Nonblocking programming

1. 정확성 (컴파일러, CPU, Cache, ABA)
2. 성능 -> Blocking, Waiting 때문에

Algorithm

1. Copy
2. ALU
3. If + goto
4. CAS

* 현실의 공유 메모리는 Atomic하지 않다.
* 하지만 프로그래밍은 Atomic Memory를 가정해야 한다.

Atomic으로 만들어주는 자료구조.

* Wait-free를 유지하면서는 불가능하다..(증명이 되어 있음)

+알파가 뭐냐?(Wait-free를 유지하면서) => CAS

* CAS(Compare And Set)연산

메모리 값이 expected면 update로 바꾸고 true를 리턴

메모리 값이 expected가 아니면 false를 리턴

- 조건부 대입(store) -> 쓰기 전에 값을 한번보고 맞으면 쓰고 아니면 안쓰고

- CAS는 CPU가 기계어로 지원을 해줘야 돌아간다.

변환(Lock-Free) 알고리즘만 써도 멀티스레드 non-blocking으로 구현이 가능하다.

하지만 성능이 나오지 않는다. (코어 수가 늘어날수록 더욱)

싱글스레드 보다 성능이 좋게 나와야 멀티스레드를 쓰는 것이다.

* 날로 먹을 수 없다!
* 대안 (최적화)

-알고리즘 별로(자료구조에 맞추어) 따로따로 최적화 구현을 해주어야 한다. (힘들다)

(최적화된 lock-free 알고리즘을 일일이 개발해야 한다.)

* 다른 데서 구해 쓸 수도 있다.

Intel TBB, VS2015 PPL

* 하지만 범용적일수록 성능이 떨어진다. 자신에게 딱 맞는 것을 만드는 것이 좋다.

# 정리

* 성능 향상을 위해 멀티스레드 프로그래밍을 해야 한다.
* Data Race라는 괴물과 싸워야 한다.
* Data Race를 최소화 해야 한다.
* Data Race는 모든 오 동작의 근원

…

# cas 사용법

#include<atomic>

Atomic\_compare\_exchange\_strong (기계어) : 비교를 해서 같으면 대입

Strong, Weak 두 종류가 있다.

Strong : 같으면 무조건 써라.(강력한) – Intel CPU는 strong밖에 없음

Weak : 같음에도 불구하고 안써질 수 있다.(허술한) – ARM 모바일 CPU는 Weak밖에 없음(그래서 실패하면 루프를 돌면서 재실행한다.)

Atomic\_int == Atomic<int>

Int x; 값이 => 0 : lock-free, 1 : 누군가 lock을 가지고 있다.

실제 HW => lock cmpxchg [A], b

* 빠르고 정확한 병렬 프로그램을 작성하는 것은 어렵다. (두마리 토끼를 다잡기)
* 포기하는 것이 불가능하다면 주의해서 프로그래밍 해야 한다.
* Atomic 변수나 mfence의 도움이 필요하다.
* 스레드 간의 동기화를 위한 자료구조가 필요하다.
* Non-Blocking 자료구조가 필수
* Non-Blocking 자료구조에는 CAS가 필수 (간단한 계산은 굳이 CAS를 쓸 필요가 없음)