Clean Code

13. 동시성

동시성의 필요성

- **동시성은 결합을 없애는 전략**
  - 무엇(what) 과 언제(when)을 분리하는 전략
  - Thread가 하나인 프로그램은 what과 when이 서로 밀접하다.
- **응답 시간과 작업 처리량 개선을 위해 동시성 구현을 하기도 한다.**

동시성에 대한 미신과 오해

미신과 오해

- **동시성은 항상 성능을 높여준다**
  - 대기 시간이 아주 길어 여러 thread가 processor 를 공유할 수 있거나, 여러 processor가 동시에 처리할 독립적인 계산이 충분히 많은 경우에만 성능이 높아진다.
- **동시성을 구현해도 설계는 변하지 않는다.**
  - Single thread system 과 Multi thread system은 설계가 판이하게 다르다.
  - 일반적으로 what과 when을 분리하면 시스템 구조가 크게 달라진다.
- **웹 프레임워크를 사용하면 동시성을 이해할 필요가 없다.**
  - 실제로는 프레임워크의 컨테이너가 어떻게 동작하는지, 어떻게 동시 수정, 데드락 등과 같은 문제를 피할 수 있는지 알아야 한다.

올바른 생각

- **동시성은 다소 부하를 유발한다.**
- **동시성은 복잡하다.**
- **일반적으로 동시성 버그는 재현하기 어렵다.**
  - 진짜 결함으로 간주되지 않고 일회성 문제로 여겨 무시하기 쉽다.
- **동시성을 구현하려면 흔히 근본적인 설계 전략을 재고해야 한다.**

동시성 방어 원칙

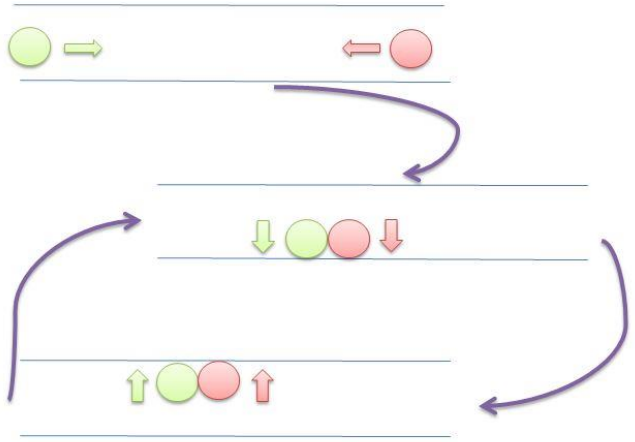
- **SRP : 동시성 관련 코드는 다른 코드와 분리해야 한다.**
  - 동시성 코드는 독자적인 개발, 변경, 조율 주기가 있다.
  - 동시성 코드에는 독자적인 난관이 있다. 다른 코드에서 겪는 난관과는 다르며 훨씬 어렵다.
  - 잘못 구현한 동시성 코드는 별의별 방식으로 실패한다. 주변에 있는 다른 코드가 발목을 잡지 않더라도 동시성 하나만으로도 충분히 어렵다.
- **자료 범위를 제한하라. (공유 자원을 캡슐화하고, 공유 자료를 최대한 줄여라)**
  - thread 간 반드시 공유해야 하는 자료의 범위를 제한하라.
    - Critical section 을 synchronized 키워드로 보호하라(JAVA)
  - 공유 자료를 수정하는 위치가 많을수록 다음 가능성도 커진다.
    - 보호할 임계영역을 빼먹는다. 그래서 공유 자료를 수정하는 모든 코드를 망가뜨린다.
    - 모든 임계영역을 올바로 보호했는지 확인하느라 똑같은 노력과 수고를 반복한다.(DRY 원칙 위반)
    - 그렇지 않아도 찾아내기 어려운 버그가 더욱 찾기 어려워진다.
- **자료 사본을 사용하라**
  - 특정 thread에서 읽기 동작만 한다면, 객체를 복사해 읽기 전용으로 사용하라
- **thread는 독립적으로 구현하라.**
  - 가급적 다른 thread와 자료를 공유하지 않는다.
  - 독자적인 thread로 가능하면 다른 processor 에서 돌려도 괜찮도록 자료를 독립적 단위로 분할하라.

라이브러리를 이해하라 (JAVA)

- **Thread 환경에 안전한 컬렉션을 사용**
  - java.util.concurrent 패키지가 제공하는 클래스는 다중 thread 환경에서 사용해도 안전하며 성능도 좋다.
  - java.util.concurrent, java.util.concurrent.atomic, java.util.concurrent.locks 를 익혀라.
- **서로 무관한 작업을 수행할 때는 executor 프레임워크를 사용**
- **가능하다면 thread가 차단 되지 않는 방법을 사용한다.**
- **일부 클래스 라이브러리는 thread 에 안전하지 못하다.**

실행 모델을 이해하라 (1/3)

- **기본적인 용어**
  - Bound Resource : multi-thread 환경에서 사용하는 자원으로, 크기나 숫자가 제한적이다. DB 연결, 길이가 일정한 read/write 버퍼 등이 있다.
  - Mutual Exclusion : 한 번에 한 thread만 공유 자료나 공유 자원을 사용할 수 있는 경우를 가리킨다.
  - Starvation : 하나 이상의 thread가 매우 긴 시간동안 혹은 영원히 자원을 기다리는 상태
  - Deadlock : 여러 thread가 서로가 끝나기를 기다린다. 모든 thread가 각기 필요한 자원을 다른 thread가 점유하는 바람에 어느 쪽도 진행하지 못한다.
  - Livelock : lock을 거는 단계에서 각 thread가 서로 방해한다. thread는 계속 진행하려 하지만, resonance로 인해 매우 긴 시간동안 혹은 영원히 진행하지 못한다.
    - **live lock 과 dead lock의 차이는 thread 들이 block 되지 않고, 계속해서 서로에 대해 응답하려고 하는 것**
    - https://stackoverflow.com/questions/6155951/whats-the-difference-between-deadlock-and-livelock



실행 모델을 이해하라 (2/3)

Multi-thread programming 실행 모델

- **Producer-Consumer**
  - 하나 이상의 producer thread가 정보를 생성해 queue에 넣는다. 하나 이상의 consumer thread가 queue에서 정보를 가져와 사용한다.
  - Producer와 Consumer 가 사용하는 queue는 한정된 자원 이다.
  - Producer는 queue에 빈공간이 있어야 정보를 채운다. (빈 공간이 생길 때까지 기다린다)
  - Consumer는 queue에 정보가 있어야 가져온다. (정보가 채워질 때까지 기다린다)
  - 대기열을 올바로 사용하고자 Producer와 Consumer는 서로에게 시그널을 보낸다.
    - Producer는 대기열에 정보가 있다, Consumer는 빈 공간이 있다는 시그널
  - Thread의 동작 순서에 따라 서로에게서 시그널을 기다릴 가능성이 존재한다
- **Readers-Writers**
  - read thread를 위한 주된 정보원으로 공유 자원을 사용하는데, write thread가 공유 자원을 이따금 갱신한다고 하자
  - 이 때, throughput을 강조하면 write가 starvation이 되거나 오래된 정보가 쌓인다.
  - 갱신을 허용하면 throughput이 낮아진다.
  - read와 write 의 밸런스를 맞춰줄 해법이 필요하다.

실행 모델을 이해하라 (3/3)

Multi-thread programming 실행 모델

- **Dining Philosophers**
  - 둥근 식탁에 철학자들이 둘러 앉아 있고, 각 철학자 왼쪽에는 포크가 놓여져 있고, 식탁의 가운데에는 커다란 스파게티 한 접시가 놓여져 있다.
  - 철학자들은 배가 고프지 않으면 생각하며 시간을 보낸다.
  - 배가 고프면 양손에 포크를 집어 들고 스파게티를 먹는다. (양손에 포크를 쥐지 않으면 먹지 못한다.)
    - 왼쪽 철학자나 오른쪽 철학자가 포크를 사용하는 중이라면 그쪽 철학자가 먹고 나서 포크를 내려놓을 때까지 기다려야 한다.
  - 스파게티를 먹고 나면 포크를 내려놓고 배가 고플 때까지 다시 생각에 잠긴다.
  - 여기서 철학자를 thread로, 포크를 자원으로 바꿔 생각해보면 여러 thread가 자원을 얻으려 경쟁하게 되는데 주의해서 설계하지 않으면 deadlock, livelock, throughput 저하 등을 겪게된다.

동기화하는 메서드 사이에 존재하는 의존성을 이해하라

- **동기화하는 메서드 사이에 의존성이 존재하면 동시성 코드에 찾아내기 어려운 버그가 생긴다.**
  - 공유 객체 하나에는 메서드 하나만 사용하라.
  - 공유 클래스 하나에 동기화된 메서드가 여럿이라면 구현이 올바른지 다시 확인하라
- **공유 객체 하나에 여러 메서드가 필요한 경우 아래의 3가지 방법을 고려하라**
  - 클라이언트에서 잠금 : 클라이언트에서 첫 번째 메서드를 호출하기 전에 서버를 잠근다. 마지막 메서드를 호출할 때까지 잠금을 유지한다.
  - 서버에서 잠금 : 서버에다 “서버를 잠그고 모든 메서드를 호출한 후 잠금을 해제하는“ 메서드를 구현한다. 클라이언트는 이 메서드를 호출한다.
  - Adapted 서버 : 잠금을 수행하는 중간 단계를 생성한다. ‘서버에서 잠금‘ 방식과 유사하지만 원래 서버는 변경하지 않는다.

클라이언트에서 잠금 Example

```
while (result == Extractor.RESULT_CONTINUE && !loadCanceled) {
    try {
        loadCondition.block();
    } catch (InterruptedException e) {
        throw new InterruptedIOException();
    }

    (...)

    result = progressiveMediaExtractor.read(positionHolder);

    long currentInputPosition = progressiveMediaExtractor.getCurrentInputPosition();
    if (currentInputPosition > position + continueLoadingCheckIntervalBytes ){
        position = currentInputPosition;
        loadCondition.close();
        handler.post(onContinueLoadingRequestedRunnable);
    }
}
```

동기화하는 부분을 작게 만들어라

- 자바에서 synchronized 키워드를 사용하면 lock이 설정된다. 같은 lock으로 감싼 모든 코드 영역은 한 번에 한 thread만 실행이 가능하다. lock은 thread를 지연시키고 부하를 가중시킨다. synchronized 문을 남발하는 코드는 바람직하지 않다.
- 반면 critical section은 반드시 보호해야 한다.
- 따라서 코드를 짤 때는 critical section의 수를 최대한 줄여야한다.

올바른 종료 코드는 구현하기 어렵다.

- **깔끔하게 종료하는 코드는 올바로 구현하기 어렵다.**
  - 가장 흔히 발생하는 문제가 deadlock이다.
    - 예1) 부모 thread가 자식 thread를 여러 개 만든 후 모두가 끝나기를 기다렸다가 자원을 해제하고 종료하는 코드가 있다라고 한다면, 자식 thread 중 하나가 deadlock에 걸렸다면? 부모 thread는 영원히 기다리고 시스템은 영원히 종료하지 못한다.
    - 예2) 예1과 유사한 시스템이 사용자에게서 종료하라는 지시를 받았을 때, 부모 thread가 모든 자식 thread에게 작업을 멈추고 종료하라는 시그널을 전달하는데, 자식 thread 중 두 개가 producer/consumer 관계라면? producer는 종료했는데, consumer가 producer로 부터 오는 메시지를 기다린다면? consumer 는 block 된 상태에 있으므로 종료하라는 시그널을 받지 못한다. consumer는 producer를 영원히 기다리고 부모 thread는 자식 thread를 영원히 기다린다.
  - 종료 코드를 개발초기부터 고민하고 동작하도록 초기부터 구현하라.

Thread 코드 테스트하기

- **테스트 코드를 작성할 때 같은 자원을 사용하는 thread가 둘 이상으로 늘어나면 복잡한 문제를 만들 수 있다.**
  - 문제를 노출하는 테스트 케이스를 작성하라. 프로그램 설정과 시스템 설정과 부하를 바꿔가며 자주 테스트하라. 테스트가 실패하면 원인을 추적하라. 다시 돌렸더니 통과하더라는 이유로 그냥 넘어가면 안된다.
- **Thread 코드 테스트에 대한 몇가지 구체적 지침**
  - 말이 안되는 실패는 잠정적인 thread 문제로 취급하라.
    - 다중 thread 코드는 때때로 '말이 안되는 오류' 를 일으킨다. 해당 오류들은 실패를 재현하기 어렵기 때문에 단순한 '일회성' 문제로 치부하고 무시하기 쉽다. 일회성 문제를 계속해서 무시한다면 잘못된 코드위에 코드가 계속 쌓인다. 일회성 문제는 존재하지 않는다고 가정하라.
  - 다중 thread를 고려하지 않은 순차 코드부터 제대로 돌게 만들자
    - Thread 환경 밖에서 코드가 제대로 도는지 반드시 확인하라. 일반적인 방법으로 thread가 호출하는 POJO(Plane Old Java Object)를 만든다. POJO는 thread를 모른다. 따라서 thread 환경 밖에서 테스트가 가능하다.
    - Thread 환경 밖에서 생기는 버그와 thread 환경에서 생기는 버그를 동시에 디버깅하지 마라. 먼저 thread 환경 밖에서 코드를 올바로 만들어라.
    - Multi thread 를 사용하는 코드 부분을 다양한 환경에 쉽게 끼워 넣을 수 있게 thread 코드를 구현하라.
      - 하나의 thread로 실행하거나 여러 thread로 실행하거나, 실행 중 thread 수를 바꿔본다.
      - Thread 코드를 실제 환경/ 테스트 환경에서 돌려본다.
      - 테스트 코드를 빨리, 천천히, 다양한 속도로 돌려본다
      - 반복 테스트가 가능하도록 테스트 케이스를 작성한다.
    - 다양한 설정에서 실행할 목적으로 다른 환경에 쉽게 끼워 넣을 수 있게 코드를 구현하라.

Thread 코드 테스트하기

- **Thread 코드 테스트에 대한 몇가지 구체적 지침**
  - Multi thread를 쓰는 코드 부분을 상황에 맞게 조율할 수 있게 작성하라
    - 적절한 thread 개수를 파악하려면 상당한 시행착오가 필요하다.
    - 처음부터 다양한 설정으로 프로그램의 성능 측정 방법을 강구한다.
    - thread 개수를 조율하기 쉽게 코드를 구현한다.
      - 프로그램이 돌아가는 도중에 thread 개수를 변경하는 방법을 고려한다.
      - 처리율과 효율에 따라 스스로 thread 개수를 조율하는 코드도 고민한다.
  - Processor 수 보다 많은 thread를 돌려보라
    - 시스템이 thread를 swapping 할 때도 문제가 발생할 여지가 있다. swapping을 일으키려면 프로세서 수 보다 많은 thread를 돌린다.
    - swapping이 잦을수록 critical section을 빼먹은 코드나 dead lock을 일으키는 코드를 찾기 쉬워진다.
  - 다른 플랫폼에서 실행해보라
    - OS 마다 thread를 처리하는 정책이 다르다. 코드가 돌아갈 가능성이 있는 플랫폼 전부에서 테스트를 수행해야 한다.
    - 처음부터, 자주 모든 목표 플랫폼에서 코드를 실행하라
  - 코드에 보조 코드를 넣어 돌려라. 강제로 실패를 일으키게 해보라
    - thread 코드의 오류는 찾기 쉽지 않다, 간단한 테스트로는 버그가 쉽게 드러나지 않는다
    - thread 버그가 산발적이고, 우발적이고, 재현이 어려운 이유는 코드가 실행되는 수천 가지 경로 중 아주 소수만 실패하기 때문이다.
    - 오류를 자주 일으키기 위해 보조 코드 (ex. java의 Object.wait(), Object.sleep(), Object.yield(), Object.priority())등과 같은 메서드를 추가(보조 코드)해 코드를 다양한 순서로 실행한다.
    - 위의 메서드들은 thread가 실행되는 순서에 영향을 미치기 때문에 버그가 드러날 가능성이 높아진다.
    - 잘못된 코드라면 가능한 초반에 그리고 자주 실패하는 편이 좋다.

Thread 코드 테스트하기

❖ 보조 코드를 추가하는 방법 2가지

1. 직접 구현하기
   - 코드에다 직접 thread의 실행 순서를 바꾸는 함수들을 추가한다.
   - 직접 구현하는 방식의 문제점
     - 보조 코드를 삽입할 적정 위치를 직접 찾아야 한다.
     - 어떤 함수를 어디서 호출해야 적당한지 알기 어렵다.
     - 배포 환경에 보조 코드를 그대로 남겨두면 프로그램 성능이 떨어진다.
     - 무작위적이다. 오류가 드러날지, 드러나지 않을지 예측이 어렵다. 사실상 드러나지 않을 확률이 더 높다.
   - 테스트 환경에서 보조 코드를 실행하고, 실행할 때 마다 설정을 바꿔줄 방법도 필요하다. (그래야 오류가 드러날 확률이 높아진다)
     - thread를 전혀 모르는 POJO와 thread를 제어하는 클래스로 프로그램을 분할하면 보조 코드를 추가할 위치를 찾기 쉬워진다.
     - 여러 상황에서 sleep, yield 등으로 POJO를 호출하게 다양한 테스트 지그를 구현할 수도 있다.
2. 자동화
   - 보조 코드를 자동으로 추가하려면 AOF, CGLIB, ASM 등 과 같은 도구를 사용한다.
     - AOF : Aspect Oriented Framework, AOP를 위한 프레임워크를 말하는 듯함(?)
     - CGLIB : Code Generate Library, 클래스의 바이트코드를 조작하여 Proxy 객체를 생성해주는 라이브러리
       - https://velog.io/@dev_leewoooo/Proxy-pattern%EC%9D%B4%EB%9E%80-with-Java
     - ASM : java의 바이트 코드를 조작하는 프레임워크

Thread 코드 테스트하기

❖ 보조 코드를 추가하는 방법 2가지

2. 자동화

- example
  - ThreadJigglePoint.jiggle은 무작위로 sleep이나 yield를 호출 하거나 아무런 동작도 하지 않는다.
  - ThreadJigglePoint 클래스를 2가지로 구현하면 편리하다.
    - 하나는 jiggle() 메서드를 구현하지 않고 배포 환경에서 사용한다.
    - 다른 하나는 무작위로 sleep,yield, nop을 구현하여 테스트환경에서 수행한다.
- 코드를 jiggle(흔드는) 이유는 thread를 매번 다른 순서로 실행하기 위해서다. 좋은 테스트 케이스와 jiggling 기법은 오류가 드러날 확률을 크게 높여준다.

```
public class ThreadJigglePoint {
    public static void jiggle() {}
}

public synchronized String nextUrlOrNull() {
    if(hasNext()) {
        ThreadJigglePoint.jiggle();
        String url = urlGenerator.next();
        ThreadJigglePoint.jiggle();
        updateHasNext();
        ThreadJigglePoint.jiggle();
        return url;
    }
    return null;
}
```