



7. 병렬 알고리즘 – STACK SKIP-LIST

멀티쓰레드 프로그래밍
정내훈

목차

- Stack

- SKIPLIST

목차

- 소개
- 무제한 무잠금 스택
- 백오프 스택
- 소거 백오프 스택
 - 무잠금 교환자
 - 소거 배열

소개

- Stack
 - 후입선출(LIFO) 구조
 - Push(), Pop() 메서드를 제공

무제한 성긴 동기화 스택

- 무제한 성긴 동기화 스택은 연결리스트로 구성되고 top 필드가 첫 노드를 가리킨다.
 - 만약 스택이 비어있을 경우는 nullptr
 - -1을 스택에 추가하는 것은 고려하지 않는다.
 - Empty일 경우 Pop()은 -2을 리턴한다.

무제한 성긴 동기화 스택

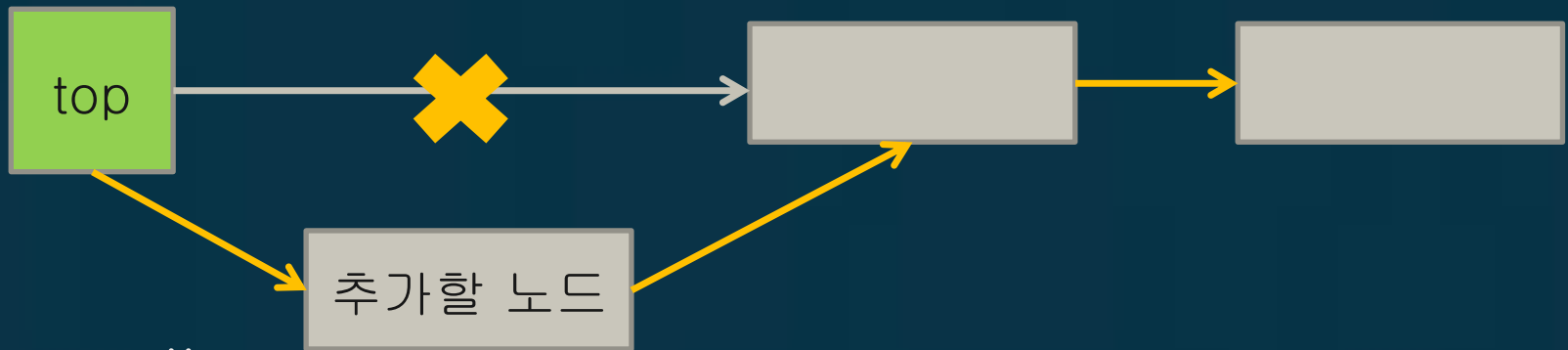
- Code

- class Node

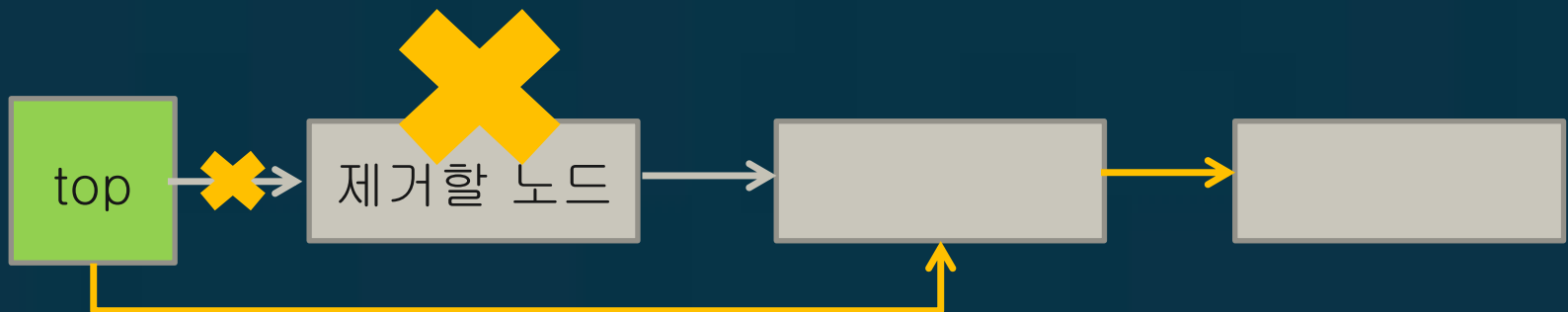
```
class Node
{
public:
    int value;
    Node *next;
    Node() { }
    ~Node() { }
    Node(int input_value) {
        value = input_value;
        next = nullptr;
    }
};
```

무제한 무잠금 스택

● Push()



● Pop()



무제한 성긴 동기화 스택

● Code

– Single Thread Code

```
int Pop()
{
    if (nullptr == top) return -2;
    int temp = top->key;
    NODE *ptr = top;
    top = top->next;
    delete ptr;
    return temp;
}
```

```
void Push(int x)
{
    NODE *e = new NODE{ x };
    e->next = top;
    top = e;
}
```


무제한 성긴 동기화 스택

- 실습 #20

- 성긴 동기화 스택을 구현하라.
- 1,2,4,8 쓰레드에서의 성능비교를 하라.
- 아래 벤치마크 프로그램을 사용하라.

```
void ThreadFunc(int threadNum)
{
    for (int i=1; i<NUM_TEST / threadNum; i++) {
        if ((rand() % 2) || (i < 1000 / threadNum)) {
            Stack.Push(i);
        } else {
            Stack.Pop();
        }
    }
}
```

Stack

- 성능 (수목반)

- Intel® Core™ i7-4770@ 2.20Hz, 4Core, Hyperthread, 16GB Memory
- Single Thread : 877ms

Thread	Coarse Stack	Lock-Free Stack	STAMPED LF QUEUE	EBR LF QUEUE
1	1245	636	1135	473
2	1545	797	1027	534
4	1661	1139	1158	652
8	1968	724	1269	774
16	1960	730	1280	790

Stack

- 성능

- Intel® Core™ i7-7700@ 3.60Hz, 4Core, Hyperthread, 64GB Memory
- Single Thread : 785ms

Thread	Coarse Stack	Lock-Free QUEUE	STAMPED LF QUEUE	EBR LF QUEUE
1	1252	757	1135	473
2	1268	636	1027	534
4	1358	797	1158	652
8	1560	1139	1269	774
16	1570	1177	1280	790

무제한 무잠금 스택

- 무잠금 스택은 CompareAndSet을 이용하여 구현한다.
- Top의 변환을 CAS를 사용하여 non-blocking으로 구현한다.
- ABA 문제가 있으므로 delete를 하지 않는다.

무제한 무잠금 스택

● 실습 #21

- 무제한 Lock-Free 스택을 구현하라.
- 1,2,4,8 쓰레드에서의 성능비교를 하라.
- 아래 벤치마크 프로그램을 사용하시오.
- 교재의 try-pop이니 try_push은 무시하시오.

```
void ThreadFunc(int threadNum)
{
    for (int i=1; i<NUM_TEST / threadNum; i++) {
        if ((rand() % 2) || (i < 1000)) {
            Stack.Push(i);
        } else {
            Stack.Pop();
        }
    }
}
```

무제한 무잠금 스택

- 숙제 #11 <<< 2020년도 생략 >>>
 - 무제한 Lock-Free 스택을 구현하라.
 - Try_pop이니 try_push 같은 것이 들어있는 프로그램을 베끼지 마시오.
 - 제출물
 - .cpp 파일
 - 실행속도 비교표 (Lock버전, Lock free)
 - CPU의 종류 (모델명, 코어 개수, 클럭)
 - 제출 : 생략

Stack

- 성능

- Intel® Core™ i7-7700@ 3.60Hz, 4Core, Hyperthread, 64GB Memory
- Single Thread : 785ms

Thread	Coarse Stack	Lock-Free Stack	STAMPED LF QUEUE	EBR LF QUEUE
1	1252	773		
2	1268	653		
4	1358	761		
8	1560	986		
16	1570	1027		

무제한 무잠금 스택

- 결론

- CAS를 사용하여 Lock-free 스택을 구현하였다.
- 메서드 호출은 스택의 top에 대해 성공한 CAS호출의 순서로 하나씩 진행되므로 순차병목현상이 나타날 수 있다.

- 메모리 문제

- new와 delete를 사용하면. ABA문제가 생긴다.
- Queue보다 문제가 생길 확률이 크다.

BACK OFF 스택

- CAS 동기화의 문제

- 경쟁이 심할 경우 CAS 실패 시 계속 재시도하는 것은 전체 시스템에 악영향을 줌
 - Thread가 많아질 수록 경쟁이 심해짐
 - 경쟁이 심할 경우 CAS가 실패할 확률이 높음
 - 실패할 경우 성공할 때 까지 반복
 - CAS를 실행할 경우 같은 CPU의 모든 Core의 메모리 접근이 중단됨
 - Thread가 많아질 수록 잦은 메모리 접근 중단

목차

- 소개
- 무제한 무잠금 스택
- 백오프 스택
- 소거 백오프 스택
 - 무잠금 교환자
 - 소거 배열

BACK OFF 스택

- BackOff

- 교재 7장에 나옴

- 경쟁이 심할 경우 경쟁을 줄이자.

- CAS의 실패는 경쟁이 심함을 뜻함.
 - CAS가 실패했을 경우 **적절한 기간** 동안 실행을 멈추었다가 재개 하자.
 - CAS의 실패 확률이 낮아짐 -> 메모리 접근 중단 감소

- **적절한 기간**

- 처음에는 짧게
 - 계속 실패하면 점점 길게
 - 첫 번 시도에 성공하면 짧게
 - Thread마다 기간을 다르게 해야 한다.

BACK OFF 스택

- BackOff 객체

- CAS가 실패 했을 경우 다음 CAS를 시도하기 전에 사용.
- 사용될 때 마다. BackOff 시간 간격 증가.
- 시간 간격은 범위 안에서 랜덤하게

BACK OFF 스택

- BackOff class

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

    void InterruptedException() {
        int delay = 0;
        if (limit != 0) delay = rand() % limit;
        limit *= 2;
        if (limit > maxDelay) limit = maxDelay;
        this_thread::sleep_for(chrono::microseconds(delay));
    }
};
```

Stack

- 성능

- Intel® Core™ i7-7700@ 3.60Hz, 4Core, Hyperthread, 64GB Memory
- Single Thread : 785ms

Thread	Coarse Stack	Lock-Free Stack	Backoff LF Stack	EBR LF QUEUE
1	1252	773	801	
2	1268	653	816	
4	1358	761	757	
8	1560	986	765	
16	1570	1027	748	

Stack

- 성능 (수목반)

- Intel® Core™ i7-4770@ 2.20Hz, 4Core, Hyperthread, 16GB Memory
- Single Thread : 877ms

Thread	Coarse Stack	Lock-Free Stack	BackOff LF Stack	BackOff-2 LF STack
1	1245	636	737	736
2	1545	797	643	599
4	1661	1139	783	833
8	1968	724	1031	1167
16	1960	730	1082	1190

Stack

- 성능 (화목반-B조)

- Intel® Core™ i7-4770@ 2.20Hz, 4Core, Hyperthread, 16GB Memory
- Single Thread : 877ms

Thread	Coarse Stack	Lock-Free Stack	BackOff LF Stack	BackOff-2 LF STack
1	1245	795	793	819
2	1545	667	832	624
4	1661	779	862	752
8	1968	1130	779	974
16	1960	1166	801	1028

BACK OFF 스택

- Better backoff

```
int delay = 0;
if (limit != 0)
    delay = rand() % limit;
limit *= 2;
if (limit > maxDelay)
    limit = maxDelay;
int start, current;
__asm RDTSC;
__asm mov start, eax;
do {
    __asm RDTSC;
    __asm mov current, eax;
} while (current - start < delay);
```

Stack

- 성능

- Intel® Core™ i7-7700@ 3.60Hz, 4Core, Hyperthread, 64GB Memory
- Single Thread : 785ms

Thread	Coarse Stack	Lock-Free Stack	Backoff LF Stack	Backoff-2 LF STack
1	1252	773	801	800
2	1268	653	816	730
4	1358	761	757	883
8	1560	986	765	1207
16	1570	1027	748	1225

BACK OFF 스택

- Better backoff – 2

- 절대 시간이 중요한가? CPU가 빠르면 Critical Section 처리 속도도 빠르다.
- 루프에서 메모리를 아예 접근하지 않는 것이 바람직하다.

```
int delay = 0;
if (0 != limit) delay = rand() % limit;
if (0 == delay) return;
limit += limit;
if (limit > maxdelay) limit = maxdelay;

_asm mov eax, delay;
myloop:
_asm dec eax
_asm jnz myloop;
```

Stack

- 성능 (수목반)

- Intel® Core™ i7-4770@ 2.20Hz, 4Core, Hyperthread, 16GB Memory
- Single Thread : 877ms

Thread	Coarse Stack	Lock-Free Stack	BackOff LF Stack	BackOff-3 LF STack
1	1245	636	737	723
2	1545	797	643	635
4	1661	1139	783	807
8	1968	724	1031	1166
16	1960	730	1082	1186

Stack

- 성능

- Intel® Core™ i7-7700@ 3.60Hz, 4Core, Hyperthread, 64GB Memory
- Single Thread : 785ms

Thread	Coarse Stack	Lock-Free Stack	Backoff LF Stack	Backoff-3 LF STack
1	1252	773	801	783
2	1268	653	816	710
4	1358	761	757	783
8	1560	986	765	942
16	1570	1027	748	965

BACK OFF 스택

● 실습 #22

- 무제한 무잠금 BackOFF Lock-Free 스택을 구현하라.
- 1,2,4,8 쓰레드에서의 성능비교를 하라.
- 아래 벤치마크 프로그램을 사용하라.

```
void ThreadFunc(int threadNum)
{
    for (int i=1; i<25000000 / threadNum; i++) {
        if ((rand() % 2) || (i < 1000)) {
            Stack.Push(i);
        } else {
            Stack.Pop();
        }
    }
}
```

BACK OFF 스택

- 숙제 12 : << 생략 >>

- Lock Free BackOff Stack의 구현

- 강의자료실에 있는 LFSTACK활용

- 제출물

- .cpp 파일
- 실행속도 비교표 (Lock버전, Lock free, Lock free BackOff)
- CPU의 종류 (모델명, 코어 개수, 클럭)

- 제출 : nhjung@kpu.ac.kr

- 제목 : [2019 멀티코어 프로그래밍 숙제 12] 학번, 이름
- 11월 19일 화요일 오전 11시까지 제출

BACK OFF 스택

- BackOff Stack 성능 (8 Core, 4 CPU XEON)

	1	2	4	8	16	32	64
BackOff STACK	1370	1158	1233	1198	1749	2977	2986
LF STACK	1392	1236	2298	2704	4739	6779	7081

목차

- 소개
- 무제한 무잠금 스택
- 백오프 스택
- 소거 백오프 스택
 - 무잠금 교환자
 - 소거 배열

소거

- 병행성 문제

- Queue나 Stack은 리스트의 말단 부분에서 잦은 충돌이 생긴다.
 - fine grain synchronization타입의 최적화가 불가능하다.
 - Queue는 $\frac{1}{2}$ 로 충돌이 분산된다.
- Stack의 경우 충돌을 회피해서 병렬로 처리하는 방법이 가능
 - 아이디어 :
 - push, pop이 동시에 발생하는 경우 꼭 Stack에 넣어야 하나?

소거

● 소거

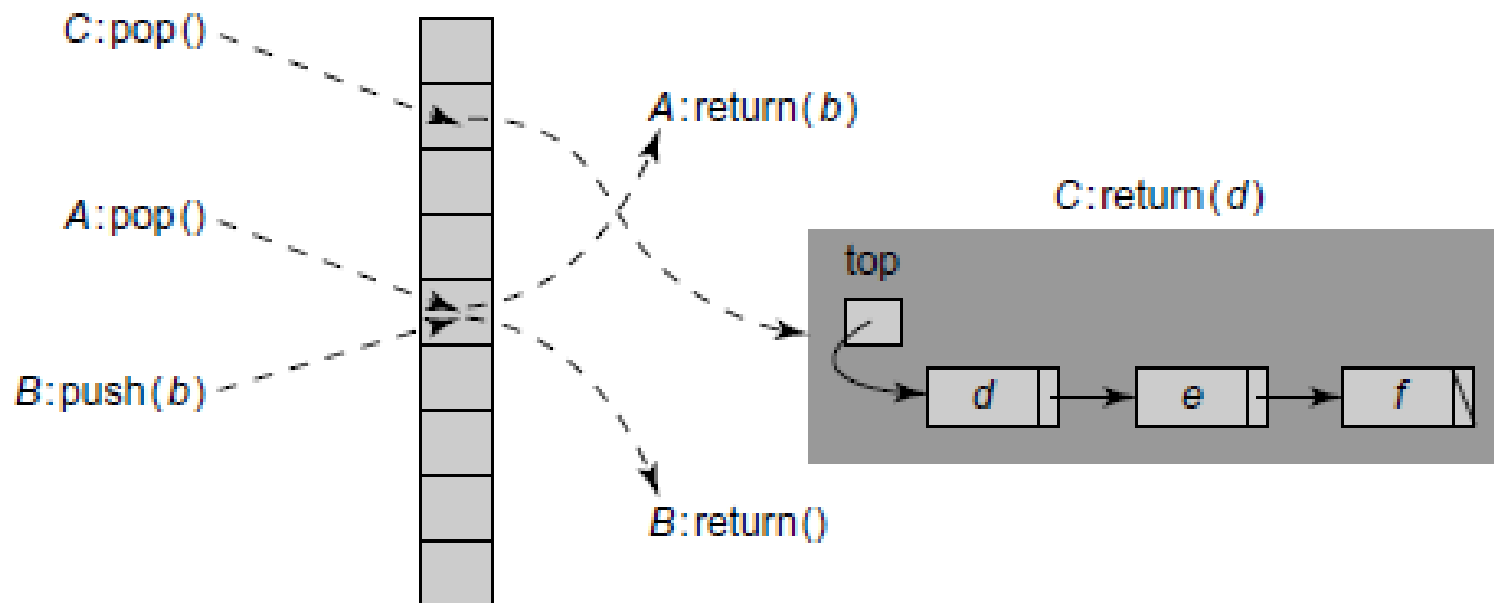
- 많은 쓰레드가 서로 충돌할 경우 Stack에 넣지 않고 직접 Data를 주고 받도록 한다.
 - Stack을 통하지 말고 다른 객체를 통해 전달
- Lock-Free로 Data를 주고 받도록 한다.
- 높은 경쟁률에 대비하여 주고 받는 별도의 객체를 복수로 준비한다.

소거

- 만약 push와 pop이 거의 동시에 실행된다면 두 연산은 서로 취소되어 없어지고 스택에 접근하지 않는다.
- 이런 경우 push를 호출하는 스레드는 스택의 변동 없이 pop을 호출하는 스레드와 값을 교환할 수 있다.
 - 이 때 서로를 소거(eliminate)하게 된다고 한다.

소거

- 소거(elimination)



소거

- EliminationArray
 - 자료교환의 장소 (교환자(Exchanger) 객체)
 - 여러 개의 원소를 갖고 부하를 분산
 - 서로 만나지 못할 경우도 있다.
 - 경쟁이 심할 경우 많은 장소가 효율적이고, 경쟁이 적을 경우는 장소가 적어야 한다.
- 스레드는 EliminationArray에서 임의의 항목을 골라서, 서로를 소거하려 시도
 - 잘못된 종류의 호출을 만난 경우는 소거 실패
 - 예) push와 push, pop과 pop
 - 임의로 하지 않고, 짝을 잘 맞추어 주면?
 - 정보 전달 자체가 오버헤드
 - Lock-free로 하는 것은 어렵다. Open-Problem

소거 백오프 스택 (2020-수목)

- 앞에서 구현했던 LockFreeStack에서 백오프를 사용하지 않고, 대신에 소거를 시도한다.
 - 소거 :
 - Elimination Array의 임의의 원소를 선택한다
 - 임의 : 협조 자체가 오버헤드
 - Array의 크기는 가변
 - 소거 시도 :
 - 아무도 없으면 기다림
 - 다른 스레드가 기다리고 있으면 교환
 - 이미 교환 중이면 재시도 [array size 증가 필요]
 - 주의점
 - Time Out 필요 [array size 감소 필요]
 - 서로 만난 스레드가 <push>, <pop>의 조합이라는 보장이 없음
 - 잘못된 만남은 실패

무잠금 교환자

- Elimination Array의 원소
- 두개의 쓰레드가 Lock free로 값을 교환할 수 있게 해주는 객체
- Exchange(myitem)의 메소드를 갖고 있으며 두개의 쓰레드가 Exchange()를 호출하면 서로의 입력 값을 리턴한다.
 - pop도 자기 값(-1)을 상대방에게 준다.
- 리턴 값은 교환값과 상태(성공, 실패, 타임아웃(-2)) 이다.


```

1 public class LockFreeExchanger<T> {
2     static final int EMPTY = ..., WAITING = ..., BUSY = ...;
3     AtomicStampedReference<T> slot = new AtomicStampedReference<T>(null, 0);
4     public T exchange(T myItem, long timeout, TimeUnit unit)
5         throws TimeoutException {
6         long nanos = unit.toNanos(timeout);
7         long timeBound = System.nanoTime() + nanos;
8         int[] stampHolder = {EMPTY};
9         while (true) {
10             if (System.nanoTime() > timeBound)
11                 throw new TimeoutException();
12             T yrItem = slot.get(stampHolder);
13             int stamp = stampHolder[0];
14             switch(stamp) {
15                 case EMPTY:
16                     if (slot.compareAndSet(yrItem, myItem, EMPTY, WAITING)) {
17                         while (System.nanoTime() < timeBound){
18                             yrItem = slot.get(stampHolder);
19                             if (stampHolder[0] == BUSY) {
20                                 slot.set(null, EMPTY);
21                                 return yrItem;
22                             }
23                         }
24                     if (slot.compareAndSet(myItem, null, WAITING, EMPTY)) {
25                         throw new TimeoutException();
26                     } else {
27                         yrItem = slot.get(stampHolder);
28                         slot.set(null, EMPTY);
29                         return yrItem;
30                     }
31                 break;
32                 case WAITING:
33                     if (slot.compareAndSet(yrItem, myItem, WAITING, BUSY))
34                         return yrItem;
35                 break;
36                 case BUSY:
37                     break;
38                 default: // impossible
39                     ...
40             }
41         }
42     }
43 }

```

무장금 교환자 (2020-화목B조)

- 구현

- Slot

- 처음 교환을 시도한 쓰레드의 입력값을 저장하는 공간
 - 두 번째 쓰레드가 자신의 값을 넘겨 주는 공간

- 교환기는 3개의 상태를 가진다.

- EMPTY

- 슬롯에 아무런 값도 저장되지 않은 상태

- WAITING

- 처음 온 쓰레드에서 슬롯에 값을 하나 저장한 상태
 - 다른 쓰레드가 와서 읽어가기를 기다리는 상태

- BUSY

- 나중에 온 쓰레드가 슬롯의 값을 읽어가고 자신의 값으로 변경한 상태
 - 처음 쓰레드는 아직 읽기를 완료하지 않음

무잠금 교환자

● 상태에 따른 구현 (1/2)

— EMPTY

- CAS를 이용하여 슬롯에 자신의 아이템을 넣고 상태를 WAITING으로 바꾸기를 시도한다.

— 실패

- 다른 스레드가 이미 CAS를 성공한 경우이다.
- 다른 스레드가 waiting으로 변경
- 처음부터 다시 시도

— 성공

- 스핀을 하면서 다른 스레드가 교환을 끝내길 기다림
- 교환이 완료(BUSY) 된 후에는 대기 중이던 스레드는 아이템을 가져간 후 상태를 empty 으로 바꾼다.
 - Empty()로 전환하는 일은 CAS를 이용하지 않는다
 - 오직 한 스레드 만 작업하기 때문

무잠금 교환자

● 상태에 따른 구현 (2/2)

— EMPTY

- Waiting으로 바꾼 후 다른 스레드를 기다림
- 다른 스레드가 나타나지 않는다면,
 - 대기중이던 스레드는 slot의 상태를 CAS를 호출하여 EMPTY로 전환
 - 성공시
 - 시간제한 예외 발생
 - 실패시
 - 다른 스레드가 나타나서 교환을 완료하고 Busy로 변경했다는 의미이므로 대기중이던 스레드는 교환을 완료한다.

무잠금 교환자

● 구현

—WAITING

- 어떤 스레드가 대기중이며 슬롯은 그 스레드의 아이템을 갖는다.
 - CAS를 이용하여 자신의 아이템을 적고 대기중인 스레드의 아이템을 얻는 것을 시도한다.(BUSY상태로 전환)
 - 실패시에는 처음부터 다시
 - 성공하면 얻은 아이템을 반환한다.

—BUSY

- 현재 다른 두 스레드가 슬롯을 써서 값을 교환하고 있으므로 처음부터 다시.

무장금 교환자 (2020-화목 A조)

● 구현

– AtomicStamperReference의 구현

- 값의 교환과 상태 변경을 Atomic하게 구현해야 한다.
- StampedQueue와 같은 구현?
 - InterlockedCompareExchange64
 - InterlockedCompareExchange128
- 해결책
 - 32비트 int의 상위 2비트 사용
 - 교환할 수 있는 값의 최대값 감소

무잠금 교환자

● ABA 문제??

— 해당 알고리즘은 ABA문제를 일으키지 않는다.

- ABA현상이 발생한다.
- 하지만 그냥 진행해도 문제없다.
- 교환대상이 바뀐 것 뿐이고 값도 그대로이기 때문에 아무 문제 없다.

— 사실은 CAS를 통해서 old값을 읽기 때문에 상태만 그대로이면 값이 바뀌는 것 까지도 문제가 없다.

● 타임아웃 문제

— 너무 짧은 교환 시간은 항상 실패하게 되므로 시간제한 기간을 고를 때 주의해야 한다.

소거 배열

- 소거 배열 (Elimination array)
 - Exchange객체의 배열.
 - capacity개의 원소를 갖는다.
 - visit()
 - 소거배열이 갖고 있는 exchange원소 중 하나를 랜덤하게 선택해서 교환을 시도한다.
 - range값을 통해 random값의 범위를 조정할 수 있다.

소거 배열

- Code

- Class Elimination

```
class EliminationArray {
    int range;
    LockFreeExchanger exchanger[MAX_THREAD];
public:
    EliminationArray() { range = 1;}
    ~EliminationArray() {}
    int Visit(int value, bool *time_out) {
        int slot = rand()% range;
        bool busy;
        int ret = exchanger[slot].exchange(value, time_out, &busy);
        if ((true == *time_out) && (range > 1)) range--;
        if ((true == busy) && (range <= num_thread / 2)) range++;
        // MAX RANGE is # of thread / 2
    }
};
```

소거 배열

- EliminationBackoffStack

- LockFreeStack에서 CAS에 실패했을 경우 백오프를 하는 대신에 EliminationArray를 써서 값의 교환을 시도한다.

- Push()

- CAS(...)

- 성공하면 리턴

- 자신의 입력값을 인자로 Visit()를 호출

- timeout이 아니면

- 다른 스레드와의 교환이 성공

- pop과의 교환이었으면 리턴 (교환된 값이 -1)

- pop과의 교환이 아니었으면 다시 CAS로~~

- timeout이면

- 다시 CAS로~~~

소거 배열

● EliminationBackoffStack

— Pop()

- CAS(...) 시도
 - 성공하면 return
- -1 을 인자로 visit()를 호출
 - timeout이 아닌 경우 : 교환 성공
 - push()호출과 교환했는지 검사 (교환값 != 0)
 - push()호출과의 교환이면 교환값을 리턴
 - 다시 CAS()로~~~
 - Timeout인 경우
 - 다시 CAS()로~~~.

소거 배열

- RangePolicy

- EliminationArray의 LoadBalancing
- EliminationArray에서 사용 중인 최대 인덱스 값을 갖는다.
- 경쟁이 심하면 값을 늘리고
 - 방문했을 때 Busy인 경우
- timeout이 발생하면 값을 줄인다.

소거 배열

● EliminationBackoffStack

— 만약 교환이 실패한다면 원인은?

- 처음부터 교환이 없었거나, 같은 종류의 연산과 값이 교환되었을 경우 실패한다.
- 이런 경우 다시 top에 대한 CAS를 재시도 한다.

소거 배열

● EliminationBackoffStack

— 교환자의 위치 선택 범위는?

- 범위의 크기와 충돌 확률은 반비례적
- 스레드의 수가 적을 경우 범위는 작아야하고 스레드수가 클 경우 범위가 넓어야 한다.
- RangePolicy객체를 이용하여 범위를 지정한다.
 - 간단하게는 time_out 하면 범위를 줄이고 exchanger에 너무 많은 스레드가 몰리면 범위를 늘린다.

실습

- 실습 #21 : EliminationBackoffStack
 - Lock-Free 소거 백오프 Stack을 완성하라
 - ST_WAIT와 ST_BUSY
 - range 변경
 - 전체 루프 타임아웃 구현
 - 1, 2, 4, 8 쓰레드에서의 성능비교를 하라.
 - 실습 #20과의 성능 비교를 하라.

속제

- 속제 12 :
 - LockFree 소거 Stack의 구현
 - 제출물
 - .cpp 파일
 - 실행속도 비교표 (Lock버전, Lock free, Lock free 소거)
 - CPU의 종류 (모델명, 코어 개수, 클럭)
 - 제출 : eclass

Stack

- 성능(화목)

- Intel® Core™ i7-7700@ 3.60Hz, 4Core, Hyperthread, 64GB Memory
- Single Thread : 857 ms

Thread	Coarse Stack	Lock-Free Stack	Backoff-2 LF Stack	Elimination LF Stack
1	1077	732	692	
2	1191	573	644	
4	1316	616	661	
8	1491	737	752	
16	1505	766	753	

결론

● EliminationBackoffStack

- LockFreeStack과 달리 확장될 가능성 있음
- 부하가 높아지면 성공적인 소거가 증가하고, 병렬적으로 여러 개의 연산이 완료된다
- 소거된 연산은 절대로 스택에 접근하지 않기 때문에 LockFreeStack에서의 경쟁이 줄어든다.

목차

- Stack

- SKIPLIST

SKIP-LIST

- 소개
- 순차 스킵 리스트
- 병행 스킵 리스트
- Lock-Free 스킵 리스트

소개

- 지금까지의 자료구조
 - Queue, Stack
 - Set
 - Linked List, 검색시간 $O(n)$
- $O(n)$ 검색시간?
 - 실제로 사용하기에는 너무 무리
 - 원소의 개수가 적을 경우에는 사용 가능
- 실제 사용하는 자료구조
 - Hash : 순서가 필요 없는 경우
 - Binary Tree 계열 (B+, Red-Black, Heap, AVL..)

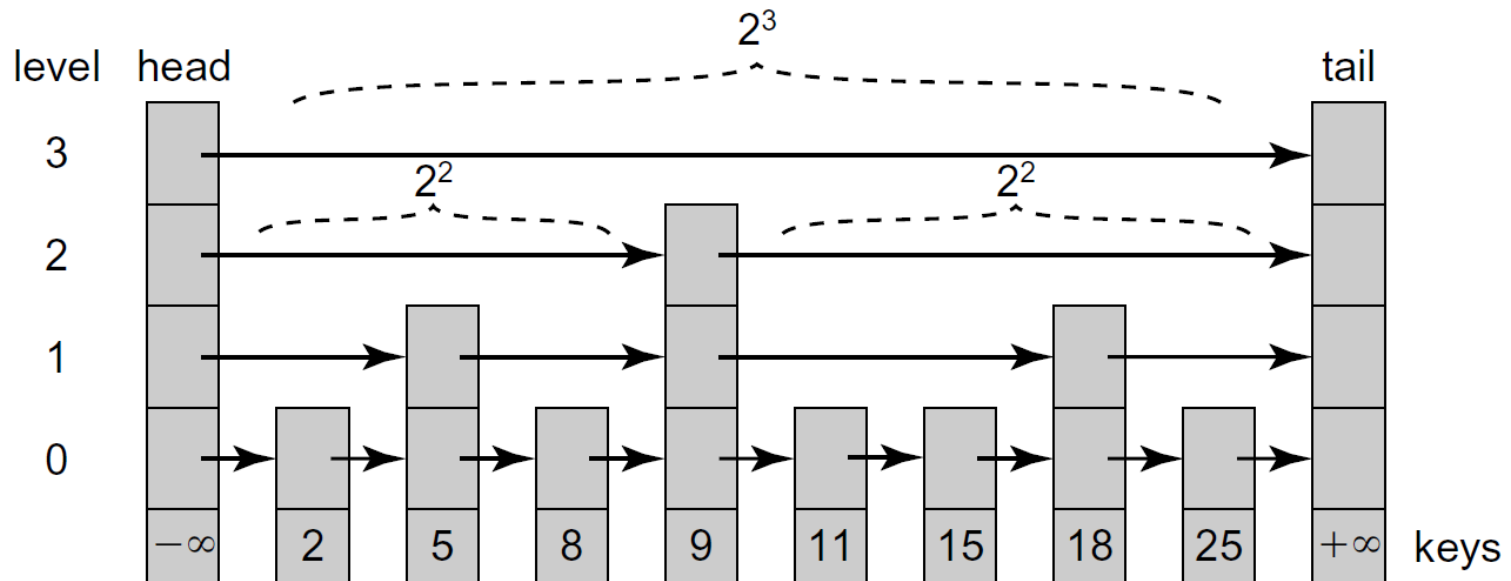
소개

- 일반적인 트리구조
 - 트리의 깊이를 유지하기 위해서 정기적인 재균형(Rebalancing) 작업이 필요(AVL트리..)
 - 하지만 병행 구조에서는 재균형작업이 병목이나 경쟁상태를 유발할 가능성이 있다.
- SkipList
 - 평균 $O(\log n)$ 검색시간을 갖는 병행 검색 구조
 - 재균형작업이 필요 없음!!!
 - 랜덤 자료구조(Randomized algorithm, Probabilistic data structure)
 - Worst Case $O(n)$ 임

소개

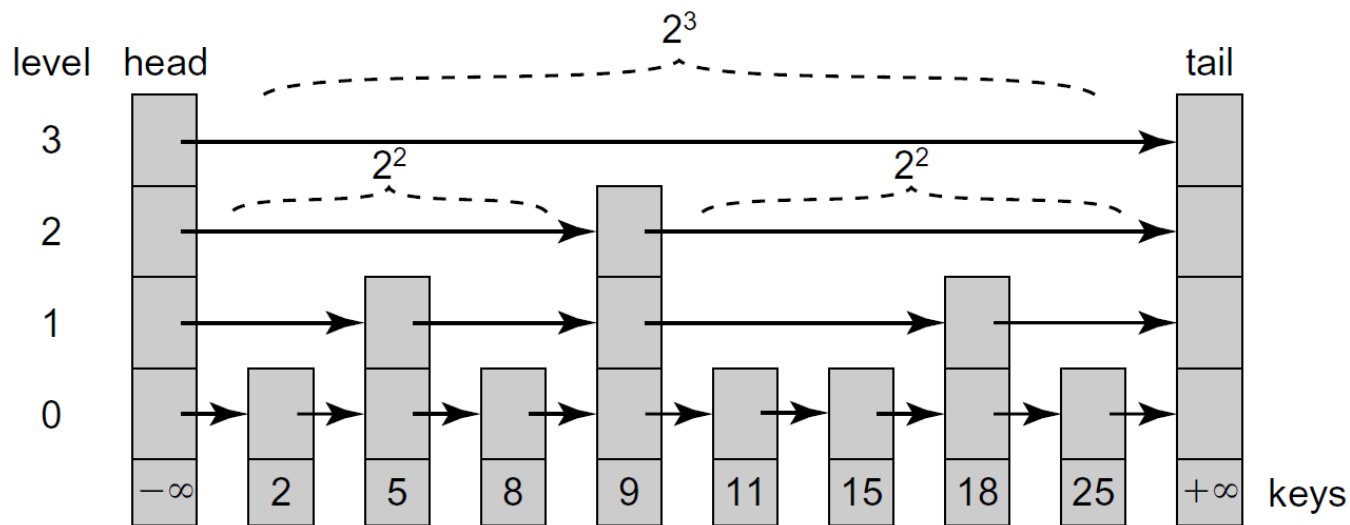
● 스킵리스트의 예

- Set을 구현할 때 사용한 연결리스트의 확장
- 노드들이 추가 포인터를 갖고 있으며, 같은 레벨의 포인터끼리 연결됨. ($\text{next} \rightarrow \text{next}[n]$)



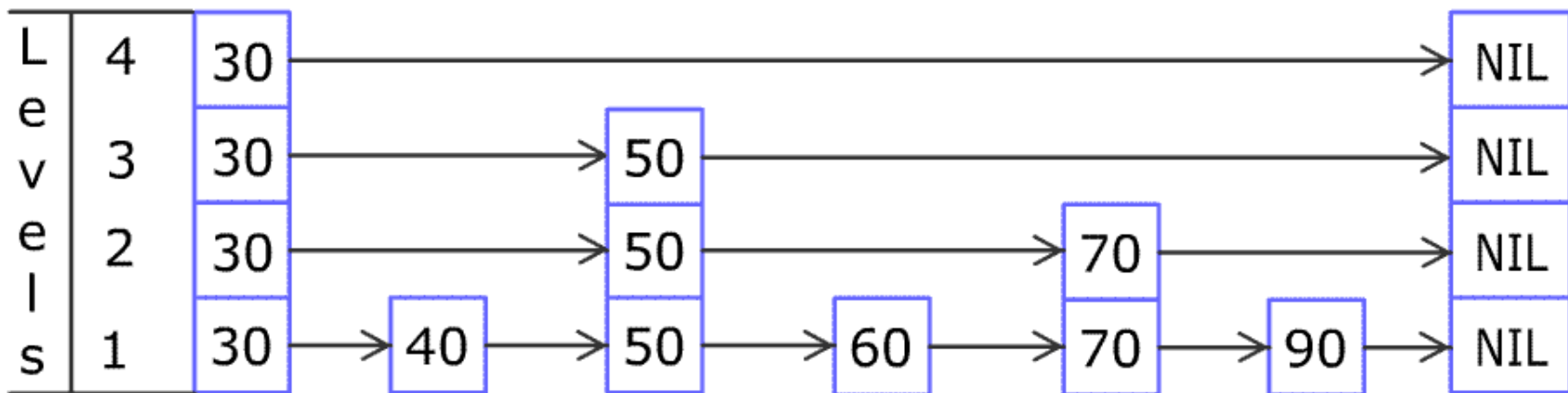
소개

- 스킵리스트의 기본 아이디어
 - 추가 링크 (레벨 1 이상) 지름길을 만든다.
 - 노드를 추가하거나 제거할 때 지름길을 유지한다.
 - 검색을 할 때 지름길을 우선 이용한다.



소개

- 스킵리스트의 동작

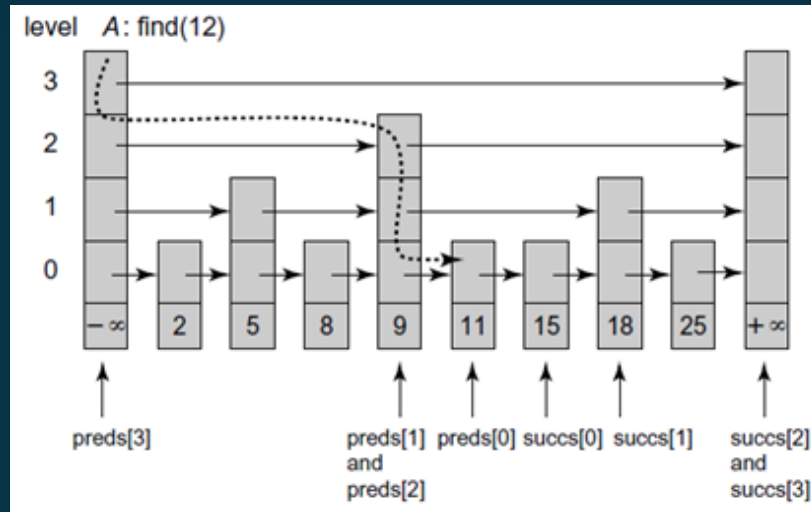


SKIP-LIST (2020-화목)

- 소개
- 순차 스킵 리스트
- 병행 스킵 리스트
- Lock-Free 스킵 리스트

순차 스킵리스트

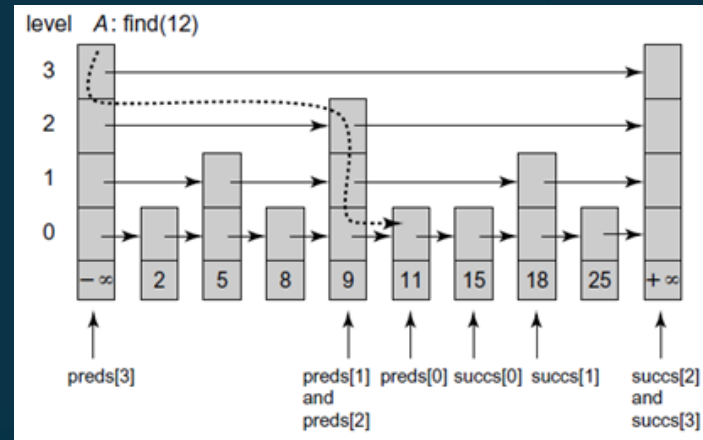
● 순차 스킵리스트 : Find()



- 언제나 시작은 head에서,
- 높은 레벨의 포인터부터 검색 시작
- 한 레벨의 검색이 끝나면 다음 레벨의 검색 시작
 - 레벨별 검색결과 저장
- 맨 아래 레벨에 도달할 경우 종료된다.

순차 스킵리스트

- 순차 스킵리스트
 - 검색의 구현 Find()

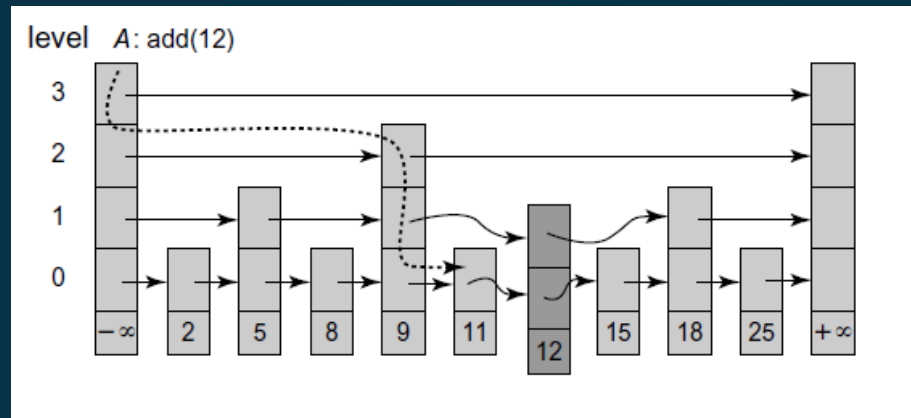


— 더 높은 층의 리스트는 낮은 층의 리스트에 대한 지름길이다.

- 계층의 각 연결은 바로 아래 계층의 2개의 노드를 건너 뛴다

순차 스킵리스트

● 순차 스킵리스트 : Add()



- 랜덤한 높이의 노드를 생성한다.
 - 높이가 높을 수록 노드의 개수는 감소 (단계 마다 $\frac{1}{2}$)
- 검색한 위치에 추가한다.
- pred, curr 가 아닌 pred[], curr[]가 필요하다.
 - remove()도 마찬가지

순차 스킵리스트

- 노드 정의

```
class Node
{
    public:
    int key;
    Node* next[MAX_LEVEL];
    int topLevel;

    Node() {
        for ( int i=0; i<MAX_LEVEL; i++ )    next[i] = NULL;
        topLevel = MAX_LEVEL;
    }
    Node(int myKey) {
        key = myKey;
        for ( int i=0; i<MAX_LEVEL; i++ ) next[i] = NULL;
        topLevel = MAX_LEVEL;
    }
    Node(int x, int height) {
        key = x;
        for ( int i=0; i<MAX_LEVEL; i++) next[i] = NULL;
        topLevel = height;
    }
};
```

순차 스킵리스트

● Node 구조

```
class NODE
{
public:
    int value;
    NODE *next[MAX_LEVEL];
    int top_level;
    NODE()
    {
        value = 0;
        for (auto i = 0; i < MAX_LEVEL; ++i)
            next[i] = nullptr;
        top_level = 0;
    }
    NODE(int x, int top)
    {
        value = x;
        for (auto i = 0; i < MAX_LEVEL; ++i)
            next[i] = nullptr;
        top_level = top;
    }
};
```

순차 스킵리스트

● 공통함수 : Find

```
void Find(int key, SLNODE *preds[], SLNODE *currs[])
{
    int cl = MAXHEIGHT - 1;
    while (cl >= 0) {
        if (MAXHEIGHT - 1 == cl) preds[cl] = &head;
        else preds[cl] = preds[cl + 1];
        currs[cl] = preds[cl]->next[cl];
        while (currs[cl]->key < key) {
            preds[cl] = currs[cl];
            currs[cl] = currs[cl]->next[cl];
        }
    }
}
```


리스트의 구현

● 실습 : #22

- 성긴 동기화 스킵리스트를 구현하시오
- 1과 1000사이의 숫자를 랜덤하게 4백만회 삽입/삭제하는 프로그램을 통해 실행 속도를 측정하시오 (다음 페이지)
- 쓰레드가 1개, 2개, 4개, 8개일 때의 속도를 비교하시오.
 - 각각 실행 전 Skip-List를 클리어 하시오

리스트의 구현

- 실습 : #22
 - test program

```
#define NUM_TEST    4000000
#define KEY_RANGE   1000
```

```
void *ThreadFunc(void *lpVoid)
{
    int key;

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

숙제

- 숙제 14 :

- 성긴 동기화 SkipList의 구현

- 수업시간에 작성한 프로그램을 디버깅하여 완성하시오

- 제출물

- .cpp 파일
- 실행속도 비교표 (쓰레드 개수 별)
- CPU의 종류 (모델명, 코어 개수, 클럭)

- 제출 : nhjung@kpu.ac.kr

- 제목 : [2019 멀티코어 프로그래밍 숙제 14] 학번, 이름
- 11월 26일 화요일 오후 1시까지 제출

SKIP-LIST

- 소개
- 순차 스킵 리스트
- 병행 스킵 리스트
- Lock-Free 스킵 리스트

잠금 기반의 병행 스킵리스트

- Non-Blocking Skip-list의 전 단계
- Mark field를 사용한 Lazy 방식의 Skip-list

잠금 기반의 병행 스킵리스트

- 조감도

- 노드

- 개별적인 잠금
 - Marked 필드(bool)
 - Remove()시 논리적으로 제거하고 있는 중이라면 true
 - next[n]
 - 각 층에 해당하는 포인터의 배열
 - 보초노드 (head, tail)
 - 초기(SkipList가 비어있을 경우)에는 head의 모든 층은 tail을 가리킨다.

잠금 기반의 병행 스킵리스트

- Add()

- Add()는 항상 Find()를 호출

- Find()

- 스킵리스트를 순회하고 모든 층에 대해, 앞 노드와 뒤 노드를 반환

- 노드가 추가되는 동안 앞 노드가 변경되는 것을 방지하기 위해 앞 노드들을 잠근다.

- fullyLinked 필드

- 모든 층에서 추가된 노드에 제대로 참조를 설정할 때까지 논리적으로 집합에 있지 않다고 판단
 - 모든 층에 연결 될 경우에 true
 - False일 경우에 접근이 허용되지 않으며 true가 될 때까지 스킵

잠금 기반의

● ADD

1. 검색한다
2. 검색노드의 안정화 대기
3. 0레벨부터 valid 검사하면서 locking
4. 수정
5. unlocking

```

boolean add(T x) {
    int topLevel = randomLevel();
    Node<T>[] preds = (Node<T>[]) new Node[MAX_LEVEL + 1];
    Node<T>[] succs = (Node<T>[]) new Node[MAX_LEVEL + 1];
    while (true) {
        int lFound = find(x, preds, succs);
        if (lFound != -1) {
            Node<T> nodeFound = succs[lFound];
            if (!nodeFound.marked) {
                while (!nodeFound.fullyLinked) {}
                return false;
            }
            continue;
        }
        int highestLocked = -1;
        try {
            Node<T> pred, succ;
            boolean valid = true;
            for (int level = 0; valid && (level <= topLevel); level++) {
                pred = preds[level];
                succ = succs[level];
                pred.lock.lock();
                highestLocked = level;
                valid = !pred.marked && !succ.marked && pred.next[level] == succ;
            }
            if (!valid) continue;
            Node<T> newNode = new Node(x, topLevel);
            for (int level = 0; level <= topLevel; level++)
                newNode.next[level] = succs[level];
            for (int level = 0; level <= topLevel; level++)
                preds[level].next[level] = newNode;
            newNode.fullyLinked = true; // successful add linearization point
            return true;
        } finally {
            for (int level = 0; level <= highestLocked; level++)
                preds[level].unlock();
        }
    }
}

```


잠금 기반의 병행 스킵리스트

● Remove()

- Find()를 호출해서 대상키를 가지고 있는 노드가 이미 리스트에 있는지 확인
 - 리스트에 있다면, 제거할 노드를 삭제할 수 있는지 확인
 - Marked, fullyLinked 필드를 이용해 확인
 - 삭제할 수 있다면, marked필드를 설정하여 논리적으로 삭제
 - 물리적인 삭제 방법은
 - 대상 노드의 앞 노드와 대상 노드를 잠그고, 각 층에서 하나씩 대상 노드를 제거(위층부터 제거)하고 다음 노드와 연결한다.

잠금 기반의

● Remove()

```

95  boolean remove(T x) {
96      Node<T> victim = null; boolean isMarked = false; int topLevel = -1;
97      Node<T>[] preds = (Node<T>[]) new Node[MAX_LEVEL + 1];
98      Node<T>[] succs = (Node<T>[]) new Node[MAX_LEVEL + 1];
99      while (true) {
100         int lFound = find(x, preds, succs);
101         if (lFound != -1) victim = succs[lFound];
102         if (isMarked |
103             (lFound != -1 &&
104              (victim.fullyLinked
105               && victim.topLevel == lFound
106                && !victim.marked)))) {
107             if (!isMarked) {
108                 topLevel = victim.topLevel;
109                 victim.lock.lock();
110                 if (victim.marked) {
111                     victim.lock.unlock();
112                     return false;
113                 }
114                 victim.marked = true;
115                 isMarked = true;
116             }
117             int highestLocked = -1;
118             try {
119                 Node<T> pred, succ; boolean valid = true;
120                 for (int level = 0; valid && (level <= topLevel); level++) {
121                     pred = preds[level];
122                     pred.lock.lock();
123                     highestLocked = level;
124                     valid = !pred.marked && pred.next[level] == victim;
125                 }
126                 if (!valid) continue;
127                 for (int level = topLevel; level >= 0; level--) {
128                     preds[level].next[level] = victim.next[level];
129                 }
130                 victim.lock.unlock();
131                 return true;
132             } finally {
133                 for (int i = 0; i <= highestLocked; i++) {
134                     preds[i].unlock();
135                 }
136             }
137         } else return false;
138     }
139 }

```

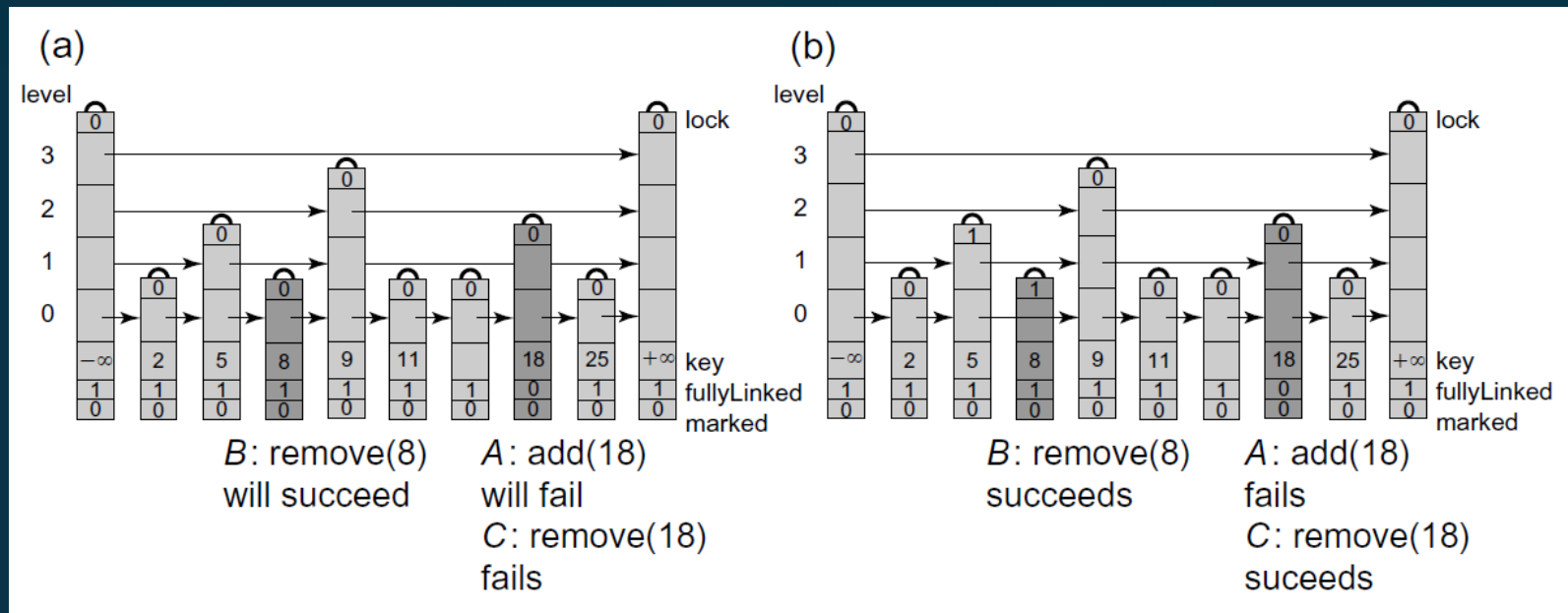
잠금 기반의 병행 스킵리스트

● Contains

```
140  boolean contains(T x) {  
141      Node<T>[] preds = (Node<T>[]) new Node[MAX_LEVEL + 1];  
142      Node<T>[] succs = (Node<T>[]) new Node[MAX_LEVEL + 1];  
143      int lFound = find(x, preds, succs);  
144      return (lFound != -1  
145          && succs[lFound].fullyLinked  
146          && !succs[lFound].marked);  
147  }
```

잠금 기반의 병행 스킵리스트

● 조감도



잠금 기반의 병행 스킵리스트

- 결과

- i7 - 920

- 연산횟수 32768

- 순차 SkipList(1Thread) : 0.37s

Thread	LazySkipLisy	Efficiency
1	0.094s	1
2	0.053s	0.885
4	0.026s	0.90
8	0.024s	0.48
16	0.020s	0.29
32	0.020s	0.14

숙제

- 숙제 13 :

- 게으른 동기화 SkipList의 구현

- 수업시간에 작성한 프로그램을 디버깅하여 완성하시오

- 제출물

- .cpp 파일
 - 실행속도 비교표 (쓰레드 개수 별)
 - CPU의 종류 (모델명, 코어 개수, 클럭)

- 제출 : nhjung@kpu.ac.kr

- 제목 : [2018 전공특강 숙제 13] 학번, 이름
 - 6월 12일 일요일 오후 1시까지 제출

SKIP-LIST

- 소개
- 순차 스킵 리스트
- 병행 스킵 리스트
- 6월 12일 일요일 오후 2시 보강
- Lock-Free 스킵 리스트

LockFreeSkipList

- LockFreeSkipList

- 3 장의 LockFreeList를 사용
- 삽입과 삭제를 위해 CAS 사용
- 삭제 시에는 노드의 next에 표시하여 논리적으로 제거한다
- 물리적인 제거는 Find()에서 이루어 진다.
 - Find()는 리스트를 순회하면서 표시된 노드를 만날 때마다 잘라내어서 표시된 노드의 키는 절대 검색하지 않는다.

LockFreeSkipList

- 클래스 정의
 - LockFree List
 - 합성 포인터

```

1 public final class LockFreeSkipList<T> {
2     static final int MAX_LEVEL = ...;
3     final Node<T> head = new Node<T>(Integer.MIN_VALUE);
4     final Node<T> tail = new Node<T>(Integer.MAX_VALUE);
5     public LockFreeSkipList() {
6         for (int i = 0; i < head.next.length; i++) {
7             head.next[i]
8                 = new AtomicMarkableReference<LockFreeSkipList.Node<T>>(tail, false);
9         }
10    }
11    public static final class Node<T> {
12        final T value; final int key;
13        final AtomicMarkableReference<Node<T>>[] next;
14        private int topLevel;
15        // constructor for sentinel nodes
16        public Node(int key) {
17            value = null; key = key;
18            next = (AtomicMarkableReference<Node<T>>[])
19                new AtomicMarkableReference[MAX_LEVEL + 1];
20            for (int i = 0; i < next.length; i++) {
21                next[i] = new AtomicMarkableReference<Node<T>>(null, false);
22            }
23            topLevel = MAX_LEVEL;
24        }
25        // constructor for ordinary nodes
26        public Node(T x, int height) {
27            value = x;
28            key = x.hashCode();
29            next = (AtomicMarkableReference<Node<T>>[])
30                new AtomicMarkableReference[height + 1];
31            for (int i = 0; i < next.length; i++) {
32                next[i] = new AtomicMarkableReference<Node<T>>(null, false);
33            }
34            topLevel = height;
35        }
36    }
37 }

```

LockFreeSkipList

- Add()

- Find()를 이용하여 노드가 이미 리스트에 있는지 확인하고 자신의 앞노드(preds[])와 뒤노드(succs[])를 얻는다
- 새 노드는 최하층 리스트에 연결하여 논리적으로 추가하고 그 다음에 순서대로 최상층까지 연결한다
 - 최상층
 - 새 노드가 고른 임의의 topLevel값이다.

LockFreeSkipList

● Add()

```

36  boolean add(T x) {
37      int topLevel = randomLevel();
38      int bottomLevel = 0;
39      Node<T>[] preds = (Node<T>[]) new Node[MAX_LEVEL + 1];
40      Node<T>[] succs = (Node<T>[]) new Node[MAX_LEVEL + 1];
41      while (true) {
42          boolean found = find(x, preds, succs);
43          if (found) {
44              return false;
45          } else {
46              Node<T> newNode = new Node(x, topLevel);
47              for (int level = bottomLevel; level <= topLevel; level++) {
48                  Node<T> succ = succs[level];
49                  newNode.next[level].set(succ, false);
50              }
51              Node<T> pred = preds[bottomLevel];
52              Node<T> succ = succs[bottomLevel];
53              newNode.next[bottomLevel].set(succ, false);
54              if (!pred.next[bottomLevel].compareAndSet(succ, newNode,
55                                                         false, false)) {
56                  continue;
57              }
58              for (int level = bottomLevel+1; level <= topLevel; level++) {
59                  while (true) {
60                      pred = preds[level];
61                      succ = succs[level];
62                      if (pred.next[level].compareAndSet(succ, newNode, false, false))
63                          break;
64                      find(x, preds, succs);
65                  }
66              }
67              return true;
68          }
69      }
70  }

```

LockFreeSkipList

● Remove()

- Find()를 호출하여 표시되지 않고 대상키를 갖는 노드가 최하층리스트에 있는지 확인
 - 최하층 리스트만 빼고 표시를 남겨서 논리적으로 제거
 - 최하층을 제외하고 전부 표시했다면 최하층의 next 참조에 표시
 - 이 표시를 남기는 것에 성공하면 Find()를 통해 물리적 제거를 한다.

LockFreeSkipList

● Remove()

```

71  boolean remove(T x) {
72      int bottomLevel = 0;
73      Node<T>[] preds = (Node<T>[]) new Node[MAX_LEVEL + 1];
74      Node<T>[] succs = (Node<T>[]) new Node[MAX_LEVEL + 1];
75      Node<T> succ;
76      while (true) {
77          boolean found = find(x, preds, succs);
78          if (!found) {
79              return false;
80          } else {
81              Node<T> nodeToRemove = succs[bottomLevel];
82              for (int level = nodeToRemove.topLevel;
83                  level >= bottomLevel+1; level--) {
84                  boolean[] marked = {false};
85                  succ = nodeToRemove.next[level].get(marked);
86                  while (!marked[0]) {
87                      nodeToRemove.next[level].attemptMark(succ, true);
88                      succ = nodeToRemove.next[level].get(marked);
89                  }
90              }
91              boolean[] marked = {false};
92              succ = nodeToRemove.next[bottomLevel].get(marked);
93              while (true) {
94                  boolean iMarkedIt =
95                      nodeToRemove.next[bottomLevel].compareAndSet(succ, succ,
96                                                                    false, true);
97                  succ = succs[bottomLevel].next[bottomLevel].get(marked);
98                  if (iMarkedIt) {
99                      find(x, preds, succs);
100                     return true;
101                 }
102                 else if (marked[0]) return false;
103             }
104         }
105     }
106 }

```

LockFreeSkipList

● Find()

- 스킵리스트를 순회하는데 각 층을 내려가며 리스트를 진행한다.
- 진행 중에 표시된 노드(marked)를 만나면 이 노드를 잘라내어서 표시된 노드의 키값을 절대 보지 않는다.
 - 현재 노드보다 더 높은 층에서 이런 현상이 발생할 수 있으므로 LazySkipList처럼 **스킵리스트 속성**은 유지되지 않는다.

LockFreeSkipList

● Find()

- LockFreeSkipList를 검색하고 대상키가 집합에 있을 때만 true를 반환
- 두가지 속성을 만족
 - 표시된 연결(marked)은 순회하지 않는다. 대신 해당 층의 리스트에서 표시된 연결이 참조된 노드를 제거
 - 모든 preds[]참조는 대상 키보다 작은 키를 갖는 노드

LockFreeSkipList

● Find()

```

107 boolean find(T x, Node<T>[] preds, Node<T>[] succs) {
108     int bottomLevel = 0;
109     int key = x.hashCode();
110     boolean[] marked = {false};
111     boolean snip;
112     Node<T> pred = null, curr = null, succ = null;
113     retry:
114     while (true) {
115         pred = head;
116         for (int level = MAX_LEVEL; level >= bottomLevel; level--) {
117             curr = pred.next[level].getReference();
118             while (true) {
119                 succ = curr.next[level].get(marked);
120                 while (marked[0]) {
121                     snip = pred.next[level].compareAndSet(curr, succ,
122                                                             false, false);
123                     if (!snip) continue retry;
124                     curr = pred.next[level].getReference();
125                     succ = curr.next[level].get(marked);
126                 }
127                 if (curr.key < key) {
128                     pred = curr; curr = succ;
129                 } else {
130                     break;
131                 }
132             }
133             preds[level] = pred;
134             succs[level] = curr;
135         }
136         return (curr.key == key);
137     }
138 }

```

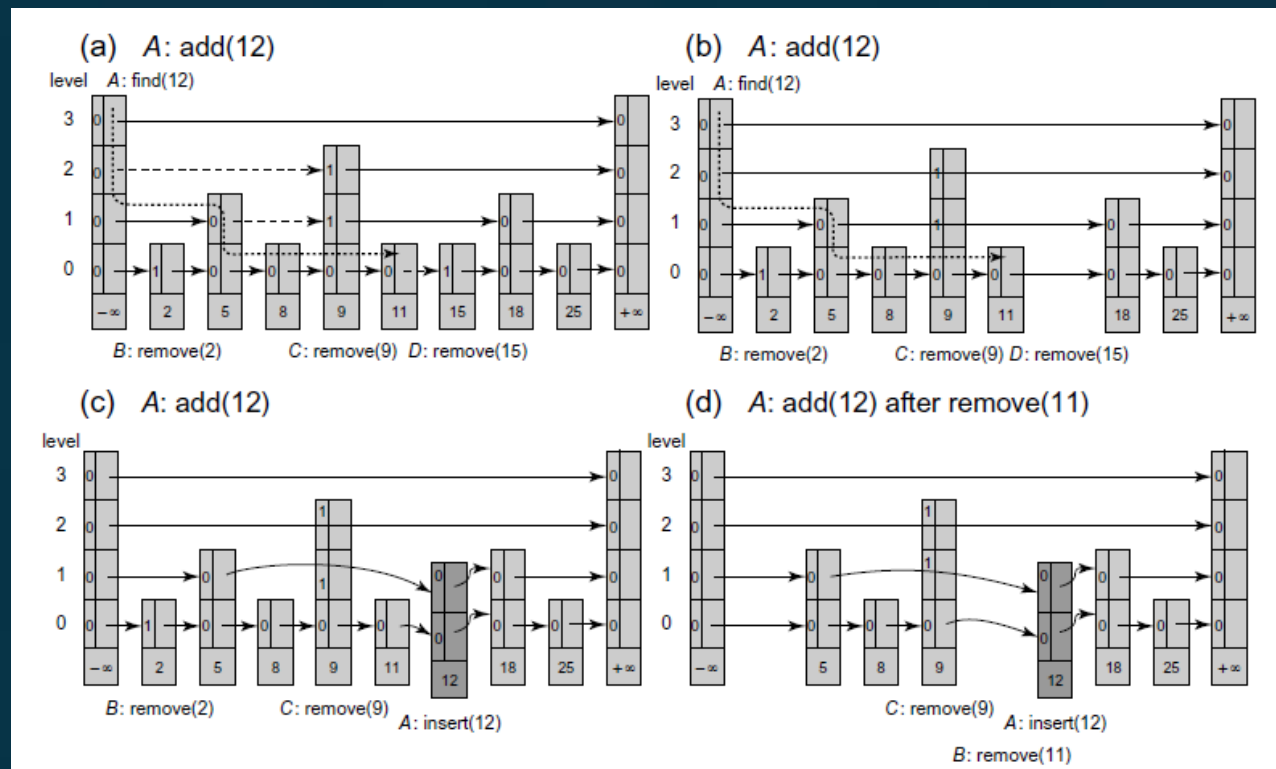

LockFreeSkipList

● Contains()

```
139  boolean contains(T x) {
140      int bottomLevel = 0;
141      int v = x.hashCode();
142      boolean[] marked = {false};
143      Node<T> pred = head, curr = null, succ = null;
144      for (int level = MAX_LEVEL; level >= bottomLevel; level--) {
145          curr = pred.next[level].getReference();
146          while (true) {
147              succ = curr.next[level].get(marked);
148              while (marked[0]) {
149                  curr = pred.next[level].getReference(); // curr.next로 변경
150                  succ = curr.next[level].get(marked);
151              }
152              if (curr.key < v) {
153                  pred = curr;
154                  curr = succ;
155              } else {
156                  break;
157              }
158          }
159      }
160      return (curr.key == v);
161  }
```

LockFreeSkipList

● LockFreeSkipList



LockFreeSkipList

- 실습 #24

- Lock Free Skip List 프로그램을 작성한다.
 - 4장에서 구현된 Free-List 연동
- 순차 Skip List와 속도 비교
- Lock Free 일반 List와 속도 비교

정리

- non-blocking Stack
 - 부하 분산テクニック
 - Elimination
 - non-blocking free list 구현
- non-blocking 검색 자료구조
 - SKIP-LIST
 - 평균 $O(\log n)$ 의 검색 시간

총정리

- 왜 멀티쓰레드 프로그래밍을 해야 하는가?
- 멀티쓰레드 프로그램은 멀티코어 CPU에서 어떻게 실행되는가?
- 멀티쓰레드 프로그래밍 API는 무엇인가?
- 내가 작성한 멀티쓰레드 프로그램이 왜 죽는가?
- Lock을 쓰면 왜 느린가?
- Lock을 제거했더니 왜 죽는가?

총정리

- Lock을 사용하지 않을 때 스레드간의 동기화는 어떻게 구현 하는 것이 좋은가?
- Non-Blocking 알고리즘이 무엇인가?
 - 어떻게 구현해야 하는가?
 - Blocking보다 왜 좋은가? 어떤 경우에 좋은가?
- Lock-Free 알고리즘의 구현이 왜 어려운가?

총정리

- OpenMP, TBB, CUDA, Transactional Memory는 무엇인가?
- 앞으로 우리는 어떠한 프로그래밍을 해야 하는가?
 - 언제까지 non-blocking 알고리즘을 써야 하나?
 - Core의 개수는 점점 늘어날 것인가?
 - C 스타일언어의 미래는?
 - 5년 후는?
 - 10년 후는?