4주차 스터디

목차

- 원자적 연산
- CAS 연산
- 락 VS CAS
- 동시성 컬렉션

원자적 연산

- 멀티스레드에서 설명하는 원자적 연산(atomic operation)은 중간에 다른 스레드의 간섭 없이 하나의 작업 단위로 수행되는 연산
- 쉽게 이야기해서 원자적 연산은 중간에 끼어들 수 없는 단일 단위의 작업을 의미합니다. 멀티스레딩 환경에서 다른 스레드의 간섭 없이 완전히 실행되는 연산

`i = 1`

- 오른쪽에 있는 1 의 값은 왼쪽의 i 변수에 대입한다.

`i = i + 1;`

- 오른쪽에있는 i의 값을 읽는다. i의 값은 10이다.
- 읽은 10에 1을 더해서 11을 만든다.
- 더한 11을 왼쪽의 i 변수에 대입한다.

`int count = max`

- max가 상수이거나, 지역 변수라면 다른 스레드가 간섭할 수 없기 때문에 원자적 연산
- max가 인스턴스 변수이거나 클래스 변수라면 다른 스레드가 간섭할 수 있기 때문에 원자적 연산이 아님

핵심은 다른 스레드가 해당 연산에 영향을 줄 수 있는가로 판단

원자적 연산

```java
public class BasicInteger implements IncrementInteger {

    private int value;

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

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

- 실행 결과를 보면 기대한 1000 이 아닌 조금 더 작은 숫자가 출력
- 여러 스레드가 동시에 원자적이지 않은 value++ 을 호출했기 때문에 발생

```java
public class SyncInteger implements IncrementInteger {

    private int value;

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

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

- 연산 자체가 나누어진 경우에는 synchronized 블럭이나 Lock 등을 사용
- 안전한 임계 영역을 만들어야 함

```java
public class VolatileInteger implements IncrementInteger
{

    volatile private int value;

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

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

- volatile은 캐시 메모리를 무시하는 기능
- volatile은 연산 차제를 원자적으로 묶어주는 기능이 아니다.

```java
public class MyAtomicInteger implements IncrementInteger {

    AtomicInteger atomicInteger = new AtomicInteger(0);

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

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

- 멀티스레드 상황에서 안전하게 값 증가 및 감소 연산을 수행할 수 있도록 동시성 코드가 내부에 구현되어 있음
- synchronized나 락을 직접 사용할 필요가 없음
- 여러 스레드가 값을 공유해야 할 때 안전하게 사용할 수 있으며, 다양한 값 증가와 감소 연산을 제공

원자적 연산 - 성능 테스트

클래스	동시성 안전성	캐시 사용	멀티스레드 안전	성능
BasicInteger	아니오	CPU 캐시 사용	아니오	단일 스레드에서 가장 빠름
VolatileInteger	아니오	메인 메모리 사용	아니오	BasicInteger보다 느림
SyncInteger	예	해당 없음	예	MyAtomicInteger보다 느림
MyAtomicInteger	예	해당 없음	예	synchronized/락 사용보다 1.5~2배 빠름

CAS 연산

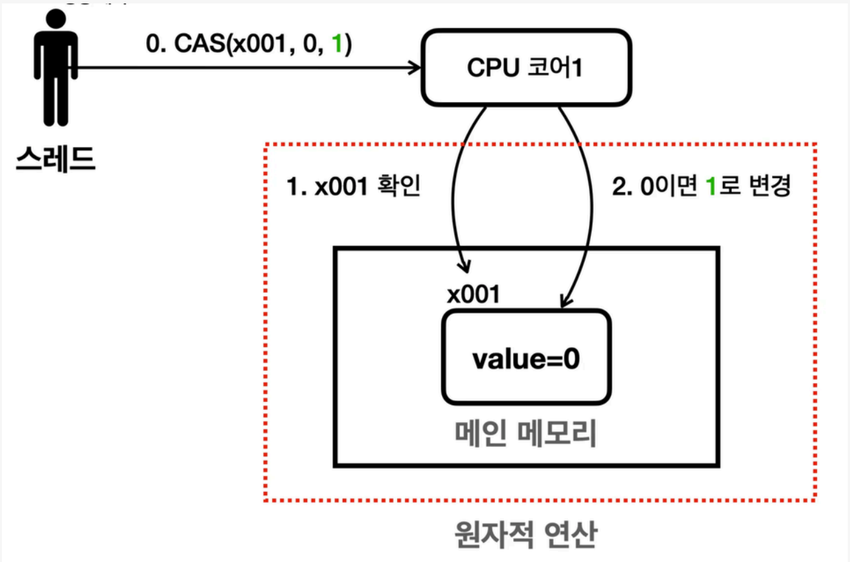
락 기반 방식의 문제점

- 락을 사용하는 방식은 상대적으로 무거운 방식
- 락을 얻고 반납하고 과정
- CPU 실행 스케줄러에서 빠졌다가 다시 들어가는 과정

CAS 연산은 락을 사용하는 방법이 아닌 또 다른 특별한 패러다임을 사용하는 방법

- 락을 완전히 대체하는 것은 아니고, 작은 단위의 일부 영역에 적용

CAS 연산



compareAndSet(0, 1)을 호출

- 매개변수의 왼쪽이 기대하는 값, 오른쪽이 변경하는 값
- 현재 value 값이 0이면 1로 변경하고 true를 반환
- 현재 value 값이 0이 아니면 값을 바꾸지 않고 false를 반환
- 값을 읽고 변경하는게 원자적으로 실행됨

CAS 연산

```
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;
}
```

동작 방식

1. 현재 변수의 값을 읽어온다.

2. 변수의 값을 1증가 시킬때, 원래 값이 같은지 확인한다.(CAS연산사용)

3. 동일하다면 증가된 값을 변수에 저장하고 종료한다.

4. 동일하지 않다면 다른 스레드가 값을 중간에 변경한 것이므로, 다시 처음으로 돌아가 위 과정을 반복한다.

CAS 연산

CAS(Compare-And-Swap)와 락(Lock) 방식의 비교

구분	락(Lock) 방식	CAS(Compare-And-Swap) 방식
접근 방식	비관적 접근법 ("다른 스레드가 방해할 것이다" 가정)	낙관적 접근법 ("대부분의 경우 충돌이 없을 것이다" 가정)
동작 방식	항상 락을 획득 후 데이터 접근, 다른 스레드의 접근 차단	락 없이 바로 데이터 접근, 충돌 발생 시 재시도
충돌 처리 방식	스레드 충돌 방지, 락을 사용해 순서대로 처리	충돌 시 재시도, 충돌이 적을 경우 빠르게 처리
성능	스레드가 락을 획득하고 반환하는 과정으로 인해 대기 시간 발생	충돌이 적은 경우 높은 성능 발휘, 대기 시간 감소
단점	대기 시간과 오버헤드 발생	충돌이 빈번한 경우 성능 저하, CPU 자원 소모 증가
예시 상황	1000개 스레드가 모두 순서대로 실행, 충돌 없음	1000개 스레드 중 50개만 충돌, 나머지는 빠르게 처리
적용 환경	충돌이 빈번한 환경에 적합	충돌이 드문 환경에 적합

CAS 연산 - 락 방식

기존 코드

```
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;
            } else {
                // 락을 획득할 때 까지 스핀 대기(바쁜 대기) 한다.
                log("락 획득 실패 - 스핀 대기");
            }
        }
        log("락 획득 완료");
    }

            .........
}
```

compareAndSet

```
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");
    }

            .......
}
```

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

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

락 VS CAS

구분	CAS(Compare-And-Swap)	동기화 락(Synchronized Lock)
장점	낙관적 동기화: 충돌이 적은 환경에서 높은 성능 발휘	충돌 관리: 하나의 스레드만 접근 가능, 충돌 없음
	락 프리(Lock-Free): 락 대기가 없어 병렬 처리 효율적	안정성: 복잡한 상황에서도 일관성 있는 동작 보장
		스레드 대기: 대기 중 CPU 사용을 최소화
단점	충돌 빈번 시 문제: 충돌 발생 시 반복적인 재시도로 CPU 자원 소모	락 획득 대기 시간: 대기 시간으로 인한 성능 저하 가능
	스핀락과 유사한 오버헤드: 충돌 빈도가 높을 경우 성능 저하	컨텍스트 스위칭 오버헤드: 락 획득 대기 중 컨텍스트 스위칭으로 인한 오버헤드 발생

동시성 컬렉션

ArrayList의 add 메서드

```
private void add(E e, Object[] elementData, int s) {
    if (s == elementData.length)
        elementData = grow();
    elementData[s] = e;
    size = s + 1;
}
```

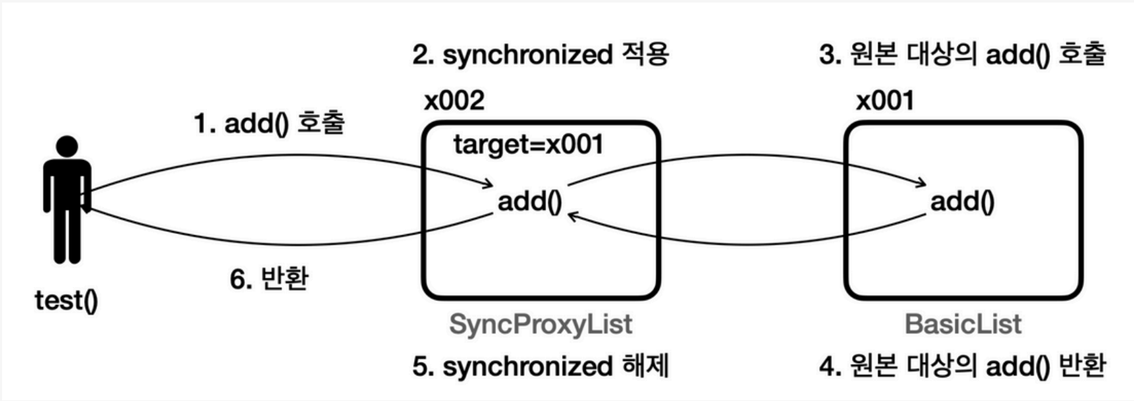
synchronized 추가

```
public class SyncList implements SimpleList {

    private static final int DEFAULT_CAPATICY = 5;

    private Object[] elementData;
    private int size = 0;

    public SyncList() {
        elementData = new Object[DEFAULT_CAPATICY];
    }

    @Override
    public synchronized int size() {
        return size;
    }

    @Override
    public synchronized void add(Object e) {
        elementData[size] = e;
        sleep(100); // 멀티스레드 문제를 쉽게 확인하는 코드
        size++;
    }

    @Override
    public synchronized Object get(int index) {
        return elementData[index];
    }

    @Override
    public synchronized String toString() {
        return Arrays.toString(Arrays.copyOf(elementData, size)) +
                " size=" + size + ", capaity=" + elementData.length;
    }
}
```

동시성 컬렉션 - 프록시 도입

```java
public class SyncProxyList implements SimpleList {

    private SimpleList target;

    public SyncProxyList(SimpleList target) {
        this.target = target;
    }

    @Override
    public synchronized int size() {
        return target.size();
    }

    @Override
    public synchronized void add(Object e) {
        target.add(e);
    }

    @Override
    public synchronized Object get(int index) {
        return target.get(index);
    }

    @Override
    public String toString() {
        return target.toString() + " by " + this.getClass().getSimpleName();
    }
}
```

SyncProxyList - add() 호출 과정



동시성 컬렉션 - 프록시 도입

new SynchronizedRandomAccessList<>(new ArrayList())

`SynchronizedRandomAccessList`는 `synchronized`를 추가하는 프록시 역할을 한다.

- 클라이언트 ➜ `ArrayList`
- 클라이언트 ➜ `SynchronizedRandomAccessList`(프록시) ➜ `ArrayList`

- synchronizedList()
- synchronizedCollection()
- synchronizedMap()
- synchronizedSet()
- synchronizedNavigableMap()
- synchronizedNavigableSet()
- synchronizedSortedMap()
- synchronizedSortedSet()

synchronized 프록시 방식의 단점

구분	설명
동기화 오버헤드	synchronized 키워드는 안전한 접근을 보장하지만, 각 메서드 호출 시 동기화 비용이 추가되어 성능 저하를 유발할 수 있음
잠금 범위의 확장	컬렉션 전체에 동기화가 이루어져 잠금 경합(lock contention)이 증가하고, 병렬 처리의 효율성이 저하될 수 있음
정교한 동기화 불가	컬렉션 전체에 대한 동기화만 가능하며, 특정 부분이나 메서드에 선택적으로 동기화를 적용하기 어려워 과도한 동기화로 이어질 수 있음

동시성 컬렉션

동시성 컬렉션의 종류

구분	컬렉션	대안 및 특징
List	CopyOnWriteArrayList	ArrayList의 대안, 쓰기 작업이 적고 읽기 작업이 많은 경우 유리
Set	CopyOnWriteArraySet	HashSet의 대안, 쓰기 작업이 적고 읽기 작업이 많은 경우 유리
	ConcurrentSkipListSet	TreeSet의 대안, 정렬된 순서 유지, Comparator 사용 가능
Map	ConcurrentHashMap	HashMap의 대안, 동시성에 안전한 맵
	ConcurrentSkipListMap	TreeMap의 대안, 정렬된 순서 유지, Comparator 사용 가능
Queue	ConcurrentLinkedQueue	비 차단(non-blocking) 동시성 큐
Deque	ConcurrentLinkedDeque	비 차단(non-blocking) 동시성 데크

블로킹 큐

큐 종류	설명
ArrayBlockingQueue	크기가 고정된 블로킹 큐. 공정(fair) 모드를 사용할 수 있지만 성능 저하 가능
LinkedBlockingQueue	크기가 무한하거나 고정된 블로킹 큐
PriorityBlockingQueue	우선순위가 높은 요소를 먼저 처리하는 블로킹 큐
SynchronousQueue	데이터를 저장하지 않으며, 생산자가 데이터를 추가하면 소비자가 받을 때까지 대기. 생산자-소비자 간의 직접적인 거래 (핸드오프) 방식
DelayQueue	지연된 요소를 처리하는 블로킹 큐. 지정된 지연 시간이 지난 후 요소가 소비되며, 스케줄링 작업에 적합

동시성 컬렉션

SynchronizedList

```
static class SynchronizedList<E>
    extends SynchronizedCollection<E>
    implements List<E> {
    @java.io.Serial
    private static final long serialVersionUID = -
7754090372962971524L;

    @SuppressWarnings("serial") // Conditionally serializable
    final List<E> list;

    SynchronizedList(List<E> list) {
        super(list);
        this.list = list;
    }
    SynchronizedList(List<E> list, Object mutex) {
        super(list, mutex);
        this.list = list;
    }

    public boolean equals(Object o) {
        if (this == o)
            return true;
        synchronized (mutex) {return list.equals(o);}
    }
    public int hashCode() {
        synchronized (mutex) {return list.hashCode();}
    }

    public E get(int index) {
        synchronized (mutex) {return list.get(index);}
    }
    public E set(int index, E element) {
        synchronized (mutex) {return list.set(index, element);}
    }
    public void add(int index, E element) {
        synchronized (mutex) {list.add(index, element);}
    }

    ...
}
```

CopyOnWriteArrayList

```
public class CopyOnWriteArrayList<E>
    implements List<E>, RandomAccess, Cloneable, java.io.Serializable
{
    private static final long serialVersionUID = 8673264195747942595L;
final transient Object lock = new Object();

    private transient volatile Object[] array;


    final Object[] getArray() {
        return array;
    }

...

public E get(int index) {
    return elementAt(getArray(), index);
}

public E set(int index, E element) {
    synchronized (lock) {
        Object[] es = getArray();
        E oldValue = elementAt(es, index);

        if (oldValue != element) {
            es = es.clone();
            es[index] = element;
        }

        setArray(es);
        return oldValue;
    }
}
public boolean add(E e) {
    synchronized (lock) {
        Object[] es = getArray();
        int len = es.length;
        es = Arrays.copyOf(es, len + 1);
        es[len] = e;
        setArray(es);
        return true;
    }
}
```

감사합니다.