



# 3. 메모리 일관성

멀티쓰레드 프로그래밍

정내훈

# 지금 까지

- 멀티쓰레드 프로그래밍의 필요성
- 멀티쓰레드 프로그래밍 API
- 멀티쓰레드 프로그래밍의 사용 예
- 멀티쓰레드 프로그래밍의 문제점
  - Data Race
  - 성능
  - Compiler
  - 피터슨 알고리즘의 BUG

# 지금 까지

- 멀티쓰레드 프로그래밍 가이드
  - Homogeneous 멀티쓰레드 프로그래밍
  - Locking의 회피
  - 공유 자료 구조의 활용

# 목차

- 멀티쓰레드 프로그램 작성시 주의점
  - 컴파일러
  - CPU
    - 상호배제의 구현
    - 메모리 일관성 문제
- Non-Blocking 프로그래밍
- 이론 시간 + CAS

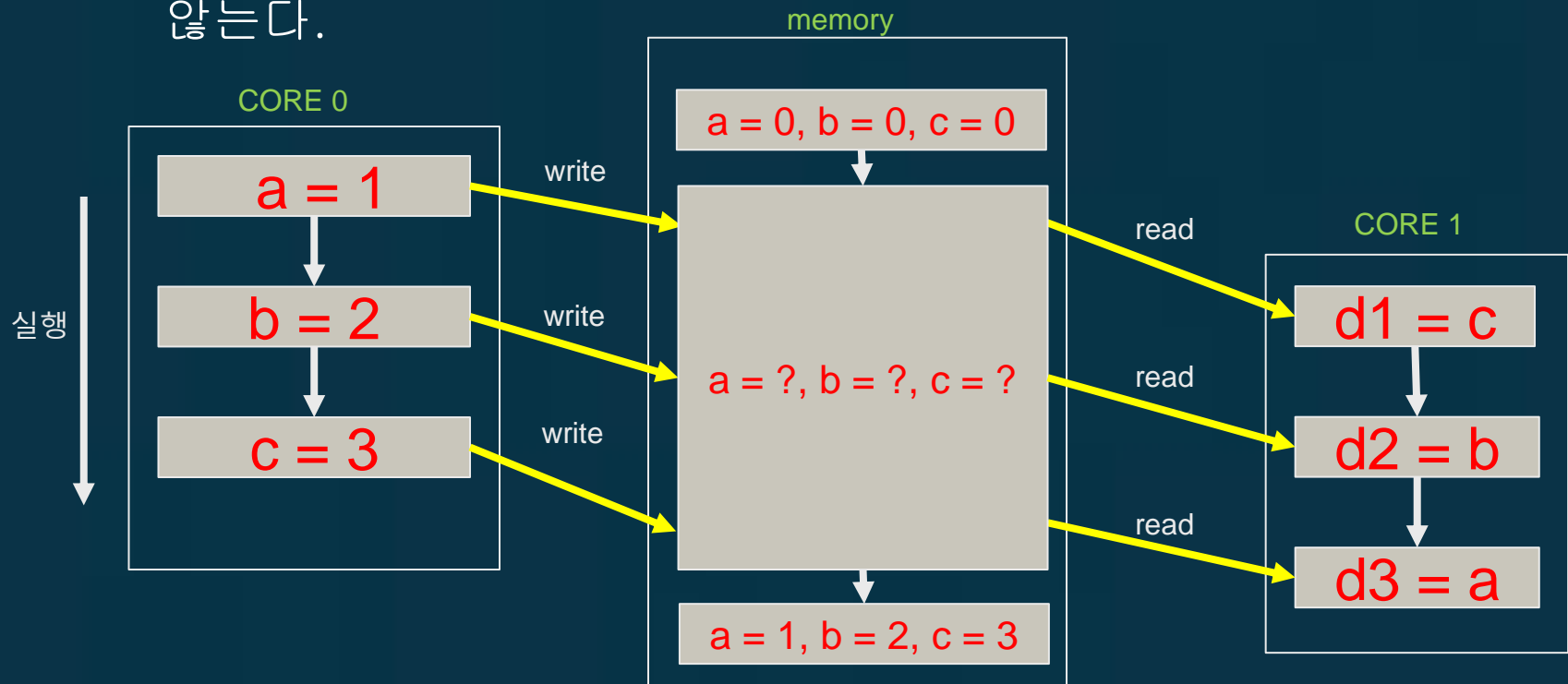
# 메모리 일관성

- 지금까지의 프로그램은 공유 메모리에 대한 접근 (쓰기/읽기)는 **Atomic**하다고 가정하고 있다. (컴파일러가 제대로 기계어로 번역했다면)
  - 정말 그런가???
- Atomic
  - 메모리의 접근이 순간적으로 행해 지며, 서로 겹쳐지지 않고
  - 실행결과 순서가 정해지면, 모든 스레드에서 같은 순서로 보인다.

# 메모리 일관성

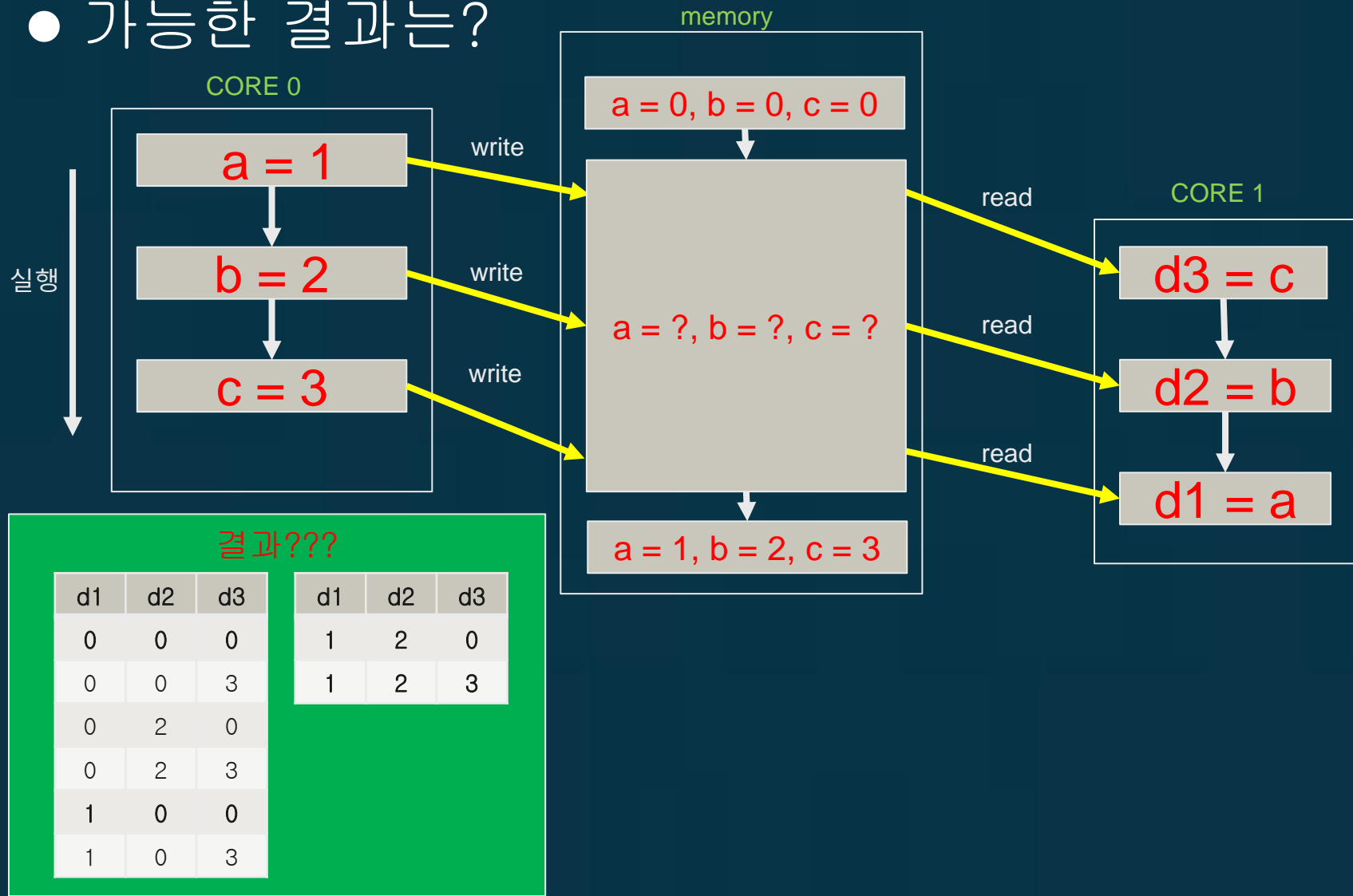
- 하지만

- PC에서의 메모리 접근은 Atomic이 아니다.
- 메모리에 쓴 순서대로 메모리가 수정되지 않는다.
  - 정확히는 메모리에 쓴 순서대로 메모리의 내용이 관측되지 않는다.



# 메모리 일관성

## ● 가능한 결과는?



# 메모리 일관성

## ● 피터슨 알고리즘 (지난 시간)

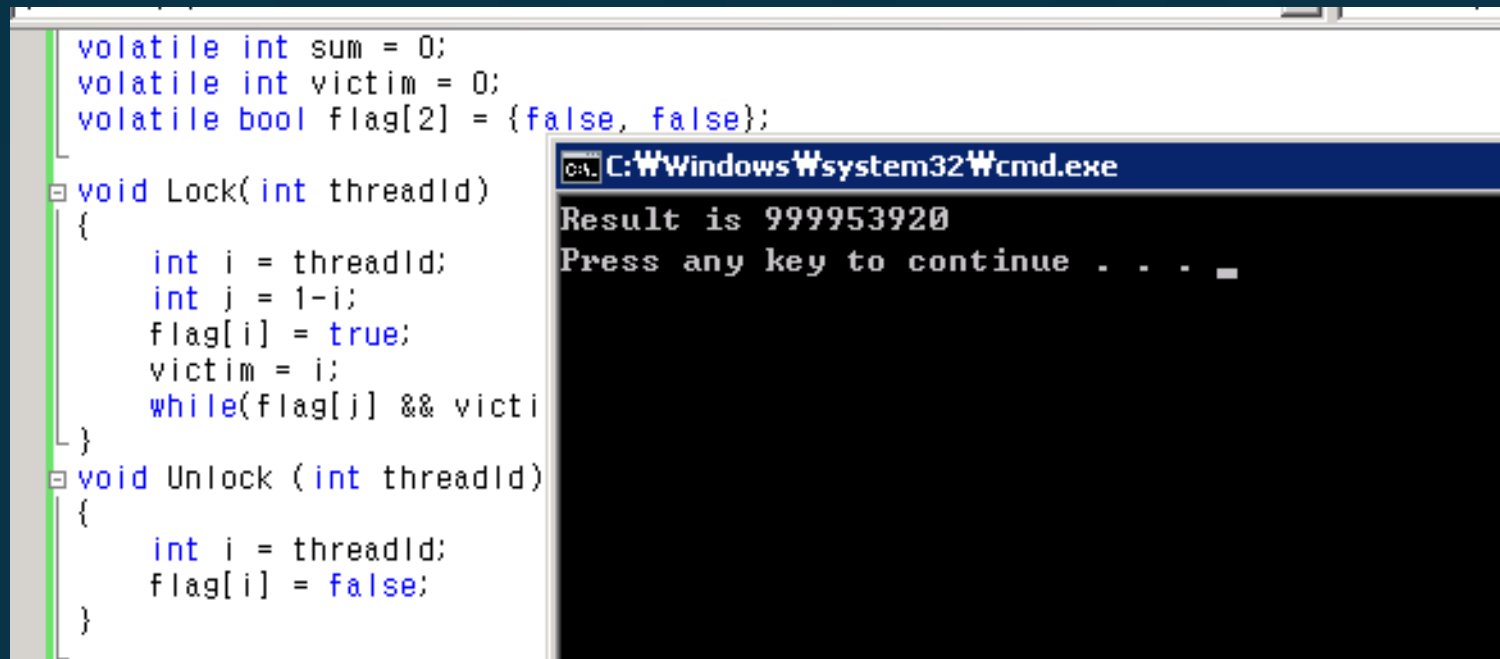
```
volatile int victim = 0;
volatile bool flag[2] = {false, false};

Lock(int myID)
{
    int yourID = 1 - myID;
    flag[myID] = true;
    victim = myID;
    while(flag[yourID] && victim == myID)
}
Unlock (int myID)
{
    flag[myID] = false;
}
```



# 메모리 일관성

## ● 상호 배제가 깨지는 결과



The image shows a C++ program in a text editor and its execution in a command prompt. The program defines a shared memory area with a sum, a victim, and flags for two threads. It includes Lock and Unlock functions to manage access. The output shows the result of the program execution.

```
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] && victi
}

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

Output:

```
C:\Windows\system32\cmd.exe
Result is 999953920
Press any key to continue . . .
```

# 메모리 일관성

- 원인은?

- PC에서의 메모리 접근은 Atomic이 아니다.
- 99.995%는 Atomic이지만 0.005%정도는 Atomic이 아니다.
  - 정확한 숫자 아님. CPU모델에 따라 비율이 다름

# 메모리 일관성

- 피터슨 알고리즘에서 역전
  - 컴파일러는 문제 없다.

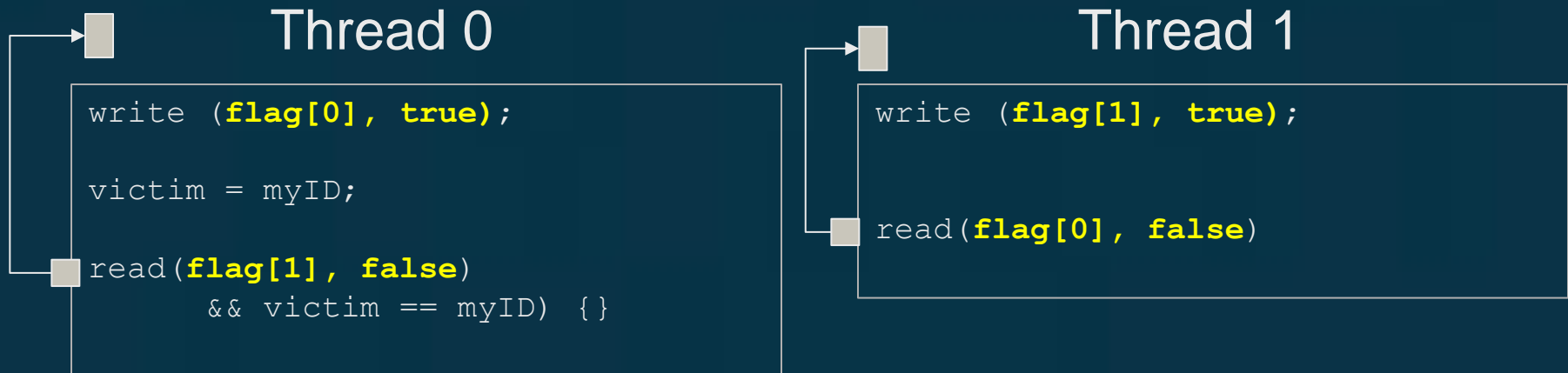
```
Lock(int threadId)
{
    int other = 1 - myID;
    flag[myID] = true;
    victim = myID;
    while(flag[other] && victim == myID) {}
}
```

```
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

## ● 피터슨 알고리즘에서 역전

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



문제 : 0번 스레드도 false를 읽고 1번 스레드도 false를 읽는다.

# 메모리 일관성

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

```
_asm mfence
```

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

# 메모리 일관성

- 리눅스

```
__asm__ __volatile__ ("mfence" ::: "memory");
```

- ARM

```
__asm__ __volatile__ ("dmb" ::: "memory")
```

- C++11

```
#include <atomic>

atomic_thread_fence(std::memory_order_seq_cst);
```

# 메모리 일관성

- 실습 #14

- 다음의 명령을 정확한 위치에 추가하여 peterson 알고리즘이 정확히 동작하도록 하시오.

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

# 메모리 일관성

- 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.
    - 이를 보조해 주는 특수한 HW가 존재한다.
    - 싱글코어에서는 절대로 들키지 않는다.



# 메모리 일관성

- Out-of-order 실행

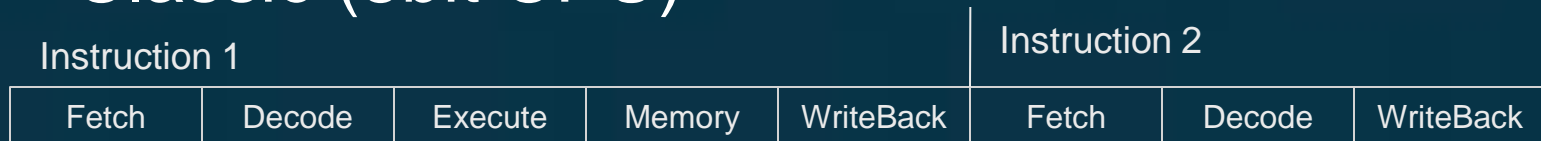
```
a = fsin(b);  
f = 3;
```

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

# Out-of-Order(CPU)

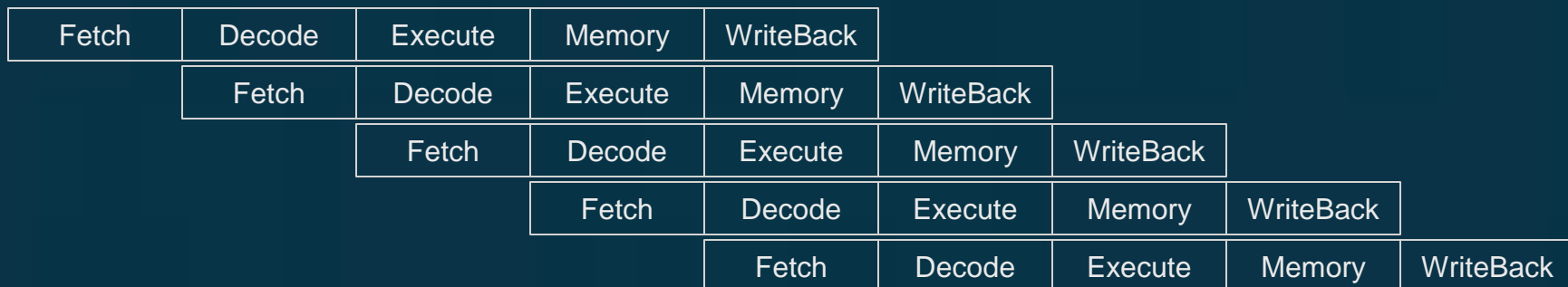
## ● CPU의 발전

### – Classic (8bit CPU)



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

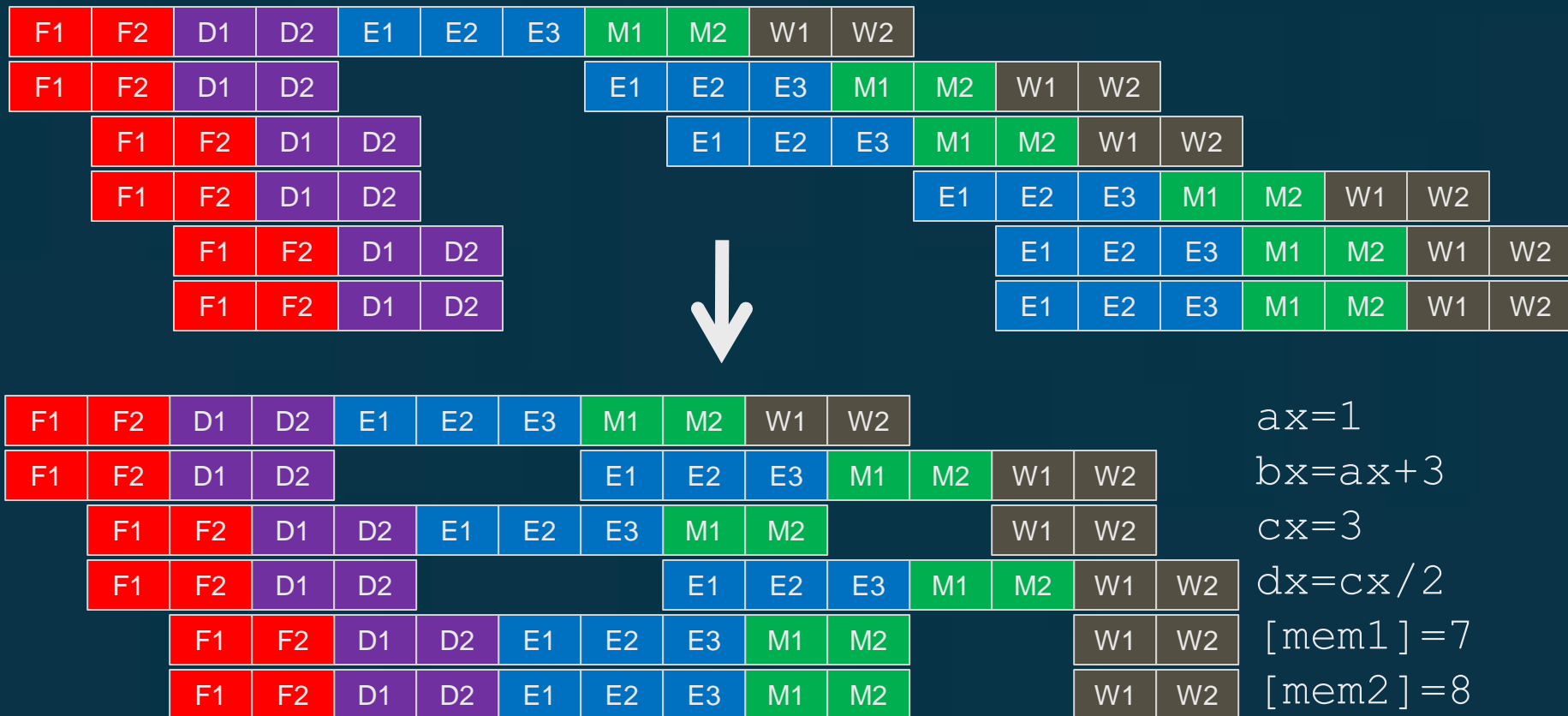
### – PipeLining (RISC CPU)





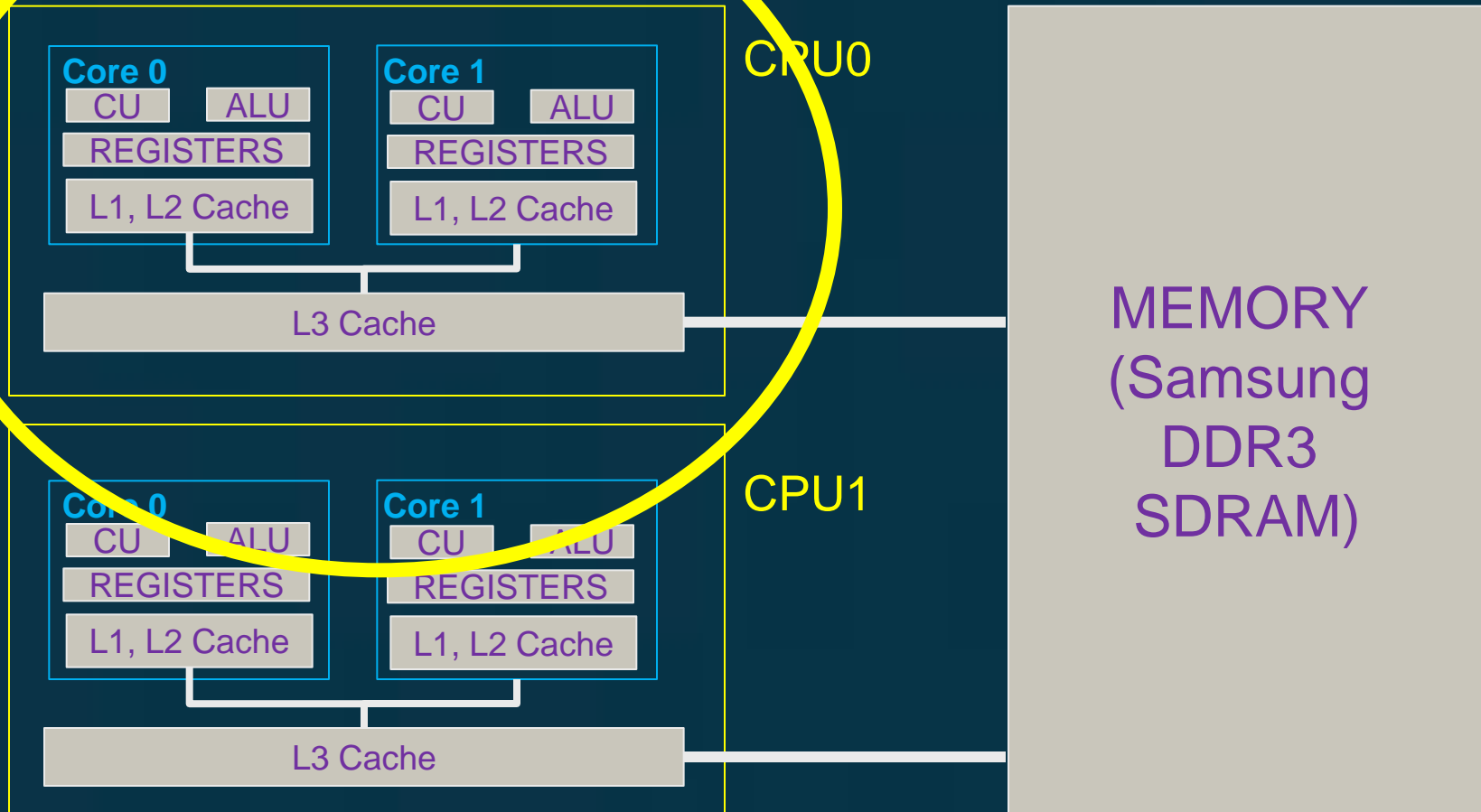
# Out-of-Order(CPU)

- CPU의 발전
  - Out of Order (Pentium Pro)



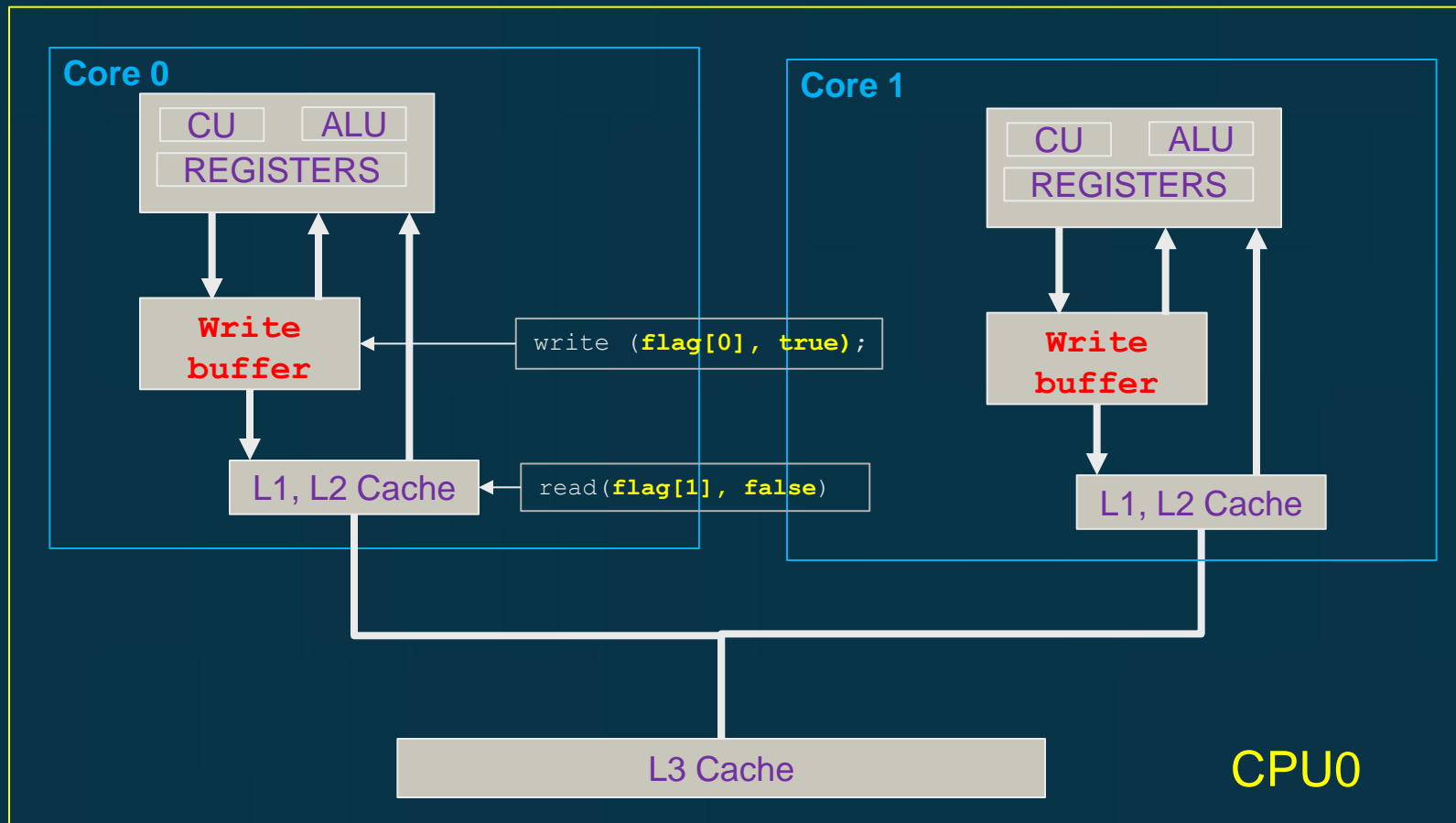
# 메모리 일관성

- 성능 향상을 위한 HW



# 메모리 일관성

- 성능 향상을 위한 HW : x86 CPU



# 메모리 일관성

- 문제는 메모리

- 읽고/쓰는 기계어가 순서대로 실행되지 않는다.

- volatile로도 해결되지 않는다.

- volatile 키워드는 기계어로 번역되지 않는다.

- 읽기와 쓰기는 시간이 많이 걸리므로.

- CPU의 입장에서 보면

- 실제 영향이 발휘되는 시간은 상대 core에 따라 다르다.

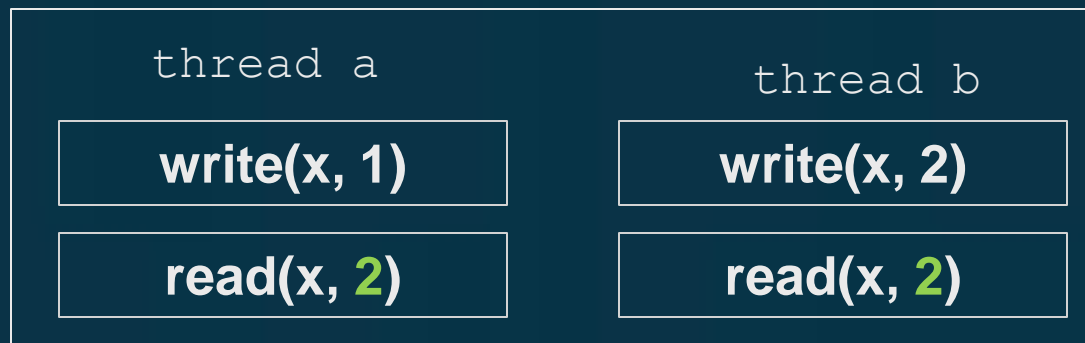
- 옆의 프로세서(core)에서 보면 어긋난 순서가 보인다, 또는 읽는 순서가 어긋난다.

- 자기 자신은 절대 알지 못한다.

- 어떠한 일이 벌어지는가?

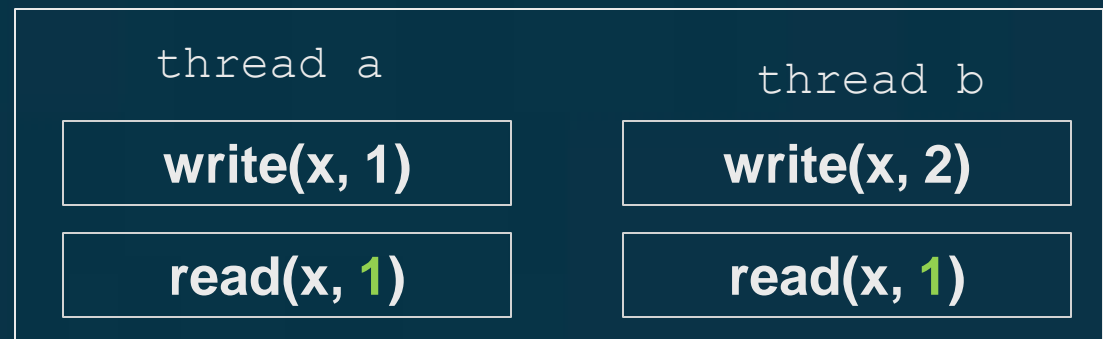
# 메모리 일관성

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



Type-A

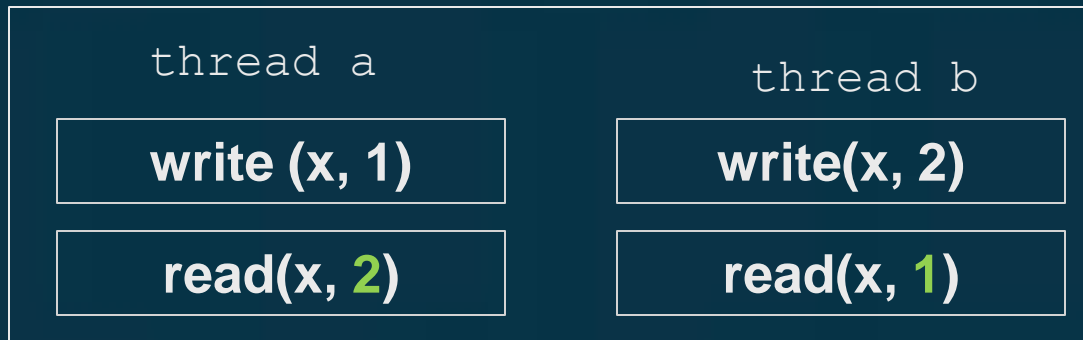
Type-B





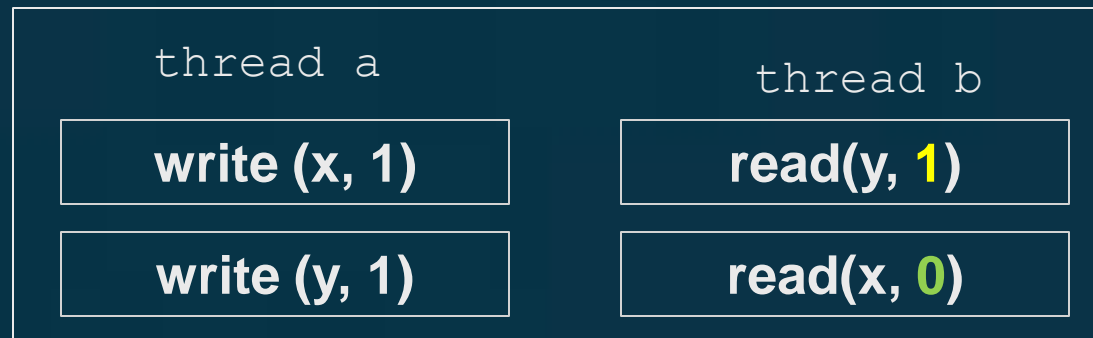
# 메모리 일관성

## ● 그러면 이것은?



Type-C!!

Type-D !!



# 메모리 일관성 (2024)

## ● 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

## ● 실습

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

# 메모리 일관성

## ● 실습 #15

– 다음의 프로그램을 실행해 볼 것

```
const auto SIZE = 50000000;
volatile int x, y;
int trace_x[SIZE], trace_y[SIZE];

void ThreadFunc0()
{
    for(int i = 0; i < SIZE; i++) {
        x = i;
        trace_y[i] = y;
    }
}

void ThreadFunc1()
{
    for(int i = 0; i < SIZE; i++) {
        y = i;
        trace_x[i] = x;
    }
}
```

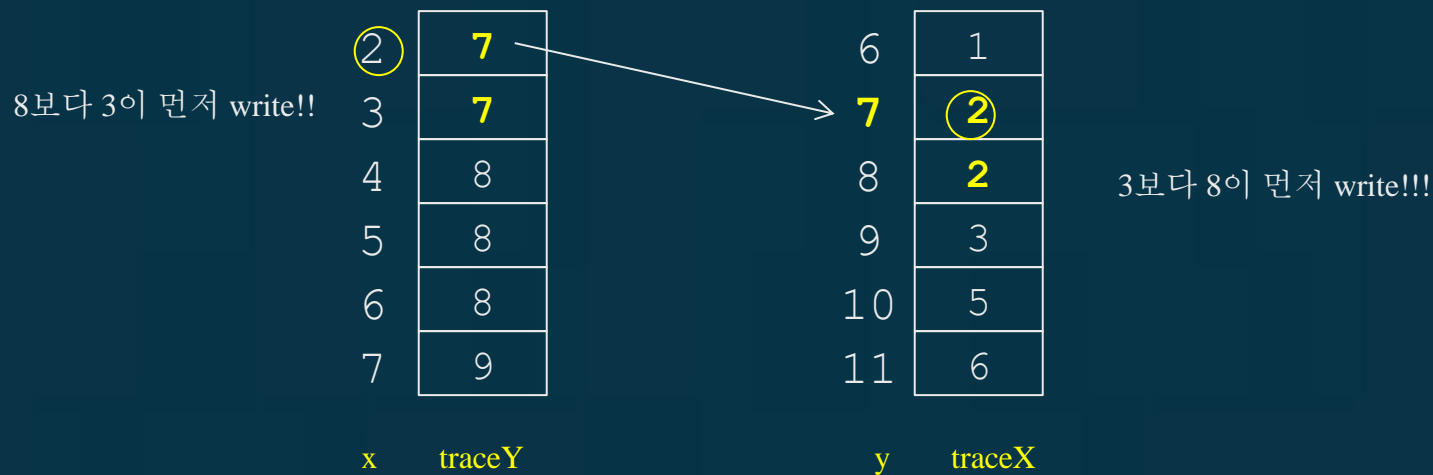
```
int main()
{
    .. // Thread 2개 실행

    int count = 0;
    for (int i=0; i< SIZE;++i)
        if (trace_x[i] == trace_x[i+1])
            if (trace_y[trace_x[i]] == trace_y[trace_x[i] + 1]) {
                if (trace_y[trace_x[i]] != i) continue;
                count++;
            }
    cout << "Total Memory Inconsistency: " << count << endl;
}
```

# 메모리 일관성

## ● 프로그램 설명

- 2개의 core에서 서로 다른 순서로 메모리가 업데이트 되는 현장을 포착



# HELL



공황상태...

# 메모리 일관성

- 실습

- 메모리 변경 순서가 뒤바뀔 확률은?
- `atomic_thread_fence()`로 오류를 없애 보자.
- `Atomic <int>`로 극복해 보자



# 메모리 일관성

- 메모리에는 유령이...
  - 변수 값을 변경하였을 경우, 변경 자체는 Atomic한가?
    - 모든 비트가 한 순간에 변하는가?

# 메모리 일관성 (2024)

## ● 실습 #16

- 아래의 프로그램을 실행하고 error의 개수를 출력하라
- bound는 적당한 변수를 pointing하도록 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 ++;
    }
}
```

# 메모리 일관성

---

- 실습 #16
  - 결과는?

# 메모리 일관성

- 실습 17 : 실습 16의 프로그램의 main()부분을 아래와 같이 수정하여 실행해 보자.

```
int ARR[32];

long long temp = (long long) &ARR[31];
temp = temp - (temp % 64);
temp -= 2;
bound = (int *) temp;
*bound = 0;

<쓰레드 생성>

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

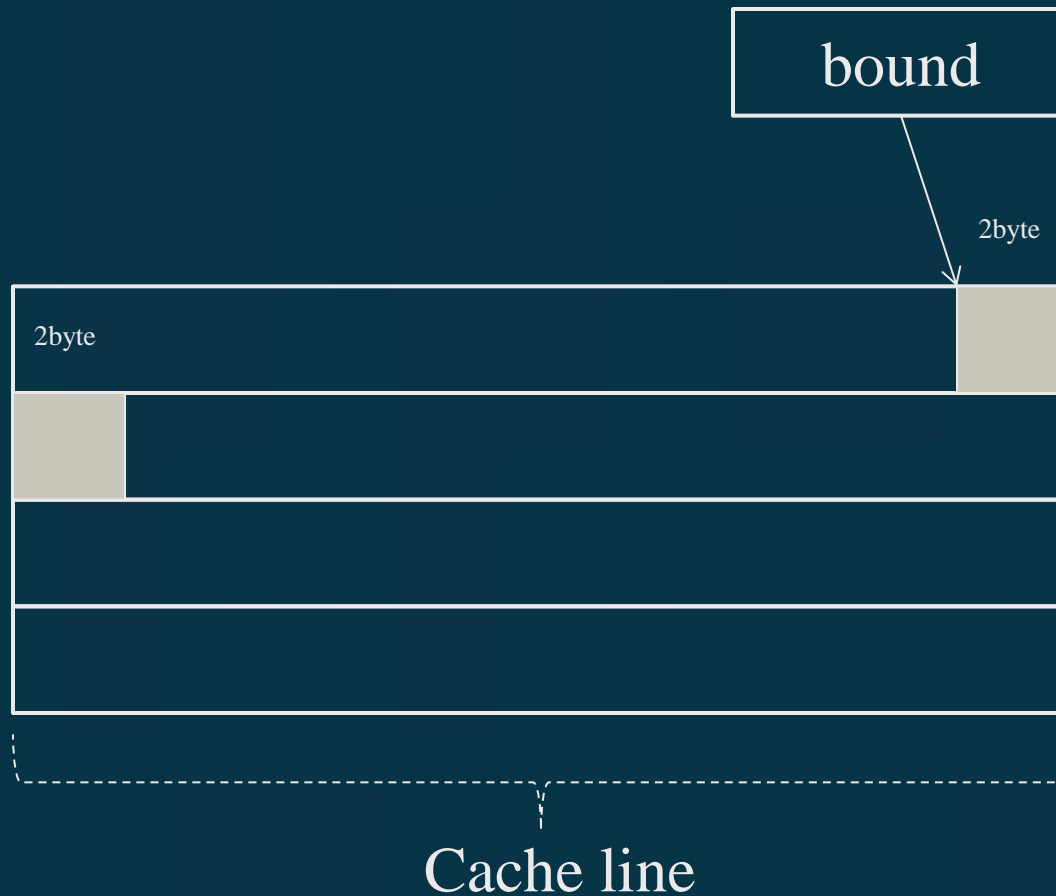
# 메모리 일관성

---

- 실습 #17
  - 결과는?

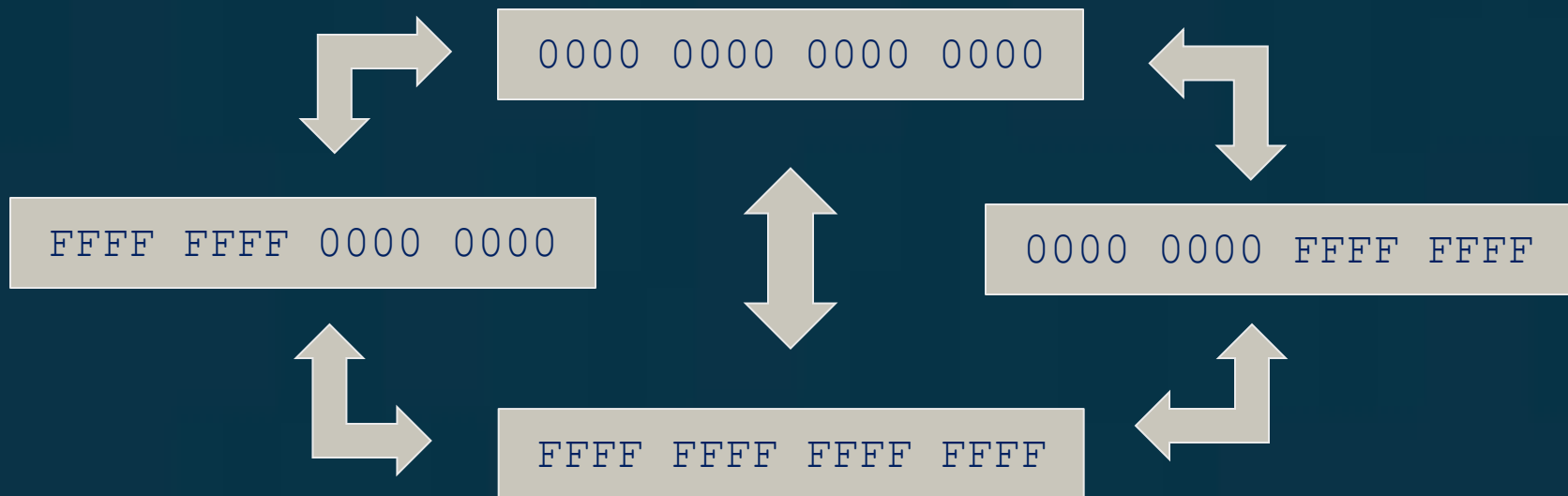
# HELL

- 어떻게 실행했길래?



# HELL

- 컴퓨터 구조 : 2의 보수
  - 0 => 0000 0000 0000 0000
  - -1 => FFFF FFFF FFFF FFFF



# HELL

- 결과가...

- 중간값

- write시 최종 값과 초기값이 아닌 다른 값이 도중에 메모리에 쓰이는 현상

- 이유는?

- Cache Line Size Boundary

- 대책은?

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

```
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 );
```



# 메모리 일관성

- 메모리 일관성

- 무모순성 (Consistency)라고 불리운다.
- 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.

# 메모리 일관성

## ● 메모리 일관성 사례

Memory ordering in some architectures <sup>[1][2]</sup>

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](http://en.wikipedia.org/wiki/Memory_ordering)

# 메모리 일관성

## ● 정리

### — 멀티쓰레드에서의 공유 메모리

- 다른 코어에서 보았을 때 업데이트 순서가 틀릴 수 있다. (쓰는 쪽과 읽는 쪽 둘 다의 문제)
- 메모리의 내용이 한 순간에 업데이트 되지 않을 때도 있다.

### — 일반적인 프로그래밍 방식으로는 멀티쓰레드에서 안정적으로 돌아가는 프로그램을 만들 수 없다.

# 메모리 일관성

- 어떻게 할 것인가?

- 위의 상황을 감안하고 프로그램 작성?

- 프로그래밍이 너무 어렵다.
- 피터슨 알고리즘도 동작하지 않는다.

- 강제로 원하는 결과를 얻도록 한다.

- 모든 메모리 접근을 <mutex>로 막으면 가능

- 성능저하!!!

- atomic\_thread\_fence()명령어를 적절한 위치에 추가

- 찾은 위치가 완벽하다는 보장은?

- 해결 방향

- 꼭 필요한 곳에 atomic\_thread\_fence() 사용
- 공유 변수로 Atomic한 메모리 사용

- 컴파일러가 적절한 위치에 atomic\_thread\_fence를 넣어줌

# 희망

- 메모리에 대한 쓰기는 언젠가는 완료 된다.
- 자기 자신의 프로그램 실행 순서는 지켜진다.
- 캐시의 일관성은 지켜진다.
  - 한번 지워졌던 값이 다시 살아나지는 않는다.
  - 언젠가는 모든 코어가 동일한 값을 본다
- 캐시라인 내부의 읽기/쓰기는 중간 값을 만들지 않는다.

# 희망

- 우리가 할 수 있는 것

- CPU의 여러 샵질에도 불구하고 Atomic Memory를 구현 할 수 있다.

- <mutex>나 fence없이 SW적인 방법 만으로도 가능
- CPU에 특수 명령어로 구현되어 있음
  - C++11에서 사용

- Atomic

- 접근(메모리는 read, write)의 절대 순서가 모든 스레드에서 지켜지는 자료구조
- 프로그래머가 필요로 하는 바로 그 자료구조
- 싱글코어에서는 모든 메모리가 Atomic Memory이다.

# 희망

- C++11 에서의 Atomic Memory
  - load(), store()를 사용해야 하지만 일반적인 ‘=’ 연산자도 Atomic하게 동작한다.

```
#include <atomic>

atomic <int> a;

a.store(3);
int d = a.load();
```

- SW로 구현할 수도 있지만 오버헤드가 커서 HW구현을 사용한다.

# Atomic

- C++11의 <atomic>에 대해 알아보자.
  - atomic <int> a;

```
#include <atomic>
#include <memory>
#include <iostream>

using namespace std;

int main()
{
    atomic<int> a, b;

    a.store(1, memory_order_release);
    b.store(2, memory_order_release);
    int c = a.load(memory_order_acquire);
    c = a.load(memory_order_seq_cst);
    b.store(3, memory_order_seq_cst);
}
```



# Atomic

- C++11에서의 Atomic 메모리의 종류
  - seq\_cst : Sequential Consistency
  - relaxed : Relaxed Consistency
  - acquire/release : Release Consistency
- 교재에서의 Consistency
  - Quiescent Consistency
  - Sequential Consistency
  - Linearization

# 희망

- 실습

- `atomic <int>` 와 `atomic <bool>`을 사용하여 `peterson` 알고리즘을 구현해 보자.
- 그냥 구현했을 때, `atomic_thread_fence`를 넣었을 때와 속도 비교를 하라.

# Atomic

- 다음 프로그램이 정상적으로 동작 할까?

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

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

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

# Atomic

## ● 다음 프로그램이 정상적으로 동작 할까?

```

00DC102B  mov     ebx,offset victim (0DC53FCh)
          Lock(t_id);
00DC1030  mov     cl,1
00DC1032  xchg    cl,byte ptr [edi]
00DC1034  mov     ecx,eax
00DC1036  xchg    ecx,dword ptr [ebx]
00DC1038  nop     dword ptr [eax+eax]
00DC1040  mov     cl,byte ptr flag (0DC53F4h)[edx]
00DC1046  test    cl,cl
00DC1048  je      make_sum+44h (0DC1054h)
00DC104A  mov     ecx,dword ptr [victim (0DC53FCh)]
00DC1050  cmp     ecx,eax
00DC1052  je      make_sum+30h (0DC1040h)
          sum += 2;
00DC1054  mov     ecx,dword ptr [sum (0DC53F8h)]
00DC105A  add     ecx,2
00DC105D  mov     dword ptr [sum (0DC53F8h)],ecx
          Unlock(t_id);
00DC1063  xor     cl,cl
00DC1065  xchg    cl,byte ptr [edi]
          for (int i = 0; i < 25000000; ++i) {
00DC1067  sub     esi,1
00DC106A  jne     make_sum+20h (0DC1030h)

```

# Atomic

- 다음 프로그램이 정상적으로 동작 할까?

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

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

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

# Atomic

## ● 다음 프로그램이 정상적으로 동작 할까?

```

007B101F  mov         edi,17D7840h
007B1024  lea         esi,flag (07B53F4h)[eax]
007B102A  nop
                                Lock(t_id);
007B1030  mov         cl,1
007B1032  xchg        cl,byte ptr [esi]
007B1034  mov         dword ptr [victim (07B53FCh)],eax
007B1039  nop         dword ptr [eax]
007B1040  mov         cl,byte ptr flag (07B53F4h)[edx]
007B1046  test        cl,cl
007B1048  je          make_sum+42h (07B1052h)
007B104A  cmp         dword ptr [victim (07B53FCh)],eax
007B1050  je          make_sum+30h (07B1040h)
                                sum += 2;
007B1052  mov         ecx,dword ptr [sum (07B53F8h)]
007B1058  add         ecx,2
007B105B  mov         dword ptr [sum (07B53F8h)],ecx
                                Unlock(t_id);
007B1061  xor         cl,cl
007B1063  xchg        cl,byte ptr [esi]
                                for (int i = 0; i < 25000000; ++i) {
007B1065  sub         edi,1
007B1068  jne         make_sum+20h (07B1030h)

```

# Atomic

- 다음 프로그램이 정상적으로 동작 할까?

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

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

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

# Atomic

## ● 다음 프로그램이 정상적으로 동작 할까?

```
volatile int victim = 0;
atomic <unsigned char> flag[2] = {0, 0};

void Lock(int myId)
{
    int other = 1 - myId;
    flag[myId] = 1;
    victim = myId;
    while ((flag[other].fetch_add(0) == 1) && victim == myId);
}

void Unlock (int myId)
{
    flag[myId] = 0;
}
```



# Atomic

- Atomic Memory 만 있으면 되는가?
  - NO
  - 실제 상용 프로그램을 int, long, float같은 기본 data type 만으로 작성할 수 있는가?
- 실제 프로그램은 기본 data type을 사용하여 다양한 자료구조를 구축하여 사용한다.
  - queue, stack, binary tree, vector...

# Atomic

- 문제
  - Atomic Memory를 사용하여 만든 자료구조가 Atomic한가?
    - NO
- 효율적인 Atomic 자료구조가 필요하다.
  - 일반 자료구조에 Lock을 추가하면
    - 너무 느리다.
  - STL은???
  - CRASH!
  - STL + LOCK은???
  - 느려서 못쓴다.
- 근데 “효율적인” 이라니?

# 목차

- 병렬 프로그램 작성시 주의점
  - 컴파일러
  - CPU
    - 상호배제의 구현
    - 메모리 일관성 문제
- Non-Blocking 프로그래밍
- 이론 시간 + CAS

# Lock없는 프로그램

- 효율적인 구현

- Lock없는 구현

- 성능 저하의 주범이므로 당연
      - Overhead
      - Critical Section(상호배제, 병렬성X)
      - Priority inversion
      - Convoying

- Lock이 없다고 성능저하가 없는가??

- 상대방 스레드에서 어떤 일을 해주기를 기다리는 한 동시실행으로 인한 성능 개선을 얻기 힘들다.
      - `while (other_thread.flag == true);`
      - lock과 동일한 성능저하
    - 상대방 스레드의 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 작업을 하고 있던 상관없이 진행
  - 예) 공유메모리 읽기/쓰기, Atomic (Interlocked) Operation

# Non-Blocking

- 블럭킹 알고리즘의 문제
  - 성능저하
  - 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 (2024)

- 년블럭킹의 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

- 년블럭킹의 등급
  - 제한된 무대기 (bounded wait-free)
    - 유한하지만 스레드의 개수에 비례한 유한일 수 있다.
  - 무간섭 (obstruction-free)
    - 한 개를 제외한 모든 스레드가 멈추었을 때, 멈추지 않은 스레드의 메소드가 유한한 단계에 종료할 때.
    - 충돌 중인 스레드를 잠깐 멈추게 할 필요가 있다.



# 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.

# 복습

- 실제 컴퓨터의 메모리는 멀티코어 프로그래밍에서 이상하게 동작한다.
- Atomic한 자료구조가 필요하다.
  - non-blocking 알고리즘이 효율적이다.
- non-blocking 알고리즘의 등급
  - wait-free
  - lock-free
  - obstruction-free

# 질문???

---