Bahria University,

Karachi Campus



# LAB EXPERIMENT NO.

**10**

# LIST OF TASKS

|  |  |
| --- | --- |
| TASK NO | OBJECTIVE |
| 1 | Queues and Message Passing. Implement a simple messaging system usingqueues for communication between threads |
| 2 | Implementation of Queues and Locks. Integrate the concepts ofqueues and locks in a more complex scenario. |
| 3 | Locks and Synchronization in a Banking System |
|  |  |

Submitted On: 13/12/2023

(Date: DD/MM/YY)

**QUESTION NO: 01** Queues and Message Passing. Implement a simple messaging system using queues for communication between threads.

### Task:

1. Create a Python program that simulates a messaging system.
2. Implement two threads, one acting as a sender and the other as a receiver.
3. Use a queue to pass messages from the sender to the receiver.
4. Ensure proper synchronization to handle multiple messages correctly.
5. Display the received messages in the console.

**CODE:**

import threading import queue import time

def sender\_thread(message\_queue, messages): for message in messages:

time.sleep(1) message\_queue.put(message)

message\_queue.put(None)

def receiver\_thread(message\_queue): while True:

message = message\_queue.get() if message is None:

break

print(f"Received message: {message}") def main():

message\_queue = queue.Queue()

messages\_to\_send = ["Hello", "How are you?", "Goodbye"]

sender = threading.Thread(target=sender\_thread, args=(message\_queue, messages\_to\_send))

receiver = threading.Thread(target=receiver\_thread, args=(message\_queue,)) sender.start()

receiver.start() sender.join() receiver.join()

if name == " main ":

main()

**OUTPUT:**



**QUESTION NO: 02** Implementation of Queues and Locks. Integrate the concepts of queues and locks in a more complex scenario.

### Task:

1. Design a program that models a restaurant with multiple chefs and waiters.
2. Use queues to represent orders placed by customers and messages sent between chefs and waiters.
3. Implement locks to synchronize access to shared resources such as the kitchen or a list of orders.
4. Simulate the flow of orders, preparation by chefs, and delivery by waiters.
5. Ensure that the program runs smoothly in a multithreaded environment.

**CODE:**

import threading import queue import time import random class Restaurant:

def init (self): self.order\_queue = queue.Queue() self.completed\_orders = [] self.lock = threading.Lock()

def place\_order(self, order): with self.lock:

print(f"Customer placed an order: {order}") self.order\_queue.put(order)

def prepare\_order(self): while True:

order = self.order\_queue.get() if order == "exit":

break

print(f"Chef is preparing order: {order}") time.sleep(random.uniform(1, 3)) print(f"Chef completed order: {order}") with self.lock:

self.completed\_orders.append(order) def serve\_order(self):

while True:

with self.lock:

if self.completed\_orders:

order = self.completed\_orders.pop(0) print(f"Waiter is serving order: {order}")

time.sleep(random.uniform(1, 3))

chef\_thread = threading.Thread(target=restaurant.prepare\_order) waiter\_thread = threading.Thread(target=restaurant.serve\_order) chef\_thread.start()

waiter\_thread.start() for i in range(5):

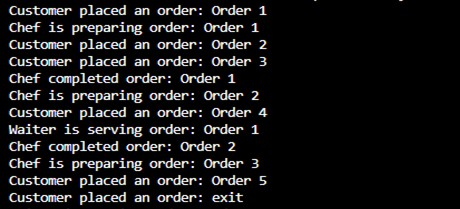
order = f"Order {i + 1}" restaurant.place\_order(order) time.sleep(random.uniform(0.5, 1.5))

restaurant.place\_order("exit") chef\_thread.join() waiter\_thread.join()

if name == " main ":

main()

**OUTPUT:**



**QUESTION NO: 03** Locks and Synchronization in a Banking System

### Task:

1. Develop a Python program simulating a banking system with multiple customer accounts (represented as balances).
2. Implement multiple threads to perform transactions such as deposits and withdrawals on these accounts.
3. Without using locks, intentionally create a scenario where race conditions or data corruption can occur during concurrent transactions.
4. Run the program and observe the unexpected behavior resulting from the lack of synchronization.
5. Modify the program to use locks to ensure that only one thread can access an account for a transaction at a time.
6. Run the modified program and verify that the accounts are accessed safely without data corruption, ensuring the integrity of each transaction.
7. Output the final balances of the customer accounts to confirm that synchronization has been achieved.

**CODE:**

import threading import time import random class Bank:

def init (self, accounts): self.accounts = accounts self.lock = threading.Lock()

def deposit(self, account\_id, amount): with self.lock:

current\_balance = self.accounts[account\_id] new\_balance = current\_balance + amount self.accounts[account\_id] = new\_balance

print(f"Deposited {amount} into Account {account\_id}. New balance:

{new\_balance}")

def withdraw(self, account\_id, amount): with self.lock:

current\_balance = self.accounts[account\_id] if current\_balance >= amount:

new\_balance = current\_balance - amount self.accounts[account\_id] = new\_balance

print(f"Withdrew {amount} from Account {account\_id}. New balance:

{new\_balance}")

else:

print(f"Insufficient funds in Account {account\_id} to withdraw

{amount}")

def simulate\_transactions(bank, num\_transactions): for \_ in range(num\_transactions):

account\_id = random.randint(0, len(bank.accounts) - 1) amount = random.randint(1, 100)

transaction\_type = random.choice(["deposit", "withdraw"]) if transaction\_type == "deposit":

bank.deposit(account\_id, amount)

else:

bank.withdraw(account\_id, amount)

def main():

num\_accounts = 3

initial\_balances = [1000, 1500, 2000] accounts = dict(enumerate(initial\_balances)) bank = Bank(accounts)

print("Simulating transactions without locks (race condition):") threads = []

for \_ in range(5):

thread = threading.Thread(target=simulate\_transactions, args=(bank, 10)) threads.append(thread)

thread.start() for thread in threads:

thread.join()

print("Balances after transactions without locks:", bank.accounts) bank.accounts = dict(enumerate(initial\_balances))

print("\nSimulating transactions with locks (ensuring synchronization):") threads = []

for \_ in range(5):

thread = threading.Thread(target=simulate\_transactions, args=(bank, 10)) threads.append(thread)

thread.start() for thread in threads:

**OUTPUT**

