6.852: Distributed Algorithms
Fall, 2015

Class 20

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]
- Next time:
  - More on wait-free computability
  - Wait-free vs. f-fault-tolerant computability
  - Reading:
    - [Borowsky, Gafni, Lynch, Rajsbaum]
    - [Attiya, Welch, Section 5.3.2]
    - [Attie, Guerraoui, Kouznetsov, Lynch]

Overview

- General goal:
  - Classify atomic object types: Which types can be used to implement which others, for which numbers of processes and failures?
  - A theory of relative computability, for objects in distributed systems.
- Herlihy considers wait-free termination only (n-1 failures).
- Considers particular 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.
- Defines a 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 (but mostly at levels 1 and 2).
- Universality result: Consensus for n processes can be used to implement 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 any object at a higher level.
- 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 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, desirable "robustness" properties.
- Yields some interesting classification results.
- But doesn't give a complete story---more work is needed.

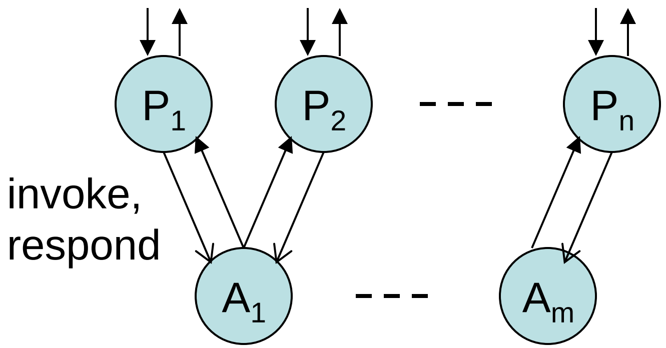
Outline

- Basic definitions
  - Consensus as an atomic object
  - Consensus numbers and the consensus hierarchy
- Queue types
  - Consensus number = 2
- Compare-and-swap types
  - Consensus number = ∞
- Universality of consensus

Basic definitions

The Model

- Concurrent system model:
  - Processes + atomic objects
- Herlihy models everything using I/O automata.
  - Uses a slight variant of tasks to define fair executions.
  - We'll keep using tasks.



- Sequential specification = variable type
- Use a concurrent system to implement an atomic object of a specified type.
- Warning: Herlihy's definition of implementation says only one object is used, but his results sometimes use many objects (of the same type).

Consensus as an atomic object

- Consensus variable type (X, x_0, invs, resps, δ):
  - V = consensus domain, $X = V \cup \{ \perp \}$.
  - $x_0 = \perp$
  - 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
- That is, the first value anyone provides in an init() operation is everyone's decision.
- Herlihy's consensus object is simply a wait-free atomic object for the consensus variable type.
- Enables him to consider atomic objects everywhere:
  - For high-level objects being implemented, and
  - For low-level objects used in the implementations.
- But, he generally treats low-level objects as shared variables.

Herlihy's consensus object vs. our consensus definition

- Herlihy's consensus atomic object is "almost the same" as our notion 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.
  - Doesn't add any new constraints.
- Some (unimportant) differences:
  - Allows repeated operations on the same port; but all get the same response 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.
- It doesn't matter:
- Theorem: Let **T** be the consensus type with domain { 0,1 }, and **T′** a consensus type with some other 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.

General consensus from binary

- Shared variables:
  - Binary consensus objects, Cons(1), …, Cons(k), where k is the length of a bit string representation for elements of V.
  - Read/write registers Init(1), …, Init(n) over $V \cup \{ \perp \}$, where V is the consensus domain, initially all $\perp$.
- Process i:
  - Post (write) initial value in Init(i), as a bit string.
  - Maintain current preferred value internally, initialized to initial value.
  - For l = 1 to k do:
    - Engage in binary consensus using Cons(l), with the 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 value.
  - Return your final preference.

What about an infinite set V?

- Theorem: Let **T** be the consensus type with domain { 0,1 }, **T′** a 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.
  - But now reach consensus on the index j for some active process, rather than the actual value (active means that it writes Init(j)).
  - Then return that j's initial value, read from Init(j).

- Moral: When we talk about solvability of consensus, we needn't specify V.

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.

- Note that read/write registers are thrown in for free.
  - Helpful in writing algorithms.
  - Reasonable because they are at the bottom of the hierarchy, consensus number 1. (Why?)
  - Follows from [Loui, Abu-Amara]: can't be used to solve even 2-process consensus.

- 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 be used (by itself, plus registers) to implement another 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:

Consensus Numbers

- 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 objects of type **T** + registers.
  - Since cons-number(**T′**) > m, there is an (m+1)-process consensus algorithm C using objects of type **T′** + registers.
  - Replace the **T′** shared variables in C with the assumed implementation of **T′** objects from **T** + registers.
  - By our composition theorem for shared-memory algorithms, 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.

Example: Snapshot types

- 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.

- 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.
    - Thus, it can be used to solve 2-process consensus.
  - Then by Corollary 3, 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.

Queue Types

Queue types

- FIFO queue type queue(V,q_0):
  - V is some 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 commonly: $q_0 = \lambda$, empty sequence.

- Theorem 5: There is a queue type **T** with cons-number(**T**) $\geq$ 2.
- Proof:

Queue types

- Theorem 5: There is a queue type **T** with cons-number(**T**) $\geq$ 2.
- Proof:
  - Construct a 2-process consensus algorithm for an arbitrary domain V, using queue shared variables.
  - Shared variables:
    - One queue of integers, initially = sequence consisting of one element, 0.
    - Registers Init(1) and Init(2) over $X = V \cup \{ \perp \}$, initially $\perp$.
  - Process i:
    - Post initial value in Init(i).
    - Perform dequeue().
    - If you get 0, then return your initial value.
    - Else (you get $\perp$), read and return Init(j), for the other process j.
  - First dequeuer wins.

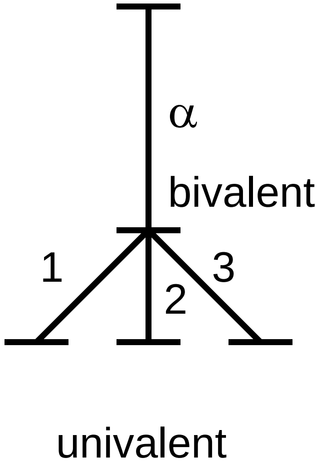
Queue types

- Theorem 5: There is a queue type T with cons-number(T) ≥ 2.
- Corollary 6: There is no wait-free implementation of an n-process atomic object of the above queue type using registers only, for any $n \geq 2$.
- 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 q_0?
- E.g., initially-empty queues?
  - Claim there's an algorithm, but it's more complicated. HW?
- Q: What about other, known initial values?

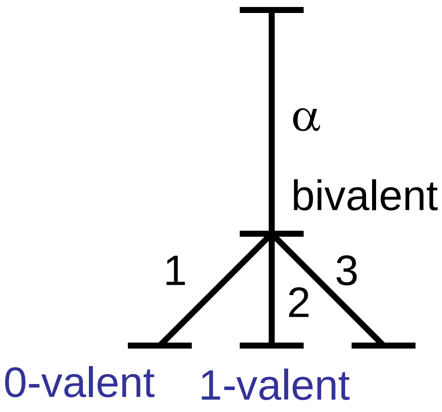
Queue lower bound

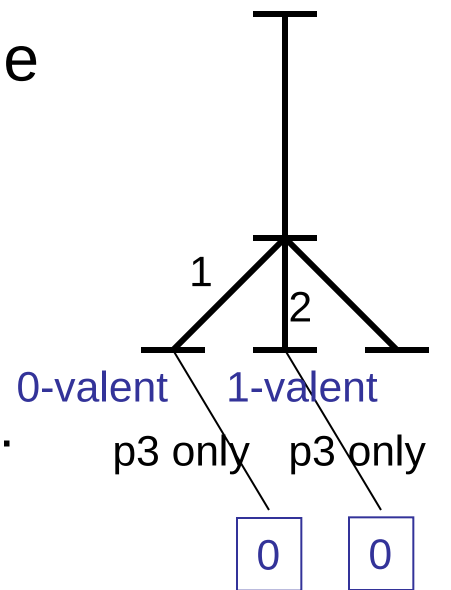
- Theorem 7: Every queue type **T** has cons-number(**T**) $\leq$ 2.
- More strongly: No combination of queue variables, with any queue types, initalized in any way, plus registers, can implement 3-process consensus.
- Proof:
  - Suppose such an algorithm, A, exists.
  - As for the register-only case, 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 yields 0-valence, process 2 yields 1-valence.
- Consider what p1 and p2 might do in their steps.
- If they access different variables, or both access the same register, we get contradictions as in the pure read/write case.
- So assume they both access the same queue q; consider cases based on types of operations they perform.
- Case 1: p1 and p2 both dequeue:
  - Then resulting states look the same to p3.
  - Running p3 alone after both yields a contradiction.



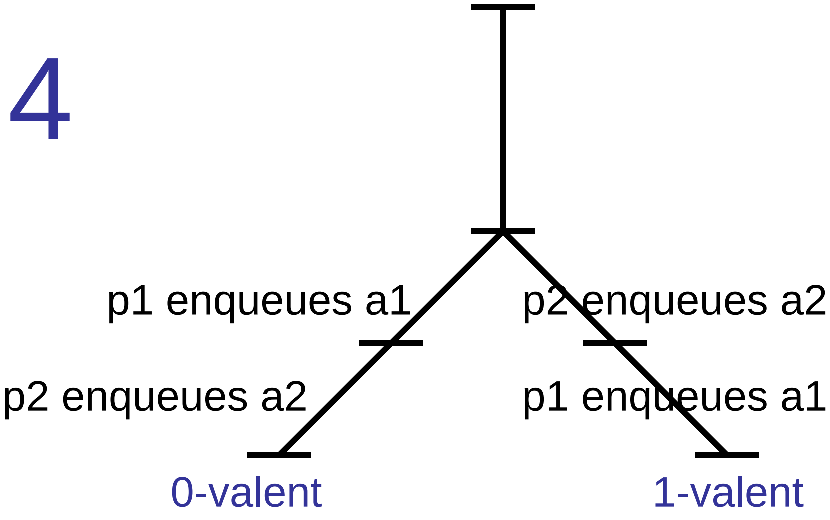


Case 2

- 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 in both cases).
  - Running p3 (or p1) alone after both yields a contradiction.

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

Case 4

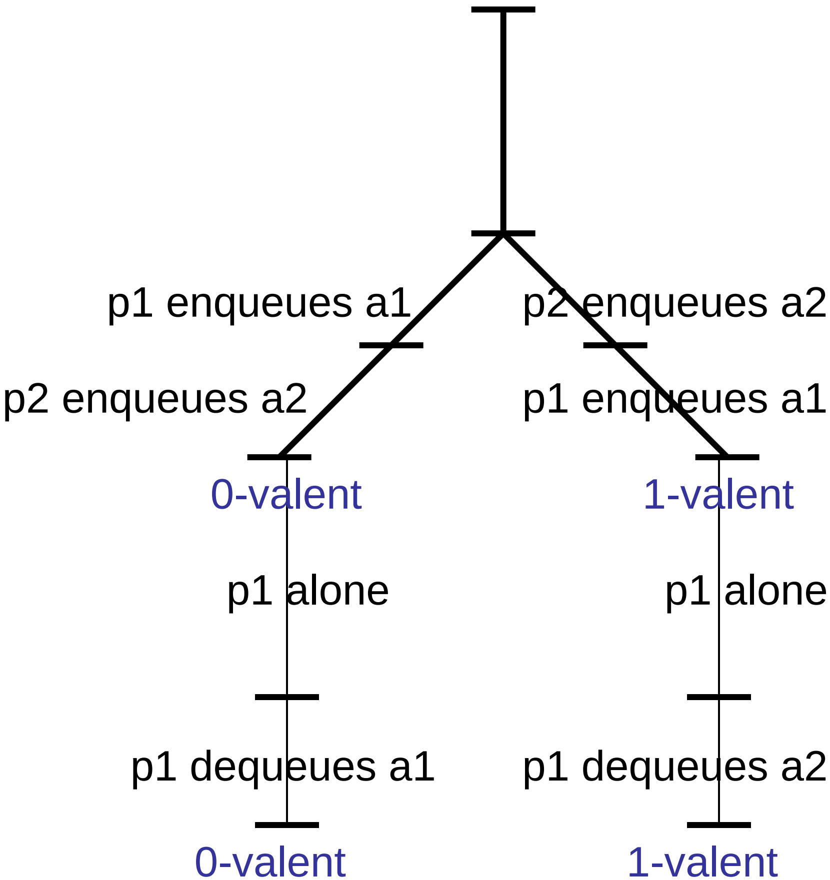


- Case 4: p1 and p2 both enqueue:
  - Consider two possible orders:
  - We will construct two executions:
    - After p1 p2, p1 runs alone until it dequeues a1, then p2 runs alone until it dequeues a2.
    - After p2 p1, p1 runs alone until it dequeues a2, then p2 runs alone until it dequeues a1.
  - These two executions are indistinguishable by p3, leading to the usual sort of contradiction.

  - But how do we construct these two executions?
    - Q: What is different after p1 p2 and p2 p1?
    - Only the queue q, which ends with 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 a1 or a2 from q, so it must eventually do so.
- So we can run p1 alone just until it dequeues a1 or a2 from q.

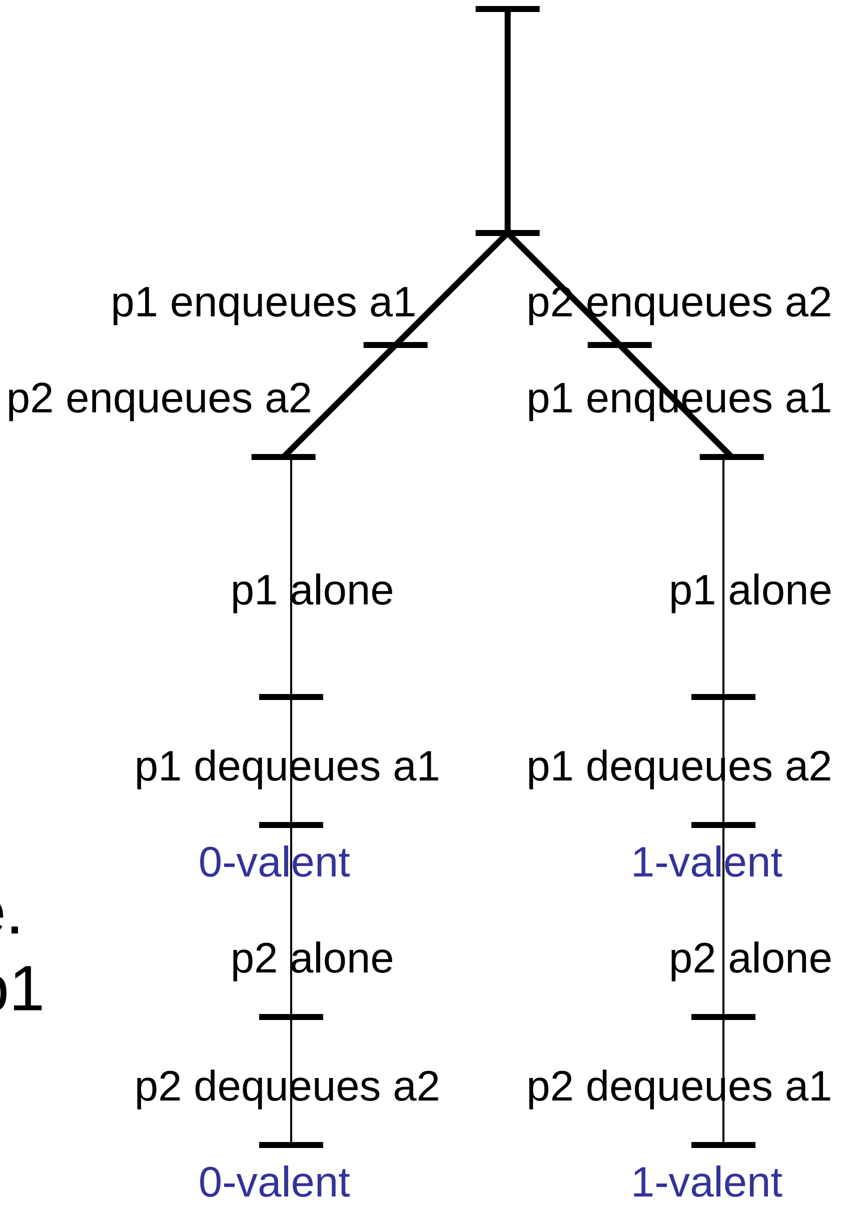


- Q: Now what is different?
- q starts with a2 on left branch, a1 on right branch, later elements are the same.
- States of all other objects are the same.
- States of p2 and p3 are the same, but p1 may be different.

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.

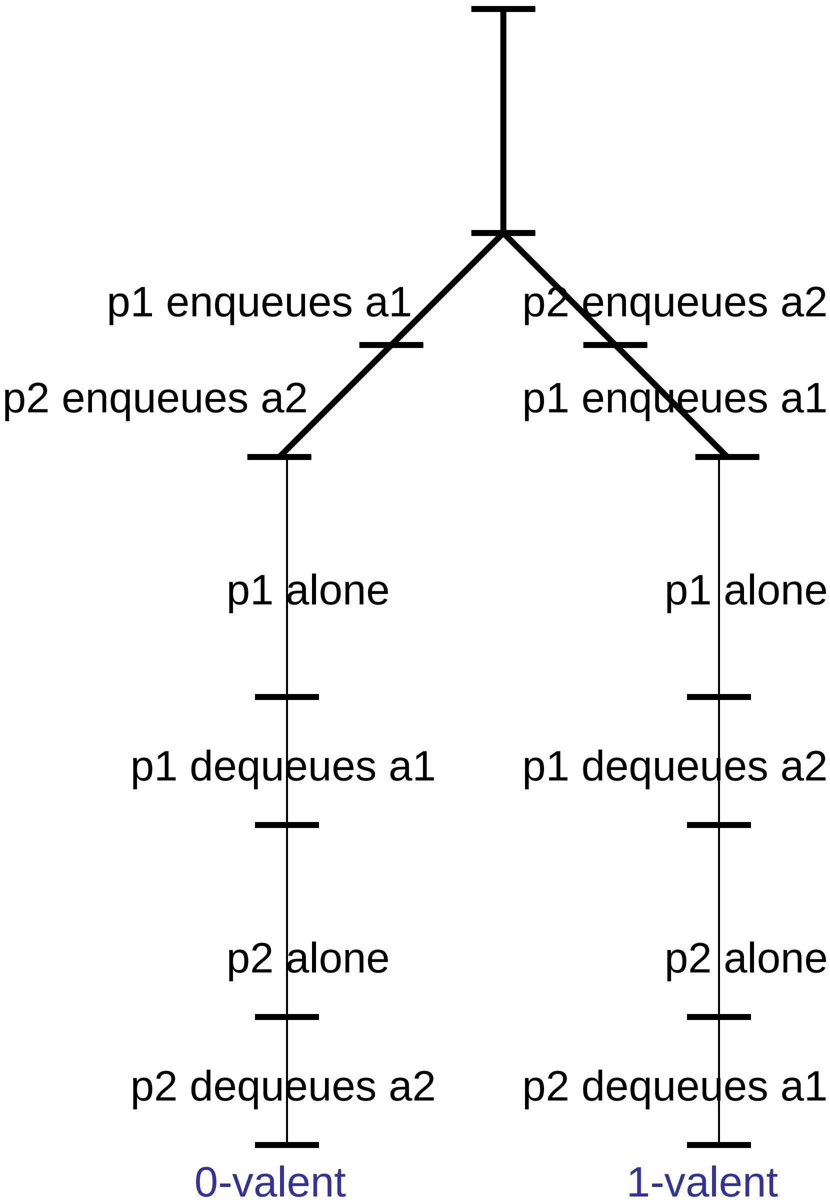
- Q: Now what is different?
- All objects, including q, are same.
- State of p3 is the same, though p1 and p2 may be different.



Constructing the executions

- This gives the needed executions:
  - 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.

- As described earlier, just run p3 alone after both to get the contradiction.



Queue types: Recap

- We just showed:
  - Theorem 7: Every queue type **T** has cons-number(**T**) $\leq$ 2.
  - In fact (stronger statement), 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**) $\geq$ 2.
- Gives quite a bit of information about the power of queue types.

Compare-and-Swap (CAS) Types

Compare-and-swap types

- 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 8: 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 = $V \cup \{ \perp \}$, initial value = $\perp$.
  - Process i:
    - If initial value = v, then access the C&S shared variable with compare-and-swap($\perp$, v), obtain the previous value w.
    - If w = $\perp$ then decide v. (You are first).
    - Otherwise, decide w. (Someone else was first and proposed w.)

Compare-and-swap types

- Corollary 9: It is impossible to implement an atomic object of this C&S type **T′** (from Theorem 8) for $n \geq 3$ processes using just queues and read/write registers.

- Proof: Like proof of Theorem 1.
  - Enough to show for $n = 3$.
  - By contradiction. Suppose there is a 3-process implementation of an atomic object of type **T′** using queues + registers.
  - By Theorem 8, there is a 3-process consensus algorithm C using just **T′** + registers.
  - Replace the **T′** shared variables in C with the assumed implementation of **T′** from queues + registers.
  - Yields a 3-process consensus algorithm using just queues + registers.
  - Contradicts (the stronger version of) Theorem 7.

- [Herlihy] classifies other data types similarly, LTTR.

Universality of Consensus

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 [Herlihy] combines:
  - A basic unfair, non-wait-free algorithm.
  - A fairness mechanism, to ensure that every operation is completed.
  - Optimizations, to reuse memory, save time.
- [Attiya, Welch, Chapter 15] separate these three aspects.
- Here, we'll simplify by ignoring the optimizations.

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

1. Non-wait-free algorithm

- Shared variables:
  - An infinite sequence of n-process consensus variables, Cons(1), Cons(2), …
  - Each consensus variable's domain is { (j, k, a) where:
    - j is a process id, $1 \leq j \leq n$,
    - k is a positive integer, a local sequence number,
    - $a \in$ invs, the set of invocations for the **T** object }
- Cons(r) is used to decide which proposed invocation on the implemented object is the r^{th} one to be performed.
- The consensus objects explicitly decide on the sequence of invocations, 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 of the form (j,k,a).
  - Knowing the sequence of consensus decisions allows process i to "run" the invocations in the sequence and compute the new states and responses for the implemented object.

Non-wait-free algorithm, process i

- When new invocation a arrives:
  - Record it in local variable current-inv, as a triple (i, k, a), where k is the first unused local sequence number.
  - For each Cons(r), starting from the first one that i hasn't yet participated in:
    - Invoke init(current-inv) on Cons(r).
    - Record decision in local variable decision(r).
    - If decision(r) = current-inv then
      - Run the sequence of invocations in decision(1), …, decision(r) to compute the response.
      - Return response to the user (and become idle).
    - Else continue on to r+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 Cons(r).
- Wait-free: No
  - Process i could submit the same operation to infinitely many Cons variables, and it could always lose.

2. Wait-free algorithm

- Add a simple priority mechanism to ensure that each operation completes.
- For Cons(r), $r \equiv i \bmod n$, any current invocation of process i gets priority.
- Priority is managed outside the consensus variables:
  - A process i sometimes "helps" another process j, by invoking 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 new invocation a arrives:
  - Record it in local variable current-inv as before, as triple (i, k, a).
  - Write value of current-inv into announce(i).
  - Then proceed as in the non-wait-free algorithm, except:
    - Before participating in Cons(r), read announce(j′), where $j' \equiv r \bmod n$.
    - If announce(j′) contains a triple inv (not $\perp$), and inv has not already won any of Cons(1), Cons(2), …, Cons(r-1), then invoke init(inv) on Cons(r).
    - Otherwise, invoke init(current-inv) on Cons(r).
  - Handle decisions as before.
  - Just before returning a response to the user, reset announce(i) := $\perp$.

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 keeps this value forever.
  - Process i participates in infinitely many consensus executions on behalf of this (i,k,a), losing all of them.
  - Choose any r such that:
    - $r \equiv i \bmod n$, and
    - r is sufficiently large so that no process accesses Cons(r), or even reads announce(i) in preparation for accessing Cons(r), 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 (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(r) that is far out of date, and have to access Cons(r), Cons(r+1),…to catch up.

- Herlihy:
  - Formulates the algorithm somewhat differently, in terms of a linked list of operations, so it’s hard to compare.
  - Time:
    - Claims a nice O(n) bound.
    - Avoids the catch-up time by allowing processes to survey others to get recent information.
  - Shared memory:
    - Still uses unbounded sequence numbers.
    - Still uses 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 he did not determine whether the hierarchy is robust.
- Herlihy's results don't imply this; what they say is:
  - 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 conceivable 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

- Basic definitions for wait-free consensus and the consensus hierarchy.
- Read/write register types, snapshot types, are at level 1.
- Queue data types are at level 2.
- Compare-and-Swap data types are at level ∞.
- Universality of consensus.
- 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]
  - [Attiya, Welch, Section 5.3.2]
  - [Attie, Guerraoui, Kouznetsov, Lynch]