6.852: Distributed Algorithms
Fall, 2015

Lecture 17

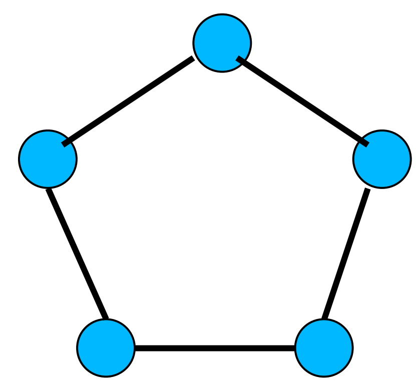
Today’s plan

- Dining Philosophers (briefly)
- Reading: Chapter 11

- Asynchronous shared-memory systems with failures.
- Consensus in asynchronous shared-memory systems.
- Impossibility of consensus [Fischer, Lynch, Paterson]
- Reading: Chapter 12

- Next:
  - Atomic objects
  - Reading: Chapter 13

Dining Philosophers



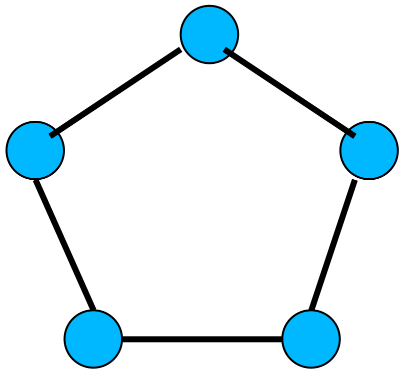
Generalized resource allocation

- Mutual exclusion: Problem of allocating a single non-sharable resource.
- Can generalize to more resources, some sharing.
- Exclusion specification $\mathcal{E}$ (for a given set of users):
  - Any collection of sets of users, closed under superset.
  - Expresses which users are incompatible, can't coexist in the critical section.

- Example: k-exclusion (any k users are OK, but not k+1)
  - $\mathcal{E} = \{ E : |E| > k \}$

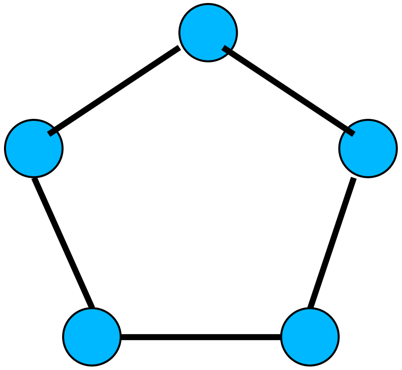
- Example: Reader-writer locks
  - Relies on classification of users as readers vs. writers.
  - $\mathcal{E} = \{ E : |E| > 1$ and E contains a writer $\}$

- Example: Dining Philosophers (Dijkstra)
  - $\mathcal{E} = \{ E : E$ includes a pair of neighbors $\}$



Resource specifications

- Some exclusion specs can be described conveniently in terms of requirements for concrete resources.
- Resource specification: Each user needs a certain particular subset of the resources.
- Can't share: Users with intersecting sets exclude each other.



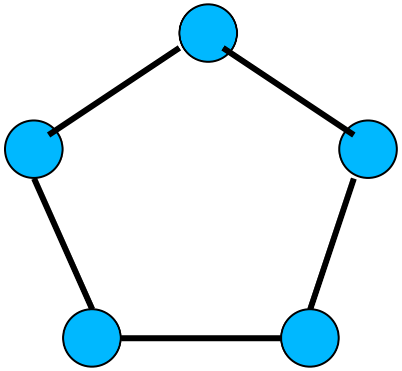
- Example: Dining Philosophers (Dijkstra)
  - $\mathcal{E}$ = { E : E includes a pair of neighbors }
  - Forks (resources) between adjacent philosophers; each needs both adjacent forks in order to eat.
  - Only one can hold a particular fork at a time, so adjacent philosophers must exclude each other.
- Not every exclusion problem can be expressed in this way.
  - k-exclusion cannot.

Resource allocation problem, for a given exclusion specification $\mathcal{E}$

- Same shared-memory architecture as for mutual exclusion (processes and shared variables).
- Well-formedness: As before.
- Exclusion: There is no reachable state in which the set of users in C is a set in $\mathcal{E}$.
- Progress: As before.
- Lockout-freedom: As before.
- But these don't capture concurrency requirements.
  - Any lockout-free mutual exclusion algorithm also satisfies all these conditions (provided that $\mathcal{E}$ doesn't contain any singleton sets).
- Can add concurrency conditions, e.g.:
  - Independent progress: If $i \in T$ and every j that could conflict with i remains in R, then eventually $i \to C$.
  - Time bound: Obtain better bounds from $i \to T$ to $i \to C$, even in the presence of conflicts, than we can for mutual exclusion.

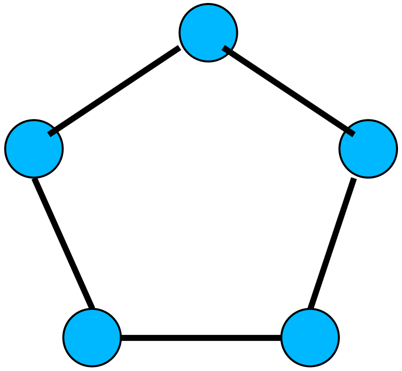
Dining Philosophers Problem

- Like Mutual Exclusion, with a different exclusion condition:
  - No two neighbors are in C at once (exclusion specification), or
  - Forks on edges, each philosopher needs both adjacent forks to eat (explicit resource specification).
- Can use progress and fairness conditions as for Mutex.
- Can add new conditions to capture concurrent access to C.
- Dijkstra posed the problem, gave a solution for a strong shared-memory model.
  - Globally-shared variables, atomic access to all of shared memory.
- Distributed version: Assume the only shared variables are RMW variables corresponding to the forks, accessible only by processes at the endpoints.



Impossibility Result

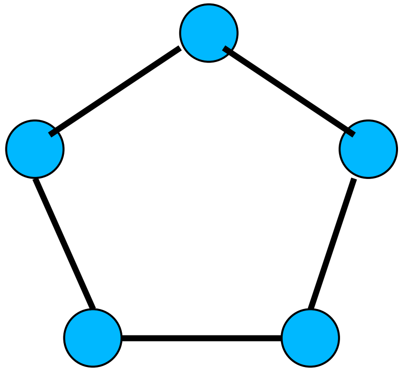
- Theorem 1: If all processes are identical and refer to forks by local names L and R, and all shared variables have the same initial value, then we can't guarantee DP exclusion + progress.
- Proof: Can't break symmetry:
  - Consider only executions that work in synchronous rounds, prove by induction on rounds that symmetry is preserved.
  - Assume all processes $\rightarrow$ T.
  - By progress, someone $\rightarrow$ C.
  - By symmetry, all do, violating DP exclusion.
- Example symmetric algorithm: Wait for R fork first, then L fork.
  - Guarantees DP exclusion.
  - Progress fails---all processes might get R fork, then wait forever for L fork (deadlock).



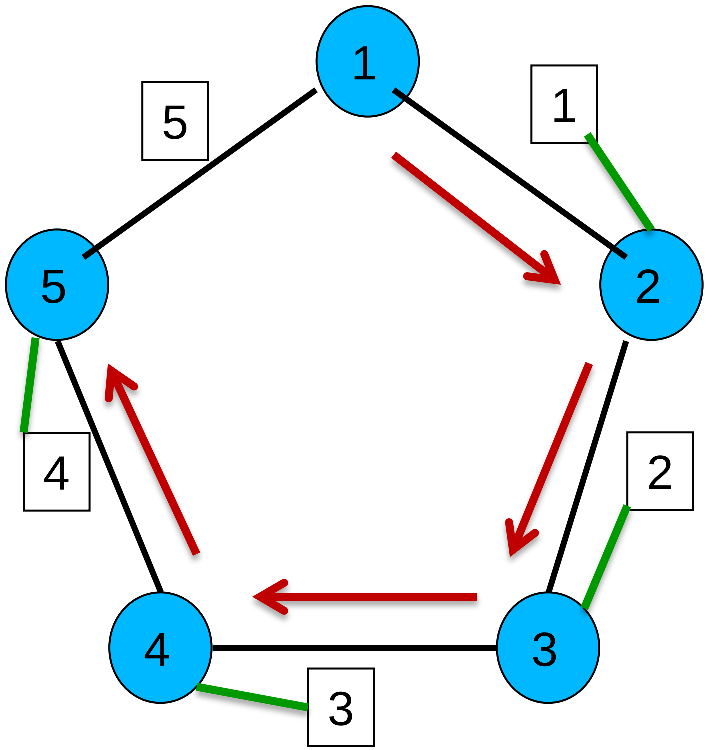
DP Algorithms

- So we need a mechanism to break symmetry.

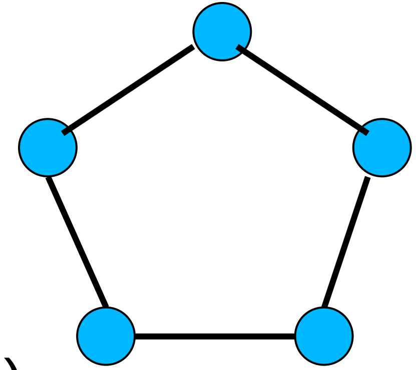
- Solution 1: Number forks in increasing order around the table; every process picks up its smaller numbered fork first.
  - Yields DP exclusion, progress, lockout freedom, independent progress.
  - But the time isn't good---we can have a long "waiting chain" of processes waiting for neighbors to release forks.



Creating a long waiting chain



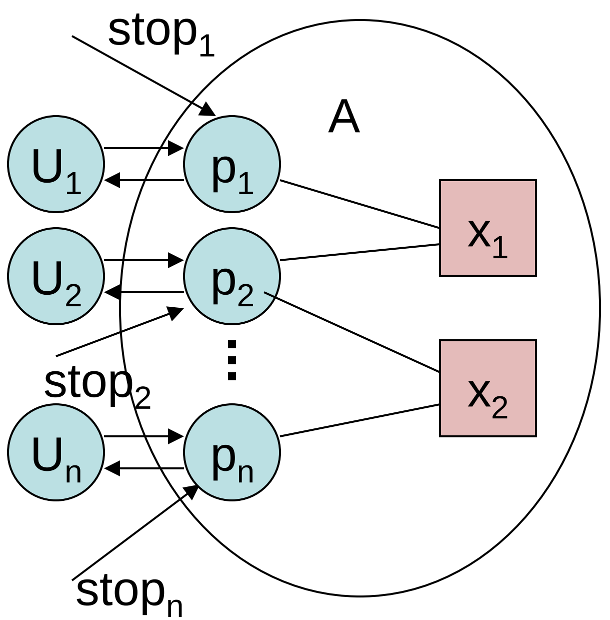
DP Algorithms



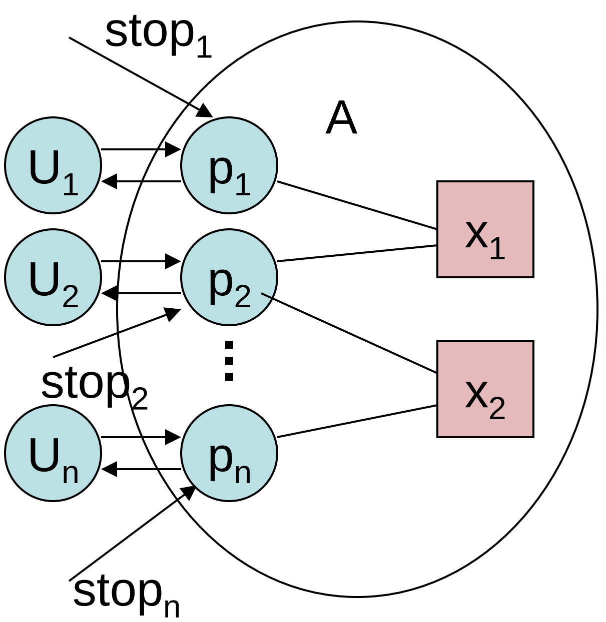
- Solution 2: Right/Left algorithm (Burns):
  - Classify processes as R or L (at least one of each).
  - R processes pick up right fork first, L processes pick up left fork first.
  - Yields DP exclusion, progress, lockout freedom, independent progress.
  - In even-sized rings in which R and L alternate, the lengths of waiting chains are limited to 2.
  - Yields a good (constant) time bound, LTTR.

- Generalize to solve any resource problem:
  - Represent the problem as an undirected graph.
  - Nodes = resources.
  - Edge between two resources if some user wants both.
  - Color the nodes of the graph; order colors.
  - All processes acquire resources in order of colors.

Asynchronous shared-memory systems with failures



Asynchronous shared-memory systems with failures



- Process stopping failures.
- Architecture as for Mutual Exclusion.
  - Processes + shared variables, one system automaton.
  - Users
- Add $stop_i$ inputs.
  - Effect is to disable all future locally controlled actions of process i.
- Fair executions:
  - Every process that doesn't fail gets infinitely many turns to perform locally-controlled steps.
  - Just ordinary fairness---stop means that nothing further is enabled.
  - Users also get turns.

The consensus problem in asynchronous shared-memory systems with failures

Consensus in Asynchronous Shared-Memory Systems

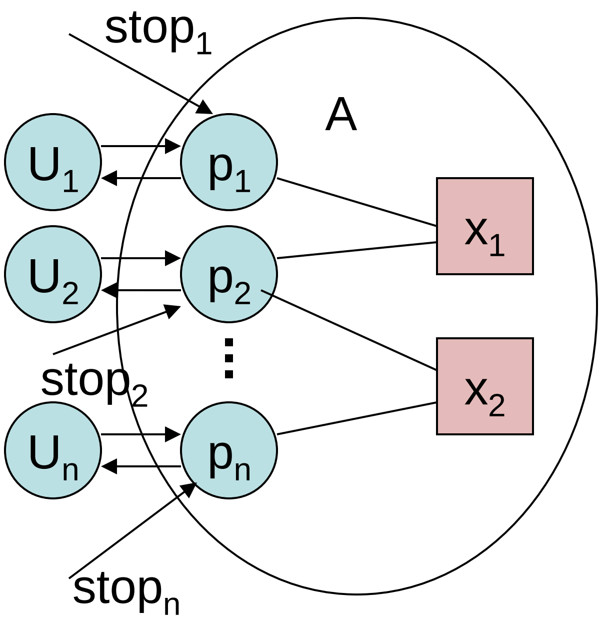
- Recall: Consensus in synchronous networks.
  - Algorithms for stopping failures:
    - FloodSet, FloodMin, Optimizations: f+1 rounds, any number of processes, low communication
  - Lower bounds: f+1 rounds
  - Algorithms for Byzantine failures
    - EIG: f+1 rounds, n > 3f, exponential communication
  - Lower bounds: f+1 rounds, n > 3f
- Asynchronous networks: Impossible
- Asynchronous shared memory:
  - Read/write variables: Impossible
  - Read-modify-write variables: Simple algorithms exist
- Impossibility results hold even if n is large and f is just 1!

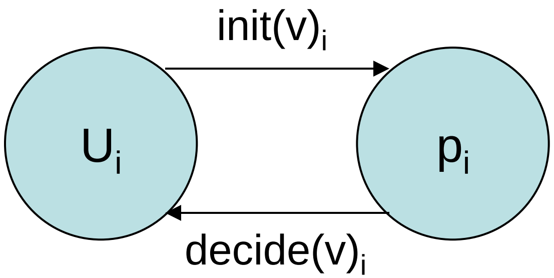
Consequences of impossibility results

- Can’t solve problems like transaction commit, agreement on choice of leader, fault diagnosis,…in the purely asynchronous model with failures.
- But these problems must be solved…what to do?
- Can strengthen the assumptions:
  - Rely on timing assumptions: Upper and lower bounds on message delivery time, on step time.
  - Probabilistic assumptions
- And/or weaken the guarantees:
  - Allow a small probability of violating safety properties, or of not terminating.
  - Conditional termination, based on stability for a “sufficiently long” interval of time.
- We’ll see some of these strategies.
- But, first, the impossibility result!

Architecture

- V, set of consensus values
- Interaction between user U_i and process (agent) p_i:
  - User U_i submits initial value v with $init(v)_i$.
  - Process p_i returns decision v with $decide(v)_i$.
  - I/O handled slightly differently from synchronous setting, where inputs and outputs were in local variables.
  - Assume each user performs at most one $init(v)_i$ in an execution.
- Shared variable types:
  - Read/write registers (for now)





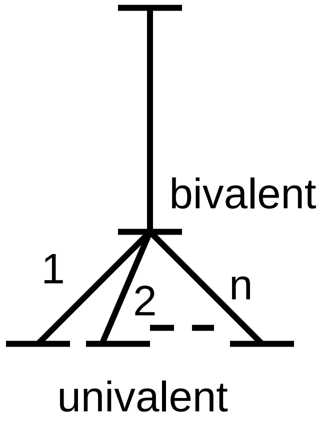
Requirements 1

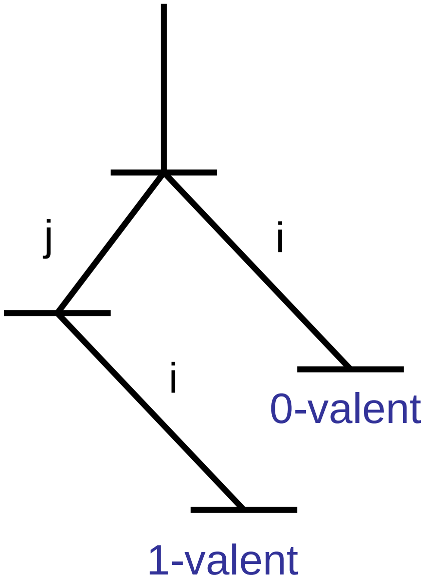
- Well-formedness:
  - At most one $\text{decide}()_i$, appears, and only after an $\text{init}()_i$.
- Agreement:
  - All decision values are identical.
- Validity:
  - If all init actions that occur contain the same v, then that v is the only possible decision value.
  - Stronger version: Any decision value is someone's initial value.
- Termination:
  - Failure-free termination (most basic requirement):
  - In any fair failure-free (ff) execution in which init events occur on all "ports", decide events occur on all ports.
- Basic problem requirements: Well-formedness, agreement, validity, failure-free termination.

Requirements 2: Fault-tolerance

- Wait-free termination (strongest condition):
  - In any fair execution in which init events occur on all ports, a decide event occurs on every port i for which no $stop_i$ occurs.
  - Similar to wait-free doorway in Lamport's Bakery algorithm: says i finishes regardless of whether the other processes stop or not.
- Also consider tolerating a limited number of failures.
- Should be easier to achieve, so impossibility results are stronger.
- f-failure termination, $0 \leq f \leq n$:
  - In any fair execution in which init events occur on all ports, if there are stop events on at most f ports, then a decide event occurs on every port i for which no $stop_i$ occurs.
- Wait-free termination = n-failure termination = (n-1)-failure termination.
- 1-failure termination: The interesting special case we will consider in our main proof.

Impossibility results for consensus in asynchronous shared-memory systems with failures





Impossibility of agreement

- Main Theorem [Fischer, Lynch, Paterson], [Loui, Abu-Amara]:
  - For $n \geq 2$, there is no algorithm in the read/write shared memory model that solves the agreement problem and guarantees 1-failure termination.
- A Weaker Theorem [Herlihy]:
  - For $n \geq 2$, there is no algorithm in the read/write shared memory model that solves the agreement problem and guarantees wait-free termination.

- We'll prove Herlihy's theorem first.

Restrictions (WLOG)

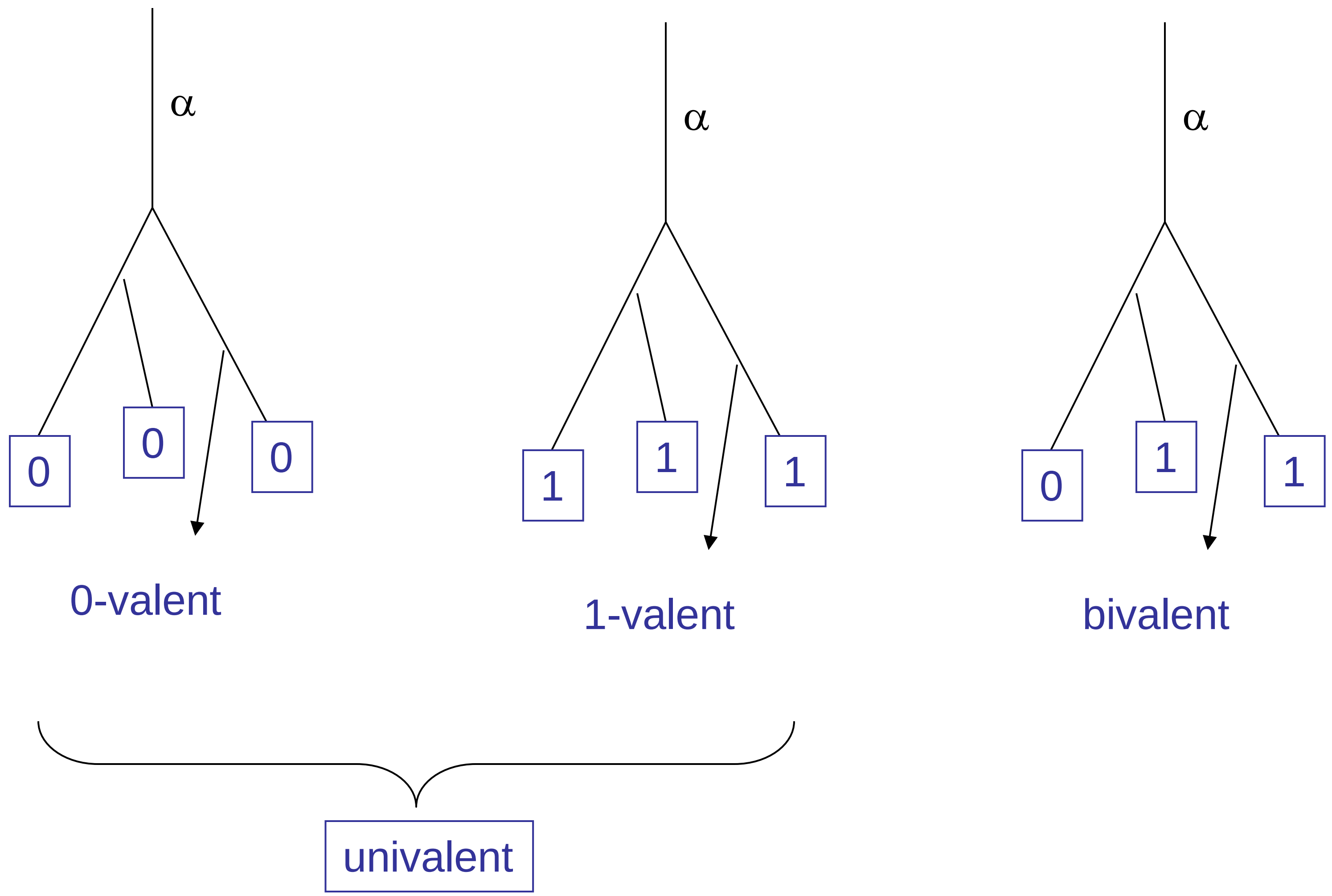
- $V = \{ 0, 1 \}$
- One task per process
- Processes are deterministic:
  - Unique start state.
  - From any state, any process has at most one locally-controlled action enabled.
  - From any state, for any enabled action, there is exactly one new state.
- Non-halting:
  - Every non-failed process always has some locally-controlled action enabled, even after it decides.

Terminology

- Initialization:
  - Sequence of n init steps, one per port, in index order: $init(v_1)_1$, $init(v_2)_2,\ldots init(v_n)_n$
- Input-first execution:
  - Begins with an initialization.

- A finite execution α is:
  - 0-valent, if 0 is the only decision value appearing in α or any extension of α, and 0 actually does appear in α or some extension.
  - 1-valent, if 1 is the only decision value appearing in α or any extension of α, and 1 actually does appear in α or some extension.
  - Univalent, if α is 0-valent or 1-valent.
  - Bivalent, if each of 0, 1 occurs in some extension of α.

Univalence and Bivalence



Exhaustive classification

- Lemma 1:
  - If A solves agreement with ff-termination, then each finite ff execution of A is either univalent or bivalent.
- Proof:
  - Can extend to a fair execution, in which everyone is required to decide.

Bivalent Initializations

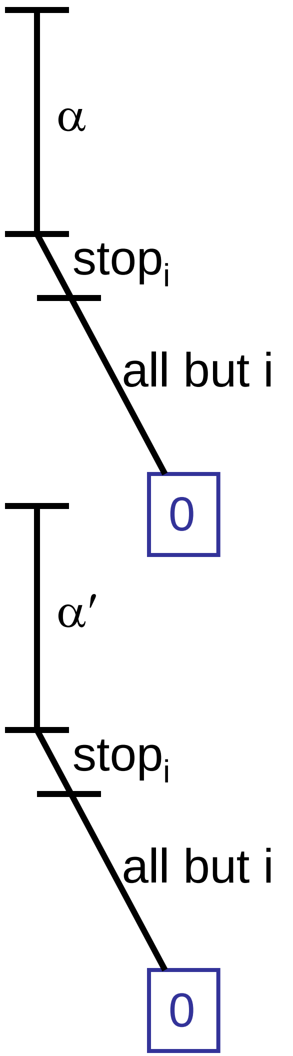
Bivalent initializations

- From now on, fix A to be an algorithm solving agreement with (at least) 1-failure termination.
  - Could also satisfy stronger conditions, like f-failure termination, or wait-free termination.

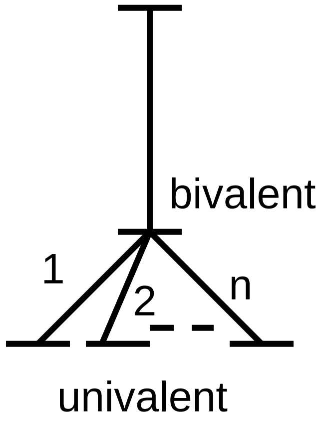
- Lemma 2: A has a bivalent initialization.
- That is, the final decision value cannot always be determined from the inputs only.
- Contrast: In non-fault-tolerant case, final decision can be determined from the inputs only; e.g., take majority.
- Proof:
  - Same argument used (later) by [Aguilera, Toueg].
  - Suppose not. Then all initializations are univalent.
  - Define initializations α_0 = all 0s, α_1 = all 1s.
  - α_0 is 0-valent, α_1 is 1-valent, by validity.

Bivalent initializations

- A solves agreement with 1-failure termination.
- Lemma 2: A has a bivalent initialization.
- Proof, cont'd:
  - Construct chain of initializations, spanning from α_0 to α_1, each differing in the initial value of just one process.
  - There must be two consecutive initializations, say α and α', where α is 0-valent and α' is 1-valent.
  - Differ only in initial value of some process i.
  - Consider a fair execution extending α, in which i fails right after α.
  - All but i must eventually decide, by 1-failure termination; since α is 0-valent, all must decide 0.
  - Extend α' in the same way, all but i still decide 0, by indistinguishability.
  - Contradicts 1-valence of α'.



Impossibility for Wait-Free Termination

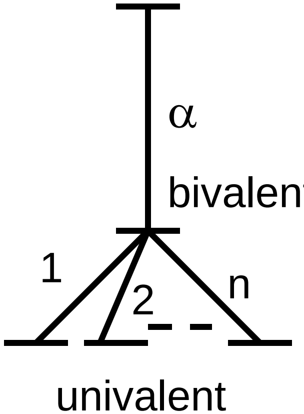


Impossibility for wait-free termination

- Weaker Theorem [Herlihy]:
  - For $n \geq 2$, there is no algorithm in the read/write shared memory model that solves the agreement problem and guarantees wait-free termination.
- Proof:
  - We already assumed A solves agreement with 1-failure termination (which yielded a bivalent initialization).
  - Now assume, for contradiction, that A (also) satisfies the stronger wait-free termination condition.
  - Proof is based on pinpointing exactly how a decision gets determined, that is, how the execution moves from bivalence to univalence.

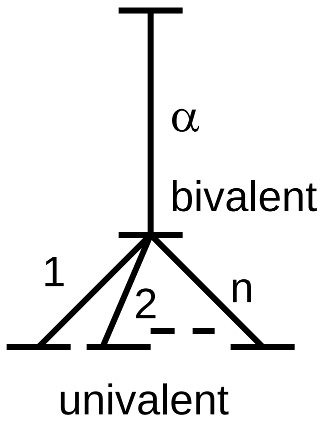
Impossibility for wait-free termination

- Definition: A decider execution α is a finite, failure-free, input-first execution such that:
  - α is bivalent.
  - For every i, $\underbrace{\text{ext}(\alpha,i)}_{\text{Extension of } \alpha \text{ with one step of } i}$ is univalent.



- Lemma 3: A (with wait-free termination) has a decider execution.

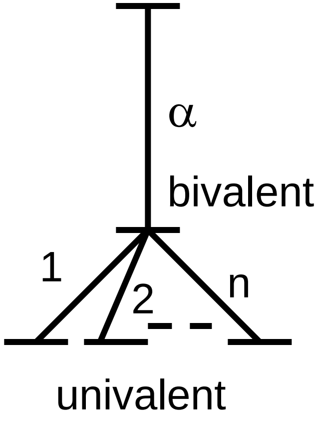
Impossibility for wait-free termination

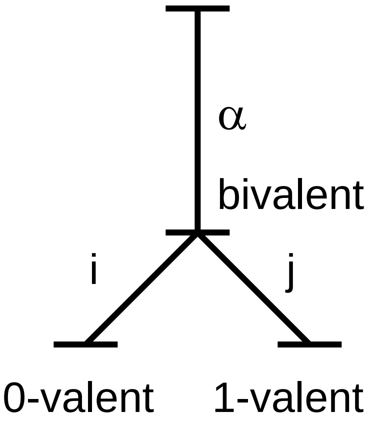


- Lemma 3: A (with w-f termination) has a decider.
- Proof:
  - Suppose not. Then any bivalent ff input-first execution has a 1-step bivalent ff extension.
  - Start with a bivalent initialization (Lemma 2), and produce an infinite ff execution α all of whose prefixes are bivalent.
    - At each stage, start with a bivalent ff input-first execution and extend by one step to another bivalent ff execution.
    - Possible by assumption.
  - α must contain infinitely many steps of some process, say i.
  - Claim i must decide in α:
    - Add stop events for all processes that take only finitely many steps.
    - Result is a fair execution α'.
    - Wait-free termination says i must decide in α'.
    - α is indistinguishable from α', by i, so i must decide in α also.
  - Contradicts bivalence.

Impossibility for wait-free termination

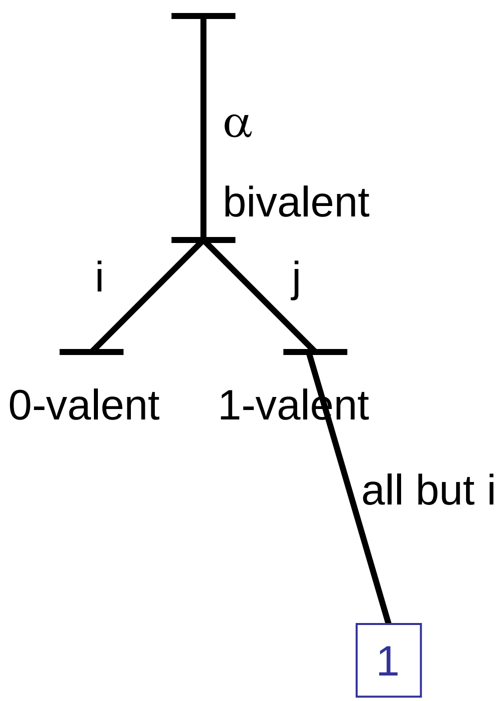
- Proof of theorem, cont'd:
  - Fix a decider, α.
  - Since α is bivalent and all 1-step extensions are univalent, there must be two processes, say i and j, leading to 0-valent and 1-valent states, respectively.
  - Case analysis yields a contradiction:
    1. i's step is a read
    2. j's step is a read
    3. Both writes, to different variables.
    4. Both writes, to the same variable.

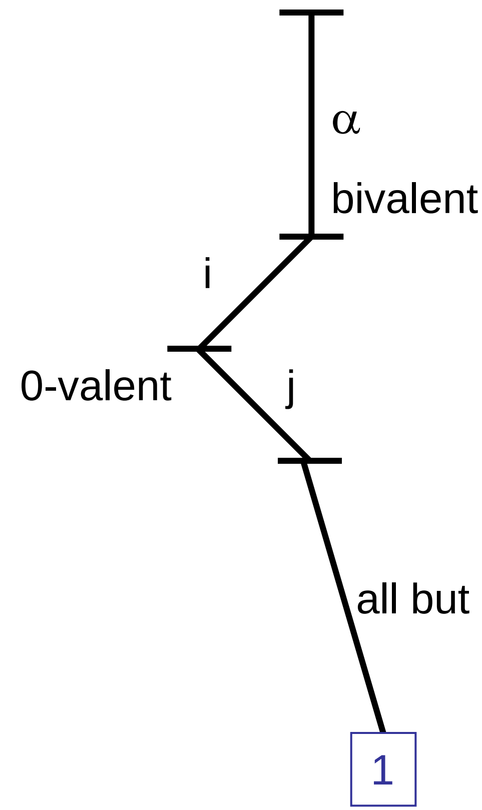




Case 1: i's step is a read

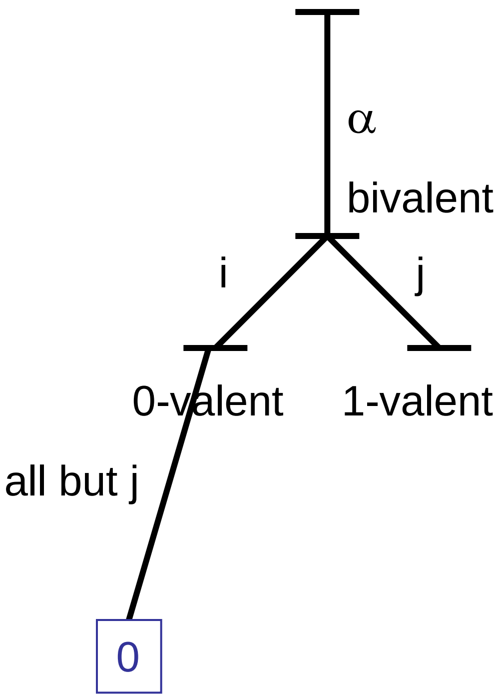
- Run all but i after ext(α,j).
- Looks like a fair execution in which i fails.
- So all others must decide; since ext(α,j), is 1-valent, they decide 1.
- Now run the same extension, starting with j's step, after ext(α,i).
- They cannot see i's read.
- So they behave the same, decide 1.
- Contradicts 0-valence of ext(α,i).

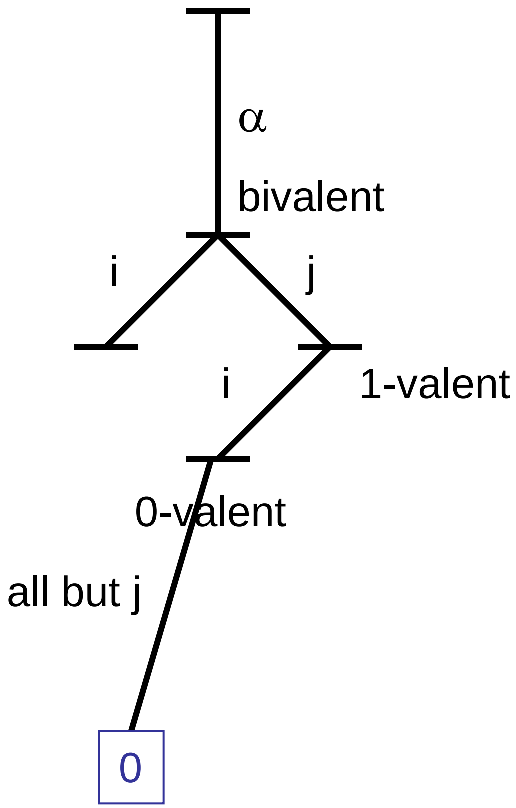




Case 2: j's step is a read

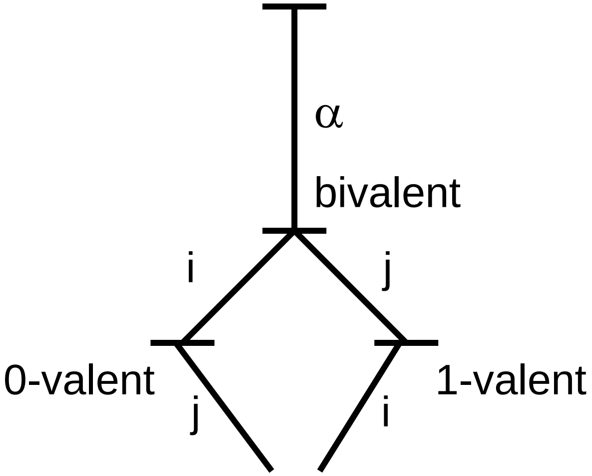
- Symmetric.





Case 3: Writes to different shared variables

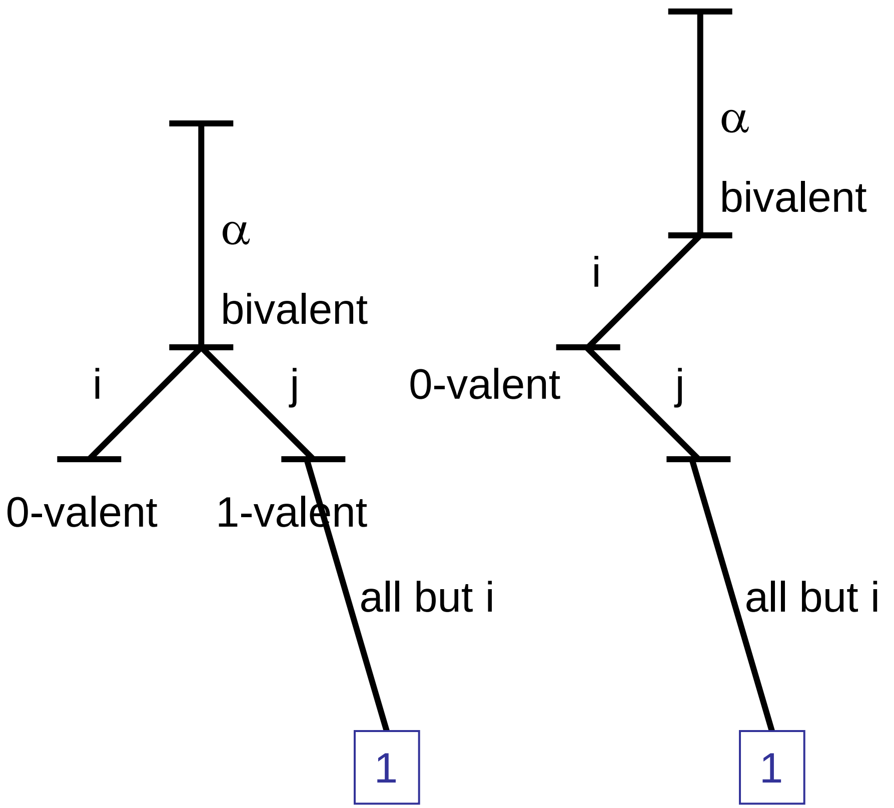
- Then the two steps are completely independent.
- Could be performed in either order, and the result should be the same.
- ext(α,ij) and ext(α,ji) are indistinguishable to all processes, and end up in the same system state.



- But ext(α,ij) is 0-valent, since it extends the 0-valent execution ext(α,i) .
- And ext(α,ji) is 1-valent, since it extends the 1-valent execution ext(α,j) .
- Contradictory requirements.

Case 4: Writes to the same shared variable x.

- Run all but i after ext(α,j); they must decide.
- Since ext(α,j), is 1-valent, they decide 1.
- Run the same extension, starting with j's step, after ext(α,i).
- Cannot see i's write to x, because j's write overwrites it.
- They behave the same, decide 1.
- Contradicts 0-valence of ext(α,i).

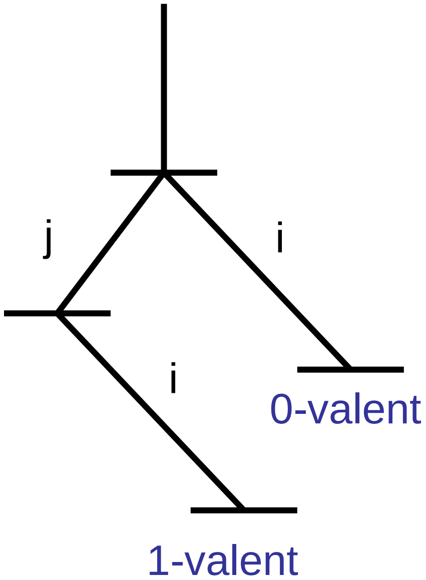


Impossibility for wait-free termination

- So we have proved:

- Weaker Theorem: [Herlihy]
  - For $n \geq 2$, there is no algorithm in the read/write shared memory model that solves the agreement problem and guarantees wait-free termination.

Impossibility for 1-failure termination



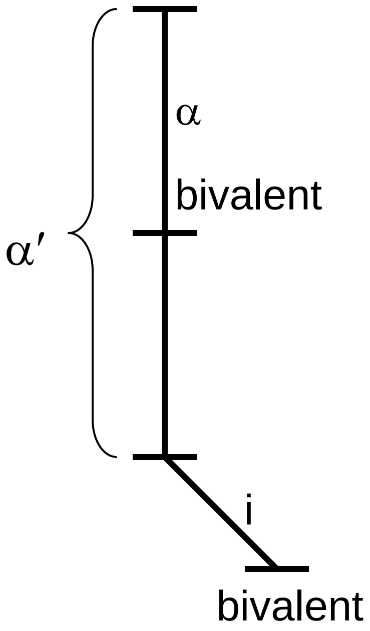
Impossibility for 1-failure temination

- Q: Why doesn’t the previous proof yield impossibility for 1-failure termination?
- Lemma 2 (bivalent initialization) works for f = 1.
- In the proof of Lemma 3 (existence of decider), wait-free termination is used to say that a process i must decide in any fair execution in which i doesn’t fail.
- 1-failure termination makes a termination guarantee only when at most one process fails.

- Main Theorem:
  – For $n \geq 2$, there is no algorithm in the read/write shared memory model that solves the agreement problem and guarantees 1-failure termination.

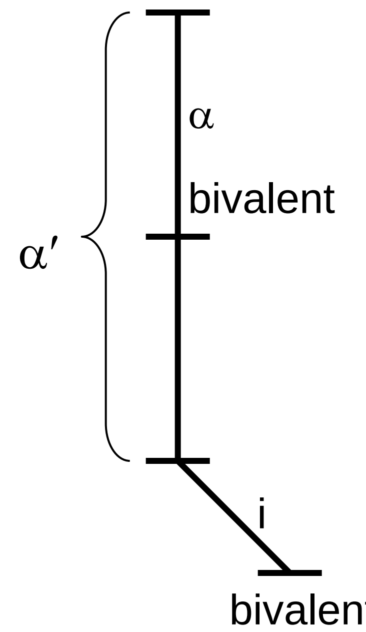
Impossibility for 1-failure temination

- From now on, assume A satisfies 1-failure termination, not necessarily wait-free termination (weaker requirement).
- Initialization lemma still works:
  - Lemma 2: A has a bivalent initialization.
- New key lemma, replacing Lemma 3:
- Lemma 4: If α is any bivalent, ff, input-first execution of A, and i is any process, then there is some ff-extension α' of α such that ext(α',i) is bivalent.



Lemma 4 $\Rightarrow$ Main Theorem

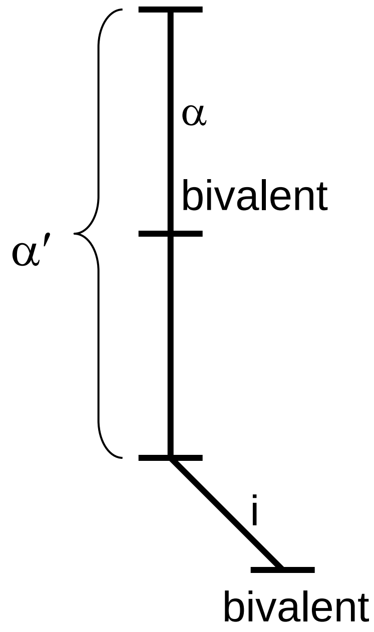
- Lemma 4: If α is any bivalent, ff, input-first execution of A, and i is any process, then there is some ff-extension α' of α such that ext(α',i) is bivalent.
- Proof of Main Theorem using Lemma 4:
  - Construct a fair, ff, input-first execution in which no process ever decides, contradicting the basic ff-termination requirement.
  - Start with a bivalent initialization.
  - Then cycle through the processes round-robin: 1, 2, …, n, 1, 2, …
  - At each step, say for i, use Lemma 4 to extend the execution, including at least one step of i, while maintaining bivalence and avoiding failures.
  - Contradiction.

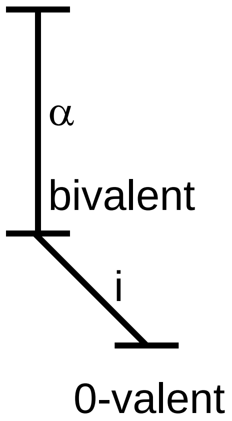


Proof of Lemma 4

- Lemma 4: If α is any bivalent, ff, input-first execution of A, and i is any process, then there is some ff-extension α' of α such that ext(α',i) is bivalent.

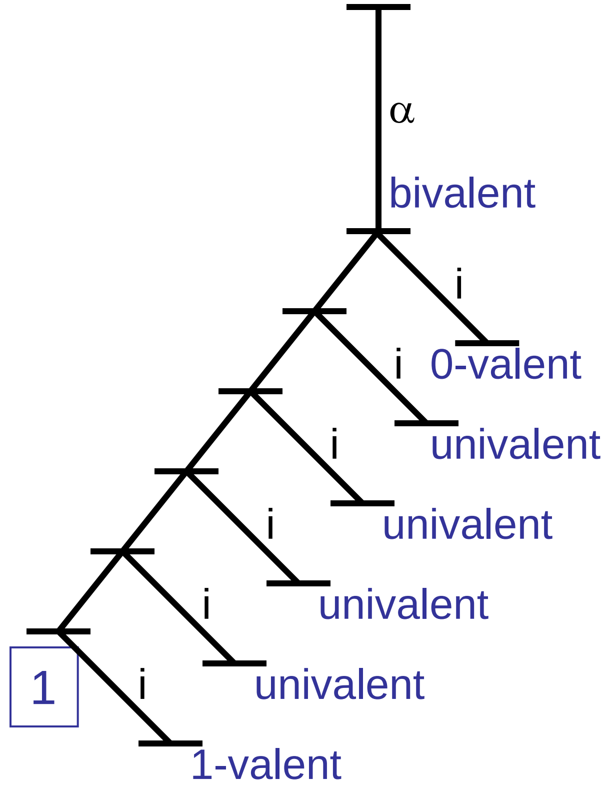
- Proof:
  - By contradiction. Suppose there is some bivalent, ff, input-first execution α of A and some process i, such that for every ff extension α' of α, ext(α',i) is univalent.
  - In particular, ext(α,i) is univalent, WLOG 0-valent.
  - Since α is bivalent, there is some extension of α in which someone decides 1, WLOG failure-free.

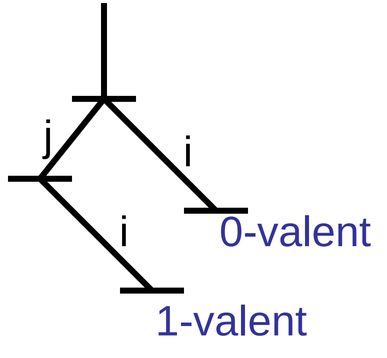




Proof of Lemma 4

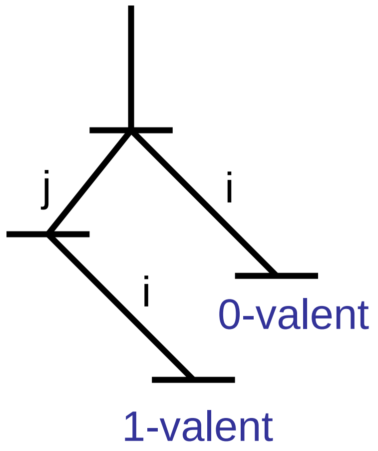
- There is some ff-extension of α in which someone decides 1.
- Consider letting i take one step at each point along the "spine".
- By assumption, results are all univalent.
- 0-valent at the beginning, 1-valent at the end.
- So there are two consecutive results, one 0-valent and the other 1-valent:
- A new kind of "decider".





New "Decider"

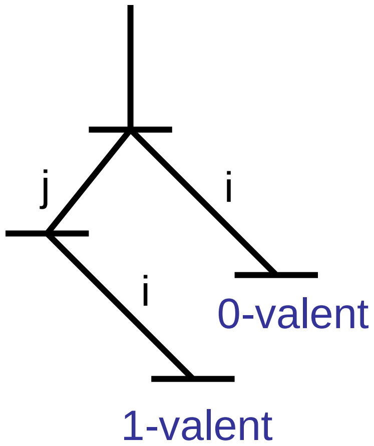
- Claim: $j \neq i$.
- Proof:
  - If $j = i$ then:
    - 1 step of i yields 0-valence
    - 2 steps of i yield 1-valence
  - But process i is deterministic, so this can't happen.
    - "Child" of a 0-valent state can't be 1-valent.

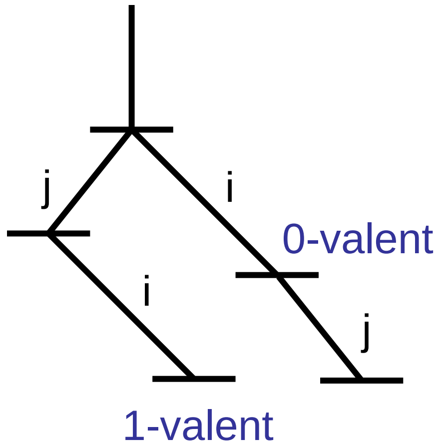


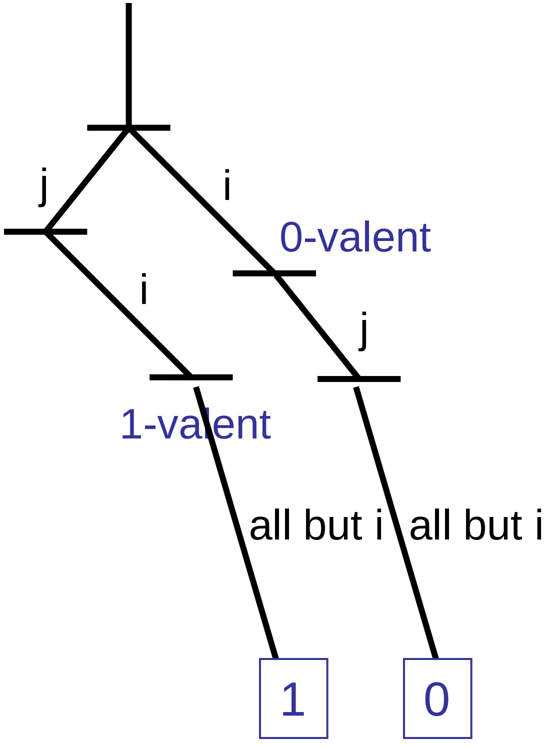
- The rest of the proof is a case analysis, similar to before…

Case 1: i's step is a read

- Run j after i.
- Executions ending with ji and ij are indistinguishable to everyone but i (because this is a read step of i).
- Run all processes except i in the same order after ji and ij.
- In each case, they must decide, by 1-failure termination.
- After ji, they must decide 1.
- After ij, they must decide 0.
- But indistinguishable, contradiction!

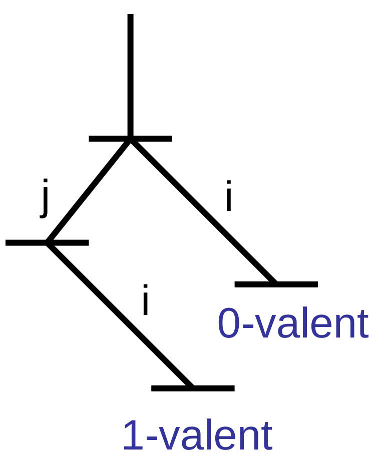


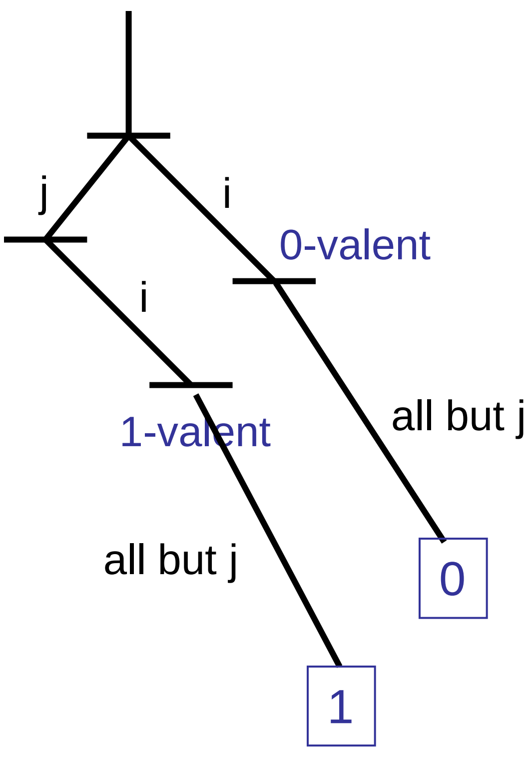




Case 2: j's step is a read

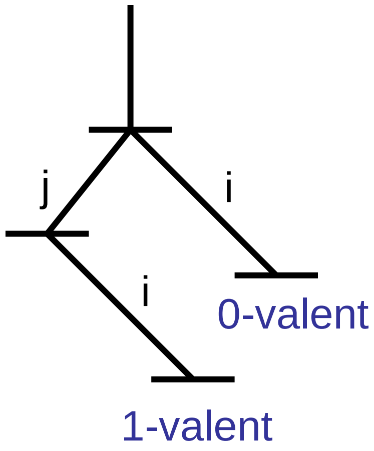
- Executions ending with ji and i are indistinguishable to everyone but j (because this is a read step of j).
- Run all processes except j in the same order after ji and i.
- In each case, they must decide, by 1-failure termination.
- After ji, they must decide 1.
- After i, they must decide 0.
- But indistinguishable, contradiction!

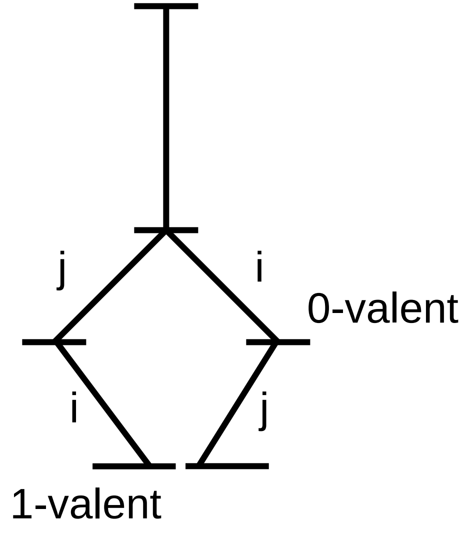




Case 3: Writes to different shared variables

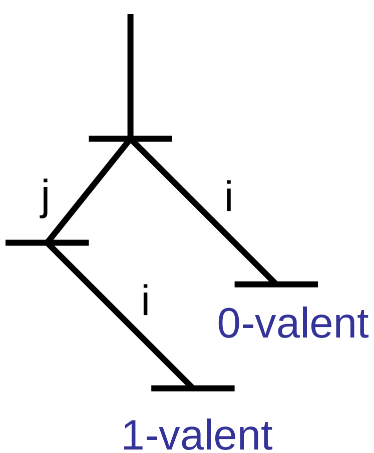
- As for the wait-free case.
- The steps of i and j are independent, could be performed in either order, indistinguishable to everyone.
- But the execution ending with ji is 1-valent, whereas the execution ending with ij is 0-valent.
- Contradiction.

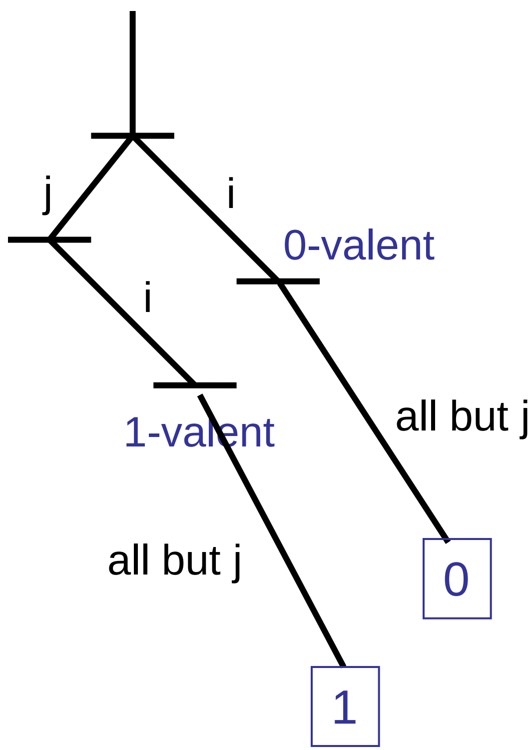




Case 4: Writes to the same shared variable x.

- As for Case 2.
- Executions ending with ji and i are indistinguishable to everyone but j (because i overwrites the write step of j).
- Run all processes except j in the same order after ji and i.
- After ji, they must decide 1.
- After i, they must decide 0.
- Indistinguishable, contradiction!





Impossibility for 1-failure termination

- So we have proved:

- Main Theorem: [Fischer, Lynch, Paterson] [Loui, Abu-Amara]
  - For $n \geq 2$, there is no algorithm in the read/write shared memory model that solves the agreement problem and guarantees 1-failure termination.

Variations, extensions, significance,…

Extension to networks

- Result also holds in asynchronous networks---see this soon.
- [Fischer, Lynch, Paterson 82, 85] proved first for networks; 2001 Dijkstra Prize.
- [Loui, Abu-Amara 87] extended result and proof to shared memory.

Significance of FLP impossibility result

- For distributed computing practice:
  - Reaching agreement is important in practice:
    - Agreeing on aircraft altimeter readings.
    - Database transaction commit.
    - Agreeing on updates to data replicas.
    - Agreeing on BitCoin transaction logs.
  - FLP shows limitations on the kind of algorithm one can look for.
- For distributed computing theory:
  - Variations:
    - [Loui, Abu-Amara 87] Read/write shared memory.
    - [Herlihy 91] Stronger fault-tolerance requirement (wait-free termination); simpler proof.
  - Circumventing the impossibility result:
    - Strengthening the assumptions.
    - Weakening the guarantees.

Strengthening the assumptions

- Using limited timing information [Dolev, Dwork, Stockmeyer 87].
  - Bounds on message delays, processor step time.
  - Makes the model more like the synchronous model.
- Using randomness [Ben-Or 83][Rabin 83] [Attiya, Censor].
  - Allow random choices in local transitions.
  - Also weakens guarantees:
    - Small probability of a wrong decision, or
    - Small probability of not terminating, in any bounded time (Probability of not terminating approaches 0 as time approaches infinity.)

Weakening the guarantees

- Agreement, validity must always hold.
- Termination guaranteed if system behavior “stabilizes”:
  - No new failures.
  - Timing (of process steps, messages) within “normal” bounds.
- Good solutions have been developed, both theoretical and practical.
- [Dwork, Lynch, Stockmeyer 88] Dijkstra Prize, 2007
  - Keeps trying to choose a leader, who tries to coordinate agreement.
  - Coordination attempts can fail.
  - Once system stabilizes, a unique leader is chosen, coordinates agreement.
  - Tricky part: Ensuring failed attempts don’t lead to inconsistent decisions.
- [Lamport 89] Paxos algorithm.
  - Improves on [DLS] by allowing more concurrency.
  - Refined, engineered for practical use.

Weakening the guarantees

- Agreement, validity must always hold.
- Termination required if system behavior "stabilizes":
  - No new failures.
  - Timing (of process steps, messages) within "normal" bounds.
- Good solutions, both theoretically and in practice.
- [Dwork, Lynch, Stockmeyer 88]: Dijkstra Prize, 2007
- [Lamport 89] Paxos algorithm.
- [Chandra, Hadzilacos, Toueg 96] Failure detectors (FDs)
  - Services that encapsulate use of time for detecting failures.
  - Develop similar algorithms to [DLS 88] and [Lamport 89] using FDs.
  - Studied properties of FDs, identified weakest FD to solve consensus.

Extension to k-consensus

- At most k different decisions may occur overall.
- Solvable for k-1 process failures but not for k failures.
  - Algorithm for k-1 failures: [Chaudhuri 93].
  - Impossibility result:
    - [Herlihy, Shavit 93], [Borowsky, Gafni 93], [Saks, Zaharoglu 93]
    - Godel Prize, 2004.
    - Techniques from algebraic topology: Sperner's Lemma.
    - Similar to those used for lower bound on rounds for k-agreement, in synchronous model.

- Question (recent results):
  - What is the weakest failure detector to solve k-consensus with k failures?

Importance of the read/write data type

- Consensus impossibility result doesn't hold for more powerful data types.
- Example: Read-modify-write shared memory
  - Very strong primitive.
  - In one step, can read variable, do local computation, and write back a value.
  - Easy algorithm:
    - One shared variable x, value in $V \cup \{\perp\}$, initially $\perp$.
    - Each process i accesses x once.
    - If it sees:
      - $\perp$, then it changes the value in x to its own initial value and decides on that value.
      - Some v in V, then it decides on that value.
- Read/write registers are similar to asynchronous FIFO reliable channels---we'll see the precise connection later.

Next time…

- Atomic objects
- Reading: Chapter 13