Concurrency-4 Synchronization with Semaphores

Agenda

- Synchronisation using Semaphores
  - Producer Consumer Problem using Semaphores
  - Producer Consumer Problem using Concurrent Data Structure (Queue)
  - Print In Order LeetCode Problem
- Deadlocks
- Additional Topics
  - wait(), notify() methods
  - Producer Consumer Using wait() & notify()
- Coding Projects (Optional)
  - Traffic Intersection Control
  - Resource Pooling in Library
- LeetCode Problems
- Additional Resources

Synchronisation using Semaphores

Producer Consumer Problem : A T-Shirt Store Example

The Producer-Consumer problem is a classic synchronization problem where two processes, the producer and the consumer, share a common, fixed-size buffer or store. The producer produces items and adds them to the buffer, while the consumer consumes items from the buffer. Semaphores are synchronization primitives that can be used to solve this problem efficiently.

Problem Description Let's use a T-shirt store as an analogy for the Producer-Consumer problem. The T-shirt store has a limited capacity to store T-shirts. Producers can create T-shirts and add them to the store, and consumers can buy T-shirts from the store. The challenge is to ensure that the store doesn't overflow with T-shirts or run out of stock.

Java Implementation-1 Using Semaphores (simplified implementation than what is covered in class, here we don't maintain an actual queue for T-Shirts, just the count)

```
import java.util.concurrent.Semaphore;

public class TShirtStore {
    private static final int STORE_CAPACITY = 5;
    private static Semaphore mutex = new Semaphore(1); // Controls access to critical sections
    private static Semaphore empty = new Semaphore(STORE_CAPACITY); // Represents empty slots in the store
    private static Semaphore full = new Semaphore(0); // Represents filled slots in the store
    private static int tShirtCount = 0;
```

```
    static class Producer implements Runnable {
        @Override
        public void run() {
            try {
                while (true) {
                    empty.acquire(); // Wait for an empty slot
                    mutex.acquire(); // Enter critical section

                    // Produce a T-shirt
                    System.out.println("Producer produces a T-shirt. Total T-shirts: "
+ ++tShirtCount);

                    mutex.release(); // Exit critical section
                    full.release(); // Signal that a T-shirt is ready to be consumed
                    Thread.sleep(1000); // Simulate production time
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    static class Consumer implements Runnable {
        @Override
        public void run() {
            try {
                while (true) {
                    full.acquire(); // Wait for a T-shirt to be available
                    mutex.acquire(); // Enter critical section

                    // Consume a T-shirt
                    System.out.println("Consumer buys a T-shirt. Total T-shirts: " + -
-tShirtCount);

                    mutex.release(); // Exit critical section
                    empty.release(); // Signal that a slot is available for production
                    Thread.sleep(1500); // Simulate consumption time
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    public static void main(String[] args) {
        Thread producerThread = new Thread(new Producer());
        Thread consumerThread = new Thread(new Consumer());

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

Semaphores

- **mutex**: Controls access to the critical sections (mutex stands for mutual exclusion).
- **empty**: Represents the number of empty slots in the store, initially set to the store's capacity.
- **full**: Represents the number of filled slots in the store, initially set to 0.

Producer:

- The producer acquires an empty slot using empty.acquire() and enters the critical section with mutex.acquire().
- It produces a T-shirt, increments the count, releases the mutex, and signals that a T-shirt is ready for consumption using full.release().

Consumer:

- The consumer acquires a filled slot using full.acquire() and enters the critical section with mutex.acquire().
- It consumes a T-shirt, decrements the count, releases the mutex, and signals that an empty slot is available for production using empty.release().

Simulated Production and Consumption: `Thread.sleep()` is used to simulate the time it takes to produce and consume T-shirts. **Execution**: When you run this program, you will observe the producer producing T-shirts and the consumer buying T-shirts. The store's capacity is maintained, and semaphores ensure proper synchronization between the producer and the consumer.

This example demonstrates how semaphores can be used to solve the Producer-Consumer problem efficiently, preventing issues such as overproduction or stockouts.

Java Implementation -2 using Semaphores Producer.java

```
public class Producer implements Runnable{
    private Queue<Object> queue;
    private int maxSize;
    private String name;
    private Semaphore producerSemaphore;
    private Semaphore consumerSemaphore;

    Producer(Queue<Object> queue, int maxSize, String name, Semaphore ps, Semaphore
cs){
        this.queue = queue;
        this.maxSize = maxSize;
        this.name = name;
        this.producerSemaphore = ps;
        this.consumerSemaphore = cs;
    }
    @Override
    public void run() {
        while(true){
            try {
                producerSemaphore.acquire();
            } catch (InterruptedException e) {
```

```
                throw new RuntimeException(e);
            }
            if(queue.size()<this.maxSize){
                    System.out.println(this.name + " adding to queue, Size " +
queue.size());
                    queue.add(new Object());
                }
            consumerSemaphore.release();
        }
    }
}
```

Consumer.java

```
package Multithreading.ProducerConsumer;

import java.util.Queue;
import java.util.concurrent.Semaphore;

public class Consumer implements Runnable{
    private Queue<Object> queue;
    private int maxSize;
    private String name;
    private Semaphore producerSemaphore;
    private Semaphore consumerSemaphore;

    Consumer(Queue<Object> queue, int maxSize, String name, Semaphore ps, Semaphore
cs){
        this.queue = queue;
        this.maxSize = maxSize;
        this.name = name;
        this.producerSemaphore = ps;
        this.consumerSemaphore = cs;
    }

    @Override
    public void run() {
            while(true){
                try {
                    consumerSemaphore.acquire();
                } catch (InterruptedException e) {
                    throw new RuntimeException(e);
                }
                if(queue.size()>0){
                        System.out.println(this.name + " removing from queue, Size " +
queue.size());
                        queue.remove();
                    }
                producerSemaphore.release();
            }
    }
}
```

Client.java

Here we create multiple producers and multiple consumers.

```
public class Client {
    public static void main(String[] args) {
        Queue<Object> objects = new ConcurrentLinkedQueue<>();
        int maxSize = 6;
        Semaphore producerSemaphore = new Semaphore(maxSize);
        Semaphore consumerSemaphore = new Semaphore(0);

        Producer p1 = new
Producer(objects,6,"p1",producerSemaphore,consumerSemaphore);
        Producer p2 = new
Producer(objects,6,"p2",producerSemaphore,consumerSemaphore);
        Producer p3 = new
Producer(objects,6,"p3",producerSemaphore,consumerSemaphore);

        Consumer c1 = new
Consumer(objects,6,"c1",producerSemaphore,consumerSemaphore);
        Consumer c2 = new
Consumer(objects,6,"c2",producerSemaphore,consumerSemaphore);
        Consumer c3 = new
Consumer(objects,6,"c3",producerSemaphore,consumerSemaphore);
        Consumer c4 = new
Consumer(objects,6,"c4",producerSemaphore,consumerSemaphore);
        Consumer c5 = new
Consumer(objects,6,"c5",producerSemaphore,consumerSemaphore);

        Thread t1 = new Thread(p1);
        Thread t2 = new Thread(p2);
        Thread t3 = new Thread(p3);
        Thread t4 = new Thread(c1);
        Thread t5 = new Thread(c2);
        Thread t6 = new Thread(c3);
        Thread t7 = new Thread(c4);
        Thread t8 = new Thread(c5);

        t1.start();
        t2.start();
        t3.start();
        t4.start();
        t5.start();
        t6.start();
        t7.start();
    }
}
```

Java Implementation -3 using Concurrent Data Structure Here is another implementation in which you can use a ConcurrentLinkedQueue in place of semaphores to ensure concurrency is handled well. **Producer.java**

```java
import java.util.Queue;

public class Producer implements Runnable{
    private Queue<Object> queue;
    int maxSize;
    String name;

    public Producer(Queue<Object> queue,int maxSize, String name){
        this.queue = queue;
        this.maxSize = maxSize;
        this.name = name;
    }

    @Override
    public void run() {
        //Each producer wants to continuously produces
        // T-Shirts and add them to the queue if there is space available

        while(true){
            synchronized (queue){
                if(queue.size()<this.maxSize){
                    System.out.println("Adding - "+ queue.size());
                    queue.add(new Object());
                }
            }
        }
    }
}
```

Consumer.java

```java
import java.util.Queue;

public class Consumer implements Runnable{
    private Queue<Object> queue;
    int maxSize;
    String name;

    public Consumer(Queue<Object> queue,int maxSize, String name){
        this.queue = queue;
        this.maxSize = maxSize;
        this.name = name;
    }

    @Override
    public void run() {
        //Each producer wants to continuously produces
        // T-Shirts and add them to the queue if there is space available
        while(true){
            synchronized (queue) {
                if (queue.size() > 0) {
                    System.out.println("Removing - "+ queue.size());
```

```
                    queue.remove();
                }
            }
        }
    }
}
```

Main.java

```
public class Main {
    public static void main(String[] args) {
        //Shared Object
        Queue<Object> q = new ConcurrentLinkedQueue<>();
        int maxSize = 6;

        Producer p1 = new Producer(q,maxSize,"p1");
        Producer p2= new Producer(q,maxSize,"p2");
        Producer p3  = new Producer(q,maxSize,"p3");

        Consumer c1 = new Consumer(q,maxSize,"c1");
        Consumer c2 = new Consumer(q,maxSize,"c2");
        Consumer c3 = new Consumer(q,maxSize,"c3");
        Consumer c4 = new Consumer(q,maxSize,"c4");
        Consumer c5 = new Consumer(q,maxSize,"c5");

        Thread t1 = new Thread(p1);
        Thread t2 = new Thread(p2);
        Thread t3 = new Thread(p3);
        Thread t4 = new Thread(c1);
        Thread t5 = new Thread(c2);
        Thread t6 = new Thread(c3);
        Thread t7 = new Thread(c4);
        Thread t8 = new Thread(c5);

        t1.start();
        t2.start();
        t3.start();
        t4.start();
        t5.start();
        t6.start();
        t7.start();
        t8.start();
    }
}
```

Time To Try - Print In Order (LeetCode)

Try to solve the following problem using Semaphores Concept.

- Print In Order - LeetCode

Solution

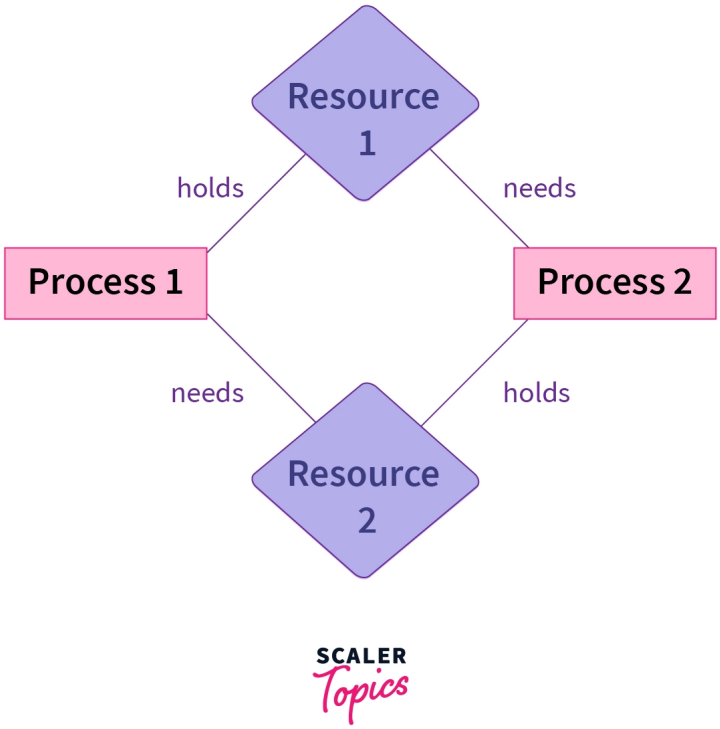
```
class Foo {
    Semaphore semaSecond = new Semaphore(0);
    Semaphore semaThird = new Semaphore(0);
    public Foo() {

    }
    public void first(Runnable printFirst) throws InterruptedException {
        printFirst.run();
        semaSecond.release();
    }
    public void second(Runnable printSecond) throws InterruptedException {
        semaSecond.acquire();
        printSecond.run();
        semaThird.release();
    }
    public void third(Runnable printThird) throws InterruptedException {
        semaThird.acquire();
        printThird.run();
    }
}
```

DeadLocks

A deadlock in OS is a situation in which more than one process is blocked because it is holding a resource and also requires some resource that is acquired by some other process.

Conditions for a deadlock

- `Mutual exclusion` - The resource is held by only one process at a time and cannot be acquired by another process.
- `Hold and wait` - A process is **holding** a resource and **waiting** for another resource to be released by another a process.
- `No preemption` - The resource can only be released once the execution of the process is complete.
- `Circular wait` - A set of processes are waiting for each other circularly. Process `P1` is waiting for process `P2` and process `P2` is waiting for process `P1`.



Process P1 and P2 are in a deadlock because:

- Resources are non-shareable. (Mutual exclusion)
- Process 1 holds "Resource 1" and is waiting for "Resource 2" to be released by process 2. (Hold and wait)
- None of the processes can be preempted. (No preemption)
- "Resource 1" and needs "Resource 2" from Process 2 while Process 2 holds "Resource 2" and requires "Resource 1" from Process 1. (Circular wait)

Tackling deadlocks

There are three ways to tackle deadlocks:

- Prevention - Implementing a mechanism to prevent the deadlock.
- Avoidance - Avoiding deadlocks by not allocating resources when deadlocks are possible.
- Detecting and recovering - Detecting deadlocks and recovering from them.
- Ignorance - Ignore deadlocks as they do not happen frequently.

1. Prevention and avoidance

Deadlock prevention means to block at least one of the four conditions required for deadlock to occur. If we are able to block any one of them then deadlock can be prevented. Spooling and non-blocking synchronization algorithms are used to prevent the above conditions. In deadlock prevention all the requests are granted in a finite amount of time.

In Deadlock avoidance we have to anticipate deadlock before it really occurs and ensure that the system does not go in unsafe state.It is possible to avoid deadlock if resources are allocated carefully. For deadlock avoidance we use Banker’s and Safety algorithm for resource allocation purpose. In deadlock avoidance the maximum number of resources of each type that will be needed are stated at the beginning of the process.

2. Detecting and recovering from deadlocks

We let the system fall into a deadlock and if it happens, we detect it using a detection algorithm and try to recover.

Some ways of recovery are as follows:

- Aborting all the deadlocked processes.
- Abort one process at a time until the system recovers from the deadlock.
- Resource Preemption: Resources are taken one by one from a process and assigned to higher priority processes until the deadlock is resolved.

3. Ignorance

The system assumes that deadlock never occurs. Since the problem of deadlock situation is not frequent, some systems simply ignore it. Operating systems such as UNIX and Windows follow this approach. However, if a deadlock occurs we can reboot our system and the deadlock is resolved automatically.

4. Tackling deadlocks at an application level

- Set timeouts for all the processes. If a process does not respond within the timeout period, it is killed.
- Implementing with caution: Use interfaces that handle or provide callbacks if locks are held by other processes.
- Add timeout to locks: If a process requests a lock, and it is held by another process, it will wait for the lock to be released until the timeout expires.

Additonal Topics

Inter-thread Communication using wait() and notify()

wait() & notify() - Recording Link Certainly! In Java, the wait() and notify() methods are part of the built-in mechanism for inter-thread communication and synchronization. These methods are used to coordinate the activities of multiple threads, allowing them to work together effectively. Let's break down these concepts for beginners:

wait() Method:

- The wait() method is called on an object within a synchronized context (i.e., within a method or block synchronized on that object).
- It causes the current thread to release the lock on the object and enter a state of waiting. Purpose:
- wait() is used when a thread needs to wait for a certain condition to be met before proceeding.

For example, if a thread is waiting for a shared resource to be available, it can call wait() until another thread notifies it that the resource is ready. Example:

```
synchronized (sharedObject) {
    while (!conditionMet) {
        try {
```

```
            sharedObject.wait(); // Releases the lock and waits for notification
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    // Continue with the critical section
}
```

notify() Method:

- The notify() method is called on an object within a synchronized context.
- It wakes up one of the threads that are currently waiting on that object. Purpose:
- notify() is used to signal that a condition (for which threads are waiting) has been met and that one of the waiting threads can proceed.
- It is essential to note that notify() only wakes up one waiting thread. If there are multiple waiting threads, it is not determined which one will be awakened. Example:

```
synchronized (sharedObject) {
    // Perform some operations and change the condition
    conditionMet = true;

    // Notify one of the waiting threads
    sharedObject.notify();
}
```

Important Points:

- Both wait() and notify() must be called within a synchronized context to avoid illegal monitor state exceptions.
- The calling thread must hold the lock on the object on which it is calling wait() or notify().
- The wait() method releases the lock, allowing other threads to access the synchronized block or method.
- The notify() method signals a waiting thread to wake up, allowing it to reacquire the lock and continue execution.

Example Scenario: Consider a scenario where multiple threads are working on a shared resource. If a thread finds that the resource is not yet available (e.g., a buffer is empty), it can call wait() to release the lock and wait until another thread populates the buffer and calls notify() to signal that the resource is ready for consumption.

In summary, wait() and notify() are fundamental methods for thread synchronization in Java, enabling threads to communicate and coordinate their activities efficiently.

Producer Consumer using Wait() and Notify() Recording Link

Implementation -1 (Simplified) Here is a simplifed version of Producer Consumer as discussed in above LIVE Class. **ProducerConsumer.java**

```
public class ProducerConsumer {

    public void produce() throws InterruptedException {
        synchronized (this){
            System.out.println("Produced - T-shirt");
            //release the lock on the shared resource and wait till some other invokes
this
            wait();
            System.out.println("Going to produce another T-Shirt");
        }
    }

    public void consume() throws InterruptedException {
       Thread.sleep(1000);
       Scanner sc = new Scanner(System.in);

       synchronized (this) {
           System.out.println("Take t-shirt? ");
           sc.nextLine();
           System.out.println("Recieved T-shirt");
           notify();
           Thread.sleep(3000);
       }
    }
}
```

PCDemo.java

```
public class PCDemo {
    public static void main(String[] args) {
        ProducerConsumer pc = new ProducerConsumer();

        Thread t1 = new Thread(()->{
            try {
                pc.produce();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        });
        Thread t2 = new Thread(()->{
            try {
                pc.consume();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        });

        t1.start();
        t2.start();
    }
}
```

Implementation-2 A more robust Implementation is as follows:

```
import java.util.LinkedList;

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

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

    public void produce() throws InterruptedException {
        synchronized (this) {
            while (buffer.size() == capacity) {
                // Buffer is full, wait for consumer to consume
                wait();
            }

            // Produce an item and add to the buffer
            int newItem = (int) (Math.random() * 100);
            buffer.add(newItem);
            System.out.println("Produced: " + newItem);

            // Notify the consumer that an item is available
            notify();
        }
    }

    public void consume() throws InterruptedException {
        synchronized (this) {
            while (buffer.isEmpty()) {
                // Buffer is empty, wait for producer to produce
                wait();
            }

            // Consume an item from the buffer
            int consumedItem = buffer.removeFirst();
            System.out.println("Consumed: " + consumedItem);

            // Notify the producer that a slot is available in the buffer
            notify();
        }
    }
}

class Producer implements Runnable {
    private final SharedBuffer sharedBuffer;

    public Producer(SharedBuffer sharedBuffer) {
        this.sharedBuffer = sharedBuffer;
    }
```

```
    @Override
    public void run() {
        try {
            while (true) {
                sharedBuffer.produce();
                Thread.sleep(1000); // Simulate production time
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

class Consumer implements Runnable {
    private final SharedBuffer sharedBuffer;

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

    @Override
    public void run() {
        try {
            while (true) {
                sharedBuffer.consume();
                Thread.sleep(1500); // Simulate consumption time
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

public class ProducerConsumerExample {
    public static void main(String[] args) {
        SharedBuffer sharedBuffer = new SharedBuffer(5);

        Thread producerThread = new Thread(new Producer(sharedBuffer));
        Thread consumerThread = new Thread(new Consumer(sharedBuffer));

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

In this example:

- `SharedBuffer` is the shared buffer where the producer produces and the consumer consumes items.
- The Producer class produces items and adds them to the buffer.
- The Consumer class consumes items from the buffer.

- The main method creates instances of the shared buffer, producer, and consumer, and starts their respective threads.
- This solution uses the `wait()` and `notify()` methods to ensure that the producer waits when the buffer is full and the consumer waits when the buffer is empty, allowing for proper coordination and synchronization between the two threads.

Coding Projects

Coding Problem 1 : Traffic Intersection Control

Implement a program that simulates a traffic intersection control system using Java Semaphores. The intersection has two roads, each with its own traffic signal. The traffic lights control the flow of traffic through the intersection. **Requirements:**

- There are two roads, Road A and Road B, crossing at the intersection.
- Each road has its own traffic signal (Semaphore) controlling the traffic flow.
- The traffic lights for Road A and Road B have a cycle of Green, Yellow, and Red signals. Only one road should have a green light at a time, while the other road has a red light. The intersection should allow a smooth transition between green lights for both roads.

Constraints:

- The time duration for each signal (Green, Yellow, Red) can be adjusted based on the program's design.
- Use Semaphores to control access to the traffic signals and ensure a safe transition. Each road signal should run in a separate thread.
- Implement a way to visually represent the current state of the traffic signals and indicate which road has the green light.

Sample Output:

Road A: Green Road B: Red

[... Some time passes ...]

Road A: Yellow Road B: Red

[... Some time passes ...]

Road A: Red Road B: Green

Notes:

- The program should demonstrate proper synchronization to ensure that only one road has a green light at any given time.
- You may choose to implement additional features such as a pedestrian signal or a button for road switching.
- Consider the safety and efficiency of the intersection control system.
- This problem statement reflects a real-world scenario where semaphores can be used to control access to shared resources (in this case, the green light for each road) in a concurrent environment. Students can implement the solution to gain hands-on experience with semaphore-based synchronization in a practical setting.

Solution:

```java
import java.util.concurrent.Semaphore;

class TrafficIntersectionControl {
    private Semaphore roadASemaphore = new Semaphore(1); // Semaphore for Road A's traffic signal
    private Semaphore roadBSemaphore = new Semaphore(0); // Semaphore for Road B's traffic signal

    // Simulate time passing
    private void sleep(int seconds) {
        try {
            Thread.sleep(seconds * 1000);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }

    // Switch the traffic lights for Road A and Road B
    private void switchLights() {
        System.out.println("Switching lights...");

        try {
            roadASemaphore.acquire(); // Acquire the semaphore for Road A
            roadBSemaphore.release(); // Release the semaphore for Road B
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }

    // Simulate traffic on Road A
    private void trafficOnRoadA() {
        while (true) {
            System.out.println("Road A: Green");
            sleep(5); // Green light duration

            System.out.println("Road A: Yellow");
            sleep(2); // Yellow light duration

            System.out.println("Road A: Red");
            switchLights(); // Switch to Road B
        }
    }

    // Simulate traffic on Road B
    private void trafficOnRoadB() {
        while (true) {
            try {
                roadBSemaphore.acquire(); // Acquire the semaphore for Road B
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
```

```
            System.out.println("Road B: Green");
            sleep(5); // Green light duration

            System.out.println("Road B: Yellow");
            sleep(2); // Yellow light duration

            System.out.println("Road B: Red");
            switchLights(); // Switch to Road A
        }
    }

    public static void main(String[] args) {
        TrafficIntersectionControl control = new TrafficIntersectionControl();

        // Create and start threads for traffic on Road A and Road B
        Thread roadAThread = new Thread(control::trafficOnRoadA);
        Thread roadBThread = new Thread(control::trafficOnRoadB);

        roadAThread.start();
        roadBThread.start();
    }
}
```

Coding Problem - 2 Real-Life Problem Statement: Resource Pooling in a Library

Imagine a library that has a collection of books, and multiple students want to borrow and return books from the library. Implement a program that uses Java Semaphores to control access to the books, ensuring that the available resources (books) are used efficiently.

Requirements: The library has a fixed number of books (resources) available for borrowing. Students can borrow books from the library and return them after reading. The library enforces a limit on the maximum number of students who can borrow books simultaneously. When a student returns a book, another student can borrow it if there is an available slot.

Constraints: Use Semaphores to control access to the shared resource (books). Each student should run in a separate thread. The program should handle the borrowing and returning of books concurrently. Implement a way to visually represent the current state of the library, indicating which books are borrowed and available.

Example Output:

Student 1 borrows Book A Library: [Book A is borrowed, Book B is available, Book C is available]

Student 2 borrows Book B Library: [Book A is borrowed, Book B is borrowed, Book C is available]

Student 1 returns Book A Library: [Book A is available, Book B is borrowed, Book C is available]

… **Notes:** Ensure that the library's state is properly synchronized to avoid race conditions. You may choose to implement additional features, such as a waitlist for students. Consider scenarios where a student may need to wait if all books are

currently borrowed. This problem statement reflects a real-world scenario where semaphores can be used to control access to a limited set of resources, ensuring that they are utilized efficiently and concurrently by multiple entities. Students can implement the solution to gain practical experience with semaphores in a resource pooling scenario.

Sample Code

```
import java.util.concurrent.Semaphore;

class Library {
    private static final int MAX_STUDENTS = 2;
    private static final int MAX_BOOKS = 3;

    private Semaphore availableBooks = new Semaphore(MAX_BOOKS, true);
    private Semaphore studentSlots = new Semaphore(MAX_STUDENTS, true);

    // Simulate time passing
    private void sleep(int seconds) {
        try {
            Thread.sleep(seconds * 1000);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }

    // Borrow a book from the library
    private void borrowBook(String book, int studentId) {
        try {
            availableBooks.acquire(); // Acquire a book
            System.out.println("Student " + studentId + " borrows " + book);
            sleep(2); // Simulate reading time
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }

    // Return a book to the library
    private void returnBook(String book, int studentId) {
        System.out.println("Student " + studentId + " returns " + book);
        availableBooks.release(); // Release the returned book
    }

    // Simulate a student using the library
    private void student(int studentId) {
        while (true) {
            try {
                studentSlots.acquire(); // Acquire a student slot
                String bookToBorrow = "Book " + (studentId % MAX_BOOKS + 1);
                borrowBook(bookToBorrow, studentId);
                returnBook(bookToBorrow, studentId);
                studentSlots.release(); // Release the student slot
                sleep(1); // Wait before the next operation
```

```
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
    }

    public static void main(String[] args) {
        Library library = new Library();

        // Create and start multiple threads for students using the library
        for (int i = 1; i <= MAX_STUDENTS; i++) {
            int finalI = i;
            new Thread(() -> library.student(finalI)).start();
        }
    }
}
```

LeetCode Multithreading Problems (HomeWork)

- Dining Philoshpers - LeetCode
- Fizz Buzz Multithreaded - LeetCode
- Building H20 - LeetCode
- Traffic Light Control

Additonal Resources

- Udemy - Concurrency, Multithreading and Parallel Computing in Java
- Memory Management in Operating Systems - Thrasing, Paging etc Interview Topics

-- End --