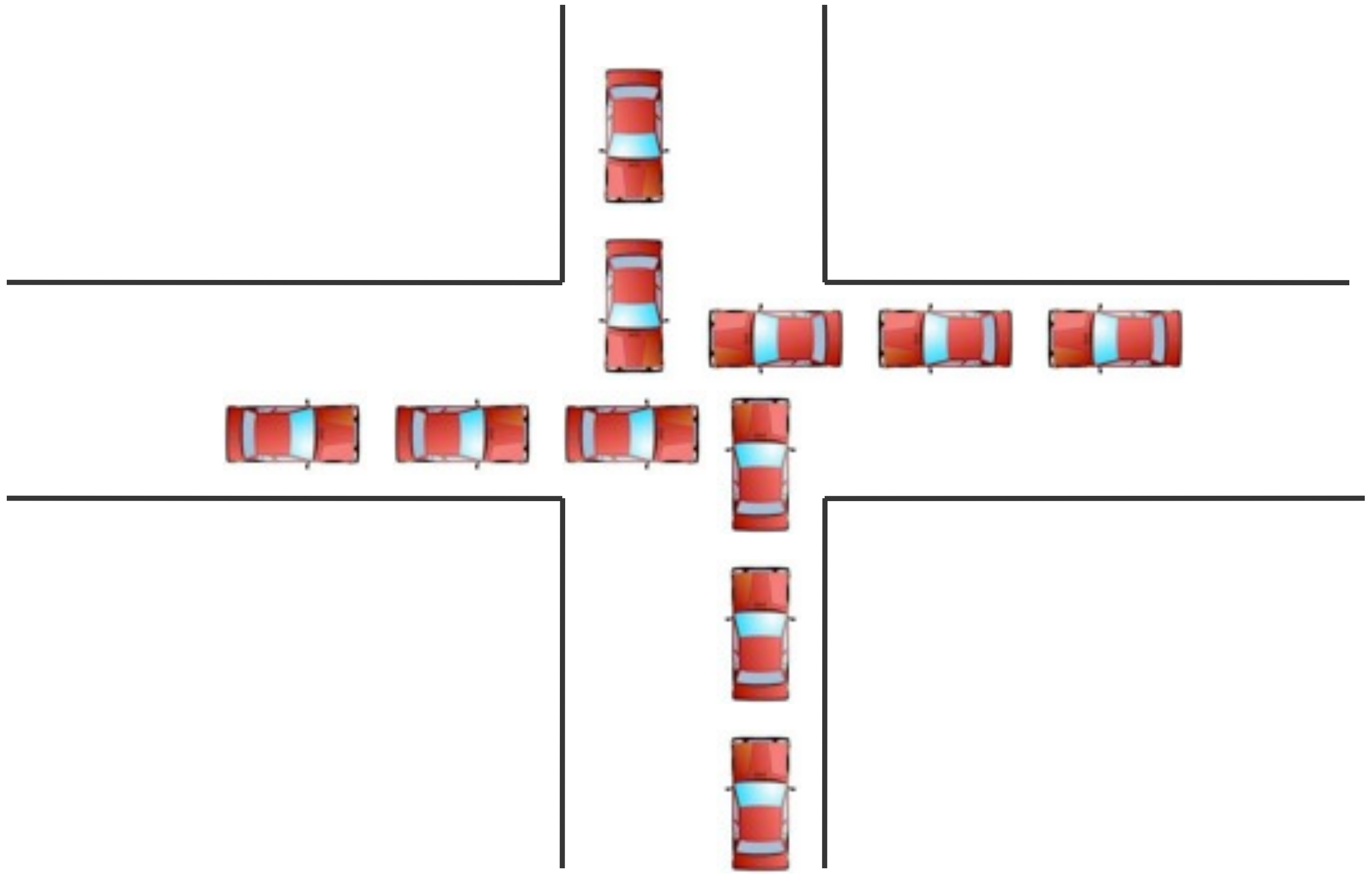
Deadlocks

Deadlock



Deadlock Definition

- A process is **deadlocked** if it is waiting for an event that will never occur.
  - Typically, but not necessarily, more than one process will be involved together in a deadlock
- Is deadlock the same as starvation (i.e., indefinitely postponed)?
  - A process is **indefinitely postponed** if it is delayed repeatedly over a *long* period of time while the attention of the system is given to other processes. (Logically the process may proceed but the system never gives it the CPU.)

Necessary Conditions for Deadlock

- **Mutual exclusion**
  - Processes claim **exclusive** control of the resources they require
- **Hold-and-wait (a.k.a. wait-for) condition**
  - Processes hold resources already allocated to them while waiting for additional resources
- **No preemption condition**
  - Resources cannot be removed from the processes holding them until used to completion
- **Circular wait condition**
  - A **circular chain** of processes exists in which each process holds one or more resources that are requested by the next process in the chain

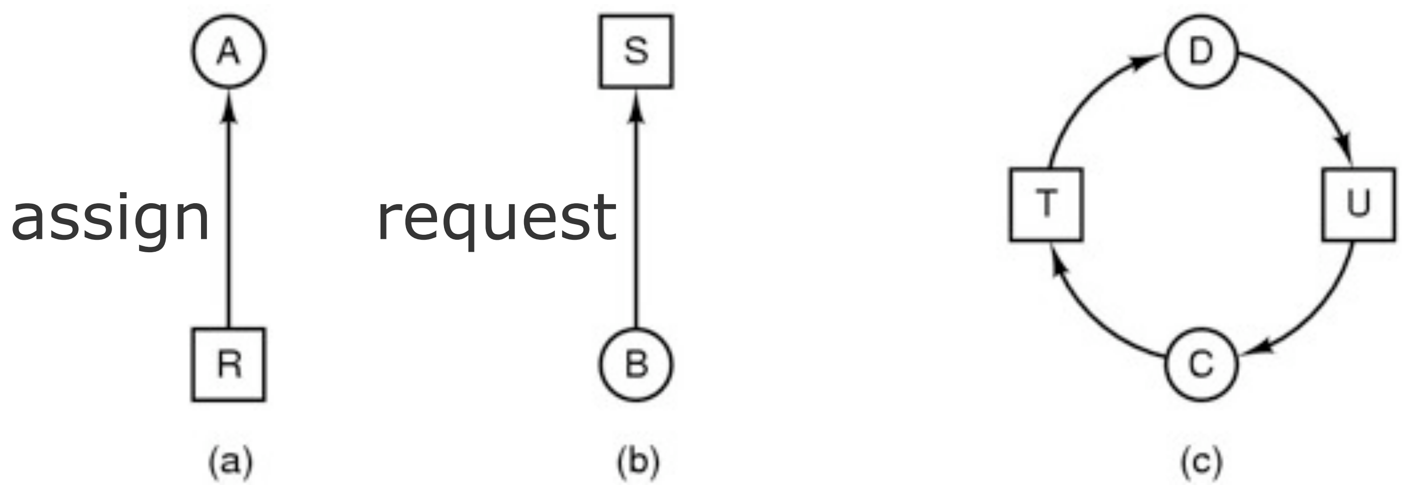
Dining Philosophers had it all

- **Mutual exclusion:** Exclusive use of chopsticks
- **Hold and wait:** Hold 1 chopstick, wait for next
- **No preemption:** Cannot force another philosopher to undo their hold
- **Circular wait:** Each waits for next neighbor to put down chopstick

Formalizing circular wait: The resource allocation graph

- **Nodes**
  - Processes
  - Resources
- **Arcs**
  - From resource to process = **resource assigned to process**
  - From process to resource = **process requests (and is waiting for) resource**

Formalizing circular wait: The resource allocation graph



(a) resource R assigned to process A

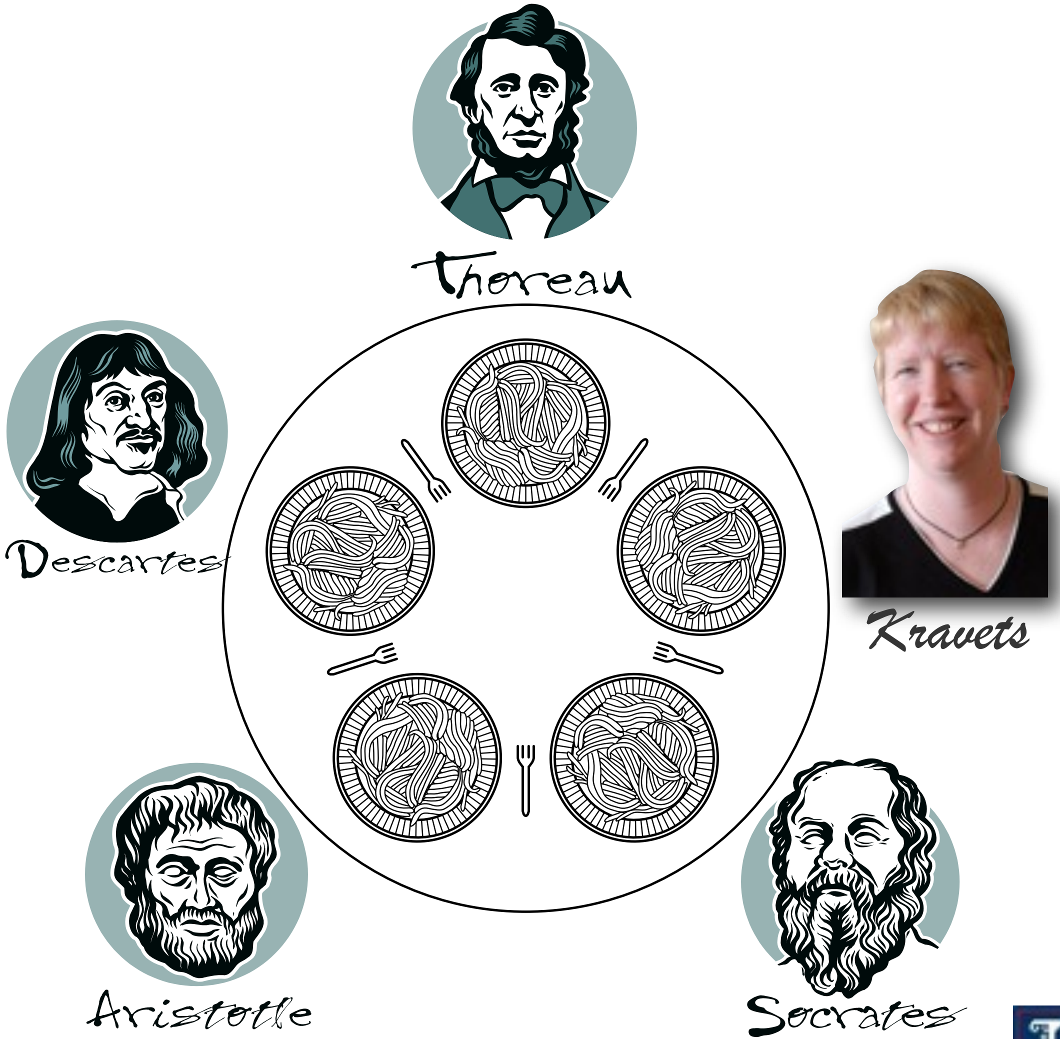
(b) process B is requesting/waiting for resource S

(c) process C and D are in deadlock over resources T and U

Dining Philosophers resource allocation graph

If we use the trivial broken "solution"...

```
void philosopher(i) {
  while true {
    take left fork;
    take right fork;
    eat();
    put left fork;
    put right fork;
  }
}
```

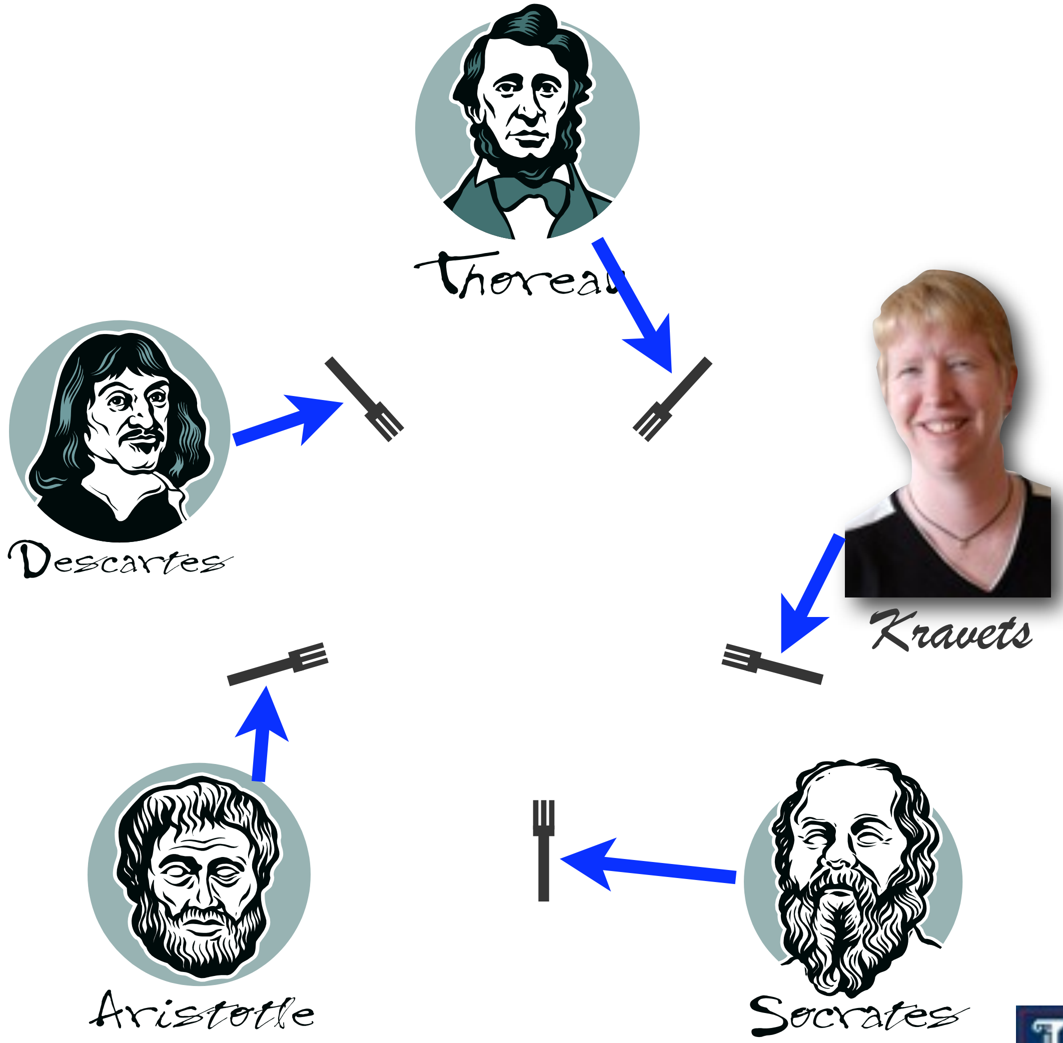


Dining Philosophers resource allocation graph

If we use the trivial broken "solution"...

One node per philosopher and per fork

1. Everyone tries to pick up left fork (request edges)

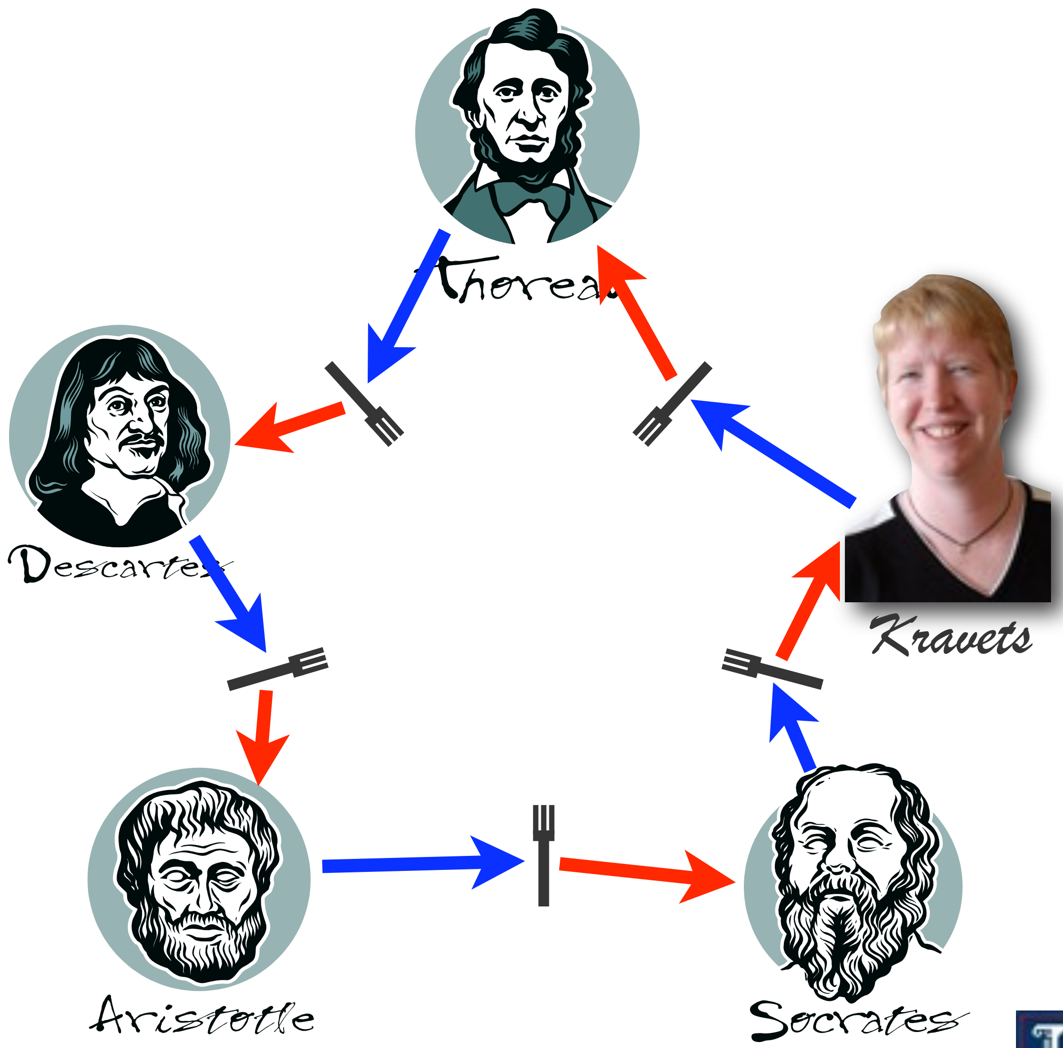


Dining Philosophers resource allocation graph

If we use the trivial broken "solution"...

One node per philosopher and per fork

1. Everyone tries to pick up left fork (request edges)
2. Everyone succeeds! (request edges become assignment edges)
3. Everyone tries to pick up right fork (request edges)
4. Cycle => deadlock



Summary so far

- Definition of deadlock
- 4 conditions for deadlock to happen
  - Mutual exclusion
  - Hold-and-wait
  - No preemption
  - Circular wait
    - = cycle in resource allocation graph
- Next: How to deal with deadlock

How to deal with deadlocks

- **The default**
  - The "ostrich solution"
- **Prevention**
  - design a system in such a way that deadlocks cannot occur, at least with respect to serially reusable resources.
- **Detection**
  - in a system that allows the possibility of deadlock, determine if deadlock has occurred, and which processes and resources are involved.
- **Recovery**
  - after a deadlock has been detected, clear the problem, allowing the deadlocked processes to complete and the resources to be reused. Usually involves destroying (at least one of) the affected processes and starting them over.
- **Avoidance**
  - impose less stringent conditions than for prevention, allowing the possibility of deadlock, but sidestepping it as it approaches.

The Default:Ostrich solution

- Do nothing! Deadlocked processes just stay stuck.
- Rationale: Make the common path faster and more reliable
  - Deadlock prevention, avoidance or detection/recovery algorithms are expensive
  - If deadlock occurs only rarely, it is not worth the overhead to implement any of these algorithms.

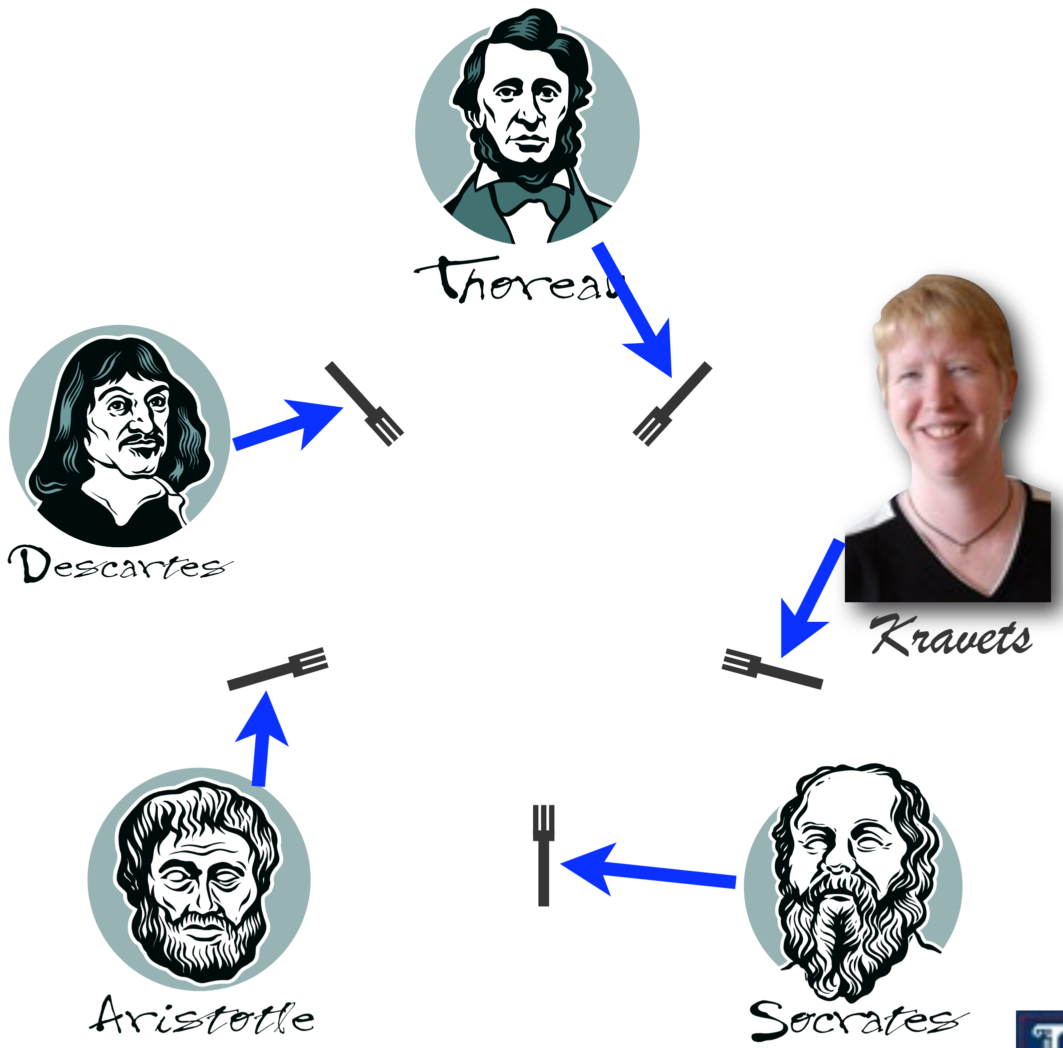
Deadlock Prevention

- Build the system so as to break one of the deadlock conditions.
  - *Mutual exclusion*
    - Solution: Allow multiple processes to access CS. E.g., reading a file.
  - *Hold-and-Wait condition*
    - Solution: Force each process to request all required resources at once (i.e., in one shot). It cannot proceed until all resources have been acquired, i.e., process either acquires all resources or stops. Also called *two-phase locking*
  - *No preemption condition*
    - Solution: Allow a process to be aborted or its resources reclaimed by another or by system, when competing over a resource
  - *Circular wait condition*
    - Solution: All resource types are numbered by an integer resource id. Processes must request resources in numerical (decreasing) order of resource id.

Dining Philosophers solution with numbered resources

First, recall the trivial broken "solution"...

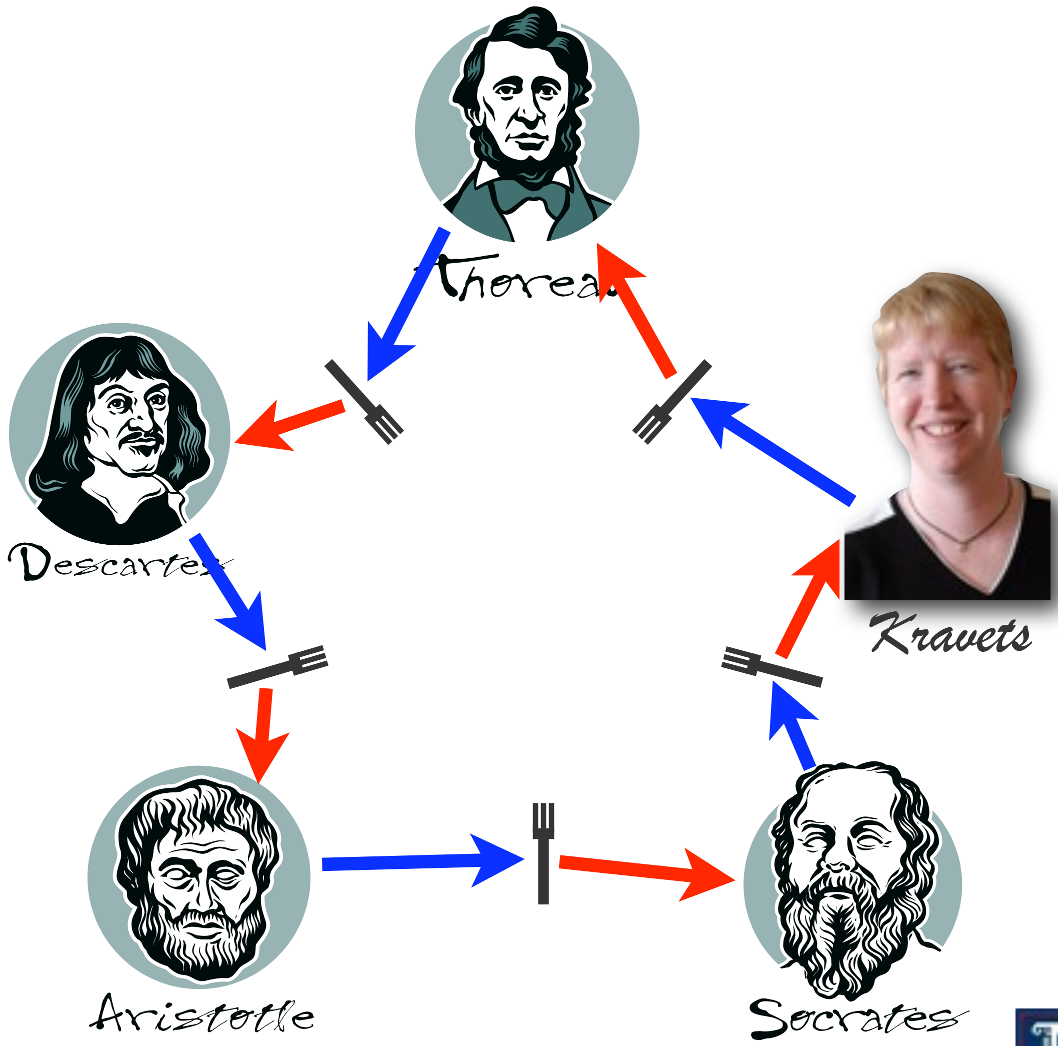
```
void philosopher(i) {
  while true {
    take left fork;
    take right fork;
    eat();
    put left fork;
    put right fork;
  }
}
```



Dining Philosophers solution with numbered resources

First, recall the trivial broken "solution"...

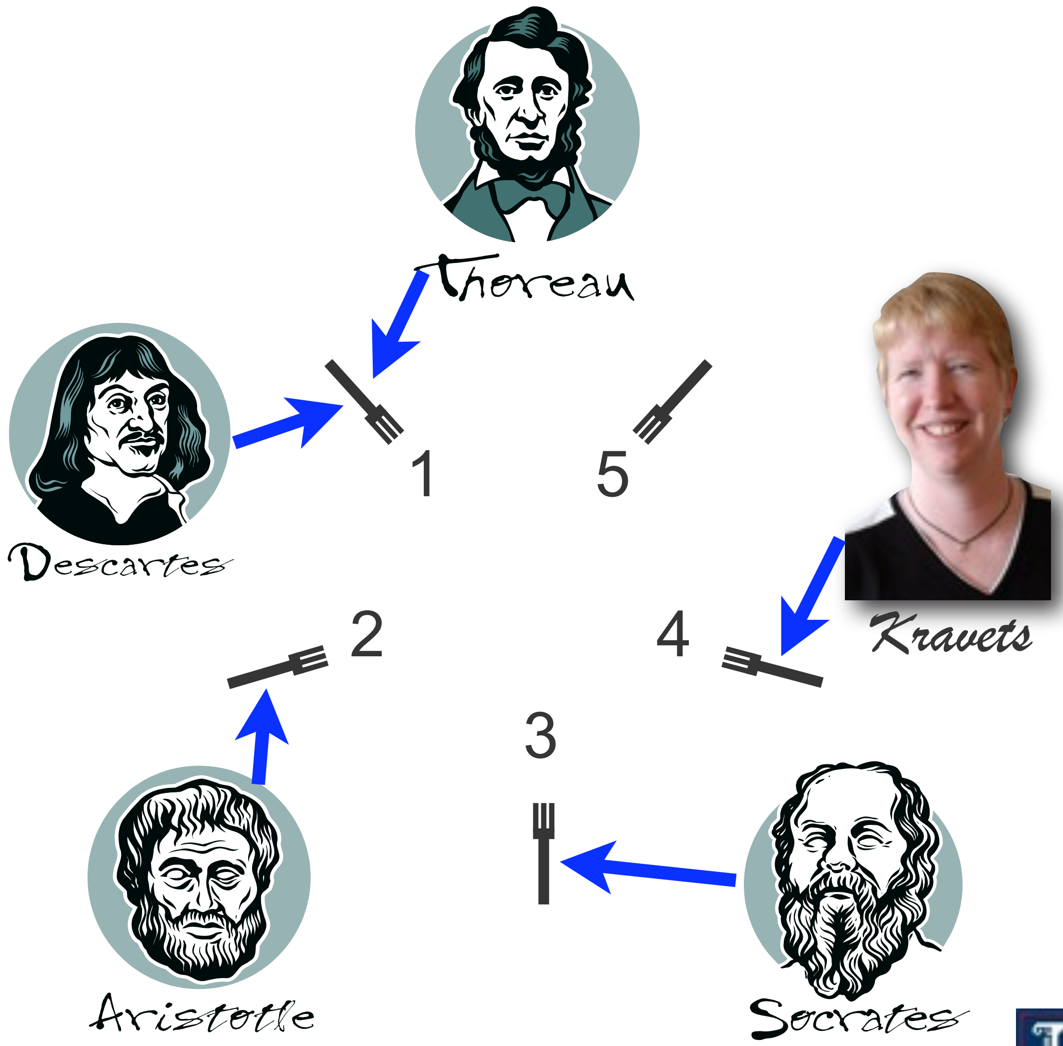
```
void philosopher(i) {
  while true {
    take left fork;
    take right fork;
    eat();
    put left fork;
    put right fork;
  }
}
```



Dining Philosophers solution with numbered resources

Instead, number resources...

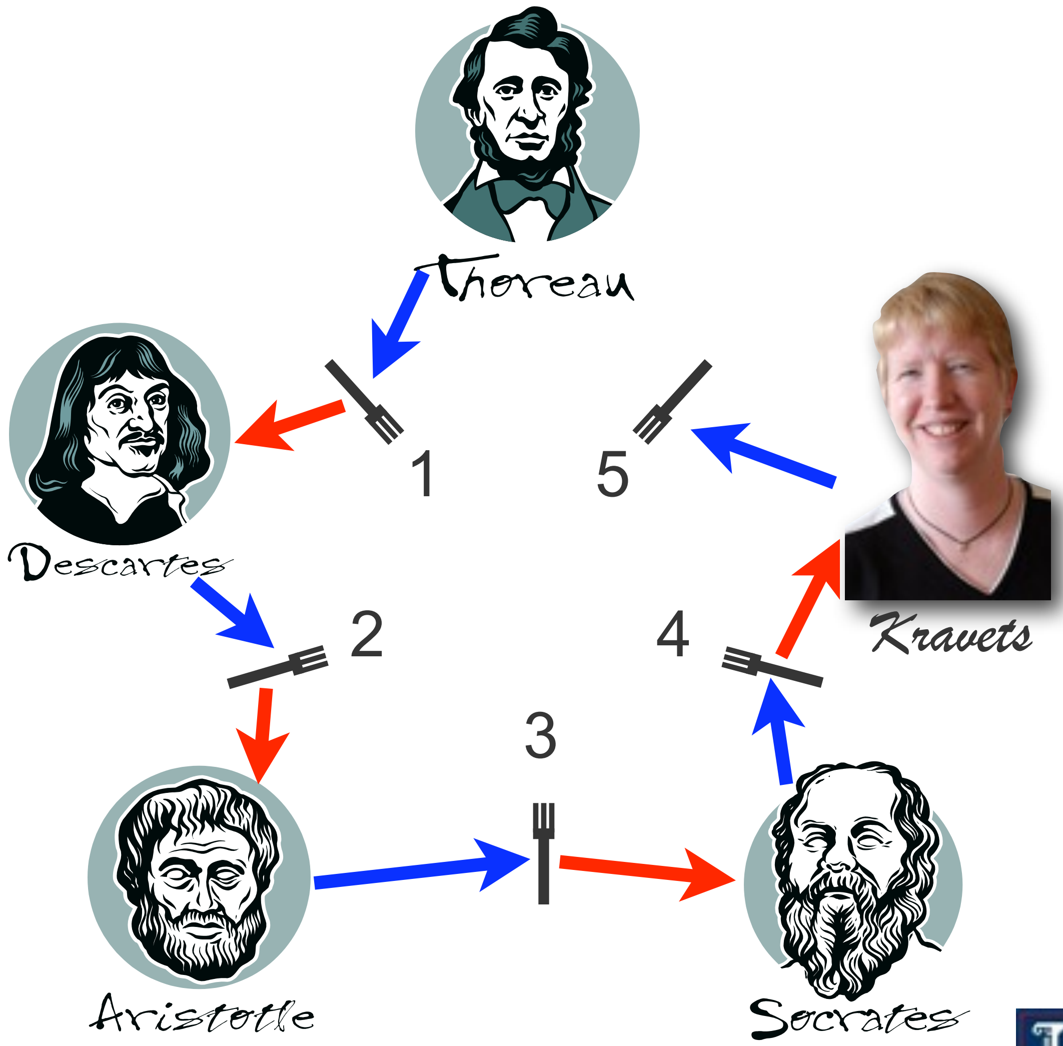
```
void philosopher(i) {
  while true {
    take lower-# fork;
    take higher-# fork;
    eat();
    put lower-# fork;
    put higher-# fork;
  }
}
```



Dining Philosophers solution with numbered resources

Instead, number resources...

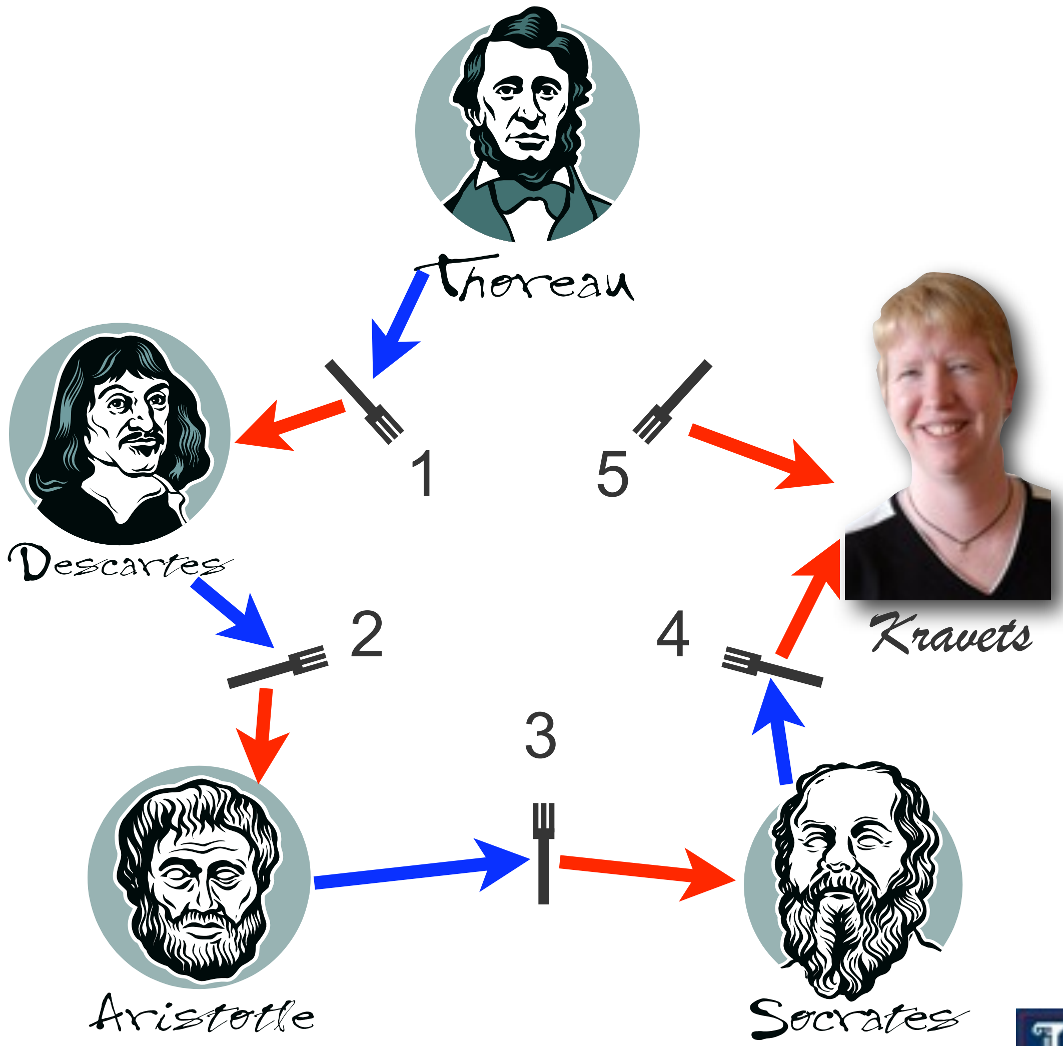
```
void philosopher(i) {
  while true {
    take lower-# fork;
    take higher-# fork;
    eat();
    put lower-# fork;
    put higher-# fork;
  }
}
```



Dining Philosophers solution with numbered resources

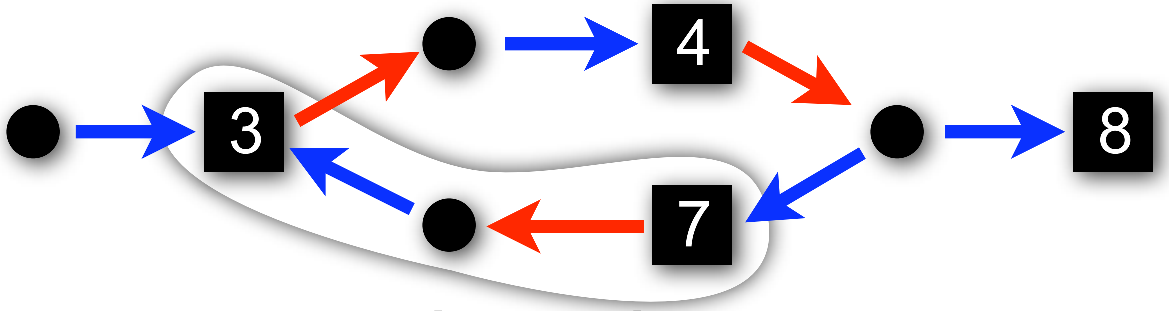
Instead, number resources...

```
void philosopher(i) {
  while true {
    take lower-# fork;
    take higher-# fork;
    eat();
    put lower-# fork;
    put higher-# fork;
  }
}
```



Ordered resource requests prevent deadlock: the proof

- Suppose we have a deadlock (proof by contradiction).
- Then what does the resource allocation graph look like?
  - Cycle!

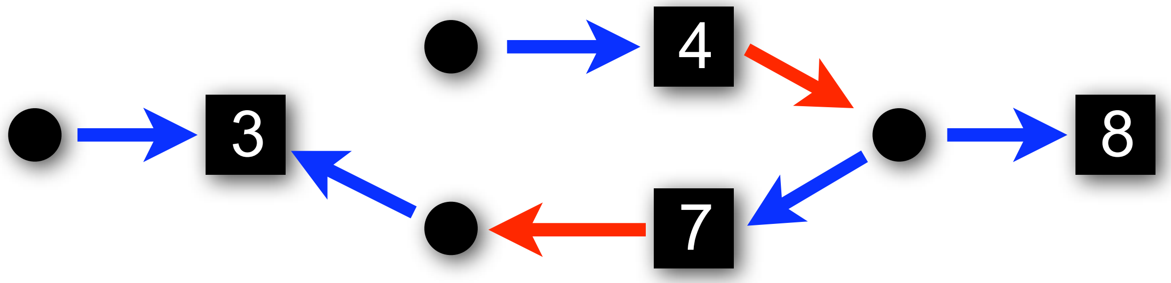


- Resources are numbered...
- and increasing around the cycle

Contra-diction!

Are we always in trouble without ordering resources?

- Not always:



- Ordered resource requests are sufficient to avoid deadlock, but not necessary.
- Convenient, but may be conservative.

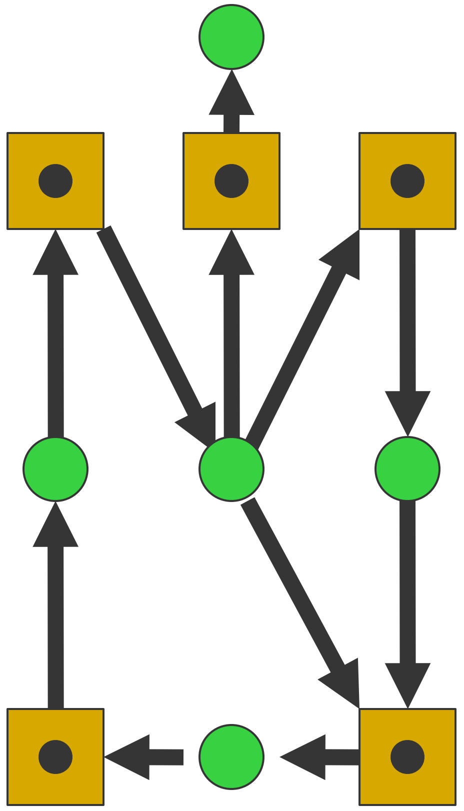
How to deal with deadlocks

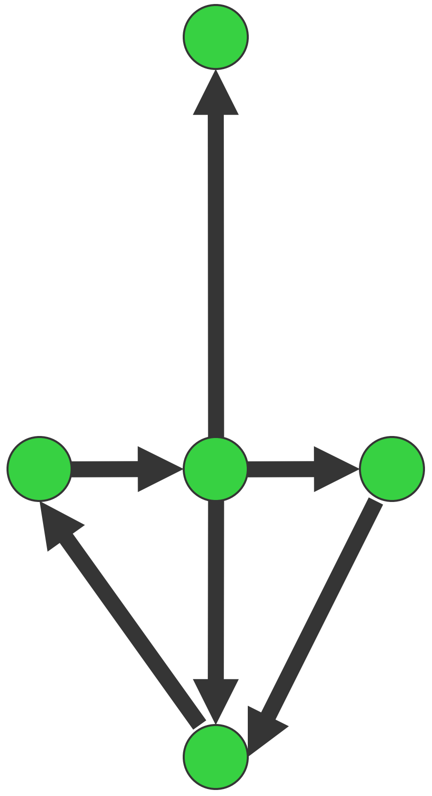
- **The default**
  - The "ostrich solution"
- **Prevention**
  - design a system in such a way that deadlocks cannot occur, at least with respect to serially reusable resources.
- **Detection**
  - in a system that allows the possibility of deadlock, determine if deadlock has occurred, and which processes and resources are involved.
- **Recovery**
  - after a deadlock has been detected, clear the problem, allowing the deadlocked processes to complete and the resources to be reused. Usually involves destroying (at least one of) the affected processes and starting them over.
- **Avoidance**
  - impose less stringent conditions than for prevention, allowing the possibility of deadlock, but sidestepping it as it approaches.

Deadlock Detection

- Check to see if a deadlock has occurred!
- Single resource per type
  - Can use wait-for graph
  - Check for cycles
    - How?

Wait for Graphs


Resource Allocation Graph


Corresponding Wait For Graph

Recovery From Deadlock

- OPTIONS:
  - Kill all deadlocked processes and release resources
  - Kill one deadlocked process at a time and release its resources
  - Rollback all or one of the processes to a checkpoint that occurred before they requested any resources
- Note: with rollback, difficult to prevent indefinite postponement

How to deal with deadlocks

- **The default**
  - The “ostrich solution”
- **Prevention**
  - design a system in such a way that deadlocks cannot occur, at least with respect to serially reusable resources.
- **Detection**
  - in a system that allows the possibility of deadlock, determine if deadlock has occurred, and which processes and resources are involved.
- **Recovery**
  - after a deadlock has been detected, clear the problem, allowing the deadlocked processes to complete and the resources to be reused. Usually involves destroying (at least one of) the affected processes and starting them over.
- **Avoidance**
  - impose less stringent conditions than for prevention, allowing the possibility of deadlock, but sidestepping it as it approaches.