Dining Philosophers Problem Simulation using Processes and Semaphores

Operating Systems CS211

Mahnoor Omer 2022912

Tammir Atif 2023907

Bushra Nas 2022149

### **Problem Overview**

The **Dining Philosophers Problem** is a classical synchronization problem that illustrates the challenges of allocating shared resources (forks) among competing entities (philosophers) without causing deadlock or starvation.

In this scenario:

* There are **5 philosophers** seated around a table.
* Each philosopher alternates between **thinking** and **eating**.
* To eat, a philosopher must pick up both the left and right forks (shared resources).

### **Objective**

* Simulate a concurrent system using multiple processes.
* Prevent **deadlocks** and **starvation** using semaphores.
* Demonstrate **practical usage of system calls** to solve real-world concurrency issues.

### **System Design and Implementation**

#### ****4.1 Representation****

* **Philosophers** are modeled as **separate processes** created using fork().
* **Forks** are represented using **named semaphores** created with sem\_open().

#### ****4.2 Philosopher Routine****

Each philosopher follows a loop in the function philosopher(int id):

1. **Think** — Simulated using sleep() and printf().
2. **Pick up forks** — Controlled using sem\_wait() on the fork semaphores.
3. **Eat** — Simulated using sleep() and printf().
4. **Put down forks** — Done using sem\_post() to release the semaphores.

### **Synchronization Mechanism**

* Semaphores (sem\_t) are used to control access to forks.
* Each fork is represented by a **named semaphore**, initialized to 1.
* All philosopher processes **share the same named semaphores**, ensuring inter-process communication.

### **Deadlock and Starvation Prevention Strategy**

To avoid **deadlock**, we break one of the **Coffman conditions** — **circular wait** — using the following strategy:

* **Even-numbered philosophers** pick up the **right fork first**, then the **left**.
* **Odd-numbered philosophers** pick up the **left fork first**, then the **right**.

This **asymmetric fork acquisition** order ensures that circular waiting cannot occur.

Additionally, because all philosophers are periodically thinking and releasing forks, **starvation is prevented** as no philosopher can hog resources indefinitely.

### **System Calls and Libraries Used**

| Function / Call | Purpose |
| --- | --- |
| fork() | Creates a separate process for each philosopher |
| sem\_open() | Creates named semaphores |
| sem\_wait() | Decrements semaphore (locks fork) |
| sem\_post() | Increments semaphore (releases fork) |
| sem\_close() | Closes semaphore |
| sem\_unlink() | Unlinks named semaphore from system |
| sleep() | Simulates time delay for thinking and eating |
| printf() | Output trace of philosopher states |

### **Observations**

* Deadlock was avoided due to asymmetric fork acquisition.
* All processes made progress over time, indicating **no starvation**.
* The solution is **scalable** to more philosophers by extending the semaphore array.

Output of our simulation:

