Bahria University,

Karachi Campus



LAB EXPERIMENT NO.

**10**

LIST OF TASKS

|  |  |
| --- | --- |
| TASK NO | OBJECTIVE |
| **01** | **Queues and Message Passing** |
| **02** | **Implementation of Queues and Locks** |
| 03 | Locks and Synchronization in a Banking System |

Submitted On:

**Date: 13/Dec/2023**

**TASK # 01: Queues and Message Passing**

import threading

import queue

import time

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

    for message in messages:

        print(f"Sender: Sending message - {message}")

        message\_queue.put(message)

        time.sleep(1)

    message\_queue.put(None)

def receiver\_thread(message\_queue):

    while True:

        message = message\_queue.get()

        if message is None:

            break

        print(f"Receiver: Received message - {message}")

message\_queue = queue.Queue()

sender = threading.Thread(target=sender\_thread, args=(message\_queue, ["Hello", "How are you?", "Goodbye"]))

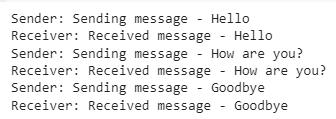
receiver = threading.Thread(target=receiver\_thread, args=(message\_queue,))

sender.start()

receiver.start()

sender.join()

receiver.join()



**TASK # 2: : Implementation of Queues and Locks**

import threading

import queue

import time

import random

class Restaurant:

    def \_\_init\_\_(self):

        self.orders\_queue = queue.Queue()

        self.kitchen\_lock = threading.Lock()

        self.orders\_in\_kitchen = []

        self.waiters\_lock = threading.Lock()

    def place\_order(self, order):

        with self.waiters\_lock:

            print(f"Waiter: Placed order - {order}")

            self.orders\_queue.put(order)

    def process\_order(self, chef\_id):

        while True:

            order = self.orders\_queue.get()

            if order is None:

                break

            with self.kitchen\_lock:

                print(f"Chef {chef\_id}: Processing order - {order}")

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

                self.orders\_in\_kitchen.append(order)

                print(f"Chef {chef\_id}: Order ready - {order}")

    def serve\_order(self, waiter\_id):

        while True:

            with self.kitchen\_lock:

                if not self.orders\_in\_kitchen:

                    continue

                order = self.orders\_in\_kitchen.pop(0)

                print(f"Waiter {waiter\_id}: Serving order - {order}")

                time.sleep(random.uniform(1, 2))

restaurant = Restaurant()

num\_chefs = 3

chefs = [threading.Thread(target=restaurant.process\_order, args=(i,)) for i in range(num\_chefs)]

num\_waiters = 2

waiters = [threading.Thread(target=restaurant.serve\_order, args=(i,)) for i in range(num\_waiters)]

for chef in chefs:

    chef.start()

for waiter in waiters:

    waiter.start()

for i in range(5):

    order = f"Order {i}"

    restaurant.place\_order(order)

    time.sleep(random.uniform(0.5, 1.5))

for \_ in range(num\_chefs):

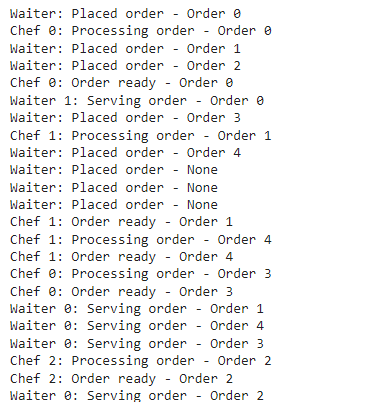
    restaurant.place\_order(None)

for chef in chefs:

    chef.join()

for waiter in waiters:

    waiter.join()



**TASK # 3: Locks and Synchronization in a Banking System**

import threading

import time

class Bank:

    def \_\_init\_\_(self):

        self.accounts = {'Ahmed': 1000, 'Ali': 1000}

    def withdraw(self, account, amount):

        current\_balance = self.accounts[account]

        time.sleep(0.1)

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

    def deposit(self, account, amount):

        current\_balance = self.accounts[account]

        time.sleep(0.1)

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

def perform\_transaction(bank, account, amount, transaction\_type):

    if transaction\_type == 'withdraw':

        bank.withdraw(account, amount)

    elif transaction\_type == 'deposit':

        bank.deposit(account, amount)

def simulate\_transactions(bank, account, num\_transactions, transaction\_type):

    for \_ in range(num\_transactions):

        perform\_transaction(bank, account, 100, transaction\_type)bank = Bank()

alice\_withdraw\_thread = threading.Thread(target=simulate\_transactions, args=(bank, 'Alice', 5, 'withdraw'))

bob\_deposit\_thread = threading.Thread(target=simulate\_transactions, args=(bank, 'Bob', 5, 'deposit'))

alice\_withdraw\_thread.start()

bob\_deposit\_thread.start()

alice\_withdraw\_thread.join()

bob\_deposit\_thread.join()

print("Final Balances:", bank.accounts)



Now, let's modify the program to use locks to ensure safe access to the shared data

import threading

import time

class Bank:

    def \_\_init\_\_(self):

        self.accounts = {'Ahmed': 1000, 'Ali': 1000}

        self.lock = threading.Lock()

    def withdraw(self, account, amount):

        with self.lock:

            current\_balance = self.accounts[account]

            time.sleep(0.1)

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

    def deposit(self, account, amount):

        with self.lock:

            current\_balance = self.accounts[account]

            time.sleep(0.1)

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

def perform\_transaction(bank, account, amount, transaction\_type):

    if transaction\_type == 'withdraw':

        bank.withdraw(account, amount)

    elif transaction\_type == 'deposit':

        bank.deposit(account, amount)

def simulate\_transactions(bank, account, num\_transactions, transaction\_type):

    for \_ in range(num\_transactions):

        perform\_transaction(bank, account, 100, transaction\_type)

bank = Bank()

alice\_withdraw\_thread = threading.Thread(target=simulate\_transactions, args=(bank, 'Ahmed', 5, 'withdraw'))

bob\_deposit\_thread = threading.Thread(target=simulate\_transactions, args=(bank, 'Ali', 5, 'deposit'))

alice\_withdraw\_thread.start()

bob\_deposit\_thread.start()

alice\_withdraw\_thread.join()

bob\_deposit\_thread.join()

print("Final Balances:", bank.accounts)

