스레드 동기화

- 스레드 고유의 리소스(**공유하지 않는 변수**)에만 접근 시 동기화 문제는 생기지 않음
- 두 개 이상의 스레드가 **같은 공유 자원(같은 변수, 객체)에 접근**하더라도 **읽기 작업만 진행**하면 문제는 생기지 않음
- **두 개 이상의 스레드가 같은 공유 자원(같은 변수, 객체)에 접근하는데 쓰기, 수정 작업이 진행될 경우에는 동기화 문제가 발생**할 수 있음
    - 즉, **값이 변화될 수 있는 여지**가 있으면 문제가 생길 수 있음
        - 이를 **스레드 안전(thread-safe) 하지 않다고 표현**하며 스레드 안전하지 않은 객체는 여러 스레드에서 동시 접근할 경우 동기화 문제가 발생 가능
    - 따라서 **한 시점에 한 스레드만 특정 자원을 쓸 수 있도록 synchronized 키워드를 통해 동기화하거나 락(Lock) 객체를 이용한 잠그기의 과정이 필요**함

원자성(Atomicity)의 개념

- 원자(atom) => 더 이상 분해할 수 없는 물리적 최소 단위
- **원자적(atomic)**인 명령어 => 더 이상 **쪼갤 수 없는 최소 단위**의 명령어
- 아래와 같은 명령어는 원자적인 명령어인지 생각해보기

```
i++;
```

- 위와 같이 명령어는 한 줄이지만 **실제로 수행되는 명령어(기계어)는 여럿일 수 있음을 주의**해야 함

```
// i++; 의 실제 수행 명령어
// (실제로는 3단계(read-modify-write)의 연산 작업 진행)
int tmp = i;    // read
tmp = tmp + 1;  // modify
i = tmp;        // write
```

스레드의 동시 접근이 문제되는 사례 살펴보기

```
// add 메서드에서 수행하는 동작
// 1. value를 읽음 (read)
// 2. value에 5를 더함 (modify)
// 3. value값에 변화된 값을 씀 (write)
public void add() {
    value = value + 5;
}
```

```
// 위의 메서드 좀 더 읽기 편하게 쓴 버전
public void add() {
    int r = value; // 1. 읽기
    value = r + 5; // 2. (수정 및) 쓰기
}
```

- **FACT)** 스레드의 실행 순서는 **특정한 순서로 진행이 보장되지 않음**
    - 스레드 스케쥴러는 **운영체제에서 관리**하므로 프로그래머가 **직접 컨트롤** 할 수 없음 (우선 순위 지정만 가능하며 우선 순위를 무조건 존중하지도 않음)
    - 즉, **"경쟁 상태"가 발생**할 수 있음

```
공학 분야에서 경쟁 상태(race condition)란 둘 이상의 입력 또는 "조작의 타이밍이나 순서"등이 결과값에 영향을 줄 수 있는 상태를 말한다.
```

- **Q)** 그렇다면 add 메서드를 **2개의 스레드**가 접근할 때 가능한 모든 시나리오는?

가능한 모든 시나리오 (트리의 형태로 그려보기)

```
- 스레드1 읽기 - 스레드 1 쓰기 - 스레드 2 읽기 - 스레드 2 쓰기
- 스레드1 읽기 - 스레드 2 읽기 - 스레드 1 쓰기 - 스레드 2 쓰기
- 스레드1 읽기 - 스레드 2 읽기 - 스레드 2 쓰기 - 스레드 1 쓰기
- 스레드2 읽기 - 스레드 2 쓰기 - 스레드 1 읽기 - 쓰레드 1 쓰기
- 스레드2 읽기 - 스레드 1 읽기 - 스레드 1 쓰기 - 쓰레드 2 쓰기
- 스레드2 읽기 - 스레드 1 읽기 - 스레드 2 쓰기 - 쓰레드 1 쓰기
```

초기 변수값이 0일 때 각 시나리오에 따른 최종 결과값 따져보기

```
- 스레드1 읽기 - 스레드 1 쓰기 - 스레드 2 읽기 - 스레드 2 쓰기
  - 0(1) → 5(1) → 5(2) → 10(2) → 최종값 10

- 스레드1 읽기 - 스레드 2 읽기 - 스레드 1 쓰기 - 스레드 2 쓰기
  - 0(1) → 0(2) → 5(1) → 5(2) → 최종값 5

- 스레드1 읽기 - 스레드 2 읽기 - 스레드 2 쓰기 - 스레드 1 쓰기
  - 0(1) → 0(2) → 5(2) → 5(1) → 최종값 5

- 스레드2 읽기 - 스레드 2 쓰기 - 스레드 1 읽기 - 쓰레드 1 쓰기
  - 0(2) → 5(2) → 5(1) → 10(1) → 최종값 10

- 스레드2 읽기 - 스레드 1 읽기 - 스레드 1 쓰기 - 쓰레드 2 쓰기
  - 0(2) → 0(1) → 5(1) → 5(2) → 최종값 5

- 스레드2 읽기 - 스레드 1 읽기 - 스레드 2 쓰기 - 쓰레드 1 쓰기
  - 0(2) → 0(1) → 5(2) → 5(1) → 최종값 5
```

- add 메서드 블록의 내부 명령어가 아래와 같은 순서로 실행되는 경우에만 **기대하는 값**을 얻을 수 있음
    - **스레드1 읽기 - 스레드 1 쓰기 - 스레드 2 읽기 - 스레드 2 쓰기**
    - **스레드2 읽기 - 스레드 2 쓰기 - 스레드 1 읽기 - 쓰레드 1 쓰기**
- 스레드 실행 시나리오가 6개인데 그 중 **2개만 적절**함
    - 그렇다면 6개 중 4개가 이상한 시나리오이므로 실패 확률은? (4/6이므로 66%)
    - 하지만 실제로는 **정상적인 시나리오로 실행할 확률이 높음**
        - 그래서 이상적인 값에 근접하는 결과를 얻을 수 있는 것, 단, **시행 횟수가 많아지면** 거의 반드시 문제가 일어남
        - 스레드를 실행시킬 때 너무 빨리 **문맥 전환(context switching)을 시키는 것은 높은 간접 비용(overhead)을 발생**시키므로 가능한 일을 많이 수행하고 문맥 전환을 하는게 유리하기 때문에 한 명령어만 수행하고 문맥 전환이 이루어지는 일은 잘 일어나지 않음

        - 단, 스레드의 문맥 전환이 이루어지는 시점을 프로그래머가 직접 제어할 수는 없음을 항상 유의해야 함

- 2개의 명령어 2개의 스레드의 접근만으로도 시나리오는 충분히 복잡함
- 그런데 내부에서 수행하는 **명령어 수가 많아진다면?**
    - 동시에 그 명령어를 실행시키는 **스레드의 개수도 많아진다면?**
    - 실행 순서에 따르는 경우의 수는 **기하급수적으로 복잡**해지고 그 중 문제를 일으킬 수 있는 **시나리오를 찾는 작업은 거의 불가능**에 가까워짐

해결 방법

- 해결 방법?
    - **동기화 (synchronization) 메서드 혹은 블록을 통해서 명령어들의 수행 순서를 동기화하여 여러 명령어들이 원자적으로 동작할 수 있도록 조정**하기

synchronized 키워드를 이용한 동기화

1. **메소드 레벨** 동기화 (모니터 객체는 **this** 객체를 사용)
2. 직접 모니터 객체를 지정하여 **블록 레벨** 동기화 (메소드 레벨 동기화에 비해 복잡)

모니터 객체

- 자바의 모든 객체는 "모니터 객체"로 사용될 수 있음
- 여기서 모니터 객체는 일종의 열쇠로 이해할 수 있음
- 한 스레드가 모니터 객체에 대한 **소유권을 얻게 되면 그 소유권을 포기할 때까지 다른 스레드에서는 접근할 수 없게 됨**
- **메서드 레벨 동기화를 할 경우 this가 자동으로 모니터 객체로 사용됨**
    - 이 경우 메서드로 진입하며 this 모니터 객체에 대한 소유권을 얻게 되고, 메서드 호출이 끝나는 시점에 this 모니터 객체에 대한 소유권을 포기함
- **블록 레벨 동기화를 할 경우 직접 모니터 객체를 지정**해주어야 함
    - 이 경우 블록으로 진입하며 직접 지정한 모니터 객체에 대한 소유권을 얻게 되고, 블록을 빠져나가는 시점에 해당 모니터 객체에 대한 소유권을 포기함

메소드 레벨 동기화

- synchronized 키워드를 사용하여 **메소드 전체**를 동기화

SynchronizedDemo1.java

```
class SharedCounter {
    public long count = 0;

    // 동기화되어 경쟁 조건이 발생하지 않는 메서드
    /*
    public synchronized void increase() {
        this.count++;
```

```
    }
    */

    // 동기화되지 않은 경쟁 조건이 발생할 수 있는 메서드
    public void increase() {
        this.count++;
    }
}

class IncrementRunnable implements Runnable {
    private SharedCounter counter;
    private String name;

    public IncrementRunnable(SharedCounter counter, String name) {
        this.counter = counter;
        this.name = name;
    }

    @Override
    public void run() {
        // 두 스레드에서 100번 반복하며 1씩 증가시키므로, 정상적인 기대값은 200
        int loop = 1_00;
        // 루프 횟수를 늘렸으므로, 기대값은 2백만 (실제 관측한 결과는?)
        // int loop = 1_000_000;

        for(int i=0;i<loop;i++) {
            this.counter.increase();
        }
    }
}

public class SynchronizedDemo1 {
    public static void main(String[] args) throws InterruptedException {
        SharedCounter counter = new SharedCounter();
        Thread t1 = new Thread(new IncrementRunnable(counter, "1"));
        Thread t2 = new Thread(new IncrementRunnable(counter, "2"));
        t1.start();
        t2.start();
        // 스레드 종료될때까지 기다리기 위해서 join 메서드 호출 (main 스레드는 waiting
상태가 됨)
        t1.join();
        t2.join();
        // 최종 결과값 확인
        System.out.println("count result : " + counter.count);
    }
}
```

동기화 메서드에서의 스레드 실행 순서 시나리오

- 스레드가 **메서드에 진입시 lock**을 걸고 메서드의 모든 코드를 실행하고 **나갈 때 unlock**함
    - 화장실에 들어갈 때 문 잠그고 볼일 보고 다시 문 열고 나와야 다른 사람이 쓸 수 있음
    - lock이 걸려있는 메서드에는 다른 스레드를 진입할 수 없음 (메서드 진입 대기를 위해서 **BLOCKED 상태**로 변함)
- 시나리오

    - **스레드1 add 메서드 lock - 스레드 1 읽기 - 스레드 1 쓰기 - 스레드1 add 메서드 unlock - 스레드2 add 메서드 lock - 스레드 2 읽기 - 스레드 2 쓰기 - 스레드2 add 메서드 unlock**
    - **스레드2 add 메서드 lock - 스레드 2 읽기 - 스레드 2 쓰기 - 스레드2 add 메서드 unlock - 스레드1 add 메서드 lock - 스레드 1 읽기 - 스레드 1 쓰기 - 스레드1 add 메서드 unlock**
        - 수행 순서가 동기화되었으므로 자원 공유 및 수정과 관련된 문제 발생 없음
- 다른 스레드가 접근 시도할 경우의 시나리오 표로 살펴보기

메서드 진입 및 잠금 과정	**작업 과정**
스레드1 add 메서드 lock	
(다른 스레드에서 접근 시도할 경우, 해당 스레드는 BLOCKED 상태로 진입하여 대기)	스레드 1 읽기
(다른 스레드에서 접근 시도할 경우, 해당 스레드는 BLOCKED 상태로 진입하여 대기)	스레드 1 쓰기
스레드 1 add 메서드 unlock	
스레드 2(혹은 다른 대기하고 있던 스레드) add 메서드 lock	
(다른 스레드에서 접근 시도할 경우, 해당 스레드는 BLOCKED 상태로 진입하여 대기)	스레드 n 읽기
(다른 스레드에서 접근 시도할 경우, 해당 스레드는 BLOCKED 상태로 진입하여 대기)	스레드 n 쓰기
add 메서드 unlock	
...	...

블록 레벨 동기화

- 성능 향상을 위해서 synchronized 키워드를 사용하여 **메서드의 일부 블록**만 동기화
- 메서드의 **내용 전체를 블록 레벨 동기화**하고 **모니터 객체로 this를 사용**하는 경우 **메서드 레벨의 동기화와 완전히 같게 동작함**

```
// 메서드의 모든 내용(전체 블록)을 동기화시키고 그 과정에서 this를 모니터 객체로 사용
public synchronized void add() {
    value += 5;
}
```

- 위의 메서드는 아래의 메서드와 완전히 같은 방식으로 동기화 됨

```
// 메서드의 전체 내용을 블록 레벨 동기화시켰고 this를 모니터 객체로 사용하므로 완전히 같음
public void add() {
    synchronized(this) {
        value += 5;
    }
}
```

- 블록 레벨 동기화는 메서드의 **내용 전체를 동기화** 할 필요가 없어서 일부만 동기화하여 **성능 향상을 가져오고 싶을 때** 사용함
    - 메서드 내에 스레드가 서로 공유하는 자원도 있지만 그렇지 않고 (같은 메모리 주소를 사용하지 않는) **독점적으로 사용하는 자원에 대한 접근도 포함**되어 있을 경우 **서로 공유하는 자원에 대해서만** 블록 레벨 동기화를 수행

SynchronizedDemo2.java (동기화를 하지 않는 코드, 문제점 살펴보기)

```
class MapManipulateRunnable implements Runnable {
    private HashMap<String, String> hashMap;
    private String name;

    public MapManipulateRunnable(HashMap hashMap, String name) {
        this.hashMap = hashMap;
        this.name = name;
    }

    @Override
    public void run() {
        for(int i=0;i<10_000;i++) {
            // 어떤 종류의 문제가 발생할 지 생각해보기
            if (hashMap.containsKey("key")) {
                String value = hashMap.remove("key");
                if (value == null) {
                    System.out.println("[" + name + "] " + "this will not
happens." + i);
                }
            } else {
                hashMap.put("key", "hello");
            }
        }
    }
}

public class SynchronizedDemo2 {
    public static void main(String[] args) {
        HashMap<String, String> shared = new HashMap<>();
        MapManipulateRunnable runnable1 = new MapManipulateRunnable(shared,
"1");
        MapManipulateRunnable runnable2 = new MapManipulateRunnable(shared,
"2");
        new Thread(runnable1).start();
        new Thread(runnable2).start();
    }
}
```

블록 레벨 동기화 코드 삽입

```
@Override
public void run() {
    for(int i=0;i<10_000;i++) {
        // 동기화 시작
        synchronized (hashMap) {
            // 중괄호 블록 내부의 코드는 모두 원자적으로 실행됨
```

```
            if (hashMap.containsKey("key")) {
                String value = hashMap.remove("key");
                if (value == null) {
                    System.out.println("[" + name + "] " + "this will not
happens." + i);
                }
            } else {
                hashMap.put("key", "hello");
            }
        }
        // 동기화 끝
    }
}
```

Q1) run 메서드를 다음과 같이 메서드 단위로 동기화하는 것과 위의 코드의 동작상의 차이점 생각해보기

```
@Override
public synchronized void run() {
    for(int i=0;i<10_000;i++) {
        if (hashMap.containsKey("key")) {
            String value = hashMap.remove("key");
            if (value == null) {
                System.out.println("[" + name + "] " + "this will not happens."
+ i);
            }
        } else {
            hashMap.put("key", "hello");
        }
    }
}
```

Q2) 위와 같이 코드를 수정하고 실행해도, 여전히 동기화 문제가 발생하는 이유 생각해보기 (힌트 : 모니터 객체는?)

Q3) 다음과 같이 외부에서 임의의 모니터 객체를 전달해주어도 동기화 문제가 해결될 수 있을지 생각해보기

```
class MapManipulateRunnable implements Runnable {
    private HashMap<String, String> hashMap;
    private String name;

    public MapManipulateRunnable(HashMap hashMap, String name) {
        this.hashMap = hashMap;
        this.name = name;
    }

    // 모니터 객체 저장 필드 및 세터 메서드 추가
    private Object monitor;
    public void setMonitor(Object monitor) {
        this.monitor = monitor;
    }

    @Override
    public void run() {
        for(int i=0;i<10_000;i++) {
```

```
            // HashMap 객체가 아닌 외부에서 전달받은 임의의 모니터 객체로 사용하여 동기화
            synchronized (monitor) {
                if (hashMap.containsKey("key")) {
                    String value = hashMap.remove("key");
                    if (value == null) {
                        System.out.println("[" + name + "] " + "this will not
happens." + i);
                    }
                } else {
                    hashMap.put("key", "hello");
                }
            }
        }
    }
}

public class SynchronizedDemo2 {
    public static void main(String[] args) {
        HashMap<String, String> shared = new HashMap<>();
        MapManipulateRunnable runnable1 = new MapManipulateRunnable(shared,
"1");
        MapManipulateRunnable runnable2 = new MapManipulateRunnable(shared,
"2");

        // 모니터 객체 생성하여 전달
        Object monitor = new Object();
        runnable1.setMonitor(monitor);
        runnable2.setMonitor(monitor);

        new Thread(runnable1).start();
        new Thread(runnable2).start();
    }
}
```

스레드 사용 팁

- 스레드 사용할 때 가급적 스레드간 **공유 자원이 없거나 적도록 설계** (태스크 분해)
- 공유 자원이 있을 경우 **한 스레드에서는 읽기 작업만, 한 스레드에서는 쓰기 작업만 수행하는 구조로 작성**한다면 좀 더 관리하기 편함