자바 스터디 2주차

- **메모리 가시성**

- Thread간 공유중인 flag값으로 다른 Thread를 중지시킬경우 정상적으로 중지되지 않을 경우 CPU는 처리성능을 위해 캐시 메모리를 사용한다.
- 메인메모리는 CPU입장에서 느리지만 가격이 저렴해 큰 용량을 구성
- CPU연산은 빠르기때문에 속도가 빠른 메모리가 필요함 그래서 속도가 매우빠르지만 가격이 비싸서 용량을 작게 구성하는 캐시 메모리 사용
- 현대의 CPU는 코어단위로 캐시메모리를 보유
- 각 코어의 캐시 메모리에 runFlag가 할당 되어있기때문에 각 스레드별로 바라보는 runFlag값이 다름
- 캐시 메모리에 있는 runFlag값이 메인메모리에 언제 반영되는지 알수 없음 반영 한다해도 캐시메모리에 다시 불러와야하며 이역시 언제 불러올지 모름
- 그럼 언제 반영될까? => CPU의 설계와 환경에 따라 다르지만 일반적으론 컨텍스트 스위칭이 발생할때 반영
- 이렇게 멀티스레드 환경에서 한 스레드에서 변경한 값이 다른 스레드에 언제 보이는가에 대한 문제를 메모리가시성 이라한다.

- **volatile**

메모리 가시성 해결방안이 바로 volatile키워드 이다. 이걸사용하면 바로 메인메모리에 메인메모리에 값이 반영되기 때문에 여러 스레드에서 접근하게 될경우 이를 사용하면 된다. 하지만 성능이 약5배정도 떨어진다는 단점이 있다.

- **자바 메모리 모델(JMM)**

자바 프로그램이 메모리에 어떻게 접근, 수정하는지 규정한다. 즉, 여러 스레드들의 작업 순서를 보장하는 happens-before 관계에 대한 정의

happens-before 관계로 메모리 가시성을 보장한다.

happens-before 관계의 예시
1. 코드가 순서대로 실행하는것
2. Volatile
3. 스레드 시작 규칙(start의 이전과 후시점)

4. 스레드 종료 규칙(join이 반환 된 후와 이전의 시점)
5. 인터럽트 규칙(인터럽트의 호출 시점과 감지하는 시점)
6. 객체 생성
7. Synchronized 블록 내 시점과 나가는 시점에 대한 시점

정리하면 volatile 또는 스레드 동기화 기법을 사용하면 메모리 가시성 문제를 해결 할 수 있다.

- **메모리 동기화 기법**

- bank예제(동시성 문제)

각각의 t1, t2 스레드 스택프레임에서 this로서접근하는 인스턴스는 같은 인스턴스이다.
balance가 공유자원이 되는것

balance에 volatile을 적용해도 문제가 발생한다. 이 문제는 메모리 가시성이 문제가 아니기 때문이다. volatile은 한 스레드의 값이 변경됐을때 다른 스레드들이 즉시 볼 수 있게하는 것일뿐이다.

문제 원인은 공유자원을 여러 단계로 나누어 사용하기 때문이다. 즉, 검증단계 이후 출금단계에도 잔액이 변하지 않았다는 가정때문이다.

이렇게 여러 스레드가 동시에 접근가능한 공유자원이 있는 부분을 임계영역이라 하며 이부분은 synchronized 키워드로 보호 가능하다. synchronized키워드로 감싸게 되면 해당 메소드에는 스레드가 순차적으로 접근한다.

- **synchronized**

- 모든 객체는 자신만의 lock을 가지고있다. 이를 ‘monitor lock’이라한다. 스레드가 synchronized 키워드의 메소드에 진입하려면 해당 인스턴스의 lock이 있어야한다.

- 스레드가 lock을 획득하면 메소드에 진입이 가능하며 lock을 획득하지 못하면 CPU 실행 스케쥴링에 들어가지 않는 BLOCKED 상태로 대기한다. 메소드의 실행이 끝나면 lock을 반납한다.

- lock을 획득하는 순서는 보장되지 않는다. volatile을 사용하지 않아도 메모리가시성의 문제를 해결 할 수 있다.

- 단점은 한번의 하나의 스레드만 실행이 가능하기때문에 성능저하가 있다는것이다. 그래서 synchronized 구간을 최소화 하고자 코드블록을 이용한다.

- 무한대기, 공정성에 대한 단점이 있다.

- **LockSupport**

- synchronized의 무한대기 단점을 보완한다.

- LockSupport의 park(), unpark()메소드로 스레드를 WATING상태로 변경하거나 RUNNABLE상태로 변경 가능하다.

- Blocked상태는 synchronzied에서 락을 얻기위해 대기하는 특별한 상태로 interrupt()를 걸어도 못 깨어난다.

- Waiting 상태는 스레드가 특정조건이나 시간동안 대기하는 상태로 interrupt()로 중간에 깨울수 있다.

- lock을 직접 구현해서 각 스레드가 lock을 가질수 있게 구현해야하는데 이는 너무 저수준이다. 그래서 실제로는 잘 안쓴다

– **ReentarntLock**

- synchronized의 무한대기와 공정성의 단점을 보완한다.

- lock() : 다른 스레드가 락을 얻으면 풀릴때까지 WATING하며 인터럽트에 응답하지 않는다.(인터럽트가 걸리면 RUNNABLE로 변하지만 다시 WATING으로 변경된다) 이때 락은 모니터락이 아닌 ReentarntLock에서 제공하는 락이다.

- lockInterruptibly() : lock()과 유사하지만 인터럽트에 응답한다.

- tryLock(time) : 락 획득을 시도하고 성공여부를 바로 반환한다. 파라미터로 시간을 받으면 특정 시간만큼만 기다린다.(내부에서 parkNanos()사용)

- unlock() : 락을 해제한다. 락이 없는 스레드가 호출하면 예외 발생

- newCondition() : Condition 객체 생성후 반환한다.

- ReentrantLock(true)로 생성하면 fair mode로 먼저 대기한 스레드가 락을 얻게하지만 성능저하가 있다.

- lock()을 하면 반드시 unlock()을 해야한다.

질문)

1. 중간에 Sleep 없으면 왜 멈추지??

2. synchronzied에서 this외에 다른게 들어가면?