13 동시성

* 동시성과 깔끔한 코드는 양립하기 어려움, 스레드 하나만 실행하는 코드는 짜기가 쉽다.,
* 겉으로 보기에 멀쩡하나 깊숙한 곳에 문제가 있는 다중 스레드 코드도 짜기 쉽다.
* 동시성이 필요한 이유?
  + 결합을 없애는 전략,, 무엇과 언제를 분리하는 전략이다.
  + 스레드가 하나인 프로그램은 무엇과 언제가 밀접
  + 무엇과 언제를 분리하면 애플리케이션 구조와 효율이 극적으로 나아진다.
  + 구조적 개선만을 위해 채택하는건 아니고 응답시간과 작업 처리량 개선이라는 요구사항으로 인해 직접적으로 동시성 구현이 불가피
  + 미신과 오해
    - 동시성과 관련된 오해들
      * 항상 성능을 높여줌
        + 때로 성능을 높여줌 대기시간이 길어 여러 스레드가 프로세스를 공유할 수 있ㅅ거나 여러 프로세서가 동시에 처리할 독립적인 계산이 충분히 많은 경우에만 성능이 옾아짐
      * 동시성을 구현해도 설계는 변하지 않음
        + 단일스레드 시스템과 다중 스레드 시스템은 설계가 판이하게 다르다. 일반적으로 무엇과 언제를 분리하면 시스템 구조가 크게 달라진다
      * 웹 또는 ejb컨테이너를 사용하면 동시성을 이해할 필요가 없다.
        + 실제로 컨테이너가 어떻게 동작하는지, 어떻게 동시에 수정하는지, 데드락 같은 문제를 피할 수 있는 지를 알아야만 한다.(좆같이 어렵다는 소리)
    - 동시성과 관련된 타당한 생각
      * 다소부하유발, 성능측면에서 부하, 코드도 더 짜야 함
      * 복잡, 간단한문제라도 복잡
      * 동시성버그는 재현하기 힘듬, 진짜 결함으로 간주되지 않고 일회성 문제로 여겨 무시하기 쉬움
      * 동시성 구현하려면 흔히 근본적인 설계 전략 재고
* 난관
  + 동시성을 구현하기 어려운 이유
    - 두개의 스레드에 같은 변수를 동시에 참조하면 일부 경로가 잘못된 결과를 내놓기 떄문에 정확히 고치려면 컴파일러가 바이트 코드를 처리하는 빙식과 자바 메모리 모델이 원자로 간주하는 최소 단위를 알아야 한다.
* 동시성 방어 원칙
  + 단일 책임 원칙
    - 동시성과 관련된 코드는 다른 코드와 분리해야한다. 불행히도 동시성과 관련이 없는 코드에 동시성을 곧바로 구현하는 사례가 너무나도 흔하다. 동시성을 구현할 떄 다음 몇가지를 고려한다.
      * 동시성 코드는 독자적인 개발 변경 조율 주기가 있다
      * 동시성 코드에는 독자적인 난관이 있다. 다른 코드에서 겪는 난관과 다르며, 훨씬 어렵다.
      * 잘못 구현한 동시선 코드는 별의 별 방식으로 실패, 주변에 있는 다른 코드가 발목을 잡지 않더라도 동시성 하나만으로 충분히 어렵다.
      * 결론 : 동시성 코드는 다른 코드와 분리하라
    - 따름 정리 : 자료 범위를 제한하라
      * 공유 객체를 사용하는 코드 냄 임계 영역을 동기화 키워드로 보로하라고 권장한다.
    - 임계 영역의 수를 줄이는 기술이 중요하다. 공유 자료를 수정하는 위치가 많을수록 다음 가능,성도 커진다
      * 보호할 임계 영역을 뺴먹는다. 그래서 공유 자료를 수정하는 모든 코드를 망가뜨린다.
      * 모든 임계영역을 올바로 보호했는지 확인하느라 똑 같은 노력과 수고를 반복한다.
      * 그렇지 않아도 찾아내기 어려운 버그가 더욱 찾기 어려워진다.
      * 결론 : 자료를 캡슐화 하라, 공유 자료를 최대한 줄여라
    - 자료 사본을 사용하라
      * 객체를 공유하지 않고 복사하여 결과를 내는 자료 사본 방식으로 결과를 가져오기
      * 공유 객체를 피하는 방법이 있다면 코드가 문제를 일으킬 가능성도 아주 낮아진다. 객체를 복사하는 시간과 부하가 걱정스럽다면 복사비용이 진짜 문제인지 실측해볼 필요가 있다. 사본으로 동기화를 피할 수 있다면 내부 잠금을 없애 절약한 수행 시간이 사본 생성과 가비지 컬랙션에 드는 부하를 상쇄할 가능성이 크다.
    - 스레드는 가능한 독립적으로 구현하라
      * 자신만의 세상에 존재하는 스레드를 구현한다. 다른 스레드와 자료를 공유하지 않는다. 각 스레드는 클라이언트 요청 하나를 처리한다. 모든 정보는 비공유 출처에서 가져오며, 로컬 변수에 저장한다.
      * 결론 : 독자적인 스레드로 가능하다면 프로세스에서 들려도 괜찮도록 자료를 독립적인 단위로 분할하라.
* 라이브러리를 이해하라
  + 자바 5는 동시성 측면에서 이전 버전보다 많이 나아졌다. 자바 5로 스레드 코드를 구현한다면
    - 스레드 환경에 안전한 컬랙션을 사용한다.
    - 서로 무관한 작업을 수행할 때는 executor 프레임 워크를 사용한다
    - 가능하다면 스레드가 차단되지 않는 방법을 사용한다
    - 일부 클래스 라이브러리는 스레드에 안전하지 못하다.
  + 스레드 환경에 안전한 컬렉션
    - 스레드에 사용해도 안전한 컬렉션 클래스 몇 개를 구현했는데 나중에 java.util.concurrent 패키지에 추가되었다. 이 클래스들은 다중 스레드 환경에서 사용해도 안전하며 성능도 좋다.
    - 실제로 concurrentHashMap은 거의 모든 상황에서 hashmap보다 빠르다. 동시 읽기 쓰기를 지원하며, 자주 사용하는 복합 연산을 다중 스레드 상에서 안전하게 만든 메서드로 제공한다. 자바 5를 사용한다면 concurrentHashMap부터 살펴보라고 권한다.
    - 좀 더 복잡한 동시성 설계를 지원하고자 자바5에서는 다른 클래스도 추가되었다.
      * reentrantLock : 한 메서드에서 잠그고 다른메서드에서 푸는 lock
      * semaphore : 전형적인 세마포어(공유 메모리 상에서 여러개 페이징 관리를 하는 프로세스), 개수가 있는 락
      * countdownLatch : 지정한 수만 큼 이벤트가 발생하고 나서야 대기중인 스레드를 모두 해체하는 락 모든 스레드에서 동시에 공평하기 시작할 기회를 준다.
* 실행 모델을 이해하라
  + 다중 스레드 애플리케이션을 분류하는 방식은 여러가지이다. 몇가지 기본용어부터 이해하면
    - 한정된 자원 : 다중 스레드 환경에서 사용하는 자원으로 크거나 숫자가 제한적. 데이터베이스 연결 길이가 일정한 읽기/쓰기 버퍼 등이 예
    - 상호 배제 : 한번에 한 스레드만 공유 자료나 공유 자원을 사용할 수 있는 경우를 가리킨다
    - 기아 : 한 스레드나 여러 스레드가 굉장히 오랫동안 혹은 영원히 자원을 기다린다. 예를 들어 항상 짧은 스레드에게 우선 순위를 준다면 짧은 스레드가 지속적으로 이어질 경우 긴 스레드가 기아상태에 빠짐
    - 데드락 : 여러 스레드가 서로가 끝나기를 기다림, 모든 스레드가 각기 필요한 사원을 다른 스레드가 점유하는 바람에 어느 쪽도 더 이상 진행하지 못한다.
    - 라이브락 : 락을 거는 단계에서 각 스레드가 서로를 방해. 스레드는 계속해서 진행하려 하지만, 공명으로 인해 굉장히 오랫동안 혹은 영원이 진행하지 못한다.
  + 생산자와 소비자
    - 하나 이상의 생산자 스레드가 정보를 생성해 버퍼나 대기열을 넣는다. 하나 이상 소비자 스레드가 대기열에서 정보를 가져와 사용한다. 생산자 스레드와 소비자 스레드가 사용하는데 대기열은 한정된 자원. 생산자 스레드는 대기열에 빈 공간이 있어야 정보를 채운다. 즉 빈 공간이 생길 떄 깢, 기다림
    - 생산자 스레드는 대기열에 정보를 채운 후 소비자 스레드에게 대기열에 정보가 있다는 시그널을 보냄 소비자 스레드는 대기열에서 정보를 읽어드린 후 대기열에 빈 공간이 있다 는 시그널을 보냄. 따라서 잘못하면 생산자 스레드와 소비자 스레드가 둘 다 진행 가능함에도 불구하고 동시에 서로에게 시그널을 기다릴 가능성이 존재한다.
  + 읽기-쓰기
    - 일기 스레드를 위한 주된 정보원으로 공유 자원을 사용하나 쓰기 스레드가 이 공유 자원을 이따금 갱신한다고 한다면 이런 경우 처리율이 문제의 핵심이다
    - 처리율을 강조하면 기아 현상이 생기거나 오래된 정보가 쌓인다. 갱신을 하용하면 처리율에 영향을 미친가. 쓰기 스레드가 버퍼를 갱신하는 동안 읽기 스레드가 버퍼를 읽지 않으려면 마찬가지로 읽기스레드가 버퍼를 읽는 동안 쓰기 스레드를 갱신하지 않으려면, 복잡한 균형 잡기가 필요하다. 대개는 쓰기 스레드가 버퍼를 오랫동안 점유하는 바람에 여러 읽기 스레드가 버퍼를 기다리느라 처리율이 떨어진다.
    - 따라서 읽기 스레드의 요구와 쓰기 스레드의 요구를 적절히 만족시켜 처리율도 적당히 높이고 기아도 방지하는 해법이 필요 간단한 전략은 읽기 스레드가 없을 때까지 갱신을 원하는 쓰기 스레드가 버퍼를 기다리는 방법
  + 식사하는 철학자들
    - 기업 어플리케이션에서 겪는 문제인데 기업 어플리케이션은 여러 프로세스가 자원을 얻으려 경쟁한다. 주의해서 설계하지 않으면 데드락 라이브락 처리율 저하 효율성 저하 등을 겪는다.
    - 일상에서 접하는 대다수 다중 스레드 제는 위 세 범주 중 하나에 속한다. 각 알고리즘을 공부하고 해법을 직접 구현해보라 그러면 나중에 실전 문제에 부닥쳤을 때 해결이 쉬워진다.
* 동기화하는 메서드 사이에 존재하는 의존성을 이해
  + 동기화하는 메서드 사이에 의존성이 존재하면ㅁ 동시성 코드에 찾아내기 어려운 버그가 생김, 자바 언어는 개별 메서드를 보호하는 synchronized라는 개념을 지원(동기화) 하지만 공유 클래스 하나에 동기화된 메서드가 여럿이라면 구현이 올바른지 다시 한번 확인하기 바란다.
  + 공유 객체 하나에 여러 메서드가 필요한 상황도 생긴다. 그럴 때는 세가지 방법을 고려한다.
    - 클라이언트에서 잠금- 클라이언트에서 첫번쨰 메서드를 호출하기 전에 서버를 잠금 마지막 메서드를 호출할 때까지 잠금을 유지
    - 서버에서 잠금-서버에다 서버를 참그고 모든 메서드를 호출 한 후 잠금을 해제하는 메서드를 구현 클라이언트는 이 메서드를 호출한다
    - 연결서버 – 잠금을 수행하는 중간 단계를 생성. 서버에서 잠금 방식과 유사하지만 원래 서버는 변경하지 않는다
* 동기화하는 부분을 작게 만들어라
  + 자바에서 synchronized 키워드를 사용하면 락을 설정한다. 같은 락으로 감싼 모든 코드 영역은 한번에 한 스레드만 실행이 가능하다. 락은 스레드를 지연시키고 부하를 가중시킨다. 그러므로 여기저기서 synhronized문을 남발하는 코드는 바람직하지 않음. 반면 임계영역은 반드시 보호해야 한다. 따라서 코드를 짤 떄는 임계 영역 수를 최대한 줄여야 한다.
  + 임계 영역 개수를 줄인답시고 거대한 임계 영역 하나로 구현하는 순진한 프로그래머들도 있다. 필요 이상으로 임계 영역 크기를 키우면 스레드간 경쟁이 늘어나고 프로그램 성능이 떨어짐
    - 동기화 하는 부분을 최대한 작게 만들어라
* 올바른 종료 코드는 구현하기 어렵다.
  + 영구적으로 돌아가는 시스템을 구현하는 방법과 잠시 돌다 깔끔하게 종료하는 시스템을 구현하는 방법은 다르다.
  + 깔끔하게 종료하는 코드는 올바로 구현하기 어렵다. 가장 흔히 발생하는 문제가 데드락 즉 스레드가 절대 오지 않을 시그널을 기다림
  + 예를 들어 부모 스레드와 자식 스레드를 여러 개를 만든 후 모두가 끝나기를 기다렸다 자원을 해제하고 종료하는 시스템이 있다 가정, 만약 자식 스레드 중 하나가 데드락에 걸렸다면, 부모 스레드는 영원히 기다리고, 시스템은 영원히 종료하지 못한다.
    - 종료코드를 개발 초기부터 고민하고 동작하게 초기부터 구현하라. 생각보다 오래걸리고 어려우니까 이미 나온 알고리즘을 검토하라
* 스레드 코드 테스트
  + 문제를 노출하는 테스트 케이스를 작성하여 프로그램 설정과 시스템 설정과 부하를 바꿔가며 자주 돌려라, 테스트가 실패하면 원인을 추적하라. 다시 돌렸더니 통과하더라는 이유로 그냥 넘어가면 절대로 안된다.
  + 고려할 사항
    - 말이 안되는 실패는 잠정적인 스레드 문제로 취급하라
      * 다중 스레드 때때로 말이 안되는 오류를 일으킨다. 시스템 실패를 일회성이라 치부하지 말것
    - 다중 스레드를 고려하지 않은 순차 코드부터 제대로 돌게 하라
      * 스레드 환경 밖에서 코드가 먼저 도는지 확인하고 스레드를 구현하는 것이 좋다.
    - 다중 스레드를 쓰는 코드 부분을 다양한 환경에서 쉽게 끼워 넣을 수 있게 스레드 코드를 구현하라
      * 한 스레드로 실행하거나 여러 스레드로 실행하거나, 실행 중 스레드 수를 바꿔본다
      * 스레드 코드를 실제 환경이나 테스트 환경에서 돌려본다
      * 테스트 코드를 빨리 천천히 다양한 속도로 돌려본다
      * 반복 테스트가 가능하도록 천천히 케이스를 작성한다
    - 다중 스레드를 쓰는 코드 부분을 상황에 맞게 조율할 수 있게 작성하라
    - 프로세스 수보다 많은 스레드를 돌려봐라
      * 스레드를 스와핑 할때도 문제가 발생하기 때문에 스와핑을 잦을수록 임계영역을 빼먹은 코드나 데드락을 일으키는 코드를 찾기 쉬워진다
    - 다른 플랫폼에 돌려보라
    - 코드에 보조 코드를 넣어 돌려라, 강제로 실패를 일으키게 해봐라
      * 직접 구현하기
      * 자동화
* 결론
  + 다중 스레드 코드는 올바로 구현하기 어렵다. 간단했던 코드가 여러 스레드와 공유자료를 추가하면서 악몽으로 변한다. 다중 스레드 코드를 작성한다면 각별히 깨끗하게 코드를 짜야 한다. 주의하지 않으면 희귀하고 오묘한 오류에 직면하게 된다.
  + 무엇보다 먼저 srp를 준수한다. Pojo를 사용하 스레드를 아는 코드와 스레드를 모르는 코드를 분리한다.
  + 동시성 오류를 일으키는 잠정적인 원인을 철저히 이해한다.
  + 사용하는 라이브러리와 기본 알고리즘을 이해한다.
  + 보호할 코드 영역을 찾아내는 방법과 특정 코드 영역을 잠그는 방법을 이해한다.
  + 어떻게든 문제는 생긴다. 초반에 드러나지 않는 문제는 일회성으로 치부하고 무시하기 십상이기 때문