Week4

CAS

- **원자적 연산**

더이상 쪼갤 수 없는 연산이다. **즉, 멀티스레드 상황에서도 안전하다**
예시) 필드에 값을 할당, 변수 증감, 비트 연산

i++ 은 원자적 연산이 아니다. => i의 값을 읽고, 더하고, 더한값을 대입하는 순서로 실행되기 때문

원자적이지 않은 연산은 임계영역이 생기게 되고 그부분을 안전하게 만들기 위해 synchronized를 적용할 수 있다.

이러한 연산을 제공하는 Atomic~ 클래스가 있으며 이는 직접 synchronized를 적용한것 보다 1.5배 ~ 2배 정도 성능적으로 좋다. 그이유는 락을 사용하지 않고 대신에 **CAS(Compare And Swap)연산**을 이용하기 때문이다.

CAS는 **compareAndSet()** 메소드를 사용하는데 이 메소드는 원자적 연산이다. 기대하는 값을 확인하는것과 변경하는 연산을 하드웨어 차원에서 하나의 연산으로 묶기때문에 논리적으로는 원자적 연산이 아닌것 처럼 보여도 실제로는 원자적 연산이다.

Lock방식에 비해 CPU연산을 계속 하기때문에(BLOCKED, WATING으로 들어가지 않음) 복잡한 연산이 있을땐 사용하지 않는게 좋다. sleep(1)만 들어가도 스핀락이 발생한다.

결론적으로 아주아주 가벼운 연산을 하는 특별한 경우에만 CAS를 사용해야한다. => 실제로 구현하기보단 AtomicInteger와 같은 CAS연산을 사용하는 라이브러리를 사용하면 된다.

Q. Atomic~ 클래스들이 CAS를 쓰는만큼 오버헤드가 크지 않을까?

동시성 컬렉션

자바에서 제공하는 컬렉션들은 대부분 원자적 연산이 아니다. 고로 동시성 문제에 직면하게된다. 이부분 역시 synchronized로 해결이 가능하지만 모든 컬렉션에 적용하기엔 어렵다.

- **proxy패턴 적용**

proxy란 대리자 라는 뜻으로 컬렉션의 synchronized 대신 처리 해주는 인터페이스를 만든다.

```java
public class SyncProxyList implements SimpleList {
    private SimpleList target;
    public SyncProxyList(SimpleList target) {
        this.target = target;
   }
    @Override
    public synchronized void add(Object e) {
        target.add(e);
    }
    @Override
    public synchronized Object get(int index) {
        return target.get(index);
    }
    @Override
    public synchronized int size() {
        return target.size();
    }
   @Override
   public synchronized String toString() {
       return target.toString() + " by " + this.getClass().getSimpleName();
   }
}
```

```java
SimpleList basicList = new BasicList();
SimpleList proxyList = new SyncProxyList(basicList);
```

이런식으로 기존 코드를 수정하지 않고 프록시를 통해 동기화 기능을 적용하는것이 가능하다.

- **프록시 방식의 단점**

1. 동기화에 대한 오버헤드
2. 스레드 대기상태가 빈번해짐
3. 과도한 동기화로 인해 특정부분에 대한 선택적 동기화가 어려움

- **라이브러리 활용**

Synchronized, lock, CAS를 다양하게 섞어서 정교하게 만들어 동기화에 대한 최적화가 적용한 동시성 컬렉션을 활용한다.

CopyOnWriteArrayList , ConcurrentHashMap 등 리스트, 맵에 대한 대안으로 사용가능한 동시성 컬렉션이 있다.