Take away concepts from the course:

• State transition systems.
• Always (invariant, safety). Nothing bad ever happens.
• Progress. Variant function. Lyapunov function. Metric. The system never gets further away from its goal and eventually gets closer.
• Global view of the designer mapped to local view of the agent.
• Data structures, e.g. graphs
**Unless Property**

**P unless Q:**
All transitions from states in which P holds are to states in which Q holds

- thinking unless hungry
- hungry unless eating
- eating unless thinking
Given: Undirected graph. Each node consists of an OS process and a client process

Specification:
Always: Neighboring clients are not in critical sections.
Eventually: Every client waiting to enter its critical section does so.
Given: Clients remain in critical sections for finite time.
Channels between neighbors.
Always: Priority graph is acyclic.
So: how should priorities change when a process eats?

v holds all its forks and eats

What should happen to edge directions when v eats?
• Make all edges directed towards v?
An agent holding a dirty fork has lower priority. An agent holding a clean fork has higher priority. Fork in channel from u to v implies v has priority.

Priority changes only when a clean fork becomes dirty.
Always properties:
• Thinking philosopher does not hold clean forks. (It may hold dirty forks.)
• Eating philosopher holds all forks incident on it, and all these forks are dirty.
• A clean fork is either held by a hungry philosopher or is in a channel to a hungry philosopher.

Proposal for an algorithm
• Eating philosopher that gets a request for a fork sends the fork after it finishes eating. (If it does not get a request for a fork it holds on to it.)
• Thinking or hungry philosopher that gets a request for a fork sends the fork if it is dirty, and holds on to the fork if it is clean.

Is the algorithm correct? Safety: obvious. Progress? Not clear
What can go wrong? Can a philosopher y remain hungry for ever because it never gets a fork from a neighbor?

Suppose y becomes hungry. Can you think of a scenario where y never gets its forks?

Could a cabal of philosophers eat repeatedly and cause others to starve for ever?
The only way to prove progress:

Find a function $f$ from states to a well-founded set such that:

**Safety**: No state transition takes the system further from its goal. **For all k**: Stable($f < k$)

**Progress**: Eventually, a state transition occurs which takes the system closer to its goal:

**For all k**: (not goal and $f = k$) leads-to (goal or $f < k$)

$f$ is bounded below. Induction on well-founded sets tells us that $f$ cannot decrease infinitely.

(Cannot carry out induction on some continuous functions. We prove convergence using limit arguments. See Lyapunov functions, CDS.)
Safety: Every transition reaches goal or decreases $f$
Progress: There exists a fair transition which either reaches goal or decreases $f$

$f$ is the variant function

Goal state

Safety: Every transition reaches goal or decreases $f$
Progress: There exists a fair transition which either reaches goal or decreases $f$
**Safety**: No transition takes the system further from its goal. 
Proof?
Consider transitions:
- thinking to hungry: decreases or no change to $f$
- hungry to eating: decreases or no change to $f$
- eating to thinking: no change to $f$
Progress: Prove that there exists a fair transition which results in the goal or decreases $f$.

Proof outline: Highest priority hungry process transits to eating or a higher priority thinking neighbor transits to hungry.

$f(s)$: Lexicographic ordering (number of higher priority thinking processes, number of higher priority hungry processes)
Writing code for a message-passing process

When a process of type \( p \) gets a message of type \( m \) then:
1. \( p \) changes its local variables (i.e. its state)
2. Sends messages \( m', m'', \ldots \) to its neighbors.

Steps in coding:
• Identify message types
  • Between Client and OS: request, resource tokens
  • Between OS neighbors: forks, request
• Identify local variables:
  • OS: List of forks, requests that the process holds.
  • Client: list of tokens it holds.
• Write actions.