Experiment#		Student ID	
Date		Student Name	[@KLWKS_bot] THANOS

11. 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?

Synchronization ensures safe and predictable execution of multiple threads by:

- **Preventing Race Conditions**: Avoids conflicts when threads modify shared resources.
- **Ensuring Data Consistency**: Maintains correct data values across threads.
- **Providing Mutual Exclusion**: Ensures only one thread accesses critical sections at a time.
- **Avoiding Deadlocks**: Prevents indefinite waiting for resources.
- **Maintaining Execution Order**: Ensures tasks execute in the correct sequence.

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

Threads communicate using:

- **Shared Memory**: Accessing common variables with synchronization (mutex, semaphore).
- **Synchronization Constructs**: Mutex, semaphore, condition variables, locks.
- **Message Passing**: Threads exchange data via message queues.
- **Atomic Variables**: Ensures safe updates without locks.

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 = 0.0;

  public synchronized void deposit(double amount) {
    if (amount > 0) balance += amount;
  }

  public synchronized void withdraw(double amount) {
    if (amount > 0 && balance >= amount) balance -= amount;
  }

  public synchronized double getBalance() {
    return balance;
  }
}
```

```
class User extends Thread {
   private BankAccount account;
   private boolean isDepositing;
   private double amount;

   public User(BankAccount account, boolean isDepositing, double
amount) {
      this.account = account;
      this.isDepositing = isDepositing;
      this.amount = amount;
   }

   public void run() {
      if (isDepositing) account.deposit(amount);
      else account.withdraw(amount);
   }
}

public class BankAccountManagement {
   public static void main(String[] args) throws InterruptedException {
      BankAccount account = new BankAccount();
      User[] users = {
         new User(account, true, 100.0),
         new User(account, false, 50.0),
         new User(account, true, 200.0),
         new User(account, false, 150.0)
      };

      for (User user : users) user.start();
      for (User user : users) user.join();

      System.out.println("Final Balance: " + account.getBalance());
   }
}
```

OUTPUT

```
Final Balance: 100.0
```

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.*;

class SharedBuffer {
  private final Queue<String> queue = new LinkedList<>();
  private final int limit;

  public SharedBuffer(int limit) { this.limit = limit; }

  public synchronized void produce(String msg) throws InterruptedException {
    while (queue.size() == limit) wait();
    queue.add(msg);
    notifyAll();
  }

  public synchronized String consume() throws InterruptedException {
    while (queue.isEmpty()) wait();
    String msg = queue.poll();
    notifyAll();
    return msg;
  }
}
```

```
class Producer implements Runnable {
   private final SharedBuffer buffer;

   public Producer(SharedBuffer buffer) { this.buffer = buffer; }

   public void run() {
      try {
         for (int i = 0; i < 10; i++) {
            buffer.produce("Message " + i);
            System.out.println("Produced: Message " + i);
            Thread.sleep(100);
         }
      } catch (InterruptedException e) {}
   }
}

class Consumer implements Runnable {
   private final SharedBuffer buffer;

   public Consumer(SharedBuffer buffer) { this.buffer = buffer; }

   public void run() {
      try {
         for (int i = 0; i < 10; i++) {
            System.out.println("Consumed: " + buffer.consume());
            Thread.sleep(150);
         }
      } catch (InterruptedException e) {}
   }
}

public class MessagingApp {
   public static void main(String[] args) {
      SharedBuffer buffer = new SharedBuffer(5);
      new Thread(new Producer(buffer)).start();
      new Thread(new Consumer(buffer)).start();
   }
}
```

- ✓ **Data and Results:**

Data

This program simulates message production and consumption using threading synchronization.

Result

Messages are successfully produced and consumed in a synchronized manner.

- ✓ **Analysis and Inferences:**

Analysis

Producer and consumer operate concurrently, ensuring controlled message exchange.

Inferences

Thread synchronization efficiently manages shared resources in concurrent programming.

VIVA-VOCE Questions (In-Lab):

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

Synchronization between producer and consumer to prevent race conditions, buffer overflow, or underflow.

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

Ensures only one thread accesses a shared resource by acquiring a lock before execution and releasing it afterward.

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

`wait()` pauses a thread when the buffer is full/empty, and `notify()` wakes a waiting thread when a resource is available.

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

Achieved using `synchronized`, locks (`ReentrantLock`), semaphores, or `BlockingQueue`.

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

Ensured through proper synchronization, thread-safe data structures (`BlockingQueue`), locks, and coordination using `wait()` and `notify()` .

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.LinkedList;
import java.util.Queue;

class BoundedBuffer {
   private final Queue<Integer> buffer = new LinkedList<>();
   private final int capacity;

   public BoundedBuffer(int capacity) { this.capacity = capacity; }

   public synchronized void produce(int item) throws InterruptedException {
      while (buffer.size() == capacity) wait();
      buffer.add(item);
      notifyAll();
   }

   public synchronized int consume() throws InterruptedException {
      while (buffer.isEmpty()) wait();
      int item = buffer.poll();
      notifyAll();
      return item;
   }
```

```
}

class Producer implements Runnable {
    private final BoundedBuffer buffer;
    public Producer(BoundedBuffer buffer) { this.buffer = buffer; }

    public void run() {
        try {
            for (int i = 0; i < 20; i++) {
                buffer.produce(i);
                System.out.println("Produced: " + i);
            }
        } catch (InterruptedException e) { Thread.currentThread().interrupt(); }
    }
}

class Consumer implements Runnable {
    private final BoundedBuffer buffer;
    public Consumer(BoundedBuffer buffer) { this.buffer = buffer; }

    public void run() {
        try {
            for (int i = 0; i < 20; i++)
                System.out.println("Consumed: " + buffer.consume());
        } catch (InterruptedException e) { Thread.currentThread().interrupt(); }
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        BoundedBuffer buffer = new BoundedBuffer(10);
        new Thread(new Producer(buffer)).start();
        new Thread(new Consumer(buffer)).start();
    }
}
```

OUTPUT

```
Produced: 0
Produced: 1
Produced: 2
Produced: 3
Produced: 4
Produced: 5
Produced: 6
Produced: 7
Produced: 8
Produced: 9
Produced: 10
Consumed: 0
Consumed: 1
Consumed: 2
Consumed: 3
```

Consumed: 4

Consumed: 5

Consumed: 6

Consumed: 7

Consumed: 8

Consumed: 9

Consumed: 10

Produced: 11

Produced: 12

Produced: 13

Produced: 14

Produced: 15

Produced: 16

Produced: 17

Produced: 18

Produced: 19

Consumed: 11

Consumed: 12

Consumed: 13

Consumed: 14

Consumed: 15

Consumed: 16

Consumed: 17

Consumed: 18

Consumed: 19

Experiment#		Student ID	
Date		Student Name	[@KLWKS_bot] THANOS

✓ **Data and Results:**

Data

The program implements a bounded buffer using Java multithreading synchronization.

Result

The producer generates items, and the consumer retrieves them successfully.

✓ **Analysis and Inferences:**

Analysis

Synchronization ensures proper coordination between producer and consumer threads efficiently.

Inferences

Bounded buffer prevents overflow and underflow, ensuring smooth data exchange.

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.