# OSC Lab Task 7: Solving a Concurrency Problem using semaphore and Simulating Deadlock in resource allocation

## Task 7-1: Concurrency problem

A coffee shop has several self-service kiosks where customers can place their coffee orders. Each kiosk can handle one customer at a time. The total number of kiosks is K. If all kiosks are in use, arriving customers must wait for a kiosk to become free. When a customer finishes placing their order, the next customer in line is allowed to use the kiosk. Using semaphores, write the C code for both the order process/thread and the completion process/thread, ensuring that the kiosks are efficiently managed.

## Task 7-2 - Simulating and Preventing Deadlock in Resource Allocation

**Objective:**
In this lab, you will learn how deadlocks occur in multithreaded programs that require access to shared resources, as well as strategies to prevent them. You will intentionally create a deadlock in a C program using threads and mutexes, then modify the program to avoid deadlock by implementing a resource ordering strategy.

### Task Description:

In this exercise, you will:

1. **Simulate a Deadlock** by creating two shared resources (e.g., `Resource A` and `Resource B`) and two threads (`Thread 1` and `Thread 2`) that each need exclusive access to both resources to complete their work. By structuring the program in a specific way, you will cause a deadlock where both threads end up waiting indefinitely for resources held by each other.
2. **Analyze Deadlock Conditions** by observing how mutual exclusion, hold and wait, no preemption, and circular wait contribute to deadlock.
3. **Implement a Deadlock Prevention Solution** using a resource ordering technique to avoid deadlock, thus allowing both threads to complete their work successfully.

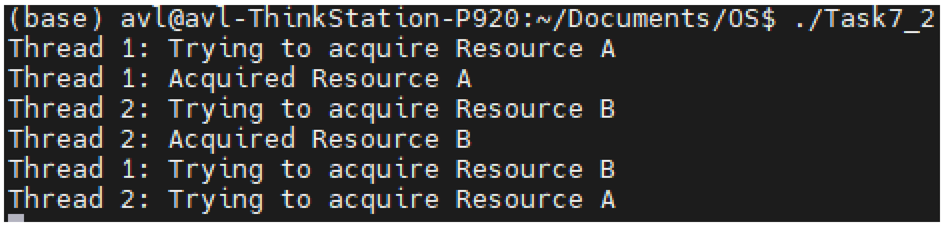
### Part 1: Simulating a Deadlock

1. **Create Shared Resources**:
   - Use two mutexes to represent two resources, `Resource A` and `Resource B`.
2. **Create Two Threads**:
   - Define two threads, `Thread 1` and `Thread 2`, each of which tries to access both resources in a different order:

        - Thread 1 should lock Resource A first and then try to lock Resource B.
        - Thread 2 should lock Resource B first and then try to lock Resource A.
    - Add a small delay (e.g., using sleep(1)) between locking the first and second resources to increase the likelihood of a deadlock.
3. **Observe the Deadlock**:
    - Run the program and observe that it hangs (deadlocks) because each thread is holding one resource and waiting for the other.

**Sample Output for Deadlock Scenario:**

Your program's output should look similar to this before it hangs due to the deadlock:



```
(base) avl@avl-ThinkStation-P920:~/Documents/OS$ ./Task7_2
Thread 1: Trying to acquire Resource A
Thread 1: Acquired Resource A
Thread 2: Trying to acquire Resource B
Thread 2: Acquired Resource B
Thread 1: Trying to acquire Resource B
Thread 2: Trying to acquire Resource A
```

// Program hangs here due to deadlock

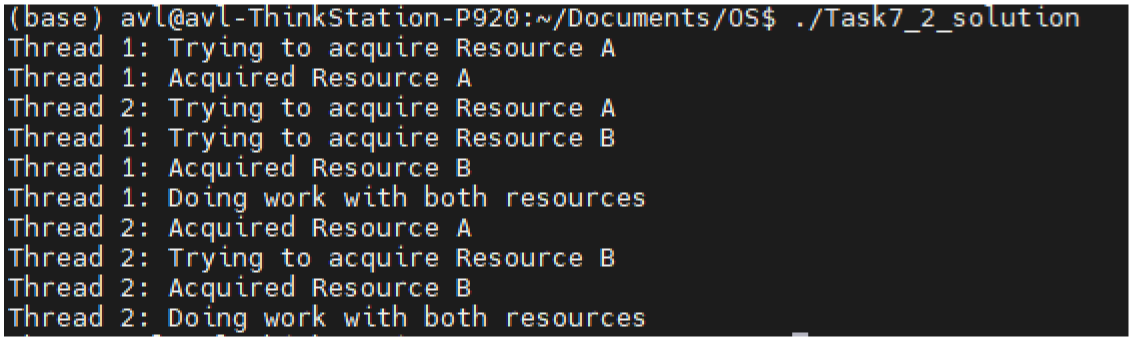
## Part 2: Preventing Deadlock

To prevent the deadlock, modify your program to use a **Resource Ordering Strategy**.

1. **Resource Ordering Strategy**:
    - Ensure that both threads lock the resources in the same order. For example:
        - Always lock Resource A first, then lock Resource B for both threads.
    - This consistent order breaks the circular wait condition, which is necessary for a deadlock to occur.
2. **Implement the Solution**:
    - Modify both threads to lock resources in the same order (Resource A first, then Resource B).
    - Test the program to confirm that it runs without deadlocking.

**Expected Output for Deadlock-Free Scenario:**

With the deadlock prevention strategy in place, your program should produce output like this and complete successfully without hanging:



```
(base) avl@avl-ThinkStation-P920:~/Documents/OS$ ./Task7_2_solution
Thread 1: Trying to acquire Resource A
Thread 1: Acquired Resource A
Thread 2: Trying to acquire Resource A
Thread 1: Trying to acquire Resource B
Thread 1: Acquired Resource B
Thread 1: Doing work with both resources
Thread 2: Acquired Resource A
Thread 2: Trying to acquire Resource B
Thread 2: Acquired Resource B
Thread 2: Doing work with both resources
```

## Learning Outcomes

By completing this assignment, you will learn:

1. How to simulate and observe a deadlock in a multithreaded program.
2. The conditions that lead to deadlock and how they apply to real-world resource allocation scenarios.
3. A practical method for preventing deadlock, which is applicable in many programming and system design contexts.