Thread Safety

멀티 스레드 환경에서의 스레드 안전에 대해 이해하기

OS-Challenge 1주 차 - 심수현

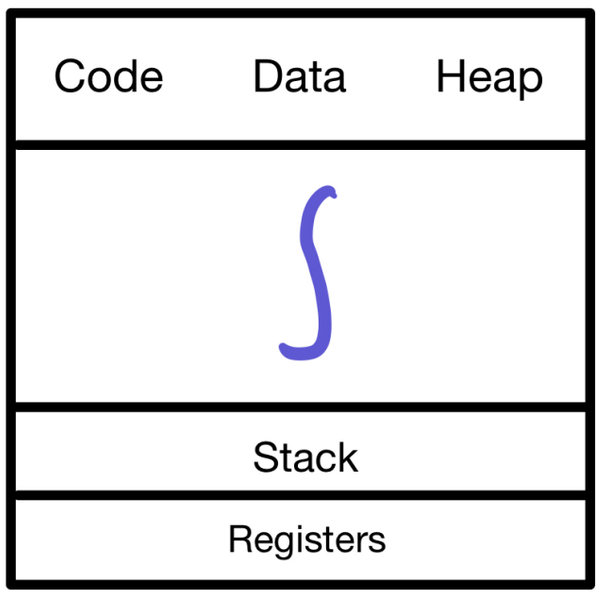
INDEX

Multi-Thread Programming

Single Thread / Multi Thread

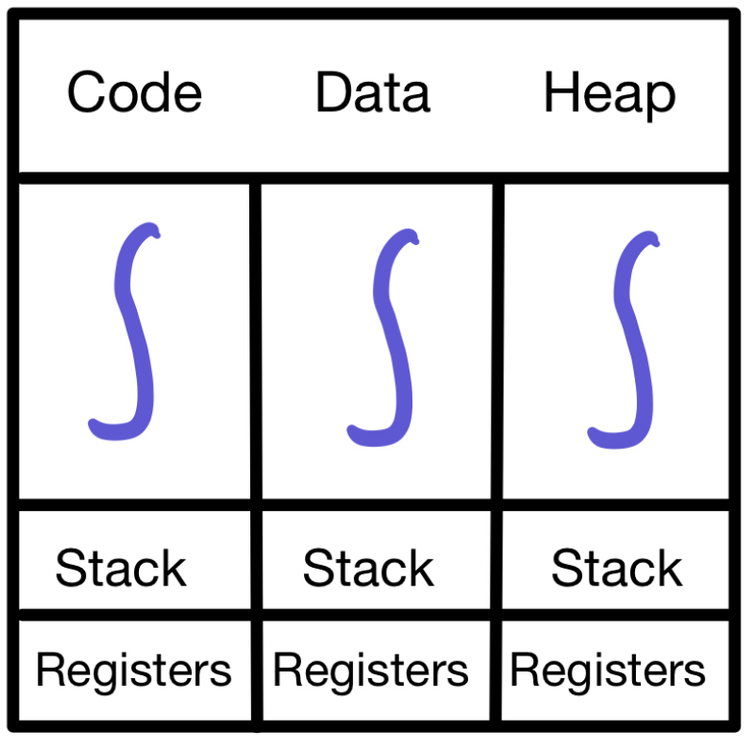
***thread*: 한 프로세스 내에서 구분지어진 실행 단위**

Single Thread



- Context Switch 작업을 요구하지 않음
- 자원 접근에 대한 동기화 신경쓰지 않음
- CPU 메모리 적게 사용

Multi Thread



- Thread 간 긴밀히 연결되어 있음
- 공유된 자원 관리 필요(통신 비용 절감, 메모리 효율성)
- 동기화 작업 필요
- Context Switching 비용이 적음

Thread-Safety란?

정의

3.407 Thread-Safe

- POSIX

A thread-safe function can be safely invoked concurrently with other calls to the same function, or with calls to any other thread-safe functions, by multiple threads. Each function defined in the System Interfaces volume of POSIX.1-2017 is thread-safe unless explicitly stated otherwise. Examples are any "pure" function, a function which holds a mutex locked while it is accessing static storage, or objects shared among threads.

In computer programming, thread safety is a property of an object or system that ensures that it remains consistent when accessed by multiple threads. A thread is a lightweight process that can run concurrently with other threads in a single program. *- BARD*

2.1 What is thread safety? - *Java Concurrency in Practicecy*

Defining thread safety is surprisingly tricky. The more formal attempts are so complicated as to offer little practical guidance or intuitive understanding, and the rest are informal descriptions that can seem downright circular. A quick Google search turns up numerous "definitions" like these:

> ...can be called from multiple program threads without unwanted interactions between the threads.
>
> ...may be called by more than one thread at a time without requiring any other action on the caller's part.

Thread-Safety란?

정의

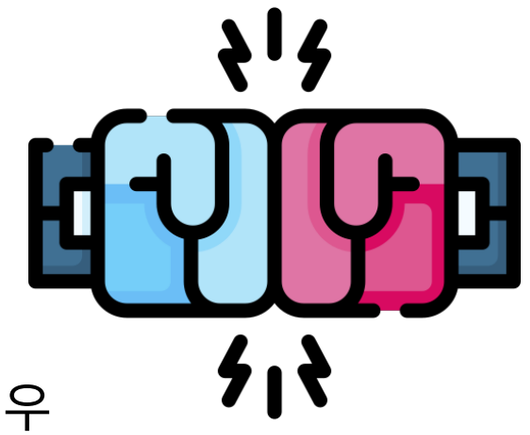
“다수의 스레드에 의한 동시 호출에서 안정성이 보장되는 상태”

두 개 이상의 스레드가 **Race Condition**에 들어가거나 같은 객체에 **동시에 접근**해도 연산결과의 **정합성**이 보장될 수 있게끔 **메모리 가시성**이 확보된 상태

Thread-Safety란?

무엇으로부터 ‘Safe’하려는 걸까요? - Race Condition (Data Race, API Race)

Race Condition; 경쟁상태



- 여러 스레드나 프로세스가 한정된 공유 자원에 **동시에(concurrently) 접근**하는 경우
- 경쟁상태는 데이터의 **불일치(inconsistency) 문제**를 야기할 수 있음
- 경쟁상태를 다루기 위해서는 **동기화(synchronize)** 되어야 함
- 현재 작업이 제어할 수 없는 또 다른 작업과의 진행 순서,
  즉, **타이밍에 따라 결과가 달라져** 여러 결과를 만들어낼 수 있는 바람직하지 않은 상황

[대표 유형]

1) Read-Modify-Write 패턴

- 이전 상태를 기준으로 객체의 현재 상태를 변경하면서 발생하는 문제

```
int count = 1000;
for (int i = 0; i < 1000; i++) {
    count++;
}
```

2) Check-Then-Act 패턴

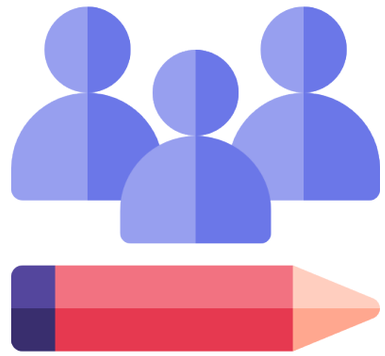
- 이전에 검증(Check)한 결과가 행동(Act) 시점에는 더 이상 유효하지 않을 때 발생하는 문제

```
// Check
if (count < 20) {
    // Act
    Thread.sleep( millis: 1);
    System.out.println("count: " + count);
}
```

Thread-Safety란?

무엇으로부터 ‘Safe’하려는 걸까요? - Race Condition (Data Race, API Race)

Data Race; 데이터 경쟁



- 다른 곳에서 **메모리 위치를 읽거나 수정할 가능성**이 있는 어떤 메모리 위치에 **쓰기 작업**을 하는 것
- 만약 프로그램의 실행이 두 개의 잠재적으로 동시에 발생하는 **충돌하는 액션**을 포함하고,
  그 중 적어도 하나가 원자적이지 않은 경우에는 해당 프로그램에 데이터 경쟁이 포함되어 있음
- **병렬 처리**를 하는 경우 흔하게 발생

1) 동기화를 위한 아무 수단도 사용하지 않은 경우 => *Race Condition, Data Race O*
2) 개별 변수를 읽을 때에만 동기화 (Mutex 등의 락 or Atomic 연산을 이용)를 적용한 경우
   => *Race Condition O, Data Race X*
3) 전체 연산에 동기화를 적용한 경우 => *Race Condition, Data Race X*

[참고]
API Race; API 경쟁(Data Race - built-in types / API Race - class types)
- 객체의 API 계약이 해당 작업을 동시에 허용하지 않을 때,

프로그램이 동일한 객체에 대해 두 개의 동시 작업을 수행하는 경우 API 경쟁이 발생

Thread-Safety와 공유 자원

자바에서 스레드가 실행될 때 어떤 자원을 공유하게 될까요?

Runtime Data Area

Local Variables 지역변수

- 각 스레드의 Stack에 저장 -> 지역변수는 스레드 간 절대 공유되지 않음

  => 모든 지역변수는 thread-safe

Local Object References 지역 레퍼런스 변수

- 레퍼런스 자체는 공유되지 않지만 참조된 객체는 스레드 각각의 Stack에 저장되지 않음
- 모든 객체들은 공유 Heap에 저장
- 만약 생성된 객체가 생성된 메소드 지역 안에서 벗어나지 않으면 thread-safe

Object Member Variables 객체 멤버 변수

- 객체 멤버 변수는 객체와 함께 Heap에 저장
- 만약 두 스레드가 같은 객체 인스턴스의 메소드를 호출하고
  그 메소드가 객체 멤버 변수를 업데이트 할 경우 메소드는 thread-safe 하지 않음
- 그러나 만약 두 스레드가 다른 인스턴스에 대해 동시적으로 호출한다면 경쟁 조건을 유발하지 않음

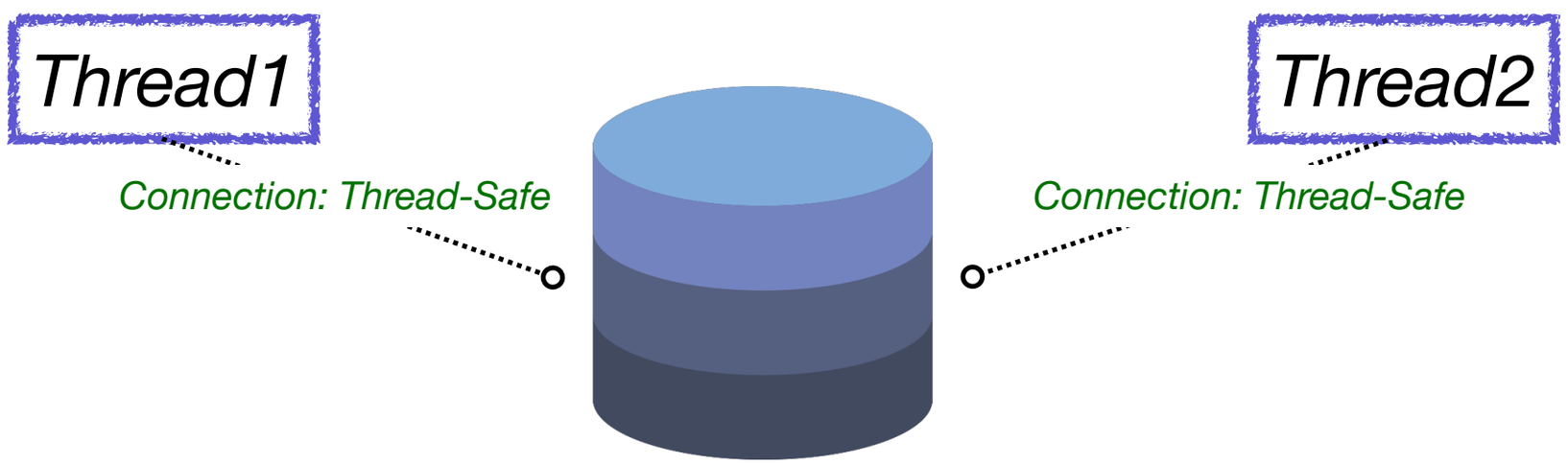
=> 즉, 객체가 thread-safe 하지 않더라도 경쟁 조건을 유발하지 않는 방법으로 사용될 수 있음

Thread Control Escape Rule

Thread-Safety 판단 방법

**“만약 한 자원의 생성, 사용, 소멸이 동일한 스레드 내에서 이루어지고,
해당 스레드에서 절대 벗어나지 않는다면
이 자원의 사용은 thread-safe라 할 수 있습니다.”**

- 자원은 객체, 배열, 파일, 데이터베이스, 커넥션, 소켓 기타 등등 어떤 방식으로든 **공유 자원**이 될 수 있고, 자바에서 객체들은 반드시 정확하게 소멸되지 않아도 됩니다. (소멸: 객체로의 참조를 잃거나 null이 되는 것)
- 객체 사용이 thread-safe해도 그 객체가 파일이나 데이터베이스 같은 공유 자원을 가리킨다면, 전체적으로 해당 어플리케이션은 thread-safe하지 않을 수 있습니다.
  - Ex. 두 개의 스레드가 각각 데이터베이스에 연결하는 커넥션을 생성하면 커넥션 자체는 thread-safe합니다.
  - ⚠️ 그러나 그 연결이 가리키는 데이터베이스의 사용은 thread-safe하지 않을 수 있습니다.



Thread-Safety 구현 방법

재진입성 / 상호 배제 / 스레드 지역 저장소 / 원자 연산 / 불변 객체

Reentrency 재진입성
Thread-Local Storage 스레드 지역 저장소
Atomic Operation 원자 연산
Mutual Exclusion Locks 락
Immutable Object 불변 객체
Producer - Consumer Pattern
Blocking Queue
Volatile
Instance Confinement 인스턴스 한정
Thread Confinement 스레드 한정
Condition Queues 조건 큐
Synchronized Collections 동기화 컬렉션
State Dependence 상태범위 제한
...

공유 변수 최소화 + 캡슐화 + 문서화

Thread-Safety 구현 방법

재진입성 / 상호 배제 / 스레드 지역 저장소 / 원자 연산 / 불변 객체

재진입성(Reentrancy)

- 어떤 함수가 한 스레드에 의해 호출되어 실행중일 때,
다른 스레드가 그 함수를 호출하더라도 그 결과가 각각 올바르게 주어져야 함

상호배제(Mutual Exclusion)

- 공유 자원을 사용할 경우 해당 자원에 대한 접근을 Semaphore와 같은 Lock으로 통제해야 함
- 임계구역을 정의하고 해당 임계구역은 한 번에 하나의 스레드만 접근해야 함

스레드 지역 저장소(Thread Local Storage)

- 공유 자원의 사용을 최대한 줄여 각각의 스레드에서만 접근 가능한 저장소를 사용함으로써 동시 접근을 막음
- 공유상태를 피할 수 없을 경우 사용

원자 연산(Atomicity)

- 공유 자원에 접근할 때 원자 연산, 원자적으로 정의된 접근 방법을 사용함

불변객체(Immutable Object)

- 객체 생성 이후에 값을 변경할 수 없도록 만들어야 함

Thread-Safety와 문서화

Effective Java Item 82: 스레드 안정성 수준을 문서화하라

Q. API 문서에 synchronized 한정자가 보이는 메서드는 스레드 안전하다?

```
public synchronized void method() {
    System.out.println("thread-safe?");
}
```

🤔 메서드 선언에 synchronized 한정자를 선언할지는 구현 이슈일 뿐 API에 속하지 않는다.
따라서 이것만으로는 그 메서드가 스레드 안전하다고 믿기 어렵다.

멀티스레드 환경에서도 API를 안전하게 사용하게 하려면
클래스가 지원하는 스레드 안전성 수준을 정확히 명시해야 한다.

synchronized 한정자는 문서화와 관련이 없다!

Thread-Safety와 문서화

Effective Java Item 82: 스레드 안정성 수준을 문서화하라

[thread-safe 수준(높은 순으로 나열)]

- **불변(immutable)** Ex. String, Long, BigInteger
- **무조건 스레드 안전(unconditionally thread-safe)** Ex. AtmoicLong, ConcurrentHashMap
- **조건부 스레드 안전(conditionally thread-safe)**
  Ex. Collections.synchronized 래퍼 메서드가 반환한 컬렉션들
- **스레드 안전하지 않음(not thread-safe)**
  Ex. ArrayList, HashMap
- **스레드 적대적(thread-hostile)**
  Ex. 동시성을 고려하지 않고 만들면
  우연히 만들어질 수 있음

> Returns a synchronized (thread-safe) map backed by the specified map. In order to guarantee serial access, it is critical that **all** access to the backing map is accomplished through the returned map.
>
> It is imperative that the user manually synchronize on the returned map when iterating over any of its collection views:

```
Map m = Collections.synchronizedMap(new HashMap());
    ...
Set s = m.keySet();  // Needn't be in synchronized block
    ...
synchronized (m) {  // Synchronizing on m, not s!
    Iterator i = s.iterator(); // Must be in synchronized block
    while (i.hasNext())
        foo(i.next());
}
```

> Failure to follow this advice may result in non-deterministic behavior.
>
> The returned map will be serializable if the specified map is serializable.
>
> Params: `m` – the map to be "wrapped" in a synchronized map.
>
> Returns: a synchronized view of the specified map.

```
@NotNull @Contract(pure = true)
public static <K,V> Map<K,V> synchronizedMap( @NotNull Map<K,V> m) {
    return new SynchronizedMap<>(m);
}
```

Quiz

- Q1. Thread-Safe와 Race Condition의 관계에 대해 설명해주세요.
- Q2. Thread-Safety하게 구현하는 방법 한 가지를 설명해주세요.
- Q3. synchronized 한정자가 보이는 메서드는 항상 thread-safe 할까요?

References

Books

- Brian Goetz, Tim Peierls, Bloch Joshua, Bowbeer Joseph, Holmes David, Lea Doug (2006). Java Concurrency in Practice. Addison-Wesley Professional.
- Joshua Bloch(2018). Effective Java(3rd ed.). Addison-Wesley Professional.

Youtube Videos

- CppCon 2018: Geoffrey Romer “What do you mean "thread-safe"?”
- Thread Safety in Singleton
- Race Conditions in Java Multithreading
- [10분 테코톡] 🌷 코다의 Process vs Thread
- [10분 테코톡] 알렉스, 열음의 멀티스레드와 동기화 In Java

Articles

- Race condition vs. Data Race: the differences explained
- Thread Safety and Shared Resources
- Race Conditions and Critical Sections