

예약 처리 프로세스와 데이터 모델에서 발생하고 있는 동시성 이슈의 해결 방안 제시

문1) 다수의 사용자가 다수의 인벤토리 예약을 하기 위한 데이터 모델의 문제점을 확인하고, 해결을 위한 방안을 제시하시오.

<제약 조건> 예약 처리 프로세스는 Async 한 방식으로 구현 되어야 한다

문2) 다수의 사용자가 하나의 인벤토리 예약을 하려고 한다, 데이터 모델의 문제점을 확인하고 해결을 위한 방안을 제시하시오.

<제약 조건> 예약 처리 프로세스는 Async 한 방식으로 구현 되어야 한다

문3) 예약된 서비스에 대해 취소 프로세스를 구성하시오

답안 표시(제미)

예약 처리 프로세스 및 데이터 모델 동시성 이슈 해결 방안

1. 다수의 사용자가 다수의 인벤토리 예약 시 데이터 모델 문제점 및 해결 방안

문제점:

- **동시 접근:** 다수의 사용자가 동시에 여러 인벤토리를 예약하려고 할 때, 데이터베이스에 동시에 접근하여 데이터 불일치가 발생할 수 있습니다.
- **재고 관리:** 예약 처리 과정에서 재고가 정확하게 차감되지 않아 실제 재고와 시스템에 표시되는 재고가 다를 수 있습니다.

해결 방안:

- **Async 처리:**
 - **Message Queue:** 예약 요청을 메시지 큐(RabbitMQ, Azure Service Bus 등)에 넣어 비동기적으로 처리합니다.
 - **Worker:** 메시지 큐에서 작업을 꺼내 처리하는 Worker를 여러 개 운영하여 처리량을 높입니다.
- **낙관적 락:**
 - 각 예약 요청에 버전 정보를 포함하여 데이터베이스 업데이트 시 버전을 비교합니다.
 - 버전이 일치하지 않으면 재시도하여 데이터 일관성을 유지합니다.
- **분산 락:**
 - Redis와 같은 In-Memory 데이터베이스를 활용하여 분산 락을 구현합니다.

- 특정 인벤토리에 대한 예약 처리 시 락을 걸어 다른 요청이 동시에 처리되지 않도록 합니다.

데이터 모델:

- **인벤토리 테이블:** 인벤토리 ID, 현재 재고, 버전
- **예약 테이블:** 예약 ID, 사용자 ID, 인벤토리 ID, 예약 시간, 상태(예약, 취소 등), 버전

예시 코드 (Python, Flask, SQLAlchemy, Redis):

Python

예약 요청 처리 함수

```
def create_reservation(inventory_id, user_id):
```

```
    # Redis에서 락 획득
```

```
    with redis.lock(f'inventory:{inventory_id}')
```

```
        # 데이터베이스에서 인벤토리 정보 조회
```

```
        inventory = Inventory.query.get(inventory_id)
```

```
        # 버전 확인 및 업데이트
```

```
        if inventory.version == request.json['version']:
```

```
            # 재고 차감 및 예약 정보 저장
```

```
            inventory.quantity -= 1
```

```
            db.session.commit()
```

```
            # 메시지 큐에 예약 처리 메시지 전송
```

```
            message_queue.send({'inventory_id': inventory_id, 'user_id': user_id})
```

```
        else:
```

```
            # 버전 불일치: 재시도
```

```
            return jsonify({'error': 'Concurrent modification'}), 409
```

코드를 사용할 때는 [주의](#)가 필요합니다.

2. 다수의 사용자가 하나의 인벤토리 예약 시 데이터 모델 문제점 및 해결 방안

문제점:

- **경합 조건:** 여러 사용자가 동일한 인벤토리를 동시에 예약하려고 할 때, 먼저 요청한 사용자만 예약에 성공해야 합니다.

해결 방안:

- **비관적 락:**
 - 데이터베이스 레벨에서 락을 걸어 다른 사용자의 접근을 차단합니다.
 - 하지만 성능 저하와 데드락 발생 가능성이 있습니다.
- **분산 락:**
 - Redis와 같은 In-Memory 데이터베이스를 활용하여 분산 락을 구현합니다.
- **Optimistic Concurrency Control (OCC):**
 - 낙관적 락과 유사하지만, 충돌 발생 시 재시도 로직을 더욱 강화합니다.

추가 고려 사항:

- **장바구니 시스템:** 장바구니에 담긴 상품을 한 번에 예약 처리하는 경우, 장바구니 자체에 락을 걸거나, 각 상품에 대한 락을 관리해야 합니다.
- **재고 관리:** 실시간 재고를 유지하기 위해 캐싱을 활용할 수 있지만, 일관성 문제가 발생할 수 있으므로 주의해야 합니다.

3. 예약 취소 프로세스

- **취소 요청:** 사용자가 예약 취소를 요청합니다.
- **취소 처리:**
 - 메시지 큐를 통해 취소 요청을 처리합니다.
 - 데이터베이스에서 예약 상태를 '취소'로 변경하고, 재고를 복원합니다.
- **환불 처리:** 결제 시스템과 연동하여 환불을 처리합니다.

데이터 모델:

- **예약 테이블에 상태 컬럼 추가:** 예약, 취소, 완료 등 상태를 관리합니다.
- **취소 이력 테이블:** 취소된 예약 정보를 별도로 저장합니다.

동시성 문제 해결:

- 위에서 설명한 낙관적 락, 분산 락 등을 적용하여 취소 처리 시 데이터 일관성을 유지합니다.

결론: 예약 시스템의 동시성 문제 해결을 위해서는 Async 처리, 낙관적 락, 분산 락, 메시지 큐 등 다양한 기법을 적절히 조합해야 합니다. 시스템의 특성과 요구사항에 맞는 최적의 해결 방안을 선택하고, 지속적인 모니터링과 개선을 통해 시스템의 안정성과 성능을 확보해야 합니다.

참고: 위 내용은 일반적인 예시이며, 실제 시스템에 적용하기 위해서는 시스템의 특징과 요구사항에 맞게 수정해야 합니다.

예약 처리 프로세스와 데이터 모델에서 발생하고 있는 동시성 이슈의 해결 방안 제시

문1) 다수의 사용자가 다수의 인벤토리 예약을 하기 위한 데이터 모델의 문제점을 확인하고, 해결을 위한 방안을 제시하시오.

<제약 조건>예약 처리 프로세스는 Async 한 방식으로 구현 되어야 한다

문2) 다수의 사용자가 하나의 인벤토리 예약을 하려고 한다, 데이터 모델의 문제점을 확인하고 해결을 위한방안을 제시하시오.

<제약 조건>예약 처리 프로세스는 Async 한 방식으로 구현 되어야 한다

문3) 예약된 서비스에 대해 취소 프로세스를 구성하시오

➔ 코파

예약 처리 프로세스 및 데이터 모델 동시성 이슈 해결 방안

1. 다수의 사용자가 다수의 인벤토리 예약 시 데이터 모델 문제점 및 해결 방안

문제점:

동시성 문제: 여러 사용자가 동시에 동일한 인벤토리를 예약하려 할 때, 실제 재고보다 더 많은 예약이 발생할 수 있는 문제.

데이터 일관성 문제: 예약 정보가 정확하게 반영되지 않아, 실제 재고와 예약 정보가 불일치할 수 있음.

Race Condition: 여러 스레드가 동시에 공유 자원(인벤토리)에 접근하여 예기치 않은 결과를 초래할 수 있음.

해결 방안:

낙관적 락(Optimistic Locking):

각 행에 version 필드를 추가하여 버전 정보를 관리

업데이트 시 버전을 비교하여 충돌 여부 확인

충돌 발생 시 재시도

비관적 락(Pessimistic Locking):

데이터 수정 전에 락을 걸어 다른 사용자의 접근을 차단

성능 저하 가능성

분산 락(Distributed Lock):

Redis, Zookeeper 등 분산 락 시스템 활용

여러 노드에서 동일한 자원에 대한 락을 관리

데이터베이스 트랜잭션:

하나의 논리적인 작업 단위로 묶어서 처리

성공하면 모두 반영, 실패하면 모두 취소

Async 처리:

메시지 큐(RabbitMQ, Kafka 등)를 이용하여 비동기적으로 처리

처리 속도 향상 및 시스템 부하 분산

데이터 모델:

Inventory 테이블:

id: Primary Key

quantity: 재고 수량

version: 낙관적 락을 위한 버전 정보

Order 테이블:

id: Primary Key

inventory_id: 외래키 (Inventory 테이블)

user_id: 외래키 (User 테이블)

version: 낙관적 락을 위한 버전 정보

status: 주문 상태 (예약, 결제 완료, 취소 등)

예시 코드 (Python, SQLAlchemy):

Python

```
from sqlalchemy import Column, Integer, ForeignKey, select
```

```
from sqlalchemy.orm import sessionmaker
```

```
# ... (생략)
```

```
async def create_order(session, inventory_id, user_id):
```

```
    inventory = await session.get(Inventory, inventory_id)
```

```
    inventory.version += 1
```

```
    try:
```

```
        new_order = Order(inventory=inventory, user=user, version=inventory.version)
```

```
        session.add(new_order)
```

```
        await session.commit()
```

```
    except SQLAlchemyError as e:
```

```
        # 낙관적 락 실패 시 재시도 로직
```

```
        print(f"Error creating order: {e}")
```

```
        await session.rollback()
```

```
        # 재시도 로직 구현
```

코드를 사용할 때는 주의가 필요합니다.

2. 다수의 사용자가 하나의 인벤토리 예약 시 데이터 모델 문제점 및 해결 방안

문제점: 위에서 언급한 문제와 동일

해결 방안: 위에서 제시한 해결 방안을 동일하게 적용

3. 예약된 서비스 취소 프로세스

취소 요청: 사용자가 예약 취소를 요청

취소 검증: 취소 가능 여부 확인 (예: 취소 마감 시간, 예약 상태 등)

재고 복원: 취소된 예약에 해당하는 재고를 복원

결제 취소: 결제가 완료된 경우 결제 취소 처리

취소 기록: 취소 내역을 로그에 기록

데이터 모델:

Order 테이블:

status: 주문 상태 (예약, 결제 완료, 취소 등)

canceled_at: 취소 시각

예시 코드:

Python

```
async def cancel_order(session, order_id):  
    order = await session.get(Order, order_id)  
  
    if order.status == "CANCELED":  
        return # 이미 취소된 경우  
  
    order.status = "CANCELED"  
  
    order.canceled_at = datetime.now()  
  
    inventory = order.inventory  
  
    inventory.quantity += 1 # 재고 복원  
  
    await session.commit()
```

코드를 사용할 때는 주의가 필요합니다.

추가 고려 사항:

분산 시스템: 다수의 서버에서 동일한 데이터에 접근하는 경우 분산 락, 분산 데이터베이스 등을 고려해야 합니다.

성능: 대규모 트래픽을 처리하기 위해 캐싱, 인덱싱 등을 활용하여 성능을 향상시켜야 합니다.

테스트: 다양한 시나리오를 기반으로 충분한 테스트를 수행하여 시스템의 안정성을 확보해야 합니다.

핵심:

동시성 문제 해결을 위한 다양한 기법을 적용

데이터 일관성 유지

Async 처리를 통한 성능 향상

명확한 예약 상태 관리

주의: 위에서 제시된 해결 방안은 일반적인 예시이며, 실제 시스템 구축 시에는 시스템의 특성과 요구사항에 맞게 수정되어야 합니다.