Laboratory 3

Synchronization in Operating System

Aim & Objectives

- Implement process synchronization using semaphores / monitors.

- Objectives:
  - To simulate the working of Semaphores (Binary)
  - To simulate the working of Monitors for any one Classical synchronization Problems.

Semaphores

- A semaphore is a simple integer variable used as a signaling mechanism to solve synchronization problems.
- It ensures that multiple processes can safely access and modify shared resources without interfering with each other, which could otherwise lead to incorrect results.
- The core issue that semaphores address is the critical section problem.

Semaphores

- A "critical section" is a piece of code where a process accesses a shared resource (like a shared memory variable, a file, or a database).
- If multiple processes execute this section simultaneously, it can lead to a "race condition," where the final state of the shared resource depends on the unpredictable order in which the processes run, often producing erroneous outcomes.

Semaphores - Working

- A semaphore controls access to a shared resource through two atomic (indivisible) operations :
    - wait(S): This operation, also known as P(), decrements the semaphore's integer value S.
    - If the value becomes less than or equal to zero, the process that called wait() is blocked and put into a waiting queue.
    - If the value is positive, the process is allowed to continue into its critical section.

Semaphores - Working

- A semaphore controls access to a shared resource through two atomic (indivisible) operations :
  - signal(S): This operation, also known as V(), increments the semaphore's value S.
  - If there are any processes waiting for the semaphore, one of them is unblocked and allowed to proceed.

Semaphores - Types

- There are two main types of semaphores:
- Counting Semaphore: The value can be any non-negative integer. It is used to control access to a resource that has multiple instances. The semaphore is initialized to the number of available resources.
- Binary Semaphore: The value can only be 0 or 1. It functions as a mutex lock, ensuring that only one process can access a resource at a time. It is initialized to 1 (available).

Semaphores - pseudocode

```
wait(Semaphore S)
{
  S = S - 1
  if (S < 0)
  {
    // Block the process
    add process to semaphore's waiting queue
    sleep()    // Process sleeps until signaled
  }
 // else continue, process enters critical section
}
```

```
signal(Semaphore S)
{
  S = S + 1
  if (S <= 0)
  {
    // Remove a process from semaphore's waiting queue
    wakeup(process)
    // Wake up one blocked process
  }
}
```

Semaphores – pseudocode - Tracing

Initial Value of Semaphore S = 1 (Semaphore starts unlocked, one resource available), No Processes waiting	
Process 1 or Thread 1	**Process 2 or Thread 2**
calls wait(S)	
S = 1 - 1 = 0;	
Since $S \geq 0$, P1 or T1 can enter the critical section immediately.	
No blocking occurs. State now: S=0S=0, 0 resources available (busy), 0 waiting processes.	
	calls wait(S)
	S = 0 – 1 = -1
	Since $S < 0$, P2 or T2 is **blocked** and added to the waiting queue.
	State now: S = −1, no resources available, 1 process waiting (P2 or T2).

Semaphores – pseudocode - Tracing

Initial Value of Semaphore S = 1 (Semaphore starts unlocked, one resource available), No Processes waiting	
Process 1 or Thread 1	**Process 2 or Thread 2**
calls signal(S)	
S = -1 + 1 = 0;	
Since S ≤ 0, **one blocked process** (P2 or T2) is removed from waiting queue and resumed.	
P2 or T2 can now enter the critical section.	
State now: S = 0, 0 resources available (P2/T2 in critical section), 0 waiting processes.	Eventually P2/T2 calls Signal(S)
	S = 0 + 1 = 1
	Since S > 0, no waiting processes to wake.
	State now: S =1, 1 resource available (semaphore unlocked), 0 waiting processes.

Semaphores - pseudocode

$S = 1$: The lock is available.

$S = 0$: The lock is held by one process.

$S < 0$: The lock is held, and one or more other processes are waiting for it.

Semaphores - Example

```
#include <stdio.h>

#include <pthread.h>

#include <semaphore.h>
```

Includes the libraries for input/output, POSIX threads, and semaphores.

Semaphores - Example

```
int x = 0;
```

Declares a shared integer variable x initialized to 0.

```
sem_t mutex;
```

Declares a semaphore mutex to control access to x.

Semaphores - Example

```
void* increment_thread(void* arg) {
    sem_wait(&mutex);
    x += 5;
    printf("Incremented x to %d\n", x);
    sem_post(&mutex);
    return NULL;
}
```

Defines the function for the thread that increments x.

sem_wait(&mutex) waits to acquire the semaphore (lock).

Adds 5 to x.

Prints the new value of x.

sem_post(&mutex) releases the semaphore (unlock).

Exits the thread function.

Semaphores - Example

```
void* decrement_thread(void* arg) {
    sem_wait(&mutex);
    x -= 5;
    printf("Decremented x to %d\n",
x);
    sem_post(&mutex);
    return NULL;
}
```

Defines the function for the thread that decrements x.

Waits to acquire the semaphore, subtracts 5 from x, prints it, then releases the semaphore and exits.

Semaphores - Example

```
int main() {

    pthread_t tid1, tid2;

    sem_init(&mutex, 0, 1);
```

Main function declares two thread identifiers.

Initializes mutex as a binary semaphore with initial value 1 (unlocked).

&mutex: This is the address of the semaphore variable to initialize.

0: The pshared parameter. A value of 0 means the semaphore is shared between threads of the same process only (not between different processes).

1: The initial value of the semaphore. Setting it to 1 means it acts as a binary semaphore (or mutex), allowing only one thread to enter the critical section at a time.

Semaphores - Example

```
pthread_create(&tid1, NULL, increment_thread, NULL);
pthread_create(&tid2, NULL, decrement_thread, NULL);
```

Creates two threads, one running increment_thread, the other decrement_thread.

Semaphores - Example

`pthread_join(tid1, NULL);`
`pthread_join(tid2, NULL);`
	Waits for both threads to complete before proceeding.

`sem_destroy(&mutex);`	Destroys the semaphore to clean up resources.
`printf("Final value of x: %d\n", x);`	Prints the final value of x after both threads finish.
`    return 0;`	
`}`	Ends the main program

Create Own Semaphores - Code

- Create a structure that holds the integer value, along with a mutex and a condition variable, which are standard tools for building synchronization primitives like semaphores.

```
// A simple semaphore
implementation
typedef struct {
    pthread_mutex_t mutex;
    pthread_cond_t condition;
    int value;
} semaphore;
```

Create Own Semaphores - Code

- semaphore_init: Initializes the semaphore with a starting value. We use 1 for a binary semaphore, meaning one resource is available.

```
void semaphore_init(semaphore* sem, int initial_value) {
    pthread_mutex_init(&(sem->mutex), NULL);
    pthread_cond_init(&(sem->condition), NULL);
    sem->value = initial_value;
}
```

Create Own Semaphores - Code

- semaphore_wait: A thread calls this before entering its critical section. It locks the mutex to ensure its check is atomic, then checks sem->value. If the value is 0, it waits. Otherwise, it decrements the value and proceeds.

```
// The 'wait' operation
void semaphore_wait(semaphore* sem) {
    pthread_mutex_lock(&(sem->mutex));
    while (sem->value <= 0) {
        // Wait on the condition variable if
semaphore value is not positive
        pthread_cond_wait(&(sem->condition),
&(sem->mutex));
    }
    sem->value--;
    pthread_mutex_unlock(&(sem->mutex));
```

Create Own Semaphores - Code

- semaphore_signal: After leaving the critical section, the thread calls this to release the resource. It increments sem->value and signals one of the waiting threads (if any) to wake up.

```
// The 'signal' operation
void semaphore_signal(semaphore* sem) {
    pthread_mutex_lock(&(sem->mutex));
    sem->value++;
    // Signal one waiting thread
    pthread_cond_signal(&(sem->condition));
    pthread_mutex_unlock(&(sem->mutex));
}
```

Create Own Semaphores - Code

- main and thread_function: The main process and a newly created thread both compete for the same semaphore. Only one can be in its "critical section" at any given time, preventing overlap and ensuring synchronized access.

```
// A function that threads will run
void* thread_function(void* arg) {
    semaphore* sem = (semaphore*)arg;

    printf("Thread is trying to acquire the semaphore...\n");
    semaphore_wait(sem);

    // --- Critical Section Start ---
    printf("Thread has acquired the semaphore and is in the critical section.\n");
    // --- Critical Section End ---

    printf("Thread is releasing the semaphore.\n");
    semaphore_signal(sem);

    return NULL;
}
```

Create Own Semaphores - Code

```
int main() {
    semaphore sem;
    // Initialize as a binary semaphore (value 1)
    semaphore_init(&sem, 1);

    pthread_t thread;
    pthread_create(&thread, NULL, thread_function, (void*)&sem);

    printf("Main process is trying to acquire the semaphore...\n");
    semaphore_wait(&sem);

    // --- Critical Section Start ---
    printf("Main process has acquired the semaphore and is in the critical section.\n");
// --- Critical Section End ---

    printf("Main process is releasing the semaphore.\n");

    semaphore_signal(&sem);

    // Wait for the other thread to finish

    pthread_join(thread, NULL);

    return 0;
}
```

Create Own Semaphores - Code

- #include <pthread.h>: This includes the POSIX threads library. It provides all the necessary functions and data types for creating and managing threads, mutexes, and condition variables (e.g., pthread_t, pthread_create, pthread_mutex_t, pthread_cond_t).

- **pthread_mutex_t mutex**: A mutex (mutual exclusion lock). This is used to protect the value field within the semaphore itself, ensuring that only one thread can modify it at a time. This prevents race conditions inside our semaphore logic.

- **pthread_cond_t condition**: A condition variable. This allows threads to wait ("sleep") efficiently without wasting CPU time (unlike busy-waiting). When a condition is met (i.e., the semaphore becomes available), one of the waiting threads can be "woken up."

- **int value:** The integer counter of the semaphore. For a binary semaphore, this will be 1 (available) or 0 (in use). For a counting semaphore, it can be any non-negative number.

Create Own Semaphores - Code

- **pthread_mutex_init(&(sem->mutex), NULL)**: Initializes the mutex inside the semaphore struct. The NULL argument specifies default attributes for the mutex.
- **pthread_cond_init(&(sem->condition), NULL):** Initializes the condition variable inside the semaphore struct with default attributes.
- **sem->value = initial_value;:** Sets the semaphore's starting count. For a binary semaphore, initial_value will be 1.

- **pthread_mutex_lock(&(sem->mutex))**: The thread first locks the semaphore's internal mutex. This ensures that no other thread can check or change sem->value at the same time.

- **while (sem->value <= 0):** The thread checks if the semaphore is available. If value is 0 or less, the resource is not available. A while loop is used instead of an if statement to handle "spurious wakeups" (where a thread might be woken up even if the condition isn't truly met).

Create Own Semaphores - Code

- **pthread_cond_wait(&(sem->condition), &(sem->mutex)):** If the resource is not available, the thread calls this function to go to sleep. Crucially, this function atomically does two things:
- **It unlocks the mutex**. This is vital so that another thread can call semaphore_signal to release the resource.
- It puts the current thread to sleep until it is signaled.
- When the thread is woken up by a pthread_cond_signal, it automatically re-acquires the lock on the mutex before proceeding.

- **sem->value--:** If the while loop condition was false (or after the thread is woken up and the condition becomes false), it means the resource is available. The thread decrements the semaphore's value to mark it as "in use."

- **pthread_mutex_unlock(&(sem->mutex)):** The thread unlocks the internal mutex, allowing other threads to interact with the semaphore.

Create Own Semaphores - Code

- **pthread_mutex_lock(&(sem->mutex)):** The thread locks the internal mutex to safely modify the semaphore's value.
- **sem->value++:** It increments the semaphore's value, indicating that a resource has been freed.
- **pthread_cond_signal(&(sem->condition)):** This function "wakes up" at least one of the threads that are currently waiting inside pthread_cond_wait. The awakened thread will then attempt to re-acquire the mutex and re-check the while loop condition in semaphore_wait.
- **pthread_mutex_unlock(&(sem->mutex)):** The thread unlocks the internal mutex.

- **semaphore* sem = (semaphore*)arg;**: The arg passed from pthread_create is a void pointer. This line casts it back to its original type, a pointer to a semaphore.
- **semaphore_wait(sem):** The thread calls our wait function to request access to the resource. The function will not return until the semaphore is acquired.
- // --- Critical Section Start ---: This marks the beginning of the code that accesses the shared resource. Only one thread can be in this section at a time.
- // --- Critical Section End ---: Marks the end of the protected code.
- **semaphore_signal(sem):** After leaving the critical section, the thread calls signal to release the semaphore, allowing another waiting thread to proceed.
- **return NULL;:** The function returns a NULL pointer, as it has no specific result to return.

Create Own Semaphores - Code

- **semaphore sem;**: Declares a variable of our custom semaphore type.
- **semaphore_init(&sem, 1);**: Initializes our semaphore with a value of 1, making it a binary semaphore that is initially available.
- **pthread_t thread;**: Declares a variable to hold the thread identifier.
- **pthread_create(&thread, NULL, thread_function, (void*)&sem);**: This function creates and starts a new thread.
- **&thread**: A pointer to store the ID of the new thread.
- **NULL**: Use default thread attributes.
- **thread_function**: The function the new thread will execute.
- **(void*)&sem**: A pointer to our semaphore, which is passed as an argument to thread_function.
- The main thread then proceeds to compete for the same semaphore by calling semaphore_wait(&sem).
- **pthread_join(thread, NULL);**: The main thread waits here until the other thread (thread) has finished its execution. This ensures the program doesn't exit prematurely.
- **return 0;**: Indicates that the program has completed successfully.

Create Own Semaphores - Code

- The main program starts.
- It creates a semaphore (our key holder) and sets its value to 1 (key available).
- It creates a new thread. Now we have two workers: the main thread and the new thread.
- Both threads will try to run semaphore_wait to get the key. It's a race!
- Whoever gets there first will lock the semaphore, change its value to 0, and enter its "critical section."
- The second thread to arrive will see the value is 0 and will be forced to wait.
- The first thread finishes its work, calls semaphore_signal, changes the value back to 1, and rings the bell.
- The waiting second thread wakes up, sees the value is now 1, and finally gets the key to do its work.
- Once both threads are done, the main program cleans up and exits.

Classical Synchronization Problem

- The classical synchronization problems in operating systems are fundamental **concurrency issues** arising when multiple processes or threads need coordinated access to shared resources.
- These problems illustrate challenges such as avoiding deadlock, ensuring mutual exclusion, and preventing race conditions.
  1. Bounded-Buffer (Producer-Consumer) Problem
  2. Dining Philosophers Problem
  3. Readers-Writers Problem
  4. Sleeping Barber Problem

Bounded-Buffer (Producer-Consumer) Problem

- This involves a **fixed-size buffer shared between two types of processes**:
  - **Producers**, who generate data and place it in the buffer, and
  - **Consumers**, who remove data from the buffer.
- Synchronization ensures that *producers don't add data when the buffer is full* and *consumers don't remove data when the buffer is empty*.
- The challenge is to coordinate access so that no data is lost or corrupted while ensuring efficient buffer usage.
- **Semaphores or mutex locks** are typically used for this synchronization.

Producer-Consumer – Semaphore Solution

- Semaphores ensure proper synchronization by controlling access to the buffer slots and mutually exclusive operations on shared data structures.
- **Shared Entities:**
  - A fixed-size buffer to hold produced data.
  - Two indices: **in** for the next slot to insert by the producer, and **out** for the next slot to remove by the consumer.
  - **Three semaphores:**
    - **mutex (binary semaphore)** for mutual exclusion to ensure that only one process accesses the buffer or modifies in/out pointers at a time.
    - **empty (counting semaphore)** initialized to the buffer size, representing the number of empty slots.
    - **full (counting semaphore)** initialized to zero, representing the number of filled slots.

Bounded-Buffer (Producer-Consumer) Problem

Initialize	
semaphore mutex = 1	// Binary semaphore for mutual exclusion
semaphore empty = BUFFER_SIZE	// Counting semaphore for empty slots
semaphore full = 0	// Counting semaphore for filled slots
integer in = 0	// Producer index
integer out = 0	// Consumer index
buffer[BUFFER_SIZE]	// Shared circular buffer

Bounded-Buffer (Producer-Consumer) Problem

Producer Process:	

while true do
        produce item
	Prepare the data to be inserted into the buffer.

wait(empty)	Block if the buffer is full (no empty slots), otherwise decrement empty.
wait(mutex)	Lock to enter the critical section so only one process can modify the buffer.
buffer[in] = item	Insert the produced item at index in.
in = (in + 1) mod BUFFER_SIZE	Move in index to the next position circularly to wrap around.
signal(mutex)	Release the lock (exit critical section).
signal(full)	Increment the count of filled slots to notify the consumer.

Bounded-Buffer (Producer-Consumer) Problem

Consumer Process:	

while true do
        wait(Full)
	Block if the buffer is empty (no full slots), otherwise decrement full.

wait(mutex)	Lock to enter the critical section to safely access the buffer.
item = buffer[out]	Remove the item at index out.
out = (out + 1) mod BUFFER_SIZE	Move out index circularly for the next removal.
signal(mutex)	Release the lock (exit critical section).
signal(empty)	Increment empty slots count to notify the producer.
consume item	Process the consumed item.

Producer-Consumer – Semaphore Solution

- **Producer Process Flow:**
  - Wait (decrement) on empty: Wait for at least one empty slot.
  - Wait (lock) on mutex: Enter the critical section.
  - Insert an item into the buffer at index in.
  - Update in index in circular fashion (e.g. (in + 1) % BUFFER_SIZE).
  - Signal (unlock) mutex: Leave critical section.
  - Signal (increment) full: Increment count of full slots.

Producer-Consumer – Semaphore Solution

- **Consumer Process Flow:**
  - Wait (decrement) on full: Wait for at least one full slot.
  - Wait (lock) on mutex: Enter the critical section.
  - Remove an item from the buffer at index out.
  - Update out index in circular fashion (e.g. (out + 1) % BUFFER_SIZE).
  - Signal (unlock) mutex: Leave critical section.
  - Signal (increment) empty: Increment count of empty slots.

Dining Philosophers Problem

- This models a scenario where **multiple philosophers sitting around a circular table need two shared chopsticks (resources) to eat.**
- Each **philosopher must pick up the two adjacent chopsticks**, **one at a time, to eat, leading to potential deadlock or starvation if all pick up one chopstick simultaneously and wait indefinitely for the other**.
- The problem highlights resource allocation and deadlock prevention techniques.

Dining Philosophers Problem

- Imagine 5 philosophers sitting around a circular table.
- Between each pair of philosophers, there is one chopstick (fork).
- Each philosopher alternates between thinking and eating.
- To eat, a philosopher needs both the chopsticks on their left and right.
- Since chopsticks are shared, a philosopher can only pick up a chopstick if it is not already being used by a neighbor.
- After eating, the philosopher puts down both chopsticks and returns to thinking.

Dining Philosophers Problem

- What Problems to be Avoided:
  - **Deadlock:** If every philosopher picks up the chopstick on their left simultaneously, then all wait forever for the right chopstick to become free. This circular waiting causes a deadlock, and no one can eat.
  - **Starvation:** Some philosophers may wait indefinitely if others continuously get access to the chopsticks, leading to unfair resource allocation.
  - **Mutual Exclusion:** Ensuring that no two adjacent philosophers eat at the same time so that no chopstick is shared simultaneously is critical.

Dining Philosophers Problem

```
semaphore chopstick[5];     // One semaphore per chopstick, initialized to 1

semaphore mutex;            // Semaphore to limit number of philosophers at the table



initialize all chopstick[i] to 1

initialize mutex to 4       // Allow maximum 4 philosophers to try picking chopsticks
```

Dining Philosophers Problem

```
function philosopher(i):
    while true:
        think()
```

Each philosopher spends some time thinking (not competing for chopsticks).

Dining Philosophers Problem

```
wait(mutex)            // Request to enter - limit philosophers to 4
```

- Philosopher requests permission to try eating from the "dining room".
- Since the mutex semaphore allows only up to 4 philosophers, this prevents all 5 trying to eat at the same time, breaking the circular wait.

Dining Philosophers Problem

```
wait(chopstick[i])        // Pick left chopstick

      wait(chopstick[(i + 1) % 5])     // Pick right chopstick
```

Philosopher tries to pick up the left chopstick by waiting on its semaphore.

Philosopher then tries to pick up the right chopstick (using modulo for circular seating).

If the chopstick is in use, the philosopher waits until it is free.

Dining Philosophers Problem

```
 eat()
```

Philosopher eats now that both chopsticks are acquired.

```
 signal(chopstick[i])       // Put down left chopstick
        signal(chopstick[(i + 1) % 5]) // Put down right chopstick
```

Philosopher puts down left and right chopsticks, signaling availability to others.

Dining Philosophers Problem

```
signal(mutex)          // Philosopher leaves the table - allows another to enter
```

Philosopher signals the mutex semaphore to indicate they have finished trying to eat.

This allows another philosopher to enter the dining room and attempt to eat, maintaining the limit of 4.

Readers-Writers Problem

- This problem deals with a **shared database accessed by two types of processes**:
  - **Readers, which can concurrently read the data without conflict**, and
  - **Writers, which require exclusive access to modify the data**.
- The challenge is to **ensure that multiple readers can access the data simultaneously**, **but writers have exclusive access**, preventing inconsistent reads or writes.
- Synchronization mechanisms manage these constraints to avoid race conditions and starvation.

Readers-Writers Problem

- The Readers-Writers Problem involves multiple processes (or threads) that want to access a shared resource (like a file or database):
  - **Readers** can read the resource simultaneously because reading does not change data.
  - **Writers need exclusive access** — only one writer at a time and no readers should read while writing, to prevent inconsistent or corrupted data.

Readers-Writers Problem

- Concerns Addressed Using Semaphores:
  - Mutual exclusion for writers to **prevent data corruption**.
  - Allowing multiple readers to **read simultaneously**.
  - Ensuring **writers have exclusive access** (no readers or other writers).
  - Preventing **starvation of readers or writers** by managing access fairness.

Readers-Writers Problem

Initialize semaphores and counter	
Semaphore mutex = 1	mutex semaphore controls mutual exclusion for incrementing/decrementing readCount safely.
Semaphore write = 1	write semaphore ensures exclusive access for writers (and blocks readers when held).
Integer readCount = 0	readCount tracks how many readers are currently reading.

function reader():
    wait(mutex)
	wait(mutex): Enter critical section to update readCount.

readCount = readCount + 1	readCount++: Increment number of active readers.

if readCount == 1
        wait(write)
    signal(mutex)
	If this is the first reader, wait(write) is called to lock out writers (block writing).
	signal(mutex): Exit critical section on readCount.

Readers-Writers Problem

// Critical section: reading resource // Reader performs reading (critical section).	
wait(mutex)	After reading, wait(mutex) is called to enter critical section again for updating readCount.
Semaphore write = 1	
readCount = readCount - 1	readCount--: Decrement number of active readers.

if readCount == 0
        signal(write)
    signal(mutex)
	If this is the last reader leaving, signal(write) releases write semaphore to allow writers to proceed.
	signal(mutex): Exit critical section.

Readers-Writers Problem

Writer Process:	

function writer():
    wait(write)
	Acquire exclusive access — no other writer or reader can enter critical section.

// Critical section: writing resource	Write to shared resource (critical section).
signal(write)	Release exclusive access for others.

Readers-Writers Problem - Tracing

- **Initial State:**
  - mutex = 1 (protects readCount)
  - write = 1 (exclusive access for writers)
  - readCount = 0
- **Step 1: First Reader Enters**
  - Reader calls wait(mutex) → decrements mutex from 1 to 0.
  - Reader increments readCount from 0 to 1.
    - Since readCount == 1, reader calls wait(write) → decrements write from 1 to 0 (blocks writers).
    - Reader calls signal(mutex) → increments mutex back to 1.
    - Reader accesses the resource for reading.
- **Semaphore values:**
  - mutex = 1
  - write = 0
  - readCount = 1

Readers-Writers Problem - Tracing

- **Step 2: Second Reader Enters (Concurrent Reading Allowed)**
  - Reader calls wait(mutex) → mutex goes 1 → 0.
  - Reader increments readCount from 1 to 2.
  - Since readCount != 1, no call to wait(write).
  - Reader calls signal(mutex) → increments mutex to 1.
  - Reader accesses resource simultaneously with the first reader.
- **Semaphore values:**
  - mutex = 1
  - write = 0
  - readCount = 2

Readers-Writers Problem - Tracing

- **Step 3: Writer Attempts to Enter**
  - Writer tries wait(write).
  - Since write is 0, writer blocks waiting for the semaphore.
  - Writer is blocked here until all readers finish.
- **Step 4: First Reader Leaves**
  - Reader calls wait(mutex) → mutex 1 → 0.
  - Reader decrements readCount from 2 to 1.
  - Since readCount != 0, no signal on write.
  - Reader calls signal(mutex) → mutex 0 → 1.
- **Semaphore values:**
  - mutex = 1
  - write = 0
  - readCount = 1

Readers-Writers Problem - Tracing

- **Step 5: Last Reader Leaves**
  - Reader calls wait(mutex) → mutex 1 → 0.
  - Reader decrements readCount from 1 to 0.
  - Since readCount == 0, reader calls signal(write) → increments write semaphore from 0 to 1, unblocking one waiting writer.
  - Reader calls signal(mutex) → mutex 0 → 1.
- Semaphore values:
  - mutex = 1
  - write = 1
  - readCount = 0

Readers-Writers Problem - Tracing

- **Step 6: Writer Proceeds**
  - Writer now successfully performs wait(write) → decrements write 1 → 0, entering critical section exclusively.
  - Writer writes to resource.

- **Semaphore values:**
  - mutex = 1
  - write = 0 (Writer holds access exclusively)
  - readCount = 0

Sleeping Barber Problem

- This problem **models a barber shop** where **one barber serves customers**.
- If **no customers are present, the barber sleeps**;
- when a **customer arrives, the barber wakes** to give a haircut.
- Customers **either wait if there are available waiting chairs or leave if the shop is full**.
- Synchronization ensures proper coordination of **customer arrivals, waiting, and service** without loss or deadlock.