6.852: Distributed Algorithms Spring, 2008

Class 19

Today's plan

- Wait-free synchronization.
- The wait-free consensus hierarchy
- Universality of consensus
- Reading:
  - [Herlihy, Wait-free synchronization] (Another Dijkstra Prize paper)
  - [Attiya, Welch, Chapter 15]

Overview

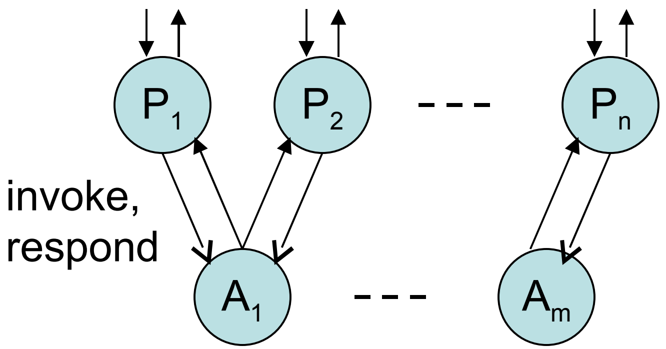
- General goal:
  - Classification of atomic object types: Which types of objects can be used to implement which other types, for which numbers of processes and failures.
  - A theory of relative computability, for objects in distributed systems.
- Follow Herlihy's approach.
- Considers wait-free termination only (n-1 failures).
- Object types:
  - Primitives used in multiprocessor memories:
    - Test-and-set, fetch-and-add, compare-and-swap
  - Standard programming data types:
    - Counters, queues, stacks
  - Consensus, k-consensus
- Hierarchy of types, with:
  - Read/write registers at the bottom, level 1
  - Consensus (viewed as an atomic object) at the top, level ∞
  - Others in between.
- Universality result: Can use consensus for n processes to implement (wait-free) any object for n processes.

Herlihy's Hierarchy

- Defines hierarchy in terms of:
  - How many processes can solve consensus using only objects of the given type, plus registers (thrown in for free).
- Shows that no object type at one level of the hierarchy can implement objects at higher levels.
- Shows:
  - Read/write registers are at level 1.
  - Stacks, queues, fetch-and-add, test-and-set are at level 2.
  - Consensus, compare-and-swap are at "level ∞".
- Hierarchy has a few limitations:
  - All of the interesting types are at level 1, 2 or ∞.
  - Gives no information about relative computability of objects at the same level.
  - Lacks some basic "robustness" properties.
- Yields some interesting classification results.
- More work is needed.

The Model

- Based on I/O automata.
  - Herlihy claims he doesn't need tasks, but we'll use them to define fair executions, as usual.
- Concurrent system:
  - Processes + atomic objects



- Sequential specification = variable type
- Use such a concurrent system to implement an atomic object, of a specified type.
- Warning: Herlihy's definition of implementation consider one object R, but results allow many objects (of one type).

Consensus as an atomic object

- Define the consensus variable type (X, x_0, invs, resps, δ):
  - Let V = consensus domain.
  - Define $X = V \cup \{ \perp \}$.
  - $x_0 = \perp$
  - invs = { init(v) | $v \in V$ }
  - resps = { decide(v) | $v \in V$ }
  - δ(init(v), $\perp$) = (decide(v), v), for any v in V
  - δ(init(w), v) = (decide(v), v), for any v, w in V
- First value provided in an init() operation is everyone's decision.

- Herlihy's consensus object is simply a wait-free atomic object for the consensus variable type.
- Lets him consider atomic objects everywhere:
  - For high-level objects being implemented, and
  - For low-level objects used in the implementations.
- Usually treats low-level objects as shared variables (as we do).

Herlihy’s consensus object vs. our consensus definition

- Herlihy’s consensus atomic object is “almost the same” as our definition of consensus:
  - Satisfies well-formedness, agreement, strong validity (every decision is someone’s initial value).
  - Wait-free termination.
    - Every init() on a non-failing port eventually receives a decide() response.
- Some (unimportant) differences:
  - Allows repeated operations on the same port; but all get the same value v.
  - Inputs needn’t arrive everywhere; equivalent requirement (Exercise 12.1)

Binary vs. arbitrary consensus

- Herlihy’s paper talks about “implementing consensus”, without specifying the domain.
- Doesn’t matter:
- Theorem: Let **T** be the consensus type with domain { 0,1 }, and **T'** the consensus type with any finite value domain V.

  Then there is a wait-free implementation of an n-process atomic object of type **T'** from n-process shared variables of type **T** and read/write registers.

Algorithm

- Shared variables:
  - Boolean consensus objects, cons(1),…,cons(k), where k is the length of a bit string representation for elements of V.
  - Registers init(1),…, init(n) over V ∪ { ⊥ }, where V is the consensus domain, initially all ⊥.
- Process i:
  - Post your initial value in init(i), as a bit string.
  - Maintain a current preference, locally, initialized to init(i).
  - For l = 1 to k do:
    - Engage in binary consensus on cons(l), with l-order bit of your current preference as input.
    - If your bit loses, then:
      - read all init(j) registers to find some value whose first l-1 bits agree with your current preference, and whose l'th bit is the winning bit from cons(l).
      - Reset your preference to this init(j).
  - Return your final preference.

Extension to infinite V

- Theorem: Let **T** be the consensus type with domain { 0,1 }, **T'** the consensus type with any value domain V.

  Then there is a wait-free implementation of an n-process atomic object of type **T'** from n-process shared variables of type **T** and read/write registers.
- Proof:
  - Similar algorithm.
  - Now reach consensus on index j, rather than value, for some active process (one that wrote init(j)).
  - Then return init(j).

- Moral: When we talk about "solving consensus", we needn't specify V (unless we care about complexity).

Consensus Numbers

- Definition: The *consensus number* of a variable type **T** is the largest number n such that shared variables of type **T** and read/write registers can be used to implement an n-process wait-free atomic consensus object.
- That is, **T** + registers solve n-process consensus.
- Notes:
  - Registers are thrown in for free.
  - Convenient in writing algorithms.
  - OK because they are at the bottom of the hierarchy (consensus number 1).
    - Follows from [Loui, Abu-Amara]
- Definition: If **T** + registers solve n-process consensus for every n, then we say that **T** *has consensus number* **∞.**

Consensus Numbers

- Consensus numbers yield a way of showing that one variable type **T** cannot implement another variable type **T',** for certain numbers of processes.

- Theorem 1: Suppose cons-number(**T**) = m, and cons-number(**T'**) > m. Then there is no (wait-free) implementation of an atomic object of type **T'** for n > m processes, from shared variables of type **T** and registers.
- Proof:
  - Enough to show for n = m+1.
  - By contradiction. Suppose there is an (m+1)-process implementation of an atomic object of type **T'** from **T** + registers.
  - Since cons-number(**T'**) > m, there is an (m+1)-process consensus algorithm C using **T'** + registers.
  - Replace the **T϶**shared variables in C with the assumed implementation of **T'** from **T** + registers.
  - By our composition theorem, this yields an (m+1)-process consensus algorithm using **T** + registers.
  - Contradicts assumption that cons-number(**T**) = m.

Example: Read/write register types

- Theorem 2: Any read/write register type, for any value domain V and any initial value v_0, has consensus number 1.
- Proof:
  - Clearly, can be used to solve 1-process consensus (trivial).
  - Cannot solve 2-process consensus [Book, Theorem 12.6].

- Corollary 3: Suppose cons-number(**T'**) > 1. Then there is no (wait-free) implementation of an atomic object of type **T'** for n > 1 processes, from registers only.
- Proof:
  - By Theorems 1 and 2.
  - Let type **T** be any register type, in Theorem 1.

Example: Snapshot types

- Corollary3: Suppose cons-number(**T'**) > 1. Then there is no (wait-free) implementation of an atomic object of type **T'** for n > 1 processes, from registers only.

- Theorem 4: Any snapshot type, for any underlying domain (W, w_0), has consensus number 1.

- Proof:
  - By contradiction.
  - Suppose there is a snapshot type **T'** with cons-number(**T'**) > 1.
    - That is, it can be used to solve 2-process consensus.
  - Then by the Corollary, there is no wait-free implementation of an atomic object of type **T'** for > 1 processes, from registers only.
  - Contradicts known implementation of snapshots from registers.

Example: Queue types

- Define a FIFO queue type queue(V,q0), where:
  - V is a value domain.
  - q_0 is a finite sequence giving the initial queue contents.
  - Operations:
    - enqueue(v), v in V: Add v to end of queue, return ack.
    - dequeue(): Return head of queue if nonempty, else $\perp$.
- Most common case: $q_0 = \lambda$, empty sequence.

- Theorem 5: There is a queue type **T** with cons-number(**T**) ≥ 2.
- Proof:
  - Construct a 2-process consensus algorithm for domain V.
  - Shared variables:
    - One queue of { 0 }, $q_0 = 0$
    - Registers, init(1) and init(2) over $V \cup \{ \perp \}$, initially $\perp$.
  - Process i:
    - Post your initial value in init(i).
    - Perform dequeue().
    - If you get 0, return your initial value.
    - Else (you get $\perp$), read and return init(j), for the other process j.

Queue algorithm

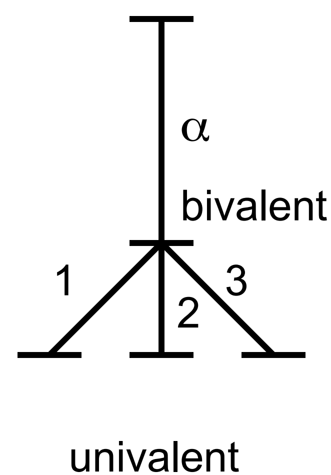
- Theorem 6: There is a queue type T with cons-number(T) ≥ 2.

- Corollary 7: There is no wait-free implementation of an n-process atomic object of the above queue type using registers only, for any n ≥ 2.
- Proof:
  - By Corollary 3.
  - Essentially: suppose there is. Plug it into the above 2-process consensus algorithm and get a 2-process consensus algorithm using registers only, contradiction.

- Q: What about queues with other initial values q0?
- E.g., initially-empty queues?
  - Claim there's an algorithm, but more complicated. Exercise?
- What about other, known initial values?

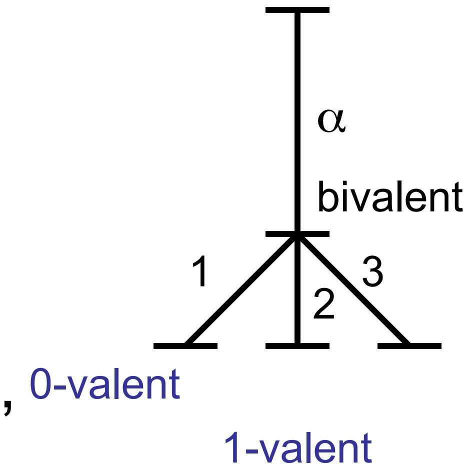
Queue impossibility

- Theorem 8: Every queue type **T** has cons-number(**T**) ≤ 2.
- More strongly: No combination of queue variables, with any queue types, initalized in any way, plus registers, can implement 3-process consensus.
- Proof:
  - Suppose such an algorithm, A, exists.
  - As for the register-only case, we can show that A has a bivalent initialization.
  - Furthermore, we can maneuver as before to a decider configuration:

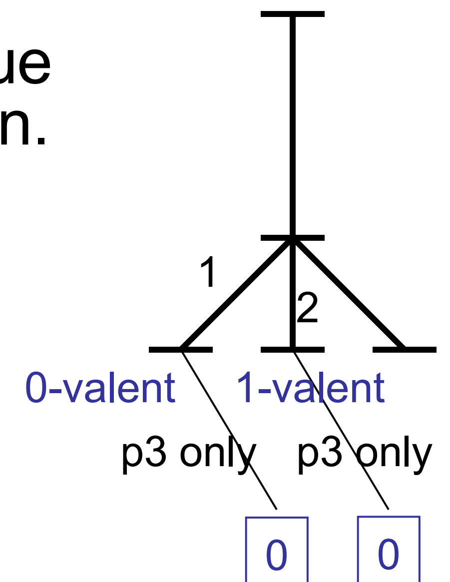


Queue impossibility

- Suppose WLOG that process 1 is 0-valent, process 2 is 1-valent.
- Consider what p1 and p2 can do in their steps.
- If they access different variables, or both access the same read/write shared variable, we get contradictions as in the purely read/write case.
- So assume they both access the same queue q; consider cases based on type of operation.



- Case 1: p1 and p2 both dequeue:
  – Then resulting states look the same to p3.
  – Running p3 after both yields a contradiction.

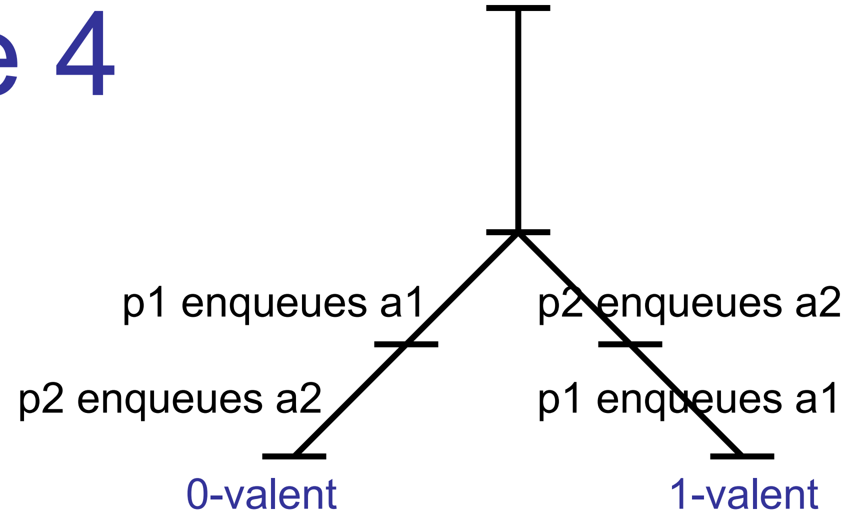


Cases 2 and 3

- Case 2: p1 enqueues and p2 dequeues:
  - If the queue is nonempty after α, the two steps commute---same system state after p1 p2 or p2 p1, yielding a contradiction.
  - If the queue is empty after α, then the states after p1 and p2 p1 look the same to all but p2 (and the queue is the same).
  - Running p3 (or p1) alone after both yields a contradiction.

- Case 3: p1 dequeues and p2 enqueues:
  - Symmetric.

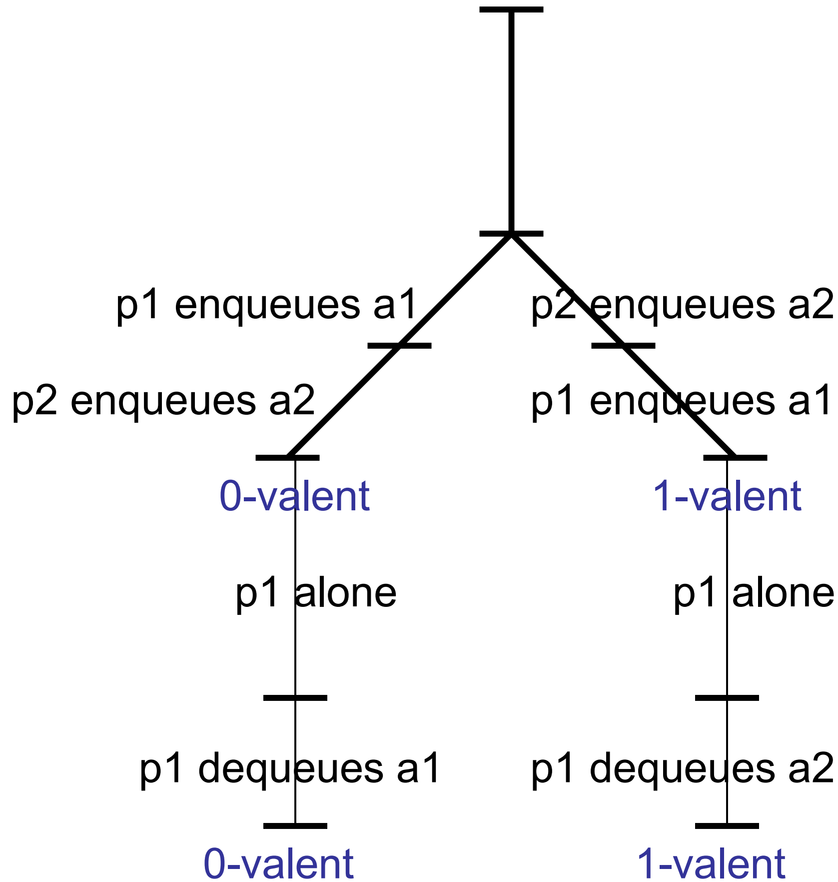
Case 4



- Case 4: p1 and p2 both enqueue:
  - Consider two possible orders:
  - Construct two executions:
    - After p1 p2, p1 runs alone until it dequeues a1, then p2 runs alone until it dequeues a2.
    - After p2 p1, p1 runs alone until it dequeues a2, then p2 runs alone until it dequeues a1.
  - These two executions are indistinguishable by p3, leading to the usual sort of contradiction.
  - Constructing the executions:
    - Q: What is different after p1 p2 and p2 p1?
    - Only the one queue q, which has a1 a2 in first case, a2 a1 in second.
    - States of all processes, values of other objects, are the same in both.

Constructing the executions

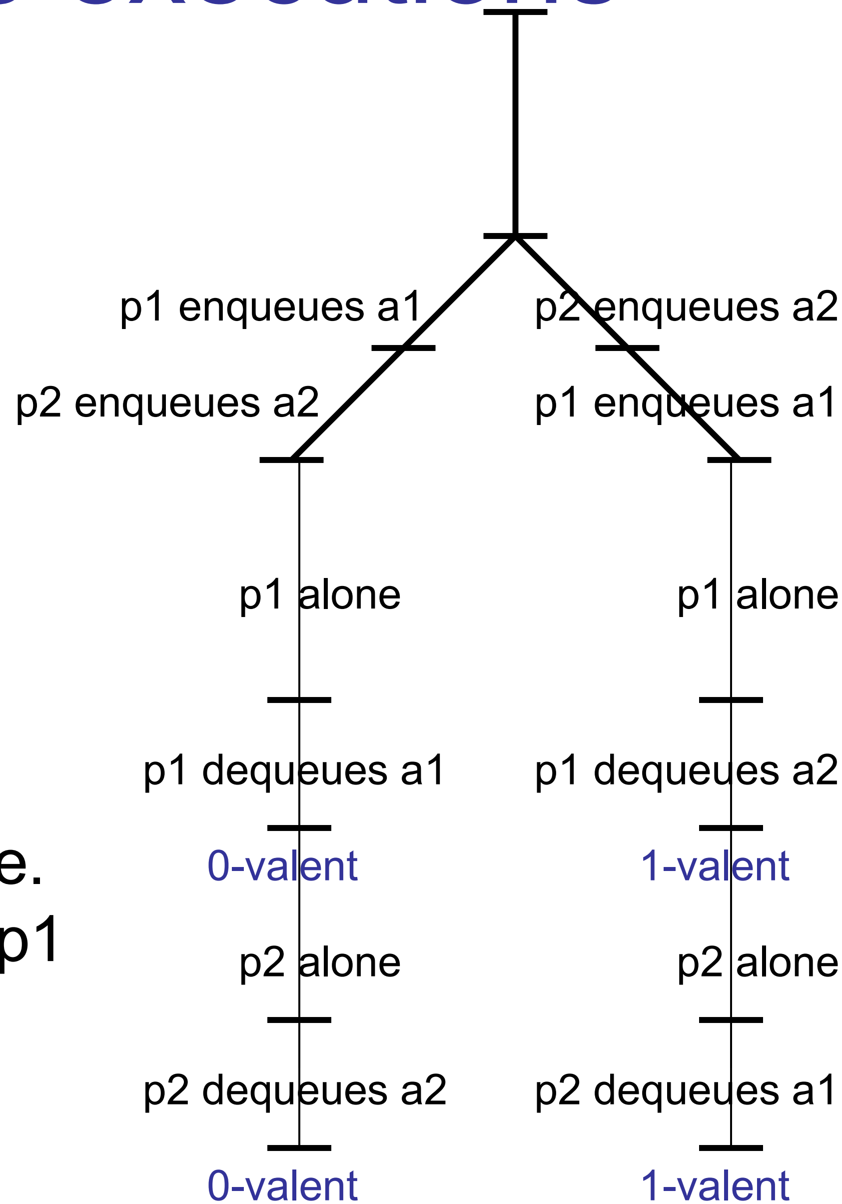
- Run p1 alone after p1 p2 and after p2 p1.
- Must eventually decide, differently in these two situations.
- But p1 can't distinguish until it dequeues from q, so it must eventually do so.
- So run p1 alone just until it dequeues from q.



- Q: Now what is different?
- q has just a2 on left branch, just a1 on right branch
- States of all other objects are the same.
- States of p2 and p3 are the same, but p1 may be different.

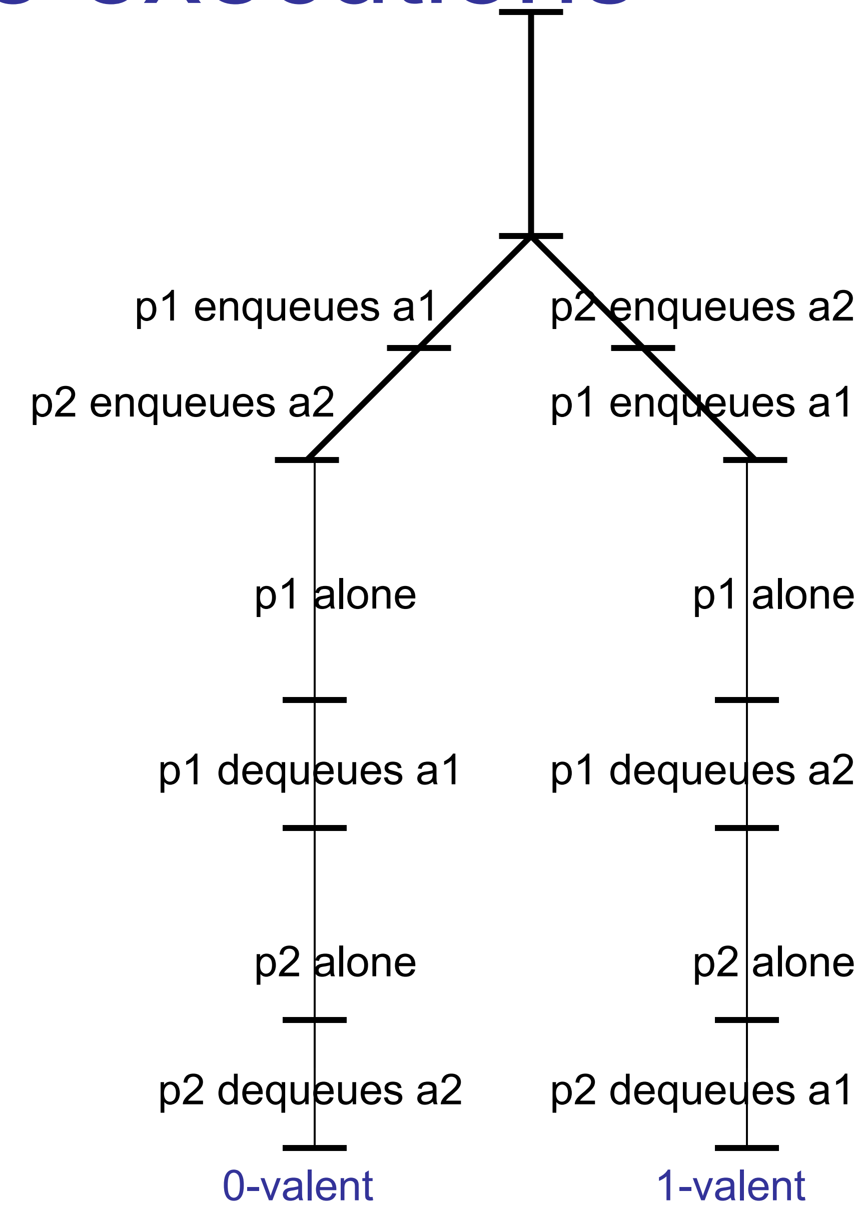
Constructing the executions

- Now run p2 alone after both branches.
- Must decide differently in the two executions.
- But p2 can't distinguish until it dequeues from q, so it must eventually do so.
- So run p2 alone just until it dequeues from q.
- Q: Now what is different?
- All objects, including q, are same.
- State of p3 is the same, though p1 and p2 may be different.



Constructing the executions

- This gives the needed executions.
- As noted earlier, just run p3 alone after both to get the contradiction.



Queue types: Recap

- We just showed:
  - Theorem 8: Every queue type **T** has cons-number(**T**) ≤ 2.
  - In fact, all queue types together can't solve 3-process consensus.
  - So cons-number(**T**) definition doesn't tell the entire story.
- Also:
  - Theorem 5: There is a queue type **T** with cons-number(**T**) ≥ 2.

Example: Compare-and-swap types

- Define a compare-and-swap type:
  - V, the value domain.
  - v_0, initial value.
  - invs = { compare-and-swap(u,v) | u, v in V }
  - resps = V
  - δ(compare-and-swap(u,v), w) =
    - (w, v) if u = w,
    - (w, w) if not.
- That is, if the variable value is equal to the first argument, change it to the second argument; otherwise leave the variable alone.
- In either case, return the former value of the variable.

Compare-and-swap types

- Theorem 9: Let **T** be the consensus type with value domain V. Then there is a compare-and-swap type **T'** that can be used to implement an n-process consensus object with type **T**, for any n.
- That is, **T'** can be used to solve n-process consensus for any n; so cons-number(**T'**) = ∞.
- Proof:
  - Use just a single C&S shared variable, value domain = V ∪ { ⊥ }, initial value = ⊥.
  - Process i:
    - If initial value = v, then access the C&S shared variable with compare-and-swap(⊥, v), obtain the previous value w.
    - If w = ⊥ then decide v. (You are first).
    - Otherwise, decide w. (Someone else was first and proposed w.)

Compare-and-swap types

- Corollary 10: It is impossible to implement an atomic object of this C&S type **T'** for $n \geq 3$ processes using just FIFO queues and read/write registers.
- Proof: By Theorem 1, with $m = 2$.

- Recall Theorem 1: Suppose cons-number(**T**) = m, and cons-number(**T'**) > m. Then there is no (wait-free) implementation of an atomic object of type **T'** for $n > m$ processes, from shared variables of type **T** and registers.

- Herlihy paper classifies other data types similarly, LTTR.

Universality of consensus

- Consensus variables and registers can implement a wait-free n-process atomic object of any variable type, for any number n.
- Algorithm in paper combines:
  - A basic unfair, non-wait-free algorithm.
  - A fairness mechanism, to ensure that every operation gets completed.
  - Optimizations, to reuse memory, save time.
- [Attiya, Welch, Chapter 15] separate these three aspects.
- Here, we'll simplify by forgetting the optimizations.

- Assume an arbitrary data type **T** = (V, v0, invs, resps, δ).
- Fix n.

Non-wait-free algorithm

- Shared variables:
  - An infinite sequence of n-process consensus variables, cons(1), cons(2),…
  - Each variable's domain is

    { (i, k, a), where:
    - $1 \leq i \leq n$, a process id,
    - k is a positive integer, a local sequence number,
    - $a \in$ invs, a particular invocation for the **T** object }
- cons(j) is used to decide which invocation on the object being implemented is the j^{th} one to be performed.
- The consensus objects explicitly decide on this sequence, and it's consistently observed everywhere.
- Process i:
  - Participates in consensus executions in order 1,2,3,…
  - Keeps track locally of the decision values for all consensus variables; these are triples (j,k,a).
  - Knowing the sequences of decisions allows process I to "run" the sequence and compute the new states and responses.

Non-wait-free algorithm, process i

- When a new invocation a arrives:
  - Record it in local state, as current-inv.
  - This is a triple (i, k, a), where k is the first unused local sequence number.
  - For each cons(j), starting from the first one that i hasn't yet participated in:
    - Invoke init(current-inv) on cons(j).
    - If returned decision = current-inv then
      - Record it in local state.
      - Run the sequence of invocations in the decisions up to j to get the response.
      - Return response to the user and quiesce.
    - If returned decision ≠ current-inv then
      - Record it in local state
      - Continue on to j+1.

Algorithm properties

- Well-formed: Yes
- Atomic: Yes
  - Everyone sees a consistent sequence of operations.
  - Serialization point for an operation can be the point where it wins at some consensus shared variable.
- Wait-free: No
  - Process i could submit the same operation to infinitely many cons variables, and it could always lose.

Wait-free algorithm

- Add a priority mechanism to ensure that each operation completes.
- For cons(j), $j \equiv i \bmod n$, current invocation of process i gets priority.
- Priority managed outside the consensus variables:
  - A process i sometimes "helps" another process j, by invoking a consensus objects with j's invocation instead of i's own.
- Additional shared variables:
  - announce(i), for each process i, a single-writer multi-reader register, written by i, read by everyone.
  - Value domain: $\{ (i, k, a) \text{ as above } \} \cup \{ \perp \}$.
  - Initial value: $\perp$

Wait-free algorithm, process i

- When a new invocation a arrives:
  - Record it in local state, as current-inv.
    - Triple (i, k, a), as before.
  - Write current-inv into announce(i).
  - Then proceed as in the non-wait-free algorithm, except:
    - Before participating in cons(j), read announce(j'), where $j \equiv j'$ mod n.
    - If announce(j') contains a triple inv (not ⊥), and inv has not already won any of cons(1), cons(2),…, cons(j-1), then invoke init(inv) on cons(j).
    - Otherwise, invoke init(current-inv) on cons(j), as before.
  - Handle decisions as before.
  - Just before returning value to the user, reset announce(i) := ⊥.

Algorithm properties

- Well-formed, Atomic: Yes, as before.
- Wait-free: Yes:
  - Claim every operation eventually completes.
  - If not, then consider some (i,k,a) that gets stuck.
  - Then after announce(i) is set to (i,k,a), it stays there forever.
  - Choose any j such that:
    - $j \equiv i \bmod n$, and
    - No one accesses cons(j), or even reads announce(i) in preparation for accessing cons(j), before announce(i) is set to (i,k,a).
  - Then for this j, everyone who participates will choose to help i by submitting (i,k,a) as input.
  - At least one process participates (i itself).
  - So the decision must be this (i,k,a).

Complexity

- Shared-memory size:
  - Infinitely many shared variables, each of unbounded size.
- Time:
  - Unbounded, because:
    - A process i may start with a cons(j) that is far out of date, have to access cons(j), cons(j+1),…to catch up.
- Herlihy:
  - Formulates the algorithm differently, in terms of a linked list of operations, so it's hard to compare.
  - Claims a nice O(n l) bound.
    - Avoids the catch-up time by allowing processes to record information they've seen, so i needn't go back to the beginning.
    - Use similar strategy for our algorithm?
  - Still has unbounded sequence numbers.
  - Still needs infinitely many consensus objects---seems unavoidable since each is good for only one decision
  - "Garbage-collects" to reclaim space taken by old objects.

Robustness

- [Jayanti] defined a robustness property for the hierarchy:
  - Robustness: If **T** is a type at level n, and **S** is a set of types, all at levels < n, then **T** has no implementation from **S** for n processes.
- But did not determine whether the hierarchy is robust.
- Herlihy's results don't imply this; they do imply:
  - If **T** is a type at level n, and **S** is a single type at a level < n, then **T** has no implementation from **S** and registers.
- But it's still possible that combining low-consensus-number types could allow implementation of a higher-consensus-number type.
- Later papers give both positive and negative results.
  - Based on technical issues.

Summary

- Work is still needed to achieve our original goals:
  - Determine which types of objects can be used to implement which other types, for which numbers of processes and failures.
  - A comprehensive theory of relative computability, for objects in distributed systems.

Next time…

- More on wait-free computability
- Wait-free vs. f-fault-tolerant computability
- Reading:
  - [Borowsky, Gafni, Lynch, Rajsbaum]
  - [Chandra, Hadzilacos, Jayanti, Toueg]
  - [Attie, Guerraoui, Kouznetsov, Lynch]