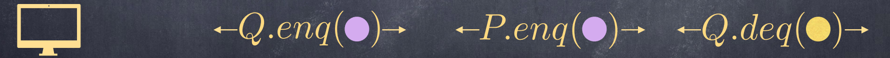
Example



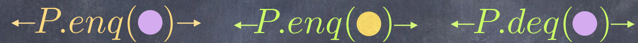
...this execution is indistinguishable from the previous one

The Case of the FIFO queue

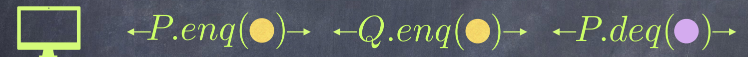


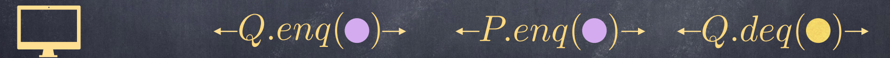


The Case of the FIFO queue



The Case of the FIFO queue





The Case of the FIFO queue

$\leftarrow Q.enq(\bullet) \rightarrow$ $\leftarrow Q.enq(\bullet) \rightarrow$ $\leftarrow Q.deq(\bullet) \rightarrow$

The Case of the FIFO queue

$\leftarrow P.enq(\bullet) \rightarrow$ $\leftarrow Q.enq(\bullet) \rightarrow$ $\leftarrow P.deq(\bullet) \rightarrow$

Not Sequentially Consistent!

$\leftarrow Q.enq(\bullet) \rightarrow$ $\leftarrow P.enq(\bullet) \rightarrow$ $\leftarrow Q.deq(\bullet) \rightarrow$

Linearizability is non blocking

- Definition

  A method is total if it is defined for every object value (e.g. Enq). Otherwise, it is partial (e.g. Deq).

- Theorem

  Linerizability is nonblocking: a pending invocation of a total method is never required to wait for another pending invocation to complete

- Note
  - Blocking (or deadlock) may occur in particular implementations of linearizability
  - Sequential consistency is <u>not</u> a nonblocking property

Linearizability is non blocking

- Definition

  A method is total if it is defined for every object value (e.g. Enq). Otherwise, it is partial (e.g. Deq).

- Theorem

  Linerizability is nonblocking: a pending invocation of a total method is never required to wait for another pending invocation to complete

- Note
  - Blocking (or deadlock) may occur in particular implementations of linearizability
  - Sequential consistency is <u>not</u> a nonblocking property

Taking a step back

$\leftarrow x.write(1) \rightarrow$ $\leftarrow y.read(0) \rightarrow$

$\leftarrow y.write(1) \rightarrow$ $\leftarrow x.read(0) \rightarrow$

- Linearizable?
- Sequentially consistent?
- Happy?

NO!

- This "flag" pattern is at the core of our mutual exclusion protocols
  - Peterson
  - Bakery
- No discussion!

Yeees?

- Sequential consistency is too expensive as default in modern hardware
  - we write to caches anyway!
- OK if flag principle
  - violated by default
  - honored by explicit request
    - e.g., if you want to synchronize

Liveness: dependent progress (blocking)

- Though linearizability is non-blocking, its implementation could delay threads
  - e.g., a thread slows down after acquiring a lock
- Implementations using locks can only guarantee dependent progress
  - as long as locks released in a timely manner
- Dependent progress conditions for blocking implementations:
  - deadlock freedom: some method call completes
  - starvation freedom: every method call completes

Liveness: dependent progress (non-blocking)

Obstruction freedom

- An obstruction-free implementation of a concurrent object guarantees that a single process executing in isolation will complete any operation in a bounded number of steps

Liveness: independent progress (non-blocking)

Lock freedom

- A lock-free implementation of a concurrent object is obstruction free and, in the presence of concurrency, it guarantees that some correct process will complete any operation in a bounded number of steps
  - A lock-free implementation is deadlock free (despite asynchrony and process crashes), but can suffer from starvation

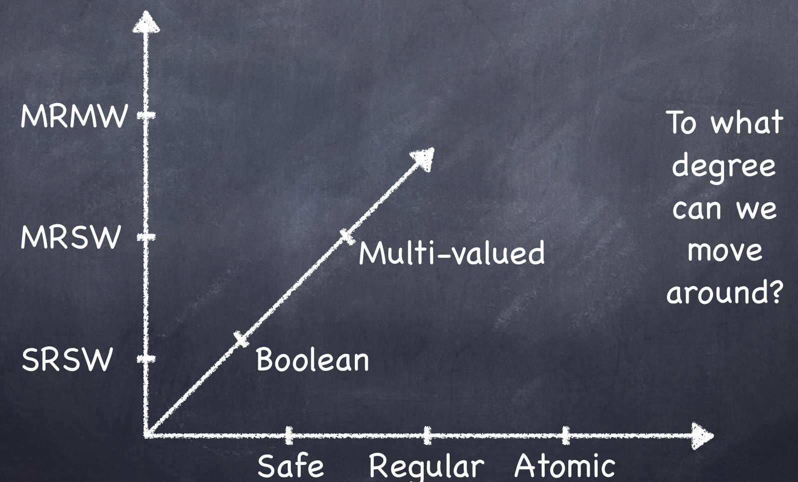
Liveness: independent progress (non-blocking)

Wait freedom

- A wait-free implementation of a concurrent object guarantees that any process can complete any operation in a finite number of steps, regardless of the execution speed of other processes
  - A wait free implementation is both deadlock and starvation free and tolerates up to $n-1$ process crashes

Register: we hardly knew ye

An excursus



Oh, the places you'll go...

Base class	Implemented class
SRSW safe	MRSW safe
MRSW Boolean safe	MRSW Boolean regular
MRSW Boolean regular	MRMW regular
MRSW regular	SRSW atomic
SRSW atomic	MRSW atomic
MRSW atomic	MRMW atomic
MRSW atomic	atomic snapshot

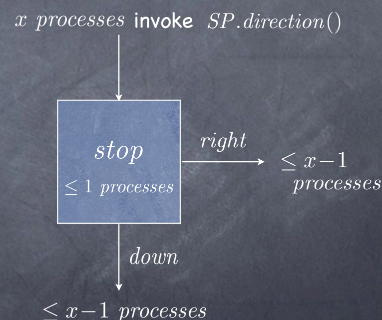
What's so cool about wait-freedom?

- We used atomic registers to implement mutual exclusion
  - we can't use mutual exclusion to implement atomic registers!
  - but there is more not to like about mutual exclusion
    - asynchronous interrupts
    - heterogeneous processors
    - fault tolerance

The Splitter

(Lamport, "A fast mutual exclusion algorithm", ACM TOCS 5(1):1-11, 1987)

- At most one process exists with *stop*
- At most $x-1$ processes exit with *right*
- At most $x-1$ processes exit with *down*
- If a single correct process invokes the splitter, it exits with *stop*



A wait-free implementation

- Two shared registers: LAST (init $\perp$) and DOOR (init $open$) (atomic MWMR)

```
operation SP.direction(i)
  LAST := id_i
  if (DOOR = closed)
    then move_i := right
    else DOOR := closed
         if (LAST = id_i) then move_i := stop
                          else move_i := down
         end if
  end if
return (move_i)
```

A wait-free implementation

- Two shared registers: LAST (init $\perp$) and DOOR (init *open*) (atomic MWMR)

```
operation SP.direction(i)
  LAST := id_i
  if (DOOR = closed)
     then move_i := right
     else DOOR := closed
          if (LAST = id_i) then move_i := stop
                           else move_i := down
          end if
  end if
return (move_i)
```

Proof:

- At least one not *right*
- At least one not *down*
- No two *stop*

A wait-free implementation

- Two shared registers: LAST (init $\perp$) and DOOR (init *open*) (atomic MWMR)

```
operation SP.direction(i)
  LAST := id_i
  if (DOOR = closed)
     then move_i := right
     else DOOR := closed
          if (LAST = id_i) then move_i := stop
                           else move_i := down
          end if
  end if
return (move_i)
```

Proof:

- At least one not *right*
  - door must be closed
  - whoever closed it, is *stop* or *down*
- At least one not *down*
- No two *stop*

A wait-free implementation

- Two shared registers: LAST (init $\perp$) and DOOR (init *open*) (atomic MWMR)

```
operation SP.direction(i)
  LAST := id_i
  if (DOOR = closed)
     then move_i := right
     else DOOR := closed
          if (LAST = id_i) then move_i := stop
                           else move_i := down
          end if
  end if
return (move_i)
```

Proof:

- At least one not *right*
  - door must be closed
  - whoever closed it, is *stop* or *down*
- At least one not *down*
  - let p be last to write LAST
  - if DOOR closed, then *right*
  - if DOOR open, then *stop*
- No two *stop*

A wait-free implementation

- Two shared registers: LAST (init $\perp$) and DOOR (init *open*) (atomic MWMR)

```
operation SP.direction(i)
  LAST := id_i
  if (DOOR = closed)
     then move_i := right
     else DOOR := closed
          if (LAST = id_i) then move_i := stop
                           else move_i := down
          end if
  end if
return (move_i)
```

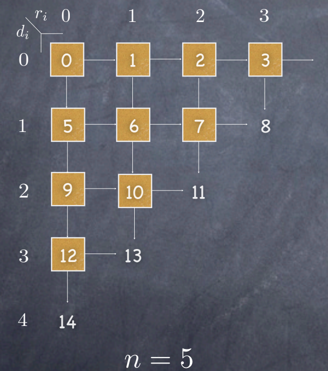
Proof:

- At least one not *right*
  - door must be closed
  - whoever closed it, is *stop* or *down*
- At least one not *down*
  - let p be last to write LAST
  - if DOOR closed, then *right*
  - if DOOR open, then *stop*
- No two *stop*
  - let p first to find LAST = p
  - no one has yet modified LAST
  - later q will find DOOR closed

The static Renaming problem

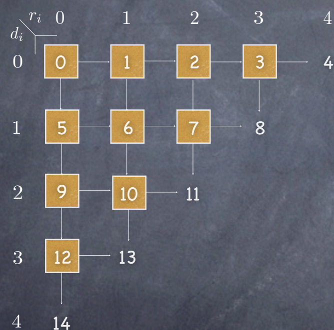
- $id_1, \ldots, id_n$: initial identities of the processes
- $id_1, \ldots, id_n \in [0, \ldots, N-1]$ where $N >>> n$
- Goal: acquire a new static unique name in the set $[0, \ldots, M-1]$, independent of current name
- M should be as small as possible
- Lower bound: $M \geq 2n-1$ (Herlihy-Shavit, JACM 1990)

The Moir-Anderson wait-free solution



$n = 5$

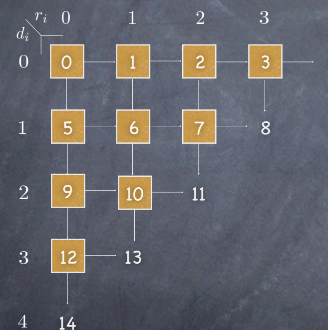
The Moir-Anderson wait-free solution



$n = 5$

```
operation get_name(id_i)
  d_i := 0; r_i := 0; move_i := down
  while (move_i ≠ stop) do
    move_i := SP.direction(id_i)
    case (move_i = right) then r_i := r_i+1
         (move_i = down) then d_i := d_i+1
         (move_i = stop) then exit_loop
    end case
  end while
  return (n×d_i + r_i − (d_i(d_i−1)/2))
```

The Moir-Anderson wait-free solution



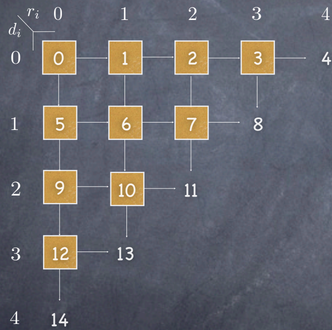
$n = 5$

```
operation get_name(id_i)
  d_i := 0; r_i := 0; move_i := down
  while (move_i ≠ stop) do
    move_i := SP.direction(id_i)
    case (move_i = right) then r_i := r_i+1
         (move_i = down) then d_i := d_i+1
         (move_i = stop) then exit_loop
    end case
  end while
  return (n×d_i + r_i − (d_i(d_i−1)/2))
```

- The new name is the position $[d_i, r_i]$

$M = n(n+1)/2$

The Moir-Anderson wait-free solution




```
operation get_name(id_i)
  d_i := 0; r_i := 0; move_i := down
  while (move_i ≠ stop) do
    move_i := SP.direction(id_i)
    case (move_i [illegible]) then r_i := r_i+1
         (move_i = down) then d_i := d_i+1
         (move_i = stop) then exit_loop
    end case
  end while
  return (n × d_i + r_i − (d_i(d_i−1)/2))
```

Proof:

- A process invokes $SP.direction()$ at most n times before $stop$
- At most one process stops at a splitter, and each splitter has a unique name
  - no two processes have same new name

Atomic Snapshot

- Array $SM[1:n]$ of n SWMR atomic registers

Atomic Snapshot

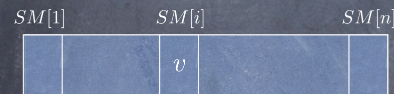
- Array $SM[1:n]$ of n SWMR atomic registers

Sequential specification

Process p_i can invoke 2 operations:

Atomic Snapshot

- Array $SM[1:n]$ of n SWMR atomic registers



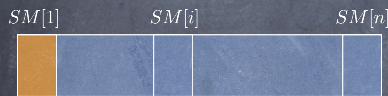
Sequential specification

Process p_i can invoke 2 operations:

- update(v)
  - sets $SM[i]$ to v

Atomic Snapshot

- Array $SM[1:n]$ of n SWMR atomic registers



Sequential specification

Process p_i can invoke 2 operations:

- update(v)
  - sets $SM[i]$ to v
- scan()
  - returns contents of $SM[1:n]$

Atomic Snapshot

- Array $SM[1:n]$ of n SWMR atomic registers

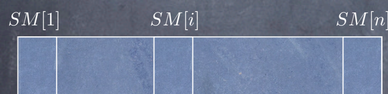
Sequential specification

Process p_i can invoke 2 operations:

- update(v)
  - sets $SM[i]$ to v
- scan()
  - returns contents of $SM[1:n]$

Atomic Snapshot

- Array $SM[1:n]$ of n SWMR atomic registers



Sequential specification

Process p_i can invoke 2 operations:

- update(v)
  - sets $SM[i]$ to v
- scan()
  - returns contents of $SM[1:n]$
  - each entry returns value of latest preceding update

How to implement scan?

- Read values one at a time (collect)
  - fine if no one updated during collect
    - clean collect
  - otherwise result may not be linearizable

How can we implement a
wait-free
atomic snapshot?

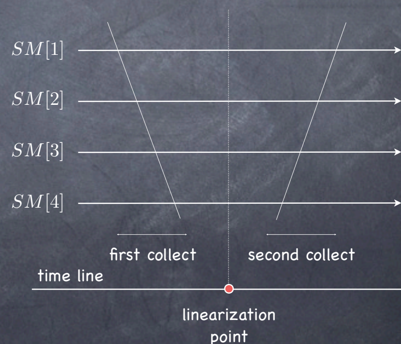
Obstruction-free Atomic Snapshot

- Add to each register a timestamp
- update(v)
  - increments timestamp $sn_i := sn_i + 1$
  - updates value $SM[i] := (v, sn_i)$
- scan()
  - perform two collects in sequence
  - if they read the same set of timestamps, then linearize scan sometime after first collect and before second

Obstruction-free Atomic Snapshot

operation update(v)
  $sn_i := sn_i + 1$ /* local seq. no. generator */
  $SM[i] := (v, sn_i)$ /*atomic write */

operation scan()
  while true do
    $A_i :=$ scan;
    $B_i :=$ scan;
    if $(\forall j : A_i[j].sn = B_i[j].sn)$ then
      return $A_i.val$
    end if
  end while



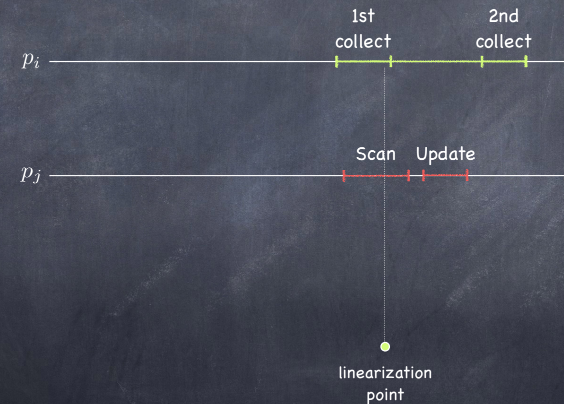
Why only obstruction-free?

If writers keep coming, may never get two successive clean collects!

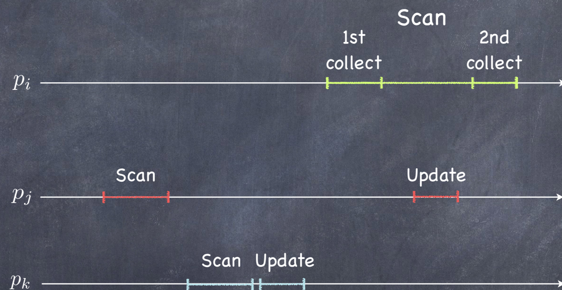
Wait-Free Atomic Snapshot

- A little help from your friends
  - add a scan before every update
  - write resulting snapshot together with new value
  - a scan continuously interrupted by updates, can take the one of the updates' snapshots

In action



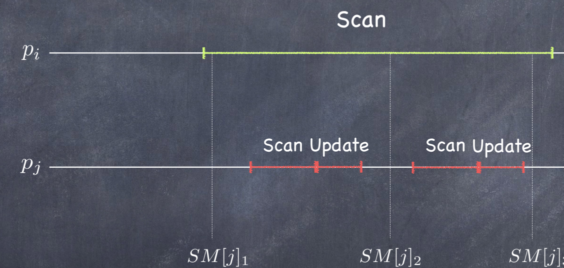
In action



- Not linearizable
  - snapshot collected by update may precede start of p_j's scan

The Gang of Six algorithm

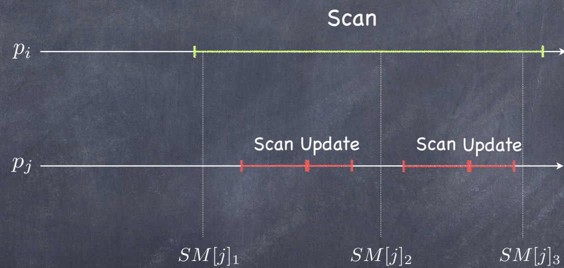
(Afek, Attiya, Dolev, Gafni, Merritt and Shavit, JACM 93)



- Suppose p_i sees three different $SM[j]$ during scan

The Gang of Six algorithm

(Afek, Attiya, Dolev, Gafni, Merritt and Shavit, JACM 93)



- Suppose p_i sees three different $SM[j]$ during scan
  - snapshot p_j captured before $SM[j]_3$ must totally overlap with p_i's scan
    - can be used as linearization point for p_i's scan

The Gang of Six algorithm

operation update(v)
  $help_array_i :=$ scan( )
  $sn_i := sn_i{+}1$
  $SM[i] := (v, sn_i, help_array_i)$

The Gang of Six algorithm

operation update(v)
  $help_array_i :=$ scan( )
  $sn_i := sn_i{+}1$
  $SM[i] := (v, sn_i, help_array_i)$

operation scan( )
  $could_help_i := \emptyset$
  while true do
    $A_i :=$ scan( )
    $B_i :=$ scan( )
    if $(\forall j : A_i[j].sn = B_i[j].sn)$
      then return $(A_i.val)$
      else for $j : 1 \leq j \leq n$ do
        if $(A_i[j].sn \neq B_i[j].sn)$ then
          if $(j \in could_help_i)$
            then return $(B_i[j].help_array)$
            else $could_help_i := could_help_i \cup \{j\}$
        end if end if
      end for
    end if
  end while

Proof

operation update(v)
  $help_array_i :=$ scan( )
  $SM[i] := (v, sn_i, help_array_i)$
  $sn_i := sn_i{+}1$

operation scan( )
  $could_help_i := \emptyset$
  while true do
    $A_i :=$ scan( )
    $B_i :=$ scan( )
    if $(\forall j : A_i[j].sn = B_i[j].sn)$
      then return$(A_i.val)$
      else for $j : 1 \leq j \leq n$ do
        if$(A_i[j].sn \neq B_i[j].sn)$then
          if $(j \in could_help_i)$
            then return $B_i[j].help_array$
            else $could_help_i := could_help_i \cup \{j\}$
        end if end if
      end for
    end if
  end while

Lemma 1: snapshot is wait free

Proof

```
operation update(v)
    help_array_i := scan()
    SM[i] := (v, sn_i, help_array_i)
    sn_i := sn_i + 1

operation scan()
    could_help_i := ∅
    while true do
      A_i := scan()
      B_i := scan()
      if (∀j : A_i[j].sn = B_i[j].sn)
        then return(A_i.val)
        else for j : 1 ≤ j ≤ n do
          if (A_i[j].sn ≠ B_i[j].sn) then
            if (j ∈ could_help_i)
              then return B_i[j].help_array
              else could_help_i := could_help_i ∪ {j}
          end if end if
       end for
     end if
    end while
```

Lemma 1: snapshot is wait free

- Let p_i invoke snapshot
  - if p_i collects < $n+1$ times, done
  - if p_i collects $n+1$ times
    - some p_j executed update() twice
    - p_j is in $could_help_i$
    - p_i returns $B_i[j].help_array$

Proof

```
operation update(v)
    help_array_i := scan()
    SM[i] := (v, sn_i, help_array_i)
    sn_i := sn_i + 1

operation scan()
    could_help_i := ∅
    while true do
      A_i := collect()
      B_i := collect()
      if (∀j : A_i[j].sn = B_i[j].sn)
        then return(A_i.val)
        else for j : 1 ≤ j ≤ n do
          if (A_i[j].sn ≠ B_i[j].sn) then
            if (j ∈ could_help_i)
              then return B_i[j].help_array
              else could_help_i := could_help_i ∪ {j}
          end if end if
       end for
     end if
    end while
```

Lemma 1: snapshot is wait free

- Let p_i invoke snapshot
  - if p_i collects < $n+1$ times, done
  - if p_i collects $n+1$ times
    - some p_j executed update() twice
    - p_j is in $could_help_i$
    - p_i returns $B_i[j].help_array$

Lemma 2: update() and scan() are atomic:

- total order of ops respects real time
- $\forall p_j$ if update$_j(v)$ is last before scan(), then $help_array[j] = v$

- update ops: linearized at time writing ends
- scans that terminate after double collect: linearized some time between the two collect
- scans that are helped terminate:
  - by induction, last helping p_k terminated by double scan
  - helped snapshot inherits linearization point

Yes, but?

- Can we make any object wait-free?
  - What primitives are necessary / sufficient for constructing wait-free objects?
- How do we build a wait-free object?
  - Is there a universal constructor?
- How do we know the implementation is correct?

You can breathe. Thank Herlihy.

- A wait-free implementation of an object can be built out of any object with the same consensus number.
- There is a universal constructor using an object of infinite consensus number.
- We can verify correctness by showing that the implementation is linearizable.

A consensus protocol is...

A system of n threads, where

- each T_i starts with an input value from some domain D
- threads communicate by operating on a set of objects
- processes eventually agree on some input value and halt

Required properties:

- agreement: distinct threads never decide distinct values
- validity: common decision value is input of some thread
- wait-freedom: each thread decides after finite number of steps

A consensus protocol is also...

- A linearizable and wait-free implementation of a consensus object
- A consensus object provides a single operation
  - decide(input: value) returns(value)
  - Sequential spec: all "decides" return value of first decide

Consensus Number

Definition The consensus number of an object of class C is the largest n for which that class solves n-thread consensus. If no largest n exists, the consensus number is said to be infinite

Corollary Suppose an object of class C can be implemented by one or more objects of class D, together with some number of atomic registers.

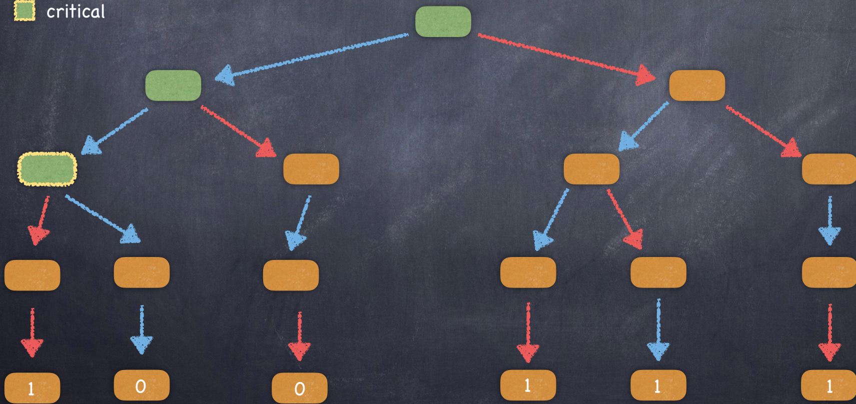
If C can solve n-thread consensus, so can D

States and Valence

- Protocol state
  - states of threads and shared objects
  - initial state, final state (with a decision value)
- Move
  - method call to an object
- Bivalent, monovalent states
  - label indicating the decision values of the final states that this state can reach
- Critical protocol state
  - bivalent, but if any thread moves, resulting state monovalent

Execution tree

(binary consensus, two threads)



Atomic registers have consensus number 1

SAD!

Lemma Every n-thread consensus protocol has a bivalent initial state

Proof. See 5414

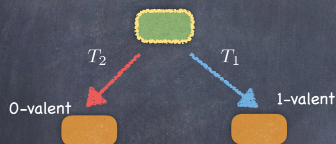
Lemma Every wait-free consensus protocol has a critical state

Proof. Start from a bivalent initial state. As long as a thread can move to a bivalent state, make it do so. If the protocol runs forever, it is not wait free - otherwise, it must enter a critical state.

Atomic registers have consensus number 1

SAD!

Proof. (Binary consensus, two threads) Run until a critical state



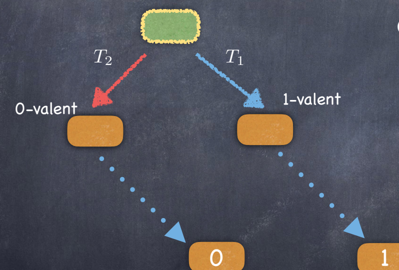
Case 1: T_2 is about to read a register

Atomic registers have consensus number 1

SAD!

Proof. (Binary consensus, two threads) Run until a critical state

T_2 T_1 0-valent 1-valent



indistinguishable by T_1

Case 1: T_2 is about to read a register

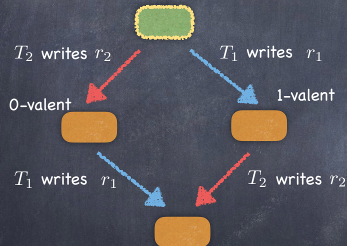
- T_1 takes a step to a 1-valent state
- T_1 runs solo to a state where the decision is 1

- T_2 reads the register
- T_1 runs solo to a state where the decision is 0

Atomic registers have consensus number 1

SAD!

Proof. (Binary consensus, two threads) Run until a critical state



Case 2: T_1, T_2 write to different registers

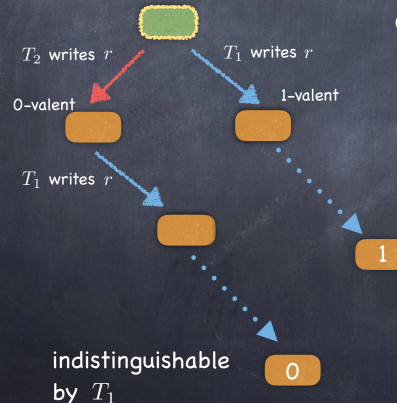
- T_1 writes r_1 (1-valent state)
- T_2 writes r_2 (0-valent state)

- T_1 writes r_1 (0-valent state)
- T_2 writes r_2 (1-valent state)

Atomic registers have consensus number 1

SAD!

Proof. (Binary consensus, two threads) Run until a critical state



indistinguishable by T_1

Case 2: T_1, T_2 write to the same register

- T_1 writes r (1-valent state)
- T_1 runs solo to a state where the decision is 1

- T_2 writes r (0-valent state)
- T_1 overwrites r
- T_1 runs solo to a state where the decision is 0

Why it matters

Corollary It is impossible to build a wait-free implementation of any object with consensus number greater than 1 using atomic registers

Hardware must support more than load/store to build lock-free concurrent data structures

FIFO queue consensus

Input values

Fifo queue

Theorem The 2-dequeuer FIFO queue has consensus number at least 2

```
void  propose (T value) {
   proposed[ThreadID.get()] := value
}

public T decide (T value) {
   propose(value);
   int status := queue.deq();
   int i := ThreadID.get();
   if (status = W)
        return proposed[i]
   else
        return proposed[i-1]
}
```

FIFO queue consensus

- Not hard to prove that FIFO queues cannot solve 3-thread consensus

Theorem FIFO queues have consensus number 2

... and so do stacks, sets, priority queues, etc

Corollary It is impossible to implement a wait-free FIFO queue (set, stack, etc) from atomic read/write registers