**You have a platform wherein you have some 'prime' users and others are normal users. You have to design a system which can prioritize the orders from the 'prime' users but should not altogether avoid the orders from the normal users.**

Let’s break it down step by step so it’s clear how this system works, especially focusing on **why we use two queues**, **how we maintain fairness**, and **how orders are processed efficiently**:

**🎯 Problem**

You have two types of users:

* **Prime users** → Their orders should be **processed faster**.
* **Normal users** → Should **not be ignored**, but get **less priority**.

So, we need a system that:

* **Prioritizes prime users**
* **Still serves normal users**
* **Handles many orders smoothly**

**💡 Solution Overview**

We’ll create a **two-queue system**:

* One queue for **prime users**
* Another queue for **normal users**

Then we use a **dispatcher** that picks orders using a **ratio**, like:

* **Process 3 prime orders**
* **Then process 1 normal order**
* Repeat…

This makes sure prime users are faster, but **normal users are not blocked** forever.

**🧱 System Parts**

**1. Order API**

A REST or gRPC service receives orders:

* When a user places an order, we check:
  + Are they a prime user?
* Then, add the order to the right queue.

**2. Order Queues**

You maintain:

Queue<Order> primeQueue;

Queue<Order> normalQueue;

These are thread-safe queues where incoming orders go.

**3. Dispatcher Logic**

This decides which order to give to workers for processing.

Example logic:

int primeCount = 0;

int PRIME\_LIMIT = 3;

Order getNextOrder() {

if (primeCount < PRIME\_LIMIT && !primeQueue.isEmpty()) {

primeCount++;

return primeQueue.poll();

} else if (!normalQueue.isEmpty()) {

primeCount = 0; // reset after normal order

return normalQueue.poll();

} else if (!primeQueue.isEmpty()) {

return primeQueue.poll(); // if normal queue is empty

} else {

return null;

}

}

This means:

* It will serve 3 prime orders.
* Then serve 1 normal order.
* Then repeat.

So **prime users get more attention**, but **normal users are never ignored**.

**⚙️ Workers**

You use a **thread pool** (like Java’s ExecutorService):

* Each worker thread will keep picking orders using getNextOrder()
* This way, you can process orders **in parallel** and fast.

**🗃️ Database**

Store order info like:

* order\_id
* user\_id
* user\_type
* status (pending, completed)
* timestamp

This helps in:

* Tracking orders
* Reprocessing failed ones
* Auditing

**📈 Scalability**

If you want to scale this to handle **millions of orders**:

* Use **Kafka or RabbitMQ** as your queue backend.
* Run multiple **dispatcher-worker** services across machines.
* Deploy components in **microservices architecture**.

**✅ Fairness Explained**

This logic is **fair** because:

* Prime users are prioritized.
* But normal users also get picked regularly.
* If you only used one queue and sorted by priority, normal users might **never get served** (called *starvation*). We avoid that.

**👨‍🏫 Analogy**

Imagine two lines in a shop:

* VIP (prime) line
* Regular (normal) line

You always serve 3 VIPs, then serve 1 regular customer.  
This makes both happy:

* VIPs get faster service
* Regulars don’t feel ignored

**🧵 Worker / Consumer Logic**

**✅ Purpose:**

* Repeatedly **poll the next order** from the dispatcher.
* **Process** it (simulate some work like saving to DB, sending email).
* **Handle errors** if processing fails.

**💻 Java-like Pseudocode for Consumer Logic**

public class OrderConsumer implements Runnable {

private OrderDispatcher dispatcher;

public OrderConsumer(OrderDispatcher dispatcher) {

this.dispatcher = dispatcher;

}

@Override

public void run() {

while (true) {

Order order = dispatcher.getNextOrder();

if (order != null) {

try {

processOrder(order); // main processing logic

System.out.println("Processed order: " + order.getId());

} catch (Exception e) {

System.err.println("Failed to process order: " + order.getId());

// optionally retry or log to dead-letter queue

}

} else {

try {

Thread.sleep(100); // wait before trying again to avoid tight loop

} catch (InterruptedException e) {

Thread.currentThread().interrupt();

}

}

}

}

private void processOrder(Order order) {

// simulate order processing

// example: save to database, call payment API, etc.

}

}

**🧰 Dispatcher Logic (as used by Consumers)**

public class OrderDispatcher {

private Queue<Order> primeQueue;

private Queue<Order> normalQueue;

private int primeCounter = 0;

private final int PRIME\_LIMIT = 3;

public synchronized Order getNextOrder() {

if (primeCounter < PRIME\_LIMIT && !primeQueue.isEmpty()) {

primeCounter++;

return primeQueue.poll();

} else if (!normalQueue.isEmpty()) {

primeCounter = 0; // reset after normal

return normalQueue.poll();

} else if (!primeQueue.isEmpty()) {

return primeQueue.poll();

} else {

return null;

}

}

}

**🧵 Spawning Consumers (with Thread Pool)**

java

CopyEdit

public class Main {

public static void main(String[] args) {

OrderDispatcher dispatcher = new OrderDispatcher();

ExecutorService executor = Executors.newFixedThreadPool(5); // 5 workers

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

executor.submit(new OrderConsumer(dispatcher));

}

// Simulate order producers adding to queues

// dispatcher.addPrimeOrder(order) / dispatcher.addNormalOrder(order)

}

}

**🔁 Flow Recap**

1. Orders come in (producer) and go to appropriate queue.
2. Dispatcher picks next order based on 3:1 prime:normal ratio.
3. Each worker (consumer) thread:
   * Calls getNextOrder()
   * Processes the order
   * Waits briefly if nothing is available