Deadlock

CS 241

March 19, 2014

University of Illinois

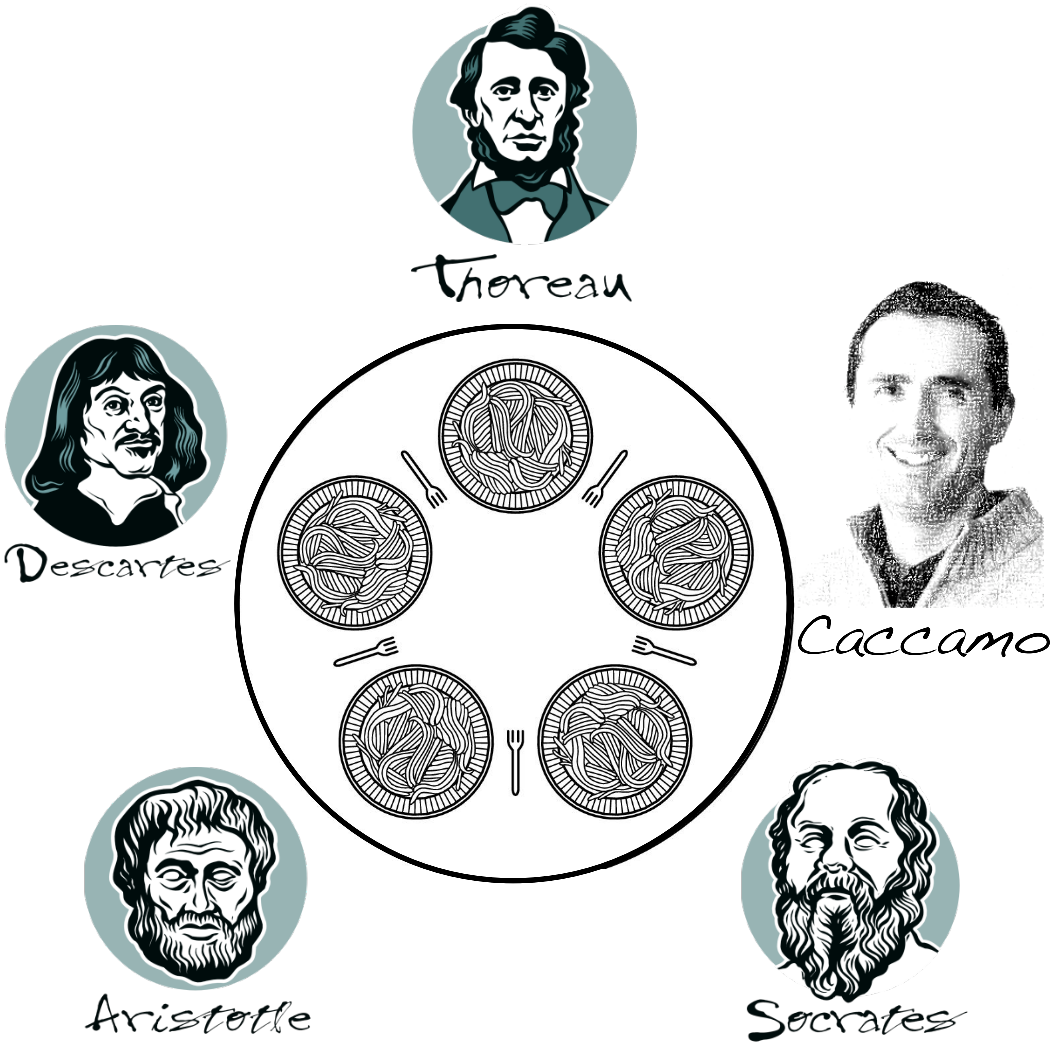
Slides adapted in part from material accompanying Bryant & O'Hallaron,
"Computer Systems: A Programmer's Perspective", 2/E

The Dining Philosophers Problem

Drinking Philosophers

[redacted]

Dining Philosophers



Dining Philosophers

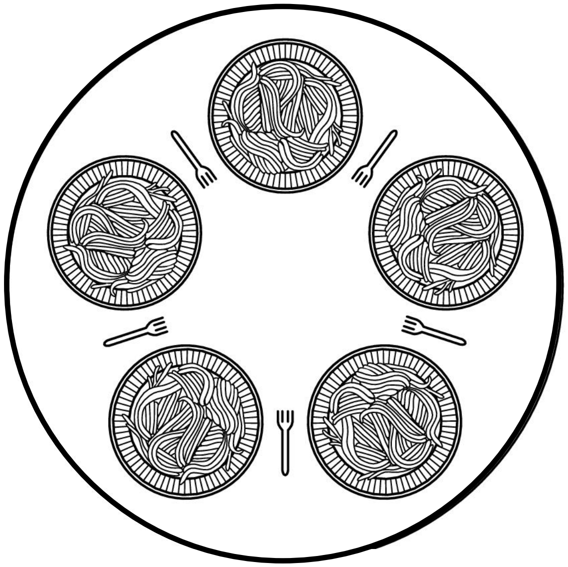
N philosophers and N forks

Philosophers eat, think

Eating needs 2 forks

Pick up one fork at a time

Each fork used by one person at a time



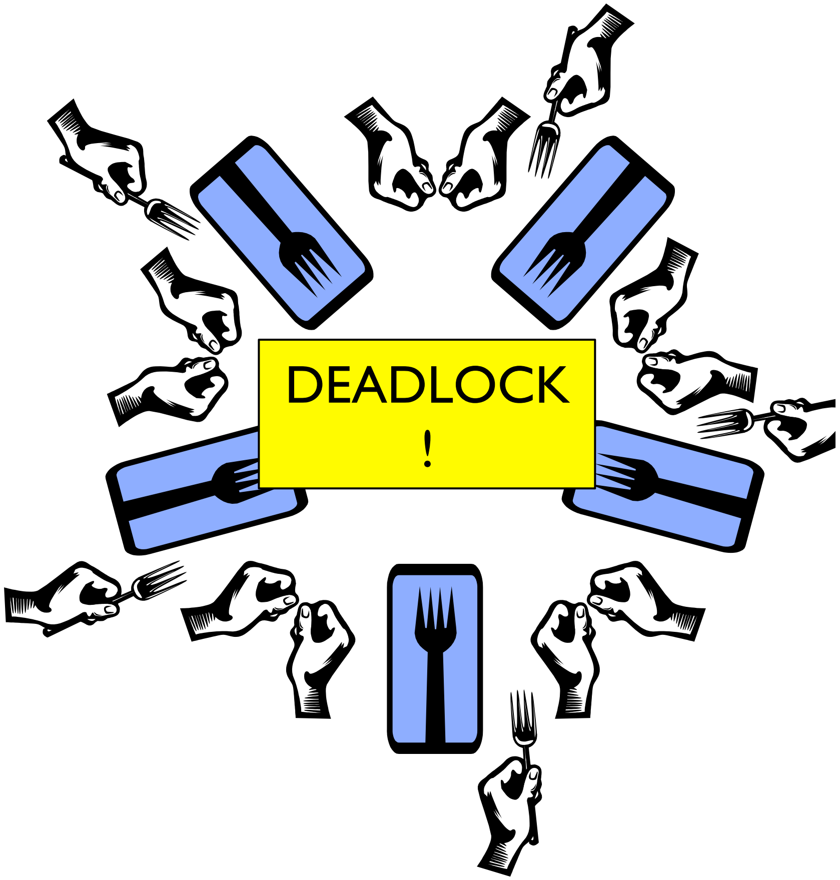
Dining Philosophers: Take 1

```
# define N 5

void philosopher (int i) {
   while (TRUE) {
      think();
      lock_fork(i);
      lock_fork((i+1)%N);

      eat(); /* yummy */

      unlock_fork(i);
      unlock_fork((i+1)%N);
  }
}
```

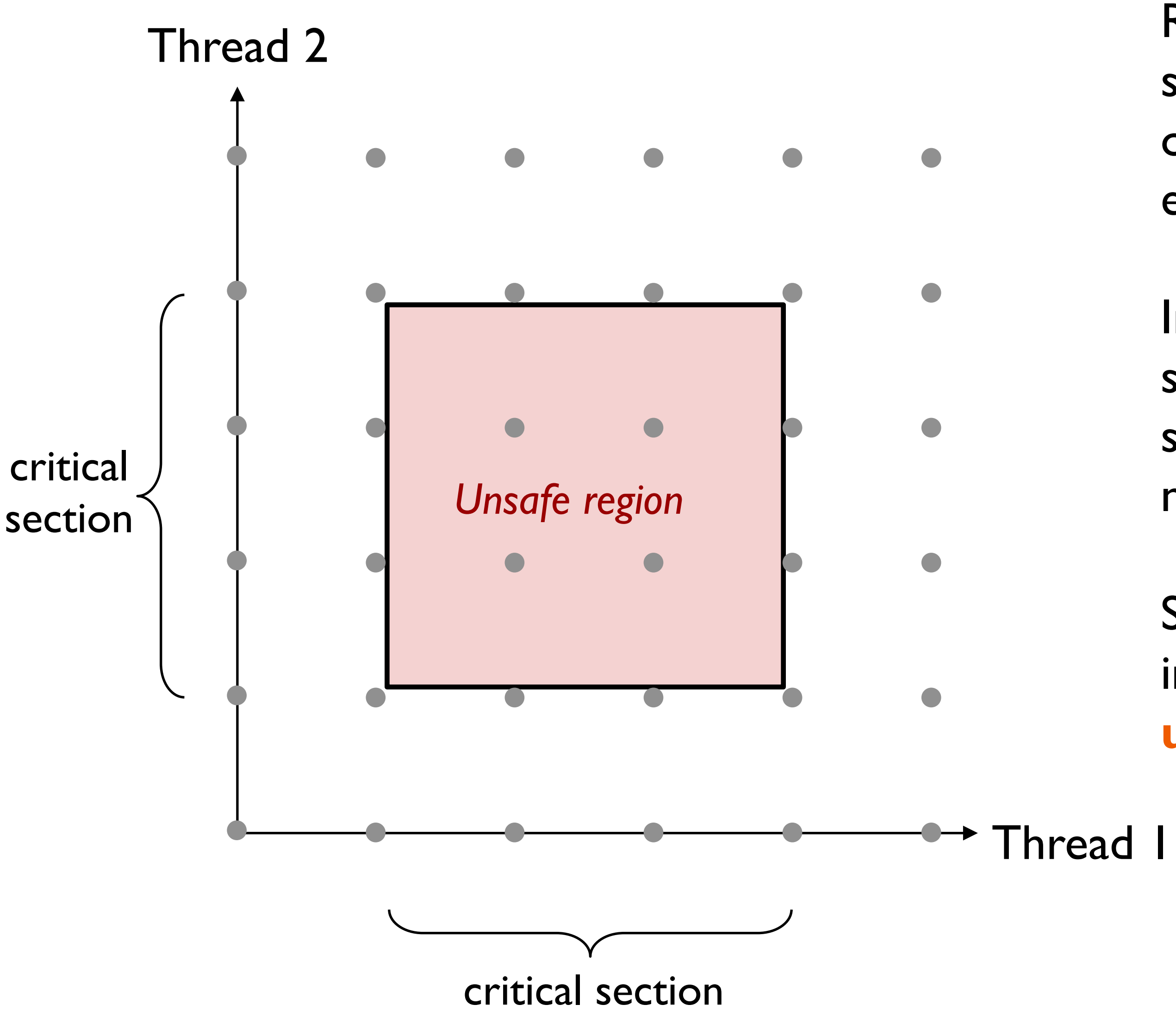
Does this work?

Dining Philosophers: Take 1

```
# define N 5

void philosopher (int i) {
   while (TRUE) {
      think();
      lock_fork(i);
      lock_fork((i+1)%N);

      eat(); /* yummy */

      unlock_fork(i);
      unlock_fork((i+1)%N);
   }
}
```



Progress diagram

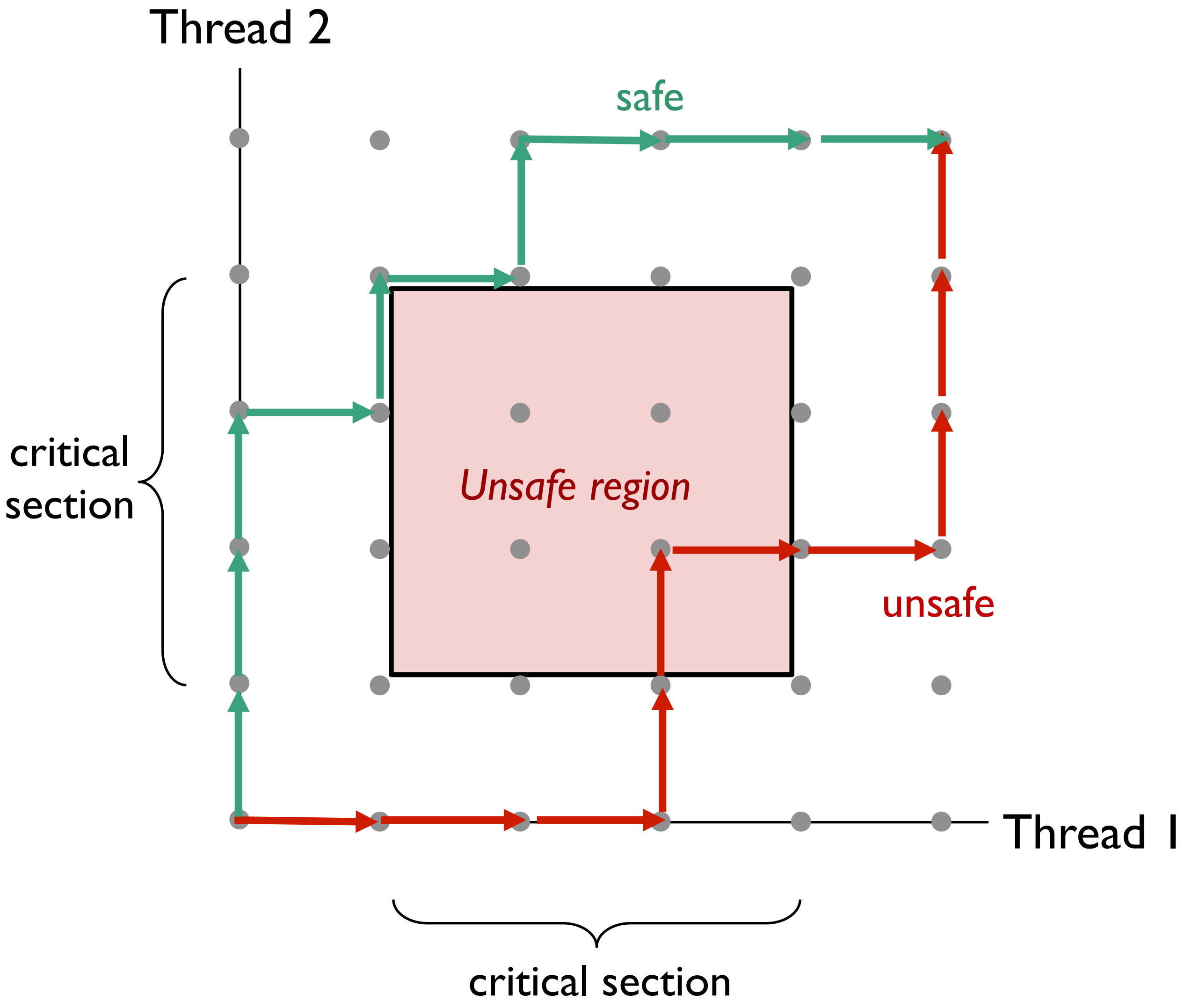


Represents state of system (position of each of the two threads in their executions)

Instructions in critical sections (wrt to some shared variable) should not be interleaved

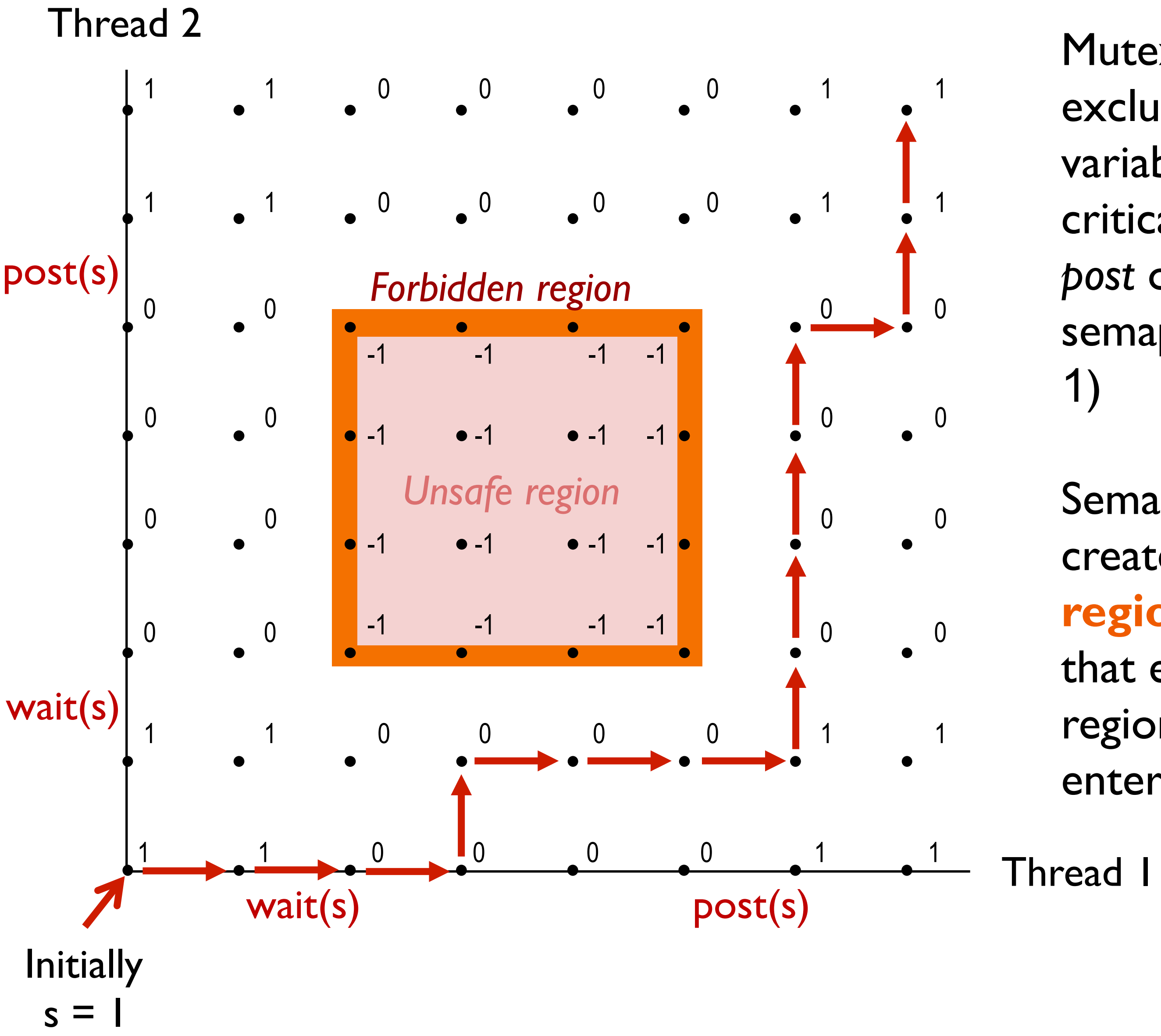
Sets of states where such interleaving occurs form **unsafe regions**

Progress diagram



But, any trajectory that goes up and to the right might occur...

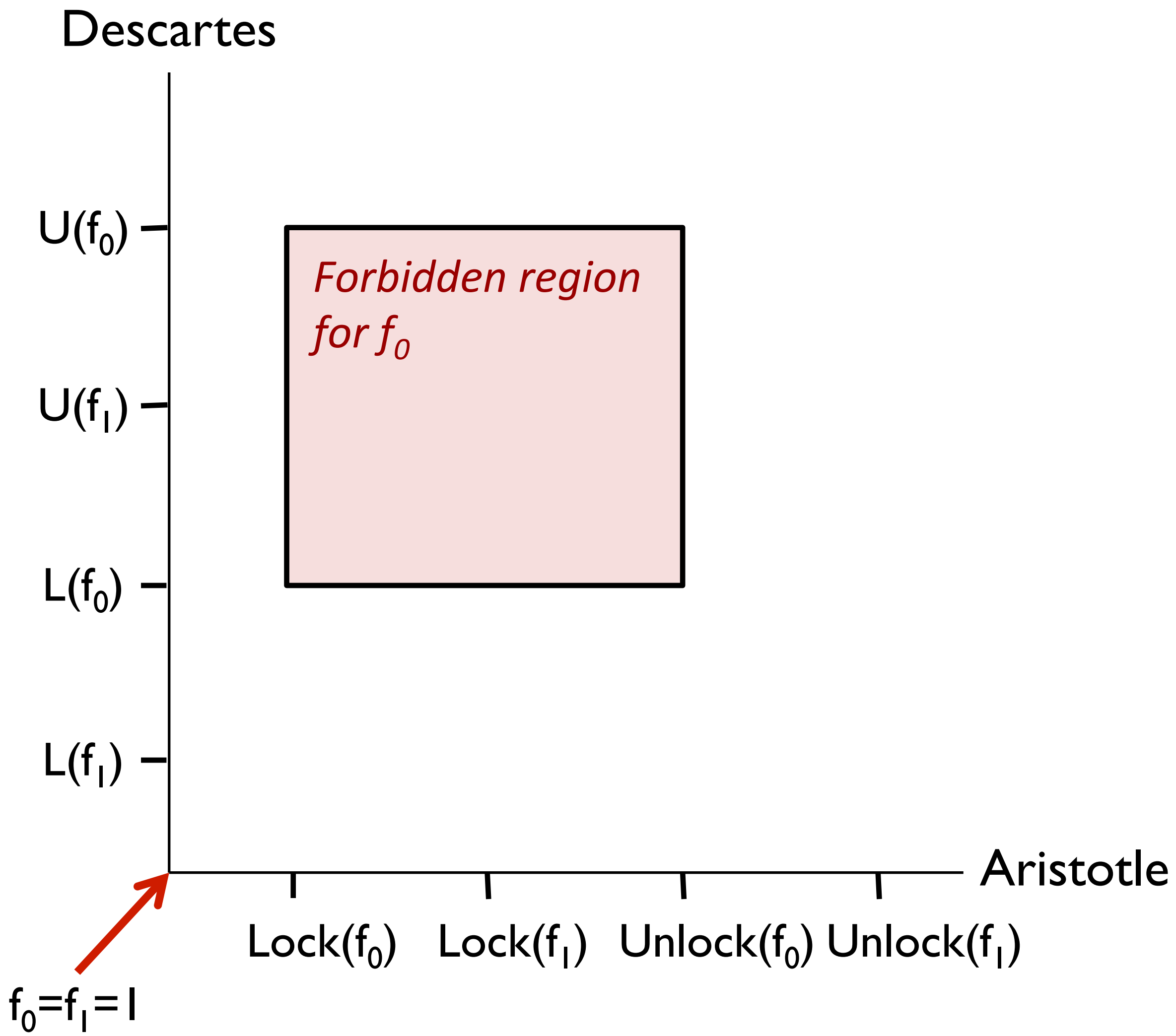
Reminder: process diagram



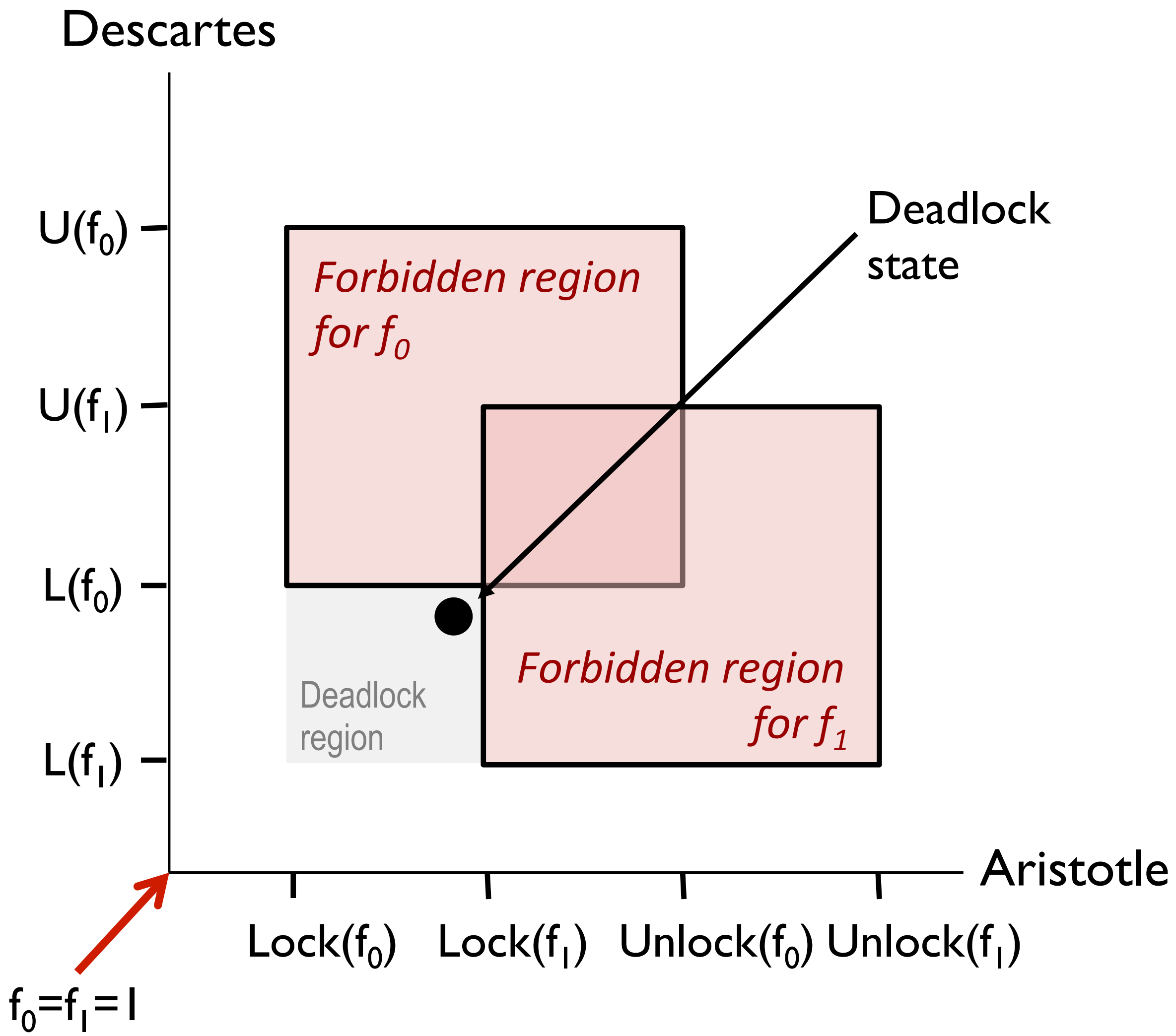
Mutexes provide mutually exclusive access to shared variable by surrounding critical section with *wait* and *post* operations on semaphore s (initially set to 1)

Semaphore invariant creates a **forbidden region** that encloses the unsafe region that must not be entered by any trajectory.

Two shared resources



Two shared resources



Any trajectory that enters the *deadlock region* will eventually reach the *deadlock state,* waiting for either f_0 or f_1 to become nonzero

Other trajectories luck out and skirt the deadlock region

Unfortunate fact: deadlock is often nondeterministic (race)

Deadlock: definition

There exists a cycle of processes such that each process cannot proceed until the next process takes some specific action.

Result: all processes in the cycle are stuck!

Example:

- P1 holds resource R1 & is waiting to acquire R2 before unlocking them
- P2 holds resource R2 & is waiting to acquire R1 before unlocking them

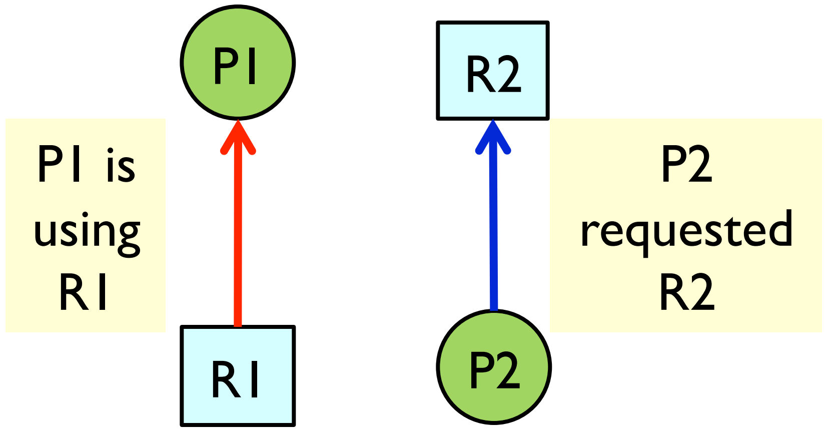
Resource allocation graphs

Nodes

- Circle: Processes
- Square: Resources

Edges

- From resource to process = resource assigned to process
- From process to resource = process requests (and is waiting for) resource



P1 is using R1

P2 requested R2

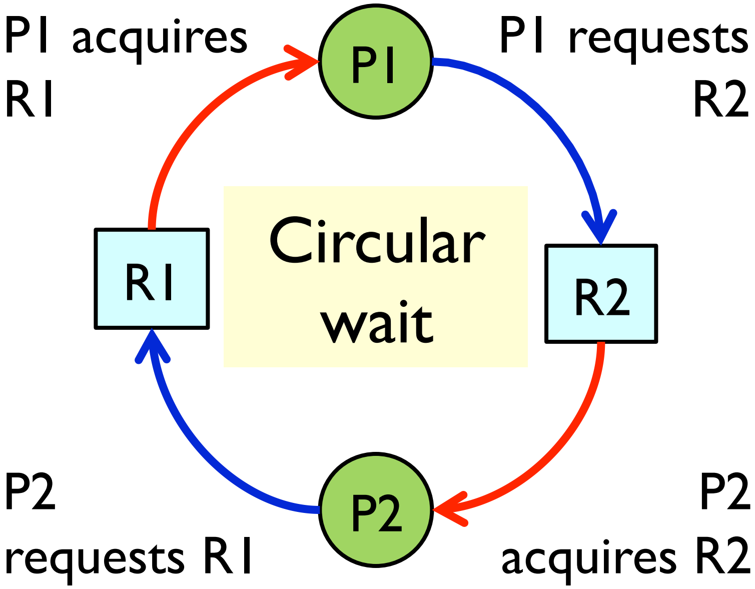
Resource allocation graphs

Nodes

- Circle: Processes
- Square: Resources

Deadlock

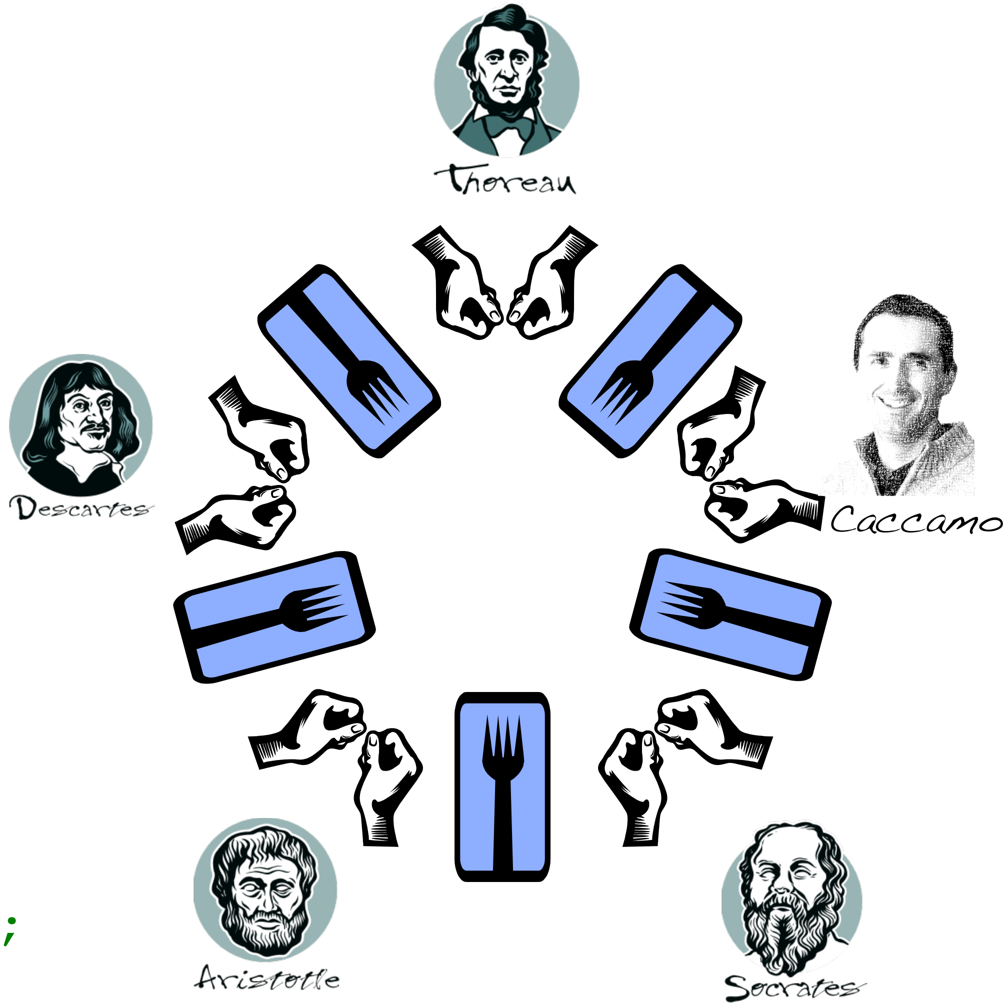
- Processes P1 and P2 are in deadlock over resources R1 and r2



Dining Philosophers resource allocation graph

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

```
# define N 5

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



Dining Philosophers resource allocation graph

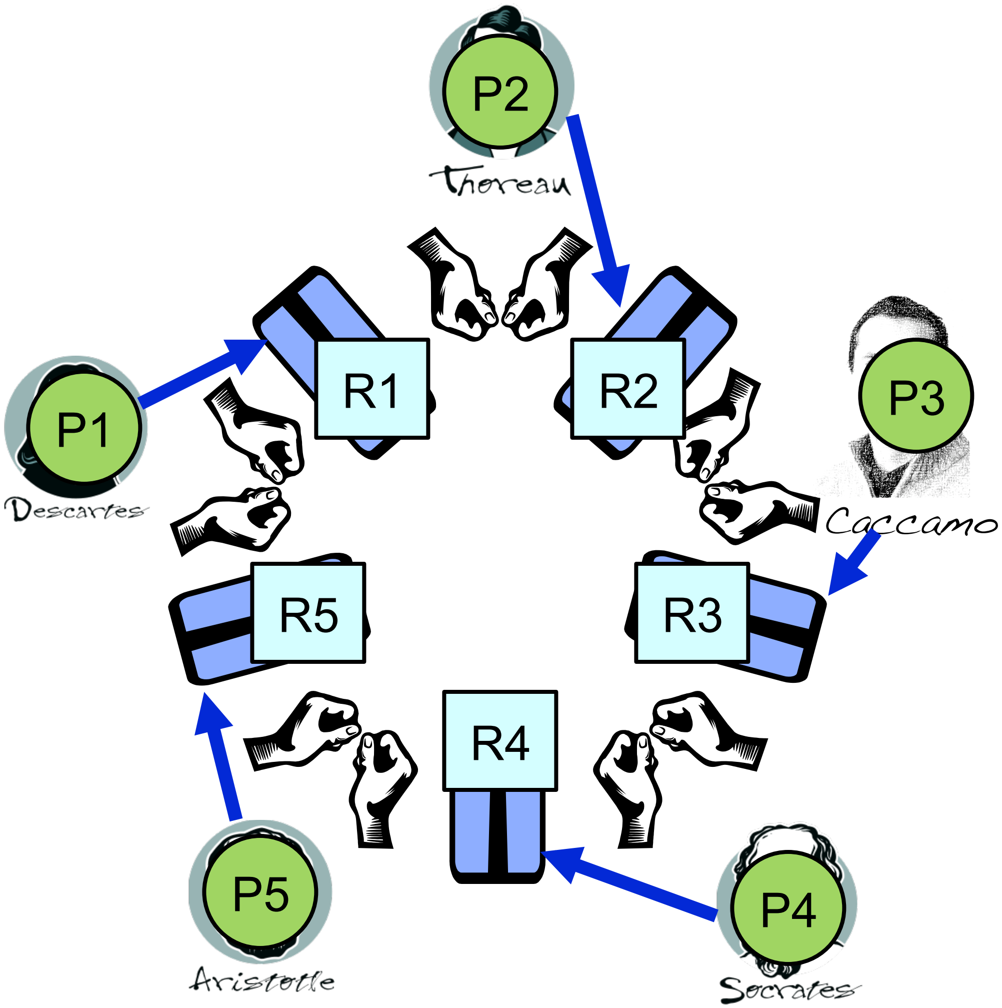
If we use the trivial broken "solution"...

One node per philosopher

One node per fork

Everyone tries to pick up left fork

- Result: Request edges



Dining Philosophers resource allocation graph

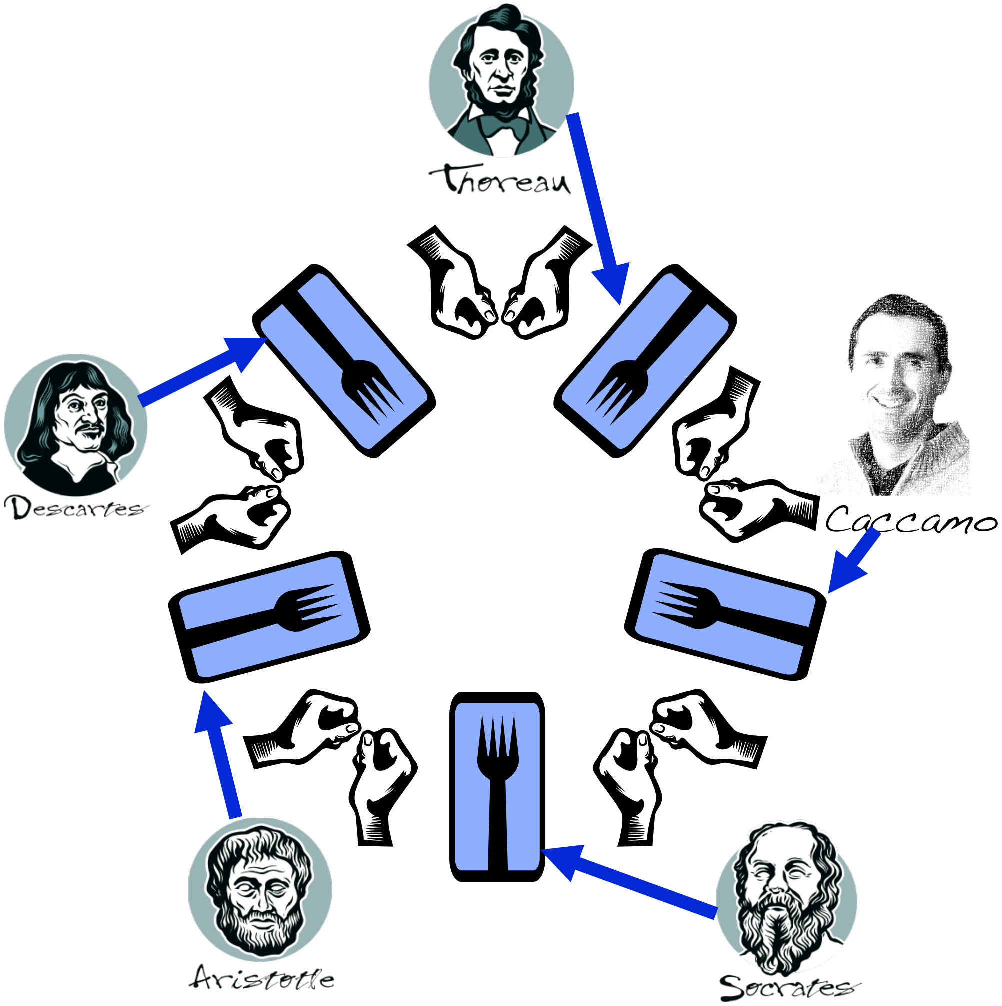
If we use the trivial broken "solution"...

One node per philosopher

One node per fork

Everyone tries to pick up left fork

- Result: Request edges
- Everyone succeeds!



Dining Philosophers resource allocation graph

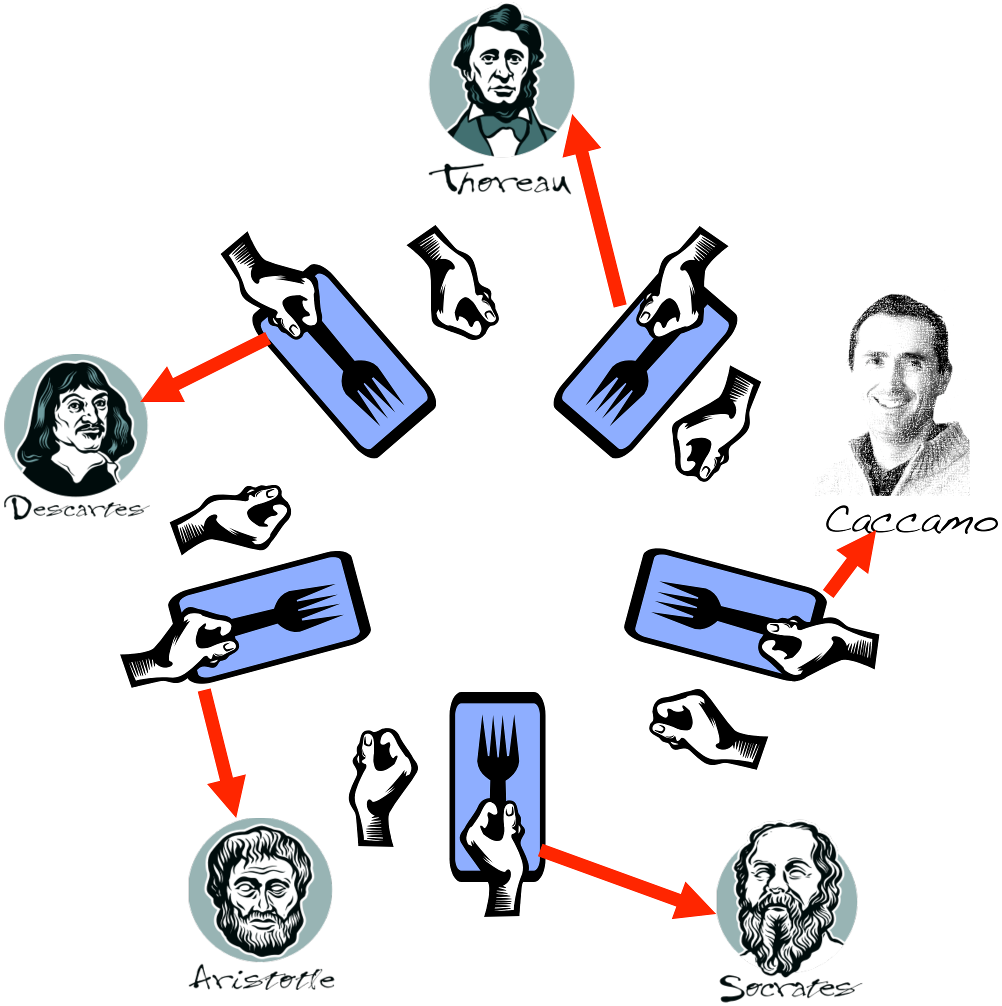
If we use the trivial broken "solution"...

One node per philosopher

One node per fork

Everyone tries to pick up left fork

- Result: Request edges
- Everyone succeeds!
- Result: Assignment edges



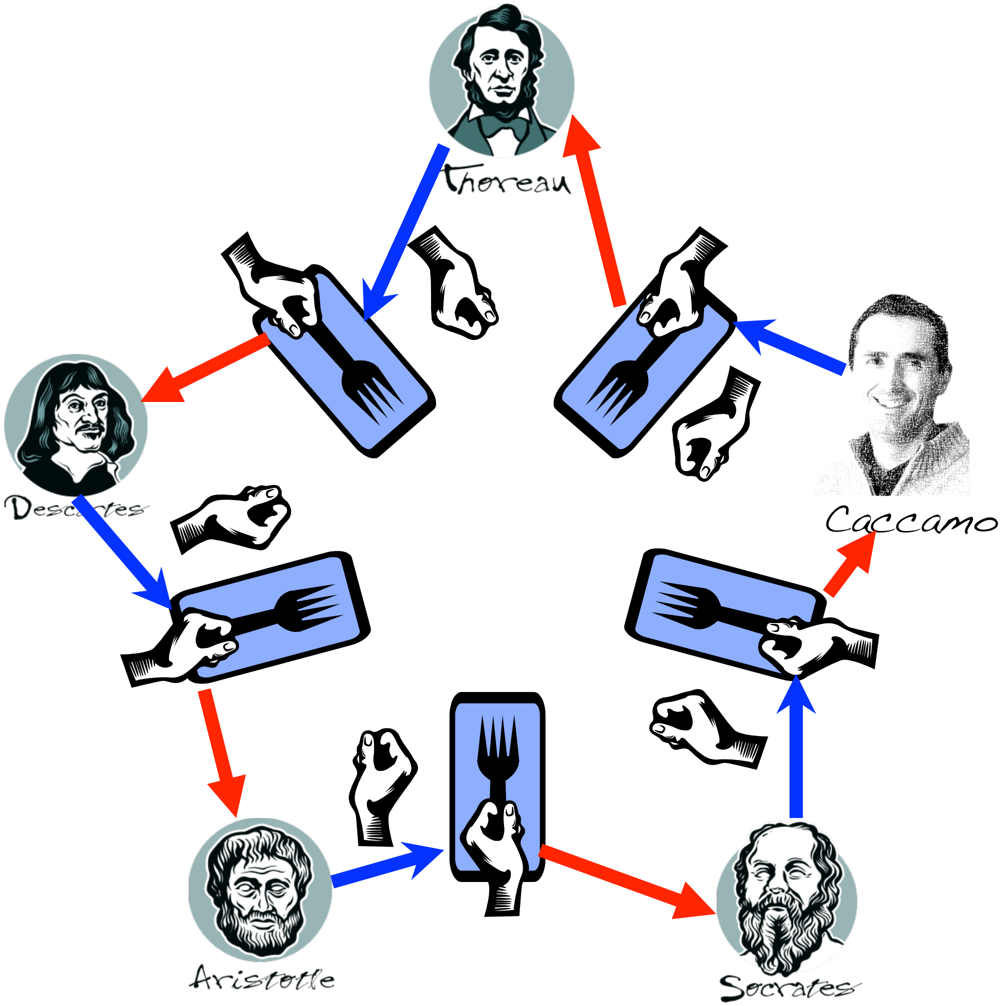
Dining Philosophers resource allocation graph

Everyone tries to pick up left fork

- Result: Request edges
- Everyone succeeds!
- Result: Assignment edges

Everyone tries to pick up right fork

- Result: Request edges



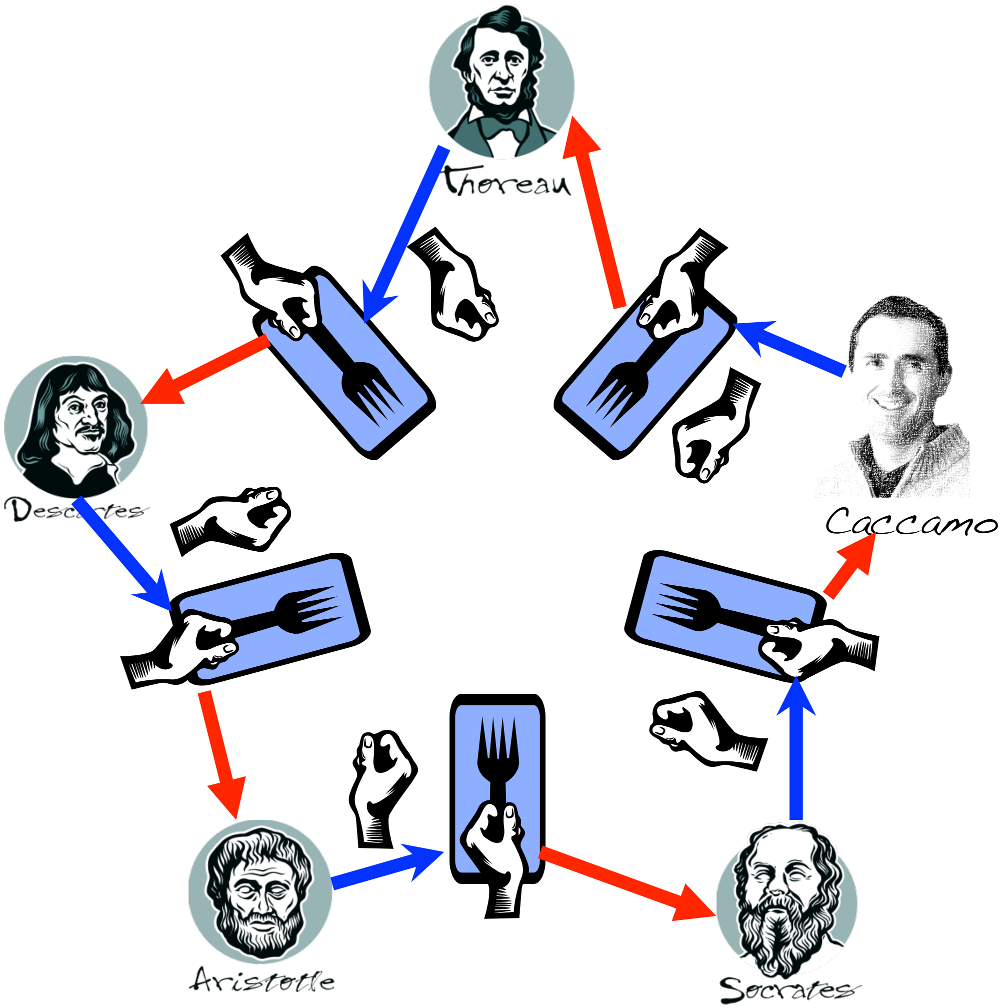
Dining Philosophers resource allocation graph

Everyone tries to pick up left fork

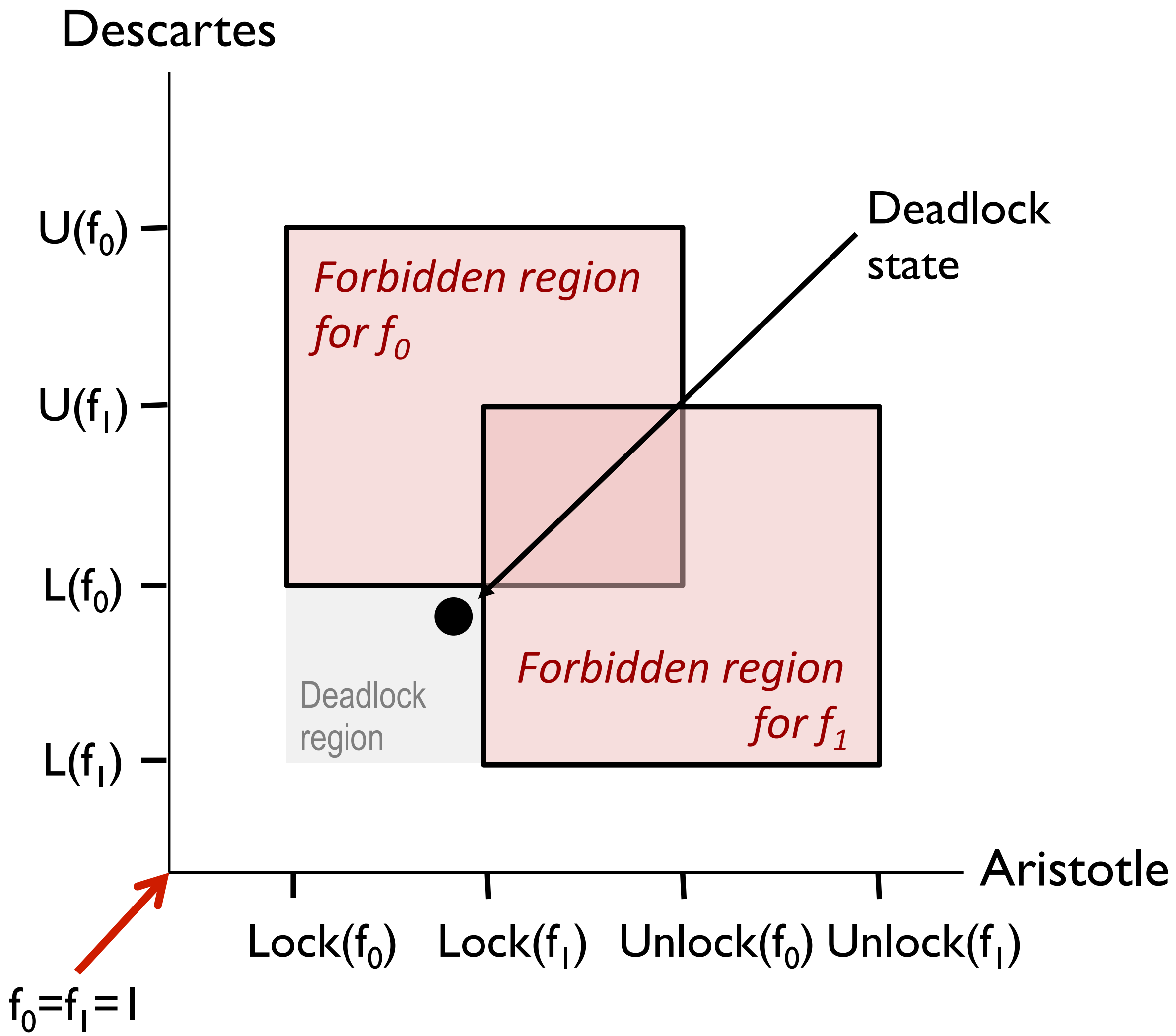
- Result: Request edges
- Everyone succeeds!
- Result: Assignment edges

Everyone tries to pick up right fork

- Result: Request edges
- DEADLOCK

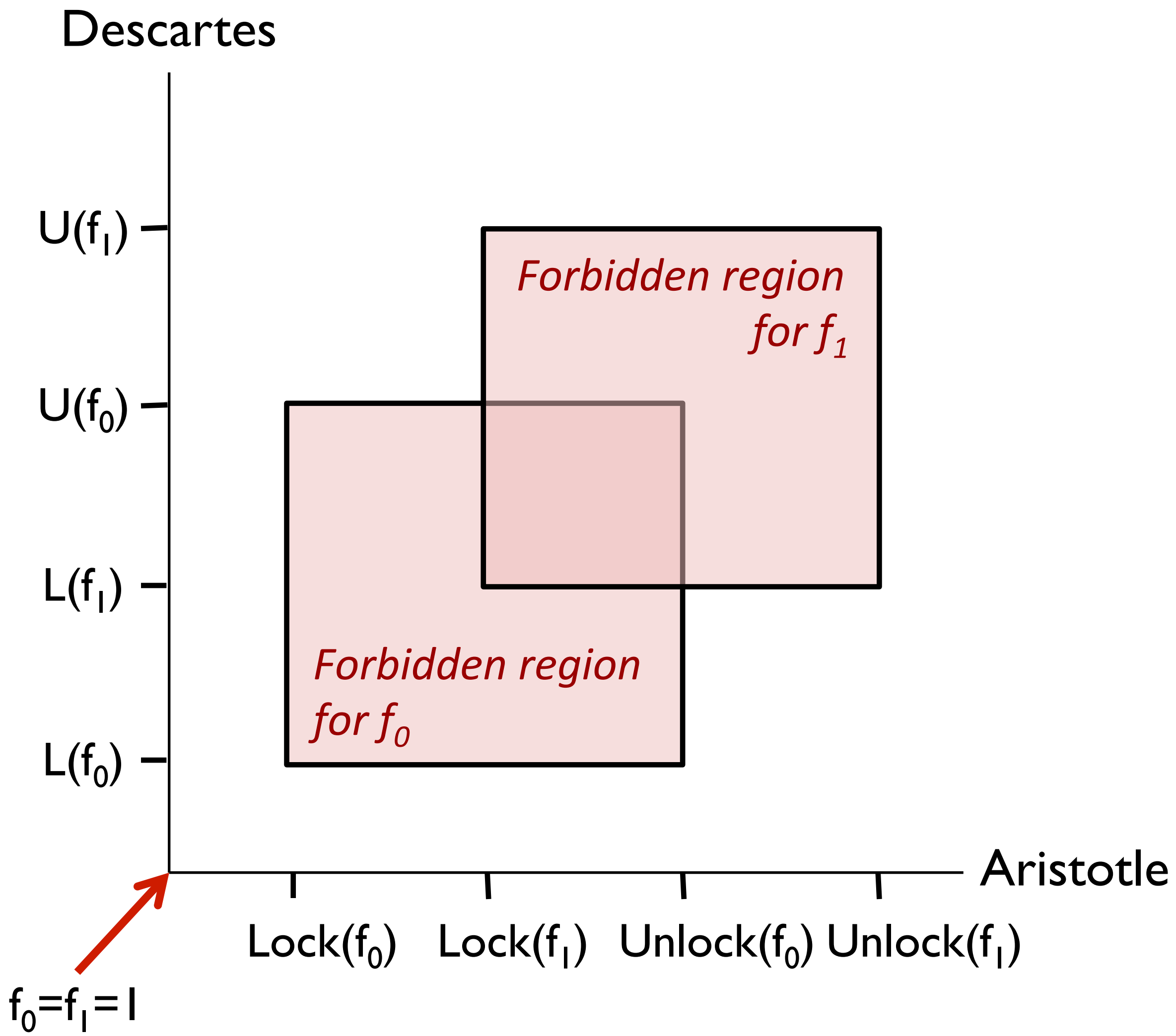


Idea: change the order



Old order

Idea: change the order



New order

Deadlock impossible!

Summary

Deadlock

- Cycle of processes / threads, each waiting on the next
- Modeled by cycle in resource allocation graph
- Often nondeterministic, tricky to debug

Next: dealing with deadlocks

- “change the order” was a nice trick... but why did it work?
- Is there a simple technique that will work always?
- Are there other ways of avoiding deadlocks?

Deadlock solutions

Prevention

- Design system so that deadlock is impossible

Avoidance

- Steer around deadlock with smart scheduling

Detection & recovery

- Check for deadlock periodically
- Recover by killing a deadlocked processes and releasing its resources

Do nothing

- Prevention, avoidance, and detection/recovery are expensive
- If deadlock is rare, is it worth the overhead?
- Manual intervention (kill processes, reboot) if needed

Deadlock Prevention

Deadlock prevention

Goal 1: devise resource allocation rules which 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

Rule #1: No Mutual Exclusion

For deadlock to happen: processes must claim exclusive control of the resources they require

How to break it?

Rule #1: No Mutual Exclusion

For deadlock to happen: processes must claim exclusive control of the resources they require

How to break it?

- Non-exclusive access only
  - Read-only access
- Battle won!
  - War lost
  - Very bad at Goal #2

Rule #2: Allow preemption

A lock can be taken away from current owner

- Let it go: If a process holding some resources is denied a further request, that process must release its original resources
- Or take it all away: OS preempts current resource owner, gives resource to new process/thread requesting it

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!

Question

Suppose every thread only tries to hold one resource at a time

- So, a thread might hold one mutex lock…
- But it will never try to acquire a second mutex lock if it has one already

Is deadlock possible? Give an example or explain why not.

- *(Hint: what would the resource allocation diagram look like?)*

Rule #3: No hold and wait

When waiting for a resource, must not hold others

- So, process can only have one resource locked
- Or, it must request all resources at the beginning
- Or, before asking for more: give up everything you have and request it all at one time

Breaks circular wait

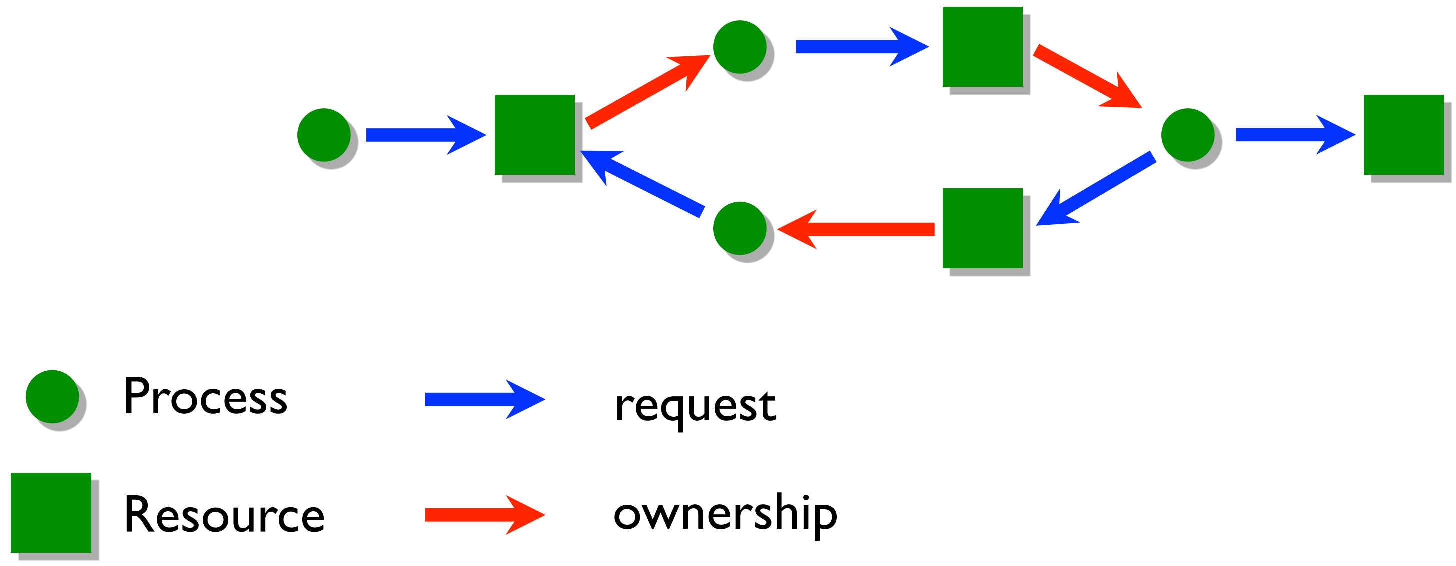
- In resource allocation diagram: process with an outgoing link must have no incoming links
- Therefore, cannot have a loop!

Rule #3: No hold and wait

Breaks circular wait

- In resource allocation diagram: process with an outgoing link must have no incoming links
- Therefore, cannot have a loop!

Q: Which of these request links would be disallowed?

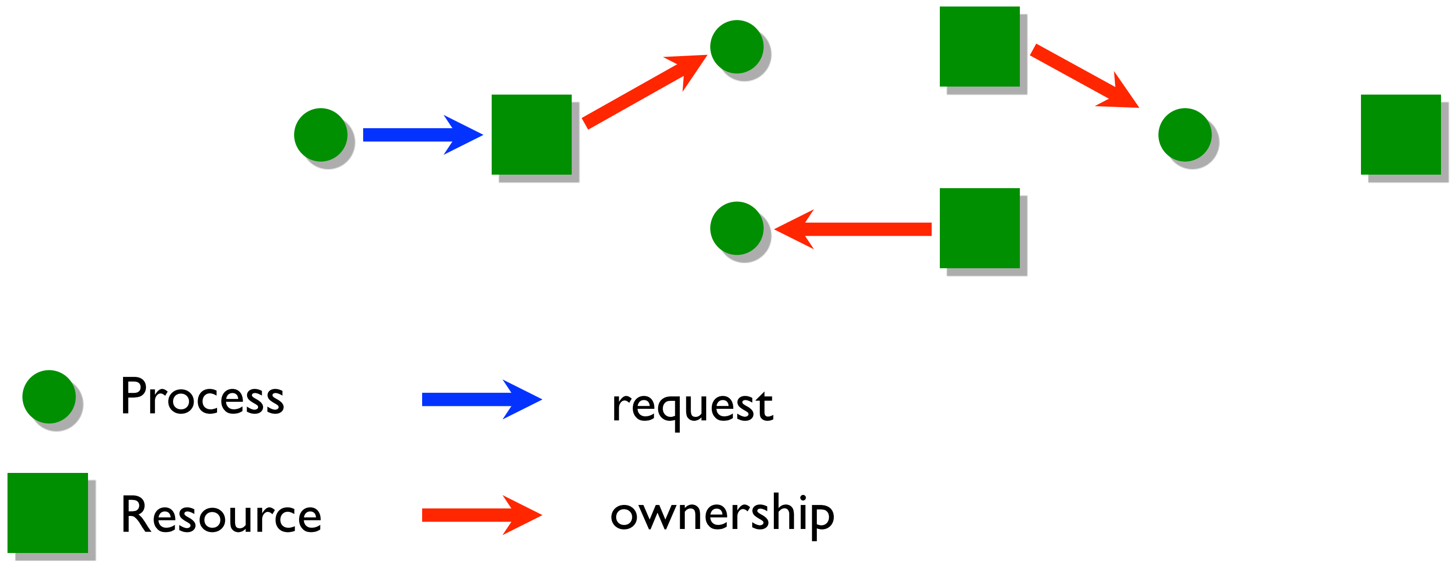


Rule #3: No hold and wait

Breaks circular wait

- In resource allocation diagram: process with an outgoing link must have no incoming links
- Therefore, cannot have a loop!

A: Legal links are...



Process

Resource

request

ownership

Rule #3: No hold and wait

Very constraining (mediocre job on Goal #2)

- Better than Rules #1 and #2, but...
- Often need more than one resource
- Hard to predict at the beginning what resources you'll need
- Releasing and re-requesting is inefficient, complicates programming, might lead to starvation

Rule #4: request resources in order

Must request resources in increasing order

- Impose ordering on resources (any ordering will do)
- If holding resource *i*, can only request resources > *i*

Much less constraining (decent job on Goal #2)

- Strictly easier to satisfy than “No hold and wait”: If we can request all resources at once, then we can request them in increasing order
- But now, we don’t need to request them all at once
- Can pick the arbitrary ordering for convenience to the application
- Still might be inconvenient at times

But why is it guaranteed to preclude circular wait?

Announcements

Problem 3: watch for announcement

Brighten out of town Friday

- Marco returns