메모리 일관성 문제 정리

- 멀티스레드에서의 공유 메모리
  - 다른 코어에서 보았을 때 업데이트 순서가 틀릴 수 있다.
  - 메모리의 내용이 한순간에 업데이트되지 않을 때도 있다.
- 일반적인 프로그래밍 방식으로는 멀티스레드에서 안정적으로 돌아가는 프로그램을 만들 수 없다.

여태까지 나왔던 멀티스레드 프로그래밍을 하면 안 되는 이유가 3개가 있었다.

1. 컴파일러 최적화 문제. - c언어가 멀티스레드용 언어가 아니기 때문에 최적화를 하면서 우리 생각대로 컴파일을 하지 않았다. 그래서 volatile를 사용해서 해결했다.

2. cpu - out of order / write buffer 문제 - cpu의 속도를 높이기 위한 이런 유용한 기법이 있어서 지금의 속도가 나오는 것이다. 그런데 이런 기법들이 멀티스레드를 생각 안 하고 싱글코어 시절에 나온 기법이라는 것이 문제다. 멀티스레드 멀티코어에서 이런 기법을 사용하면 읽고 쓰는 순서가 엉망이 된다. 그래서 atomic_thread_fence라는 c++11의 기능을 써서 out of order write buffer 실행을 막아야 한다. 기능을 끄는 게 아니라, 명령어 앞 뒤로 실행 순서를 절대로 바꾸지 말라는 명령을 하는 것이다.

3. 캐시 cache 문제 - 캐시는 멀티코어를 생각하지 않고 만든 것이다. 아무 생각 없이 했다가는 멀티코어에서 데이터가 엉망이 된다. 그래서 인텔에서는 멀티코어 만들 때 캐시에 여러 부가 장치를 덧붙였다. MESI 같은 알 필요 없는 프로토콜로 캐시가 제대로 동작하게 했지만, 캐시라인에 걸치게 되면 결과가 중간값이 나오는 현상이 일어난다. 완전하지 않기 때문이다. 이걸 막는 방법은 조심하는 방법밖에 없다. 포인터를 쓸 때 조심하고, pragma pack을 쓸 때 조심해야 한다.

(4. ABA 문제 - 사실 하나 이게 더 있다. 이거는 다다음달에 할 거고, 이걸 하기 위해서는 배워야 할 게 많다.)

이렇게가 프로그래밍을 하면 안 되는 문제들이었다. 이것만 조심하면 멀티스레드 프로그래밍할 때 에러 없이 잘 돌아갈 수 있다. 그런데 다 하나같이 까다로운 것뿐이다.

컴파일러 문제는 volatile 붙이면 되는데 마구 붙이면 또 성능이 떨어지고. cpu는 지구 상에 마음 놓고 쓸 게 하나도 없는 데다가 atomic_thread_fence 이것도 명령어 하나하나 껴 넣으면 out of order write buffer 문제는 없는데 마찬가지로 느려진다. 캐시는 조심하면 2번 access 할 때 1번 하게 된다면 성능은 좋아져서 좋지만, 프로그래밍을 할 때 신경 쓸 게 많아 까다롭다.

스레드끼리 서로 데이터를 주고받는 동기화를 해야 하고, 공유 메모리를 읽고 쓰고 하는데 옆 코어에서 봤을 때 읽고 쓰는 순서가 엉켜 보인다. 어떡하느냐? 공유 메모리를 쓰는 애만의 문제도, 읽는 애만의 문제도 아니다. 둘 다 순서가 꼬여서 어떻게 꼬일지 모른다. 그래서 아무 생각 없이 싱글 스레드 짜듯 짜면 안 된다. 대책이 있다. 이걸 주의하고 잘해야 한다.

대책은? 어떻게 할 것인가?

- 위의 상황을 감안하고 프로그램 작성?
  - 프로그래밍이 너무 어렵다
  - 피터슨 알고리즘도 동작하지 않는다
- 강제로 원하는 결과를 얻도록 한다
  - 모든 메모리 접근을 CRITICAL_SECTION으로 막으면 가능
    - 성능 저하!!!
  - mfence명령어를 적절한 위치에 추가
    - 찾은 위치가 완벽하다는 보장은?
- 해결 방향
  - 꼭 필요한 곳에 mfence 사용
  - 공유 변수 대신 atomic 한 자료구조 사용

프로그래밍이 너무 어렵다. 하나하나 주의하여 짜야하는데 쉽지 않다. 피터슨 알고리즘도 제대로 돌아가지 않고, atomic_thread_fence 같은 거 넣어줘야 하고.

한방에 해결하는 방법은? mutex를 쓰면 된다. 모든 공유 변수 읽을 때 mutex 하면 되는데 이렇게 하면 성능이 뚝 떨어진다. 함부로 넣을 수 없다. atomic_thread_fence를 적절한 위치에 잘 넣으면 되느냐? 추가를 하면 cpu문제는 막을 수 있으나 캐시 문제는 막을 수 없다. 그리고 컴파일러는 또 다른 문제이다. 성능이 떨어지니 꼭 필요한 곳에만 넣어라. 꼭 필요한 곳이 어디냐? 알기 어렵다. 제대로 돌아가는 프로그램을 짜려면 수학적으로 증명을 해야 안심을 하고 넘어갈 수 있는데, 그 증명은 대학원에 가야 배운다.

공유 변수로 그냥 int 쓰지 않고 atomic 한 메모리를 사용하면 일관성 문제가 해결되니 atomic 변수를 쓰는 것도 나쁘지 않다. atomic을 잠시 뒤 알아보자.

희망 1

- 메모리에 대한 쓰기는 언젠가는 완료된다.
- 자기 자신의 프로그램 순서는 지켜진다.
- 캐시의 일관성은 지켜진다
    - 한번 지워졌던 값이 다시 살아나지 않는다
    - 언젠가는 모든 코어가 동일한 값을 본다
- 캐시라인 내부의 쓰기는 중간 값을 만들지 않는다

인텔은 자기네들이 할 수 있는 조치는 다 했다. 그러나 성능이 떨어지는 결과를 가져오는 조치는 할 수 없었다. 그러면서도 동시에 프로그래밍이 너무 까다롭고 어려워서 아예 못하는 상황을 막는 최소한의 조치를 한 것이다. 그 조치가 무엇이냐? 이것들은 반드시 지켜진다. 이 사항들은 엉키지 않도록 cpu를 만들었다.

1. 메모리에 대한 쓰기는 언젠가는 완료된다

메모리에 썼다 그럼 메모리에 들어간다. 언제 들어가냐? 어떤 순서로 들어가냐? 그건 정해지지 않았다. 하지만 언젠가는 반드시 들어간다. 도중에 날아가지 않는다.

2. 자기 자신의 프로그램 순서는 지켜진다

이게 어긋나면 싱글 스레드 프로그램도 제대로 안됐을 것이다. 내가 썼으면 그 데이터가 반드시 읽힌다. 내가 봤을 때 내가 읽고 쓴 건 항상 올바르게 보인다.

3. 캐시 일관성은 지켜진다

메모리에 어떤 데이터를 쓰면 옛날에 있던 데이터는 지워진다. 근데 이렇게 난리법석을 치다 보면 내가 쓴 3이 나왔는데 혼란한 와중에 썼던 3이 날아가고 그전에 들어있던 0이 슬그머니 되돌아가냐? 그런 거 없다. 옛날 값이 살아나지 않는다. 한번 덮어쓰면 끝이다. **그리고 시간이 지나면 언젠가는 모든 코어가 동일한 값을 본다.** 0(아직 3이 안 써진 상태)을 보던가 3(써진 상태)을 보던가 하나로 통일되지 끝까지 갔을 때 둘이 다른 값을 보지 않는다. 한 코어는 2를 쓰고 한 코어는 3을 쓰면, 코어마다 2를 보던지 3을 보던지 다른 값을 볼 수 있는데 마지막에 가면 같은 값을 본다. 끝까지 2만 읽고 3만 읽고 그러지 않는다.

4. 캐시라인의 내부의 쓰기는 중간 값을 만들지 않는다

캐시라인에 걸쳐지지만 않으면 -1을 쓰면 이게 써지고 0을 쓰면 0을 써진다. 65535 같은 이상한 값은 읽히지 않는다.

이 내용들은 반드시 지켜진다. 근데 너무 약하다. 좀 더 강력하게 지켜줘야 하는데 너무 약해서 이것만 믿고 프로그래밍하는 게 쉽지 않다.

희망 2

- 우리가 할 수 있는 것
  - cpu의 여러 삽질에도 불구하고 atomic memory를 구현할 수 있다.
    - 더군다나 mutex나 mfence 없이 SW적인 방법 만으로 CPU에서 특수 명령어로 구현되어 있다. C++11에서 사용.
- Atomic
  - 접근(메모리는 read, write)의 절대 순서가 모든 스레드에서 지켜지는 자료구조
  - 프로그래머가 필요로 하는 바로 그 자료구조
  - 싱글코어에서는 모든 메모리가 atomic memory이다.

지금까지는 싱글 스레드 프로그램에서는 모든 명령어가 atomic 명령어기 때문에 쓰면 써지고 읽으면 최근 값이 읽혀서 신경도 안 쓰는 거였는데 멀티코어로 오면서 atomic이라는 개념이 중요해졌다. atomic 메모리. 읽고 쓰는 순서가 모든 스레드에서 똑같은 순서로 보이는 메모리가 atomic 메모리이다. 우리는 여태까지 우리가 쓰는 메모리가 atomic이라고 가정하고 해왔다. 싱글 스레드에서는 atomic이라는 개념도 없었다.

어떻게 구현하는가? 일단 int a 이런 atomic이 아니다. volatile 이것도 atomic이 아니다. **mutex를 쓰면 atomic이다.** 앞 뒤로 쓰면 atomic이 되는 것. **그리고 fence를 써도 atomic 하게 구현하는 것이다.** 근**데 이런 거 안 써도 소프트웨어적으로 구현할 수 있는 방법이 있다.** 알고리즘이 있다. int가 아니라 class sw_atomic_int {load() start()} 이런 식으로 하는 건데 방법은 안 알려줄 거다. 굉장히 복잡한 내용이다. **대학원 내용이고 오버헤드가 크다**는 것만 알아둬라.

다행스럽게도 우리가 사용하는 cpu에는 메모리 읽고 쓰는 것을 atomic 하게 해주는 특수 명령어가 있다. 이걸 사용하면 메모리 읽고 쓸 때 atomic 하게 순서 안 바뀌게 읽고 쓸 수 있다. c++11에서도 atomic 메모리가 구현되어있다. cpu에 구현된 명령어를 사용해서 구현한 것이다. 소프트웨어가 아니라 하드웨어로 구현되어 있는 방법이다. 오버헤드도 그렇게 크지 않다.

C++11의 Atomic Memory

```
#include <atomic>

atomic <int> a;
a.store(3);
int d = a.load();
```

- **SW로 구현할 수도 있지만 오버헤드가 커서 HW적인 방법으로 구현된다.**

A라는 int를 쓰고 싶은데 읽고 쓰는 순서를 cpu가 바꾸면 안 된다. 순서대로 읽고 쓰게 만들어야겠다 하면 이렇게 하면 된다. atomic이라고 하는 것과 타입을 넣어주면 a에 들어가 있는 int 값이 3이 들어가는데 atomic 하게 들어가고, 3을 넣고 d가 a를 읽을 때 그 순서는 절대로 바뀌지 않는다. 안심하고 쓸 수 있다.

그런데 int인데 그냥 d = a; 하면 되는 거 아니었냐. load() 이런 거 아토믹이라고 귀찮게 붙여야 하는가. 아니다. 그냥 d=a; 해도 똑같이 atomic 하게 돌아간다. 그래도 a.load()랑 똑같이 돌아간다. c++의 편리한 점이다. 오퍼레이터를 자기가 정의해서 쓸 수 있다.

그리고 아까 얘기했듯 위에는 c++11의 라이브러리를 쓰는 표준이고, 이건 cpu에 있는 특수 명령어로 구현이 되어있다. 그래서 cpu의 도움 없이 sw로 할 수 있는데 오버헤드가 커서 잘 사용은 안 한다.

```
#include <atomic>
#include <memory>
#include <iostream>

using namespace std;

int main()
{
	atomic<int> a, b, l;
	int p;

	a.store(1, memory_order_release);
	b.store(2, memory_order_release);
	int c = a.load(memory_order_acquire);
	c = a.load(memory_order_seq_cst);
	b.store(3, memory_order_seq_cst);
}
```

그러면 store load 하면 끝이냐? 아니다. 인터넷 뒤져보면 memory order 어쩌고 하면서 이상한 명령어를 적게 되어있다. 이게 뭐냐? 읽고 쓰면 끝인 줄 알았더니 이건 뭐 하는 거냐? 여기서 다룰 내용은 아니다. *(교수님도 학교 다니실 때 학부도 석사도 아니고 박사과정 때 배운 거라고 하셨다)*

이것만 알아둬라. memory_order_seq_cst 이게 기본이다. 이걸 사용하면 atomic 하게 된다. 다른 걸 사용하면 반 정도 atomic 하게 된다. 반 정도 되는 게 무슨 소용이냐? 소용 있다. 읽고 쓰는 순서가 뒤바뀔 수 있는데 위엣거랑은 바뀌지만 아랫 거랑은 안 바뀌고, 이렇게 반만 제한하는 키워드들이다. 파이프라인 스톨을 좀 더 줄일 수 있다. 그러나 이해하는 게 쉽지 않아서 넘어가자.

Atomic

- C++11에서의 Atomic 메모리의 종류
  - csq_cst : Sequential Consistency
  - relexed : Relexed Consistency
  - acquire/release : Release Consistency
- 교재에서의 Consistency
  - Quiescent Consistency
  - Sequential Consistency
  - Linearization

우리는 csq_cst 이걸 써야 하고 이게 기본이다. relaxed는 대학원 수준 안 가야 하고 교재에는 relaxed, acquire/release이건 없다. 그리고 교재에 있다는 Quiescent, Linearization 두 개는 해당사항이 없는데 Sequential Consistency 이거는 csq_cst랑 이름도 같다. 같은 거다.

간단하게 설명하면 Linearization는 Sequential보다 헐렁한 내용이다.

실습

- atomic <int>와 atomic <bool>을 사용하여 peterson알고리즘을 구현해보자
- 그냥 구현했을 때, _asm mfence를 넣었을 때와 속도 비교를 하라

지난번에 피터슨 알고리즘을 volatile int로 했는데 결과가 이상했다. 제대로 결과가 나오기 위해 atomic_thfread _fence를 했는데 이번엔 그거 안 하고 이렇게 해도 된다. 한번 해보자

```cpp
#include <thread>
#include <mutex>
#include <chrono>
#include <vector>
#include <atomic>
using namespace std;
using namespace chrono;

//피터슨
volatile int sum;


atomic<int> victim = 0;
atomic<bool> flag[2] = { false, false };

//피터슨 락과 언락으로 구현하기
void p_lock(int myid)
{
	int other = 1 - myid;
	flag[myid] = true;
	victim = myid;
	//atomic_thread_fence(memory_order_seq_cst);
	while ((true == flag[other]) && victim == myid); //옆에 스레드가 false일때까지 기다려라.

}
void p_unlock(int myid) //일이 끝나면 flag를 false로.
{
	flag[myid] = false;
}


void do_work2(int num_thread, int myid) {

	for (int i = 0; i < 50000000 / num_thread; ++i) {
		p_lock(myid);
		sum += 2;
		p_unlock(myid);
	}

}

int main() {

	for (int num_thread = 1; num_thread <= 2; num_thread *= 2)
	{
		sum = 0;

		vector<thread> threads;
		auto start_time = high_resolution_clock::now();

		for (int i = 0; i < num_thread; ++i)
			threads.emplace_back(do_work2, num_thread, i);

		for (auto &th : threads)
			th.join();

		auto end_time = high_resolution_clock::now();
		threads.clear();
		auto exec_time = end_time - start_time;

		int exec_ms = duration_cast<milliseconds>(exec_time).count();
		cout << "Threads[ " << num_thread << "] , sum= " << sum;
		cout << ", Exec_time =" << exec_ms << " msecs\n";
	}
```

atomic_thread_fence(memory_order_seq_cst); 사용O

atomic_thread_fence(memory_order_seq_cst); 사용X, atomic 사용 O

3.1초가 걸렸다. atomic으로 선언하면 flag 읽을 때 쓸 때마다 계속 스톨이 일어난다. 원래 atomic_thread_fence_(memory_order_seq_cst) 했을 때는 루프 때마다 한 번씩 스톨이 생겼는데, 지금은 flag에서 한 번, victim에서 한 번, while 안에 flag에서 또 한 번, victim에서 한번 해서 총 네 번의 스톨로 늘어났다.

그래서 아무 생각 없이 atomic 남발하는 게 더 느려지게 된다. 꼭 필요한 곳에만 써야 한다.

그래서 이제 atomic을 사용하면 컴파일러랑 cpu의 문제는 해결이 된다. volatile 같은 거 쓸 필요가 없다. 알아서 시켜주니까. 그리고 읽고 쓰는 순서 고민하지 않아도 된다. 그러나, 캐시 문제는 해결해주지 않는다. atomic 한 int가 캐시라인에 걸쳐져 있다? 그런 상황을 조심해야 한다. atomic 한 int에 포인터를 했을 때 이런 일이 막 생긴다. 계속 조심해야 하는 문제이다.

그럼 이제 컴파일러랑 cpu 문제는 atomic으로 해결이 된다. 그럼 다 끝난 건가?

Atomic Memory만 있으면 되는가?

- NO
- 실제 상용 프로그램을 int, long, float 같은 기본 data type만으로 작성할 수 있는가?
- 실제 프로그램은 기본 data type을 사용하여 다양한 자료구조를 구축하여 사용한다
    - queue, stack, binary tree, vector

atomic <int> sum 이렇게 했을 때 sum += 2 이거는 atomic 해서 lock 같은 거 걸 필요가 없는데 sum = sum + 2 이거는 atomic하지 않아서 이상한 값이 나온다. 이런 식으로 하면 atomic이 아니라 이거 이전의 데이터 레이스 문제를 피할 수 없다. 중요한 문제는 전역 변수에 읽고 쓰는 것.

전역 변수가 이것만 있냐? 아니다. 온갖 해괴한 자료구조가 전역 변수로 선언이 되어있다. 게임 서버라면 뭐가 전역으로 선언되어 있느냐? Class Player; map <int, Player> 이런 거. 수천 명의 플레이어가 들어가 있고 이 자료구조는 복잡한 자료구조이다. 모든 정보가 멤버로 들어가 있다. 이거의 맵은 id가 숫자로 있고, 무슨 플레이어다 하고 맵 자료구조로 전역 변수로 선언이 되어있다. 왜냐하면 플레이어 정보는 모든 스레드가 access 할 수 있어야 한다. 어떤 스레드는 access 못한다? 그럼 게임 서버가 돌아가지 않는다. 몇천 개의 패킷이 들어오는데 어떤 스레드에서 실행이 될지 모른다. Map 이게 공유 메모리이고 공유 변수이다. 이걸 읽고 쓸 때 mutex로 락을 걸어야 한다. 안 그러면 충돌을 일으키며 죽어버린다.

근데 그러면 느려지기 때문에 우리가 어떻게 해야 하는가? atomic 배웠으니까 atomic <map> 하면 atomic 하게 읽고 쓸 수 있겠네? 개념상으론 가능한데 실제 프로그래밍하면 컴파일이 안된다. 비주얼 스튜디오의 명령으로는 map을 아토믹하게 구현할 수 없다면서 뱉어낸다. 실제 프로그램은 기본 데이터 타입을 사용해서 다양한 자료구조를 만들고 벡터, 클래스, 만들어서 공유하기 때문에 이 **아토믹 키워드에만 의존할 수 없다. 왜? 복잡한 자료구조는 컴파일러가 atomic으로 못한다. 그래서 다른 방법을 사용해야 한다.**

- 문제
    - Atomic Memory를 사용하여 만든 자료구조는 atomic 한가?
        - no
    - **효율적인 atomic 자료구조가 필요하다**
        - 일반 자료구조에 lock을 추가하면
            - 너무 느리다
        - stl은?
            - crach
        - stl + lock 은?
            - 느려서 못쓴다
    - "효율적인" 이라니?????

클래스를 atomic 한 자료구조만 모아서 만들었다고 해서 그 자료구조가 atomic 하게 되는 게 아니다. atomic 한 amp 하고 atomic 한 queue. 이런 atomic 자료구조가 필요하다. 근데 이거 만드는 건 그렇게 어렵지 않다.

잘 돌아가는데 lock 걸면 너무 느리고..

atomic이니 뭐니 다 c++11에 구현되어 있으면 c++11에 벡터나 맵을 쓰면 아토믹하게 돌아가지 않을까? 절대 그렇지 않다. c++11에 있는 stl 컨테이너들은 멀티스레드를 생각하고 만든 게 아니라 멀티스레드에서 쓰는 순간 죽어버린다. 게임 서버 프로그래밍 시간에 실습하면 알 거다.

stl이랑 mutex랑 쓰면 되지 않냐? 너무 느리다.

그래서 그냥 원래는 atomic 한 자료구조가 필요하면 mutex를 쓰면 된다. 근데 그 앞에 효율적인 이라는 전제조건이 붙는다면..? 그래서 나온 게 non blocking 프로그래밍이다. 이런 문제에서 우리를 해방시키고 성능 문제에서도 해방시키는 새로운 기법이다. 나머지 시간 동안 계속 배울 기법이다.

- 효율적인 구현
  - Lock 없는 구현
    - 성능 저하의 주범이므로 당연
      - Overhead
      - CriticalSection(상호 배제, 병렬성 X)
      - Priority inversion
      - Convoying
  - Lock이 없다고 성능 저하가 없는가??
    - 상대방 스레드에서 어떤 일을 해주기를 기다리는 한 동시 실행으로 인 한성능 개선을 얻기 힘들다.
      - while (other_thread.flag == true);
      - Lock과 동일한 성능 저하
  - 상대방 스레드의 행동에 의존적이지않는 구현 방식이 필요하다.

고성능 효율적인 멀티스레드 프로그래밍은 뭐냐? 일단 lock이 없어야 한다. 최대한 lock이 없어야 한다. 근데 이게 굉장히 힘들고 아무 생각 없이 없앴다가는 컴파일러, cpu, 캐시의 함정에 빠진다. lock이 문제가 되는 이유는 싱글 스레드에서의 오버헤드 때문에 성능이 저하된다. lock을 요청했을 때 바로바로 오면 상관이 없는데 그렇지 않다. 또 lock을 사용하면 상호 배제가 되면서 동시에 실행되지가 않는다. 그래서 성능이 떨어진다. 우선순위 역전이나 그런 일이 생긴다. 이건 나중에 배울 거고.

이런 문제를 일으키니 락을 사용하면 안 된다. 그럼 어떡해야 하는가

mutex를 사용하지 않으니 fence나 volatile을 잘 써야 한다. 근본적인 질문으로 돌아가서, **락이 없다고 해서 성능 저하가 없는가? 아니다.** 왜? 아까 우리가 올린 피터슨 알고리즘은 뮤 텍스를 안 썼었다. 그냥 공유 메모리 읽고 쓰고 하면서 volatile 이런 거 주의해서 쓰고 했다. 근데 mutex를 쓴 것보다 빠른가? 아니었다. 지난주에 돌려봤다. 피터슨 알고리즘으로 mutex를 돌렸을 때 3초가 아니라 1초가 나왔었다.

mutex를 쓰면 느려진다고 해서 안 쓰고 volatile를 썼었는데 mutex보다 더 느려졌다. **그러면 mutex를 쓰는 게 문제가 아니라, mutex가 갖고 있는 성능을 저하시키는 성질이 있고, 피터슨도 그걸 피할 수 없어서 성능이 떨어지는 것이다.**

많은 사람들이 연구를 했다. 어떤 특성이 있어서 성능이 떨어지고 mutex를 안 썼는데도 왜 느리냐

결론에 도달했다. mutex가 없다고 해서 성능 저하가 없는 게 아니라 프로그램 자체의 문제다. mutex의 문제가 아니라. 프로그램이 실행하다가 옆의 스레드에서 뭘 해줄 때까지 기다리는 프로그램이 있다고 하면 위의 단점이 그대로 살아나기 때문에 성능개선을 얻기 힘들다. 그러니까 내가 실행하다가 다른 스레드가 뭘 해줄 때까지 기다리는 거 옆의 스레드가 뭘 하기를 어떤 상태로 가기를 기다리는 코드가 있으면 그게 성능 저하의 주범이다. **lock은 모두 내부에 루프가 있다.** lock이 있으면 루프를 도는 코드가 당연히 포함되어 있기 때문에 성능 저하의 주범이 되는 것이다. lock을 안 쓰더라도, 이와 비슷한 코드가 있으면 마찬가지로 성능이 떨어진다. **옆의 스레드가 뭘 해주길, 옆의 스레드가 어떤 상태로 바뀌길 기다리는 그런 옆 스레드에 의존적인 프로그래밍을 하면 성능을 저하시키는 원인이다** 라는 결론을 얻었다.

- **Blocking**
  - 다른 스레드의 진행상태에 따라 진행이 막힐 수 있음.
    - 예) while(lock!= 0)
  - 멀티스레드의 bottle neck이 생긴다.
  - lock을 사용하면 블럭킹
- **Non - Blocking**
  - 다른 스레드가 어떠한 삽질을 하고 있던 상관없이 진행
    - 예) 공유 메모리 읽기/쓰기, Atomic (Interlocked) operation

이렇게 옆 스레드가 한 개든 두 개든 어떤 일을 해줄 때까지 기다리는, **다른 스레드 진행상태에 따라서 진행이 막힐 수 있는 프로그램을 블로킹 프로그램이라고 한다. 이게 바틀 넥의 주범이다.**

우리는 이런 식으로 다른 스레드가 뭘 해줄 때까지 기다리는 프로그래밍을 하면 안 된다. **락을 사용하면 무조건 블로킹이다.**

다른 스레드가 빠져나오지 않고 계속 꾸물거리면 계속 멈춰있기 때문이다. 더 이상 진행할 수 없다.

그러니까 mutex를 사용하면 무조건 블로킹이고, 이**걸 안 쓴다 하더라도 이런 식으로 짜여있다면 마찬가지로 블로킹 프로그램이다.** 동시에 실행되지 않고 루프를 돌아가는 거 오버헤드가 큰데 계속 변수를 읽어야 하고. 이런 코드가 없는 프로그래밍을 해야 한다. mutex는 당연히 쓰면 안 되고. 이것만 안 쓰면 되는 게 아니라 이것도 안 쓰고 while 이런 코드가 없는 걸 해야 한다. **옆 스레드가 뭘 하건, 프로그램 중에 어딜 실행하고 있건, 계산 값이 뭐가 나왔건 상관없이 돌아갈 수 있는 프로그램. 이런 걸 논블로킹 프로그래밍이다.**

다른 스레드가 어떤 삽질을 하든지 상관없이. flag를 true로 하건 false로 하건 신경 쓰지 않고 그냥 수행하고, 자기 일을 할 수 있는 알고리즘이 논블로킹 알고리즘이다.

우리가 본 적 있다. 뭐가 논블로킹 알고리즘이냐. atomic <int> a; 하고 a.store(3). 이러면 3이 a에 아토믹하게 들어가는데 들어가는 이 store이라는 메서드는 논블로킹이다. 옆 스레드가 a에 뭘 쓰고 있건 읽고 있건 상관없이 3을 집어넣는다. 옆 스레드가 lock을 걸고 있는 거 아무 상관없다. 무슨 오퍼레이션을 하든지 3을 넣고 끝난다. 루프도 없다. 공유 메모리 읽고 쓰는 거. sum += 2 이것도 논블로킹이다. 옆 스레드에 sum에 무슨 짓을 하던 2를 더하고 끝난다. 기다리지 않는다.

아니 뭐 당연한 건데 이런 걸로 논블로킹이라고 생색을 내냐 싶을 수 있다. 복잡한 예도 있지만 아직 안 배워서 그렇다.

블로킹 알고리즘의 문제

- 성능 저하
- Priority Inversion
  - lock을 공유하는 덜 중요한 작업들이 중요한 작업의 실행을 막는 현상
  - Reader / Write Problem에서 많이 발생
- Convoying
  - Lock을 얻는 스레드가 스케줄링에서 제외된 경우, lock을 기다리는 모든 스레드가 공회전
  - core 보다 많은 수의 thread를 생성했을 경우 자주 발생.
- 성능이 낮아도 non - blocking이 필요할 수 있다

1. 성능 저하 - 다른 스레드의 작업을 기다려야 하기 때문에 오버헤드가 있고, 동시에 실행되지 않아서 병렬성이 떨어진다.

2. 우선순위 역전 - 한번 lock을 얻으면 unlock을 할 때까지 다른 애가 끼어들 수 없다. 당연한 건데 이게 큰 문제다. 우리가 어떤 자료구조에 대해 메서드를 막 호출하면, 예를 들어 플레이어를 더하거나 빼거나 하면, 그럴 때 락 언락을 걸어야 한다. 근데 뭐가 중요한가? 여기에 add 할 수 있고 map에 erase, 아니면 그냥 read 할 수 있다. 여러 오퍼레이션이 있는데 **우선순위가 다 똑같은 게 아니다.** 새로운 플레이어 추가되면 add 해야 하고 그건 id를 갖고 플레이어를 읽어내는 read 보다 훨씬 중요한 거다. 왜? 새로운 플레이어가 접속을 하면 추가해야 하고 안 그러면 그 플레이어는 아무런 동작도 할 수 없다.

근데 erase도 우선순위가 높다. 플레이어가 게임을 떠났는데 아이템 거래나 몬스터가 공격하는 게 되면 안 된다. 근데 read가 락을 걸고 있는 동안은 add나 erase가 안된다. **더 우선순위가 높은 중요한 작업이 빨리 실행되어야 하는데 덜 중요한 게 lock을 얻고 있으면 중요한 작업이 계속 delay 되는 경우가 있다.** lock을 얻고 시간이 계속 흐른다? 그럼 큰일이 난다. 모든 메서드가 다 똑같은 우선순위다 그럼 상관없는데, 조건이 붙는 순간 **락으로는 이걸 구현할 수가 없다.**

3. 호위 (guard) - 이건 lock 한 다음에 sum += 2 하고 unlock을 한다. 문제는? 운영체제는 시분할 운영체제이다. 또 작업 관리자 보면 알겠지만 프로세스가 많다. 그래서 시분할로 콘텍스트 스위치 하면서 돌아가면서 실행되고 있는데 쭉 실행하다가 락을 얻었다? 그럼 sum에 더하고 unlock 하려고 하는데...! 바로 그 직전에 콘텍스트 스위치가 일어난다??? 그럼 큰일 난 거다. lock을 얻은 채로 콘텍스트 스위치 되었으니 0.01초 동안 unlock이 안 되는 것이다.

옆 스레드가 lock 하고 sum += 2 해야 하는데 못하고 있다. 1/100초 동안 한 놈만 멍하니 있는 게 아니라 **모든 스레드가 멍하니 있는다.** 싱글 스레드는 1/100초 손해지만, 듀얼이면 2/100, 쿼드면 4/100 **코어가 많으면 많을수록 점점 손해가 커진다. 이게 콘 보잉(convoying) 현상이다. 락을 얻었는데 콘텍스트 스위치.** 운영체제를 호출해서 lock을 거는 게 아니었기 때문에 운영체제가 콘텍스트 스위치를 할 때 애가 락을 얻은 앤가 아닌가 알 수 없다. 이 현상을 피할 수 없고 이래서 이런 이야기를 하는 것이다.

멀티스레드 프로그래밍할 때 스레드가 많으면 많을수록 성능이 빨라지겠거니. 빨라지지 않는다 하더라도 손해가 있겠냐? 절대 아니다. 쿼드코어 cpu여서 스레드를 4개 만들면 각각 cpu마다 스레드가 하나씩 돌아간다. 그리고 콘 보잉이 없다. 왜? 계속 한 놈씩 맡아서 실행하니까. 락을 얻은 채로 콘텍스트 스위치 될 일도 없다. 만일 콘텍스트 스위치가 일어난다 하더라도 서로 자리 바꾸는, 잠깐 운영체제 들어갔다 나오는 정도로 끝나지 1/100초 이렇게 걸리지 않는다. 3 GHz다? 그럼 이동안 90억 개의 명령어가 실행될 수 있다. 1/100초 엄청난 시간이다. 코어의 개수보다 스레드 개수가 적으면 콘 보잉 현상이 일어나지 않는다. 다른 컴퓨터에서 cpu 쓰는 다른 프로그램 같이 돌리지 않을 것이다. 그런데 욕심부려서 스레드 5 6 7 8개를 만들었다 하면? 운영체제가 실행하다가 1/100초 다 됐네 하면서 다른 애 실행할 거 없나 보고 다른 스레드로 바꾼다. 스레드 0 실행하다 스레드 4 하다가 스레드 0 하다가.. 이렇게 콘텍스트 스위치가 일어난다. 그럼 락을 갖고 1/100초 동안 자게 된다. 그래서 스레드 만들면 만들수록 성능이 떨어진다. **그러니 코어의 개수보다 스레드의 개수가 많으면 안 된다. 성능 향상이 없다. 운영체제의 오버헤드도 간다. 그러나 가장 큰 이유는, 락을 얻고 콘텍스트 스위치 하면 모든 스레드가 쉬고 있는 불상사가 생길 수 있다.**

논블로킹의 등급

- 무대기(wait-free)
  - 모든 메서드가 정해진 유한한 단계에 실행을 끝마침
  - 멈춤 없는 프로그램 실행
- 무 잠금(lock-free)
  - 항상 적어도 한 개의 메서드가 유한한 단계에 실행을 끝마침
  - 무대 기이면 무 잠금이다
  - 기아(starvation)를 유발하기도 한다
  - 성능을 위해 무대기 대신 무 잠금을 선택하기도 한다

자료구조 시간에 알고리즘 등급을 배웠었다. 알고리즘은 그럼 문제를 푸는 방법. 문제를 풀면 됐지 알고리즘 사이에 무슨 등급이 존재하느냐? O(0), o(log n) o(n) o(nlog n) o(n^2) 이게 등급이다.

논블로킹에도 있다. 무대기 알고리즘. Wait free이게 맵이다? 그럼 멀티스레드에서 insert 하고 erase 하고 하면 아무 상관없이 아토믹하게 움직일 뿐만 아니라 모든 메서드가 정해진 시간에 끝난다. 스레드가 몇 개든 상관없이 다른 스레드가 같이 읽던 말던 정해진 시간에 모든 메서드가 끝난다. 병렬성이 굉장히 높다. 정말 멈추지 않는다. 기다리는 게 전혀 없다. 멀티스레드 알고리즘 중에 논블로킹 알고리즘이 좋은 거고 그중 여러 등급이 있는데 그중 wait free 알고리즘이 제일 좋다.

그다음에 나오는게 무 잠금 **lock free알고리즘.** 위에 건 생소한데 이건 좀 들어봤을 수 있다. 이건 뭐냐? **Wait free보단 좀 덜 좋다.**여러 스레드가 동시에 어떤 동일한 자료구조에 대해 메서드를실행했을 때 **wait free는 정해진 시간에 끝나는데 lock free는 그러지 않는다.** 다른 스레드가 같은 데이터를 돌린다 그럼 충돌하면서 딜레이가 생길 수 있다.

근데 블로킹 알고리즘하고 그럼 뭐가 다르냐? 두 스레드중 하나가 대기하는거면. 딜레이가 생기는거면 뭐가 다르냐

이게 다르다. **딜레이가 있을 수 있는데 정해진 시간 안에 실행을 끝마치는 메소드 또느 스레드가 적어도 1개는 반드시 존재한다. 이게 다르다.** 이게 무슨 얘기냐.

프로그램을 간단하게 이야기하면 **블로킹 알고리즘**이다 하면 a, b 스레드가 실행하다가 같은자료구조에 access해서 충돌이 일어났다? 그럼 b가 쭉 실행하는동안 a가 실행을 못한다. 근데 그러다가 b가 unlock을 하면 그때서야 실행이 된다. B가 실행하다가 **콘텍스트 스위치가 되었다?** 그럼 a는 **계속 실행못하고 한도끝도없이 기다려야함**. 이게 블로킹이었는데,

논블로킹 lock free는 상한선이 존재한다. 충돌했다? 하면 b가 실행하고 a는 잠시 기다림. 그럴 순 있는데 끝까지 기다리는게 아니라 , **limit 유한한 단계까지만 기다리고** 그 시간이 지나면 b를 신경쓰지 않고 자기 일을 한다. 콘보잉 그런거 없다. 논블로킹이다. **기다리지 않고, 잠시 멈추긴 하지만 b가 unlock을 하길 기다리지 않고 자기 할일을 한다.** 이게 lock free 알고리즘이다. 적어도 한 개 , 이건 최소한의 조건이다. 3개가 부딪혀도 상관없다.

그래서 wait free 면 lock free이다. 무대기는 무잠금의 부분집합이다.

적어도 한 개는 실행이 되는데 문제는 한 개만 실행이 되어도 lock free이다. C는 아니고 a랑 b만 번갈아가면서 실행해도 락프라. **C는 쫄쫄 굶고있어도 lock free.** 멈추지 않고 유한한 단계에 한 개의 메소드는 돌고있으니까. **근데 이게 일어날 확률이 굉장히 작아서 보통 신경을 안쓴다.**

프로그램할 때 주료 힙소트가 퀵소트가 더 좋으나 힙소트를 안쓰고 퀵소트를 쓴다…. 왜? 퀵소트가 성능이 좋아서.. 워스트 케이스로 들어오면 힙소트가 당연히 좋지만 그냥 일반적으로 사용하는 데이터는 워스트케이스가 발생할 확률이 굉장히 낮기 때문에 퀵소트 사용이 더 좋다. (오버헤드가 적어서)

락프리보다 웨잇프리 만드는게 더 까다롭다. 오버헤드가 커진다? 그럼 락프리정도만 구현하고 넘어가는경우가 많다. **논블로킹 많은데 대부분 락프리이다.** 못만드는게 아니라 락프리가 더 좋으니까 그냥 쓰는것이다.

논블로킹의 등급

- 제한된 무대기 (bounded wait-free)
  - 유한하지만 쓰레드의 개수에 비례한 유한일 수 있다.
- 무간섭 (obstruction-free)
  - 한 개를 제외한 모든 쓰레드가 멈추었을 때, 멈추지 않은 쓰레드의 메소드가 유한한 단계에 종료할 때.
  - 충돌 중인 쓰레드를 잠깐 멈추게 할 필요가 있다.

락프리랑 논블로킹이랑 거의 같은 의미로 많이 사용한다 현업에선. wait free도 lock free니까 틀린 말이 아니다.

bounded wait free - 유한하지만 스레드의 개수에 비례한다. 100클락 안에 끝나거나 해야하는데, 스레드 개수에 따라 그 숫자가 왔다갔다 한다.

무간섭 - 스탑 에이션에 걸리기가 굉장히 쉽다. 그리고 동시에 실행시켰을 때 락프리는 정해진 시간 안에 반드시 하나는 실행을 해야하는데, 아무것도 실행이 안되고 계속 딜레이 되는 그런경우도 생길 수 있다. 이게 obstruction free이다. 아무도 실행을 끝마치지 못하면 그게 블로킹이지 무슨 논블로킹이냐.

만약 지금 데드락에 빠져서 딜레이가 되고 있다? 그럼 한 개를 제외한 나머지가 멈춘다. 그럼 한 개는 실행을 한다. 그리고 끝나고 다음에 뭘 하냐. 나머지 애들 중 또 한 개만 깨워서 하나하나 실행을 해서 데드락을 풀 수 있다. 이게 무간섭이다.

해괴하지만 이렇게 하는게 더 효율적인 알고리즘도 있다. 여기서 다룰 건 아니고 등등 있는데 별로 중요하진 않다. 실제로 쓰는 경우도 별로 없다. 현업에서 들을 일은 없을 거고 학부 레벨도 아니다.

정리

- Wait-free, Lock-free
  - Lock을 사용하지 않고
  - 다른 쓰레드가 어떠한 행동을 하기를 기다리는 것 없이
  - 자료구조의 접근을 Atomic하게 해주는 알고리즘의 등급
- **멀티 쓰레드 프로그램에서 쓰레드 사이의 효율적인 자료 교환과 협업을 위해서는 Non- Blocking 자료 구조가 필요하다.**

논블로킹 프로그램은 락의 단점을 없앤 프로그래밍 기법이다. 락을 사용하지 않아야 하고 안사용한다고 논블로킹 되는게 아니고, 다른 스레드가 어떤 행동을 하길 기다리지 않고 그냥 실행.

논블로킹 알고리즘은 atomic 한 알고리즘이다. 여러 스레드에서 동시에 실행했을 때 데이터레이스 없이 올바른 결과를 내주는 알고리즘. 그래서 멀티스레드 프로그램에서 스레드간의 효율적인 자료교환과 협업을 위해선 논블로킹 자료구조를 써야한다.

정말 얘네가 얼마나 빠르냐? 그건 하나하나 만들어서 성능을 고려해야 한다. 실제 컴퓨터의 메모리는 멀티코어에서 이상하게 동작하니까 아토믹하게 자료구조를 만들어서 짜야한다. **atomic 하게 하겠다 하면 mutex나 쓰면 되는데, 효율적이기까지 해야 하기 때문에 논블로킹 알고리즘을 사용한다.** 논블로킹에는 wait free, lock free가 있다.

Atomic한 자료구조가 필요하다 -> non-blocking알고리즘이 효율적이다.