Different Problems

1) Parking lot problem-

Parking lot design in Java is a design problem that deals with how vehicles are parked in a parking lot. It is mainly asked in the HLD or LLD (High Level or Low-Level Design) round of the top multinational companies such as Amazon, Google, Facebook, etc. Note that there is no right or wrong answer to this design problem. Therefore, such type of design problem requires a good discussion with the interviewer about what kind of parking design the interviewer wants. For example, one interviewer may want a parking lot to be of 5 floors, whereas another interviewer is fine with a single-floor parking lot.

Code-

```
import java.util.*;

class ParkingLot {
   private int capacity;
   private List<Car> parkedCars;

   public ParkingLot(int capacity) {
      this.capacity = capacity;
      parkedCars = new ArrayList<>();
   }

   public boolean parkCar(Car car) {
      if (parkedCars.size() < capacity) {
         parkedCars.add(car);
         System.out.println("Car " + car.getLicensePlate()
+ " parked successfully.");
         return true;
      } else {
```

```
            System.out.println("Parking lot is full. Cannot park car " + car.getLicensePlate());
            return false;
        }
    }

    public void removeCar(String licensePlate) {
        for (Iterator<Car> iterator = parkedCars.iterator(); iterator.hasNext();) {
            Car car = iterator.next();
            if (car.getLicensePlate().equals(licensePlate)) {
                iterator.remove();
                System.out.println("Car " + licensePlate + " removed from parking lot.");
                return;
            }
        }
        System.out.println("Car with license plate " + licensePlate + " not found in parking lot.");
    }

    public void displayParkingLotStatus() {
        System.out.println("Parking Lot Status:");
        System.out.println("Capacity: " + capacity);
        System.out.println("Occupied Spaces: " + parkedCars.size());
        System.out.println("Available Spaces: " + (capacity - parkedCars.size()));
        System.out.println("List of parked cars:");
        for (Car car : parkedCars) {
            System.out.println(car);
        }
    }
}
```

```
class Car {
    private String licensePlate;
    private String color;

    public Car(String licensePlate, String color) {
        this.licensePlate = licensePlate;
        this.color = color;
    }

    public String getLicensePlate() {
        return licensePlate;
    }

    public String getColor() {
        return color;
    }

    @Override
    public String toString() {
        return "License Plate: " + licensePlate + ", Color: " +
color;
    }
}

public class Main {
    public static void main(String[] args) {
        ParkingLot parkingLot = new ParkingLot(5);

        Car car1 = new Car("ABC123", "Red");
        Car car2 = new Car("XYZ456", "Blue");
        Car car3 = new Car("DEF789", "Green");

        parkingLot.parkCar(car1);
        parkingLot.parkCar(car2);
        parkingLot.parkCar(car3);
```

```
        parkingLot.displayParkingLotStatus();

        parkingLot.removeCar("XYZ456");

        parkingLot.displayParkingLotStatus();
    }
}
```

2) Sleeping barber-

The Sleeping Barber problem is a classic problem in process synchronization that is used to illustrate synchronization issues that can arise in a concurrent system. The problem is as follows:

There is a barber shop with one barber and several chairs for waiting customers. Customers arrive at random times and if there is an available chair, they take a seat and wait for the barber to become available. If there are no chairs available, the customer leaves. When the barber finishes with a customer, he checks if there are any waiting customers. If there are, he begins cutting the hair of the next customer in the queue. If there are no customers waiting, he goes to sleep.

The problem is to write a program that coordinates the actions of the customers and the barber in a way that avoids synchronization problems, such as deadlock or starvation.

One solution to the Sleeping Barber problem is to use semaphores to coordinate access to the waiting chairs and the barber chair. The solution involves the following steps:

Initialize two semaphores: one for the number of waiting chairs and one for the barber chair. The waiting chairs semaphore is initialized to the number of chairs, and the barber chair semaphore is initialized to zero.

Customers should acquire the waiting chairs semaphore before taking a seat in the waiting room. If there are no available chairs, they should leave.

When the barber finishes cutting a customer's hair, he releases the barber chair semaphore and checks if there are any waiting customers. If there are, he acquires the barber chair semaphore and begins cutting the hair of the next customer in the queue.

The barber should wait on the barber chair semaphore if there are no customers waiting.

The solution ensures that the barber never cuts the hair of more than one customer at a time, and that customers wait if the barber is busy. It also ensures that the barber goes to sleep if there are no customers waiting.

However, there are variations of the problem that can require more complex synchronization mechanisms to avoid synchronization issues. For example, if multiple barbers are employed, a more complex mechanism may be needed to ensure that they do not interfere with each other.

Prerequisite – Inter Process Communication Problem : The analogy is based upon a hypothetical barber shop with one barber. There is a barber shop which has one barber, one barber chair, and n chairs for waiting for customers if there are any to sit on the chair.

If there is no customer, then the barber sleeps in his own chair.

When a customer arrives, he has to wake up the barber.

If there are many customers and the barber is cutting a customer's hair, then the remaining customers either wait if there are empty chairs in the waiting room or they leave if no chairs are empty.

Solution-

The solution to this problem includes three semaphores. First is for the customer which counts the number of customers present in the waiting room (customer in the barber chair is not included because he is not waiting). Second, the barber 0 or 1 is used to tell whether the barber is idle or is working, And the third mutex is used to provide the mutual exclusion that is required for the process to execute. In the solution, the customer has the record of the number of customers waiting in the waiting room if the number of customers is equal to the number of chairs in the waiting room then the upcoming customer leaves the barbershop. When the barber shows up in the morning, he executes the procedure barber, causing him to block on the semaphore customers because it is initially 0. Then the barber goes to sleep until the first customer comes up. When a customer arrives, he executes customer procedure the customer acquires the mutex for entering the critical region,

if another customer enters thereafter, the second one will not be able to anything until the first one has released the mutex. The customer then checks the chairs in the waiting room if waiting customers are less then the number of chairs then he sits otherwise he leaves and releases the mutex. If the chair is available then customer sits in the waiting room and increments the variable waiting value and also increases the customer's semaphore this wakes up the barber if he is sleeping. At this point, customer and barber are both awake and the barber is ready to give that person a haircut. When the haircut is over, the customer exits the procedure and if there are no customers in waiting room barber sleeps.

Code-

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


class BarberShop {
   private Semaphore availableSeats;
   private Semaphore barberReady;
   private Semaphore customerReady;


   public BarberShop(int numSeats) {
      availableSeats = new Semaphore(numSeats);
      barberReady = new Semaphore(0);
      customerReady = new Semaphore(0);
   }
```

```
    public void customerWalksIn() throws InterruptedException {
        if (availableSeats.tryAcquire()) {
            System.out.println("Customer walks in and sits down.");
            barberReady.release();
            customerReady.acquire();
            System.out.println("Customer got haircut and leaves.");
            availableSeats.release();
        } else {
            System.out.println("No available seats, customer leaves.");
        }
    }

    public void barberCutsHair() throws InterruptedException {
        while (true) {
            barberReady.acquire();
            System.out.println("Barber starts cutting hair.");
            Thread.sleep(2000); // Simulating haircut
            customerReady.release();
        }
    }
}
```

```
class Barber extends Thread {
    private BarberShop barberShop;

    public Barber(BarberShop barberShop) {
        this.barberShop = barberShop;
    }

    @Override
    public void run() {
        try {
            barberShop.barberCutsHair();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

class Customer extends Thread {
    private BarberShop barberShop;

    public Customer(BarberShop barberShop) {
        this.barberShop = barberShop;
    }
```

```
    @Override
    public void run() {
        try {
            barberShop.customerWalksIn();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

public class Main {
    public static void main(String[] args) {
        BarberShop barberShop = new BarberShop(3); // 3 seats in the waiting area

        Barber barber = new Barber(barberShop);
        barber.start();

        for (int i = 0; i < 5; i++) { // 5 customers arriving
            Customer customer = new Customer(barberShop);
            customer.start();
            try {
                Thread.sleep(1000); // Customers arrive with 1 second interval
            } catch (InterruptedException e) {
```

```
            e.printStackTrace();
          }
        }
      }
}
```

3) Diner Philosopher problem-

Dining Philosophers Problem States that 5 Philosophers are engaged in two activities Thinking and Eating. Meals are taken communally on a table with five plates and five forks in a cyclic manner as shown in the figure.

Solution :

Correctness properties it needs to satisfy are :

- Mutual Exclusion Principle –
- No two Philosophers can have the two forks simultaneously.
- Free from Deadlock –
- Each philosopher can get the chance to eat in a certain finite time.
- Free from Starvation –When few Philosophers are waiting then one gets a chance to eat in a while.
- No strict Alternation.

- Proper utilization of time.

Code-

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

class DiningPhilosophers {
   private Semaphore[] forks;
   private Semaphore mutex;

   public DiningPhilosophers(int numPhilosophers) {
      forks = new Semaphore[numPhilosophers];
      for (int i = 0; i < numPhilosophers; i++) {
         forks[i] = new Semaphore(1);
      }
      mutex = new Semaphore(1);
   }

   public void pickUpForks(int philosopherId) throws InterruptedException {
      mutex.acquire();
      forks[philosopherId].acquire();
      forks[(philosopherId + 1) % forks.length].acquire();
      mutex.release();
   }
```

```
    public void putDownForks(int philosopherId) throws InterruptedException {
        forks[philosopherId].release();
        forks[(philosopherId + 1) % forks.length].release();
    }
}

class Philosopher extends Thread {
    private int id;
    private DiningPhilosophers diningPhilosophers;

    public Philosopher(int id, DiningPhilosophers diningPhilosophers) {
        this.id = id;
        this.diningPhilosophers = diningPhilosophers;
    }

    @Override
    public void run() {
        try {
            while (true) {
                System.out.println("Philosopher " + id + " is thinking.");
                Thread.sleep(1000);
                System.out.println("Philosopher " + id + " is hungry.");
```

```
            diningPhilosophers.pickUpForks(id);
            System.out.println("Philosopher " + id + " is eating.");
            Thread.sleep(1000);
            diningPhilosophers.putDownForks(id);
          }
        } catch (InterruptedException e) {
          e.printStackTrace();
        }
    }
}

public class Main {
    public static void main(String[] args) {
        int numPhilosophers = 5;
        DiningPhilosophers diningPhilosophers = new
DiningPhilosophers(numPhilosophers);

        Philosopher[] philosophers = new
Philosopher[numPhilosophers];
        for (int i = 0; i < numPhilosophers; i++) {
          philosophers[i] = new Philosopher(i,
diningPhilosophers);
          philosophers[i].start();
        }
    }
}
```

4) Producer consumer problem-

In computing, the producer-consumer problem (also known as the bounded-buffer problem) is a classic example of a multi-process synchronization problem. The problem describes two processes, the producer and the consumer, which share a common, fixed-size buffer used as a queue.

- The producer's job is to generate data, put it into the buffer, and start again.
- At the same time, the consumer is consuming the data (i.e. removing it from the buffer), one piece at a time.

Problem:

To make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.

Solution

The producer is to either go to sleep or discard data if the buffer is full. The next time the consumer removes an item from the buffer, it notifies the producer, who starts to fill the buffer again. In the same way, the consumer can go to sleep if it finds the buffer to be empty. The next time the producer puts data into the buffer, it wakes up the sleeping consumer.

An inadequate solution could result in a deadlock where both processes are waiting to be awakened.

Code-

```
import java.util.LinkedList;
```

```
import java.util.Queue;

class Buffer {
   private Queue<Integer> queue;
   private int capacity;

   public Buffer(int capacity) {
      this.capacity = capacity;
      queue = new LinkedList<>();
   }

   public synchronized void produce(int item) throws InterruptedException {
      while (queue.size() == capacity) {
         wait(); // Buffer is full, wait for consumer to consume
      }
      queue.add(item);
      System.out.println("Produced: " + item);
      notify(); // Notify consumer that a new item is available
   }

   public synchronized int consume() throws InterruptedException {
      while (queue.isEmpty()) {
```

```
            wait(); // Buffer is empty, wait for producer to produce
        }
        int item = queue.poll();
        System.out.println("Consumed: " + item);
        notify(); // Notify producer that space is available
        return item;
    }
}

class Producer extends Thread {
    private Buffer buffer;

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

    @Override
    public void run() {
        try {
            for (int i = 0; i < 5; i++) { // Producing 5 items
                buffer.produce(i);
                Thread.sleep(1000); // Producer waits for 1 second between producing items
            }
```

```
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

class Consumer extends Thread {
    private Buffer buffer;

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

    @Override
    public void run() {
        try {
            for (int i = 0; i < 5; i++) { // Consuming 5 items
                buffer.consume();
                Thread.sleep(2000); // Consumer waits for 2 seconds between consuming items
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
```

```
}

public class Main {
    public static void main(String[] args) {
        Buffer buffer = new Buffer(2); // Buffer size is 2

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

        producer.start();
        consumer.start();
    }
}
```

5) Deadlock-

Synchronized keyword is used to make the class or method thread-safe which means only one thread can have lock of synchronized method and use it, other threads have to wait till the lock releases and anyone of them acquire that lock.

It is important to use if our program is running in multi-threaded environment where two or more threads execute simultaneously. But sometimes it also causes a problem which is called Deadlock. Below is a simple example of Deadlock condition.

Code-

```
public class DeadlockExample {
    public static void main(String[] args) {
```

```
        // Resources
        Object resource1 = new Object();
        Object resource2 = new Object();

        // Thread 1
        Thread thread1 = new Thread(() -> {
            synchronized (resource1) {
                System.out.println("Thread 1 acquired resource 1");

                try {
                    Thread.sleep(100); // Introducing delay to increase chances of deadlock
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }

                synchronized (resource2) {
                    System.out.println("Thread 1 acquired resource 2");
                }
            }
        });

        // Thread 2
        Thread thread2 = new Thread(() -> {
            synchronized (resource2) {
                System.out.println("Thread 2 acquired resource 2");

                try {
                    Thread.sleep(100); // Introducing delay to increase chances of deadlock
                } catch (InterruptedException e) {
                    e.printStackTrace();
```

```
            }

            synchronized (resource1) {
                System.out.println("Thread 2 acquired
resource 1");
            }
        }
    });

    // Start threads
    thread1.start();
    thread2.start();
  }
}
```