## 원자적 연산 (atomic operation)

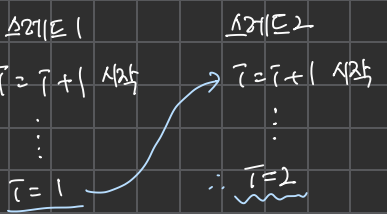
→ 해당 연산이 더 이상 나눌 수 없는 단위로 수행된다는 것
(다른 연산 간섭 X, all or nothing)

ex) volatile int i = 0;

- 쪼갤 수 없는 원자적 연산 i = 1
- 원자적 연산이 아님 i++
  1. i의 값을 읽음
  2. 읽어온 i에 1을 더함
  3. 계산된 값을 i에 대입

  ) synchronized 나 Lock을 사용해서 임계영역 안에서 처리

① 스레드 2개가 순차적으로 실행




② 스레드 2개가 동시에 실행

스레드1 i = i+1 수행
스레드2 i = i+1 수행
스레드1 i의 값을 읽음 (0)
스레드2 i의 값을 읽음 (0)
스레드1 0+1 → 1
스레드2 0+1 → 1
스레드1 i 변수에 1을 대입
스레드2 i 변수에 1을 대입

⇓

∴ i = 1

## volatile과 synchronized

- volatile은 CPU 사이에 발생하는 캐시 메모리와 메인메모리가 동기화되지 않는 문제를 해결, 모든 연산을 원자적으로 묶어주지 않음!
- synchronized은 임계영역 안에서 안전하게 수행됨

AtomicInteger - 멀티스레드에서 안전한 연산을 할 수 있는 클래스
특정 값을 공유하여 증가, 감소 등 연산을 처리해야할 때

※ 성능 비교

- BasicInteger(멀티스레딩 고려X) : 39ms
- VolatileInteger(volatile) : 455ms → 임계영역X, 멀티스레딩환경에서 사용불가
- SyncInteger(synchronized) : 625ms → 멀티스레딩 환경 사용O, 느림
- MyAtomicInteger(AtomicInteger) : 369ms → 멀티스레딩O, synchronized/ReentrantLock보다 빠름

참고

네, 맞습니다! JVM은 **탈출 분석(Escape Analysis)**이라는 기술을 사용하여 `synchronized` 블록을 최적화할 수 있습니다. 탈출 분석을 통해 객체가 해당 스레드 밖으로 "탈출"하지 않는다는 사실을 알 수 있다면, JVM은 특정 경우에 `synchronized` 블록을 제거하거나 최적화할 수 있습니다.

탈출 분석은 컴파일러가 객체가 메서드나 스레드의 범위를 벗어나지 않는다는 것을 감지할 때 작동합니다. 예를 들어, 어떤 객체가 단일 메서드 내부에서만 사용되고 다른 스레드로 공유되지 않는다면, 그 객체는 "탈출"하지 않는 것으로 간주됩니다. 이 경우 JVM은 동기화 비용을 제거하거나 줄이는 최적화를 적용할 수 있습니다.

**대표적인 최적화 방법:**

1. **스레드 로컬 객체**: 객체가 다른 스레드로 공유되지 않는다고 판단되면, `synchronized` 블록이 의미 없으므로 JVM은 해당 동기화 코드를 최적화하여 제거할 수 있습니다.
2. **스택 할당 최적화**: 탈출하지 않는 객체는 힙이 아닌 스택에 할당될 수 있고, 이를 통해 동기화 비용뿐만 아니라 메모리 할당 비용도 절감할 수 있습니다.

하지만, 이러한 최적화는 JVM에 따라 다를 수 있고, JVM이 항상 탈출 분석을 적용한다고 보장할 수는 없습니다. 따라서 개발자가 수동으로 동기화를 관리하는 것이 더 안전한 경우가 많습니다.

**결론:** 네, 탈출 분석 덕분에 JVM이 일부 경우에 동기화를 제거할 수 있지만, 이 최적화는 자동으로 항상 적용되는 것이 아니므로 동기화가 정말 필요 없는 상황이라면, 코드 레벨에서 불필요한 동기화를 제거하는 것이 좋습니다.

## CAS 연산

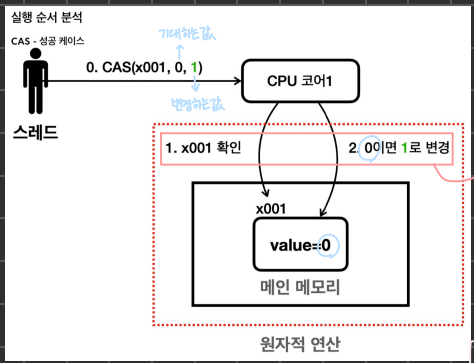
락 기반 방식의 문제점 - 락획득 대기, 획득, 해제 등 추가적으로 시간 소요

⇓ Lock-free

CAS(Compare-And-Swap, Compare-And-Set)

※ 완전히 락을 대체하지 않고 "작은 단위의 일부 영역에 적용"

compareAndSet(0,1) : atomicInteger가 갖고 있는 값이 현재 0이면, 이 값을 1로 변경하라는 메서드
↳ 원자적으로 실행
- 변경 성공시 true, 현재 값이 0이 아니라면 변경되지 않고 false




2개로 나뉘진 명령어(원자적이지 않음)

CAS는 원자적이지 않은 두개의 연산을 CPU에서 특별하게 하나의 원자적인 연산으로 묶어 제공하는 기능

↓

1번과 2번 사이에 다른 스레드가 x001의 값을 변경하지 못하게 막음

[① 기대하는 값인지 확인 ② 값 변경 연산]

<하드웨어>

lock을 걸지않는 연산

```
private static int incrementAndGet(AtomicInteger atomicInteger) {
    int getValue;
    boolean result;
    do {
        getValue = atomicInteger.get();
        log("getValue: " + getValue);
        result = atomicInteger.compareAndSet(getValue, getValue + 1);
        log("result: " + result);
    } while (!result);
    return getValue + 1;
}
```

성공 true
실패 false

충돌이 발생하더라도 while문에서 다시 시도하므로 락없이 데이터를 안전하게 변경가능

충돌이 빈번하게 발생할 경우, 성능 이슈 발생

→ 여러 스레드에서 자주 동시에 동일한 변수의 값을 변경하려하면 CAS도 자주 실패하고 재시도해야 하므로 성능 저하발생우려 ⇒ 그만큼 CPU 낭비

| Lock 방식 | CAS 방식 |
|---|---|
| · pessimistic 접근법 | · optimistic 접근법 |
| · 데이터에 접근하기 전에 항상 락획득 | · 락없이 바로 데이터 접근 |
| · 다른 스레드의 접근를 막음 | · 충돌 발생시 그때 재시도 |
| · 다른 스레드가 방해할 것이다 라는 가정 | · 대부분 경우 충돌이 없을 것이다 가정 |

언제 CAS를 사용하면 좋은지? 충돌이 많이 발생하는 연산은?

ex) 1000개 스레드를 동시에 생성하고 동시에 실행

- 락방식 : 1000개 스레드 모두 락 획득하고 반환, 순서대로 하나씩실행 → 50개 충돌
- CAS방식 : 1000개 스레드 모두 한 번에 실행 → 충돌이 나는 50개 경우만 재시도

간단한 CPU연산에서는 우세

CAS(Compare-And-Swap) 연산에서 충돌이 많이 발생하는 상황은 주로 여러 스레드가 동시에 동일한 메모리 위치에 대해 변경을 시도할 때 발생합니다. 다음은 CAS 충돌이 빈번하게 발생하는 일반적인 상황들입니다:

1. **경합이 심한 공유 자원**: 다수의 스레드가 동시에 동일한 공유 자원에 접근해 CAS 연산을 수행하는 경우, 충돌이 발생할 가능성이 높습니다. 스레드들이 경쟁적으로 데이터를 수정하려 할 때 CAS 연산이 성공하지 못하고 재시도하게 됩니다.
2. **짧은 연산 사이클**: CAS 연산이 적용되는 코드의 연산이 매우 짧을 때 충돌 가능성이 큽니다. 즉, 스레드들이 아주 짧은 시간 동안 메모리의 특정 위치에 대해 연산을 반복적으로 시도하면 CAS 연산의 성공률이 떨어질 수 있습니다.
3. **다중 CPU 환경**: 다중 코어 또는 다중 프로세서 시스템에서 여러 CPU가 동시에 동일한 메모리 위치에 대해 CAS 연산을 수행하려고 하면 충돌이 자주 발생할 수 있습니다. 특히, 캐시 일관성을 유지하기 위한 오버헤드가 커지면서 성능 저하도 일어날 수 있습니다.
4. **큰 객체나 복잡한 상태 업데이트**: CAS가 적용되는 데이터가 복잡하거나 큰 경우, 여러 스레드가 각기 다른 값을 예상하고 동시에 변경하려 하면 충돌이 자주 발생할 수 있습니다. 예를 들어, 링크드 리스트나 트리 구조와 같은 복잡한 자료구조를 다루는 경우입니다.
5. **빈번한 스레드 컨텍스트 전환**: 스레드가 자주 중단되고 다시 시작되는 환경에서는 CAS 연산에서 충돌이 자주 발생할 수 있습니다. 스레드가 특정 작업을 완료하기 전에 다른 스레드가 동일한 메모리 위치에서 CAS 연산을 시도하는 경우 충돌 가능성이 증가합니다.

??

# CAS 단점

Runnable 상태로 락을 얻을 때까지 while문을 반복함
→ 락을 기다리는 스레드가 CPU를 계속 사용하면서 대기함
(blocked, waiting 상태에서는 CPU를 거의 사용X)

스레드를 계속 runnable 로 살려둔 상태에서 계속 락 획득을 반복체크하는것이
더효율적일 때 사용! → 스레드의 상태를 전환 하지 않아 매우 빠르게 락을 획득하여 실행 가능

- 임계영역은 짧아지만 연산이 길지 않고 영원! 짧게 끝날 때 사용
  ex) 숫자 값증가, 자료구조의 데이터 추가 O
  I/O, DB 작업 X

SpinLock - 스레드가 락이 해제되기를 기다리면서 반복문을 통해 계속해서 확인
(제자리에서 spin 하는 것같다고 SpinLock)

- Busy-wait 방식 ( CPU 자원을 계속 사용하면서 바쁘게 대기 )

〈정리〉

| | CAS | 동기화락 |
|---|---|---|
| 장점 | 낙관적 동기화<br>Lock-free | 충돌 관리<br>안정성<br>스레드 대기 (runnable X) |
| 단점 | 충돌이 빈번할 때<br>성능저하 | 락 획득 대기 시간<br>context switching의<br>오버헤드 |