



5. Non-Blocking 알고리즘 - LIST

멀티쓰레드 프로그래밍
정내훈

목표 및 소개

● 목표

- Non Blocking 자료 구조의 제작 실습
- 일반 자료구조를 멀티쓰레드 자료구조로 변환한다.
- Blocking자료구조부터 시작하여 단계별로 성능향상 기법을 적용한다.
- 최종적으로 Lock-Free 자료구조를 제작한다.
- 각 자료구조의 성능을 비교한다.

목표 및 소개

● 목표 자료구조

— SET

- 아이템의 중복을 허용하지 않는다.
- 검색의 효율성을 위해 아이템은 정렬되어 저장된다.
- 삽입 삭제의 효율성을 위해 링크드리스트로 구현된다.

— 구현 할 Method

- add
- remove
- contains

리스트로 만든 집합

- 필드

- key : 리스트에 저장 되는 값
- next : 다음 노드의 포인터

- 메서드

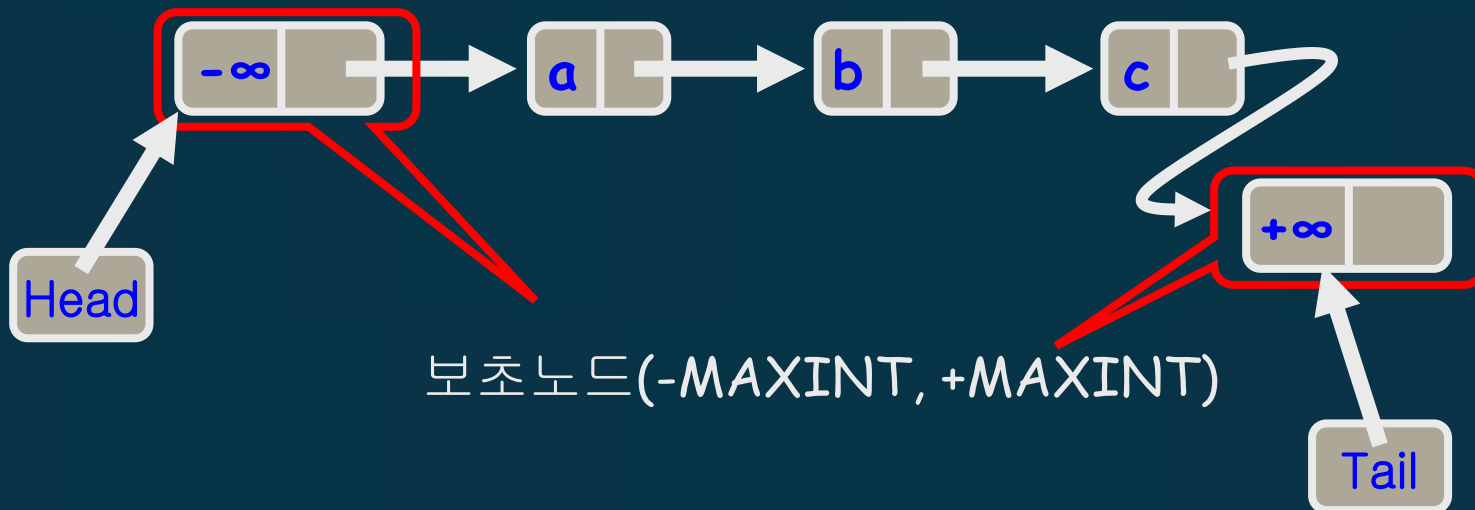
- add(x) : 집합에 x 추가, 성공시 true 반환
- remove(x) : 집합에서 x 제거, 성공시 true 반환
- contains(x) : 집합에 x가 있다면 true 반환

리스트로 만든 집합

- 추가적인 구현

- 보조 노드

- 검색의 효율성을 위해 항상 존재하는 Head와 Tail노드를 갖도록 한다.
 - Head는 MAXINT, TAIL은 -MAXINT를 키로 갖는다.



구현 차례

- 성긴 동기화(coarse-grained synchronization)
 - Lock하나로 동기화 객체 전체를 감싸는 경우
- 세밀한 동기화 (fine-grained synchronization)
- 낙천적인 동기화 (optimistic synchronization)
- 게으른 동기화 (lazy synchronization)
- 비멈춤 동기화 (nonblocking synchronization)

리스트의 구현

● 성긴 동기화

— 구현

- 리스트는 하나의 잠금을 갖고 있으며, 모든 메서드호출은 이 잠금을 통해 Critical Section으로 진행된다.
 - 모든 메서드는 잠금을 가지고 있는 동안에만 리스트에 접근한다.

— 문제점

- 경쟁이 낮을 경우 이 동기화가 좋은 선택이지만 경쟁이 높아질 경우 성능이 저하된다.
- Blocking이다.

리스트의 구현

● 성긴 동기화 — 구현

```
class NODE {
public:
    int key;
    NODE *next;

    NODE() { next = NULL; }

    NODE(int key_value) {
        next = NULL;
        key = key_value;
    }

    ~NODE() {}
};
```

```
class CLIST {
    NODE head, tail;
    mutex glock;
public:
    CLIST()
    {
        head.key = 0x80000000;
        tail.key = 0x7FFFFFFF;
        head.next = &tail;
    }
    ~CLIST() {}

    void Init()
    {
        NODE *ptr;
        while(head.next != &tail) {
            ptr = head.next;
            head.next = head.next->next;
            delete ptr;
        }
    }

    bool Add(int key)
    {
    }

    bool Remove(int key)
    {
    }

    bool Contains(int key)
    {
    }
};
```


리스트의 구현

● 성긴 동기화 — 구현

```
bool Add(int key)
{
    NODE *pred, *curr;

    pred = &head;
    glock.lock();
    curr = pred->next;
    while (curr->key < key) {
        pred = curr;
        curr = curr->next;
    }

    if (key == curr->key) {
        glock.unlock();
        return false;
    } else {
        NODE *node = new NODE(key);
        node->next = curr;
        pred->next = node;
        glock.unlock();
        return true;
    }
}
```

```
bool Remove(int key)
{
    NODE *pred, *curr;

    pred = &head;
    glock.lock();
    curr = pred->next;
    while (curr->key < key) {
        pred = curr;
        curr = curr->next;
    }

    if (key == curr->key) {
        pred->next = curr->next;
        delete curr;
        glock.unlock();
        return true;
    } else {
        glock.unlock();
        return false;
    }
}
```

리스트의 구현

- 성긴 동기화
 - 구현

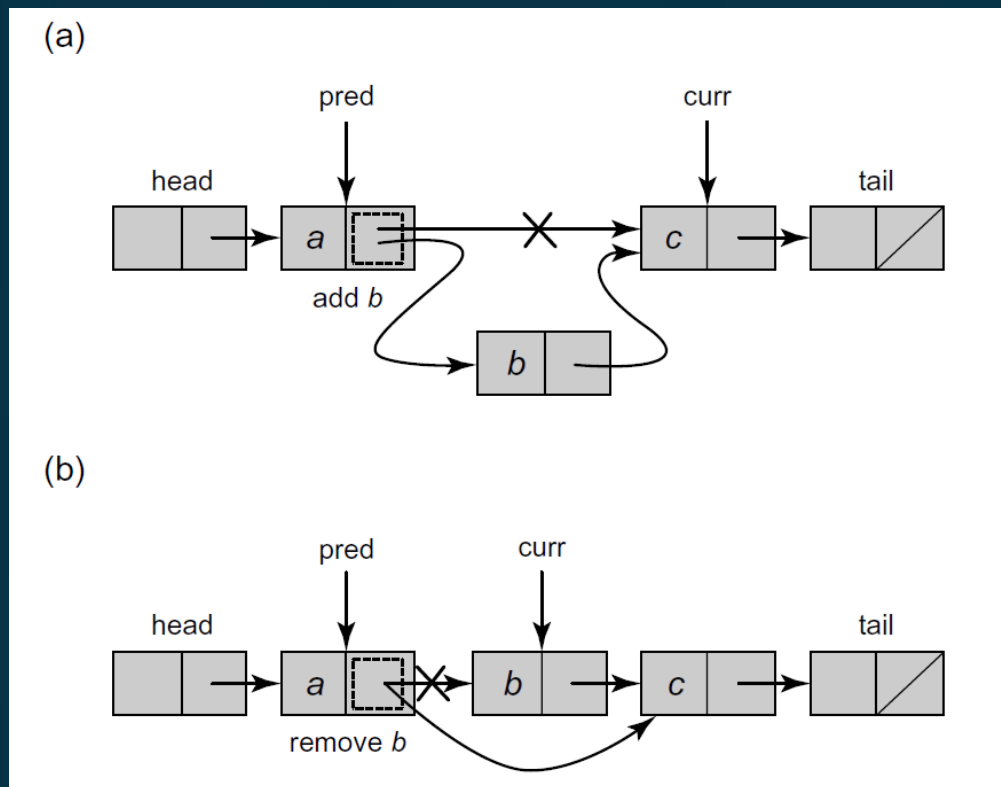
```
bool Contains(int key)
{
    NODE *pred, *curr;

    pred = &head;
    glock->lock() ;
    curr = pred->next;
    while (curr->key < key) {
        pred = curr;
        curr = curr->next;
    }
    if (key == curr->key) {
        glock->unlock() ;
        return true;
    } else {
        glock->unlock() ;
        return false;
    }
}
```

리스트의 구현

● 성긴 동기화

— 구현 : pred와 curr를 사용한 add & remove



리스트의 구현

● 실습 : #19

- 성긴 동기화 리스트를 구현하시오
- 0과 999사이의 숫자를 랜덤하게 4백만회 삽입/삭제하는 프로그램을 통해 실행 속도를 측정하시오 (다음 페이지)
- 쓰레드가 1개, 2개, 4개, 8개일 때의 속도를 비교하시오.
 - 각각 실행 전 List를 클리어 하시오

리스트의 구현

- 실습 : #19
 - test program

```
const auto NUM_TEST = 4000000;  
const auto KEY_RANGE = 1000;
```

```
void ThreadFunc(int num_thread)  
{  
    int key;  
  
    for (int i=0; i < NUM_TEST / num_thread; i++) {  
        switch (rand() % 3) {  
            case 0: key = rand() % KEY_RANGE;  
                    clist.Add(key);  
                    break;  
            case 1: key = rand() % KEY_RANGE;  
                    clist.Remove(key);  
                    break;  
            case 2 : key = rand() % KEY_RANGE;  
                     clist.Contains(key);  
                     break;  
            default : cout << "Error\n";  
                     exit(-1);  
        }  
    }  
}
```

성긴 동기화

- 속제 2 :
 - 성긴 동기화 리스트의 구현 (실습시간에 완성 못한 사람만)
 - 제출물
 - .cpp 파일
 - 실행속도 비교표
 - CPU의 종류 (모델명, 코어 개수, 클럭)
 - 제출 : eclass

성긴 동기화

● 결과

	성긴동기화						
1	2819						
2	2902						
4	3153						
8	4111						

싱글 쓰레드 : 2560

구현 차례

- 성긴 동기화(coarse-grained synchronization)
 - Lock하나로 동기화 객체 전체를 감싸는 경우
- 세밀한 동기화 (fine-grained synchronization)
- 낙천적인 동기화 (optimistic synchronization)
- 게으른 동기화 (lazy synchronization)
- 비멈춤 동기화 (nonblocking synchronization)

리스트의 구현

● 세밀한 동기화

— 전체 리스트를 한꺼번에 잠그는 것보다 개별 노드를 잠그는 것이 병행성을 향상시킬 수 있다.

- 전체 리스트에 대한 잠금을 두는 것이 아니라, 각각의 노드에 잠금을 둔다.
- Node에 Lock()과 Unlock()메소드를 구현해야 한다.
- Node의 next field를 변경할 경우에는 반드시 Lock()을 얻은 후 변경해야 한다.

리스트의 구현

● 세밀한 동기화

– 주의점

- Add()와 Remove()시점의 Pred, Curr가 가리키는 노드는 Locking이 되어 있어야 한다.
- Head 부터 Node이동을 할 때 Lock을 잠그면서 이동해야 한다.
 - 예를 들어 a의 잠금을 풀고 나서 b(a->next 였던)의 잠금을 한다면 그 사이에 다른 스레드에 의해 b가 제거될 수 있기 때문이다.
 - 즉, 이동 시 pred이 잠금상태 일 때 동안 curr의 잠금을 획득한다.

리스트의 구현

● 세밀한 동기화

– 구현

- java임
- C++로 번역하시오
- hashCode는 삭제
 - item이 key
- try/finally가 수행하는
unlock을 제대로 된 위치로
옮겨야 한다.

```

1  public boolean add(T item) {
2      int key = item.hashCode();
3      head.lock();
4      Node pred = head;
5      try {
6          Node curr = pred.next;
7          curr.lock();
8          try {
9              while (curr.key < key) {
10                 pred.unlock();
11                 pred = curr;
12                 curr = curr.next;
13                 curr.lock();
14             }
15             if (curr.key == key) {
16                 return false;
17             }
18             Node newNode = new Node(item);
19             newNode.next = curr;
20             pred.next = newNode;
21             return true;
22         } finally {
23             curr.unlock();
24         }
25     } finally {
26         pred.unlock();
27     }
28 }

```

리스트의 구현

● 세밀한 동기화

– 구현

- add()와 동일
- C로 번역할 때는 삭제된 노드를 delete 해주어야 함.

– Contains()

- add(), remove()와 마찬가지로 locking을 하면서 이동하기만 하면 됨.

```
29 public boolean remove(T item) {  
30     Node pred = null, curr = null;  
31     int key = item.hashCode();  
32     head.lock();  
33     try {  
34         pred = head;  
35         curr = pred.next;  
36         curr.lock();  
37         try {  
38             while (curr.key < key) {  
39                 pred.unlock();  
40                 pred = curr;  
41                 curr = curr.next;  
42                 curr.lock();  
43             }  
44             if (curr.key == key) {  
45                 pred.next = curr.next;  
46                 return true;  
47             }  
48             return false;  
49         } finally {  
50             curr.unlock();  
51         }  
52     } finally {  
53         pred.unlock();  
54     }  
55 }
```

리스트의 구현

- 실습 : #20

- 세밀한 동기화 리스트를 구현하시오
- 실습 #19에서 사용한 테스트 프로그램을 사용하시오
- 스레드가 1개, 2개, 4개, 8개일 때의 속도를 비교하시오.
 - 각각 실행 전 List를 클리어 하시오

세밀한 동기화

- 속제 2 :

- 세밀한 동기화 리스트의 구현 (실습시간에 완성 못한 사람만)
 - 테스트 횟수를 400만번에서 40만번으로 줄일 것.
- 제출물
 - .cpp 파일
 - 실행속도 비교표
 - CPU의 종류 (모델명, 코어 개수, 클럭)
- 제출 : Eclass

성긴 동기화

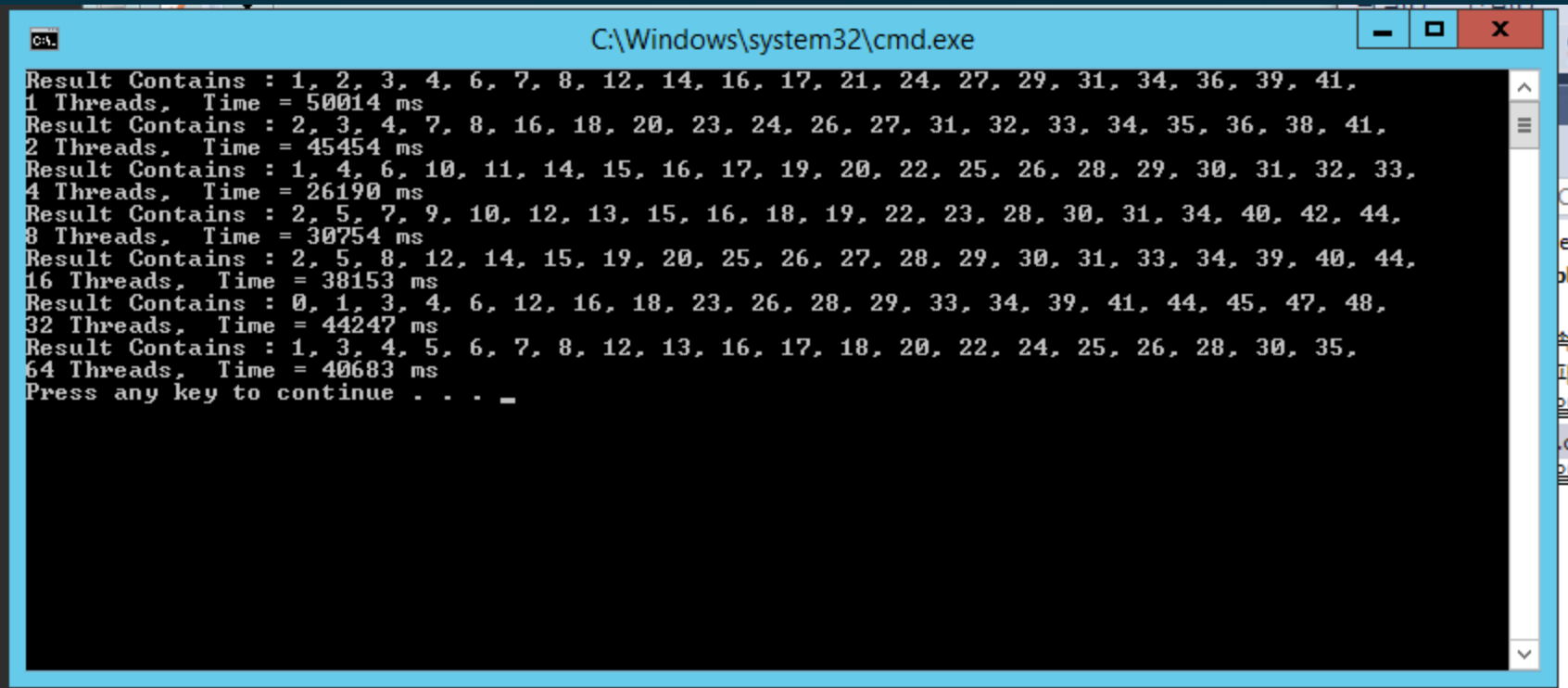
● 결과

	성긴동기화	세밀한 동기화					
1	2819	25310					
2	2902	23125					
4	3153	19000					
8	4111	14171					

싱글 쓰레드 : 2560

성능 비교

- 8Core – 4CPU Zeon



```

C:\Windows\system32\cmd.exe

Result Contains : 1, 2, 3, 4, 6, 7, 8, 12, 14, 16, 17, 21, 24, 27, 29, 31, 34, 36, 39, 41,
1 Threads, Time = 50014 ms
Result Contains : 2, 3, 4, 7, 8, 16, 18, 20, 23, 24, 26, 27, 31, 32, 33, 34, 35, 36, 38, 41,
2 Threads, Time = 45454 ms
Result Contains : 1, 4, 6, 10, 11, 14, 15, 16, 17, 19, 20, 22, 25, 26, 28, 29, 30, 31, 32, 33,
4 Threads, Time = 26190 ms
Result Contains : 2, 5, 7, 9, 10, 12, 13, 15, 16, 18, 19, 22, 23, 28, 30, 31, 34, 40, 42, 44,
8 Threads, Time = 30754 ms
Result Contains : 2, 5, 8, 12, 14, 15, 19, 20, 25, 26, 27, 28, 29, 30, 31, 33, 34, 39, 40, 44,
16 Threads, Time = 38153 ms
Result Contains : 0, 1, 3, 4, 6, 12, 16, 18, 23, 26, 28, 29, 33, 34, 39, 41, 44, 45, 47, 48,
32 Threads, Time = 44247 ms
Result Contains : 1, 3, 4, 5, 6, 7, 8, 12, 13, 16, 17, 18, 20, 22, 24, 25, 26, 28, 30, 35,
64 Threads, Time = 40683 ms
Press any key to continue . . . _
```


구현 차례

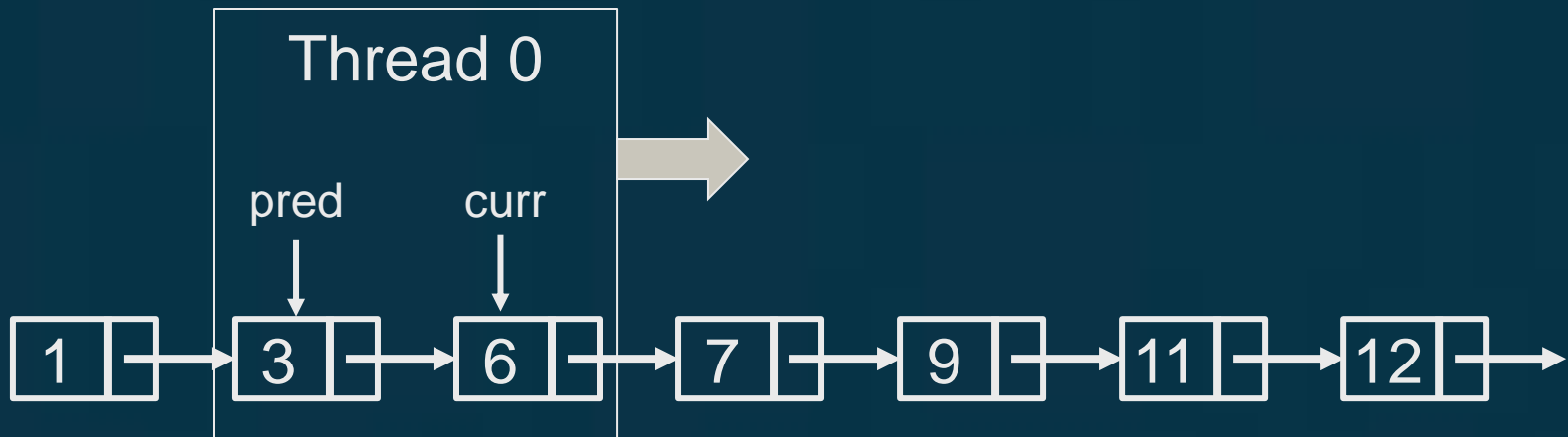
- 성긴 동기화(coarse-grained synchronization)
 - Lock하나로 동기화 객체 전체를 감싸는 경우
- 세밀한 동기화 (fine-grained synchronization)
- 낙천적인 동기화 (optimistic synchronization)
- 게으른 동기화 (lazy synchronization)
- 비멈춤 동기화 (nonblocking synchronization)

낙천적 동기화

- 세밀한 동기화의 성능 저하 원인
 - 잠금의 획득과 해제가 너무 빈번하다.
 - 리스트가 길어지는 경우 성능이 매우 떨어진다.
 - 해결 아이디어???
- 이동 시 잠금을 하지 않는다.
 - Add/Remove를 위해 pred를 수정하기 전에 pred를 잠근다.

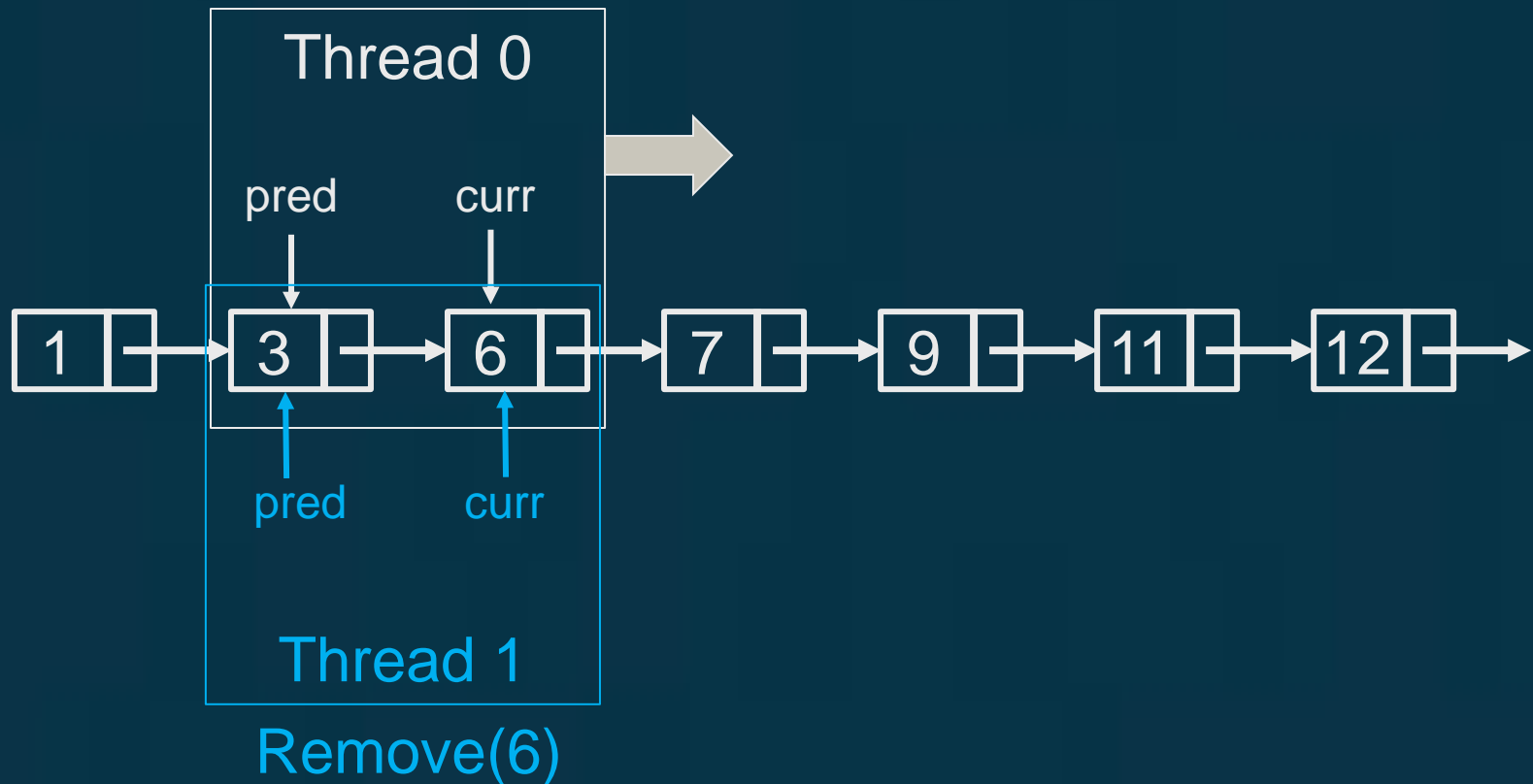
낙천적 동기화

- 이동 시 잠금을 하지 않는다???



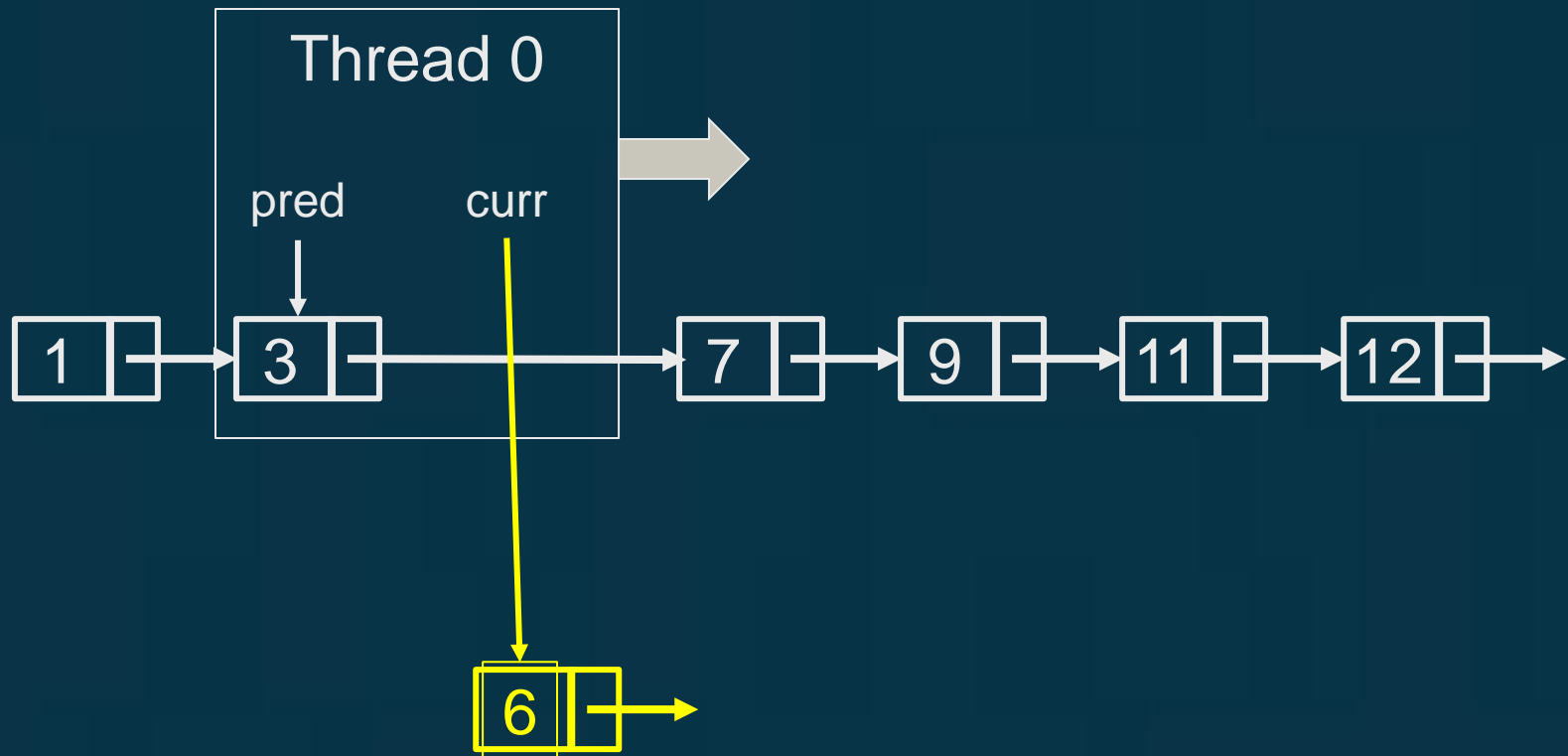
낙천적 동기화

- 이동 시 잠금을 하지 않는다???



낙천적 동기화

- 이동 시 잠금을 하지 않는다???



이후 시나리오

리스트에 재사용 되어서 엉뚱한 위치에 추가 => 오동작

다른 자료구조에 재사용 되어서 next에 다른 값 저장 => CRASH

낙천적 동기화

- 이동 시 잠금을 하지 않는다.
 - Data Race
 - 세밀한 동기화에서 이동시 잠그는 이유가 있음.
- 해결
 - Crash (또는 무한루프)
 - 제거된 Node의 next가 crash를 발생시키는 값을 갖지 않게 한다.
 - 제거된 Node라도 next를 따라가면 TAIL이 나오게 한다.
 - 오동작
 - pred와 curr를 잘못 잠갔을 경우 처음부터 다시 실행
 - pred와 curr를 잠근 후 제대로 잠갔는지 검사. (Validation)

낙천적 동기화

● 임시 해결

— 제거된 노드를 'delete' 하지 않는다.

- next 필드의 오염 방지, 결국엔 TAIL 만남.
- 하지만 **Memory Leak** => 나중에 해결
- 교재의 Java 예제는 문제가 없음
 - Java언어는 더 이상 사용되지 않는 Node를 자동적으로 delete (Garbage Collection)
 - C++의 shared_ptr와 비슷 (같지는 않음)

— Validation 조건 검사

- 잠겨진 pred와 curr가 제거되지 않았고
- pred와 curr사이에 다른 노드가 끼어들지 않았다.

낙천적 동기화

● Validation

Pred, Curr가 리스트에 존재한다.
Pred와 Curr사이에 다른 노드가 없다.

— 충분한가? 충분하다.

- locking이 되어 있으므로 validation조건이 만족된 이후에는 다른 스레드가 pred와 curr를 변경할 수 없다.
- 다시 검색을 실행해도 항상 같은 pred, curr가 선택된다.
- 따라서 add/remove 연산을 해도 안전하다.

낙천적 동기화

- `validate()` : 유효성 검사
 - 다시 처음부터 검색해서 원래 `pred`, `curr`로 다시 올 수 있는지 확인한다.
 - `pred`와 `curr`가 리스트에 존재하는 지 확인
 - `pred->next == curr`인 것을 확인한다.
 - 중간에 다른 노드가 끼어들지 않았음을 확인

낙천적 동기화

● 문제점

- 낙천적 동기화 알고리즘은 기아를 겪을 수 있다.
 - Validate가 실패하면 (false를 리턴하면) 처음 부터 다시 실행한다. (head부터 검색을 다시)
 - 다른 스레드들이 pred와 curr를 계속 수정하는 경우 계속 재시도를 하면서 지연될 수 있다.
 - 기아상태를 겪는 경우는 흔치 않은 경우이기 때문에 실제로는 잘 동작할 가능성이 크다.

낙천적 동기화

- Add

- lock을 걸고 해제하는 순서에 주의
- 전체 재시도를 위한 while loop 존재

```
1  public boolean add(T item) {
2      int key = item.hashCode();
3      while (true) {
4          Node pred = head;
5          Node curr = pred.next;
6          while (curr.key < key) {
7              pred = curr; curr = curr.next;
8          }
9          pred.lock(); curr.lock();
10         try {
11             if (validate(pred, curr)) {
12                 if (curr.key == key) {
13                     return false;
14                 } else {
15                     Node node = new Node(item);
16                     node.next = curr;
17                     pred.next = node;
18                     return true;
19                 }
20             }
21         } finally {
22             pred.unlock(); curr.unlock();
23         }
24     }
25 }
```

낙천적 동기화

- Remove

```
26 public boolean remove(T item) {
27     int key = item.hashCode();
28     while (true) {
29         Node pred = head;
30         Node curr = pred.next;
31         while (curr.key < key) {
32             pred = curr; curr = curr.next;
33         }
34         pred.lock(); curr.lock();
35         try {
36             if (validate(pred, curr)) {
37                 if (curr.key == key) {
38                     pred.next = curr.next;
39                     return true;
40                 } else {
41                     return false;
42                 }
43             }
44         } finally {
45             pred.unlock(); curr.unlock();
46         }
47     }
48 }
```

낙천적 동기화

● Contains

— 구현

```
49 public boolean contains(T item) {
50     int key = item.hashCode();
51     while (true) {
52         Entry pred = this.head; // sentinel node;
53         Entry curr = pred.next;
54         while (curr.key < key) {
55             pred = curr; curr = curr.next;
56         }
57         try {
58             pred.lock(); curr.lock();
59             if (validate(pred, curr)) {
60                 return (curr.key == key);
61             }
62         } finally { // always unlock
63             pred.unlock(); curr.unlock();
64         }
65     }
66 }
```

낙천적 동기화

● Validate

- Pred가 List에 있는지 검사
- Pred다음에 Next가 있는지 검사

```
67  private boolean validate(Node pred, Node curr) {  
68      Node node = head;  
69      while (node.key <= pred.key) {  
70          if (node == pred)  
71              return pred.next == curr;  
72          node = node.next;  
73      }  
74      return false;  
75  }
```

리스트의 구현

- 실습 : #21

- 낙천적인 동기화 리스트를 구현하시오
- 실습 #19의 프로그램을 통해 실행 속도를 측정하시오
- 스레드가 1개, 2개, 4개, 8개일 때의 속도를 비교하시오.
 - 각각 실행 전 List를 클리어 하시오

성긴 동기화

● 결과 (목금반)

	성긴동 기화	세밀한 동기화	낙천적 동기화				
1	2819	25310	4541				
2	2902	23125	3799				
4	3153	19000	2782				
8	4111	14171	1709				

싱글 쓰레드 : 2560

낙천적 동기화

- 숙제 4 :

- 낙천적 동기화 리스트의 구현 (실습시간에 완성 못한 사람만)
- 제출물
 - .cpp 파일
 - 실행속도 비교표 (성긴/세밀한 동기화와 비교)
 - CPU의 종류 (모델명, 코어 개수, 클럭)
- 제출 : eClass

구현 차례

- 성긴 동기화(coarse-grained synchronization)
 - Lock하나로 동기화 객체 전체를 감싸는 경우
- 세밀한 동기화 (fine-grained synchronization)
- 낙천적인 동기화 (optimistic synchronization)
- 게으른 동기화 (lazy synchronization)
- 비멈춤 동기화 (nonblocking synchronization)

리스트의 구현

● 게으른 동기화

- 낙천적 동기화는 Lock의 횟수는 비약적으로 감소했으나 리스트를 두 번 순회해야 한다는 눈에 보이는 오버헤드가 있다.
- 이를 극복하여 다시 순회하지 않는 알고리즘을 작성하였다.
 - validate()가 노드를 처음부터 다시 순회하지 않고 validation을 수행한다.
 - pred와 curr의 잠금은 여전히 필요하다.

리스트의 구현

- 게으른 동기화
 - Contains() 메소드는 자주 호출되는 메소드인데 이 메소드를 **Wait-Free**로 만들 수 있으면 좋겠다.
 - 목적이 아니라 부수효과에 가까움...

리스트의 구현

● 게으른 동기화의 아이디어

- 각 노드에 marked 필드를 추가하여 그 노드가 집합에서 제거되어 있는지 표시한다.
 - marked가 true이면 제거되었다는 표시
 - marking을 실제 제거 보다 반드시 **먼저** 수행한다.
 - 또한 marking은 잠금을 획득한 후 수행된다.
 - 순회를 할 때 대상 노드를 잠글 필요가 없고 노드가 head에서 접근할 수 있는지 확인하기 위해 전체리스트를 다시 순회하지 않아도 된다.

리스트의 구현

- 게으른 동기화
 - 구현

```
1  private boolean validate(Node pred, Node curr) {  
2      return !pred.marked && !curr.marked && pred.next == curr;  
3  }
```

```
1  public boolean contains(T item) {  
2      int key = item.hashCode();  
3      Node curr = head;  
4      while (curr.key < key)  
5          curr = curr.next;  
6      return curr.key == key && !curr.marked;  
7  }
```

리스트의 구현

● 게으른 동기화

— 구현

- 낙천적인 방법과 같다.

```
1  public boolean add(T item) {
2      int key = item.hashCode();
3      while (true) {
4          Node pred = head;
5          Node curr = head.next;
6          while (curr.key < key) {
7              pred = curr; curr = curr.next;
8          }
9          pred.lock();
10         try {
11             curr.lock();
12             try {
13                 if (validate(pred, curr)) {
14                     if (curr.key == key) {
15                         return false;
16                     } else {
17                         Node node = new Node(item);
18                         node.next = curr;
19                         pred.next = node;
20                         return true;
21                     }
22                 }
23             } finally {
24                 curr.unlock();
25             }
26         } finally {
27             pred.unlock();
28         }
29     }
30 }
```

리스트의 구현

● 게으른 동기화

— 구현

- 낙천적인 방법과 같으나
노드를 물리적으로
제거하기 전에 마킹을
하여 논리적으로
제거한다.

```
1  public boolean remove(T item) {  
2      int key = item.hashCode();  
3      while (true) {  
4          Node pred = head;  
5          Node curr = head.next;  
6          while (curr.key < key) {  
7              pred = curr; curr = curr.next;  
8          }  
9          pred.lock();  
10         try {  
11             curr.lock();  
12             try {  
13                 if (validate(pred, curr)) {  
14                     if (curr.key != key) {  
15                         return false;  
16                     } else {  
17                         curr.marked = true;  
18                         pred.next = curr.next;  
19                         return true;  
20                     }  
21                 }  
22             } finally {  
23                 curr.unlock();  
24             }  
25         } finally {  
26             pred.unlock();  
27         }  
28     }  
29 }
```


리스트의 구현

- 게으른 동기화

- 이게 제대로 돌아 갈까????

- 다음 명제를 주목

- marking되어 있지 않은 모든 Node는 실제 리스트에 존재하는 살아있는 Node이다!!!

- 헐...

- 보충

- validate에서의 marking검사는 locking이후에 이루어지므로 validate가 OK이면 안전하다.

리스트의 구현

- 실습 : #22

- 게으른 동기화 리스트를 구현하시오
- 실습 #19의 프로그램을 통해 실행 속도를 측정하시오
- 스레드가 1개, 2개, 4개, 8개일 때의 속도를 비교하시오.
 - 각각 실행 전 List를 클리어 하시오

게으른 동기화

- 숙제 5 :

- 게으른 동기화 리스트의 구현 (수업시간에 검사 받은 수강생은 숙제를 제출하지 말것)
- 제출물
 - .cpp 파일
 - 실행속도 비교표 (성긴/세밀한/낙천적 동기화와 비교)
 - CPU의 종류 (모델명, 코어 개수, 클럭)
- 제출 : Eclass에 제출

성긴 동기화

● 결과 (목금반)

	성긴동 기화	세밀한 동기화	낙천적 동기화	게으른 동기화			
1	2819	25310	4541	3125			
2	2902	23125	3799	2142			
4	3153	19000	2782	1337			
8	4111	14171	1709	849			

싱글 스프레드 : 2560

게으른 동기화

- 단점

- 게으른 알고리즘은 **Blocking** 이다.
- 한 스레드가 Lock을 얻은 채로 지연되면, 다른 스레드 역시 지연되게 된다. (Convoying)

- 주의

- Flag를 사용할 때 메모리 업데이트 순서가 중요하므로 `volatile`과 `atomic_thread_fence`를 적절히 사용하던가 `atomic memory`를 사용해야 한다.

리스트의 구현

● 메모리 릭의 해결

— Free List

- Delete하지 않고 모아 놓음
 - marking이 해제되는 순간 오작동 가능
- 언젠가는 재사용 해야함.
 - when????? :
 - 아무도 remove된 node를 가리키지 않을 때
 - remove 시점에서 중복실행 중인 모든 method의 호출이 종료되었을 때

— C++11의 shared_ptr

- 아무도 가리키지 않는 노드 제거에 적합.

메모리 Leak 해결

- shared_ptr란?

- C++11에서 제공하는 일종의 스마트 포인터
- 객체에 reference counter를 두고 이를 통해 앞으로 쓰이지 않을 객체를 판별해서 자동 삭제
- reference counter 증감을 atomic 하게 구현

- shared_ptr를 사용한 구현

- Node의 next를 shared_ptr로 구현
- 각 스레드에서 사용하는 모든 포인터를 shared_ptr로 대체

리스트의 구현

● C++11의 shared_ptr???

```
class SPZLIST {
    shared_ptr <SPNODE> head;
    shared_ptr <SPNODE> tail;
public:
    SPZLIST()
    {
        head = make_shared<SPNODE>(0x80000000);
        tail = make_shared<SPNODE>(0x7fffffff);
        head->next = tail;
    }
    ~SPZLIST()
    {
    }

    void init()
    {
        head->next = tail;
    }
}
```

```
bool Remove(int key)
{
    shared_ptr<SPNODE> pred, curr;

    while(true) {
        pred = head;
        ...
    }
}
```


리스트의 구현

● 실습 : #24

- Shared_ptr을 사용하여 게으른 동기화 리스트에서 메모리 릭을 제거하시오
- 실습 #19의 프로그램을 통해 실행 속도를 측정하시오
- 스레드가 1개, 2개, 4개, 8개일 때의 속도를 비교하시오.
 - 각각 실행 전 List를 클리어 하시오

게으른 동기화

- 속제 6 :

- Shared_ptr을 사용한 메모리 릭이 없는 게으른 동기화 리스트의 구현
- 제출물
 - .cpp 파일
 - 실행속도 비교표
 - CPU의 종류 (모델명, 코어 개수, 클럭)
 - 멀티쓰레드에서 실행시간이 5분 이상 걸리거나, 크래시가 발생할 때
 - 이러한 이상 현상이 생기는 원인에 대한 예측
 - **가능하면** 오류 없이 동작시키고 성능 또한 최적화를 해볼것.
- 제출 : eClass

게으른 동기화

● 결과 (화목반)

	성긴동기화	세밀한 동기화	낙천적 동기화	게으른 동기화	게으른 shared_ptr		
1	1926	25503	4512	2082	18719		
2	1725	23041	3708	1514	X		
4	2043	18275	2560	1030	X		
8	6972	14256	1671	662	X		

싱글 쓰레드 : 1871

구현 차례

- 성긴 동기화(coarse-grained synchronization)
 - Lock하나로 동기화 객체 전체를 감싸는 경우
- 세밀한 동기화 (fine-grained synchronization)
- 낙천적인 동기화 (optimistic synchronization)
- 게으른 동기화 (lazy synchronization)
- 복습
- 비멈춤 동기화 (nonblocking synchronization)

복습

- 현실의 멀티쓰레드 프로그램은?
 - 여러 쓰레드가 동시에 멀티 코어에서 실행된다.
 - 쓰레드간의 데이터 공유 및 동기화는 안전한 Lock-free 자료구조를 통해서 이루어진다.
 - 언리얼 3 : 디스플레이 리스트 Queue
 - 각종 게임 서버 : Log buffer Queue, timer priority queue, 시야 리스트, Party vector, guild list

복습

- Lock-free 알고리즘을 사용하여야 한다.
- 사용하지 않으면
 - 병렬성 감소
 - Priority Inversion
 - Convoying
 - /* 성능이 떨어지고 락이 발생한다 */

복습

- Lock-free 알고리즘이란?
 - 여러 개의 스레드에서 동시에 호출했을 때에도 정해진 단위 시간마다 적어도 한 개의 호출이 완료되는 알고리즘.

??????

복습

- Lock-free **알고리즘**이란?
 - 자료구조 및 그것에 대한 접근 방법
 - 예) QUEUE : enqueue, dequeue
 - 예) STACK : push, pop
 - 예) 이진 트리 : insert, delete, search

복습

● Lock-free 알고리즘이란?

- 멀티쓰레드에서 동시에 호출해도 정확한 결과를 만들어 주는 알고리즘
 - STL 탈락
 - Atomic한 동작
- Non-Blocking 알고리즘
 - 다른 스레드가 어떤 상태에 있건 상관없이 호출이 완료된다.
- 호출이 다른 스레드와 충돌하였을 경우 적어도 하나의 승자가 있어서, 승자는 delay없이 완료된다.

복습

- (보너스)

- Wait-free 알고리즘은?

- 호출이 다른 스레드와 충돌해도 모두 delay없이 완료 된다.

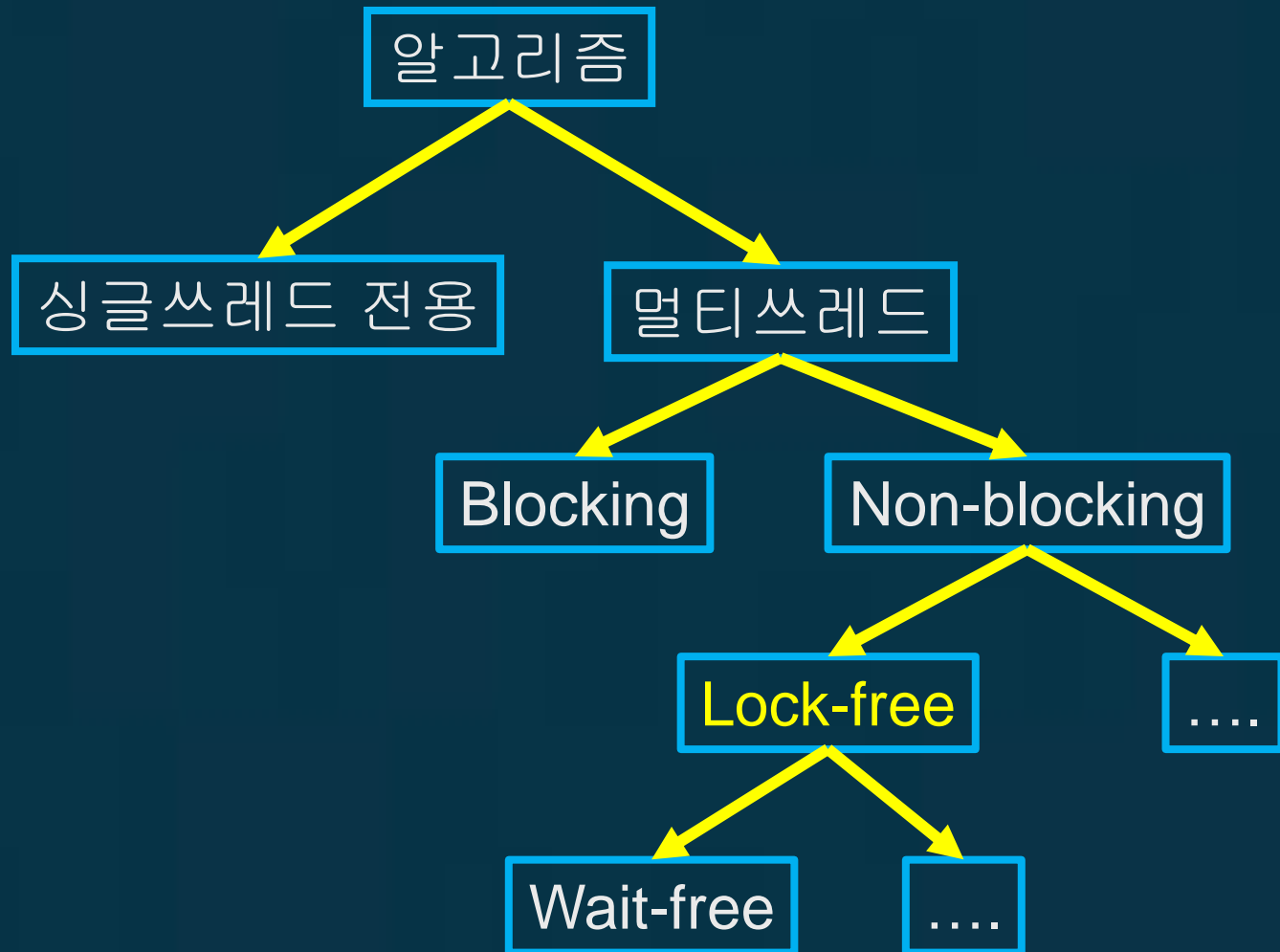
- 추가 상식

- LOCK을 사용하지 않는다고 lock-free 알고리즘이 아니다!!!

- LOCK을 사용하면 무조건 lock-free 알고리즘이 아니다.

복습

● 알고리즘의 분류



복습

- 예) Blocking 알고리즘

int sum;

```
mylock.lock();  
sum = sum + 2;  
mylock.unlock();
```

BLKQUEUE q;

```
mylock.lock();  
q.push(35);  
mylock.unlock();
```

```
while (dataReady == false);  
temp = g_data;
```

복습

● 왜 Blocking인가?

```
while (dataReady == false);  
temp = g_data;
```

- dataReady에 true가 들어가지 않으면 이 알고리즘은 무한 대기, 즉 다른 스레드에서 무언가 해주기를 기다린다.
- 여러 가지 이유로 dataReady에 true가 들어오는 것이 지연될 수 있다.
 - Schedule out, 다른 스레드 때문에 대기

복습

● Non-blocking은?

```
atomic_int sum;  
  
sum += 2;
```

```
BLK_QUEUE::push(int x) {  
    Node *e = New_Node(x);  
    glock.lock();  
    tail->next = e;  
    tail = e;  
    glock.unlock();  
}
```



```
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);  
    }  
}
```

복습

- Non-blocking은?

```
while (dataReady == false);  
temp = g_data;
```



```
if (dataReady == false) return false;  
temp = g_data;
```

복습

● CAS?

- CAS가 없이는 대부분의 non-blocking 알고리즘들을 구현할 수 없다.
 - Queue, Stack, List...
- CAS를 사용하면 모든 싱글쓰레드 알고리즘들을 Lock-free 알고리즘으로 변환할 수 있다!!!
- Lock-free 알고리즘의 핵심

복습

- 정리

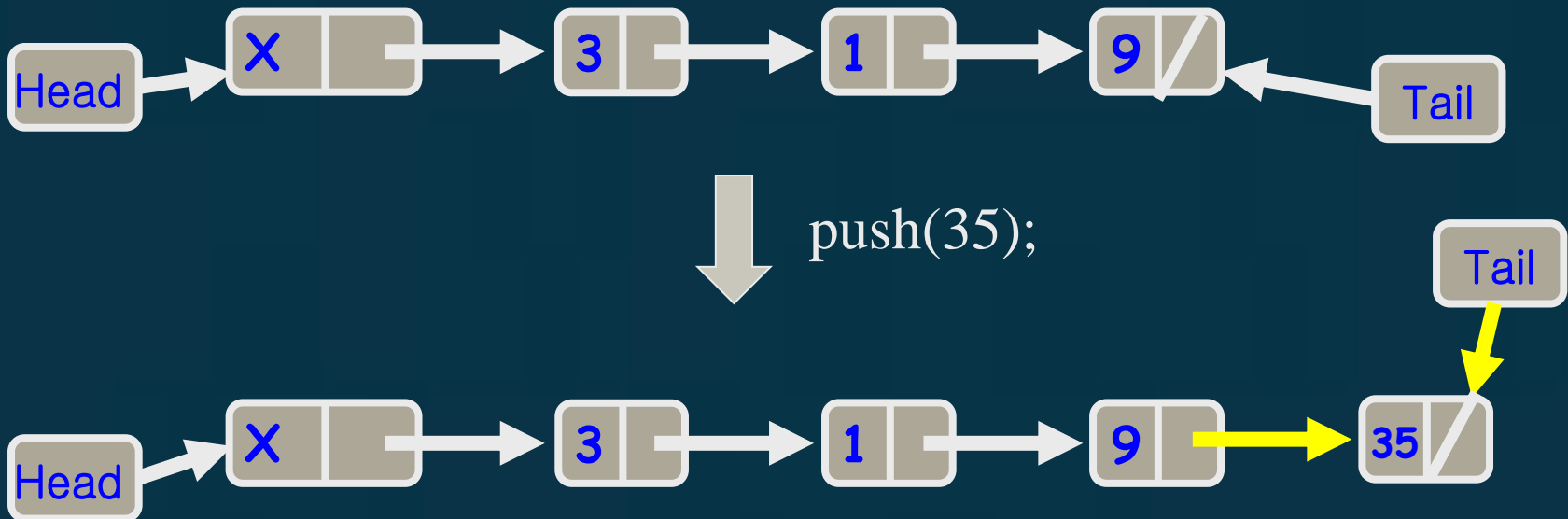
- Lock-free 알고리즘을 써야한다.
 - 성능때문이다.
 - CAS가 꼭 필요하다.

- CAS

- CAS(&A, old, new);
- 의미 : A의 값이 old면 new로 바꾸고 true를 리턴
- 다른 버전의 의미 : A메모리를 다른 스레드가 먼저 업데이트 해서 false가 나왔다. 모든 것을 포기하라.

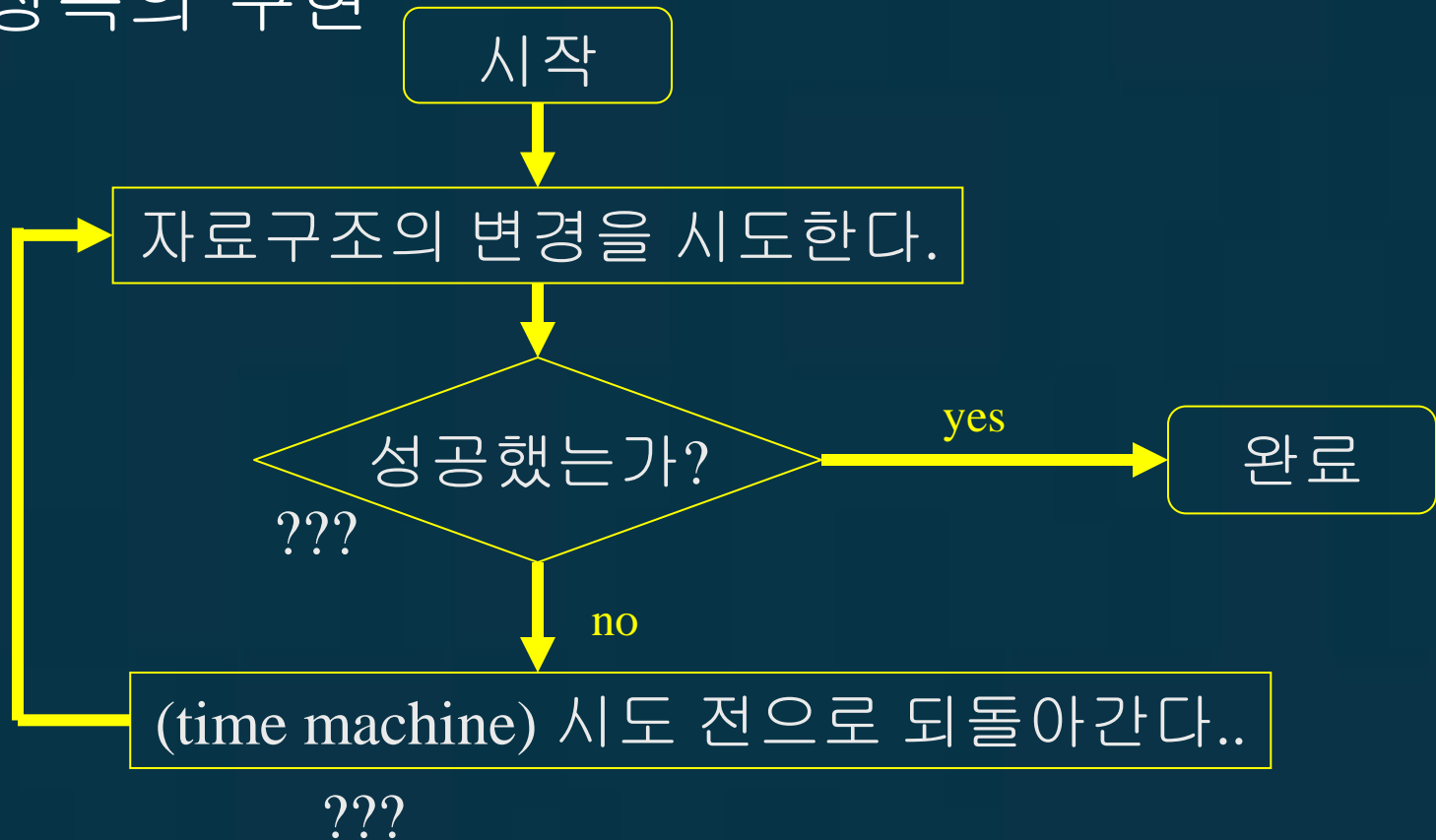
복습

- Lock-free 알고리즘은 어떻게 구현되는가?
- 알고리즘의 동작이란?
 - 기존의 자료구조의 구성을 다른 구성으로 변경하거나 자료구조에서 정보를 얻어내는 행위



복습

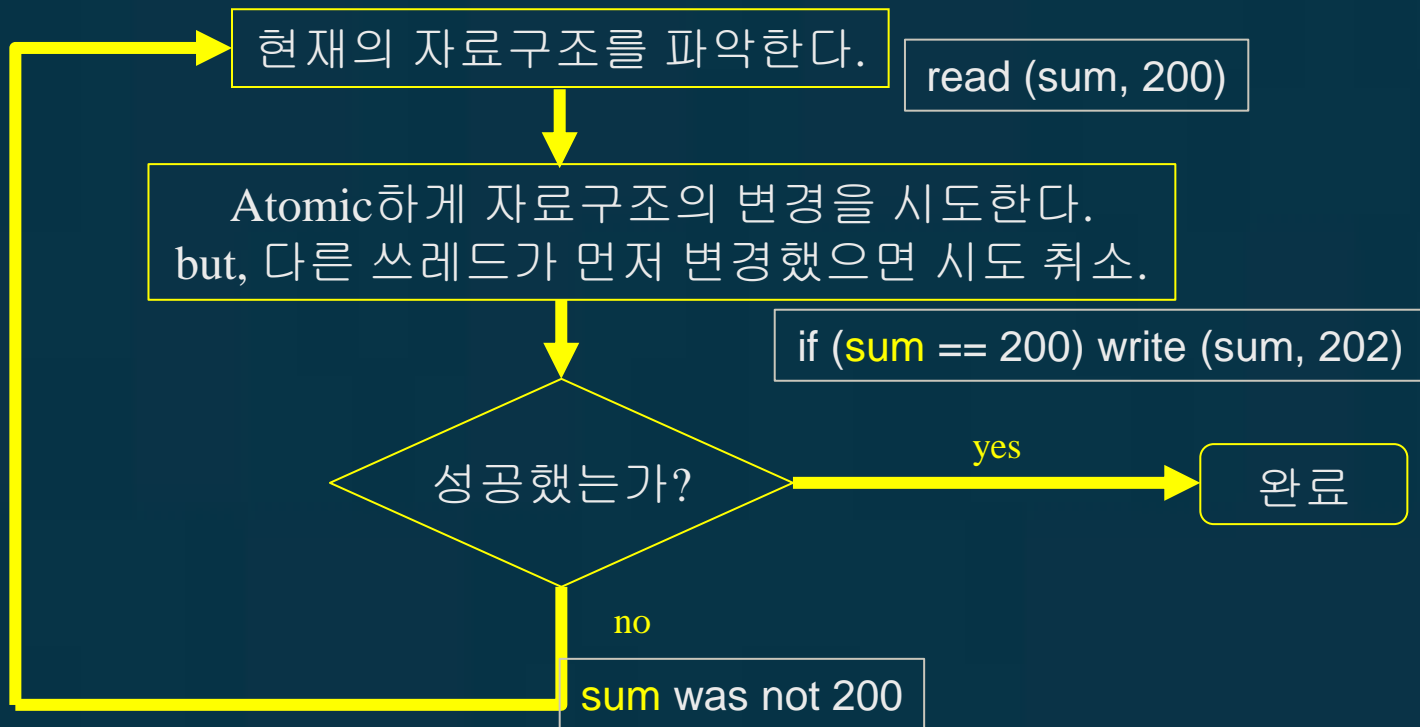
- Lock-free 알고리즘은 어떻게 구현되는가?
 - 상상속의 구현



복습

- Lock-free 알고리즘은 어떻게 구현되는가?
- 앞의 알고리즘이 불가능 하므로

$sum = sum + 2$



복습

Atomic하게 자료구조의 변경을 시도한다.
but, 다른 스레드가 먼저 변경했으면 시도 취소.

```
mylock.lock();  
sum = sum + 2;  
mylock.unlock();
```



```
while (true) {  
    int old_sum = sum;  
    int new_sum = old_sum + 2;  
    if (true == CAS(&sum, old_sum, new_sum)) break;  
}
```



CAS

결과물


대책

atomic하게 자료구조의 변경을 시도한다.
but, 다른 스레드가 먼저 변경했으면 시도 취소.

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



CAS



```
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;  
    }  
}
```

복습

```

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!

현실

```

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);
    }
}

```

복습

● ...

- 알고리즘이 많이 복잡하다.
- 그래서 작성시 실수하기가 쉽다.
- 실수를 적발하기가 어렵다.
 - 하루에 한두 번 서버 크래시
 - 가끔 가다가 아이템 증발
- 제대로 동작하는 것이 **증명된** 알고리즘을 사용해야 한다.

복습

● 결론

- 믿을 수 있는 non-blocking container들을 사용하라.
 - Intel TBB, Visual Studio PPL
- 자신을 포함한 출처가 의심스러운 알고리즘은 정확성을 증명하고 사용하라.
 - 정확성이 증명된 논문에 있는 알고리즘은 OK.

구현 차례

- 성긴 동기화(coarse-grained synchronization)
 - Lock하나로 동기화 객체 전체를 감싸는 경우
- 세밀한 동기화 (fine-grained synchronization)
- 낙천적인 동기화 (optimistic synchronization)
- 게으른 동기화 (lazy synchronization)
- 복습
- 비멈춤 동기화 (nonblocking synchronization)

리스트의 구현

● Non-Blocking 구현

- 게으른 동기화를 통해 만족할 만한 멀티쓰레드 성능향상을 얻었다.
- 하지만 Blocking 구현이어서 성능향상의 여지가 있고, Priority Inversion이나 Convoying에서 자유롭지 못하다.
- Non-Blocking 구현은 게으른 동기화에서 출발한다.
 - Lock으로 보호 받는 CriticalSection을 CAS로 대체해야 하는데, CS구간이 충분히 작아졌다.

리스트의 구현

- Non-Blocking 구현이란?

- Lock()을 사용하지 않는다.

- 서로 충돌하는 thread는 CAS로 **승부**를 낸다.

- CAS 성공

- 무조건 method가 성공적으로 종료해야 한다.

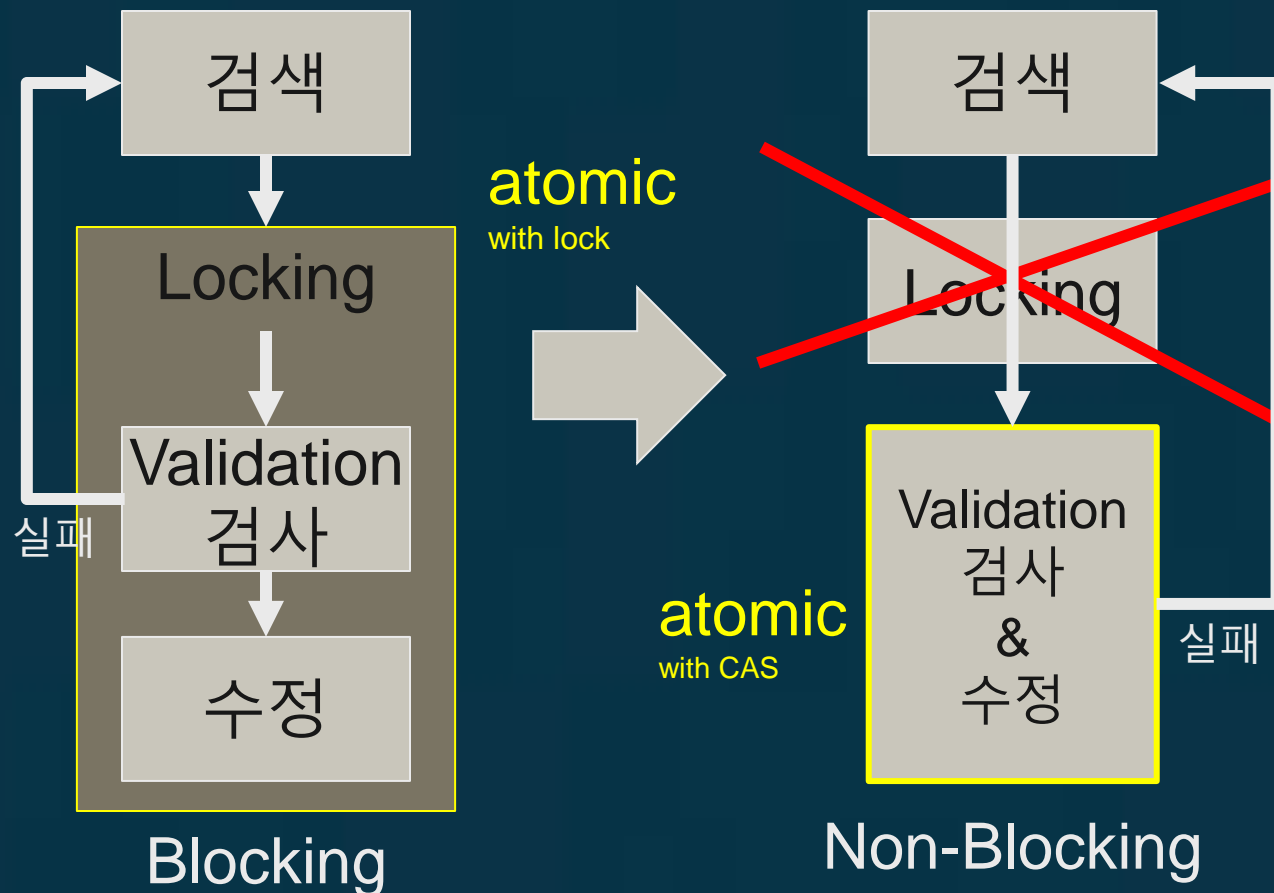
- 적어도 이전 보다는 더 진전된 상태로 바뀌어야 한다.

- CAS 실패

- 졌다는 이야기는 다른 쓰레드에서 먼저 자료구조를 수정했다는 이야기 이므로 지금까지 수집한 자료구조 정보를 더 이상 사용할 수 없고, 다시 수집해야 한다

리스트의 구현

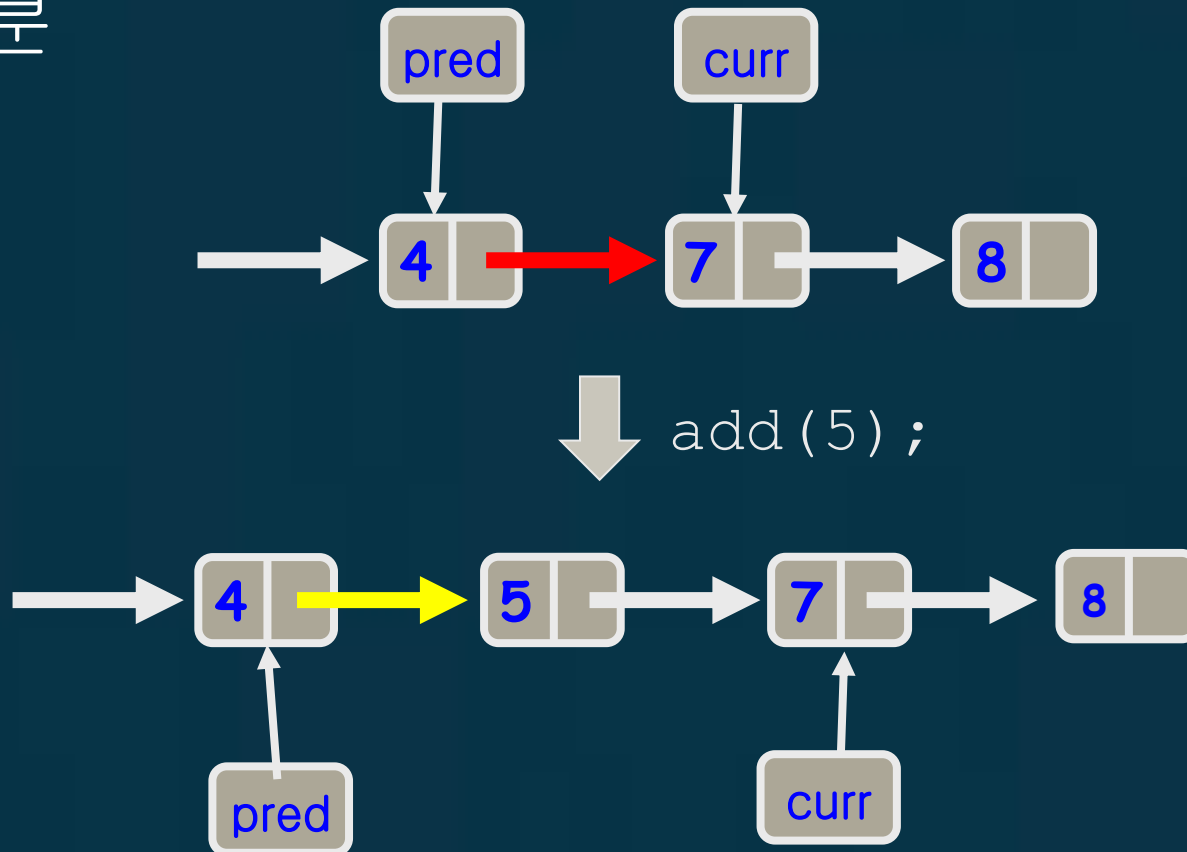
- Non-Blocking 구현이란?



리스트의 구현

● Add의 구현

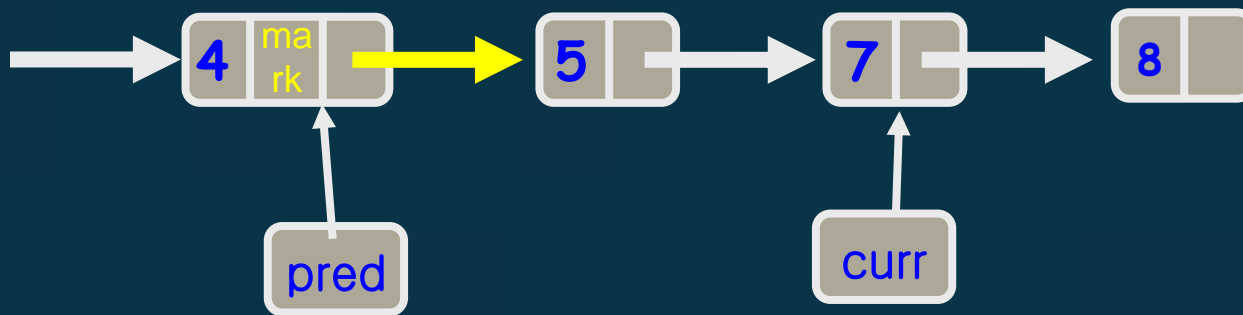
- 자료 구조 중 우리가 **atomic**하게 변경해야 할 부분



리스트의 구현

- Add의 구현

- 다른 쓰레드가 *pred를 Remove하면?
 - Pred의 marking을 확인하면서 변경해야 한다.
- 다른 쓰레드가 *pred를 Remove하면?
 - Pred의 mark가 false인 것을 확인하면서 변경해야 한다.
- 다른 쓰레드가 *pred, *curr사이에 새로운 노드를 끼워 넣으면?
 - Pred의 next가 curr인 것을 확인하면서 변경해야 한다.



리스트의 구현

- Add의 구현의 의문

- Pred와 curr를 locking하지 않기 때문에 당연히 발생할 수 있는 불상사들
- 이를 검출하려면 next와 marking을 동시에 감시하면서 CAS를 수행하면 된다.
 - 즉 next와 marking을 다른 스레드가 건드렸다면 아무것도 하지 않고 처음부터 다시 수행해야 한다.
 - Next와 marking을 동시에 CAS 해야 한다.
 - `CAS(pred->making, false, false, pred->next, curr, new_node);`

- 동시 CAS??? Double CAS?

- 그런 연산은 x86 CPU에 존재하지 않는다.

```
class LFNODE {  
    int    key;  
    bool  marking;  
    NODE * next;  
}
```


리스트의 구현

- 동시 CAS의 구현

- CAS의 한계

- 한번에 하나의 변수 밖에 바꾸지 못한다.
 - Marking과 Next의 Atomic한 동시 변환이 가능해야 한다.

- 극복

- 한 장소에 주소와 Marking을 동시에 저장
 - 주소와 marking을 access하는 별도의 API작성

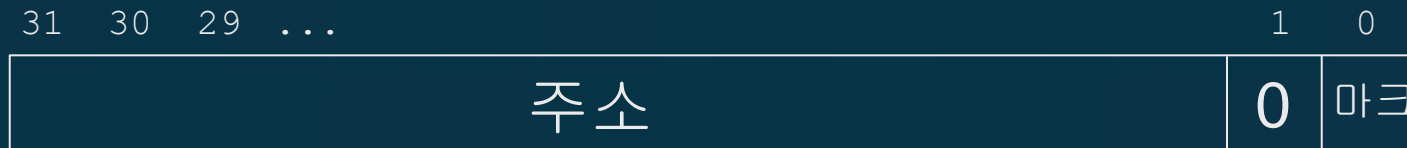
- 일종의 꼼수, 특수한 경우에만 사용 가능

리스트의 구현

● 비멈춤 동기화

– 유사 멀티 CAS 구현

- NEXT.CAS(oldmark, mark, old_next, new_next)
- 32비트 주소 중 LSB를 마크로 사용 (1비트를 mark로 사용)
 - NEXT 필드를 포인터로 직접 사용할 수 없게 되었으므로, 모든 next 필드를 통한 Node이동 시 type변환이 필요하다.
 - Debugging이 어려워짐
 - 포인터가 아닌 다른 데이터 타입으로 선언하는 경우



NEXT의 구성

비임출 동기화

- Mark 주소의 CAS 구현 예제
 - LFNODE의 메소드

```
bool LFNODE::CAS(int old_v, int new_v)
{
    return atomic_compare_exchange_strong(
        reinterpret_cast<atomic_int*>(&next), &old_v, new_v);
}

bool LFNODE::CAS(LFNODE *old_node, LFNODE *new_node,
    bool oldMark, bool newMark) {
    int oldvalue = reinterpret_cast<int>(old_node);
    if (oldMark) oldvalue = oldvalue | 0x01;
    else oldvalue = oldvalue & 0xFFFFFFFF;

    int newvalue = reinterpret_cast<int>(new_node);
    if (newMark) newvalue = newvalue | 0x01;
    else newvalue = newvalue & 0xFFFFFFFF;

    return CAS(oldvalue, newvalue);
}
```

비멈춤 동기화

- API

- 마킹 : CAS사용

- CAS(LFNODE *old_node, LFNODE *old_node, false, true);

- 그 외 구현이 필요한 함수

- GetNext() : 합성 자료구조(next)에서 주소만 가져오기
 - GetNextWithMark() : 주소와 마킹값을 동시에 가져오기
 - GetMark() : 마킹값을 가져오기
 - AtomicMarkableReference : 주소와 마킹값으로 합성 자료구조 만들기

비임춤 동기화

- curr 를 delete하려면?
 - 두 단계가 필요하다.
 - curr 를 마킹하고 (CAS 사용)
 - pred의 next를 curr의 next로 변경한다. (CAS 사용)
 - pred의 marking을 확인하면서 변경
 - 하지만 우리는 pred와 curr를 잠그지 못한다!
 - 따라서 위의 2번째 과정에서 실패할 수 있다.
 - curr를 마킹한 순간 다른 스레드가 pred를 지울 수 있다!
 - 마킹되지 않은 이전 노드를 찾아서 next를 curr의 next로 변경해야 한다.
 - 이 과정에서 다른 스레드들과 충돌 할 수 있고, 이를 해결하려면 알고리즘이 복잡해 진다..
 - Pred와 curr사이에 다른 노드가 추가되었을 수도 있다

비멈춤 동기화

- 현실적인 대안 : 정책 변경
 - Remove시 제거를 시도하지만 실패하면 무시한다.
 - 리스트 중간 중간에 marking된 노드가 존재하는 것을 받아들인다.
 - 리스트의 정의를 변경하고, 변경된 리스트에서도 올바르게 동작하도록 **모든** 메소드를 수정한다.
 - Add(), Remove(), Conatins() 메소드를 마킹되었지만 remove되지 않은 노드가 있는 경우에도 동작하도록 수정한다.
 - 마킹된 노드를 고려한 수정은 너무 어렵다.
 - **메소드 호출 시 마킹된 노드를 제거하고 진행한다.**
 - 제거하면서 검색한다..

리스트의 구현

● 검색

- Add와 Remove에서 행하는 검색의 변경
- 검색 시 마킹 노드의 삭제를 동시에 행한다.

```

1  class Window {
2      public Node pred, curr;
3      Window(Node myPred, Node myCurr) {
4          pred = myPred; curr = myCurr;
5      }
6  }
7  public Window find(Node head, int key) {
8      Node pred = null, curr = null, succ = null;
9      boolean[] marked = {false};
10     boolean snip;
11     retry: while (true) {
12         pred = head;
13         curr = pred.next.getReference();
14         while (true) {
15             succ = curr.next.get(marked);
16             while (marked[0]) {
17                 snip = pred.next.compareAndSet(curr, succ, false, false);
18                 if (!snip) continue retry;
19                 curr = succ;
20                 succ = curr.next.get(marked);
21             }
22             if (curr.key >= key)
23                 return new Window(pred, curr);
24             pred = curr;
25             curr = succ;
26         }
27     }
28 }

```

리스트의 구현 (2021-화목, 목금)

● 구현

— 9번 줄의 메모리 낭비 주의

```
1  public boolean add(T item) {  
2      int key = item.hashCode();  
3      while (true) {  
4          Window window = find(head, key);  
5          Node pred = window.pred, curr = window.curr;  
6          if (curr.key == key) {  
7              return false;  
8          } else {  
9              Node node = new Node(item);  
10             node.next = new AtomicMarkableReference(curr, false);  
11             if (pred.next.compareAndSet(curr, node, false, false)) {  
12                 return true;  
13             }  
14         }  
15     }  
16 }
```


리스트의 구현

- 구현

```
17  public boolean remove(T item) {
18      int key = item.hashCode();
19      boolean snip;
20      while (true) {
21          Window window = find(head, key);
22          Node pred = window.pred, curr = window.curr;
23          if (curr.key != key) {
24              return false;
25          } else {
26              Node succ = curr.next.getReference();
27              snip = curr.next.attemptMark(succ, true);
28              if (!snip)
29                  continue;
30              pred.next.compareAndSet(curr, succ, false, false);
31              return true;
32          }
33      }
34  }
```

리스트의 구현

● 구현

```
35    public boolean contains(T item) {  
36        boolean[] marked = false{};  
37        int key = item.hashCode();  
38        Node curr = head;  
39        while (curr.key < key) {  
40            curr = curr.next;  
41            Node succ = curr.next.get(marked);  
42        }  
43        return (curr.key == key && !marked[0])  
44    }
```

* 40행을 `curr = curr.next.GetReference();`로 수정할 것

리스트의 구현

- 구현 단계 (1/2)

- Next필드 관련 메소드 추가

- GetReference()
 - 합성 주소에서 주소만 리턴
 - Get(*mark)
 - 합성 주소에서 주소를 리턴하고 동시에 마킹여부를 call by reference로 리턴
 - AttemptMark(*nextnode, mark)
 - CAS를 사용하여 마킹 시도
 - CompareAndSet(oldmark, newmark, *oldnode, *newnode)
 - 합성 주소에 대한 CAS

- 두 가지 구현 옵션

- Next 필드를 “Node*” 에서 “MarkableReference”로 타입 변경
 - 프로그래밍 시 혼돈 감소
 - 디버깅 불편
 - Node에 Next 필드 관련 메소드 추가

리스트의 구현

- 구현 단계 (2/2)
 - 검색 함수 구현 : `find(pred, curr)`
 - 덩치가 커졌으므로 공통 함수로 분리가 바람직하다.
 - Java와 달리 C++는 `window`라는 구조체가 굳이 필요 없음.

리스트의 구현

- 실습 : #23

- 비멈춤 동기화 리스트를 구현하시오
- 실습 #19의 프로그램을 통해 실행 속도를 측정하시오
- 스레드가 1개, 2개, 4개, 8개일 때의 속도를 비교하시오.
 - 각각 실행 전 List를 클리어 하시오

리스트의 구현

● 주의사항

```
LFNODE* get_next(bool *removed)
{
    long long value = next;
    *removed = 1 == (value & 0x1);
    return reinterpret_cast<LFNODE*>(value & LSB_MASK);
}
```

```
LFNODE* get_next(bool *removed)
{
    *removed = 1 == (next & 0x1);
    return reinterpret_cast<LFNODE*>(next & LSB_MASK);
}
```

속제 7

● 비멈춤 동기화 리스트의 구현

— 제출물

- .cpp 파일
 - (주의) Eclass에 올려놓은 Sample을 가지고 완성 시킬것
- 실행속도 비교표 (성긴동기화, 세밀한 동기화, 낙천적, 게으른)
- CPU의 종류 (모델명, 코어 개수, 클럭)

— 제출 : e-class

Non Blocking 동기화

- 결과 (목금반)

	성긴동기화	세밀한 동기화	낙천적 동기화	게으른 동기화	게으른 shared_ptr	LF 동기화	
1				6408		4709	
2				4978	X	4226	
4				3064	X	3296	
8				2016	X	1880	

싱글 쓰레드 : 5119

리스트의 구현

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

	1	2	4	8	16	32	64
Course	2203	5969	7672	9547	22970	22829	22861
Fine	63626	52207	32080	19610	19657	28000	27205
Opt	4422	3922	2984	1672	1281	875	1109
Lazy	1922	1672	921	531	562	468	578
LF	2390	2062	1922	781	359	1125	259

리스트의 구현

- Lock-Free List의 메모리 릭
 - Lazy와 같다.
 - 메소드 실행 중에 참조하고 있는 노드가 재사용되는 경우 오동작의 위험성이 매우 높다.
- 재사용을 위해서는?
 - Free-List를 사용한다.
 - shared_ptr를 사용한다.

리스트의 구현

● Free-List의 구현

- Remove되는 Node들을 Free-List에 넣는다.
- New 대신 Free-List에 있는 Node들을 사용한다.
- 재사용해도 안전한 노드들만 재사용한다.
 - 최근(?)에 remove된 노드들은 재사용하지 않는다..

● Lock Free Free-List의 필요성

- 당연히 Lock Free 알고리즘에서는 Lock Free, Free-List가 필요하다.

리스트의 구현

- 메모리 릭의 해결방법
 - Atomic shared ptr
 - stamped pointer
 - LF QUEUE 챕터에서 다룸
 - EPOCH (Epoch Based Reuse)
 - Lock Free Free-List
 - Hazard Pointer

EPOCH

- 모든 공유 객체는 Epoch Counter와 Thread Epoch Counter[쓰레드 개수]를 갖고 있다.
- Method호출 시
 - EPOCH Counter를 증가시킨다.
 - EPOCH Counter를 Thread Epoch Counter[My_Thread_ID]에 저장한다.
 - 메소드 실행
 - Thread Epoch Counter[My_Thread_ID] = 0;
- 각 쓰레드들은 자신만의 memory pool을 관리한다.
- delete된 노드를 자신의 memory pool에 MAX(Thread Epoch Counter[])와 같이 넣는다.
- 일정 시간이 지나면 memory pool에 있는 객체들을 실제로 delete한다.
 - Thread Epoch Counter[]의 0이 아닌 최소값 보다 작은 값을 갖고 있는 객체만 delete한다.
- 단점
 - 어느 한 쓰레드의 Thread Epoch Counter가 증가하지 않는 경우 memory leak과 다름 없다.

EPOCH

- EPOCH

- eclass sample code 참조

- 참조 :

- https://www.cs.rochester.edu/u/scott/papers/2018_PPoPP_IBR.pdf

EPOCH

● EPOCH

— 구현 :

```
// globals
1 int epoch
2 int reservations[thread_cnt]
3 thread_local int counter
4 thread_local list retired
5 epoch_freq // freq. of increasing epoch
6 empty_freq // freq. of reclaiming retired

// private function
7 Function empty() : void
8   max_safe_epoch = reservations.min()
9   for block ∈ retired do
10     /* all blocks retired IN or AFTER
11       max_save_epoch will be protected */
12     if block.retire_epoch < max_safe_epoch then
13       free(block)
```

```
// public interface
12 Function retire(block* ptr) : void
13   retired.append(ptr)
14   ptr->retire_epoch = epoch
15   counter++
16   if counter % epoch_freq == 0 then
17     fetch_and_increment(epoch)
18   if retired.cnt % empty_freq == 0 then
19     empty()

20 Function start_op() : void
21   reservations[tid] = epoch
22 Function end_op() : void
23   reservations[tid] = MAX
```

EPOCH

● EPOCH

– 사용 :

```
// set-ups
1 memory_manager<node> mm; // 3 nodes with values 0, 2 and
  4
2 node n0(0), n1(2), n2(4)
3 n0.next = &n1
4 n1.next = &n2
5 n2.next = NULL
6 node* head = &n
```

```
// 2 worker threads
7 Function tA() : int // Thread A tries
  to read n1
8     mm.start_op()
9     node* target = &n1
10    node* p1 = mm.read(&target)
11    if p1 , NULL // n1 still exists
12    then
13        int ret = p1->val; mm.end_op()
14        return ret;
15    else
16        mm.end_op()
17    return NULL;
```

```
// 2 worker threads
18 Function tB() : void // Thread B tries to
  update n1
19    while true do
20        mm.start_op()
21        node* new_n1 = mm.alloc(sizeof(node))
22        new_n1->val = 3
23        new_n1->next = &n2
24        node* target = &n0
25        node* p0 = mm.read(&target)
26        if p0 ≠ NULL then
27            if mm.CAS(&p0->next, &n1, new_n1) then
28                mm.retire(&n1)
29                mm.end_op()
30            return
```


리스트의 구현

- 다시 한번 더 shared_ptr
 - LFNODE에서 사용 가능한가?
 - NO!
 - 우리는 합성 포인터 자료구조를 사용하고 있다.
 - 하지만 조금 더 보충을 해보면...

리스트의 구현

- shared_ptr의 문제점

- 느리다

- 일반적인 포인터의 load, store는 32비트 integer의 load, store와 같다.
 - 하지만 shared_ptr의 store는 atomic counter의 두 번 업데이트를 필요로 한다.
 - 기존 데이터의 counter 감소 새 데이터의 counter증가.

- Atomic하지 않다.

- Counter의 관리는 atomic하다.
 - Pointer 자체의 접근이 atomic하지 않다.
 - 예) `g_shared_ptr_a = g_shared_ptr_b;`
 - shared_ptr가 기본 자료구조가 아니라 구조체이다.

리스트의 구현

● shared_ptr의 문제점 해결

— 느리다면.

- Pointer의 update가 자주 발생하는 경우 shared_ptr로 구현하지 않는다.
- 함수의 parameter에는 const reference를 사용한다.

— Atomic하지 않다면.

- atomic_shared_ptr를 사용한다.
(<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4162.pdf>)
- 상용 atomic_shared_ptr를 구매해서 사용한다.
- 주의해서 atomic하게 사용한다.

리스트의 구현

- shared_ptr의 atomic한 사용.
 - Load를 atomic하게 하는 법
 - `atomic_load(&shared_ptr_a);`
 - Store를 atomic하게 하는 법
 - `atomic_exchange(&shared_ptr_a, shared_ptr_b)`
 - 문제점
 - 위의 방법 사용 시 글로벌 lock이 걸린다.

리스트의 구현

- shared_ptr의 atomic한 사용.
 - 자체적으로 구현 Atomic_shared_ptr 사용
 - 돈주고 산다
 - 구현한다.
 - 기다린다.
 - C++20 표준

정리

- List를 사용한 병렬 Set의 구현
 - add, remove, contains method 구현
- Lock-Free 까지 구현 및 성능 비교
- CAS를 사용한 Non-Blocking 프로그래밍 기법
- Marking을 사용한 Lazy Programming 기법
- Mark와 Pointer의 합성

이후의 전개

- CAS없이 non-blocking 자료구조를 만들수 없음을 증명
- Lock-free Queue의 구현
 - ABA 문제
 - Reference Counting의 구현
- Lock-free Stack의 구현
 - TOP 노드에서의 bottleneck 해소
- $O(\log n)$ 검색 List구현
 - Lock-free SkipList
 - Free List를 통한 Node 재사용의 구현

질문???
