- **Atomic**
  - 모든 Atomic 연산은 다른 Atomic 연산과 동시에 수행되지 않는다
  - Atomic연산은 이전의 명령어의 실행이 모두 끝날 때까지 실행되지 않는다.
  - Atomic연산 이후의 명령어는 Atomic연산이 완전히 종료한 후에 시작된다.
  - 모든 Atomic연산의 실행 순서는 모든 스레드에서 동일하게 관찰된다 (이건 나중에)

Atomic은 뭐냐? 크리티컬 섹션 작업을 수행하는데, 어떻게 실행을 해야 하느냐? 나 실행할 때 다른 애 아무도 실행 못하게 막고 실행해야 하느냐? 그건 아니다. 같이 실행해도 되는데, 결과가 뒤섞이지 않고 다른 애랑 같이 실행해서 올바른 결과가 나오게만 하면 같이 실행해도 상관없지 않으냐. 그게 안되니까 문제지. 상호 배제 이건 성능개선과 거리가 있다. 내가 할 땐 다른 애 아무도 하지 말라는 이야기니까. 그래서, Atomic은 결과만 제대로 나오면 된다는 해결책 중에 하나이다. 원자. 더 이상 나눌 수 없는 것. Atomic이라 함은, 중간에 아무도 끼어들지 말아야 하는 게 이것이다. **실행중 다른 cpu가 끼어들지 못하게 한다.** 다른 아토믹 연산과 동시에 수행되지 않는 것. 동시에 실행되더라도 결과는 동시에 실행되지 않는 결과가 나오면 된다.

그래서 이 연산 어떻게 정의하느냐? 어떻게 다른 Atomic연산과 동시에 수행되지 않고 따로 수행되느냐. 또 이런 조건들이 붙는다. Atomic연산이 실행하기 시작할 땐 이전의 다른 연산이 다 끝난 뒤 Atomic연산부터 실행해야 하고, 이 연산이 실행이 끝날 때까지 다른 명령어들을 실행하면 안 된다. Atomic연산이 여러 개 있을 때 어떤 게 어떤 것보다 먼저 위에 실행되는지 그 순서가 모든 스레드에서 동일하게 되어야 한다. Atomic 하게 실행을 했는데, a가 먼저 실행되느냐 b가 먼저 실행되느냐에 따라 결과가 다르다. 서로 겹치지 않는 것뿐만 아니라 어느 스레드에서 봐도 똑같은 값이 나와야 한다.

왜 이렇게 정하냐? sum+=2가 어떻게 동작해야하는지 직관적으로 프로그래머가 생각한 것과 똑같이 동작해야 하기 때문. 사람의 생각과 똑같은 결과가 나와야 한다.

```
#include <atomic>

std::atomic <int> sum;
```

c++11에는 Atomic이라는 것을 제공한다. 2를 더하는 것 이걸 Atomic하게 해야겠다 그러면 그 정도는 lock을 쓰지 않아도 해결해주마 해준다. 그러나 성능이 다를 이유가 없다. **호환성이 올라가는 것 뿐이고 성능은 변함이 없다.**

Atomic에는 volatile이라는 뜻도 같이 들어가 있다. 그렇기 때문에 volatile 넣어도 되지만 큰 차이 없다.
Atomic으로 하니까 올바른 결과는 나온다. 성능에 변함은 없지만.

- **정답은?**
  - 처음부터 data race가 적도록 프로그램을 재작성하는 것이 좋다
    - lock이나 atomic 연산 개수를 줄일 수 있다.
  - 하지만, lock이나 atomic연산을 완전히 없애는 것은 불가능하다.

기존의 프로그램에 lock 을 깨작깨작 붙여서 되는 경우는 거의 없다. 알고리즘을 다 뜯어고치고 자료구조 새로 정의하고 해서 처음부터 새로 작성해야 한다.

lock이나 atomic의 연산 개수를 줄여야 하지만, 완전히 없앨 수는 없다. 스레드간 커뮤니케이션을 완전히 없애는 것은 불가능하다. 그게 가능하면 멀티스레드 프로그래밍하지 않는다. 멀티 프로세서 프로그래밍을 따로 하는 게 낫다.

그러면 어떻게 줄일 수 있을까??

- **실습**
  - Data Race를 최소화 하도록 프로그램을 변경해 실행해보자.
  - 역시 1,2,4,8 thread에서의 실행시간을 재보자.

```
#include <iostream>
#include <thread>
#include <mutex>
#include <chrono>
#include <vector>

using namespace std;
using namespace chrono;

volatile int sum;
mutex mylock;

void do_work(int num_thread) {

	volatile int local_sum = 0;

	for (int i = 0; i < 50000000/ num_thread; ++i) {
		local_sum += 2;
	}

	mylock.lock();
	sum += local_sum;
	mylock.unlock();
}

int main() {

	for (int num_thread = 1; num_thread <= 16; 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_work, num_thread);

		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";
	}

}
```

```
Threads[ 1] , sum= 100000000, Exec_time =72 msecs
Threads[ 2] , sum= 100000000, Exec_time =35 msecs
Threads[ 4] , sum= 100000000, Exec_time =19 msecs
Threads[ 8] , sum= 100000000, Exec_time =10 msecs
Threads[ 16] , sum= 100000000, Exec_time =10 msecs
계속하려면 아무 키나 누르십시오 . . .
```

올바른 실행결과. 쿼드코어이기에 4배이상 빨라지지 않는다.

do_work 함수 앞뒤로 lock/unlock을 거는 것이 아니다. sum에 굳이 2를 더하는 게 아니라 결과를 저장하자. 근데 그거에 lock을 걸자. sum을 더해주는 곳에 걸지 말고. 그때 새로 만드는 local sum 변수는 volatile로 정의해줘야 컴파일러가 사기를 안친다. (5천만 번을 루프를 다 돌지 않고 한 번에 더해주는 행위)

올바른 결과가 나오고, 실행속도는 듀얼코어에서 2배, 쿼드코어에서 4배. **근데 8배는 안빨라진다. 왜??? 코어가 4개여서 동시에 실행되는 스레드는 4개까지 밖에 없기 때문이다.** 그래스 하드웨어 코어의 개수가 왜 중요한지 알 수 있다. 쿼드코어 cpu는 4배까지밖에 안 빨라진다. 32 코어 cpu면 32배까지 빨라진다. 이게 제대로 된 멀티스레드 프로그래밍이다.

결국 공유변수 access 회수가 최소가 되도록 해야 한다. 5000만 번 access 하는 게 아니라.

	실행시간	결과
1 Thread	116	100000000
2 Thread	62	52661344
4 Thread	49	34748866
8 Thread	43	28327790

No LOCK

	실행시간	결과
1 Thread	3,226	100000000
2 Thread	15,642	100000000
4 Thread	11,245	100000000
8 Thread	14,191	100000000

With LOCK

	실행시간	결과
1 Thread	379	100000000
2 Thread	555	100000000
4 Thread	568	100000000
8 Thread	566	100000000

With Atomic

	실행시간	결과
1 Thread	113	100000000
2 Thread	61	100000000
4 Thread	30	100000000
8 Threads	33	100000000

정답

정리 (i7-920 - 4core)

- **정리**
  - 병렬 컴퓨터란 무엇인가?
  - 쓰레드란 무엇인가?
  - 왜 멀티스레드 프로그래밍을 해야 하는가?
  - 멀티스레드 프로그래밍은 어떻게 해야 하는가? (c++ 표준 라이브러리)
  - 멀티스레드 프로그래밍의 어려움 (Data Race, 성능)

병렬 컴퓨터란 무엇인가?
하드웨어 여러 개를 동시에 사용해서 한 어플리케이션의 성능을 올리는 것이 특징.

왜 멀티스레드 프로그래밍을 해야 하는가?
CPU의 성능을 올리기 위해서. 이 이유가 아니라 하드디스크나, 네트워크 성능을 위해서는 스레드를 쓸 게 아니라 I/O를 잘해야 되는 거고, CPU가 너무 느려서 성능이 안 나올 때 해야 한다.

멀티스레드 프로그래밍은 어떻게 해야 하는가? (c++ 표준 라이브러리)
C++11 표준을 쓰면 되는 거라 어렵진 않다.

멀티스레드 프로그래밍의 어려움 (Data Race, 성능)
멀티스레드 SDK를 가져와서 설치할 필요 없다. 겁먹지 마라. 프로젝트 옵션에서 뭐 세팅해야 하고 CPU 할당하고 스레드마다 속성에 넣어주고 그런 거 없다. 그냥 싱글 스레드 프로그램 짜듯 하면 되고, 단지 그 안에 스레드 객체만 만들면 된다. 그런데 실제 어려운 이유는 data race들을 다 잡아줘야 하고, 그걸 다 없앨 수 없기 때문이다. 최소한으로 억제해서 성능이 올라가도록 잘 프로그래밍을 해야 하는데 그게 어렵다.

• • •

멀티스레드 프로그래밍이 무엇인지는 이제 다 이야기했다. 이렇게 한 학기 강의가 끝난다면 좋겠지만 이렇게 간단하지 않다. 외부에서도 보면 멀티스레드 관한 강의도 많고 책도 많고 스터디도 많지만 거기서 다루는 내용은 여기까지이다. 더 깊게 내려가지 않는다. 왜?? 어려우니까. 모르니까. 제대로 된 멀티스레드 프로그래머다 그럼 여기서 끝내지 않고 더 깊게 내려가야한다. 그래서 위 내용을 2주동안 소개했고, 본격적으로 제대로된 프로그래밍이 왜 어렵고 어떻게 극복하느냐 자세히 이야기 한다. 이 학기를 끝내고 나면 어디가서 다른사람이 하는 이야기를 못알아 듣진 않는다. 누가 잘못된 애기를 하고 있구나 그정도는 판별할 수 있다. 근데 제대로된 프로그래밍을 설계하기엔 한 학기로는 부족하다. 두 학기 정도 대학원에서 더 들어야 한다. 미국 대학교도 마찬가지이고 학부 때는 이 정도 맛보기만 한다.