Experiment#		Student ID	
Date		Student Name	

16. Thread Synchronization and Coordination of Thread.

Aim/Objective: To understand the Thread Synchronization and Producer-Consumer thread coordination in a multi-threaded Java program.

Description: The student will understand the concepts of the Producer-Consumer pattern, which is used to solve the problem of synchronizing access to a shared resource between multiple threads.

Pre-Requisites: Classes, Objects, Understanding of multi-threading and synchronization in Java

Tools: Eclipse IDE for Enterprise Java and Web Developers

Pre-Lab:

1) Explain the need of Synchronization in a multithreading environment?

Need for Synchronization in Multithreading:

In a multithreading environment, multiple threads can access shared resources simultaneously. This can lead to data inconsistency or race conditions if threads interfere with each other's operations on shared data.

Synchronization ensures that

1. Data Consistency: Only one thread can access a shared resource at a time, preventing conflicts and maintaining consistency.

2. Prevents Race Conditions: Synchronization controls access to critical sections of code where shared resources are modified, avoiding unpredictable outcomes

3. Thread Safety: It ensures that shared resources are used in a thread-safe manner, preserving correct results even when multiple threads are involved.

Without synchronization, threads might corrupt shared data or cause unexpected behaviors in a program.

2) How threads establish communication mechanism among them in a multithreading environment?

Threads in a multithreading environment can communicate through:

1. Wait/Notify: Threads use wait(), notify(), and notifyAll() to pause and resume execution, co-ordinating tasks.

2. Shared Objects: Threads share variables or objects, and synchronization ensures consistent access to them.

3. Thread Join: One thread waits for another to finish using join().

4. Flags/Variables: Shared variables can signal task completion between threads.

5. Locks/Semaphores: These control thread access to shared resources, ensuring safe co-ordination.

These mechanisms help threads collaborate effectively and avoid conflicts.

In-Lab:

1) You are tasked with designing a Bank Account Management System where multiple users can perform transactions on a shared bank account simultaneously. To ensure the consistency and correctness of the account balance, synchronization is necessary.

Requirements

A. **BankAccount Class**:

a. Attributes: balance (double).

b. Methods:

deposit(double amount): Adds the specified amount to the account balance.

withdraw(double amount): Subtracts the specified amount from the account balance if sufficient funds are available.

getBalance(): Returns the current balance.

B. **Thread Safety**:

a. Use synchronization to ensure that deposit and withdrawal operations are thread-safe.

C. **Operations**:

a. Multiple threads will simulate users performing deposit and withdrawal operations concurrently.

Procedure/Program:

```
class BankAccount{
    private double balance;
    public BankAccount(double initialBalance){
        this.balance = initialBalance;
    }
    public synchronized void deposit(double amount){
        if(amount>0){
            balance += amount;
            system.out.println(Thread.currentThread().getName()+"deposited:"
            +amount+", Current Balance: "+balance);
        }
    }
```

```
public synchronized void withdraw(double amount) {
    if(amount >0 && balance >=amount) {
        balance -=amount;
        System.out.println(Thread.currentThread().getName()+" withdrew :"
         +amount +", Current Balance:"+ balance);
    }else {
        System.out.println(Thread.currentThread().getName() +"tried to withdraw:"
         +amount+" but insufficient balance.");
    }
}
public synchronized double getBalance() {
    return balance;
}
}
class TransactionTask implements Runnable {
    private Bankaccount account;
    public TransactionTask (BankAccount account) {
        this.account= account;
    }
    public void run() {
        account.deposit(100);
        account.withdraw(50);
    }
}
```

```
public class Main{
    public static void main(String[] args) {
        BankAccount account = new BankAccount(1000);
        Thread user1 = new Thread(new TransactionTask(account), "User1");
        Thread user2 = new Thread(new TransactionTask(account), "User2");
        Thread user3 = new Thread(new TransactionTask(account), "User3");
        user1.start();
        user2.start();
        user3.start();
    }
}
```

2) Implement a messaging application where a Producer class generates messages and a Consumer class consumes them. The communication between the producer and consumer will be synchronized to ensure proper message exchange without data loss or race conditions.

Requirements

A. **Producer Class**:

   a. Generates messages and puts them into a shared buffer.
   b. Uses synchronization to ensure thread safety.

B. **Consumer Class**:
   a. Consumes messages from the shared buffer.
   b. Uses synchronization to ensure thread safety

C. **Shared Buffer**:
   a. A thread-safe queue to store messages.

Procedure/Program:

```
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;
class Producer implements Runnable {
    Private BlockingQueue<String> sharedQueue;
    public Producer(BlockingQueue<String> sharedQueue) {
        this.sharedQueue = sharedQueue;
    }
    public void run() {
        for(int i=1; i<=5; i++) {
            try {
                String message = "Message "+i;
                sharedQueue.put(message);
                System.out.println("Produced: "+message);
                Thread.sleep(100);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
```

```
        }
    }
    try {
        sharedQueue.put("END");
    } catch(InterruptedException e) {
        Thread().currentThread().interrupt();
    }
}
}class Consumer implements Runnable{
    private BlockingQueue<String> sharedQueue;
    public Consumer(BlockingQueue<string> sharedQueue) {
        this.sharedQueue = sharedQueue;
    }
    public void run() {
        try {
            String message;
            While(!(message= sharedQueue.take()).equals("END")){
                System.out.println("Consumed: "+message);
                Thread.sleep(150);
            }
        } catch(InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }
}
```

```java
public class MessagingApp {
    public static void main(String[] args) {
        BlockingQueue<String> sharedQueue = new ArrayBlockingQueue<>(10);

        Producer producer = new Producer(sharedQueue);
        Consumer consumer = new Consumer(sharedQueue);

        Thread producerThread = new Thread(producer, "Producer");
        Thread consumerThread = new Thread(consumer, "Consumer");

        producerThread.start();
        consumerThread.start();
    }
}
```

✓ **Data and Results:**

Output:

Produced: Message1

Consumed: Message1

Produced: Message 2

Consumed: Message 2

Produced: Message 3

Consumed: Message 3

Produced: Message4

Consumed: Message4

Produced: Message5

Consumed: Message5 Produced: END

✓ **Analysis and Inferences:**

Synchronization: BlockingQueue ensures thread safety and prevents data loss by blocking the producer if the queue is full and the consumer if the queue is empty.

Coordination: Producer and consumer operate independently; messages are processed as they become available, allowing flexible consumption rates.

Efficiency: Using BlockingQueue simplifies thread coordination and avoids race conditions without manual locking.

Termination: The "END" message provides a clear, safe way to signal the consumer to stop once all messages are handled.

Experiment#		Student ID	
Date		Student Name	

VIVA-VOCE Questions (In-Lab):

1) What is the main challenge in implementing the producer-consumer problem?

The key challenge is managing synchronization between threads to avoid issues like data corruption, race conditions, or deadlocks. Producers and consumers need coordinated access to a shared buffer so that data is neither overwritten nor lost. The buffer must also manage capacity limits, ensuring producers pause if the buffer is full and consumers wait if it's empty.

2) How does the synchronized keyword ensure that only one thread can access a shared resource at a time?

The synchronized keyword locks an object or method, so only one thread can execute the synchronized code block or method at a time. When a thread enters a synchronized block, it acquires a lock on that object; other threads attempting to access the same synchronized block are blocked until the lock is released.

3) How does the wait() and notify() methods facilitate inter-thread communication in the Producer-Consumer pattern?

Wait() causes a thread to release its lock and wait until another thread invokes notify() or notifyAll() on the same object. In the producer-consumer pattern, a producer can wait() if the buffer is full, and a consumer can wait() if it's empty. The notify() method signals waiting threads to wake up and continue processing when conditions change, enabling smooth coordination between producers and consumers.

4) How can you ensure mutual exclusion between the producers and consumers while accessing the shared buffer?

Mutual exclusion is ensured by synchronizing access to the shared buffer, either by using synchronized blocks, methods, or a thread-safe data structure. This prevents producers and consumers from accessing the buffer concurrently, ensuring data consistency.

5) How does the Producer-Consumer pattern ensure thread safety and synchronization?

The pattern uses the thread-safe data structures or synchronized blocks to manage shared resources, preventing concurrent modification issues. Methods like wait() and notify() allow the producer and consumer to communicate effectively, coordinating buffer access and ensuring data integrity without busy-waiting or conflicts.

Post-Lab:

1) Write a code to implement a bounded buffer using the concepts learned in the experiment. Ensure that the buffer has a maximum capacity of 10 items, and the producer and consumer threads operate correctly while avoiding race conditions.

Procedure/Program:

```
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;
class Producer implements Runnable{
  private final BlockingQueue<Integer> buffer;
  public Producer(BlockingQueue<Integer> buffer) { this.buffer = buffer; }
  public void run() {
      try{
         for(int item=0;;item++) {
            buffer.put(item);
            System.out.println("Produced: "+ item);
            Thread.sleep(500);
        }
    } catch (InterruptedException e) { Thread.currentThread().interrupt(); }
  }
}
class Consumer implements Runnable {
  Private final BlockingQueue<Integer> buffer;
  public Consumer(BlockingQueue<Integer> buffer) { this.buffer = buffer; }
  public void run() {
     try {
```

```
        while(true) {
            int item = buffer.take();
            System.out.println("Consumed: "+item);
            Thread.sleep(1000);
        }
    } catch(InterruptedException e) {Thread.currentThread().interrupt();}
}
}public class BoundedBuffer{
    public static void main(String[] args) {
        BlockingQueue<Integer> buffer = new ArrayBlockingQueue<>(10);
        new Thread(new Producer(buffer)).start();
        new Thread(new Consumer(buffer)).start();
    }
}
```

Experiment#		Student ID	
Date		Student Name	

✓ **Data and Results:**

Buffer: A 10-item BlockingQueue.

Producer and Consumer: Producer generates items continuously, and the consumer retrieves them in the same order.

Output: Shows alternating "Produced" and "Consumed" messages, with each waiting if the buffer is full or empty.

✓ **Analysis and Inferences:**

Synchronization: BlockingQueue manages synchronization, preventing race conditions.

Order: FIFO order ensures items are consumed exactly as produced.

Efficiency: BlockingQueue simplifies code by handling waits, removing the need for manual locks.

Scalability: Supports easy buffer resizing and potential addition of more producers or consumers.

Evaluator Remark (if Any):	Marks Secured: _____out of 50
	Signature of the Evaluator with Date

Evaluator MUST ask Viva-voce prior to signing and posting marks for each experiment.