**Task No. 1:** Queues and Message Passing

Implement a simple messaging system using queues for communication between threads.

Create a Python program that simulates a messaging system.

Implement two threads, one acting as a sender and the other as a receiver.

Use a queue to pass messages from the sender to the receiver.

Ensure proper synchronization to handle multiple messages correctly.

Display the received messages in the console.

**Solution:**

import threading

import queue

import time

message\_queue = queue.Queue()

# Function to simulate the sender thread

def sender\_thread():

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

for message in messages\_to\_send:

# Put the message in the queue

message\_queue.put(message)

print(f"Sent: {message} \n")

time.sleep(1)

# Function to simulate the receiver thread

def receiver\_thread():

while True:

# Get the message from the queue

message = message\_queue.get()

print(f"Received: {message}\n")

# Create and start the sender and receiver threads

sender = threading.Thread(target=sender\_thread)

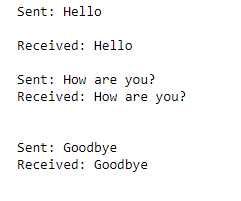
receiver = threading.Thread(target=receiver\_thread)

sender.start()

receiver.start()

sender.join()

receiver.join()

**Output:**

**Task No. 2:** Implementation of Queues and Locks

Integrate the concepts of queues and locks in a more complex scenario.

Design a program that models a restaurant with multiple chefs and waiters.

Use queues to represent orders placed by customers and messages sent between chefs and waiters.

Implement locks to synchronize access to shared resources such as the kitchen or a list of orders.

Simulate the flow of orders, preparation by chefs, and delivery by waiters.

Ensure that the program runs smoothly in a multithreaded environment.

**Solution:**

import threading

import queue

import time

order\_queue = queue.Queue()

message\_queue = queue.Queue()

# Create a lock for synchronization

lock = threading.Lock()

# Function to simulate the customer placing an order

def place\_order(customer, item):

order\_queue.put((customer, item))

print(f"{customer} placed an order for {item}")

# Function to simulate the chef preparing an order

def prepare\_order(chef):

while True:

with lock:

if not order\_queue.empty():

customer, item = order\_queue.get()

print(f"{chef} is preparing {item} for {customer}")

time.sleep(2) # Simulate cooking time

message\_queue.put((customer, f"{item} is ready"))

else:

break

# Function to simulate the waiter delivering an order

def deliver\_order(waiter):

while True:

with lock:

if not message\_queue.empty():

customer, message = message\_queue.get()

print(f"{waiter} delivered {message} to {customer}")

else:

break

# Create and start chef threads

chef1 = threading.Thread(target=prepare\_order, args=("Chef 1",))

chef2 = threading.Thread(target=prepare\_order, args=("Chef 2",))

chef1.start()

chef2.start()

# Create and start waiter threads

waiter1 = threading.Thread(target=deliver\_order, args=("Waiter 1",))

waiter2 = threading.Thread(target=deliver\_order, args=("Waiter 2",))

waiter1.start()

waiter2.start()

place\_order("Customer 1", "Burger")

place\_order("Customer 2", "Pizza")

place\_order("Customer 3", "Pasta")

chef1.join()

chef2.join()

waiter1.join()

waiter2.join()

A white text with black text

Description automatically generated**Output:**

**Task No. 3:** Locks and Synchronization in a Banking System

Develop a Python program simulating a banking system with multiple customer accounts (represented as balances).

Implement multiple threads to perform transactions such as deposits and withdrawals on these accounts.

Without using locks, intentionally create a scenario where race conditions or data corruption can occur during concurrent transactions.

Run the program and observe the unexpected behavior resulting from the lack of synchronization.

Modify the program to use locks to ensure that only one thread can access an account for a transaction at a time.

Run the modified program and verify that the accounts are accessed safely without data corruption, ensuring the integrity of each transaction.

Output the final balances of the customer accounts to confirm that synchronization has been achieved.

**Solution:**

import threading

accounts = {

    "Alice": 100,

    "Bob": 200,

    "Charlie": 300

}

lock = threading.Lock()

class TransactionThread(threading.Thread):

    def \_\_init\_\_(self, name, account, amount, operation):

        super().\_\_init\_\_()

        self.name = name

        self.account = account

        self.amount = amount

        self.operation = operation

    def run(self):

        global accounts

        with lock:  # Acquire lock before accessing account

            if self.operation == "deposit":

                accounts[self.account] += self.amount

                print(f"{self.name} deposited {self.amount} to {self.account}")

            elif self.operation == "withdraw":

                if accounts[self.account] >= self.amount:

                    accounts[self.account] -= self.amount

                    print(f"{self.name} withdrew {self.amount} from {self.account}")

                else:

                    print(f"Insufficient funds in {self.account} for {self.name} to withdraw {self.amount}")

threads = []

threads.append(TransactionThread("Alice", "Bob", 50, "deposit"))

threads.append(TransactionThread("Bob", "Alice", 30, "withdraw"))

threads.append(TransactionThread("Charlie", "Alice", 20, "withdraw"))

for thread in threads:

    thread.start()

for thread in threads:

    thread.join()

print(f"Final account balances: {accounts}")

**Output:**

A white background with black text

Description automatically generated