Java에서의 동기화

Synchronized, Volatile, Atomic

배서은

목차

Java에서의 동기화

경쟁상태(race condition)

공유자원과 임계영역

- 경쟁상태 : 둘 이상의 스레드가 공유자원을 병행적으로 읽거나 쓰는 동작을 할때 타이밍이나 접근순서에 따라 결과가 변하는 상황

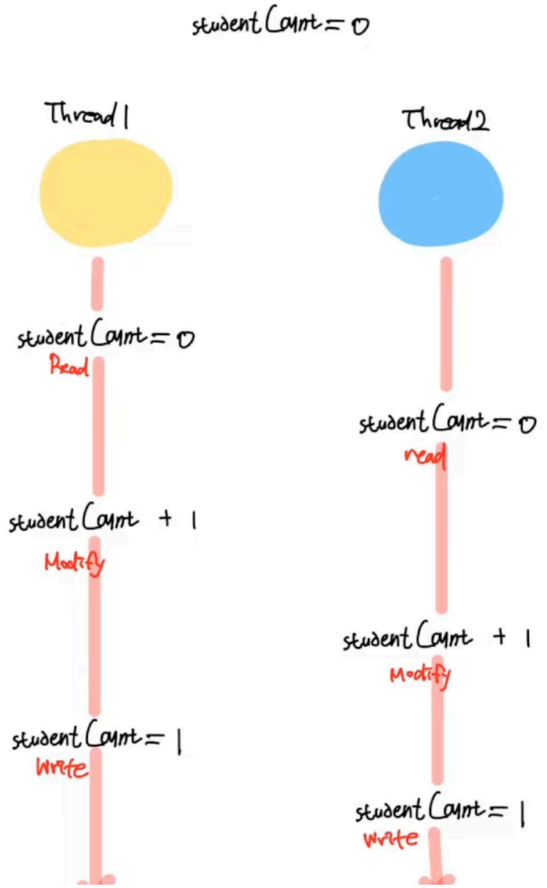
- EX) Read-Modify-Write , Check-then-act

Read-Modify-Write

경쟁상태 - 1

- 경쟁상태가 발생하는 패턴 중 하나

- -> 두 연산 모두0을 읽어오고,
  두 연산 모두 1로 업데이트를 하려고한다.

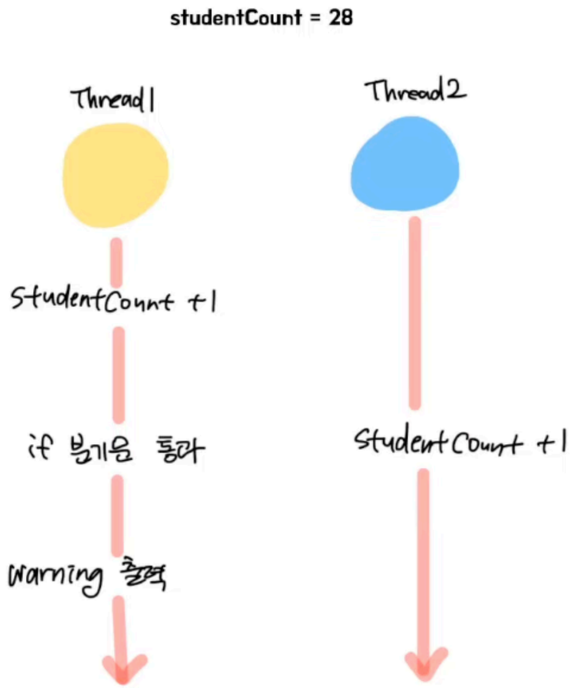


Check-then-act

경쟁상태 - 2

- 스레드1이 출력되는 과정에서 +1이 되면서 출력값은 30이 된다(스레드1의 입장에서 잘못된 데이터!)

- **원자성과 가시성을 보장해야한다.**



원자성

원자성과 가시성

- 공유 자원에 대한 작업의 단위가 더이상 쪼갤수없는 하나의 연산인 것처럼 동작하는 것

Read-modify-write

1. 기존 메모리 GET(Read)
2. Get한 데이터 수정(Modify)
3. 수정데이터 덮어쓰기(write)

Check-then-act

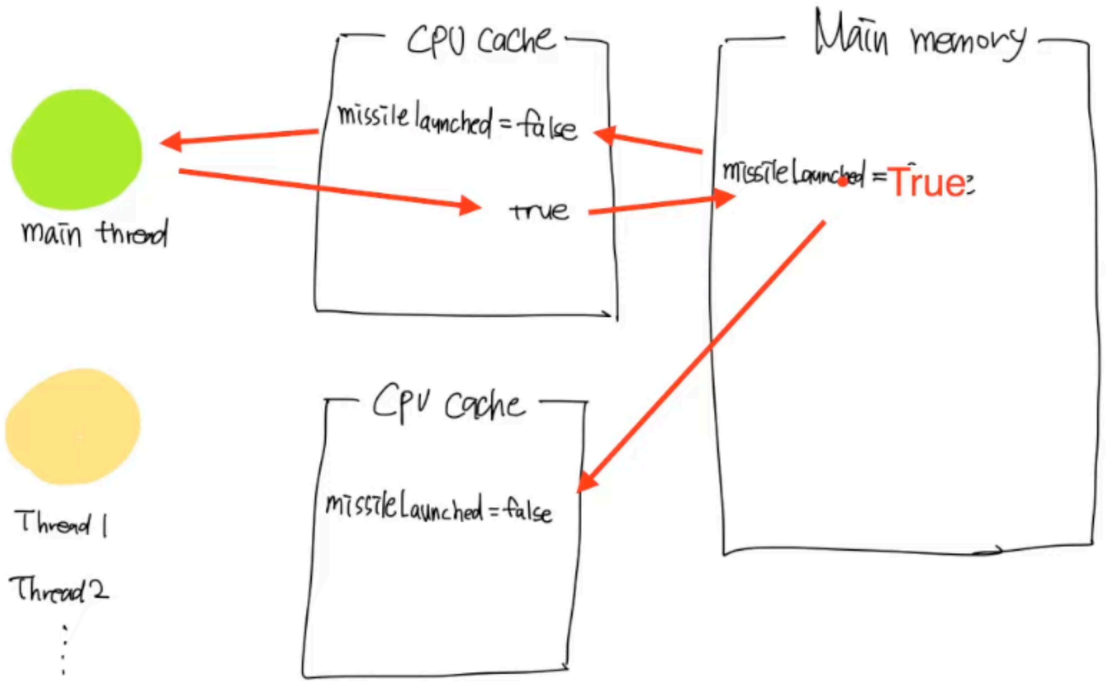
1. 기존 데이터Get한뒤 if문 비교(Read)
2. 로직 실행 (act)

기계어 레벨에서 여러번의 연산으로 분리 -> 스레드끼리 연산이 뒤죽박죽된다.

가시성

원자성과 가시성

- 메인메모리에 있는 진짜 값을 스레드가 보지못해서 가시성이라고 한다.(CPU cache때문에)



가시성을 위해?

- **Volatile 키워드**
- CPUcache를 쓰지않고 메인메모리에서만 값을 읽어온다.

```
public class MyClass {
    public volatile int counter = 0; // 이렇게 volatile 키워드를 공유변수에
    // 사용하면 멀티 쓰레드 환경에서 메인메모리의 변수를 공유할 수 있습니다.

    private void Mytest() {
      // ... (비즈니스 로직)
    }
}
```

volatile 은 언제쓸까?

앞서 말했듯이, volatile 키워드로 변수를 선언해서 여러개의 쓰레드에서 공유할 수 있도록할 때 사용하면 됩니다. 그러나 주의할점은, Volatile 은 Write 를 하나의 쓰레드에서만 할때 유용하고 **여러 쓰레드에서 Write 한다면 부적절합니다.** 지난번에 살펴봤던 경쟁상태인 Read-Modify-Write 패턴이 발생하 수 있기 때문이죠. 여러개의 변수가 동시에 읽고 그 시점을 기준으로 데이터를 변경할 때 혼란이 생기기 때문입니다.

아래에서 설명드릴 synchronized 와 비교해볼 경우 동기화 이슈로 인해 volatile 대신에 synchronized 키워드를 사용하면 되지 않나? 라는 생각이 들수 있습니다. 하지만 synchronized 를 사용할경우 대규모 서비스에서 심각한 성능저하가 생긴다는 장점이 생기므로, 상황을 적절히 고려하여 어떤걸 사용할지 결정해야합니다.

동기화의 방식

Java의 동기화

- 블로킹 방식
- 논블로킹 방식

블로킹 방식

동기화 방식 - 1

- 블로킹 : 특정 스레드가 작업을 수행하는 동안 다른 작업은 진행하지않고 대기하는 방식
- EX) Monitor , Synchronized 키워드

Monitor

블로킹

- 임계영역 접근하려는 여러 스레드
- 임계영역에는 한개의 스레드만 락을 쥐고 들어갈 수 있다.

- 배타동기큐(=Mutual Exclusion Queue)

Monitor

블로킹

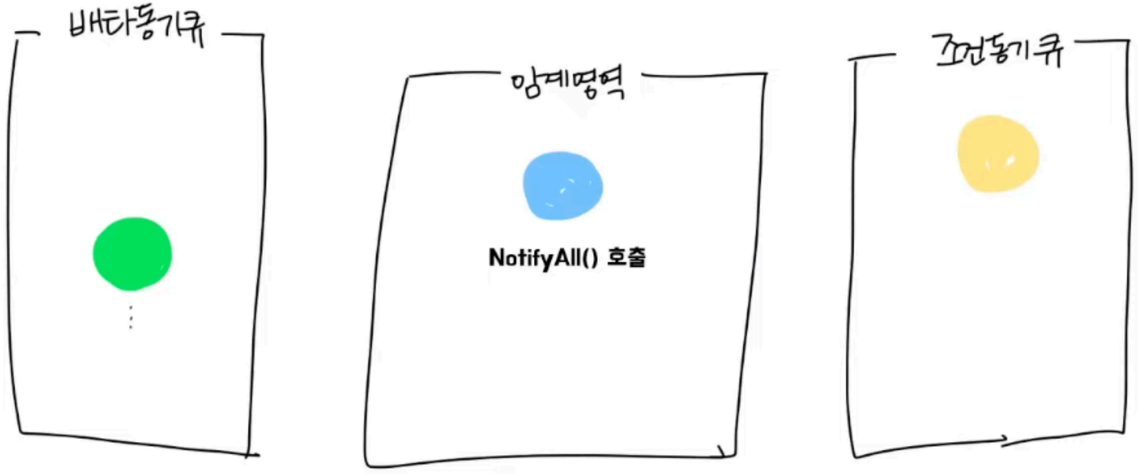
- 하나의 스레드가 임계영역에서 동작을 하다가
  wait()연산을 만나면

- sleep상태가 되면서,
  조건동기큐(ConditionalSynchronizationQ)로 들어간다.

Monitor

블로킹

- 스레드가 임계영역에서 작업을 하다가
  notify()/nofityAll()을 수행을 하게되면,
- 조건동기큐에서 sleep했던스레드가 깨어나면서,
  임계영역이 비었을때, 다시 임계영역으로 돌아온다.



Synchronized

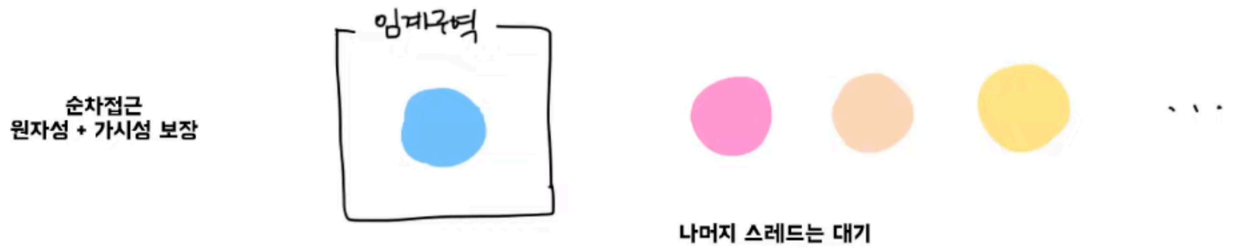
블로킹 - 모니터

- **Synchronized**
  - 자바에서 모니터메커니즘의 Mutual Exclusion을 선언하는 키워드
  - 키워드 선언한 코드에 한개의 스레드만 접근가능

Synchronized

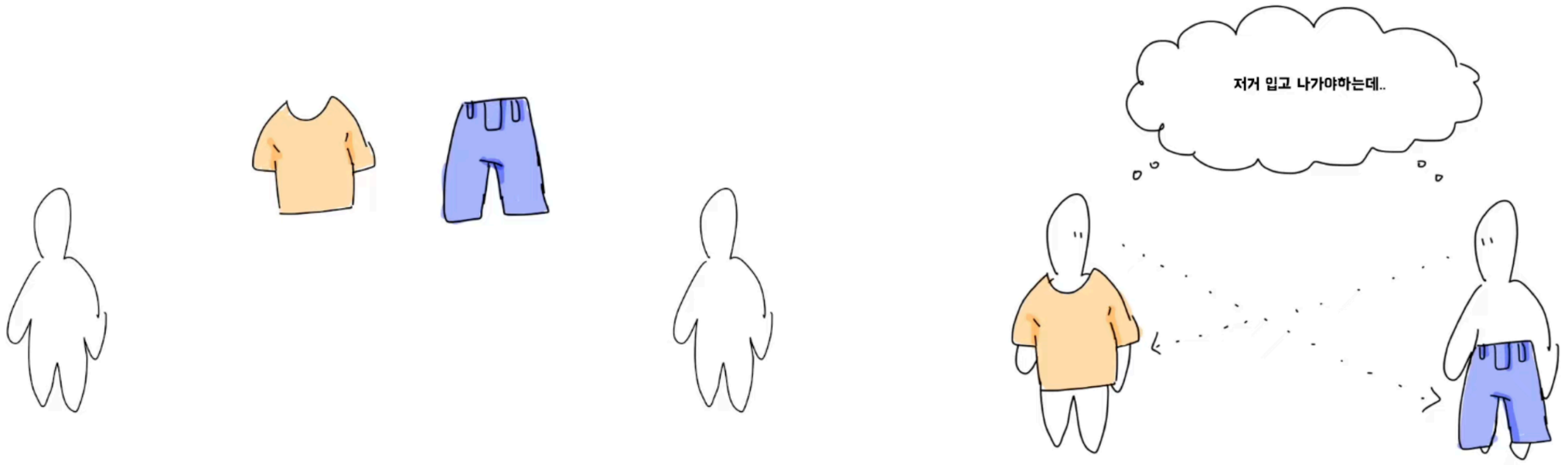
블로킹 - 모니터

- 단점
- 대기 스레드 == 성능 저하
- 데드락 발생



데드락

Synchronized 단점



- 서로 자신의 자원(에 대한 락)을 쥔채로, 상대방의 자원(에 대한 락)을 대기하는 상태

어떻게 사용해야할까?

Synchronized의 사용

1. 메소드보다는 **객체 레벨**로 지정한다.

```
public class SynchronizedTest implements Runnable {

  public void run() {

          synchronized (this.object) {

              long threadId = Thread.currentThread().getId();

              System.out.println("critical section start [" + threadId + "]");

              try {
                  System.out.println("Work within critical section[" + threadId + "]");
                  Thread.sleep(10000);
              } catch (Exception e) { /* No-op */ }

              System.out.println("critical section end [" + threadId + "]");
          }
      }
}
```

어떻게 사용해야 할까?

Synchronized의 사용

2. Synchronized블록 내에서 참조 객체를 수정하지 않는다.

다른 객체가 되어버려서, 싱크로나이즈드가 작동하지않고, 스레드가 동시에 실행된다.

```
public void run() {
        synchronized (this.object) {

            long threadId = Thread.currentThread().getId();

            System.out.println("critical section start [" + threadId + "]");

            System.out.println("Work within critical section [" + threadId +
"]");       this.object = new TestObject(200);

            try {

                Thread.sleep(10000);
            } catch (Exception e) { /* No-op */ }

            System.out.println("critical section end [" + threadId + "]");
        }
    }
```

논블로킹 방식

동기화 방식 - 2

- 다른 스레드의 작업여부와 상관없이 자신의 작업을 수행하는 방식
- EX) Atmoic 타입

CAS

Compare and set

기댓값과 현재의 자원값이 같으면 기존 자원값을 새로운 값으로 수정하고 True, 다르면 수정하지 않고 False

원자성 확보

Atomic타입

논블로킹

- 동시성을 보장하기 위해서 자바에서 제공하는 Wrapper Class

CAS(Compare and Set)

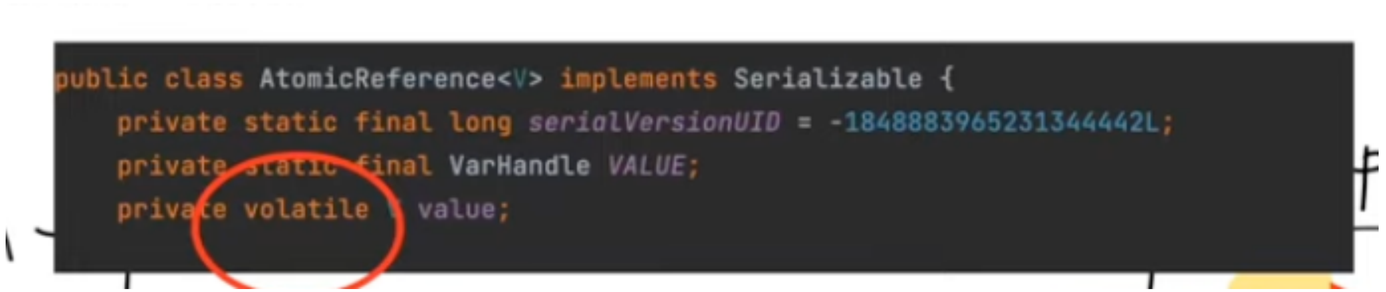
+

Volatile

AtomicReference

논블로킹

- 일반적, JVM 과 CPU 사이에 있는 캐시에서 값을 끌어온다.
- AtomicReference내부에 Voratile이 박혀있어서, 메모리->쓰레드 바로 가능
- CompareAndSet()메소드 사용
- Volatile을 통한 가시성을, CAS를 통해 원자성을 확보한다.



A-B-A 문제

AtomicReference 유의사항

CAS연산에서 공유 객체에 대한 변화를 감지하지 못할때 발생하는 문제.

- 원인

CAS내부적으로 포인터를 통한 비교를 하기 때문에 결국 주소값을 비교한다.

- 공유변수가 A->B->A 이렇게 돌아온 경우, 수정이 존재했었다는 것을 감지못한다는 점.
- 메모리 주소는 같아도, A->B로 변경이 일어났을 경우를 감지하지 못한다는 점.

AtmoicStampedReference

ABA문제 해결책

The `AtomicStampedReference` class provides an object reference variable which can be read and written atomically. By atomic is meant that multiple threads attempting to change the same `AtomicStampedReference` will not make the `AtomicStampedReference` end up in an inconsistent state.

The `AtomicStampedReference` is different from the **`AtomicReference`** in that the `AtomicStampedReference` keeps both an object reference and a stamp internally. The reference and stamp can be swapped using a single atomic compare-and-swap operation, via the `compareAndSet()` method.

The `AtomicStampedReference` is designed to be able to solve the A-B-A problem which is not possible to solve with an `AtomicReference` alone. The A-B-A problem is explained later in this text.

2. Why Do We Need *AtomicStampedReference*?

First, ***AtomicStampedReference*** **provides us with both an object reference variable and a stamp that we can read and write atomically**. **We can think of the stamp a bit like a timestamp or a version number**.

Simply put, adding a stamp **allows us to detect when another thread has changed the shared reference from the original reference A, to a new reference B, and back to the original reference A**.

Let's see how it behaves in practice.

- Timestamp 나 VersionNumber를 내부에 가지고 있고, 이를 같이 업데이트하는 방법을 사용해서 해결한다.

출처

[AtomicReferecence]

https://www.baeldung.com/java-atomicstampedreference
https://jenkov.com/tutorials/java-util-concurrent/atomicstampedreference.html

[Synchronized 유의할점]

https://hbase.tistory.com/179

[Java의 동기화-테코톡영상]

https://www.youtube.com/watch?v=ktWcieiNzKs

퀴즈

Java의 동기화

1. race condition 방지를 위해 보장되어야하는 두가지 속성은?
2. synchronized 키워드 사용시에 주의할 점 2 가지는?
3. atomicReferecence는 1)어떤 방식의 동기화인지 그리고 내부에서 사용되는 2)두가지 알고리즘과 키워드는?