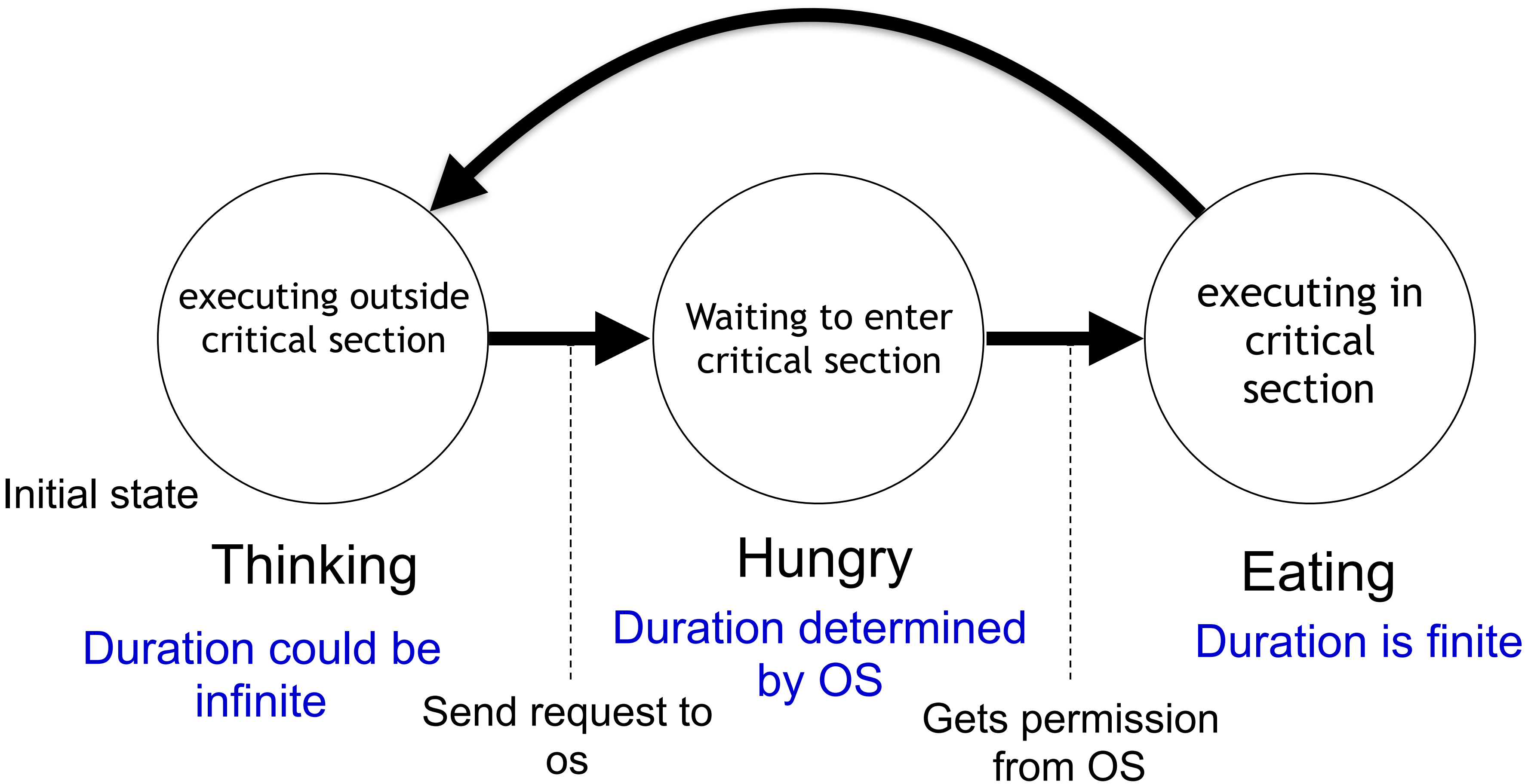
# Distributed Dining Philosophers

CS172
K. Mani Chandy
California Institute of Technology

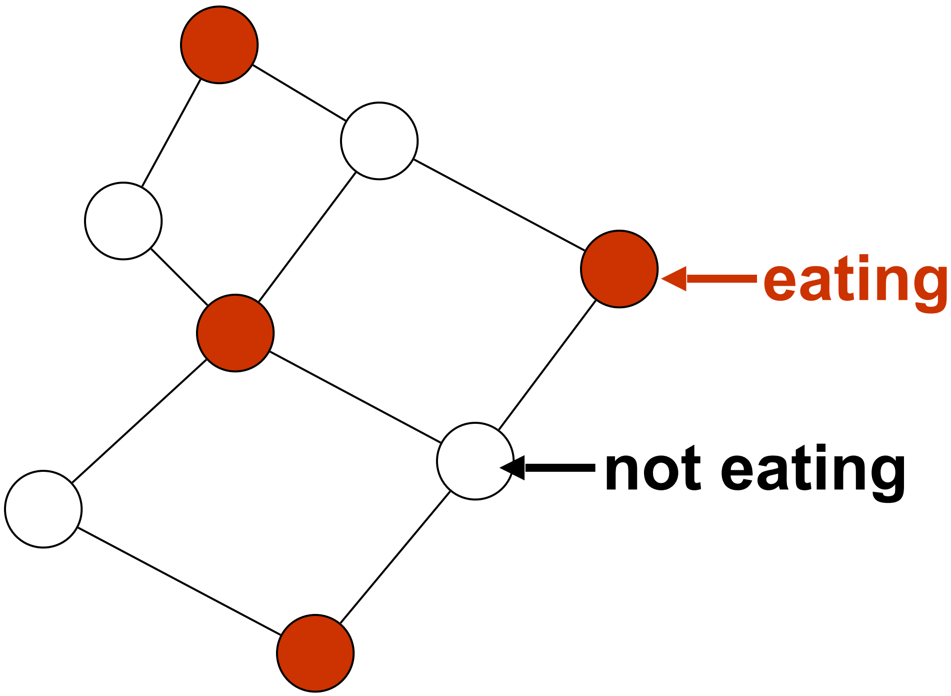
## Key ideas of distributed dining philosophers algorithm

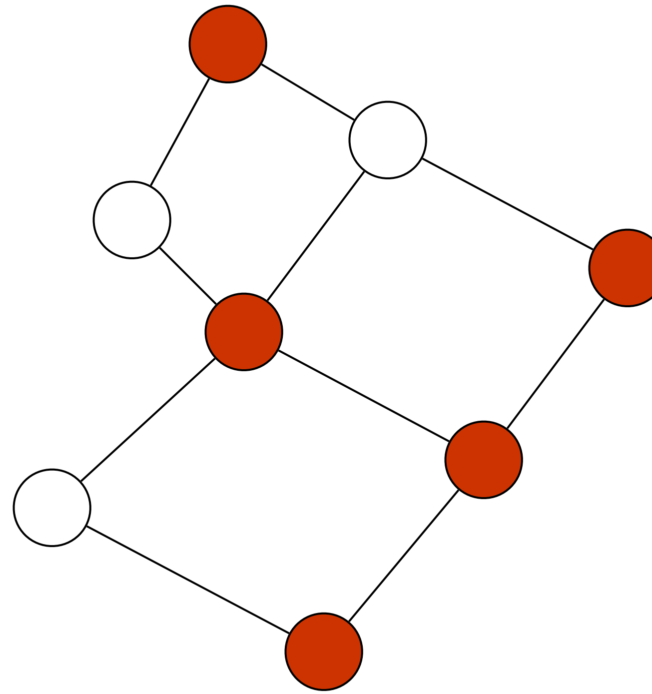
1. **Conflict resolution in distributed systems**.
2. **Priority among agents in conflict.** Some agents win and others lose. Fair winning: every agent that wants to win gets to win *eventually*.
3. **Tokens.** An agent that holds a token knows that other agents don’t hold the same token.
4. **Dynamic Data Structures**: Acyclic graph structures maintained by actions of all agents.

# Client Life Cycle for Dining Philosopher



# Safety property: Always neighbors aren't eating

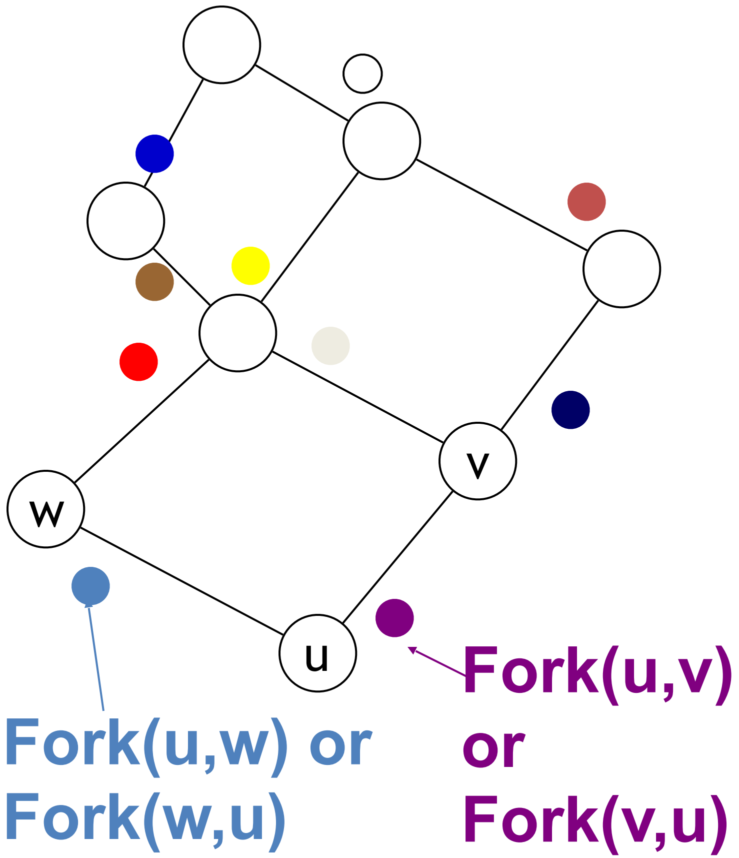


Safety property satisfied:
Neighbors aren't eating

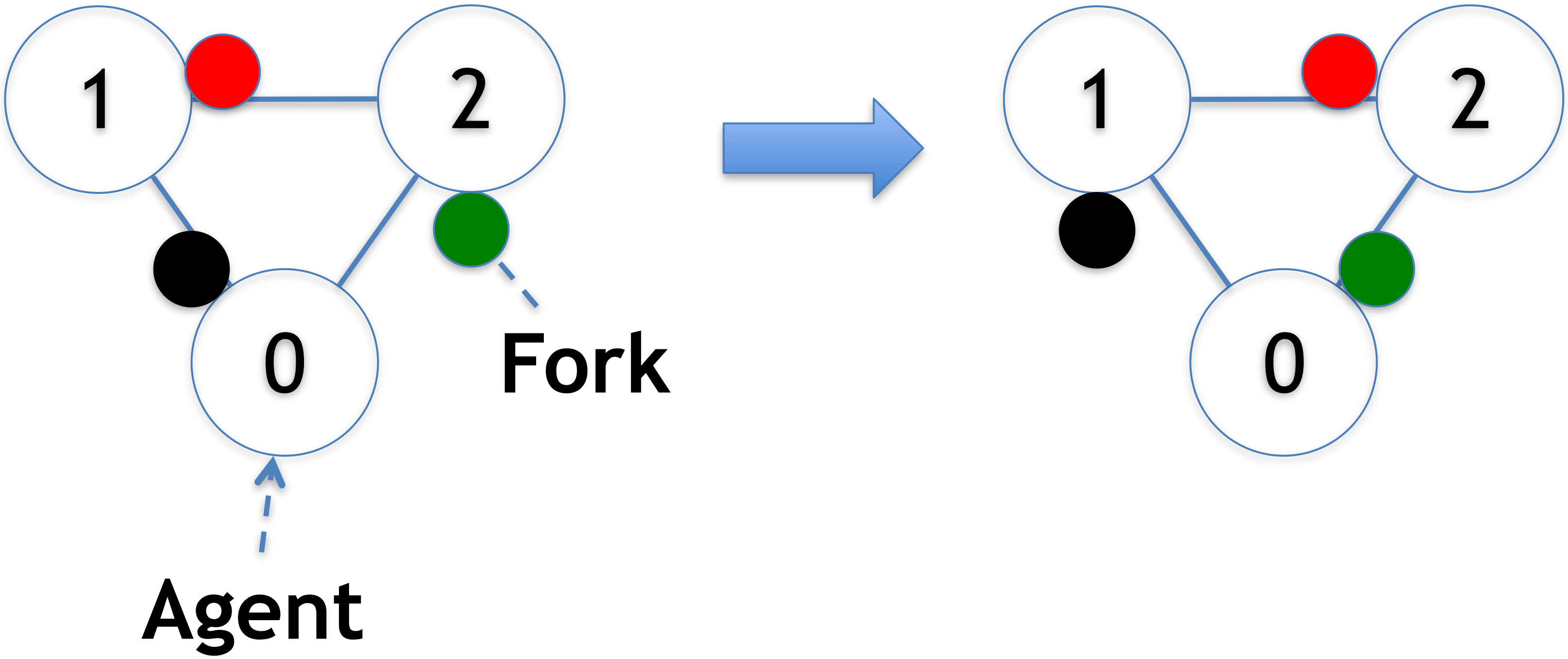
Safety property violated
Neighbors are eating

# Another example of tokens: Introduction of forks




- There is exactly one fork on each edge.
- Forks on different edges have different colors: Color (u,v) is different from color (u,w).
- A fork on an edge (u, v) is at u or at v or in the channel from u to v or in the channel from v to u.
- Philosopher eats only if it holds all its forks.
- Safety property satisfied

# Hungry agents give forks to requestors




| | |
|---|---|
| Agent j holds fork it shares with agent (j+1) mod 3 | Agent j holds fork it shares with agent (j-1) mod 3 |

**Conflict resolution** What to do when multiple agents want the same resource at the same time?

There have to be winners and losers.

Fair winning: Every agent that wants to win gets to win eventually.

**Conflict resolution** What to do when multiple agents want the same resource at the same time?

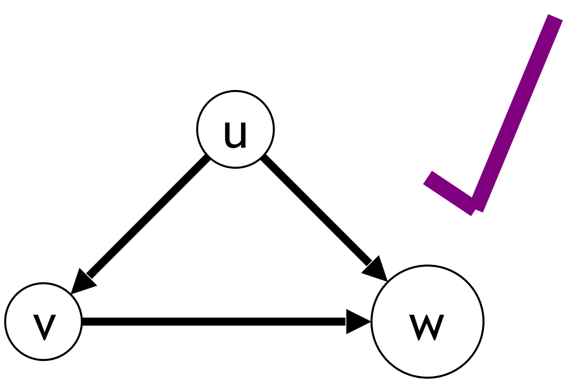
There have to be winners and losers.

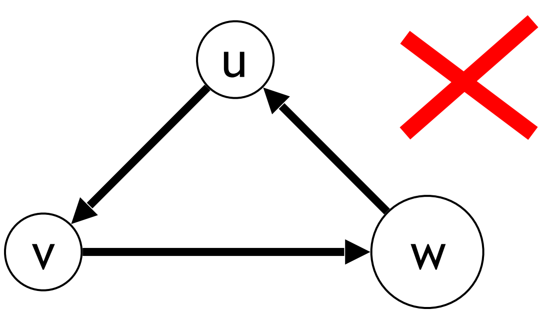
*If the state is symmetric, with all agents exactly like each other, then the state can remain symmetric for ever.*

So, ensure that there is a priority structure which is asymmetric, e.g., there is an agent with highest priority.

**Conflict resolution** What to do when u and v want fork(u,v) at the same time?
**Priority**: Give the fork to the agent with higher priority.

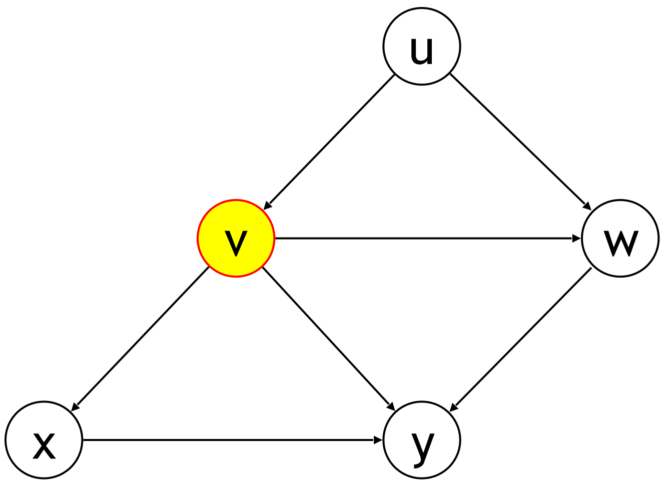
- The vertices of a priority graph represent agents.
- The directed edges represent priority. There is an edge (u, v) exactly when agent u has priority over agent v.
- Maintain the invariant that the priority graph is acyclic. Why? Because a symmetric state can persist forever.



# How should priorities change when a process eats?

v holds all its forks and eats

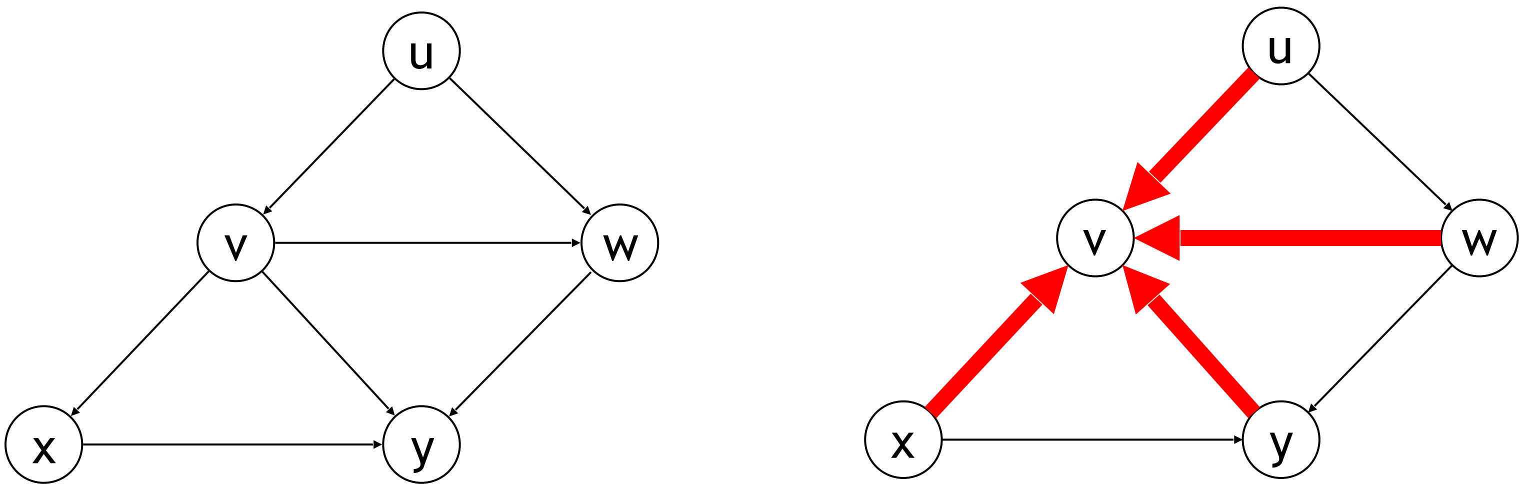



What should happen to edge directions after v eats?

- Flip edges incident on v?
- Make all edges directed towards v?

# How should priorities change when a process eats?

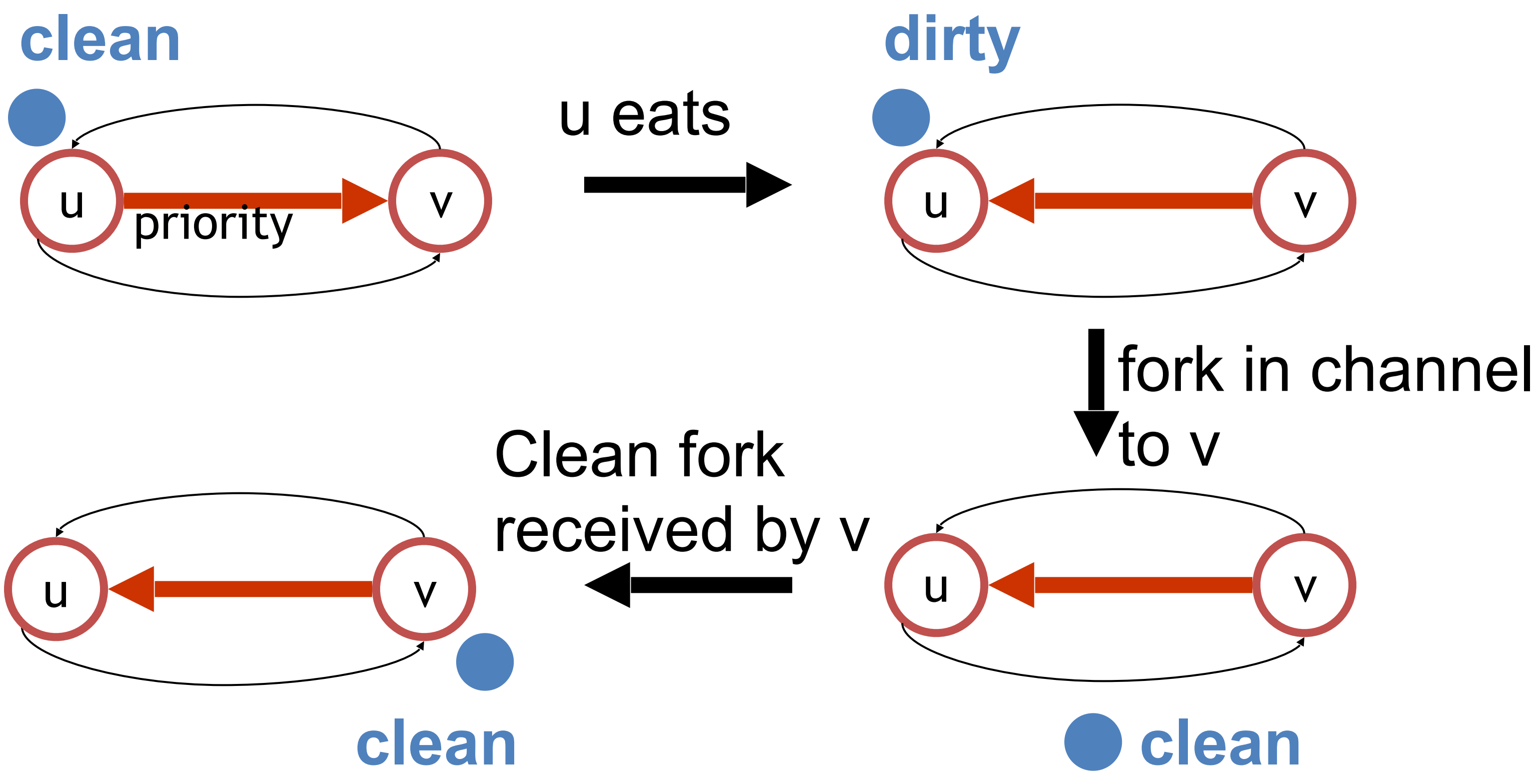
v holds all its forks and eats




What should happen to edge directions after v eats?

- Flip edges incident on v? No. may cycle.
- Make all edges directed towards v? Yes. Prove that the graph remains acyclic.

**How can we represent priorities in terms of forks?**
**All forks held by an eating agent are dirty.**
**An agent holding a dirty fork has lower priority.**




**Priority changes only when a clean fork becomes dirty**

**Proposal for an algorithm**

- An **eating** philosopher that gets a request for a fork does what?

**Proposal for an algorithm**

- An eating philosopher that gets a request for a fork does what?

**Finishes eating (in finite time) and gives the cleaned fork to requester**

## Proposal for an algorithm

A **thinking** philosopher v that gets a request for a fork does what?

**Always:** Thinking philosophers hold only *dirty* forks.

If v is thinking then the fork that it shares with a neighbor is:

1. at v *and dirty* or
2. at w or in the channel to w.

## Proposal for an algorithm

A **thinking** philosopher v that gets a request for a fork does what?

<u>**Always:** Thinking philosophers hold only *dirty* forks.</u>

**Cleans the dirty fork and gives it to the requester.**

**Proposal for an algorithm**

- A hungry philosopher that gets a request for a fork does what?

**Proposal for an algorithm**

- A hungry philosopher that gets a request for a fork does what?

**If the fork is clean then holds on to the request and the fork.** (Does not give the fork because the holder has higher priority.)

**If the fork is dirty then gives the cleaned fork to the requester.** (Yields the fork because the requester has higher priority.)

## Proposal for an algorithm

- When hungry philosopher holds all its forks then what happens?

**Proposal for an algorithm**

- When hungry philosopher holds all its forks then what happens?

If the hungry philosopher holds a request for a dirty fork it gives the dirty fork (after cleaning it) to the requester.

**Otherwise, the hungry philosopher starts eating and dirties all the forks that it holds.**
(So, the eating philosopher has lower priority than its neighbors.)

## Proposal for an algorithm

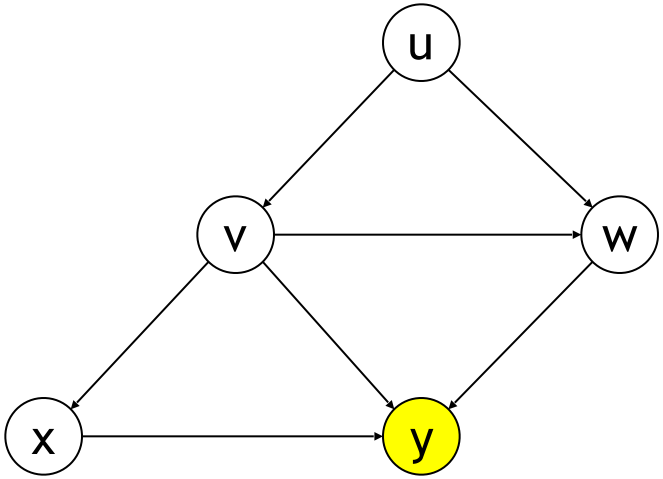
Philosopher yields a requested fork if:
*the philosopher is not eating and the fork is dirty*

## Proposal for an algorithm

- An eating philosopher that gets a request for a fork sends the fork when it finishes eating. If it does not get a request for a fork then when it transits to thinking it continues to hold on to the (dirty) fork.
- A thinking philosopher that gets a request for a fork sends the fork.
- A hungry philosopher that gets a request for a fork sends the fork if the fork is dirty. It holds on to the fork if the fork is clean.
- A hungry philosopher transits to eating if it holds all forks and it does not have a request for a fork which is dirty.

Is the algorithm correct? Safety: obvious. Progress? Not clear

# Can a philosopher remain hungry for ever?




Could a cabal of philosophers eat repeatedly and cause others to starve for ever? For example, could philosopher y remain hungry forever because u, v, w eat repeatedly, one after the other?

Could hungry philosophers forever hold some, but not all, forks that they need to eat?

To prove progress find a variant function f such that for all k:

**(v.hungry and f = k)      leads-to**

**(v.eating or (v.hungry and f < k))**

To prove progress find a variant function f such that for all k:

**(v.hungry and f = k)      leads-to**

**(v.eating or (v.hungry and f < k))**

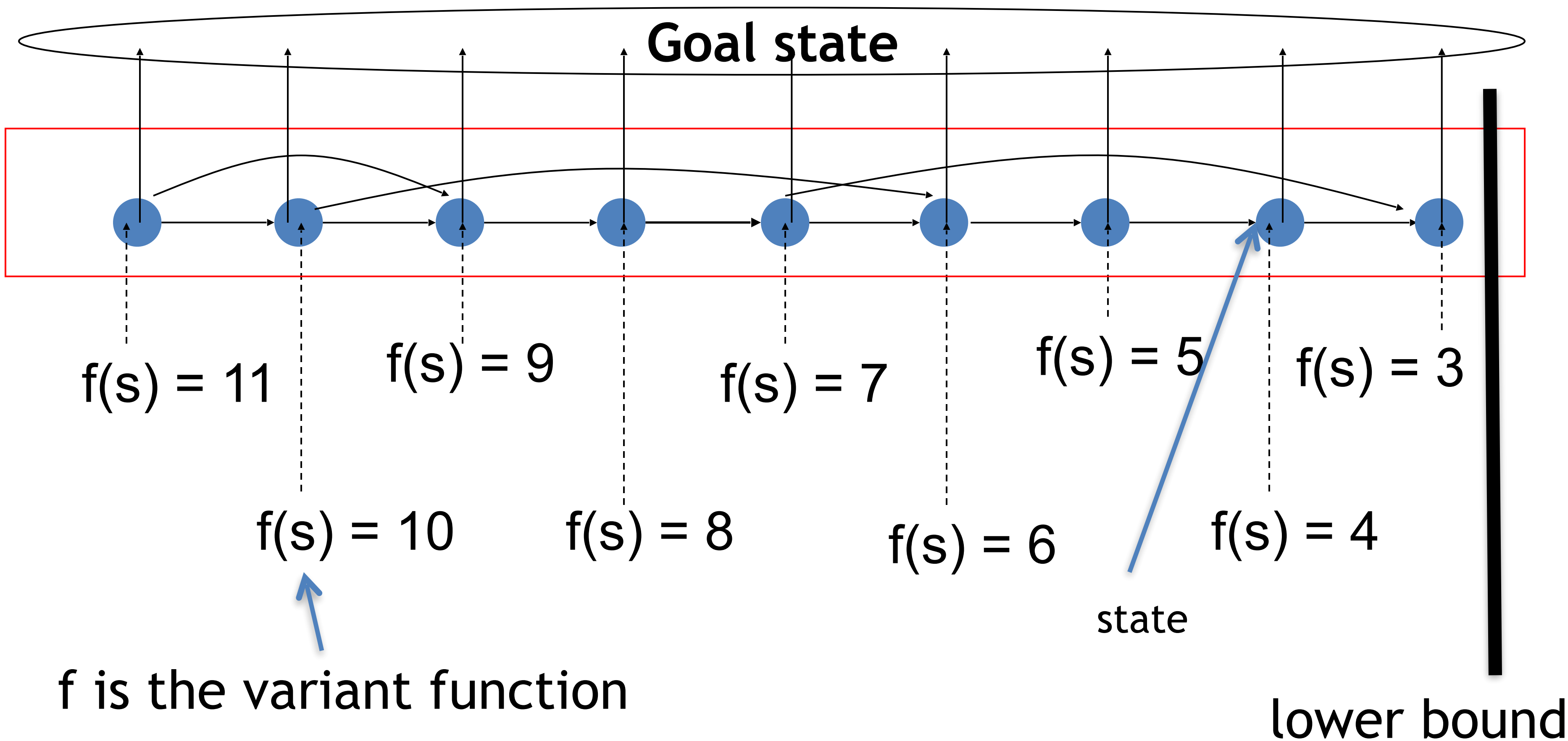
One way to prove this is to show:

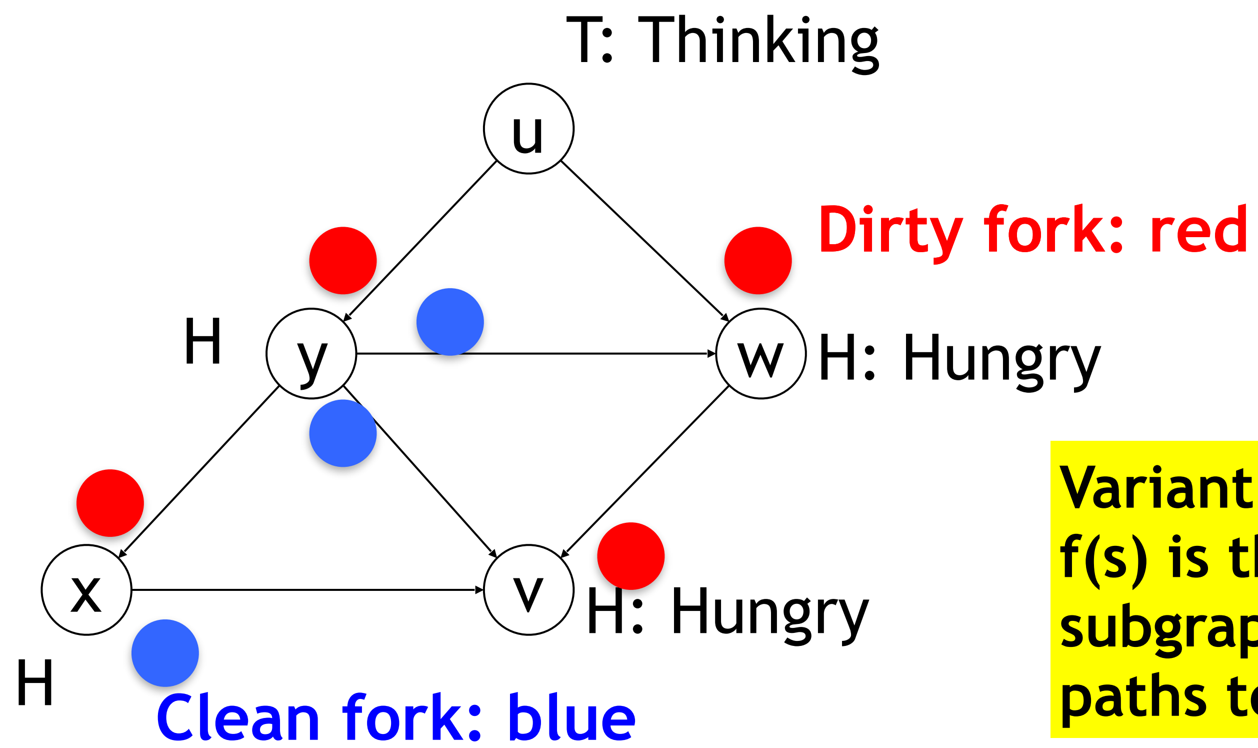
1. For all statements s:

{v.hungry and f=k} s {v.eating or (v.hungry and f <= k)}

2. (v.hungry and f=k)  leads-to NOT(v.hungry and f=k)

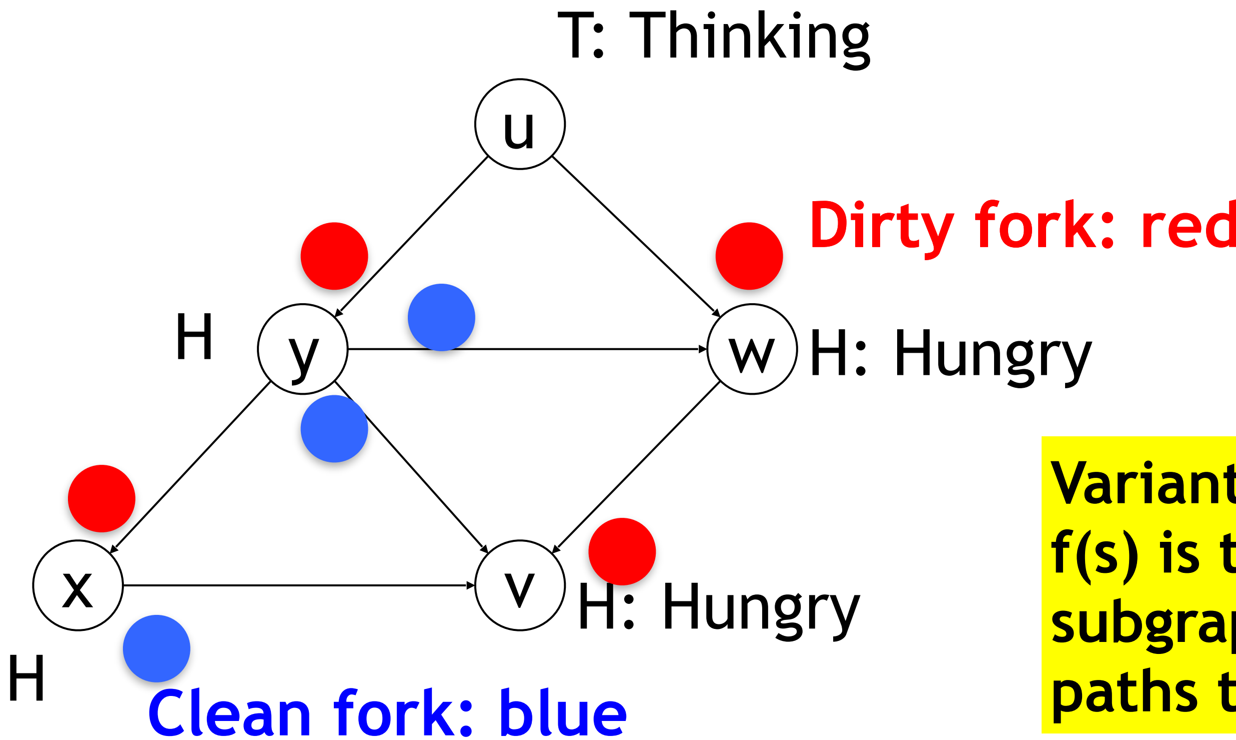
# The pictorial idea of the progress proof






**Variant Function**
**f(s) is the directed acyclic subgraph of vertices with paths to v**

**Example Priority Graph.**
Forks located at vertices; blue for clean; red for dirty.

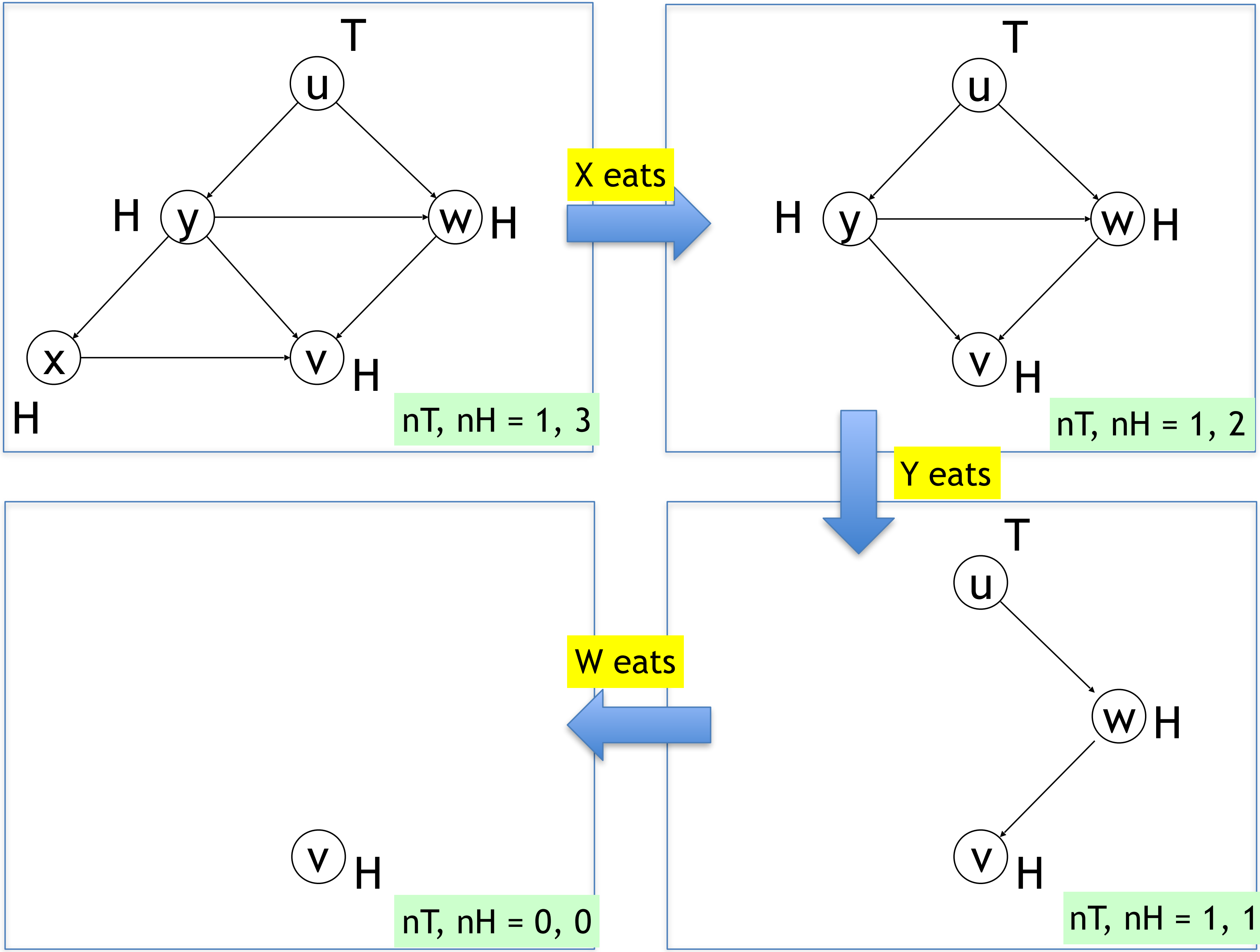



**Variant Function**
**f(s) is the directed acyclic subgraph of vertices with paths to v**

**Example Priority Graph.**
Forks located at vertices; blue for clean; red for dirty.

Variant function to prove that v will eat eventually is: (nT, nH), the number of thinking and hungry philosophers with paths to v.

nT = 1 because of vertex u
nH = 3 because of vertices w, x, y



T
u
H
y
w
H
x
v
H
H
nT, nH = 1, 3
X eats
T
u
H
y
w
H
v
H
nT, nH = 1, 2
Y eats
T
u
w
H
v
H
nT, nH = 1, 1
W eats
v
H
nT, nH = 0, 0

One way to prove this is to show:

1. For all statements s:

{v.hungry and f=k} s {v.eating or (v.hungry and f <= k)}

How do we do this when f = (nT, nH)?

One way to prove this is to show:

1. For all statements s:

{v.hungry and f=k} s {v.eating or (v.hungry and f <= k)}

How do we do this when f = (nT, nH)?

nT does not increase because no path to v is created while v remains hungry.

nH increases only if nT decreases.

One way to prove this is to show:

1. For all statements s:

{v.hungry and f=k} s {v.eating or (v.hungry and f <= k)}

2. (v.hungry and f=k) leads-to NOT(v.hungry and f=k)

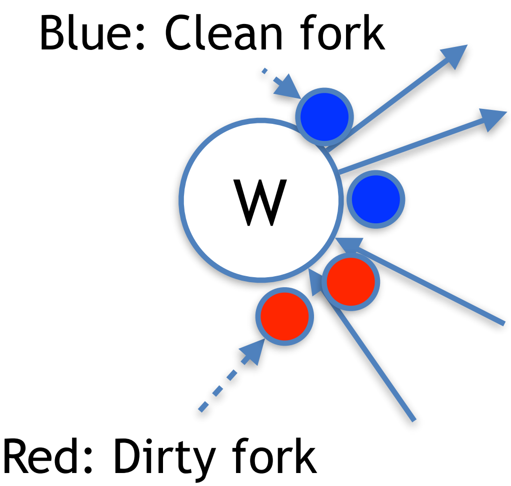
How do we do this when f = (nT, nH)?

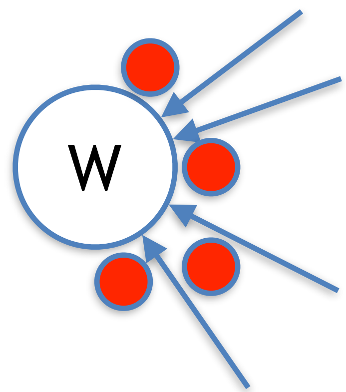
1. For all statements s:

{v.hungry and f=k} s {v.eating or (v.hungry and f <= k)}

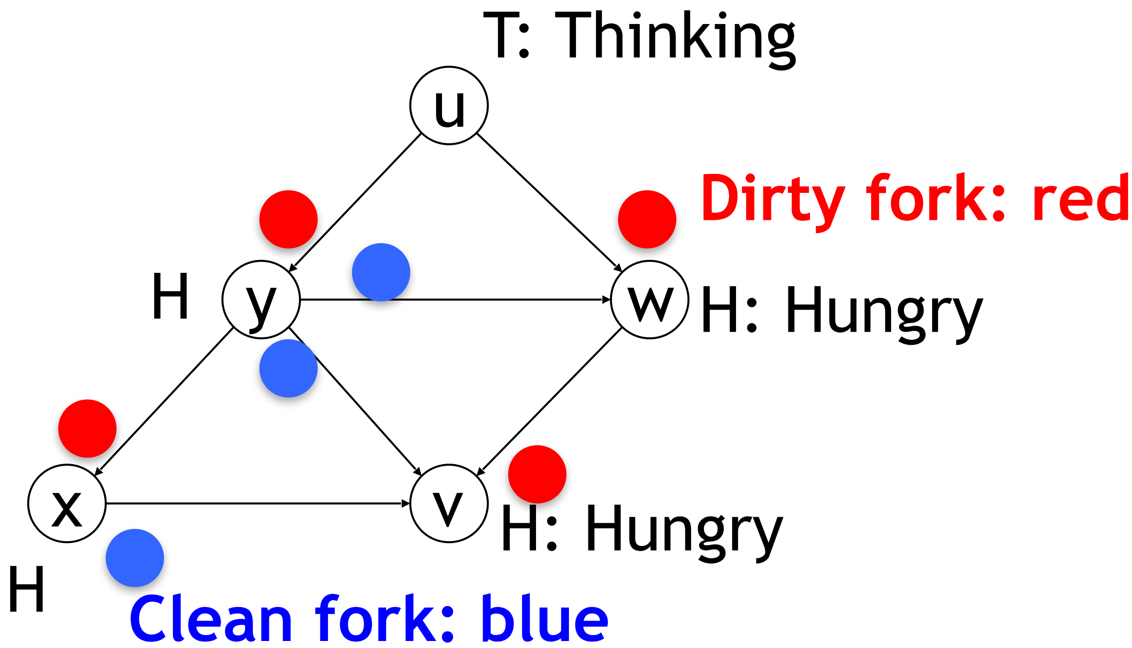
Show that (nT, nH) cannot increase while v remains hungry

Any change to the priority graph does not increase nT, or nT + nH




**The only change to the priority graph directs all edges incident on a vertex towards that vertex. So, if v remains hungry, then changes to the priority graph do not create paths from any vertex to v.**

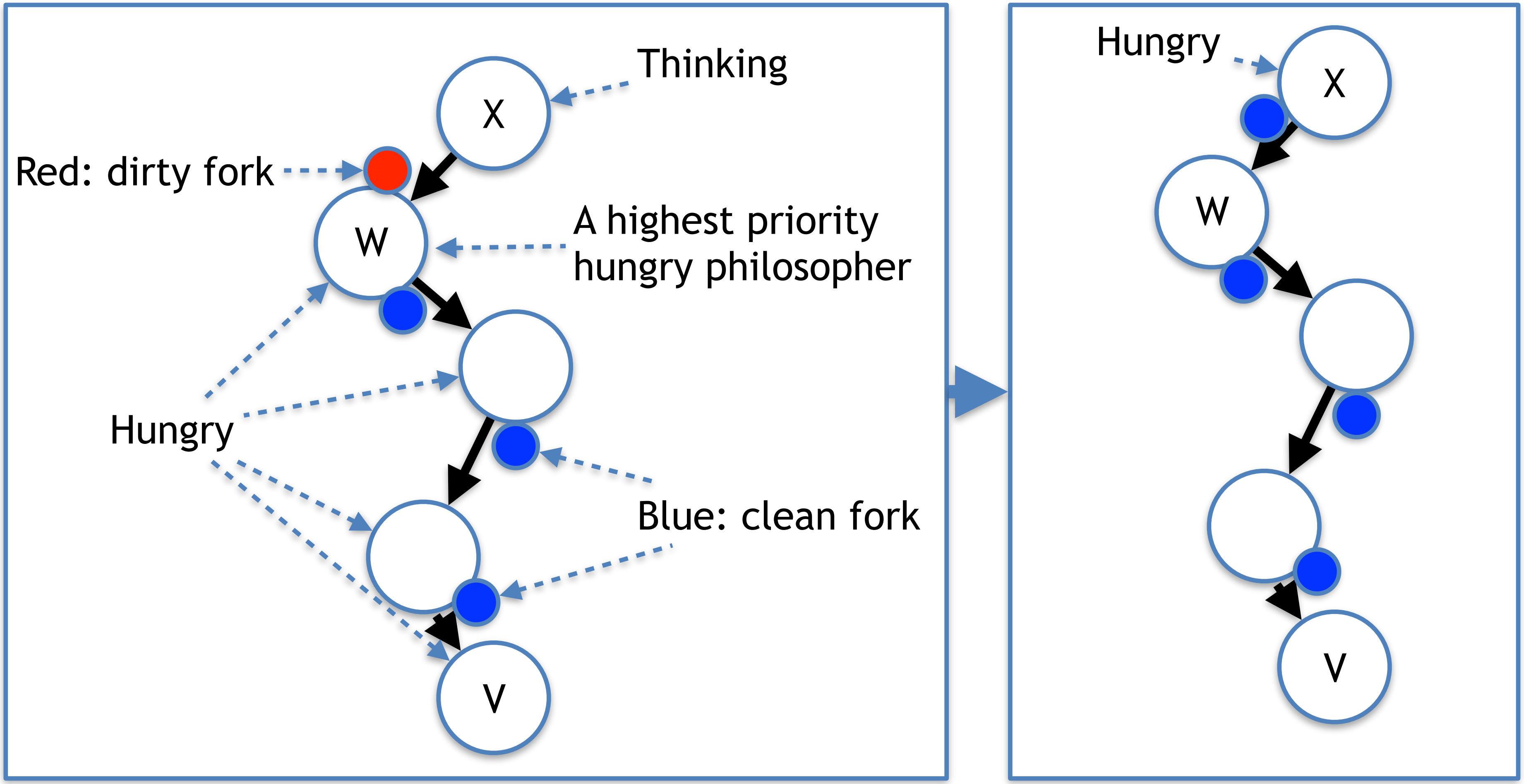



*2. (v.hungry and f=k) leads-to NOT(v.hungry and f=k)*

How do we prove this when f = (nT, nH)?

Eventually highest priority hungry philosopher (e.g, y) gets all its forks, or

a higher priority thinking philosopher becomes hungry (see next slide).




In the left-hand diagram, W is the highest priority hungry philosopher.

A higher priority thinking philosopher (e.g. X) can become hungry (right-hand diagram). nT decreases, and nH increases, but (nT, nH) decreases.

(Note: X can get the fork from W before W eats in which case W eats only after X.)

## Key ideas of distributed dining philosophers algorithm

1. **Conflict resolution in distributed systems**.
2. **Priority among agents in conflict.** Some agents win and others lose. Fair winning: every agent that wants to win gets to win *eventually*.
3. **Tokens.** An agent that holds a token knows that other agents don't hold the same token.
4. **Dynamic Data Structures**: Acyclic graph structures maintained by actions of all agents.