LAB NO 10

Bahria University,

Karachi Campus

LAB EXPERIMENT NO.
09

LIST OF TASKS

TASK NO	OBJECTIVE
1	Basic Concurrent Account Operations In this task, students are required to implement a simple banking system that supports concurrent account operations. They should create a BankAccount class with methods for deposit, withdrawal, and balance inquiry, ensuring the proper synchronization to handle concurrent transactions.
2	Ensuring Thread Safety in a Messaging System In this task, students are tasked with designing and implementing a simple messaging system that ensures thread safety. The system should allow multiple threads to send and receive messages concurrently without the risk of data corruption or race conditions. Students should create a Message Queue class with methods for sending and receiving messages, and they must incorporate synchronization mechanisms such as locks or semaphores to guarantee thread safety.
3	GUI-Based Bidirectional Chat System with Socket Programming The task involves creating a graphical user interface (GUI) for a bidirectional chat system using socket programming. Students are required to design an intuitive GUI with features like message input fields, chat logs, and message display areas. Through socket programming, they need to implement the server to handle multiple clients, manage connections, and facilitate real-time communication. The bidirectional nature of the system should allow users to send and receive messages, complete with timestamps and sender identification. Additionally, students should ensure thread safety to manage concurrent connections and implement robust error handling mechanisms.

Submitted On:
05/12/2023

LAB NO 10

(Date: DD/MM/YY)

Task No 1: Basic Concurrent Account Operations

Solution:

```
import threading
import time
import random
class BankAccount:
    def __init__(self, account_id, initial_balance=0):
        self.account_id = account_id
        self.balance = initial_balance
        self.lock = threading.Lock()
    def deposit(self, amount):
        with self.lock:
            current_balance = self.balance
            new_balance = current_balance + amount
            time.sleep(1)
            self.balance = new_balance
            print(f"Deposited ${amount} to Account {self.account_id}. New balance: 
${new_balance}")
    def withdraw(self, amount):
        with self.lock:
            current_balance = self.balance
            if current_balance >= amount:
                new_balance = current_balance - amount
                time.sleep(1)
                self.balance = new_balance
                print(f"Withdrew ${amount} from Account {self.account_id}. New balance: 
${new_balance}")
            else:
                print(f"Insufficient funds in Account {self.account_id} to withdraw 
${amount}")
    def check_balance(self):
        with self.lock:
            print(f"Balance in Account {self.account_id}: ${self.balance}")
def perform_operations(account, num_operations):
    for _ in range(num_operations):
        operation = random.choice(["deposit", "withdraw"])
        amount = random.randint(1, 100)
        if operation == "deposit":
            account.deposit(amount)
        else:
            account.withdraw(amount)
        time.sleep(0.5)
if __name__ == "__main__":
    accounts = [BankAccount(account_id) for account_id in range(1, 4)]
    threads = []
    for account in accounts:
        thread = threading.Thread(target=perform_operations, args=(account, 2))
        threads.append(thread)
        thread.start()
    for thread in threads:
        thread.join()
    for account in accounts:
```

```
        account.check_balance()
```

Output:

```
PS C:\Users\ah030\OneDrive\Desktop> & C:/Users/ah030/AppData/
Insufficient funds in Account 1 to withdraw $65
Insufficient funds in Account 2 to withdraw $24
Insufficient funds in Account 2 to withdraw $45
Deposited $81 to Account 3. New balance: $81
Deposited $73 to Account 1. New balance: $73
Deposited $97 to Account 3. New balance: $178
Balance in Account 1: $73
Balance in Account 2: $0
Balance in Account 3: $178
```

Task No 2: Ensuring Thread Safety in a Messaging System

In this task, students are tasked with designing and implementing a simple messaging system that ensures thread safety. The system should allow multiple threads to send and receive messages concurrently without the risk of data corruption or race conditions. Students should create a Message Queue class with methods for sending and receiving messages, and they must incorporate synchronization mechanisms such as locks or semaphores to guarantee thread safety. The goal is to demonstrate the ability to handle concurrent operations on shared resources securely. Evaluation will be based on the correctness of the messaging system, the effectiveness of thread safety measures, and the demonstration of proper synchronization in a multi-threaded environment.

Solution:

```
import threading
import time
from queue import Queue
class MessageQueue:
    def __init__(self):
        self.messages = Queue()
        self.lock = threading.Lock()
    def send_message(self, sender, content):
        with self.lock:
            message = f"{sender}: {content}"
            self.messages.put(message)
            print(f"Message sent by {sender}: {content} :)")
    def receive_message(self, receiver):
        with self.lock:
            if not self.messages.empty():
                message = self.messages.get()
                print(f"Message received by {receiver}: {message} :)")
            else:
                print(f"No messages for {receiver}")
def user_function(user, message_queue):
    for _ in range(3):
        time.sleep(1)
        message_queue.send_message(user, f"Hi, How are you doing {_ + 1}")
    for _ in range(3):
        time.sleep(1)
```

```
        message_queue.receive_message(user)
if __name__ == "__main__":
    message_queue = MessageQueue()
    user1_thread = threading.Thread(target=user_function, args=("User1", 
message_queue))
    user2_thread = threading.Thread(target=user_function, args=("User2", 
message_queue))
    user1_thread.start()
    user2_thread.start()
    user1_thread.join()
    user2_thread.join()
```

Output:

```
PS C:\Users\ah030\OneDrive\Desktop> & C:/Users/ah030/AppData/Local/Programs/Python/Python311/
Message sent by User1: Hi, How are you doing 1 :)
Message sent by User2: Hi, How are you doing 1 :)
Message sent by User1: Hi, How are you doing 2 :)
Message sent by User2: Hi, How are you doing 2 :)
Message sent by User1: Hi, How are you doing 3 :)
Message sent by User2: Hi, How are you doing 3 :)
Message received by User1: User1: Hi, How are you doing 1 :)
Message received by User2: User2: Hi, How are you doing 1 :)
Message received by User1: User1: Hi, How are you doing 2 :)
Message received by User2: User2: Hi, How are you doing 2 :)
Message received by User1: User1: Hi, How are you doing 3 :)
Message received by User2: User2: Hi, How are you doing 3 :)
```

Task No 3: GUI-Based Bidirectional Chat System with Socket Programming

The task involves creating a graphical user interface (GUI) for a bidirectional chat system using socket programming. Students are required to design an intuitive GUI with features like message input fields, chat logs, and message display areas. Through socket programming, they need to implement the server to handle multiple clients, manage connections, and facilitate real-time communication. The bidirectional nature of the system should allow users to send and receive messages, complete with timestamps and sender identification. Additionally, students should ensure thread safety to manage concurrent connections and implement robust error handling mechanisms. The final evaluation will consider the completeness and correctness of the GUI, the effectiveness of bidirectional communication, the implementation of thread safety measures, and the overall reliability of the chat system. Additional features, such as file sharing or encryption, can be explored for extra credit.

Solution:

```
import tkinter as tk
from tkinter import scrolledtext
import socket
import threading
from datetime import datetime
class ChatApp:
def __init__(self, master):
self.master = master
self.master.title("Chat Application")
self.message_area = 
scrolledtext.ScrolledText(self.master, 
wrap=tk.WORD, width=50, height=20)
self.message_area.tag_configure("server", 
foreground="blue")
```

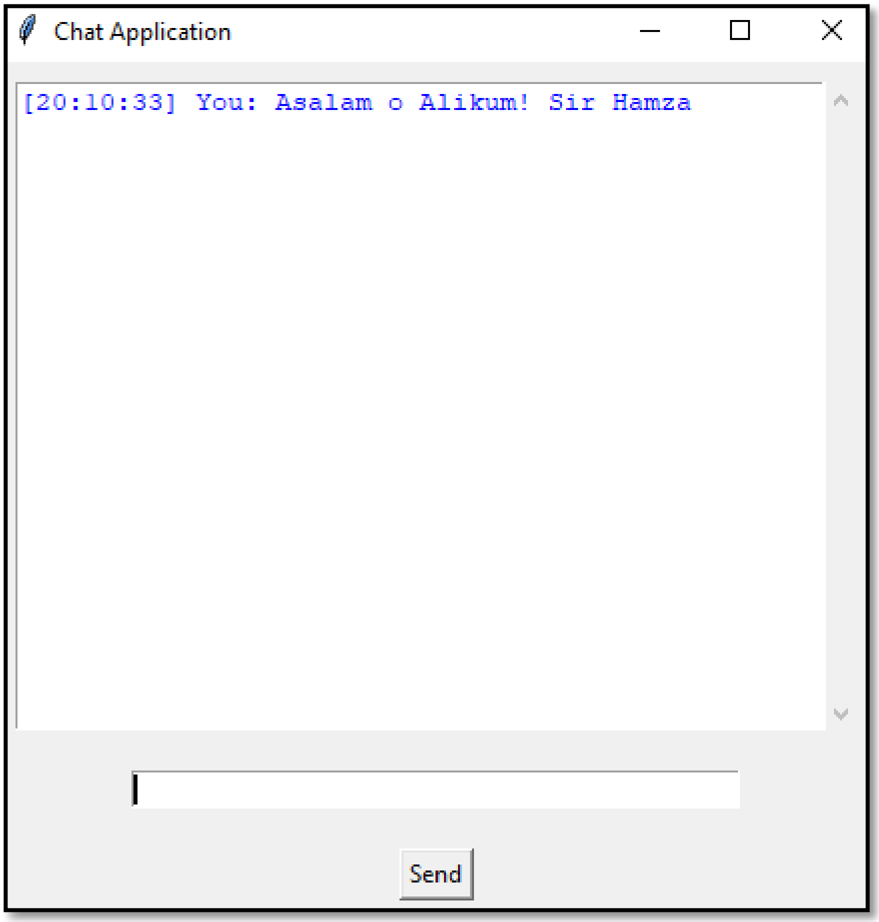
```
args=(client_socket,))
client_thread.start()
def handle_client(self, 
client_socket):
while True:
try:
message = 
client_socket.recv(1024).decode('utf-
8')
if message:
timestamp = 
datetime.now().strftime("%H:%M:%S")
formatted_message = f"[{timestamp}] 
Client: {message}"
```

LAB NO 10

self.message_area.tag_configure("client", foreground="green")
self.message_area.pack(padx=10, pady=10)
self.entry_var = tk.StringVar()
self.message_entry = tk.Entry(self.master, textvariable=self.entry_var, width=50)
self.message_entry.pack(padx=10, pady=10)
self.send_button = tk.Button(self.master, text="Send", command=self.send_message)
self.send_button.pack(pady=10)
self.server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
self.server.bind(('localhost', 5555))
self.server.listen(5)
self.clients = []
self.accept_thread = threading.Thread(target=self.accept_connections)
self.accept_thread.start()
def accept_connections(self):
while True:
client_socket, client_address = self.server.accept()
self.clients.append(client_socket)
client_thread = threading.Thread(target=self.handle_client,
	self.message_area.insert(tk.END, formatted_message + "\n", "client")
	except (socket.error, ConnectionResetError):
	self.clients.remove(client_socket)
	break
	def send_message(self):
	message = self.entry_var.get()
	if message:
	timestamp = datetime.now().strftime("%H:%M:%S")
	formatted_message = f"[{timestamp}] You: {message}"
	self.message_area.insert(tk.END, formatted_message + "\n", "server")
	
	for client_socket in self.clients:
	try:
	client_socket.send(message.encode('utf-8'))
	except socket.error:
	pass
	self.entry_var.set("")
	def main():
	root = tk.Tk()
	app = ChatApp(root)
	root.mainloop()
	if __name__ == "__main__":
	main()

Output:

LAB NO 10



LAB NO 10

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:

05/12/2023

(Date: DD/MM/YY)

*****LAB TASKS*****

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:

```
Received message: Hello
Received message: How are you?
Received message: Goodbye
```

QUESTION NO: 02 **Task#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:

```
Customer placed an order: Order 1
Chef is preparing order: Order 1
Customer placed an order: Order 2
Customer placed an order: Order 3
Chef completed order: Order 1
Chef is preparing order: Order 2
Customer placed an order: Order 4
Waiter is serving order: Order 1
Chef completed order: Order 2
Chef is preparing order: Order 3
Customer placed an order: Order 5
Customer placed an order: exit
```

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:
        thread.join()
```

OUTPUT

```
Withdrew 77 from Account 1. New balance: 1586
Withdrew 83 from Account 2. New balance: 1867
Withdrew 35 from Account 1. New balance: 1551
Deposited 69 into Account 0. New balance: 780
Withdrew 70 from Account 1. New balance: 1481
Deposited 70 into Account 0. New balance: 850
Withdrew 8 from Account 1. New balance: 1473
Deposited 69 into Account 0. New balance: 919
Withdrew 75 from Account 2. New balance: 1792
Deposited 51 into Account 2. New balance: 1843
Deposited 63 into Account 0. New balance: 982
Withdrew 89 from Account 2. New balance: 1754
Withdrew 22 from Account 0. New balance: 960
Deposited 78 into Account 0. New balance: 1038
Withdrew 71 from Account 1. New balance: 1402
Deposited 73 into Account 1. New balance: 1475
Withdrew 27 from Account 0. New balance: 1011
Final balances after transactions with locks: {0: 1011, 1: 1475, 2: 1754}
```