KARNAVATI UNIVERSITY

Operating Systems

Course Code: **71203002004**

Case Studies in synchronization

by -

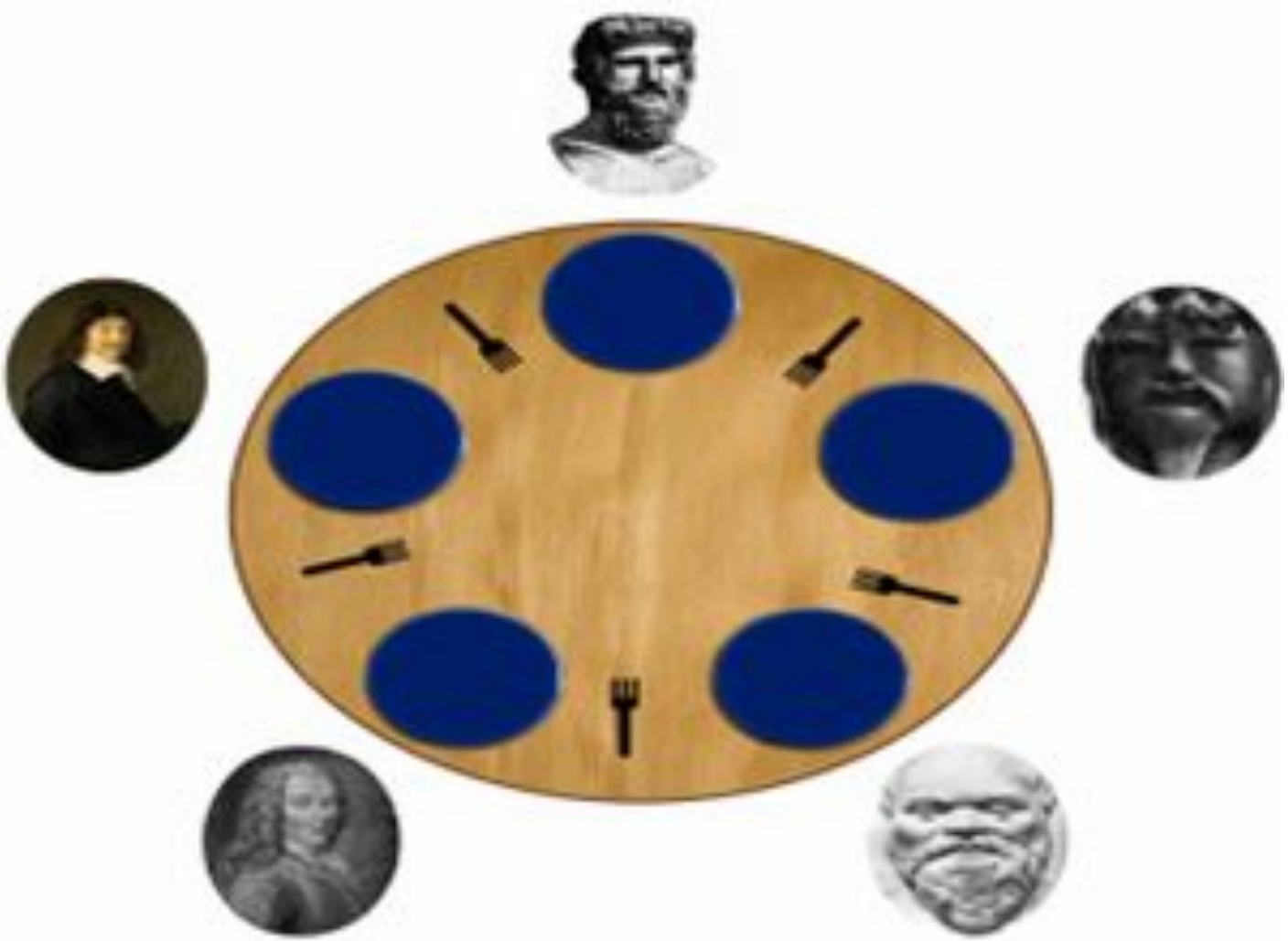
Minal Rajwar

Dining Philosophers Problem

It's a classic computer science problem about **sharing limited resources** (like forks) among multiple processes (philosophers) without causing **deadlock** or **starvation**.

Problem Setup

- **5 philosophers** sit around a round table.
- Each philosopher alternates between **thinking** and **eating**.
- To eat, a philosopher needs **two chopsticks** (one on the left, one on the right).
- There are **only 5 chopsticks**, each shared between neighbors.



Rules

A philosopher can pick up **only one chopstick at a time**.

They can eat **only if they have both chopsticks**.

We must design a method so:

- Two philosophers don't eat with the same chopstick at the same time (**mutual exclusion**).
- No one is stuck waiting forever (**no deadlock**).
- Everyone eventually gets to eat (**no starvation**).

First Attempt (Simple Semaphore)

- Each chopstick is a **semaphore** (1 = available, 0 = taken).
- Philosopher waits for the left fork, then the right fork.
- **Problem:** If everyone picks the left fork first, they'll all wait forever for the right fork → **deadlock**.

Second Attempt (Limit Access)

- Allow **only 4 philosophers** into the dining room at a time.
- At least one philosopher can always get both forks → avoids deadlock.
- Drawback: Doesn’t fully follow the “all philosophers can sit” scenario.

Third Attempt (Asymmetric Approach)

- First 4 philosophers pick **left fork first**, then **right fork**.
- The last philosopher picks **right fork first**, then **left fork**.
- This breaks the cycle and avoids deadlock.
- Known as the **Chandy/Misra Solution**.
- **Advantages:** No deadlock, no starvation, fair, efficient.

Monitor Solution

- Use a **monitor** with:
  - An array to track forks available to each philosopher.
  - takeForks() waits until both forks are available.
  - releaseForks() frees forks and signals neighbors if they can eat.
- **Ensures:** safety, fairness, no deadlock/starvation automatically.

Key Points to Remember:

- Problem teaches **synchronization** and **resource allocation** in concurrent systems.
- Solutions aim to balance **mutual exclusion**, **deadlock freedom**, and **fairness**.
- Can be implemented using **semaphores** or **monitors**.

Sleeping Barbers Problem

Barber sleeping: If no customer → barber waits (sleeps).

Customer arrives:

- If there's a free chair → sits and waits.
- If no free chair → leaves.

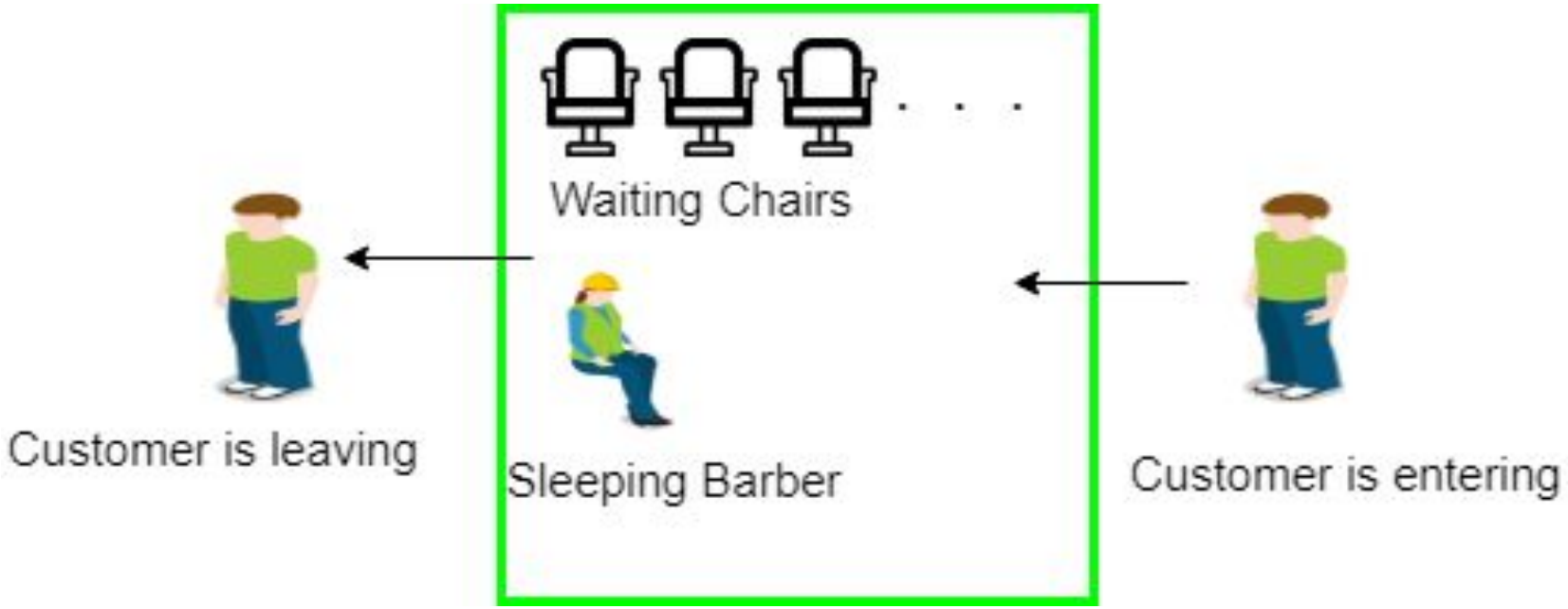
Haircut:

- Barber wakes up if sleeping.
- Cuts one customer's hair at a time.

After haircut:

- If more customers → takes next one.
- If none → goes back to sleep.

Sleeping Barbers Problem



Sleeping Barbers Problem

Semaphores Used

- **customers** → Counts waiting customers.
- **barber** → 0 or 1, shows if barber is ready.
- **mutex** → Ensures only one process changes waiting chair count at a time.

Why Use Semaphores?

- Prevents **two customers** from taking the same chair.
- Ensures **only one haircut at a time**.
- Handles sleeping/waking of barber properly.

Synchronization in the Sleeping Barber Problem

Advantages

- **Efficient Resource Use** – Barber chair and waiting chairs are never wasted.
- **No Race Conditions** – Only one customer in the chair at a time, no conflicts.
- **Fairness** – Every waiting customer gets a fair turn.

Synchronization in the Sleeping Barber Problem

Disadvantages

1. **Complexity** – Harder to design and code, especially in big systems.
2. **Overhead** – Uses extra CPU time, memory, and resources.
3. **Risk of Deadlocks** – Wrong implementation can make everyone stuck waiting.

Classical Synchronization Problem - using Semaphores

1. Producer–Consumer Problem

- **Idea:** Producer makes items, consumer uses them.
- **Buffer:** Temporary storage with limited size.
- **Semaphores:**
  - empty → empty slots count
  - full → filled slots count
  - mutex → lock for safe access

Classical Synchronization Problem - using Semaphores

2. Traffic Light Control

- **Idea:** Semaphores control lights for each direction at an intersection.
- **Semaphores:** One per light direction to ensure only one is green at a time.

3. Bank Transaction Processing

- **Idea:** Only one transaction can change an account at a time.
- **Semaphore Value:** 1 → ensures exclusive access.

Classical Synchronization Problem - using Semaphores

4. Print Queue Management

- **Idea:** Only one print job uses the printer at a time.
- **Semaphore Value:** 1 → lock until printing is done.

5. Railway Track Management

- **Idea:** Only one train can be on a track section at a time.
- **Semaphore Value:** 1 → ensures safe crossing.

Classical Synchronization Problem - using Semaphores

6. Dining Philosophers

- **Idea:** Philosophers share forks (two needed to eat).
- **Semaphore Value:** 1 per fork → prevents deadlock and starvation.

7. Reader–Writer Problem

- **Idea:** Many readers can read at once, but only one writer can edit at a time.
- **Semaphores:**
  - Reader → multiple access allowed
  - Writer → exclusive access

DISCUSSION & REVISION

1. In the Producer–Consumer problem, what stores the items temporarily?
2. Which mechanism is used in all these problems to control access?
3. In Traffic Light Control, what changes from green to red?
4. In Bank Transaction Processing, what is being updated?
5. In the Sleeping Barber problem, where do customers wait?
6. In the Dining Philosophers problem, what is needed to eat?

REFERENCES

1. https://www.geeksforgeeks.org/operating-systems/semaphores-in-process-synchronization/
2. https://www.geeksforgeeks.org/operating-systems/dining-philosophers-problem/
3. https://www.geeksforgeeks.org/operating-systems/sleeping-barber-problem-in-process-synchronization/