**Week 10 (Multithreading)**

**Example 1:**

You are managing the inventory for a small online store. You have a shared inventory count for a popular item. Two background processes—one handling incoming shipments (Producer) and one handling customer orders (Consumer)—run concurrently to update this inventory count.

**Shared Resource:** Define a global variable, INVENTORY\_COUNT, initialized to 100.

Create two functions:

* add\_inventory(num\_adds): Simulates receiving shipments by adding a fixed number of items to the inventory.
* remove\_inventory(num\_removes): Simulates fulfilling orders by subtracting a fixed number of items from the inventory.

**Synchronization:** Launch **two** threads for add\_inventory and **three** threads for remove\_inventory. Each thread should perform its operation **50,000** times. Implement a **threading.Lock** to ensure that all 50,000 additions and 50,000 subtractions occur correctly, preventing any lost updates (race conditions) to the shared INVENTORY\_COUNT.

*import threading*

*import time*

*# --- 1. Shared Resources ---*

*INVENTORY\_COUNT = 100*

*NUM\_OPERATIONS = 50000*

*# Initialize the Lock for synchronization*

*INVENTORY\_LOCK = threading.Lock()*

*# --- 2. Thread Worker Functions ---*

*def add\_inventory(num\_adds):*

*"""Adds items to the shared inventory count safely."""*

*global INVENTORY\_COUNT*

*thread\_name = threading.current\_thread().name*

*for \_ in range(num\_adds):*

*# Acquire the lock to enter the critical section*

*with INVENTORY\_LOCK:*

*# Critical Section: Only one thread can execute this at a time*

*INVENTORY\_COUNT += 1*

*# The lock is automatically released upon exiting the 'with' block*

*def remove\_inventory(num\_removes):*

*"""Removes items from the shared inventory count safely."""*

*global INVENTORY\_COUNT*

*thread\_name = threading.current\_thread().name*

*for \_ in range(num\_removes):*

*# Acquire the lock to enter the critical section*

*with INVENTORY\_LOCK:*

*# Critical Section: Only one thread can execute this at a time*

*INVENTORY\_COUNT -= 1*

*# --- 3. Execution ---*

*if \_\_name\_\_ == '\_\_main\_\_':*

*threads = []*

*# Calculate the expected result for verification*

*expected\_result = INVENTORY\_COUNT + (2 \* NUM\_OPERATIONS) - (3 \* NUM\_OPERATIONS)*

*start\_time = time.time()*

*# 2 Producer Threads (Adding)*

*for i in range(2):*

*t = threading.Thread(target=add\_inventory, args=(NUM\_OPERATIONS,), name=f"Producer-{i+1}")*

*threads.append(t)*

*t.start()*

*# 3 Consumer Threads (Removing)*

*for i in range(3):*

*t = threading.Thread(target=remove\_inventory, args=(NUM\_OPERATIONS,), name=f"Consumer-{i+1}")*

*threads.append(t)*

*t.start()*

*# Wait for all 5 threads to complete*

*for t in threads:*

*t.join()*

*end\_time = time.time()*

*print("-" \* 40)*

*print(f"Total time taken: {end\_time - start\_time:.4f} seconds.")*

*print(f"Initial Inventory: {100}")*

*print(f"Total Additions: {2 \* NUM\_OPERATIONS}")*

*print(f"Total Removals: {3 \* NUM\_OPERATIONS}")*

*print("-" \* 40)*

*print(f"Expected Final Inventory: {expected\_result}")*

*print(f"Actual Synchronized Inventory: {INVENTORY\_COUNT}")*

*if INVENTORY\_COUNT == expected\_result:*

*print("\n SUCCESS: Synchronization with Lock was perfect!")*

*else:*

*# This branch indicates a problem, likely due to a missed lock or excessive complexity*

*print(f"\n FAILURE: Lost updates detected. Error: {INVENTORY\_COUNT - expected\_result}")*

**Exercise:**

1. Design a Python application to simulate a banking system involving two shared bank accounts, Account A (starting balance: $1000) and Account B (starting balance: $500).

Write a function, transfer\_money(sender\_account, receiver\_account, amount), and launch five threads that concurrently attempt to call this function with random amounts between $1 and $100.

1. Implement Necessary Synchronization: Use two separate threading.Lock objects (one for each account) to protect the integrity of the individual account balances.
2. Demonstrate Deadlock (Optional): Show how a deadlock can occur if the two locks are acquired in different orders by different threads (e.g., Thread 1 acquires Lock A then Lock B, while Thread 2 acquires Lock B then Lock A).

The final combined balance (A+B) must always equal the initial combined balance ($1500), regardless of the transfer order.

1. A company wants to analyze large text files. You are asked to design a Python program that:

Splits a given text file into **five equal parts**.

Creates **five threads**, where each thread:

* 1. Reads one part of the file,
  2. Counts the number of words in its chunk,
  3. Stores the count in a shared list.

After all threads complete, the main thread should sum up the word counts and display the **total number of words in the file**.

1. Design a Python program to perform **matrix multiplication** using multithreading. You are given two square matrices A and B of size N x N. Create **N threads**, where each thread computes one row of the result matrix C = A × B. Use the threading module to launch the threads and join them after completion. Store the results in a shared 2D list (matrix C). Ensure synchronization so that no race condition occurs when multiple threads update the shared matrix. Compare the performance of the threaded version with a **sequential version** for large matrices.