멀티스레드 프로그래밍을 하면 안되는 이유

1. 컴파일러가 최적화를 하면서 생각한데로 컴파일을 하지 않는다.

* Volatile (컴파일러 최적화를 하지 않겠다)

1. CPU -> atomic\_thread

* Out of order, write buffer를 기다려라. (단 무자비로 쓰면 속도저하를 야기한다.)

1. Cache(캐시)( MESI, 캐시라인 걸치기)

* Pointer 조심, Pragma pack 조심

1. ABA

같은 문제들이 있다.

공유 메모리이기 때문에 읽고 쓸 때 순서가 뒤죽박죽이 된다.

이러한 대책을 고려하며 멀티 스레드 프로그래밍을 해야 한다.

thread\_fence => CPU 문제 (atomic하게)

# 희망

* 메모리에 대한 쓰기는 언젠가는 완료 된다.
* 반드시 된다!
* 자기 자신의 “프로그램 순서”는 지켜진다.
* 캐시의 일관성은 지켜진다.
* 한번 지워졌던 값이 다시 살아나지 않는다. (절대로)
* 언젠가는 모든 코어가 동일한 값을 본다.
* 캐시라인 내부의 쓰기는 중간 값을 만들지 않는다.
* 너무 임펙트가 없다. (이것만으로 지켜지기는 힘들다.)
* 우리가 할 수 있는 것 => 소프트웨어적인 방법으로 Atomic Memory 구현
* Atomic
* 접근의 절대 순서가 모든 쓰레드에서 지켜지는 자료구조
* 싱글코어는 모든 메모리가 Atomic Memory 이다.

- Mutex를 사용 -> Atomic

- C++11 Atomic<int> a; a.store(3), a.load() -> Atomic

- Class sw\_atomic\_int -> 메서드 Load, Store -> Atomic

1. load() => ‘=’ 연산자도 Atomic하게 동작한다.

* <atomic>에 대해 알아보자.

a.store(1, memory\_order\_...)

default => memory\_order\_acquire

(대학원 과정이므로 …)

atomic\_thread\_fence를 필요한 곳에 쓰는 것이 atomic을 남발한 것보다 훨씬 빠르다.

이제 최적화와 CPU문제는 Atomic을 이용하여 해결이 되었다.

* Atomic Memory만 있으면 되는가?
* NO!
* 실제 상용 프로그램을 int, long, float같은 기본 data type만으로 작성할 수 있는가?

(map<int(id), PLAYER> 🡨 전역변수, 공유메모리변수) 게임서버로 쓴다고 가정했을 때

위 자료형을 atomic하게 변수로 만들수 있을까? => 비주얼 스튜디오에서 컴파일 에러

조금이라도 복잡한 자료구조는 atomic하게 만들 수 없다.

* 효츌적인 Atomic 자료구조가 필요하다.
* 일반 자료구조에 Lock을 추가하면 => 잘 돌아가지만 너무 느리다.
* STL을 써서 Atomic하게 => 멀티 쓰레드를 생각하지 않고 만든 라이브러리이므로 CRASH
* STL + LOCK => 느려서 못씀
* 이걸 해결할 것은…. =========🡺 Non-Blocking 프로그래밍 (Lock을 없앤다.)\

# Lock없는 프로그램

* 효율적인 구현
* 성능 저하의 주범이므로 당연
* Overhead 발생
* 동시에 실행되지 않는다. 상호배제 (이건 병렬성이 없다는 것과 같다.)
* Priority inversion
* Convoying
* Lock이 없다고 성능저하가 없는가?? NO!
* 상대방 쓰레드에서 어떤 일을 해주기를 기다리는 한 동시 실행으로 인한 성능 개선을 얻기 힘들다.

=>while(other\_thread.flag==true); 옆 쓰레드에 의존적인 작업들…

=>loack과 동일한 성능저하

\*(이러한 프로그램을 블록킹 프로그램이라고 한다.)\*

* 블록킹(blocking)
* 다른 쓰레드의 진행상태에 따라 진행이 막힐 수 있음. Ex) while(lock != 0)
* 멀티쓰레드의 bottle neck이 생긴다.
* lock을 사용하면 블록킹
* 넌블럭킹(non-blocking)
* 다른 쓰레드가 어떠한 삽질을 하고 있던 상관없이 진행…

예) 공유메모리 읽기/쓰기, Atomic(Interlocked) Operation

Atomic<int> a;

a.store(3) <= 넌블럭킹

volatile int sum;

sum += 2; <= 넌블럭킹

* 왜넌블럭킹? => 블록킹 알고리즘의 문제
* 성능 저하
* Priority Inversion

더 우선순위가 높은 작업이 빨리 실행되어야 하는데.. 그걸 lock으로 막는 것.

Reader/Write Problem에 많이 발생

* Convoying

한번 lock을 하면 다른 쓰레드는 unlock을 할때까지 기다려야 한다.

Context Switch => Unlock이 안되는 상황

코어 수가 많을수록 그 손해는 점점 더 커진다.

그러므로 코어보다 스레드 개수를 많이 만들면 안된다. => Context Switch를 예방

* 성능이 낮아도 Non-blocking이 필요할 수 있다…
* 넌블럭킹의 등급
* 무대기(wait-free)

모든 메소드가 정해진 유한한 단계에 실행을 끝마침

멈춤 없는 프로그램 실행

* 무잠금(lock-free)

항상, 적어도 한 개의 메소드가 유한한 단계에서 실행을 끝마침 – 한도 끝도 기다리는게 아니라 (limit) 범위만큼 기다렸으면 다른 스레드 상관없이 그냥 실행 or 충돌이 되면 지연은 되겠지만 다른 쓰레드 상관없이 그냥 실행

무대기이면 무잠금이다. == wait-free

기아(starvation)을 유발하기도 한다. => 근데 잘 안 발생하기 때문에 신경쓰지 않을 수 있다.

성능을 위해 무대기 대신 무잠금을 선택하기도 한다. => 성능이 중요!

알고리즘이 무대기에 비해 훨씬 더 간단하기 때문에 오버헤드가 적다.

Non-Blocking Programming == lock-free Programming

정리

* Wait-free, Lock-free => Lock을 사용하지 않고..
* 멀티 쓰레드 프로그램에서 쓰레드 사이의 효율적인 자료 교환과 협업을 위해서는 Non-Blocking 자료 구조가 필요하다.