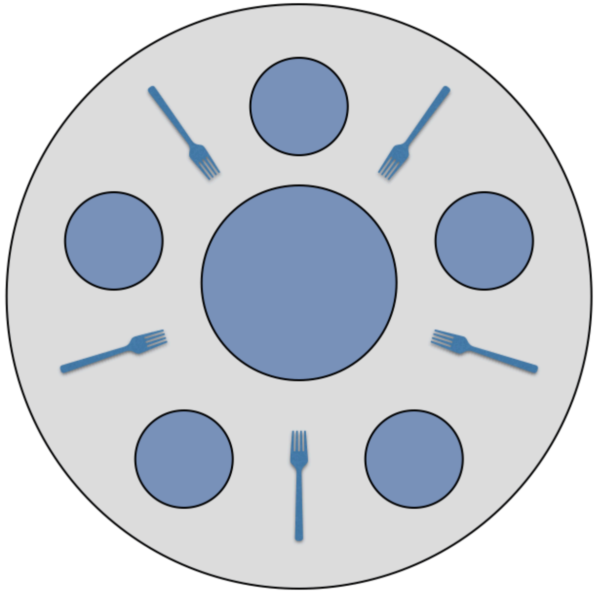
🍽️ Dining Philosophers in Python

> “Five philosophers walk into a table. None eat. Concurrency ensues.”



💡 The Setup

- Five **threads** (philosophers) share **five forks** (locks).
- Each must acquire **two forks** to eat.
- We want:
  - No deadlock
  - No starvation
  - Everyone eats eventually

⚙️ Imports & Base Code

```python
import threading, time, random

N = 5
forks = [threading.Lock() for _ in range(N)]
names = ["Aristotle", "Kant", "Spinoza", "Marx", "Russell"]

def think(name):
    print(f"{name} is thinking...")
    time.sleep(random.uniform(0.5, 1.5))

def eat(name):
    print(f"🍝 {name} is eating!")
    time.sleep(random.uniform(0.5, 1.5))
```

😬 Attempt #1 – Naïve Approach

```
def philosopher(i):
    left = forks[i]
    right = forks[(i + 1) % N]
    name = names[i]

    while True:
        think(name)
        left.acquire()
        right.acquire()
        eat(name)
        right.release()
        left.release()
```

💥 Deadlock: everyone grabs left fork, waits forever for right fork.

🧙 Attempt #2 – The “One Weird Philosopher” Fix

```python
def philosopher(i):
    left = forks[i]
    right = forks[(i + 1) % N]
    name = names[i]

    while True:
        think(name)
        if i == N - 1:
            right.acquire()
            left.acquire()
        else:
            left.acquire()
            right.acquire()
        eat(name)
        right.release()
        left.release()
```

✅ Avoids deadlock

😐 But brittle — depends on arbitrary ordering

🍽️ Attempt #3 – The Waiter (Semaphore) Solution

Only 4 philosophers can pick up forks at once.

```python
waiter = threading.Semaphore(N - 1)

def philosopher(i):
    left = forks[i]
    right = forks[(i + 1) % N]
    name = names[i]

    while True:
        think(name)
        waiter.acquire()
        left.acquire()
        right.acquire()
        eat(name)
        right.release()
        left.release()
        waiter.release()
```

🧑‍🍳 Attempt #4 – Fair Waiter (Queueing)

Simulate fairness with timed waiting and priority.

```
lock = threading.Lock()
next_turn = 0

def philosopher(i):
    global next_turn
    left, right = forks[i], forks[(i + 1) % N]
    name = names[i]

    while True:
        think(name)
        with lock:
            my_turn = next_turn
            next_turn += 1

        while True:
            with lock:
                if my_turn == 0:
                    break
            time.sleep(0.1)  # politely wait

        left.acquire()
        right.acquire()
        eat(name)
        right.release()
        left.release()
```

🧩 Alternative: Using `threading.Condition`

Python’s closest thing to a monitor.

```
condition = threading.Condition()
state = ["thinking"] * N

def left(i):  return (i + N - 1) % N
def right(i): return (i + 1) % N

def test(i):
    return (state[i] == "hungry" and
            state[left(i)] != "eating" and
            state[right(i)] != "eating")

def pickup(i):
    with condition:
        state[i] = "hungry"
        while not test(i):
            condition.wait()
        state[i] = "eating"

def putdown(i):
    with condition:
        state[i] = "thinking"
```

🧮 Summary

Version	Deadlock	Starvation	Difficulty	Comment
Naïve	❌	❌	🪑 Easy	Everyone selfish
Weird Philosopher	✅	❌	🫠 Hacky	Works by luck
Semaphore Waiter	✅	⚠️ Maybe	🙂 Medium	Practical
Fair Waiter	✅	✅	😌 Moderate	Orderly
Monitor	✅	✅	🤓 Advanced	Best conceptually

🧩 Why Monitors Matter

- They **simplify synchronization logic** by hiding low-level lock management.
- They prevent common race-condition nightmares (where a mutex might be forgotten or misused).
- They influenced **modern concurrency abstractions**, like:
  - Java's `synchronized` methods and `wait()/notify()`
  - Python's `threading.Condition`
  - C++'s condition variables inside class methods

⚖️ Comparison: Monitor vs Semaphore

Feature	Monitor	Semaphore
Level	High-level (abstract)	Low-level (primitive)
Encapsulation	Yes — includes data + methods	No — global synchronization variable
Automatic mutual exclusion	Yes	Must be done manually
Ease of use	Easier, safer	Prone to errors (e.g., forgetting `signal()`)

🗣️ Plain-English Summary

A **monitor** is like a **traffic light with built-in rules**:

- Only one car (thread) can be in the intersection (shared resource) at a time.
- Cars can **wait** when the light is red (condition not met).
- When conditions change, the light turns green (`signal`) for the next waiting car.

🧩 In Pseudocode

```
monitor BoundedBuffer {
    int buffer[N];
    int count = 0, in = 0, out = 0;
    condition notFull, notEmpty;

    procedure insert(item) {
        if (count == N) wait(notFull);
        buffer[in] = item;
        in = (in + 1) % N;
        count++;
        signal(notEmpty);
    }
```

🧠 Discussion

- Why does resource ordering matter?
- How could this generalize to **N resources**?
- How might you simulate this visually (e.g., using `asyncio` or `pygame`)?

🏁 Closing Thought

> “The Dining Philosophers problem doesn’t teach you how to eat.
> It teaches you **how not to starve while sharing.**”