Name:Testing

SECTION 1: THEORY QUESTIONS

Q1: Discuss the importance of Software Architecture and the differences between Tiered and Microservices architectures.

Answer: Software architecture serves as the blueprint for a software system, providing an abstraction to manage the system's complexity and establish a communication coordination mechanism among components. It is the high-level structure of a software system and the discipline of creating such structures and systems. Each structure comprises software elements, relations among them, and properties of both elements and relations. The architecture of a software system is a metaphor, analogous to the architecture of a building. It functions as a critical design tool for systems that are increasingly large and complex. In a Tiered architecture, typically a three-tier model, the application is organized into the presentation tier (UI), the logic tier (application processing), and the data tier (database). This provides a clear separation of concerns and allows for easier maintenance of specific layers. However, tiered systems often suffer from being monolithic, where a failure in one layer can affect the entire system.

In contrast, Microservices architecture structures an application as a collection of loosely coupled services. In a microservices architecture, services are fine-grained and the protocols are lightweight. The benefit of decomposing an application into different smaller services is that it improves modularity. This makes the application easier to understand, develop, test, and become more resilient to architecture erosion. It parallelizes development by enabling small autonomous teams to develop, deploy, and scale their respective services independently. It also allows the architecture of an individual service to emerge through continuous introspection. However, it introduces complexity in terms of network latency, data consistency, and distributed logging. Choosing the right architecture is paramount because it dictates the scalability, maintainability, and deployment velocity of the software throughout its entire lifecycle.

Q2: Analyze the concept of Deadlocks in Operating Systems and the four necessary conditions for a deadlock to occur.

Answer: A deadlock is a situation in which two or more processes are unable to proceed because each is waiting for the other to release a resource. In an operating system, deadlocks occur when processes are granted exclusive access to devices, files, and other resources. The problem of deadlocks is a classic one in concurrent programming and operating systems. For a deadlock to occur, four conditions, known as the Coffman conditions, must hold simultaneously within the system. The first condition is Mutual Exclusion, which states that at least one resource must be held in a non-sharable mode; only one process can use the resource at any given time. The second condition is Hold and Wait, where a process is currently holding at least one resource and is requesting additional resources that are being held by other processes.

The third condition is No Preemption, which means that resources cannot be forcibly taken from a process; they must be released voluntarily by the process holding them after that process has completed its task. The fourth and final condition is Circular Wait, where a group of processes exists such that each process is waiting for a resource held by the next process in the chain, with the last process waiting for a resource held by the first. If any one of these conditions is not met, a deadlock cannot occur. Operating systems use various strategies to handle deadlocks, including deadlock prevention by ensuring at least one condition never holds, deadlock avoidance by using algorithms like the Banker's Algorithm to check the system state before granting resource requests, and deadlock detection and recovery, where the system allows a deadlock to occur but has mechanisms to identify and break it.

Q3: Describe the features of Cloud Computing and the shared responsibility model between providers and customers.

Answer: Data integrity refers to the accuracy, consistency, and reliability of data stored in a database. It is a fundamental aspect of database design and management, ensuring that the information remains correct and trustworthy throughout its lifecycle. Maintaining data integrity is crucial because decisions made based on incorrect or inconsistent data can lead to significant business failures or system errors. In a Database Management System (DBMS), integrity is enforced through several types of constraints. Entity Integrity ensures that each table has a primary key and that the primary key is unique and not null, allowing every row to be uniquely identified. Referential Integrity ensures that relationships between tables remain consistent; specifically, a foreign key value must either match a primary key value in a related table or be null.

Domain Integrity ensures that all data items in a column fall within a defined set of valid values, which is managed through data types, formats, and check constraints. Beyond these constraints, DBMS use Transaction Management to ensure integrity. The ACID properties (Atomicity, Consistency, Isolation, Durability) are essential here. Atomicity ensures that all operations in a transaction are completed; if not, the transaction is aborted. Consistency ensures that a transaction brings the database from one valid state to another. Isolation ensures that concurrent transactions do not interfere with each other. Durability ensures that once a transaction is committed, it remains so even in the event of a system failure. By combining these constraints and transactional controls, the DBMS provides a robust framework for preventing data corruption and ensuring that the data reflects the real-world entities it represents

Q4: Explain the importance of Data Integrity and the mechanisms used to ensure it in a DBMS.

Answer: Data integrity refers to the accuracy, consistency, and reliability of data stored in a database. It is a fundamental aspect of database design and management, ensuring that the information remains correct and trustworthy throughout its lifecycle. Maintaining data integrity is crucial because decisions made based on incorrect or inconsistent data can lead to significant business failures or system errors. In a Database Management System (DBMS), integrity is enforced through several types of constraints. Entity Integrity ensures that each table has a primary key and that the primary key is unique and not null, allowing every row to be uniquely identified. Referential Integrity ensures that relationships between tables remain consistent; specifically, a foreign key value must either match a primary key value in a related table or be null.

Domain Integrity ensures that all data items in a column fall within a defined set of valid values, which is managed through data types, formats, and check constraints. Beyond these constraints, DBMS use Transaction Management to ensure integrity. The ACID properties (Atomicity, Consistency, Isolation, Durability) are essential here. Atomicity ensures that all operations in a transaction are completed; if not, the transaction is aborted. Consistency ensures that a transaction brings the database from one valid state to another. Isolation ensures that concurrent transactions do not interfere with each other. Durability ensures that once a transaction is committed, it remains so even in the event of a system failure. By combining these constraints and transactional controls, the DBMS provides a robust framework for preventing data corruption and ensuring that the data reflects the real-world entities it represents.

Q5: Compare the TCP and UDP protocols, discussing their use cases and delivery mechanisms.

Answer: Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) are the two primary protocols used for data transmission at the Transport Layer of the OSI model. TCP is a connection-oriented protocol, meaning that before data can be sent, a formal connection must be established between the sender and the receiver through a process known as the three-way handshake. TCP provides reliable delivery by using sequence numbers, acknowledgments, and retransmission of lost packets. It also features flow control and congestion control to ensure that the sender does not overwhelm the receiver or the network. This makes TCP ideal for applications where data accuracy is paramount, such as web browsing (HTTP/HTTPS), email (SMTP), and file transfers (FTP). However, the overhead of establishing a connection and ensuring reliability makes TCP slower than UDP.

UDP, on the other hand, is a connectionless protocol. It sends data packets, called datagrams, directly to the destination without establishing a connection first. UDP does not provide any guarantee of delivery, nor does it order the packets or check for duplicates. This lack of overhead makes UDP significantly faster and more efficient for real-time applications where speed is more important than perfect accuracy. Common use cases for UDP include online gaming, video streaming, and Voice over IP (VoIP), where a few lost packets are preferable to the delays caused by retransmission. While TCP provides a "reliable stream" of data, UDP provides a "best-effort" delivery service. Developers must choose between these protocols based on the specific needs of their application, balancing the trade-offs between reliability and performance.

SECTION 2: CODING / LOGIC QUESTIONS

Q6: Implement a Python script to demonstrate multi-level inheritance in Object-Oriented Programming.

Answer:

Python

```
class Vehicle:
    def __init__(self, brand, year):
        self.brand = brand
        self.year = year
        self.is_running = False

    def start_engine(self):
        self.is_running = True
        return f"The {self.brand} engine is now running."

class Car(Vehicle):
    def __init__(self, brand, year, doors):
        # Call the constructor of the parent class
        super().__init__(brand, year)
        self.doors = doors

    def drive(self):
        if self.is_running:
            return f"The {self.brand} car is driving with {self.doors} doors."
        return "Start the engine first!"

class ElectricCar(Car):
    def __init__(self, brand, year, doors, battery_capacity):
        # Call the constructor of the Car class
        super().__init__(brand, year, doors)
        self.battery_capacity = battery_capacity

    def check_battery(self):
        # Logic to simulate battery check
        return f"Battery is at {self.battery_capacity}kWh capacity."

# Simulation of Multi-level Inheritance
my_ev = ElectricCar("Tesla", 2024, 4, 100)
print(my_ev.start_engine())
print(my_ev.drive())
print(my_ev.check_battery())
```

Q7: Write a Python program to implement a Bubble Sort algorithm with a flag for optimization.

Answer:

Python

```
def optimized_bubble_sort(data_list):
    n = len(data_list)

    # Outer loop to traverse through all elements
    for i in range(n):
        # Flag to detect if any swapping happened in the inner loop
        swapped = False

        # Last i elements are already in place, so skip them
        for j in range(0, n - i - 1):
            # Compare adjacent elements
            if data_list[j] > data_list[j + 1]:
                # Swap if the element found is greater than the next
                data_list[j], data_list[j + 1] = data_list[j + 1], data_list[j]
                swapped = True

        # Edge case: If no two elements were swapped by inner loop, then break
        if not swapped:
            print(f"List optimized and sorted at iteration {i+1}")
            break

    return data_list

# Testing the logic with an unsorted and nearly sorted list
sample_data = [64, 34, 25, 12, 22, 11, 90]
print("Original List:", sample_data)
sorted_result = optimized_bubble_sort(sample_data)
print("Sorted List:", sorted_result)

nearly_sorted = [1, 2, 3, 5, 4]
print("Nearly Sorted:", optimized_bubble_sort(nearly_sorted))
```

Q8: Create a Python script that reads a text file and counts the frequency of each word using a dictionary.

Answer:

Python

```
import string


def count_word_frequency(filename):
  word_count = {}


  try:
    # Open file in read mode
    with open(filename, 'r') as file:
      for line in file:
        # Remove punctuation and convert to lowercase
        clean_line = line.translate(str.maketrans('', '', string.punctuation))
        words = clean_line.lower().split()


        # Update word counts in the dictionary
        for word in words:
          if word in word_count:
            word_count[word] += 1
          else:
            word_count[word] = 1


    # Sort dictionary by frequency (descending)
    sorted_counts = dict(sorted(word_count.items(), key=lambda item: item[1],
reverse=True))
    return sorted_counts


  except FileNotFoundError:
        return "Error: The specified file was not found."
    except Exception as e:
        return f"An unexpected error occurred: {e}"

# Simulation Logic
# Create a temporary file for the test
with open("test_content.txt", "w") as f:
    f.write("Python is great. Python is fast. Fast is good!")

results = count_word_frequency("test_content.txt")
print(results)
```

Q9: Implement a Python function that uses Recursion to calculate the Fibonacci sequence up to n terms.

Answer:

Python

```
def fibonacci_recursive(n):
    # Base cases
    if n <= 0:
        return 0
    elif n == 1:
        return 1
    else:
        # Recursive step
        return fibonacci_recursive(n - 1) + fibonacci_recursive(n - 2)

def generate_fib_series(count):
    series = []
    # Loop to collect n terms
    for i in range(count):
        # Edge case: Ensure count is positive
        if count < 0:
            return "Invalid count"
        series.append(fibonacci_recursive(i))
    return series

# Execute and print result
num_terms = 10
print(f"Fibonacci series for {num_terms} terms:")
print(generate_fib_series(num_terms))

# Comment: Recursion is elegant but note that for large n,
# this approach is inefficient without memoization.
```

Q10: Write a Python program to validate a user's password based on multiple security criteria.

Answer:

Python

```
def validate_password(password):
    # Check for minimum length
    if len(password) < 8:
        return "Fail: Password must be at least 8 characters long."

    has_upper = False
    has_lower = False
    has_digit = False
    has_special = False
    special_chars = "!@#$%^&*"
    # Loop through each character to check conditions

    for char in password:

        if char.isupper():

            has_upper = True

        elif char.islower():

            has_lower = True

        elif char.isdigit():

            has_digit = True

        elif char in special_chars:

            has_special = True


    # Logical verification of all conditions

    if not has_upper:

        return "Fail: Missing uppercase letter."

    if not has_lower:

        return "Fail: Missing lowercase letter."

    if not has_digit:

        return "Fail: Missing numerical digit."

    if not has_special:

        return "Fail: Missing special character (!@#$%^&*)."


    return "Success: Password is secure and valid."


# Testing the validation logic

test_passwords = ["12345", "password", "Password123", "Secure@123"]

for pwd in test_passwords:

    print(f"Testing '{pwd}': {validate_password(pwd)}")
```

SECTION 3: PROGRAMMING LANGUAGE QUESTIONS

Q11: What is the purpose of the __init__ method in Python classes?

Answer: The __init__ method is a special method in Python classes, known as a constructor in other object-oriented languages. It is automatically called when a new instance (object) of a class is created. Its primary purpose is to initialize the object's attributes with default or user-provided values. It allows the class to set up the initial state of the object, ensuring that it is ready for use immediately after instantiation.

Q12: Explain the concept of List Comprehension in Python with an example.

Answer: List comprehension is a concise and elegant way to create lists in Python. It provides a shorter syntax when you want to create a new list based on the values of an existing list or iterable. Instead of using a multi-line for loop and the .append() method, you can perform the operation in a single line. For example, [x**2 for x in range(5)] creates a list of squares: [0, 1, 4, 9, 16].

Q13: What are Python "Args" and "Kwargs" and when should they be used?

Answer: *args and **kwargs are special keyword patterns that allow a function to accept a variable number of arguments. *args is used to send a non-keyworded, variable-length argument list to the function (passed as a tuple). **kwargs is used to pass a keyworded, variable-length argument list (passed as a dictionary). They are useful when you do not know beforehand how many arguments might be passed to your function.

Q14: Describe the difference between 'Shallow Copy' and 'Deep Copy' in Python.

Answer: A shallow copy creates a new object but inserts references into it to the objects found in the original. If the original object contains nested objects, the shallow copy will still point to the same nested objects. A deep copy, created using the copy.deepcopy() function, creates a new object and recursively adds copies of the objects found in the original, ensuring that the new object is entirely independent of the original.

Q15: What is the Global Interpreter Lock (GIL) and how does it affect multi-threading?

Answer: The Global Interpreter Lock (GIL) is a mechanism used in CPython (the standard Python implementation) that ensures only one thread executes Python bytecode at a time, even on multi-core processors. This is done to simplify memory management and prevent race conditions in the interpreter's internal state. However, it means that CPU-bound multi-threaded programs do not see a performance gain on multi-core systems. For CPU-bound tasks, multiprocessing is preferred over multi-threading.