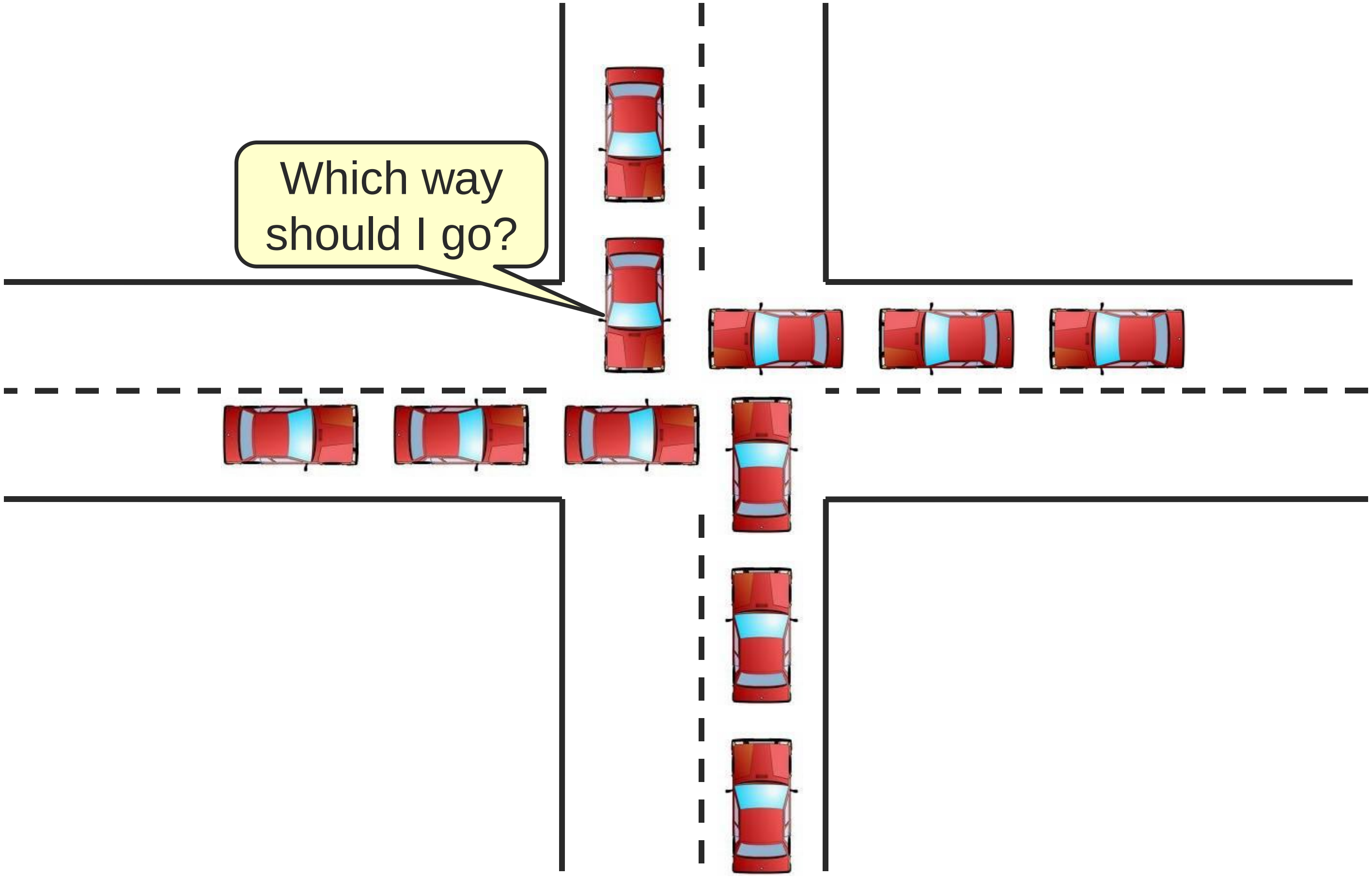
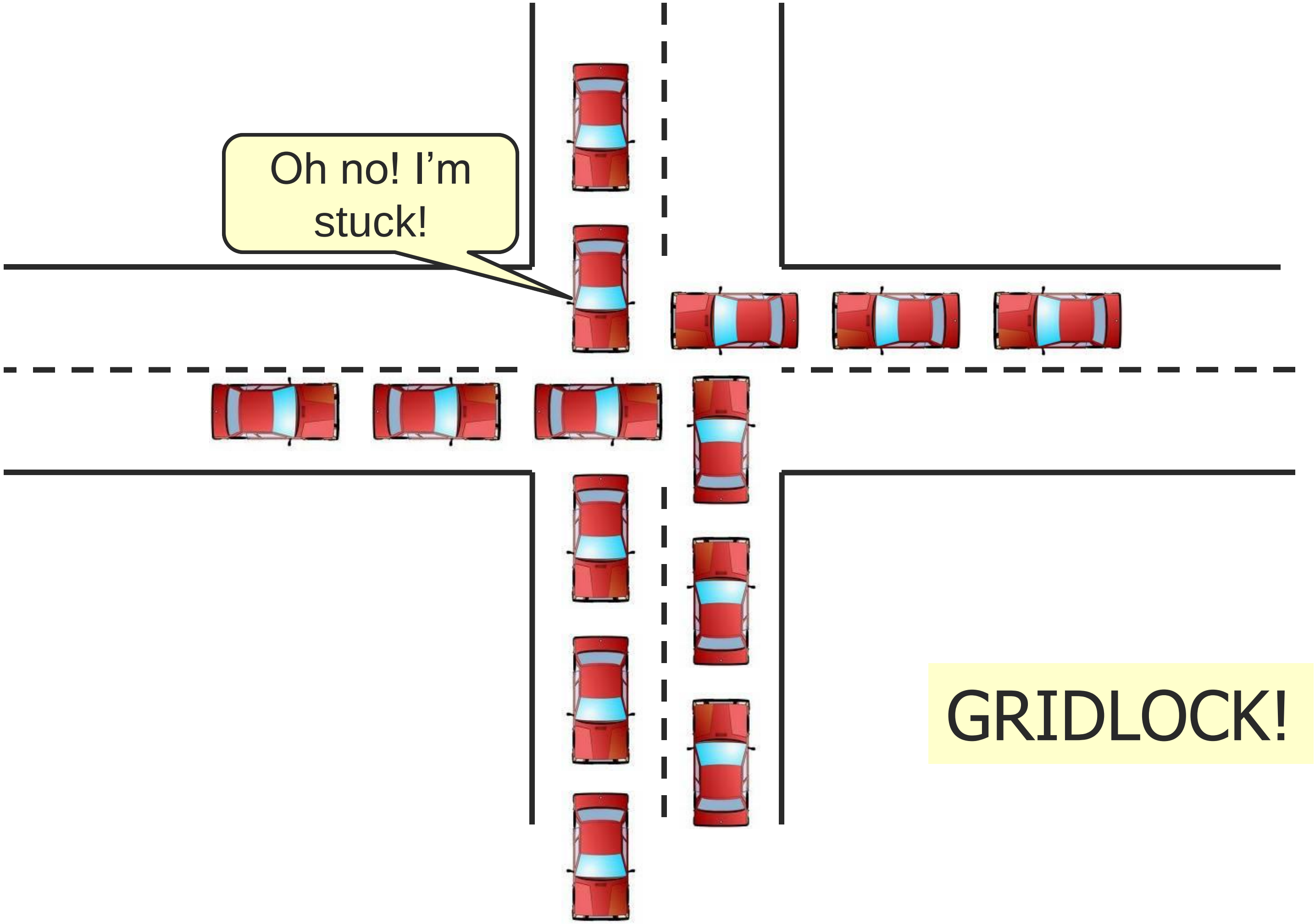
Deadlocks

Deadlock



Deadlock

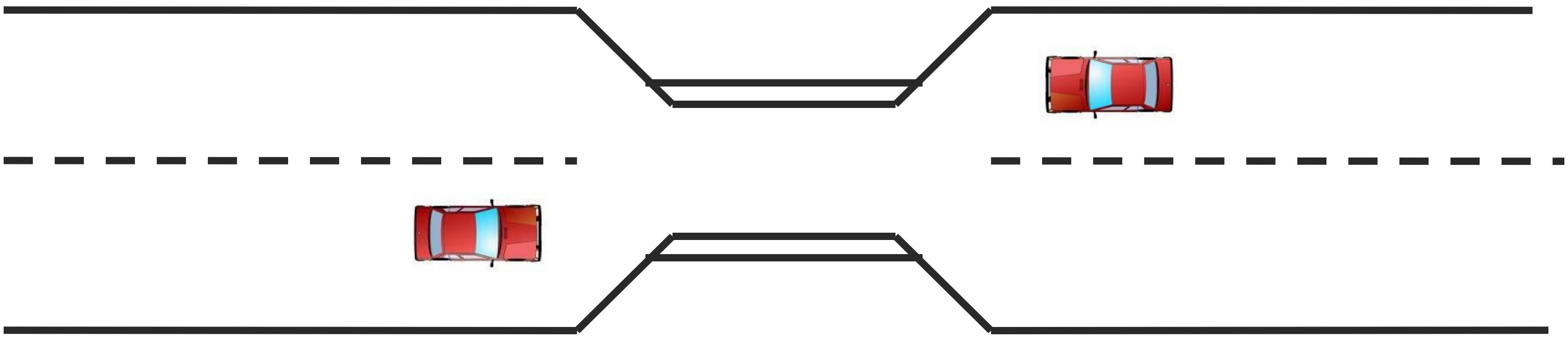


Deadlock Definition

- Deadlocked process
  - Waiting for an event that will never occur
  - Typically, but not necessarily, involves more than one process
    - A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause

How can a single process deadlock itself?

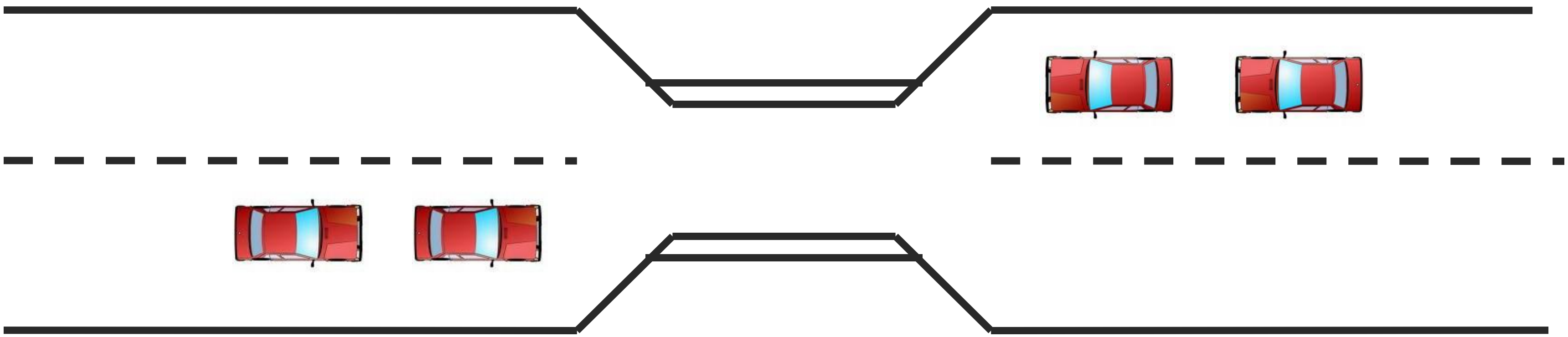
Deadlock: One-lane Bridge



- Traffic only in one direction
- Each section of a bridge can be viewed as a resource

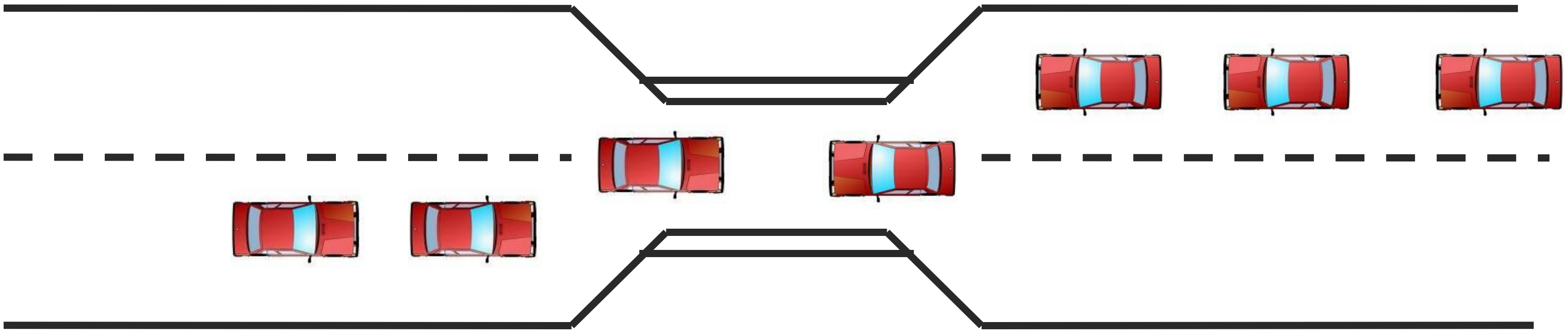
What can happen?

Deadlock: One-lane Bridge



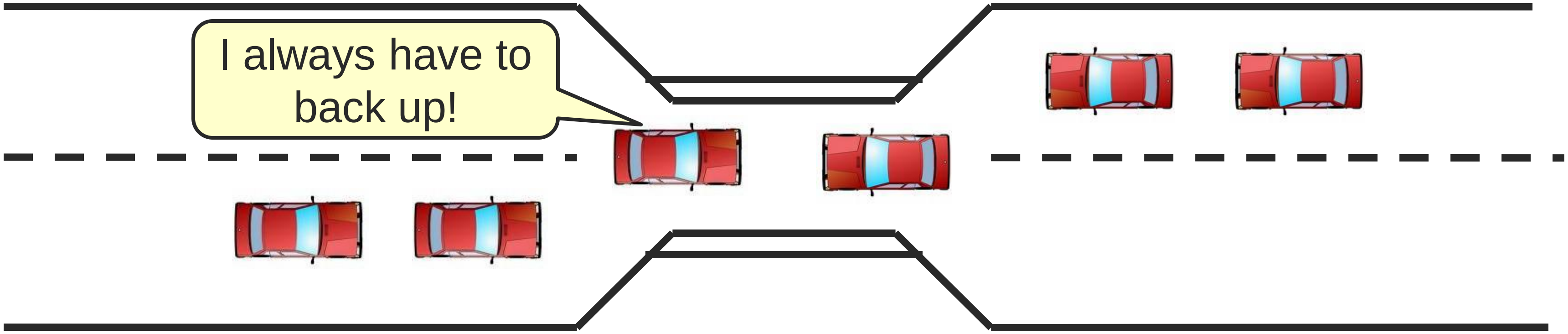
- Traffic only in one direction
- Each section of a bridge can be viewed as a resource
- Deadlock
  - Resolved if cars back up (preempt resources and rollback)
  - Several cars may have to be backed up

Deadlock: One-lane Bridge



- Traffic only in one direction
- Each section of a bridge can be viewed as a resource
- Deadlock
  - Resolved if cars back up (preempt resources and rollback)
  - Several cars may have to be backed up
- But, starvation is possible
  - e.g., if the rule is that Westbound cars always go first
- Note
  - Most OSes do not prevent or deal with deadlocks

Deadlock: One-lane Bridge



- Deadlock vs. Starvation
  - Starvation = Indefinitely postponed
    - 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

Addressing Deadlock

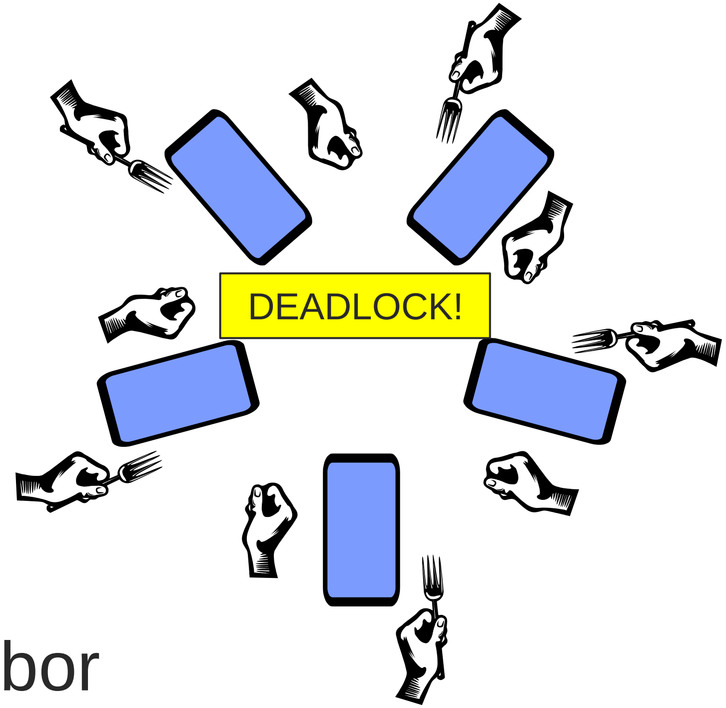
- Prevention
  - Design the system so that deadlock is impossible
- Avoidance
  - Construct a model of system states, then choose a strategy that, when resources are assigned to processes, will not allow the system to go to a deadlock state
- Detection & Recovery
  - Check for deadlock (periodically or sporadically) and identify which processes and resources are involved
  - Recover by killing one of the deadlocked processes and releasing its resources
- Manual intervention
  - Have the operator reboot the machine if it seems too slow

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

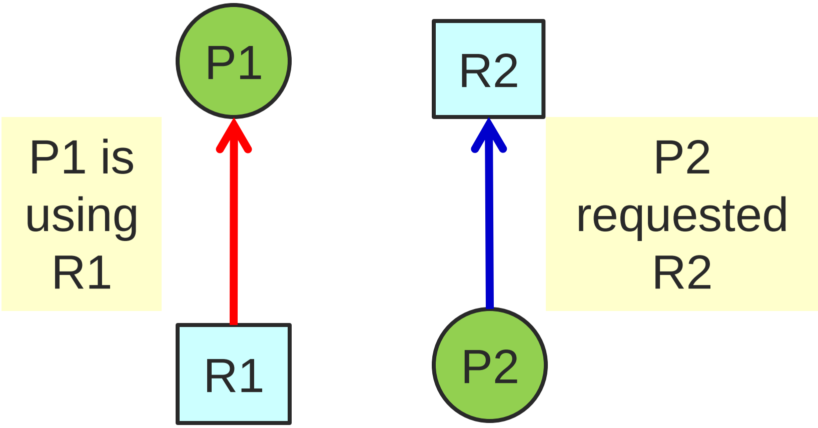
Dining Philosophers had it all

- Mutual exclusion
  - Exclusive use of forks
- Hold and wait condition
  - Hold 1 fork, wait for next
- No preemption condition
  - Cannot force another to undo their hold
- Circular wait condition
  - Each waits for next neighbor to put down fork



This is the best one to tackle

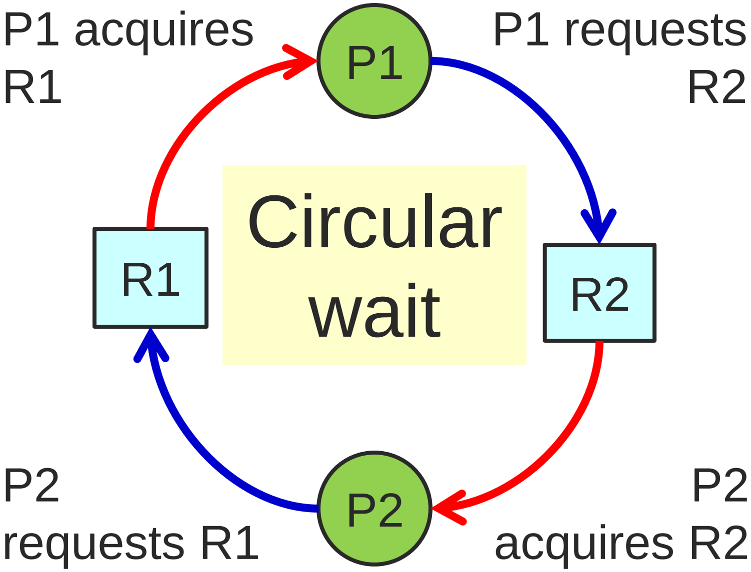
Formalizing circular wait: Resource allocation graphs



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

Resource allocation graphs

- Nodes
  - Circle: Processes
  - Square: Resources



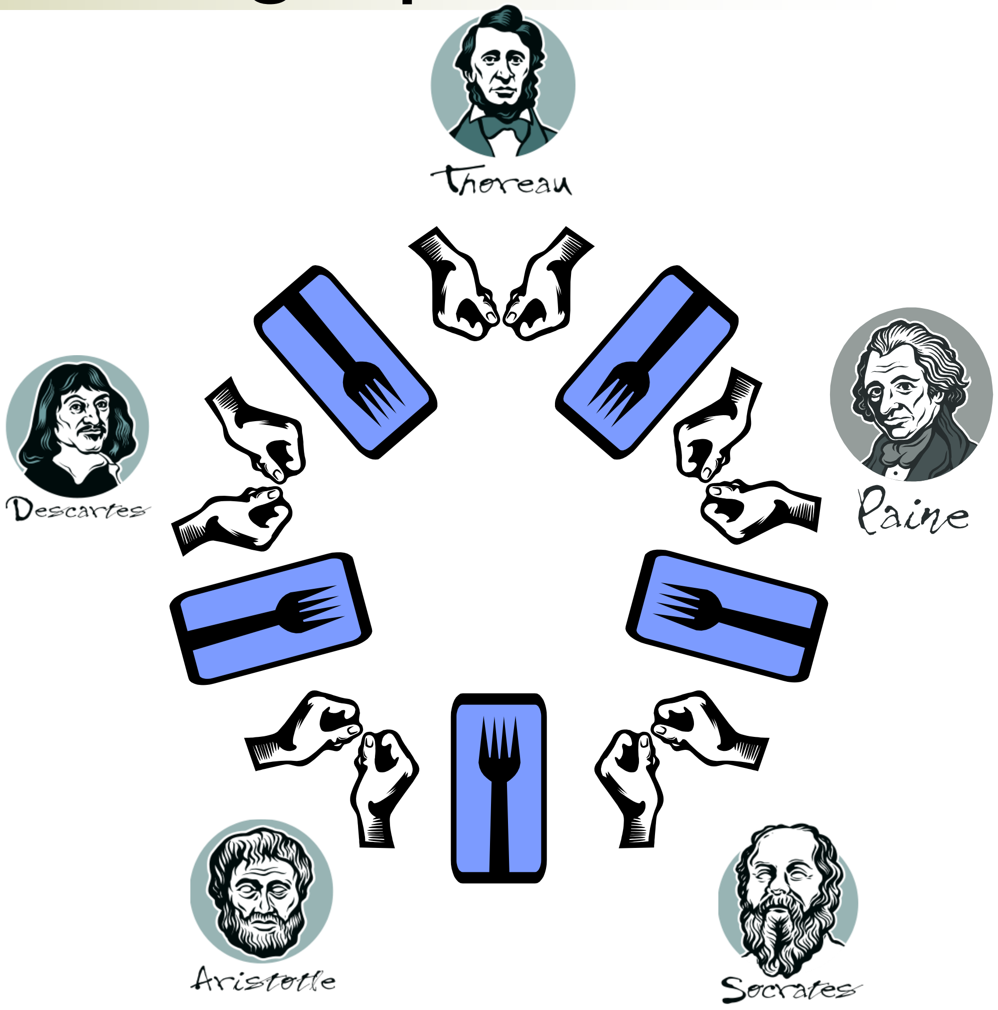
- Deadlock
  - Processes P1 and P2 are in deadlock over resources R1 and r2

Dining Philosophers resource allocation graph

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

```
# define N 5

void philosopher (int i) {
   while (TRUE) {
      think();
      take_fork(i);
      take_fork((i+1)%N);
      eat(); /* yummy */
      put_fork(i);
      put_fork((i+1)%N);
  }
}
```



Dining Philosophers resource allocation graph

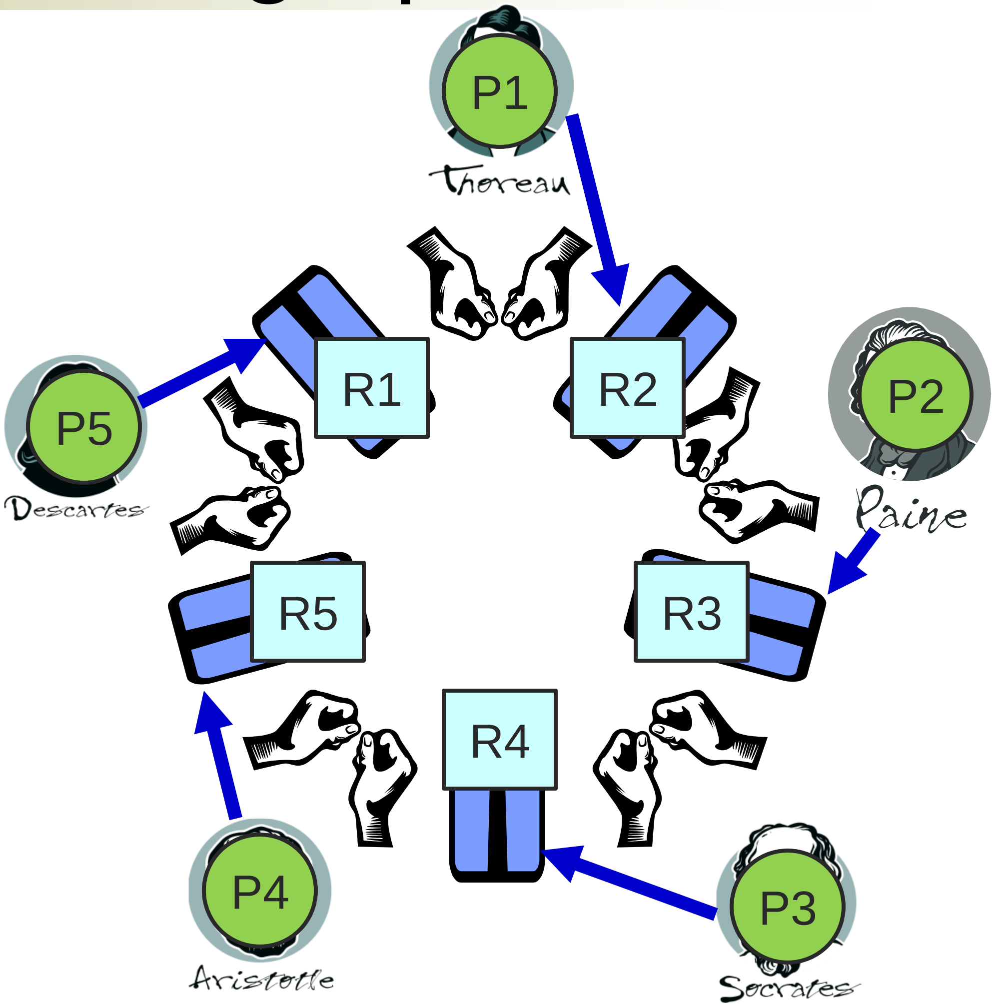
If we use the trivial broken "solution"...

One node per philosopher

One node per fork

⇒ Everyone tries to pick up left fork

⇒ Request edges



Dining Philosophers resource allocation graph

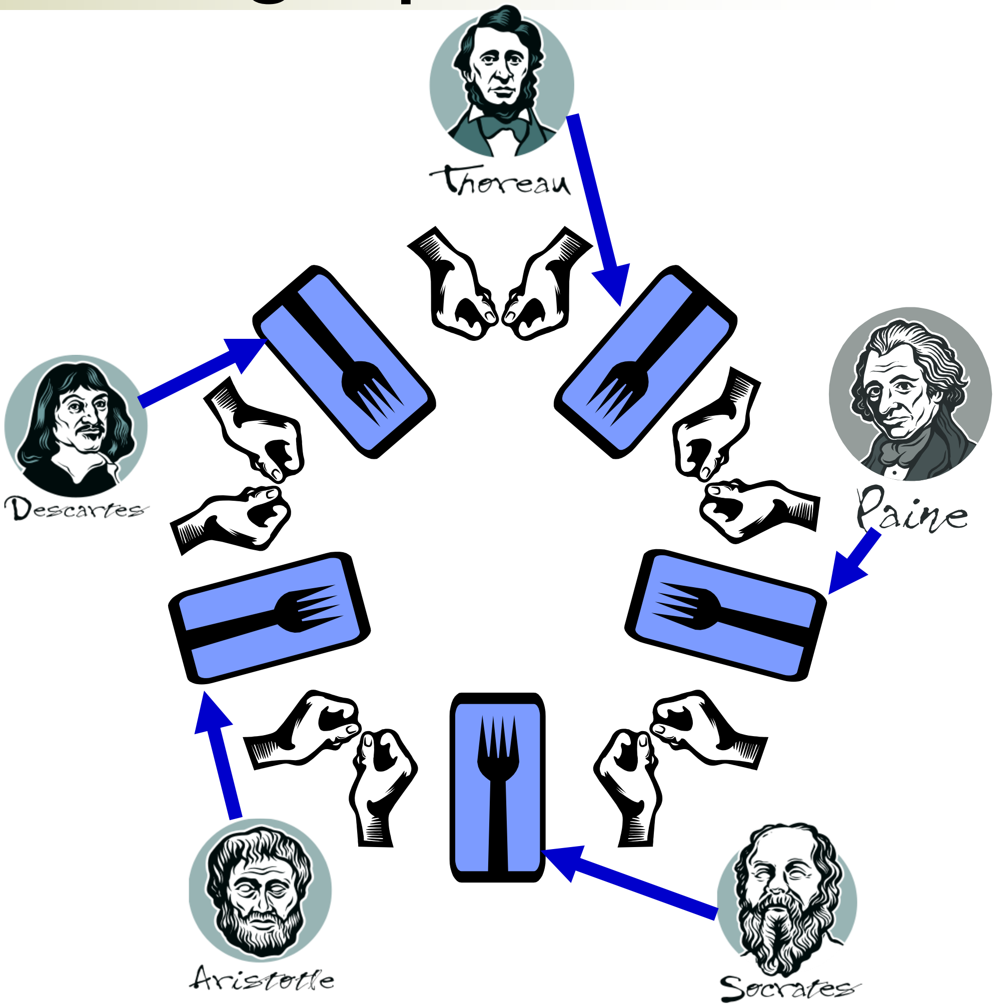
If we use the trivial broken "solution"...

One node per philosopher

One node per fork

⇒ Everyone tries to pick up left fork

⇒ Everyone succeeds



Dining Philosophers resource allocation graph

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

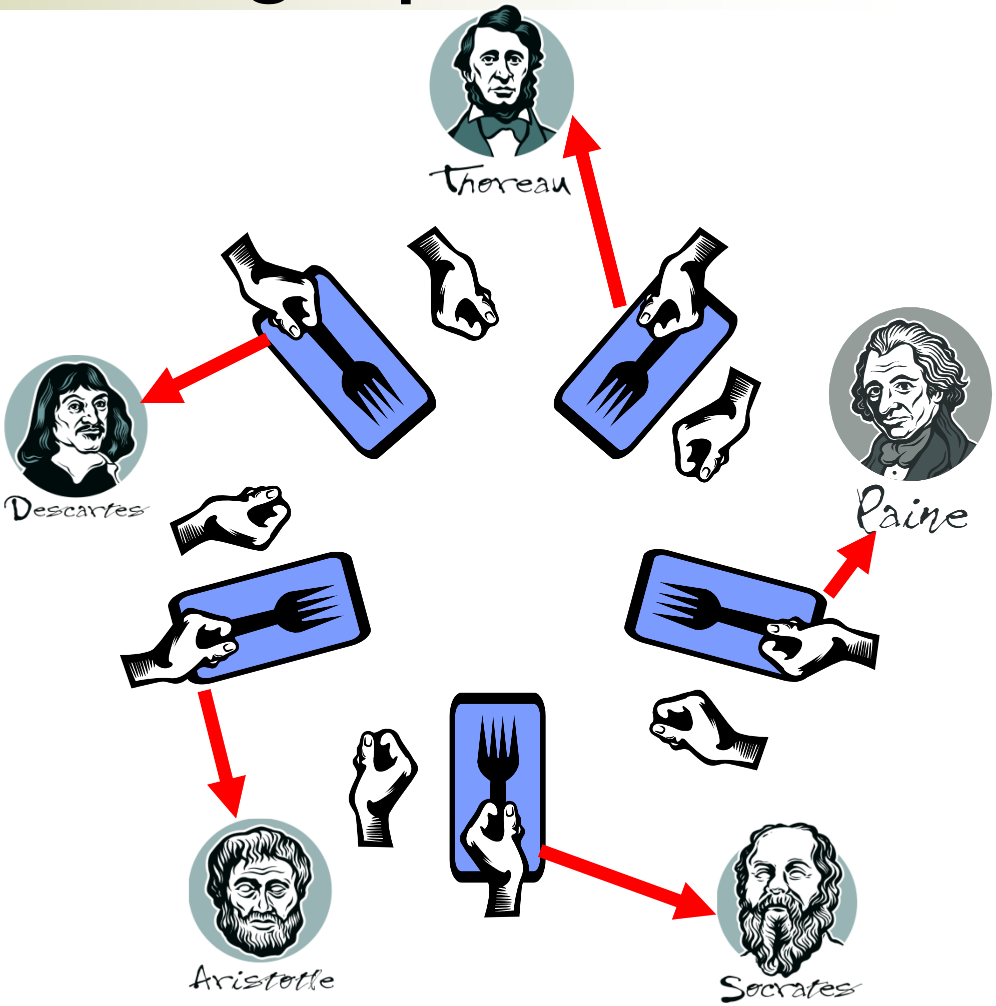
One node per philosopher

One node per fork

⇒ Everyone tries to pick up left fork

⇒ Everyone succeeds

⇒ Assignment edges



Dining Philosophers resource allocation graph

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

One node per philosopher

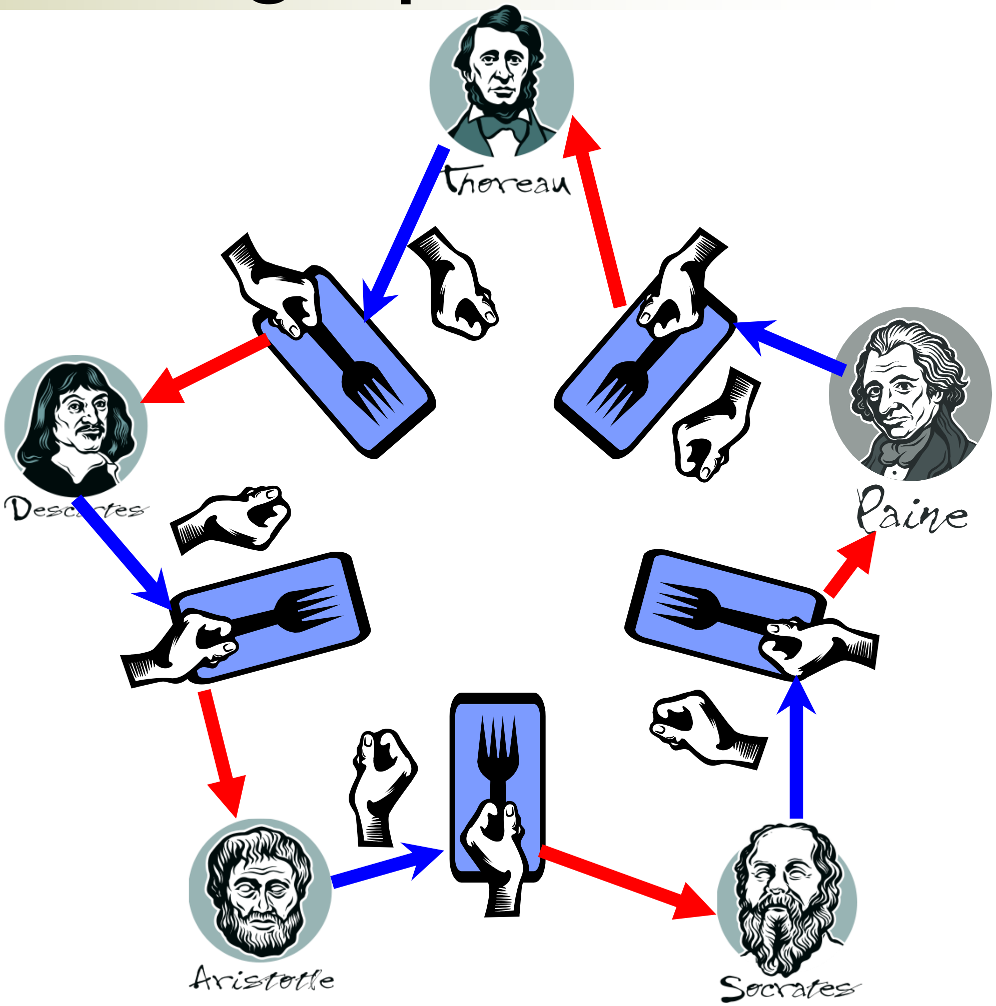
One node per fork

⇒ Everyone tries to pick up left fork

⇒ Everyone succeeds

⇒ Everyone tries to pick up right fork

⇒ Request edges



Dining Philosophers resource allocation graph

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

One node per philosopher

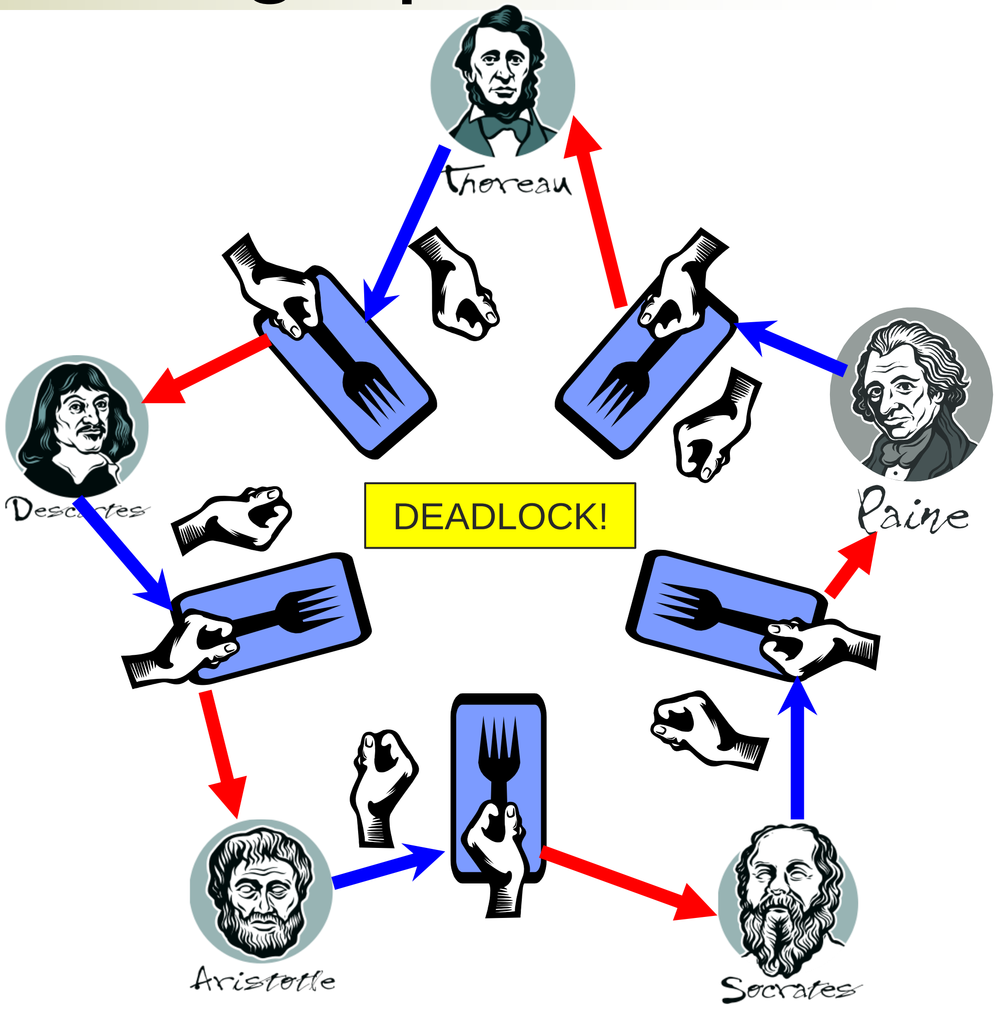
One node per fork

⇒Everyone tries to pick up left fork

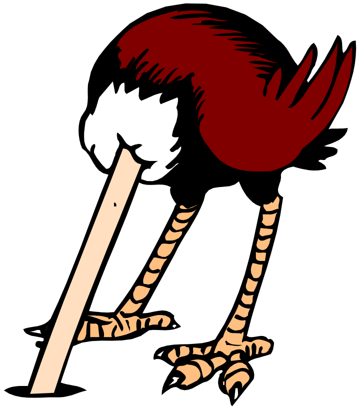
⇒ Everyone succeeds

⇒Everyone tries to pick up right fork

⇒ Cycle = deadlock



Default Solution: Be an Ostrich



- Approach
  - Do nothing!
  - Deadlocked processes stay stuck
- Rationale
  - Keeps the common path faster and more reliable
  - Deadlock prevention, avoidance and detection/recovery are expensive
  - If deadlock is rare, is it worth the overhead?

Deadlock Prevention

- Goal 1: devise resource allocation rules that make circular wait impossible
  - Resources include mutex locks, semaphores, pages of memory, ...
  - ...but you can think about just mutex locks for now

- Goal 2: make sure useful behavior is still possible!
  - The rules will necessarily be conservative
    - Rule out some behavior that would not cause deadlock
  - But they shouldn't be to be too conservative
    - We still need to get useful work done

Deadlock Prevention

- Prevent any one of the 4 conditions
  - Mutual exclusion
  - Hold-and-wait
  - No preemption
  - Circular wait

Mutual Exclusion

- Processes claim exclusive control of the resources they require
- How to break it?

Mutual Exclusion

- Processes claim exclusive control of the resources they require
- How to break it?
  - Non-exclusive access only
    - Read-only access
  - Probably not an option for most scenarios
    - But be smart and try to use shared resources wisely
  - Battle won!
    - War lost
    - Very bad at Goal #2

Hold and Wait Condition

- Processes hold resources already allocated to them while waiting for additional resources
- How to break it?

Hold and Wait Condition

- Processes hold resources already allocated to them while waiting for additional resources
- How to break it?
  - All at once
    - Force a process to request all resources it needs at one time
    - Get all or nothing
  - Release and try again
    - If a process needs to acquire a new resource, it must first release all resources it holds, then reacquire all it needs
  - Both
    - Inefficient
    - Potential of starvation

Hold and Wait Condition

- Processes hold resources already allocated to them while waiting for additional resources
- How to break it?
  - Only one
    - Process can only have one resource locked
- Result
  - No circular wait!

Hold and Wait Condition

- Processes hold resources already allocated to them while waiting for additional resources
- Result
  - No circular wait!
  - Very constraining (mediocre job on Goal #2)
    - Better than Rules #1 and #2, but...
    - Often need more than one resource
    - Hard to predict resource needs at the beginning
    - Releasing and re-requesting is inefficient, complicates programming, might lead to starvation

No Preemption Condition

- Resources cannot be taken from processes holding them until used to completion
- How to break it?

No Preemption Condition

- Resources cannot be taken from processes holding them until used to completion
- How to break it?
  - Let it all go
    - If a process holding some resources is denied a further request, that process must release its original resources
    - Inefficient!
  - Take it all away
    - If a process requests a resource that is held by another process, the OS may preempt the second process and force it to release its resources
    - Waste of CPU and other resources!

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

On success, pthread_mutex_trylock() returns 0. On error, one of the following values is returned:

EBUSY: The mutex is already locked.

No Preemption Condition

- Resources cannot be taken from processes holding them until used to completion
- Result
  - Breaks circular wait
    - Because we don’t have to wait
  - Reasonable strategy sometimes
    - e.g. if resource is memory: “preempt” = page to disk
  - Not so convenient for synchronization resources
    - e.g., locks in multithreaded application
    - What if current owner is in the middle of a critical section updating pointers? Data structures might be left in inconsistent state!

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
- How to break it?

Circular Wait Condition

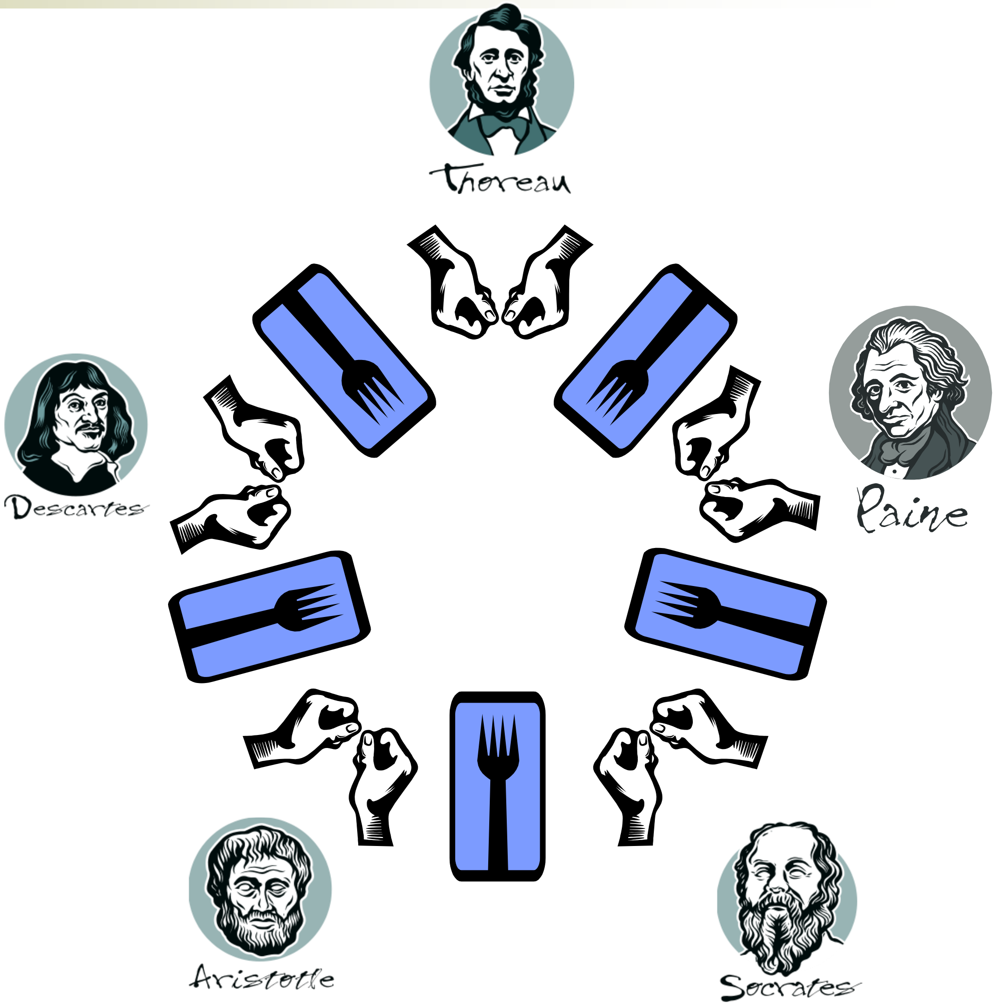
- A circular chain of processes exists in which each process holds one or more resources that are requested by the next process in the chain
- How to break it?
  - Guarantee no cycles
    - Allow processes to access resources only in increasing order of resource id
    - Not really fair or necessarily efficient …

Dining Philosophers solution with numbered resources

Back to the trivial broken “solution”...

```
# define N 5

void philosopher (int i) {
   while (TRUE) {
      think();
      take_fork(i);
      take_fork((i+1)%N);
      eat(); /* yummy */
      put_fork(i);
      put_fork((i+1)%N);
  }
}
```

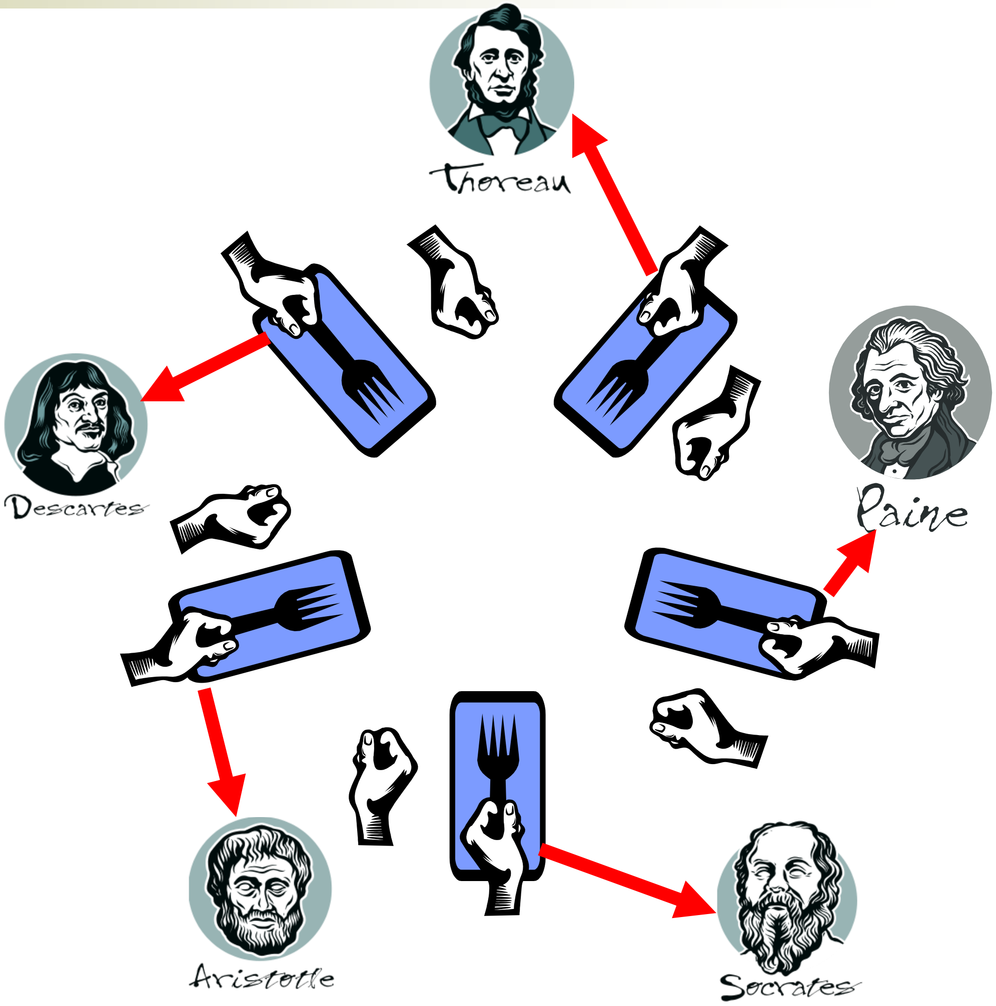


Dining Philosophers solution with numbered resources

Back to the trivial broken "solution"...

```
# define N 5

void philosopher (int i) {
   while (TRUE) {
       think();
       take_fork(i);
       take_fork((i+1)%N);
       eat(); /* yummy */
       put_fork(i);
       put_fork((i+1)%N);
  }
}
```

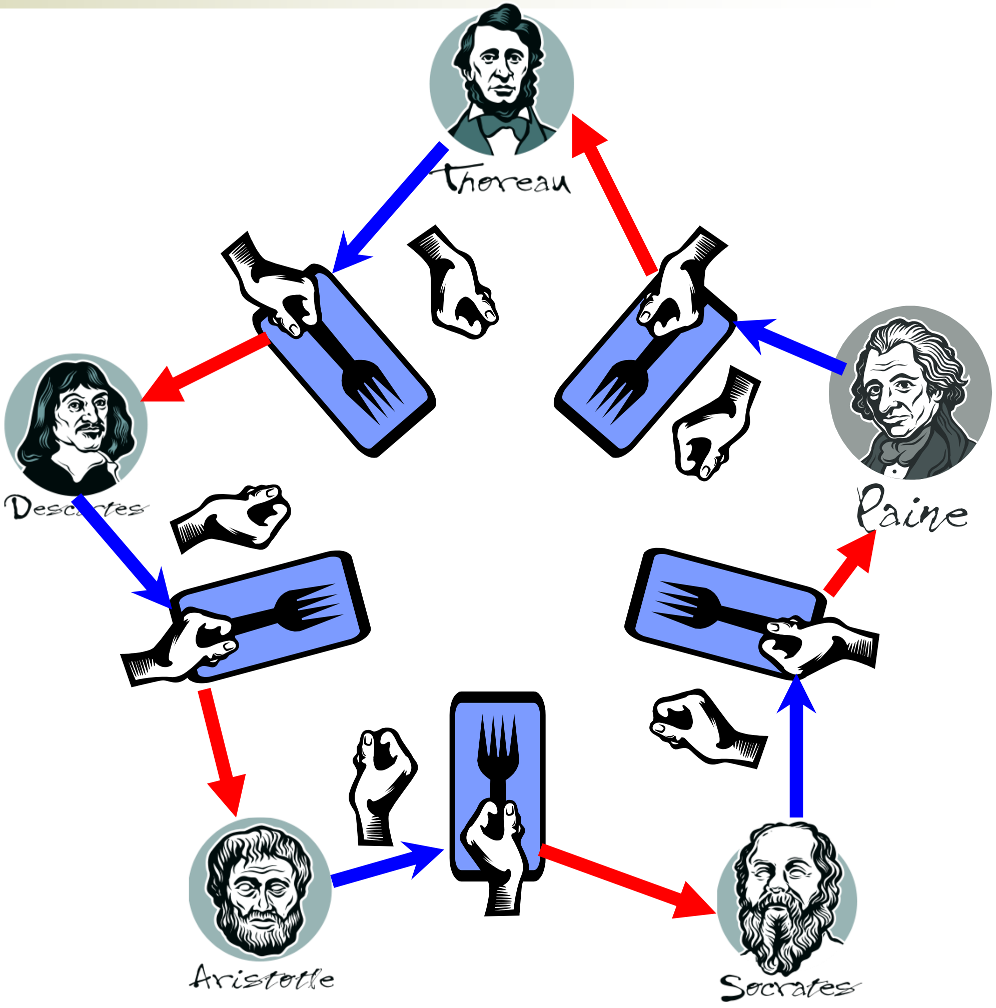


Dining Philosophers solution with numbered resources

Back to the trivial broken "solution"...

```
# define N 5

void philosopher (int i) {
   while (TRUE) {
      think();
      take_fork(i);
      take_fork((i+1)%N);
      eat(); /* yummy */
      put_fork(i);
      put_fork((i+1)%N);
   }
}
```



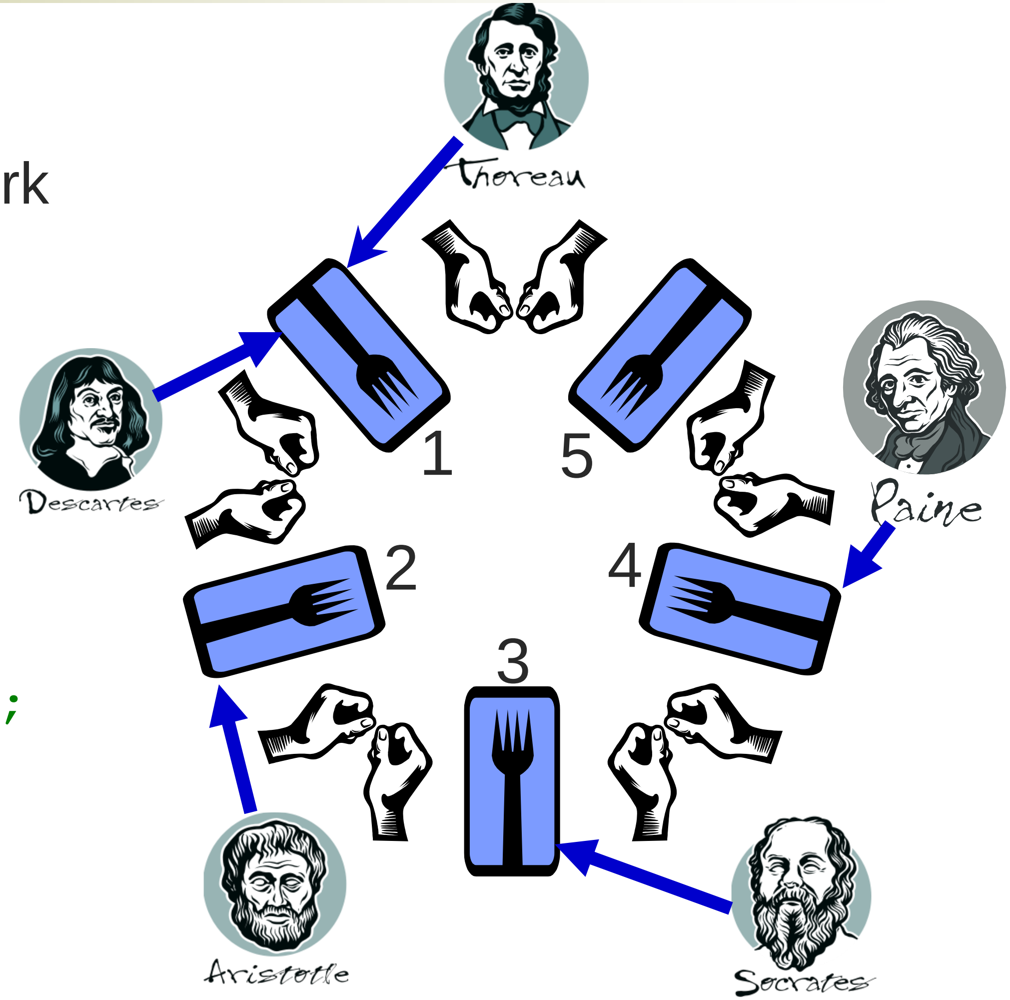
Dining Philosophers solution with numbered resources

Instead, number resources...

First request lower numbered fork

```
# define N 5

void philosopher (int i) {
    while (TRUE) {
        think();
        take_fork(LOWER(i));
        take_fork(HIGHER(i));
        eat(); /* yummy */
        put_fork(LOWER(i));
        put_fork(HIGHER(i));
   }
}
```



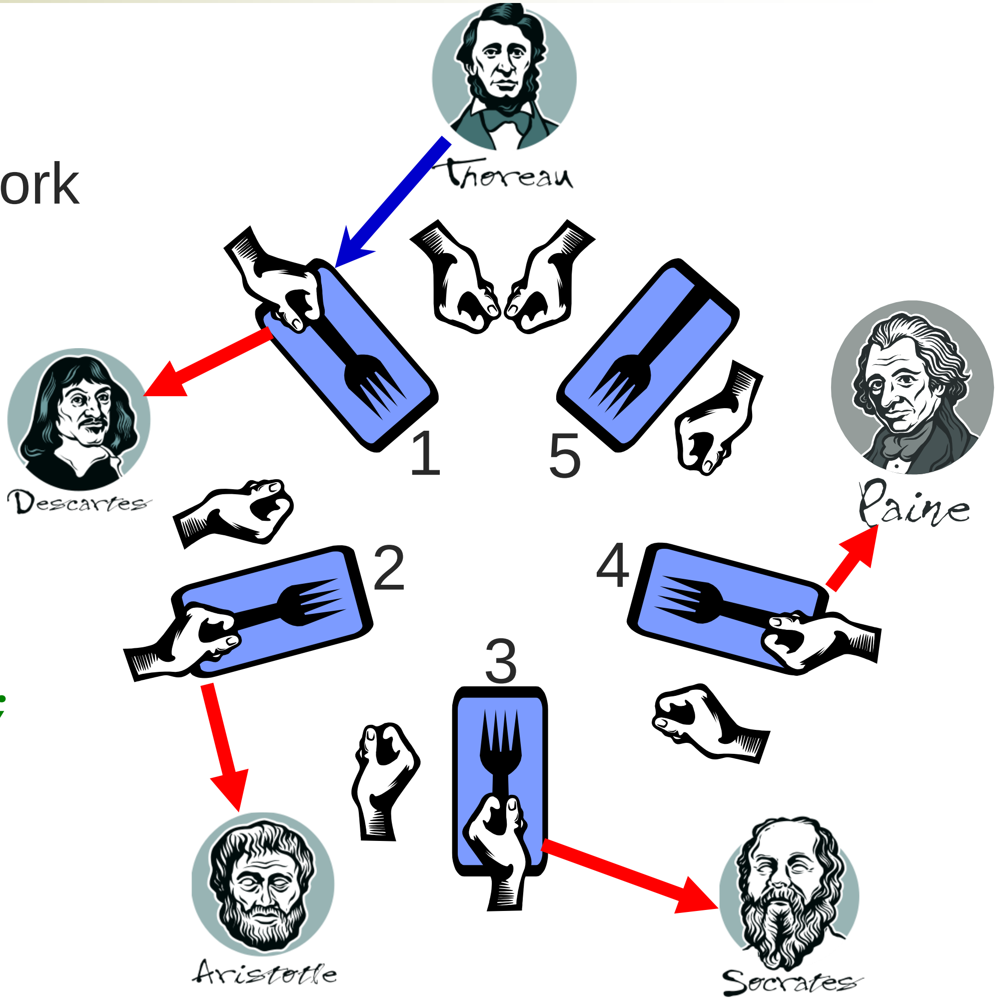
Dining Philosophers solution with numbered resources

Instead, number resources...

Then request higher numbered fork

```
# define N 5

void philosopher (int i) {
   while (TRUE) {
      think();
      take_fork(LOWER(i));
      take_fork(HIGHER(i));
      eat(); /* yummy */
      put_fork(LOWER(i));
      put_fork(HIGHER(i));
   }
}
```



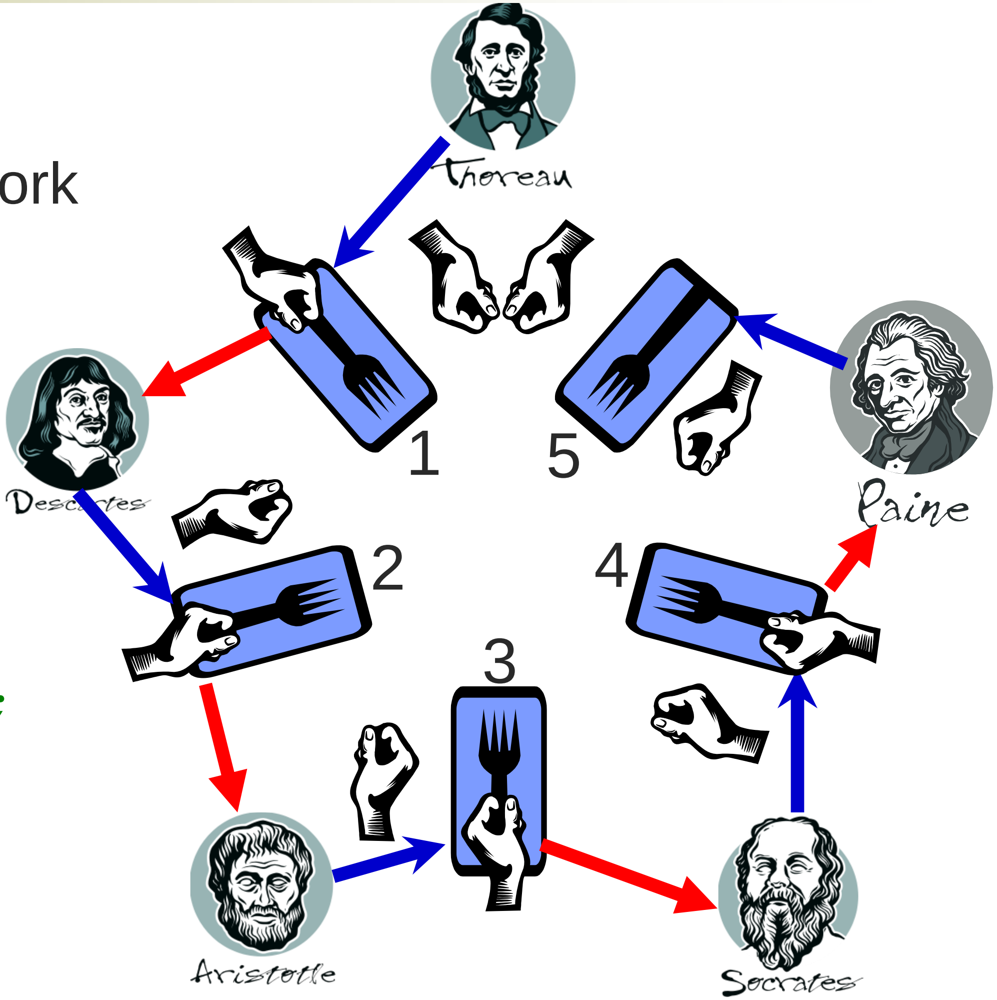
Dining Philosophers solution with numbered resources

Instead, number resources...

Then request higher numbered fork

```
# define N 5

void philosopher (int i) {
   while (TRUE) {
      think();
      take_fork(LOWER(i));
      take_fork(HIGHER(i));
      eat(); /* yummy */
      put_fork(LOWER(i));
      put_fork(HIGHER(i));
  }
}
```



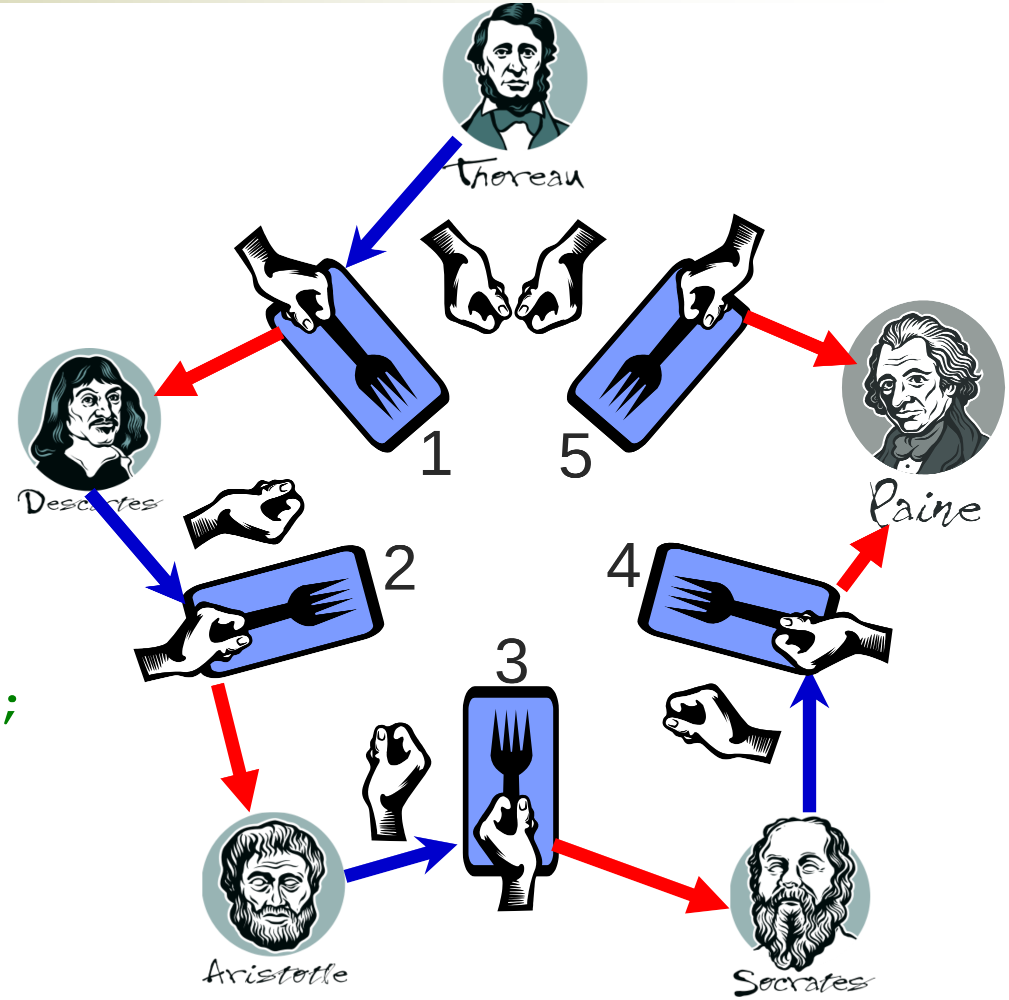
Dining Philosophers solution with numbered resources

Instead, number resources...

One philosopher can eat!

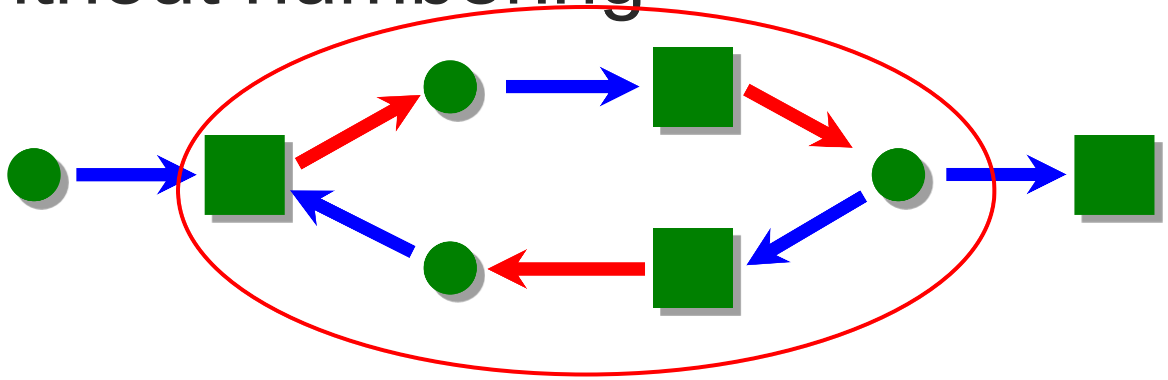
```
# define N 5

void philosopher (int i) {
   while (TRUE) {
      think();
      take_fork(LOWER(i));
      take_fork(HIGHER(i));
      eat(); /* yummy */
      put_fork(LOWER(i));
      put_fork(HIGHER(i));
   }
}
```



Ordered resource requests prevent deadlock

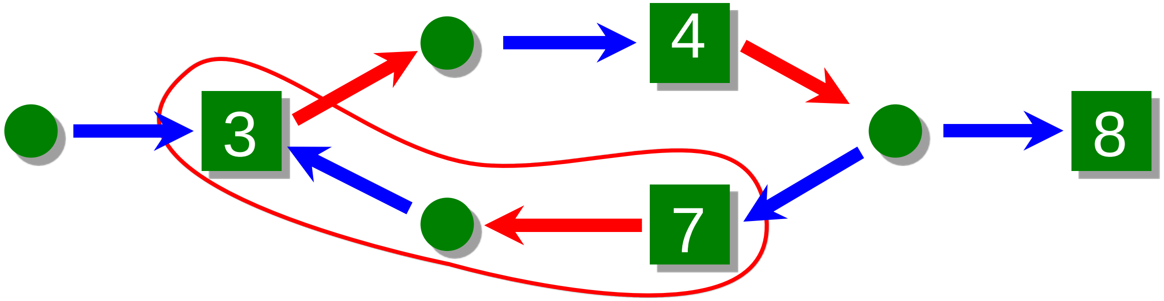
- Without numbering



Cycle!

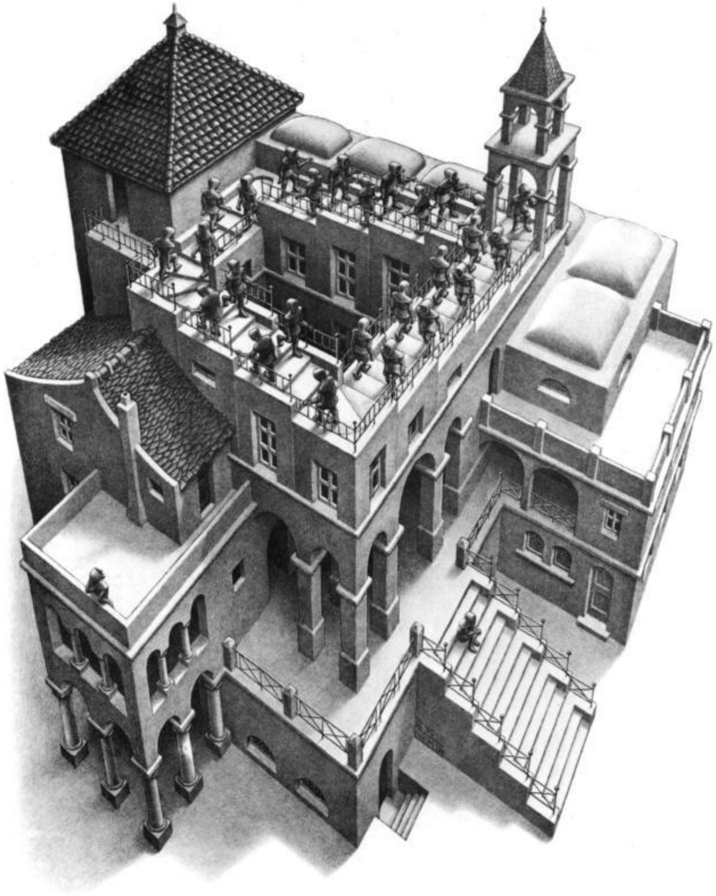
Ordered resource requests prevent deadlock

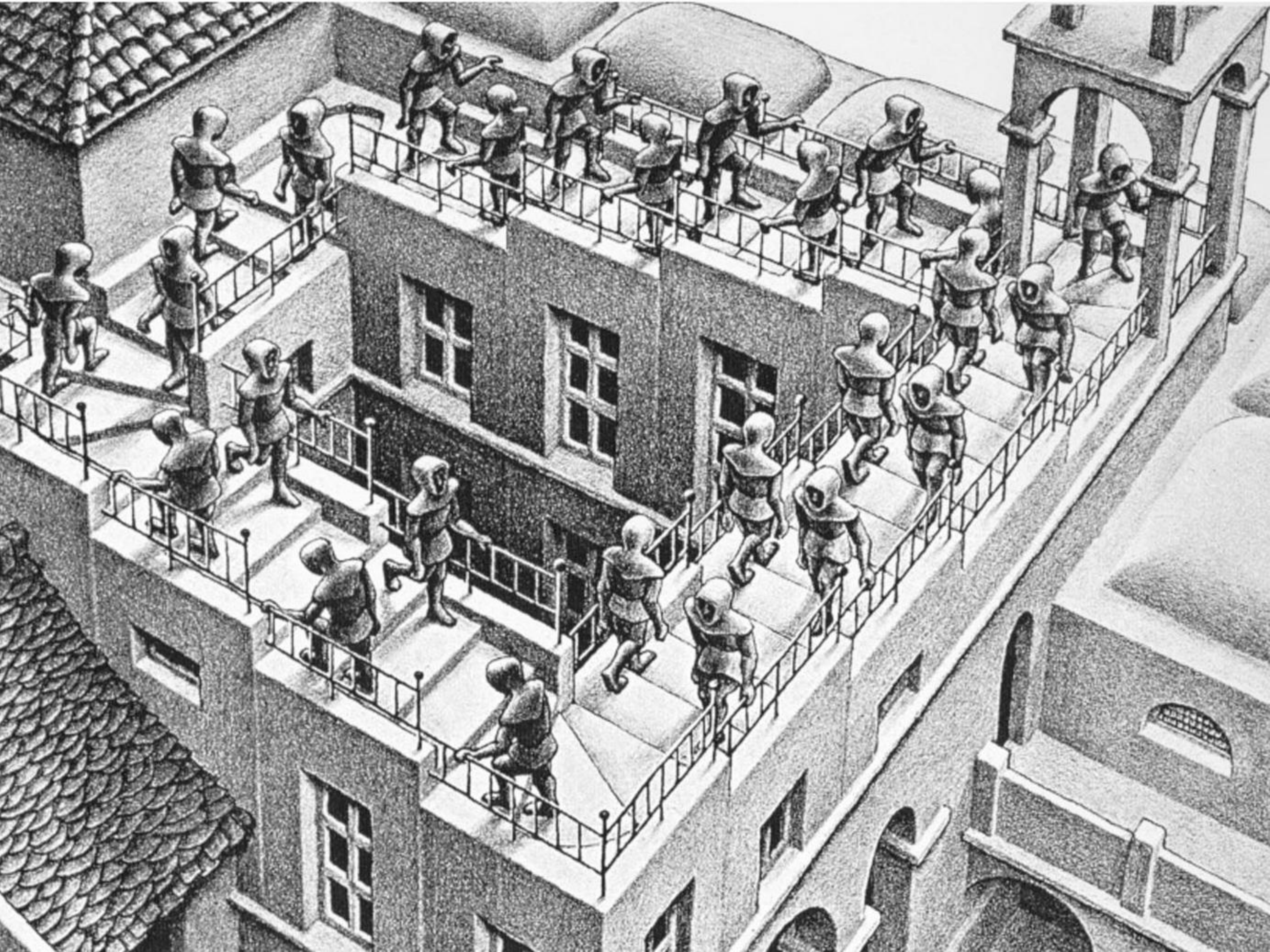
- With numbering



Contradiction:
Must have requested
3 first!

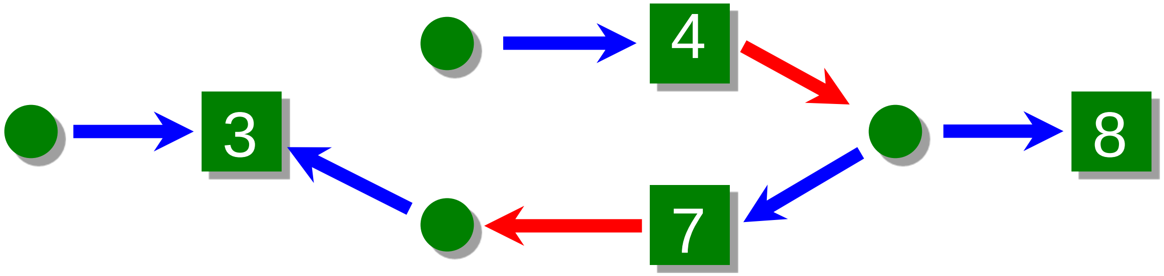
Proof by M.C. Escher





Are we always in trouble without ordering resources?

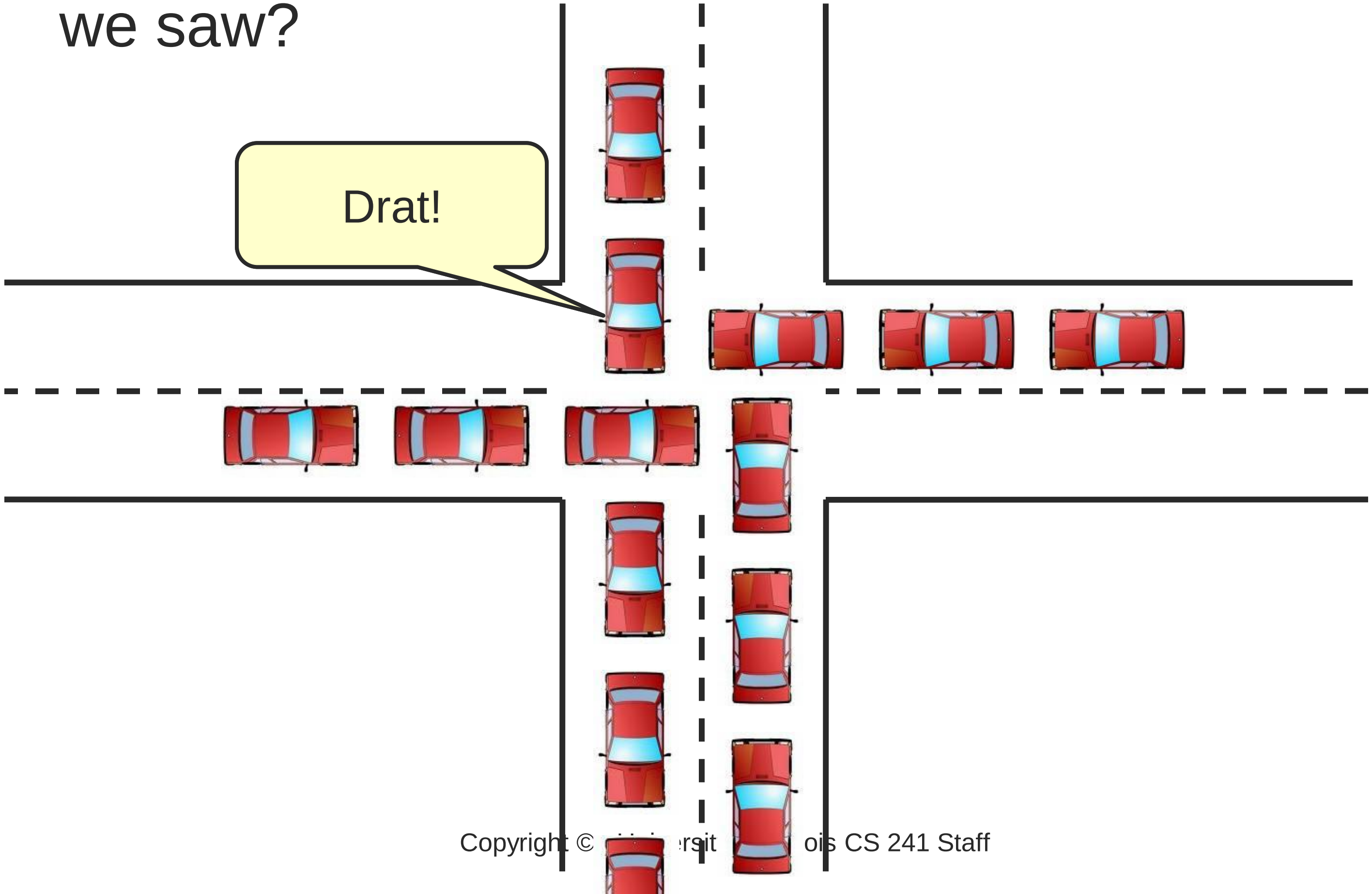
- Not always



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

Q: What's the rule of the road?

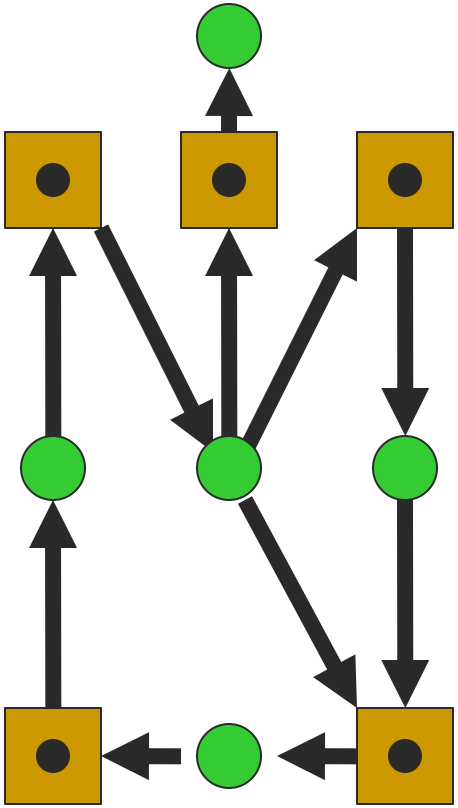
- What's the law? Does it resemble one of the rules we saw?



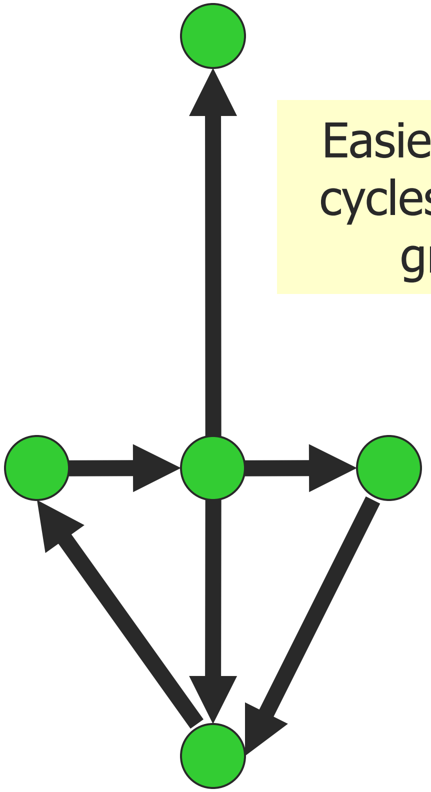
Deadlock Detection

- 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

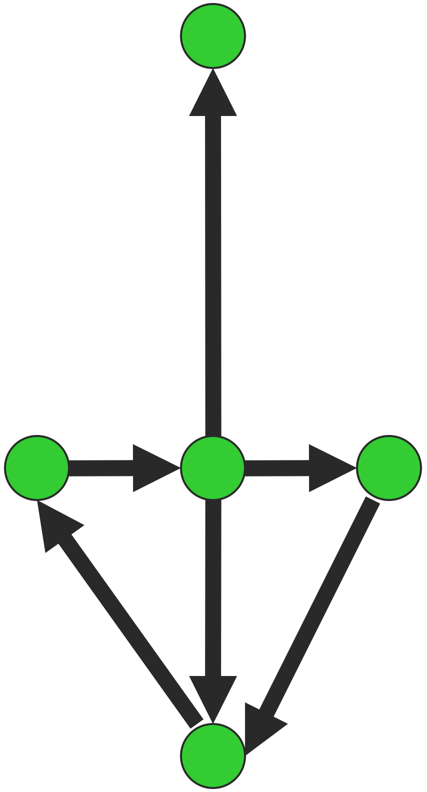


Easier to find cycles on this graph

Corresponding Wait For Graph

Deadlock Recovery

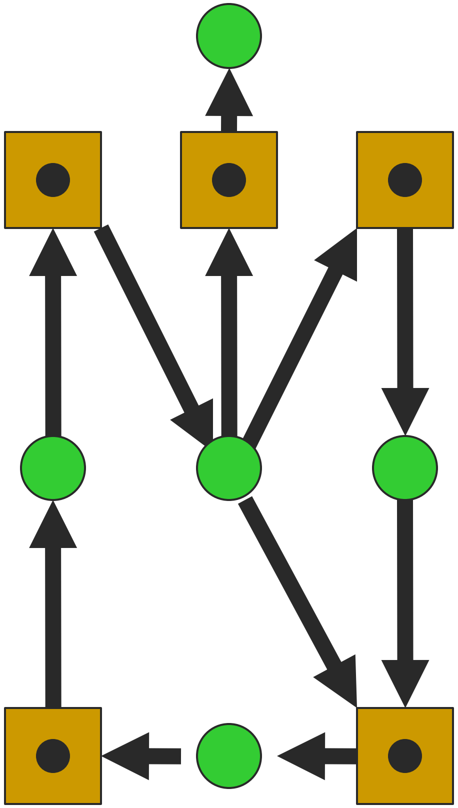
- Get rid of the cycles in the wait for graph
- How many cycles are there?



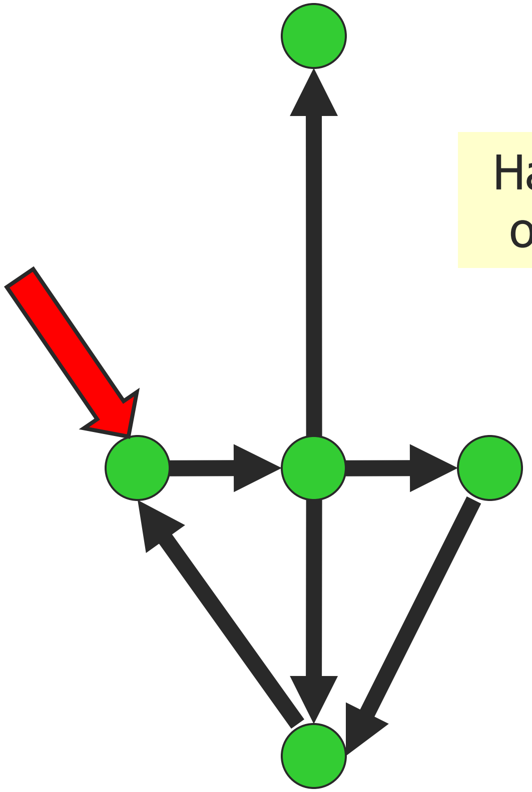
Deadlock Recovery

- Options
  - Kill all deadlocked processes and release resources
  - Kill one deadlocked process at a time and release its resources
  - Steal one resource at a time
  - Rollback all or one of the processes to a checkpoint that occurred before they requested any resources
    - Difficult to prevent indefinite postponement

Deadlock Recovery



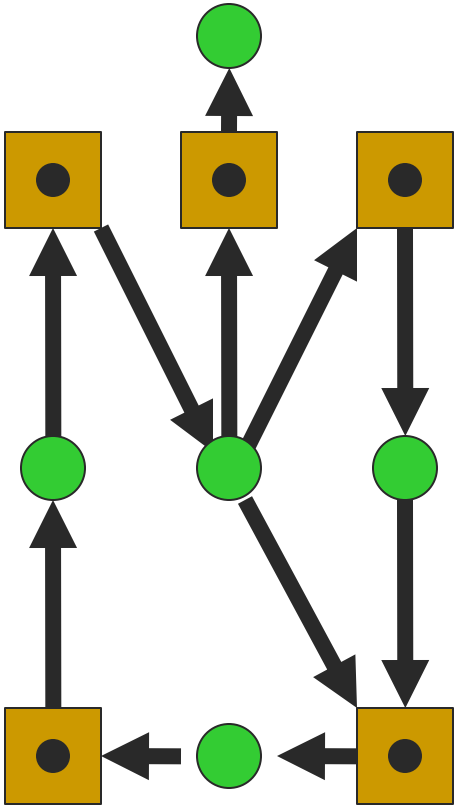
Resource
Allocation Graph



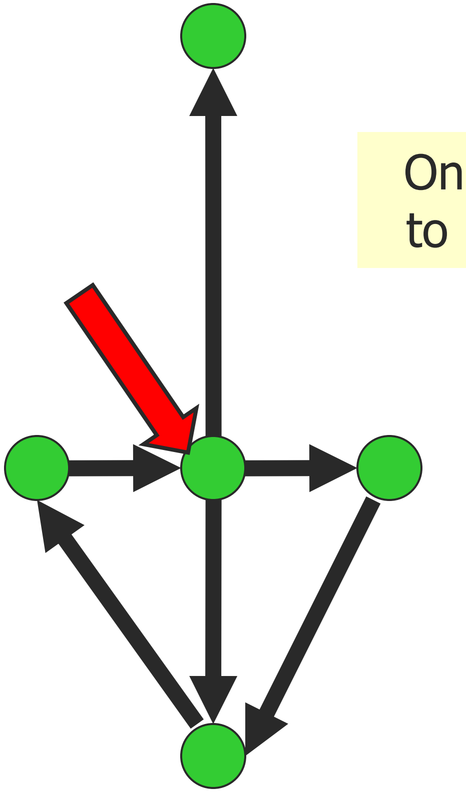
Have to kill
one more

Corresponding Wait
For Graph

Deadlock Recovery



Resource
Allocation Graph



Only have
to kill one

Corresponding Wait
For Graph

Deadlock Recovery: Process Termination

- How should the aborted process be chosen?
  - Process priority
  - Current computation time and time to completion
  - Amount of resources used by the process
  - Amount of resources needed by the process to complete
  - If this process is terminated, how many other processes will need to be terminated?
  - Is process interactive or batch?

Deadlock Recovery: Resource Preemption

- Selecting a victim
  - Minimize cost
- Rollback
  - Return to some safe state
  - Restart process for that state
- Challenge: Starvation
  - Same process may always be picked as victim
  - Fix: Include number of rollbacks in cost factor

Deadlock Avoidance

- Basic idea
  - Resource manager tries to see the worst case that could happen
  - It does not grant an incremental resource request to a process if this allocation might lead to deadlock

Deadlock Avoidance

- Approach
  - Define a model of system states (SAFE, UNSAFE)
  - Choose a strategy that guarantees that the system will not go to a deadlock state
- Multiple instance of each Resources
  - Requires the maximum number of each resource needed for each process
    - For each resource **`i`**, **`p.Max[i]`** = maximum number of instances of **`i`** that **`p`** can request

Safe vs. Unsafe

- Safe
  - Guarantee
    - There is some scheduling order in which every process can run to completion even if all of them suddenly and simultaneously request their maximum number of resources
  - From a safe state
    - The system can guarantee that all processes will finish
- Unsafe state: no such guarantee
  - A deadlock state is an unsafe state
  - An unsafe state may not be a deadlock state
  - Some process may be able to complete
- Overall
  - a conservative/pessimistic approach

How to Compute Safety

- Banker’s Algorithm (Dijkstra, 1965)
  - Each customer tells banker the maximum number of resources it needs, before it starts
  - Customer borrows resources from banker
  - Customer returns resources to banker
  - Banker only lends resources if the system will stay in a safe state after the loan