Deadlock Solutions: Prevention

CS 241

March 31, 2014

University of Illinois

Announcement

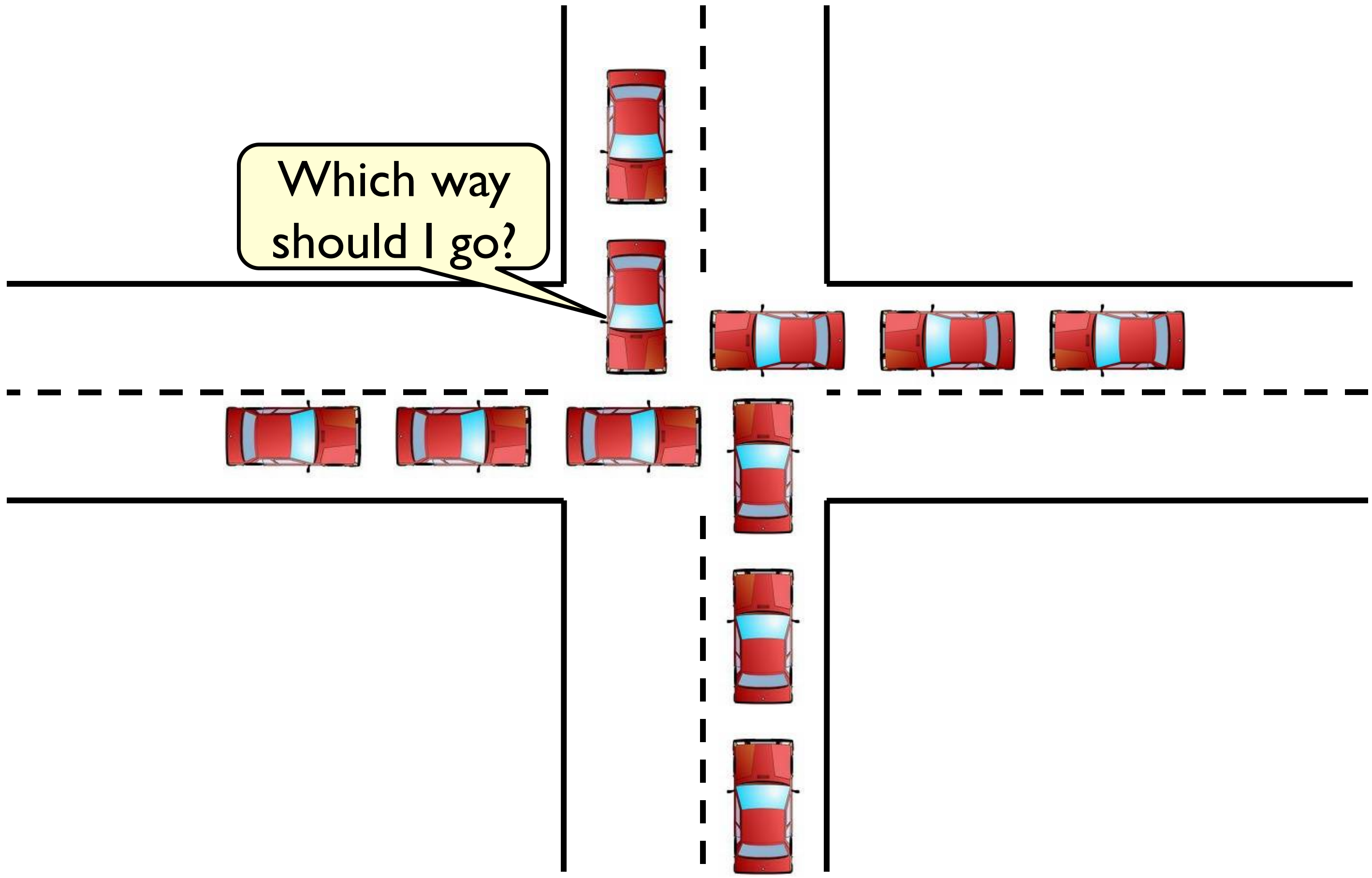
Brighten's office hours today, 12-1

Deadlock: definition

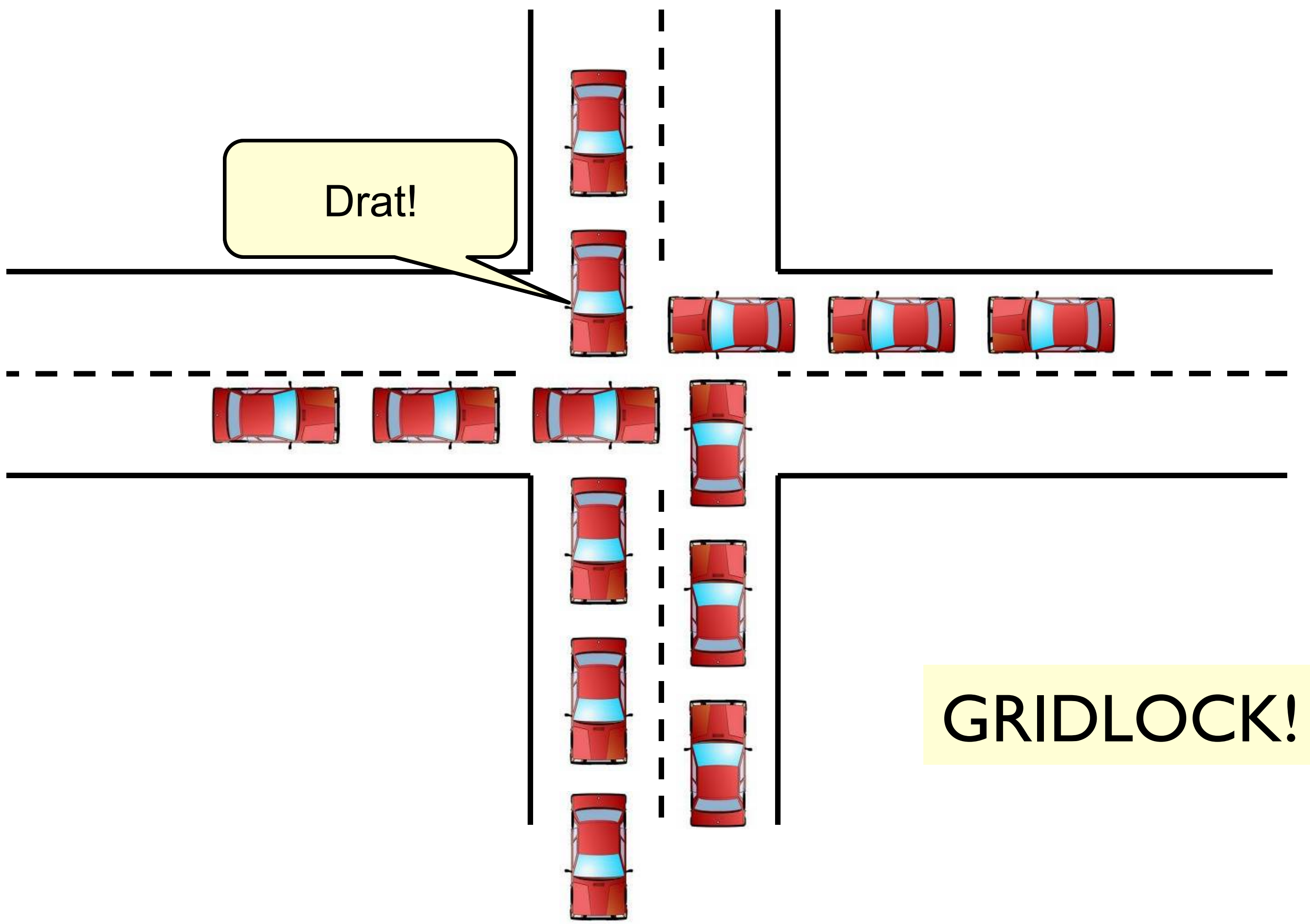
There exists a cycle of processes such that each process cannot proceed until the next process takes some specific action.

Result: all processes in the cycle are stuck!

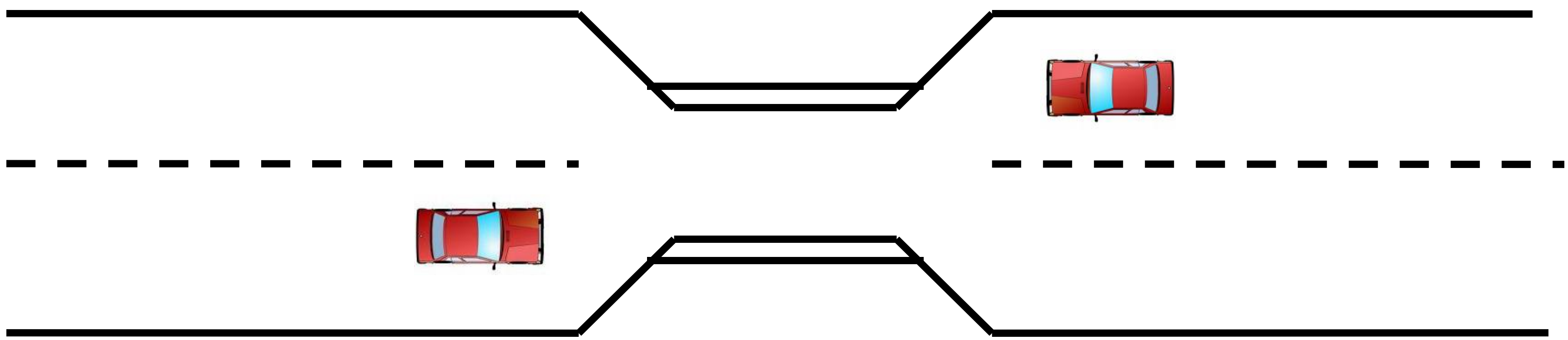
Deadlock in the real world



Deadlock in the real world



Deadlock: One-lane Bridge

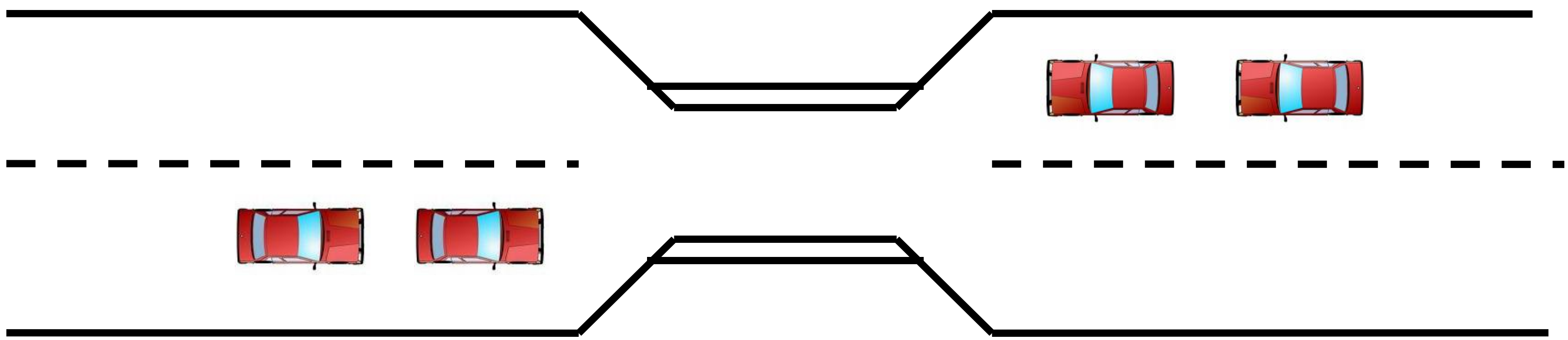


Traffic only in one direction

Each section of a bridge can be viewed as a resource

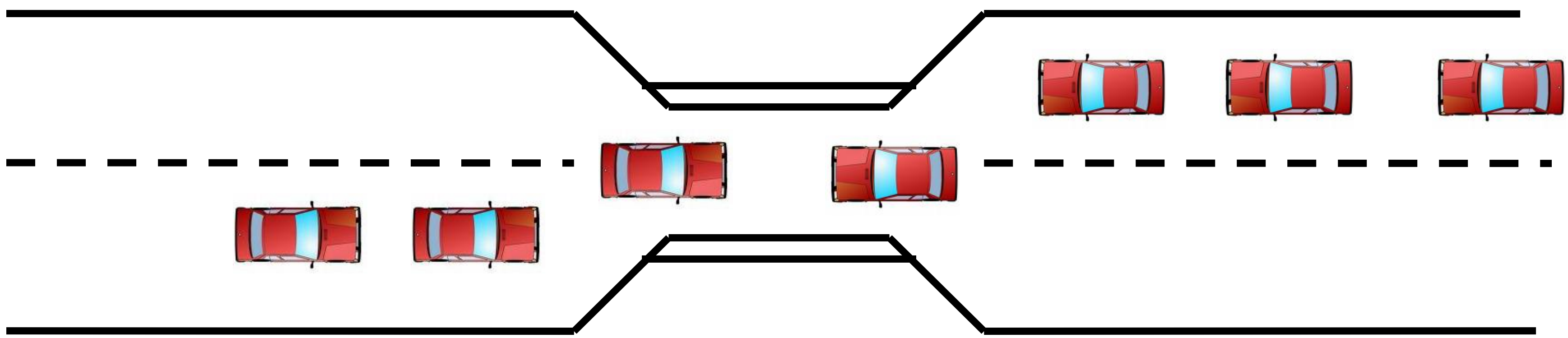
What can happen?

Deadlock: One-lane Bridge



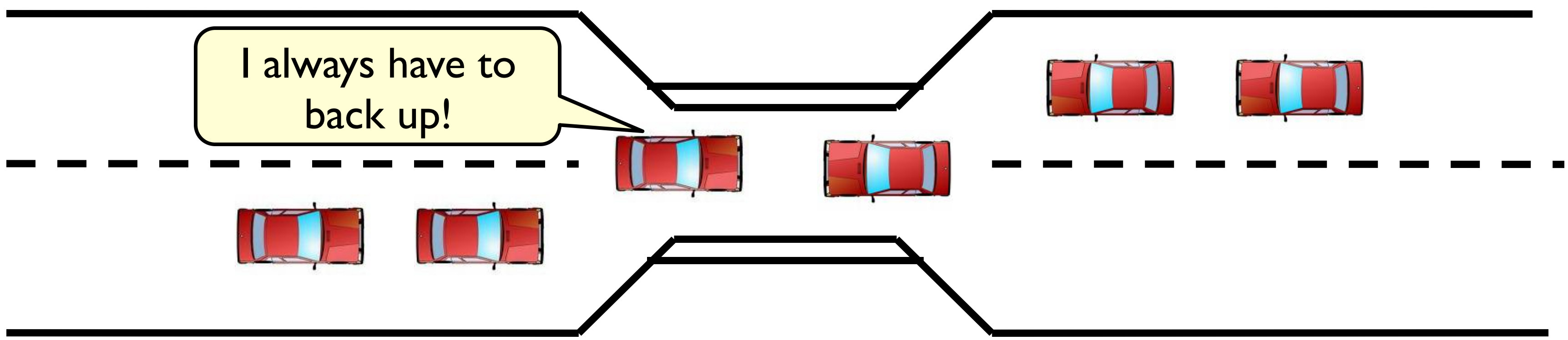
- Traffic only in one direction
- Each section of a bridge can be viewed as a resource
- Deadlock
  - Resolved if cars back up (preempt resources and rollback)
  - Several cars may have to be backed up

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 when present

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 (unfortunate scheduling)
- Deadlock = no hope
  - All processes blocked; scheduling change won't help

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!

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

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*

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

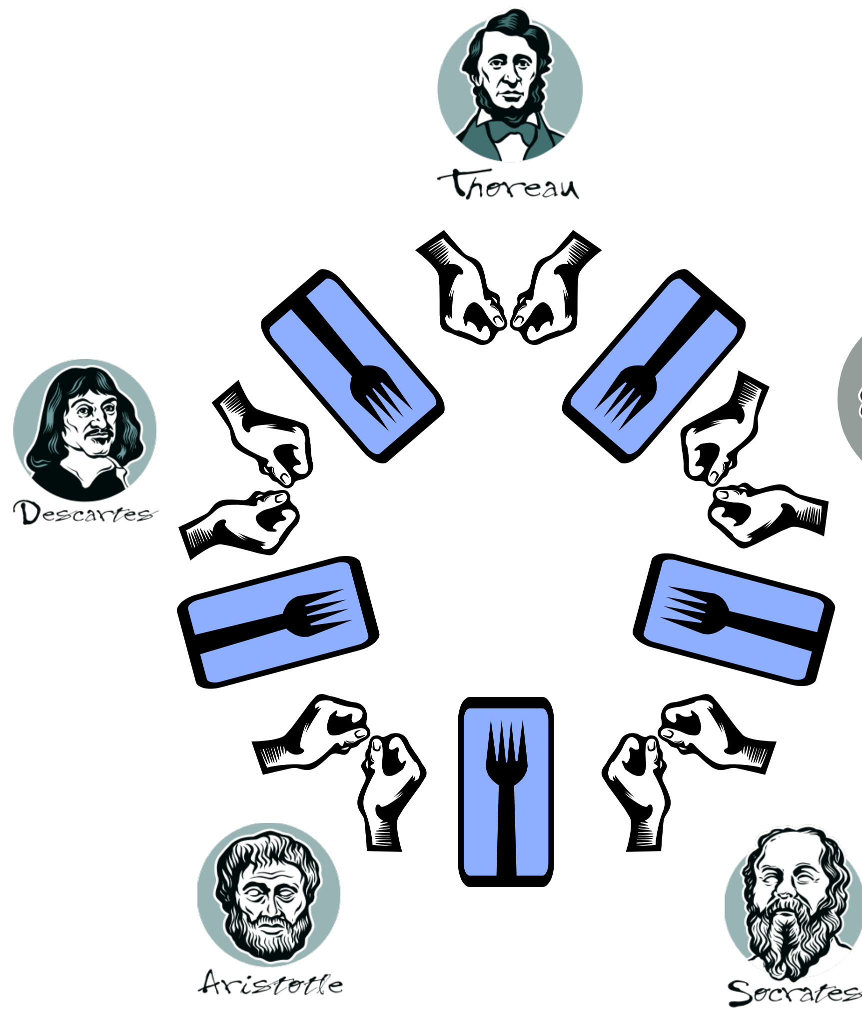
But why is it guaranteed to preclude circular wait?

Dining Philosophers solution with unnumbered 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);
    }
}
```

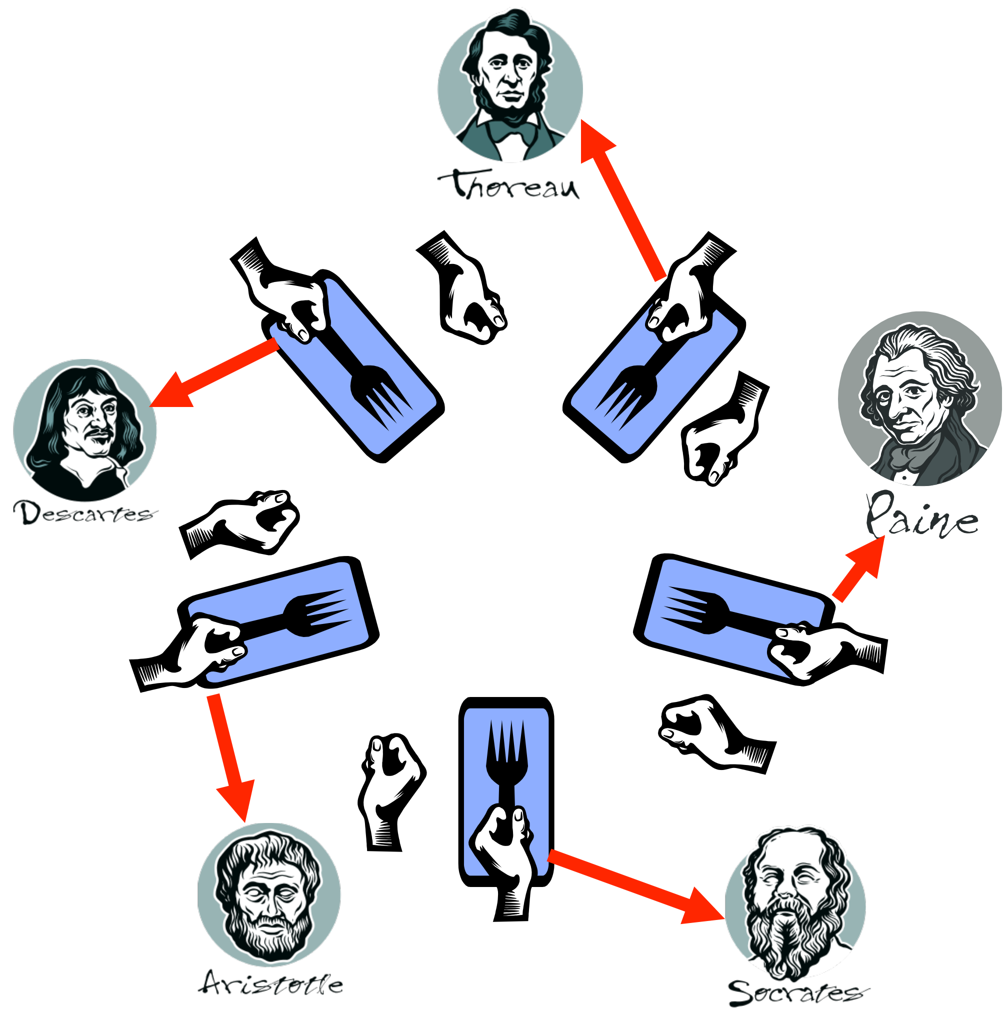


Dining Philosophers solution with unnumbered resources

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);
    }
}
```

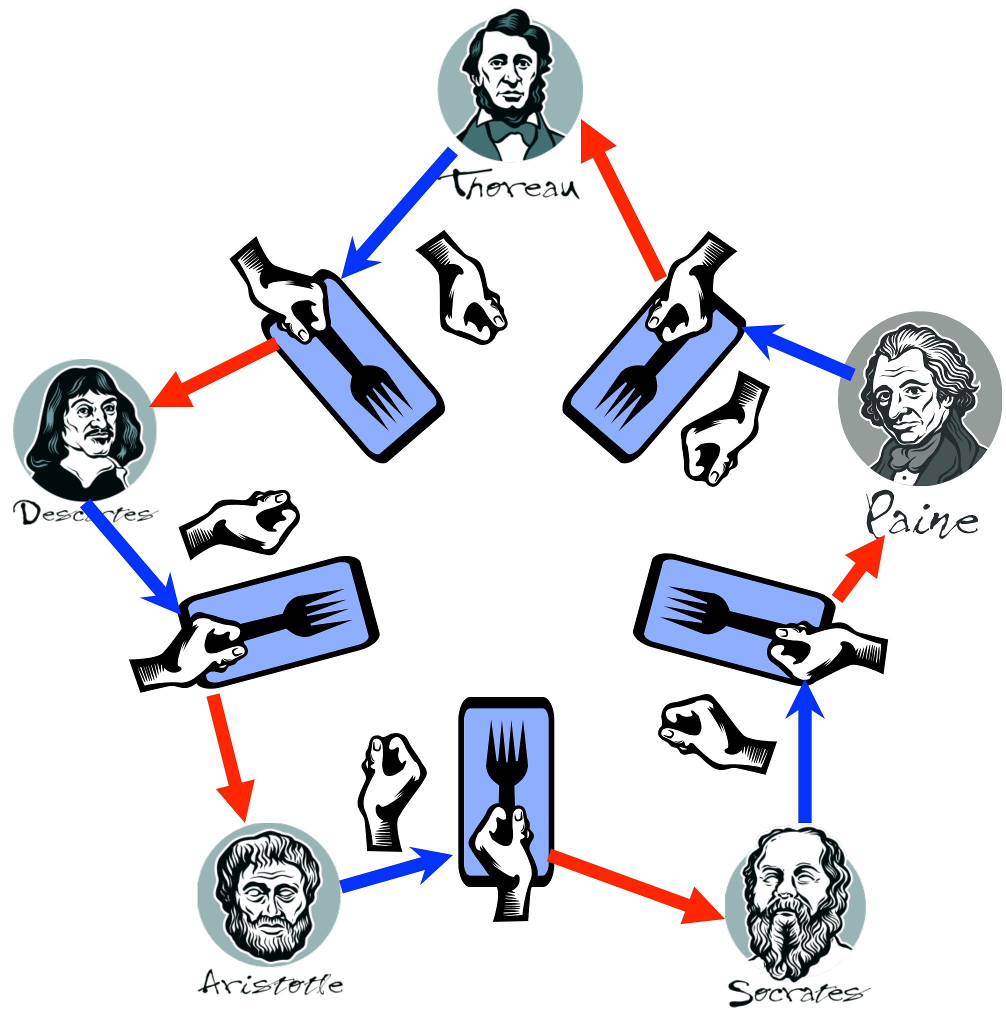


Dining Philosophers solution with unnumbered resources

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);
    }
}
```



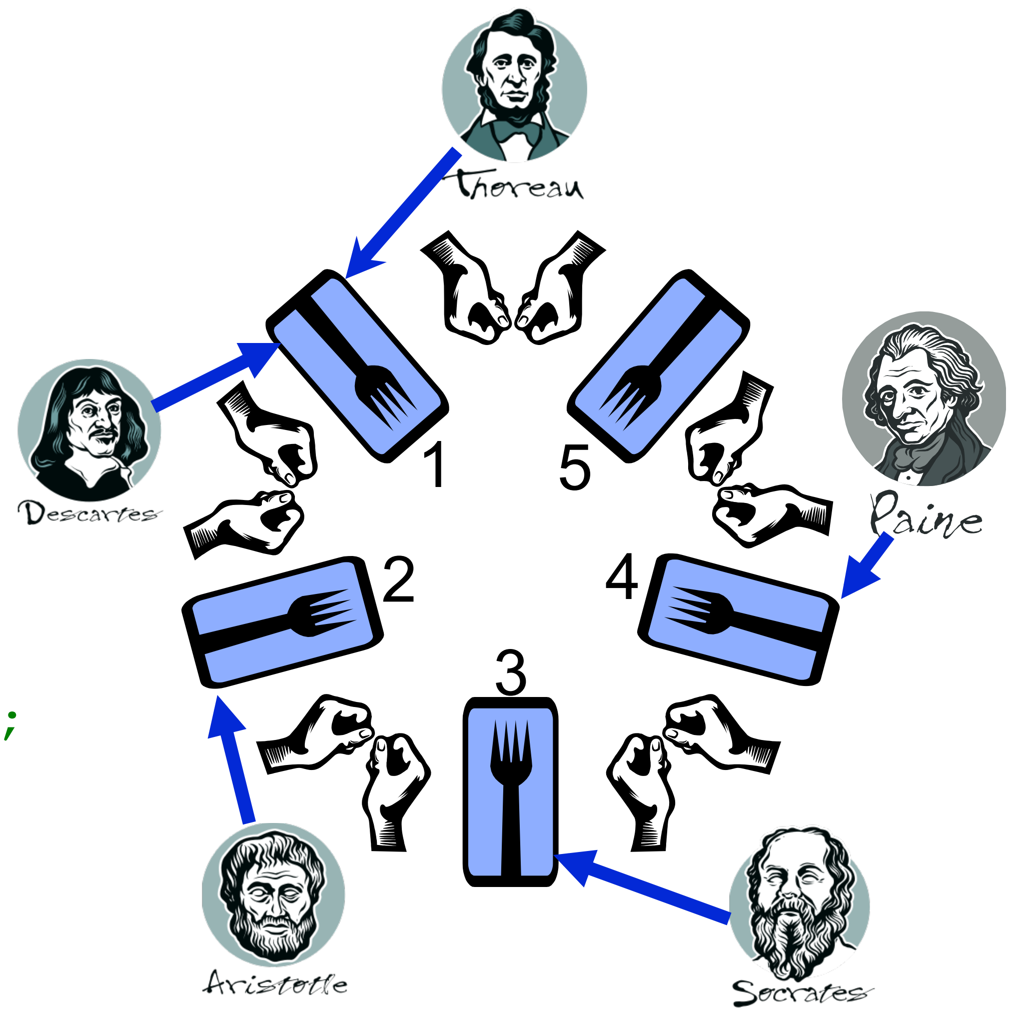
Dining Philosophers solution with numbered resources

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));
    }
}
```



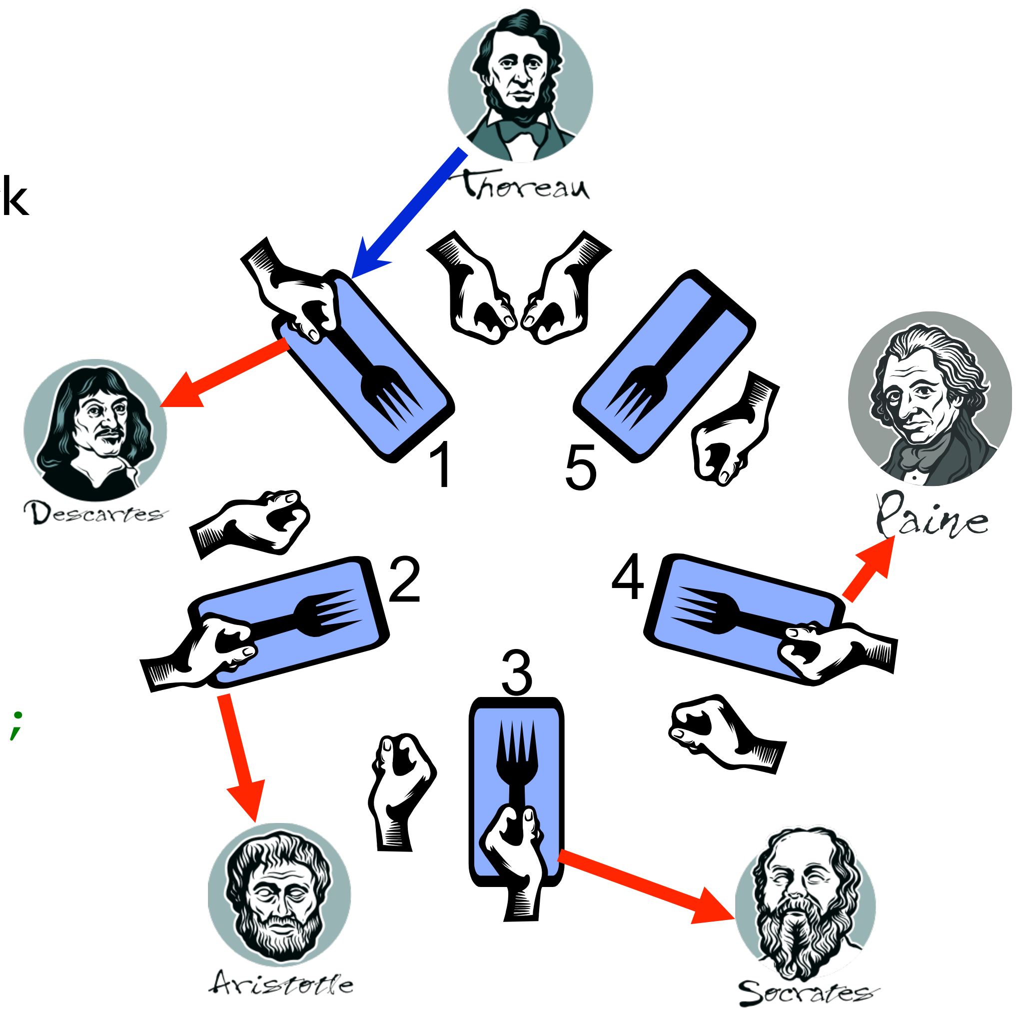
Dining Philosophers solution with numbered resources

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));
  }
}
```



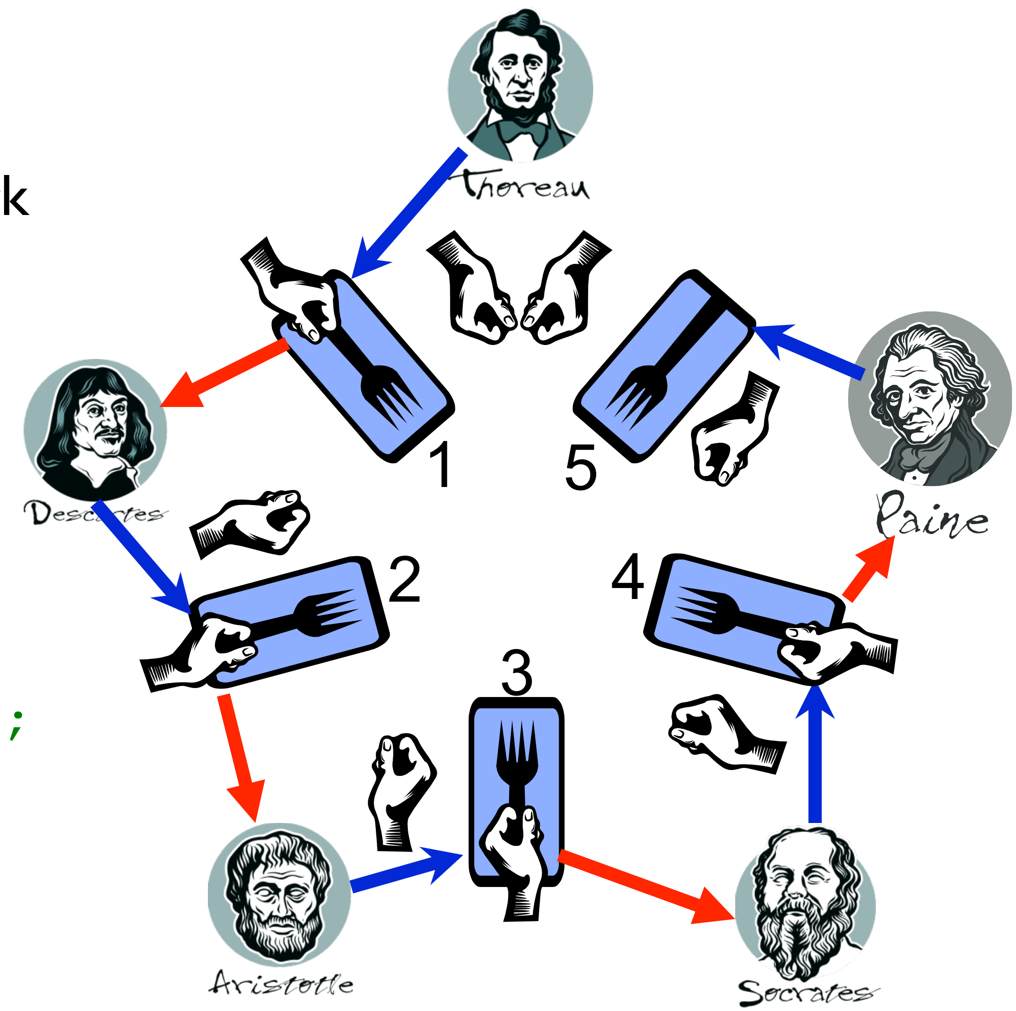
Dining Philosophers solution with numbered resources

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));
  }
}
```



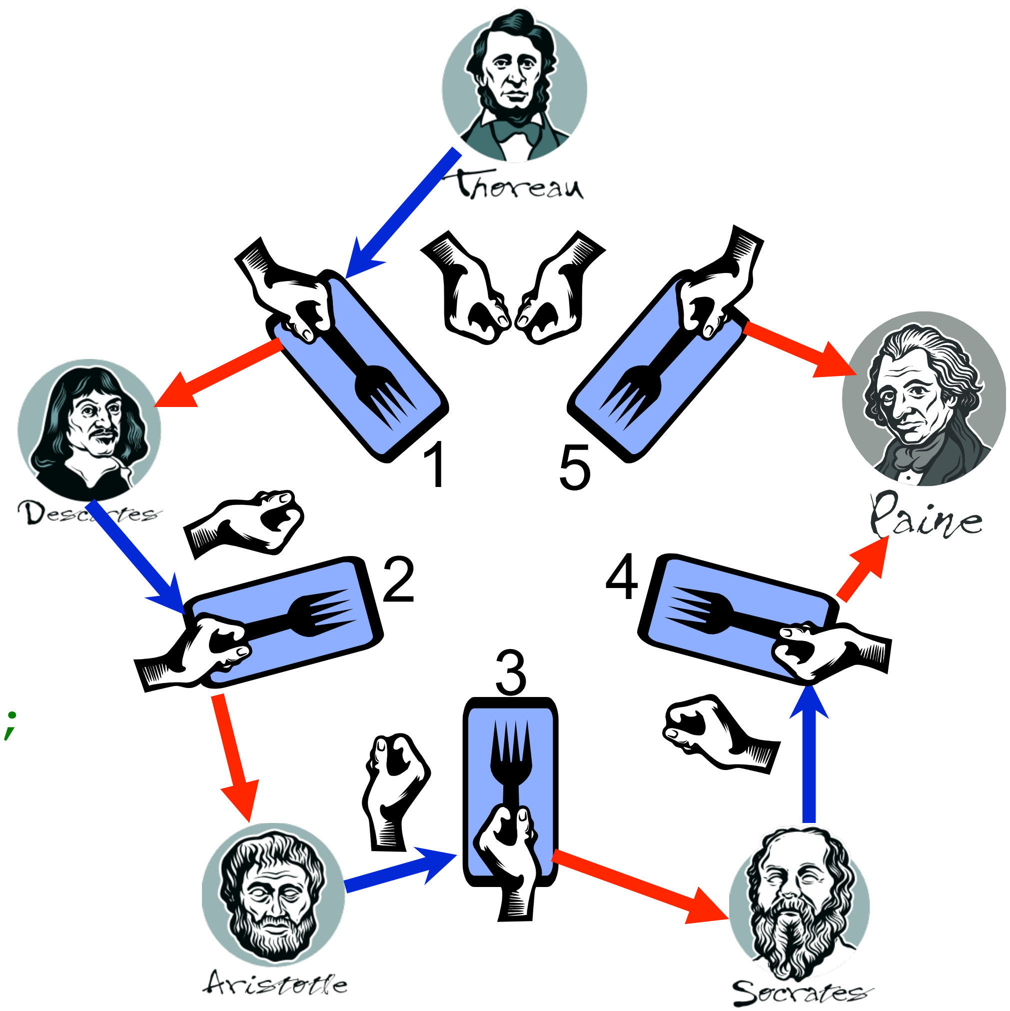
Dining Philosophers solution with numbered resources

Instead, number resources...

One philosopher can eat!

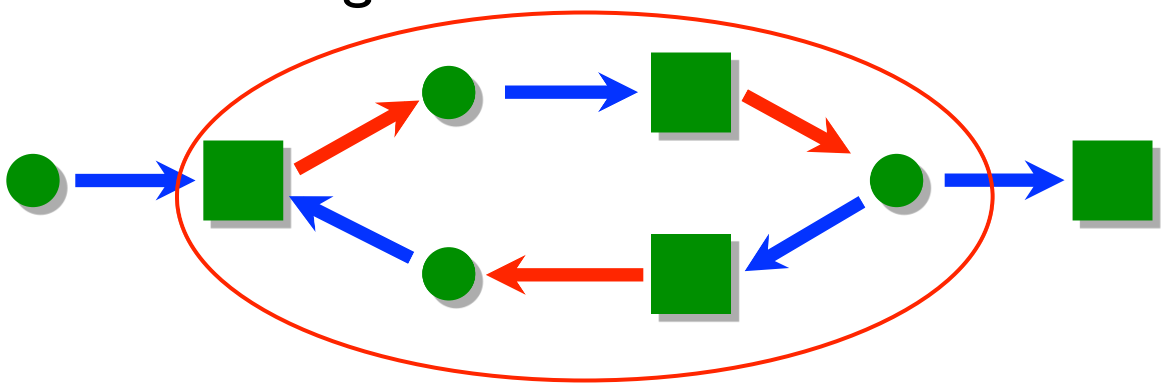
```
# define N 5

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

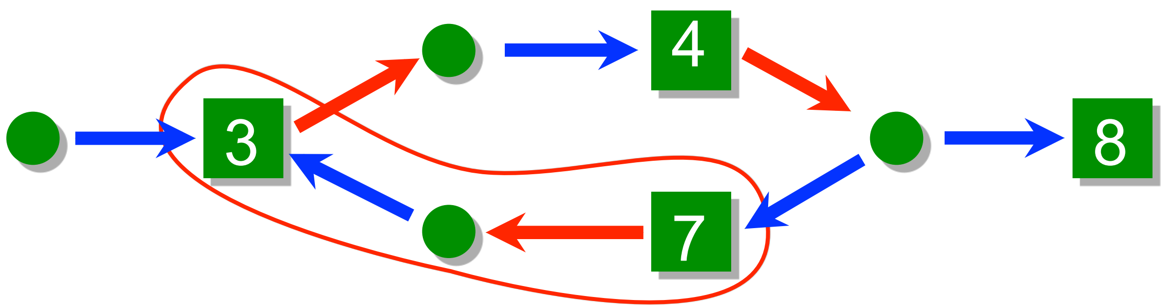
Without numbering



Cycle!

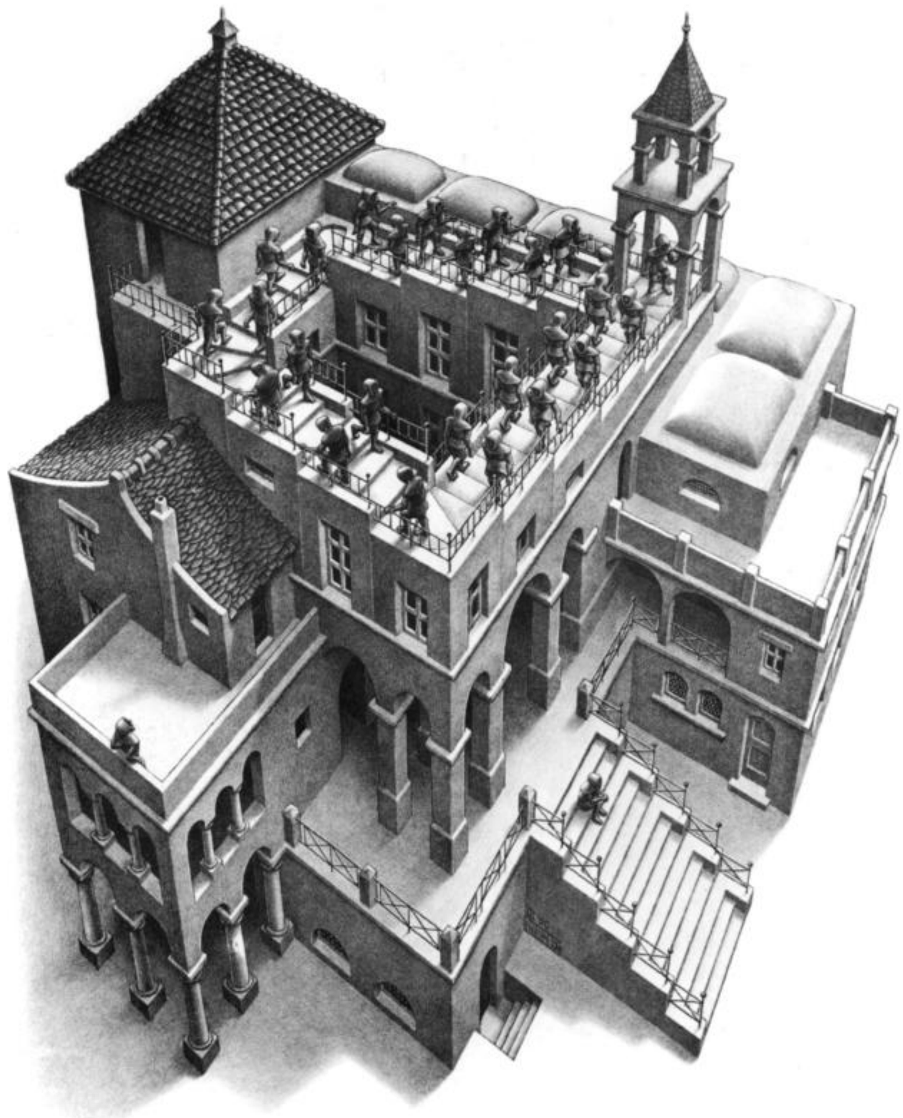
Ordered resource requests prevent deadlock

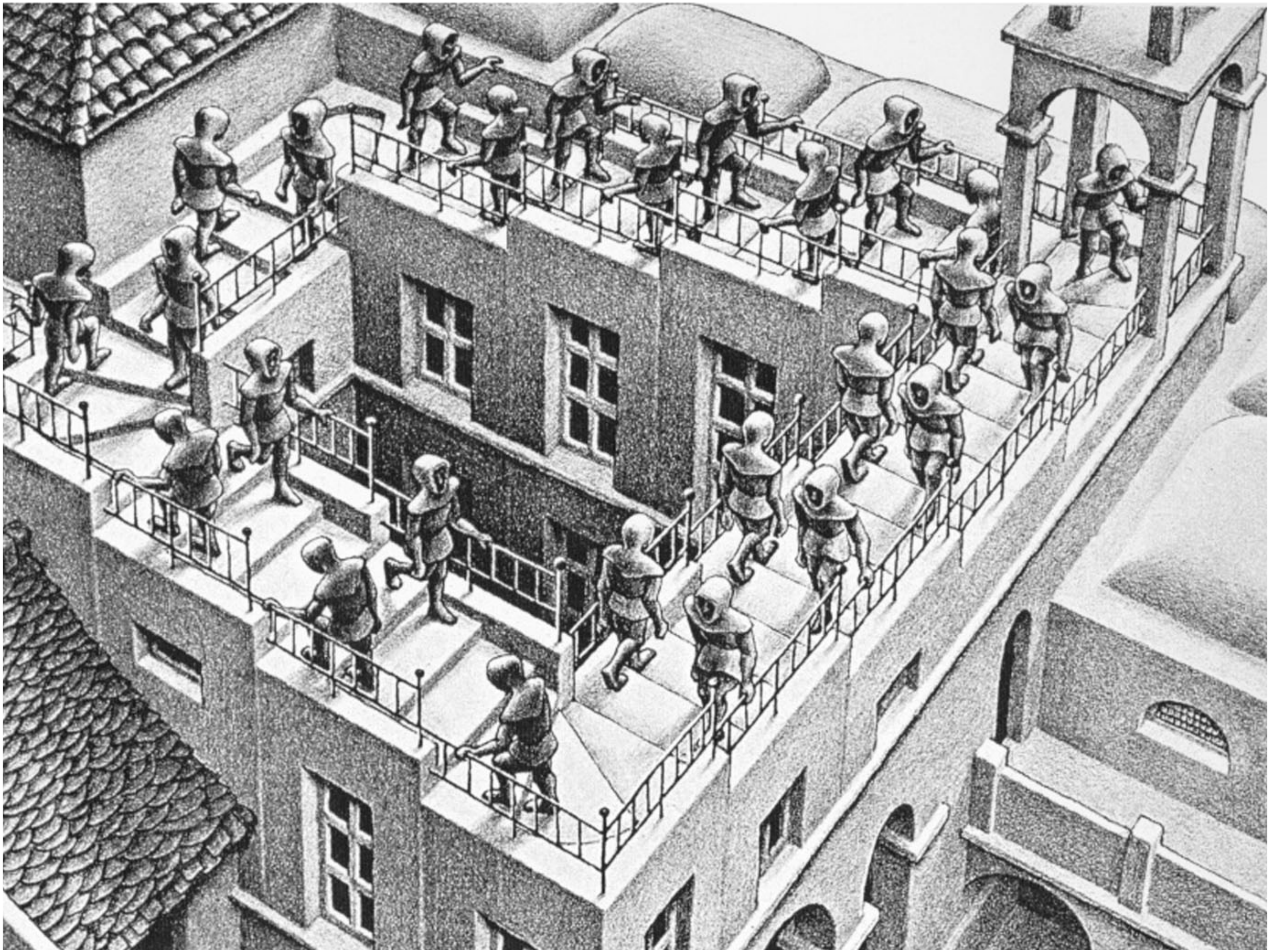
With numbering



Ordering violation:
Process holds 7,
is requesting 3

Proof by M.C. Escher





Summary: Deadlock prevention methods

#1: No mutual exclusion

- Thank you, Captain Obvious

#2: Allow preemption

- OS can revoke resources from current owner

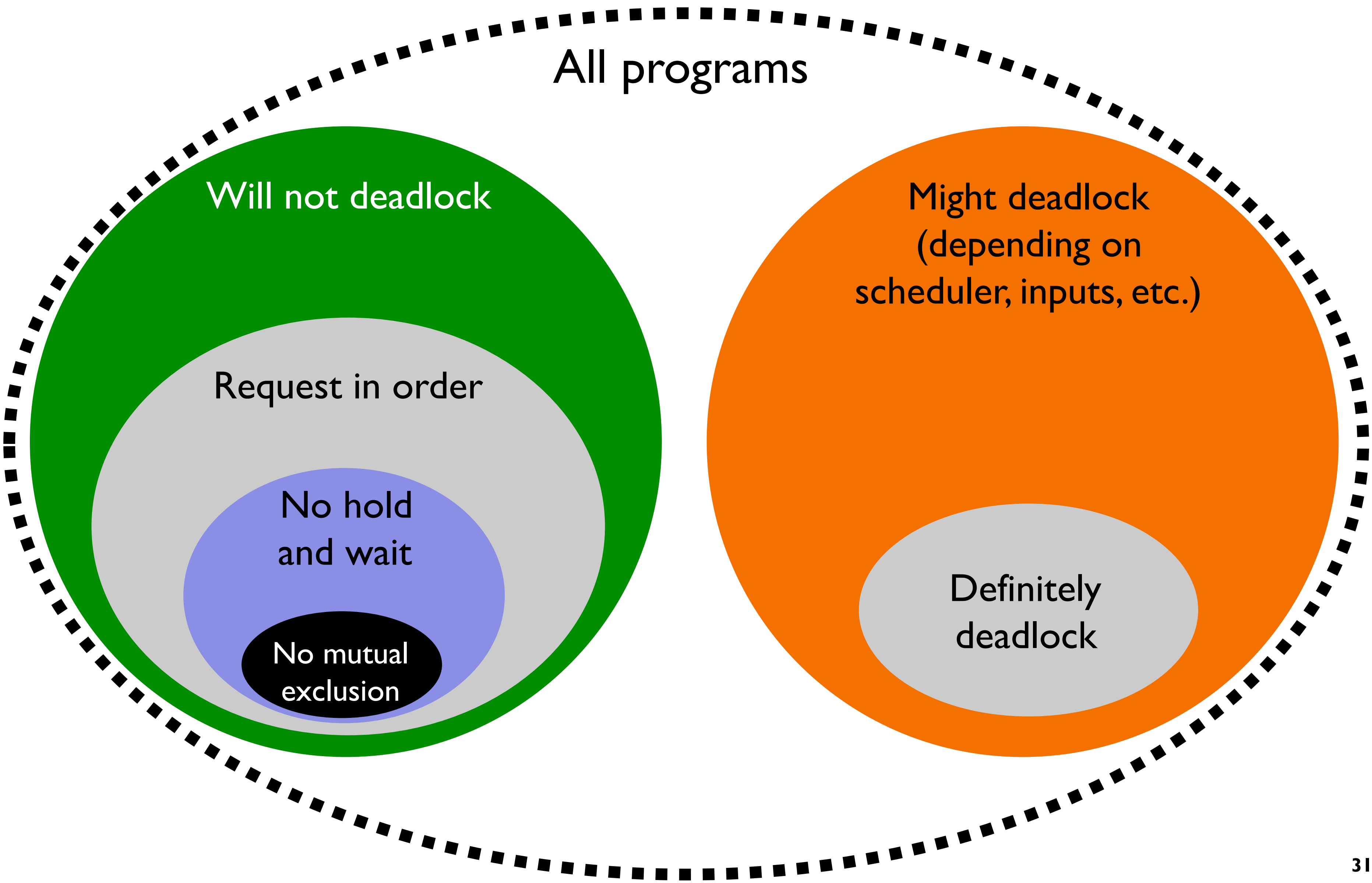
#3: No hold and wait

- When waiting for a resource, must not currently hold any resource

#4: Request resources in order

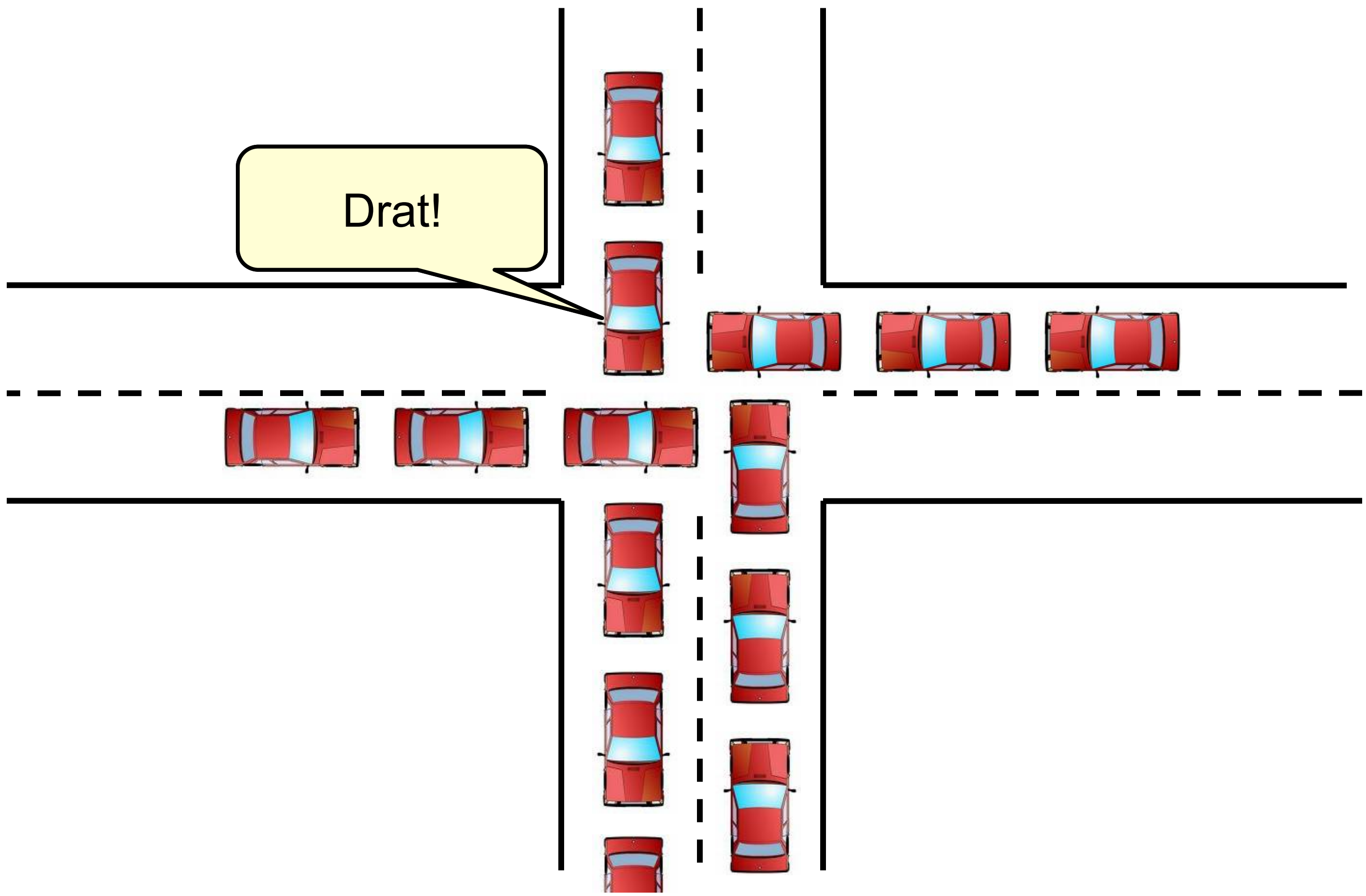
- When waiting for resource i, must not currently hold any resource $j > i$
- As you can see: If your program satisfies #3 then it satisfies #4

“Request In Order” is more permissive



Q: What's the rule of the road?

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



Summary

Deadlock prevention

- Imposes rules on what system can do
- These rules are conservative
- Most useful technique: ordered resources
- Application can do it; no special OS support

Next: dealing with deadlocks other ways

- Avoidance
- Detection & recovery

Deadlock Avoidance

Deadlock Avoidance

Idea: Steer around deadlock with smart scheduling

Assume OS knows:

- Number of available units of each resource
  - Each individual mutex lock is a resource with one unit available
  - Each individual semaphore is a resource with possibly multiple units available
- For each process, current amount of each resource it owns
- For each process, maximum amount of each resource it might ever need
  - For a mutex this means: Will the process ever lock the mutex?

Assume processes are independent

- If one blocks, others can finish if they have enough resources

How to guide the system down a safe path of execution

Helper function: is a given state safe?

- Safe = there's definitely a way to finish the processes without deadlock

When a resource allocation request arrives

- Pretend that we approve the request
- Call function: Would we then be safe?
- If safe,
  - Approve request
- Otherwise,
  - Block process until its request can be safely approved
  - Some other process is scheduled in the meantime

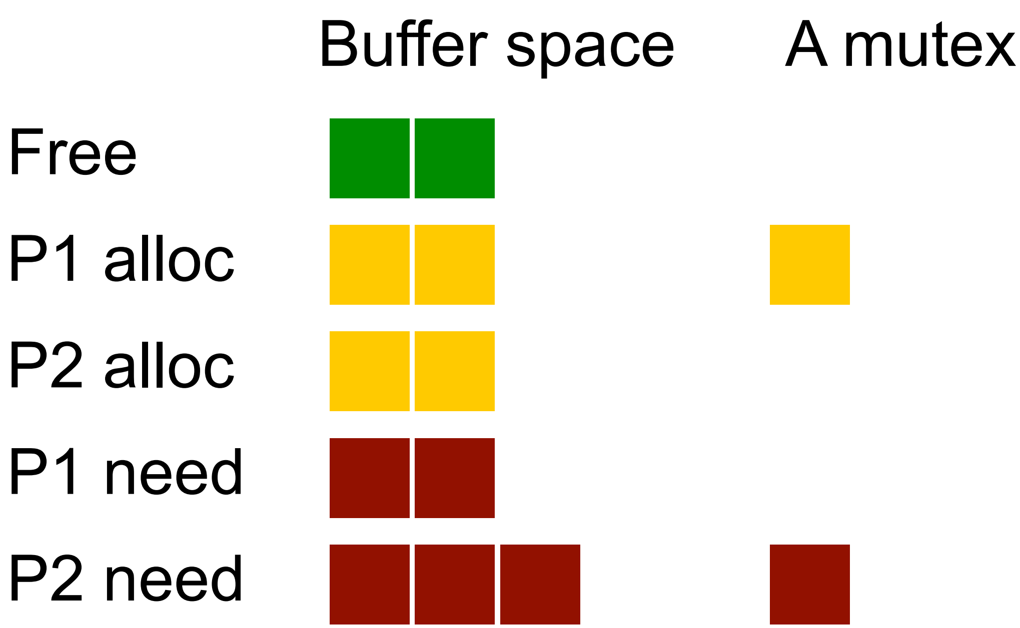
This is called the Banker's Algorithm

- Dijkstra, 1965

What is a state?

For each resource,

- Current amount available
- Current amount allocated to each process
- Future amount needed by each process (maximum)



When is a state safe?

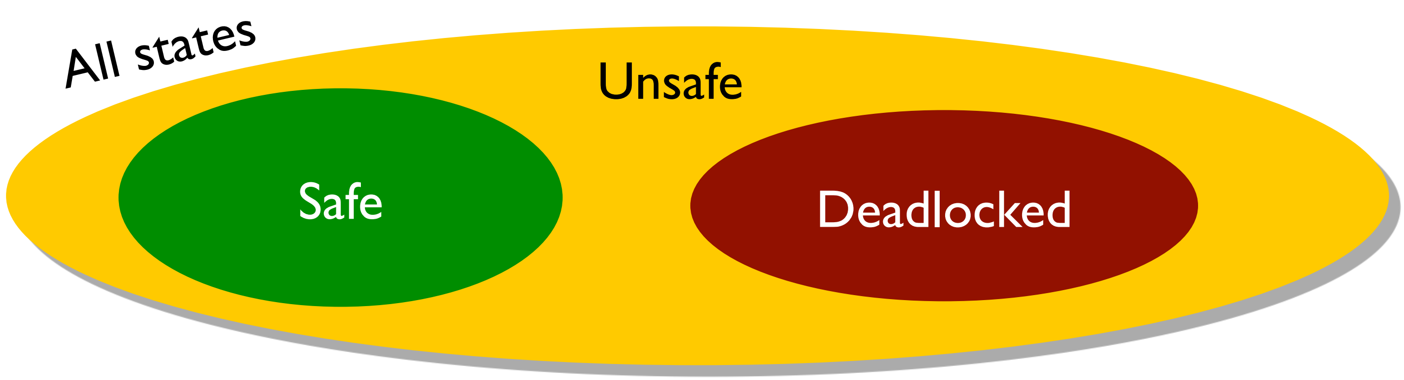
There is an execution order that can finish

In general, that's hard to predict

- So, we're conservative: find sufficient conditions for safety
- i.e., make some pessimistic assumptions

Pessimistic assumptions:

- A process might request its maximum resources at any time
- A process will never release its resources until it's done



Computing safety

"There is an execution order that can finish"

Search for an order P1, P2, P3, ... such that:

- P1 can finish using what it has plus what's free
- P2 can finish using what it has + what's free + what P1 releases when it finishes
- P3 can finish using what it has + what's free + what P1 and P2 will release when they finish
- ...

Computing safety

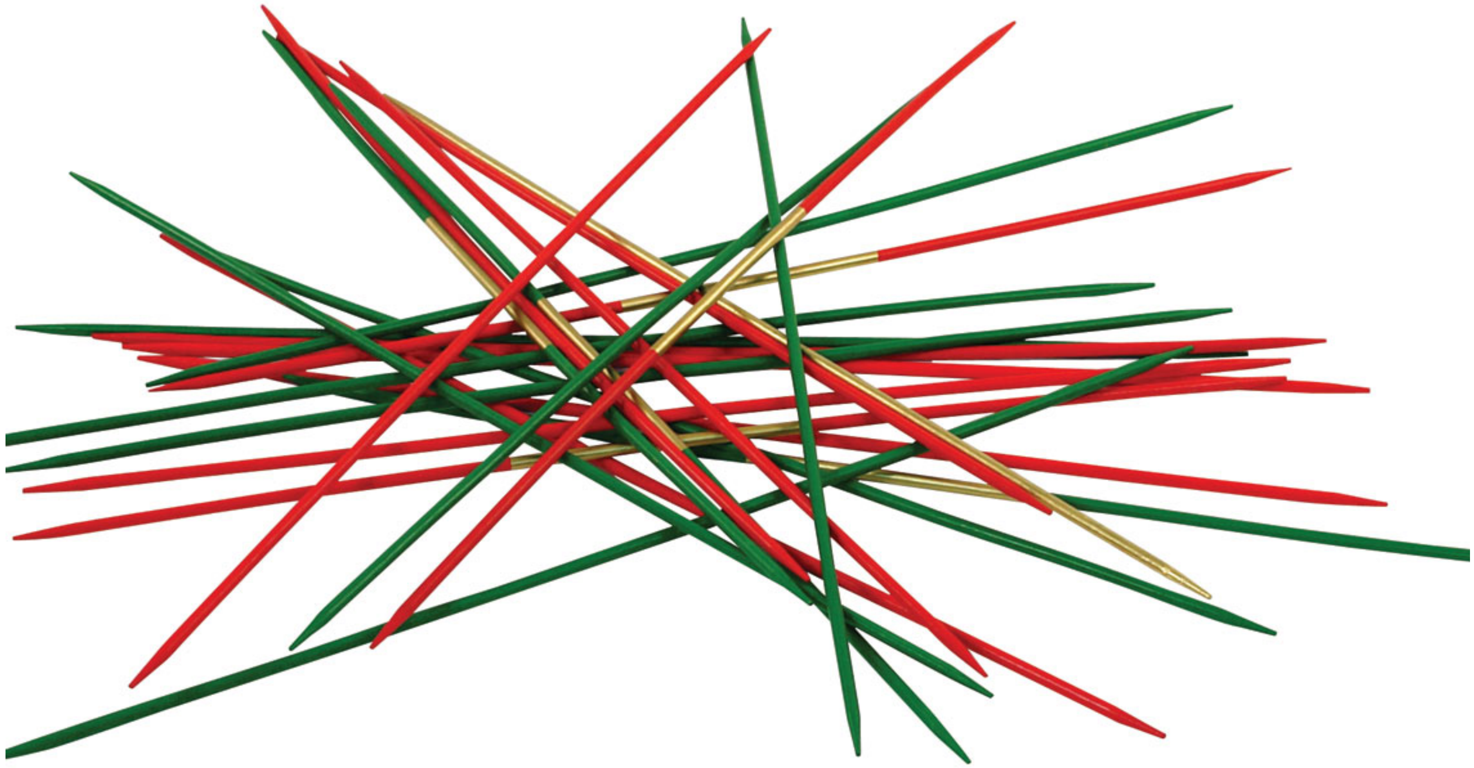
“There is an execution order that can finish”

More specifically... Search for an order P1, P2, P3, ... such that:

- P1’s max resource needs ≤ what it has + what’s free
- P2’s max resource needs ≤ what it has + what’s free + what P1 will release when it finishes
- P3’s max resource needs ≤ what it has + what’s free + what P1 and P2 will release when they finish
- ...

But how do we find that order?

Inspiration



Playing Pickup Sticks with Processes

Pick up a stick on top

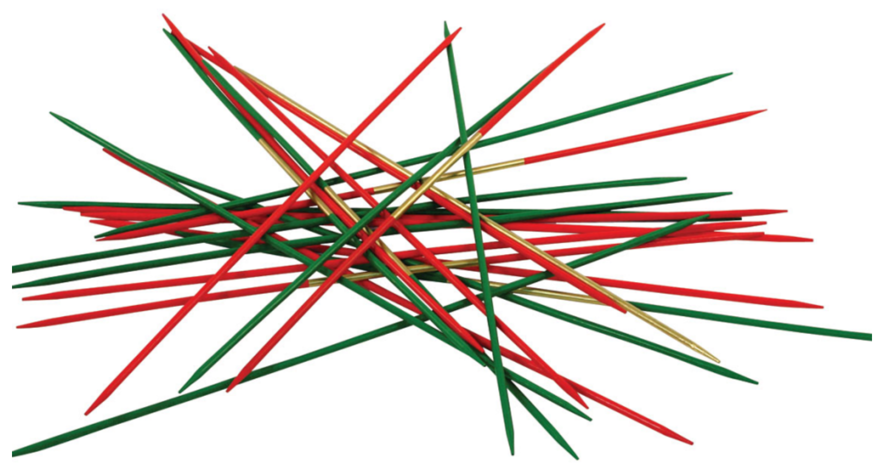
- = Find a process that can finish with what it has plus what's free

Remove stick

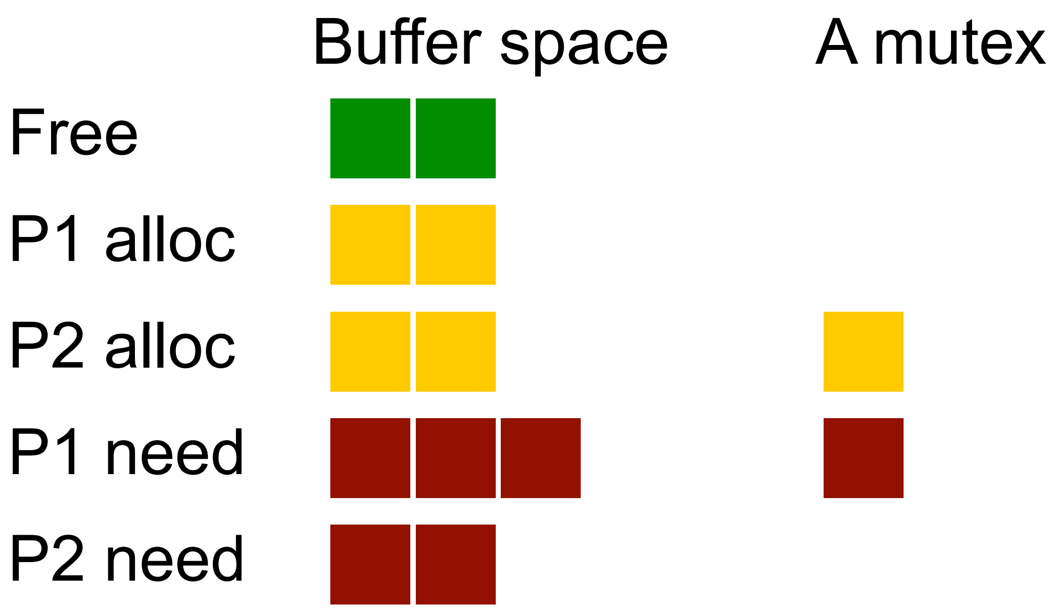
- = Process finshes & releases its resources

Repeat until...

- ...all processes have finished
  - Answer: safe
- ...or we get stuck
  - Answer: unsafe

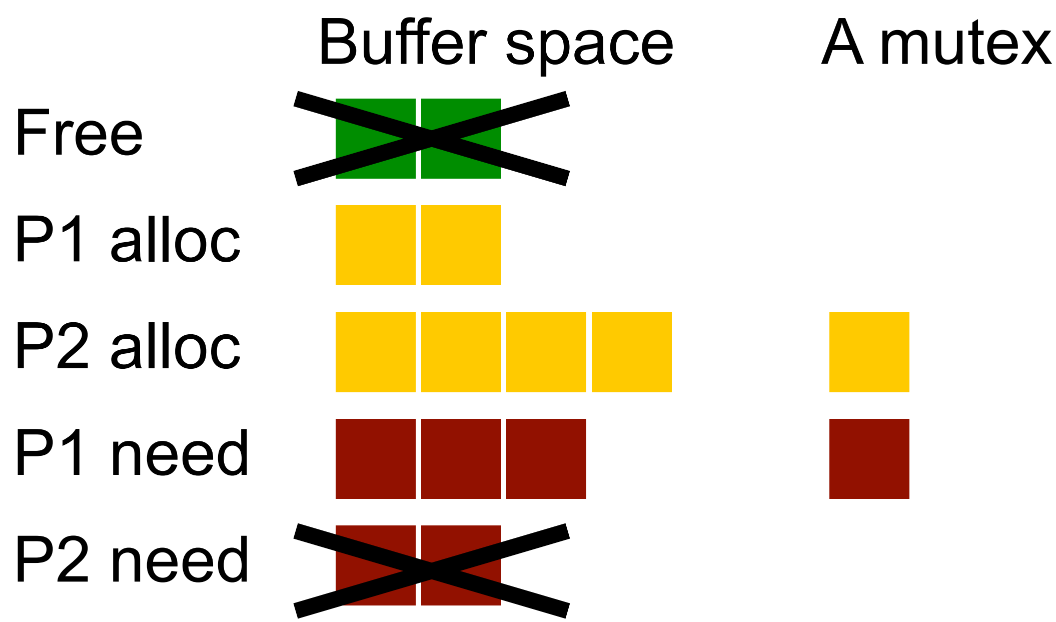


Try it: is this state safe?



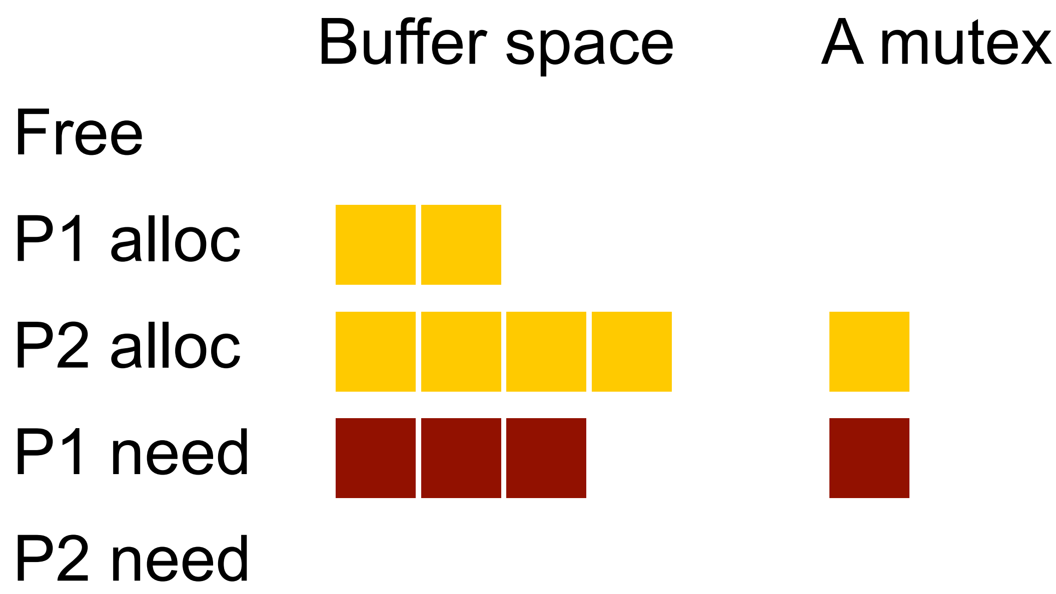
Which process can go first?

Try it: is this state safe?



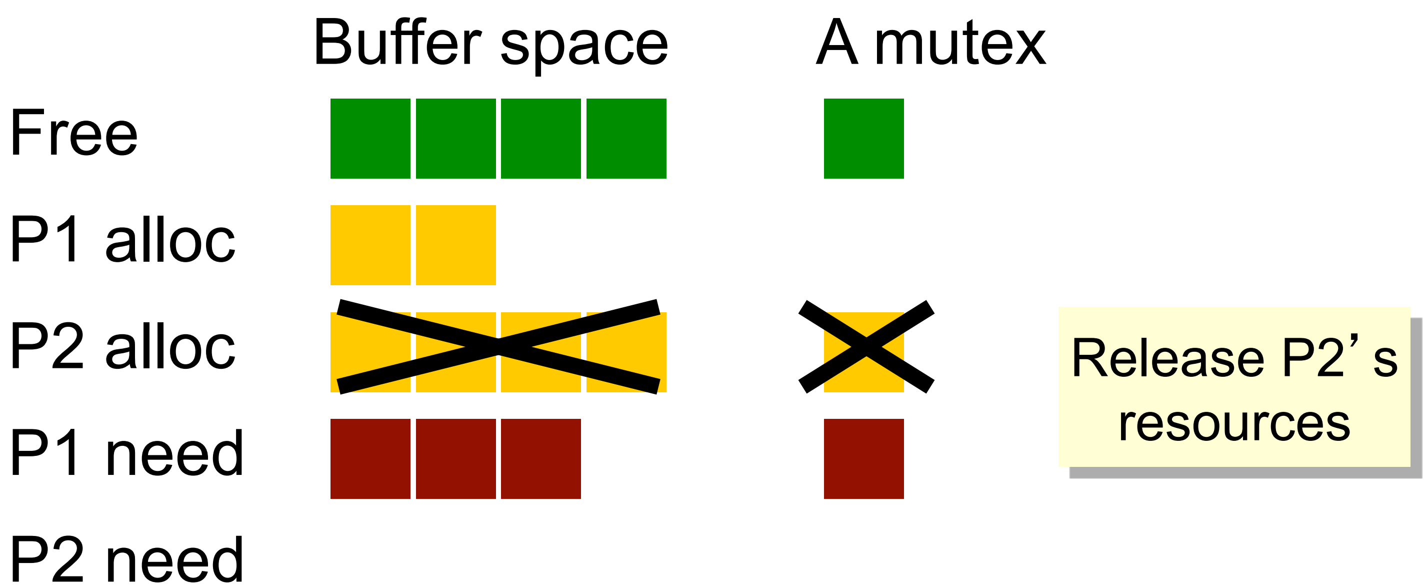
Start with P2

Try it: is this state safe?

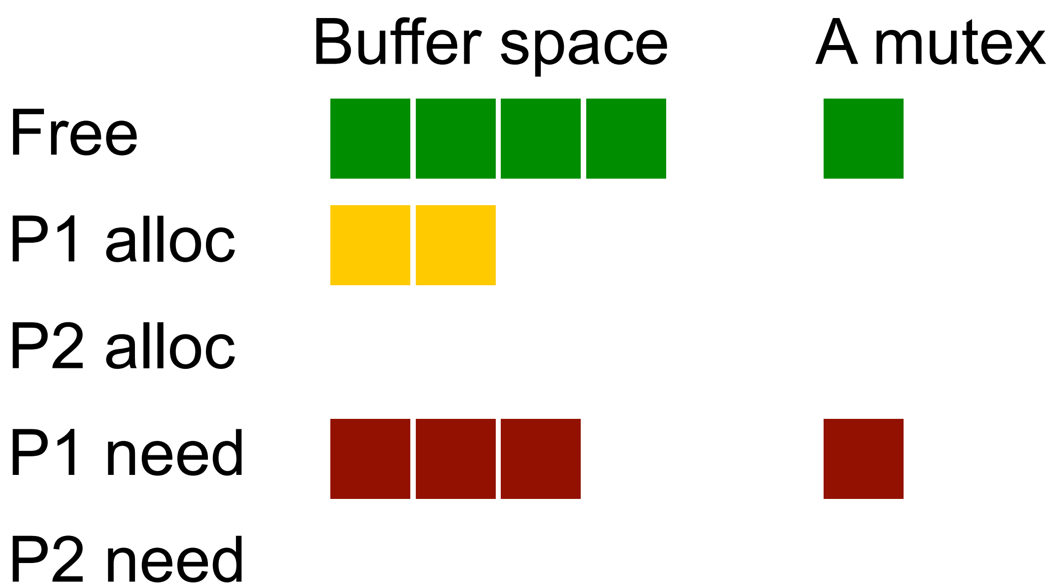


Release P2's resources

Try it: is this state safe?

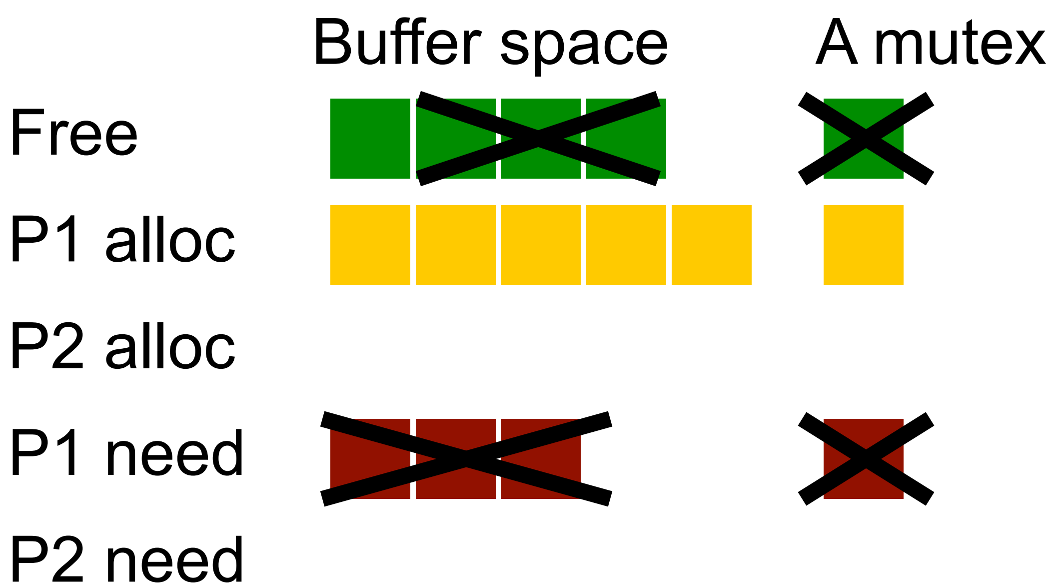


Try it: is this state safe?



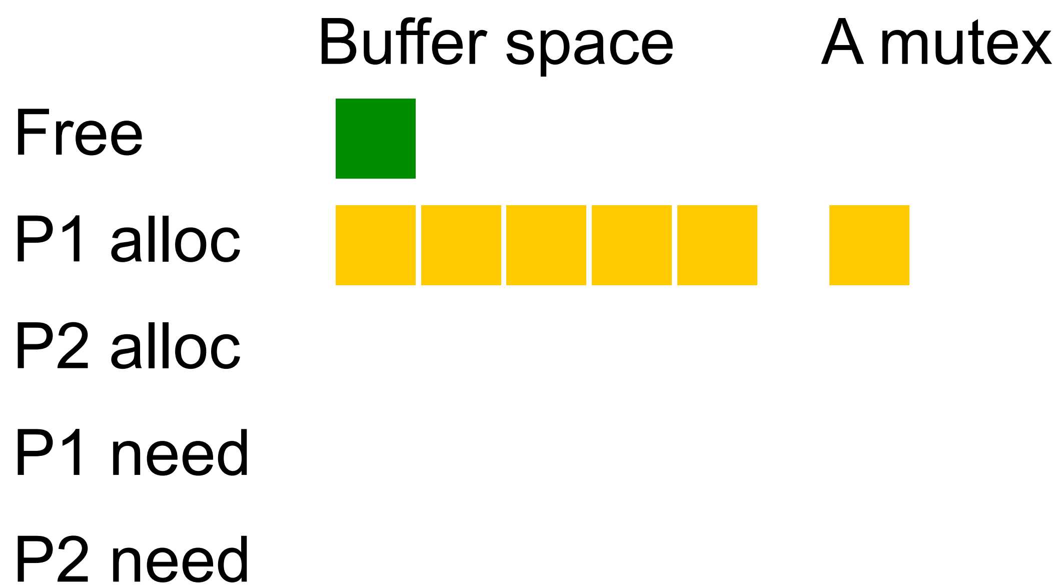
Continue with P1

Try it: is this state safe?



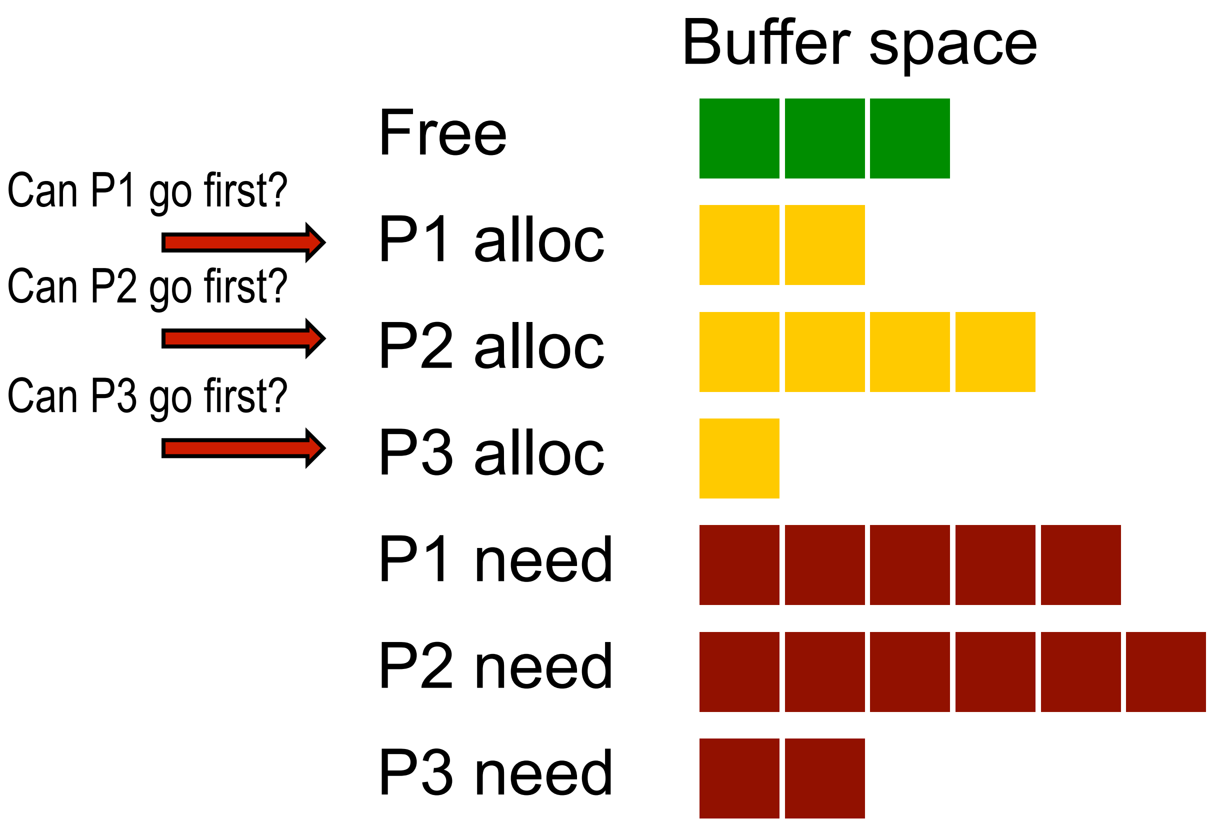
Continue with P1

Try it: is this state safe?

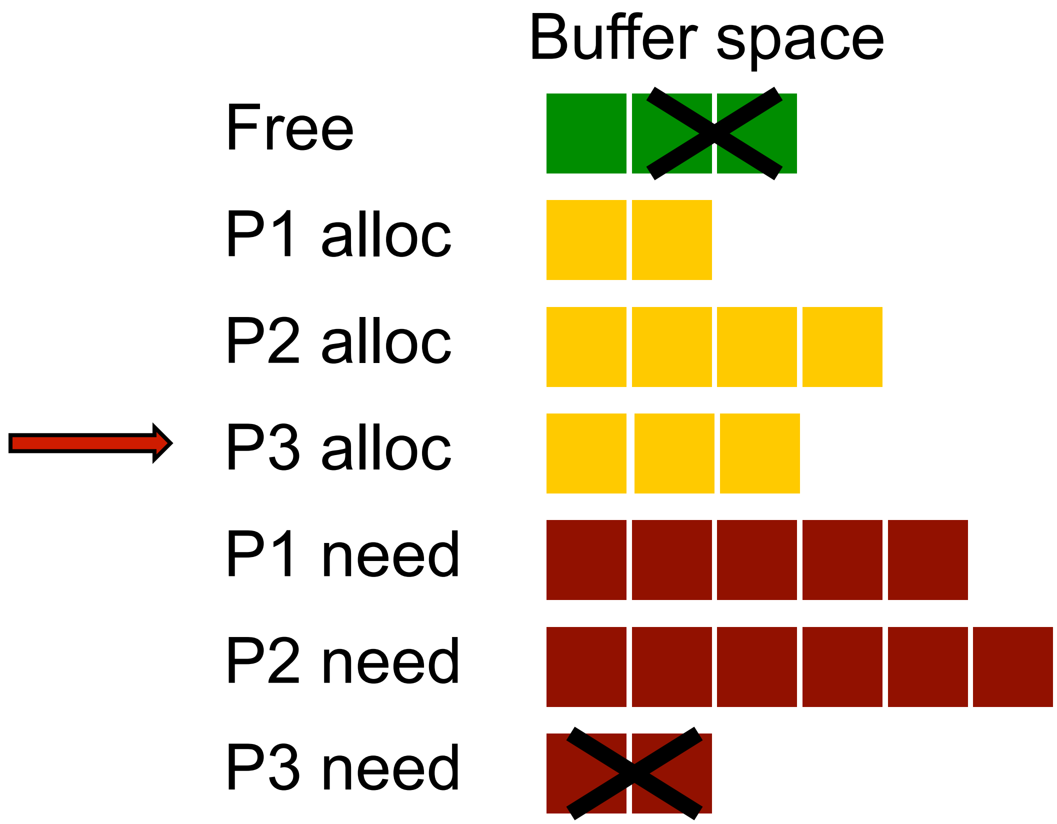


Yes, it’s safe: Order is P2, P1

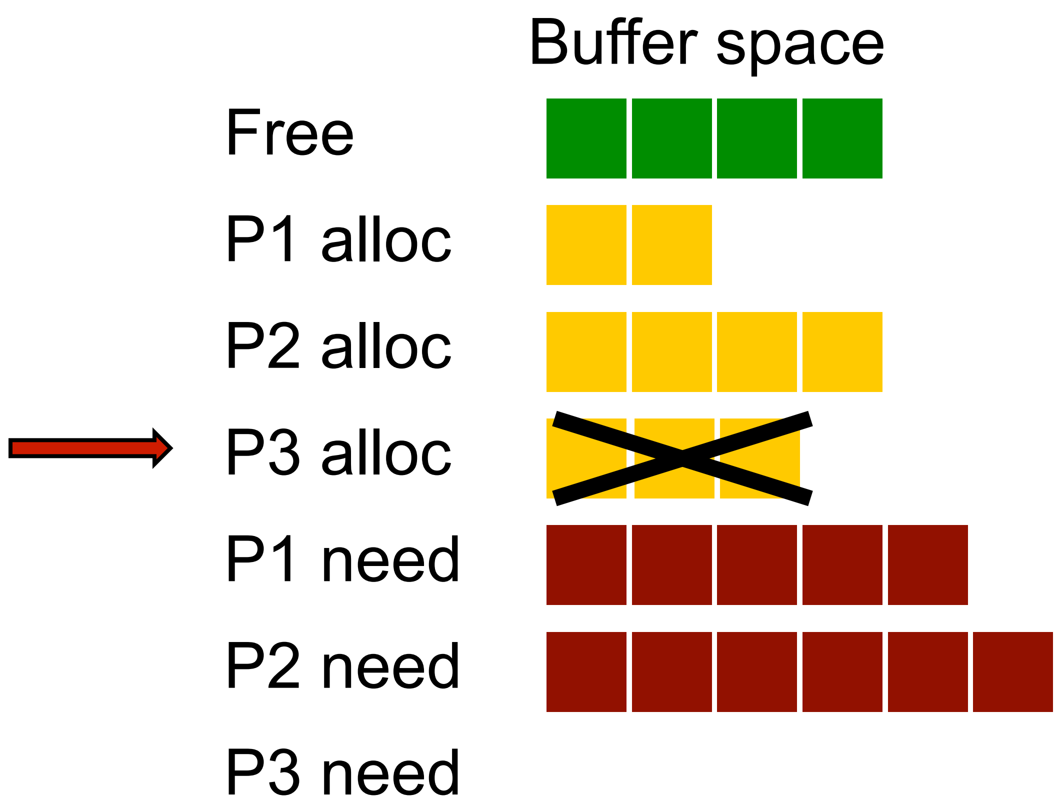
Example 2: Is this state safe?



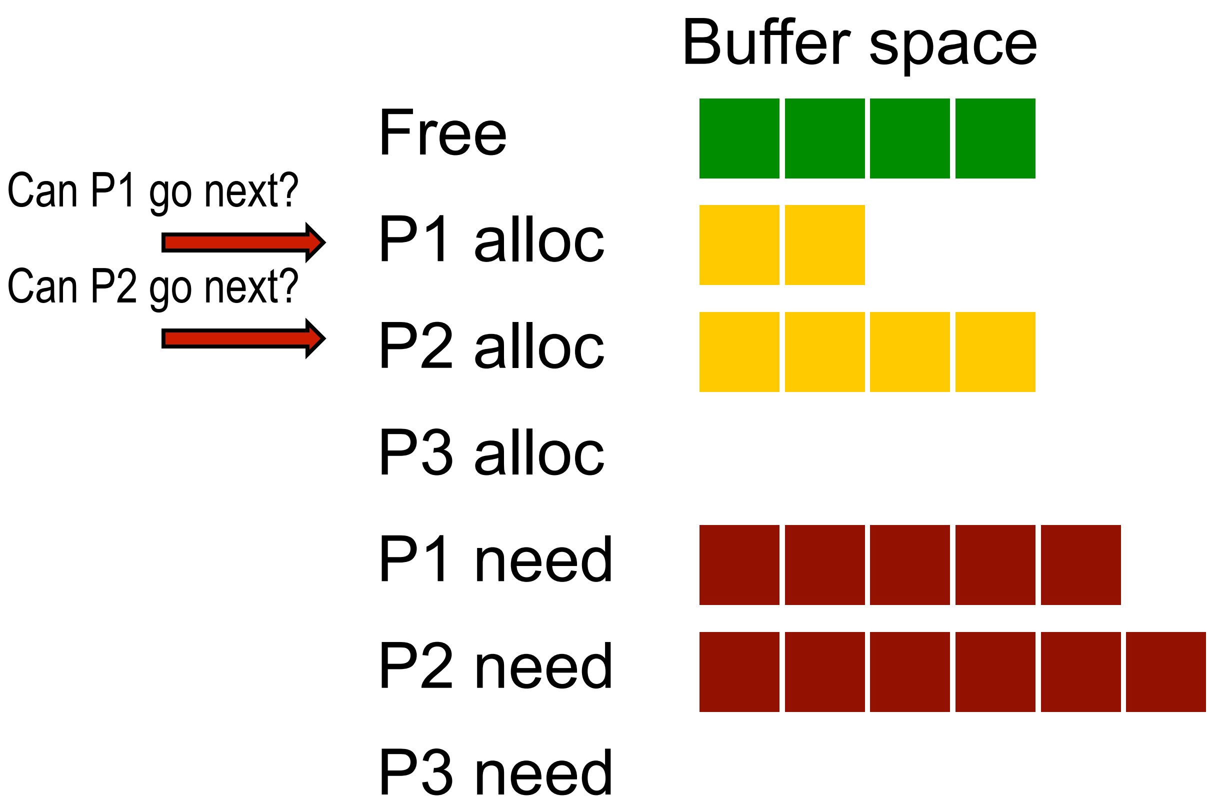
Example 2: Is this state safe?



Example 2: Is this state safe?



Example 2: Is this state safe?



Unsafe!