**Question 1: Thread Implementation - Java Threads [10 marks]**

**Implement a Java program using Thread class to demonstrate the concept of different ways of implementing threads. [5 marks , partial marks are possible]**

// Thread implementation by extending Thread class

class MyThread extends Thread {

public void run() {

for (int i = 1; i <= 5; i++) {

System.out.println("Thread (extended): " + i);

try {

Thread.sleep(500); // Adding a small delay for demonstration

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}

// Thread implementation by implementing Runnable interface

class MyRunnable implements Runnable {

public void run() {

for (int i = 1; i <= 5; i++) {

System.out.println("Thread (implemented): " + i);

try {

Thread.sleep(500); // Adding a small delay for demonstration

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}

public class ThreadDemo {

public static void main(String[] args) {

// Creating threads using both approaches

MyThread thread1 = new MyThread();

Thread thread2 = new Thread(new MyRunnable());

// Starting threads

thread1.start();

thread2.start();

// Main thread continues its execution

for (int i = 1; i <= 5; i++) {

System.out.println("Main Thread: " + i);

try {

Thread.sleep(500); // Adding a small delay for demonstration

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}

**Compare the usage of extending the Thread class and implementing the Runnable interface. Include comments explaining the advantages and disadvantages of each approach. [5 marks, partial marks are possible]**

Advantages and Disadvantages:

Extending Thread class:

Advantages:

Simplicity: It is straightforward and easy to implement as you only need to extend the Thread class and override the run method.

Direct access to thread-specific methods: Since the class itself is a thread, it can directly access the methods like getId(), getName(), etc.

Disadvantages:

Limited flexibility: Java does not support multiple inheritance, so if your class is already extending another class, it cannot extend Thread as well.

Less reusable: Since your class is tied to the Thread class, it becomes less reusable in scenarios where you want the class to be used in a non-threaded context.

Implementing Runnable interface:

Advantages:

Improved flexibility: Since Java supports multiple interfaces, a class can implement Runnable and still extend another class, providing better flexibility.

Enhanced reusability: The class is not bound to the Thread class, making it more reusable in various scenarios.

Disadvantages:

Indirect access to thread-specific methods: Since the class is not a thread itself, it needs to be passed to a Thread constructor, which may make accessing thread-specific methods less straightforward.

Slightly more code: Implementing the Runnable interface usually requires a bit more code compared to extending the Thread class.

**Question 2: Thread Life Cycle - Understanding Thread States [10 marks]**

**Write a Java program that illustrates the life cycle of a thread. Include code segments for each thread state (NEW, RUNNABLE, BLOCKED, WAITING, TIMED\_WAITING, and TERMINATED). Use appropriate sleep and synchronization methods to control the thread's state transitions. [10 marks, partial marks can be awarded ]**

public class ThreadLifeCycleDemo {

public static void main(String[] args) {

// NEW state

Thread newThread = new Thread(() -> {

// RUNNABLE state

System.out.println("Thread is in RUNNABLE state.");

// Simulating some task

for (int i = 0; i < 3; i++) {

System.out.println("Executing task in RUNNABLE state: " + i);

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

// BLOCKED state (synchronized block)

synchronized (ThreadLifeCycleDemo.class) {

System.out.println("Thread is in BLOCKED state (synchronized block).");

}

// WAITING state (waiting for a notification)

synchronized (ThreadLifeCycleDemo.class) {

try {

System.out.println("Thread is in WAITING state (waiting for a notification).");

ThreadLifeCycleDemo.class.wait();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

// TIMED\_WAITING state (sleeping for a specific time)

try {

System.out.println("Thread is in TIMED\_WAITING state (sleeping for 3 seconds).");

Thread.sleep(3000);

} catch (InterruptedException e) {

e.printStackTrace();

}

// TERMINATED state

System.out.println("Thread is in TERMINATED state.");

});

// Starting the thread

newThread.start();

// Main thread

for (int i = 0; i < 5; i++) {

System.out.println("Main thread executing: " + i);

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

// NOTIFY the waiting thread to move from WAITING to RUNNABLE

synchronized (ThreadLifeCycleDemo.class) {

System.out.println("Notifying the waiting thread.");

ThreadLifeCycleDemo.class.notify();

}

// Waiting for the newThread to complete

try {

newThread.join();

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println("Main thread is in TERMINATED state.");

}

}

**Extra points to understand the answer: [not expected in the answer but for the purpose of understanding the code]**

In this program:

The thread starts in the NEW state when it is created.

It transitions to the RUNNABLE state when start() is called.

It enters the BLOCKED state when it tries to access a synchronized block.

It enters the WAITING state when it calls wait() and waits for a notification.

It enters the TIMED\_WAITING state when it calls sleep() for a specific duration.

Finally, it enters the TERMINATED state when the run() method completes.

Note: The synchronized blocks are used for simplicity to illustrate state transitions and synchronization mechanisms. In real-world scenarios, you may use more sophisticated synchronization mechanisms based on your application requirements.

**Question 3: Producer-Consumer Problem - Synchronization in Java [10 marks]**

**Implement the classic Producer-Consumer problem in Java using synchronised methods or blocks to ensure proper synchronization. Clearly demonstrate the shared buffer, the producer, and consumer threads. Include proper handling of buffer size and use wait and notify mechanisms. [10 marks, partial marks can be awarded if part of the code is correct]**

import java.util.LinkedList;

import java.util.Queue;

class SharedBuffer {

private final Queue<Integer> buffer;

private final int maxSize;

public SharedBuffer(int maxSize) {

this.buffer = new LinkedList<>();

this.maxSize = maxSize;

}

// Producer adds an item to the buffer

public synchronized void produce(int item) {

while (buffer.size() == maxSize) {

// Buffer is full, wait for the consumer to consume items

try {

System.out.println("Buffer is full. Producer is waiting.");

wait();

} catch (InterruptedException e) {

Thread.currentThread().interrupt();

}

}

buffer.add(item);

System.out.println("Produced: " + item);

// Notify the consumer that an item is available

notify();

}

// Consumer removes an item from the buffer

public synchronized int consume() {

while (buffer.isEmpty()) {

// Buffer is empty, wait for the producer to produce items

try {

System.out.println("Buffer is empty. Consumer is waiting.");

wait();

} catch (InterruptedException e) {

Thread.currentThread().interrupt();

}

}

int item = buffer.poll();

System.out.println("Consumed: " + item);

// Notify the producer that space is available in the buffer

notify();

return item;

}

}

class Producer extends Thread {

private final SharedBuffer buffer;

public Producer(SharedBuffer buffer) {

this.buffer = buffer;

}

@Override

public void run() {

for (int i = 1; i <= 5; i++) {

buffer.produce(i);

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}

class Consumer extends Thread {

private final SharedBuffer buffer;

public Consumer(SharedBuffer buffer) {

this.buffer = buffer;

}

@Override

public void run() {

for (int i = 1; i <= 5; i++) {

int item = buffer.consume();

try {

Thread.sleep(1500);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}

public class ProducerConsumerDemo {

public static void main(String[] args) {

SharedBuffer buffer = new SharedBuffer(3);

Producer producer = new Producer(buffer);

Consumer consumer = new Consumer(buffer);

producer.start();

consumer.start();

}

}

**Extra points to understand the answer: [not expected in the answer but for the purpose of understanding the code]**

In this example:

The SharedBuffer class represents the shared buffer with methods produce and consume.

The Producer class represents the producer thread that adds items to the buffer.

The Consumer class represents the consumer thread that removes items from the buffer.

The wait and notify mechanisms are used to synchronize the access to the shared buffer.

The buffer has a maximum size (maxSize), and the producer waits when the buffer is full, while the consumer waits when the buffer is empty.

The Thread.sleep statements are used to simulate the production and consumption of items over time.

This example demonstrates proper synchronization to avoid issues like race conditions and deadlock in a producer-consumer scenario.**Question 4: Finite State Process [10 marks]**

**Give the FSP process definitions for the two processes ChocVM and Animals based on their following Labelled Transition System (LTS) graphs:**

A diagram of a diagram of a diagram

Description automatically generated with medium confidence

i) ChocVM = ({crunchie, flake} -> runout -> ChocVM | mars -> deliverMars -> ChocVM | twirl -> deliverTwrirl -> ChocVM). [5 marks]

**Or condition order can be changed.**

ii) Animals = (cat -> (purr -> END | meow -> Animals) | cow -> oink -> ERROR). [5 marks]

**Or condition order can be changed. ERROR process name can be anything other than Animals, END, STOP**

**All the process names should start with Uppercase letters, otherwise reduce the marks.**

**Incorrect answers no marks.**

**Question 5: Monitor in Java - Synchronization Mechanisms [15 marks]**

**Design a concurrent Java program that utilizes the monitor concept for synchronization. Implement a scenario where multiple threads access a shared resource and demonstrate the use of synchronized methods or blocks to prevent data corruption and race conditions.**

class SharedResource {

private int counter = 0;

// Synchronized method to increment the counter

public synchronized void increment() {

int oldValue = counter;

System.out.println(Thread.currentThread().getName() + " - Before Increment: " + oldValue);

// Simulating some processing time

try {

Thread.sleep(100);

} catch (InterruptedException e) {

e.printStackTrace();

}

counter++;

int newValue = counter;

System.out.println(Thread.currentThread().getName() + " - After Increment: " + newValue);

}

}

class SharedResourceUser extends Thread {

private final SharedResource sharedResource;

public SharedResourceUser(SharedResource sharedResource) {

this.sharedResource = sharedResource;

}

@Override

public void run() {

for (int i = 0; i < 3; i++) {

sharedResource.increment();

}

}

}

public class ConcurrentProgramWithMonitor {

public static void main(String[] args) {

SharedResource sharedResource = new SharedResource();

// Creating multiple threads that access the shared resource

Thread thread1 = new SharedResourceUser(sharedResource);

Thread thread2 = new SharedResourceUser(sharedResource);

// Starting the threads

thread1.start();

thread2.start();

}

}

**Not part of the answer but to understand it better:**

In this example:

The SharedResource class has a counter and a synchronized method (increment) that increments the counter in a thread-safe manner.

The SharedResourceUser class represents threads that use the shared resource. Each thread calls the increment method multiple times.

The synchronized keyword in the increment method ensures that only one thread can execute this method at a time, preventing race conditions and data corruption.

Output may vary due to thread scheduling, but the output will always show that the increments are done in a safe and synchronized manner, avoiding data corruption:

Thread-0 - Before Increment: 0

Thread-0 - After Increment: 1

Thread-0 - Before Increment: 1

Thread-0 - After Increment: 2

Thread-0 - Before Increment: 2

Thread-0 - After Increment: 3

Thread-1 - Before Increment: 3

Thread-1 - After Increment: 4

Thread-1 - Before Increment: 4

Thread-1 - After Increment: 5

Thread-1 - Before Increment: 5

Thread-1 - After Increment: 6

This demonstrates the use of synchronized methods to ensure the proper synchronization of access to shared resources, preventing race conditions and maintaining data integrity.

**Question 6: Advanced Synchronization - ReentrantLock and Condition [15 marks]**

**Implement a concurrent program using ReentrantLock and Condition to manage multiple threads. Create a scenario where threads need to wait for specific conditions before proceeding.**

[10 marks]

import java.util.concurrent.locks.Condition;

import java.util.concurrent.locks.ReentrantLock;

class SharedResource {

private int counter = 0;

private final ReentrantLock lock = new ReentrantLock();

private final Condition condition = lock.newCondition();

public void waitForCondition(int targetValue) throws InterruptedException {

lock.lock();

try {

while (counter < targetValue) {

// Wait until the counter reaches the target value

condition.await();

}

} finally {

lock.unlock();

}

}

public void incrementCounter() {

lock.lock();

try {

counter++;

System.out.println(Thread.currentThread().getName() + " - Counter Incremented: " + counter);

// Signal waiting threads when the counter is incremented

condition.signalAll();

} finally {

lock.unlock();

}

}

}

class SharedResourceUser extends Thread {

private final SharedResource sharedResource;

public SharedResourceUser(SharedResource sharedResource) {

this.sharedResource = sharedResource;

}

@Override

public void run() {

try {

// Wait for the condition (counter value) to be met

sharedResource.waitForCondition(5);

System.out.println(Thread.currentThread().getName() + " - Condition Met!");

// Perform some task after the condition is met

// For demonstration purposes, simply print a message

System.out.println(Thread.currentThread().getName() + " - Task Completed!");

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

public class ConcurrentProgramWithReentrantLock {

public static void main(String[] args) {

SharedResource sharedResource = new SharedResource();

// Creating multiple threads that use the shared resource

Thread thread1 = new SharedResourceUser(sharedResource);

Thread thread2 = new SharedResourceUser(sharedResource);

// Starting the threads

thread1.start();

thread2.start();

// Simulating the increment of the counter over time

for (int i = 0; i < 6; i++) {

try {

Thread.sleep(1000); // Simulating some delay

} catch (InterruptedException e) {

e.printStackTrace();

}

sharedResource.incrementCounter();

}

}

}

**Discuss the advantages of using ReentrantLock and Condition over traditional synchronization methods.**

[5 marks]

Advantages of using ReentrantLock and Condition over traditional synchronization methods like synchronized blocks/methods:

Fine-grained control: With ReentrantLock, you have more fine-grained control over the locking and unlocking process. You can choose when to acquire and release the lock, providing more flexibility.

Interruptible locking: The ReentrantLock supports interruptible locking. Threads waiting on a lock can be interrupted, allowing for more responsive handling of thread interruptions.

Condition signaling: Condition provides a more expressive way to manage the waiting and signaling process. It allows you to have multiple conditions associated with a single lock, enabling more complex synchronization scenarios.

Try-locking: The tryLock method in ReentrantLock allows you to attempt to acquire the lock without waiting. This can be useful in situations where waiting for a lock indefinitely might not be desirable.

Lock fairness: ReentrantLock provides options for fair locking, ensuring that the longest-waiting thread gets the lock, which can help prevent thread starvation.

While traditional synchronization methods like synchronized are simpler and often sufficient for many scenarios, ReentrantLock and Condition provide additional features and flexibility for advanced synchronization requirements.

**Not part of marking scheme but to understand:**

In this example:

The SharedResource class contains a counter, a ReentrantLock, and a Condition.

The waitForCondition method is used by threads to wait until a specific condition (counter-reaching a target value) is met.

The incrementCounter method increments the counter and signals waiting threads when the condition is met.

The SharedResourceUser class represents threads that use the shared resource. They wait for the condition to be met before proceeding with their task.

**Question 7: Finite State Machine [15 marks]**

**The following FSP program SYSTEM models two processes Successor and Predecessor sharing a Counter.**

const MAX = 2

range RANGE = 0..MAX

Counter ( N = MAX ) = CVal[ N ],

CVal[ cv : RANGE ] = ( when ( cv < MAX ) succ  -> CVal[ cv + 1 ]

  | when ( cv > 0 ) pred  -> CVal[ cv - 1 ]

  | succVal[ cv ]  -> CVal[ cv ]

  | when ( cv == 0 ) isZero  -> CVal[ cv ]

  ) .

Successor = ( succ -> Successor

  | succVal[ i : RANGE ] -> Successor ) .

Predecessor = ( pred -> Predecessor

  | isZero -> wait -> Predecessor ) .

||SYSTEM = ( Counter(2) || Successor || Predecessor ) .

Answer the following questions:

* Complete the following *alphabet diagram* for ||SYSTEM, by stating which actions are in each of the five areas*A, B, C, D, E*  in the following diagram:

A diagram of a number of circles

Description automatically generated

* For each action in SYSTEM's alphabet state: the type of action (synchronous or asynchronous) and all the processes that perform it. [10 marks]
* What actions can the SYSTEM initially perform? [5 marks]

A: wait

B: isZero, pred

C:

D: succ, succVal[0], succVal[1], succVal[2]

E:

isZero : synchronous

pred : synchronous

succ : synchronous

succVal[0] : synchronous

succVal[1] : synchronous

succVal[2] : synchronous

wait : asynchronous

What actions can the SYSTEM initially perform? : pred, succVal[2]

**[No partial marks if answer is wrong]**

**Question 8: Basic Thread Synchronization [15 marks]**

**Write a Java program that creates two threads, ThreadA and ThreadB. Both threads should increment a shared counter variable inside a synchronized method. Ensure that the increment operation is atomic and that the final value of the counter reflects the total increments performed by both threads.**

class SharedCounter {

private int counter = 0;

// Synchronized method to increment the counter

public synchronized void increment() {

counter++;

}

// Getter method to retrieve the current value of the counter

public int getCounter() {

return counter;

}

}

class ThreadA extends Thread {

private final SharedCounter sharedCounter;

public ThreadA(SharedCounter sharedCounter) {

this.sharedCounter = sharedCounter;

}

@Override

public void run() {

for (int i = 0; i < 10000; i++) {

sharedCounter.increment();

}

}

}

class ThreadB extends Thread {

private final SharedCounter sharedCounter;

public ThreadB(SharedCounter sharedCounter) {

this.sharedCounter = sharedCounter;

}

@Override

public void run() {

for (int i = 0; i < 10000; i++) {

sharedCounter.increment();

}

}

}

public class SharedCounterExample {

public static void main(String[] args) {

SharedCounter sharedCounter = new SharedCounter();

// Creating two threads, ThreadA and ThreadB, sharing the same counter

ThreadA threadA = new ThreadA(sharedCounter);

ThreadB threadB = new ThreadB(sharedCounter);

// Starting both threads

threadA.start();

threadB.start();

// Waiting for both threads to complete

try {

threadA.join();

threadB.join();

} catch (InterruptedException e) {

e.printStackTrace();

}

// Displaying the final value of the counter

System.out.println("Final Counter Value: " + sharedCounter.getCounter());

}

}

[Not part of the answer but to understand the answer]

In this example:

The SharedCounter class contains a counter variable and a synchronized method (increment) that increments the counter. The getCounter method is used to retrieve the current value of the counter.

Both ThreadA and ThreadB instances are created, each with a reference to the shared SharedCounter object.

The run method of each thread increments the counter in a loop (10,000 times in this example) using the synchronized increment method.

The main method creates the shared counter, starts both threads, waits for them to complete using join, and then displays the final value of the counter.

The use of synchronization ensures that the increment operation is atomic, and the final value of the counter represents the total increments performed by both threads.