



멀티스레드 2

MM4220 게임서버 프로그래밍

정내훈

내용

- 복습
- 간단한 동기화
- 메모리 일관성
- Lock 없는 프로그램
- CAS
- Non Blocking 알고리즘

복습

- Thread 2개로 합계 1억을 만드는 프로그램

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

using namespace std;
int sum;

void thread_func()
{
    for (auto i = 0; i < 25000000; ++i) sum += 2;
}

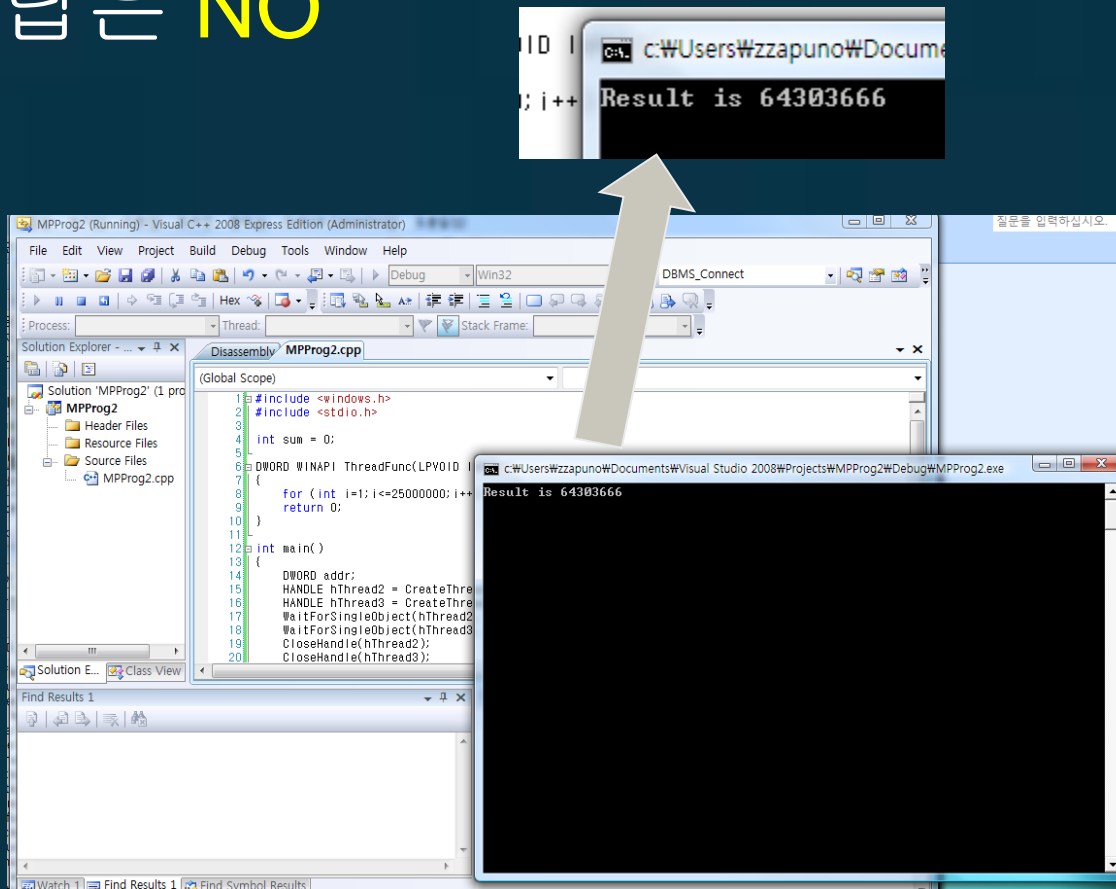
int main()
{
    thread t1 = thread{ thread_func };
    thread t2 = thread{ thread_func };

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

    cout << "Sum = " << sum << "\n";
}
```

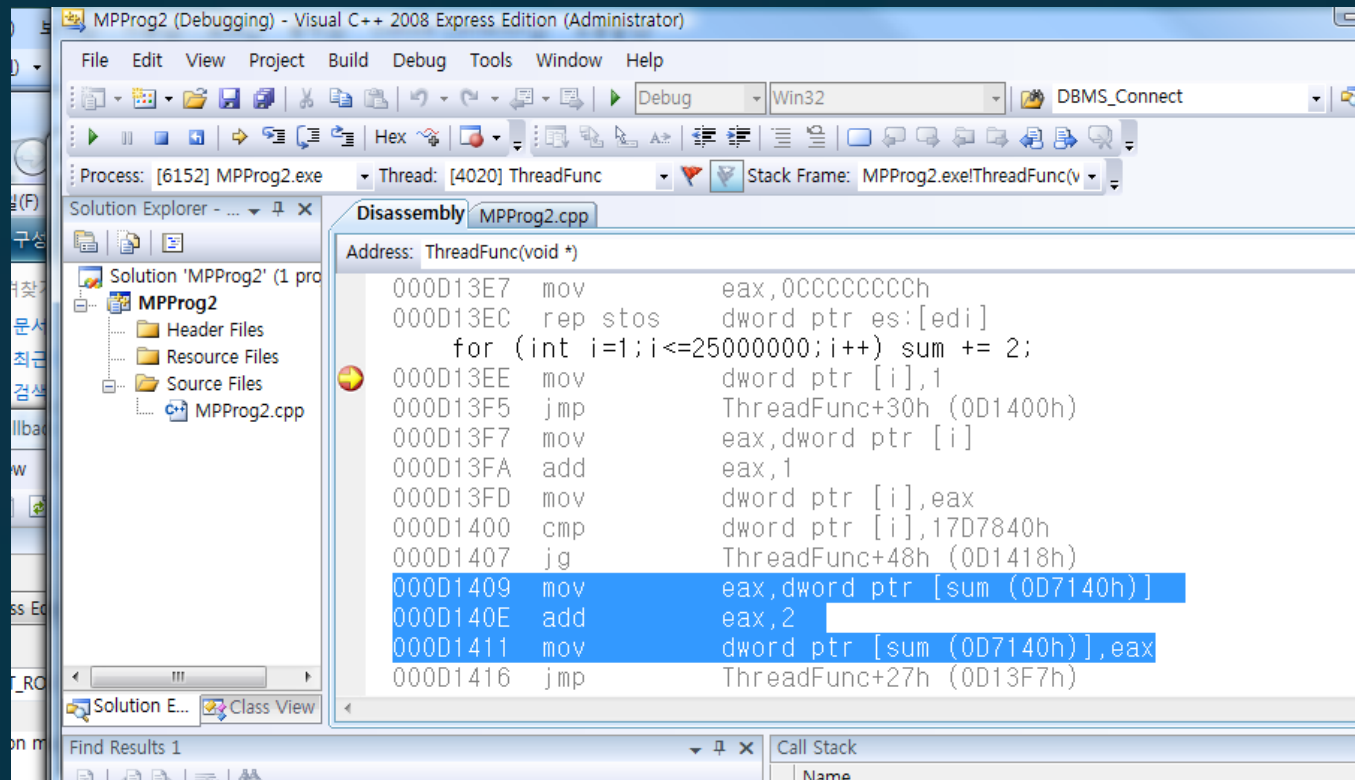
복습

- 쉬운가?
- 대답은 **NO**



복습

- 왜 틀린 결과가 나왔을까?
– “sum+=2”가 문제이다.



복습

- 왜 틀린 결과가 나왔을까?
 - **DATA RACE** (공유 메모리에 WRITE한다.)
 - “sum+=2”가 문제이다.

쓰레드 1

쓰레드 2

MOV EAX, SUM

ADD EAX, 2

MOV SUM, EAX

MOV EAX, SUM

ADD EAX, 2

MOV SUM, EAX

sum = 200

sum = 200

sum = 202

sum = 202

내용

- 복습
- 간단한 동기화
- 메모리 일관성
- Lock 없는 프로그램
- CAS
- Non Blocking 알고리즘

지금까지

- Data Race가 모든 문제의 근원이다.
- Data Race만 조심하면 싱글쓰레드 프로그래밍과 다를 바가 없다.
- Data Race를 없애는 가장 직접적인 방법은
 - Lock을 사용
 - 속도저하 문제
 - Data Race를 없애도록 프로그램을 변경
 - 변경은 힘든 작업이다.
 - Data Race를 완전히 없애는 것은 힘들다.

성능 향상

- Lock을 회피하는 프로그래밍
 - Data Race를 줄여서 Lock의 필요성 자체를 줄인다.
 - 아무 문제 없다.
 - 효과가 매우 크다.
 - **어렵다!**
 - Data Race가 있지만 전후 사정을 잘 파악해서 Lock을 넣지 않아도 잘 수행되도록 프로그래밍 한다.
 - 효과가 있다.
 - 많은 **함정**들이 있다.
 - Lock의 구현은 이 방법 밖에 없다.

간단한 동기화 (2023 화목)

- 동기화 (Synchronization)
 - 스레드 끼리 데이터를 주고 받거나
 - 실행 순서를 맞추는 행위
 - 협업을 위해 필수
- 동기화 구현
 - 공유 메모리를 통해 정보를 주고 받음.
 - 당연히 Data Race이니 mutex 필요.
- 우리의 시도
 - 정말 간단한 동기화에 mutex가 꼭 필요한가?

간단한 동기화

- 출발 : 간단한 동기화

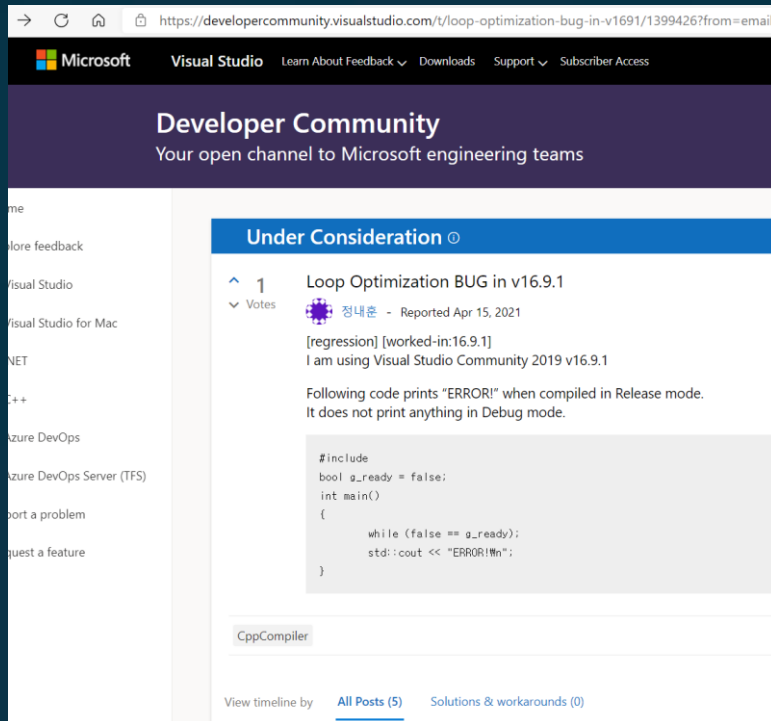
```
bool flag = false;
int send_data;

void Receiver()
{
    while(flag == false);
    cout << send_data;
}
```

```
void Sender()
{
    send_data = 1;
    flag = true;
}
```

간단한 동기화

● 비주얼 스튜디오 버그



Karen Huang [MSFT]

Under Investigation

...

Thanks for taking the time to report this issue to us. We've filed a bug for this issue on the C++ team here. The status on this Developer Community item will be updated as that bug is looked at. Thanks again for reporting this to us.

0

Apr 16, 2021

간단한 동기화

● Mutex를 사용한 해결.

```
bool flag = false;
int send_data;
mutex a;

void Receiver()
{
    a.lock();
    while (false == flag) { a.unlock(); a.lock(); }
    cout << send_data;
    a.unlock();
}
```

```
void Sender()
{
    a.lock();
    send_data = 999;
    flag = true;
    a.unlock();
}
```

간단한 동기화

- Visual Studio의 사기를 피하는 방법

- volatile을 사용하면 된다.

- 반드시 메모리를 읽고 쓴다.
- 변수를 레지스터에 할당하지 않는다.
- 읽고 쓰는 순서를 지킨다.

- 참 쉽죠?

- “어셈블리를 모르면 Visual Studio의 사기를 알 수 없다” 흠좀무...

간단한 동기화

● volatile을 사용한 해결???

```
volatile int *send_data = nullptr;

void Receiver()
{
    while (nullptr == send_data);
    cout << *send_data;
}
```

```
void Sender()
{
    send_data = new int {999};
}
```

고생길

- volatile의 사용법

- volatile int * a;

- *a = 1; // 최적화로 인한 오동작을 하지 않음
 - a = b; // 최적화로 인한 메모리 접근 생략, 순서 변경가능

- int * volatile a;

- *a = 1; // 최적화로 인한 메모리 접근 생략, 순서 변경가능
 - a = b; // 최적화로 인한 오동작을 하지 않음

간단한 동기화

- 아래의 Lock과 Unlock이 동작할까?
 - 피터슨 알고리즘 (운영체제 교과서에 실려 있음)
- 실행해 보자
- 결과는?

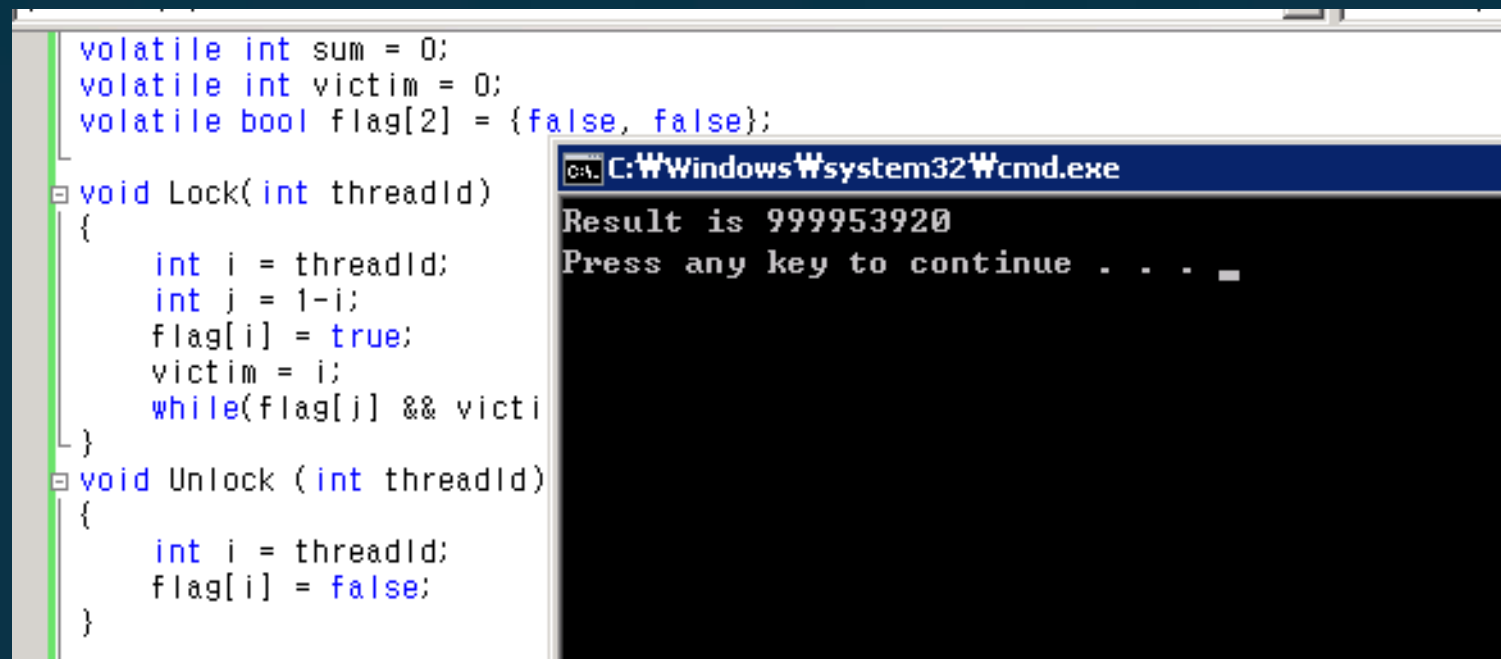
```
volatile int victim = 0;
volatile bool flag[2] = {false, false};

Lock(int myID)
{
    int other = 1 - myID;
    flag[myID] = true;
    victim = myID;
    while(flag[other] && victim == myID) {}
}

Unlock(int myID)
{
    flag[myID] = false;
}
```

Lock의 구현

- 이유는?



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

void Lock(int threadId)
{
    int i = threadId;
    int j = 1-i;
    flag[i] = true;
    victim = i;
    while(flag[j] && victim == j)
    {
        // Busy wait
    }
}

void Unlock(int threadId)
{
    int i = threadId;
    flag[i] = false;
}

int main()
{
    // ... (thread creation and execution) ...
    cout << "Result is " << sum << endl;
    system("pause");
}
```

C:\Windows\system32\cmd.exe

Result is 999953920

Press any key to continue . . .

메모리 일관성

- 피터슨 알고리즘의 오류?
 - 컴파일러는 문제 없다.

```

Lock(int threadId)
{
    int other = 1 - myID;
    flag[myID] = true;
    victim = myID;
    while(flag[other] && victim == myID) {}
}

```

```

                                Lock(threadid);
01281014  mov                byte ptr flag (1283378h)[eax],1
0128101B  mov                dword ptr [victim (1283374h)],eax
01281020  mov                bl,byte ptr flag (1283378h)[esi]
01281026  test               bl,bl
01281028  je                ThreadFunc2+34h (1281034h)
0128102A  mov                ebx,dword ptr [victim (1283374h)]
01281030  cmp                ebx,eax
01281032  je                ThreadFunc2+20h (1281020h)

```

메모리 일관성

- 이유는?
 - CPU는 사기를 친다.
 - Out of order execution
 - write buffering
 - CPU는 프로그램을 순차적으로 실행하는 척만한다.
 - 싱글코어에서는 절대로 들키지 않는다.
 - 이를 메모리 일관성 문제(Memory Consistency Problem) 이라고 한다.

메모리 일관성

- 정말? -> 확인해 보자 -> 실행 순서의 역전을 강제로 막아보자.
- 다음의 명령을 중간에 삽입

```
_asm mfence
```

또는

```
std::atomic_thread_fence(std::memory_order_seq_cst);
```

- 메모리 접근 순서를 강제하는 명령어
- 앞의 명령어들의 메모리 접근이 끝나기 전까지 뒤의 명령어들의 메모리 접근을 시작하지 못하게 한다.

메모리 일관성

- Out-of-order 실행

```
a = b * c * d * e;  
f = 3;
```

f = 30이 먼저 종료

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

c = d 가 먼저 종료

Out-of-Order(CPU)

● CPU의 발전

— Out of Order (Pentium Pro)

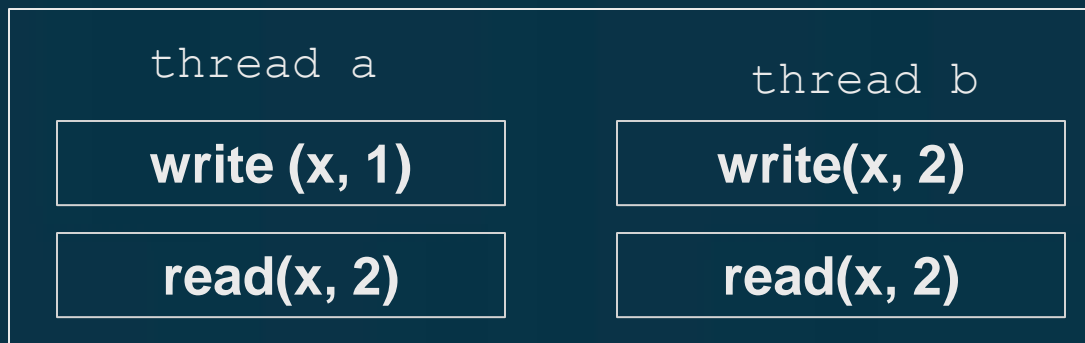


메모리 일관성

- 문제는 메모리
 - 프로그램 순서대로 읽고 쓰지 않는다.
 - volatile로도 해결되지 않는다.
 - volatile 키워드는 기계어로 번역되지 않는다.
 - 읽기와 쓰기는 시간이 많이 걸리므로.
 - CPU의 입장에서 보면
 - 실제 메모리 접근 시간은 상대 core에 따라 다르다.
 - 옆의 프로세서(core)에서 보면 뒤바뀐 순서가 보인다.
 - 자기 자신은 절대 알지 못한다.
- 어떠한 일이 벌어지는가?

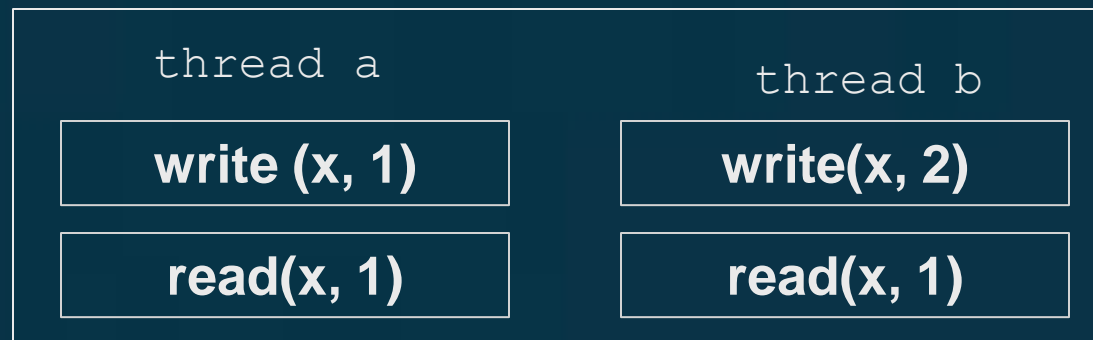
메모리 일관성

- 아래의 두 개의 실행결과는 서로 다르다
어떠한 것이 정확한 결과인가?



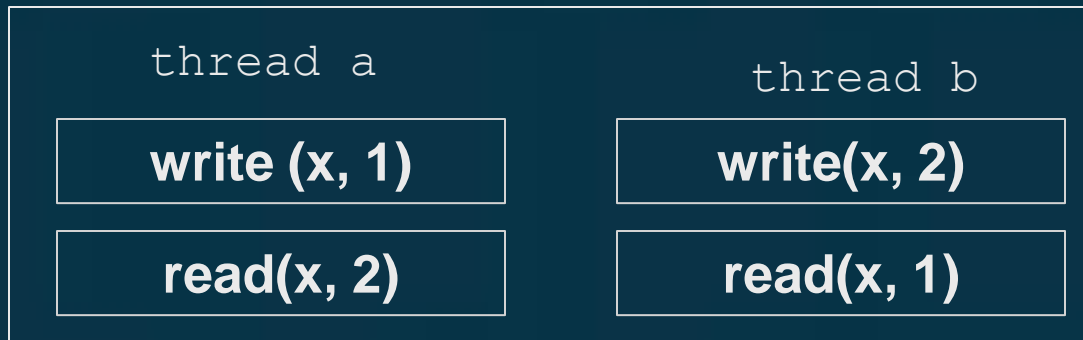
Type-A

Type-B



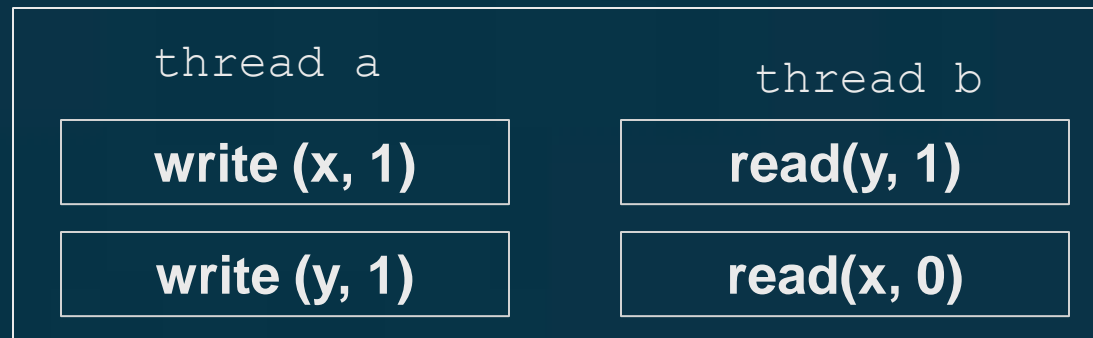
메모리 일관성

- 그러면 이것은?



Type-C!!

Type-D !!



메모리 일관성

- 그러면 이것은?

thread a

write (x, 1)

read (y, 0)

thread b

write(y, 1)

Read(x, 0)

Type-E

Type-F

thread a

write (x, 1)

thread b

write (x, 2)

thread c

read(x, 1)

read(x, 2)

thread d

read(x, 2)

read(x, 1)

메모리 일관성

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

메모리 일관성

● 메모리 일관성 테스트 2

```
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 ++;
    }
}
```

메모리 일관성

- bound를 다음과 같이 세팅

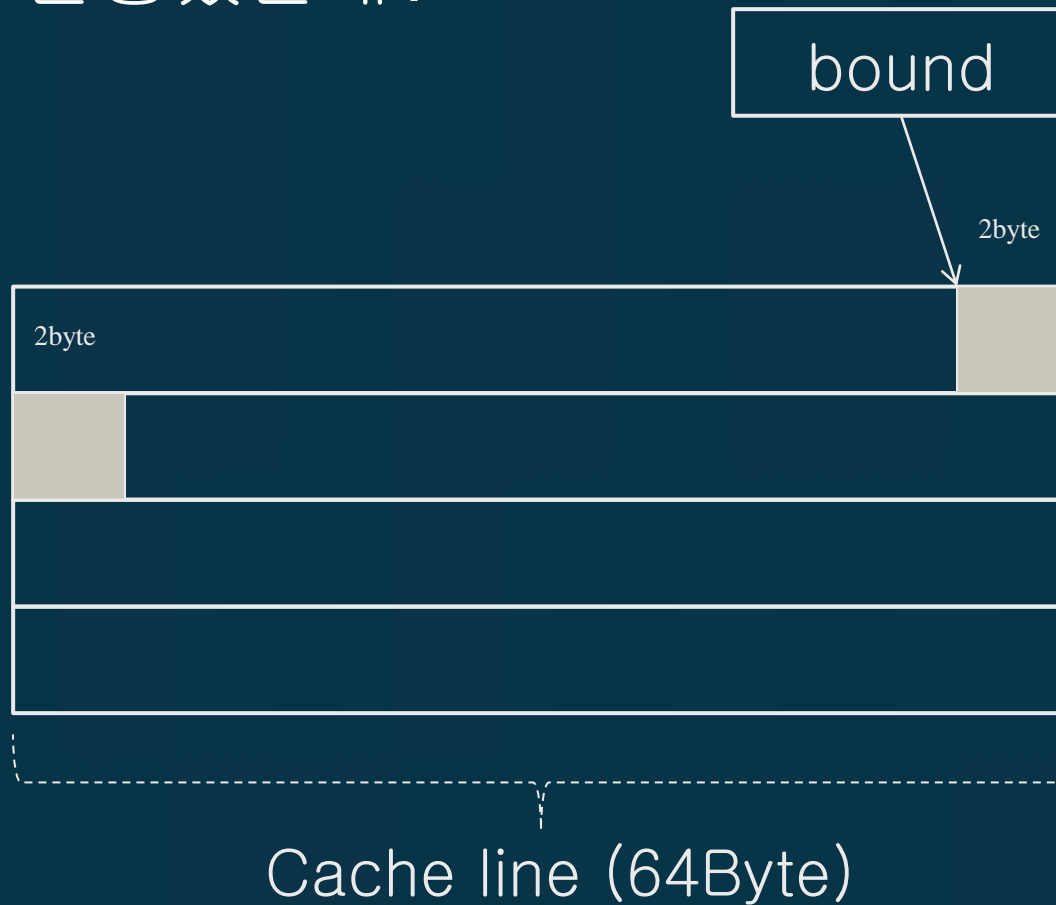
```
int a[32];  
long long addr = reinterpret_cast<long long>(&a[31]);  
addr = (addr / 64) * 64;  
addr = addr - 2;  
bound = reinterpret_cast<int *>(addr);  
*bound = 0;
```

<쓰레드 생성>

```
cout << "Result is " << error << "\n";
```

HELL

- 어떻게 실행했길래?



메모리 일관성

- 메모리 일관성 테스트 2 실행 결과

- 중간값

- write시 최종값과 초기값이 아닌 다른 값이 도중에 메모리에 쓰이는 현상
 - Read 도중 다른 스레드가 값을 변경

- 이유는?

- Cache Line Size Boundary
 - 확인 : -2를 -1, -3, -4로 교체

- 대책은?

- Byte 밖에 믿을 수 없다.
 - Pointer가 아닌 변수는 Visual C++가 잘 해준다.
 - Pointer를 절대 믿지 마라.
 - #pragma pack을 조심하라

메모리 일관성

- PC에서의 공유 메모리
 - 다른 코어에서 보았을 때 업데이트 순서가 틀릴 수 있다.
 - 메모리의 내용이 한 순간에 업데이트 되지 않을 때 도 있다.
- 그래도 희망적
 - 언젠가는 메모리에 대한 쓰기가 실행 된다.
 - 자기 자신의 프로그램 순서는 지켜진다.
 - 캐시의 일관성은 지켜진다.
 - 한번 지워졌던 값이 다시 살아나지는 않는다.
 - 언젠가는 모든 코어가 동일한 값을 본다
 - 캐시라인 내부의 쓰기는 중간 값을 만들지 않는다.

메모리 일관성

● 메모리 일관성 사례

Memory ordering in some architectures ^{[1][2]}

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 일관성

- 해결책
 - 이러한 상황을 고려하면서 프로그래밍을 한다.
 - 가능하지만 프로그래밍이 너무 어려워진다.
 - Lock()을 넣어서 순서의 어긋남을 제거한다.
 - 프로그램이 너무 느려진다.
 - Lock()의 구현은??
 - 문제가 되는 부분 어긋남을 제거한다.
 - atomic_thread_fence()의 사용
 - 어렵다. 남발 시 성능 저하
 - 기존의 멀티쓰레드용 라이브러리를 사용한다.
 - C++11 <atomic>

메모리 일관성

- 메모리 일관성 문제를 없애는 법
 - Atomic 메모리사용
- Atomic
 - 접근(메모리는 read, write)의 절대 순서가 모든 스레드에서 지켜지는 메모리
 - 대부분의 알고리즘이 Atomic 메모리를 바탕으로 한다.
 - 싱글코어에서는 모든 메모리가 Atomic Memory이다.

Atomic

- Atomic한 int를 만들 수 있는가?
 - SW적으로 구현할 수 있다.
 - 자세한 내용은 생략, 대학원 과정
 - 복잡, 고비용
 - HW적으로 구현되어 있다.
 - CPU에 특수 명령어로 구현되어 있다.
 - C++11의 `atomic<>` 템플릿을 사용해도 된다.

```
#include <atomic>
atomic <int> sum;
```

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

Atomic

- Atomic int 만 있으면 되는가?
 - NO
 - 예) “map <int, SESSION> players”
 - 실제 상용 프로그램을 int, long, float같은 기본 data type 만으로 작성되지 않는다.
- 실제 프로그램은 기본 data type을 사용하여 다양한 자료구조를 구축하여 사용한다.
 - 특히 쓰레드 사이의 자료 공유에서.
 - queue, stack, binary tree, vector...

Atomic

- Atomic 자료 구조
 - Atomic Memory를 사용하여 자료구조를 만들면 Atomic한가?
 - NO
 - 예) `sum = sum + 2;`
- 효율적인 Atomic 자료구조가 필요하다.
 - Lock으로 Atomic 자료구조를 만들면
 - 병렬성이 떨어진다.
 - 너무 느리다.
- 근데 “효율적인” 이라니?

Lock없는 프로그램

- 효율적인 자료구조의 구현
 - Lock없는 구현
 - 성능 저하의 주범이므로 당연
 - Lock이 없다고 성능저하가 없는가??
 - 상대방 쓰레드에서 어떤 일을 해주기를 기다리는 한 동시실행으로 인한 성능 개선을 얻기 힘들다.
 - while (other_thread.flag == true);
 - lock과 동일한 성능저하
 - 상대방 쓰레드의 행동에 의존적이지 않는 구현방식이 필요하다.

Non-Blocking

- 블럭킹 (blocking)
 - 다른 쓰레드의 진행상태에 따라 진행이 막힐 수 있음
 - 예) `while(lock != 0);`
 - 멀티쓰레드의 bottle neck이 생긴다.
 - Lock을 사용하면 블럭킹
- 논블럭킹 (non-blocking)
 - 다른 쓰레드가 어떠한 삽질을 하고 있던 상관없이 진행
 - 예) 공유메모리 읽기/쓰기, `atomic_int` 에서 `'+='`

Non-Blocking

- 블럭킹 알고리즘의 문제
 - 성능저하: lock 오버헤드, 병렬성 저하
 - Priority Inversion
 - Lock을 공유하는 덜 중요한 작업들이 중요한 작업의 실행을 막는 현상
 - Reader/Write Problem에서 많이 발생
 - Convoying
 - Lock을 얻은 스레드가 스케줄링에서 제외된 경우, lock을 기다리는 모든 스레드가 공회전
 - Core보다 많은 수의 thread를 생성했을 경우 자주 발생.
- 성능이 낮아도 Non-Blocking이 필요할 수 있다.

Non-Blocking

- 다른 해설
 - Non-Blocking자료 구조는 여러 개의 method를 갖고 있는 class 이다.
 - 여러 스레드에서 동시에 method를 호출 해도 오류 없이 동작 한다. (= Atomic 하다)
 - 이것은 lock을 사용했을 때와 같은 결과가 나온다.
 - 다른 스레드에서 동시에 method를 호출했다고 해서 속도가 느려지지 않는다. (wait-free)
 - 속도가 좀 느려질 수 있지만, 어느 스레드가 멈추었다고 해서, 다른 스레드가 멈추는 경우는 절대 없다. (lock-free)

Non-Blocking

- 언블럭킹의 등급

- 무대기 (wait-free)

- 모든 메소드가 정해진 유한한 단계에 실행을 끝마침
- 멈춤 없는 프로그램 실행

- 무잠금 (lock-free)

- 항상, 적어도 한 개의 메소드가 유한한 단계에 실행을 끝마침
- 무대기이면 무잠금이다
- 기아(starvation)을 유발하기도 한다.
- 성능을 위해 무대기 대신 무잠금을 선택하기도 한다.

Non-Blocking

- 정리

- Wait-free, Lock-free

- Lock을 사용하지 않고
 - 다른 스레드가 어떠한 행동을 하기를 기다리는 것 없이
 - 자료구조의 접근을 Atomic하게 해주는 알고리즘의 등급

- 멀티 스레드 프로그램에서 스레드 사이의 효율적인 자료 교환과 협업을 위해서는 Non-Blocking 자료 구조가 필요하다.

Non-Blocking

- Atomic Memory로 Non-blocking 자료구조를 만들 수 있는가?
 - Queue, map, stack...
- 아니다.
 - 공유 메모리를 읽고 쓰면서 동기화하는 방식으로 구현할 수 없다
- Atomic Memory만으로는 다중 스레드 무대기 큐를 만들 수 없다!!!!!!
 - (증명) : 멀티 코어 프로그래밍 과목

CAS

- 다중 쓰레드 무대기 큐를 만들려면?
 - CAS 연산이 필요하다.
 - 기존의 공유메모리는 Read와 Write 연산만 있었다.
- CAS?
 - CompareAndSet
 - Atomic하고 Wait-Free하게 동작
 - Read와 Write만으로는 절대로 만들 수 없음
 - HardWare이 도움이 필수!!!

CAS

- CAS?

- MEM.CAS(A,B);

- MEM의 값이 A면 MEM의 값을 B로 바꾸고 true를 리턴
 - MEM의 값이 A가 아니면 아무것도 하지 않고 false를 리턴

CAS

- 실제 HW (CPU) 구현

- `lock cmpxchg [A], b` 기계어 명령어로 구현

- `eax`에 비교값, `A`에 주소, `b`에 넣을 값

```
if (eax == [a]) {  
    ZF = true;  
    [a] = b;  
} else {  
    ZF = false;  
    eax = [a];  
}
```

CAS

- 실제 CAS의 구현 : C++11
 - Atomic_compare_and_set은 없고
atomic_compare_exchange를 대신 사용

```
bool CAS(atomic_int *addr, int expected, int new_val)
{
    return atomic_compare_exchange_strong(
        addr, &expected, new_val);
}
```

```
bool CAS(volatile int *addr, int expected, int new_val)
{
    return atomic_compare_exchange_strong(
        reinterpret_cast<volatile atomic_int *>(addr),
        &expected, new_val);
}
```

CAS (생략)

- Windows API를 사용한 구현
 - `int y = Interlockedcompareexchange(&X, a, b);`
 - X의 값이 b이면 X의 값을 a로 바꾼다.
 - X의 값의 b가 아니면 X의 값을 바꾸지 않는다.
 - X에 원래 들어 있던 값을 리턴한다.
 - a,b의 순서에 주의
 - true/false를 얻으려면 y와 b를 비교하면 된다.

CAS

- CAS로 Lock의 구현

- 0으로 초기화 되어 있는 공유 메모리 X가 있을 때. 모든 스레드에서

```
CAS (&X, 0, 1);
```

- 를 실행 시키면 오직 하나의 스레드만 X가 0이 되면서 true를 리턴한다.

```
void Lock(atomic_int *x)
{
    while(false == CAS(&X, 0, 1));
}
void Unlock(atomic_int *x)
{
    *x = 0;
}
```

CAS

- 역사

- 1970년대 x86 CPU에 lock cmpxchg 명령 존재
- 선견지명?
- SW만으로 구현된 lock(), unlock()이 비효율적이므로 간단하게 lock()을 구현할 명령어 추가
 - 싱글스레드 프로그램도 lock 필요
 - 멀티 프로세스 사이의 동기화.
 - 당시에 멀티 CPU 기능이 있었음.
 - 하지만, 잘 못 사용하면 엄청난 성능 저하.

- 구현

- 파이프라인을 멈추고, 다른 Core와 CPU에서 같은 주소의 메모리 접근을 하지 못하도록 막고 실행
- 다른 Core와 CPU에서도 메모리의 접근이 제한됨
- 엄청난 성능 페널티 (= 민폐)

CAS

- CAS_Lock 성능
 - i7 quad core CPU
 - 단위 ms

#of threads	data race	mutex	atomic_int	CAS_lock
1	79	1471	270	630
2	62	1219	506	4221
4	48	1353	762	8025
8	54	1678	794	13535
16	64	1626	794	24981

CAS

- CAS의 위엄

CAS를 사용하면 기존의 모든 자료구조를 멀티쓰레드 Wait-free 자료구조로 변환할 수 있다.

- 증명 가능 (멀티코어 프로그래밍 과목)
- 간단한 변환 알고리즘 존재

CAS

- “간단한 변환 알고리즘”??
 - 존재하지만 너무 비효율적이다.
 - 100배 이상의 성능 저하.
- 각각의 자료구조에 맞추어 최적화를 해야한다.
 - 최적화는 힘들지만, 고성능이 가능하다.
- 이미 최적화 되어있는 라이브러리를 쓰는 것이 좋다.
 - visual studio 2010: PPL (Parallel Patterns Library)
 - Intel : TBB(Thread Building Block)

재정의

- 고성능 멀티 쓰레드 프로그래밍
 - 비멈춤(non-blocking) 자료구조를 사용한 프로그래밍
 - Lock은 사용하지 않지만 CAS는 필요하다.
(CAS 없이는 만들 수 없음이 증명되어 있다.)

LF-Queue의 구현

● 예) Lock-Free queue의 구현 (1/3)

```
int CAS(Node **addr, Node *old_v, Node *new_v)
{
    return atomic_compare_exchange_strong(
        reinterpret_cast<atomic_int*>(addr),
        reinterpret_cast<atomic_int*>(&old_v),
        reinterpret_cast<int>(new_v));
}

class LFqueue {
private:
    Node *head, *tail;
public:
    LFqueue() {
        head = tail = new Node(0);
    }
}
```

LF-Queue의 구현

● 구현 (2/3)

```
void enq(int x) {
    Node *e = new Node(x);
    while (true) {
        Node *last = tail;
        Node *next = last->next;
        if (last != tail) continue;
        if (nullptr == next) {
            if (true == CAS(&(last->next), nullptr, e)) {
                CAS(&tail, last, e);
                return;
            }
        }
        else CAS(&tail, last, next);
    }
}
```

LF-Queue의 구현

- 구현 (3/3)

```
int deq(int x) {  
    while (true) {  
        Node *first = head;  
        if (nullptr == first->next) EMPTY();  
        if (!CAS(&head, first, first->next))  
            continue;  
        int value = first->next->item;  
        safe_delete(first);  
        return value;  
    }  
};
```

LF-Queue의 구현

● 무엇이 바뀌었는가?

싱글 쓰레드 Queue

```
QUEUE::push(int x) {
    Node *e = new Node(x);
    tail->next = e;
    tail = e;
}
```



멀티쓰레드 Atomic Queue

```
QUEUE::push(int x) {
    Node *e = new Node(x);
    q_lock.lock();
    tail->next = e;
    tail = e;
    q_lock.unlock();
}
```



LF-Queue

```
LF_QUEUE::push(int x) {
    Node *e = new Node(x);
    while (true) {
        Node *last = tail;
        Node *next = last->next;
        if (last != tail) continue;
        if (nullptr != next) continue;
        if (CAS(&(last->next), nullptr, e,
                &tail, last, e)) return;
    }
}
```

LF-Queue의 구현

```

LF_QUEUE::push(int x) {
    Node *e = New_Node(x);
    while (true) {
        Node *last = tail;
        Node *next = last->next;
        if (last != tail) continue;
        if (NULL != next) continue;
        if (CAS(&(last->next), NULL, e,
            &tail, last, e)) return;
    }
}

```

하지만 2개의 변수에 동시에
CAS를 적용할 수는 없다!

실제 구현

```

LF_QUEUE::push(int x) {
    Node *e = New_Node(x);
    while (true) {
        Node *last = tail;
        Node *next = last->next;
        if (last != tail) continue;
        if (NULL == next) {
            if (CAS(&(last->next), NULL, e)) {
                CAS(&tail, last, e);
                return;
            }
        } else CAS(&tail, last, next);
    }
}

```

성능 비교

- 정렬된 리스트 삽입/삭제/검색 프로그램
성능 비교 (단위 초)
— XEON 2 CPU 8Core

쓰레드 개수	Lock 사용	Lock-Free
1	2.59	2.30
2	4.36	1.76
4	4.88	1.61
8	5.01	0.83
16	4.92	1.10

리스트의 구현

- 성능 비교 예 (단위 ms)
 - XEON E5-4620 (8 core X 4 CPU)

	1	2	4	8	16	32	64
Lock	2203	5969	7672	9547	22970	22829	22861
Lock-Free	2390	2062	1922	781	359	1125	259

Visual Studio PPL

- 병렬 컨테이너

- 종류

- Concurrent_vector
 - Concurrent_queue
 - Concurrent_unordered_map
 - Concurrent_unordered_multimap
 - Concurrent_unordered_set
 - Concurrent_unordered_multiset

- 사용시 주의점

- 메소드들 이 표준 컨테이너와 약간 다르다.
 - 멀티쓰레드에서 사용가능한 메소드가 있고 **아닌** 메소드가 있다.

정리

- 멀티 스레드 프로그래밍에서는 공유 메모리를 사용해서 스레드간의 협업을 한다.
- Data Race로 인한 잘못된 결과
 - DataRace를 없애는 것은 Lock() 이 기본
- Lock()으로 인한 심각한 성능저하.
- Lock()을 제거
 - 직접 공유메모리를 사용했을 때 생기는 문제
 - 컴파일러, 일관성, 중간 값
- Atomic한 공유 자료 구조를 만들어서 사용해야 한다.
 - Non-Blocking 자료 구조가 바람직하다.
- 좋은 공유 자료 구조는 만들기 힘들다.
 - 무대기(wait free)알고리즘의 작성은 까다롭다.
 - 상용 라이브러리도 좋다. Intel TBB, Visual Studio 2015 PPL 등