The Dining Philosophers
Authors: James Reinke and Will Tachau

- The Algorithm Summary
    - The algorithm we have implemented is very similar to the hygienic solution presented in class. The key to avoiding deadlock, the situation where no progress/change is made during after a cycle of the algorithm, is ensuring that our graph is acyclic. If the graph is acyclic, we ensure that philosophers that fail to make progress do not hold forks. We keep the graph acyclic in the following way. Philosophers have 5 states (not including gone, where a philosophers fails to exist in memory) joining, thinking, hungry, eating and leaving. If these states are ranked in this order, our algorithm ensures that a philosopher only makes progress as they move from the states. Progress is made by gaining priority of forks. The only time progress is lost is when a philosopher finishes eating. In this event the philosopher loses priority over every fork he holds. Because progress can only be made when progress for another philosopher is lost, we know that graph only changes when a philosopher finishes eating or a new philosopher is added. If a philosopher gives up priority for every fork, we know this philosophers arrows point away form the node, disenabling a cyclic traversal of the graph. In this scenario, we always have at least one philosopher that has priority over 0 forks. Thus, our graph remains acyclic. When a new philosopher is added, he gives his fork to every single neighbor and ensures they are clean. Thus, the newest philosopher maintains the acyclic property. The addition of this philosopher in this manner will never change the states of its neighbors, so adding a new philosopher becomes trivial.
- States
    - Joining, Thinking, Hungry, Eating, Leaving, Gone
- Information Stored by Each Process
    - The processes need to store certain information based on what state they are in.
    - Joining
        - If a philosopher is joining, he needs to hold two lists of information. One contains the philosopher's neighbors, which is a list of unique ID's. The other list is a list of the neighbors that have confirmed the philosopher's presence.
    - Thinking
        - A thinking philosophers only needs to store a list its neighbors and a list of forks, which are tuples containing the ID of the philosopher it shares a fork with and a state dirty/clean. We do <u>NOT</u> need to store hungry requests. In this state, if a neighbor sends a message that it is hungry we immediately send them a fork.

    - Hungry
        - A hungry philosopher stores four different things. A list of neighbors, a list of forks, a list of hungry neighbors and the process ID of the controller. We also store the message reference from the controller so we can send it back when the philosopher starts eating.
    - Eating
        - An eating philosopher stores four different things. A list of neighbors, a list of forks, a list of hungry neighbors and the process ID of the controller.
    - Leaving
        - Leaving philosopher needs to store the controller's process ID and a list of neighbors so that it can inform both that it is leaving. We also store the message reference from the controller so we can send it back when the philosopher leaves.
    - Gone
        - A gone philosopher does not exist in memory and does not store anything.
- Message Types
    - joining
        - A joining atom is sent to every neighbor when a philosopher enters the system along with a unique processor name for the joining philosopher.
    - become_hungry
        - This is sent by the controller to any thinking philosopher.
    - eating
        - A philosopher can send an eating atom to the controller when the philosopher transitions from hungry to eating. This allows the controller to know which processes are eating so that it can tell them to eventually stop eating.
    - stop_eating
        - The controller can send a stop_eating atom to a philosopher. This can only be sent to an eating philosopher.
    - leave
        - The controller can send a leaving atom to a philosopher. This can be sent to a philosopher in any state other than leaving and joining.
    - gone
        - A philosopher sends each neighbor and the controller a gone atom when it leaves the system. This is sent by a philosopher when transitioning from leaving to gone.
    - give_fork
        - A philosopher can send a give_fork atom to another philosopher. This only happens when the receiving philosopher is hungry. If the sending philosopher is hungry, the receiving philosopher must have greater priority.

- eat_request
  - A philosopher sends another philosopher an eat_request atom. A philosopher sends this atom only when they transition from thinking to hungry. When a philosopher transitions from thinking to hungry, he sends this atom to every single neighbor.

- Algorithm Rules
  - Any State
    - joining
      - Adds the name of philosopher that sent the join atom and adds a clean fork with the joining philosopher's name.
    - leave
      - The philosopher sends a gone atom to each neighbor and the controller and then exists the system.
    - gone
      - The receiving philosopher deletes the PID of the sending philosopher from its neighbor list and deletes the key associated between the two of them if the philosopher holds the key.
  - Joining
    - We assume this philosopher does not receive command messages from other philosophers until they have acknowledged its presence. We accept that a philosopher might leave before sending its confirmation, in which case this philosopher will never be actualized to thinking.
    - confirmed
      - When a joining philosopher receives a confirmed atom, it stores the PID of the sending philosopher to a list of confirmed philosophers. If this list is the same size as the neighbors list, the philosopher transitions from joining to thinking. Otherwise, it remains thinking waiting for the rest of the neighbors to confirm its existence.
  - Thinking
    - become_hungry
      - When a thinking philosopher receive this atom from the controller, he immediately becomes hungry and sends an eat_request to every single neighbor.
    - eat_request
      - If a thinking philosopher receives an eat_request atom, he immediately sends his fork to that philosopher, regardless of priority. The fork is cleaned in the process.
  - Hungry

        - eat_request
            - If a hungry philosopher receives an eat_requst atom while hungry, he only sends his fork to that philosopher if that philosopher has priority (i.e the fork is dirty). He does this by sending a give_fork atom and deletes the fork from its list. If it does not have the fork, the receiving philosopher does nothing. If the philosopher has greater priority and has the fork, he stores the request in a list of PID's which represent hungry neighbors.
        - give_fork
            - When a hungry philosopher receives a give_fork atom, he stores a tuple (the fork) of the PID of the philosopher who sent the fork and a clean atom.
        - Hungry -> Eating
            - Upon receiving any message, the philosopher checks if he has all forks. If he does, he becomes eating and sends a message to the controller notifying it that this philosopher has started eating. We also send it the reference we received from the eat_request.
    - Eating
        - eat_request
            - The philosopher immediately saves the PID of the sending philosopher to a list, which represents hungry neighbors.
        - stop_eating
            - When the philosopher stops eating, he sends a give_fork request to every philosopher on his hungry neighbor list. He then dirties every fork (giving priority to every neighbor but not explicitly sending the fork until a philosopher sends a eat_request).
    - Leaving
        - Leaving -> Gone
            - The philosopher always goes from leaving to gone after sending its messages to the neighbors. When it transitions from leaving to gone it sends the reference receiving from the leave atom back to the controller.

These rules ensure that our graph remains acyclic. One only has to read the eating and joining procedures to see that whenever priority is given to others, it is done so in a manner that ensures the philosopher losing priority does so for every fork it is holding. These outward arrows ensure the acyclic property. By only making progress until we reach eating, we ensure every hungry philosopher eventually eats. This algorithm only allows neighbors of eating -> thinking or any state -> gone philosophers to make progress. Thus, the runtime would be slow if only one

philosopher finishes per iteration. But it is possible that we have many philosophers going from eating -> thinking, but they can't be neighbors.