

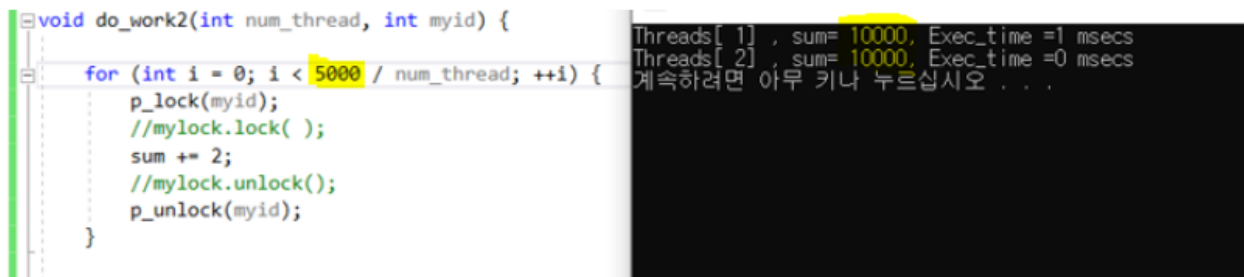
시간 시간까지 한 이야기

- 멀티스레드 프로그래밍의 필요성
- 멀티스레드 프로그래밍 API
- 멀티스레드 프로그래밍의 사용 예
- 멀티스레드 프로그래밍의 문제점
 - Data Race
 - 성능
 - 컴파일러
 - 피터슨 알고리즘의 bug

지난 시간에 1억이 제대로 안 나오는 문제에 대해서 이야기하고 있었다. 이것은 cpu의 문제다.

cpu는 정해진 순서대로 메모리를 읽고 쓰지 않는다. 인텔이 만든 cpu가 그렇게 하지 않기 때문이다. 왜? 기계어에서 메모리를 읽어라 라고 했을 때 메모리의 실행이 순간적으로 변하는 게 아니다. 명령어가 실행할 때 여러 클락이 걸린다. 메모리를 읽어라 했을 때 cpu에서 메모리를 읽어올 때까지 시간이 매우 오래 걸린다. 여러 클락이 걸리니까, 메모리 x를 읽고 y를 읽는다? 그럼 x를 먼저 읽고 y를 읽고 딱딱 끊어지는 게 아니다. x를 읽는 게 시간이 걸리고 y를 읽는데 시간이 걸리고. 근데 이게 겹친다. x를 먼저 읽었지만, y의 값이 먼저 도달할 수도 있다. 그래서 읽고 쓰는 순서가 바뀔 수 있다는 게 문제다. 그래서 이 알고리즘이 이런 문제가 있는 것이다.

1억에 가까운 값이 나오는 게 더 문제다. 스레드 2개를 합쳐서 5천만번 돌렸는데, 1억에 48 부족했으니 24번 틀린 것이다. 24/5천만 확률로 틀린 것. 엄청나게 약질적이다. 100번 10000번 돌리면 에러가 안 난다.



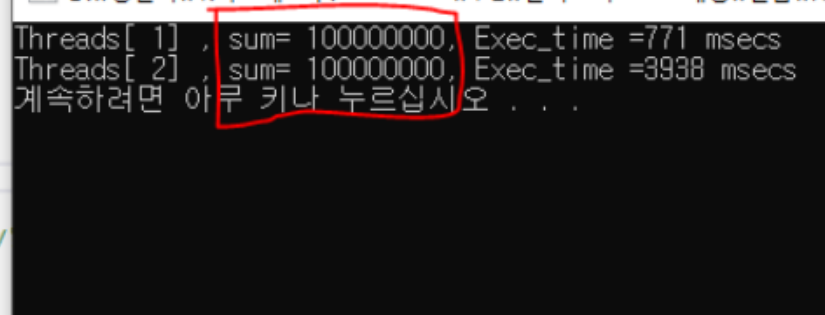
이렇게 5천만 번에서 5천 번으로 줄였을 때, 제대로 된 결과가 나온다. 동점 5천 명이 몰려서 하루 24시간 게임을 하면? 루프의 개수가 늘어나고, 그러면 틀린 값이 나오면서 죽는다. 그러니까 코드를 짤 때도 문제가 없고, QA팀에서도 문제가 없다 판단하고 통과했는데, 오픈 베타만 하면 죽는다. 소스코드도 문제가 없다. 그럼 어떻게 버그를 고치냐? 못 고친다. 게임을 접어야 한다. 두 시간마다 서버가 죽게 된다. 그런데도 왜 죽는지 모른다. 사람이 많이 몰려서 부하가 걸리면 바로 죽는다. 아주 악질적인 에러다.

여태까지 나타난 문제

1. 컴파일러 -> volatile 최적화 금지.

2. CPU (out of order) -> cpu도 실행 시 최적화를 한다. 메모리 읽고 쓰는 순서가 바뀐다. 메모리 내놔했을 때 시간이 걸린다. request를 여러 개 하는 경우 속도가 제멋대로라 순서가 바뀔 수 있다. 그래서 피터슨 알고리즘이 1억이 안 나오는 이유가 이것이다. 이것은 추측이고 정말 그런가 확인을 해봐야 한다. 컴파일러는 기계어로 보면 알 수 있었다. 그런데 cpu를 지금 우리가 까 볼 수 없다. 예전엔 뚜껑 까보고 했지만, 요즘은 뚜껑 까는 순간 동작을 안 한다. 그래서 어떡하느냐? 간접적으로 유추를 한다. 인텔이 만든 설계도가 있으면 그대로 돌리면 되는데, 기업 비밀이니까 알 수 없고. 최적화 방지 명령어를 넣어서 돌렸을 때, 제대로 동작하면 cpu최적화 때문에 벌어지는구나 알 수 있다. 그래서 문제는 메모리를 읽고 쓰고 하는데 쓰기도 전에 읽고, 읽기도 전에 쓰고, 이렇게 뒤바뀌니까 생기는 문제다 라는 가설. 그러니까 순서를 바꾸는 걸 강제로 막아보자. 이랬을 때 올바른 결과가 나오면 순서때문이었구나. 알 수 있다. 그리고 cpu의 out of order 파이프라인은 우리가 통제할 수 없고 하드웨어적으로 통제하려면 cpu의 명령어가 있어야 한다. 인텔은 명령어가 있다. 파이프라인이 여기서 멈춰서 위의 명령과 아래 명령이 섞이지 말라는 명령어가 있다. mfence. m = memory / Fence = 담장 / _asm 어셈블리 명령어를 프로그램 안에 강제로 집어주는 키워드. gcc는 __asm__.

```
//피터슨 락과 연락으로 구현하기
void p_lock(int myid)
{
    int other = 1 - myid;
    flag[myid] = true;
    __asm__ mfence
    victim = myid;
    while ((true == flag[other]) && victim == myid); //
}
```



```
Threads[ 1 ] , sum= 100000000, Exec_time =771 msecs
Threads[ 2 ] , sum= 100000000, Exec_time =3938 msecs
계속하려면 아무 키나 누르십시오...
```

_asm mfence 명령어를 넣어주니까 결괏값이 제대로 나온 것을 확인할 수 있다. cpu의 최적화 문제로 1억이 제대로 안 나왔던 것이다. 그래서 우리는 또 하나의 문제 해결을 했다. 이걸 충격적이다. cpu를 믿을 수 없다니. 왜 이런 일이 일어나는가? 먼저 첫 번째 문제는, c언어가 멀티스레드에서 사용하라고 만든 언어가 아니었고 그래서 멀티스레드가 오동작을 해도 고소해서 이길 수 없었다. 마찬가지로 두 번째, X86 CPU도 멀티코어용 CPU가 아니다. 나중에 성능을 도저히 올릴 수 없으니 최후의 발악으로 나온 게 멀티코어, 그래서 대비가 되어있지 않았다. 그러니까 이런 일이 벌어지고, 다행스럽게도 이런 명령이 있어서 극복 가능했다.

그럼 멀티스레드에 특화된 cpu는 없는가? 없다. 왜? 멀티스레드에 특화된 cpu를 만들 수 있지만, 싱글 스레드에서 성능이 많이 떨어진다. 싱글스레드에서 느리면 기존에 멀티스레드가 아닌 프로그램들에서 다 성능이 떨어진다. 싱글코어에서 느리면 사람들은 더 안쓴다. 증명이 되어있다. 옛날 AMD가 가격이 기존에 싼었다. 왜? 코어는 많은데 싱글코어의 성능이 인텔보다 성능이 떨어져서. 똑같은 가격에 코어가 더 많았다. 그래서 프로그래밍을 잘 하면 더 빠르데도 불구하고 아무도 안샀다. 그래서 지금의 CPU는 멀티스레드에 특화된 CPU가 아니고 싱글코어, 싱글스레드에서 잘 돌아가는 최적화가 되어있다.

비주얼 스튜디오에서는 이렇게 쓰면 되는데, 리눅스에서 프로그래밍을 한다? 그럼 gcc를 쓰니까

`__asm__ __volatile__ ("mfence" ::: "memory");` 이런 형식으로 써야 한다. cpu가 똑같아서 mfence라는 기계어는 똑같다. 그런데 cpu가 다르면? 모바일 게임에서 만들면? 거긴 비주얼 스튜디오도 없고 윈도우로 만들어진 핸드폰이 없으니까 아무도 안 쓰는데, 요즘엔 ms가 안드로이드 컴파일도 지원하긴 해서 `_asm` 해서 프로그래밍할 수는 있다. 근데 문제는 cpu가 다르니 mfence명령어가 없다. 어셈블리 명령어가 완전히 다르다. 이건 dmb가 명령어이다. data memory barrier.

이렇게 컴파일러마다 다 다르게 집어넣고 cpu마다 다 다르게 써야 하고. 프로그램이 지정해주지 않느냐? 혼란을 정리해주려 나온 게 c++11이다. 이거 쓰면 이런 지저분한 거 하지 않고 모든 운영체제와 모든 컴파일러와 모든 cpu에서 다 똑같이 동작하는 명령어를 제공한다. 그게 이거다.

```
#include <atomic>
atomic_thread_fence(std::memory_order_seq_cst);
```

atomic 이건 뭐냐? 멀티스레드로 실행하는데 이 앞뒤로는 메모리 읽고 쓰는 순서는 절대로 침범하지 말아라. 이 위에서 어떻게 제멋대로 하고, 아래서는 순서 바꾸는 거 상관없는데, 이 사이는 절대로 바꾸지 말아라. 이거 프로그래밍하고 컴파일하면 환경에 따라(리눅스, ARM) 기계어로 위에 언급한 두 줄이 나온다. 다른 명령어로 나올 수도 있지만, 어쨌든 같은 일을 하는 명령어가 나온다. 혼란을 잠재우려고 c++11에서 만들어줬다. memory_order 이건 뭐냐. 메모리 읽고 쓰는 순서를 지키고, seq 이건 Sequential 순서대로. cst 이건 constancy 일관성. 순서 일관성을 지켜라. 순서 일관성은 뒤에서 다시 이야기한다.

이 방식으로 하면 왜 느려지나? 이게 들어가면서 파이프라인이 스톱되고, 그러니까 당연히 느려진다. 이걸 쓰지 않으면 틀린 결과가 나오더라도 속도는 조금 더 빠르다. 성능 차이가 생각보다 크다. 2초정도 차이난다. 일부 cpu에서는 성능차이가 별로 안나는 cpu가 있다. cpu 구현에 따라 다른거고 여기서는 성능차이가 나는 것을 확인했다. 이 결과는 atomic_thread_fence를 써도 결과가 똑같다.

```
void p_lock(int myid)
{
    int other = 1 - myid;
    flag[myid] = true;
    atomic_thread_fence(memory_order_seq_cst);
    victim = myid;
    while ((true == flag[other]) && victim == myid);
}
```

```
Threads[ 1] , sum= 100000000, Exec_time =413 msecs
Threads[ 2] , sum= 100000000, Exec_time =3429 msecs
계속하려면 아무 키나 누르십시오 . . .
```

- 메모리 접근이 Atomic하지 않은 이유는?
 - CPU는 사기를 친다.
 - Line Based Cache Sharing
 - Out of order execution
 - write buffering
 - 사기를 치지 않으면 실행속도가 느려진다.
 - CPU는 프로그램을 순차적으로 실행하는 척만한다.
 - 싱글코어에서는 절대로 들키지 않는다.

내가 지금까지 4년동안 프로그래밍하는데 수많은 프로그램을 짜고 읽고 쓰고 하고 한 번도 메모리 읽고 쓰는 순서가 뒤박여서 엉터리였던적이 없다 이게 무슨애 기냐?

싱글 스레드 프로그래밍할 때는 메모리 읽고 쓰는 순서가 바뀌는 게 나타나질 않는다. Cpu는 순차적으로 실행하는 척만 하기 때문에. 싱글코어에서는 절대로 들 키지 않는다. 읽는게먼저다 그럼어떻게되나?

X =3 ; 다음에 if(x==3) 하면 쓰는게 반드시 실행되는데 위의 말대로 순서가 바뀐다면, 실행되지 않을때가 있어야 하는 거 아닌가? 아니다. 이게 들키지 않도록 감 시하는 하드웨어가 있다. X에 3을 썼다. 그럼 메모리에 들어가지 않고 cpu어딘가에 들어가서 실행하고 있다.x를 읽는다, 그럼 x메모리에 가서읽지않고 혹시 내가 x에 쓰고 있는 게 메모리에 아직 안들어가고헤매고 있는 게 어디있나 cpu를 탈탈탈 뒤진다. 그래서 아까 실행했는데 메모리에 안들어간게 파이프라인에 걸쳐있 거나 하면 거깃네 하고 거기서 읽는다. 메모리에서 읽는 게 아니다. X에 쓰는 게아직 끝나지 않은 게 어디에도 없다? 그럼 비로소 메모리에서 읽는다. 이런 하드웨 어가 있고 이런하드웨어를 internal forwarding 로직이라고한다. 컴퓨터 구조에서 배운다. 내앞에 쓰는거있으면 절대로 놓치지 않는다.. 원래는 메모리에 넣고 가져와야하는데, 메모리에 아직안넣었다? 그럼 직접간다. 그런 하드웨어가 있기 때문에 싱글스레드 프로그래밍에서는 절대로 읽고쓰는순서가 어긋나는일을 우 리가 발견할 수 없다. 어긋나는데 그걸 커버해주는, 프로그램에 영향을 끼치는 것을 막아주는 그런 하드웨어가 있어서. 문제는 이게 코어안에서만 동작한다. 검사 를 할 때 코어 안에 있는 것만 검사하지 옆 코어는 검사하지 않는다.사하지않는다. 그래서 outoforder로 막 실행하는데 옆에 메모리에서 갖고 와야지 그런 거 안 한다. 옆 코어까지 forwardingforwarding 해주지 않는다. 이 로직을 넣으면 cpu가 느려져서 어차피 구현도 못한다.

```
a = fsin(b);  
f = 3;
```

```
a = b;    // a,b는 cache miss  
c = d;    // c,d는 cache hit
```

A에 계산값이 들어가야하는데 한참시간이걸린다. 그럼 파이프라인에 실행이되고 f에 3을 넣는다, a에 sin값이 들어가는것보다 3이들어가는게 먼저다.

A에 b가들어가는게 먼저인데 먼저아니냐먼저 아니냐? a나 b 둘 중 하나에서캐시 미스가 난다? 그럼 메모리에서 읽어오느라고 시간이 몇십클럭 엄청 걸린다. 그 동안 cpu가 멍하게 있지 않는다. 놀면뭐하나 미리 해둘 거 없나. 밑의 문장이 실행한다. 둘다 캐시있네? 그럼 즉시 읽어서 즉시 실행한다. 그러면 a에 b값이 들어가기전에 c에 d값이 들어간다.

CPU의 발전

Fetch : 명령어 읽어오기
Decode : 명령어 해석
Execute : 명령어 실행
Memory : 메모리 입출력
WriteBack : 실행결과 쓰기

Instruction 1					Instruction 2		
Fetch	Decode	Execute	Memory	WriteBack	Fetch	Decode	WriteBack

Fetch	Decode	Execute	Memory	WriteBack				
	Fetch	Decode	Execute	Memory	WriteBack			
		Fetch	Decode	Execute	Memory	WriteBack		
			Fetch	Decode	Execute	Memory	WriteBack	
				Fetch	Decode	Execute	Memory	WriteBack

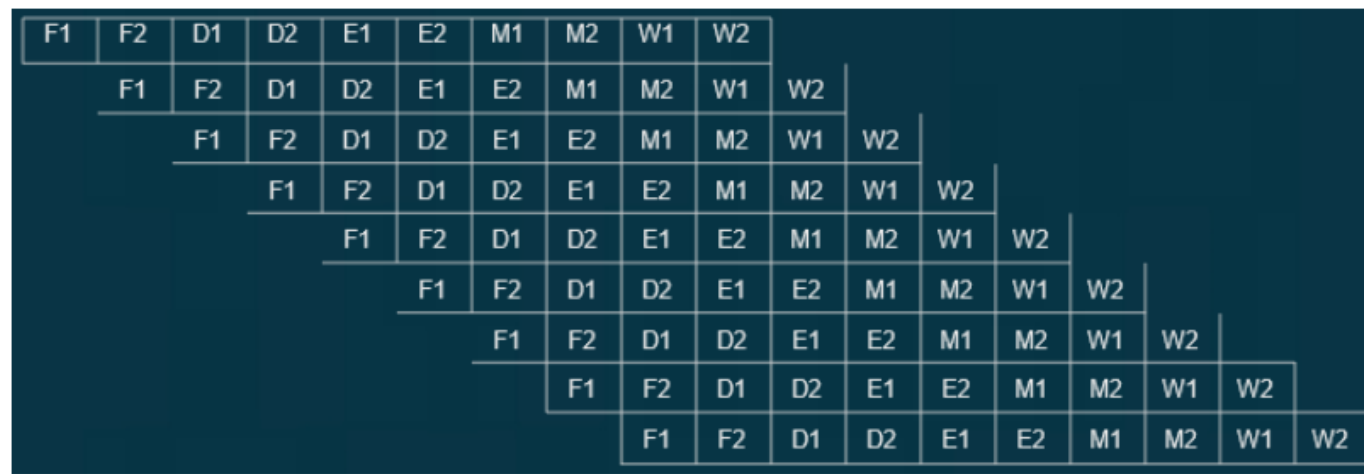
pipelining (RISC CPU)

80년대 후반이 되면서 기존 방식은 너무 느렸고 해서 파이프라인으로 돌아가기로 했다. 매 클럭 execute alu를 매 클럭 돌아가면서 쓴다.

Fetch	Decode	Execute	Memory	WriteBack				
Fetch	Decode	Execute	Memory	WriteBack				
	Fetch	Decode	Execute	Memory	WriteBack			
	Fetch	Decode	Execute	Memory	WriteBack			
		Fetch	Decode	Execute	Memory	WriteBack		
		Fetch	Decode	Execute	Memory	WriteBack		
			Fetch	Decode	Execute	Memory	WriteBack	
			Fetch	Decode	Execute	Memory	WriteBack	
				Fetch	Decode	Execute	Memory	WriteBack
				Fetch	Decode	Execute	Memory	WriteBack

Super scalar (Pentium)

명령어를 두 개 세 개 한꺼번에 동시에 패치해서 동시에 실행하자. 패치할 때 여러개 명령어를 패치하다 보니 메모리 인터페이스가 느려졌다. 한 클럭에 64비트를 패치한다. 64비트에는 명령어가 한 개가 아니라 여러개 일 수 있다. alu가 2개 3개 4개 들어있다. 동시에 실행할 수 있다. 펜티엄부터 이런 일이 벌어지고 너무 느리다 해서 나온게 밑의 슈퍼파이프라인. 여기서는 동일한 시간에도 두개씩밖에 패치를 못하고, 세개 하려면 alu가 3개여야 해서 하드가 너무 복잡하다. 그래서 이걸 잘게 잘게 쪼개자 한 개 슈퍼파이프라인이다.



superpipelining (pentium 4)

그리고 이걸 냈는데 펜티엄 4가 망하게 되었다. 파이프라인이 100단계가 넘는다. f1스테이지에서 f2스테이지로 넘어갈 때 열이 발생한다. 앞에 펜티엄 3보다 2배의 열이 난다. 전기 엄청 잡아먹고 해서 망하게 되었다.



out of order (Pentium Pro)

그래서 나온 게 이것이다. 파이프라인 풀로 채우고 실행하면 좋은데 못하더라. 왜? 명령어를 보니까 $ax=1$, $bx = ax + 3$ 이런 명령어 두 개를 동시에 패치했는데 동시에 실행할 수 없다. 대입 같은 건 기다릴 수밖에 없는 게 있다. 그럼 뒤로 실행이 밀리고 속도가 느려지기도 한다. 이 명령어가 끝날 때까지 기다렸다가 3을 더해야 하는 건 어쩔 수 없다. 근데 실행이 계속 밀려야 되냐. 실행해도 되는 녀석들은 실행하자. 그래서 $cx=3$ 같은 건 실행이 된다.

이러니까 순서가 뒤죽박죽 바뀐다. 메모리 읽고 쓰는 순서도 뒤죽박죽 바뀌면서 피터슨 알고리즘이 제대로 돌아가지 않는 일이 생기고, 이것 막기 위해서는 mfence를 써야 한다.



hyper thread (pentium 4)

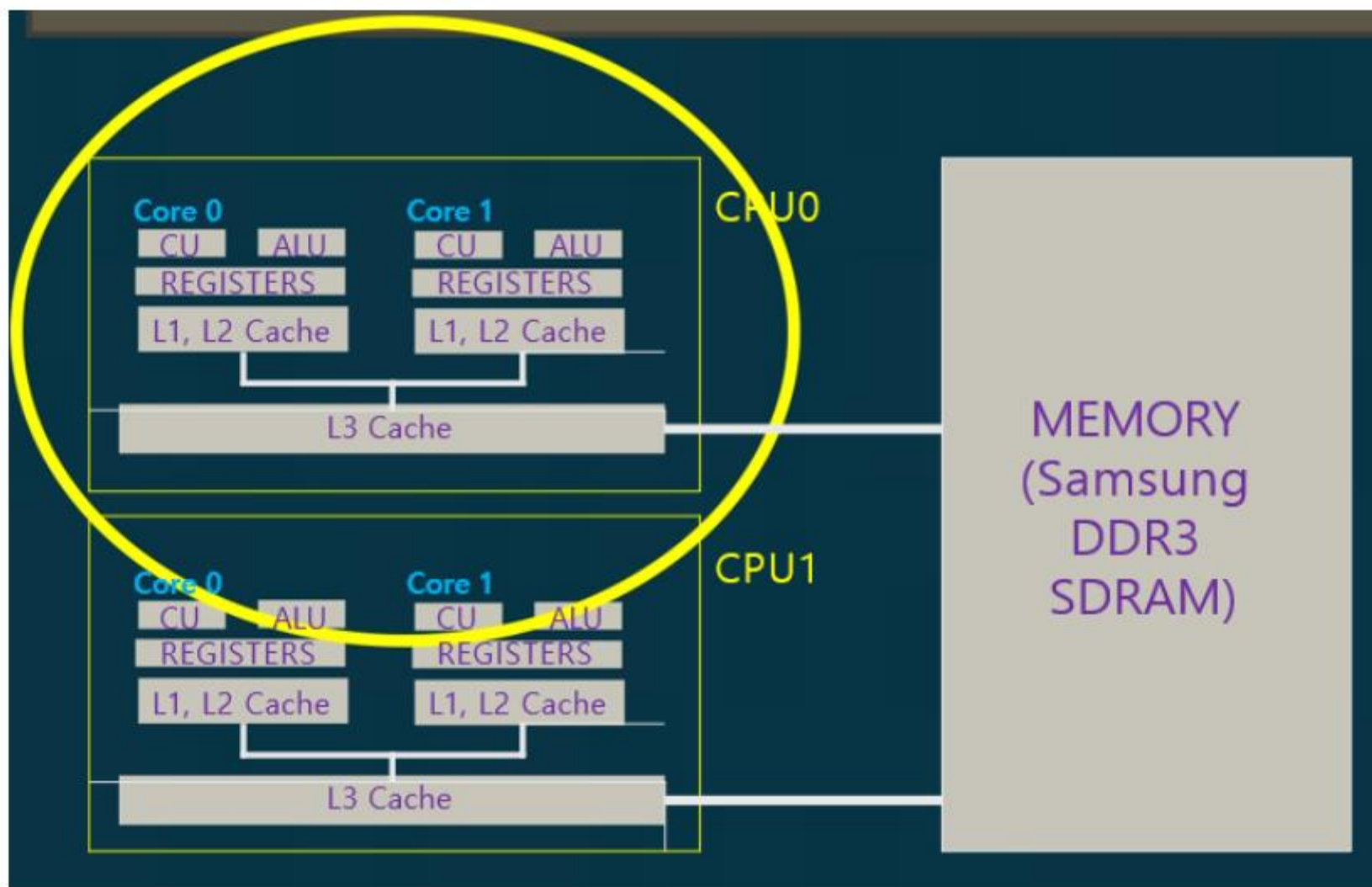
인텔이 i7에 하이퍼스레드가 있다 광고하는데 그건 이거다.

out of order를 해도 빈자리를 메꿀수없다. alu가 논다. 어떡해야 하나?? 아무리 out of order를 해서 빈자리를 메꿀수없다면 스레드 동시에 실행하자. 스레드 A와 스레드 b를 동시에 실행하고 그걸 합친다. 빈자리에 옆 스레드의 부분을 집어넣을 수 있다. 왜? 서로전혀 다른프로그램이라 메모리 dependency 상관없고 순서 상관없이 그냥 합칠수가 있다. 근데 그냥 합치면 성능이 안나오기때문에 fetch 디코드는 두 세트로 마련을 해놓았다. Alu는 하나인데는하나인데 디코드만 뺏기게해서 패치 디코드만 더많이할수있고 실행 execution은 원래 코어 역할 하나밖에안하고. 이게 하이퍼스레드다. 이것로 했다고 해서 코어개수가 2배 뺏기된 효과가 나오느냐 그렇지 않다. 프로그램 두 개를 실행시키면 실행시간이 두배로 걸리는데 하이퍼스레드는 두개 합한 시간만큼 걸리지는 않지만, 멀티코어 보다는 느리다. 근데 싱글코어에서 2개를 번갈아시키는데보단 빠르다. 없는것보단 빠르는데 멀티코어가 훨씬 좋다.

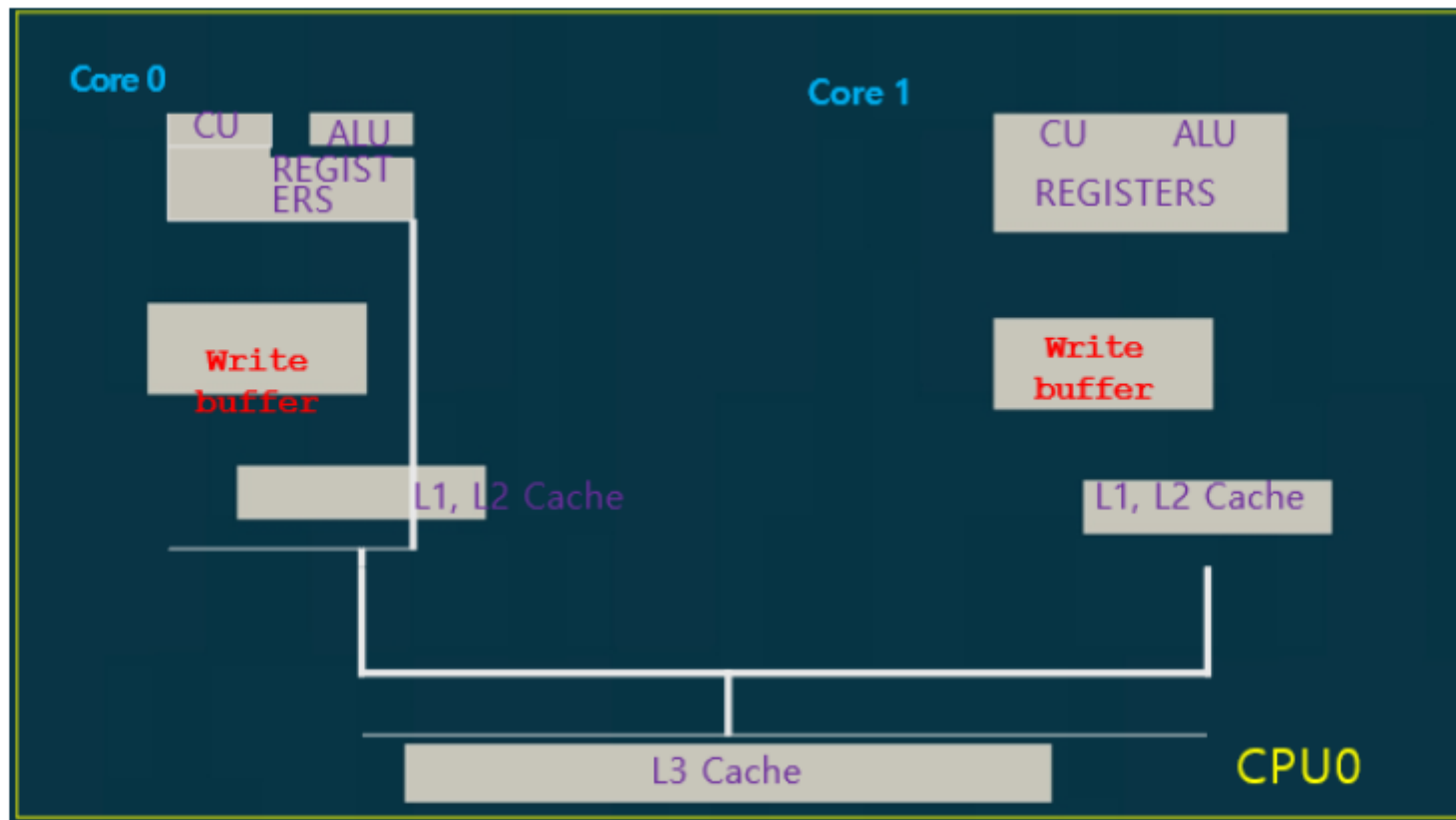


Multi core (pentium 4)

멀티코어를 했다면, 하이퍼스레드는 반까지 안된다. 빈공간 다 채워넣음 좋은데, 프로그램에 따라서 메모리안쓰고 cpu만 쓰는 게 있을 수 있다. 그럼 성능 향상이 없을 수 있다. 아님 빈자리가 너무 크거나. 그리고 메모리 많이 쓴다 그럼 메모리 유닛도 여러 개가 아니고 하나밖에 없기 때문에 거기서 또 병목이 생기고 하이퍼스레드 그렇게 좋지 않다. 스레드 4개돌리는거랑 스레드 8배돌리는게 22배 차이 나야 하는데 그 정도 아닌 걸 체험할 수 있다. 실습실에서.



멀티코어 cpu는 이렇게 생겼다. 코어를 좀 더 자세히 보자.



그럼 이렇다. write buffer는 왜 있냐. 메모리에 값을 쓰고 다음 명령을 실행해야 하는데 명령어 쓰는 실행시간이 굉장히 길다. 캐시에 넣고 땡 하면 끝나는데, 캐시 미스가 나고 그럼 메모리 캐시에서 올 때까지 기다려야 한다. 그냥 기다리고 있으면 파이프라인 스톱이니깐 cpu가 논다. 다음 명령어를 실행하자. 그러다 또 write buffer를 만난다면? 두 번째 만나면 그럼 그땐 기다리느냐? 아니다. 그래서 버퍼라는 걸 만들어서 write 몇 개씩 쌓아놓고 다음으로 넘어갈 수 있게 만들자. read 버퍼는 없다. 왜? 명령을 실행하다가 메모리를 읽어야 한다? 읽지 않고 다음으로 내려가는 거 안된다. 읽은 값을 가지고 계산을 하거나 해야 하는데, 뒤에 진행을 못한다. 메모리에 들어가지 않았다고 해서 밑의 명령을 실행할 수 없는 건 아니다. 그러니까 버퍼에 쌓아놓고, 계속 실행하고 메모리 버스에 여유를 두고 천천히 write를 하면 된다.

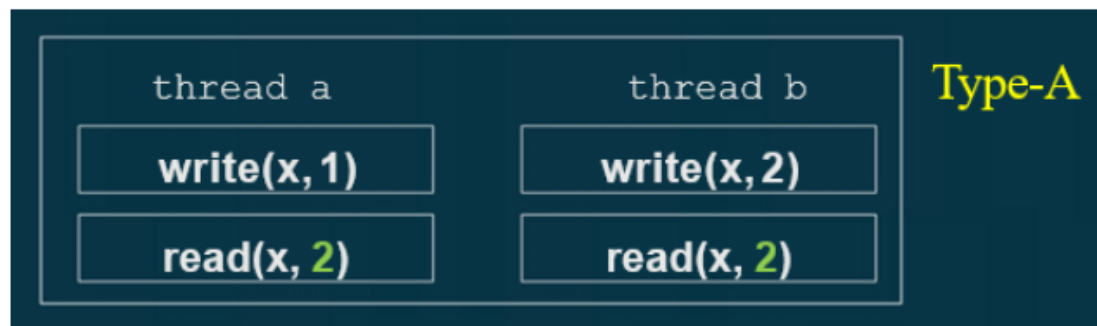
그래서 멀티코어에선 문제가 생긴다. 애가 flag에 true를 넣었는데, 가지 않고 쌓이고. 코어 1가 옆 스레드에 분명 썼는데 L3 캐시에서 가져올 때 버퍼에 있는 게 반영이 안 되고, 그럼 버퍼 때문에 write가 안돼서 순서가 딜레이 되고.

문제는 메모리

- 프로그램 순서대로 읽고 쓰지 않는다.
 - volatile로도 해결되지 않는다
 - volatile 키워드는 기계어로 번역되지 않는다.
 - 읽기와 쓰기는 시간이 많이 걸리므로
 - cpu의 입장에서 보면
 - 실제 영향이 발휘되는 시간은 상대 코어에 따라 다르다.
 - 옆의 프로세서에서 보면 어긋난 순서가 보인다. 또는 읽는 순서가 어긋난다.
 - 자기 자신은 절대 알지 못한다
 - 어떠한 일이 벌어지는가????

예전에 착각을 하는 사람들은 이렇게 말하는 사람도 있었다. cpu가 왜 순서가 뒤죽박죽인가! volatile 하면 cpu도 컴파일러처럼 최적화 안 하고 순서대로 실행하지 않느냐. 그런데 그렇지 않다. **volatile은 컴파일러에게 요청하는 것이다. cpu와는 아무런 관계도 없다.** 그렇기 때문에 **cpu는 제멋대로 실행을 하고**, 읽고 쓰기 시간이 많이 걸리고 언제 실행될지도 알 수가 없고, 또 write buffer 때문에 순서가 어긋나는데, 싱글코어에서는 다 검색해서 올바른 순서대로 실행시켜주지만 멀티코어에서는 그렇게 할 수 없다. 그렇기 때문에 그냥 순서 바뀐 게 그냥 전달이 된다. 그래서 틀린 결과가 나오게 된다.

예를 들면 어떤 결과가 나오는가??



여기서는 스레드 a가 읽으면 1이 나올 수도 있지만, 2가 나올 수도 있다. 이랬을 때 제대로 된 실행이나? 제대로 됐다. 1 쓰고 2를 덮으면 2가 읽힐 것이고, 2 쓰고 1을 덮으면 1이 읽힐 것이다.

Type-B

thread a

write(x, 1)

read(x, 1)

thread b

write(x, 2)

read(x, 1)

이게 그 상황이다. 2를 쓰고 1을 덮으면 1이 읽힐 것이다. 이것도 우리가 생각한 대로 맞게 실행이 되는 것이다.

그렇다면 이 밑의 상황들은 어떤가?

thread a

write (x, 1)

read(x, 2)

thread b

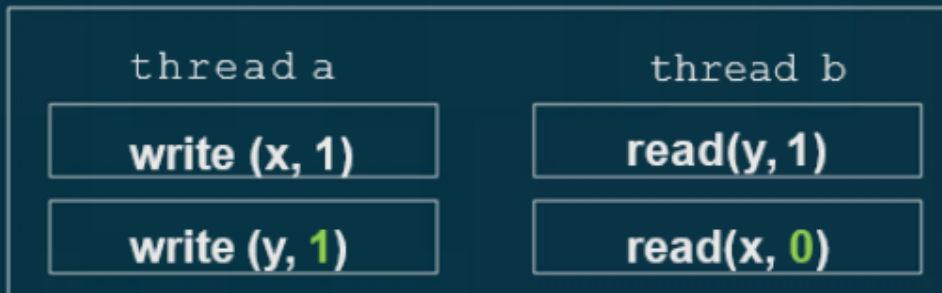
write(x, 2)

read(x, 1)

Type-C!!

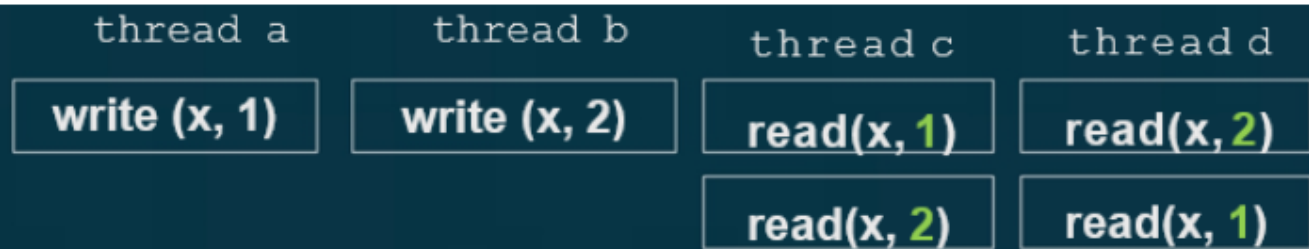
처음 스레드 a의 read가 2인걸 보면 그래, a가 처음에 1을 쓴걸 b가 2로 덮어썼구나. 그런데 스레드 b가 읽었더니 1이 나왔다? 이 얘기는 b가 2를 썼는데 a가 1을 덮었다는 이야기다. 그 얘기는 b다음에 a가 실행되었다는 건데, 스레드 a에서의 read가 2가 나왔다는 건 반대로 a다음에 b가 실행이 되었다는 이야기. **실행 순서가 서로 뒤바뀌어 보이는 것이다.** 결과가 다르게 나오는 두 개의 스레드. 이런 결과가 생기면 안 된다. 에이 이런 건 안 생기겠지? 이렇게 생각하고 프로그램이 하는데 이런 결과가 나온다. 왜? 읽고 쓰는 순서가 바뀔 수 있기 때문에.

Type-D !!



이 상황은 a스레드가 x에 1을 쓰고, y에 1을 쓰고. b스레드가 y를 읽으니까 1이 나왔다는 건? x에 이미 1이 들어가는 게 실행이 되고, y를 쓰는 작업이 수행되었다는 것. 그런데 스레드 b에서 x를 읽으니 0이 나왔다? 1이 아니라? 이런 실행은 제대로 실행을 했다면 있을 수 없다. 그런데 이런 일이 cpu에서는 벌어진다. **이게 피터슨 알고리즘이 제대로 동작하지 않는 상황이다.** flag에 true true를 넣었는데, 상대방이 읽을 땐 false false로 읽히고. 왜 이런 일이 벌어지냐? 둘 중 하나는 무조건 1이 나와야 하는데 둘 다 0이 나오고 이런 일은 있으면 안 된다. 그런데 **x에 1을 썼는데 write buffer에 들어가서 잠자고 있고, y에 1을 썼는데 write buffer에 들어가서 잠자고 있고. 읽었더니 0이나 오고. 또 0이 나오고.**

Type-F



또 스레드를 여러 개 수정했는데 x에다 1을 쓰고, x에다 2를 쓰고. x를 읽으니까 1이 나오고, 한번 더 읽으니까 2가 나오고. c까지 보면 이럴 수 있다. 순서가 1 다음에 2가 쓰인 것. 스레드가 a, b 순서로 작동한 것. 그런데 d 스레드는 실행 순서가 반대로 나온다. 2가 읽히고 1이 읽히고. 이런 일이 생기면 안 되는데 다 일어날 수 있는 일이다. 지금 cpu에서는.

그러면 프로그래밍할 때 어떻게 해야 하는가? 항상 올바른 결과가 나오는 걸 가정하고 프로그램이 하면 안 되고, 이런 식으로 순서가 뒤죽박죽 섞어놓는 괴상한 결과가 나올 수도 있다는 것을 가정하고 프로그래밍해야 한다.

왜? cpu가 write buffer를 바로 실행하지 않으니까. 그게 싫으면 mfence를 넣어야 하는데 그럼 느려진다.

- 현실

- 앞의 여러 형태의 결과는 전부 가능하다.
- 부정확해 보이는 결과가 나오는 이유?
 - 현재의 CPU는 Out-of-order 실행을 한다.
 - 메모리의 접근은 순간적이 아니다.
- 멀티 코어에서는 옆의 코어의 Out-of-order 실행이 관측된다.

나는 나의 프로그래밍 순서를 지키는데, 옆 코어에서 봤을 때는 아니다. 나는 순서를 엉터리로 해도 제대로 수습해서 보여주는데, 옆 코어는 수습되기 전의 엉터리 결과를 보는 것이다. 이런 일이 얼마나 자주 벌어지는지 살펴보자.

- 실습

- Atomic하지 않은 메모리 접근을 직접 확인해보자.
- 메모리 수정이 서로 다른 코어에서도 같은 순서로 관찰되는지 비교해 보자.
- 메모리를 수정하면서 다른 스레드에서 수정한 내용을 기록한다.
- 두 개의 스레드에서의 기록을 비교한다.

```

#include <thread>
#include <iostream>
#include <atomic>
using namespace std;

const int MAX = 50'000'000;

volatile int x, y;
volatile int trace_x[MAX], trace_y[MAX];

void thread_x()
{
    for (int i = 0; i < MAX; ++i)
    {
        x = i;
        trace_y[x] = y;
    }
}

void thread_y()
{
    for (int i = 0; i < MAX; ++i)
    {
        y = i;
        trace_x[y] = x;
    }
}

int main()
{
    int count = 0;

    thread t1{ thread_x };
    thread t2{ thread_y };

    t1.join();
    t2.join();

    //연속된 숫자가 있나 없나 검사
    for (int i = 0; i < (MAX - 1); ++i)
    {
        if (trace_x[i] != trace_x[i + 1]) continue;
        int x = trace_x[i];
        if (trace_y[x] != trace_y[x + 1]) continue;
        if (trace_y[x] == i) //이러면 두개의 스레드의 메모리 읽고쓰는 순서가 다른것.
            count++;
    }
    cout << "Number of Error = " << count << endl;
}

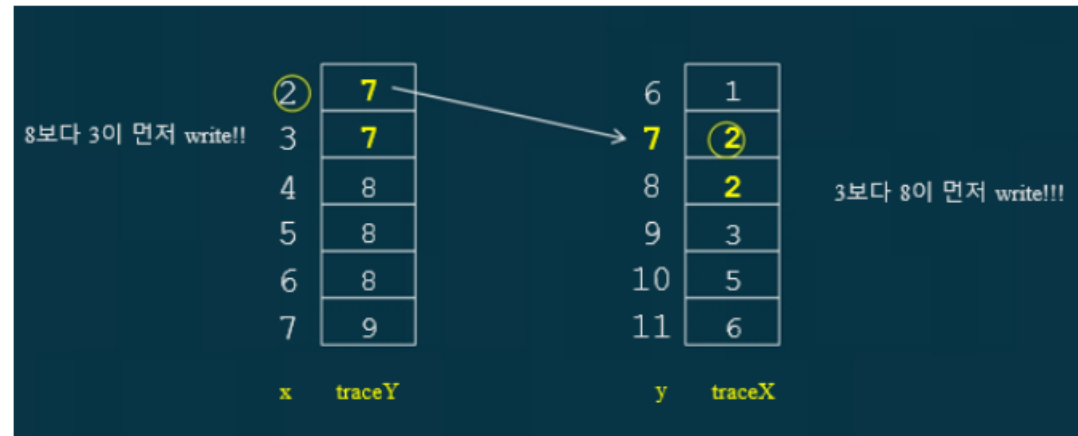
```

이 프로그램을 돌려보자. 이렇게 두 개의 스레드가 서로 x y에 저장을 하고, 그럼 cpu가 올바른 순서대로 메모리를 읽고 썼는지 히스토리를 가지고 판별을 할 수 가 있다. 이거 왜 이렇게 짰냐? 한다면, 메모리를 읽고 쓰는 순서와, 메모리 액세스 순서가 얼마나 어긋나는지 그것을 확인해보기 위한 것이다. 아까 피터슨 알고리즘으로 확인한 거 아니냐? 아니다. 그건 cpu가 순서를 바꿔 실행해 1억이 안 나온다는 것을 확인한 것이다. 그걸로는 언제 틀렸는지 알 수 없다. 여기는 뭐냐? 메모리를 읽고 쓰는데 언제 무슨 값이 잘못됐는지 다 증거를 남긴다. 증거를 얻어내는 프로그램이다. 그래서 이렇게 실행되면 어떻게 되는가?

trace_x는 0123456789 이렇게 들어가는데, 실행 속도가 둘이 같지 않다. 그래서 차이가 있어서 01233345567 이렇게 들어간다. 똑같은 값이 몇 개가 들어가기
도 한다. 스레드가 반대쪽이 느려지면 어떤 건 13556 이렇게 빼먹고 들어가고, 같은 값이 들어가고, 한다. 단! 절대로 일어나면 안 되는 일은 1234554 이렇게 역
전이 일어나면 안 된다. 이런 일은 cpu가 막아주기 때문에 역전은 일어나지 않을 것이다. 계속 증가하거나, 같은 값이 나오거나 한다. 우리 이걸 가지고 cpu에 메
모리 읽고 쓰는 순서가 뒤집혔구나 라는 걸 알 수 있다.

x의 값이 쪽 있는데 연속되는 x의 값이 서로 같은 경우, 같다고 해서 에러가 아니다. 같은데 x값을 인덱스로 하는 y를 트레이스하고 그 값이 같다 하면 그런 경우 어
떤 값이 저장되어있을 거고, 인덱스에서 y를 봤던 | 연속될 수 있는데 만약에 y에 저장된 값이 i와 같다면 일관성이 깨진 것이다. ++로 에러를 세서 몇 번 메모리
일관성이 깨졌는지 알 수 있다.

이게 무슨 소리냐.. 소스코드로는 모르겠다. 설명을 들어도 모른다.



2개의 core에서 서로 다른 순서로 메모리에 업데이트 되는 현장을 포착

이런 이야기다. trace_x, trace_y를 쪽 저장하고 같은 값이 반복될 수도, 중간에 빠질 수도 있다. 하지만 계속 증가는 한다. 이랬을 때 y의 흔적을 보면 778889 하
고 같은 값이 연속인 걸 볼 수 있다. 그러면 먼저 **x가 2일 때 trace_y가 7이 나왔다.** 그럼 x스레드에서 y 값을 봤는데 7이 읽혔다는 것이다. **x가 y를 봤을 때 y가 7이었다.**

그럼 y가 7일 때 과연 x가 2가 읽힐 것인가 하는 이야기. **y가 7일 때, 이때 x값을 읽은 게 몇이 나오냐? 2이다.** 서로 같은 값이 나오니까 문제가 없다.

여기까진 문제가 없다. 숫자가 다르다고 해서 문제가 생기는 것도 아니다 2를 썼는데 2가 아직 오지 않아서 1이 들어갔구나. 그럴 수 있다.

문제는... y가 7일 때도 x가 2, y가 8일 때도 x가 2라는 것이다. 반대로, x가 2일 때 y가 7, x가 3일 때 y가 7이라는 것이다. 읽고 쓰는 순서가 엇갈린 것이다. 왜???
x가 2일 때->y가 7이면, x가 3으로 바뀌어도 ->y는 7이라는 이야기다. 그러니까, **x가 3으로 바뀌는 게 y가 8로 바뀌는 것보다 먼저다. x가 3일 때도 y는 아직 7 이니까.**

그런데, y 쪽을 보면. y를 8로 바꾸었는데 -> x는 2이다. y는 8로 바꾸었는데 x가 아직도 안 바뀌고 있다는 것이다. 위에서는 x가 3으로 바뀌는 순서가 y가 8로 바뀌는 순서보다 빨랐다고 말했는데, **둘이 다른 순서로 메모리 업데이트가 되는 것이다.**

이렇게 프로그램을 돌려보면, 프로그램이 실행되다가 언제 어긋나고 무슨 값이 어긋났는지 확실히 알 수 있다.

뭐, 무슨 얘기인지 잘 모를 수 있다. 잘 생각해봐라.

그럼 이제 확실하게 테스트를 해볼 수 있다. 이 오류가 많이 나는 프로그램을 살펴보고, mfence로 오류를 제거할 수 있는지 까지 살펴보자.

```
Number of Error = 212251
계속하려면 아무 키나 누르십시오 . . .
```

위의 코드로 실행했을 때 212251번 메모리 일관성이 깨졌다.

그럼 아래 코드처럼, mfence를 추가해서 실행시켜보자.

```
void thread_x()
{
    for (int i = 0; i < MAX; ++i)
    {
        x = i;
        atomic_thread_fence(memory_order_seq_cst);
        trace_y[x] = y;
    }
}

void thread_y()
{
    for (int i = 0; i < MAX; ++i)
    {
        y = i;
        atomic_thread_fence(memory_order_seq_cst);
        trace_x[y] = x;
    }
}
```

```
Number of Error = 0
계속하려면 아무 키나 누르십시오 . . .
```

이렇게 mfence를 넣으면, 제대로 메모리를 읽고 쓰는 순서가 어긋나지 않고 지켜진다. 그러니까 0이 나온다. 그럼 둘 중 하나만 mfence를 넣으면?

```
Number of Error = 31893
계속하려면 아무 키나 누르십시오 . . .
```


둘 다 어긋나는 것보다 한쪽만이라도 cpu가 순서를 지키면 오류가 줄어들긴 한다. 근데 의미 없다. 제대로 돌아간다는 보장을 할 수 없다.

읽고 쓰는 순서가 다른 것 외에 또 다른 문제가 있다. 캐시라인 문제.

- 메모리에 유령이 있다?
 - 변수 값을 변경했을 경우, 변경 자체는 atomic 한가?
 - 모든 비트가 한순간에 변하는가?
- 실습
 - 아래의 프로그램을 실행하고 error의 개수를 출력하라
 - bound는 적당한 변수를 pointing 하도록 한다.

```
volatile bool done = false;
volatile int *bound;
int error;

void ThreadFunc1()
{
    for (int j = 0; j <= 25000000; ++j) *bound = -(1 + *bound); done = true;
}

void ThreadFunc2()
{
    while (!done)
    {
        int v = *bound;
        if ((v != 0) && (v != -1)) error ++;
    }
}
```

bound에 0이 들어가 있다. 바운드에 0과 -1을 바꿔가며 집어넣는 프로그램이다. 별 문제없다. 그냥 int로 하지 *로 했지만 별 상관없다. 그다음에 done 다했으면 true로 하고,

옆 스레드는 뭐냐? 옆에선 true가 아니면 계속 바운드를 읽는다. 그다음에 bound 값을 검사해서 0인가 -1인가. 둘 다 아니면 에러다. 둘 중 하나면 그럴 수 있지.

0과 -1 이외에는 나오면 안 되는 프로그램이다. 제대로 되나 실행해보자

```

#include <thread>
#include <iostream>
#include <atomic>
using namespace std;

const int MAX = 25'000'000;

volatile int x, y;
volatile int* b;
volatile bool done = false;
int error; //volatile로 할필요없다. 두개의 스레드에서 동시에쓰는게 아니라 한 스레드만 쓰는것.

void thread_1()
{
    for (int i = 0; i < MAX; ++i)
    {
        *b = -(1 + *b);
    }
    done = true;
}

void thread_2()
{
    while (false == done)
    {
        int temp = *b;
        if ((temp != 0) && (temp != -1)) error++;
    }
}

int main()
{
    int a = 0;
    b = &a;

    thread t1{ thread_1 };
    thread t2{ thread_2 };

    t1.join();
    t2.join();

    cout << "Number of Error = " << error << endl;
}

```

Number of Error = 0
계속하려면 아무 키나 누르십시오 . . .

에러가 0이 나온다. 그럼 이제 코드를 좀 바꿔보자.

```

#include <iostream>
#include <atomic>
using namespace std;

const int MAX = 25'000'000;

volatile int x, y;
volatile int* b;
volatile bool done = false;
int error; //volatile로 할필요없다. 두개의 스레드에서 동시에쓰는게 아니라 한 스레드만 쓰는것.

void thread_1()
{
    for (int i = 0; i < MAX; ++i)
    {
        *b = -(1 + *b);
    }
    done = true;
}

void thread_2()
{
    while (false == done)
    {
        int temp = *b;
        if ((temp != 0) && (temp != -1)) error++;
    }
}

int main()
{
    int a[64];
    int temp = reinterpret_cast<int>(&a[31]);
    temp = temp / 64 * 64;
    temp = temp - 2;
    b = reinterpret_cast<int*>(temp);
    *b = 0;

    //이게 도대체 뭐하는짓이냐? 배열의 한가운데 주소를 얻어서 주소를 가공함.
    //temp를 64의 배수로 가공함.
    //배열의 크기는 256바이트. 그 중간의 값을 가리키는것. 그리고 0으로 초기화시키고 돌리는것.
    //이게 무슨 짓이냐?

    thread t1{ thread_1 };
    thread t2{ thread_2 };

    t1.join();
    t2.join();

    cout << "Number of Error = " << error << endl;

    system("pause");
}

```

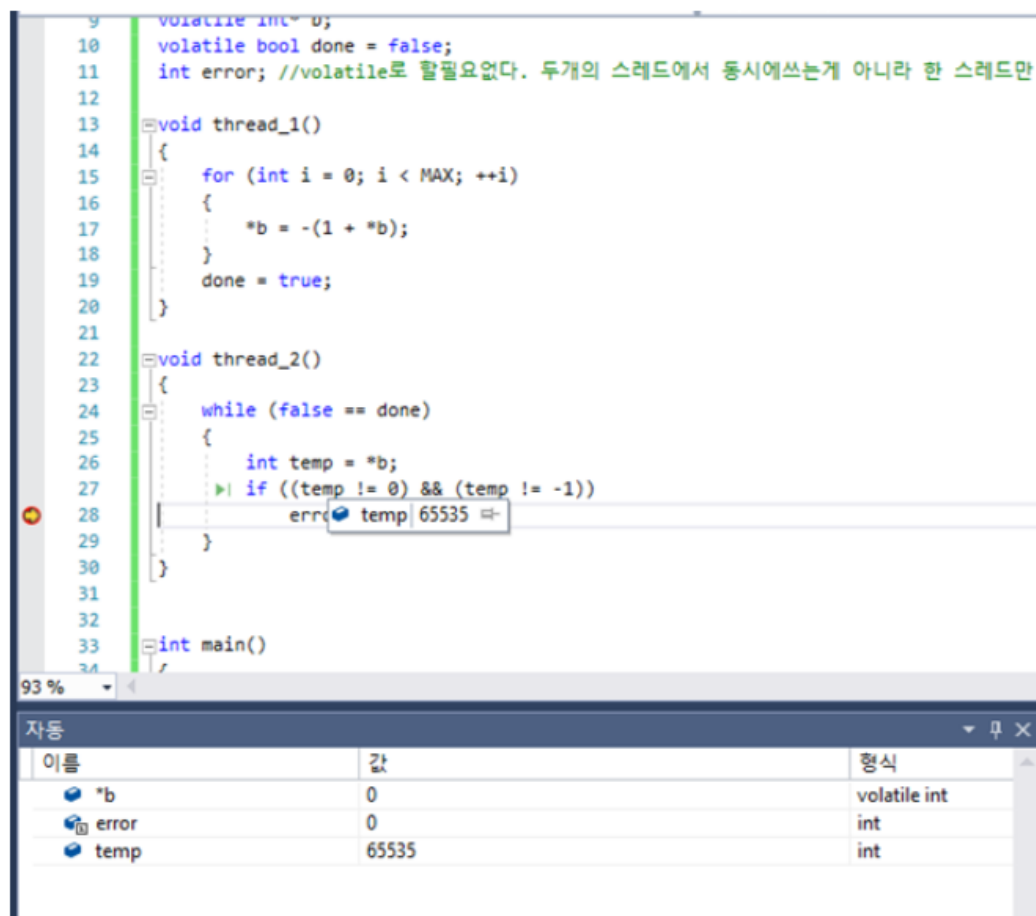
Number of Error = 1509286
계속하려면 아무 키나 누르십시오 . . .

이렇게 조금만 수정했더니 에러가 150만 개가 난다. 아깐 에러가 0이었는데..

```
int a[64];
int temp = reinterpret_cast<int>(&a[31]);
temp = temp / 64 * 64;
temp = temp - 2;
b = reinterpret_cast<int*>(temp);
*b = 0;
```

수정한 이 부분이 좀 수상하다. 여기서 뭐가 잘못됐나? 잘못된 거 없다. 다른 메모리 침범하는 거 없고, 그냥 배열 안에서만 조작한 거고 그걸 0으로 초기화한다. 어디서 잘못됐을까? 알 수 없다. 멀티스레드 프로그래밍했을 때 스레드 1은 옆 스레드 읽었을 때 0과 -1 이외의 다른 값이 나와서 에러라고 체크가 되었다. 이게 무슨 일이나? 확인을 해보아야 한다.

도대체 temp에 무슨 값이 들어있길래 에러가 나오나?



```
9 volatile int* b;
10 volatile bool done = false;
11 int error; //volatile로 할필요없다. 두개의 스레드에서 동시에쓰는게 아니라 한 스레드만
12
13 void thread_1()
14 {
15     for (int i = 0; i < MAX; ++i)
16     {
17         *b = -(1 + *b);
18     }
19     done = true;
20 }
21
22 void thread_2()
23 {
24     while (false == done)
25     {
26         int temp = *b;
27         if ((temp != 0) && (temp != -1))
28             error = temp;
29     }
30 }
31
32
33 int main()
34 {
```

93 %

이름	값	형식
*b	0	volatile int
error	0	int
temp	65535	int

이 사진에서 temp는 분명 65535인데 *b는 0이 들어가 있다. 어떡하면 이럴 수 있나? 이런 일이 벌어지는 게 현실이고 원인은 모른다. **멀티스레드 프로그래밍이 어려운 이유 3번째 캐시 문제이다.**

탐정이 되어서 하나하나 범인의 행적을 쫓아야 한다. 첫 번째 단서. 틀린 값이 나오는데 이 값에 무슨 의미가 있는가? 이걸 살펴봐야 한다. 65535? 계속하면 -65536이 나오고 65535가 나오고... 랜덤 한 값이 아닌 특정한 값이다. 예러는 계속 늘어가고.. 이 값이 과연 무슨 의미가 있는가?

무엇을 의미하는지 한번 출력해보자

```
void thread_2()
{
    while (false == done)
    {
        int temp = *b;
        if ((temp != 0) && (temp != -1))
        {
            cout << hex << temp << " ";
            error++;
        }
    }
}
```

```
ffff ffff0000 ffff0000 ffff0000 ffff ffff0000 ffff0000 ffff0000 ffff ffff0000 ffff0000 ffff0000 ffff0000 ffff f
fff ffff0000 ffff ffff0000 ffff ffff ffff0000 ffff ffff0000 ffff0000 ffff0000 ffff ffff ffff0000 ffff ffff ffff
0000 ffff ffff0000 ffff0000 ffff0000 ffff ffff0000 ffff ffff ffff ffff0000 ffff ffff0000 ffff0000 ffff ffff ffff ff
fff0000 ffff ffff ffff ffff0000 ffff ffff ffff ffff0000 ffff ffff0000 ffff ffff0000 ffff ffff0000 ffff0000 ffff ffff
0000 ffff ffff0000 ffff0000 ffff0000 ffff0000 ffff0000 ffff0000 ffff0000 ffff ffff0000 ffff0000 ffff0000 ffff0000 ffff00
00 ffff ffff ffff ffff ffff ffff ffff0000 ffff0000 ffff ffff0000 ffff ffff ffff0000 ffff ffff0000 ffff ffff ffff ff
fff0000 ffff ffff0000 ffff ffff ffff ffff ffff ffff0000 ffff ffff0000 ffff ffff Number of Error = 73
계속하려면 아무 키나 누르십시오 . . .
```

ffff와 ffff0000 이 쪽 출력되는 것을 볼 수 있다.

0은 0000 0000 이게 0을 의미하는 것이었다.

-1은 ffff ffff 이게 -1을 의미하는 것이었다.

그럼 ffff 0000은 뭐냐? 0000 0000에서 ffff ffff가 될 때, 한 번에 되지 않고 중간에 0000 ffff가 되었다가, ffff ffff가 되고, 여기서 또 ffff 0000이 되었다가 000 0 0000이 되는 것이다. **한방에 바뀌는 게 아니고, 중간값을 옆에 스레드에서 써버린 것.** 0을 -1로 썼는데 0이나 -1이 아닌 전혀 다른 값이 읽히는 것. 이것이 멀티 스레드 프로그래밍에서의 3번째 문제이다.


```

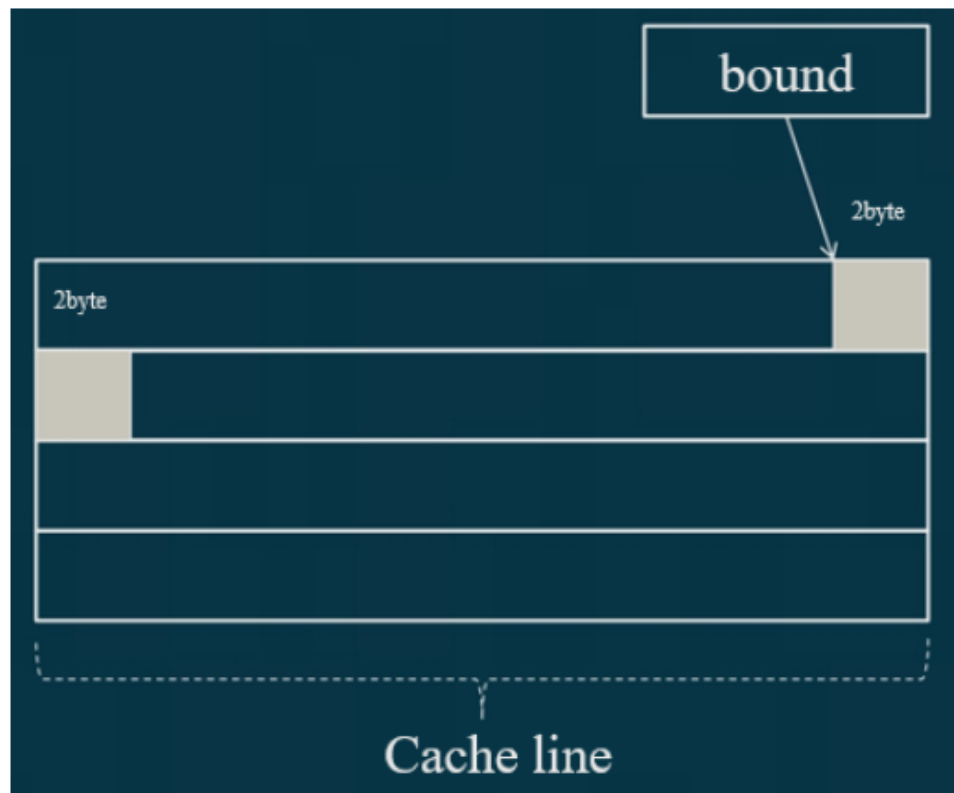
int main()
{
    int a[64];
    int temp = reinterpret_cast<int>(&a[31]);
    temp = temp / 64 * 64;
    //temp = temp - 2;
    b = reinterpret_cast<int*>(temp);
    *b = 0;
}

```

저 `temp = temp - 2;` 부분을 주석으로 지우면 에러가 없다고 결과가 나올 것이다. 그럼 2 말고 4 빼면? 에러가 마찬가지로 안 난다. 그럼 1을 빼면? 에러가 난다. 그런데! 숫자가 바뀐다. 캐시는 메모리에 있는 값을 빠르게 읽고 쓰고 할 수 있게 하는 것이다. **A는 4바이트, 메모리에 있어서 읽어야 하는데 캐시에 올려놓고 읽는다. 4바이트가 캐시에 올라오느냐? 아니다. A가 포함되어 있는 캐시라인을 통째로 읽는다.** 1바이트 읽으려 해도 메모리를 A가 포함된 한 라인 전체를 읽는다. **캐시라인의 사이즈는 64바이트.** 우리가 A를 읽으면 A가 포함된 64바이트를 통째로 읽어서 캐시에 갖다 놓는다.

메모리 낭비가 아니냐? 아니다. 다 이유가 있다. **첫 번째 이유는 4바이트를 읽나 64바이트를 읽나 속도 차이가 없다.** 왜? 메모리를 읽을 때 처음에는 읽는 레이턴시가 문제지 한 번에 4바이트 읽나 64바이트 읽나 속도 차이가 없다. CPU의 입장에서서는 어차피 읽는데 옆에꺼 읽어두면 캐시 히트 범위가 많아진다. 많이 읽으면 좋으니까 많이 읽는 거고. 그럼 128바이트씩 읽지 왜? 숫자가 커지면 읽는 속도의 차이가 생기고.. 한 번에 많이 읽으면 나머지를 다 사용할 확률이 올라가지 않는다. 조금 읽어도 안되고, 많이 읽어도 안된다. 적절한 사이즈가 딱 64바이트다. 이걸 인텔이 정한 것이다. 수많은 벤치마크 프로그램을 돌리고 알아냈다.

두 번째 이유는, 캐시 관리를 해야 해서다. 캐시에 메모리를 읽을 때 데이터가 캐시에 있나 알아봐야 한다. 메모리가 어디에서 꺼냈는지 주서가 붙어있다. **캐시 메모리는 실제 데이터도 있지만, 태그가 붙어있다.** 1바이트마다 태그가 있으면 실제 데이터보다 태그가 메모리를 훨씬 더 차지한다. 이러면 안 된다. **태그와 실제 데이터의 비율이 적절해야 한다.** 태그 하나에 데이터가 많이 붙어 있을수록 태그로 낭비되는 메모리가 줄어든다. 그래서 라인 단위로 관리를 한다. 아까 프로그램에서 한 짓이 뭔가? `temp = temp / 64 * 64`는 바운드를 64배 수로 만들고, 바운드가 캐시라인 시작 위치에 딱 놓이게 된다. 그런데 여기서 `temp = temp - 2`를 하면? 2바이트가 캐시라인에 걸처진다. 이게 무슨 이야기냐?



위 사진처럼 걸쳐있게 된다. 한방에 쓰면 괜찮은데, 다른 캐시라인이면 캐시에 쓸 때 한 클럭에 한 줄을 쓰고, 다음 클럭에 다음 줄 쓰고, 클럭이 쪼개진다. 동시에 쓰질 못한다. **왜? cpu가 메모리를 읽고 쓸 때 캐시라인 단위로 읽는다. 한 번에 읽을 수 있는 데이터 단위는 캐시라인이다.** 한 방에 업데이트하고 싶으면 2라인씩 읽어야 하는데 그걸 못하고 2번에 나눠 읽어야 한다.

싱글 스레드에서는 문제가 없다. 2번 읽고 0, 또 2번 읽고 -1 리턴하면 되는데, 한 줄 업데이트하고 중간에 다른 코어가 끼어들어서 두줄을 읽는다. 그럼 중간값이 나오는 것이다. 그래서 -1이 나올 때 반만 읽혀서 나오는 것이다. 0으로 바뀔 때도 마찬가지이다. 잘리는 위치가 중간에서 시프트 된 것. 그러면 $temp = temp - 1$ 했을 때 오류의 16진수 순서가 00ff ffff / ff00 0000여야 하는데 왜 0000 00ff 이런 식이나? 리틀 엔디 안이어서. 그래서 숫자가 반대로 보이게 나오는 것이다. $temp = temp - 3$ 을 하면 또 다른 결과가 나온다.

스레드 1은 중간값을 읽지 않는다. 그런데 스레드 2에서는 중간값이 보이는 것이다. 그래서 에러가 난다. 이런 걸 막으려면 어떻게 해야 하는가? 캐시를 안 쓰면 된다. 캐시를 쓰지 않으면 이런 오류가 안 생긴다. 근데 그럼 속도가 10배 느려진다. 멀티스레드 에러를 줄이려고 캐시를 안 쓴다? 그럼 멀티스레드로 4배 빨라지고, 캐시를 안 써서 10배 느려진다.

캐시를 disable 하는 명령어가 있지만 쓸 이유가 없다. 그래서 이 캐시라인에 걸처지면 황당한 값이 나온다는 걸 알아야 한다. 근데 누가 이딴 식으로 포인터 주소를 만드는가? 우리가 만든다.

int a 한 다음, int b = &a 하면 이게 캐시라인에 걸처질 확률은 0이다. 모든 int 변수를 정할 때 , 딱 4의 배수에 맞게 컴파일하니까.

가끔 이런 의문이 들 때 있다. int 선언하고, char 선언하고 , int 선언하면 그럼 주소는 100 104 104 이런 식일까. 아니다. 100 104 108 이런 식으로 나온다. 그럼 c의 3바이트는? 그냥 놓고 있다. 메모리 낭비가 아니다. 100 104 105 이딴 식으로 메모리가 할당되면 멀티스레드에서 개판이 난다. 내가 데미지를 받았다 그럼 - 데미지인데, 멀티스레드에서 +가 되면 hp가 뺏기된다. 만약 저게 포인터였다면 프로그램이 죽는다.

우리가 만드는 게임 서버 소스의 패킷이 저런 식이다.

```
char buf[256]

buf[0] = length;
buf[1] = OP_MOVE;
*((float *)&buf[2]) = x;
*((float *)&buf[6]) = y;
*((float *)&buf[10]) = z;
*((float *)&buf[14]) = dx;
*((float *)&buf[18]) = dy;
*((float *)&buf[22]) = dz;
*((float *)&buf[26]) = ax;
*((float *)&buf[30]) = ay;
*((float *)&buf[34]) = az;
*((int *)&buf[38]) = h;
...
send( fd, buf, (size_t)buf[0], 0 );
```

- 결과
 - 중간값
 - write시 최종 값과 초기 값이 아닌 다른 값이 도중에 메모리에 써지는 현상
 - 이유는?
 - Cache Line Size Boundary
 - 대책은?
 - Pointer를 절대 믿지 마라.
 - Byte밖에 믿을 수 없다.
 - Pointer가 아닌 변수는 visual c++ 이 잘해준다.

위 사진처럼 데이터를 뭉쳐서 차곡차곡 뽁뽁하게 채워서 집어넣는다. 저런 이상한 짓을 우리가 서버 만들 때 쓴다. 그럼 마지막 int는 캐시라인에 걸려질 수 있다.

버퍼는 4바이트 단위로 할당되는데 중간에 채워 넣으면 마지막은 걸려질 수 있다. 운나쁘면. 확률은 1/16이다. 왜? 1/64인데 내가 가질 수 있는 주소는 4바이트 단위니까 4/64 해서 1/16이 된다. 16개의 위치. 그래서 이걸 매우 높은 확률이다. 그리고 int는 하나밖에 없는데 float 4바이트짜리, 다른 것도 마찬가지다. 그럼 이 중 하나라도 걸려질 확률은 10/16이 넘는다.

이걸 어떻게 해결해야 하는가? **mutex를 사용하면 된다.** 근데 이런 고생을 하는 건 mutex를 안 쓰기 위해서였으니, 안 쓰고 해결해보자. **모든 포인터 변수를 다 검사하고 사용해야 한다.** 포인터 변수. 또 pragma pack 이거 사용하면 안 된다.

Struct pos{ char id; int x; int y;} 했을 때 x, y는 절대 캐시라인에 걸려지지 않는데, pragma pack 하면 높은 확률로 걸친다. 각 위치는 1, 5이다. pragma pack을 쓰거나 포인터 연산을 한다고 하면 주의해야 한다. 걸치나 안 걸치나 검사하면서 프로그래밍해야 하고, pragma pack 쓰면 안 되는데 어쩔 수 없이 쓴다. 패킷 정의할 때.

조심해서 프로그래밍하는 수밖에 없다. 그런 일이 없도록. 포인터가 아닌 다른 변수는 비주얼 스튜디오가 잘해주니까 int* b는 아무 문제가 없는데, 실제 주소를 넣어주거나 할 때는 조심하는 수밖에 없다.

이게 세 번째 캐시라인 문제였다. 이것의 해결방법은 조심. 이것밖에 없다. mfence를 넣는다? 하나도 통하질 않는다.

- 메모리 일관성

- 무모 순성 (Consistenct)라고 불린다.
- Memory Consistency 또는 Memory Ordering이라고 불린다.
- 강한 모델과 약한 모델이 있고 각각 허용되는 결과들을 제한하는 정도가 다르다.
- 실제 컴퓨터가 제공하는 모델과 프로그래밍에 사용되는 모델을 동기화 명령을 사용해서 일치시켜야 한다.

이런 걸 메모리 일관성이라고 부른다. 순서나 값이 맞지 않아서 일관성이 어긋난다. Memory Consistency문제라고 부른다. 이런 문제를 자세히 알고 싶으면 Memory Consistency로 검색하면 논문이 몇 편이 나오고, 자세한 내용은 대학원에서 배운다. 학부 레벨에서는 조심하는 수밖에 없다. 그리고 consistency는 여러 종류가 있다. 몇 가지만 하자.

Type	Alpha	ARMv7	PA-RISC	POWER	SPARC RMO	SPARC PSO	SPARC TSO	x86	x86 oostore	AMD64	IA64	zSeries
Loads reordered after Loads	Y	Y	Y	Y	Y				Y		Y	
Loads reordered after Stores	Y	Y	Y	Y	Y				Y		Y	
Stores reordered after Stores	Y	Y	Y	Y	Y	Y			Y		Y	
Stores reordered after Loads	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Atomic reordered with Loads	Y	Y		Y	Y						Y	
Atomic reordered with Stores	Y	Y		Y	Y	Y					Y	
Dependent Loads reordered	Y											
Incoherent Instruction cache pipeline	Y	Y		Y	Y	Y	Y	Y	Y		Y	Y

http://en.wikipedia.org/wiki/Memory_ordering

메모리 일관성이 깨지는 경우 cpu가 어디까지 허용을 하는가? cpu별로 다르다. 읽고 쓰고, 읽고 읽고, 이 모든 조합에 대해서 이다. atomic? 서로 엇갈리지 않게 해주는 연산이 atomic 연산이다. int 선언을 atomic으로 하면 atomic끼리는 안 어긋나도록 한다고 배웠는데, 이것하고 노말 연산 사이에서는 서로 어긋날 수 있다. 읽는 거랑 atomic이 어긋나는가, 쓰는 거랑 어긋나는가, dependent로 읽는데 아무 상관없는 값이 아니라 같은 주소 값을 읽을 때 어긋날 수 있는가. 다 따져본다. 캐시라인에 걸치는 거, cpu 굉장히 많다. 우리가 쓰는 x86은 굉장히 양반이다.

피터슨 알고리즘이 어긋나는 이유가 이거다. 저장과 읽는 게 순서가 바뀔 수 있는 것. 그나마 우리가 쓰는 게 상황이 낫지만 좋아할 건 아니다. 성능 제한을 많이 걸었다는 뜻이다. 그래서 파이프라인 스톱이 굉장히 많이 일어난다. 그래서 arm이 성능은 더 좋다. Y가 하나도 없는 cpu는 없다. 그래서 프로그래밍할 때 주의해야 한다.