Operating System

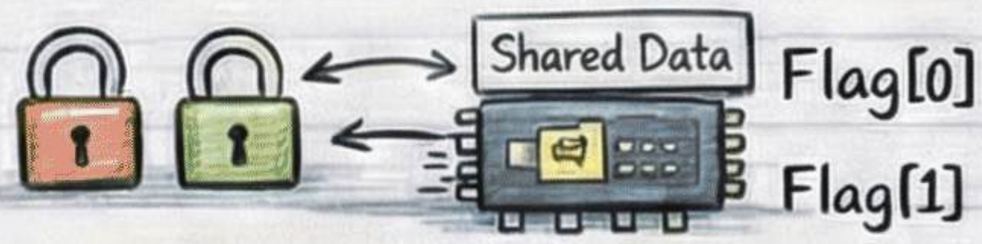
Module-3 Notes

by pyqfort.com

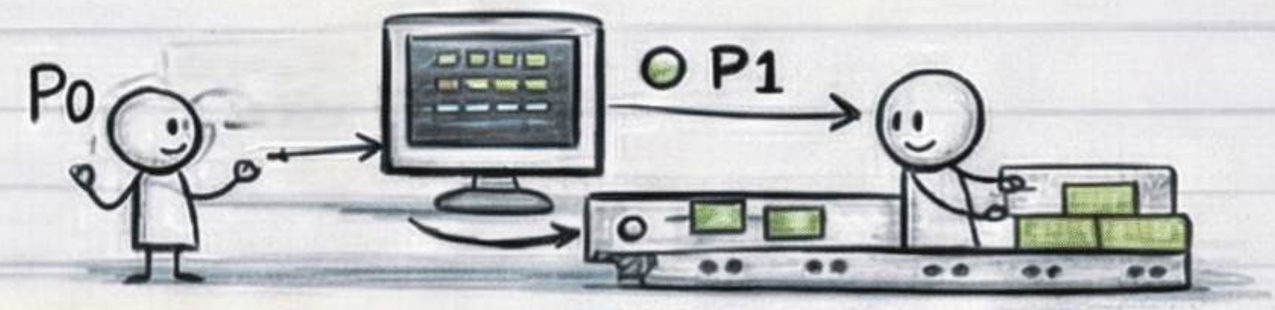
Contents Covered:

- Intro
- Race Conditions

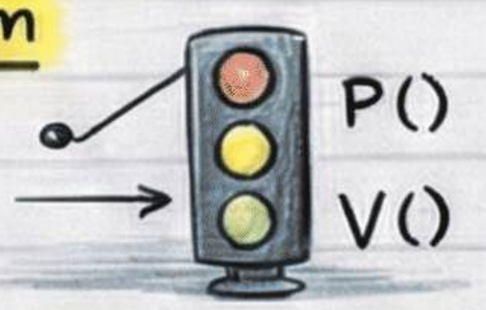
- Mutual Exclusion & Hardware Solution

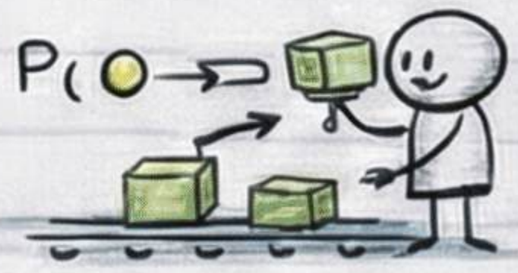


- Peterson's Solution

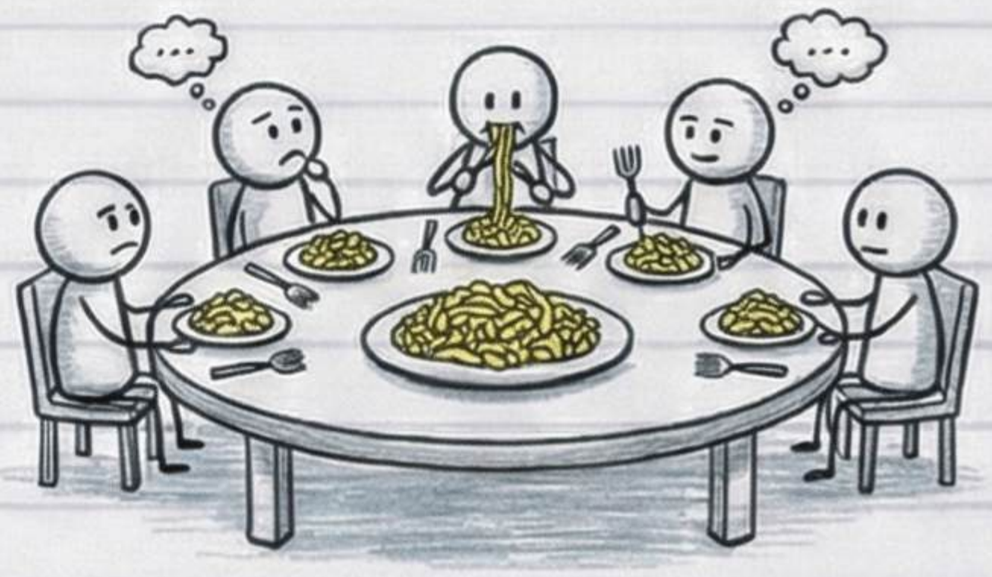


- Producer/Consumer Problem



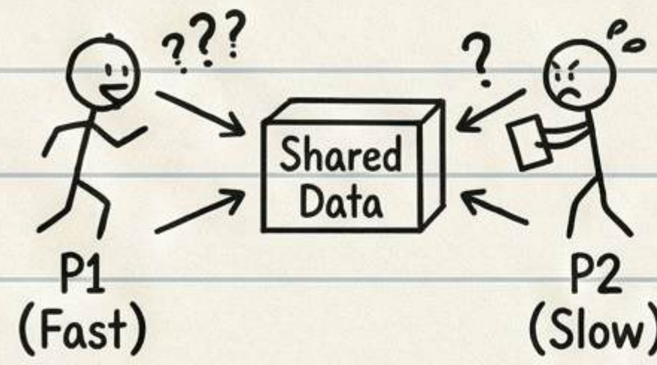


- Semaphores
- Monitors
- Dining Philosophers Problem



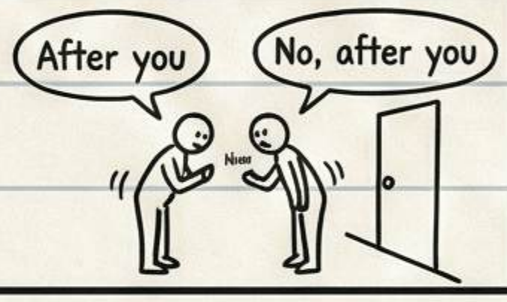
MODULE 3: INTER-PROCESS COMMUNICATION (IPC)

1. The Problem: Concurrency & Shared Data



Concurrency (running at same time, different speeds) + **Shared Data** = Potential **Race Condition** (Outcome depends on timing!)

2. Key Concepts & Definitions

Term	Meaning
Mutual Exclusion	Only ONE process in its Critical Section at a time. 
Critical Section (CS)	Code part accessing/modifying shared resources. MUST BE PROTECTED! 
Livelock	Processes repeatedly yielding to each other, no progress. 

3. The Critical Section Protocol (The Structure)

```
PROCESS P () {
    do {
        Entry Criteria
        Critical Section (CS)
            <update          </α:>
              Shared Data <"here!">
              ...         = code areun!">
        Exit Criteria
        ... Remainder Section ...
    } while (true);
}
```

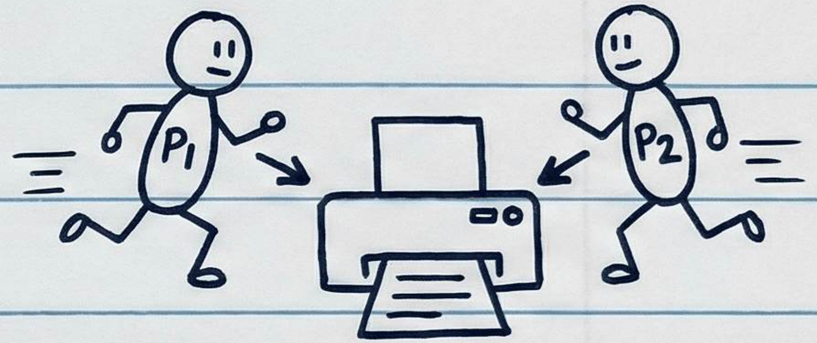
Check conditions. Stop if busy!

Access/Update Shared Data (Only one here!)

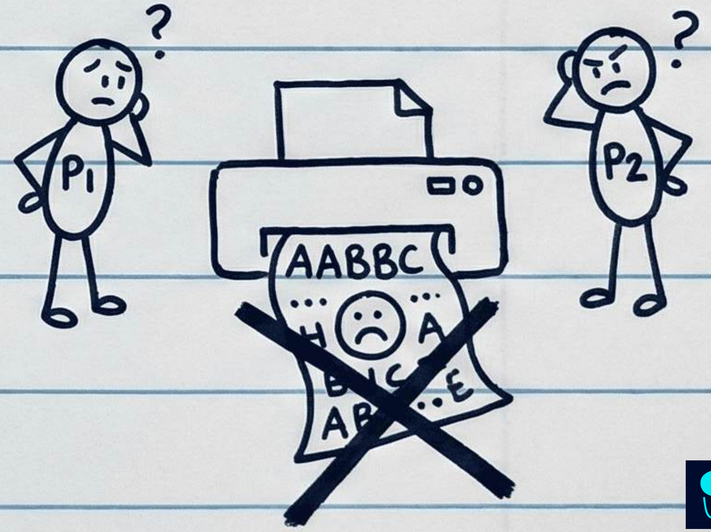
Signal: I'm done! Next process can enter.

Race Conditions

This happens when two (or more) processes try to access a non-shareable resource (like a printer) at the same time.



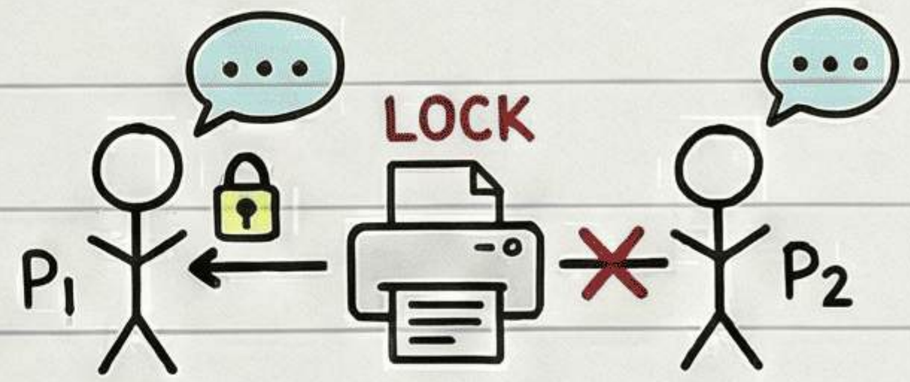
The problem is that the output from the processes gets mixed up, and no process gets the desired result.



Mutual Exclusion & Hardware Solution

Mutual Exclusion

When a process accesses a non-shareable resource (e.g., printer), others must be excluded to prevent mixed output.
This is Mutual Exclusion.



It ensures processes do not access a resource concurrently.

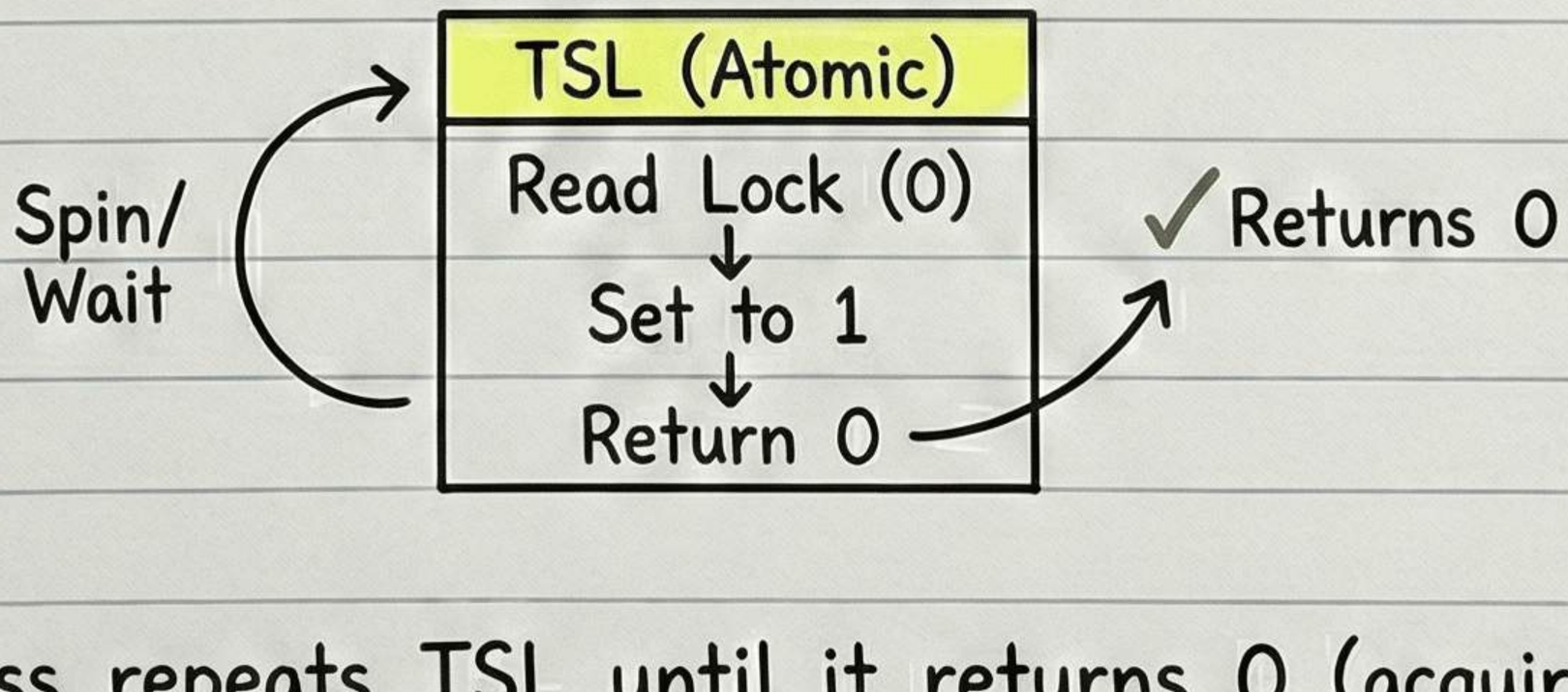
Hardware Solution (Test-and-Set / TSL)

The "Test and Set" (TSL) is a special atomic hardware instruction to solve the critical section problem.

It works as a single, indivisible operation.

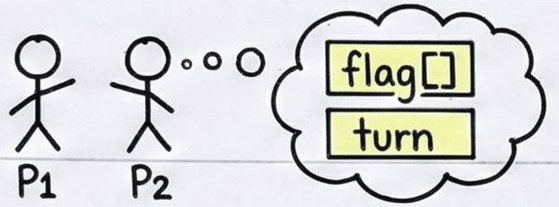
Steps:

1. Reads lock value (e.g., 0 for unlocked).
2. Sets lock to 1 (locked) regardless.
3. Returns original value read.



Process repeats TSL until it returns 0 (acquired lock).
Atomicity prevents simultaneous access.

Strict Alternation: Peterson's Solution



A software solution for **mutual exclusion** between **two processes** using two shared variables.

Process P1

```
do {
    process_flag[0] = true;   // I want in!
    process_turn = 1;         // Your turn (polite)

    while (process_flag[1] && process_turn == 1);
                              // Wait...

    // Critical Section

    process_flag[0] = false;  // I'm done!
    ...
} while (true);
```

Process P2

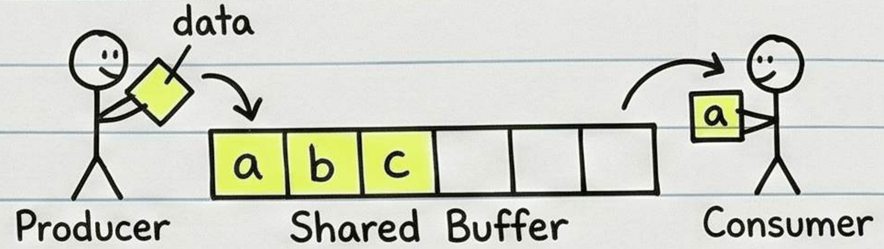
```
do {
    process_flag[1] = true;   // I want in!
    process_turn = 0;         // Your turn (polite)

    while (process_flag[0] && process_turn == 0);
                              // Wait...

    // Critical Section

    process_flag[1] = false;  // I'm done!
    ...
} while (true);
```

Summary: Ensures **Mutual Exclusion**, **Progress**, and **Bounded Waiting**. A fair & correct solution!

Checklist: Understand `flag` & `turn`? How the `while` loop works?

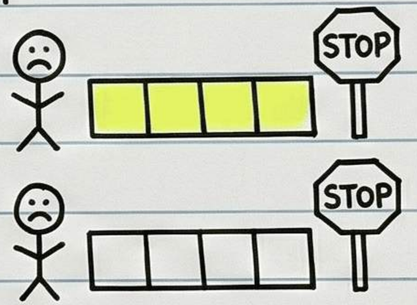
The Producer/Consumer Problem

This is a classic synchronization problem with **two processes**: a **Producer** and a **Consumer**, who share a **fixed-size buffer**.



How it Works & The Rules

- The **Producer** generates data and puts it into the **buffer**.
- The **Consumer** takes data out of the **buffer** and uses it.
- Crucial Rule 1 (**Producer**): If the buffer is **FULL**, the Producer must **WAIT** until a slot becomes empty.
- Crucial Rule 2 (**Consumer**): If the buffer is **EMPTY**, the Consumer must **WAIT** until data is added.



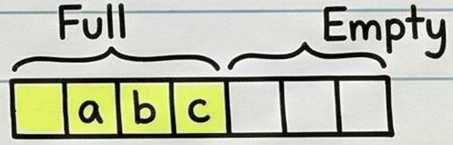
Key Terms & State

Buffer: A temporary storage area (like a queue).

Full: Number of slots containing data.

Empty: Number of free slots.

Goal: We need a way to **synchronize** them so they don't overwrite data or try to consume nothing!

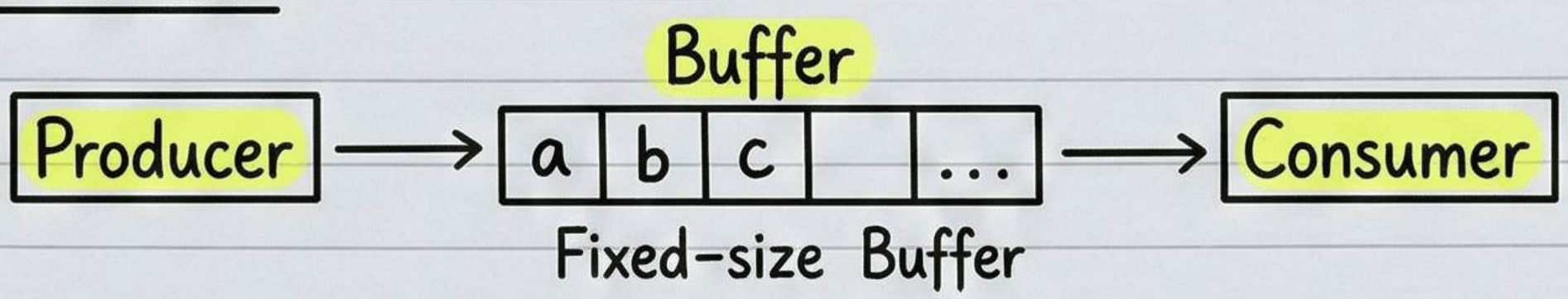


"**Full = 3**" (items a, b, c)

"**Empty = n - 3**" (where n is total slots)

The Producer/Consumer Problem Codes

The Problem



Producer creates items, adds to buffer. Consumer removes items. Must synchronize to avoid overflow/underflow & ensure mutual exclusion.

The Solution (Semaphores)

Uses three semaphores:

1. Empty: Count of empty slots (init. N). 

2. Full: Count of filled slots (init. 0). 

3. Buffer_access: Mutex for buffer (init. 1). 

```
Producer() {
  do {
    Produce an item;
    P(Empty); // Wait for slot
    P(Buffer_access); // Lock buffer
    Add item to buffer;
    V(Buffer_access); // Unlock
    V(Full); // Signal full
  } while(true);
}
```

```
Consumer() {
  do {
    P(Full); // Wait for item
    P(Buffer_access); // Lock buffer
    Consume item from buffer;
    V(Buffer_access); // Unlock
    V(Empty); // Signal empty
  } while(true);
}
```

P() is wait (decrement). V() is signal (increment).
Ensures only one accesses buffer at a time.

Semaphores

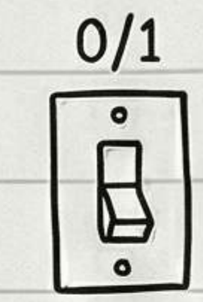
A semaphore is a tool for process synchronization. It acts like a guard or lock on a shared resource, preventing race conditions.

Types of Semaphores

Binary Semaphore: Value is only 0 or 1. Used for mutual exclusion of a single resource.

Counting Semaphore: Can have any positive integer value. Used when there are multiple instances of a resource (e.g., 3 printers → starts at 3).

Mutex: A special binary semaphore where *only* the process that locked it can unlock it.



3 resources

Operations (Atomic!)

Two indivisible operations:

wait (P): Checks the value. If > 0, it decrements and proceeds. If <= 0, it waits.

signal (V): Increments the value, potentially allowing a waiting process to enter.

Implementation: Busy Waiting vs. Blocking

Problem: A simple while(S<=0); loop creates busy waiting or a spinlock, which wastes CPU time.

Solution: Instead of spinning, the process should be blocked and put into a waiting queue.
A signal operation then wakes up a process from the queue.

Queue (Zzz...)

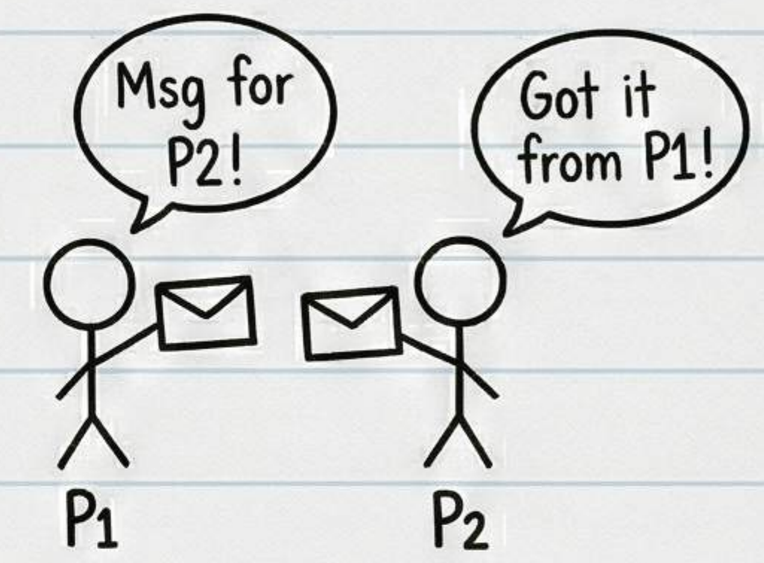
This uses a structure: struct { int value; queue q; }

Monitors: Message Passing

Process communication can happen in three ways: shared memory, message passing, and signals.

Message Passing

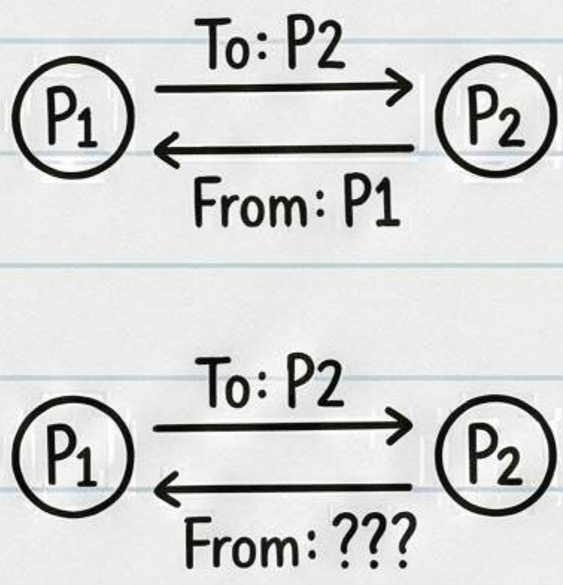
Processes communicate by sending messages and are explicitly aware of each other.



Direct Addressing

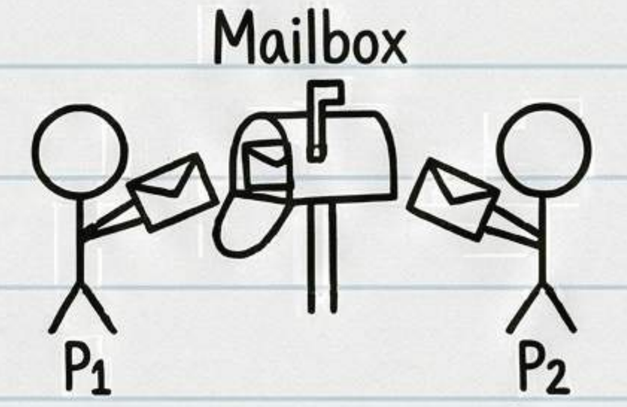
Processes must know each other's names.

- Symmetric: Both sender & receiver know each other's names.
- Asymmetric: Sender knows receiver's name, but receiver doesn't know sender's.



Indirect Addressing

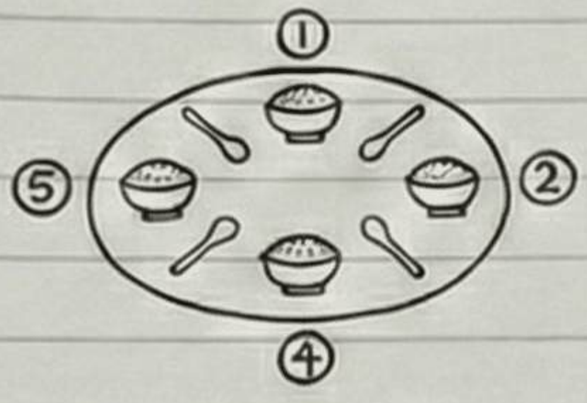
Processes use a shared mailbox to store messages. They don't need to know each other's names directly.



- Link types: One-to-one, one-to-many, many-to-one, many-to-many

NOTE: Each philosopher needs both adjacent spoons simultaneously to eat — one from left and one from right neighbor.

Dining Philosophers Problem (Dijkstra): Classic synchronization IPC problem. Five philosophers around a round table, think & eat. Infinite rice, but only 5 spoons, one between each pair. Need 2 spoons to eat.



Deadlock: If all pick up left spoon simultaneously.

Livelock: All put down & retry simultaneously. System won't progress.

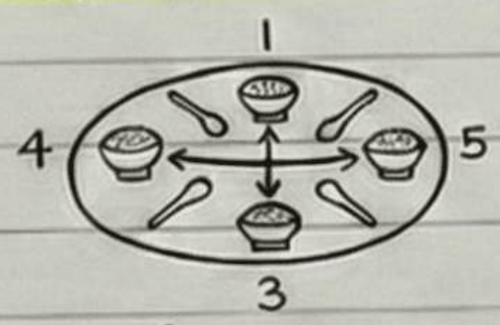
SOLUTIONS:

1. **Limit Philosophers**: Allow only 4 to sit. Fifth prevented from sitting. Deadlock avoided.

Algorithm (Semaphore):

```
do {
        wait(Sem_Philosopher);
        wait(Sem_Spoon[n]);
        wait(Sem_Spoon[(n+1)%5]);
        eat();
        signal(Sem_Spoon[n]);
        signal(Sem_Spoon[(n+1)%5]);
        signal(Sem_Philosopher);
        think();
} while(true);
```

2. **Asymmetric Solution**: Odd-numbered pick left first, then right. Even-numbered pick right first, then left.



Odd can pick up left. Even cannot pick up right. Philosophers 1 & 3 can eat.

```
do {    if (n%2!=0) {
            wait (Sem_Spoon[n]);
            wait (Sem_Spoon[(n+1)%5]);
            eat();
        } else {
            wait (Sem_Spoon[(n+1)%5]);
            wait (Sem_Spoon[n]);
            eat();
        }
        think();
} while (true);
```

Conditions: One philosopher per spoon. No deadlock. No starvation. Max concurrent eating.