문제 해결 보고서

동시성 문제 해결 보고서

1. 현황 분석 (AS-IS)

1.1 현재 시스템 구조

1.1.1 주문 및 결제 서비스

현재 OrderService는 주문부터 결제까지 모든 프로세스를 하나의 @Transactional 어노테이션으로 처리하고 있습니다.

```
@Transactional
public ResponseOrder orderProduct(RequestOrder requestOrder) {
    // 1. 사용자 조회 및 잔고확인
    User user = userService.getUserAndCheckBalance(requestOrder.userId
(), requestOrder.requestPrice(), status);

    // 2. 포인트 차감
    user.usePoint(requestOrder.requestPrice());

    // 3. 상품 재고 차감
    Product product = productService.getProductInfo(requestOrder.productI
d());
    product.decreaseStock(requestOrder.requestQuantity());

    // 4. 주문 생성
    Order order = new Order(user, product, ...);
    Order returnOrder = orderRepository.save(order);

    // 5. 결제 처리
    paymentService.paymentProduct(requestOrder, user, product, returnOrd
er);

    return responseOrder;
}
```

1.1.2 쿠폰 발급 서비스

```
@Transactional
    public ResponseUserCoupon getCoupon(RequestUserCoupon requestU
serCoupon) {

        // 1. 발급하려는 쿠폰을 hist에서 사용자가 받았는지 조회
        Boolean couponYn = couponHistService.getCouponHist(requestUserC
oupon);

        if(couponYn) {
            throw new CustomException("쿠폰을 이미 발급받았음");
        }



        // 쿠폰발급
        Coupon coupon = couponRepository.findById(requestUserCoupon.co
uponId())
                .orElseThrow(() -> new CustomException("쿠폰을 찾을 수 없음"));

        try {
            // 쿠폰수량 감소
            coupon.issueCoupon();

            couponRepository.save(coupon);
        } catch (CustomException e) {
            throw e;
        } catch (ObjectOptimisticLockingFailureException e) {
            throw new CustomException("쿠폰 발급에 실패했습니다. (동시성 충돌 발
생)");
        } catch (Exception e) {
            throw new CustomException("알 수 없는 오류 발생");
        }

        // 쿠폰이력추가
        CouponHist couponHist = couponHistService.addCouponHist(request
UserCoupon, coupon);
```

```
        ResponseUserCoupon responseUserCoupon = new ResponseUserCo
upon(
                couponHist.getUserId(),
                couponHist.getCouponId(),
                couponHist.getProductId()
        );

        return responseUserCoupon;
    }
```

1.2 식별된 문제점

1.2.1 과도한 트랜잭션 범위

- **문제**: 사용자 조회, 포인트 차감, 재고 관리, 주문 생성, 결제 처리가 모두 하나의 트랜잭션에 포함
- **영향**: 트랜잭션 지속 시간이 길어져 Lock 경합 가능성 증가

1.2.2 동시성 처리 부재

- **재고 관리**: 동시 주문 시 재고 부족 상황에서 race condition 발생 가능
- **포인트 차감**: 동일 사용자의 동시 주문 시 잔고 부족 검증 실패 가능

1.2.3 시스템 확장성 제약

- **단일 장애점**: 결제 서비스 장애 시 전체 주문 프로세스 중단
- **리소스 낭비**: 긴 트랜잭션으로 인한 DB 커넥션 점유 시간 증가

2. 해결 방안 (TO-BE)

2.1 트랜잭션 분리 전략

2.1.1 핵심 트랜잭션과 부가 처리 분리

OrderFacade (트랜잭션 분리)

```
@Service
public class OrderFacade {

    @Transactional
```

```
    public ResponseOrder processOrder(RequestOrder request) {

        // 주문 생성
        Order order = createOrderCore(request);

        // 비동기 후속 처리 이벤트 발행
        orderEventPublisher.publishOrderCreated(OrderCreatedEvent.of(orde
r));

        return ResponseOrder.from(order);
    }
}
```

OrderEventPublisher (이벤트 발행자)

```
@Component
public class OrderEventPublisher {

    public void publishOrderCreated(OrderCreatedEvent event) {
        eventPublisher.publishEvent(event);
    }

    public void publishProductUpdated(ProductUpdatedEvent event) {
        eventPublisher.publishEvent(event);
    }

    public void publishPointDeducted(PointDeductedEvent event) {
        eventPublisher.publishEvent(event);
    }

    public void publishPaymentCompleted(PaymentCompletedEvent event) {
        eventPublisher.publishEvent(event);
    }
}
```

OrderCreatedEvent (이벤트 객체)

```
public class OrderCreatedEvent {

    private final Long orderId;
    private final Long userId;
    ...
    private OrderCreatedEvent(Long orderId, Long userId,
    Long productId, int requestQuantity, int requestPrice) {
        this.orderId = orderId;
        this.userId = userId;
        ...
    }

    public static OrderCreatedEvent of(Order order) {
        return new OrderCreatedEvent(
            order.getId(),
            order.getUser().getId(),
            ...
        );
    }
}
```

2.1.2 이벤트 기반 비동기 처리

OrderEventHandler (비동기 이벤트 처리)

```
@Component
public class OrderEventHandler {



    @EventListener
    @Async("orderTaskExecutor")
    @Transactional
    public void handlePointDeduction(OrderCreatedEvent event) {
        try {
            // 실제 포인트 차감 (동시성 제어 적용)
            userService.deductPointsWithLock(
                event.getUserId(),
                event.getRequestPrice()
```

```
            );

            // 다음 단계 이벤트 발행
            orderEventPublisher.publishPaymentReady(
                PaymentReadyEvent.of(event)
            );

        } catch (InsufficientPointException e) {
            // 포인트 부족 시 재고 롤백 및 주문 취소
            inventoryService.increaseStock(event.getProductId(), event.getReq
uestQuantity());
            orderService.cancelOrder(event.getOrderId(), "포인트 부족");
        } catch (Exception e) {
            handlePointError(event, e);
        }
    }

    @EventListener
    @Async("orderTaskExecutor")
    @Transactional
    public void handlePayment(PointDeductedEvent event) {
        try {

            Payment result = paymentService.processPayment(
                    event.getOrderId(),
                    event.getRequestPrice()
            );

            if ("01".equals(result.getStatus())) {
                // 주문 완료 처리
                orderService.completeOrder(event.getOrderId());

                pointHistService.createPointHist(
                        userService.getUserInfo(event.getUserId(), "01"),
                        TransactionType.USE,
                        event.getRequestPrice(),
                        userService.getUserInfo(event.getUserId(), "01").getPoint(),
                        result.getId()  // 결제 ID
```

```
            );

            orderEventPublisher.publishPaymentCompleted(
                    PaymentCompletedEvent.of(event, result.getId(),true)
            );
        } else {
            // 결제 실패 시 보상 트랜잭션
            handlePaymentFailure(event);
        }

    } catch (Exception e) {
        throw new CustomException("결제실패");
    }
  }

}
```

비동기 처리를 위한 ThreadPoolTaskExecutor 설정

```
@Configuration
@EnableAsync
public class AsyncConfig {

  @Bean(name = "orderTaskExecutor")
  public ThreadPoolTaskExecutor orderTaskExecutor() {
    ThreadPoolTaskExecutor executor =
      new ThreadPoolTaskExecutor();
    executor.setCorePoolSize(10);       // 기본 스레드 수
    executor.setMaxPoolSize(50);        // 최대 스레드 수
    executor.setQueueCapacity(100);     // 큐 용량
    executor.setThreadNamePrefix("Order-");
    executor.setRejectedExecutionHandler(new ThreadPoolExecutor.Caller
RunsPolicy());
    executor.initialize();
    return executor;
  }
}
```

추가 이벤트 객체들

```
}

public class PointDeductedEvent {
    ...
}

public class PaymentCompletedEvent {
        ...
}
```

2.2 동시성 제어 방안

2.2.1 비관적 락 (Pessimistic Lock) 적용(결제서비스)

```
@Entity
public class Product {

    @Lock(LockModeType.PESSIMISTIC_WRITE)
    @Query("UPDATE Product p SET p.stock = p.stock - :quantity WHERE p.i
d = :productId AND p.stock >= :quantity")
    public int decreaseStockWithLock(@Param("productId") Long productId,
@Param("quantity") int quantity);
}

@Entity
public class User {

    @Lock(LockModeType.PESSIMISTIC_WRITE)
    @Query("UPDATE User u SET u.point = u.point - :amount WHERE u.id = :
userId AND u.point >= :amount")
    public int deductPointWithLock(@Param("userId") Long userId, @Param
("amount") int amount);
}
```

2.2.2 낙관적 락 (Optimistic Lock) 활용(쿠폰서비스)

```
public class Coupon {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "COUPON_ID")
    private Long id;
        ...
        // version 추가
    @Version
    private Long version = 0L;
}
```

2.2.3 트랜잭션 타임아웃 설정

```
@Transactional(timeout = 30) // 30초 타임아웃
public ResponseOrder processOrder(RequestOrder request) {

}
```

3. 기대 효과

확장성 확보

- **서비스 분리**: 각 도메인별 독립적 확장 가능
- **장애 격리**: 결제 장애가 주문 생성에 미치는 영향 최소화
- **유지보수성**: 각 컴포넌트별 독립적 배포 및 테스트 가능

데이터 정합성 관리

- **보상 트랜잭션**: 각 단계별 롤백 로직 구현