# SCC.211 Operating Systems

## Lecture 7 – Common Problems in Concurrent Programs

Amit K. Chopra
School of Computing & Communications, Lancaster University, UK
amit.chopra@lancaster.ac.uk

# Objectives

- Common pitfalls in concurrent programming
- Race conditions
- Deadlock
- Livelock
- Dining philosophers

# Race conditions

It’s very easy to introduce subtle programing errors into concurrent programs

Unidentified or incorrectly-protected critical sections leading to race conditions

**Why subtle?**

- In a sequential program, given identical input, things happen deterministically (i.e. in the same order)
- This is not the case in a concurrent program (sometimes work)
- This has led to the design of special concurrent programming languages that try to aid correctness (e.g., Eiffel, CSP)

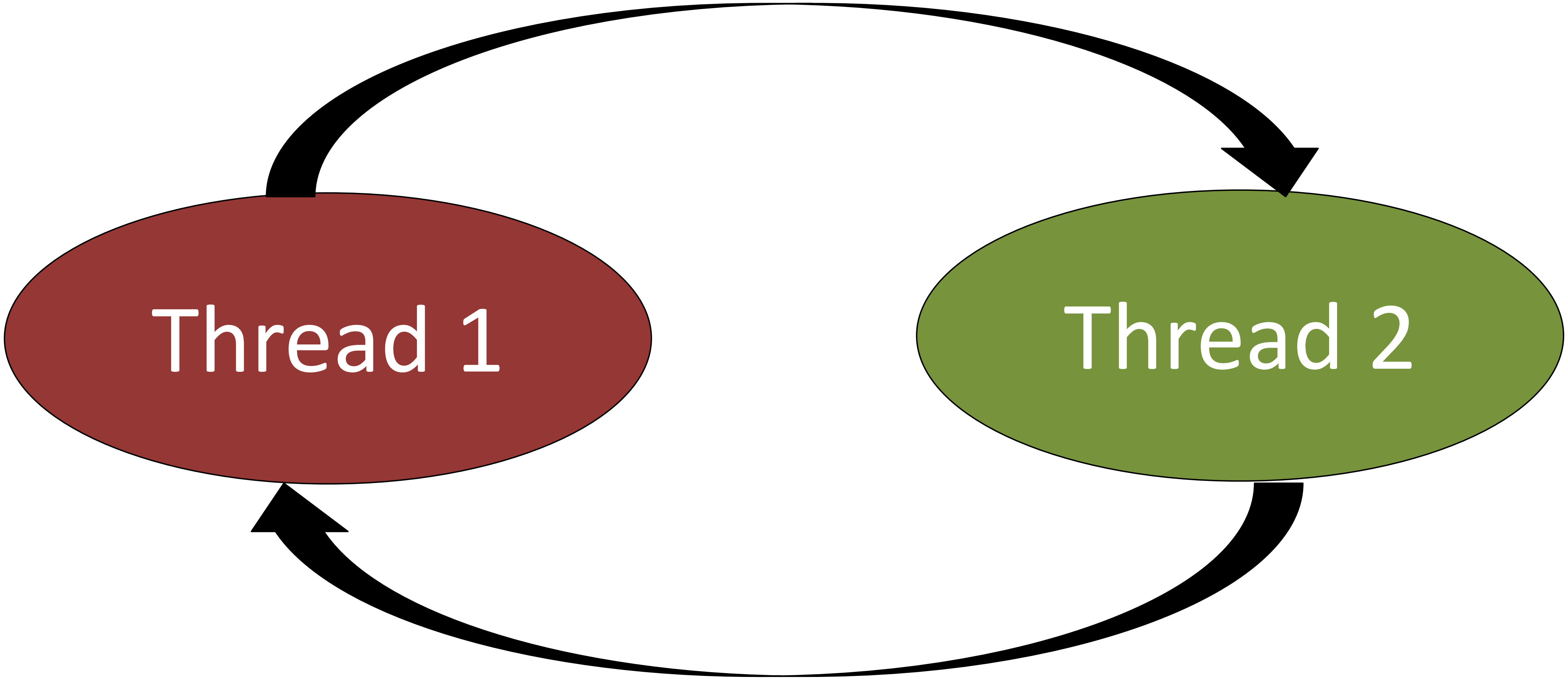
# Deadlock Scenario

Wants the other car to leave so can take the space

Wants the other car to move so can leave the space

# Deadlock

Two or more threads are blocked forever waiting for each other



# Deadlock

1

**Mutual exclusion of resources**

- Processes need exclusive access to the resources they are attempting to obtain

2

**Hold and wait**

- Processes are allowed to hold a resource while waiting for another resource

3

**Non-preemption of resources**

- Resources may not be forcibly taken away from a process

4

**Circular wait**

- Possibility of getting into a cycle of processes where each process waits for a resource held by the 'next' process

# What is the problem?

```
Lock lock;
transfer(Account x, Account y, int amt) {
        acquire(lock);
        x = x – amt;
        y = y+ amt;
        release(lock);
}
```

Bank has millions of accounts, named say A1, A2, …

# What is the problem now?

Assuming a lock per account increase concurrency; however…

```
transfer(Account x, Account y, int amt) {
        acquire(x.lock);
        acquire(y.lock);
         x = x – amt;
        y = y+ amt;
        release(y.lock);
        release(x.lock);
}
```

# Deadlock in Threads

Very typical with incorrect lock ordering

Thread 1

```
public void run() {
  synchronized (lockA) {
          …
          synchronized (lockB) { …}
  }
```

Thread 2

```
public void run() {
  synchronized (lockB) {
          …
          synchronized (lockA) {…}
  }
```

Main

```
Thead1.start();
Thread2.start();
```

# Dealing with Deadlock

1

**Program design so circular wait never occurs**

- Impose total ordering on resources and requiring all processes request resources in that same order

2

**Prove formally a program is deadlock free before running**

- Very difficult, except for small and simple programs
- Satellites, mission critical systems use this…

3

**Detect deadlock at runtime, and try recover automatically**

- Selectively abort processes
- Roll back to an earlier state

4

**Manual corrective action**

- Ctrl + C
- Reboot!
- Kick the machine?

# Deadlocks and Locking Order

```
//Critical section A
Lock1.acquire()
Lock2.acquire()
```

Deadlock

```
//Critical Section B
Lock2.acquire()
Lock1.acquire()
```

```
//Critical section A'
Lock1.acquire()
Lock2.acquire()
```

Safe

```
//Critical Section B'
Lock1.acquire()
Lock2.acquire()
```

# Livelock Intuition

Two cars coming from opposite directions have to cross a single-lane bridge.

Cars get on the bridge, see each other. Each back offs so the other can pass.

*Ad infinitum*

# Livelock

Process can’t make progress for a reason other than deadlock (typically the process is still actively changing)

Unlike deadlock, livelock may eventually resolve spontaneously

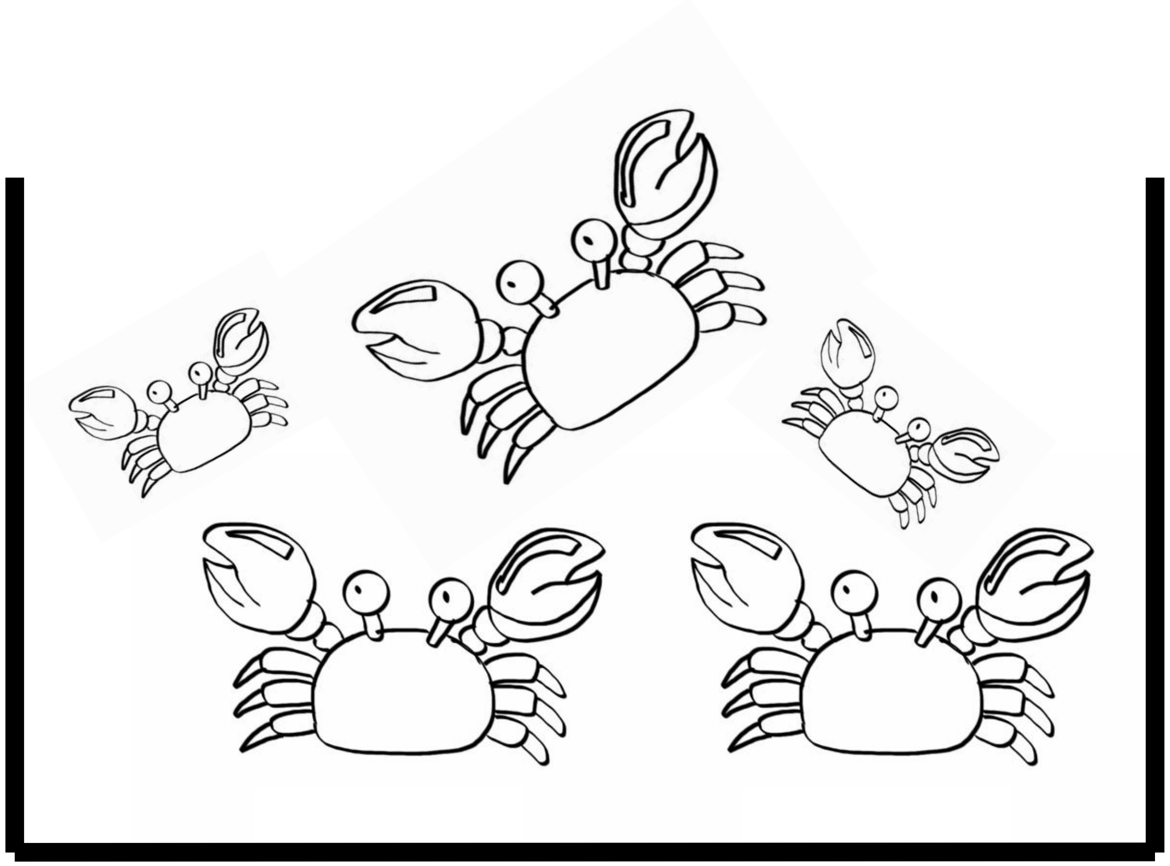
However, often drops into an indefinite livelock ‘groove’, hence livelock is often as undesirable as deadlock

– Can occur if deadlock detection repeatedly triggered

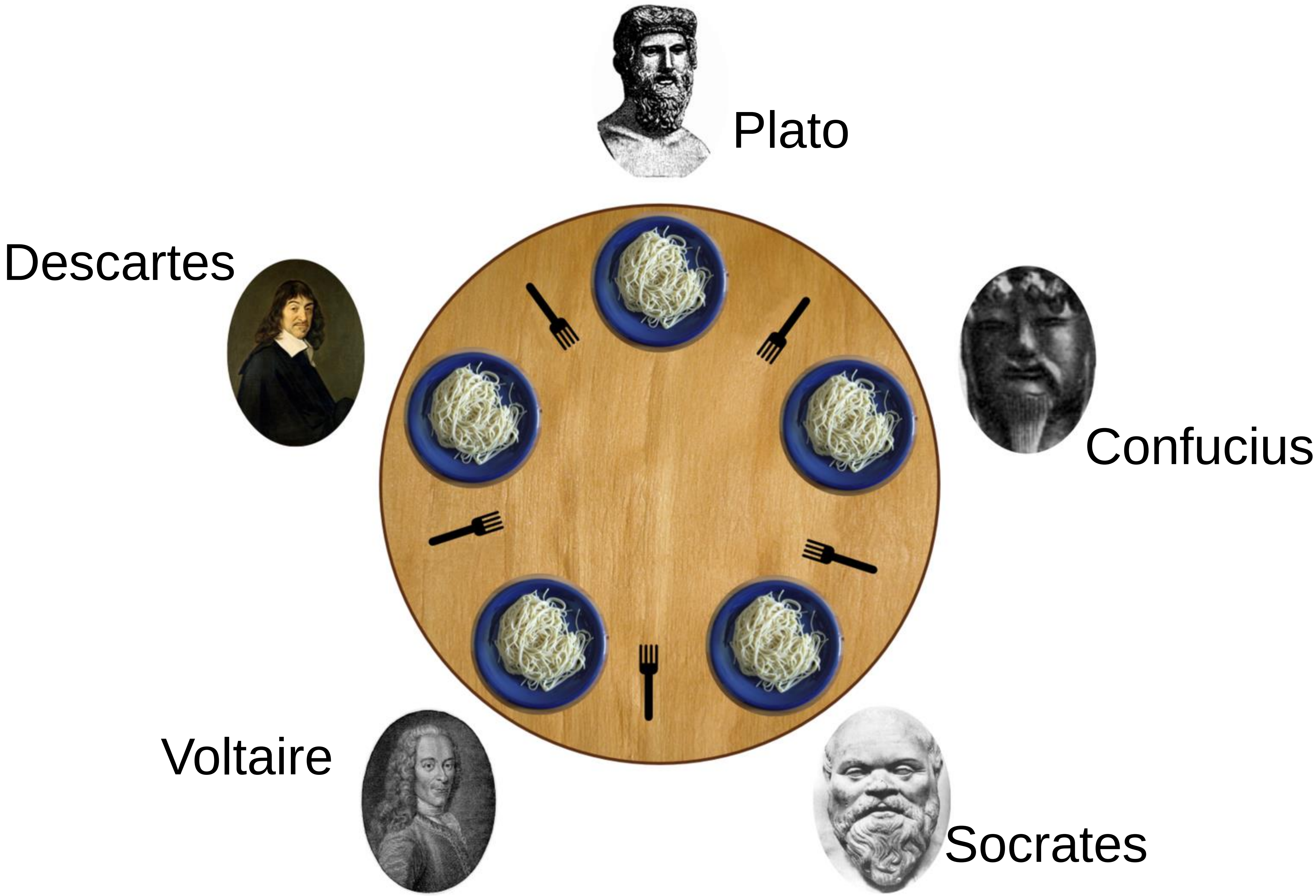
# Starvation

Some process is unable to obtain access to shared resources

- May happen because other processes repeatedly beat the process in obtaining access
- Deadlock implies starvation but not vice versa



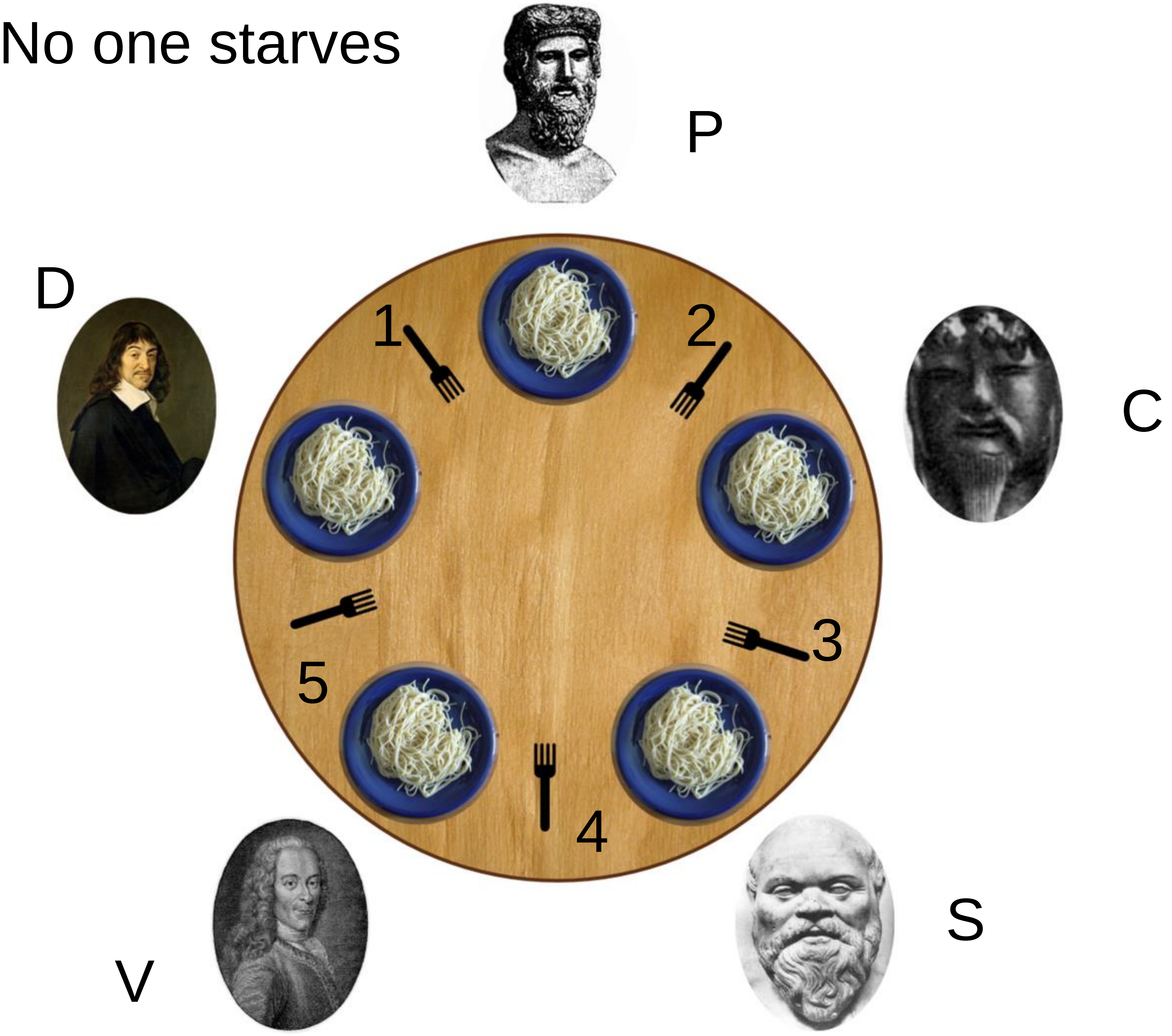
# Dining Philosopher



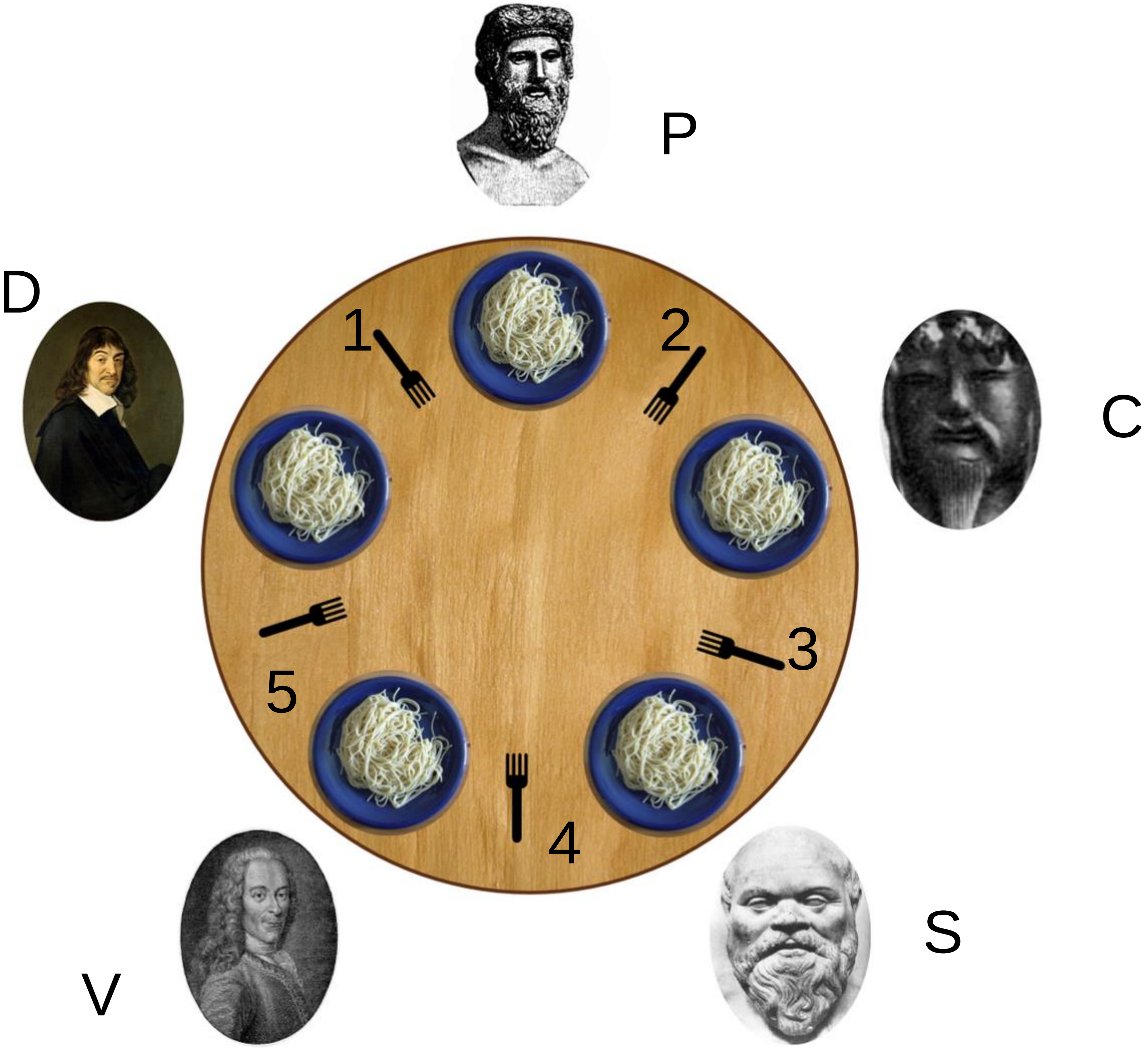
*An_illustration_of_the_dining_philosophers_problem,* Created by Benjamin D. Esham,
Creative Commons 3.0 https://creativecommons.org/licenses/by-sa/3.0/

Each alternates indefinitely between eating and thinking. To eat, each needs both left and right forks and when finished eating, puts them down. Each picks and puts forks down one by one. Two cannot simultaneously hold a fork.

Challenge: No one starves



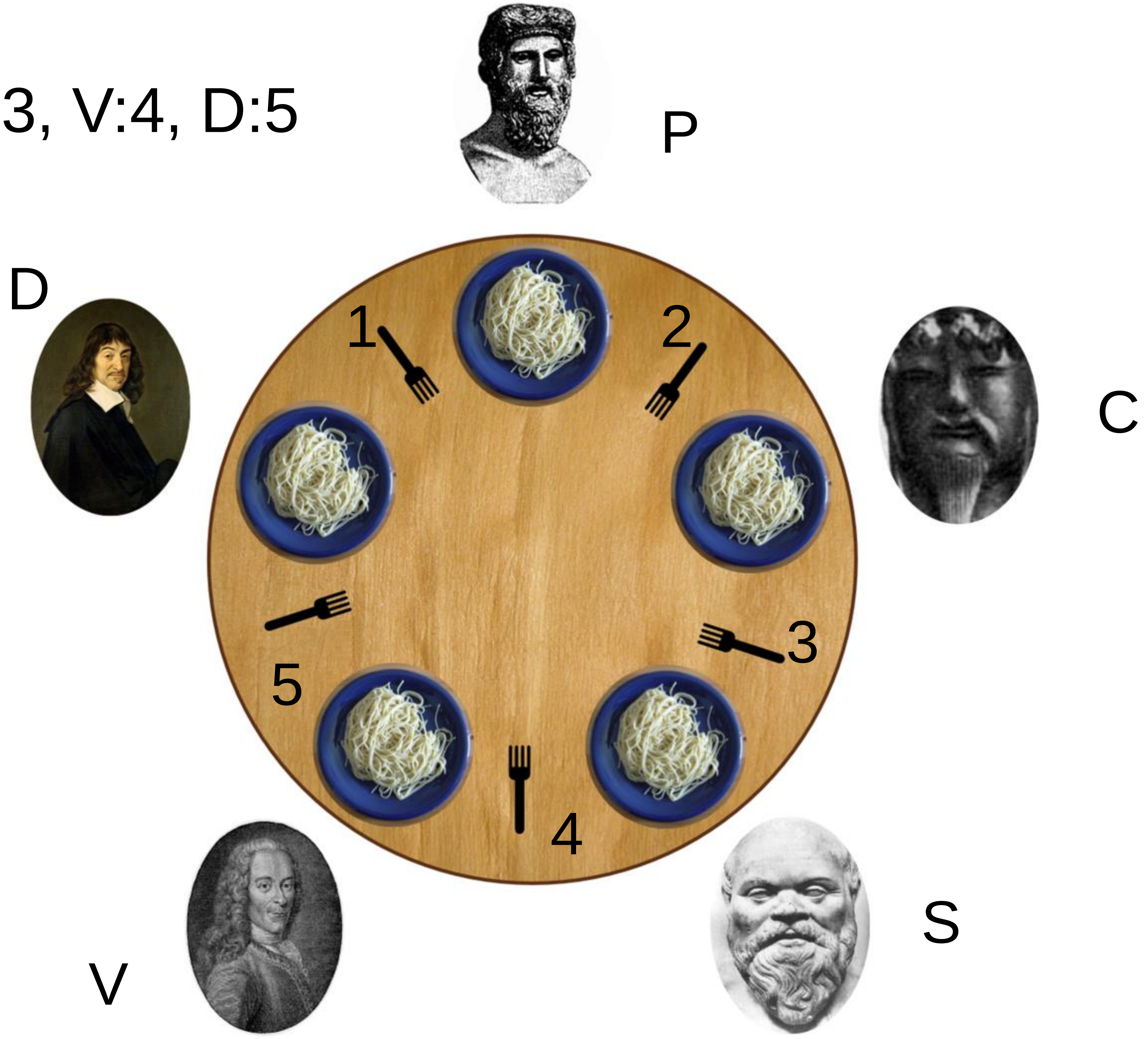
# Solution? Each picks up an available fork, then waits for the other fork to become available.



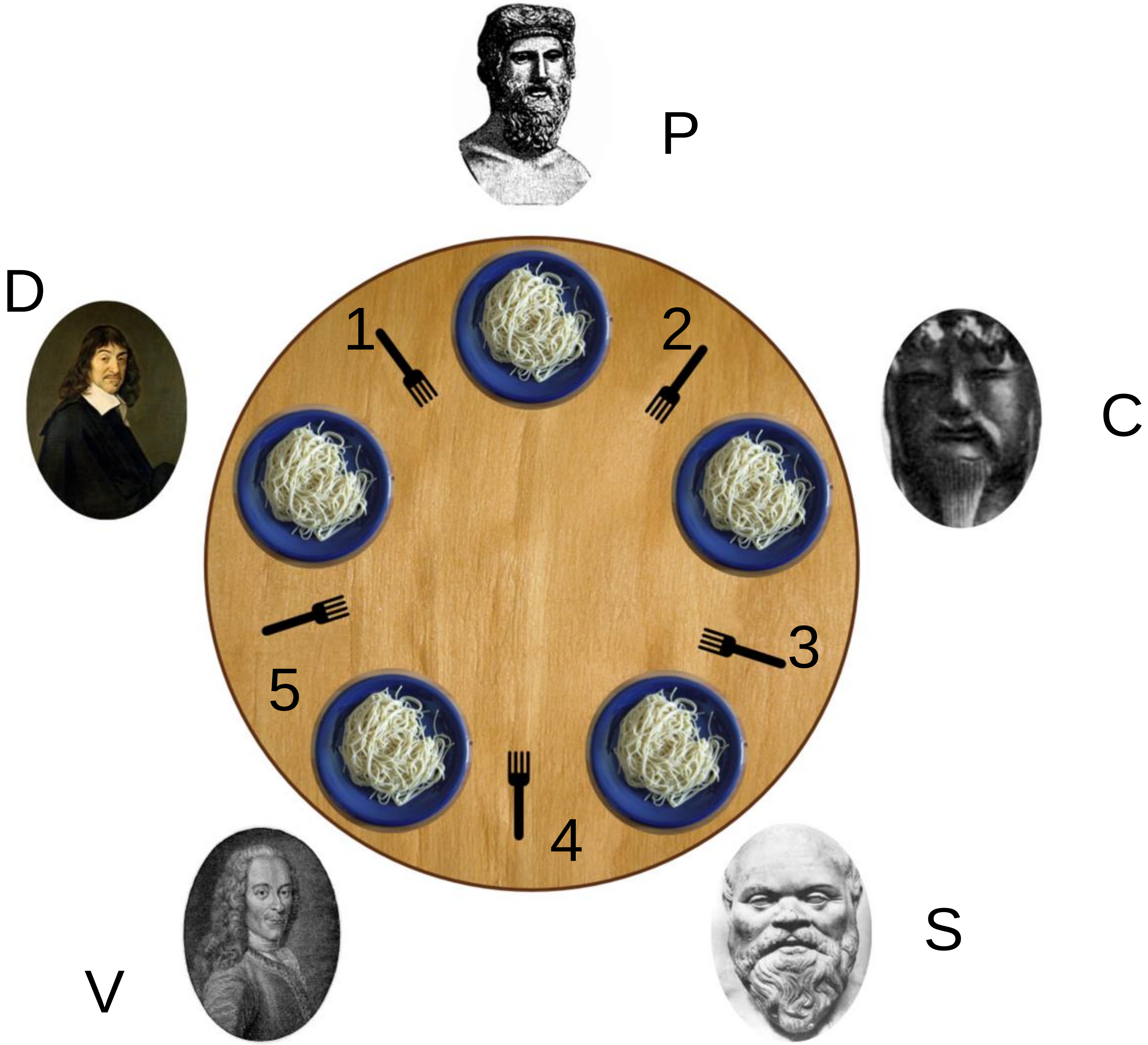
# Each picks up an available fork, then waits for the other fork to become available. Solution?

Deadlock!

P:1, C:2, S:3, V:4, D:5

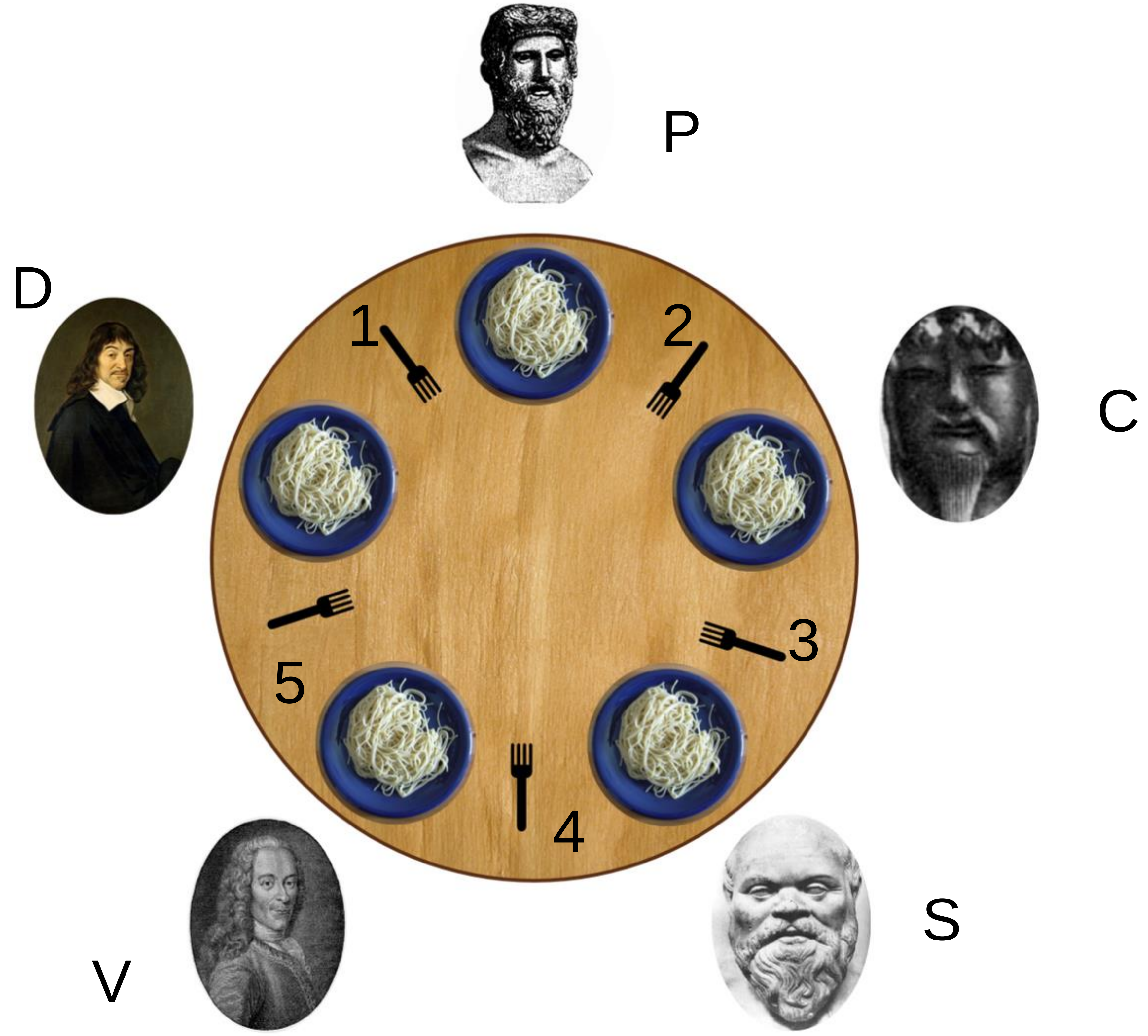


# Each picks up an available fork, then checks for the other fork. If unavailable, put downs the held fork. Retries. Solution?



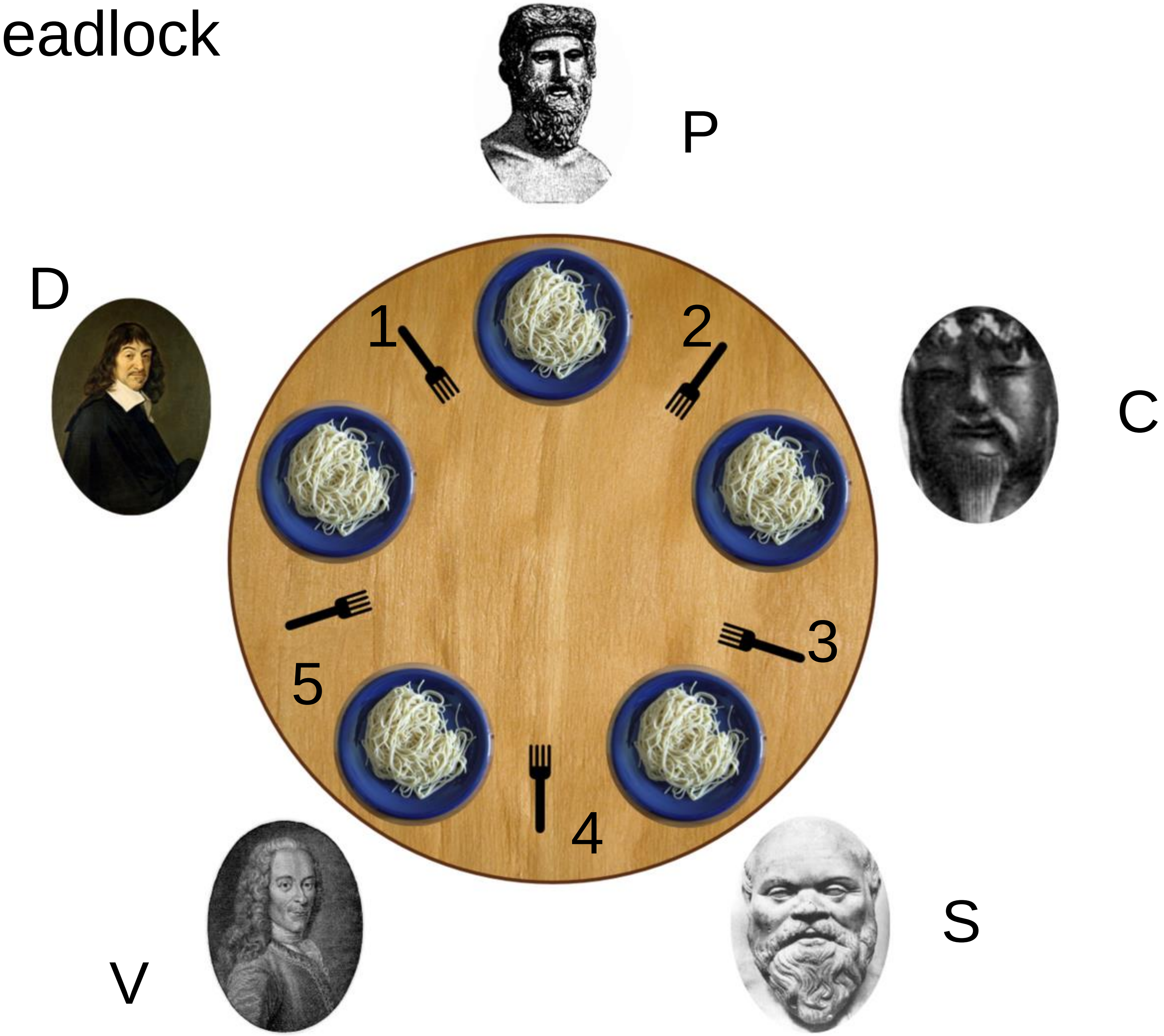
# Each picks up an available fork, then checks for the other fork. If unavailable, put downs the held fork. Retries. Solution?

Livelock!



# Each picks lower-numbered fork first and then higher-numbered fork.

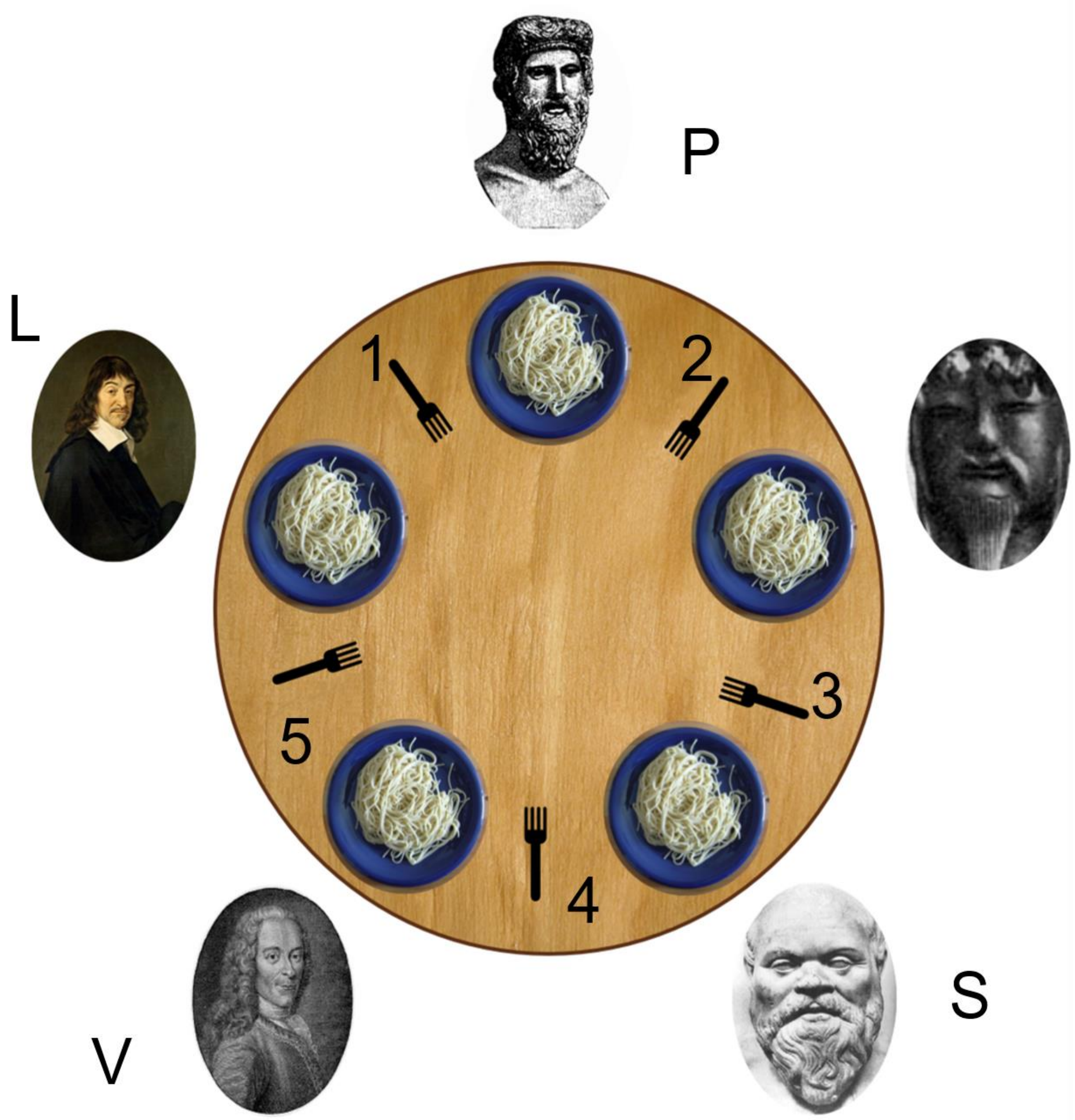
Prevents deadlock



# Each picks lower-numbered fork first and then higher-numbered fork.

But not starvation: C eats, then thinks, but before P can pick up 2, picks it up again…*ad infinitum*

Need *fair scheduling* to guarantee that P eventually eats



# Paranoid Programming

If you are dealing with concurrent design patterns,
Approach the problem methodically

- Its very unlikely you can ‘code your way’ out of the problem
- (Especially with 500,000 lines of code..)
- Checks before, during, and after concurrency ‘hotspots’
- Pen and paper does wonders

# Summary

- Some common pitfalls in concurrent programming
  - Race conditions, deadlock, livelock, starvation
- Dining philosophers
  - Thought experiment
  - Illustrates the problems