

지난 시간에 2를 5000만 번 더하는 프로그램을 만들었었다. 이 프로그램을 멀티스레드로 바꿔보자.

- 이번엔 Thread 2개를 만드는 프로그램을 작성해보자.
 - 전역 변수 sum
 - 스레드가 하는 일은 $\text{sum} += 2$; 오천만 번 수행. (하나의 스레드는 이천오백만 번 수행)
 - 스레드 종료 후 sum 출력

```
#include <iostream>
#include <thread>

int sum;
using namespace std;

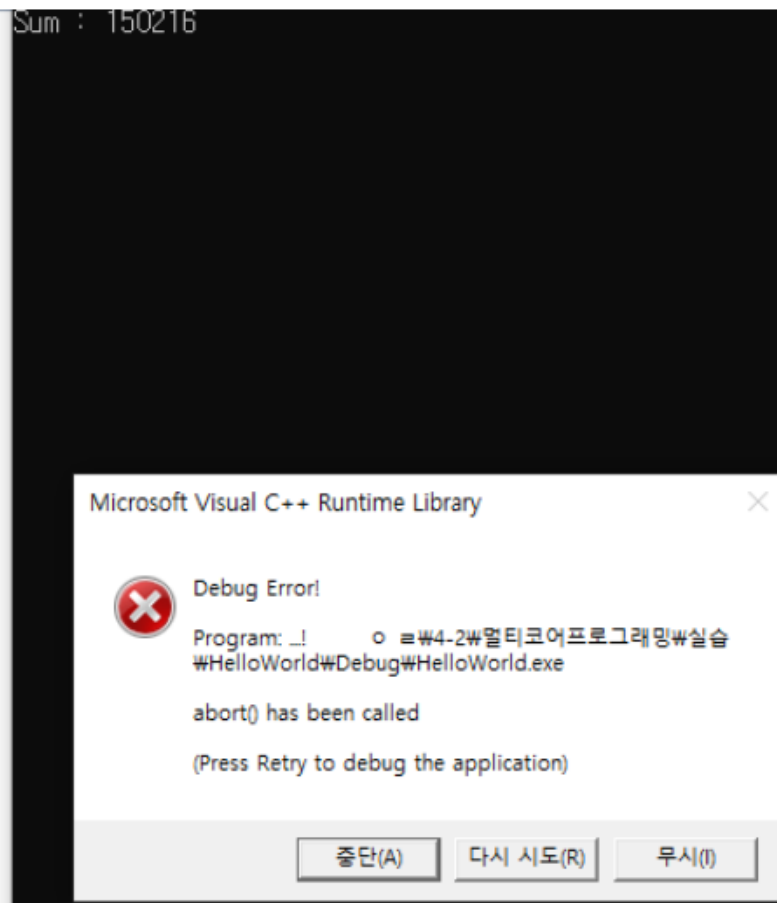
void do_work() {
    for (int i = 0; i < 25000000; ++i) {
        sum += 2;
    }
}

int main() {
    while (1)
    {
        sum = 0;

        thread t1{ do_work };
        thread t2{ do_work };

        cout << "Sum : " << sum << endl;

        system("pause");
    }
}
```



이렇게 짜면 제대로 된 결과값이 나오지 않는다. 택도 없이 작은 값. 왜일까???

스레드가 작업이 끝나지 않았는데 sum을 출력해버리기 때문이다. 그리고 abort() has been called라고 하는 디버깅 에러가 뜨는데, 이건 스레드가 종료되지도 않았는데 프로그램이 끝났다는 것이다. 파일을 오픈했는데 닫지 않거나, 메모리를 new 했는데 delete 하지 않거나, 모두 똑같은 얘기다. 스레드가 종료될 때까지 기다려야 하기 때문에 join으로 기다려야 한다.

```

#include <iostream>
#include <thread>

int sum;
using namespace std;

void do_work() {
    for (int i = 0; i < 25000000; ++i) {
        sum += 2;
    }
}

int main() {
    while (1)
    {
        sum = 0;

        thread t1{ do_work };
        thread t2{ do_work };

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

        cout << "Sum : " << sum << endl;

        system("pause");
    }
}

```

```

Sum : 56198740
Sum : 57193988
Sum : 53643868
Sum : 51338588
Sum : 56780232
Sum : 54553588
Sum : 52979382
Sum : 55483180
Sum : 54549354
Sum : 56474218
Sum : 54535526
Sum : 53312422
Sum : 52726696
Sum : 52260972
Sum : 57234410
Sum : 54011716
Sum : 52492864
Sum : 52865302
Sum : 53329846
Sum : 56054528
Sum : 59534852
Sum : 66253200
Sum : 54753784
Sum : 54473170
Sum : 59832428
Sum : 53359378
Sum : 57158638
Sum : 51707828
Sum : 57523216

```

이렇게 join을 써도 제대로 된 결과값이 나오지 않는다. 왜일까??

버그가 있기 때문에. 그런데 문제는 이렇게 간단한 프로그램인데, 소스코드를 봐도 에러가 있을 리가 없는 프로그램이라고 생각한다. 어디가 버그인지 알 수 없다.

그런데 운영체제를 들었다면 왜 버그가 있는지 안다.

멀티스레드 프로그래밍에서 중요한 사항

- 올바른 결과가 나와야 한다.
 - 무한루프에 빠지거나 프로그램이 오류로 종료하면 안 된다. 당연하지만 힘들다
- 멀티스레드로 인한 성능 향상이 커야 한다.
 - 성능 향상이 적으면 멀티스레드 프로그래밍을 할 이유가 없다. (여러 가지 요인이 얹혀서, 만족할 만한 성능 향상을 얻는 것은 어렵다)

지금은 단순히 덧셈이라 결과 값만 틀린 것이지만, 포인터 가지고 뭘 한다던가 stl 컨테이너 갖고 뭘 한다거나 하면 틀린 값뿐 아니라 무한루프에 빠진다거나 종료되거나 온갖 일을 겪게 될 것이다. 올바른 결과가 나오는 게 너무나 당연하지만, 이렇게 간단한 프로그램조차 올바른 결과를 얻는 게 어렵다.

그리고 멀티스레드를 쓰기 전보다 2배 빨라야 가장 좋다. 스레드가 2개이기 때문이다. 스레드를 2개 썼는데 3배 빨라질 수는 없다. 2배까진 안 빨라도 최소한 느려지면 안 된다. 그런데 그게 어렵다. 실행 속도, 성능은 뒤로 미루고 올바른 값을 만들어 내는 것부터 해보자.

그럼 위의 실습에서의 문제는 뭐였을까?

sum += 2 이게 문제다.

멀티스레드 프로그래밍에서는 이렇게 프로그래밍하면 안 된다. sum은 전역 변수이므로 data 영역에 저장되고, 두 스레드 모두 접근이 가능하다. 그리고 읽고, 2를 더하고, 쓸 수 있다.

이때, cpu가 어떻게 실행하나 봐 보자. 기계어로 아래와 같이 컴파일된다

```
for (auto i = 0; i < 25000000; ++i) sum += 2;
00D46B1E mov     dword ptr [ebp-8],0
00D46B25 jmp     thread_func+30h (0D46B30h)
00D46B27 mov     eax,dword ptr [ebp-8]
00D46B2A add     eax,1
00D46B2D mov     dword ptr [ebp-8],eax
00D46B30 cmp     dword ptr [ebp-8],17D7840h
00D46B37 jge     thread_func+48h (0D46B48h)
00D46B39 mov     eax,dword ptr ds:[00D51390h]
00D46B3E add     eax,2
00D46B41 mov     dword ptr ds:[00D51390h],eax
00D46B46 jmp     thread_func+27h (0D46B27h)
}
```

debug -> windows -> disassembly

기계어를 몰라도 괜찮다.

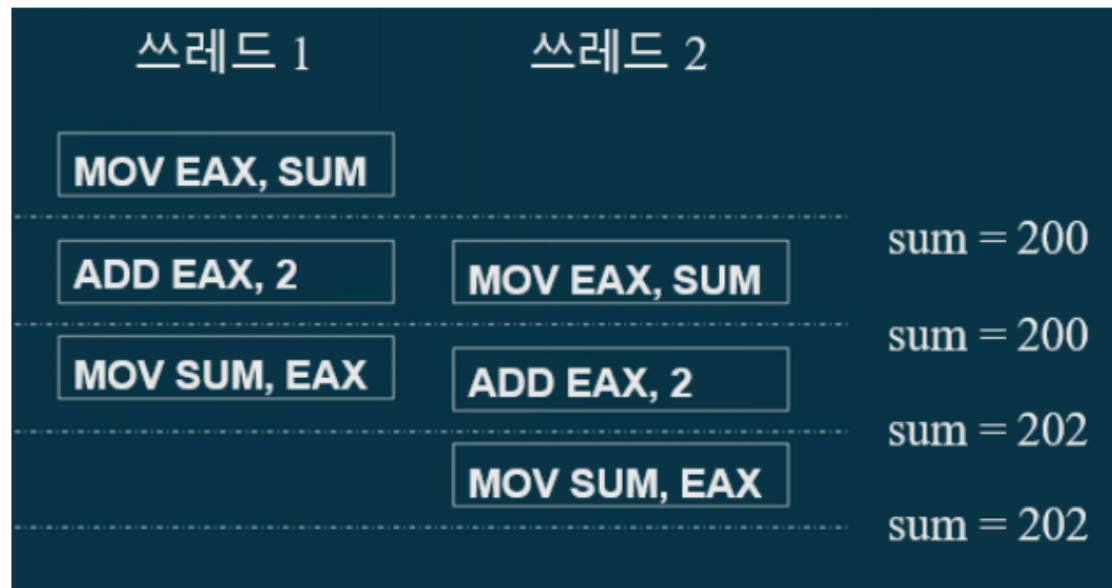
저 옆에 보이는 [00D51390h] 이게 sum이라는 변수의 주소이다. 2를 더한 뒤 같은 주소에 집어넣는 것이다.

cpu는 오퍼랜드가 2개 있는 거 있고 3개, 1개, 없는 게 있다. 여기는 2개가 있다.

우리가 쓰고 있는 cpu에는 메모리에 직접 2를 더하는 명령어는 없다. 사실 있긴 한데 내부적으로 여러 개로 갈라져서 마이크로 코드로 실행되니 기본적으로 그런 동작은 없다. 왜냐면, 메모리에 2를 더한다? 삼성에서 만든 ddr 코어 메모리 표준 규격에 2를 더한다는 건 없다. 메모리에 대한 그런 산술 연산은 지원하지 않는다. 삼성에서 파는 메모리는 2를 읽고, 쓰는 기능만 제공을 하지 2를 더한다거나 3을 빼는 메모리에 대한 산술연산은 제공이 안된다. 그래서 메모리에 있는 값에 2를 더하고 싶다면 cpu가 메모리에 있는 값을 읽은 뒤 2를 더해서 그 값을 쓰는 수밖에 없다.

그럼 비주얼 스튜디오가 이걸 어떻게 동작을 할까?

이렇게 컴파일되고 실행되는 게 문제이다. 스레드 t1 t2가 동시에 실행된다. 1번 코어에 t1, 2번 코어에 t2 동시에 실행되는데 그럼 순서가 엇갈리면서 실행이 된다.



이런 순서로 실행이 될 수 있는데 각자 2를 읽어서 2를 더하고 2를 쓴다.

스레드 1이 sum을 200일 때 읽고 2를 더했을 때 동시에 스레드 2가 sum을 읽으면 아직 sum이 200이었기 때문에 스레드 2는 202가 아닌 200을 가져간다. 스레드 1이 아직 쓰지 않았기 때문이다. 그런 뒤 스레드 1이 sum을 202로 수정을 했지만, 스레드 2는 반영되지 않고 200에서 2를 더한 202를 다시 쓴다. t1이 쓴 것을 t2가 덮어썼다. 상대방이 202를 쓴 것을 기다렸다가 그것을 읽어서 204를 써야 하는데, 순서를 안 맞추고 자기 멋대로 돌아가니 틀린 결과가 나오는 것이다. 그래서 절대로 1억이 나오지 않는다.

운이 좋으면 1억이 나올 수 있지만, 확률은 0에 수렴하고 무조건 1억보다 작은 숫자가 나온다. 1억보다 큰 값은 절대로 나올 수 없다. 2가 더해지는 것이 무시가 될 수는 있지만 기존의 숫자보다 많이 더할 수는 없다. 상한은 1억. 하한은 4가 된다. **어떻게 하한이 2가 아닌 4가 되는지는 한 번 잘 생각을 해보아라.**

그리고 중요한 건 여기서 맨날 실행할 때마다 결과값이 달라진다. 그러니까 우리가 프로그래밍해서 죽는 거면 계속 죽고, 잘못된 결과면 계속 같은 잘못된 값이 나왔는데 멀티스레드 프로그래밍을 하니까 값이 매번 다르다. 이게 특징이다. 이건 잘못 만든 프로그램이다. 실행할 때마다 동작이 다르기 때문에 디버깅하기가 힘들어진다. 계속 다뤄보도록 하자.

Data Race

- 원인
 - 공유 메모리를 여러 스레드에서 읽고 쓴다
 - 읽고 쓰는 순서에 따라 실행결과가 달라진다. (프로그래머가 예상 못한 결과가 발생)
 - 이것을 Data Race라고 한다.
- Data Race의 정의
 - 복수개의 스레드가 하나의 메모리에 동시 접근
 - 적어도 한 개는 write

틀린 결과가 왜 나오는지 계속 얘기했듯, sum이 전역 변수여서 데이터를 두 개의 스레드가 동시에 읽고 쓰고 하는데 모든 전역 변수는 공유 메모리여서. 읽고 쓰는 순서에 따라 실행결과가 달라진다. 내가 읽고 쓰는데 중간에 읽으면 내가 업데이트한 결과가 날아가고, 잠시 내가 interrupt 걸려서 해매고 있는데 상대방이 10번 업데이트를 한 뒤 내가 다시 쓸 때 상대방의 업데이트 결과를 날려버린다. 이 모든 결과는 우리가 의도한 결과가 아니고, 이런 현상을 data race라고 한다. 우리나라 말로는 책마다 다르게 번역을 했지만 경쟁상태라고 한다. 그런데 경쟁상태라고 하면 업계에 있는 사람들이 안 좋아한다. data race라고 하자. 이건 잘못된 결과를 일으키는 원인이다. 복수개의 스레드가 하나의 메모리에 접근하기 때문이다. 스레드 1개만 읽고 쓰고 하면 아무 문제가 없다. 여러 개의 스레드가 읽기만 한다면 문제가 없는데, 한놈이라도 쓰면 데이터 레이스가 일어난다. data race가 없으려면 복수개의 스레드를 만들지 않으면 되는데 그럼 멀티스레드 프로그램을 쓰지 말라는 거. 동시 접근을 하지 말라는 거? 그건 스레드끼리 서로 데이터를 주고받지 말라는 것. 한 곳에서 메모리를 읽고 있으면 다른 곳에서는 절대로 못 읽게 한다? 그런 식으로 코딩할 순 없다. read만 하고 write하지 말라 이게 어떻게 가능한가. 우리가 배운 길로는 불가능하다. 이걸 가능하게 해주는 프로그래밍 언어가 있다. 학기말쯤 하겠지만 pure functional 언어라고, write가 하나도 없어서 data race가 없는 천국 같은 멀티스레드 프로그래밍을 할 수 있다. 다른 얘기고, 데이터 레이스는 이런 것이다.

Data Race 제대로 이해하고 있나?

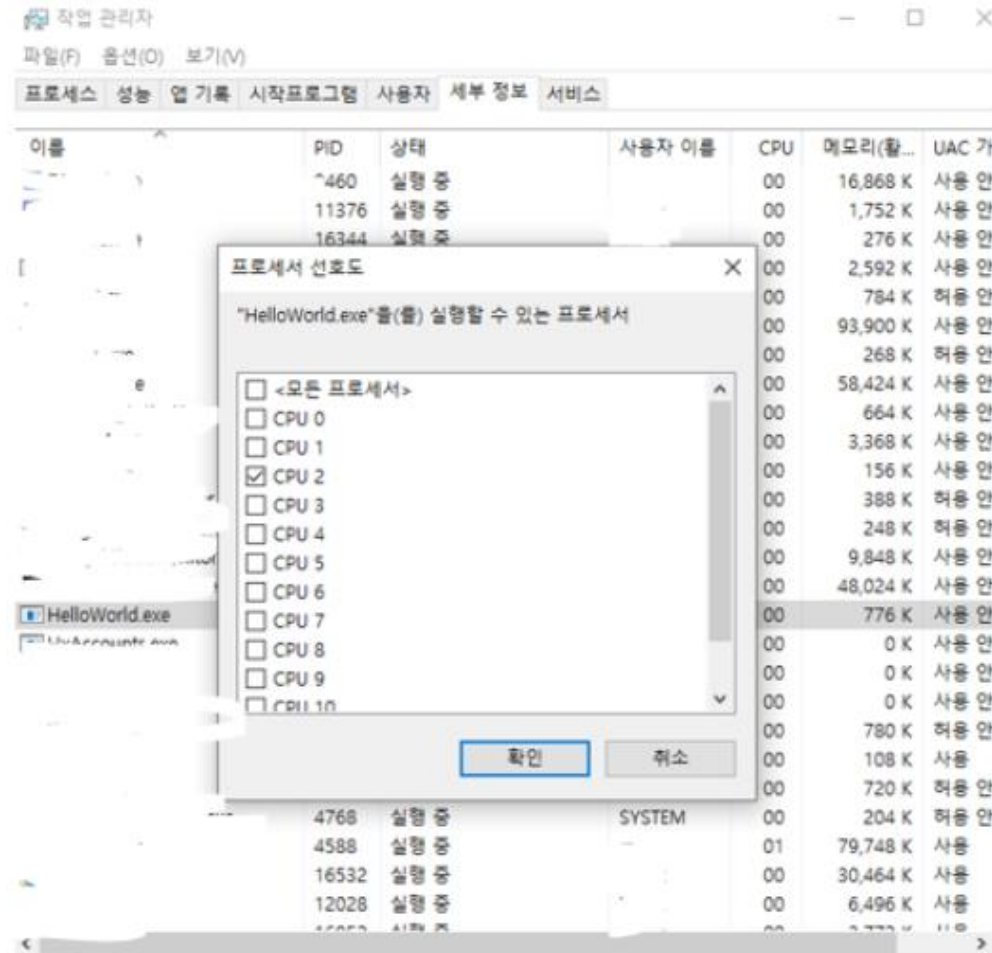
- 앞의 프로그램을 싱글코어에서 동작하게 하면 결과는?
- 앞의 프로그램의 `sum += 2`를 `_asm add sum, 2`로 바꾸면?
- `_asm add sum, 2`로 바꾼 후 싱글코어에서 동작하게 하면?

앞의 프로그램을 싱글코어에서 동작하게 하면 결과는?

싱글코어에서만 동작하게 하는 방법은

```
1  #include <stdio.h>
2  #include <iostream>
3  #include <thread>
4
5
6  int sum;
7
8
9  void do_work() {
10     for (int i = 0; i < 250000000; ++i) {
11         _asm add sum, 2;
12     }
13 }
14
15 int main() {
16
17     system("pause");
18     while (true) {
19         sum = 0;
20         std::thread t1{ do_work };
21         std::thread t2{ do_work };
22         t1.join();
23         t2.join();
24         std::cout << "sum= " << sum << std::endl;
25     }
26     system("pause");
27 }
```

이렇게 while 루프 돌기 전에 pause를 걸고,



이렇게 작업 관리자에서 실행파일을 찾아 마우스 우클릭, 프로세서 선호도, 에서 아무 코어 한 개만 체크하면 싱글코어에서만 돌아가게 할 수 있다. 디폴트는 모든 cpu에서 다 실행할 수 있다. 스레드를 2개 만들었는데 두 곳에서 실행되지 않고 하나의 코어에서 시분할로 왔다 갔다 하면서 실행이 되는 것이다.

이렇게 하고 pause 걸었던 것을 다시 돌아가게 하면? 틀린 값이 나올 수 있다. 확률이 큰 것은 아니다. 20번 돌리면 1번 정도 틀린 값이 나온다. (강의실에서 2명 정도 다른 값이 나왔었다)

싱글코어에서 돌려도 제대로 안 돌아간다. 어쩔 땐 맞고, 어쩔 땐 틀린 값이 나오는 프로그램이 더 악질적이다. 프로그래머 입장에선 아까 버그 없었는데 하면서 다른 엉뚱한 곳에서 헤매게 된다. 싱글코어에서 애러 날 확률이 줄어드는 거? 절대로 좋은 게 아니다.

앞의 프로그램의 `sum += 2`를 `_asm add sum, 2`로 바꾸면?

컴파일러에서 읽고, 더하고, 쓰고 이렇게 명령어를 3개를 쓰는데. 명령어 3개를 명령어 1개로 바꾸면 어떨까?

우리 cpu에서는 메모리에 2를 더하는 명령어가 있다. 이 명령어로 바꾸면 제대로 된 결과값이 나온다?? 별 차이 없다.

읽고, 2를 더하고, 쓰고 한다. 명령어 1개라고 해도 읽는데 한 클락, 더하는데 한 클락, 쓰는데 한 클락이 쓰인다. 명령어가 따로 나누어지지 않았다 뿐이지 스레드 두 개가 동시에 실행하면서 읽고 쓰고 접근하는 건 똑같다. **똑같이 정확하지 않은 값이 나온다.**

`_asm add sum, 2`로 바꾼 후 싱글코어에서 동작하게 하면?

절대로 이상한 값 안 나온다. 놀랍게도 100만번 돌려도 틀린 값이 안나온다. **제대로 된 결과가 나온다.** 왜?????????

위에 두 상황은 콘텍스트 스위치가 일어나서 메모리에 쓰려고 그럴 때 바로 콘텍스트 스위치가 일어나버리기 때문. 싱글코어는 한 코어에서 번갈아 실행이 되는 데 `add sum`는 명령어 하나이다. 명령어 실행 중엔 절대 interrupt가 일어나지 않는다. 명령이 끝나면 interrupt 들어온 게 있나 없나 커널에서 검사를 하기 때문에 절대 중간에 다른 스레드가 절대로 끼어들 수 없다. 그래서 항상 옳은 결과가 나오는 것임.

Data Race 해결 방법

- c++11에 lock과 unlock을 사용한다
- 복수개의 스레드가 동시에 접근할 수 없도록 하는 것. (동시에는 하나의 스레드만 접근할 수 있도록)

다행스럽게도 c++11을 만든 사람들은 이런 상황이 일어날 것을 이미 알고 있었다. 그래서 해결책을 만들었는데,

lock과 unlock을 쓰는 것이다. 락과 언락은 뭐냐? 데이터 레이스를 없애는 것이다. 복수개의 스레드를 없앨 수는 없고, 적어도 1개의 write를 막을 수는 없다. **단지 복수개의 스레드가 동시에 접근할 수 없도록 하는 것이다.** 내가 일을 끝낸 뒤 접근해라. lock을 걸어놓고 수정하고, unlock 걸고 빠져나오고. 화장실과 같은 것이다.

어떻게 쓰느냐? c++11에서 mutex라는 표준 라이브러리가 있다. 이것은 mutual(상호) exclusion(배제)을 줄여 mutex라고 한다. 상호 배제. 뮤텝스 선언(`#include <mutex>`)을 하고, 클래스 객체를 하나 만든 뒤 (`mutex mylock;`), 거기에 있는 락 언락 메서드를 실행을 해서 사용하는 것이다.

Lock Unlock 주의점

- mutex 객체는 전역 변수로
- 같은 객체 사이에서만 lock unlock이 동작.
 - 다른 mutex 객체는 상대방을 모름
- 서로 동시에 실행돼도 괜찮은 critical section이 있다면 다른 mutex 객체로 보호하는 것이 성능에 좋다.
 - 같은 mutex 객체로 보호하면 동시에 실행이 안됨.

락 연락 그냥 하지 뭐하러 객체를 만들어서 붙이냐? c++11이니까 객체 만드는 건 그렇다 쳐도 왜 그렇게 해야 하나? 그냥 깔끔하게 lock unlock 되는데? mutex는 서로 동시에 실행해도 괜찮다면 다른 mutex로 객체를 보호하는 것이 성능에 좋다. 이게 무슨 소리냐?



이 상황에서 2를 더하는 것과 3을 곱하는 것. 동시에 실행되면 안 된다. 서로서로 덮어쓰기 때문에 같은 lock을 써야 한다. 근데 sum 변수 사용과 counter 변수 사용은 실행결과엔 서로 아무런 상관이 없다. 동시에 실행되어도 상관이 없다. 아무 문제가 없다면 적극적으로 동시에 실행시켜야 한다. 왜? 그래야 성능이 올라가니까. 그러나 동시에 실행되면 안 되는 것은 같은 lock을 써야 한다. 그룹으로 나누어 따로따로 lock을 걸 수 있도록 이렇게 하는 것이다. 괜히 클래스 사용하는 것이 아니라 이렇게 lock을 쓰기 위험이다.

mutex는 전역 변수로 써야 한다. 지역변수로 쓰겠다는 것은 스레드가 서로 다른 lock을 사용한다는 것이다. 같은 lock을 쓸 때 상호 배제가 되는 것이다.

critical section은 우리나라 말로 임계 영역으로 번역을 한다. 실행했을 때 데이터 레이스를 일으키는 프로그램 영역을 **critical section**이라고 한다. 명령어 한 줄일 수도, 여러 줄 일 수도 있다.

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

using namespace std;
int sum;
mutex mylock;

void do_work() {
    for (int i = 0; i < 25000000; ++i) {
        mylock.lock();
        sum += 2;
        mylock.unlock();
    }
}

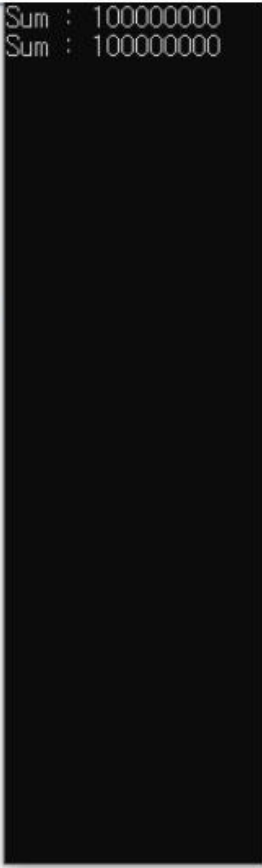
int main() {
    while (1)
    {
        sum = 0;

        thread t1{ do_work };
        thread t2{ do_work };

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

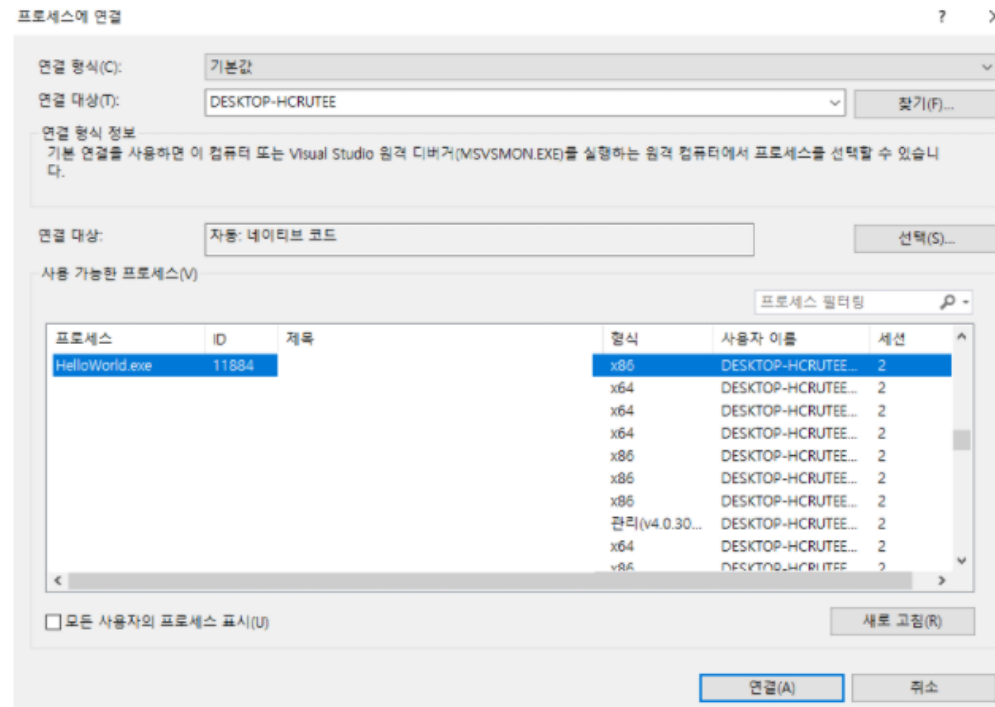
        cout << "Sum : " << sum << endl;
    }

    system("pause");
}
```



mutex를 써 보았다. 옳은 결과가 나오지만 속도가 2초 3초에 하나씩 결과가 나올 정도로 매우 느리다.

너무 느린데 이 2초 3초 사이에 지금 i가 어디까지 돌아갔는지 보고 싶으면



디버그 -> 프로세스에 연결 -> 실행파일 찾아서 연결

이렇게 연결을 한 뒤

```
1  #include <iostream>
2  #include <thread>
3  #include <mutex>
4
5  using namespace std;
6  int sum;
7  mutex mylock;
8
9  void do_work() {
10     for (int i = 0; i < 25000000; ++i) {
11         mylock.lock();
12         sum += 2;
13         mylock.unlock();
14     }
15 }
```

이렇게 확인하면 i가 몇 번쨰지 확인할 수 있다.

아깐 lock을 안 하면 틀리지만 빠른 결과가 나왔다. 지금은 멀티스레드를 해서 결과는 맞으나 10배 이상 느려졌다. 성능이 너무 안 나온다. 얼마나 너무 느린가? 느낌으로도 알 수 있다. 그런데 프로그래밍을 느낌으로 할 수 없으니 시간을 재보아야 한다. 어떻게 잴 것이냐? 스톱워치로?? c++11에서는 시간을 재는 라이브러리가 있다.

```
#include <chrono>
using namespace std::chrono;

auto t = high_resolution_clock::now();
// 측정하고 싶은 프로그램을 이곳에 위치시킨다.
auto d = high_resolution_clock::now() - t;

cout << duration_cast<milliseconds>(d).count() << " msecs\n";
//milli second를 측정하는 프로그램
```

[1000 milli seconds(ms, msecs, 밀리초) = 1 sec(초) , 즉 3000ms = 3 sec]

• 실습

- lock을 사용한 프로그램과 lock을 사용하지 않은 프로그램의 속도를 비교하시오
- Thread 개수를 1,2,4,8개로 변경하면서 측정하시오.
 - 스레드 개수가 늘어나면 하나의 스레드에서 돌아가는 루프의 회수가 줄어들어야 함 (전체 회수 = 5000만 번)
- 힌트
 - n개의 스레드를 저장할 수 있는 벡터(또는 배열) 필요..
 - 스레드 함수가 수행해야 할 루프의 회수는 5000만 번/스레드 개수
 - 2중 루프 필요
 - 스레드 개수 루프
 - 스레드 생성 루프
 - 스레드 함수에 매개 변수 전달

```

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

using namespace std;
using namespace chrono;

int sum;
mutex mylock;

void do_work(int num_thread) {
    for (int i = 0; i < 50000000/ num_thread; ++i) {
        mylock.lock();
        sum += 2;
        mylock.unlock();
    }
}

int main() {
    for (int num_thread = 1; num_thread <= 16; num_thread *= 2)
    {
        sum = 0;

        vector<thread> threads; //스레드 저장
        auto start_time = high_resolution_clock::now(); //시작시간

        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(); //종료시간
        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"; // 1sec = 1000msec
    }

    system("pause");
}

```



```
Threads[ 1] , sum= 100000000, Exec_time =7930 msecs
Threads[ 2] , sum= 100000000, Exec_time =8444 msecs
Threads[ 4] , sum= 100000000, Exec_time =8376 msecs
Threads[ 8] , sum= 100000000, Exec_time =9002 msecs
Threads[16] , sum= 100000000, Exec_time =9548 msecs
계속하려면 아무 키나 누르십시오 . . .
```

lock과 unlock을 했을 때 (debug 모드)

```
Threads[ 1] , sum= 100000000, Exec_time =141 msecs
Threads[ 2] , sum= 51963502, Exec_time =162 msecs
Threads[ 4] , sum= 30449534, Exec_time =165 msecs
Threads[ 8] , sum= 13955336, Exec_time =182 msecs
Threads[16] , sum= 11343740, Exec_time =165 msecs
```

lock과 unlock을 주석으로 뺐을 때 (debug 모드)

실행 속도 차이가 엄청 많이 난다. 70배 정도 차이가 난다. 이런 현실에서 뭐가 문제인가? 오히려 느려진다. 8개의 스레드를 돌려도 8배 빨라지지 않는다. 스레드를 늘려서 작업을 잘게 쪼개도 성능이 좋아지지 않고, 오히려 더 느려지고 있다.

스레드가 많으면 많을수록 점점 더 느려진다. 개수와 상관없이 결과는 올바르다. (lock을 쓸 경우)

멀티스레드 프로그래밍을 해야 되냐 말아야 하나 더 느려지는데. 여러 개의 코어에서 돌리면 돌릴수록 느려진다. 과연 지금 cpu를 풀로 사용하고 있는가?

HelloWorld.exe	28508	실행 중	60
----------------	-------	------	----

cpu 개수에 비례해서 사용량이 증가하고 결국 cpu를 풀로 사용하고 있다. 종료 직전까지도 100%까지 안될 텐데 다른 곳에서 조금씩 cpu를 잡아먹기 때문이다.

잠깐! 성능을 측정하는데 debug 모드로 하고 있었다. **디버그 모드에서의 성능 측정은 무의미하다. 앞으로 모든 성능은 릴리즈 모드에서 측정하자.**

Release x86

```
Threads[ 1] , sum= 100000000, Exec_time =1304 msecs
Threads[ 2] , sum= 100000000, Exec_time =1305 msecs
Threads[ 4] , sum= 100000000, Exec_time =1312 msecs
Threads[ 8] , sum= 100000000, Exec_time =1332 msecs
Threads[16] , sum= 100000000, Exec_time =1362 msecs
계속하려면 아무 키나 누르십시오 . . .
```

lock과 unlock을 했을 때 (release모드)

```
Threads[ 1] , sum= 100000000, Exec_time =2 msecs
Threads[ 2] , sum= 100000000, Exec_time =0 msecs
Threads[ 4] , sum= 100000000, Exec_time =0 msecs
Threads[ 8] , sum= 100000000, Exec_time =1 msecs
Threads[16] , sum= 100000000, Exec_time =2 msecs
계속하려면 아무 키나 누르십시오 . . .
```

lock과 unlock을 주석으로 뺐을 때 (release모드)

갑자기 10배 빨라졌다. 이게 뭐냐? 비주얼 스튜디오의 농간이다. 우리가 쓰는 비주얼 스튜디오는 제대로 돈을 내면 1년에 2백만 원 줘야 쓸 수 있는 프로그램이다. 비싼 만큼 ms에서 열심히 만든 프로그램이기 때문에 굉장히 똑똑하다. 이런 프로그램이 있다 그럼 5천만 번 일을 하는게 아니다. 대충 보아하니 5천만번 2를 더하는구나. 그럼 1억을 더하는 거랑 마찬가지로네. 그래서 컴파일하면 `sum += 1억`; 이렇게 컴파일된다. 사실이냐?

```
void do_work(int num_thread) {
00CE10D0  push     ebp
00CE10D1  mov     ebp,esp
        for (int i = 0; i < 50000000/ num_thread; ++i) {
00CE10D3  mov     eax,2FAF080h
00CE10D8  cdq
00CE10D9  idiv     eax,dword ptr [num_thread]
00CE10DC  test    eax,eax
00CE10DE  jle     do_work+1Eh (0CE10EEh)
00CE10E0  mov     ecx,dword ptr [sum (0CE5490h)]
00CE10E6  lea     eax,[ecx+eax*2]
00CE10E9  mov     dword ptr [sum (0CE5490h)],eax
        //mylock.lock();
        sum += 2;
        //mylock.unlock();
    }
}
00CE10EE  pop     ebp
00CE10EF  ret
```

정말이다

컴파일러가 영악하게 이 프로그램이 뭐하는지 알고 한방에 해버렸다. 릴리즈 모드와 디버깅 모드의 성능 차이는 거의 오천만 배 정도. 루프도 안 돌고 한방에 끝내는 5천만 배의 성능 차이가 난다. 칭찬을 해주어야 하는데, 멀티스레드 프로그래밍이 얼마나 빠르니 알아야 하기 때문에 이렇게 최적화를 하면 안 된다. 제대로 측정이 안된다. 그래서 2ms라는 말도 안 되는 속도가 나온다. 최적화가 하지 말라고 알려줘야 하고, 그렇다고 디버그 모드를 쓰는 건 아니다. **메모리 읽고 쓰는 것 제대로 하라고 알려주어야 한다. 그건 뭐냐? VOLATILE이라는 키워드이다.**

예전에 volatile과 register라는 키워드를 배웠을 것이다.

register는 메모리 저장하지 말고, 레지스터 하나 할당해라. 그래서 메모리 읽고 쓰느라 시간낭비하지 말고 레지스터에 다 저장하다가 맨 마지막에 출력할 때만 메모리나 컴퓨터에 전달을 해라. 최적화를 해라. 그런데 register int sum; 이렇게 하면 오류가 날 것인데, **최신 버전 c 표준은 register라는 키워드는 더 이상 쓰지 않겠다고 삭제**를 했다. 그래서 이런 키워드는 더이상 존재하지 않는다.

volatile 그런 거 절대로 하지 마라 레지스터에 넣거나 컴파일러가 최적화하지 말고 소스코드에 있는 건 무조건 읽고 쓰려면 써라. 그래서 **이 키워드가 붙은 것을 읽고 쓸 땐 최적화를 하지 않는다**. 그래서 실행시간을 올바르게 잴 수 있다.

```
Threads[ 1] , sum= 100000000, Exec_time =1306 msecs
Threads[ 2] , sum= 100000000, Exec_time =1308 msecs
Threads[ 4] , sum= 100000000, Exec_time =1312 msecs
Threads[ 8] , sum= 100000000, Exec_time =1345 msecs
Threads[16] , sum= 100000000, Exec_time =1362 msecs
계속하려면 아무 키나 누르십시오 . . .
```

lock과 unlock을 했을 때 (release모드) - volatile을 추가했으나 차이가 없다

```
Threads[ 1] , sum= 100000000, Exec_time =72 msecs
Threads[ 2] , sum= 50543640, Exec_time =50 msecs
Threads[ 4] , sum= 37871440, Exec_time =66 msecs
Threads[ 8] , sum= 27935764, Exec_time =38 msecs
Threads[16] , sum= 20047820, Exec_time =58 msecs
계속하려면 아무 키나 누르십시오 . . .
```

lock과 unlock을 주석으로 뺐을 때 (release모드)

멀티스레드를 해서 속도가 빨라지긴 했으나 엉터리 값이 나오는데 좋아할 게 아니다.

lock unlock을 하면서 더 느려지게 된다.

일단 mutex라는 건 일을 나눠서 하는 건 맞다. 근데 lock을 거니까 애가 일할 땐 재가 쉬어야하고, 재가 일할땐 애가 쉬어야 한다. 그럼 일을 같이하는 의미가 없다. 한 번에 하나의 스레드만 실행되고 나머지 스레드는 계속 unlock되길 기다리며 있는것이다. 화장실은 4칸이 있는데 한번에 1칸씩만 쓰고 있으니 성능이 저하된다.

또 다른 문제는 lock unlock 오버헤드가 엄청 크다.

해결방법? lock을 쓰면 안 된다. 이 얘기는 무슨 소리냐. lock 없이 어떻게 sum +=2를 하는 동안 다른 스레드가 실행되지 못하도록 하는가? 해결책이 몇 개 있다. cpu 레벨에서 막는 것이다. lock add sum, 2;라고 cpu 자체에 lock을 거는 것. 다른 코어에서 sum을 건드릴 수 없다. 그렇기 때문에 원하는 결과가 나온다.

해결방법? lock을 쓰면 안 된다. 이 얘기는 무슨 소리냐. lock 없이 어떻게 `sum += 2`를 하는 동안 다른 스레드가 실행되지 못하도록 하는가? 해결책이 몇 개 있다. cpu 레벨에서 막는 것이다. `lock add sum, 2`라고 cpu 자체에 lock을 거는 것. 다른 코어에서 `sum`을 건드릴 수 없다. 그렇기 때문에 원하는 결과가 나온다.

```
void do_work(int num_thread) {  
    for (int i = 0; i < 50000000/ num_thread; ++i) {  
        //mylock.lock();  
        //sum += 2;  
        //mylock.unlock();  
        _asm lock add sum, 2;  
    }  
}
```

```
Threads[ 1] , sum= 100000000, Exec_time =242 msecs  
Threads[ 2] , sum= 100000000, Exec_time =555 msecs  
Threads[ 4] , sum= 100000000, Exec_time =748 msecs  
Threads[ 8] , sum= 100000000, Exec_time =854 msecs  
Threads[16] , sum= 100000000, Exec_time =889 msecs  
계속하려면 아무 키나 누르십시오 . . .
```

`_asm lock add sum, 2; (release 모드)`

빨라졌다! 빨라지긴 했는데... 점점 느려진다. 왜 자꾸 이런 문제가 있는 건지 다음 시간에 계속 하자.