10. CAS - 동기화와 원자적 연산

#1.인강/0.자바/5.자바-고급1편

원자적 연산 - 소개

컴퓨터 과학에서 사용하는 **원자적 연산**(atomic operation)의 의미는 해당 연산이 더 이상 나눌 수 없는 단위로 수행된다는 것을 의미한다. 즉, 원자적 연산은 중단되지 않고, 다른 연산과 간섭 없이 완전히 실행되거나 전혀 실행되지 않는 성질을 가지고 있다. 쉽게 이야기해서 멀티스레드 상황에서 다른 스레드의 간섭 없이 안전하게 처리되는 연산이라는 뜻이다.

> **참고**: 과거에 원자는 더 이상 나눌 수 없는 가장 작은 단위로 여겨졌다. 그래서 더는 나눌 수 없다는 뜻으로 **원자적 연산**이라는 단어를 사용한다. 물론 현대 물리학에서는 원자가 더 작은 입자들로 구성되어 있다는 것이 밝혀졌다. 하지만 원자적 연산이라는 단어는 그대로 사용한다.

예를 들어서 다음과 같은 필드가 있을 때
`volatile int i = 0;`

다음 연산은 둘로 쪼갤 수 없는 원자적 연산이다.

- `i = 1`

왜냐하면 이 연산은 다음 단 하나의 순서로 실행되기 때문이다.

1. 오른쪽에 있는 `1`의 값은 왼쪽의 `i` 변수에 대입한다.

하지만 다음 연산은 원자적 연산이 아니다.

```
i = i + 1;
```

왜냐하면 이 연산은 다음 순서로 나누어 실행되기 때문이다.

1. 오른쪽에 있는 `i`의 값을 읽는다. `i`의 값은 10이다.
2. 읽은 10에 1을 더해서 11을 만든다.
3. 더한 11을 왼쪽의 `i` 변수에 대입한다.

원자적 연산은 멀티스레드 상황에서 아무런 문제가 발생하지 않는다. 하지만 원자적 연산이 아닌 경우에는 `synchronized` 블럭이나 `Lock`등을 사용해서 안전한 임계 영역을 만들어야 한다.

순서대로 실행

예를 들어서 2개의 스레드가 해당 로직을 수행하는데, 하나가 완전히 끝나고 나서, 나머지 하나가 수행된다고 가정해보자.

```
처음에 i = 0이라고 가정하겠다.

스레드1: i = i + 1 연산 수행
스레드1: i의 값을 읽는다. i는 0이다.
스레드1: 읽은 0에 1을 더해서 1을 만든다.
스레드1: 더한 1을 왼쪽의 i변수에 대입한다.
결과: i의 값은 1이다.

스레드2: i = i + 1 연산 수행
스레드2: i의 값을 읽는다. i는 1이다.
스레드2: 읽은 1에 1을 더해서 2을 만든다.
스레드2: 더한 2을 왼쪽의 i변수에 대입한다.
결과: i의 값은 2이다.
```

2개의 스레드가 각각 한 번 연산을 수행했으므로 변수 `i`의 값은 0 ➜ 2가 된다.

동시에 실행

이번에는 문제가 되는 경우를 알아보자. 2개의 스레드가 해당 로직을 동시에 함께 수행하면, 문제가 발생한다.

```
처음에 i = 0이라고 가정하겠다.

스레드1: i = i + 1 연산 수행
스레드2: i = i + 1 연산 수행
스레드1: i의 값을 읽는다. i는 0이다.
스레드2: i의 값을 읽는다. i는 0이다.
스레드1: 읽은 0에 1을 더해서 1을 만든다.
스레드2: 읽은 0에 1을 더해서 1을 만든다.
스레드1: 더한 1을 왼쪽의 i변수에 대입한다.
스레드2: 더한 1을 왼쪽의 i변수에 대입한다.
```

```
결과: i의 값은 1이다.
```

2개의 스레드가 각각 한 번 연산을 수행했지만 변수 `i`의 값은 0➜ 1이 된다. 한 번의 연산이 사라진 것이다.

그렇다면 다음 연산은 원자적 연산일까?

`i++`

이 연산은 원자적 연산처럼 보이지만 사실은 원자적 연산이 아니다. 왜냐하면 이 연산은 앞서 살펴본 `i = i + 1`을 축약한 것이기 때문이다. 결과적으로 `i++`은 `i = i + 1`와 똑같이 동작한다.

원자적 연산 - 시작

원자적이지 않은 연산을 멀티스레드 환경에서 실행하면 어떤 문제가 발생하는지 코드로 알아보자.

`IncrementInteger`는 숫자 값을 하나씩 증가시키는 기능을 제공한다. 예를 들어서 지금까지 접속한 사용자의 수 등을 계산할 때 사용할 수 있다.

```
package thread.cas.increment;

public interface IncrementInteger {
    void increment();

    int get();
}
```

- `IncrementInteger`는 값을 증가하는 기능을 가진 숫자 기능을 제공하는 인터페이스다.
- `increment()` : 값을 하나 증가
- `get()` : 값을 조회

```
package thread.cas.increment;

public class BasicInteger implements IncrementInteger {

    private int value;

    @Override
    public void increment() {
        value++;
    }
```

```
    @Override
    public int get() {
        return value;
    }
}
```

- `IncrementInteger` 인터페이스의 가장 기본 구현이다.
- `increment()` 를 호출하면 `value++` 를 통해서 값을 하나 증가한다.
  - `value` 값은 인스턴스의 필드이기 때문에, 여러 스레드가 공유할 수 있다. 이렇게 공유 가능한 자원에 `++` 와 같은 원자적이지 않은 연산을 사용하면 멀티스레드 상황에 문제가 될 수 있다.

```
package thread.cas.increment;

import java.util.ArrayList;
import java.util.List;

import static util.ThreadUtils.sleep;

public class IncrementThreadMain {

    public static final int THREAD_COUNT = 1000;

    public static void main(String[] args) throws InterruptedException {
        test(new BasicInteger());
    }

    private static void test(IncrementInteger incrementInteger)
                                            throws InterruptedException {

        Runnable runnable = new Runnable() {
            @Override
            public void run() {
                sleep(10); //너무 빨리 실행되기 때문에, 다른 스레드와 동시 실행을 위해 잠깐 쉬었
다가 실행
                incrementInteger.increment();
            }
        };

        List<Thread> threads = new ArrayList<>();
        for (int i = 0; i < THREAD_COUNT; i++) {
            Thread thread = new Thread(runnable);
            threads.add(thread);
```

```
            thread.start();
        }

        for (Thread thread : threads) {
            thread.join();
        }

        int result = incrementInteger.get();
        System.out.println(incrementInteger.getClass().getSimpleName() + "
result: " + result);
    }

}
```

- `THREAD_COUNT` 수 만큼 스레드를 생성하고 `incrementInteger.increment()`를 호출한다.
- 스레드를 1000개 생성했다면, `increment()` 메서드도 1000번 호출하기 때문에 결과는 1000이 되어야 한다.
- 참고로 스레드가 너무 빨리 실행되기 때문에, 여러 스레드가 동시에 실행되는 상황을 확인하기 어렵다. 그래서 `run()` 메서드에 `sleep(10)`을 두어서, 최대한 많은 스레드가 동시에 `increment()`를 호출하도록 한다.

실행 결과

```
BasicInteger result: 950
```

실행 결과를 보면 기대한 1000이 아니라 다른 숫자가 보인다. 아마도 실행 환경에 따라서 다르겠지만 1000이 아니라 조금 더 적은 숫자가 보일 것이다. 물론 실행 환경에 따라서 1000이 보일 수도 있다.
이 문제는 앞서 설명한 것 처럼 여러 스레드가 동시에 원자적이지 않은 `value++`을 호출했기 때문에 발생한다.
그럼 혹시 `volatile`을 적용하면 될까?

원자적 연산 - volatile, synchronized

예제 - volatile

다음과 같이 `VolatileInteger` 클래스를 만들고, `volatile`을 적용해보자.

```
package thread.cas.increment;

public class VolatileInteger implements IncrementInteger {

    private volatile int value;

    @Override
```

```
    public void increment() {
        value++;
    }

    @Override
    public int get() {
        return value;
    }
}
```

- `BasicInteger`와 같고 `volatile`만 적용해주었다.

IncrementThreadMain에 추가

```
public class IncrementThreadMain {

    public static void main(String[] args) throws InterruptedException {
        test(new BasicInteger());
        test(new VolatileInteger()); // 추가
    }
    ...
}
```

- `test(new VolatileInteger());`를 한 줄 추가하고 실행해보자.

실행 결과

```
BasicInteger result: 950
VolatileInteger result: 961
```

실행 결과를 보면 `VolatileInteger`도 여전히 1000이 아니라 더 작은 숫자가 나온다.
`volatile`은 여러 CPU 사이에 발생하는 캐시 메모리와 메인 메모리가 동기화 되지 않는 문제를 해결할 뿐이다.
`volatile`을 사용하면 CPU의 캐시 메모리를 무시하고, 메인 메모리를 직접 사용하도록 한다. 하지만 지금 이 문제는 캐시 메모리가 영향을 줄 수는 있지만, 캐시 메모리를 사용하지 않고, 메인 메모리를 직접 사용해도 여전히 발생하는 문제이다.
이 문제는 연산 자체가 나누어져 있기 때문에 발생한다. `volatile`은 연산 차제를 원자적으로 묶어주는 기능이 아니다.

이렇게 연산 자체가 나누어진 경우에는 `synchronized` 블럭이나 `Lock`등을 사용해서 안전한 임계 영역을 만들어야 한다.

예제 - synchronized

다음과 같이 `SyncInteger` 클래스를 만들고 `synchronized`를 적용해서 안전한 임계 영역을 만들어보자.

```
package thread.cas.increment;

public class SyncInteger implements IncrementInteger {
    private int value;

    @Override
    public synchronized void increment() {
        value++;
    }

    @Override
    public synchronized int get() {
        return value;
    }
}
```

- `value++` 연산은 `synchronized`를 통해 임계 영역 안에서 안전하게 수행된다. 쉽게 이야기해서 한 번에 하나의 스레드만 해당 연산을 수행할 수 있다.

IncrementThreadMain에 추가

```
public class IncrementThreadMain {

    public static void main(String[] args) throws InterruptedException {
        test(new BasicInteger());
        test(new VolatileInteger());
        test(new SyncInteger()); //추가
    }
    ...
}
```

- `test(new SyncInteger());`를 한 줄 추가하고 실행해보자.

실행 결과

```
BasicInteger result: 950
VolatileInteger result: 961
SyncInteger result: 1000
```

`synchronized`를 통해 안전한 임계 영역을 만들고 `value++` 연산을 수행했더니 정확히 1000이라는 결과가 나왔다. 1000개의 스레드가 안전하게 `value++` 연산을 수행한 것이다.

원자적 연산 - AtomicInteger

자바는 앞서 만든 `SyncInteger`와 같이 멀티스레드 상황에서 안전하게 증가 연산을 수행할 수 있는 `AtomicInteger`라는 클래스를 제공한다. 이름 그대로 원자적인 `Integer`라는 뜻이다.
다음과 같이 `MyAtomicInteger` 클래스를 만들고, 자바가 제공하는 `AtomicInteger`를 사용해보자.

```
package thread.cas.increment;

import java.util.concurrent.atomic.AtomicInteger;

public class MyAtomicInteger implements IncrementInteger {

    AtomicInteger atomicInteger = new AtomicInteger(0);

    @Override
    public void increment() {
        atomicInteger.incrementAndGet();
    }

    @Override
    public int get() {
        return atomicInteger.get();
    }
}
```

- `new AtomicInteger(0)` : 초기값을 지정한다. 생략하면 `0`부터 시작한다.
- `incrementAndGet()` : 값을 하나 증가하고 증가된 결과를 반환한다.
- `get()` : 현재 값을 반환한다.

IncrementThreadMain에 추가

```
public class IncrementThreadMain {

    public static void main(String[] args) throws InterruptedException {
        test(new BasicInteger());
        test(new VolatileInteger());
        test(new SyncInteger());
        test(new MyAtomicInteger()); // 추가
    }
```

```
    ...
}
```

- `test(new MyAtomicInteger());`를 한 줄 추가하고 실행해보자.

실행 결과

```
BasicInteger result: 950
VolatileInteger result: 961
SyncInteger result: 1000
MyAtomicInteger result: 1000
```

실행 결과를 보면 `AtomicInteger`를 사용하면 `MyAtomicInteger`의 결과도 1000인 것을 확인할 수 있다.
1000개의 스레드가 안전하게 증가 연산을 수행한 것이다.

`AtomicInteger`는 멀티스레드 상황에 안전하고 또 다양한 값 증가, 감소 연산을 제공한다. 특정 값을 증가하거나 감소해야 하는데 여러 스레드가 해당 값을 공유해야 한다면, `AtomicInteger`를 사용하면 된다.

> 참고: `AtomicInteger`, `AtomicLong`, `AtomicBoolean`등 다양한 `AtomicXxx` 클래스가 존재한다.

원자적 연산 - 성능 테스트

이제 `AtomicInteger`의 진짜 모습을 하나씩 알아보자.
우선 `AtomicInteger`와 우리가 직접 만든 각 클래스의 성능을 비교해보자.

```
package thread.cas.increment;

public class IncrementPerformanceMain {

    public static final long COUNT = 100_000_000;

    public static void main(String[] args) {
        test(new BasicInteger());
        test(new VolatileInteger());
        test(new SyncInteger());
        test(new MyAtomicInteger());
    }
```

```
    private static void test(IncrementInteger incrementInteger) {
        long startMs = System.currentTimeMillis();
        for (long i = 0; i < COUNT; i++) {
            incrementInteger.increment();
        }
        long endMs = System.currentTimeMillis();
        System.out.println(incrementInteger.getClass().getSimpleName() + ": ms="
+ (endMs - startMs));
    }
}
```

- 단일 연산은 너무 빠르기 때문에 성능을 확인하려면 어느 정도 반복적인 연산이 필요하다.
- 각각 `COUNT` 만큼 반복해서 연산을 수행해 보자. 여기서는 1억 번 값 증가 연산을 수행했다.

실행 결과

```
BasicInteger: ms=39
VolatileInteger: ms=455
SyncInteger: ms=625
MyAtomicInteger: ms=367
```

BasicInteger

- 가장 빠르다.
- CPU 캐시를 적극 사용한다. CPU 캐시의 위력을 알 수 있다.
- 안전한 임계 영역도 없고, `volatile` 도 사용하지 않기 때문에 멀티스레드 상황에는 사용할 수 없다.
- 단일 스레드가 사용하는 경우에 효율적이다.

VolatileInteger

- `volatile` 을 사용해서 CPU 캐시를 사용하지 않고 메인 메모리를 사용한다.
- 안전한 임계 영역이 없기 때문에 멀티스레드 상황에는 사용할 수 없다.
- 단일 스레드가 사용하기에는 `BasicInteger` 보다 느리다. 그리고 멀티스레드 상황에도 안전하지 않다.

SyncInteger

- `synchronized` 를 사용한 안전한 임계 영역이 있기 때문에 멀티스레드 상황에도 안전하게 사용할 수 있다.
- `MyAtomicInteger` 보다 성능이 느리다.

MyAtomicInteger

- 자바가 제공하는 `AtomicInteger` 를 사용한다. 멀티스레드 상황에 안전하게 사용할 수 있다.
- 성능도 `synchronized` , `Lock(ReentrantLock)` 을 사용하는 경우보다 1.5 ~ 2배 정도 빠르다.

`SyncInteger`처럼 락을 사용하는 경우보다, `AtomicInteger`가 더 빠른 이유는 무엇일까? `i++` 연산은 원자적인 연산이 아니다. 따라서 분명히 `synchronized`, `Lock(ReentrantLock)`와 같은 락을 통해 안전한 임계 영역을 만들어야 할 것 같다.
놀랍게도 `AtomicInteger`가 제공하는 `incrementAndGet()` 메서드는 락을 사용하지 않고, 원자적 연산을 만들어낸다.

CAS 연산1

참고: CAS 연산은 심화 내용이다. 이해가 어렵다면 가볍게 듣고 넘어가도 괜찮다. 왜냐하면 우리가 직접 CAS 연산을 사용하는 경우는 거의 없기 때문이다. 대부분 복잡한 동시성 라이브러리들이 CAS 연산을 사용한다. 우리는 AtomicInteger와 같은 CAS 연산을 사용하는 라이브러리들을 잘 사용하는 정도면 충분하다.

락 기반 방식의 문제점

`SyncInteger`와 같은 클래스는 데이터를 보호하기 위해 락을 사용한다.
여기서 말하는 락은 `synchronized`, `Lock(ReentrantLock)` 등을 사용하는 것을 말한다.
락은 특정 자원을 보호하기 위해 스레드가 해당 자원에 대한 접근하는 것을 제한한다. 락이 걸려 있는 동안 다른 스레드들은 해당 자원에 접근할 수 없고, 락이 해제될 때까지 대기해야 한다.
또한 락 기반 접근에서는 락을 획득하고 해제하는 데 시간이 소요된다.

예를 들어서 락을 사용하는 연산이 있다고 가정하자. 락을 사용하는 방식은 다음과 같이 작동한다.

1. 락이 있는지 확인한다.
2. 락을 획득하고 임계 영역에 들어간다.
3. 작업을 수행한다.
4. 락을 반납한다.

여기서 락을 획득하고 반납하는 과정이 계속 반복된다. 10000번의 연산이 있다면 10000번의 연산 모두 같은 과정을 반복한다.
이렇듯 락을 사용하는 방식은 직관적이지만 상대적으로 무거운 방식이다.

CAS

이런 문제를 해결하기 위해 락을 걸지 않고 원자적인 연산을 수행할 수 있는 방법이 있는데, 이것을 CAS(Compare-And-Swap, Compare-And-Set) 연산이라 한다. 이 방법은 락을 사용하지 않기 때문에 락 프리(lock-free) 기법이라

한다. 참고로 CAS 연산은 락을 완전히 대체하는 것은 아니고, **작은 단위의 일부 영역에 적용**할 수 있다. 기본은 락을 사용하고, 특별한 경우에 CAS를 적용할 수 있다고 생각하면 된다.

지금부터 코드를 통해 CAS 연산을 알아보자.

```
package thread.cas;

import java.util.concurrent.atomic.AtomicInteger;

public class CasMainV1 {

    public static void main(String[] args) {
        AtomicInteger atomicInteger = new AtomicInteger(0);
        System.out.println("start value =  " + atomicInteger.get());

        boolean result1 = atomicInteger.compareAndSet(0, 1);
        System.out.println("result1 = " + result1 + ", value = " +
atomicInteger.get());

        boolean result2 = atomicInteger.compareAndSet(0, 1);
        System.out.println("result2 = " + result2 + ", value = " +
atomicInteger.get());
    }
}
```

- `new AtomicInteger(0)` : 내부에 있는 기본 숫자 값을 `0`으로 설정한다.
- 자바는 `AtomicXxx`의 `compareAndSet()` 메서드를 통해 CAS 연산을 지원한다.

실행 결과

```
start value =  0
result1 = true, value = 1
result2 = false, value = 1
```

compareAndSet(0, 1)

`atomicInteger`가 가지고 있는 값이 현재 0이면 이 값을 1로 변경하라는 매우 단순한 메서드이다.

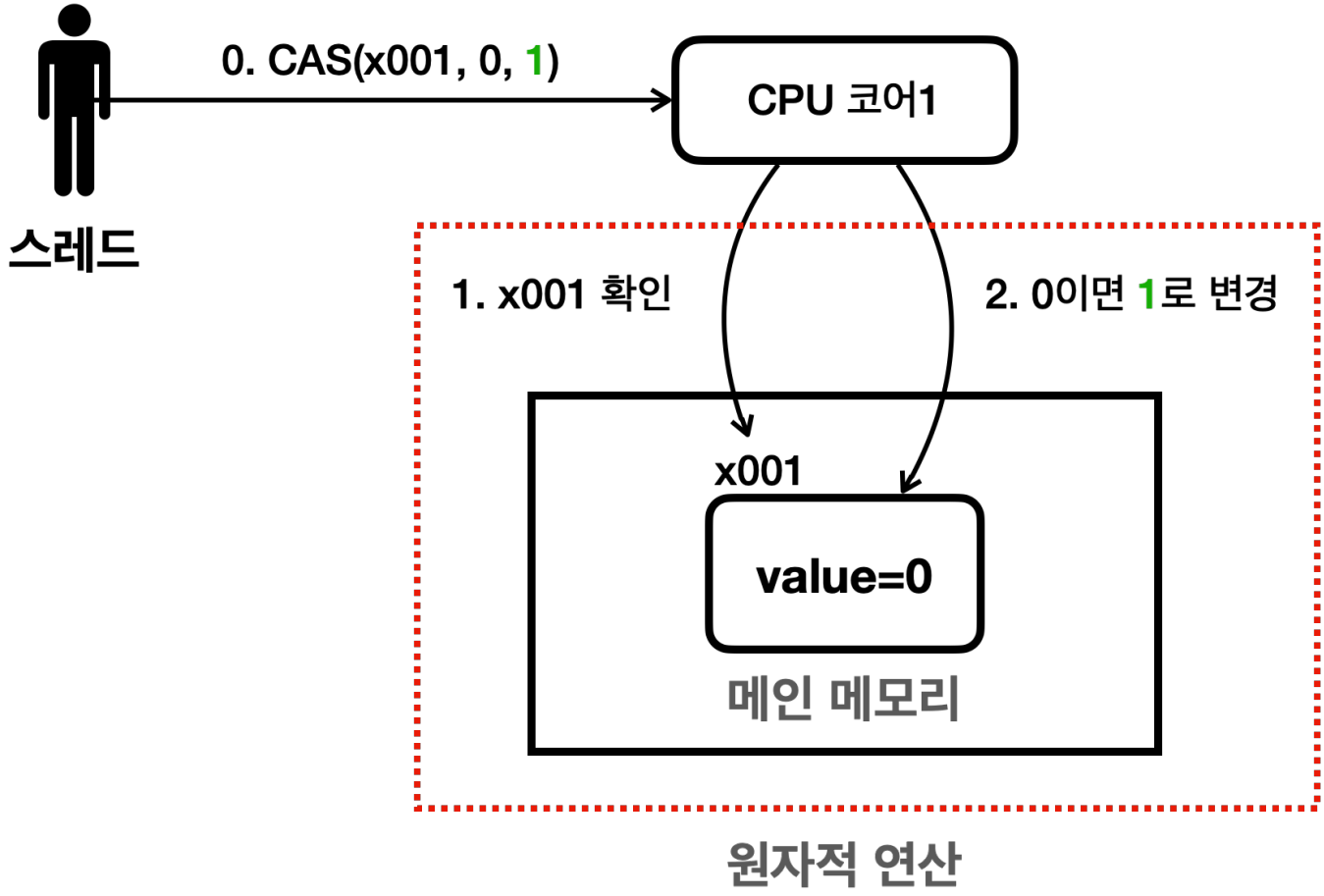
- 만약 `atomicInteger`의 값이 현재 0이라면 `atomicInteger`의 값은 1로 변경된다. 이 경우 `true`를 반환한다.
- 만약 `atomicInteger`의 값이 현재 0이 아니라면 `atomicInteger`의 값은 변경되지 않는다. 이 경우 `false`를 반환한다.

여기서 가장 중요한 내용이 있는데, **이 메서드는 원자적으로 실행**된다는 점이다.

그리고 이 메서드가 제공하는 기능이 바로 CAS(compareAndSet) 연산이다.

실행 순서 분석

CAS - 성공 케이스



- 여기서는 `AtomicInteger` 내부에 있는 `value` 값이 0이라면 1로 변경하고 싶다.
- `compareAndSet(0, 1)` 을 호출한다. 매개변수의 왼쪽이 기대하는 값, 오른쪽이 변경하는 값이다.
- `CAS` 연산은 메모리에 있는 값이 기대하는 값이라면 원하는 값으로 변경한다.
- 메모리에 있는 `value` 의 값이 0이므로 1로 변경할 수 있다.
- 그런데 생각해보면 이 명령어는 2개로 나누어진 명령어이다. 따라서 원자적이지 않은 연산처럼 보인다.
    1. 먼저 메인 메모리에 있는 값을 확인한다.
    2. 해당 값이 기대하는 값(0)이라면 원하는 값(1)으로 변경한다.

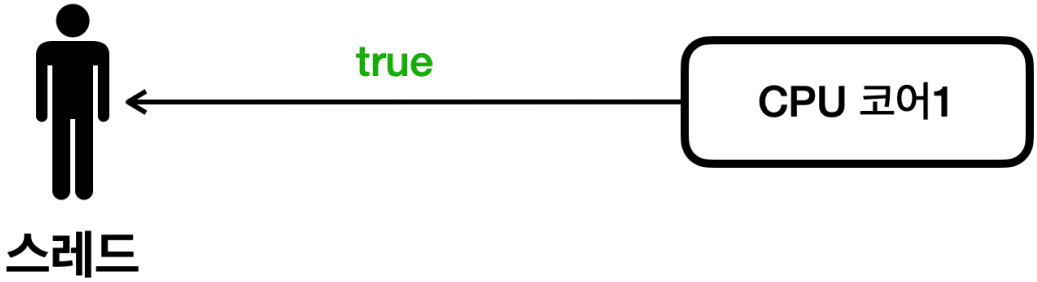
CPU 하드웨어의 지원

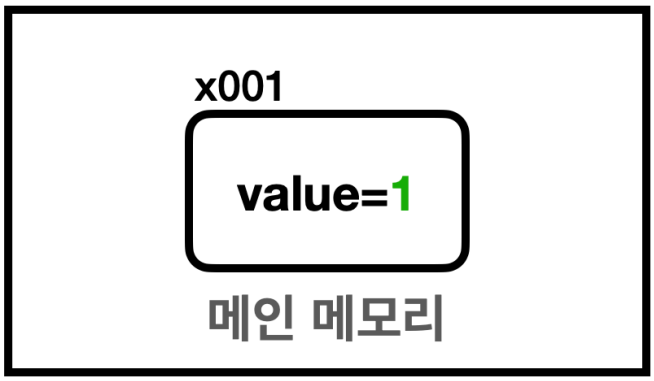
CAS 연산은 이렇게 원자적이지 않은 두 개의 연산을 CPU 하드웨어 차원에서 특별하게 하나의 원자적인 연산으로 묶어서 제공하는 기능이다. 이것은 소프트웨어가 제공하는 기능이 아니라 하드웨어가 제공하는 기능이다. 대부분의 현대 CPU들은 CAS 연산을 위한 명령어를 제공한다.

CPU는 다음 두 과정을 묶어서 하나의 원자적인 명령으로 만들어버린다. 따라서 중간에 다른 스레드가 개입할 수 없다.

1. x001의 값을 확인한다.
2. 읽은 값이 0이면 1로 변경한다.

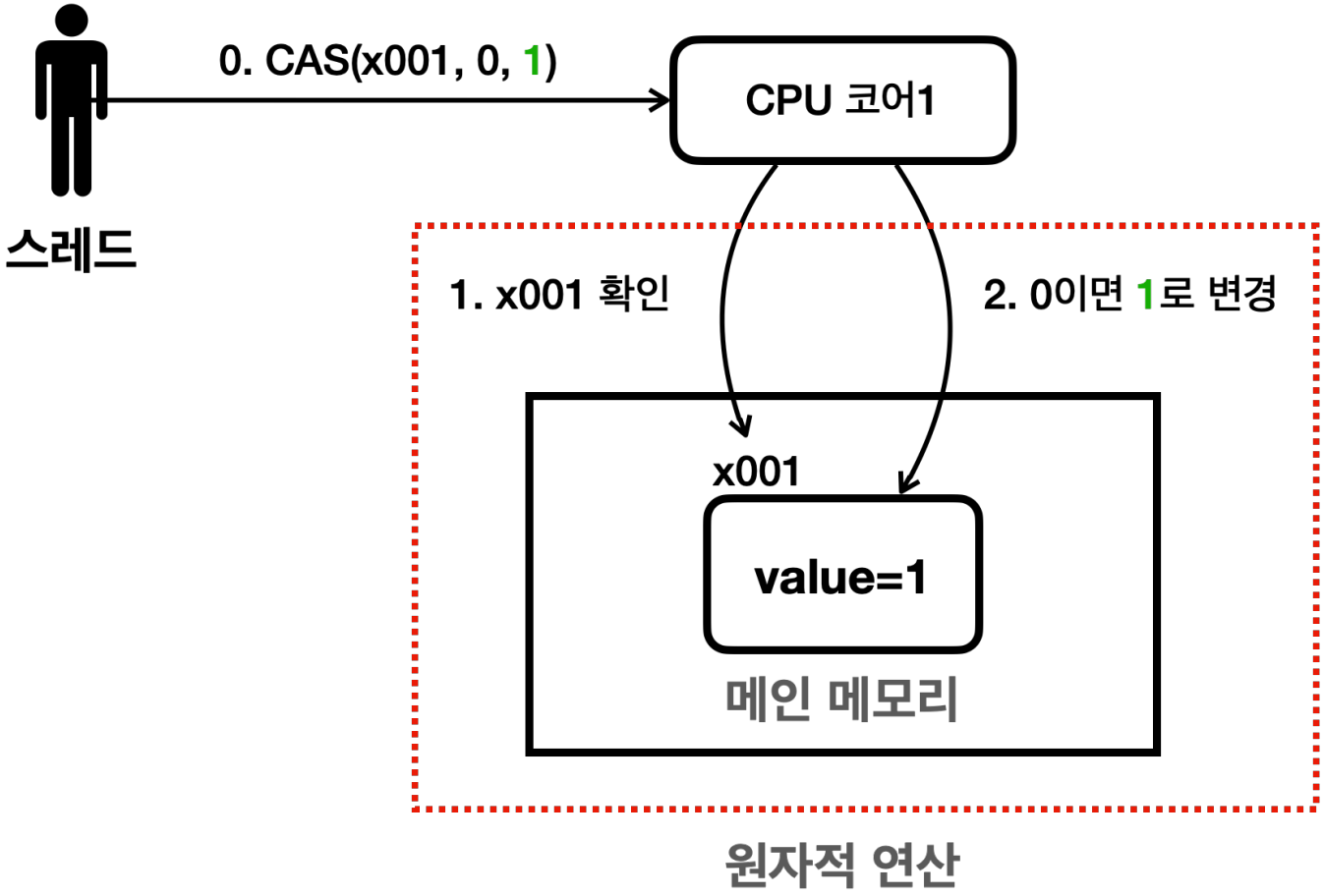
CPU는 두 과정을 하나의 원자적인 명령으로 만들기 위해 1번과 2번 사이에 다른 스레드가 `x001`의 값을 변경하지 못하게 막는다. 참고로 1번과 2번 사이의 시간은 CPU 입장에서 보면 아주 잠깐 찰나의 순간이다. 그래서 성능에 큰 영향을 끼치지 않는다. CPU가 1초에 얼마나 많은 연산을 수행하는지 생각해보면 이해가 될 것이다.



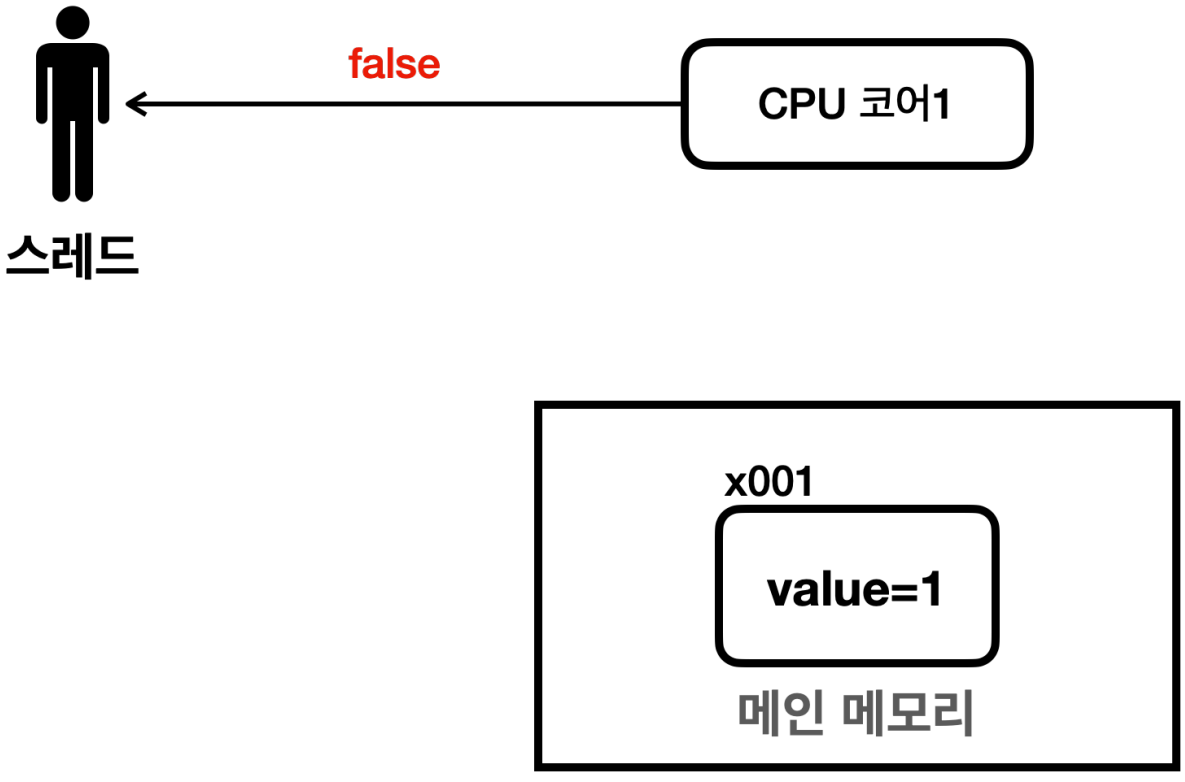


- `value`의 값이 0 ➜ 1이 되었다.
- CAS 연산으로 값을 성공적으로 변경하고 나면 `true`를 반환한다.

CAS - 실패 케이스



- `CAS` 연산은 메모리에 있는 값이 기대하는 값이라면 원하는 값으로 변경한다.
- 여기서는 `AtomicInteger` 내부에 있는 `value` 값이 0이라면 1로 변경하고 싶다.
- 현재 `value`의 값이 기대하는 0이 아니라 1이므로 아무것도 변경하지 않는다.



- CAS 연산으로 값 변경에 실패하면 `false`를 반환하고, 값도 변경하지 않는다.

여기까지 듣고 보면 CAS 연산을 사용하면, 1. 기대하는 값을 확인하고 2. 값을 변경하는 두 연산을 하나로 묶어서 원자적으로 제공한다는 것은 이해했을 것이다. 그런데 이 기능이 어떻게 락을 일부 대체할 수 있다는 것일까?

CAS 연산2

어떤 값을 하나 증가하는 `value++` 연산은 원자적 연산이 아니다. 이 연산은 다음 연산과 같다.

```
i = i + 1;
```

이 연산은 다음 순서로 나누어 실행된다. `i`의 초기 값은 0으로 가정하겠다.

1. 오른쪽에 있는 `i`의 값을 읽는다. `i`의 값은 0이다.
2. 읽은 0에 1을 더해서 1을 만든다.
3. 더한 1을 왼쪽의 `i` 변수에 대입한다.

1번과 3번 연산 사이에 다른 스레드가 `i`의 값을 변경할 수 있기 때문에, 문제가 될 수 있다. 따라서 `value++` 연산을 여러 스레드에서 사용한다면, 락을 건 다음에 값을 증가해야 한다.

CAS 연산을 활용해서 락 없이 값을 증가하는 기능을 만들어보자.
`AtomicInteger`가 제공하는 `incrementAndGet()` 메서드가 어떻게 CAS 연산을 활용해서 락 없이 만들어졌는지 직접 구현해보자.

```
package thread.cas;

import java.util.concurrent.atomic.AtomicInteger;

import static util.MyLogger.log;

public class CasMainV2 {

    public static void main(String[] args) {
        AtomicInteger atomicInteger = new AtomicInteger(0);
        System.out.println("start value = " + atomicInteger.get());

        // incrementAndGet 구현
        int resultValue1 = incrementAndGet(atomicInteger);
        System.out.println("resultValue1 = " + resultValue1);
```

```
        int resultValue2 = incrementAndGet(atomicInteger);
        System.out.println("resultValue2 = " + resultValue2);
    }

    private static int incrementAndGet(AtomicInteger atomicInteger) {
        int getValue;
        boolean result;
        do {
            getValue = atomicInteger.get();
            log("getValue: " + getValue);
            result = atomicInteger.compareAndSet(getValue, getValue + 1);
            log("result: " + result);
        } while (!result);
        return getValue + 1;
    }
}
```

여기서 만든 `incrementAndGet()`은 `atomicInteger` 내부의 `value` 값을 하나 증가하는 메서드이다. 사실 `atomicInteger`도 이 메서드를 제공하지만, 여기서는 이해를 위해 직접 구현해보자.

CAS 연산을 사용하면 여러 스레드가 같은 값을 사용하는 상황에서도 락을 걸지 않고, 안전하게 값을 증가할 수 있다. 여기서는 락을 걸지 않고 CAS 연산을 사용해서 값을 증가했다.

- `getValue = atomicInteger.get()`을 사용해서 `value` 값을 읽는다.
- `compareAndSet(getValue, getValue + 1)`을 사용해서, 방금 읽은 `value` 값이 메모리의 `value` 값과 같다면 `value` 값을 하나 증가한다. 여기서 CAS 연산을 사용한다.
- 만약 CAS 연산이 성공한다면 `true`를 반환하고 `do~while`문을 빠져나온다.
- 만약 CAS 연산이 실패한다면 `false`를 반환하고 `do~while`문을 다시 시작한다.

실행 결과

```
start value = 0
17:48:20.440 [     main] getValue: 0
17:48:20.442 [     main] result: true
resultValue1 = 1
17:48:20.442 [     main] getValue: 1
17:48:20.442 [     main] result: true
resultValue2 = 2
```

지금은 순서대로 실행되기 때문에, 결과는 다음과 같다.

incrementAndGet 첫 번째 실행

- `atomicInteger.get()` 을 사용해서 `value` 값을 읽는다. ➜ `getValue` 는 0이다.
- `compareAndSet(0,1)` 을 수행한다.
  - `compareAndSet(getValue, getValue + 1)`
- CAS 연산이 성공했으므로 `value` 값은 0에서 1로 증가하고 `true` 를 반환한다.
- `do~while` 문을 빠져나간다.

incrementAndGet 두 번째 실행

- `atomicInteger.get()` 을 사용해서 `value` 값을 읽는다. ➜ `getValue` 는 1이다.
- `compareAndSet(1,2)` 을 수행한다.
  - `compareAndSet(getValue, getValue + 1)`
- CAS 연산이 성공했으므로 `value` 값은 1에서 2로 증가하고 `true` 를 반환한다.
- `do~while` 문을 빠져나간다.

지금은 `main` 스레드 하나로 순서대로 실행되기 때문에 CAS 연산이 실패하는 상황을 볼 수 없다. 우리가 기대하는 실패하는 상황은 연산의 중간에 다른 스레드가 값을 변경해버리는 것이다. 멀티스레드로 실행해서 CAS 연산이 실패하는 경우에 어떻게 작동하는지 알아보자.

CAS 연산3

멀티스레드를 사용해서 중간에 다른 스레드가 먼저 값을 증가시켜 버리는 경우를 알아보자. 그리고 CAS 연산이 실패하는 경우에 어떻게 되는지 알아보자. 이 경우에도 값을 정상적으로 증가시킬 수 있을까?

```
package thread.cas;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.atomic.AtomicInteger;

import static util.MyLogger.log;
import static util.ThreadUtils.sleep;

public class CasMainV3 {

    private static final int THREAD_COUNT = 2;

    public static void main(String[] args) throws InterruptedException {
        AtomicInteger atomicInteger = new AtomicInteger(0);
```

```
        System.out.println("start value = " + atomicInteger.get());

        Runnable runnable = new Runnable() {
            @Override
            public void run() {
                incrementAndGet(atomicInteger);
            }
        };

        List<Thread> threads = new ArrayList<>();
        for (int i = 0; i < THREAD_COUNT; i++) {
            Thread thread = new Thread(runnable);
            threads.add(thread);
            thread.start();
        }

        for (Thread thread : threads) {
            thread.join();
        }

        int result = atomicInteger.get();
        System.out.println(atomicInteger.getClass().getSimpleName() + "
resultValue: " + result);

    }

    private static int incrementAndGet(AtomicInteger atomicInteger) {
        int getValue;
        boolean result;
        do {
            getValue = atomicInteger.get();
            sleep(100); // 스레드 동시 실행을 위한 대기
            log("getValue: " + getValue);
            result = atomicInteger.compareAndSet(getValue, getValue + 1);
            log("result: " + result);
        } while (!result);

        return getValue + 1;
    }
}
```

- 2개의 스레드가 `incrementAndGet()`를 호출해서 `AtomicInteger` 내부의 `value` 값을 동시에 하나씩 증가시킨다.
- 이때 두 스레드는 `incrementAndGet()` 메서드를 함께 호출한다. 여기서 스레드가 동시에 같은 값을 읽고

CAS를 수행하는 상황을 쉽게 만들기위해 중간에 `sleep()` 코드를 추가했다.

실행 결과

```
start value = 0
18:13:37.623 [ Thread-1] getValue: 0
18:13:37.623 [ Thread-0] getValue: 0
18:13:37.625 [ Thread-1] result: true
18:13:37.625 [ Thread-0] result: false
18:13:37.731 [ Thread-0] getValue: 1
18:13:37.731 [ Thread-0] result: true
AtomicInteger resultValue: 2
```

실행 결과를 보면 마지막에 `AtomicInteger`가 정상적으로 2 증가된 것을 확인할 수 있다.

두 스레드의 실행 결과를 분석해보자. 보기 쉽게 스레드 별로 로그를 나누었다.

Thread-1 실행

```
18:13:37.623 [ Thread-1] getValue: 0
18:13:37.625 [ Thread-1] result: true
```

- `atomicInteger.get()`을 사용해서 `value` 값을 읽는다. ➜ `getValue`는 0이다.
- `compareAndSet(0,1)`을 수행한다.
  - `compareAndSet(getValue, getValue + 1)`
- CAS 연산이 성공했으므로 `value` 값은 0에서 1로 증가하고 `true`를 반환한다.
- `do~while` 문을 빠져나간다.

Thread-0 실행

```
//[Thread-0] do~while 첫 번째 시도
18:13:37.623 [ Thread-0] getValue: 0
18:13:37.625 [ Thread-0] result: false

//[Thread-0] do~while 두 번째 시도
18:13:37.731 [ Thread-0] getValue: 1
18:13:37.731 [ Thread-0] result: true
```

[Thread-0] do~while 첫 번째 시도

- `atomicInteger.get()`을 사용해서 `value` 값을 읽는다. ➜ `getValue`는 0이다.
- `compareAndSet(0,1)`을 수행한다.
  - `compareAndSet(getValue, getValue + 1)`

- 그런데 `compareAndSet(0,1)` 연산은 실패한다.
  - CAS 연산에서 현재 `value` 값으로 0을 기대했지만 `Thread-1`이 중간에 먼저 실행되면서 `value`의 값을 0 ➜ 1로 변경해버렸다.
- CAS 연산이 실패했으므로 `value` 값은 변경하지 않고, `false`를 반환한다.
- 실패했으므로 `do~while` 문을 빠져나가지 못한다. `do~while`문을 다시 시작한다.
  - `while (!result)` ➜ `while(!false)` ➜ `while(true)` 이므로 다시 반복

[Thread-0] do~while 두 번째 시도

- `do~while`문이 다시 시작된다.
- `atomicInteger.get()`을 사용해서 `value` 값을 읽는다. ➜ `getValue`는 1이다.
- `compareAndSet(1,2)`을 수행한다.
  - `compareAndSet(getValue, getValue + 1)`
- CAS 연산이 성공했으므로 `value` 값은 1에서 2로 증가하고 `true`를 반환한다.
- `do~while` 문을 빠져나간다.

정리

`AtomicInteger`가 제공하는 `incrementAndGet()` 코드도 앞서 우리가 직접 작성한 `incrementAndGet()` 코드와 똑같이 CAS를 활용하도록 작성되어 있다. CAS를 사용하면 락을 사용하지 않지만, 대신에 다른 스레드가 값을 먼저 증가해서 문제가 발생하는 경우 루프를 돌며 재시도를 하는 방식을 사용한다.

이 방식은 다음과 같이 동작한다

1. 현재 변수의 값을 읽어온다.
2. 변수의 값을 1 증가시킬 때, 원래 값이 같은지 확인한다. (CAS 연산 사용)
3. 동일하다면 증가된 값을 변수에 저장하고 종료한다.
4. 동일하지 않다면 다른 스레드가 값을 중간에 변경한 것이므로, 다시 처음으로 돌아가 위 과정을 반복한다.

두 스레드가 동시에 실행되면서 문제가 발생하는 상황을 스레드가 충돌했다고 표현한다.
이 과정에서 충돌이 발생할 때마다 반복해서 다시 시도하므로, 결과적으로 락 없이 데이터를 안전하게 변경할 수 있다.
CAS를 사용하는 방식은 충돌이 드물게 발생하는 환경에서는 락을 사용하지 않으므로 높은 성능을 발휘할 수 있다. 이는 락을 사용하는 방식과 비교했을 때, 스레드가 락을 획득하기 위해 대기하지 않기 때문에 대기 시간과 오버헤드가 줄어드는 장점이 있다.

그러나 충돌이 빈번하게 발생하는 환경에서는 성능에 문제가 될 수 있다. 여러 스레드가 자주 동시에 동일한 변수의 값을 변경하려고 시도할 때, CAS는 자주 실패하고 재시도해야 하므로 성능 저하가 발생할 수 있다. 이런 상황에서는 반복문을 계속 돌기 때문에 CPU 자원을 많이 소모하게 된다.

CAS(Compare-And-Swap)와 락(Lock) 방식의 비교

락(Lock) 방식

- 비관적(pessimistic) 접근법
- 데이터에 접근하기 전에 항상 락을 획득
- 다른 스레드의 접근을 막음
- "다른 스레드가 방해할 것이다"라고 가정

CAS(Compare-And-Swap) 방식

- 낙관적(optimistic) 접근법
- 락을 사용하지 않고 데이터에 바로 접근
- 충돌이 발생하면 그때 재시도
- "대부분의 경우 충돌이 없을 것이다"라고 가정

정리하면 충돌이 많이 없는 경우에 CAS 연산이 빠른 것을 확인할 수 있다.
그럼 충돌이 많이 발생하지 않는 연산은 어떤 것이 있을까? 언제 CAS 연산을 사용하면 좋을까?
사실 간단한 CPU 연산은 너무 빨리 처리되기 때문에 충돌이 자주 발생하지 않는다. 충돌이 발생하기도 전에 이미 연산을 완료하는 경우가 더 많다.

앞서 여러 스레드가 `value++` 연산을 수행했던 `BasicInteger`, `VolatileInteger`의 예를 보자.
실행 결과

```
BasicInteger result: 950
VolatileInteger result: 961
SyncInteger result: 1000
MyAtomicInteger result: 1000
```

이 경우 최대한 많이 충돌하게 만들기 위해 1000개의 스레드를 동시에 쉬게 만든 다음에 동시에 실행했다.

```
sleep(10); //너무 빨리 실행되기 때문에, 다른 스레드와 동시 실행을 위해 잠깐 쉬었다가 실행
incrementInteger.increment();
```

`BasicInteger`의 실행 결과를 보면 최대한 스레드를 충돌하게 만들었는데도, 1000개 중에 약 50개의 스레드만 충돌한 사실을 확인할 수 있다.

락 방식

- 스레드 충돌을 방지하기 위해 1000개의 스레드가 모두 락을 획득하고 반환하는 과정을 거친다.
- 락을 사용하기 때문에 1000개의 스레드는 순서대로 하나씩 수행된다.
- 사실 이 중에 스레드가 충돌하는 경우는 50개의 경우 뿐이다.

CAS 방식

- 1000개의 스레드를 모두 한 번에 실행한다.
- 그리고 충돌이 나는 50개의 경우만 재시도 한다.

이 예제는 억지로 충돌을 만들기 위해서 `sleep(10)`을 넣었다. 만약 이 코드를 제거한다면 충돌 가능성은 100개 중에 1개도 안될 것이다.

정리하면 간단한 CPU 연산에는 락 보다는 CAS를 사용하는 것이 효과적이다.

CAS 락 구현1

CAS는 단순한 연산 뿐만 아니라, 락을 구현하는데 사용할 수도 있다.
`synchronized`, `Lock(ReentrantLock)` 없이 CAS를 활용해서 락을 구현해보자.
먼저 CAS의 필요성을 이해하기 위해 CAS 없이 직접 락을 구현해보자.

우선 코드를 먼저 작성하고 실행해보자.

```
package thread.cas.spinlock;

import static util.MyLogger.log;
import static util.ThreadUtils.sleep;

public class SpinLockBad {
    private volatile boolean lock = false;

    public void lock() {
        log("락 획득 시도");
        while(true) {
            if (!lock) { // 1. 락 사용 여부 확인
                sleep(100); // 문제 상황 확인용, 스레드 대기
                lock = true; // 2. 락의 값 변경
                break; // while 탈출
            } else {
                // 락을 획득할 때 까지 스핀 대기(바쁜 대기) 한다.
                log("락 획득 실패 - 스핀 대기");
            }
        }
        log("락 획득 완료");
    }
```

```
    public void unlock() {
        lock = false;
        log("락 반납 완료");
    }

}
```

구현 원리는 매우 단순하다.

- 스레드가 락을 획득하면 `lock`의 값이 `true`가 된다.
- 스레드가 락을 반납하면 `lock`의 값이 `false`가 된다.

- 스레드가 락을 획득하면 while문을 탈출한다.
- 스레드가 락을 획득하지 못하면 락을 획득할 때 까지 while문을 계속 반복 실행한다.

```
package thread.cas.spinlock;

import static util.MyLogger.log;
import static util.ThreadUtils.sleep;

public class SpinLockMain {
    public static void main(String[] args) {
        SpinLockBad spinLock = new SpinLockBad();

        Runnable task = new Runnable() {
            @Override
            public void run() {
                spinLock.lock();
                try {
                    // critical section
                    log("비즈니스 로직 실행");
                    //sleep(1); // 오래 걸리는 로직에서 스핀 락 사용X
                } finally {
                    spinLock.unlock();
                }
            }
        };

        Thread t1 = new Thread(task, "Thread-1");
        Thread t2 = new Thread(task, "Thread-2");

        t1.start();
        t2.start();
```

```
    }

}
```

실행 결과

```
09:58:35.387 [ Thread-1] 락 획득 시도
09:58:35.387 [ Thread-2] 락 획득 시도
09:58:35.388 [ Thread-1] 락 획득 완료
09:58:35.389 [ Thread-2] 락 획득 완료
09:58:35.389 [ Thread-1] 비즈니스 로직 실행
09:58:35.389 [ Thread-2] 비즈니스 로직 실행
09:58:35.389 [ Thread-1] 락 반납 완료
09:58:35.389 [ Thread-2] 락 반납 완료
```

실행 결과를 보면 기대와는 다르게 `Thread-1`, `Thread-2` 둘다 동시에 락을 획득하고 비즈니스 로직을 동시에 수행해버린다.
이제는 왜 이런 문제가 발생하는지 이제는 쉽게 이해할 수 있을 것이다.

스레드 둘이 동시에 수행되기 때문에 문제가 발생했다. 실행 결과를 분석해보자.

```
public void lock() {
    log("락 획득 시도");
    while(true) {
        if (!lock) { // 1. 락 사용 여부 확인
            sleep(100); // 문제 상황 확인용, 스레드 대기
            lock = true; // 2. 락의 값 변경
            break; // while 탈출
        } else {
            // 락을 획득할 때 까지 스핀 대기(바쁜 대기) 한다.
            log("락 획득 실패 - 스핀 대기");
        }
    }
    log("락 획득 완료");
}
```

실행 결과 분석

`lock`의 초기값은 `false`이다.
`Thread-1`과 `Thread-2`는 동시에 실행된다.

- `Thread-1` : `lock()`을 호출해서 락 획득을 시도한다.
- `Thread-2` : `lock()`을 호출해서 락 획득을 시도한다.

- `Thread-1` : `if (!lock)` 에서 락의 사용 여부를 확인한다. `lock` 의 값이 `false` 이다.
  - `if (!lock)` ➜ `if (!false)` ➜ `if (true)` 이므로 `if` 문을 통과한다.
- `Thread-2` : `if (!lock)` 에서 락의 사용 여부를 확인한다. `lock` 의 값이 `false` 이다.
  - `if (!lock)` ➜ `if (!false)` ➜ `if (true)` 이므로 `if` 문을 통과한다.

- `Thread-1` : `lock = true;` 를 호출해서 락의 값을 변경한다.
  - 이 시점에 `lock` 은 `false` ➜ `true` 가 된다.
- `Thread-2` : `lock = true;` 를 호출해서 락의 값을 변경한다.
  - 이 시점에 `lock` 은 `true` ➜ `true` 가 된다.

- `Thread-1` , `Thread-2` 둘다 `break;` 를 통해 while문을 탈출한다.
- `Thread-1` , `Thread-2` 둘다 락 획득을 완료하고, 비즈니스 로직을 수행한 다음에 락을 반납한다.

여기서 어떤 부분이 문제일까?
바로 다음 두 부분이 원자적이지 않다는 문제가 있다.

- **1. 락 사용 여부 확인**
- **2. 락의 값 변경**

이 둘은 한 번에 하나의 스레드만 실행해야 한다. 따라서 `synchronized` 또는 `Lock` 을 사용해서 두 코드를 동기화해서 안전한 임계 영역을 만들어야 한다.

여기서 다른 해결 방안도 있다. 바로 두 코드를 하나로 묶어서 원자적으로 처리하는 것이다.
CAS 연산을 사용하면 두 연산을 하나로 묶어서 하나의 원자적인 연산으로 처리할 수 있다.
락의 사용 여부를 확인하고, 그 값이 기대하는 값과 같다면 변경하는 것이다. 이것은 CAS 연산에 딱 들어 맞는다!

참고로 락을 반납하는 다음 연산은 연산이 하나인 원자적인 연산이다. 따라서 이 부분은 여러 스레드가 함께 실행해도 문제가 발생하지 않는다.

```
public void unlock() {
    lock = false; //원자적인 연산
    log("락 반납 완료");
}
```

CAS 락 구현2

이번에는 CAS를 사용해서 락을 구현해보자.

```
package thread.cas.spinlock;

import java.util.concurrent.atomic.AtomicBoolean;

import static util.MyLogger.log;

public class SpinLock {
    private final AtomicBoolean lock = new AtomicBoolean(false);

    public void lock() {
        log("락 획득 시도");
        while (!lock.compareAndSet(false, true)) {
            // 락을 획득할 때 까지 스핀 대기(바쁜 대기) 한다.
            log("락 획득 실패 - 스핀 대기");
        }
        log("락 획득 완료");
    }

    public void unlock() {
        lock.set(false);
        log("락 반납 완료");
    }

}
```

CAS 연산을 지원하는 `AtomicBoolean`을 사용했다.

구현 원리는 단순하다.

- 스레드가 락을 획득하면 `lock`의 값이 `true`가 된다.
- 스레드가 락을 반납하면 `lock`의 값이 `false`가 된다.

- 스레드가 락을 획득하면 while문을 탈출한다.
- 스레드가 락을 획득하지 못하면 락을 획득할 때 까지 while문을 계속 반복 실행한다.

락을 획득할 때 매우 중요한 부분이 있다. 바로 다음 두 연산을 하나로 만들어야 한다는 점이다.

- **1. 락 사용 여부 확인**
- **2. 락의 값 변경**

락을 획득하기 위해 먼저 락의 사용 여부를 확인했을 때 `lock`의 현재 값이 반드시 `false`여야 한다. `true`는 이미 다른 스레드가 락을 획득했다는 뜻이다. 따라서 이 값이 `false`일 때만 락의 값을 변경할 수 있다.
락의 값이 `false`인 것을 확인한 시점부터 `lock`의 값을 `true`로 변경할 때 까지 `lock`의 값은 반드시 `false`를 유지해야 한다.
중간에 다른 스레드가 `lock`의 값을 `true`로 변경하면 안된다. 그러면 여러 스레드가 임계 영역을 통과하는 동시성 문제가 발생한다.

CAS 연산은 이 두 연산을 하나의 원자적인 연산으로 만들어준다.
`lock.compareAndSet(false, true)`

- **1. 락 사용 여부 확인**: `lock`의 값이 `false`이면
- **2. 락의 값 변경**: `lock`의 값을 `true`로 변경해라.

SpinLockMain - 코드 변경

```
package thread.cas.spinlock;

import static util.MyLogger.log;
import static util.ThreadUtils.sleep;

public class SpinLockMain {
    public static void main(String[] args) {
        //SpinLockBad spinLock = new SpinLockBad();
        SpinLock spinLock = new SpinLock();
        ...
    }

}
```

- `SpinLockBad` 대신에 `SpinLock`을 사용하도록 코드를 변경하자.
- 참고로 예제를 단순화 하려고 인터페이스는 사용하지 않았다.

실행 결과

```
09:41:34.602 [ Thread-1] 락 획득 시도
09:41:34.602 [ Thread-2] 락 획득 시도
09:41:34.604 [ Thread-1] 락 획득 완료
09:41:34.604 [ Thread-2] 락 획득 실패 - 스핀 대기
09:41:34.604 [ Thread-1] 비즈니스 로직 실행
09:41:34.604 [ Thread-2] 락 획득 실패 - 스핀 대기
09:41:34.604 [ Thread-1] 락 반납 완료
09:41:34.605 [ Thread-2] 락 획득 완료
09:41:34.605 [ Thread-2] 비즈니스 로직 실행
09:41:34.605 [ Thread-2] 락 반납 완료
```

- 실행 결과는 환경에 따라 달라질 수 있다.

실행 결과를 보면 락이 잘 적용된 것을 확인할 수 있다.

실행 결과 분석

```
public void lock() {
    log("락 획득 시도");
    while (!lock.compareAndSet(false, true)) {
        // 락을 획득할 때 까지 스핀 대기(바쁜 대기) 한다.
        log("락 획득 실패 - 스핀 대기");
    }
    log("락 획득 완료");
}
```

Thread-1 로그

```
09:41:34.602 [ Thread-1] 락 획득 시도
09:41:34.604 [ Thread-1] 락 획득 완료
09:41:34.604 [ Thread-1] 비즈니스 로직 실행
09:41:34.604 [ Thread-1] 락 반납 완료
```

Thread-2 로그

```
09:41:34.602 [ Thread-2] 락 획득 시도
09:41:34.604 [ Thread-2] 락 획득 실패 - 스핀 대기
09:41:34.604 [ Thread-2] 락 획득 실패 - 스핀 대기
09:41:34.605 [ Thread-2] 락 획득 완료
09:41:34.605 [ Thread-2] 비즈니스 로직 실행
09:41:34.605 [ Thread-2] 락 반납 완료
```

`lock`의 초기값은 `false`이다.

- `Thread-1`: `lock()`을 호출해서 락 획득을 시도한다.
- `Thread-2`: `lock()`을 호출해서 락 획득을 시도한다.
- `Thread-1`: `while (!lock.compareAndSet(false, true))`를 사용해서 락의 사용 여부를 확인하면서 변경을 시도한다.
  - `lock.compareAndSet(false, true)`
    - CAS 연산을 사용했다. `lock`이 `false`면 `lock`의 값을 `true`로 변경한다.
    - `lock`의 값이 `false`이므로 `true`로 변경한다. 변경에 성공했기 때문에 `true`를 반환한다.
  - `while (!true)` ➜ `while(false)`가 되므로 while문을 빠져나온다.
  - 락 획득을 완료한다.
- `Thread-2`: `while (!lock.compareAndSet(false, true))`를 사용해서 락의 사용 여부를 확인하면

서 변경을 시도한다.

  - `lock.compareAndSet(false, true)`
    - CAS 연산을 사용했다. `lock`이 `false`면 `lock`의 값을 `true`로 변경한다.
    - `lock`의 값이 `true`이므로 값을 변경할 수 없다. 변경에 실패했기 때문에 `false`를 반환한다.
  - `while (!false)` ➜ `while(true)`가 되므로 while문을 시작한다.
  - 락 획득에 실패하고, 락을 획득할 때 까지 while문을 반복한다.
    - `락 획득 실패 - 스핀 대기` 로그가 반복해서 남는다.
- `Thread-1`: 비즈니스 로직을 수행하고 `lock.set(false)`을 수행해서 락을 반납한다.
- `Thread-2`: 락을 획득하고 while문을 탈출한다. 비즈니스 로직을 수행한 다음에 락을 반납한다.

기존 코드와 비교

기존 코드는 바로 다음 두 부분이 원자적이지 않다는 문제가 있었다.

- **1. 락 사용 여부 확인**
- **2. 락의 값 변경**

```
while(
    if (!lock) { //1. 락 사용 여부 확인
        lock = true; //2. 락의 값 변경
    }
)
```

이 문제를 CAS 연산을 통해서 원자적으로 바꾸었다.

```
while(lock.compareAndSet(false, true)) {}
```

CAS 연산 덕분에 원자적이지 않은 두 연산을 다음과 같이 하나의 원자적인 연산으로 바꿀 수 있었다.

- **1. 락을 사용하지 않는다면 락의 값을 변경**

원자적인 연산은 스레드 입장에서 쪼갤 수 없는 하나의 연산이다. 따라서 여러 스레드가 동시에 실행해도 안전하다.
이렇게 CAS를 사용해서 원자적인 연산을 만든 덕분에 무거운 동기화 작업 없이 아주 가벼운 락을 만들 수 있었다.
동기화 락을 사용하는 경우 스레드가 락을 획득하지 못하면 `BLOCKED`, `WAITING` 등으로 상태가 변한다. 그리고 또 대기 상태의 스레드를 깨워야 하는 무겁고 복잡한 과정이 추가로 들어간다. 따라서 성능이 상대적으로 느릴 수 있다. 반면에 CAS를 활용한 락 방식은 사실 락이 없다. 단순히 while문을 반복할 뿐이다. 따라서 대기하는 스레드도 `RUNNABLE` 상태를 유지하면서 가볍고 빠르게 작동할 수 있다.

CAS 단점

하지만 이렇게 반복문과 CAS를 사용해서 락을 대체하는 방식에도 단점이 있다.

`SpinLockMain` 코드에 있는 `sleep(1)` 주석을 풀고 실행해보자.

```
public void run() {
    spinLock.lock();
    try {
        // critical section
        log("비즈니스 로직 실행");
        sleep(1); // 오래 걸리는 로직에서 스핀 락 사용X
    } finally {
        spinLock.unlock();
    }
}
```

실행 결과

```
09:41:04.925 [ Thread-1] 락 획득 시도
09:41:04.925 [ Thread-2] 락 획득 시도
09:41:04.927 [ Thread-1] 락 획득 완료
09:41:04.927 [ Thread-1] 비즈니스 로직 실행
09:41:04.927 [ Thread-2] 락 획득 실패 - 스핀 대기
09:41:04.927 [ Thread-2] 락 획득 실패 - 스핀 대기
09:41:04.927 [ Thread-2] 락 획득 실패 - 스핀 대기
09:41:04.927 [ Thread-2] 락 획득 실패 - 스핀 대기
09:41:04.927 [ Thread-2] 락 획득 실패 - 스핀 대기
09:41:04.927 [ Thread-2] 락 획득 실패 - 스핀 대기
09:41:04.927 [ Thread-2] 락 획득 실패 - 스핀 대기
09:41:04.927 [ Thread-2] 락 획득 실패 - 스핀 대기
09:41:04.927 [ Thread-2] 락 획득 실패 - 스핀 대기
09:41:04.928 [ Thread-2] 락 획득 실패 - 스핀 대기
09:41:04.928 [ Thread-2] 락 획득 실패 - 스핀 대기
09:41:04.928 [ Thread-2] 락 획득 실패 - 스핀 대기
09:41:04.928 [ Thread-2] 락 획득 실패 - 스핀 대기
09:41:04.928 [ Thread-2] 락 획득 실패 - 스핀 대기
09:41:04.928 [ Thread-2] 락 획득 실패 - 스핀 대기
09:41:04.928 [ Thread-2] 락 획득 실패 - 스핀 대기
09:41:04.928 [ Thread-2] 락 획득 실패 - 스핀 대기
09:41:04.928 [ Thread-2] 락 획득 실패 - 스핀 대기
09:41:04.928 [ Thread-1] 락 반납 완료
09:41:04.928 [ Thread-2] 락 획득 완료
09:41:04.929 [ Thread-2] 비즈니스 로직 실행
09:41:04.930 [ Thread-2] 락 반납 완료
```

이 방식은 락을 기다리는 스레드가 `BLOCKED`, `WAITING` 상태로 빠지지는 않지만, `RUNNABLE` 상태로 락을 획득할 때 까지 while문을 반복하는 문제가 있다. 따라서 락을 기다리는 스레드가 CPU를 계속 사용하면서 대기하는 것이다!
`BLOCKED`, `WAITING` 상태의 스레드는 CPU를 거의 사용하지 않지만, `RUNNABLE` 상태로 while문을 반복 실행하는 방식은 CPU 자원을 계속해서 사용하는 것이다.

동기화 락을 사용하면 `RUNNABLE` 상태의 스레드가 `BLOCKED`, `WAITING` 상태에서 다시 `RUNNABLE` 상태로 이동한다. 이 사이에 CPU 자원을 거의 사용하지 않을 수 있다.
그래서 동기화 락을 사용하는 방식보다 스레드를 `RUNNABLE`로 살려둔 상태에서 계속 락 획득을 반복 체크하는 것이 더 효율적인 경우에 이런 방식을 사용해야 한다. 이 방식은 스레드의 상태가 변경되지 않기 때문에 매우 빠르게 락을 획득하고, 또 바로 실행할 수 있는 장점이 있다.

그럼 어떤 경우에 이런 방식이 효율적일까?
안전한 임계 영역이 필요하지만, 연산이 길지 않고 매우매우매우! 짧게 끝날 때 사용해야 한다.
예를 들어 숫자 값의 증가, 자료 구조의 데이터 추가와 같이 CPU 사이클이 금방 끝나는 연산에 사용하면 효과적이다.
반면에 데이터베이스의 결과를 대기한다거나, 다른 서버의 요청을 기다린다거나 하는 것 처럼 오래 기다리는 작업에 사용하면 CPU를 계속 사용하며 기다리는 최악의 결과가 나올 수도 있다.

스핀 락
스레드가 락이 해제되기를 기다리면서 반복문을 통해 계속해서 확인하는 모습이 마치 제자리에서 회전(spin)하는 것처럼 보인다. 그래서 이런 방식을 "스핀 락"이라고도 부른다. 그리고 이런 방식에서 스레드가 락을 획득 할 때 까지 대기하는 것을 스핀 대기(spin-wait) 또는 CPU 자원을 계속 사용하면서 바쁘게 대기한다고 해서 바쁜 대기(busy-wait)라 한다.
이런 스핀 락 방식은 아주 짧은 CPU 연산을 수행할 때 사용해야 효율적이다. 잘못 사용하면 오히려 CPU 자원을 더 많이 사용할 수 있다.
정리하면 "스핀 락"이라는 용어는, 락을 획득하기 위해 자원을 소모하면서 반복적으로 확인(스핀)하는 락 메커니즘을 의미한다. 그리고 이런 스핀 락은 CAS를 사용해서 구현할 수 있다.

정리

락 VS CAS 사용 방식

동기화 락(`synchronized`, `Lock(ReentrantLock)`)을 사용하는 방식과 CAS를 활용하는 락 프리 방식의 장단점을 비교해보자.

CAS의 장점

1. **낙관적 동기화**: 락을 걸지 않고도 값을 안전하게 업데이트할 수 있다. CAS는 충돌이 자주 발생하지 않을 것이라고 가정한다. 이는 충돌이 적은 환경에서 높은 성능을 발휘한다.
2. **락 프리(Lock-Free)**: CAS는 락을 사용하지 않기 때문에, 락을 획득하기 위해 대기하는 시간이 없다. 따라서 스레드가 블로킹되지 않으며, 병렬 처리가 더 효율적일 수 있다.

CAS의 단점

1. **충돌이 빈번한 경우**: 여러 스레드가 동시에 동일한 변수에 접근하여 업데이트를 시도할 때 충돌이 발생할 수 있다. 충돌이 발생하면 CAS는 루프를 돌며 재시도해야 하며, 이에 따라 CPU 자원을 계속 소모할 수 있다. 반복적인 재시도로 인해 오버헤드가 발생할 수 있다.
2. **스핀락과 유사한 오버헤드**: CAS는 충돌 시 반복적인 재시도를 하므로, 이 과정이 계속 반복되면 스핀락과 유사한 성능 저하가 발생할 수 있다. 특히 충돌 빈도가 높을수록 이런 현상이 두드러진다.

동기화 락의 장점

1. **충돌 관리**: 락을 사용하면 하나의 스레드만 리소스에 접근할 수 있으므로 충돌이 발생하지 않는다. 여러 스레드가 경쟁할 경우에도 안정적으로 동작한다.
2. **안정성**: 복잡한 상황에서도 락은 일관성 있는 동작을 보장한다.
3. **스레드 대기**: 락을 대기하는 스레드는 CPU를 거의 사용하지 않는다.

동기화 락의 단점

1. **락 획득 대기 시간**: 스레드가 락을 획득하기 위해 대기해야 하므로, 대기 시간이 길어질 수 있다.
2. **컨텍스트 스위칭 오버헤드**: 락을 사용하면, 락 획득을 대기하는 시점과 또 락을 획득하는 시점에 스레드의 상태가 변경된다. 이때 컨텍스트 스위칭이 발생할 수 있으며, 이로 인해 오버헤드가 증가할 수 있다.

결론

일반적으로 동기화 락을 사용하고, 아주 특별한 경우에 한정해서 CAS를 사용해서 최적화해야 한다.
CAS를 통한 최적화가 더 나은 경우는 스레드가 `RUNNABLE` ➜ `BLOCKED`, `WAITING` 상태에서 다시 `RUNNABLE` 상태로 가는 것 보다는, 스레드를 `RUNNABLE`로 살려둔 상태에서 계속 락 획득을 반복 체크하는 것이 더 효율적인 경우에 사용해야 한다. 하지만 이 경우 대기하는 스레드가 CPU 자원을 계속 소모하기 때문에 대기 시간이 아주아주아주 짧아야 한다. 따라서 임계 영역이 필요는 하지만, 연산이 길지 않고 매우매우매우! 짧게 끝날 때 사용해야 한다.
예를 들어 숫자 값의 증가, 자료 구조의 데이터 추가, 삭제와 같이 CPU 사이클이 금방 끝나지만 안전한 임계 영역, 또는 원자적인 연산이 필요한 경우에 사용해야 한다.
반면에 데이터베이스를 기다린다거나, 다른 서버의 요청을 기다리는 것 처럼 오래 기다리는 작업에 CAS를 사용하면 CPU를 계속 사용하며 기다리는 최악의 결과가 나올 수도 있다. 이런 경우에는 동기화 락을 사용해야 한다.

또한 CAS는 충돌 가능성이 낮은 환경에서 매우 효율적이지만, 충돌 가능성이 높은 환경에서는 성능 저하가 발생할 수 있다. 이런 경우에는 상황에 맞는 적절한 동기화 전략을 사용하는 것이 중요하다. 때로는 락이 더 나은 성능을 발휘할 수 있으며, CAS가 항상 더 빠르다고 단정할 수는 없다. 따라서, 각 접근 방식의 특성을 이해하고, 애플리케이션의 특정 요구사항과 환경에 맞는 방식을 선택하는 것이 필요하다.

실무 관점

실무 관점에서 보면 대부분의 애플리케이션들은 공유 자원을 사용할 때, 충돌할 가능성 보다 충돌하지 않을 가능성이 훨씬 높다. 예를 들어서 여러 스레드에서 발생하는 주문 수를 실시간으로 증가하면서 카운트 한다고 가정해보자. 그리고 특정 피크시간에 주문이 100만건 들어오는 서비스라고 가정해보자.

- 1,000,000 / 60분 = 1분에 16,666건, 1초에 277건

CPU가 1초에 얼마나 많은 연산을 처리하는지 생각해보면, 백만 건 중에 충돌이 나는 경우는 아주 넉넉하게 해도 몇 십 건 이하일 것이다. 따라서 실무에서는 주문 수 증가와 같은 단순한 연산의 경우, 락을 걸고 시작하는 것 보다는, CAS처럼 낙관적인 방식이 더 나은 성능을 보인다.

그런데 여기서 중요한 핵심은 주문 수 증가와 같은 단순한 연산이라는 점이다. 이런 경우에는 `AtomicInteger`와 같은 CAS 연산을 사용하는 방식이 효과적이다. 이런 연산은 나노 초 단위로 발생하는 연산이다.
반면에 데이터베이스를 기다린다거나, 다른 서버의 요청을 기다리는 것 처럼 수 밀리초 이상의 시간이 걸리는 작업이라면 CAS를 사용하는 것 보다 동기화 락을 사용하거나 스레드가 대기하는 방식이 더 효과적이다.

우리가 사용하는 많은 자바 동시성 라이브러리들, 동기화 컬렉션들은 성능 최적화를 위해 CAS 연산을 적극 활용한다. 덕분에 실무에서 직접 CAS 연산을 사용하는 사용하는 일은 매우 드물다. 대신에 CAS 연산을 사용해서 최적화 되어 있는 라이브러리들을 이해하고 편리하게 사용할 줄 알면 충분하다.

CAS의 개념을 알아두면 앞으로 멀티스레드와 관련된 다양한 라이브러리들을 분석할 때, "아~ 이 부분은 CAS를 사용해서 최적화 했구나"라는 점을 이해할 수 있을 것이다.

참고: CAS 연산은 심화 내용이다. 이해가 어렵다면 가볍게 듣고 넘어가도 괜찮다. 왜냐하면 우리가 직접 CAS 연산을 사용하는 경우는 거의 없기 때문이다. 대부분 복잡한 동시성 라이브러리들이 CAS 연산을 사용한다. 우리는 AtomicInteger와 같은 CAS 연산을 사용하는 라이브러리들을 잘 사용하는 정도면 충분하다.