



# [사전 학습 자료] 동시성 이슈

## 동시성 이슈란 ?

분산 애플리케이션 환경에서 여러 스레드가 동시에 **공유 자원(shared resources)** 을 변경 하면서 발생하는 문제를 의미합니다. 따라서, **공유 자원(shared resources)** 이 무엇인지에 대한 이해가 중요합니다.

**경쟁 상태(race condition)**란 두 개 이상의 스레드가 공유 데이터에 액세스할 수 있고, 동시에 변경을 하려고 할 때 발생하는 문제입니다. 다수의 프로세스 혹은 스레드가 동기화 없이 **공유 자원(shared resources)** 에 접근하여 값을 변경하려는 현상을 의미합니다.

## 공유 자원(shared resources)

코드 측면에서는 Service 등에 전역 변수로 선언되어있는 필드가 있으며, 데이터베이스 측면에서는 컬럼의 값 일 수 있습니다.

```
@Service class OrderService { // 전역 변수 = 공유 자원(shared resources) private val products = mutableMapOf( "apple" to 100, "banana" to 50, "orange" to 75 ) ... fun order(...) { // 여기서 products 에 대한 값을 읽고 변경 } }
```

2개의 스레드(A,B)가 동시에 order 메서드에 접근하여 apple 을 구매하는 경우, 아래와 같은 시나리오가 발생할 수 있습니다.

1. A, B 모두 apple 조회 시, 재고가 100개 2. A 가 apple 10개를 구매 -> 아직 products 에는 반영 전 3. B 가 apple 20개를 구매하고 products 에 있는 apple 을 80 개로 업데이트 4. A 주문 로직이 완료되어 products 에 있는 apple 을 90 개로 업데이트

이러한 현상을 **Lost Update(write-write conflict)** 라고 합니다.

**Lost Update** 또는 **Write-Write Conflict**는 다중 트랜잭션이 같은 데이터를 동시에 업데이트하려고 할 때, 한 트랜잭션의 변경 사항이 다른 트랜잭션에 의해 덮어씌워지는 문제입니다. 이로 인해 **한 트랜잭션의 작업 결과가 사라지는 현상**이 발생합니다.

다시 돌아가서, products 가 왜 공유 자원인지를 이해하기 위해서는 CS 지식이 필요합니다.

## 프로세스와 스레드의 메모리 영역

프로세스 메모리 영역은 크게 4가지로 구성되어 있습니다.

- **Stack**: 매개변수, 지역변수 등 임시적인 자료
- **Heap**: 동적으로 할당되는 메모리
- **Data**: 전역 변수
  - **BSS**: 초기화되지 않은 데이터가 저장됨
  - **GVAR**: 초기화된 데이터가 저장됨
- **Text**: Program 의 코드

하나의 프로세스에서는 여러개의 스레드가 동작합니다. 위 4개의 영역 중에서, 스레드간 메모리를 공유하는 영역과, 그렇지 않은 영역이 존재합니다.

지역 변수가 저장되는 Stack 영역은, 스레드간 공유하지 않고 자신 만의 영역을 갖습니다. 그 외 나머지 영역은 공유하게 됩니다. 따라서 **products 전역 변수는 Data 영역에 저장되며**, 모든 스레드가 공유하게 되기 때문에 **공유 자원(shared resources)**에 해당됩니다.

동시성 이슈를 해결하기 위한 첫 번째 스텝은, 공유 자원(shared resources)을 식별하는 것입니다.

## 스레드별로 전역변수를 관리하기 위해서는 ?

사전 과제에서 주문 내역을 저장하기 위한 전역 변수를 보셨을 것입니다.

```
// 주문 정보를 저장하는 DB private val orderDatabase = mutableMapOf<String, MutableList<OrderInfo>>>()
```

주문은 스레드 별로 생성되며, 스레드 별로 관리되어야 합니다. 이때 ThreadLocal 을 사용하여 동시성 이슈를 해결할 수 있습니다.

## ThreadLocal 을 활용하여 주문 내역을 저장

```
// ThreadLocal 을 사용하여 각 스레드마다 독립적인 주문 데이터 보관 private val threadLocalOrderDatabase = ThreadLocal.withInitial { mutableMapOf<String, MutableList<OrderInfo>>>() }
```

하지만 ThreadLocal 을 ThreadPool 과 함께 사용할 때 주의할 점이 있습니다. ThreadLocal 을 모두 사용하고 나면 **ThreadLocal.remove()** 를 호출해서 스레드 로컬에 저장된 값을 제거해주어야 합니다. 제거하지 않으면 다른 스레드가 ThreadPool 에서 스레드를 빌릴때, 제거되지 않은 데이터를 참조할 수 있습니다.

*ThreadLocal* provides an easy-to-use API to confine some values to each thread. This is a reasonable way of achieving thread-safety in Java. However, **we should be extra careful when we're using *ThreadLocals* and thread pools together.**

In order to better understand this possible caveat, let's consider the following scenario:

1. First, the application borrows a thread from the pool.
2. Then it stores some thread-confined values into the current thread's *ThreadLocal*.
3. Once the current execution finishes, the application returns the borrowed thread to the pool.
4. After a while, the application borrows the same thread to process another request.
5. **Since the application didn't perform the necessary cleanups last time, it may re-use the same *ThreadLocal* data for the new request.**

## 데이터베이스 관점에서의 공유 자원

사전 과제에서 주어졌던 products 는 Database 에서 관리되어야 하는 상품(products) 입니다. 상품은 재고(stock) 을 가지고 있습니다.

Column Name	Data Type	Constraints	Description
id	BIGINT	PRIMARY KEY, AUTO_INCREMENT	상품 고유 ID
name	VARCHAR(255)	NOT NULL	상품 이름
description	TEXT	NULLABLE	상품 설명
price	DECIMAL(10, 2)	NOT NULL	상품 가격
stock	INT	DEFAULT 0, NOT NULL	상품 재고 수량
created_at	DATETIME	DEFAULT CURRENT_TIMESTAMP	상품 등록일
updated_at	DATETIME	DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP	마지막 수정일

여기서 멀티 스레드로 부터 데이터의 정합성을 보장해야 하는 것은 stock 입니다. 즉, stock 이 **공유 자원(shared resources)** 임을 알 수 있습니다.

## Thread-safe

멀티 스레드 환경에서 다수의 스레드가 동시에 접근하여 자원을 변경하려는 경우, 데이터 불일치 등의 문제가 발생할 수 있습니다. 이러한 환경에서도 데이터의 일관성을 유지할 수 있는 코드를 **Thread-safe** 한 코드라고 합니다.

### synchronized 는 실무에서 쓰면 혼나요 !

synchronized 는 성능이 좋지 않다고 알려져 있습니다. 내부적으로 모니터 락(Monitor Lock)이라는 것을 사용하여 스레드 간 **상호 배제**(<sup>㉜</sup> Wikipedia **Mutual exclusion**)를 보장합니다.




Mutual Exclusion - Allows concurrent access & updates to shared resources without race conditions.

모니터 락이란, 특정 객체에 대해 락을 획득하고 해제하는 JVM의 메커니즘입니다. `synchronized` 블록에 진입할 때 락을 획득해야 하고, 블록을 빠져나갈 때 락을 해제합니다.

- 이 과정에서 JVM은 다음 작업을 수행해야 합니다:
  - 락을 얻기 위한 경쟁 처리.
  - 락 상태 업데이트.
  - 락 소유자가 해제될 때 다른 대기 중인 스레드에게 락을 전달.
- 락 경합이 심할 경우, 이러한 작업이 지연 되면서 성능 저하로 이어집니다

## Chapter 17. Threads and Locks

The Java programming language provides multiple mechanisms for communicating between threads. The most basic of these methods is synchronization, which is implemented using monitors. Each object in Java is associated with a monitor, which a thread can lock or unlock. Only one thread at a time may hold a lock on a monitor. Any other threads attempting to lock that monitor are blocked until they can obtain a lock on that monitor.



-  GeeksforGeeks [Object Level Lock vs Class Level Lock in Java - GeeksforGeek...](#)
-  bhakti [Difference Between Semaphore and Mutex \(with Comparison Char...](#)
-  bhakti [Difference Between Semaphore and Monitor in OS \(with Compari...](#)

사실, 성능 외에도 분산 애플리케이션 환경에서 `synchronized` 를 사용하면 안되는 이유가 있습니다.

`synchronized` 는 단일 JVM 의 Java 프로세스 내부의 스레드 동기화만 보장합니다. 서버가 2대 이상인 경우에는 문제가 발생할 수 있습니다.

## java.util.Concurrent package

`java.util.concurrent` package 에 있는 클래스를 사용함으로써, Thread-safe 한 코드를 작성할 수 있습니다.

-  Baeldung [A Guide to ConcurrentHashMap | Baeldung](#)
- `ConcurrentHashMap` 은 내부적으로 CAS 연산을 사용하며 스레드가 락을 획득하거나 해제하는 과정 없이 원자적(atomic)으로 값을 변경하기 때문에, 전통적인 락 메커니즘에 비해 성능이 우수합니다.
-  LAYER6AI [번외편. ConcurrentHashMap](#)

## Alternative ways of synchronization

- 분산 락(Distributed Lock)
- DB Transaction Isolation Level
- Message Queue 기반 동기화
  - e.g 여러 인스턴스가 동시에 작업을 수행하더라도, 큐에 들어온 순서대로 처리 하게끔 한다. 또는 재고 변경이나 계좌 이체와 같은 중요한 작업을 직렬화 하여 순차적으로 처리하게끔 한다.