Goals:

• Describe how to refine specifications for complex problems
• Work through design and proof of dining philosophers (as example)

Reading:

• P. Sivilotti, *Introduction to Distributed Algorithms*, Chapter 8
Dining Philosophers on a Graph

Classic formulation of “dining philosophers”
- N philosophers, N forks, shared pairwise
- Philosophers (agents) are in one of three states
  - hungry (TRY)
  - thinking (NC)
  - eating (CS) - requires two forks
- Similar to mutual exclusion, with multiple resources available (and required) to complete operation (eat)

Generalization to a graph
- Consider an arbitrary (but finite) undirected graph.
- Vertices = agents, edges define neighbors (assume connected, no self-loops).
- Edges = shared resource. Agent requires all shared resources (edges) to “eat”
- Classic formulation is special case with ______________________________
- Mutual exclusion is a special case with ______________________________

Other variants
- “Drinking” philosophers: allow a pool of resources (drinks), with each philosopher needing some resource from the pool, possibly with preferences
Program Specification (Dining Philosophers)

User process specification
- udn1: u.t unless u.h in user
- udn2: stable(u.h) in user
- udn3: u.e unless u.t in user
- udn4: (∀u,v : E(u,v) : ¬(u.e ∧ v.e)) ⇒ (∀u :: u.e ∼ ¬u.e))

Specification of composite program
- dn1: (safety): invariant (¬(u.e ∧ v.e ∧ E(u,v))) in user | os
- dn2: (progress): u.h ∼ u.e in user | os

Constraints on conflict resolution layer (os)
- odn1: invariant(u.t) in os
- odn2: stable(u.e) in os
- Derived properties of os
  - stable(¬u.h) in os
  - u.h unless u.e in os

Given these specs, how do we proceed?
- Need to define a “program” that implements the “os” function in a distributed fashion
- OK to assume listed properties about agents
- Approach: write specs for os, then write code

CM88 key:
- dn = dining (philosophers)
- udn = user process spec
- odn = os process spec
Solution Approaches

Naive: ask for permission for all forks and wait
  • Problem: deadlock (e.g., if a cycle exists)

Require all forks to be acquired at once
  • Request all forks; if not successful, release & retry
  • Problem: ________________________________

“Hygenic” solution
  • Resolve symmetry by establishing a priority order
  • Can break the symmetry in the undirected links by giving each link a direction (establishes priority)
  • Choosing a direction establishes a partial order, but need to make sure not to introduce any cycles
  • As in mutual exclusion, as algorithm proceeds we have to maintain this structure (= update directions)

Key properties
  • Conflicts for a shared resource (i.e., the fork) are resolved in favor of the process with higher priority (according to the partial order).
  • After a process “wins” a conflict (and gets to eat), its neighbors should be given a chance to win the next round. So the priority of the winner is lowered.
Specification Refinement #1: Safety

Original specification of composite program:

\[ \text{dn1}: \left( \forall u, v :: \text{invariant}. \left( \neg (E(u, v) \land u.e \land v.e) \right) \right) \]

- Can implement this invariant by making use of a token (*a la* mutual exclusion)
- For each edge \((u, v)\) in the graph, establish a token \(fork(u, v)\) that keeps track of who has access to the shared resource (fork) at the current time
- New spec: if \(u\) is eating (in CS), then it must have the token

\[ \text{odn9}: \left( \forall u, v :: \text{invariant}. \left( u.e \land E(u, v) \Rightarrow fork(u, v) = u \right) \right) \]

- New spec satisfies the old spec since token can only be in one place at a time

Implement that idea of a token by *refining* the specification

- Add new variables/functions and write specification in term of those quantities
- New specification should satisfy the original specification
- In setting up the new specification, you are making a choice about program structure
  - For dining philosophers, this refinement means we will use a token-based approach to enforce mutual exclusion on each edge
Specification Refinement #2: Priority

Need to break the symmetry between philosophers

- Basic idea: establish some sort of priority on the graph

\[
u \leq v \equiv (fork(u, v) = v \land clean(u, v)) \lor (fork(u, v) = u \land \neg clean(u, v))\]

Establish desired properties (informal refinement)

1. An eating process holds all its forks and the forks are dirty.
2. A process holding a clean fork continues to hold it (and it remains clean) until the process eats.
3. A dirty fork remains dirty until it is sent from one process to another (at which point it is cleaned)
4. Clean forks are held only by hungry philosophers

Example (right)
Formal Specification Refinement: Priority

Initial specification

- **odn9**: \((\forall u, v :: \text{invariant.}(u.e \land E(u, v) \Rightarrow fork(u, v) = u))\)
- **dn2**: (progress): \(u.h \leadsto u.e\)

Refinement

- **odn10**: \(\text{invariant}(u.e \land E(u, v) \Rightarrow fork(u, v) = u \land \neg clean(u, v))\)
  - An eating process holds all fork and the forks are dirty
- **odn11**: \(fork(u, v) = u \land clean(u, v)\) **unless** \(u.e\)
  - Process holding a clean fork continues to hold it (and remains clean) until it eats
- **odn12**: \(fork(u, v) = u \land \neg clean(u, v)\) **unless** \(fork(u, v) = u \land clean(u, v)\)
  - Dirty fork remains dirty until it is sent to another agent (and then it is cleaned)
- **odn13**: \(\text{invariant}(fork(u, v) = u \land clean(u, v) \Rightarrow u.h)\)
  - Clean forks are only held by hungry philosophers

Claim: if new specs are satisfied, so are the old ones

- odn9 established tokens as means of safety; odn10 refines this (adds a condition)
- odn11 + odn13 provide a way to capture \(u.h \leadsto u.e\): if I have a clean fork then I am hungry [odn13] and if I have a clean fork then I will eat (eventually) eat [odn11]
- odn12 + odn13 says that I only pass fork to hungry neighbor, and they get clean fork
Specification Refinement #3: Token Request

Problem: how do we know if our neighbor is hungry?
• Need this in order to implement previous spec

Solution: add a “request token” req(u,v) to each edge
• Idea: if agent is hungry, doesn’t have fork, and has the request token, then send request to v (set req(u,v) = v)

Refined specifications
• odn14: invariant( (fork(u,v)=u) ∧ (req(u,v) = u) ⇒ v.h )
• odn16: fork(u,v) = v ∧ req(u,v) = u ∧ u.h \textbf{ensures} req(u,v) = v
• odn18: fork(u,v) = u ∧ req(u,v) = u ∧ ¬clean(u,v) ∧ ¬ u.e \textbf{ensures} fork(u,v) = v
  - Establishes conditions under which u sends a fork to v
• odn19: u.h ∧ (∀v : E(u,v) : fork(u,v) = u ∧ (clean(u,v) ∨ req(u,v) = v) \textbf{ensures} u.e ∧ ¬clean(u,v)

Claim: new specs ⇒ previous specs
Recall: P \textbf{ensures} Q ⇒ P \sim Q
• Need to show that if v is hungry, then u will know this (odn14 + odn16)
• Need to show we don’t violate any previous invariants (required for correctness)
• Details are (relatively) straightforward ⇒ see CM88 (beware changes in notation!)

Note: getting very close to a program description (ensures = guarded actions)
Proof of Correctness: Priority

Goal: show that refinements above lead to satisfaction of the original specification
- Safety: (safety): invariant \((\neg(u.e \land v.e \land E(u,v))\) in user | os
- Progress: (progress): \(u.h \sim u.e\)

Key part is showing that priority refinement allows hungry agents to eventually eat
- Approach: establish a metric (variant) that guarantees progress
- Metric (for each agent): \(u.m = (\# v \geq u) + (\# v \geq u : v.t)\)
  - Number of agents w/ higher priority + number w/ higher priority that are thinking
- If \(v.m = 0\) then I get to eat (nobody of higher priority => my requests will be honored)
- Can show that the metric \(u.m\) never increases:
  - if agent \(w\) above me eats, then they make a vertex into a sink => number of processes above me has to go down (since I can’t go through \(w\) anymore)
  - if a process becomes thinking then it must have been eating => became a sink
- Can show that the metric \(u.m\) eventually decreases
  - Define \(u.top = true\) if a process is a the top of the partial order
    \[ u.top \equiv (\forall v : E(u,v) \land v.h : v < u) \]
  - Can show that a process at the top eats eventually
  - If a process is not at the top, there is a hungry process above it that is at the top
Unity Algorithm for Dining Philosophers

Final set of refined specifications (from slide 8):

- \( \text{fork}(u,v) = v \land \text{req}(u,v) = u \land u.h \) \textbf{ensures} \( \text{req}(u,v) = v \)  
- \( \text{fork}(u,v) = u \land \text{req}(u,v) = u \land \neg \text{clean}(u,v) \land \neg u.e \) \textbf{ensures} \( \text{fork}(u,v) = v \)  
- \( u.h \land (\forall v : E(u,v) : \text{fork}(u,v)=u \land (\text{clean}(u,v) \lor \text{req}(u,v)=v)) \) \textbf{ensures} \( u.e \land \neg \text{clean}(u,v) \)

<table>
<thead>
<tr>
<th>Program</th>
<th>hungry</th>
</tr>
</thead>
<tbody>
<tr>
<td>initially</td>
<td>( p.\text{state} = \text{thinking} )</td>
</tr>
<tr>
<td></td>
<td>( (\forall q : E(p,q) : \text{clean}(p,q) = \text{false} ) )</td>
</tr>
<tr>
<td></td>
<td>Priorities form a partial order</td>
</tr>
<tr>
<td>always</td>
<td>( p.t \equiv p.\text{state} = \text{thinking} )</td>
</tr>
<tr>
<td></td>
<td>( p.h \equiv p.\text{state} = \text{hungry} )</td>
</tr>
<tr>
<td></td>
<td>( p.e \equiv p.\text{state} = \text{eating} )</td>
</tr>
</tbody>
</table>

**assign**

- \( [H_p] \) \( p.h \land \text{fork}(p,q) = q \)  
  \( \rightarrow \text{req}(p,q) := q; \)  
- \( [E_p] \) \( p.h \land (\forall q : E(p,q) : \text{fork}(p,q) = p \land (\text{clean}(p,q) \lor \text{req}(p,q) = q) ) \)  
  \( \rightarrow p.\text{state} := \text{eating}; \)  
  \( \quad \text{clean}(p,q) := \text{false}; \)  
- \( [R_p] \) \( \text{req}(p,q) = p \land \text{fork}(p,q) = p \land \neg \text{clean}(p,q) \land \neg p.e \)  
  \( \rightarrow \text{fork}(p,q) := q; \)  
  \( \quad \text{clean}(p,q) := \neg \text{clean}(p,q); \)
Add an animation here that shows all of the tokens moving around for the previous example (slide 6).

Also include a sketch of the proof technique?
Summary: Specifications and Refinement

Key ideas:

- Specifications for composed systems
  - Properties of the underlying process (user)
  - Properties of the composed system (user | os)
  - Constraints on access to user processes

- Design via successive refinement
  - Refine properties to establish program structure
  - Each refinement solves problem from previous level (and satisfies the prior specs)
  - Final specification can be converted to code

- Advantages of this approach
  - Maintain a formal proof structure throughout
  - Painful, but necessary for safety critical systems!

Next week: fault tolerance (Byzantine agreement, Paxos)

- Q: What happens when we have failures (eg, messages are sent but don’t arrive, processes fail, etc)
- A: still possible to make everything work correctly (with formal certificates)