올바른 결과와 성능 향상이 동시에 만족되어야 하는데, 그게 힘들다. 올바른 결과를 위해서는 mutex를 사용해야 하지만, 그걸 사용하면 성능이 떨어지니까. 그러니까 성능 향상과 정확성 사이에서 빙글빙글 돌지 말고, 재작성해야 한다. 최소한의 mutex로 올바른 결과가 나올 수 있도록 프로그래밍해야 올바른 성능이 나오는 것이다.

재작성이란? mutex를 줄이자. 이것이 재작성이다. 어떻게 줄일 수 있을까? 데이터레이스를 줄이자. 데이터 레이스를 줄이면, mutex를 줄일 수 있다. 그런데 꼭 이렇게 해야 되느냐? 데이터 레이스가 있을 때 꼭 mutex를 써야 하는가? 데이터 레이스가 있어도 mutex를 안 쓰고 돌릴 수 있지 않은가? 데이터 레이스를 잘 관리하면 되지 않는가? 이런 야심 찬 생각을 가진 사람들이 있다. mutex 없이 데이터 레이스를 관리하자. 그럼 이제 거하게 뒤통수를 맞고 멀티스레드 프로그래밍을 포기하게 된다. 하나씩 실습해보자.

주의점 1. 컴파일러

아래 코드와 같은 멀티스레드 프로그램을 작성해보자. 이 프로그램은 어떤 문제가 있을까?

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

using namespace std;

int message;
bool ready = false;

void recv()
{
	while (false == ready);
	cout << "I got " << message << endl;
}

void send()
{
	message = 999;
	ready = true;
}

int main()
{
	thread reciver{ recv };
	thread sender{ send };

	reciver.join();
	sender.join();

}
```

이 프로그램은 무엇을 하는 내용인가? 두 개의 스레드가 동시에 돌아가는 프로그램이다. sender 스레드가 reciver 스레드에게 데이터를 넘겨주는 프로그램이다. 문제는 실행 순서를 알 수 없다. recv가 먼저 실행될수도, send가 먼저 실행될 수도 있다. 근데 recv가 먼저 실행되면 숫자가 999가 들어가지 않는다. 그럼 flag를 써서 이게 true일 때까지 기다렸다가, 데이터를 읽을 수 있다. 그럼 send에서 저장한 값이 1에 확실히 전달이 될 것이다.

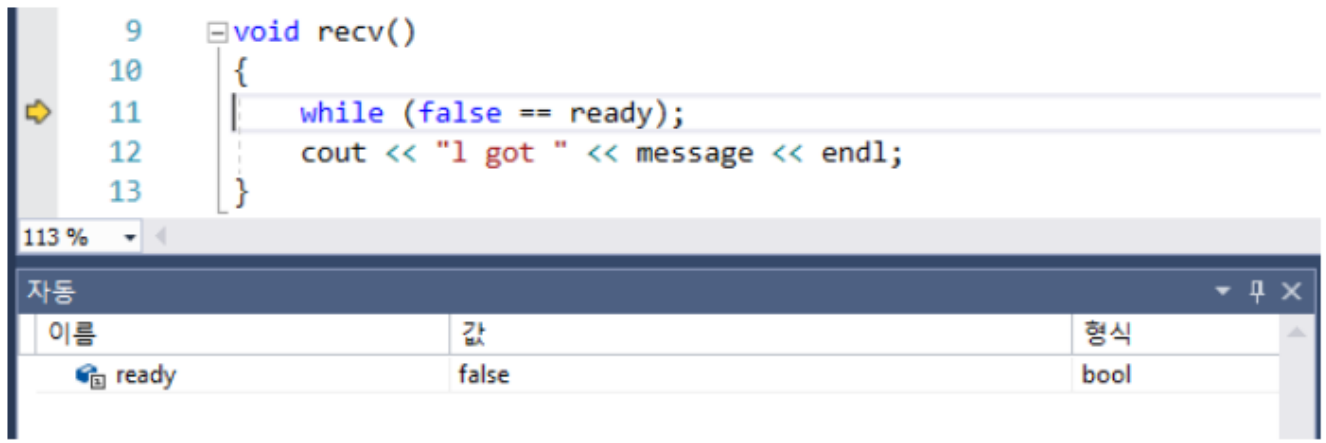
그럼 999가 뜨는가? 제대로 당연히 된다. 아주 간단한 두 줄짜리 프로그램이다.

debug 모드

그런데 잠깐, 디버그 모드로 하고 있진 않았는가?? 릴리즈 모드로 돌리면 ??

release 모드

결과가 나오지 않는다. 어디서 멈춰있는지 확인해보자.



reciver스레드가 여기에서 멈춰있다. ready의 값이 false여서 while문에서 멈춰있는 것이다. 왜 ready의 값이 false인가? send에서 분명 flag를 true로 해주었을 텐데. 그리고 분명 recv send 모두 다 했을 텐데. sender스레드가 뒤에 선언되어서 그런 거 아닌가? 그렇진 않다. send내부에 중단점을 걸고 돌리면 잘 불리는 걸 알 수 있다.

분명 true로 flag를 만들어 주었는데 계속 무한루프이다. 왜 true로 바꾸었음에도 무한루프냐? 원인을 알 수가 없다. 그래서 많은 사람들이 포기하게 된다. 왜 이게 false인지 알 수가 없어서. 그럼 어떻게 해야 하냐? 실제 cpu가 무슨 짓거리를 해서 false인지 확인해야 한다. 디스 어셈블리 창을 확인해보아야 한다.

```
void recv()
{
00E71040  mov         al,byte ptr [ready (0E75484h)]
    while (false == ready);
00E71045  test        al,al
00E71047  je          recv+5h (0E71045h)
    cout << "1 got " << message << endl;
```

디버그 - 창 - 디스어셈블리

이 과정은 무엇이냐? ready가 false이면 계속 무한루프를 돌아라. 그게 je이다. 이 명령은 jump이다. false이면 점프를 해라. 어디로? 0E71045로. 밑으로 내려가지 말고 저기로 점프해서 다시 값을 비교해라. al은 레지스터이니까, 0E71040으로 가서 ready의 값을 다시 읽어야 하는데, al에 적힌 새로운 true라는 값을 읽지 않고 이전에 읽었던 값인 false만 계속 읽으면서 무한루프를 돌고 있다.

al에 있는 값을 확인해보자. 진짜 ready는 true인가?? ready의 메모리를 읽어보면

```
메모리 1
주소: 0x00E75484
0x00E75484  01 00 00 00 e7 03 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

01로 true의 값이 들어가 있다.

그러나

```
레지스터
EAX = 00E71000 EBX = 00A5A100 ECX = 00A557D0 EDX = 00000000 ESI = 0073F944 EDI = 6C92DA00 EIP = 00E71045 ESP = 00F7F83C EBP = 00F7F85C EFL = 00000246
```

레지스터에는 여전히 00으로 false라고 저장되어 있다. **즉, 메모리에 적힌 새로운 값을 읽어와야 하는데 레지스터에 있는 예전 값을 계속 읽기 때문에 문제인 것이다.**

비주얼 스튜디오가 엉터리로 컴파일했다! ready를 넣고 비교하는 건 좋은데, false이면 새로 값을 읽어야지 옛날에 읽은 값만 계속해서 읽으면서 무한루프를 돌고 있으니까. 비주얼 스튜디오의 버그를 발견했다. ms에 편지를 보내야 한다. 만약 이렇게 해서 내가 짠 프로그램의 버그를 못 잡아서 손해가 100억이 났다? 그럼 손해배상 청구를 해야 한다. 그럼 어떻게 되는가? 재판이 벌어지고 우리가 지게 된다. 왜 우리가 질까? ms는 컴파일을 만들 때 잘못한 게 하나도 없기 때문이다.

왜 잘못한 게 하나도 없는가? c로 멀티스레드 프로그래밍하라고 누가 정했는가? 아무도 정하지 않았다. 그때 C를 만들 때 멀티스레드 프로그래밍이 없었다. 멀티스레드 프로그래밍하고 아무 상관없는 언어이다. 멀티스레드 프로그램에서 돌렸더니 이상하다? 그건 MS와 아무 관련이 없다. **C는 멀티스레드용 프로그래밍 언어가 아니다.** 이상하게 동작한다고 고소할 수 없다. 싱글 스레드 프로그램에선 이렇게 컴파일했을 때 아무 문제가 없었다. 싱글 스레드에서는 애초에 무한루프를 돌면 그 루프를 빠져나갈 수 있는 방법이 없다. 절대 빠져나갈 수 없다. 왜? 싱글 스레드에서 while 도중에 flag를 고칠 수 있는 방법이 없다.

그러니까 멀티스레드 프로그램에 C를 쓴 우리가 잘못이다. 그러니까 우리는 리눅스에 있는 gcc를 써야 한다? 아니다. 그것도 똑같이 컴파일한다. 똑같이 무한루프를 돈다. 컴파일의 문제가 아니고 **오히려 이렇게 컴파일 한 컴파일러가 더 잘한 거다. 아까 0E71040로 점프를 했다면, 메모리를 계속 건드리게 되지만 0E71045로 점프를 했기 때문에 메모리를 안 건드렸던 거다.** 다른 프로그램이 메모리를 읽고 쓰고 하는데 방해하지 않는다. 메모리가 놀기 때문에 열도 덜나고 다른 프로그램에 방해도 하지 않고, 훨씬 좋은 프로그램이다. 만약 이게 싱글 스레드 프로그램이었다면.

그래서 mutex 없이 프로그램하면 이렇게 간단한 프로그램 조차 잘 돌아가지 않는다. 소스코드 백날 쳐다봐도 왜 이게 잘못됐는지 절대 알 수 없다. 디스 어셈블러 해서 안을 들여다보기 전엔.

그래서 결론적으로, c는 멀티스레드 용 언어가 아니다. 컴파일러가 어떻게 컴파일할지 알 수 없고, 이상하게 컴파일해서 돌리면 이상한 결과가 나오던지 무한루프에 빠지게 되는 일이 일어난다. 그럼 어떡하지? 멀티스레드로 프로그래밍하지 않으면 된다. 그러나 꼭 성능의 문제로 반드시 이걸 해결해야만 한다면 어떡하는가? volatile를 사용한다.

- 컴파일러의 사기를 피하는 방법
  - volatile를 사용하면 된다.
    - 반드시 메모리를 읽고 쓴다
    - 변수를 레지스터에 할당하지 않는다
    - 읽고 쓰는 순서를 지킨다
  - 어셈블리를 모르면 visual studio의 사기를 알 수 없다

volatile를 사용하면 된다. 비주얼 스튜디오가 너무 잘 최적화시켜서 생긴 문제이다. 최적화 금지 키워드인 volatile를 넣어주어야 한다.

그럼 이 키워드를 사용해서 아까의 코드를 수정해보도록 하자.

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

using namespace std;

int message;
volatile bool ready = false; //여기에 volatile를 붙인다.

void recv()
{
	while (false == ready);
	cout << "I got " << message << endl;
}

void send()
{
	message = 999;
	ready = true;
}

int main()
{
	thread reciver{ recv };
	thread sender{ send };

	reciver.join();
	sender.join();

	system("pause");
}
```

그러면 999가 제대로 나온다. 0이 만약 나왔다? 그럼 로또 맞을 확률이다. 0이 나왔다면 그건 다른 이야기고, 어쨌든 volatile를 붙이면 해결이 된다.

그럼 message는 믿을 수 있나? 얘도 volatile를 하는 게 낫다. 메시지를 읽는 명령어를 while 밑에 둘지 위에 둘지 모른다. message를 읽는 기계어를 while위에 둘 수도 있다. 왜? 싱글 스레드에서는 message를 while 전에 읽나, 후에 읽나 문제가 안된다. 그러니 volatile를 읽어서 반드시 저 위치에서 읽으라고 명령을 해야 한다. 컴파일의 실행 순서가 중요하고, 데이터 읽고 쓰는 게 중요한 변수는 volatile를 쓰면 되는구나 하고 배우자.

그렇다면 정말 이게 전부인가? 그렇지 않다. 우리가 생각했던 것보다 조금 더 복잡하다.

```
struct Qnode {
	volatile int data;
	volatile Qnode* next;
};


void ThreadFunc1()
{
	...
	while ( qnode->next == NULL ) { }
	my_data = qnode->next->data;
	...
}
```

예를 들어 이런 코드가 있다고 해보자. 노드가 있고, data와 next가 있다. qnode란 linked list의 어떤 노드를 가리키고, 그 노드가 마지막 노드일 수도, 마지막 노드가 아닐 수도 있다. 만약 마지막 노드이다? 그럼 대기하고 있다가 마지막 노드가 아니면 다음 노드를 읽어라. 이런 식으로 프로그래밍할 수 있다. 논리적으로 아무 문제가 없다. next가 null일 때 읽으면 당연히 오류인데, null이 아니니까 아무 문제가 없지 않나? 그런데 우리는 이런 루프의 문제가 있다는 걸 알 수 있다. 처음에 한번 읽고 volatile 해주자. 그럼 아무 문제가 없지 않나? 아무 문제가 없었으면 예제가 아니었을 것이다. 이것도 무한루프에 빠진다. 옆에 스레드가 노드에 다른 노드를 끼워 넣어도 무한루프에서 빠져나오지 못한다. 왜? 한번 읽고 루프 돌면서 계속 새로 읽지 않고 옛날에 읽었던 값을 레지스터에 읽어놓고 그대로 쓰니까. volatile를 했는데 왜??? 대체 왜 메모리를 읽는 걸 생략해버리냐?? volatile를 제대로 하지 않아서 그렇다.

volatile의 사용법

- volatile int * a;
  - *a = 1; // 순서를 지킴
  - a = b; // 순서를 지키지 않는다.
- int * volatile a;
  - *a = 1; // 순서를 지키지 않음,
  - a = b; // 이것은 순서를 지킴

데이터가 volatile이라는 뜻. 뭐가? *a가.

a가 가리키는 게 volatile이라는 뜻이지 a가 volatile이라는 게 아니다. a라는 변수가 있을 때, 얘는 데이터를 갖고 있는 게 아니라 a라는 변수는 포인터다. 그러니까 어떤 다른 주소를 가리키고 있고, 이 주소는 int이다. 그러니까 그 가리키고 있는 int인 주소가 volatile이라는 것이다.

volatile이라는 int를 가리키는 포인터가 a이다. 그래서 volatile를 붙여도 해결이 안 된다. int 자체는 상관이 없다. a를 volatile로 만들어야 한다. 그럼 int* volatile a 이런 식으로 선언해줘야 한다. 둘 다 volatile로 해주고 싶다? 그럼 volatile int* volatile a로 선언해주어야 한다.

위의 리스트의 경우에도 node* volatile로 선언해야 한다. 그래야 제대로 된다.

특히 포인터를 사용했을 때는, 어느 것을 원하느냐에 따라 정확히 volatile 붙이는 위치를 잘 써야 한다. 잘 모르겠으면 volatile를 다 붙여줘야 한다.

컴파일러 문제 정리

- 여러 개의 스레드가 공유하는 변수는 volatile를 써야 한다.
- volatile를 사용하면 컴파일러는 프로그래머가 지시한 대로 메모리에 접근한다
- 하지만, cpu는 volatile를 모른다. (지금은 skip)

여러 개의 스레드가 공유하는 변수는 volatile를 써야 한다. 안 그러면 비주얼 스튜디오가 읽어야 하는데 읽는 걸 생략할 수 있고, 읽고 쓰고 할 때 위치를 제멋대로 바꿀 수 있다.

근데, mutex를 쓸 땐 그런 이야기가 없었는데? **mutex를 쓰면 volatile를 쓸 필요가 없다.** mutex를 쓰면 mutex를 만남과 즉시 새로 값을 읽는다. lock 다음 메모리 읽는 것이 있으면 반드시 읽는다. lock 안에 루프를 돈다? 그런 경우 새로 읽지 않는다. volatile하지 않으면. 근데 상관이 없다. **lock안에 들어왔으니까 다른 스레드가 그 안에 있는 변수들을 바꿀 수 없다.** 왜? data race 없으려고 lock을 건 거니까. 그런 경우 어떻게 컴파일하든 상관이 없다. 그러니 mutex를 써야 한다면, volatile를 쓸 필요가 전혀 없다. **mutex 없이 쓰겠다 하면 volatile를 써야 한다.** 이런 경우에는 프로그래밍할 때 굉장히 신경을 써야 한다. 컴파일 문제니까 volatile로 해결한다.

정리하자면, mutex를 쓰지 않고 데이터 레이스를 없애려면 컴파일러 문제가 생기는 사실을 알 수 있고, 엄밀히 말하면 c가 원래 그렇게 생겨서 생기는 문제다. 함부로 최적화하지 않게 주의해서 프로그래밍해야 한다.

주의점 2. CPU

- **상호 배제의 구현**
- **메모리 일관성 문제**

그렇구나 그럼 주의해야지 하고 끝일까? 또 다른 문제가 있다. 우리는 지금 mutex 오버헤드가 너무 커서 성능이 안 나오는 문제가 있다. mutex 없이 프로그램을 짜고 있는데 생기는 또 다른 문제는?

- 상호 배제의 구현
  - 멀티스레드 프로그램에서의 문제는 하나의 자원을 여러 스레드에서 동시에 사용해서 생기는 경우가 대부분.
  - 해결책 : 공유 자원을 업데이트하는 부분은 한 번에 하나의 스레드에서만 실행할 수 있도록 하자.
  - 이것을 상호 배제(mutual exclusion)라 부른다.

멀티스레드 프로그래밍을 하다 보면 상호 배제가 필요하다. 어떤 자원이 있다. 그럼 내가 file 업데이트를 할 때 다른 스레드가 와서 업데이트를 하면 엉망이니까, 내가 업데이트를 하면 넌 잠깐 쉬고 있고 내가 끝나면 업데이트해라. 이게 상호 배제이고 mutex이다. 그런데 성능 때문에 mutex를 쓰지 않고, 상호 배제를 자체적으로 구현해서 써야겠다 하고 마음을 먹게 된다. 그럼 밑에서 같이 mutex를 쓰지 않고 상호 배제하는 것을 구현해보자.

- 임계 영역
    - Critical Section
    - 프로그램 중 상호 배제로 보호받고 있는 구간
    - 오직 하나의 스레드만 실행할 수 있음
- 임계 영역 구현
    - lock & unlock을 사용해서 lock과 unlock사이에 임계 영역을 둔다.
    - lock은 다른 스레드가 lock을 통과했고 unlock을 하기 전이라면 unlock을 실행할 때까지 프로그램의 실행을 멈춘다.

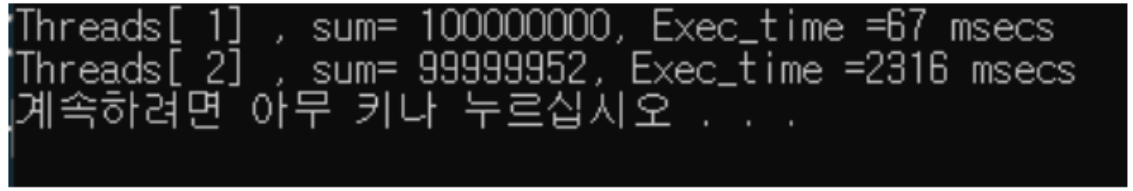
상호 배제를 한다면 크리티컬 섹션이 있어서 lock unlock으로 구현을 하는데 실제로 구현을 해보자. mutex가 오버헤드가 크니까 가만히 생각해보면 c++11에 mutex가 구현이 어떻게 되어있나 그걸 알 필요가 있다. 실제로 그걸 구현해보는 게 의미가 있겠다. 그러니 c++11의 라이브러리를 사용하지 않고 직접 구현해보자. 최대한 단순하게, 오버헤드가 작게 쓸 수 있지 않을까

- 실제로 구현해보자
    - c++11의 라이브러리를 사용하지 않고
    - 최대한 단순하고 가볍게
- lock & unlock의 구현
    - 공유 메모리를 통해서 구현한다
    - 여러 가지 알고리즘이 있다
        - 피터슨, 데커, 빵집...
- 피터슨 알고리즘으로 구현해보자
    - 2개의 스레드 사이의 lock과 unlock을 구현하는 알고리즘
    - 매개변수로 스레드 아이디를 전달받으며 값은 0과 1이라고 가정
    - 운영체제 시간에 배우고 유명한 알고리즘이다.

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

using namespace std;
using namespace chrono;

volatile int sum;
mutex mylock;

volatile int victim = 0;
volatile bool flag[2] = { false, false };

//피터슨 락과 언락으로 구현하기
void p_lock(int myid)
{
	int other = 1 - myid;
	flag[myid] = true;
   victim = myid;
	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);
		//mylock.lock( );
		sum += 2;
		//mylock.unlock();
		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";
	}

}
```

빵집 알고리즘이 복잡하니 피터슨 알고리즘을 써보자. 이건 간단하고 굉장히 유명하다. 왜 간단하냐? 스레드가 2개일 경우에만 돌리자. 스레드가 2개면 알고리즘이 굉장히 심플해진다. lock unlock이 있으면 volatile 해주어야 하고, 그렇지 않으면 무한루프에 빠진다. 그러니까 volatile 해주고, 동작은 어떻게 되냐?

한 스레드 id가 0이고, 다른 스레드는 id가 1이다. 내 아이디가 1이면 상대방은 0, 내가 0이면 상대방은 1. 상대방 아이디를 얻고, 내가 임계 영역에 들어가려 한다. lock걸기 위해 플래그를 세팅하고, 상대방이 들어가고자 하는지를 봐야 한다. 이게 true면 상대방도 들어가려고 하고 있다는 것. 그러면 기다리고, 상대방이 unlock 하길 기다리고. 그러면서 상호 배제를 해야 한다. 문제는 동시에 lock을 하려고 하면 flag [0], flag [1] 모두 true이다. 이러면 무한루프에 빠지게 된다. 그래서 victim이라는 변수를 주었다. 만약 둘 다 true인 경우라 하더라도, 둘다 무한루프에 빠지지 않고 한 명은 빠져나오게 하자. 그게 이 희생자라는 것이다. 내가 희생자다? 그럼 나는 계속 루프를 돌지만, 희생자가 아닌 스레드는 빠져나오자. 그러니까 상호 배제도하고 무한루프도 되지 않는 것이다. 이해가 안 되면 계속 들여다보면 언젠가 이해가 될 것이다. 이 알고리즘은 아무 문제가 없다. 이걸 사용해서 제대로 돌아가도록 같이 해보자.



빈번한 메모리 참조로 성능도 안나오고, 1억도 제대로 안나오는 오동작을 일으켰다.

결과가 1억이 안 나오고, 실행시간도 확 떨어진다. 이게 mutex를 쓸 때보다 성능이 안 좋다. 이게 뭐냐? 1억이 안 나오는 것도 놀랍지만 속도도 느리다. 프로그램에 버그가 있는 거다.

일단 1억이 왜 안 나오냐? p_lock부분은 문제가 없다. 근데 lock내부의 알고리즘이 문제다. 아까 알고리즘은 문제가 없다고 했었는데? 교과서에도 나온 알고리즘이고, 이걸 c로 옮겼는데 옮길 때 뭔가 실수한 게 있나? 없다. 원래 알고리즘과 c로 구현한걸 비교해봐도 틀린 것이 없다. 괜히 헛수고 할까봐 이야기하지만 알고리즘으로 옮길때 실수는 없었다. 이게 두 번째 문제다. 첫 번째는 컴파일러가 마음대로 최적화하는 최적화 문제, 두 번째는 cpu 최적화 때문에 일어나는 문제. cpu가 프로그램을 엉터리로 수행했기 때문에. 그래서 1억이 안 나온 것이다.

일단 속도는, 피터슨 알고리즘으로 했을 땐 2.3초 정도. 그리고 mutex로 lock, unlock 했을 땐 1.7초 정도였다. 다 쓰면 안 된다. 다 제대로 된 해결책이 아니다. 이런 속도 차이가 왜 나는지는 알 필요 없고, 대학원을 가서 배워야 한다. 설명하는데 시간이 오래 걸린다. 1억이 안 나오는 게 일단 제일 문제이다.

오동작 원인은 cpu가 제대로 돌아가지 않기 때문에. 그럼 이번엔 인텔을 고소해야 하는가? 아니다. mutex 없이 데이터 레이스 놔두고 프로그램 돌리는 건 알고리즘이 아무리 제대로 되어있다 하더라도 위험하다. cpu에 대해서는 다음장에서 배우도록 하자.