Class 16
Today’s plan

- Generalized resource allocation
- Asynchronous shared-memory systems with failures.
- Consensus in asynchronous shared-memory systems.
- Impossibility of consensus [Fischer, Lynch, Paterson]
- Reading: Chapter 11, Chapter 12
- Next: Chapter 13
Generalized resource allocation

- Mutual exclusion: Problem of allocating a single non-sharable resource.
- Can generalize to more resources, some sharing.

- Exclusion specification $E$ (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)
  $$E = \{ E : |E| > k \}$$

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

- Example: Dining Philosophers [Dijkstra]
  $$E = \{ E : E \text{ includes a pair of neighbors} \}$$
Resource specifications

• Some exclusion specs can be described conveniently in terms of requirements for concrete resources.

• **Resource specification:** Different users need different subsets of resources
  – Can't share: Users with intersecting sets exclude each other.

• **Example:** Dining Philosophers
  \[ E = \{ E : E \text{ includes a pair of neighbors} \} \]
  – 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.
  – E.g., k-exclusion cannot.
Resource allocation problem, for a given exclusion spec \( \mathbf{E} \)

- Same shared-memory architecture as for mutual exclusion (processes and shared variables, no buses, no caches).
- **Well-formedness**: As before.
- **Exclusion**: No reachable state in which the set of users in \( C \) is a set in \( \mathbf{E} \).
- **Progress**: As before.
- **Lockout-freedom**: As before.
- But these don’t capture concurrency requirements.
- Any lockout-free mutual exclusion algorithm also satisfies \( \mathbf{E} \) (provided that \( \mathbf{E} \) 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 \( i \rightarrow C \).
  - **Time bound**: Obtain better bounds from \( i \rightarrow T \) to \( i \rightarrow C \), even in the presence of conflicts, than we get for mutual exclusion.
Dining Philosophers

• Dijkstra’s paper posed the problem, gave a solution using strong shared-memory model.
  – Globally-shared variables, atomic access to all of shared memory.
  – Not very distributed.

• More distributed version: Assume the only shared variables are on the edges between adjacent philosophers.
  – Correspond to forks.
  – Use RMW shared variables.

• **Impossibility result:** If all processes are identical and refer to forks by local names “left” and “right”, and all shared variables have the same initial values, then we can’t guarantee DP exclusion + progress.

• **Proof:** Show we can’t break symmetry:
  – Consider subset of executions that work in synchronous rounds, prove by induction on rounds that symmetry is preserved.
  – Then by progress, someone $\rightarrow$ C.
  – So all do, violating DP exclusion.
Dining Philosophers

- **Example:** Simple symmetric algorithm where all wait for R fork first, then L fork.
  - Guarantees DP exclusion, because processes wait for both forks.
  - But progress fails---all might get R, then deadlock.
- So we need something to break symmetry.
- **Solutions:**
  - Number forks around the table, pick up smaller numbered fork first.
  - Right/left algorithm (Burns):
    - Classify processes as R or L (need 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, and good time bound (constant, for alternating R and L).
- **Generalize to solve any resource problem**
  - Nodes represent resources.
  - Edge between resources if some user needs both.
  - Color 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 $\text{stop}_i$ inputs.
  - Effect is to disable all future non-input 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.
Consensus 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

• 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…
• 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 Uᵢ and process (agent) pᵢ:
  – User Uᵢ submits initial value v with init(v)ᵢ.
  – Process pᵢ returns decision v with decide(v)ᵢ.
  – I/O handled slightly differently from synchronous setting, where we assumed I and O in local variables.
  – Assume each user performs at most one init(v)ᵢ in an execution.

• Shared variable types:
  – Read/write registers (for now)
Problem requirements 1

• **Well-formedness:**
  – At most one $\text{decide}(*)_i$, appears, and only if there’s a previous $\text{init}(*)_i$.

• **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 an 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.
Problem requirements 2: Fault-tolerance

- **Failure-free termination:**
  - In any fair failure-free (ff) execution in which init events occur on all ports, decide events occur on all ports.

- **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 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 proof.
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.

- **Simpler 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.

- Let’s prove the simpler theorem first.
Restrictions (WLOG)

• $V = \{ 0, 1 \}$
• Algorithms are deterministic:
  – Unique start state.
  – From any state, any process has $\leq 1$ 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: $\text{init}(v_1)_1$, $\text{init}(v_2)_2, \ldots \text{init}(v_n)_n$

- **Input-first execution:**
  - Begins with an initialization.

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

0-valent

1-valent

bivalent

univalent
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 initialization

• 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 $\alpha_0 = \text{all } 0\text{s}, \alpha_1 = \text{all } 1\text{s}.$
  – $\alpha_0$ is 0-valent, $\alpha_1$ is 1-valent, by validity.
**Bivalent initialization**

- A solves agreement with 1-failure termination.
- **Lemma 2:** A has a bivalent initialization.
- **Proof, cont’d:**
  - Construct chain of initializations, spanning from $\alpha_0$ to $\alpha_1$, each differing in the initial value of just one process.
  - There must be 2 consecutive initializations, say $\alpha$ and $\alpha'$, where $\alpha$ is 0-valent and $\alpha'$ is 1-valent.
  - Differ in initial value of some process $i$.
  - Consider a fair execution extending $\alpha$, in which $i$ fails right after $\alpha$.
  - All but $i$ must eventually decide, by 1-failure termination; since $\alpha$ is 0-valent, all must decide 0.
  - Extend $\alpha'$ in the same way, all but $i$ still decide 0, by indistinguishability.
  - Contradicts 1-valence of $\alpha'$.
Impossibility for wait-free termination

- **Simpler 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.
  - Now assume, for contradiction, that $A$ (also) satisfies wait-free termination.
  - 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 $\alpha$ is a finite, failure-free, input-first execution such that:
  – $\alpha$ is bivalent.
  – For every $i$, $\text{ext}(\alpha, i)$ is univalent.

• **Lemma 3:** A (with wait-free termination) has a decider execution.
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 $\alpha$ 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.
- $\alpha$ must contain infinitely many steps of some process, say $i$.
- Claim $i$ must decide in $\alpha$:
  - Add stop events for all processes that take only finitely many steps.
  - Result is a fair execution $\alpha'$.
  - Wait-free termination says $i$ must decide in $\alpha'$.
  - $\alpha$ is indistinguishable from $\alpha'$, by $i$, so $i$ must decide in $\alpha$ also.
- Contradicts bivalence.
Impossibility for wait-free termination

• Proof of theorem, cont’d:
  – Fix a decider, $\alpha$.
  – Since $\alpha$ 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(\(\alpha, j\)).
- Looks like a fair execution in which i fails.
- So all others must decide; since ext(\(\alpha, j\)), is 1-valent, they decide 1.
- Now run the same extension, starting with j’s step, after ext(\(\alpha, i\)).
- They behave the same, decide 1.
  - Cannot see i’s read.
- Contradicts 0-valence of ext(\(\alpha, i\)).
Case 2: j’s step is a read

- Symmetric.
Case 3: Writes to different shared variables

- Then the two steps are completely independent.
- They could be performed in either order, and the result should be the same.
- \( \text{ext}(\alpha, ij) \) and \( \text{ext}(\alpha, ji) \) are indistinguishable to all processes, and end up in the same system state.
- But \( \text{ext}(\alpha, ij) \) is 0-valent, since it extends the 0-valent execution \( \text{ext}(\alpha, i) \).
- And \( \text{ext}(\alpha, ji) \) is 1-valent, since it extends the 1-valent execution \( \text{ext}(\alpha, j) \).
- Contradictory requirements.
Case 4: Writes to the same shared variable x.

- Run all but i after $\text{ext}(\alpha, j)$; they must decide.
- Since $\text{ext}(\alpha, j)$ is 1-valent, they decide 1.
- Run the same extension, starting with j’s step, after $\text{ext}(\alpha, i)$.
- They behave the same, decide 1.
  - Cannot see i’s write to x.
  - Because j’s write overwrites it.
- Contradicts 0-valence of $\text{ext}(\alpha, i)$.
Impossibility for wait-free termination

- So we have proved:

- Simpler 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

- **Why doesn’t the previous proof yield impossibility for 1-failure termination?**
- Lemma 2 (bivalent initialization) works for $f = 1$.
- In 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 termination

• 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 $\alpha$ is any bivalent, ff, input-first execution of A, and $i$ is any process, then there is some ff-extension $\alpha'$ of $\alpha$ such that $\text{ext}(\alpha',i)$ is bivalent.
Lemma 4 $\Rightarrow$ Main Theorem

- **Lemma 4:** If $\alpha$ is any bivalent, ff, input-first execution of A, and $i$ is any process, then there is some ff-extension $\alpha'$ of $\alpha$ such that $\text{ext}(\alpha', i)$ is bivalent.

- **Proof of Main Theorem:**
  - 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: $1, 2, \ldots, n, 1, 2, \ldots$
  - 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.
Proof of Lemma 4

- **Lemma 4**: If $\alpha$ is any bivalent, ff, input-first execution of A, and $i$ is any process, then there is some ff-extension $\alpha'$ of $\alpha$ such that $\text{ext}(\alpha',i)$ is bivalent.

- **Proof**:
  - By contradiction. Suppose there is some bivalent, ff, input-first execution $\alpha$ of A and some process $i$, such that for every ff extension $\alpha'$ of $\alpha$, $\text{ext}(\alpha',i)$ is univalent.
  - In particular, $\text{ext}(\alpha,i)$ is univalent, WLOG 0-valent.
  - Since $\alpha$ is bivalent, there is some extension of $\alpha$ in which someone decides 1, WLOG failure-free.
Proof of Lemma 4

– There is some ff-extension of $\alpha$ 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, as 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 both ji and ij.
- In each case, they must decide, by 1-failure termination.
- After ji, they decide 1.
- After ij, they 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 decide 1.
- After i, they 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 decide 1.
- After i, they 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.
Shared memory vs. networks

- Result also holds in asynchronous networks---revisit shortly.
- [Loui, Abu-Amara 87] extended result and proof to shared memory.
Significance of FLP impossibility result

• For distributed computing practice:
  – Reaching agreement is sometimes important in practice:
    • Agreeing on aircraft altimeter readings.
    • Database transaction commit.
  – 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 requirements/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].
  - Allow random choices in local transitions.
  - 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 requirements

• **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
  – Keeps trying to choose a leader, who tries to coordinate agreement.
  – Coordination attempts can fail.
  – Once system stabilizes, 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.
• [Chandra, Hadzilacos, Toueg 96] Failure detectors (FDs)
  – Services that encapsulate use of time for detecting failures.
  – Develop similar algorithms 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.

• Open question (currently active):
  – What is the weakest failure detector to solve k-consensus with k failures?
Importance of 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 \( V \cup \{\bot\} \), initially \( \bot \).
    - Each process \( i \) accesses \( x \) once.
    - If it sees:
      - \( \bot \), then it changes the value in \( x \) to its own initial value and decides on that value.
      - Some \( v \) in \( V \), then 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