5-2 배경이론

멀티쓰레드 프로그래밍

정내훈

개요

- 지금까지 배운 내용의 이론적 토대
- 교재의 용어를 그대로 사용하므로 좀 낯선 용어들이 등장.
  - 추후 교재를 통한 복습을 위해 교재 용어 그대로 사용.

목차

- 합의 객체 (Consensus Object)
  - Non-blocking 알고리즘을 만들기 위해 필요한 객체
- 합의수 (Consensus Number)
  - non-blocking 알고리즘을 만드는 능력
- 만능성 (Universality)
  - 모든 알고리즘을 멀티 쓰레드 무대기로

복습

- 우리가 사용하고 있는 컴퓨터의 메모리는 멀티코어 프로그래밍에서는 믿을 놈이 못 된다. => atomic이 아니다
- 하지만 우리는 atomic하게 사용할 수 있다.
  - atomic<T>를 사용해서 자료구조 T를 atomic하게 사용하도록 컴파일 한다.
  - 적절한 위치에 atomic_thread_fence를 추가한다.

복습

- atomic<int> a;
  - a에 대한 모든 연산을 atomic으로 수행하며, wait-free로 수행된다.
  - 예) a = 3, a += 7, sum = a;
- atomic<vector> a;
  - 컴파일 에러, 복잡한 자료구조는 atomic하게 변경할 수 없다.
- atomic<point> pos;
  - pos에 대한 load, store가 atomic
  - 내부적으로 mutex를 사용해서 구현

```
struct point {
    int x,y,z;
};
```

복습

- atomic한 복잡한 자료구조가 필요하면?
  - vector, tree, hash_table, priority-queue
- 적절한 동기화기법을 사용해서 Non-blocking으로 변환해서 사용해야 한다.

동기화

- 동기화
  - 자료구조의 동작을 Atomic하게 구현하는 것
  - 우리는 성긴/세밀한/낙천적인/게으른/비멈춤 동기화를 구현해 보았다.
- 동기화를 구현하기 위해서는 기본 동기화 연산들을 사용해야 한다.
  - 예) 메모리 : atomic_load(), atomic_store()
  - 예) LF_SET : ADD(), REMOVE(), CONTAINS()
- 이 기본 동기화 연산들은 무대기(wait-free)혹은 무잠금(lock-free)이어야 한다.
  - 아니면 무대기나 무잠금 동기화를 구현할 수 없다.

진행

- atomic_load(), atomic_store()와 non-blocking자료구조의 관계는?
- 직접적으로 비교가 어렵기 때문에 중간다리를 도입할 예정
- 중간다리) 합의 객체
- 합의 객체로 무엇을 할 수 있는가? 합의 객체를 구현하려면 무엇이 필요한가? 등을 살펴볼 예정.

합의(Consensus) 객체

- 새로운 동기화 연산을 제공하는 가상의 객체
- 동기화 연산 : decide
  - 선언
    - `Ttype_t decide(Type_t value)`
  - 동작
    - n 개의 스레드가 decide를 호출한다.
    - 각각의 스레드는 한번 이하로만 호출한다.
    - Decide는 모든 호출에 대해 같은 값을 반환한다.
    - Decide가 반환하는 값은 전달된 value중 하나이다.
    - Atomic하고 Wait-Free로 동작한다.

합의(Consensus)

- 의미
  - 모든 쓰레드가 같은 결론을 얻는 방법
  - decide()를 사용해 모든 쓰레드가 wait-free로 같은 결론을 얻는다.
  - 여러 경쟁 쓰레드들 중 하나를 선택하고, 누가 선택되었는지 모든 쓰레드가 알게 한다.
    - 높은 확률로 제일 처음 decide()를 호출한 쓰레드가 선택됨

합의(Consensus)

- Blocking 구현
  - wait-free가 아니므로 합의 객체는 아님

```
class Consensus {
private:
        bool decided;
        Type_t d_value;
        mutex l;
public:
        Consensus() { decided = false; }
        Type_t decide(Type_t value)   {
                l.lock();
                if (false == decided) {
                    d_value = value;
                    decided = true;
                 }
                l.unlock();
                return d_value;
        }
};
```

합의(Consensus)

- NonBlocking 구현

```
class Consensus {
private:
        Type_t d_value;
public:
        Consensus() {
                d_value = INIT;  // INIT은 절대로 사용되지 않는 값
        }
        Type_t decide(Type_t value)   {
                CAS(&d_value, INIT, value);
                return d_value;
        }
};
```

목차

- 합의 객체
  - Non-blocking 알고리즘을 만들기 위해 필요한 객체
- 합의수
  - non-blocking 알고리즘을 만드는 능력
- 만능성
  - 모든 알고리즘을 멀티 쓰레드 무대기로

합의 수(Consensus number)

- 정의
  - 동기화 연산을 제공하는 클래스 C가 있을 때
  - 클래스 C와 atomic 메모리를 여러 개 사용해서 n개의 스레드에 대한 합의 객체를 구현 할 수 있다 => 클래스 C가 n-스레드 합의 문제를 해결한다(solve)고 한다.
  - 클래스 C의 합의 수(consensus number)
    - C를 이용해서 해결 가능한 n-스레드 합의 문제 중 최대의 n을 말한다. 만약 최대 n이 존재하지 않는다면, 그 클래스의 합의 수를 무한하다(infinite)고 한다.
  - 동기화 객체 C가 얼마나 파워풀한 가를 계측
    - 0, 1 : 있으나 마나
    - 2 : 2개 쓰레드 해결 가능, 3개 쓰레드 해결 불가능
    - 무한대 : 가장 파워풀한 객체

근본 문제

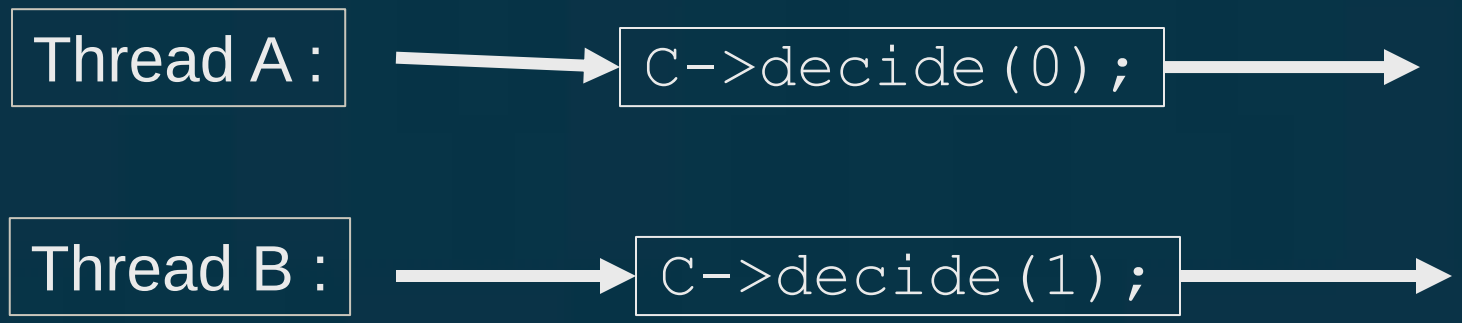
- Atomic 메모리로 n개 스레드의 합의 문제를 해결할 수 있는가?
  - Wait Free혹은 Lock Free로

  ⇒atomic_load(), atomic_store()연산만을 사용해서 n개 스레드 합의 객체를 만들 수 있는가?

  ⇒일단 2개 쓰레드에서 합의 객체를 구현할 수 있나 살펴보자.

Decide 구현

- 문제 단순화
  - 쓰레드 2개 : A, B
  - 각각 0 과 1로 합의 시도
  - C는 합의 객체 (atomic 메모리 read/write로만 구현)
  - return 값은 모두 0이던가, 모두 1이어야 한다.

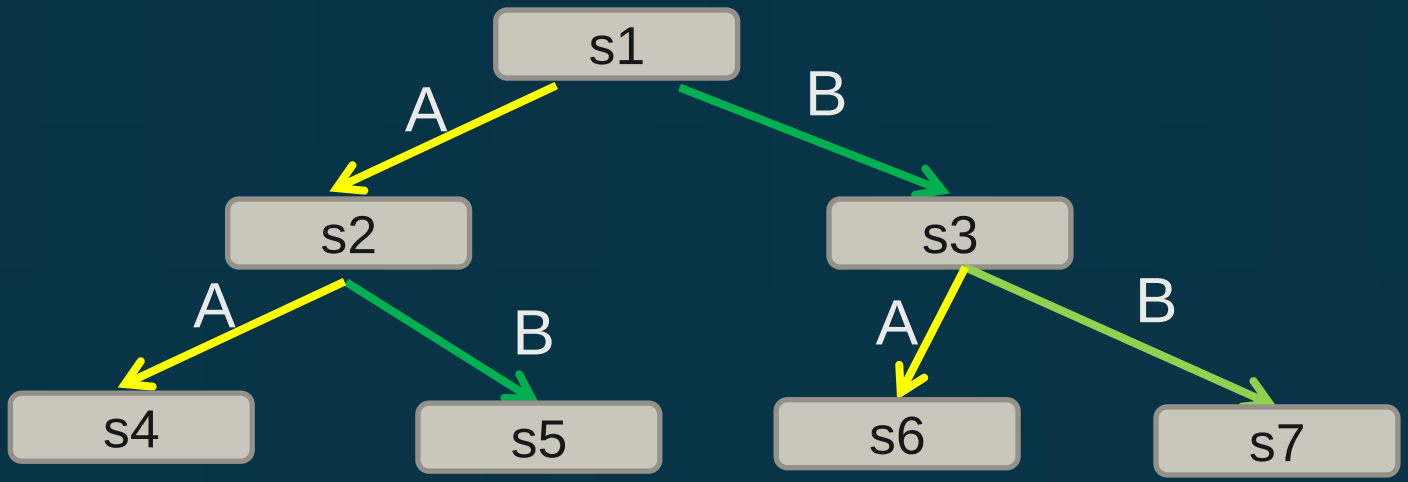


Decide 구현 (2024)

- 문제 단순화
  - decide 알고리즘의 실행
    - 쓰레드 A와 쓰레드 B가 동시에 실행되며 return값을 결정.
    - atomic_load()/atomic_store()를 read/write라 부르자.
    - A와 B는 임의의 개수의 공유메모리에 대해 read/write연산을 수행하면서 알고리즘을 수행
    - return값에 영향을 미치는 연산은 공유메모리에 대한 read/write밖에 없다.
      - 공유메모리의 값은 일종의 Input과 Ouput이다. Input이 같으면 로컬변수는 항상 같은 값을 갖는다. Output도 같지만 실행 순서에 따라 공유메모리에 저장되는 Output이 달라진다.
    - return 값은 공유메모리 연산이 어떠한 순서대로 실행되었느냐 에 의해 결정된다.
      - 같은 순서로 공유메모리 연산이 실행되면 항상 같은 return값이 나온다.
    - 알고리즘의 실행 과정 중 공유메모리에 대한 read/write를 따로 분리해서 생각해 보자.
      - 이를 이동(MOVE)라고 부르자.

Decide 구현

- 문제 단순화
  - 알고리즘의 모든 실행 가능한 경로를 이진 트리로 나타낼 수 있다. 이를 프로토콜이라 부르자.
    - 프로토콜의 생김새는 구현 알고리즘과 input 값이 결정한다.
    - 왼쪽 : A가 이동, 오른쪽 : B가 이동
      - A, B는 read이거나 wrtie이다.
    - 실제 실행 시 어느 방향으로 진행할 지는 정해져 있지 않다.
      - 어느 방향으로 진행하더라도 올바른 결과가 나와야 한다.

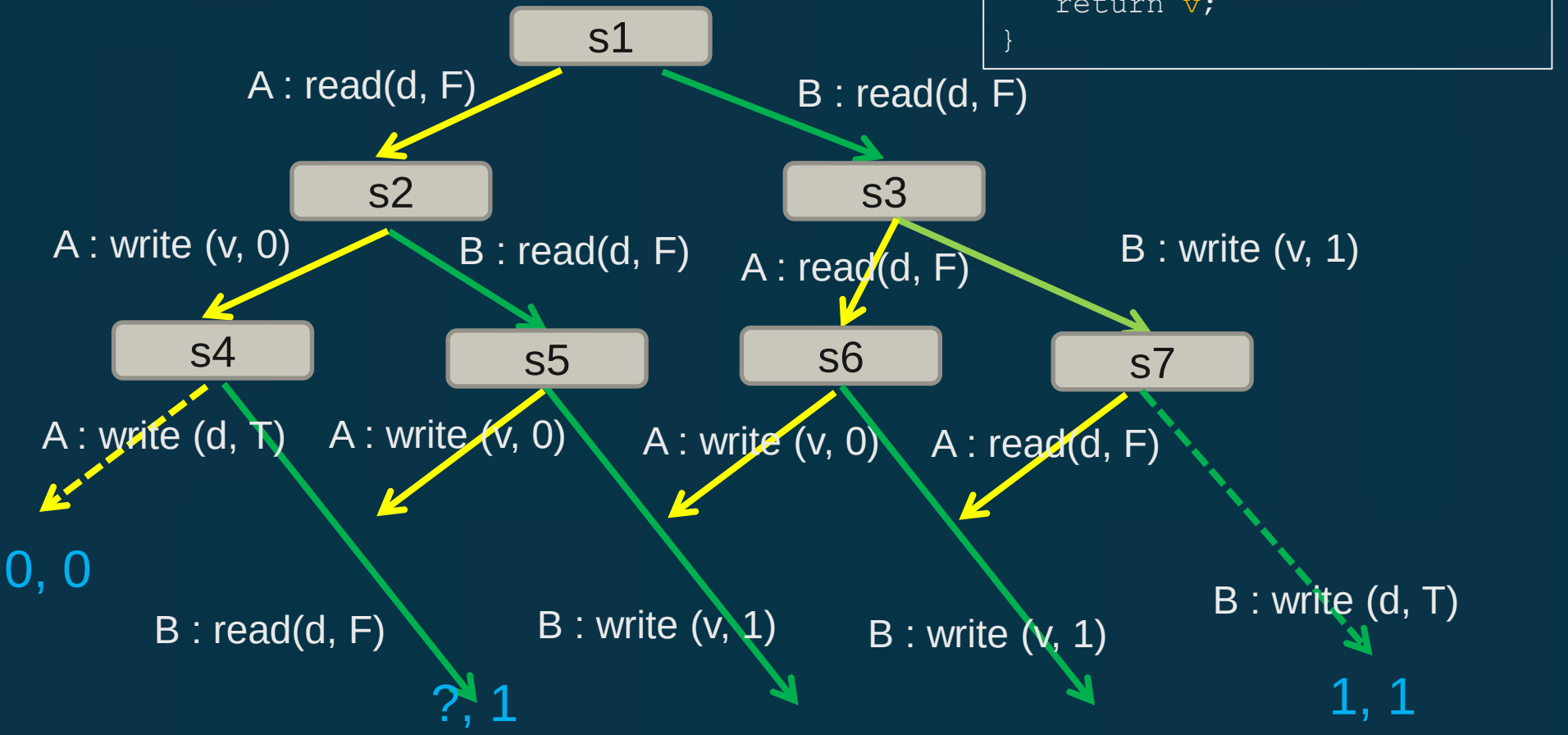


Decide 구현

- 프로토콜

Thread A : --decide(0)--
Thread B : --decide(1)--

```
Type_t decide(Type_t value) {
    if (false == d) {
        v = value;
        d = true;
    }
    return v;
}
```

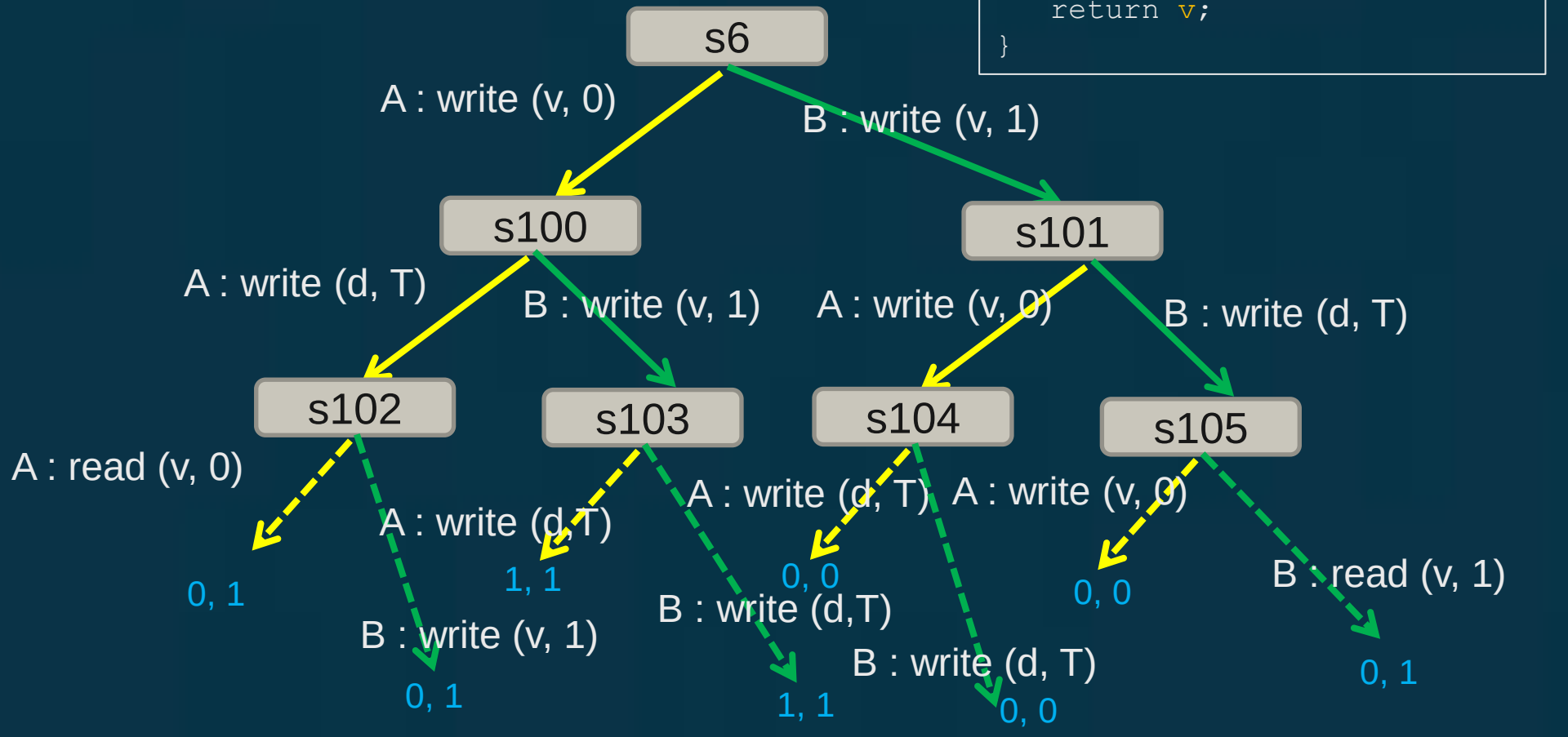


Decide 구현

- 프로토콜

Thread A : --decide(0)--
Thread B : --decide(1)--

```
Type_t decide(Type_t value) {
    if (false == d) {
        v = value;
        d = true;
    }
    return v;
}
```

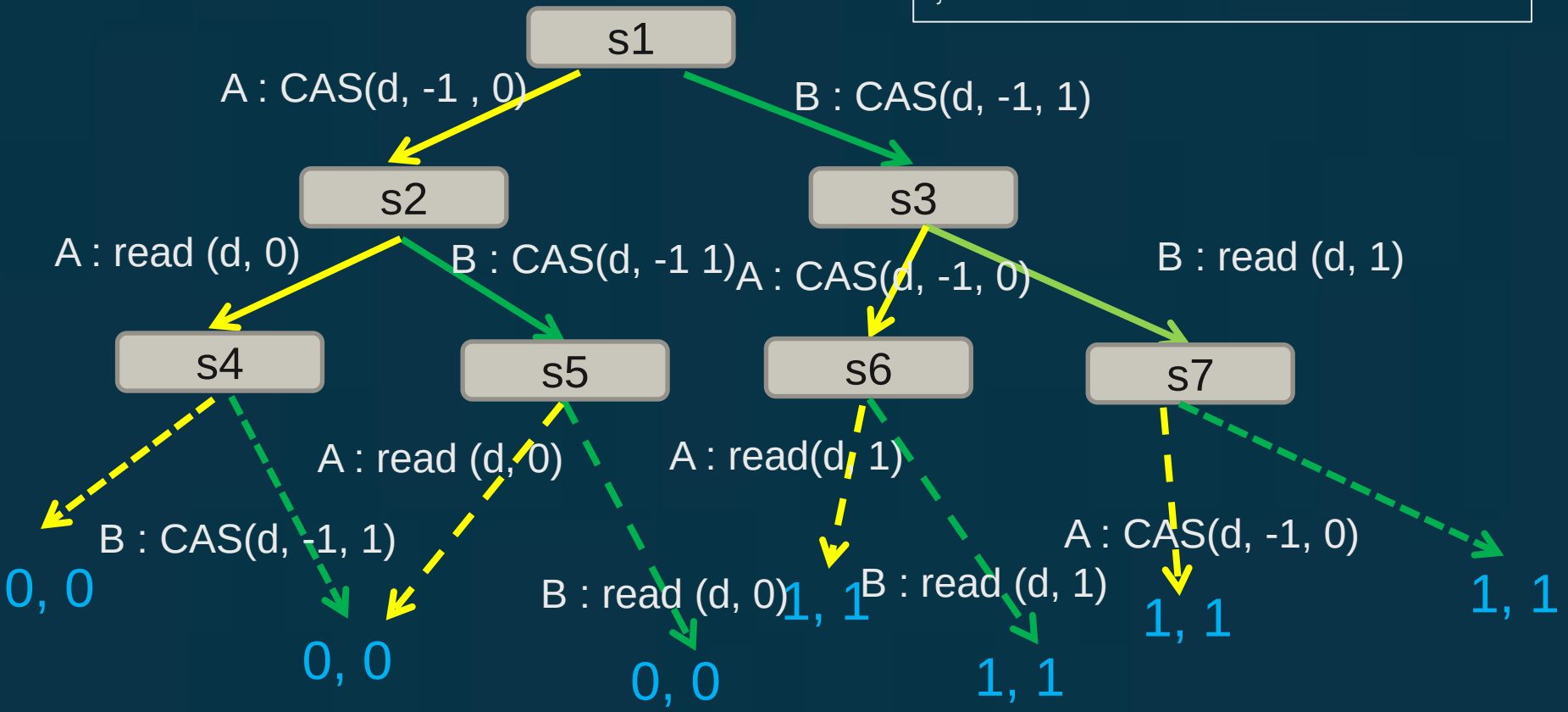


Decide 구현

- 프로토콜

Thread A : --decide(0)--
Thread B : --decide(1)--

```
Type_t decide(Type_t value)
{
    CAS(&d, INIT, value);
    return d;
}
```



Decide 구현

- 문제 단순화
  - 스레드는 합의를 이룰 때 까지 계산하면서 이동(move)한다.
    - Wait-free이므로 언젠가는 이동이 끝난다.
      - A와 B의 decide호출이 완료된다.
      - 이 시점에서는 return값이 1또는 0으로 결정되어 있다.
    - 이동할 때만. 전체 실행 상태가 변경될 수 있다.
      - 상태 : return 값의 결정 영향을 미치는 메모리나 레지스터의 값
    - 이동이 아니면 상태가 변경되지 않는다.
      - 이동 결과에 의해서만 상태가 변경된다.
      - 결정은 A,B가 같은 값을 return 하게 되는 것이기 때문에, 결정되지 않은 상태에서 이동없이 결정이 되었다면 결정이 되었다는 것을 상대 쓰레드에 알려줄 방법이 없다.

합의 수

- 문제 단순화
  - 초기상태 : 아무런 이동이 없는 경우
  - 최종상태 : 모든 스레드들이 이동을 마친 상태 (프로토콜의 Leaf)
    - Decide 메소드가 결정된 값을 리턴한다.
- 일가(univalent) 상태
  - 앞으로 어떠한 이동을 하더라도 결정 값의 변화가 없는 경우
- 이가(bivalent)상태
  - 최종 결정 값이 결정되지 않은 상태
- 임계(critical)상태
  - 현재 상태가 이가이다.
  - 다음의 이동으로 무조건 일가 상태가 된다.
    - 두개의 child가 모두 일가 상태이다.

일가 상태 이가 상태

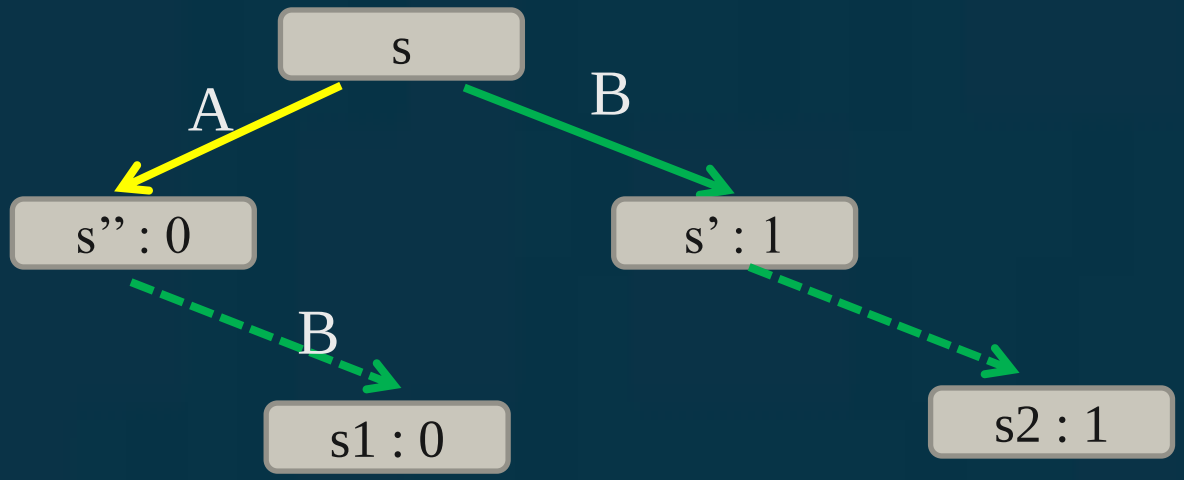
- 보조 정리 (교재 참조)
  - 모든 2-스레드 합의 프로토콜의 초기상태는 이가이다.
    - A가 0, B가 1을 합의시킬 경우
    - A만 실행하면 0, B만 실행하면 1을 결정해야 한다. => 실행 순서에 따라 결과가 바뀐다 => 따라서 일가가 아니다
  - 모든 무대기 합의 프로토콜은 임계 상태가 반드시 존재한다.
    - 트리의 높이는 유한하다.
    - 마지막 층의 노드들은 모두 1가이다.
    - 마지막 2가 노드가 존재하는 높이가 있다.
    - 그 높이에 존재하는 모든 2가 노드는 임계상태이다.

증명

- 명제 : Atomic 메모리로 2 쓰레드 합의 문제를 해결할 수 있다.
- 증명 : (교재 참조)
  - Atomic 메모리로 구현했다고 가정하면?
    - decide를 알고리즘으로 구현했고 protocol을 그릴 수 있다.
    - 임계상태가 반드시 존재하며 그 때 가능 한 이동 시나리오는
      - A : Read, B : Any
        - 이 상태에서 A thread는 공유메모리를 읽으려고 하고 있고, B thread는 공유메모리를 읽거나 쓰려고 하고 있다.
      - A : write r0, B : write r1 (r0 != r1)
      - A : write r, B : write r

증명

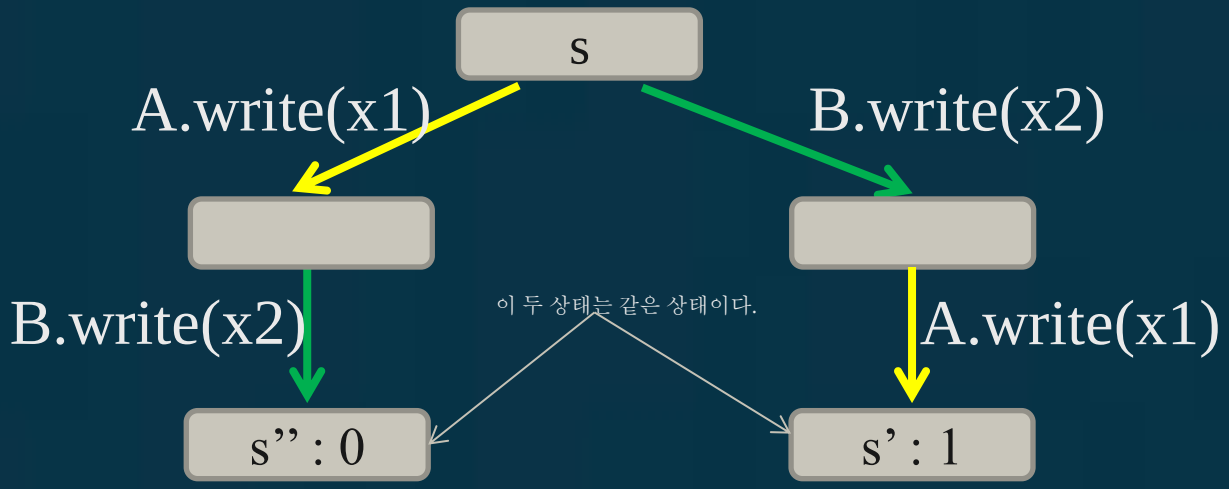
- A : read, B : write or read
  - 예상 과정 ->



  - "s 상태에서 B만 수행" 과 "s''상태에서 B만 수행"
  - A가 읽기 연산을 했는지 안했는지 B는 알 수 없다.
    - B가 알 수 없는 A의 내부상태에서만 변경 되기 때문에 B는 s'과 s''를 구분할 수 없으므로 B는 결과 값을 결정할 수 없다. 따라서 이러한 경우는 존재할 수 없다. (임계상태에서 A가 read일 수 없다.)

증명

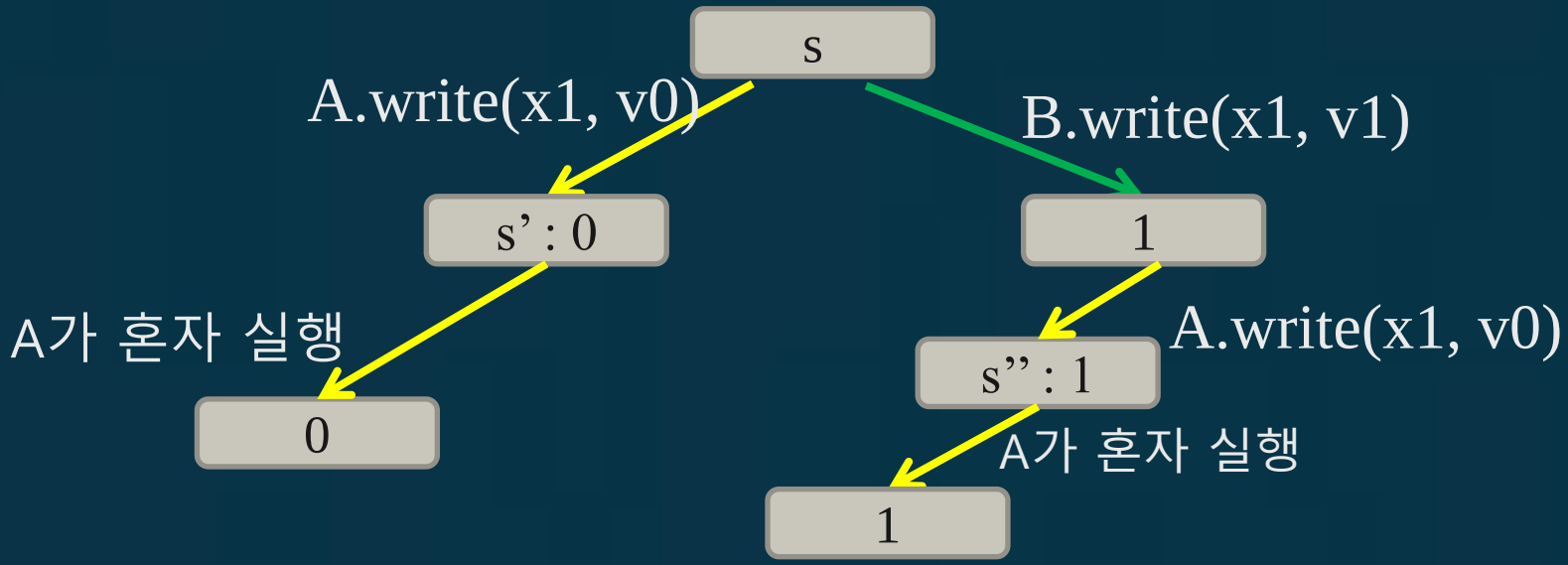
- A : write(x1), B : write(x2)
  - 예상 과정 ->



  - s''와 s'는 모든 것이 똑같은 같은 상태이다. 따라서 최종 값은 같아야 한다.
  - 이 경우 A, B 둘 중 누가 먼저 실행되었는지 구분할 수 없다. 그런데, s가 임계상태 이므로, 최종결과는 0또는 1이 되야 하는 모순이 있다. (이러한 임계 상태 s는 존재할 수 없다.)

증명 (2024)

- A : write(x1), B : write(x1)
  - 예상 과정 ->



  - A의 입장에서 s'와 s''을 구분할 수 없으므로(A는 B의 기록을 덮어씌운다.) 이러한 임계상태는 존재할 수 없다.

증명

- 증명
  - A : Read, B : Any
    - A가 읽었는지 안 읽었는지 모르는 채로 B는 0일지 아닌지를 구분해야 한다.
  - A : write r0, B : write r1 (r0 != r1)
    - 어느 순서로 썼는지 구분할 수 없다.
  - A : write r, B : write r
    - A가 썼는지 안썼는지 알 수 없는 채로 B는 0과 1을 결정해야 하는 경우가 생긴다.
      - A의 write r을 B의 write r이 덮어 쓴 경우와 아닌 경우를 B는 구분할 수 없다.

증명

- 증명
  - 어떠한 경우든 모순으로 인해 임계영역이 존재할 수 없다. 따라서 Read/Write에 의한 이동만으로는 2스레드 합의가 불가능 하다.
- 결론
  - atomic 메모리의 read/write 만으로는 두개의 스레드에서 합의를 non-blocking으로 구현하는 것이 불가능하다.

증명

- 보충
  - Non Blocking 프로그램은 언젠가는 결과를 정해야 하는데, Atomic Memory의 READ/WRITE만으로는 정해진 결과를 모든 쓰레드에게 알리는 것이 불가능하다.
    - Wait-free이므로 결과를 정한 쓰레드가 이후의 공유메모리 연산으로 결과를 알려주지 않아도 다른 쓰레드들은 결과를 알아야 함.

FIFO QUEUE

- Atomic 메모리로 2쓰레드 합의객체를 구현할 수 없다면 무엇이 필요한가?
- Queue객체가 있으면 해결이 가능한가?

- 가상의 Queue를 가정하자
  - 2 Dequeuer Queue
  - 2개의 Thread에서 동시에 Deque를 했을 때 atomic하고 wait-free하게 동작하는 queue

FIFO QUEUE

- 가정 : Dequeuer가 2인 QUEUE가 있다면?
- 결론 : 적어도 2의 합의 수 갖는다.

```
#define WIN 0
#define LOSE 1
class QueueConsensus : public ConsensusProtocol {
private:
        Queue   queue;
public:
        int proposed[2];
        QueueConsensus() { queue.enq(WIN); queue.enq(LOSE); }
        int Decide(int value) {
                int i = Thread_id();
                proposed[i] = value;
                int status = queue.deq();
                if (WIN == status ) return proposed[i];
                else return proposed[1-i];
        }
};
```

FIFO QUEUE

- 그래서?
  - Atomic 메모리의 합의수는 1
  - 2 Dequeuer QUEUE의 합의수는 적어도 2
  - 결론 : 2 Dequeuer QUEUE는 atomic 메모리로 구현 불가능
- 결론
  - atomic 메모리만 가지고는 큐, 스택, 우선순위 큐, 집합, 리스트등의 무대기 구현을 작성할 수 없다.

FIFO QUEUE

- 하나 더
  - n thread FIFO Queue의 합의 수는 2이다.
- 증명 (1/3)
  - A,B,C 세개의 스레드가 합의가능한 프로토콜이 있다고 가정하자 (합의수가 3이상이라고 가정)
  - S라는 임계 상태가 반드시 존재한다.
  - S에서 A가 이동하면 0인상태, B가 이동하면 1인상태로 간다고 가정하자
    - 역도 마찬가지, 0과 1만 갖는 간단한 문제를 가정

FIFO QUEUE

- 증명 (2/3)
  - A,B의 이동은 같은 객체에 대한 호출이다.
    - 다른 객체에 대한 호출은 프로그램적으로 순서를 구분 할 수 없으므로 임계 상태가 될 수 없다.
  - 그 객체는 Queue다
    - read/write로는 앞의 증명에 의해 합의 수 2 이상을 구현할 수 없다.

FIFO QUEUE

- 증명 (3/3)
  - A:deq, B:deq인 경우
    - A가 deq하자마자 B가 deq한 경우, B가 deq하자마자 A가 deq한 경우
    - C는 위의 두 경우를 구별할 수 없다.
  - A:deq, B:enq인 경우
    - C가 볼 때 A가 먼저 수행되나 B가 먼저 수행되나 똑 같다 (구별할 수 없다)
  - A:enq, B:enq인 경우
    - A가 enq하는 것을 a, B가 enq하는 것을 b라 하자.
    - 큐에 a,b가 들어가는 경우와 b,a가 들어가는 경우 두 가지가 있다.
    - 이 때 A와 B를 각각 queue의 내용을 읽을 때 까지만 실행한다.
      - A와 B는 (a,b)의 순서를 알기 전까지는 결정 결과를 알 수 없다.
      - 따라서 A만 실행하면 결정결과를 모르는 채로 (a, b)의 dequeue를 시도하게 된다.
      - (a, b)를 A가 하나 deq한 즉시 A를 멈추고 B를 실행시켜 B가 나머지를 deq한 순간 B를 멈춘다.
      - => queue에 들어간 a와 b가 사라지고, 결정에 대한 아무 정보도 남지 않는다.
    - 이 후 C가 실행된다면 C는 위의 두 가지 경우를 구별할 수 없다.

FIFO QUEUE

- 결론
  - Wait-Free Queue로는 3개 쓰레드 합의 객체를 만드는 것이 불가능하다.
  - 따라서, Wait-Free Queue의 합의 수는 2이다.

다중 대입 객체

- 그렇다면, n개 스레드의 합의 객체를 구현할 수 있는 합의수 n의 동기화 객체가 존재하는가?

- 존재한다.

- 다중 대입 객체 : 배열로 구성되며 복수의 원소를 atomic하게 변경할 수 있는 객체

다중 대입 객체

- 정의 : (m,n)-대입 문제
  - 멤버로 Size가 n인 배열을 가짐
  - Assign() 메소드 (원자적으로 수행, wait-free)
    - 매개변수로 m개의 값과 m개의 인덱스를 받는다.
    - 값들을 배열의 해당 인덱스에 대입한다.
  - Read() 메소드는 인덱스 i를 받아 i번째 값을 반환한다.
- 스냅샷의 반대 기능
  - 스냅샷 : One-write, multiple-read
  - 읽을 때 배열전체의 값을 원자적으로 읽음.
  - 스냅샷은 원자적 메모리로 구현 가능하므로 합의 수 1이다.

다중 대입 객체

- 예) 잠금 기반의 (2,3) 대입 객체 구현

```
class Assign23 {
private:
        int r[3];
        mutex AL;
public:
        void assign(int val0, int val1, int i0, int i1) {
                AL.lock()
                r[i0] = val0;
                r[i1] = val1;
                AL.unlock();
        }
        int read(int i)  {
                AL.lock();
                int val = r[i];
                AL.unlock();
                return val;
        }
};
```

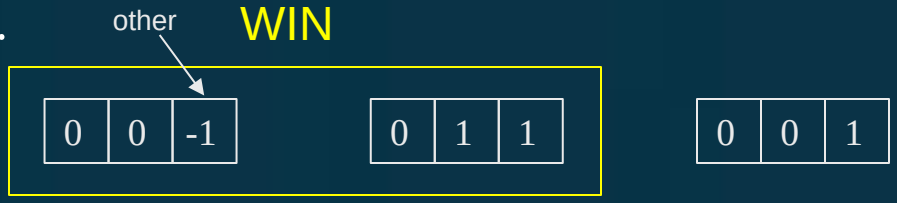
다중 대입 객체

- 원자적인 레지스터를 이용하여 (m, n)-대입 객체를 대기 없이 구현하는 것은 불가능하다.(n>m>1)
  - 증명 : (2,3)-대입 객체로 2 스레드의 이진 합의를 풀 수 있다.
    - 따라서 (2,3)-대입 객체를 원자적인 레지스터로 구현하는 것은 불가능하다.

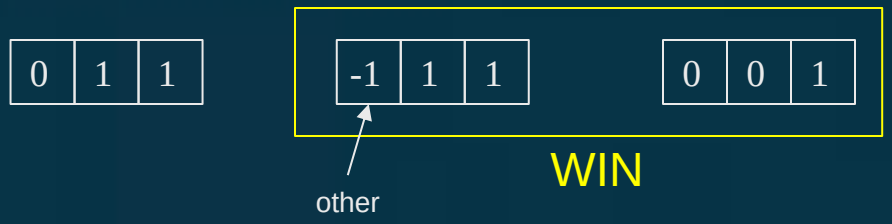
```
class MultiConsensus :public ConsensusProtocol {
private:
        Assign23 assign23;
         int proposed[2] = {INIT, INIT};
public:
        MultiConsensus() { assign23.init(INIT); }
        int decide(int value) {
                int i = Thread_id();
                proposed[i] = value;
                int j = 1 - i;
                assign23.assign(i,i,i,i+1); // 값1, 값2, 인덱스1, 인덱스2
                int other = assign23.read((i+2) % 3);
                if (other == INIT || other == assign23.read(1))
                        return proposed[i];
                else return proposed[j];
        }
};
```

다중 대입 객체

- 증명 : (2,3)-대입 객체로 2 스레드의 이진 합의를 풀 수 있다.
  - 먼저 실행된 스레드의 값을 합의값으로 하면 된다.
  - Thread 0이 보았을 때 객체가 가질 수 있는 상태는 다음 3개 뿐이다.

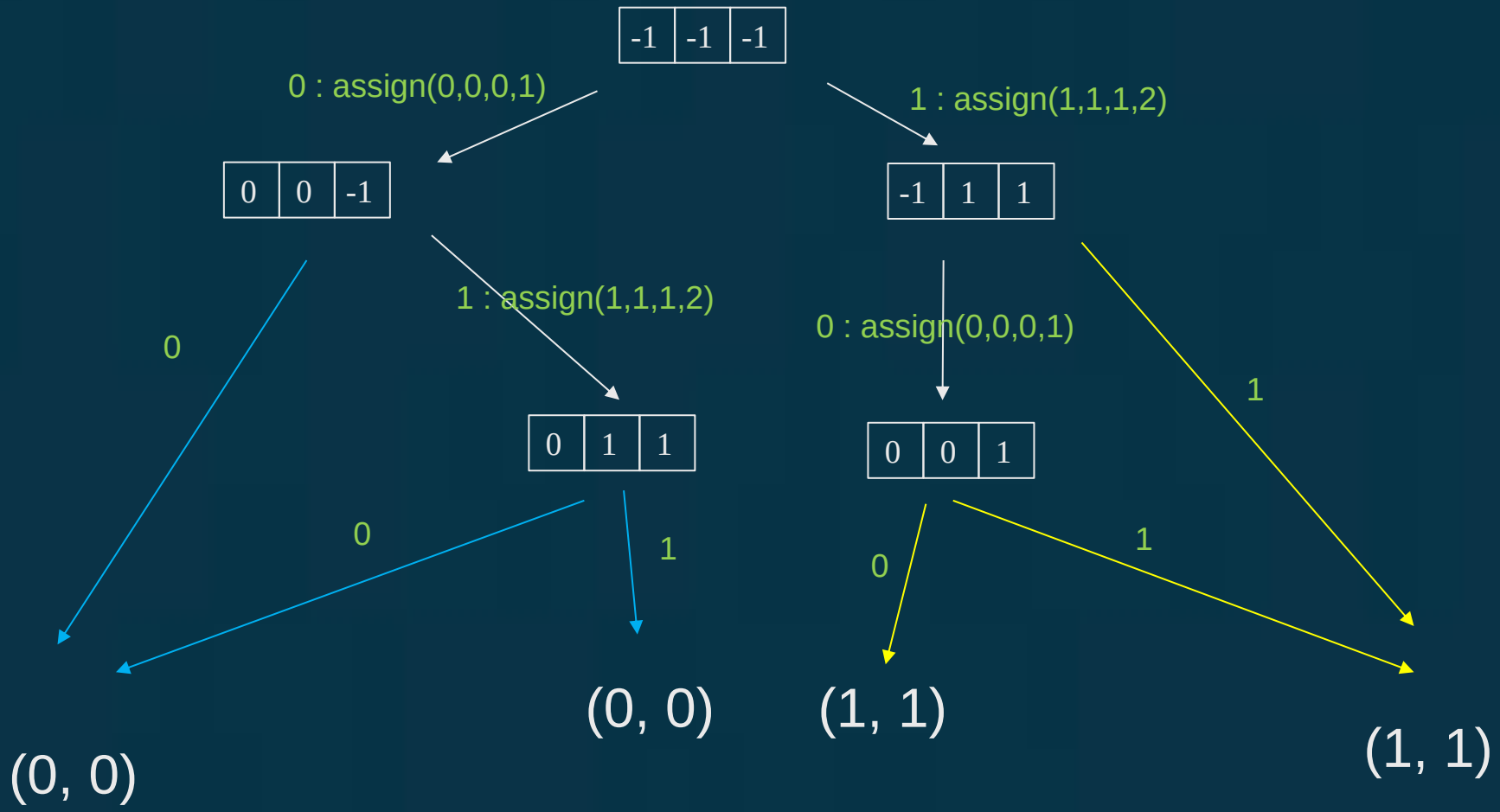


  - Thread 0의 입장에서는 i[2]가 INIT이거나 i[1]이 1인 경우 Thread0의 assign이 먼저 실행된 것이다.
  - 마찬가지로, Thread 1의 입장에서는 i[0]가 INIT이거나 i[2]가 0인 경우 Thread1의 assign이 먼저 실행된 것이다.



다중 대입 객체

- 프로토콜


-1 -1 -1
0 : assign(0,0,0,1)
1 : assign(1,1,1,2)
0 0 -1
-1 1 1
1 : assign(1,1,1,2)
0 : assign(0,0,0,1)
0
1
0 1 1
0 0 1
0
1
0
1
(0, 0)
(0, 0)
(1, 1)
(1, 1)

다중 대입 객체

- 최종 결론
- 원자적인 (n, n(n+1)/2)-대입 객체(n > 1)는 최소 n의 합의 수를 가진다.
- 따라서, 대입 객체들을 HW적으로 지원하면 합의문제를 무대기로 해결할 수 있다.
  - HW구현 비용이 너무 크다.
  - 대안은? RMW

동기화 연산들

- RMW (read-modify-write)연산
  - 하드웨어가 지원하는 동기화 연산의 한 종류
  - 특수 명령어가 반드시 필요 (wait-free가 되기 위해서는)
- 메소드 M은 함수 f에 대한 RMW이다.
  - 메소드 M이 원자적으로 현재의 메모리의 값을 v에서 f(v)로 바꾸고 원래 값 v를 반환한다.

동기화 연산들

- RMW 연산의 종류 (원래 값이 x인 경우)
  - GetAndSet(v) : f(x) = v
  - GetAndIncrement : f(x) = x+1
  - GetAndAdd(k) : f(x, k) = x + k
  - compareAndSet(x, e, u)
    - f(x, e, u) = u(if x = e) , x(if x != e)
    - 메모리의 값이 e면 u로 바꾼다. 리턴 값은 메모리의 원래 값
  - get()
    - 항등 함수 : f(x) = x
- 항등 함수가 아닌 함수를 지원할 때 그RMW를 명백하지 않은(nontrivial)이라고 한다.

동기화 연산들

- 정리:
  - 임의의 명백하지 않은 RMW연산은 합의 수가 최소한 2이다.

```
class RMWConsensus: public ConsensusProtocol {
private:
          RMWobject r;
          int proposed[2] = {INIT, INIT};
public:
          RMWConsensus() { r.init(INIT); }
          int decide(int value) {
                    int i = Thread_id();
                    proposed[i] = value;
                    int j = i - 1;
                    if (r.rmw() == INIT) return proposed[i];
                    else return proposed[j];
          }
};
```

  - => 원자적 메모리만을 사용해선 2개나 그 이상의 스레드에 대한 어떠한 명백하지 않은 RMW도 구현할 수 없다.

동기화 연산들 (2023 화금)

- Common2 RMW연산
  - 많은 RMW연산이 여기에 속한다.
- 정의
  - 함수 집합 F는 모든 값 v와 F에 속하는 모든 함수 f_i와 f_j에 대해 다음이 성립하면 Common2라고 한다.
  - fi와 fj는 교환이 가능하거나 : $f_i(f_j(v)) = f_j(f_i(v))$
  - 한 함수가 다른 함수를 덮어쓰는 경우 : $f_i(f_j(v)) = f_i(v)$ 이거나 $f_j(f_i(v)) = f_j(v)$
- 특징
  - 합의수 2를 갖는다.
  - 최근의 프로세서들에서는 제거되는 추세이다.
- 예
  - getAndSet() : 덮어 쓴다.
  - getAndIncrement() : 교환이 가능하다.

동기화 연산들

- Common2 RMW 레지스터의 합의 수는 2이다.
  - 두개의 쓰레드 A, B에 대한 임계영역이 존재하고, 같은 객체에 대한 RMW연산이다.
  - 3스레드일 경우, A 스레드(합의 과정 승자) 는 항상 자신이 제일 처음이었다는 것을 알 수 있어야 하고, B와 C 스레드는 자신이 패자라는 것을 알 수 있어야 한다. 그러나 상태를 정의하는 함수들은 Common2의 연산을 따르므로 교환이나 덮어쓰기가 가능하여, C 스레드가 나머지 중 어느 스레드가 먼저 실행되었는지 판단할 수 없다. 또한, 프로토콜은 무대기(wait-free)이므로, 승자를 확인하려고 기다릴 수도 없다.

동기화 연산들

- CAS(Compare And Set)연산
  - CAS(expected, update)
    - 레지스터의 값이 expected면 update로 바꾸고 true를 리턴
    - 레지스터의 값이 expected가 아니면 false를 리턴
  - CAS는 무한대의 합의 수를 갖는다.

```
class CASConsensus {
private:
        int FIRST = -1;
        AtomicInt r = FIRST;
public:
        value decide(value v) {
                propose(v);
                int i = thread_id();
                if (r.CAS(FIRST, i)) return proposed[i];
                else return proposed[r];
        }
```

정리

- Atomic 메모리 read/write만을 사용해서는 멀티쓰레드에서 무대기인 일반적인 자료구조를 구현할 수 없다.
- CAS가 무한대 합의수를 갖는다.
  - 임의의 합의수를 갖는 자료구조를 구현할 수 있는 희망을 가진다.
  - 다음에 증명할 사항 : 합의수 무한대인 동기화 연산으로 모든(n개의 쓰레드에서 non-blocking인) 자료구조를 구현할 수 있는가?

목차

- 합의 객체
- 합의수
  - non-blocking 알고리즘을 만드는 능력
- 만능성
  - 모든 알고리즘을 멀티 쓰레드 무대기로
- Lock
  - 효율적인 Lock의 구현

내용

- 합의의 의미
- 만능성
- 무잠금 만능 구성
- 무대기 만능 구성

합의의 의미

- 모든 무대기(wait free) 동기화 객체는 합의 수(consensus number)라는 능력의 차이가 있다.
- 적은 합의 수를 갖는 객체로 큰 합의 수 객체를 구현할 수 없다.

합의 수	객체
1	원자적 메모리
2	getAndSet(), getAndAdd(), 큐, 스택
m	(m, m(m+1)/2)-대입 객체
무한대	메모리 이동, compareAndSet(), LL-SC

합의의 의미

- 의의
  - 불가능한 시도를 미연에 방지할 수 있다.
    - 예) 원자적 메모리를 가지고 4개 스레드 무대기 병렬 큐를 작성하려 하는 행위
  - 구현 가능한 방법을 알고, 왜 그것이 구현 가능한지를 안다면 이를 최적화 할 때 더 잘 할 수 있다.

만능

- 모든 자료구조의 무대기 동기화가 가능 한가?
  - 그렇다! 가능하다.
- 만능 객체
  - 어떠한 객체든 무대기 병렬객체로 변환시켜 주는 객체
    - 예) 싱글 스레드에서만 돌아가는 큐를 무대기 병렬 큐로 변환 시켜 줄 수 있다.
  - n개의 스레드에서 동작하는 만능객체는 합의 수 n이상의 객체만 있으면 구현 가능하다.
    - 무한대의 합의 수 객체 CAS를 사용하면 쓰레드개수에 상관없이 만능 객체를 구현할 수 있다.

만능

- 만능의 정의
  - 클래스 C 객체들과 원자적 메모리로 모든 객체를 무대기 구현으로 변환하는 것이 가능하다면 클래스 C는 만능이다.
    - 클래스 C로 모든 객체를 무대기로 변환 가능한데, 직접적으로 변환하지 않고 원자적 메모리를 사용해서 대상 객체를 약간 변형한 후 변환한다.
    - 변형 : 여러 개의 메소드와 파라미터, 리턴값을 통합
- 일단 무잠금 만능객체를 알아보고 그것을 무대기 만능객체로 변형한다.

무잠금 만능 구성

- 준비(변형)
  - 순차 객체 A 가 있고 이를 n-thread상에서 무대기로 구현하려고 한다.
  - 조건 : A는 결정적이다.(deterministic)
    - 모든 객체의 초기상태는 항상 같은 상태이다.
    - 같은 상태에서 같은 입력을 주면 항상 같은 결과와 같은 완료 상태가 나온다.

      => 초기 상태에서 같은 입력 값을 동일한 순서로 입력하면 항상 같은 결과가 나온다.

      - 입력의 순차적인 리스트를 로그(log)라고 한다.

무잠금 만능 구성

- 준비
  - 순차 객체 A
    - 병렬화 하고자 하는 객체를 감싼 객체
    - 호출 메소드를 apply하나로 통일

```
class SeqObject {
public:
        Response apply(Invocation invoc};
};
```

  - Invocation 객체
    - 호출하고자 하는 원래 객체의 메소드와 그 입력값을 갖는 객체
  - Reponse
    - 여러메소드 들의 결과 값의 타입 을 압축한 객체

무잠금 만능 구성

- 순차 객체의 예

```
enum MethodType { DEQUE, ENQUE, CLEAR };
typedef int InputValue;
typedef int Response;

class Invocation {
  MethodType type;
  InputValue v;
};

class SeqObject_Queue {
   queue <int> m_queue
public:
   Response apply(Invocation invoc)
   {
     int res = -1;

     if (ENQUE == invoc.type) m_queue.enque(invoc.v);
     else if (DEQUE == invoc.type) res = m_queue.deque();
     else if (CLEAR == invoc.type) m_queue.clear();
     return res;
   }
};
```

무잠금 만능 구성

- Log
  - Log는 Node들의 리스트이다.

```
class NODE
{
public:
    Invocation invoc;
    Consensus decideNext;
    NODE *next;
    volatile int seq;

    NODE() { seq = 0; next = nullptr; }
    ~NODE() { }
    NODE(const Invocation &input_invoc)
    {
        invoc = input_invoc;
        next = nullptr;
        seq = 0;
    }
};
```

무잠금 만능 구성

- Log
  - Consensus
    - 합의 객체
    - Node를 입력으로 받아 그 중 한 Node를 선발
  - 로그는 Node의 링크드 리스트이다.
    - 순차객체 A의 초기값이 일정하므로 순차객체 A의 모든 상태를 A와 Log를 조합해서 알 수가 있다.

무잠금 만능 구성

```
class LFUniversal {
private:
        Node *head[N],  Node tail;
public:
        LFUniversal() {
                tail.seq = 1;
                for (int i=0;i<N;++i) head[i] = &tail;
        }
        Response apply(Invocation invoc) {
                int  i = Thread_id();
                Node prefer = Node(invoc);
                while (prefer.seq == 0) {
                        Node *before = tail.max(head);
                        Node *after = before->decideNext->decide(&prefer);
                        before->next = after; after->seq = before->seq + 1;
                        head[i] = after;
                }
                SeqObject myObject;
                Node *current = tail.next;
                while (current != &prefer) {
                        myObject.apply(current->invoc);
                        current = current->next;
                }
                return myObject.apply(current->invoc);
        } };
```

무잠금 만능 구성

- 무잠금 만능 구현
  - 지금 까지 객체 가해진 모든 메소드 호출의 리스트인 Log를 보관
    - tail부터 시작하는 Node의 리스트
  - 새로운 호출이 오면 Node를 생성한 후 Log의 head에 덧 붙인다
    - 합의 객체 활용
  - A 객체를 생성한 후 Log에 있는 Invocation을 새로운 호출까지 적용시키고 그 결과를 반환한다.

무잠금 만능 구성

- Head추가의 정당성

```
int  i = Thread_id();
Node prefer = Node(invoc);
while (prefer.seq == 0) {
// prefer가 성공적으로 head에 추가 되었는지 검사
      Node *before = tail.max(head);
// Log의 head를 찾지만 다른 스레드와 겹쳐져서 잘 못 찾을 수도 있음
      Node *after = before->decideNext->decide(&prefer);
// before의 합의는 항상 유일하다!
      before->next = after; after->seq = before->seq + 1;
// 여러 스레드가 같은 작업을 반복 할 수 있지만, 상관없다.
      head[i] = after;
// 자신이 본 제일 앞의 head는 after니까 업데이트 시켜준다
// 이러지 않으면 동일한 합의 객체를 두 번 호출할 수 있다.
// 운좋게 after가 prefer가 되면 성공이다!
}
```

무잠금 만능 구성

- 구현의 트릭
  - 노드 마다 합의 객체를 갖고 있다.
    - 하나의 스레드는 HEAD배열을 통해 한번 호출한 합의 객체는 다시 호출하지 않도록 한다.
  - 순차객체 A는 매 호출마다 새로 생성된다.
    - 다른 스레드는 절대 그 순차객체를 호출하지 않는다.
  - 합의(Consensus)객체로 인해 한 Node의 Next Node는 어떤 스레드에서도 유일하다는 것을 보장 받는다.

무잠금 만능 구성

- 왜 무잠금인가?
  - 유한 스텝에 끝난 다면?
    - 당연히 무잠금
  - 무한히 실행된다면 (끝나지 않는다면)
    - 다른 노드에서 계속 HEAD 배열을 업데이트 하고 있다.
    - 하지만 누군가는 계속 실행되고 있으므로 무잠금

무대기 만능 구성

```
class WFUniversal {
private:
        Node *announce[N];
        Node *head[N];
        Node tail;
public:
        WFUniversal() {
                tail.seq = 1;
                for (int i=0;i<N;++i) { head[i] = &tail; announce[i] = &tail; }
        }
        Response apply(Invocation invoc) {
                int i = Thread_id();
                announce[i] = new Node(invoc);
                head[i] = tail.max(head);
                while (announce[i]->seq == 0) {
                        Node *before = head[i];
                        Node *help = announce[((before->seq + 1) % N)];
                        Node *prefer;
                        if (help->seq == 0) prefer = help;
                          else prefer = announce[i];
                        Node *after = before->decideNext->decide(prefer);
                        before->next = after;
                        after->seq = before->seq + 1;
                        head[i] = after;
                }
                SeqObject myObject;
                Node *current = tail.next;
                while (current != announce[i]) {
                        myObject.apply(current->invoc);
                        current = current->next;
                }
                head[i] = announce[i];
                return myObject.apply(current->invoc);
        }
};
```

무대기 만능 구성

- 변경점
  - 합의할 때 자기대신 우선순위가 높은 다른 스레드의 Node를 합의 한다. (Helping)
    - 물론 자기 자신이 합의 될 때 까지 계속 합의를 시도한다.
  - Help할 객체의 스레드를 유일하고 공평하게 결정할 수 있으면 기아를 막을 수 있다.
    - Node의 seqnumber % MAX_THREAD를 Help하도록 한다.
      - 모든 스레드가 같은 결정을 한다.
      - help만 하다가 자기자신이 기아상태에 빠지는 것을 막는다.

무대기 만능 구성

- 옳음 증명 (교재 번역 미스, 원본도 오타…)
  - Log에 노드가 누락 되는가? No
    - 모든 apply는 자신의 Node가 추가될 때 까지 실행
  - Log에 한 노드가 여러 번 추가 될 수 있는가?
    - 스레드 A, B가 노드 a를 동시에 추가하려고 한다.
    - 같은 자리에 추가 할 경우 => 합의가 해결
    - 다른 자리에 추가 할 경우
      - 다음 페이지 참조

무대기 만능 구성

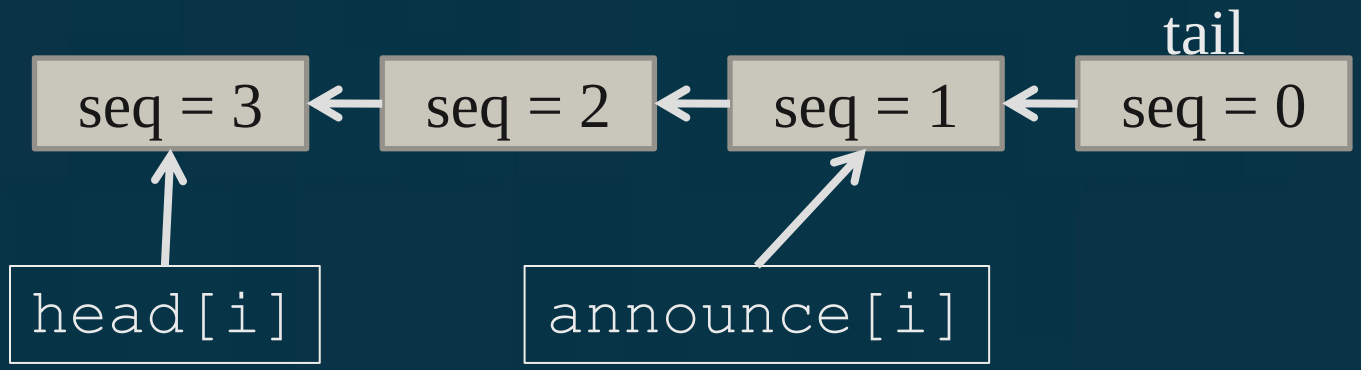
- 쓰레드 A, B가 동시에 node a를 추가할 수 있는가?
  - 같은 자리에 a가 두번 추가 되는 것은 상관없다. 단지 덮어쓰기일 뿐
  - 다시 말해서 a가 다른 두 개의 자리에 추가될 수 있는가?
    - “이 노드는 두 번째 추가 전에 적어도 한 번은 head[]에 추가되어야 한다.”
    - 먼저 추가된 a가 있고 뒤에 추가된 a’가 있다는 이야기
    - 먼저 추가된 a가 어디선가 한번은 before가 된 적이 있다는 이야기
    - before는 head[]에서 꺼내는 수 밖에 없기 때문에 적어도 한번 어디에서선가 head[]에 a가 들어간 적이 있다.
    - 따라서 head[]가 정해 지는 시점에서 a.seq != 0
      - anounce.seq나 help.seq 검사를 통과할 수가 없다.

무대기 만능 구성

- 정리하면
  - 두 개의 쓰레드에서 a를 다른 위치에 넣으려고 한다. 이 때 앞서 하나의 쓰레드가 a를 넣었고, 쓰레드 X가 뒤를 이어서 a를 넣으려고 한다.
  - 쓰레드X에서 before는 head[X]에서 왔다.
    - 이 때 head[X]는 a이거나 a의 후계자이다. (두 번째로 다른 위치에 넣으려는 것이기 때문)
    - X가 추가하려는 노드 a (즉 prefer)가
      - help인경우. : 불가능, help를 얻기 전에 이미 help.seq(=a.seq)가 0이 아니기 때문
        - 왜냐 하면 a는 이미 어떤 쓰레드에서 적어도 한번 head[]에 들어간 적이 있다. 왜냐하면 head[X]는 a이거나 a의 후계자여야 하기 때문,
        - [보조정리 1] head[x] == a이면 당연히 a.seq != 0, head[x]가 a의 후계이면 a가 어디선가 before가 되서 decide를 실행한 적이 있다. before는 head[]에서만 올 수 있으므로 a.seq != 0
        - 11번을 실행 할 때 이미 a.seq != 0 이므로 13번에서 걸린다. (모든 head[].seq는 0이 아니다 && a는 head[]에 들어간 적이 있다.)
      - announce[x]인 경우. : X == A를 뜻한다. (A는 a를 만든 쓰레드)
        - before=head[X]이고, head[X]에 쓰는 것은 X(자기자신) 뿐이다. 10번의 테스트를 통과하기 이전에 이미 한번 head[x]에 썼고 그것은 9번 아니면 20번이다.
          - before에 쓴 head[x]는 a이거나 a의 후계이기 때문에 9,20번에서 이미 a.seq != (announce[x] == a) , 따라서 10번 통과는 불가능 ([보조정리 1] 번에 의해)

무대기 만능 구성

- 옳음 증명
  - #28의 head[i] = announce[i]는 문제 없는가?
    - announce[i]가 Log에서 head[i]이후면 문제 없음
      - head를 전진시켜서 더 최신의 값을 가리키므로 문제 없음
    - head[i]가 announce[i] 이후 노드를 가리킬 수가 있는가?
      다시말해서 head[i]를 후퇴시키는 경우가 있는가? <그림 참조> 있다.
      - announce이후 MAX를 하기전에 다른 쓰레드에서 마구 추가한 경우
      - decide가 알아서 하므로 문제는 없다. 단지 while루프를 몇 번 더 돌아서 비효율적일 뿐이다.



무대기 만능 구성

- 옳음 증명
  - #28의 head[i] = announce[i]는 왜 있는가?
  - 나중에 증명할 때 유용하게 쓰인다. <보조 정리 6.4.4>

무대기 만능 구성

- 무대기 증명
  - Thread i가 기아가 되려할 때
  - 다른 실행 중인 스레드들 중 하나는 head[K]를 반드시 거쳐야 한다.
    - K MOD N == i
  - 따라서 스레드 i는 기아가 아니다.
  - head[k]를 거치지 않는 다면? 무잠금이 아니다.

정리

- 어떤 객체이던 무대기 병렬 구현이 가능하다.
  - 성능은 논외

과제 #8

- 만능객체를 사용하여 무잠금으로 구현된 std::set<int>를 Wait-Free만능객체를 사용하여 무대기로 구현하시오
  - LIST벤치마크 프로그램을 사용해서 성능 측정
    - 기존은 400만번 루프인데 4만번으로 루프회수를 줄일 것.
  - 속도 비교를 하시오
    - Single Thread
      - std::set<int>
    - 쓰레드 개수가 1/2/4/8/16/32
      - 수업시간에 구현한 Lock-Free Set
      - mutex를 사용해서 구현한 멀티쓰레드 std::set<int>
      - LF 만능객체로 구현된 LF std::set<int>

성긴 동기화

- 결과

	성긴동기화	세밀한 동기화	낙천적 동기화	게으른 동기화	Std::SET	LF 동기화	EBR LF
1	1364	19044	5859	2710	513	2166	2209
2	1620	18321	3534	2060	579	2242	2363
4	2625	15263	2192	1278	779	1782	1851
8	12251	15901	1711	907	1094	1335	1065
16	12057	16873	1730	914	1114	1390	1077

싱글 쓰레드 : 1350

Std::SET싱글 쓰레드 : 460

싱글 쓰레드 shared_ptr 게으른 : 18242

성능

- LIST

	싱글스레드	성긴동기화	게으른 동기화	LF 동기화
1	1781	2540	2125	2132
2		2442	1590	1677
4		2575	978	1260
8		10810	628	908

- std::set

	싱글스레드	성긴동기화	SeqObject	LF 만능(4000)
1	681	884	758	?
2		851		?
4		891		?
8		1085		?

효율적의 LOCK구현

- HW 동기화 연산을 사용해서 Lock을 구현하고 있으나, 그대로 사용하면 매우 안좋은 성능이 나오므로, 적절한 알고리즘을 통해 HW동기화 연산을 사용해야 한다.
- 어떠한 알고리즘을 사용해야 하는지 살펴보자
- 현대 CPU의 복잡성으로 인해 벤치마크 성능이 들쭉날쭉 나오므로 강의에서는 생략

차례

- 스핀락과 경쟁
- 효율적인 Lock에 대해 살펴본다.

용어

- 스핀락(Spin Lock)
  - 계속해서 잠금을 시도 하려 할 때
- 스피닝(Spinning)
  - 잠금을 획득하기 위해서 계속해서 시도하는 과정
- 멈춤(Blocking)
  - 운영체제가 현재 스레드를 멈추고 다른 스레드를 할당하는 방법

현대 다중프로세서

- Lock의 구현
  - TAS(Test And Set)를 사용
    - 빵집알고리즘 같은 원자적메모리만 사용해 구현한 Lock은 너무 비효율적이다.
    - CAS를 사용해도 되지만 Lock의 구현을 위해서는 TAS만으로 충분하다.

TAS 구현

```
bool state = false;

LONG TestAndSet(LONG input)
{
        bool old_state = false;
        return atomic_compare_exchange_strong(&state, &old_state, input);
}

void TASLock()
{
        while (TestAndSet(true)) {}
}

void ThreadFuncTAS()
{
        for(int i = 0; i <25000000/threadNum;i++)
        {
                TASLock();
                sum+=2;
                Unlock();
        }
        return 0;
}
```

TAS 속도

- Core2Duo

Threads	Second
1	0.580
2	1.58
4	3.15
8	6.00
16	10.11
32	16.16

TAS 속도

- i7 920

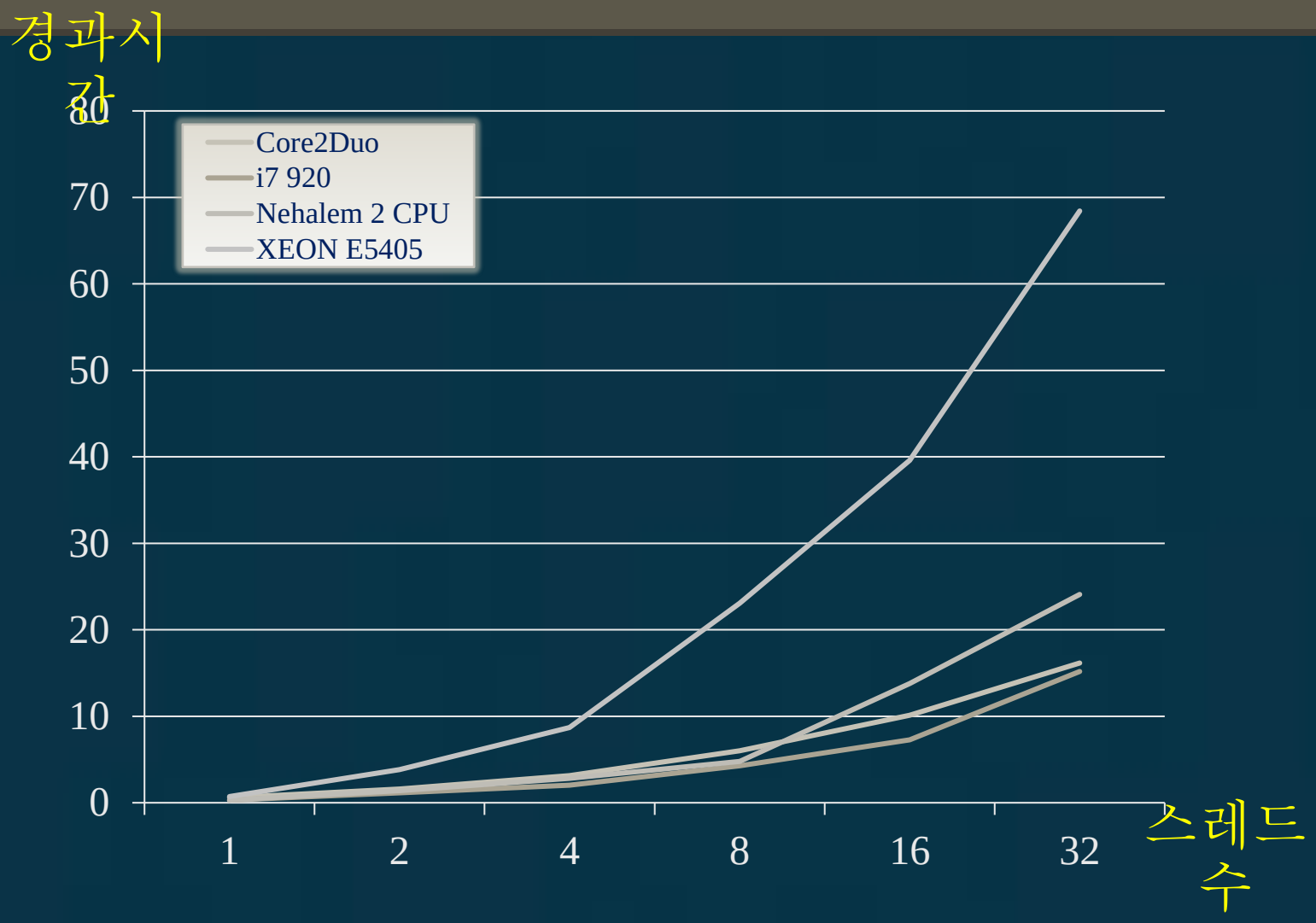
Threads	Second
1	0.30
2	1.14
4	2.03
8	4.26
16	7.26
32	15.18

TAS 속도

- Nehalem 2 CPU

Threads	Second
1	0.30
2	1.39
4	2.81
8	4.77
16	13.79
32	24.09

TAS 결과

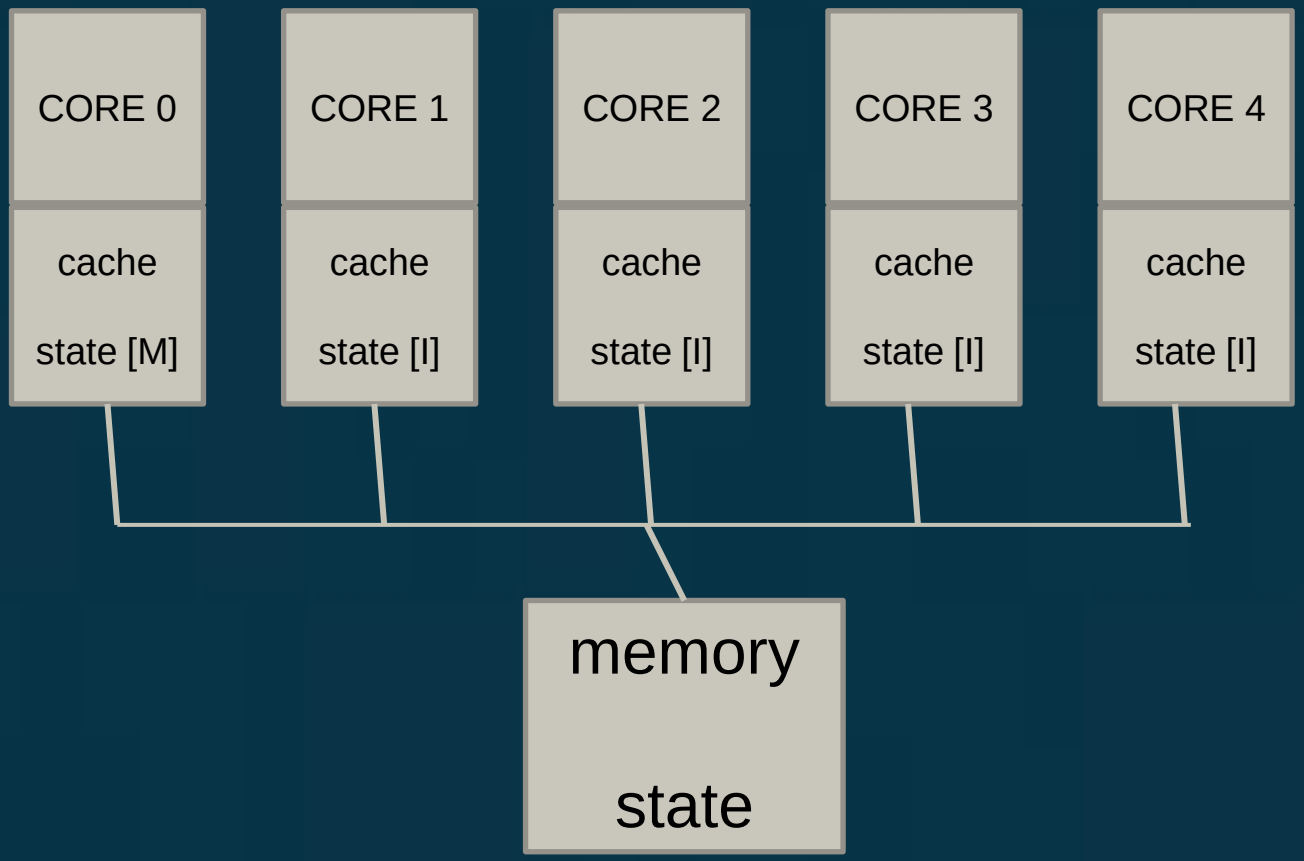

경과시간
80
70
60
50
40
30
20
10
0
Core2Duo
i7 920
Nehalem 2 CPU
XEON E5405
1
2
4
8
16
32
스레드 수

test-and-set 잠금

- 결과
  - 스레드 수가 증가함에 따라 소요시간이 계속하여 증가한다. (쓰레드의 수 ^ 2)
    - TAS 성능 저하의 원인은?.
      - 모든 스핀중인 스레드들이 거의 캐시미스가 발생하게 되고, 최신 값을 버스로부터 새로 얻어와야만 하게 만든다.
      - 스핀중에 (lock xchg 명령 실행 중) 스레드에 의해 버스가 독점되어 새로 값을 얻어오는 과정에서도 지연이 발생

test-and-set 잠금

- 원인
  - 다른 core에 cache있는 state변수를 무효화(Invalidate)시키고, 메모리에서 state를 읽어와야 한다.
    - 무효화 : 만일 변경(Modified)상태였다면 메모리에 write back해야 한다.



test-and-set 잠금

- Cache 동기화 (Cache Coherence)
  - Core끼리 캐시에 들어있는 내용을 맞추는 시스템
    - write한 내용이 날라가지 않게, 언젠가는 write한 내용이 메모리에 반영되도록, 덮어 씌여진 내용이 살아나지 않도록 => regular메모리를 만드는 시스템
  - Intel은 MESI 프로토콜을 사용한다.
    - M : Modified = Cache line이 valid하고 내용이 변경되어 있다.
    - E : Exclusive = Cache line이 valid하고 다른 core의 cache 에는 이 주소에 해당하는 cache line이 존재하지 않는다.
    - S : Shared = Cache line이 valid하고 다른 core에도 이 주소에 해당하는 cache line이 존재할 수 있다.
    - I : Invalid = Cache line이 invalid하다.

test-test-and-set 잠금

- TTAS

```
void TTASLock()
{
      while(true) {
            while (state) {}
            if (TestAndSet(true)) continue;
            return;
      }
}
```

test-test-and-set 잠금

- TTAS의 경우는?.
  - TTAS의 Lock의 경우 캐시히트가 발생하여 버스에 부하를 줄이고 메모리 접근이 느리게 되지 않는다.
    - spin할 때 메모리 locking을 하지 않는다
      - InterlockedExchange로 검사를 하지 않기 때문에
    - 서로의 cache를 invalidate하지 않는다
      - Read만 하면서 spin하기 때문에 Cache는 공유된다.
  - 하지만 문제점은 Unlock에서 발생
    - 잠금을 해제할 때 각 스레드들은 캐시미스가 발생하고 새로운 값을 읽는다.
    - 그 후에 한스레드가 잠금을 획득하면 또 다시 캐시미스가 발생되고 새로운 값을 읽으려 하면서 버스에 소통량이 일시적으로 증가한다.

TTAS 속도

- Core2Duo

Threads	Second
1	0.60
2	1.36
4	2.31
8	4.12
16	6.87
32	9.10

TTAS 속도

- i7 920 (2.67GHz)

Threads	Second
1	0.29
2	1.94
4	2.19
8	2.88
16	5.34
32	10.24

TTAS 속도

- Nehalem E5520, 2 CPU, 2.27GHz

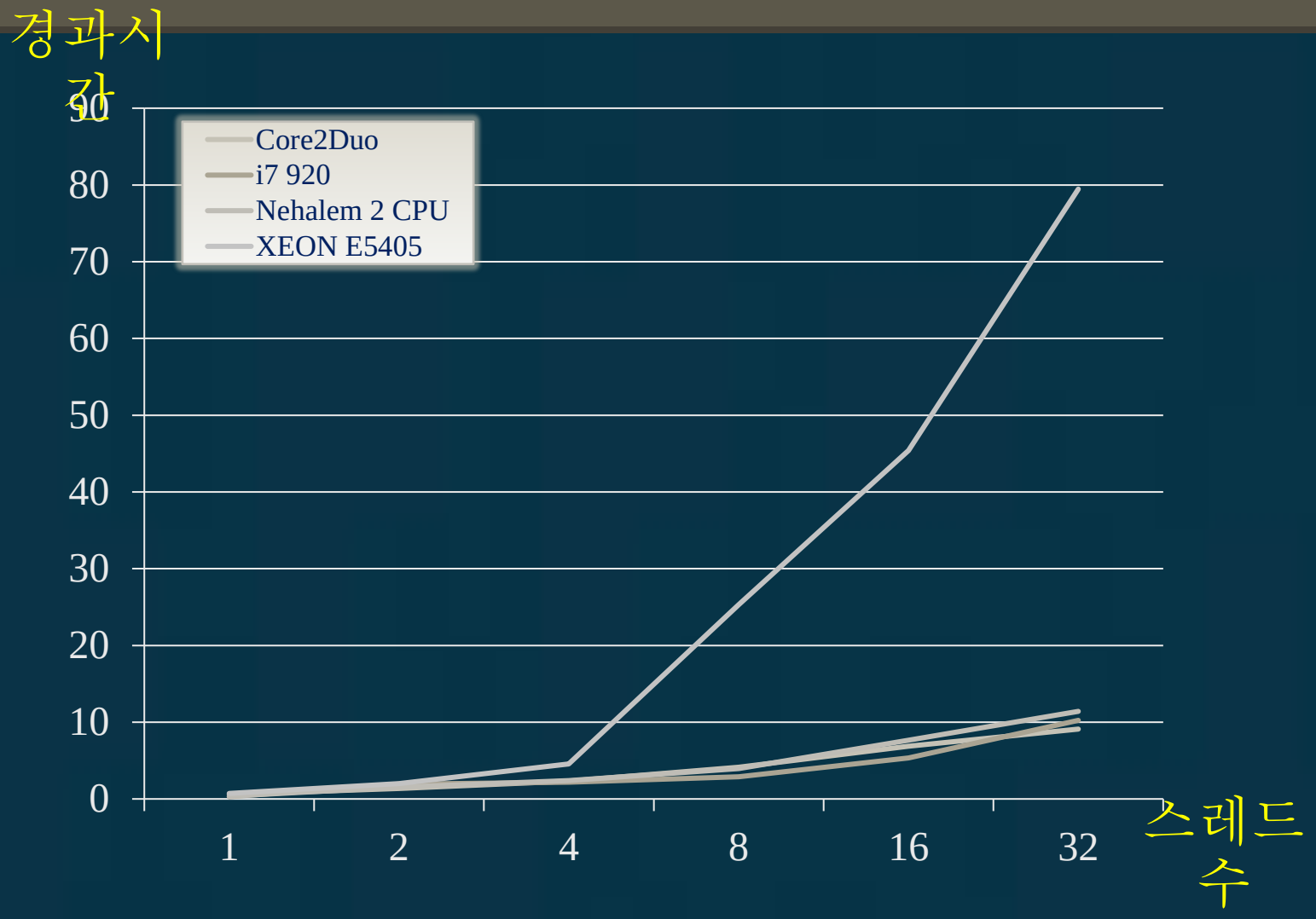
Threads	Second
1	0.36
2	1.54
4	2.39
8	3.97
16	7.66
32	11.41

TTAS 속도

- XEON (E5405, Harpertown, 2CPU)

Threads	Second
1	0.72
2	1.99
4	4.55
8	25.33
16	45.40
32	79.46

TTAS 결과


경과시간
90
80
70
60
50
40
30
20
10
0
Core2Duo
i7 920
Nehalem 2 CPU
XEON E5405
1
2
4
8
16
32
스레드 수

그래프 작성 팁

- TAS와 TTAS를 비교하는 이쁜 그래프를 그려보자.
  - 3D 그래프가 필요할 지도 모름
- 앞의 그래프를 볼 때 “Nehalem같은 경우 성능 감소 속도가 완만하니까 쓸 만 하네”라는 오해를 불러일으키기 쉽다.
  - 이런 경우 Y축으로 efficiency를 사용한다.
    - Speed up = Thread 1개 로 실행했을 때 걸린 시간 / 걸린 시간
    - efficiency = Speed up / Thread 개수

그래프 작성 팁

- 병렬 알고리즘의 속도 비교는 speed up과 efficiency를 많이 사용한다.
  - 위키피디아 참조

In parallel computing, **speedup** refers to how much a parallel algorithm is faster than a corresponding sequential algorithm.

Look up ***speedup*** in Wiktionary, the free dictionary.

Definition [edit]

Speedup is defined by the following formula:

$$S_p = \frac{T_1}{T_p}$$

where:

- p is the number of processors
- T_1 is the execution time of the sequential algorithm
- T_p is the execution time of the parallel algorithm with p processors

Linear speedup or **ideal speedup** is obtained when $S_p = p$. When running an algorithm with linear speedup, doubling the number of processors doubles the speed. As this is ideal, it is considered very good scalability.

Efficiency is a performance metric defined as

$$E_p = \frac{S_p}{p} = \frac{T_1}{pT_p}.$$

It is a value, typically between zero and one, estimating how well-utilized the processors are in solving the problem, compared to how much effort is wasted in communication and synchronization. Algorithms with linear speedup and algorithms running on a single processor have an efficiency of 1, while many difficult-to-parallelize algorithms have efficiency such as $\frac{1}{\log p}$ [citation needed] that approaches zero as the number of processors increases.

When attempting to understand parallel performance, efficiency is generally a better metric to plot than speedup, since

- all of the area in the graph is useful (whereas in a speedup curve 1/2 of the space is wasted)
- it is easy to see how well parallelization is working
- there is no need to plot a "perfect speedup" line

Engineers therefore tend to prefer it. On the other hand, marketing people prefer speedup curves because they go up and to the right.

결과

- OLD XEON에서 오히려 느려지는 이유
  - 싱글 쓰레드
    - 아키텍쳐의 차이.
      - Old Xeon은 Pentium 4기반
      - Nehalem은 Core 아키텍쳐 기반
    - FB-DIMM으로 인한 성능 차이는 거의 없음
      - 벤치마크 프로그램이 충분히 작으므로 캐시에 다 올라옴.
  - 멀티 쓰레드
    - 메모리 Bottle Neck
      - 단일 메모리 버스 : 모든 CPU가 하나의 메모리 컨트롤러에 의지
      - 느린 메모리 컨트롤러 : 네할렘은 메모리 컨트롤러를 CPU에 내장
    - 느린 캐시 동기화
      - 캐시 데이터 전송을 메모리 컨트롤러를 통해서 해야 함
      - 네할렘은 QPI 연결을 통해 CPU끼리 직접 해결
    - 따라서, 쓰레드의 개수가 1개 CPU의 코어의 개수를 넘어가면 급격히 느려진다.
      - TTAS 그래프를 보라.
  - OLD XEON에서는 TTAS가 왜 TAS보다도 느린가???

Back-Off

- Back-Off
  - 여러 스레드가 잠금을 획득하려고 하는 것은 좋은 방법이 아니다.
    - 잠금을 획득할 확률이 낮은 상태에서 잠금 획득을 시도하는 것은 버스 트래픽에 많은 무담이 된다.
    - 잠시 양보해서 다른 경쟁 스레드가 끝낼 기회를 주는 것(Back-off)이 효율적이다.

Back-Off

- 구현
  - 최소 딜레이 값과 최대 딜레이 값을 설정한다.
  - limit는 현재 딜레이 값을 제한하고 random은 0과 limit사이의 렌덤한 딜레이 값을 정해서 그 시간동안 스레드를 멈춘다.
    - 잠금을 얻는 데 실패한다면 2배로 늘린다.
  - 잠금을 획득하는데 실패하는 경우만 스레드가 양보한다.
  - 컴퓨터 별로 최적의 MaxDelay와 MinDelay를 결정하는 것은 어렵다.

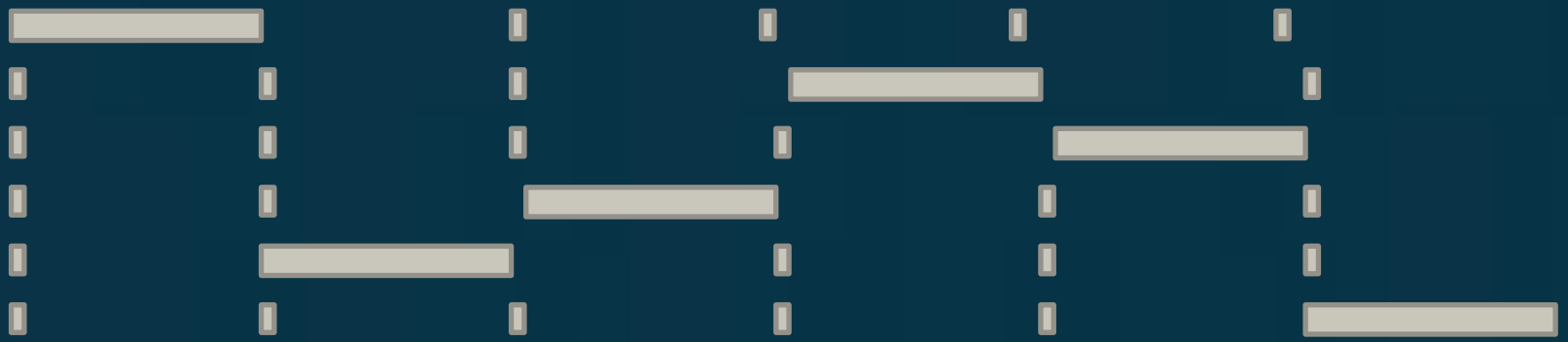
Back-Off

- 구현

```
class BackOff{
            int minDelay, maxDelay;
            int limit;
            int random;

public:
            BackOff(int min, int max) {
                        minDelay = min;
                        maxDelay = max;
                        limit = minDelay;
            }

            void InterruptedException() {
                        int delay = rand()%limit+1;
                        if(limit<maxDelay)          limit = 2*limit;
                        Sleep(delay);
            }
};


void Lock()
{
            BackOff backOff(MIN_DELAY, MAX_DELAY);
            while (true) {
                        while (state) {}
                        if (!TestAndSet(true)) return;
                        else
                                    backOff.InterruptedException();
            }
}
```

Back-Off

- 벤치마크 문제
  - Sleep()이 너무 오랜시간 thread를 재운다.
    - 병렬로 수행되지 않고 그냥 sequential하게 수행된다.



BackOff 속도

- Core2Duo
  - MinDelay : 10, MaxDelay : 200

Threads	Second
1	
2	
4	
8	
16	
32	

BackOff 속도

- i7 920
  - MinDelay : 10, MaxDelay : 200

Threads	Second
1	0.34
2	0.42
4	0.45
8	0.61
16	0.87
32	0.83

BackOff 속도

- Nehalem 2 CPU
  - MinDelay : 10, MaxDelay : 200

Threads	Second
1	
2	
4	
8	
16	
32	

BackOff 속도

- XEON (E5405, Harpertown, 2CPU)
  - MinDelay : 10, MaxDelay : 200
  -

Threads	Second
1	
2	
4	
8	
16	
32	

Back-Off

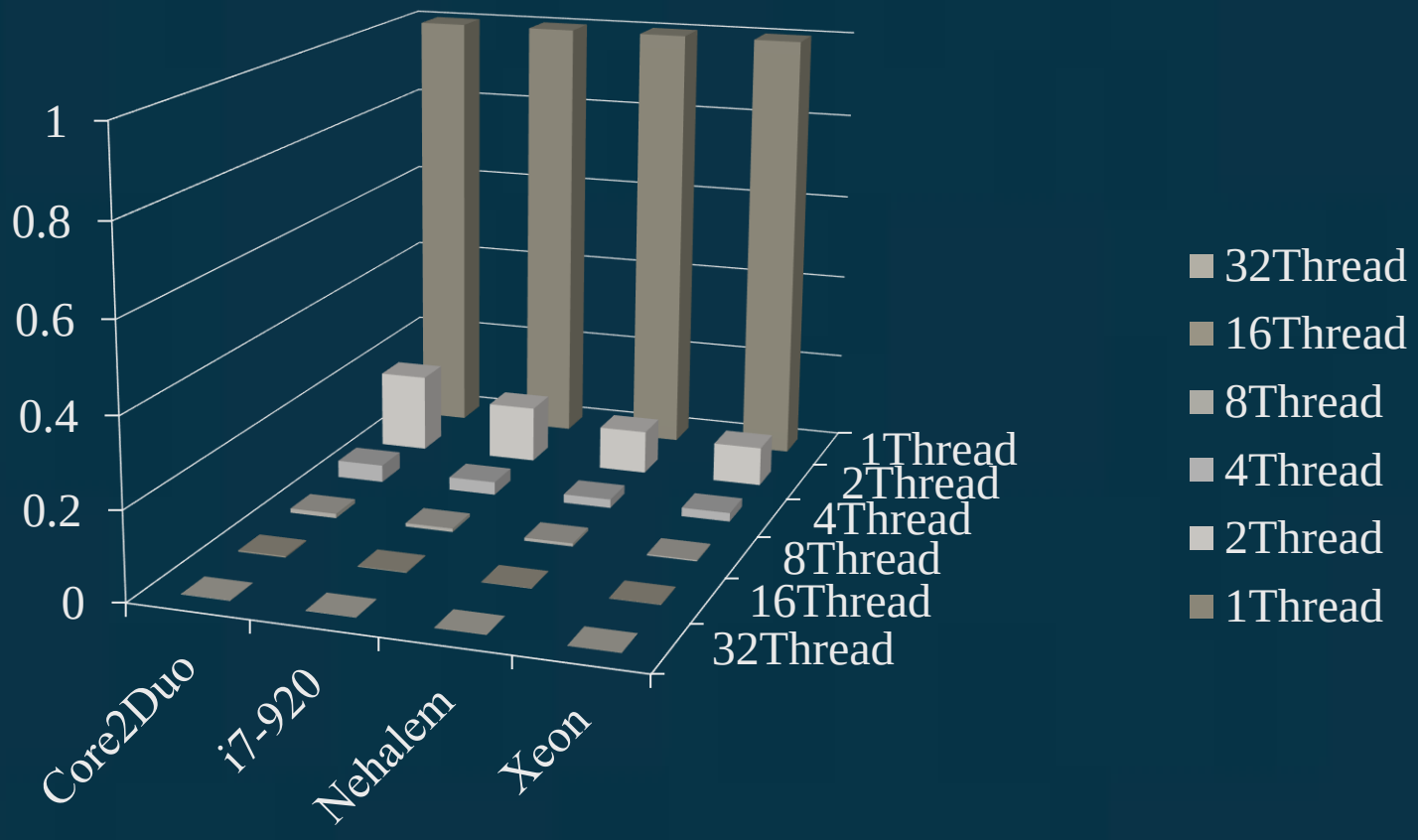
- 벤치마크 문제
  - Sleep()이 너무 오랜시간 thread를 재운다.
    - 차라리 아래 프로그램이 훨씬 효율적

```
void Lock()
{
      while (true) {
            while (state) {}
            if (!TestAndSet(true)) return;
            else Sleep(0);
      }
}
```

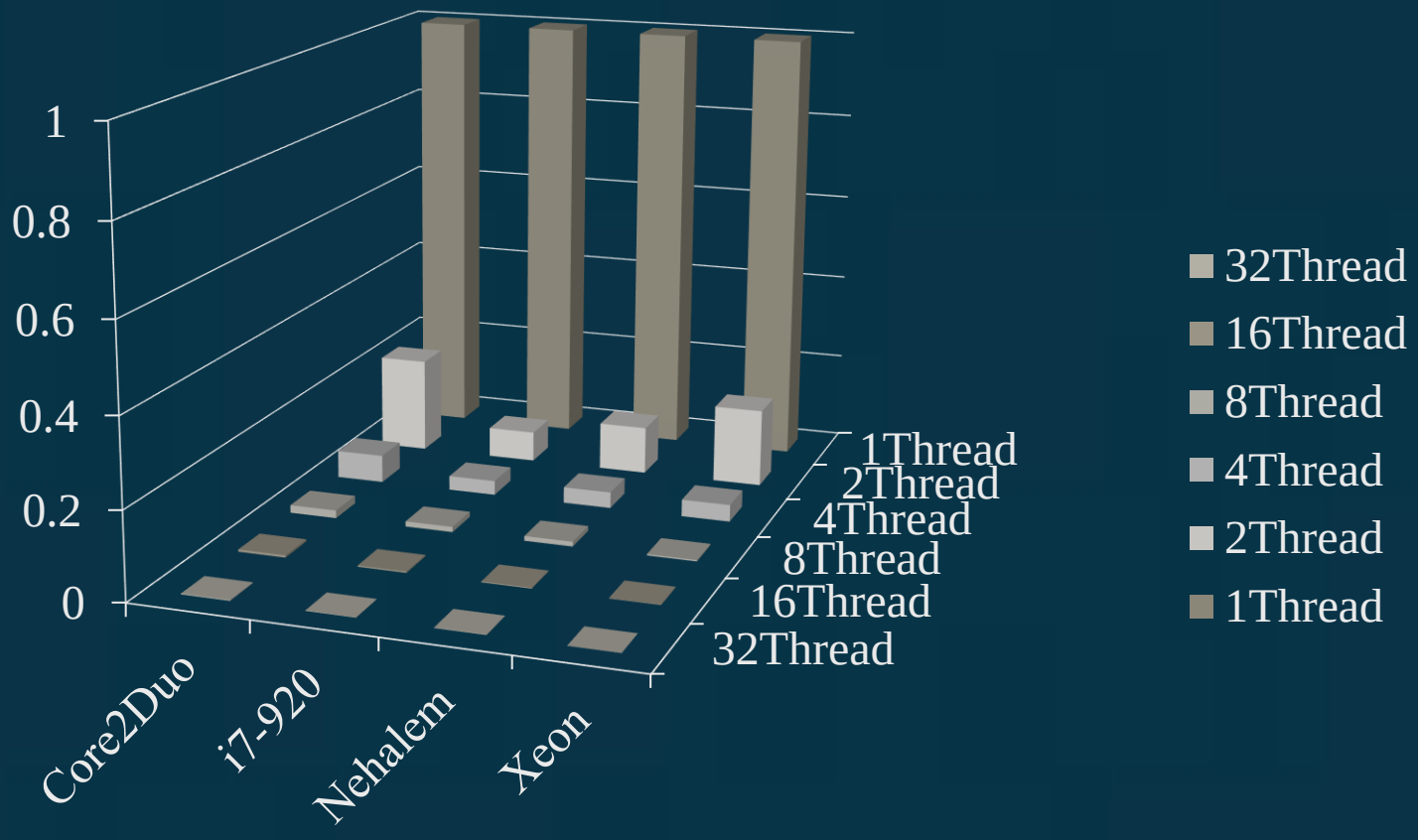
Back-Off

- 벤치마크 문제
  - Sleep()이 너무 오랜시간 thread를 재운다.
    - 운영체제에 따라 결과가 틀린듯…
    - 슬립에 의존한 시간조정은 물 한잔 마시는데 유조선을 사용하는 격.
    - 전체적인 Throughput은 높아지지만 Sleep()한 thread들의 response time은 최악이다.

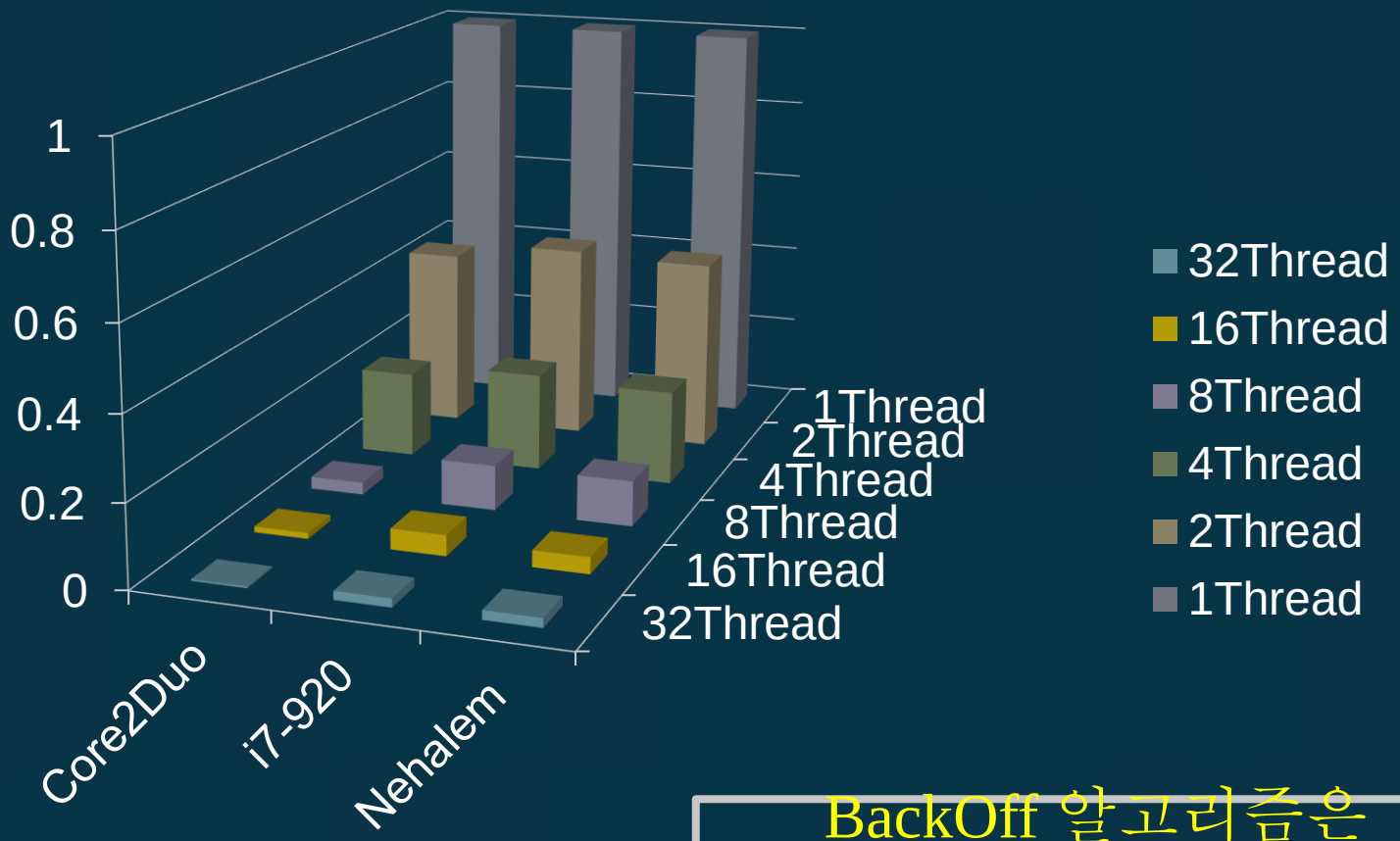
TAS : Efficiency값



TTAS : Efficiency값

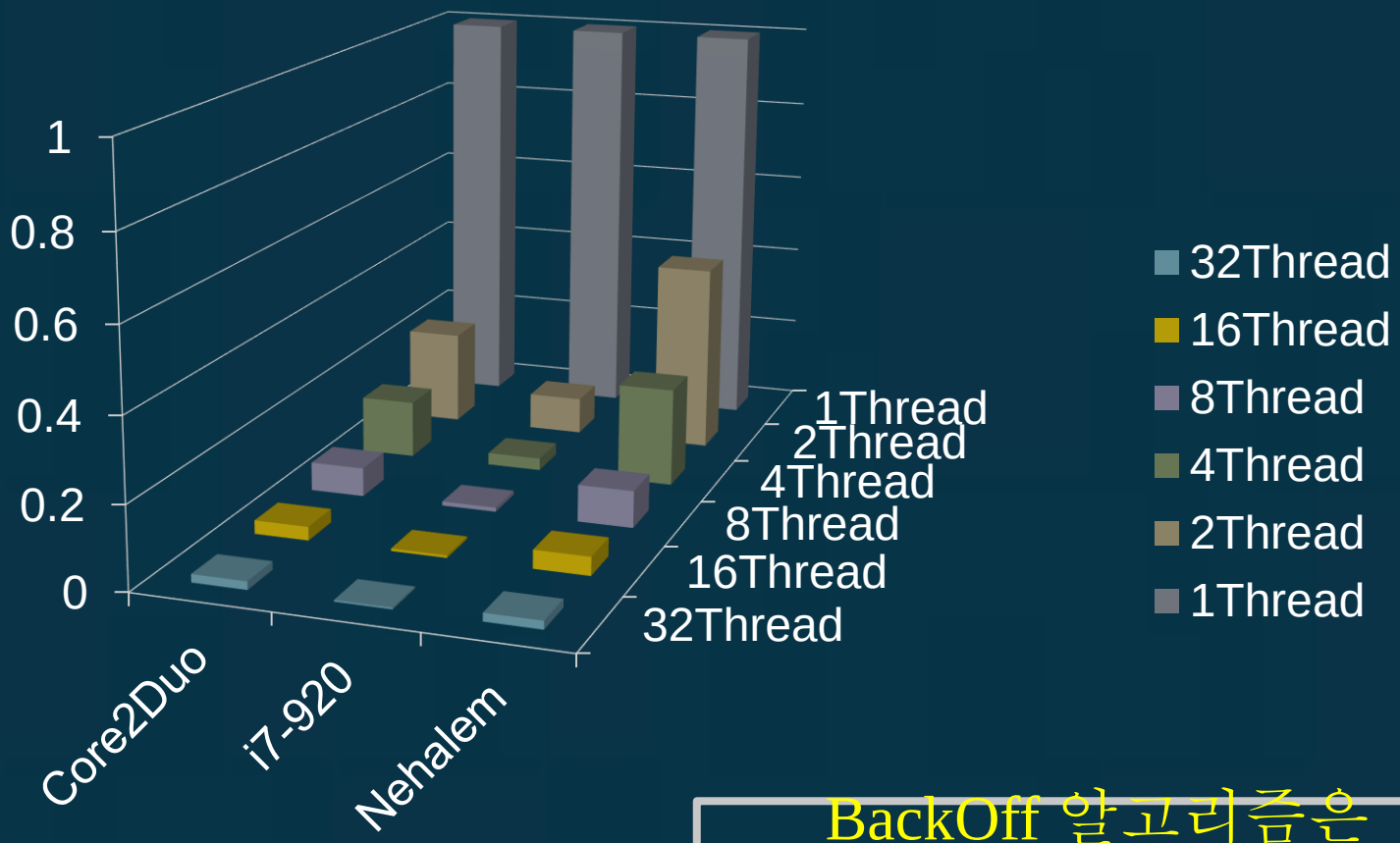


BackOff : Efficiency값



BackOff 알고리즘은
연산횟수 10배

BackOff(Sleep): Efficiency값



BackOff 알고리즘은
연산횟수 10배

큐 잠금

- Back-Off의 문제점
  - TAS보다 적지만 버스 트래픽이 발생
  - 스레드가 필요한 정도 보다 더 오래 지연될수 있으므로 임계영역이 비효율적으로 이용될 수 있다.
- 큐 잠금
  - Back-off알고리즘의 보완
  - 각 스레드들은 자신의 앞에 있는 스레드가 완료했는지 체크해봄으로써 자신의 차례가 되었는지 알 수 있다.

Alock 큐 잠금

- 구현 및 설명
  - 각 스레드들은 tail을 공유한다.
    - tail값은 단조증가하고 slot값을 정하는 데에 쓰인다.
  - flag배열은 해당 인덱스(slot)가 true이면 그 스레드가 잠금을 획득할 권한이 있음을 뜻한다.
    - 잠금을 해제 할 때는 해당 flag값을 false로 바꾸고 다음 배열 값(slot+1)을 true로 설정한다.
  - mySlotIndex는 스레드 지역 변수.
    - _declspec(thread) static 로 선언

Alock 큐 잠금

- 구현 및 설명

```
_declspec(thread) static int mySlotIndex = 0;

volatile LONG tail = 0;
volatile bool flag[THREAD_MAX];
int size;

void Lock()
{
          int slot = InterlockedIncrement((long*)&tail) % size;
          mySlotIndex = slot;
          while (!flag[slot]) {}
}

void UnLock()
{
          int slot = mySlotIndex;
          flag[slot] = false;
          flag[(slot+1)%size] = true;
}
```

Alock 속도

- Core2Duo

Threads	Second
1	0.78
2	1.40
4	XXX
8	XXX
16	XXX
32	XXX

Alock 속도

- i7 920

Threads	Second
1	0.73
2	2.28
4	3.25
8	4.61
16	X
32	X

Alock 속도

- Nehalem 2 CPU

Threads	Second
1	0.95
2	2.59
4	3.27
8	3.60
16	9.81
32	XXXX

큐 잠금

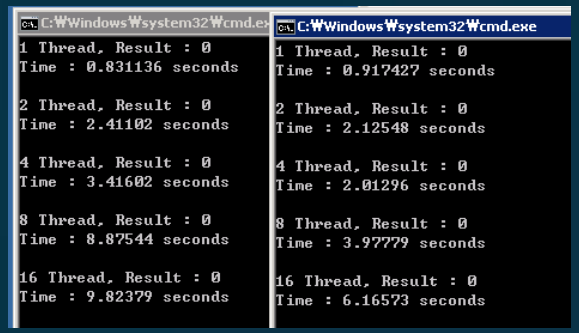
- 실행결과
  - TAS보다는 성능이 좋고, TTAS보다 약간 떨어지고, Backoff보다는 스레드가 많아 질수록 성능이 많이 떨어진다.
  - Thread가 많아질 경우 현격하게 느려진다.
    - 하드웨어 코어개수 보다 많을때

큐 잠금

- 문제점
  - false sharing
    - 배열과 같이 서로 붙어있는 데이터가 하나의 캐시라인에 있을 때 발생
      - 한 스레드의 쓰기연산은 그 항목이 속한 캐시라인을 무효화하게 되어 가짜 무효화가 발생하고 무효화 트래픽을 일으킨다.
        - 가짜 무효화 : 값은 실제로 변하지 않았지만 무효화가 되었다고 판단하는 현상

Threads	Second
1	0.84
2	2.83
4	3.26
8	3.78
16	6.74
32	XXXXX

flag간의 간격을 cache line사이즈보다 크게한 경우



sum++를 제거하여 기본 false sharing을 없앤 경우의 속도비교

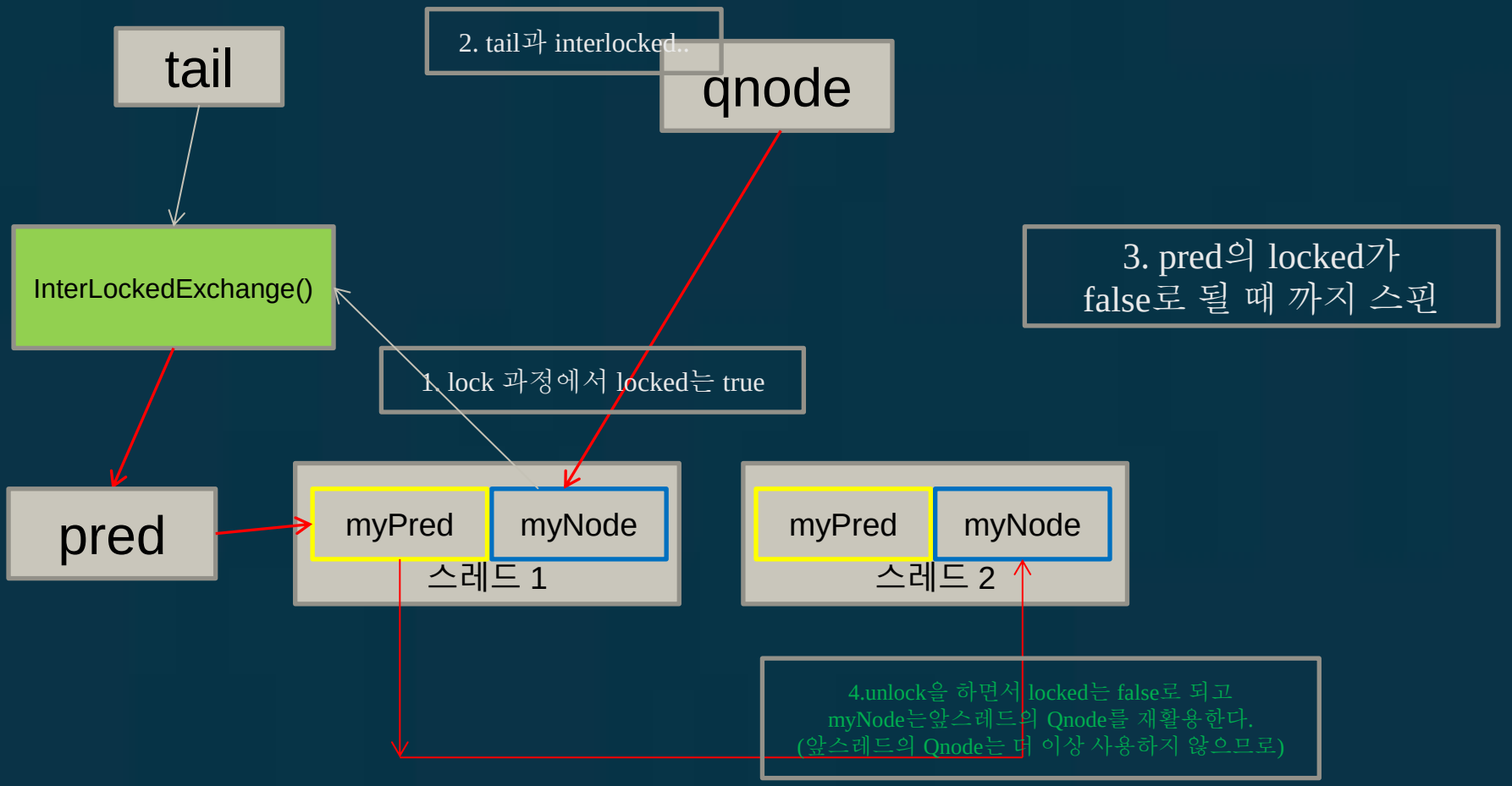
큐 잠금

- 문제점
  - Busy waiting
    - 쓰레드의 개수가 코어의 개수보다 많을 경우 다음 쓰레드에게 양보했음에도 불구하고 다음 쓰레드가 Sleep 스테이트에 있기 때문에 진행을 하지 못하고 다음 Thread schedule시간까지 전체 시스템이 멈춰있게 된다.
    - Sleep()을 넣으면 되지만 이 역시 무지 느리다.

Threads	Second
1	0.82
2	3.42
4	5.21
8	8.49
16	10.92
32	24.61

CLH 큐 잠금

- 구현 및 설명



CLH 큐 잠금

- 구현 및 설명

```
<thread local>Qnode *myNode, *myPred;

void Lock()
{
        QNode *qnode = myNode;
        qnode->locked=true;
        QNode *pred = TestAndSet(&tail, qnode));
        myPred = *pred;

        while(pred->locked) {}
}

void UnLock()
{
        myNode->locked = false;
        myNode = myPred;
}
```

CHL 속도

- Nehalem 2 CPU

Threads	Second
1	3.77, 3.76, 3.76
2	22.03, 21.58, 21.97
4	70.25, 35.06, 36.16
8	84.66, 96.59, 81.11
16	97.25, 98.59, 98.40
32	XXXX

CHL 속도

- OLD XEON 2 CPU

Threads	Second
1	7.54, 7.53
2	74.81, 73.07
4	110.37, 112.69
8	141.81, 144.225
16	XXXX
32	XXXX

CLH 큐 잠금

- 단점
  - Cache가 없는 NUMA 컴퓨터에서 낮은 성능을 보인다.
  - 현재의 서버 컴퓨터
    - OLD XEON : NUMA가 아니다.
    - Nehalem : NUMA이다. Cache가 있다.
  - NUMA란?
    - Non Uniform Memory Access
      - Physical 메모리 주소에 따라 접근 속도가 다를 수 있는 컴퓨터

성능

벤치마킹을 결과 CHL이 더 빠른 결과가 나타남

평균	3.564	17.083	27.913	36.774
SpeedUp(평균)	1	0.208628	0.127682	0.096916
Efficiency(평균)	1	0.104314	0.031921	0.012115
최대	3.57	17.21	31.42	44.4
SpeedUp(최대)	1	0.207438	0.113622	0.080405
Efficiency(최대)	1	0.103719	0.028405	0.010051
최소	3.55	16.71	26.5	32.63
SpeedUp(최소)	1	0.212448	0.133962	0.108796
Efficiency(최소)	1	0.106224	0.033491	0.013599

CHLock

평균	7.31	21.856	30.501	37.991
SpeedUp(평균)	1	0.334462	0.239664	0.192414
Efficiency(평균)	1	0.167231	0.059916	0.024052
최대	7.32	22.1	31.08	43.98
SpeedUp(최대)	1	0.331222	0.235521	0.166439
Efficiency(최대)	1	0.165611	0.05888	0.020805
최소	7.3	21.69	29.57	34.11
SpeedUp(최소)	1	0.336561	0.246872	0.214013
Efficiency(최소)	1	0.16828	0.061718	0.026752

Alock

MCS 큐 잠금

- 생략