

# Process Synchronization

---

# What is process synchronization?

---

Mechanism to ensure that multiple processes or threads can execute concurrently without conflicting with each other, especially when accessing shared resources.

**Goal:** avoid race conditions, ensure data consistency, and coordinate the execution order of processes.

# Race Condition

---

- A race condition occurs when multiple processes or threads read and write shared data concurrently, and the final outcome depends on the sequence of their execution.
- This can lead to unpredictable behavior and bugs that are hard to reproduce and debug.

# Example

---

Consider two threads trying to increment a shared counter:

```
int counter = 0;  
void increment() {  
    counter++;  
}
```

If **both threads read the counter's value simultaneously**, say counter = 0, and then both increment it, the final value might be 1 instead of 2 because they might **overwrite each other's increments**.

# Critical Section

---

- A critical section is a segment of code that accesses shared resources (like data structures or hardware devices) that must not be concurrently accessed by more than one thread or process.
- The main goal is to prevent data inconsistency and corruption.

# Example

---

Imagine a bank account shared between multiple ATMs. The critical section here would be the part of the code that updates the account balance. If two ATMs try to update the **balance simultaneously without proper synchronization**, it might lead to **inconsistent balance updates**.

```
int balance = 1000; // Shared resource
```

```
void withdraw(int amount) {  
    pthread_mutex_lock(&mutex); // Enter critical section  
    if (balance >= amount) {  
        balance -= amount;  
    }  
    pthread_mutex_unlock(&mutex); // Exit critical section  
}
```

# Mutual Exclusion

---

- Mutual exclusion (often shortened to "mutex") is a principle used to prevent race conditions by ensuring that only one process or thread can access the critical section at any one time.
- This guarantees that the critical section is executed atomically.

# Example

---

Using a mutex lock to ensure mutual exclusion:

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void increment() {
    pthread_mutex_lock(&mutex); // Lock to enter critical section
    counter++;
    pthread_mutex_unlock(&mutex); // Unlock to exit critical section
}
```

In this example, **pthread\_mutex\_lock** ensures that only one thread can increment the counter at a time, thus preventing race conditions.



# Peterson's Solution for Mutual Exclusion

---

- **Goal:** Ensure only one of two processes enters the critical section at a time.

## Pros

- Simple and elegant.
- No complex atomic operations or special hardware needed.
- Good theoretical foundation for mutual exclusion.

## Cons

- Limited to two processes.
- Busy waiting wastes CPU cycles.
- Not suitable for modern multiprocessor systems.

```
do {  
    flag[i] = TRUE;  
    turn = j;  
    while (flag[j] && turn == j)  
        critical section  
    flag[i] = FALSE;  
    remainder section  
}  
while (TRUE);
```

**turn:** whose turn it is to enter the critical section  
**flag:** indication of whether or not a process is ready to enter critical section

# Synchronization Mechanisms

---

## **Locks:**

- **Mutex:** Provides mutual exclusion; only one thread can lock at a time.
- **Spinlock:** Threads spin in a loop, checking for lock availability; efficient for short waits.

## **Semaphores:**

- **Binary Semaphore:** Like a mutex; can be 0 or 1.
- **Counting Semaphore:** Manages access for multiple threads by counting available resources.

**Monitors:** High-level construct with a lock and condition variables; only one thread can execute at a time.

**Condition Variables:** Used with mutexes to let threads wait for specific conditions.

**Barriers:** Threads wait until all reach a synchronization point before continuing.

# Solution Using Locks “Mutexes”

---

- **Implementation:** Only one thread can acquire the lock and enter the critical section at a time.
- **Status Indicators:**
  - **Lock = 0:** Critical section is vacant (initial value).
  - **Lock = 1:** Critical section is occupied.

```
do {  
    acquire lock;  
    // Critical section  
    release lock;  
    // Remainder section  
} while (TRUE);
```

# Modern Machines: Atomic Hardware Instructions

---

- **Definition:** Atomic operations are non-interruptible and complete in a single step.

## Examples of Atomic Instructions

- **Test-and-Set:** Sets a value and returns the old value.
- **Compare-and-Swap (CAS):** Compares a variable to an expected value and swaps it with a new value if they match.
- **Fetch-and-Add:** Adds a value to a variable and returns the old value.

## Benefits

- **Efficient Synchronization:** Avoids complex locking mechanisms.
- **Non-Interruptible:** Ensures operations are complete without interruption, preventing race conditions.

# Solution Using Test-And-Set

---

- **Lock Variable:** Initialized to false.
- **Algorithm:** Returns current value and sets lock to true.
- **Critical Section Entry:**
  - **First Process:** Enters immediately (TestAndSet(lock) returns false).
  - **Others:** Wait (lock is true).

```
do {  
    while (TestAndSet(&lock))  
        ; // Do nothing  
    // Critical section  
    lock = FALSE;  
    // Remainder section  
} while (TRUE);
```

# Solution Using Swap

---

- Mechanism:

- Initial: **key = true**, then swap with **lock**.

- Execution:

- First Process:

- key = false**, **lock = true**

- Enters critical section (**while(key)** breaks).

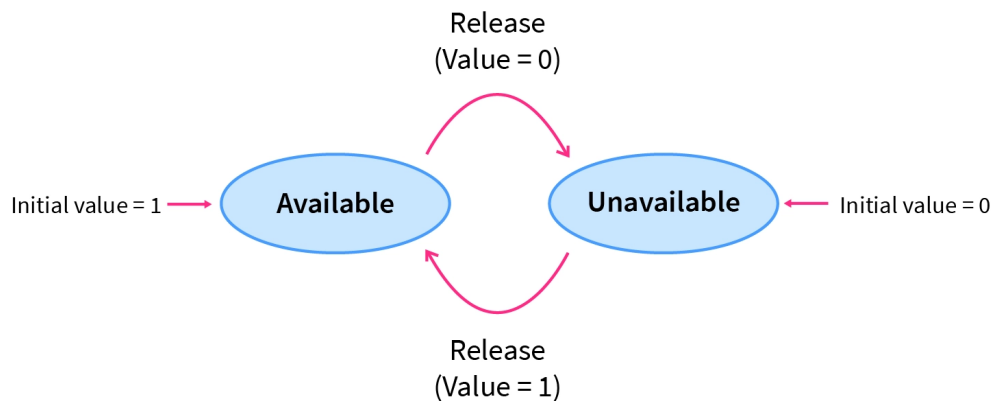
- Others:

- key = true**, **while(key)** continues (lock remains true).

```
do {  
    &key;  
    key = TRUE;  
    while (key == TRUE)  
        Swap(&lock, &key);  
    // Critical section  
    lock = FALSE;  
    // Remainder section  
} while (TRUE);
```

# Semaphore

- Synchronization tool that doesn't require busy waiting.
- **Operations:** Wait () -> Decreases value; if negative, waits. & Signal () -> Increases value; wakes up waiting processes/threads.
- Can be counting or binary.



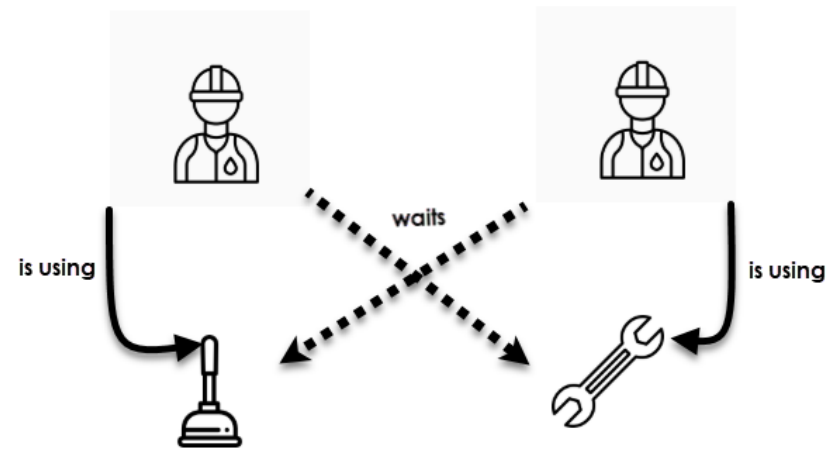
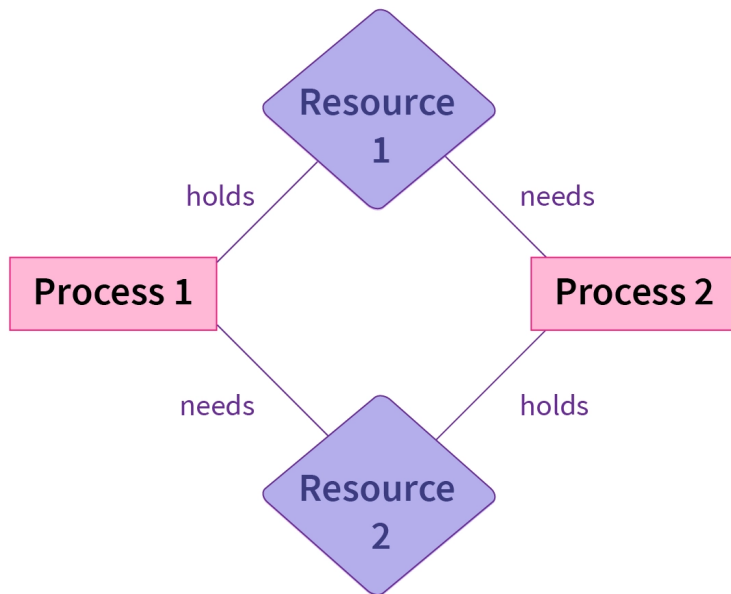
```
Semaphore mutex(1); // Binary semaphore
```

```
// Thread 1  
mutex.Wait(); // Enter critical section  
// Critical Section  
mutex.Signal(); // Exit critical section
```

```
// Thread 2  
mutex.Wait(); // Enter critical section  
// Critical Section  
mutex.Signal(); // Exit critical section
```

# Deadlock

When two or more processes wait indefinitely for each other to release resources, causing all of them to be stuck.





# Conditions

---

- **Mutual Exclusion:** Only one process can use a resource at a time.
- **Hold and Wait:** Processes hold resources while waiting for more.
- **No Preemption:** Resources can't be forcibly taken away from processes.
- **Circular Wait:** A set of processes are waiting for each other in a circular chain.

# Necessary Conditions for Deadlock (Traffic Analogy)



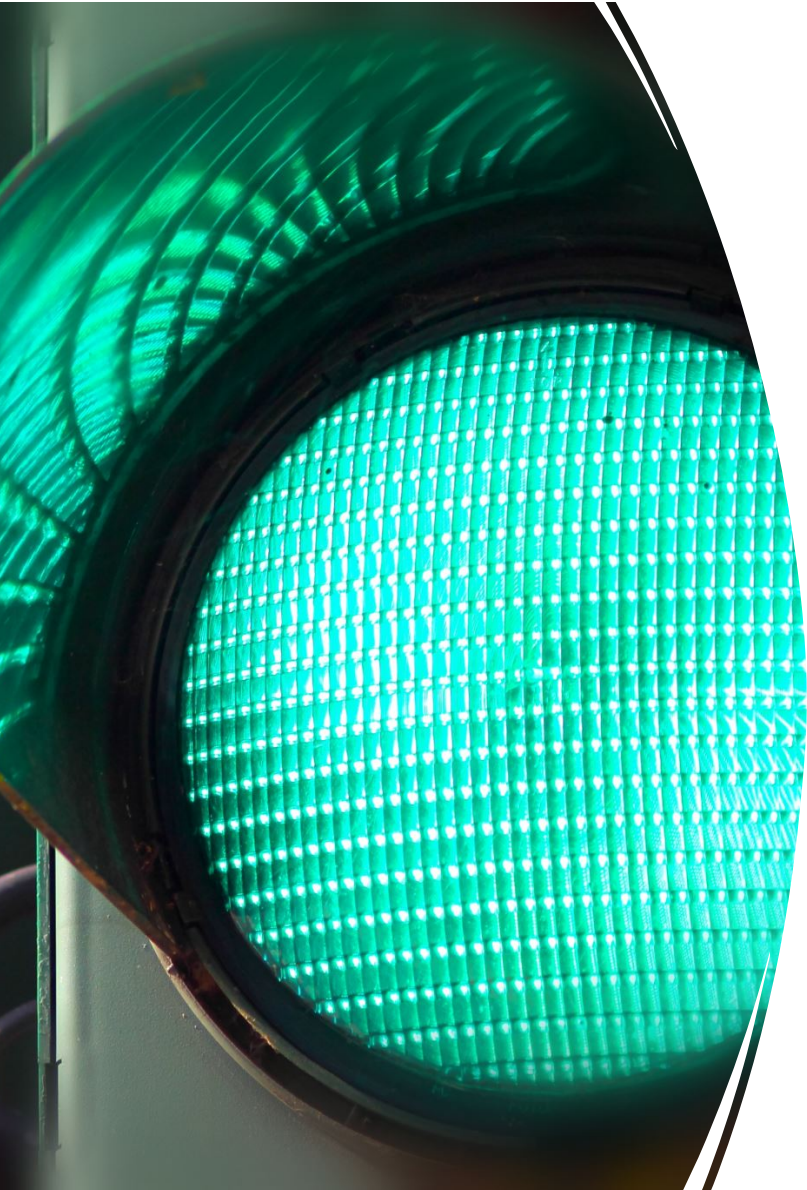


# Mutual Exclusion (One-lane Bridge)

---

- **Explanation:** Only one car can be on a one-lane bridge at a time.
- **Traffic Analogy:** If a car is on the bridge, no other car can enter the bridge until the first car exits.





## Hold and Wait (Intersection with Traffic Lights)

---

- Explanation:** Cars already in the intersection hold their position and wait for the light to turn green.
- Traffic Analogy:** A car is waiting at a red light, holding its place in the intersection while waiting for the green light to proceed. Meanwhile, it blocks the way for other cars that need to pass through the intersection.



# No Preemption (Toll Booth)

---

- Explanation:** Once a car is paying at a toll booth, it cannot be forced to leave until it finishes the transaction.
- Traffic Analogy:** A car at the toll booth can't be pushed out of the way by another car; the waiting cars must wait until the car at the booth finishes paying and moves on.





# Circular Wait (Roundabout with Four Cars)

---

- Explanation:** Each car in a circular roundabout is waiting for the next car to move before it can proceed.
- Traffic Analogy:** Four cars are in a roundabout, each waiting for the car in front to move. Car A is waiting for Car B, Car B is waiting for Car C, Car C is waiting for Car D, and Car D is waiting for Car A. None of the cars can move because they are all waiting for each other, creating a deadlock.



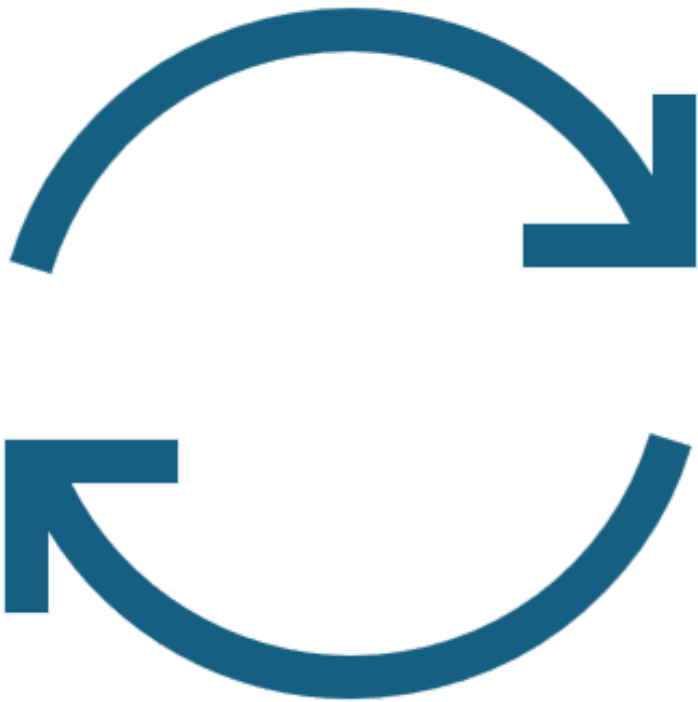
# What is Starvation in OS?

- **Definition:** A process never gets the resources it needs to run because other processes keep getting those resources.
- **Cause:** Typically occurs in priority-based scheduling systems where high-priority requests are processed first.
- **Problems Caused:** Reduced system responsiveness, Violation of response time guarantees, and Potential system failure.



# Concept of Priority Aging

- **Definition:** A scheduling approach used to prevent starvation.
- **Mechanism:** As a process waits in the system, its priority gradually increases.
- **Purpose:** Ensures fair resource distribution by improving the priority of processes that have been waiting longer, reducing the chance of indefinite waiting.



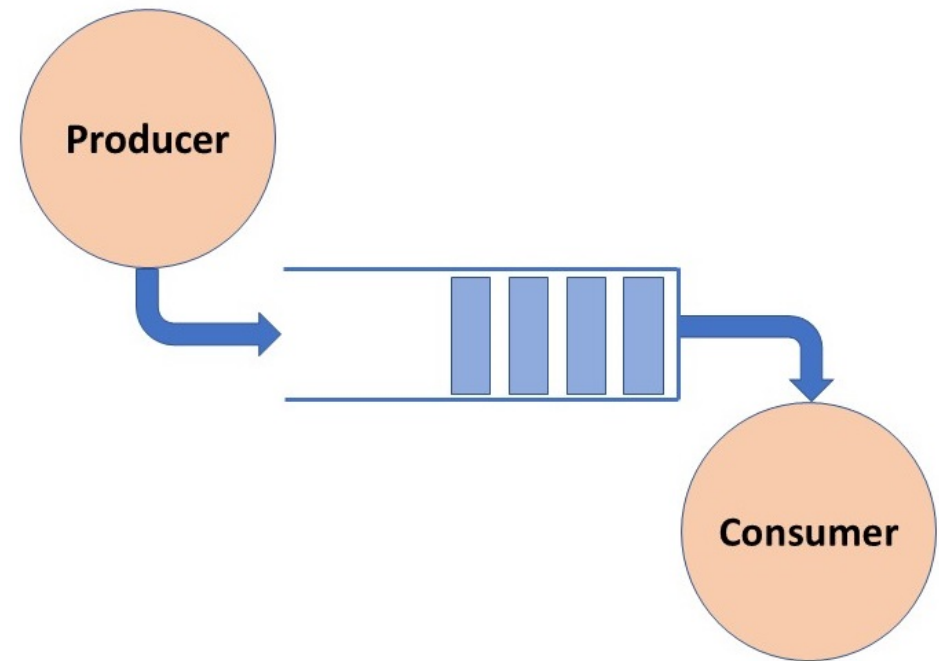


# Classic Synchronization Problems

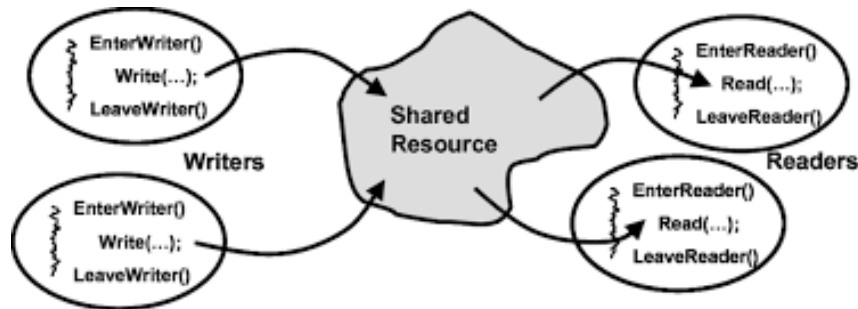
---

## Producer-Consumer Problem:

- Involves synchronization between producer and consumer threads that share a common buffer.
- The producer adds items to the buffer, and the consumer removes items from it.
- **Solution:** Use semaphores to track the number of empty and full slots and a mutex for mutual exclusion.



# Classic Synchronization Problems



## Readers-Writers Problem:

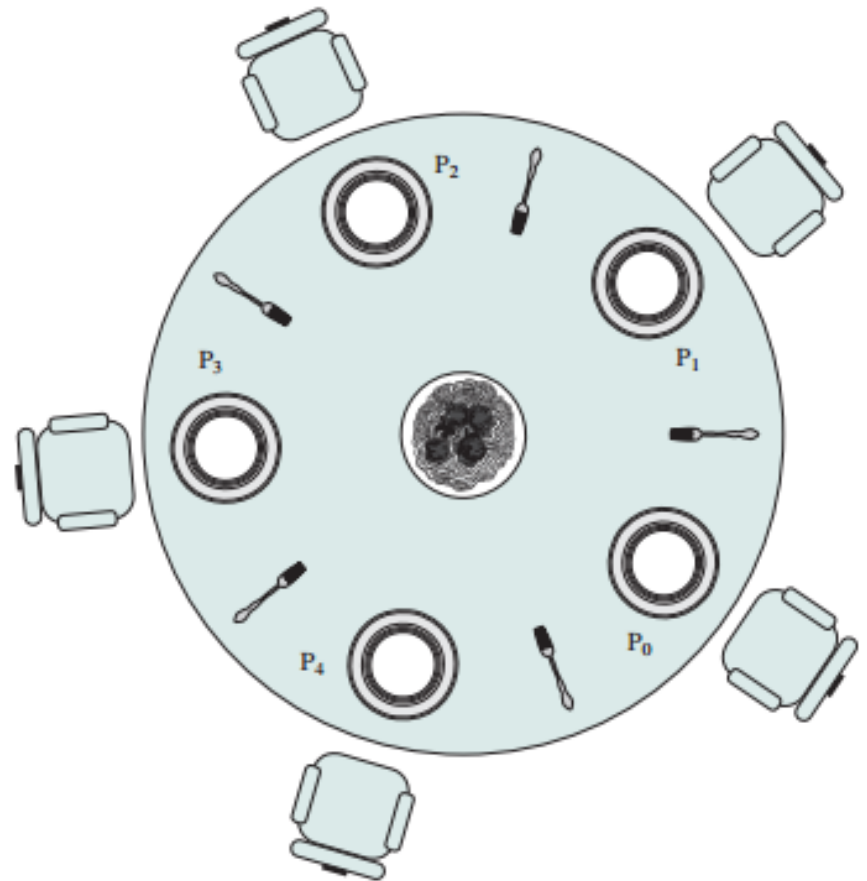
- Involves synchronization between readers and writers of a shared resource.
- Multiple readers can read simultaneously, but writers require exclusive access.
- **Solution:** Use read and write locks to manage access.

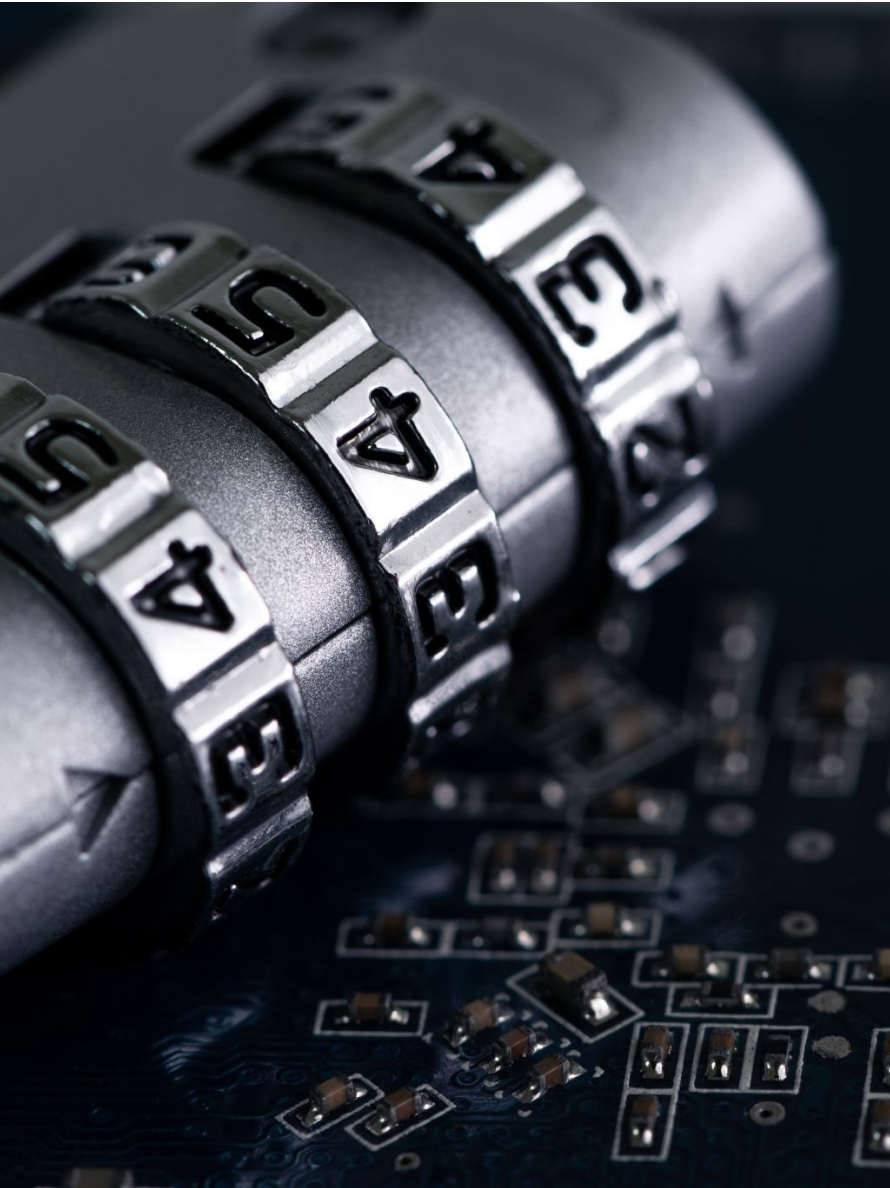
# Classic Synchronization Problems

---

## Dining Philosophers Problem:

- Models five philosophers sitting at a table, each needing two forks to eat.
- The challenge is to design a protocol such that no philosopher starves.
- **Solution:** Various approaches like resource hierarchy, waiter, or Chandy/Misra solution.





---

Another example of deadlock that is not related to a computer system environment.

### **Resource Allocation in Businesses:**

- Two departments wait on resources held by the other, such as a printer and a scanner, causing a deadlock in operations.

# Implementing Synchronization in OS

---

Operating systems provide several APIs and constructs for synchronization:

## **1.POSIX Threads (pthreads):**

- Functions like `pthread_mutex_lock`, `pthread_mutex_unlock`, `pthread_cond_wait`, and `pthread_cond_signal`.

## **2.Windows Threads:**

- Functions like `CreateMutex`, `WaitForSingleObject`, `ReleaseMutex`, `InitializeCriticalSection`, `EnterCriticalSection`, and `LeaveCriticalSection`.

## **3.Java:**

- `synchronized` keyword, `wait`, `notify`, and `notifyAll` methods for object synchronization.

# Activity: Understanding the Sleeping Barber Problem

---

## Scenario:

You own a small barber shop with one barber, one barber chair, and a waiting room with three chairs. When a customer arrives, they will either take a seat in the waiting room or leave if no chairs are available. If the barber is idle and a customer arrives, the barber begins cutting their hair immediately. Once a customer's hair is cut, they leave, and the barber checks for waiting customers before either starting on the next customer or going to sleep if none are waiting.

## Instructions:

- 1. Read the scenario carefully:** Understand the flow of events in the barber shop.
- 2. Answer the questions:** Think critically about the synchronization challenges and propose solutions.

# Questions

1. Describe the main entities in this problem and their roles.
2. What synchronization issues can arise in this scenario?
3. How would you use semaphores to manage access to the barber chair and the waiting room chairs?
4. Explain how the barber transitions between sleeping, cutting hair, and checking for waiting customers.
5. What happens if a customer arrives and all waiting room chairs are occupied?
6. How would you ensure that the barber does not starve and gets time to rest?
7. Discuss potential improvements to this basic synchronization scheme.

Describe the main entities in this problem and their roles.

---

**Entities:** Barber, Customers, Barber Chair, Waiting Room Chairs.

**Roles:**

- The **Barber** cuts hair and sleeps if there are no customers.
- Customers** come to get a haircut, wait if necessary, or leave if the waiting room is full.
- The **Barber Chair** is where haircuts take place.
- Waiting Room Chairs** are where customers wait if the barber chair is occupied.



What synchronization issues can arise in this scenario?

—

- **Race Conditions:** Multiple customers arriving at the same time.
- **Deadlock:** Barber waiting for customers, while customers leave due to perceived unavailability.
- **Starvation:** If customers continually arrive when the barber is about to sleep, the barber might never get a chance to rest.

How would you use semaphores to manage access to the barber chair and the waiting room chairs?

---

- **Barber Chair Semaphore:** Initialize to 1 to ensure only one customer can be in the chair.
- **Waiting Room Semaphore:** Initialize to the number of waiting room chairs (3 in this case) to limit the number of customers waiting.
- **Customer Semaphore:** Incremented when a customer arrives and decremented when the barber starts a haircut, signaling the barber when customers are present.
- **Barber Semaphore:** Signaled when the barber is ready to cut hair, ensuring synchronization between customer arrival and barber's readiness.

Explain how the barber transitions between sleeping, cutting hair, and checking for waiting customers.

---

- The barber starts by checking the **Customer Semaphore**:
- If the semaphore indicates customers are waiting, the barber proceeds to decrement the **Waiting Room Semaphore** and increments the **Barber Semaphore**.
- If no customers are waiting, the barber goes to sleep (blocks on the **Customer Semaphore**).
- When a customer arrives, they increment the **Customer Semaphore**. If the barber is sleeping, this wakes the barber.
- The barber then performs the haircut, and upon finishing, checks the **Customer Semaphore** again to see if more customers are waiting.

What happens if a customer arrives and all waiting room chairs are occupied?

—

- The customer checks the **Waiting Room Semaphore**:

- If the semaphore value is zero (all chairs are occupied), the customer leaves the shop.
- Otherwise, the customer takes a seat in the waiting room (decrements the semaphore) and waits for their turn.

How would you ensure that the barber does not starve and gets time to rest?

---

- Use a condition where after a certain number of haircuts or after a set period, the barber takes a mandatory break.
- Implement a fair scheduling policy to prevent the barber from continuously working without rest due to a steady stream of customers.

Discuss potential improvements to this basic synchronization scheme.

—

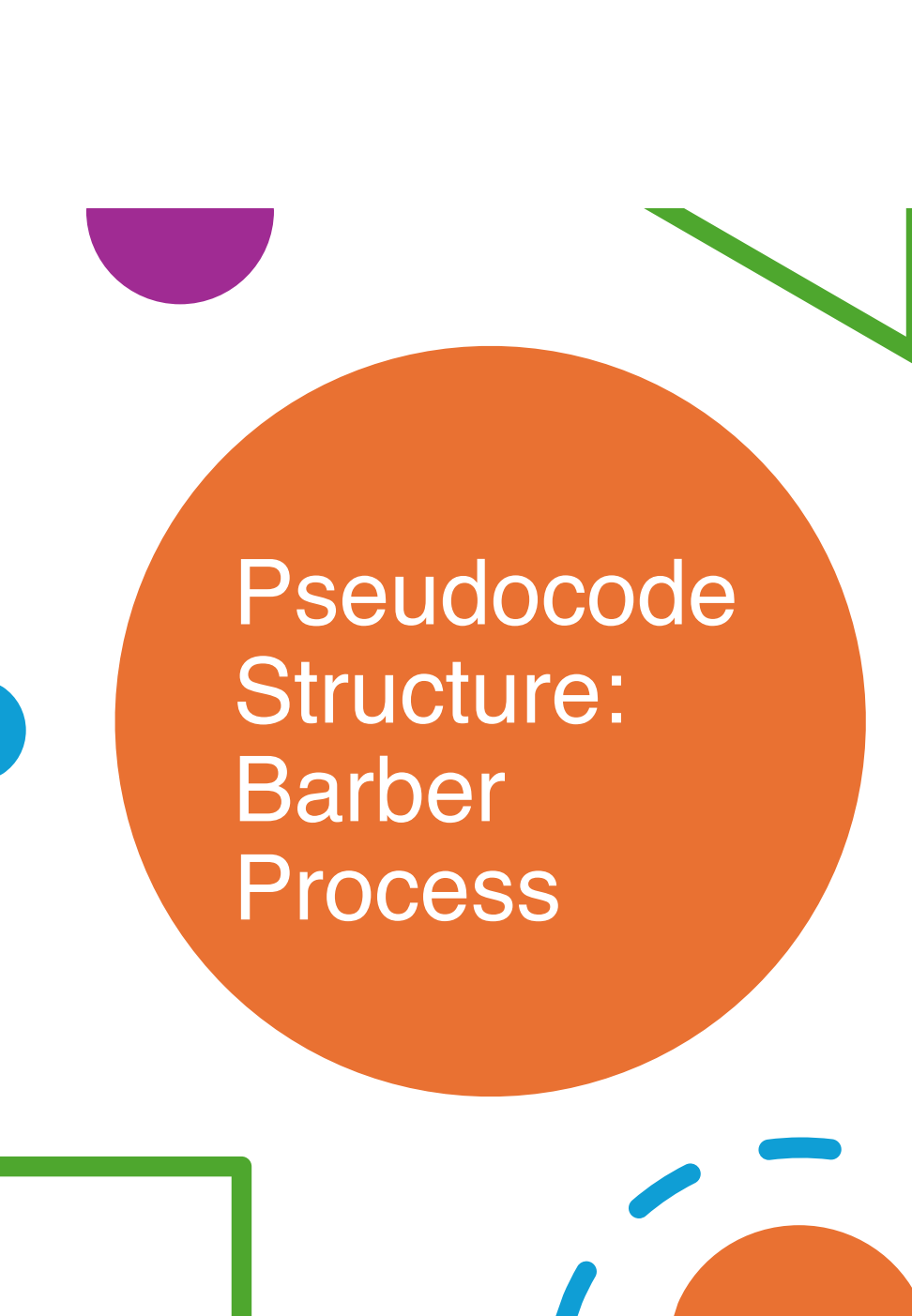
- Priority Queue:** Implement a priority queue for customers, prioritizing those who have waited longer.
- Timed Semaphores:** Use timed semaphores to allow the barber to rest periodically.
- Additional Barbers:** If the shop is busy, consider adding more barbers and expanding the synchronization mechanism to handle multiple barbers and chairs.

Write pseudocode to synchronize the barber and the customers using semaphores.

---

### **Pseudocode Requirements:**

1. Use semaphores to manage the barber chair and the waiting room chairs.
2. Ensure that the barber goes to sleep when no customers are present and wakes up when a customer arrives.
3. Ensure that customers leave if no waiting room chairs are available.
4. Prevent race conditions and ensure mutual exclusion where necessary.



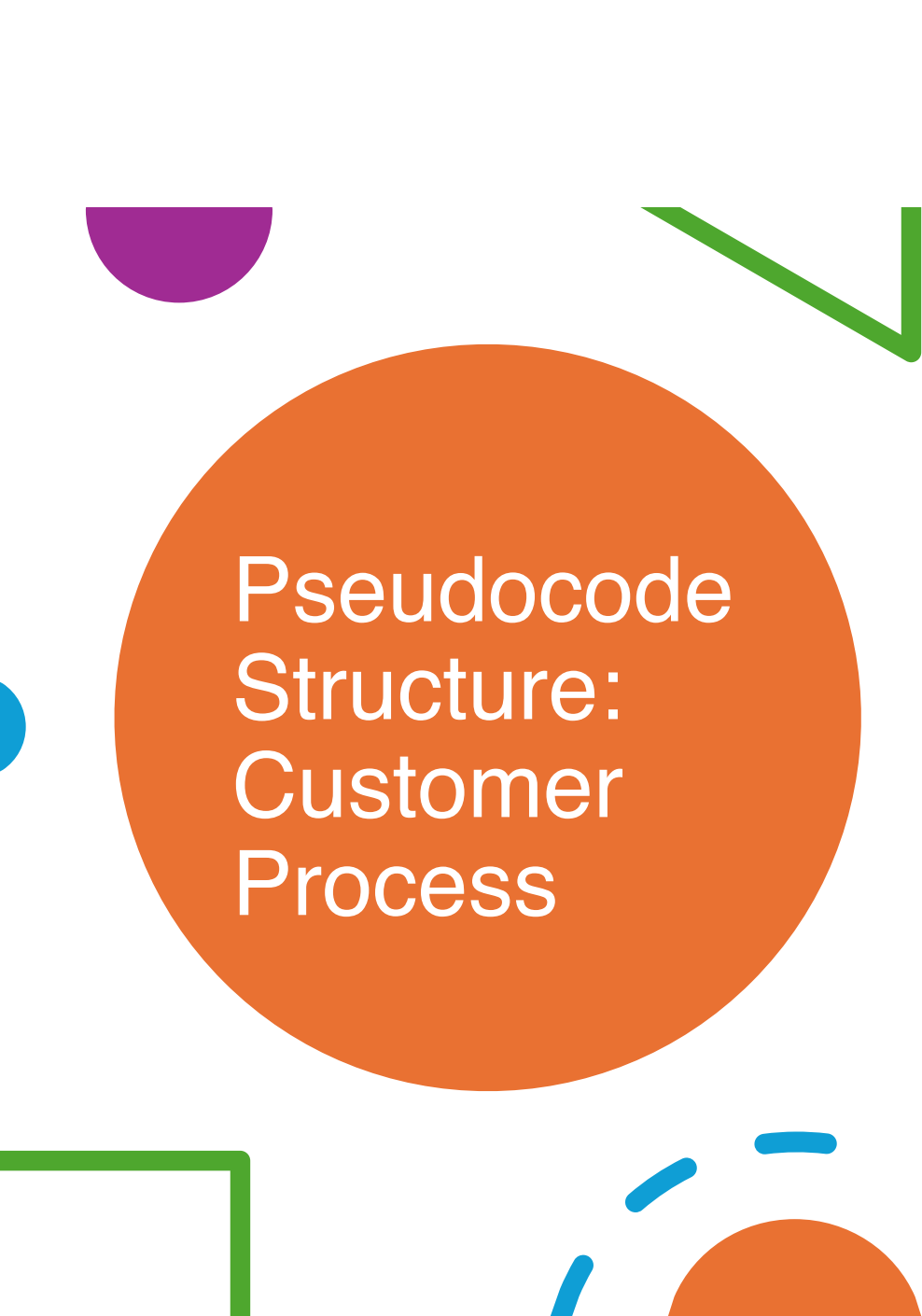
# Pseudocode Structure: Barber Process

```
initialize semaphore barberReady to 0
initialize semaphore accessWRSeats to 1
initialize semaphore customers to 0
initialize integer waitingCustomers to 0
initialize integer numberOfSeats to 3

procedure Barber()
  while true do
    wait(customers)      // Wait for a customer to arrive
    wait(accessWRSeats)  // Acquire access to waiting room
seats
    waitingCustomers = waitingCustomers - 1
    signal(barberReady)  // Notify a customer that barber is ready
    signal(accessWRSeats) // Release access to waiting room
seats
    // Cut hair
    cutHair()
  end procedure

procedure cutHair()
  // Barber cuts the customer's hair
end procedure
```





# Pseudocode Structure: Customer Process

```
procedure Customer()
    wait(accessWRSeats)      // Acquire access to waiting room seats
    if waitingCustomers < numberOfSeats then
        waitingCustomers = waitingCustomers + 1
        signal(customers)    // Notify the barber that there is a customer
        signal(accessWRSeats) // Release access to waiting room seats
        wait(barberReady)    // Wait for the barber to be ready
        // Get hair cut
        getHairCut()
    else
        signal(accessWRSeats) // Release access to waiting room seats
        // Leave the shop as no seats are available
        leaveShop()
    end procedure

procedure getHairCut()
    // Customer gets their hair cut
end procedure

procedure leaveShop()
    // Customer leaves the shop because no waiting room seats are available
end procedure
```

# Best Practices

---

- **Minimize Locking:** Lock only when necessary and keep the critical section as short as possible.
- **Avoid Deadlocks:** Ensure that locks are always acquired and released in a consistent order.
- **Use High-Level Abstractions:** Where possible, use higher-level synchronization constructs like monitors or condition variables instead of low-level locks.