SUBMITTED BY: AREEBA KHAN

REG NO: FA22-BSE-008

SUBMITTED TO: SIR MUKHTIAR ZAMIN

(LAB ASSIGNMENT 2)

5 ARCHITECTURAL PROBLEMS IN SOFTWARE SYSTEMS:

- Lack of Fault Tolerance in Systems
- Poor Performance due to Synchronous Communication
- Single Point of Failure (SPOF)
- Versioning Issues in APIs
- Inefficient Resource Utilization

1. Lack of Fault Tolerance in Systems

- **Problem**:
  In early systems, when one component of the system failed (e.g., a server or a service), the entire system would stop working. For example, if a payment service went down in an e-commerce application, customers couldn't check out, resulting in lost revenue. These systems lacked fault tolerance, meaning they couldn't handle failures gracefully.
- **Solution**:
  To make systems fault-tolerant, techniques like **Load Balancers**, **Redundant Systems**, and **Circuit Breakers** were introduced. These ensure that if one component fails, another takes its place, or the system gracefully handles the failure.
- **Real Example**:
  Netflix introduced **Chaos Engineering**, a practice where they deliberately cause failures in their system to test its ability to handle issues without affecting users. For instance, if one server fails, traffic is redirected to another server seamlessly.

2. Poor Performance due to Synchronous Communication

- **Problem**:
  In systems with synchronous communication, all requests and responses occur in real-time. For example, in a food delivery app, if a "Payment Service" is slow, the entire process (placing an order) gets delayed because the system waits for the payment to finish. This creates bottlenecks during high traffic, such as festival sales.
- **Solution**:
  Asynchronous communication was introduced using **Message Queues** like RabbitMQ or Kafka. Instead of waiting for real-time responses, the system sends requests to a queue and processes them independently. This decouples services and makes the system faster.

- **Real Example**:
  Uber uses an event-driven architecture where asynchronous communication ensures that ride bookings and driver allocations happen in parallel, avoiding delays even during peak times.

3. Single Point of Failure (SPOF)

- **Problem**:
  In many traditional systems, a single failure point could bring the entire system down. For example, if a database server crashes, no user can access the application. This is called a **Single Point of Failure (SPOF)**.
- **Solution**:
  To avoid SPOF, **Replication** and **Clustering** techniques are used. In database replication, multiple copies of the database are maintained (e.g., Master-Slave Replication). If one database fails, another takes over without downtime. Clustering involves running multiple servers together to handle requests.
- **Real Example**:
  AWS (Amazon Web Services) provides database replication across multiple regions. If a primary database in one region fails, the system automatically switches to the replicated database in another region, ensuring zero downtime.

4. Versioning Issues in APIs

- **Problem**:
  When APIs (Application Programming Interfaces) are updated, older client applications often stop working because they are not compatible with the new changes. For example, if an API that fetches weather data changes its format, older mobile apps using that API may break.
- **Solution**:
  API Versioning was introduced to maintain backward compatibility. Different versions of the same API (e.g., /API/v1/weather and /API/v2/weather) are maintained so that older clients continue to work while newer features are added in updated versions.
- **Real Example**:
  Twitter introduced versioned APIs so that developers using older API versions could continue working while Twitter added new features in newer versions like v2.

5. Inefficient Resource Utilization

- **Problem**:
  Older systems running on physical servers were inefficient because they used fixed resources (CPU, RAM) even when the system wasn't fully active. For instance, a server with a large capacity might only handle a few requests, leaving most of the resources unused, leading to high costs.

- **Solution**:
  Companies migrated to **Containerized Architecture** using tools like Docker and Kubernetes. Containers allow multiple applications to run on the same server efficiently, optimizing resource usage. Kubernetes helps in automatically scaling resources up or down based on system demand.
- **Real Example**:
  Spotify moved to Kubernetes for container orchestration, enabling them to manage thousands of microservices efficiently and reduce costs significantly.

SELECTING ONE PROBLEM:

Problem: Poor Performance due to Synchronous Communication

Synchronous Processing - The Problem

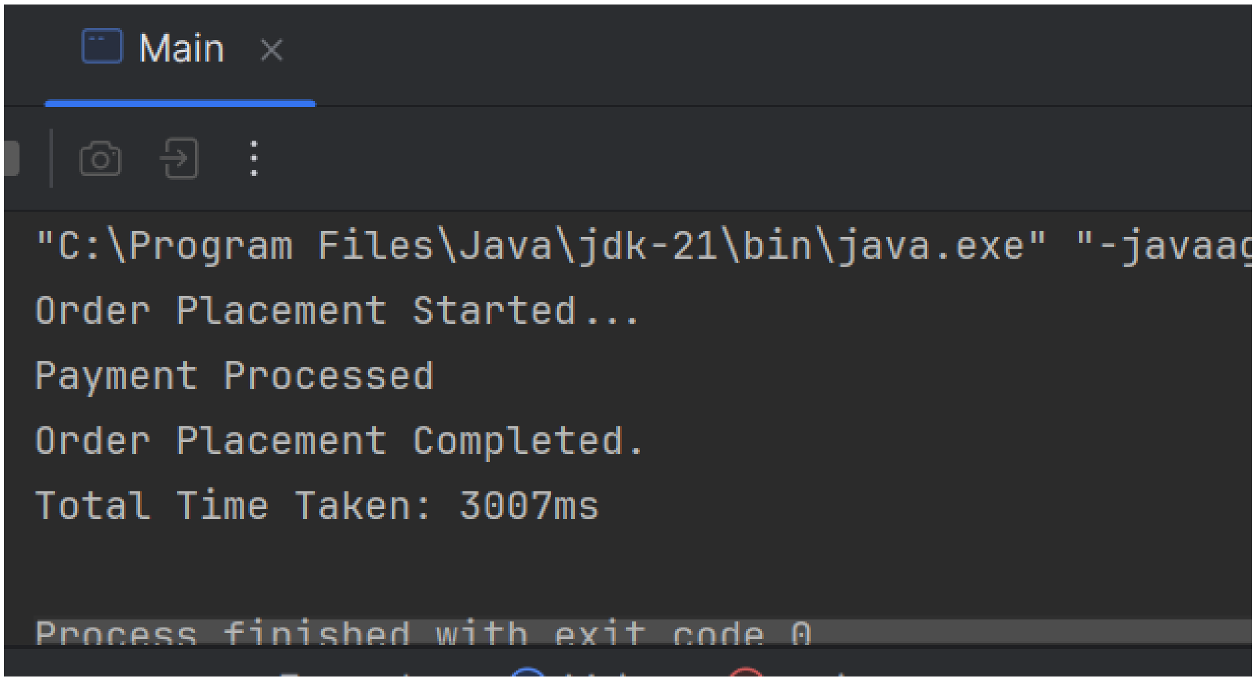
In synchronous processing, tasks are executed sequentially, meaning one task must complete before the next can begin. This can cause delays and inefficiencies when multiple tasks need to be processed simultaneously.

Let's consider a **Payment Processing** system, where we place an order, process the payment, and save the payment status to the database. In synchronous mode, the entire flow of tasks must complete before moving on to the next task.

Synchronous Example Code:

```
package Example;


import Example.DataAccess.PaymentRepository;

import Example.Service.PaymentService;

import Example.Service.OrderService;


public class Main {

    public static void main(String[] args) {

        // Initialize services and repository

        PaymentService paymentService = new PaymentService();

        PaymentRepository paymentRepository = new PaymentRepository();

        OrderService orderService = new OrderService(paymentService, paymentRepository);


        long startTime = System.currentTimeMillis();
        // Place an order
        String paymentStatus = orderService.placeOrder();

        // Display payment status and completion
        System.out.println(paymentStatus);
        System.out.println("Order Placement Completed.");

        long endTime = System.currentTimeMillis();
        System.out.println("Total Time Taken: " + (endTime - startTime) + "ms");
    }
}
```

Output



```
"C:\Program Files\Java\jdk-21\bin\java.exe" "-javaag
Order Placement Started...
Payment Processed
Order Placement Completed.
Total Time Taken: 3007ms

Process finished with exit code 0
```

Explanation of the Synchronous Process:

1. The placeOrder method in the OrderService class is executed in a linear sequence.
2. The payment processing (which might involve some time delays) must complete before the order placement process can proceed to the next step.
3. The total time taken is the cumulative time of each operation, leading to longer delays, especially when handling multiple orders.

Problem with Synchronous Processing:

- **Blocking**: The system waits for each task to complete before continuing to the next.
- **Inefficiency**: If you need to process multiple orders, each order will be handled one at a time, increasing the total time taken for the entire batch of orders.
- **Scaling Issues**: Handling a large number of orders synchronously will cause significant delays, making the system less scalable.

Asynchronous Processing - The Solution

In asynchronous processing, tasks are executed independently, allowing the system to move on to other tasks while waiting for a particular operation to complete. This reduces delays and increases efficiency, especially when multiple tasks can run in parallel.

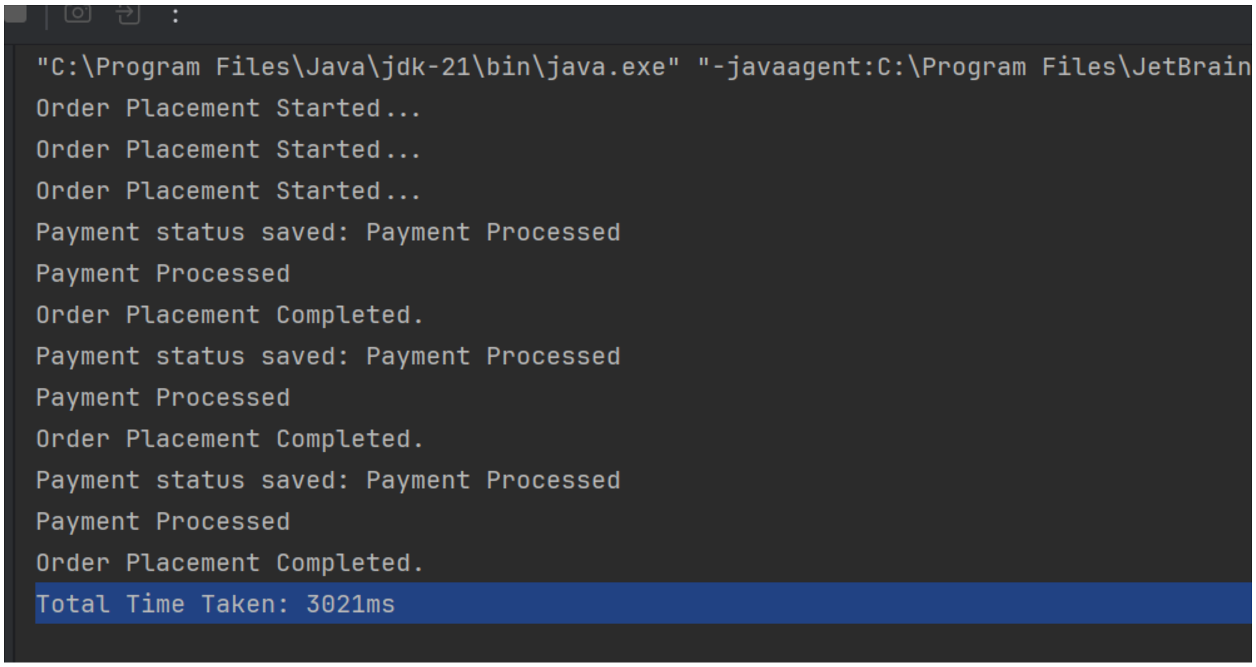
Let's modify the payment processing example to handle multiple orders concurrently using asynchronous processing.

Asynchronous Example Code:

```
package Example;


import Example.DataAccess.PaymentRepository;

import Example.Service.PaymentService;

import Example.Service.OrderService;


import java.util.concurrent.CompletableFuture;

import java.util.concurrent.ExecutionException;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;


public class Main {

  public static void main(String[] args) {

    // Initialize services and repository

    PaymentService paymentService = new PaymentService();

    PaymentRepository paymentRepository = new PaymentRepository();

    OrderService orderService = new OrderService(paymentService, paymentRepository);
```

```
        // Use ExecutorService for better thread management
        ExecutorService executorService = Executors.newFixedThreadPool(5);  // More threads for
concurrency

        long startTime = System.currentTimeMillis();

        // Simulate placing multiple orders concurrently
        CompletableFuture<Void> order1 = CompletableFuture.runAsync(() -> {
            try {
                orderService.placeOrder();
            } catch (Exception e) {
                System.out.println("Error processing payment for order 1: " + e.getMessage());
            }
        }, executorService);

        CompletableFuture<Void> order2 = CompletableFuture.runAsync(() -> {
            try {
                orderService.placeOrder();
            } catch (Exception e) {
                System.out.println("Error processing payment for order 2: " + e.getMessage());
            }
        }, executorService);

        CompletableFuture<Void> order3 = CompletableFuture.runAsync(() -> {
            try {
                orderService.placeOrder();
            } catch (Exception e) {
                System.out.println("Error processing payment for order 3: " + e.getMessage());
            }
        }, executorService);

        // Wait for all orders to be processed
        try {
            CompletableFuture.allOf(order1, order2, order3).get();  // Wait for all orders to complete
        } catch (InterruptedException | ExecutionException e) {
            System.out.println("Error waiting for the tasks to complete: " + e.getMessage());
        }

        long endTime = System.currentTimeMillis();
        System.out.println("Total Time Taken: " + (endTime - startTime) + "ms");

        // Shutdown the executor
        executorService.shutdown();
    }
}
```

Output



```
"C:\Program Files\Java\jdk-21\bin\java.exe" "-javaagent:C:\Program Files\JetBrain
Order Placement Started...
Order Placement Started...
Order Placement Started...
Payment status saved: Payment Processed
Payment Processed
Order Placement Completed.
Payment status saved: Payment Processed
Payment Processed
Order Placement Completed.
Payment status saved: Payment Processed
Payment Processed
Order Placement Completed.
Total Time Taken: 3021ms
```

Explanation of the Asynchronous Process:

1. The placeOrder method is executed asynchronously using CompletableFuture.runAsync(), which runs each order in parallel.
2. Each order is processed independently, without waiting for the other orders to complete. The ExecutorService manages multiple threads efficiently.
3. The CompletableFuture.allOf() method ensures that the program waits for all the orders to complete before finishing.

Benefits of Asynchronous Processing:

- **Non-blocking**: The system can initiate multiple tasks and continue processing without waiting for each task to finish.
- **Efficiency**: Multiple orders are processed in parallel, which reduces the overall time.
- **Scalability**: As the number of orders increases, the system can handle more concurrent tasks without significant performance degradation.